EXPERT INSIGHT

Developing IoT Projects with ESP32

Unlock the full Potential of ESP32 in IoT development to create production-grade smart devices

Second Edition

packt

Vedat Ozan Oner

Developing IoT Projects with ESP32

Second Edition

Unlock the full Potential of ESP32 in IoT development to create production-grade smart devices

Vedat Ozan Oner

BIRMINGHAM—MUMBAI

Developing IoT Projects with ESP32

Second Edition

Copyright © 2023 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 capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Rahul Nair

Acquisition Editor – Peer Reviews: Gaurav Gavas

Project Editor: Namrata Katare

Content Development Editor: Soham Amburle

Copy Editor: Safis Editing

Technical Editor: Anjitha Murali

Proofreader: Safis Editing

Indexer: Rekha Nair

Presentation Designer: Ganesh Bhadwalkar

Developer Relations Marketing Executive: Meghal Patel

First published: September 2021

Second edition: November 2023

Production reference: 1231123

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80323-768-8

www.packt.com

Contributors

About the author

Vedat Ozan Oner is an IoT product developer and software architect with more than 15 years of experience. He is also the author of *Developing IoT Projects with ESP32, First Edition*, published by *Packt*, one of the best-sellers in the field. Vedat has a bachelor's degree in computer engineering from Middle East Technical University, Ankara, Turkey and holds several industry-recognized credentials and qualifications, including **PMP®**, **ITIL®**, and **AWS** Certified Developer. He established his own company, Mevoo Ltd, in 2018 in London to provide consultancy services to his clients and develop his own IoT products. Vedat currently lives in Gloucester, England with his family.

Heart-felt thanks to my wife for her relentless support and patience. Her teas kept my mind fresh during the long nights while I was working on this book. I owe special thanks to the readers of the first edition. Your feedback on the first book was invaluable and helped me a lot to decide on the content of this second edition.

About the reviewers

Emmanuel Odunlade, a hardware design engineer, solution architect, and entrepreneur, has an extensive background in embedded hardware design and has led the development of several hardware product categories, including consumer, medical, industrial, and military, from conception to production.

He currently leads the electrical engineering team at Sure Grip Controls, building industry-leading control solutions featured in the cabins of some of the world's leading off-highway vehicles.

Before working at Sure Grip, Emmanuel was an ML/IoT hardware architecture specialist at Vision X, leading the development of edge hardware for Fortune 500 companies, and principal IoT solution architect at Hinge, overseeing the development and deployment of several bespoke IoT solutions for customers across diverse sectors.

When not architecting solutions or playing melodious (some people may not agree) tunes on the saxophone, Emmanuel loves to write and is a contributor to several magazines and blogs with over 500 published articles, and is currently working on his first book.

> *Thank you to God, to Eyitope for the reminders, to Ibukun, Audrey, and Tinuke for always being there, to Vedat for the opportunity to be part of this journey, and finally to Manish, Namrata, and the incredible team that worked on this book.*

Royyan Abdullah Dzakiy is an IoT developer, currently the manager of eFishery's R&D team in Indonesia. He leads technical

teams (firmware, electrical, mechanical, AI, and full stack), product managers, and research teams (PhD researchers and aquaculture scientists). His focus has mainly been on solving complex aquaculture challenges, contributing to products and patents like fish and shrimp feeders, aquatic livestock sensors, LoRa and BLE for rural connectivity, livestock behavioral AI, GIS image processing, digitizing written forms with OCR, etc.

He teaches IoT in rural areas, including topics such as tech product development and research. Beyond his work, he's also an FPV drone pilot and licensed ham radio enthusiast, runs an IoT-based NGO, and was a member of Edinburgh Hacklab.

I would like to thank Allah for His mercy and kindness, for giving me this wonderful opportunity. I'd like to thank my lovely wife, for allowing me to take up some of our precious time to contribute to this book. And finally, I would like to thank the author, Vedat, for trusting me in this role.

Carlos Bugs has been working with technology for more than 18 years. He started with electronics projects from scratch, then worked with embedded firmware in assembly and later in C. He has worked on many products in areas like agriculture, instrumentation, automotive, industry, and sustainability.

He has also worked for large companies as a consultant, where he learned about managing business goals, as well as how to deal with stakeholders.

He is also an entrepreneur and was the co-founder and CTO of a tech organization called *Syos*, whose goal was to connect the cold

chain through IoT and AI to ensure safety, efficiency, and sustainability in the health and food sector.

> *I would like to congratulate Vedat for the great job he did. Writing a technical book is a big challenge and Vedat really utilized his knowledge and experience in real projects. Also, I would like to thank Packt Publishing for all the support that they gave during this amazing journey.*

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://discord.gg/3Q9egBjWVZ`

Contents

- Preface
  - Who this book is for
  - What this book covers
  - To get the most out of this book
  - Get in touch
1. Introduction to IoT development and the ESP32 platform
   - Technical requirements
   - Understanding the basic structure of IoT solutions
     - IoT security
   - The ESP32 product family
     - ESP32 series
     - Other SoCs
   - Development platforms and frameworks
   - RTOS options
   - Summary
2. Understanding the Development Tools
   - Technical requirements
   - ESP-IDF
     - The first application
     - ESP-IDF Terminal
   - PlatformIO
     - Hello world with PlatformIO
     - PlatformIO Terminal

- FreeRTOS
  - Creating the producer-consumer project
  - Coding application
  - Running the application
- Debugging
- Unit testing
  - Creating a project
  - Coding the application
  - Adding unit tests
  - Running unit tests
- Summary
- Questions
- Further reading

3. Using ESP32 Peripherals
   - Technical requirements
   - Driving General-Purpose Input/Output (GPIO)
     - Turning an LED on/off by using a button
     - Creating a project
     - Coding the application
     - Troubleshooting
   - Interfacing with sensors over Inter-Integrated Circuit (I2C)
     - Developing a multisensor application
     - Creating a project
     - Coding the application
     - Troubleshooting
   - Integrating with SD cards over Serial Peripheral Interface (SPI)
     - Adding SD card storage

- Creating the project
- Coding the application
- Testing the application
- Troubleshooting

Audio output over Inter-IC Sound (I²S)

- Developing a simple audio player
- Coding the application
- Testing the application

Developing graphical user interfaces on Liquid-Crystal Display (LCD)

- A simple graphical user interface (GUI) on ESP32
- Creating the project
- Coding the application
- Testing the application

Summary

Questions

Further reading

4. Employing Third-Party Libraries in ESP32 Projects

Technical requirements

LittleFS

- Creating a project
- Coding the application
- Testing the application

Nlohmann-JSON

- Creating a project
- Coding the application
- Testing the application

Miniz

        - Creating a project
        - Coding the project
        - Testing the application
    - FlatBuffers
        - Creating a project
        - Coding the application
        - Testing the application
    - LVGL
        - Designing the GUI
        - Creating a project
        - Coding the application
        - Testing the application
    - ESP-IDF Components library
    - Espressif frameworks and libraries
    - Summary
    - Questions
5. Project – Audio Player
    - Technical requirements
    - The feature list of the audio player
    - Solution architecture
    - Developing the project
        - Designing the GUI
        - Creating the IDF project
        - Coding the application
    - Testing the Project
    - New features
    - Troubleshooting
    - Summary
6. Using Wi-Fi Communication for Connectivity

Technical requirements
Connecting to local Wi-Fi
- Creating a project
- Coding the application
- Testing the application
- Troubleshooting

Provisioning ESP32 on a Wi-Fi network
- Creating a project
- Coding the application
- Testing application
- Troubleshooting

Communicating over MQTT
- Installing the MQTT broker
- Creating a project
- Coding the application
- Testing the application
- Troubleshooting

Running a RESTful server on ESP32
- Creating the project
- Coding the application
- Testing the application

Consuming RESTful services
- Creating the project
- Coding the application
- Testing the application
- Troubleshooting

Summary
Questions
Further reading

7. ESP32 Security Features for Production-Grade Devices
    - Technical requirements
    - ESP32 security features
        - Secure Boot v1
        - Secure Boot v2
        - Digital Signature (DS)
        - ESP Privilege Separation
    - Over-the-air updates
    - Upgrading firmware from an HTTPS server
        - Preparing the server
        - Creating a project
        - Coding the application
        - Testing the application
        - Troubleshooting
    - Utilizing RainMaker for OTA updates
        - Configuring RainMaker
        - Creating a project
        - Coding the application
        - Testing the application
        - Troubleshooting
    - Sharing data over secure MQTT
        - Creating a project
        - Coding the application
        - Testing the application
        - Troubleshooting
    - Summary
    - Questions
    - Further reading

8. Connecting to Cloud Platforms and Using Services
    - Technical requirements
    - Developing on AWS IoT
        - Hardware setup
        - Creating an AWS IoT thing
        - Configuring a project
        - Coding the application
        - Testing the application
        - Troubleshooting
    - Visualizing with Grafana
        - Creating a Timestream database
        - Creating a Grafana workspace
        - Creating a Grafana dashboard
        - Troubleshooting
    - Integrating an ESP32 device with Amazon Alexa
        - Updating the thing shadow
        - Creating the lambda handler
        - Coding the lambda handler
        - Creating the smart home skill
        - Troubleshooting
    - Summary
    - Questions
    - Further reading
9. Project – Smart Home
    - Technical requirements
    - The feature list of the smart home solution
    - Solution architecture
        - Setting up plug hardware
        - Setting up multisensor hardware

        - Software architecture
    - Implementation
        - Preparing common libraries
            - Creating IDF component
            - Coding IDF component
        - Developing plug
            - Adding plug node
            - Coding application
        - Developing multisensor
            - Adding sensor node
            - Adding a GUI
            - Coding the application
    - Testing project
        - Testing plug
        - Testing the multisensor application
        - Using smart home features
    - Troubleshooting
    - New features
    - Summary
10. Machine Learning with ESP32
    - Technical requirements
    - Learning the ML basics
        - ML approaches to solve computing problems
            - Supervised learning
            - Unsupervised learning
            - Reinforced learning
        - TinyML pipeline

Data collection and preprocessing
Designing and training a model
Optimizing and preparing the model for deployment
Running inference on an IoT device
Running inference on ESP32
Creating the project
Coding the application
Testing the application
Developing a speech recognition application
Creating the project
Coding the application
Testing the application
Troubleshooting
Summary
Questions
Further reading
11. Developing on Edge Impulse
Technical requirements
An overview of Edge Impulse
Cloning an Edge Impulse project
Using the ML model on ESP32
The model library
The application code
Testing the application
Troubleshooting
Next steps for TinyML development
The Netron app
TinyML Foundation

ONNX format
Project ideas
Image processing with ESP32-S3-EYE
Anomaly detection
Summary
Questions
Further reading
12. Project – Baby Monitor
Technical requirements
The feature list of the baby monitor
Solution architecture
Implementation
Generating the ML model
Creating an IDF project
Developing the application
Testing the project
Troubleshooting
New features
Summary
Answers
Chapter 02:
Chapter 03:
Chapter 04:
Chapter 06:
Chapter 07:
Chapter 08:
Chapter 10:
Chapter 11:
Other Books You May Enjoy
Index

Preface

It has been a long time since the first **Internet of Things** (**IoT**) devices entered our lives, and now they are helping us in many ways. We have smart TVs, voice assistants, connected appliances at home, or **Industrial IoT** (**IIoT**) devices being used in the transportation, healthcare, agriculture, and energy sectors – virtually everywhere. The new generation has been growing up with this technology and using IoT devices effectively (my 3-year-old daughter's music box, for example, is an Echo device). Furthermore, new IoT products are introduced on the market every day with novel features or improved capabilities.

We all appreciate how fast technology is changing. It is hard for everybody to keep up with all those changes: technology manufacturers, technology consumers, and, in between them, people like us – IoT developers that make technology available to consumers. Since the 1st edition of this book, **Espressif Systems** has added many chips to their portfolio in response to market needs. For instance, we see the single-core ESP32-C family of **System-on-Chip** (**SoC**) devices with RISC-V architecture. They have a reduced set of capabilities and memory but are much cheaper compared to the first ESP32. There is also the ESP32-S family as a new branch of the original ESP32 SoCs with more capabilities and peripherals to support **Artificial Intelligence-of-Things** (**AIoT**) solutions. On top of hardware, we see state-of-the-art frameworks and libraries that

enable us to use those SoCs in different types of applications. In this book, I've tried to cover them from a bit of a different perspective in addition to the basics of ESP32 development as a starting point.

There are several key differences between the first edition and this one. First of all, the examples of this edition are developed in C++ by employing ESP-IDF, compared to the C programming language and the PlatformIO environment in the first edition. We will also use different development kits from Espressif Systems in this edition, which makes hardware setup easier in some examples. In terms of content, we will discuss machine learning on ESP32 with hands-on projects, but the Bluetooth/BLE topics have been excluded from the book and some others have been condensed to make room for the machine learning examples. A noteworthy addition that I expect you would find interesting in this edition is the exploration of integration with third-party libraries. In the relevant chapter, various methods of incorporating third-party libraries into ESP32 projects will be discussed.

This doesn't mean the 1st edition is now obsolete. On the contrary, it is still perfectly valid if you are new to IoT with ESP32. With this edition of the book, we have a chance to discuss the subjects where the 1st edition With this edition of the book, we have a chance to discuss in detail about the emerging new technology in terms of new technology. I really enjoyed preparing the examples for this book, and I hope you enjoy them, too. I want to share a wise quote from a distinguished historian and women's rights activist, Mary Ritter Beard, before delving into the topics.

"Action without study is fatal. Study without action is futile."

- Mary Ritter Beard

Who this book is for

This book is targeted at embedded software developers, IoT software architects/developers, and technologists who want to learn how to employ ESP32 effectively in their IoT projects.

What this book covers

Chapter 1, *Introduction to IoT Development and the ESP32 Platform*, discusses IoT technology in general and introduces the ESP32 platform in terms of both hardware and software.

Chapter 2, *Understanding the Development Tools*, talks about the popular development environments ESP-IDF and PlatformIO, and teaches you how to utilize the toolchain to develop and test ESP32 applications.

Chapter 3, *Using ESP32 Peripherals*, gives practical examples of integrating with sensors and actuators by interfacing with common ESP32 peripherals, including audio and graphics.

Chapter 4, *Employing Third-Party Libraries in ESP32 Projects*, talks about different methods of importing third-party libraries with examples. LVGL is one of the libraries discussed in this chapter.

Chapter 5, *Project – Audio Player*, is the first reference project in the book with audio, graphics, and button interactions to engage its users.

Chapter 6, *Using Wi-Fi Communication for Connectivity*, shows how to communicate over different application layer protocols, such as MQTT and REST, after connecting to a local Wi-Fi network.

Chapter 7, *ESP32 Security Features for Production-Grade Devices*, explores the security features of ESP32 by giving examples of secure firmware updates and secure communication techniques. ESP

RainMaker is the IoT platform that provides the backend services in the examples.

Chapter 8, *Connecting to Cloud Platforms and Using Services*, explains how to pass data to AWS IoT Core and visualize it on Grafana. Amazon Alexa integration is also covered with a step-by-step project example.

Chapter 9, *Project – Smart Home*, builds a full-fledged smart home solution on the ESP RainMaker platform to show how different devices can operate together in the same product.

Chapter 10, *Machine Learning with ESP32*, introduces the basics of machine learning and tinyML on ESP32, and discusses Espressif's machine learning frameworks with a speech recognition example.

Chapter 11, *Developing on Edge Impulse*, explains how to develop machine learning applications on ESP32 by utilizing the Edge Impulse platform.

Chapter 12, *Project – Baby Monitor*, is the last project of the book, which shows how to design and develop a connected machine learning product. Edge Impulse and ESP RainMaker are the platforms employed in the project.

To get the most out of this book

The examples are written in **modern C++** by using **ESP-IDF** (the major development framework for ESP32, maintained by Espressif Systems). Therefore, a basic understanding of modem C++ concepts would be beneficial to get a better grasp of the subjects discussed in the book. Although not required, some familiarity with using

command-line tools in a terminal window could also help to follow the examples.

I tried to explain all the subjects in the scope of the book in as much detail as possible. Nevertheless, IoT is a vast field to talk about in a single book, so I appended a *Further reading* section at the end of most of the chapters in case you need some background information. If you find it difficult to follow any of the underlying subjects in a chapter, reading the reference books listed in the *Further reading* sections will support you in understanding the examples of that specific chapter better.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://packt.link/gbp/9781803237688.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system.”

A block of code is set as follows:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.con
     /etc/asterisk/cdr_mysql.conf
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Select **System info** from the **Administration** panel.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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 reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit 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@packtpub.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 http://authors.packtpub.com.

Share your thoughts

Once you've read *Developing IoT Projects with ESP32, Second Edition*, 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/9781803237688

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

1

Introduction to IoT development and the ESP32 platform

Internet of Things (**IoT**) is a common term that refers to devices that we interact with, in our daily lives and share data between them over the internet to harness the power of information. When connected, a device has access to more information to process and can better decide what to do next in the scope of its design goals. Although this defines a basic understanding of IoT, it has more aspects with wider implications beyond this fundamental description, which we will discuss throughout this book.

Espressif's ESP32 is a powerful tool in the toolbox of a developer for many types of **IoT** projects. We are all developers, and we all know how important it is to select the right tool for a given problem in a domain. To solve a problem, we need to understand the domain, and we need to know the available tools and their features in order to find the right one (or perhaps several combined). After selecting the tool, we eventually need to figure out how to use it in the most efficient and effective way possible so as to maximize the added value for end users. When it comes to IoT, tool selection becomes more important. It is not only software tools but also the selection of hardware tools that can make a paramount difference in deciding

the success of a product. The ESP32 product family has a special place in the IoT world with diverse application areas. We can develop simple connected sensors to be used in homes as well as industry-grade **Artificial Intelligence of Things** (**AIoT**) applications in manufacturing. Despite its low price tag, it provides a good amount of processing power with a high degree of connectivity capabilities and modern security features, which makes it a strong option in many types of IoT projects.

In this chapter, I will discuss IoT technology, in general, what an IoT solution looks like in terms of basic architecture, and how ESP32 fits into those solutions as a tool. If you are new to IoT technology or are thinking of using ESP32 in your next project, this chapter will help you to understand the big picture from a technology perspective by describing what ESP32 provides, its capabilities, and its limitations.

The main topics covered in this chapter are as follows:

- Understanding the basic structure of IoT solutions
- The ESP32 product family
- Development platforms and frameworks
- RTOS options

Technical requirements

In this book, we are going to go through many practical examples where we can learn how to use ESP32 effectively in real-world scenarios. Although links to the examples are provided within each chapter, you can take a sneak peek at the online repository here: https://github.com/PacktPublishing/Developing-IoT-

Projects-with-ESP32-2nd-edition. The examples are placed in their relative directories of the chapters for easy browsing. There is also a common source code directory that contains the shared libraries across the chapters.

The programming language of the examples is usually C++11 (the default C++ standard supported by the toolchain). However, there are several chapters where Python 3 is required to support the subject.

The hardware tools, development kits, and sensors that you will need throughout the book are the following:

- ESP32-S3-BOX-Lite (approx. $35)
- ESP32-C3-DevKitM-1 (approx. $8)
- BME280 temperature, humidity, pressure breakout board (approx. $15)
- TSL2561 ambient light breakout board (approx. $6)
- SPI SD card breakout board (approx. $3)
- A micro-SD memory card (any micro-SD would work)
- LEDs, tactile switches, various resistors, and hook-up cables

The total cost of the hardware that you need for the projects is around 70 USD. However, it might change a bit according to the store you buy from.

The examples are designed for these devkits and tested on them. If you want to use any other hardware that you might have, you need to choose the right drivers and/or **board support packages** (**BSPs**) where necessary.

Understanding the basic structure of IoT solutions

Although the definition of IoT might change slightly from different viewpoints, there are some key concepts in the IoT world that differentiate it from other types of technologies:

- **Connectivity**: An IoT device is connected either to the internet or to a local network. An old-style thermostat on the wall waiting for manual operation with basic programming features doesn't count as an IoT device.
- **Identification**: An IoT device is uniquely identified in the network so that data has a context identified by that device. In addition, the device itself is available for remote update, remote management, and diagnostics.
- **Autonomous operation**: IoT systems are designed for minimal or no human intervention. Each device collects data from the environment where it is installed, and it can then communicate the data with other devices to detect the current status of the system and respond as configured. This response can be in the form of an action, a log, or an alert if required.
- **Interoperability**: Devices in an IoT solution talk to one another, but they don't necessarily belong to a single vendor. When devices designed by different vendors share a common application-level protocol, adding a new device to that heterogeneous network is as easy as clicking on a few buttons on the device or on the management software.

- **Scalability**: IoT systems are capable of horizontal scalability to respond to an increasing workload. A new device is added when necessary to increase capacity instead of replacing the existing one with a superior device (vertical scalability).
- **Security**: I wish I could say that every IoT solution implements at least the minimal set of mandatory security measures, but unfortunately, this is not the case, despite a number of bad experiences, including the infamous Mirai botnet attack. On a positive note, I can say that IoT devices mostly have secure boot, secure update, and secure communication features to ensure confidentiality, integrity, and availability (also known as the CIA triad).

Although these key concepts define some commonality between IoT solutions in general, not many IoT products have implemented them before showing up on the market. They have different levels of maturity in these areas. The limitations or constraints may come from a specific use case; however, in addition to its value proposition, a modern IoT product should apply basic good practices in these areas to protect its users and its brand value.

An IoT solution combines many different technologies into a single product, starting from a physical device and covering all layers up to end user applications. Each layer of the solution aims to implement the same vision set by the business but requires a different approach while designing and developing. We definitely cannot talk about one-size-fits-all solutions in IoT projects, but we can still apply an organized approach to developing products. Let's see which layers a solution has in a typical IoT product:

- **Device hardware**: Every IoT project requires hardware with a **System-On-Chip** (**SoC**) or **Microcontroller Unit** (**MCU**) and sensors/actuators to interact with the physical world. In addition to that, every IoT device is connected, so we need to select the optimal communication medium, such as wired or wireless. Power management is also another consideration under this category.
- **Device firmware**: We need to develop device firmware to run on the SoC in order to fulfill the project's requirements. This is where we collect data and transfer it to the other components in the solution.
- **Communication**: Connectivity issues are handled in this category of the solution architecture. The physical medium selection corresponds to one part of the solution, but we still need to decide on the protocol between devices as a common language for sharing data. Some protocols may provide a whole stack of communication by defining the physical medium up to the application layer. If this is the case, you don't need to worry about anything else, but if your stack leaves the context management at the application layer up to you, then it is time to decide on what IoT protocol to use.
- **Backend system**: This is the backbone of the solution. All data is collected on the backend system and provides the management, monitoring, and integration capabilities of the product. Backend systems can be implemented on on-premises hardware or cloud providers, again depending on the project requirements. Moreover, this is where IoT encounters other disruptive technologies. You can apply big data analytics to extract more

in-depth information from data coming from sensors, or you can use AI algorithms to feed your system with more smart features, such as anomaly detection or predictive maintenance.

- **End user applications**: You will very likely require an interface for your end users to let them access the functionality. Ten years ago, we were only talking about desktop, web, or mobile applications. But today we have voice assistants. You can think of them as a modern interface for human interaction, and it might be a good idea to add voice assistant integration as a feature, especially in the consumer segment.

This is the list of aspects, more or less, that we need to take into account in many types of IoT projects before starting.

The following diagram depicts the general structure of IoT solutions:

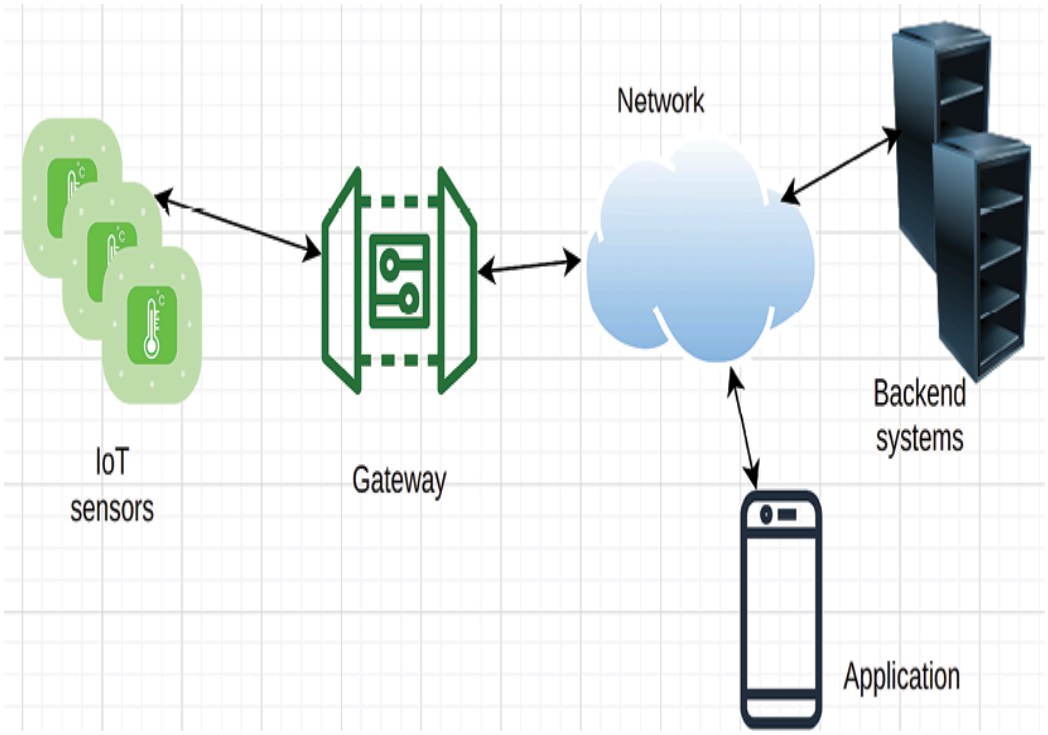


Figure 1.1: Basic structure

Comparing this figure to the layers we discussed above, IoT sensors or devices carry the device hardware and firmware layers. The communication layer can be shared between IoT devices and gateways, depending on the selected communication technology and the capabilities of the IoT device hardware. An IoT solution doesn't have to have a gateway, for instance, if the SoC on the IoT devices already supports the network protocol in the solution. Please remember that this structure doesn't represent the entire IoT world; it cannot. The requirements of a project decide the optimal architecture of an IoT solution.

IoT security

One important consideration that remains is security. I cannot overemphasize its importance. IoT devices are connected to the real world and any security incident has the potential for causing serious damage in the immediate environment, let alone other cybersecurity crimes. Therefore, it should always be on your checklist while designing any hardware or software components of the solution. Although security, as a subject, definitely deserves a book by itself, I can list some golden rules for devices in the field:

- Always look to reduce the attack surface for both hardware and firmware.
- Prevent physical tampering wherever possible. No physical port should be open if this is not necessary.
- Keep secret keys on a secure medium.
- Implement secure boot, secure firmware updates, and encrypted communication.

- Do not use default passwords; TCP/IP ports should not be open unnecessarily.
- Put health check mechanisms in place along with anomaly detection where possible.

We should embrace secure design principles in general as IoT developers. Since an IoT product has many different components, end-to-end security becomes the crucial point while designing the product. A risk impact analysis should be done for each component to decide on the security levels of data in transit and data at rest. There are many national/international institutions and organizations that provide standards, guidelines, and best practices regarding cybersecurity. One of these, which works specifically on IoT technology, is the **IoT Security Foundation**. They are actively developing guidelines and frameworks on the subject and publish many of those guidelines, which are freely available. If you want to check those guidelines, you can visit the IoT Security Foundation website for their publications here: https://www.iotsecurityfoundation.org/best-practice-guidelines/.

Now that we are equipped with sufficient knowledge of IoT and its applications, we can propel our journey with ESP32, a platform perfectly suited for beginner-level projects as well as end products. In the remaining sections of this chapter, we are going to talk about the ESP32 hardware, development frameworks, and RTOS options available on the market.

The ESP32 product family

Since the launch of the first ESP32 chip, Espressif Systems has extended the family with new designs for different purposes. They now have more than 200 different SoCs and modules in the inventory. Although it is impossible to discuss each of them one by one, we can talk about the ESP32 product family in general to understand their intended use cases. It is always a good idea to check what we need and what is available in the arsenal before starting a new IoT project.

There is an online tool on the Espressif website to find an SoC/module by filtering them based on selected features. You check the filter boxes, and the tool lists the hardware matching your requirements: https://products.espressif.com/#/product-selector

The following is a basic checklist for selecting an SoC/module:

- Performance requirements (core frequency, single/double core)
- Memory requirements (RAM, ROM)
- Embedded flash requirements
- Power requirements (power consumption at different power modes, low-power co-processor)
- Cost requirements
- Wi-Fi and/or BLE requirements (including antenna type)
- Peripherals (number of GPIOs, other peripherals)
- Physical environment (operating temperature, humidity)
- Security requirements

- Package type/size

You can extend this list for your specific project, but it is usually good enough to meet these requirements in many cases. With that, we can look at the different series of SoCs in the ESP32 product family from Espressif Systems.

ESP32 series

This series is the first product in the ESP32 product family and gives its name to the entire family of SoCs. It contains the most common variants of chips in the ESP32 product family. Let's have a quick look at the functional block diagram on its datasheet:

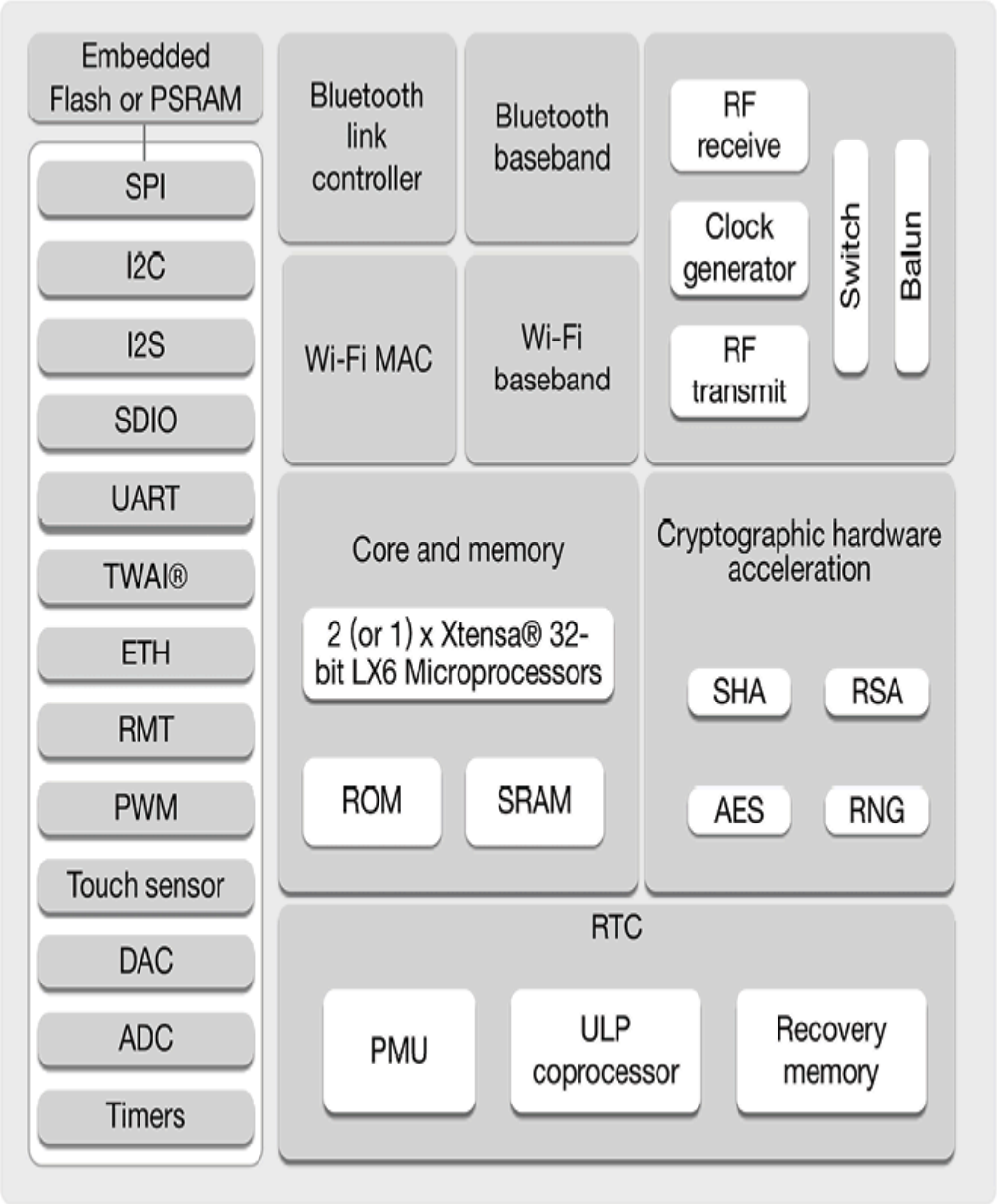


Figure 1.2: The functional block diagram of ESP32 series chips (source: ESP32 Series Datasheet, Version 3.9, Espressif Systems)

The key features are the following:

- It has an Xtensa 32-bit LX6 core. It can have 1 or 2 cores, which means there are variants based on the number of cores that an SoC has.

- It has ROM and SRAM memories (448 KB of ROM and 520 KB of on-chip SRAM to be exact).
- It can have embedded flash or **Pseudostatic-Ram** (**PSRAM**), which implies there are more variants here.
- It has an incredible range of peripherals (with 34 GPIOs)
- It has integrated RF components for Wi-Fi (802.11 b/g/n – 2.4 GHz) and Bluetooth (BLEv4.2 and BR/EDR) communication
- It has cryptographic hardware acceleration
- It has low-power management features with a **Real-Time Clock** (**RTC**) and an **ultra-low-power** (**ULP**) co-processor

The variants have different part numbers, and it is good to know this numbering convention (at least the existence of it since you can always open and read the datasheet) when ordering or talking to technical support. The modules and development kits also specify the SoC part number, which is useful to understand the capabilities of the device. Let's see how it works.

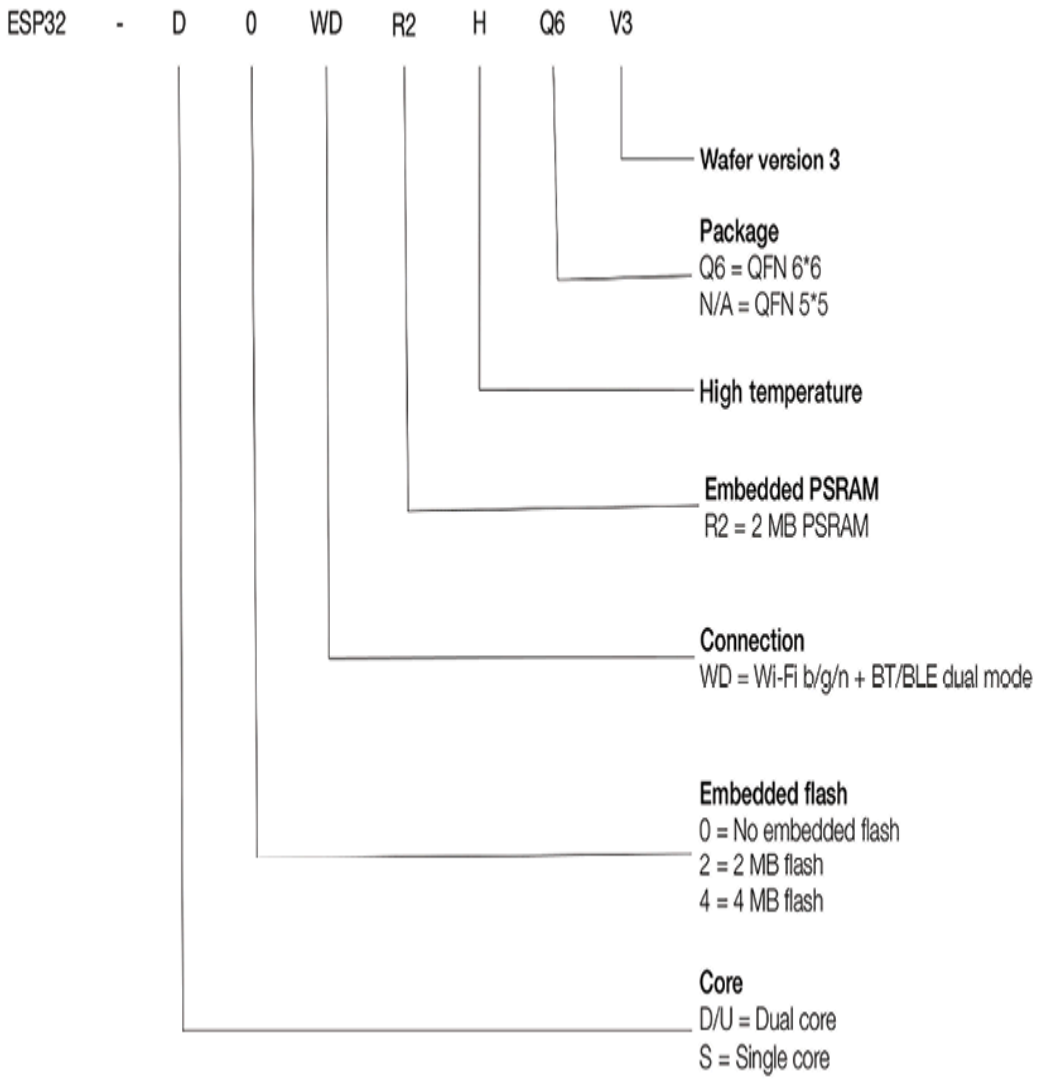


Figure 1.3: ESP32 part numbering convention (source: ESP32 Series Datasheet, Version 3.9, Espressif Systems)

The fields in the part number show the different features of an SoC, as you can see in the preceding figure. For example, ESP32-D0WDR2-V3 has a dual core, no embedded flash, Wi-Fi and BT/BLE dual mode, and 2 megabytes of PSRAM, and it uses the ECO V3 wafer.

Espressif marks some of their SoCs as **Not recommended for new designs** (**NRND**). It basically means that the part number will be discontinued in the near future. This could be because of some silicon changes on a newer replacement version, or the part is going to end-of-life will probably drop it from the inventory in the future. When you see the NRND label next to an SoC, it is wise to look for something else if you are starting a new IoT project. You can find all the datasheets and technical documentation here: https://www.espressif.com/en/support/documents/technical-documents

I strongly recommend reading the ESP32 series datasheet, which is publicly available on the Espressif website, to learn more about the SoC features in this series. In particular, the pages where the peripherals are talked about are important to understand whether the high-level requirements can be met in an IoT project. The datasheet is also a good starting point to discover and experiment with the technologies coming with ESP32.

Let's see what other series of ESP32 are on offer.

Other SoCs

After the successful launch of the ESP32 series of SoCs, Espressif has continued to answer the needs of IoT developers with new series featuring different technologies. The following is a quick summary:

- **ESP32-S2**: This series boasts superior security features, such as software inaccessible security keys. It has a single-core Xtensa 32-bit LX7 CPU and less memory compared to the SoCs in the ESP32 series, but more variants with embedded PSRAM and flash in different sizes. Espressif removed Bluetooth from this series; instead, it has a full-speed USB-OTG interface that allows us to develop USB devices. Another interesting feature of ESP32-S2 is the LCD and camera interfaces, as can be seen in its peripherals list.
- **ESP32-S3**: The selling point of this series is the support for the **Artificial Intelligence of Things** (**AIoT**) with its high-performance dual-core microprocessor (Xtensa 32-bit LX7). It has everything that the ESP32-S2 series has and more. It supports BLE 5 with enhanced range, more bandwidth, and less power. Again, there are variants with different PSRAM and flash options. The development kit, **ESP32-S3-BOX-Lite**, that we are going to use in this book makes use of the ESP32-S3-WROOM-1-N16R8 module with 16 MB flash and 8 MB PSRAM.
- **ESP32-C2**: This targets the same market as ESP8266, the very first product of Espressif Systems on the market, with its low-cost approach yet enhanced feature set. It employs a RISC-V 32-bit single-core processor. Despite its low cost, it provides Wi-Fi (802.11b/g/n) and BLE5 connectivity with a good enough peripheral for basic IoT applications.
- **ESP32-C3**: This series shares the same processor as ESP32-C2 but has more memory and better security features to make it suitable for cloud applications. The other development kit in this book, **ESP32-C3-DevKitM-1**, uses an SoC from this series.

- **ESP32-C6**: Probably the most interesting thing about the ESP32-C6 series is its connectivity features. In addition to Wi-Fi (802.11b/g/n) and BLE5, it supports Wi-Fi 6 (802.11ax), the new Wi-Fi standard to support more devices in a network with more bandwidth. It also adds IEEE 802.15.4 radio connectivity, which enables Zigbee and Thread as **Wireless Personal Area Networks** (**WPANs**) to be selected as local communication infrastructure in products.
- **ESP32-H2**: With this series, Espressif turns a new page. ESP32-H2 doesn't have Wi-Fi on it! Instead, ESP32-H2 combines the IEEE 802.15.4 and Bluetooth 5 radio protocols on the same chip. IEEE 802.15.4 defines the physical and media access layers for a **low-rate Wireless Personal Area Network** (**LR-WPAN**). Zigbee, Thread, and several other WPAN protocols operate on top of IEEE 802.15.4.

Apart from the SoCs, Espressif also manufactures modules with different SoCs. They are ready-to-assemble parts with integrated antennas or antenna connectors as well as external flash and SPIRAMs for different needs. These modules remove a lot of hassle for hardware designers while working on a new IoT device.

There are countless scenarios in the IoT world, but I believe you can find the right SoC solution for your project from this wide range of ESP32 products in many cases.

After this hardware review, let's look at what we have on the software side to drive the hardware, in the next section.

Development platforms and frameworks

ESP32 is quite popular. Therefore, there are a good number of options that you can select as your development platform and framework.

The first framework, of course, comes directly from Espressif itself. They call it the **Espressif IoT Development Framework** (**ESP-IDF**). It supports all three main OS environments – Windows, macOS, and Linux. After installing some prerequisite packages, you can download the ESP-IDF from the GitHub repository and install it on your development PC. They have collected all the necessary functionality into a single Python script, named `idf.py`, for developers. You can configure project parameters and generate a final binary image by using this command-line tool.

You can also use it in every step of your project, starting from the build phase to connecting and monitoring your ESP32 board from the serial port of your computer. If you are a more graphical UI person, then you need to install Visual Studio Code as an IDE and install the ESP-IDF extension on it. You can find the ESP-IDF documentation at this link: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html

The second option is the Arduino IDE and **Arduino Core for ESP32**. If you are familiar with Arduino, you'll know how easy it is to use. However, it comes at the cost of development flexibility compared to

ESP-IDF. You are constricted in terms of what Arduino allows you to do and you need to obey its rules.

The third alternative you can choose is **PlatformIO**. This is not a standalone IDE or tool but comes as an extension in Visual Studio Code as an open-source embedded development environment. It supports many different embedded boards and frameworks, including ESP32 boards and ESP-IDF. Following installation, it integrates itself with the VSCode UI, where you can find all the functionality that `idf.py` of ESP-IDF provides. In addition to VSCode's IDE features, PlatformIO has an integrated debugger, unit testing support, static code analysis, and remote development tools for embedded programming. PlatformIO is a good choice for balancing ease of use and development flexibility.

The programming language for those three frameworks is **C/C++**, so you need to know C/C++ in order to develop within those frameworks. However, C/C++ is not the only programming language for ESP32. You can use **MicroPython** for Python programming or **Espruino** for JavaScript programming. They both support ESP32 boards, but to be honest, I wouldn't use them to develop any product to be launched on the market. Although you may feel more comfortable with them because of your programming language preferences, you won't find ESP-IDF capabilities in either of them. **Rust** is another option as a programming language to develop IoT applications on ESP32. It is getting some attention in the embedded world and seems worth trying on ESP32. However, in this book, we will develop the applications in C++11 as the modern C++ standard supported by ESP-IDF.

Throughout the book, we will use Visual Studio Code as the IDE and ESP-IDF as the development framework with its integral tools. PlatformIO is also very popular among developers, and we will see some of its features in the next chapter.

We will discuss the real-time operating systems available for ESP32 next.

RTOS options

Basically, a **real-time operating system** (**RTOS**) provides a deterministic task scheduler. Although the scheduling rules change depending on the scheduling algorithm, we know that the task we create will complete in a certain time frame within those rules. The main advantages of using an RTOS are the reduction in complexity and improved software architecture for easier maintenance.

The main real-time operating system supported by ESP-IDF is **FreeRTOS**. ESP-IDF uses its own version of the Xtensa port of FreeRTOS. The fundamental difference compared with vanilla FreeRTOS is the dual-core support. In ESP-IDF FreeRTOS, you can choose one of two cores to assign a task, or you can let FreeRTOS choose it. Other differences compared with the original FreeRTOS mostly stem from the dual-core support. FreeRTOS is distributed under the MIT license. You can find the ESP-IDF FreeRTOS documentation at this URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html

If you want to connect your ESP32 to the **Amazon Web Services** (**AWS**) IoT infrastructure, you can do that by using **Amazon FreeRTOS** as your RTOS choice. ESP32 is in the AWS partner device catalog and is officially supported. Amazon FreeRTOS brings the necessary libraries together to connect to AWS IoT and adds other security-related features, such as TLS, OTA updates, secure communication with HTTPS, WebSocket, and MQTT – pretty much everything needed to develop a secure, connected device. An example of using Amazon FreeRTOS in an ESP32 project is given here: https://freertos.org/quickconnect/index.html

Zephyr is another RTOS option with a permissive free software license, Apache 2.0. Zephyr requires an ESP32 toolchain and ESP-IDF installed on the development machine. Then, you need to configure Zephyr with them. When the configuration is ready, we use the command-line Zephyr tool, **west**, for building, flash, monitoring, and debugging purposes.

If you are a fan of Apache Software, then **NuttX** is also an option as an RTOS in your ESP32 projects. As it is valid for other Apache products, standards compliance is a paramount driving factor in the NuttX design. It shows a high degree of compliance with POSIX and ANSI standards; therefore, the APIs provided in NuttX are almost the same as defined in them.

The last RTOS that I want to share here is **Mongoose OS**. It provides a complete development environment with its web UI tool, **mos**. It has native integration with several cloud IoT platforms, namely, AWS IoT, Google IoT, Microsoft Azure, and IBM Watson, as well as any other IoT platform that supports MQTT or REST endpoints if

you need a custom platform. Mongoose OS comes with two different licenses, one being an Apache 2.0 community edition, and the other an enterprise edition with a commercial license.

Our choice in this book will be FreeRTOS and we will also discuss Amazon FreeRTOS features when we connect to AWS IoT.

Summary

In this first chapter, we discussed what the **Internet of Things** (**IoT**) is in general and its building blocks to understand the basic structure of IoT solutions. One of the building blocks is device hardware that interacts with the physical world. **System-on-chip** (**SoC**) is a major component while designing an IoT device. Espressif Systems empowers us, as IoT developers, with its **ESP32** product family, which includes many SoCs with different configurations in response to the needs of IoT projects. In addition to hardware, Espressif maintains the **Espressif IoT Development Framework** (**ESP-IDF**), the main software framework to develop IoT applications on ESP32 SoCs. ESP-IDF comes with all the necessary tools to deliver full-fledged IoT products. We also talked about **real-time operating system** (**RTOS**) options that we can select from in ESP32 development. ESP-IDF integrates **FreeRTOS**, a prominent RTOS option in the embedded world, as a component.

Following the background information in this chapter, upcoming chapters are going to focus on how to use ESP32 effectively in our projects. With the help of practical examples and explanations, we will see different aspects of ESP32 for different use cases and apply

them in the projects at the end of each part. We will begin by using sensors and actuators in the next chapter.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

2

Understanding the Development Tools

After getting a quick overview of Espressif's ESP32 technology in the first chapter, we are now ready to start on some development with the real hardware. For this, we need to understand the basics and how to use the available tools for the job. It is a learning process and takes some time; however, we'll have acquired the fundamental knowledge and gained hands-on experience to develop actual applications on ESP32 by the end of the chapter.

In this chapter, we're going to install the development environment on our machines and use our development kits to run and debug the applications. The topics covered are as follows:

- ESP-IDF
- PlatformIO
- FreeRTOS
- Debugging
- Unit testing

Let's start by looking into the development framework by Espressif Systems, *ESP-IDF*.

Technical requirements

Throughout the book, we're going to use **Visual Studio Code** (**VSCode**) as our **Integrated Development Environment** (**IDE**). If you don't have it, you can find it here: https://code.visualstudio.com/. VSCode is updated monthly, but its version is **1.77.3** as of writing this book.

The main development framework that we will use in this chapter and in the book is **ESP-IDF v4.4.4**. It is available on GitHub here: https://github.com/espressif/esp-idf. If you are a Windows user, you can download the installer from this link as another installation option: https://dl.espressif.com/dl/esp-idf/.

ESP-IDF requires **Python 3** to be installed on the development machine. We will also need Python in some of the examples. The Python version that is used is **3.10.11**.

The examples in the book are developed on Ubuntu 22.04.2 LTS (Linux 5.15.0-71-generic x86_64). However, they compile regardless of the development platform after the installation of ESP-IDF on the development machine.

For hardware, we need both devkits in this chapter, **ESP32-C3-DevKitM-1** and **ESP32-S3-Box-Lite**.

The source code in the examples is located in the repository found at this link: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch2

ESP-IDF

ESP-IDF is the official framework by Espressif Systems. It comes with all the necessary tools and an SDK to develop ESP32-based products. After installing ESP-IDF, we will have:

- An SDK for ESP32 development: **ESP-IDF v4.4.4** supports the ESP32, ESP32-S2, ESP32-S3, and ESP32-C3 families of chips.
- GCC-based toolchains for the **Xtensa** and **RISC-V** architectures.
- **GNU debugger** (**GDB**).
- Toolchain for the ESP32 **Ultra Low Power** (**ULP**) coprocessor.
- **cmake** and **ninja** build systems.
- **OpenOCD** for ESP32.
- Python utilities: The most notable one is **idf.py** and is the one we will use throughout the book. It collects all the development tasks in the same Python script.

The installation of ESP-IDF differs from platform to platform, so make sure to follow the steps as described in the official documentation for your target development machine. The documentation is provided at this link: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/#installation

My personal choice of development environment is VSCode on Canonical Ubuntu 22.04 LTS. However, you can install ESP-IDF on any platform without any problem if you follow the guidance in the official documentation. The other IDE option is to use ESP-IDF through it.

On top of ESP-IDF, Espressif offers a VSCode extension in the Visual Studio Marketplace to manage ESP32 projects in a similar fashion to

the command-line tools. It makes development in the VSCode environment easier but its capabilities are limited compared to the command-line tools. Its installation is explained here: https://github.com/espressif/vscode-esp-idf-extension. We will use this extension with VSCode to develop the very first ESP32 application.

The first application

In this example, we will simply print `'Hello World'` on the serial output of ESP32 – surprise! Let's do it step by step:

1. Make sure you have installed **ESP-IDF v4.4.4**. Follow the steps as described in the documentation (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/#installation).
2. Make sure you have installed **VSCode** and the **Espressif IDF** extension (https://github.com/espressif/vscode-esp-idf-extension).
3. Plug ESP32-S3-BOX-Lite into a USB port of your development machine and observe that it is shown on the device list (the following command is on a Linux terminal but you can choose any relevant relevant tool on your development machine):

```
$ lsusb | grep -i espressif
Bus 001 Device 025: ID 303a:1001 Espressif USB J
```

4. Run VSCode and enable the ESP-IDF extension if it is not already enabled.

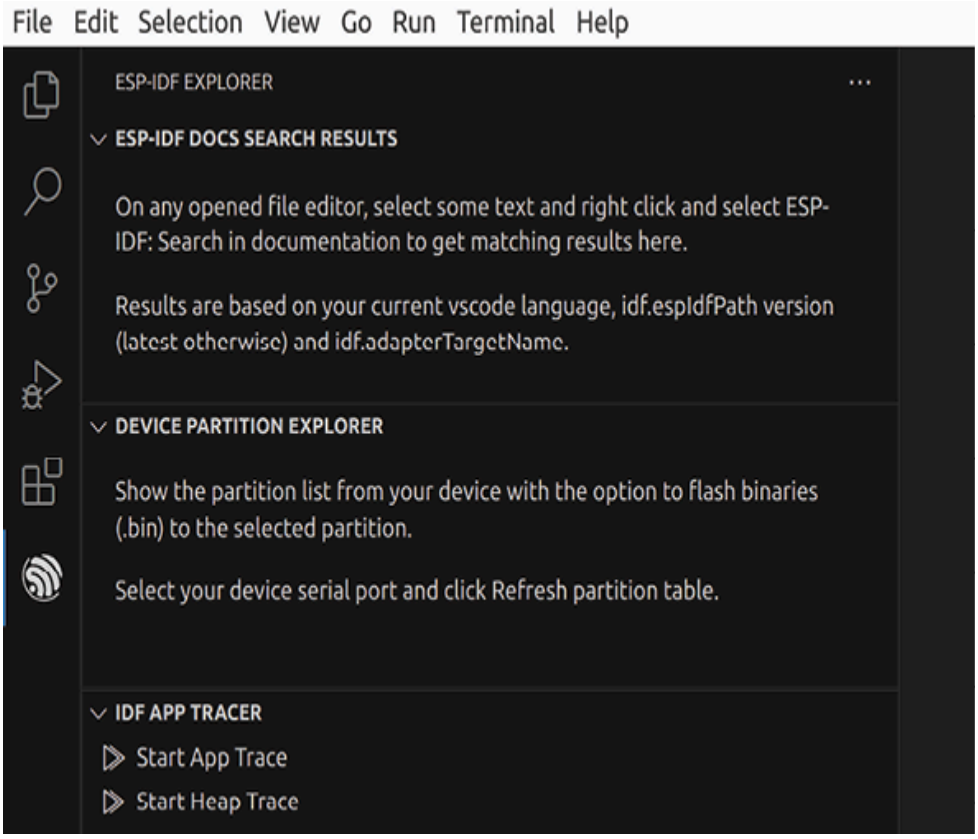


Figure 2.1: Espressif IDF in VSCode

5. Create an ESP32 project. There are two ways to do that. You can either select **View | Command Palette | ESP-IDF: New Project** or press the *Ctrl + E + N* key combination to open the **New Project** dialog. Fill in the input boxes as shown in the following screenshot (don't use any whitespace in the project path, and select ESP32-S3 and the serial port that the devkit is connected to). Then click on the **Choose Template** button.

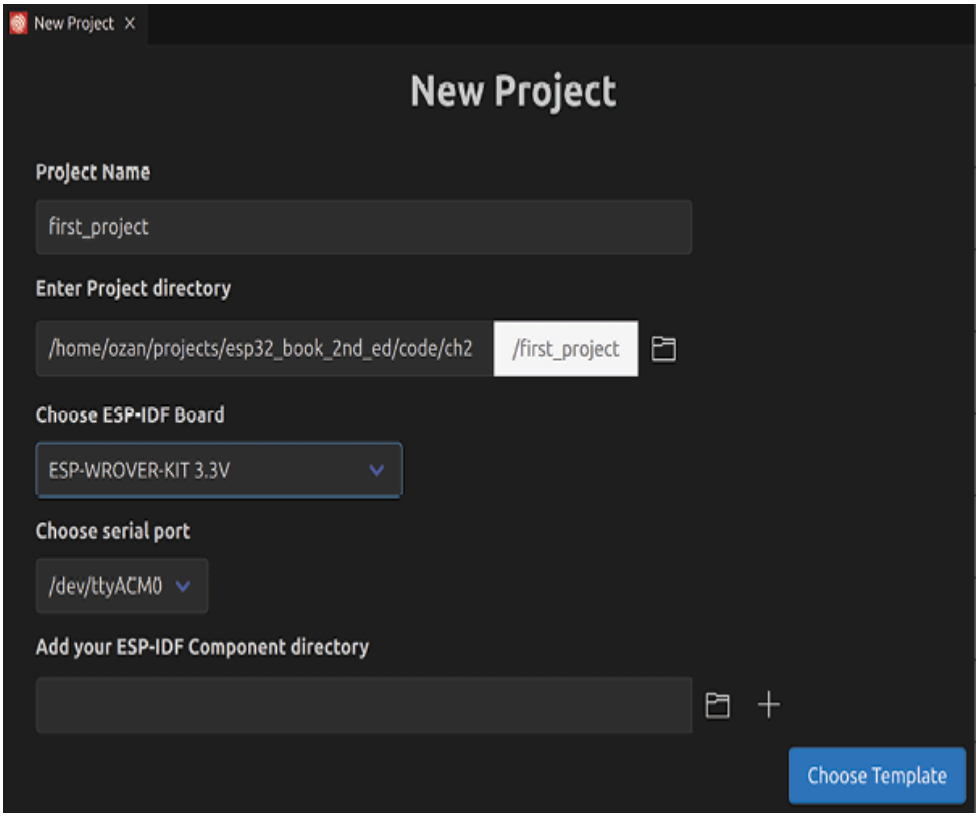


Figure 2.2: New project

6. On the next screen, select **ESP-IDF** from the drop-down box and **sample_project** from the list. Complete this step by clicking on the **Create project using template sample_project** button. On the bottom right of the screen, a pop-up window will show up asking to open the project in a new window. Answer **Yes** and a new VSCode window will appear with the new project.

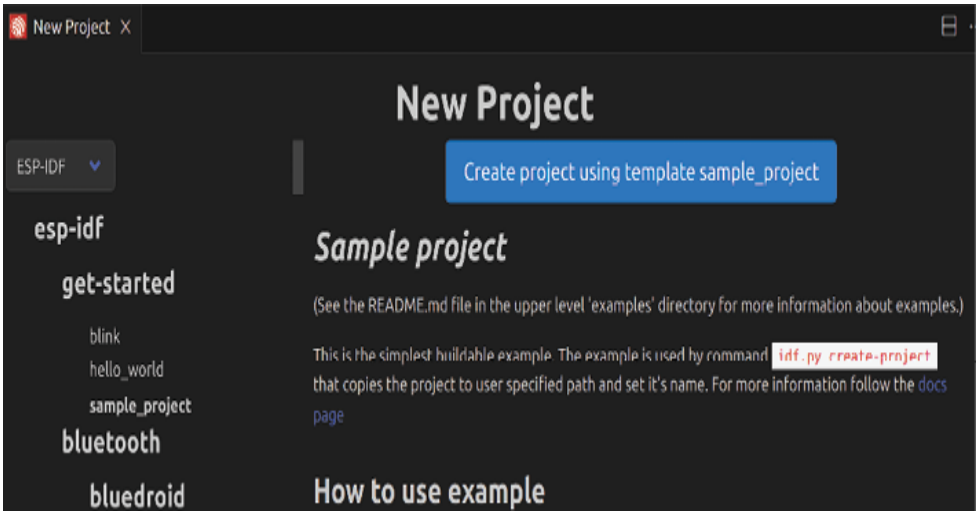


Figure 2.3: Select sample_project

7. In the new VSCode window, you now have the environment for the ESP32 project. You can see the file names in Explorer on the left side of the window.

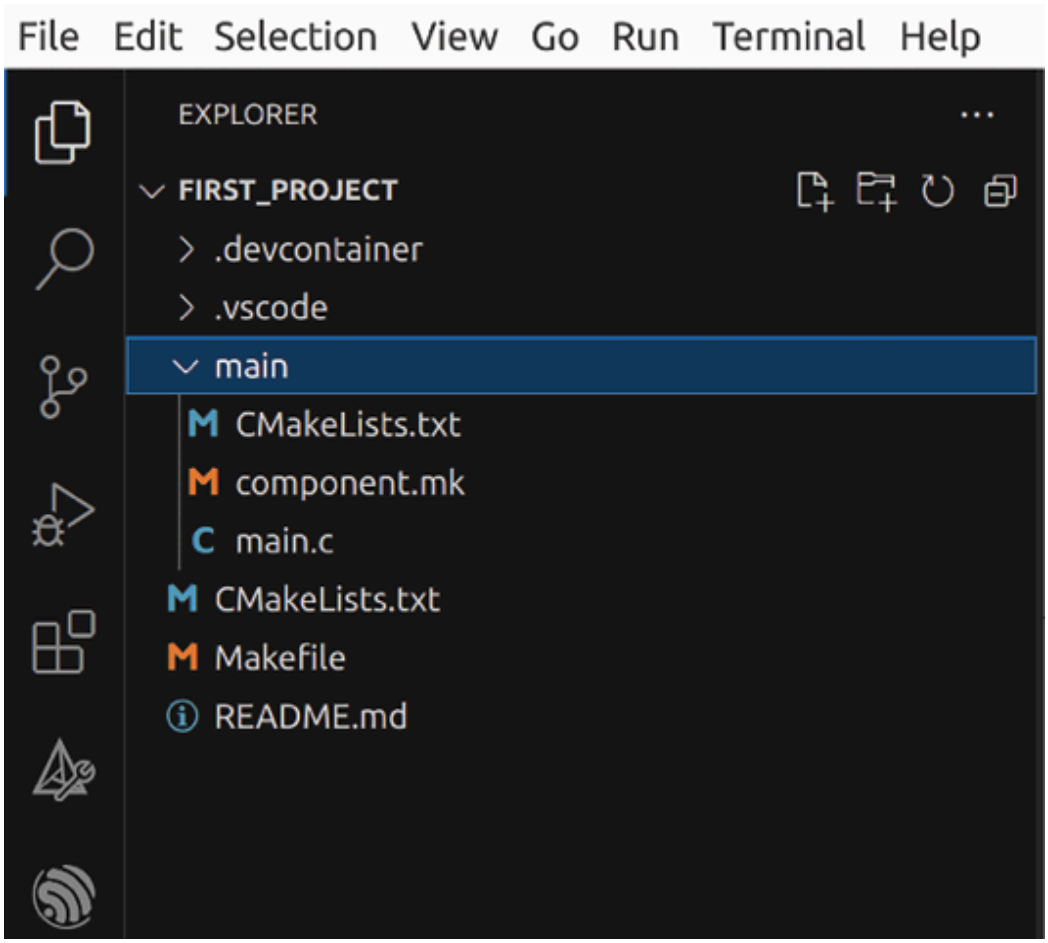


Figure 2.4: Explorer with VSCode files

8. Rename `main/main.c` to `main/main.cpp` and edit the content with the following simple and famous code:

```
#include <iostream>
extern "C" void app_main(void)
{
    std::cout << "Hello world!\n";
}
```

9. To compile, flash, and monitor the application, we can simply press the *Ctrl* + *E* + *D* key combination (there are other ways to

the same thing: we can click on the fire icon in the bottom menu or we can select the same CP in the command palette in VSCode). The ESP-IDF extension will ask you to specify the flash method. There are three options: **JTAG, UART and DFU**. We select **UART**.

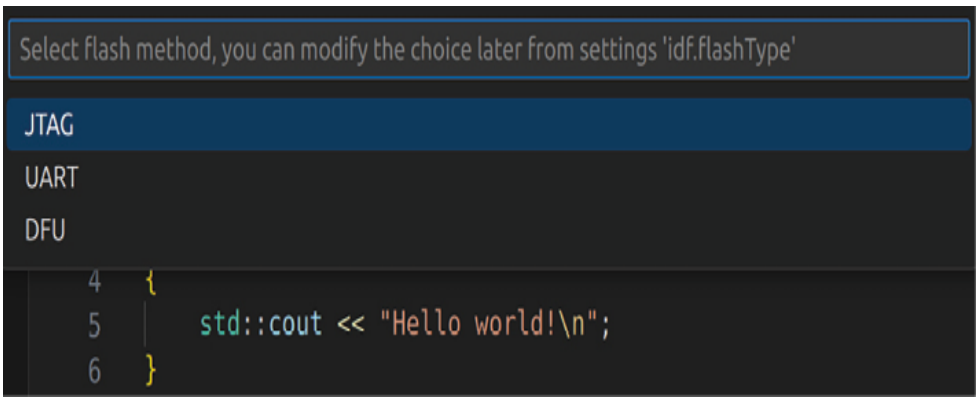


Figure 2.5: Selecting the flash method

10. Then the next step is to select the project – we only have `first_example`. VSCode will open a terminal tab where you can see the compilation output. If everything goes well, it will connect to the serial monitor and all the logs will be displayed as follows.

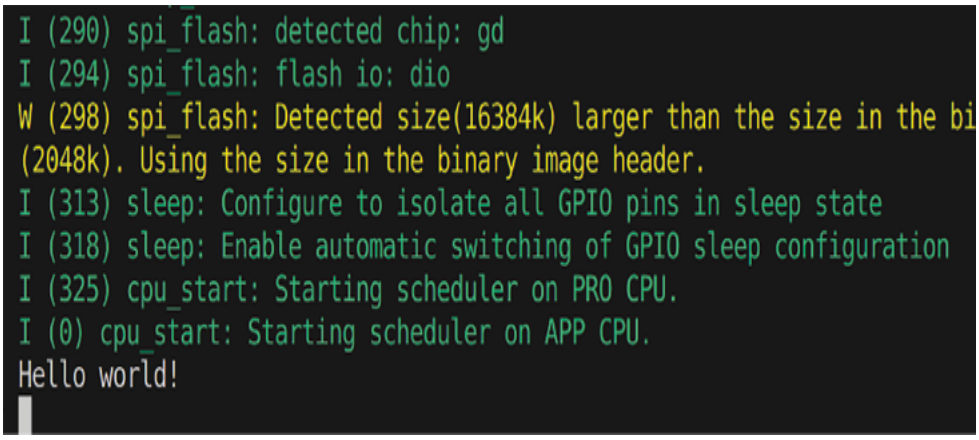


Figure 2.6: The serial monitor output

11. We have compiled the application, flashed it to the devkit, and can see its output in the serial monitor. To close the serial monitor, press *Ctrl +]*.

Let's go back to the application code in *step 8* and discuss it briefly. In the first line, we include the C++ `iostream` header file to be able to use the standard output (`stdout`) to print text on the screen:

```
#include <iostream>
```

In the next line, we define the application entry point:

```
extern "C" void app_main(void)
{
```

`extern "C"` means that we will next add some C code and the C++ compiler will not mangle the symbol name that comes after this. It is `app_main` here. The `app_main` function is the entry point of ESP32 applications, so we will have this function in every ESP32 application that we develop. We only print `"Hello world!\n"` on the standard output in `app_main`:

```
    std::cout << "Hello world!\n";
} // end of app_main
```

The standard output is redirected to the serial console by default, thus we see **Hello world!** printed on the screen in *step 10* when we monitor ESP32 by connecting its serial port. We've completed our

first ESP32 application. We can use these steps as a blueprint when starting a new project. Let's discuss more about the internal workings of ESP-IDF and other files in a typical ESP32 project.

ESP-IDF uses *cmake* as its build configuration system. Therefore, we see the `CMakeLists.txt` files in various places. The one in the root defines the ESP-IDF project:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(first_project)
```

The other one is `main/CMakeLists.txt`. It registers the application component by providing the source code files and include directories. In our case, it is only `main.cpp` and the current directory of `main.cpp` where it will look for any header files:

```
idf_component_register(SRCS "main.cpp"
                       INCLUDE_DIRS ".")
```

We frequently edit this file to add new source code files and include directories in our projects.

If you are not familiar with the *cmake* tool, you can visit its documentation at this link: https://cmake.org/cmake/help/latest/. We will discuss the `CMakeLists.txt` files in the example projects but if you want a preview of how to configure

an ESP-IDF component, the documentation is here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html#minimal-component-cmakelists

You may have noticed that when we compiled the application, new additions appeared in the project root as can be seen in the following figure:

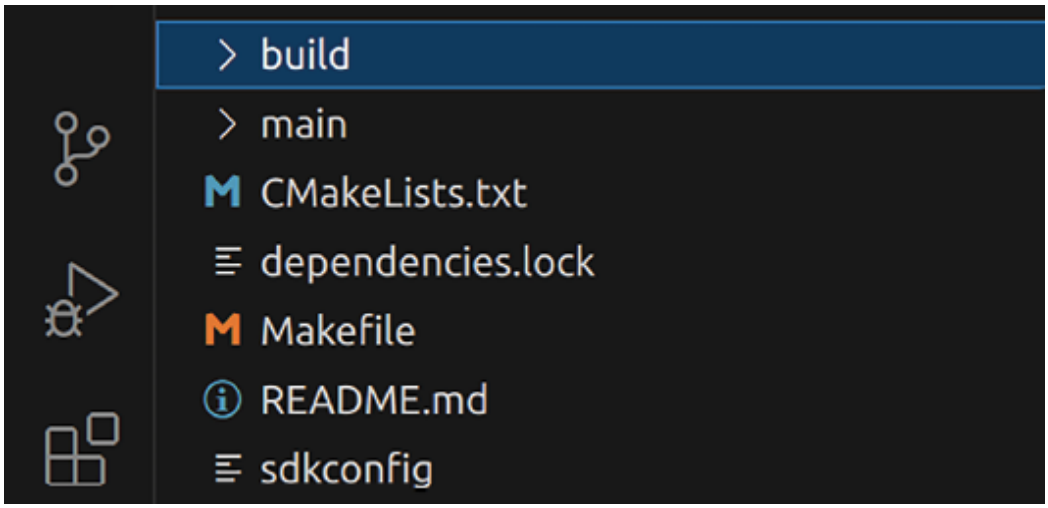


Figure 2.7: The files and directories after compilation

We can see a new directory, `build`, and a file, `sdkconfig`. As the name implies, all the artifacts from the build process are located under the `build` directory. The tools used during the build process generate their files in there. The most interesting files are probably the JSON files. Let's list them:

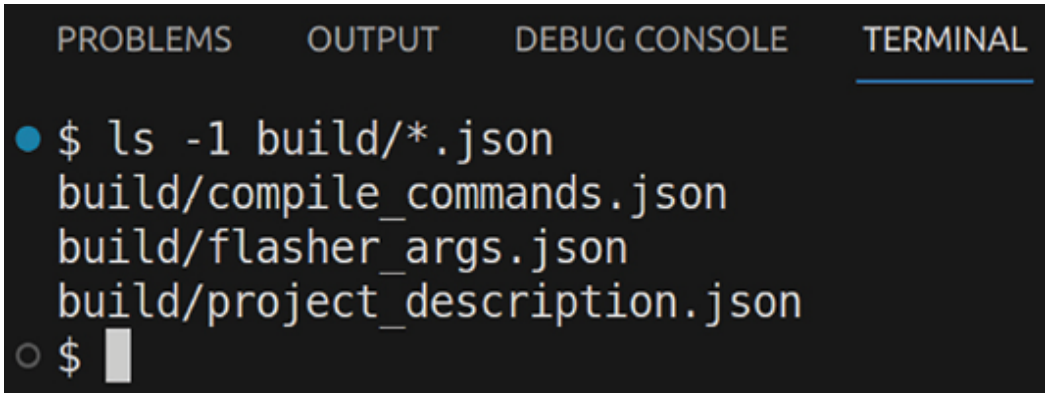


Figure 2.8: JSON files in the build directory

The names of these files are self-explanatory regarding their content so I don't want to repeat them. However, I suggest you check their content after building the project since they can be very helpful for troubleshooting purposes and understanding the build system in general.

`sdkconfig` in the project root is important. It is generated by ESP-IDF automatically during the build if there is none. This file contains the entire configuration of the application and defines the default behavior of the system. The editor is `menuconfig`, which is accessible from **View | Command Palette | ESP-IDF: SDK Configuration editor** (`menuconfig`) or simply *Ctrl + E + G*:

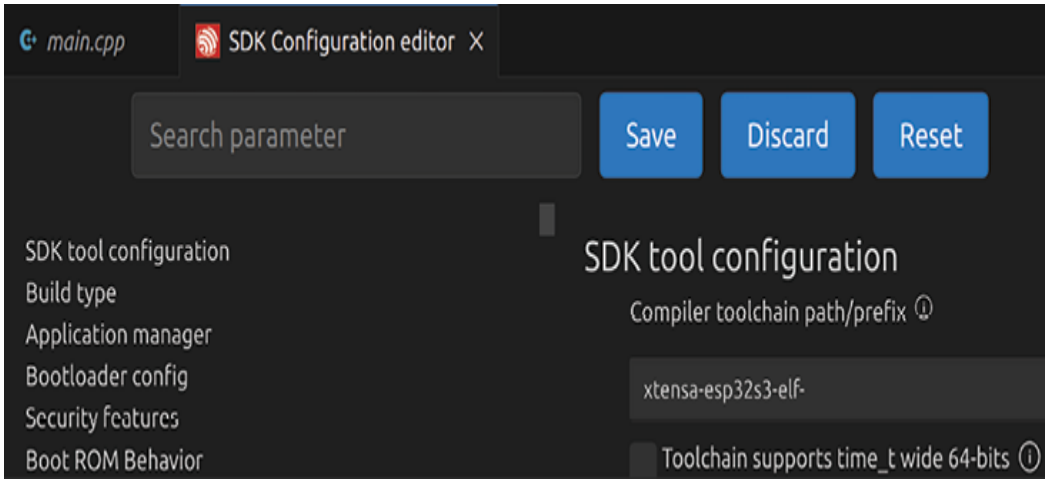


Figure 2.9: SDK configuration (sdkconfig)

We will use `menuconfig` often to configure our projects, and even provide our custom configuration items to be included in `sdkconfig` (`menuconfig` is a part of the Linux configuration system, `Kconfig`, and adapted in ESP-IDF to configure ESP32 applications).

ESP-IDF Terminal

One last thing worth noting with the ESP-IDF extension is the **ESP-IDF Terminal**. It provides command-line access to the underlying Python scripts that come with the ESP-IDF installation. To open a terminal, we can select **View | Command Palette | ESP-IDF: Open ESP-IDF Terminal** or press the key combination of *Ctrl + E + T*. It opens an integrated command-line terminal. The two powerful and popular tools are `idf.py` and `esptool.py`. We can manage the entire development environment and build process by only using `idf.py` and we will use this tool a lot throughout the book. Just go

ahead and write `idf.py` in the terminal to see all the options. As a quick example:

```
$ idf.py clean flash monitor
Executing action: clean
Running ninja in directory <your_directory>/ch2/firs
Executing "ninja clean"...
[0/1] Re-running CMake...
-- Project is not inside a git repository, or git re
-- Building ESP-IDF components for target esp32s3
-- Project sdkconfig file <your_directory>/ch2/first
-- App "first_project" version: 1
<The rest of the build and flashing logs. Next comes
I (0) cpu_start: Starting scheduler on APP CPU.
Hello world!
```

This simple command cleans the project (removes the previous build output if any), compiles the application, flashes the generated firmware to the devkit, and finally starts the serial monitor to show the application output.

Similarly, you can see what `esptool.py` can do by writing its name and pressing *Enter* in the terminal. The main purpose of this tool is to provide direct access to the ESP32 memory and low-level application image management. As an example, we can use `esptool.py` to flash the application binary:

```
$ esptool.py --chip esp32s3 write_flash -z 0x10000 h
esptool.py v3.2
```

```
Found 1 serial ports
Serial port /dev/ttyACM0
Connecting...
Chip is ESP32-S3
Features: WiFi, BLE
Crystal is 40MHz
MAC: 7c:df:a1:e8:20:30
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Flash will be erased from 0x00010000 to 0x00079fff.
Compressed 430672 bytes to 205105...
Wrote 430672 bytes (205105 compressed) at 0x0001000(
Hash of data verified.
Leaving...
Hard resetting via RTS pin…
```

The `esptool.py` tool is especially useful for production. It directly communicates with the ROM bootloader of Espressif SoCs and is independently available on *PyPI* so we can install it on any machine for firmware binary management with a simple command, without using the entire ESP-IDF:

```
$ pip install esptool
```

The Python installation adds two other tools:

- **espefuse.py** for reading/writing eFuses of ESP32

- **espsecure.py** for managing secure boot and flash encryption

We will not use `esptool.py` and the other two that come with the installation in the examples but you will probably need them in production to flash the firmware images on your IoT devices and enable security features. The official documentation for these tools can be found at this link: https://docs.espressif.com/projects/esptool/en/latest/esp32/index.html

With this overview under your belt, you are now ready to start using VSCode and the ESP-IDF extension together to develop ESP32 projects. The next tool is *PlatformIO*.

PlatformIO

PlatformIO supports many different platforms, architectures, and frameworks with modern development capabilities. It comes as an extension in VSCode and so is very easy to install and configure with just a few clicks. After launching VSCode, go to the VSCode **Extensions** screen (*Ctrl*+*Shift*+*X*) and search for `platformio` in the marketplace. It appears in the first place in the match list. Click on the **Install** button, and that is it:

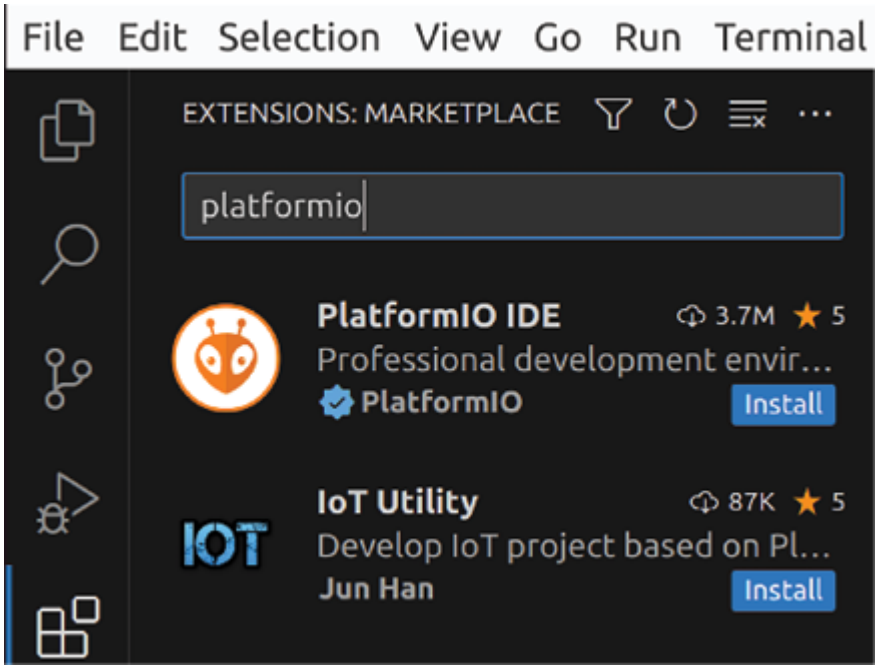


Figure 2.10: Installing PlatformIO

After a few minutes, the installation completes and we have PlatformIO installed in the VSCode IDE. PlatformIO has some unique features. The most notable one is probably the declarative development environment. With PlatformIO, we only need to specify what we're going to use in our project, including the chip type (not limited to Espressif products), which framework and which version of that framework, other libraries with version constraints, and any combination of them. We'll see what all these things mean and how to configure a project shortly. Apart from that, PlatformIO has all the utilities that you would need when developing an embedded project, such as debugging, unit testing, static code analysis, and firmware memory inspection. When I used PlatformIO for the first time roughly 8 years ago, the debug feature was not available in the free version. It is now a free and open-source

project with all features at our disposal. Thank you, guys! Enough talking, let's develop the same application with PlatformIO.

Hello world with PlatformIO

Now, we're going to use PlatformIO. Here are the steps to develop the application:

1. Go to the PlatformIO Home screen.

Figure 2.11: PlatformIO Home

2. Click on the **New Project** button on the right of the same screen.

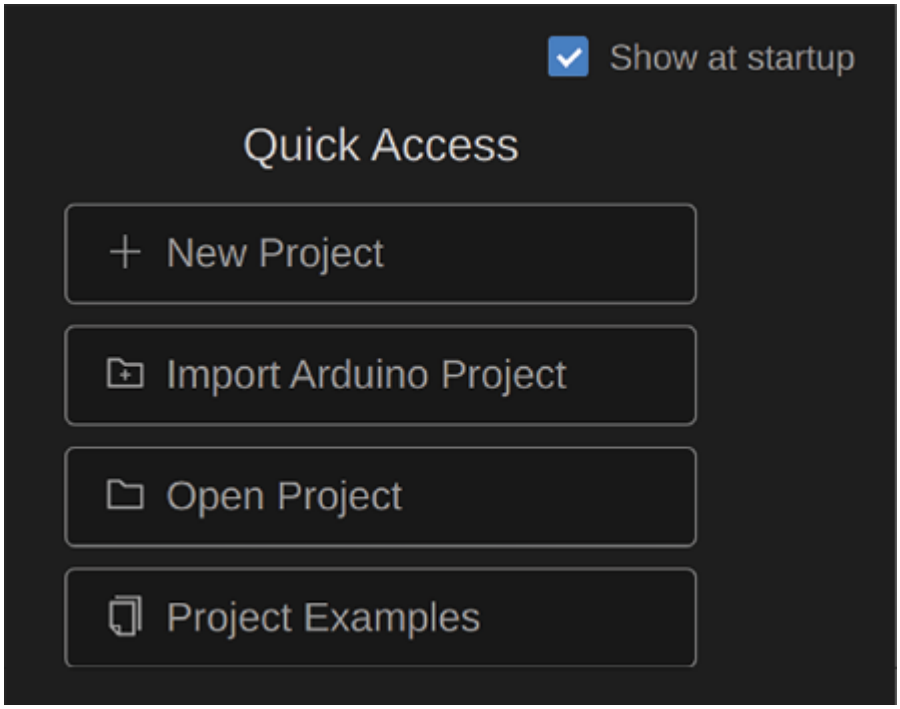


Figure 2.12: Quick access buttons on the PlatformIO Home screen

3. A pop-up window appears. Set the project name, select **Espressif ESP32-S3-Box** for the board, and specify the framework as **Espressif IoT Development Framework**. You can choose a directory for the project or leave it at the PlatformIO default. Click on **Finish** to let PlatformIO do its job.

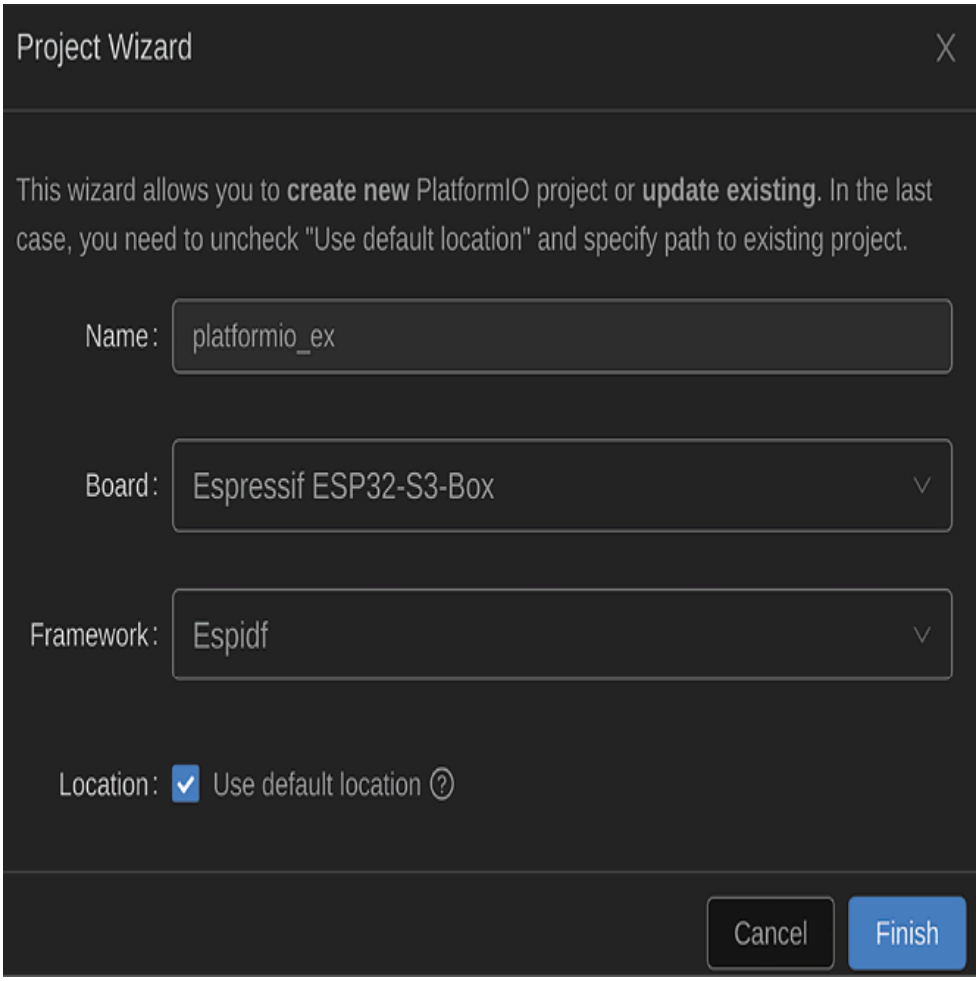


Figure 2.13: PlatformIO Project Wizard

4. When the project is created, we have the following directory structure.

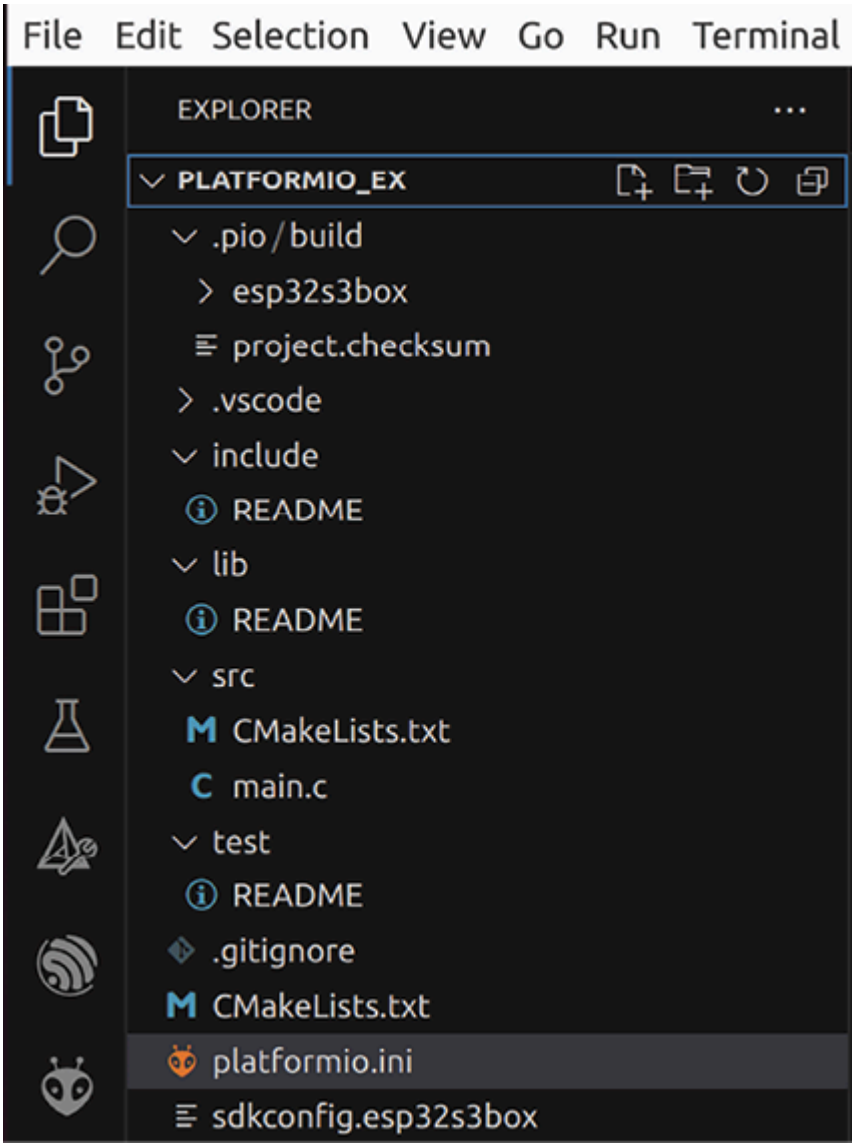


Figure 2.14: Project directory structure

5. Rename `src/main.c` to `src/main.cpp` and copy-paste the same code that we have already developed with the ESP-IDF extension.

```
#include <iostream>
extern "C" void app_main()
{
```

```
    std::cout << "Hello World!\n";
}
```

6. Edit the `platformio.ini` file to have the following configuration settings.

```
[env:esp32s3box]
platform = espressif32
board = esp32s3box
framework = espidf
monitor_speed=115200
monitor_rts = 0
monitor_dtr = 0
```

7. On the PlatformIO tasks list, you will see the **Upload and Monitor** task under the **PROJECT TASKS** | **esp32s3box** | **General** menu. It will build, flash, and monitor the application.

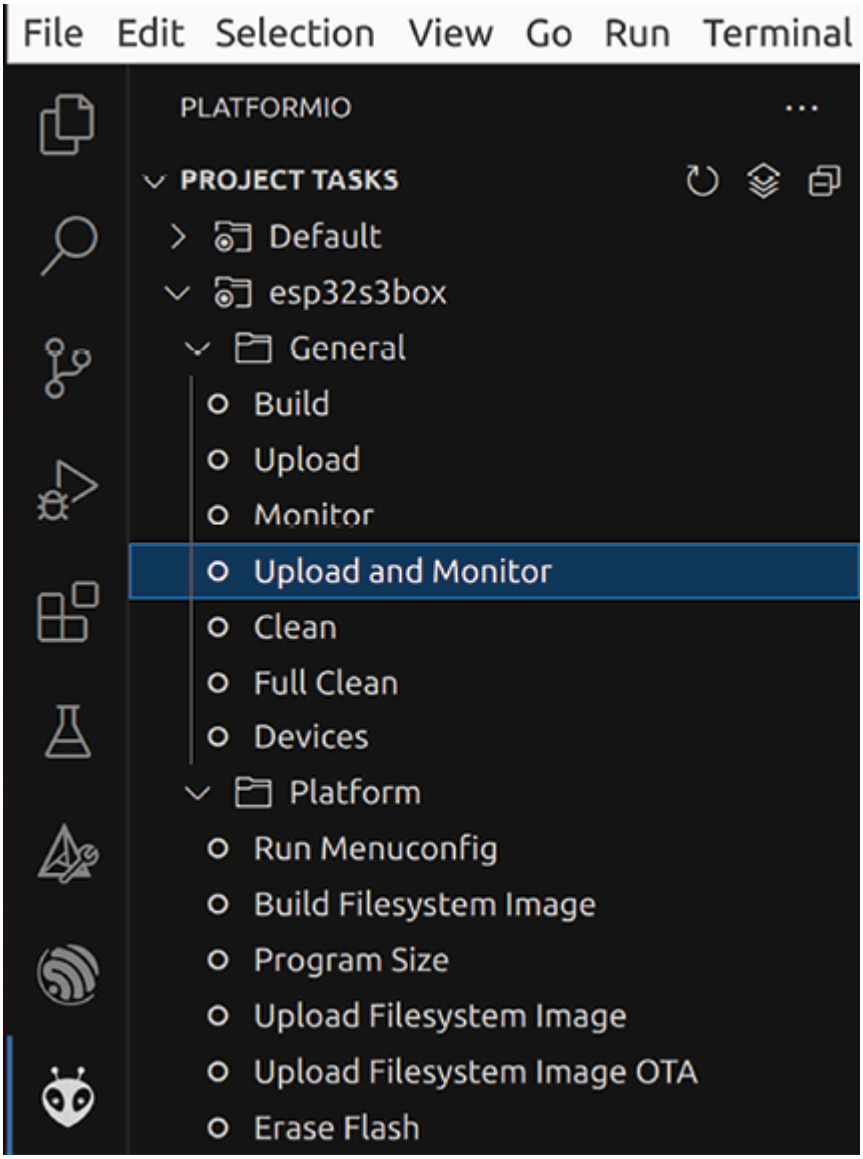


Figure 2.15: PlatformIO project tasks

8. You can observe the application output in the integrated terminal.

```
[0;32mI (334) app_start: Starting scheduler on CPU0[0m
[0;32mI (338) app_start: Starting scheduler on CPU1[0m
[0;32mI (338) main_task: Started on CPU0[0m
[0;32mI (348) main_task: Calling app_main()[0m
Hello World!
[0;32mI (348) main_task: Returned from app_main()[0m
```

Figure 2.16: Application output in the terminal

As you might have already noticed, we didn't download or install anything except PlatformIO. It handled all these low-level configuration and installation tasks for us. PlatformIO uses the `platformio.ini` file for this purpose. Let's investigate its content.

The first line defines the environment. The name of the environment is `esp32s3box`. We can write anything as the environment name:

```
[env:esp32s3box]
```

The second line shows the platform – `espressif32`:

```
platform = espressif32
```

As of writing this chapter, PlatformIO supports 48 different platforms. `espressif32` is one of them. We can specify the platform version if needed and PlatformIO will find and download it for us. If none is specified, it will assume the latest version of the platform. Then the board that we use in the project is listed:

```
board = esp32s3box
```

The board in the project is `esp32s3box`. There are 11,420 different boards supported by PlatformIO, 162 of which are in the `espressif32` platform. Next, we see the framework:

```
framework = espidf
```

The framework is `espidf`. This category contains 24 more frameworks in the PlatformIO registry.

The `platform`, `board`, and `framework` settings were added automatically in *step 3* with the PlatformIO project wizard. PlatformIO collected them as user inputs at the project definition stage and set the initial content of `platformio.ini` with these values.

Then, we added the next three lines manually to define the serial monitor behavior:

```
monitor_speed=115200
monitor_rts = 0
monitor_dtr = 0
```

We set the serial baud rate to 115,200bps, and RTS and DTR to 0 in order to reset the chip when the serial monitor connects so that we can see the entire serial output of the application.

You can browse the PlatformIO registry at this link to see all platforms, boards, frameworks, libraries, and tools: https://registry.platformio.org/search.

Before moving on, let's include our other board, ESP32-C3-DevKitM-1, in the project and see how easy it is to update the configuration of

the project for different boards. To do that, just append the following lines at the end of `platformio.ini` and save the file:

```
[env:esp32c3kit]
platform = espressif32
board = esp32-c3-devkitm-1
framework = espidf
monitor_speed=115200
monitor_rts = 0
monitor_dtr = 0
```

When you save the file, PlatformIO will detect this and create another entry in the project tasks for the new environment as can be seen in the following figure:

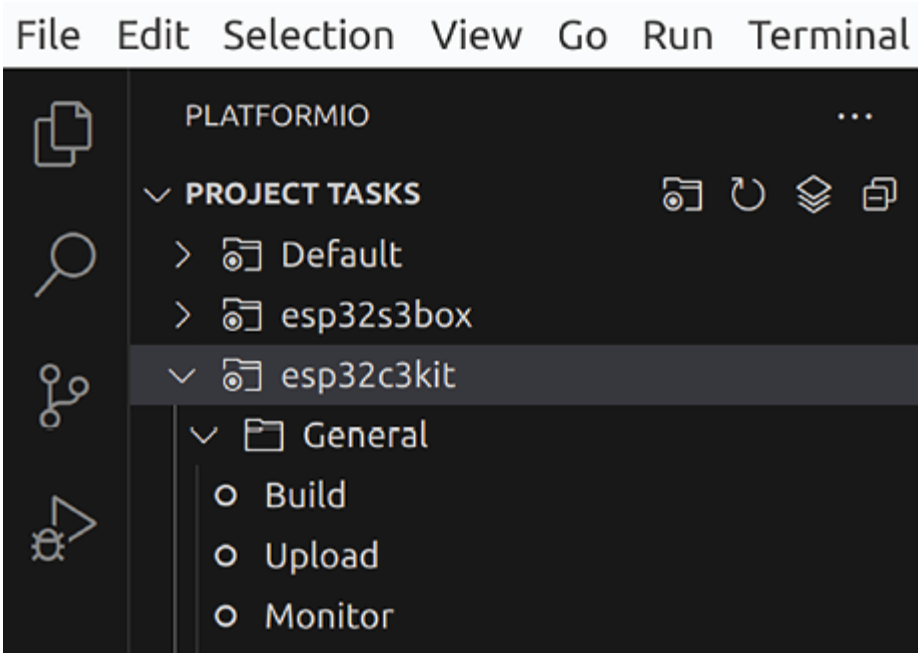


Figure 2.17: New environment under PROJECT TASKS

After plugging the new devkit, you can upload and monitor the same application without making any other modifications in the project. Again, we didn't manually download or install anything for ESP32-C3-DevKitM-1; it was all handled by PlatformIO. If you're wondering where those downloads go, you can find them in the `$HOME/.platformio/platforms/` directory of your development machine. The PlatformIO documentation provides complete information about what can be configured in `platformio.ini` with examples: https://docs.platformio.org/en/latest/projectconf/index.html.

PlatformIO Terminal

In addition to the GUI features, PlatformIO also provides a command-line tool – `pio` – which is accessible through **PlatformIO Terminal**. It can be quite useful in some cases, especially if you enjoy command-line tools in general. To start PlatformIO Terminal, you can click on the **PlatformIO: New Terminal** button in the bottom toolbar of VSCode.

Figure 2.18: VSCode bottom toolbar

This toolbar also has other quick-access buttons for the frequently used features, such as compilation, upload, monitor, etc. When you click on the **Terminal** button (the labels appear when you hover the

mouse pointer over the buttons), it will redirect you to a command-line terminal where you can enter `pio` commands. Write `pio` and press the *Enter* key to display the `pio` options.

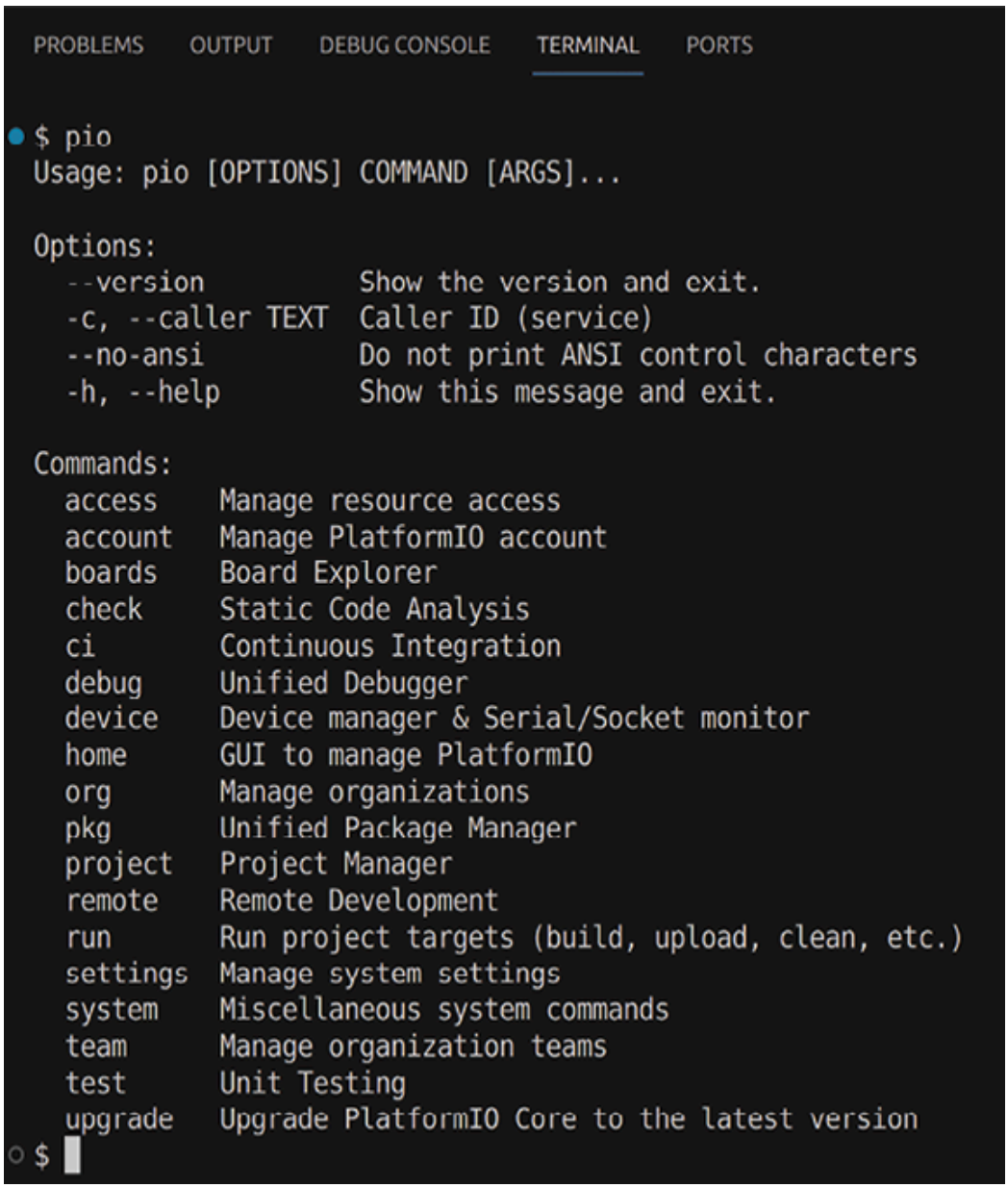


Figure 2.19: PlatformIO Terminal and the pio command-line tool

We can flash ESP32-C3-DevKitM-1 by using `pio` as the following:

```
$ pio run -t upload -e esp32c3kit
Processing esp32c3kit (platform: espressif32; board
-----------------------------------------------
Verbose mode can be enabled via `-v, --verbose` opti
CONFIGURATION: https://docs.platformio.org/page/boar
PLATFORM: Espressif 32 (5.1.1) > Espressif ESP32-C3-
HARDWARE: ESP32C3 160MHz, 320KB RAM, 4MB Flash
…
Leaving...
Hard resetting via RTS pin...
============ [SUCCESS] Took 24.90 seconds ==========
Environment    Status    Duration
-------------  --------  ------------
esp32c3kit     SUCCESS   00:00:24.902
========================================
```

And we can monitor the serial output with the following command:

```
$ pio device monitor -e esp32c3kit
--- forcing DTR inactive
--- forcing RTS inactive
--- Terminal on /dev/ttyUSB0 | 115200 8-N-1
<removed>
ESC[0;32mI (324) cpu_start: Starting scheduler.ESC[0m
Hello World!
```

The `pio` tool has all the functions that you can do with the GUI. To see how to use any other command, just append the `-h` option after

the command's name. The online documentation provides more detailed information about the commands: https://docs.platformio.org/en/latest/core/userguide/index.html#commands.

This completes the introduction to PlatformIO. In the next topic, we will discuss FreeRTOS, the official **Real-Time Operating System** (**RTOS**) supported by ESP-IDF.

FreeRTOS

There are different flavors of FreeRTOS. FreeRTOS was originally designed for single-core architectures. However, ESP32 has two cores, and therefore the Espressif port of FreeRTOS is designed to handle dual-core systems. Most of the differences between vanilla FreeRTOS and ESP-IDF FreeRTOS stem from this. The following list shows some of those differences:

- **Creating a new task**: There is a new function in ESP-IDF FreeRTOS where we can specify on which core to run a new task; it is `xTaskCreatePinnedToCore`. This function takes a parameter to set the task affinity to the specified core. If a task is created by the original `xTaskCreate`, it doesn't belong to any core, and any core can choose to run it at the next tick interrupt.
- **Scheduler suspension**: The `vTaskSuspendAll` function call only suspends the scheduler on the core on which it is called. The other core continues its operation. Therefore, it is not the right way to suspend the scheduler and protect shared resources.

- **Critical sections**: Entering a critical section stops the scheduler and interrupts only on the calling core. The other core continues its operation. However, the critical section is still protected by a mutex, preventing the other core from running the critical section until the first core exits. We can use the `taskENTER_CRITICAL(mux)` and `taskEXIT_CRITICAL(mux)` macros for this purpose.

Another flavor of FreeRTOS is Amazon FreeRTOS, which adds more features. On top of the basic kernel functionality, with Amazon FreeRTOS developers also get common IoT libraries, such as **coreHTTP**, **coreJSON**, **coreMQTT**, and Secure Sockets, for connectivity. Amazon FreeRTOS aims to allow any embedded devices to be connected to the AWS IoT platform easily and securely. We will talk about Amazon FreeRTOS in more detail later in the book. For now, let's stick to ESP-IDF FreeRTOS and see a classic example of the producer-consumer pattern.

Creating the producer-consumer project

In this example, we will simply implement the producer-consumer pattern to show some functionality of Espressif FreeRTOS. There will be a single producer and two consumer FreeRTOS tasks, one on each core of ESP32. As you might guess, the devkit is ESP32-S3-BOX-Lite (ESP32-C3 has a single RISC-V core). The producer task will generate numbers and push them to the tail of a queue. The consumers will pop numbers from the head. The following figure depicts what we will develop in this example:

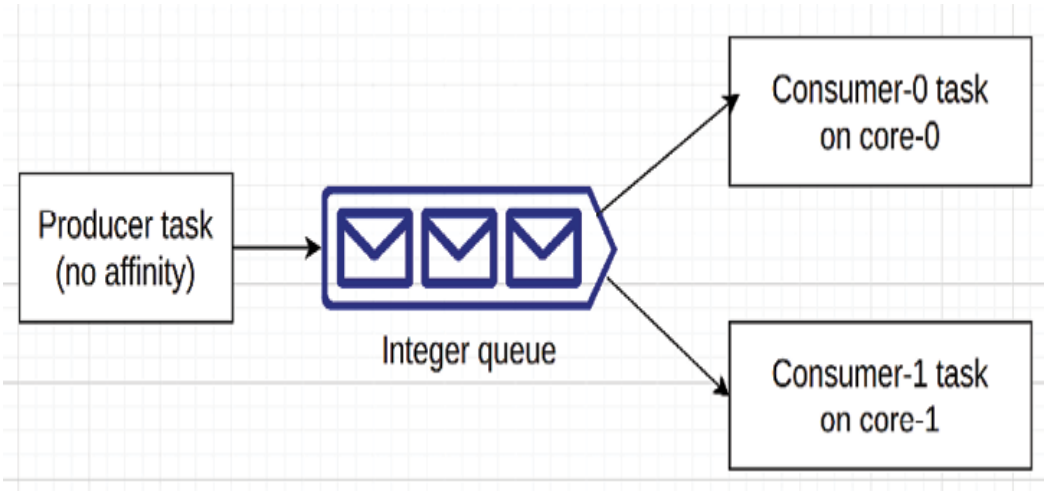


Figure 2.20: Producer-consumer pattern

The producer task will have no affinity, meaning that the FreeRTOS scheduler will assign it to a core at runtime. We will pin a consumer task to each core. There will be a FreeRTOS queue to pass integer values between the producer and the consumers. FreeRTOS queues are thread-safe, so we don't need to think about protecting the queue against reading/writing by multiple tasks. We will simply push values to the back of the queue and pop from the front (there is a good article here about how FreeRTOS queues work: https://www.freertos.org/Embedded-RTOS-Queues.html).

Let's prepare the project in steps:

1. Plug the devkit in a USB of your development machine and start a new PlatformIO project with the following parameters:
    - Name: `espidf_freertos_ex`
    - Board: **Espressif ESP32-S3-Box**
    - Framework: **Espressif IoT Development Framework**
2. Edit `platformio.ini` and append the following lines (the last two lines will provide a nice, colorful output on the serial

monitor):

```
monitor_speed=115200
monitor_rts = 0
monitor_dtr = 0
monitor_filters=colorize
monitor_raw=yes
```

3. Rename `src/main.c` to `src/main.cpp` and edit it by adding the following temporary code:

```
#include <iostream>
extern "C" void app_main()
{
    std::cout << "hi\n";
}
```

4. Run `menuconfig` by selecting **PLATFORMIO** | **PROJECT TASKS** | **esp32s3box** | **Platform** | **Run Menuconfig**.

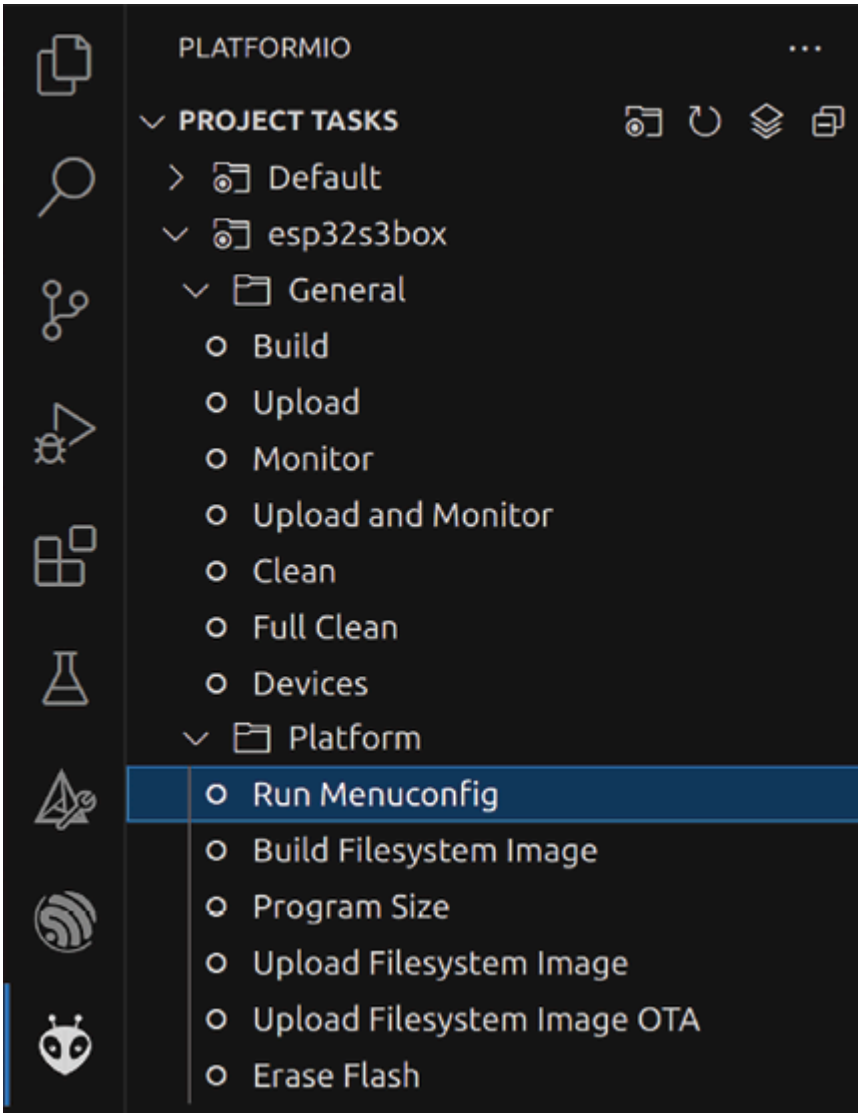


Figure 2.21: Running menuconfig

5. This is the first time we run `menuconfig` to configure ESP-IDF. We need to change a configuration value in order to enable a FreeRTOS function that lists the FreeRTOS tasks in an application. When `menuconfig` starts, navigate to **(Top)** → **Component config** → **FreeRTOS** → **Kernel** and check the following options (the latter two are dependent on the first one, and will become visible when the first is enabled):
    - Enable **FreeRTOS trace utility**

- Enable **FreeRTOS stats formatting functions**
- Enable display of `xCoreID` in `vTaskList`

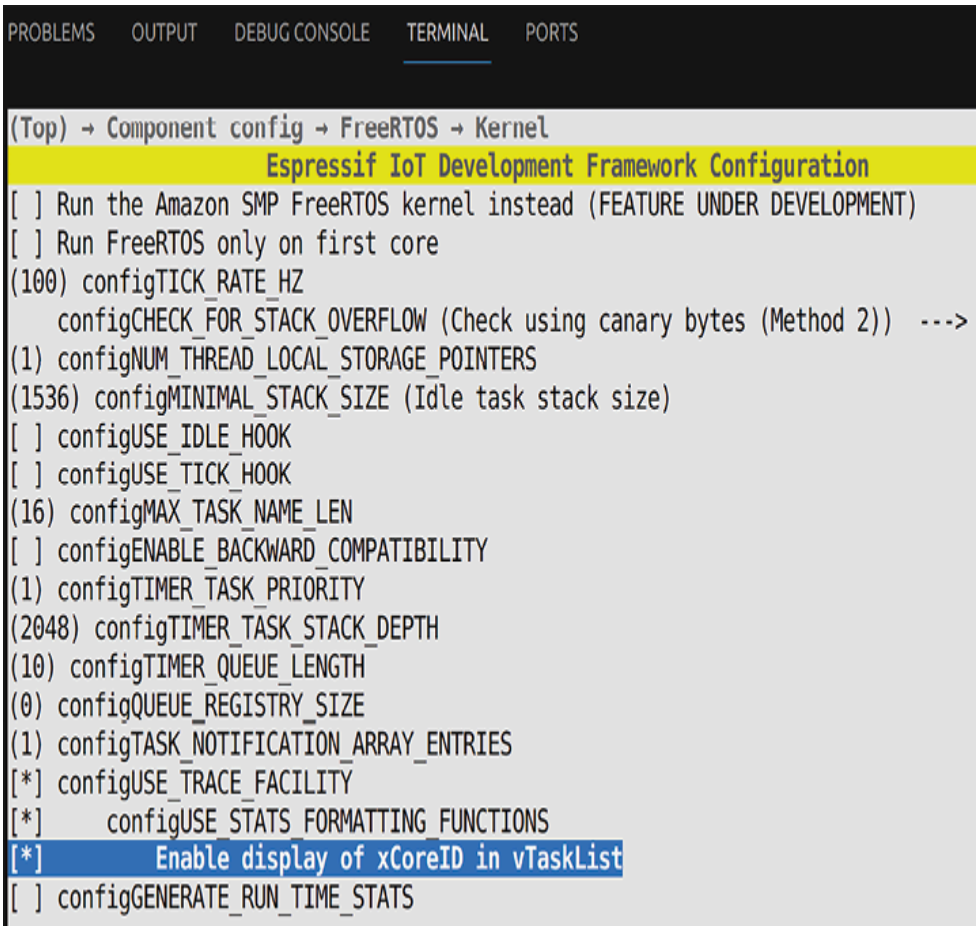


Figure 2.22: Configuring FreeRTOS in menuconfig

6. Build the project (**PLATFORMIO | PROJECT TASKS | esp32s3box | General | Build**).
7. Flash and monitor the application to see the `hi` text on the serial monitor (**PLATFORMIO/PROJECT TASKS | esp32s3box | General | Upload and Monitor**).

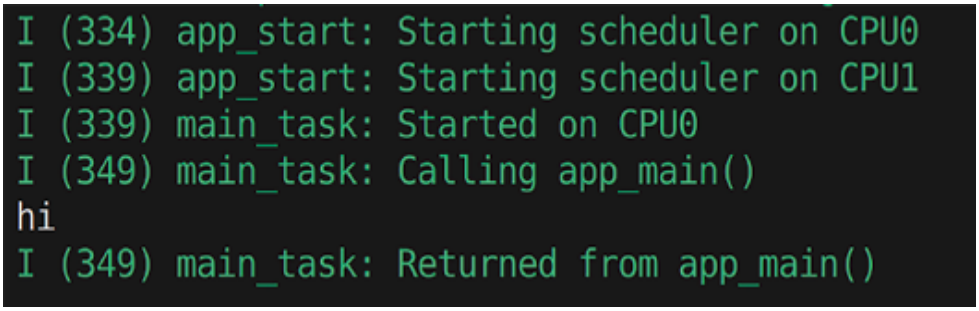


Figure 2.23: The serial monitor output when the application is configured successfully

Now that we have the project configured, we can develop the application, next.

Coding application

So far, so good. Now, we can implement the producer-consumer pattern in the `src/main.cpp` file. First, we clear the temporary code inside the file and then add the following headers:

```
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/queue.h>
#include <esp_log.h>
```

The `freertos/FreeRTOS.h` header file contains the backbone definitions based on the configuration. When we need a FreeRTOS function, we first include this header file, then the specific header where the needed function is declared. In our example, we will create tasks and a queue for the producer-consumer pattern; thus, we include `freertos/task.h` and `freertos/queue.h` respectively. The last header file, `esp_log.h`, is for printing log

messages on the serial console. Instead of direct access to the serial output via `iostream`, we will use the ESP-IDF logging macros in this application. Then we can define the global variables in the file scope:

```
namespace
{
    QueueHandle_t m_number_queue{xQueueCreate(5, si
    const constexpr int MAX_COUNT{10};
    const constexpr char *TAG{"app"};
    void producer(void *p);
    void consumer(void *p);
} // end of namespace
```

In the anonymous namespace, we define a FreeRTOS queue, `m_number_queue`. This will be the medium in which to exchange data between the producer and consumers. The `xQueueCreate` function (in fact, it is a macro) creates a queue to hold 5 integers. The producer will generate integers to push into the queue. The `MAX_COUNT` constant shows the maximum number of integers to be generated by the producer. `TAG` is required by the logging macros. We will use it as a parameter when we want to log something. A logging macro prints the provided tag before any message. `producer` and `consumer` are the functions to be passed to the FreeRTOS tasks. We will see how to do this next:

```
extern "C" void app_main()
{
```

```
    ESP_LOGI(TAG, "application started");
    xTaskCreate(producer, "producer", 4096, nullptr
```

Now, we're implementing the `app_main` function. Remember that this is the application entry point. The first statement is the `ESP_LOGI` macro call with `TAG` and a message. **application started** will be printed on the serial monitor when the application starts. There are other macros in the logging family, such as `ESP_LOGE` for errors and `ESP_LOGW` for warnings. In the next line after printing the log message, we create our first FreeRTOS task by calling `xTaskCreate`. It has the following syntax in the `freertos/task.h` header file:

```
xTaskCreate(task_function, task_name, stack_depth,
    function_parameters, priority, task_handle_addre
```

Looking at this prototype, `xTaskCreate` will create a FreeRTOS task that runs the `producer` function that we declared earlier. The task name will be `producer` with a stack size of `4096` bytes. We don't pass any parameters to the task. The task priority is `5`, and finally, we don't provide any address for the task handle since we don't need it in this example. The FreeRTOS scheduler will create the producer task with these parameters.

Then, we need the consumers:

```
    xTaskCreatePinnedToCore(consumer, "consumer-0",
                            5, nullptr, 0);
    xTaskCreatePinnedToCore(consumer, "consumer-1",
                            5, nullptr, 1);
```

We will have two consumers. For this, we use the `xTaskCreatePinnedToCore` function this time. It is very similar to `xTaskCreate`. Its prototype is:

```
xTaskCreatePinnedToCore(task_function, task_name, st
    function_parameters, priority, task_handle_addre
```

In addition to the parameters that `xCreateTask` uses, `xTaskCreatePinnedToCore` needs a task affinity defined – i.e., on which core to run the task. In our example, the first consumer task will run on `cpu-0`, and the second one will run on `cpu-1`. This function is specific to ESP-IDF FreeRTOS in order to support dual-core processors as we mentioned earlier.

We have now created all the tasks. Let's see the list of the FreeRTOS tasks that we have in this application with the following lines of code:

```
    char buffer[256]{0};
    vTaskList(buffer);
```

```
    ESP_LOGI(TAG, "\n%s", buffer);
} // end of app_main
```

To list the tasks, we call `vTaskList` with a `buffer` parameter. It fills the buffer with the task information and we print the buffer on the serial output. `vTaskList` has been enabled by a `menuconfig` entry during the project initialization phase. This completes the `app_main` function. Next, we will implement the producer task function in the anonymous namespace:

```
namespace
{
    void producer(void *p)
    {
        int cnt{0};
        vTaskDelay(pdMS_TO_TICKS(500));
```

In the `producer` function, we define a variable, `cnt`, to count the numbers that we push into the queue. Then, we implement a 500 ms delay in the task execution. We add a loop for enqueueing the numbers as follows:

```
        while (++cnt <= MAX_COUNT)
        {
            xQueueSendToBack(m_number_queue, &cnt, p
            ESP_LOGI(TAG, "p:%d", cnt);
        }
```

In the loop, we use the `xQueueSendToBack` function of FreeRTOS to send the numbers into the queue. The `xQueueSendToBack` function takes the queue reference, a pointer to the value to be pushed into the queue, and the maximum time for which to block the task if the queue is full. The number that is passed to the queue is the value of the `cnt` variable itself. Therefore, we will see the numbers starting from 1 up to 10 in the queue. We finish the producer task function as follows:

```
        vTaskDelete(nullptr);
    } // end of producer
```

A FreeRTOS task cannot return, else the result would be an application crash. When we are done with a task and we don't need it anymore, we simply delete it by calling the `vTaskDelete` function. This function takes the task handle as a parameter, and passing `nullptr` means that the current task is the one to be deleted. Since there is no task after that point, we can safely return from the `producer` function. Then we implement the `consumer` function:

```
    void consumer(void *p)
    {
        int num;
```

The `consumer` function will run on both cores of ESP32-S3. When we defined two consumer tasks in the `app_main` function, we passed the `consumer` function as the task function and the core

number as the parameter to be passed to the `consumer` function. Therefore, the `p` argument of the function shows the core number. In the `consumer` function body, we first define a variable, `num`, to hold the values that come from the queue. Next comes the task loop:

```
        while (true)
        {
            xQueueReceive(m_number_queue, &num, port
            ESP_LOGI(TAG, "c%d:%d", (int)p, num);
            vTaskDelay(2);
        }
    } // end of consumer
} // end of namespace
```

The task loop is an infinite loop, so the function will never return as it should be. The `xQueueReceive` function takes the same parameters as with the `xQueueSendToBack` function that we used in the `producer` function. However, the `xQueueReceive` function pops the value at the front of the queue. When all values in the queue are consumed, it will block the task until a new value arrives. If no value comes, then the `xQueueReceive` function will block forever since we passed `portMAX_DELAY` as its third argument. The application is ready to run on the devkit, let's do it next.

Running the application

We can upload and monitor it by clicking on the **Upload and Monitor** project task of the PlatformIO IDE. Let's discuss the output

briefly:

```
<Previous logs are removed ...>
I (280) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
I (301) app: application started
```

After the start of the FreeRTOS schedulers on both CPUs, our application prints its first log as `application started`. Then we see the `vTaskList` output as follows:

```
I (301) app:
consumer-1      R   5       3580        9       1
main              X   1       1936        4       0
IDLE               R   0       892         6       1
IDLE               R   0       1012         5
producer           B   5       3500        7       2
esp_timer          S   22      3432        3       0
ipc1               B   24      884          2
consumer-0      B 5       3412        8       0
ipc0               B   24      892          1
```

The columns in this table are:

- Task name
- Task state
- Priority
- Used stack in bytes
- The order in which the tasks are created

- Core ID

We can see our tasks in the list in addition to other default tasks. They are (in the order of creation) as follows:

- The **Inter-Processor Call** (**IPC**) tasks (`ipc0` and `ipc1`) for triggering execution on the other CPU
- `esp_timer` for RTOS tick period
- The `main` task that calls the `app_main` function (entry point) of the application
- The `IDLE` tasks of FreeRTOS

After the default FreeRTOS tasks, our tasks start. When you look at the last column of the table, **consumer-0** has started on `cpu0`, **consumer-1** has started on `cpu1`, and for **producer**, the core ID value is displayed as **-1**, which means it can run on both CPUs.

The logs from the tasks come next on the serial output:

```
I (801) app: p:1
I (801) app: p:2
I (801) app: c1:1
I (801) app: p:3
I (801) app: c0:2
I (801) app: p:4
I (801) app: p:5
I (801) app: p:6
I (801) app: p:7
I (821) app: c1:3
I (821) app: p:8
I (831) app: c0:4
I (831) app: p:9
```

```
I (841) app: c1:5
I (841) app: p:10
I (851) app: c0:6
I (861) app: c1:7
I (871) app: c0:8
I (881) app: c1:9
I (891) app: c0:10
```

Because of the delays in the consumer tasks, the producer fills up the queue faster than the consumers remove numbers and the producer has to wait for the consumers to make some space so it can insert a new number. When **consumer-1** removes `3` from the queue, then the producer can enqueue `8`. It stops pushing new numbers when it gets to `10` as we coded. The rest of the job is only for the consumers to dequeue all numbers remaining in the queue.

This example demonstrated how to utilize FreeRTOS for a simple producer-consumer problem and the basic usage of the ESP32 cores with different tasks. We will continue to employ FreeRTOS in the examples of the upcoming chapters and learn about more of its features. The official ESP-IDF FreeRTOS API documentation is here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html.

In the next topic, we will discuss how we can debug our applications.

Debugging

All families of ESP32 MCUs support **Joint Test Action Group** (**JTAG**) debugging. ESP-IDF makes use of **OpenOCD**, an open-source software, to interface with the **JTAG probe/adapter**, such as an **ESP-Prog** debugger. To debug our applications, we use an Espressif version of **gdb** (the GNU debugger), depending on the architecture of ESP32 that we have in a particular project. The next figure shows a general ESP32 debug setup:

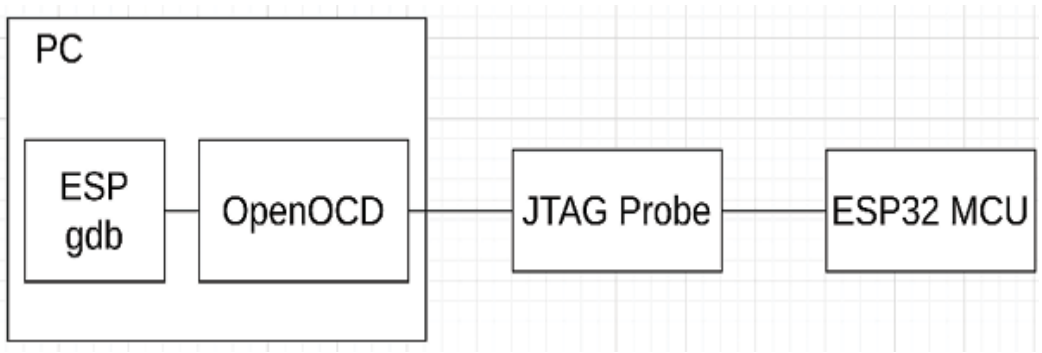


Figure 2.24: JTAG debugging

When we develop our own custom ESP32 devices, we can connect to the standard JTAG interface of ESP32 to debug the application. With this option, we need to use a JTAG probe between the development machine and the custom ESP32 device. The JTAG pins are listed on the official documentation for each family of ESP32 (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/jtag-debugging/configure-other-jtag.html).

The issue with JTAG debugging is that it requires at least 4 GPIO pins to carry the JTAG signals, which means 4 GPIO pins less available to your application. This might be a problem in some projects where you need more GPIO pins. To address this issue,

Espressif introduces direct USB debugging (built-in JTAG) without a JTAG probe. In the preceding figure, the JTAG probe in the middle is not needed for debugging and OpenOCD running on the development machine talks directly to the MCU over USB. The built-in JTAG debugging requires only two pins on ESP32, which saves two pins compared to the ordinary JTAG debugging with a probe. This feature is not available in all ESP32 families but ESP32-C3 and ESP32-S3 do have it; thus, we will prefer this method in this example with our ESP32-S3-BOX-Lite devkit. We don't need a JTAG probe but we still need a USB cable with the pins exposed outside to be able to connect them to the corresponding pins of the devkit. The connections are:

```
ESP32-S3 Pin    USB Signal
GPIO19            D-
GPIO20            D+
5V                V_BUS
GND             Ground
```

We can find a USB cable on many online shops with all lines exposed but it is perfectly fine to cut a USB cable and solder a 4-pin header to use its pins. You can see my simple setup below:

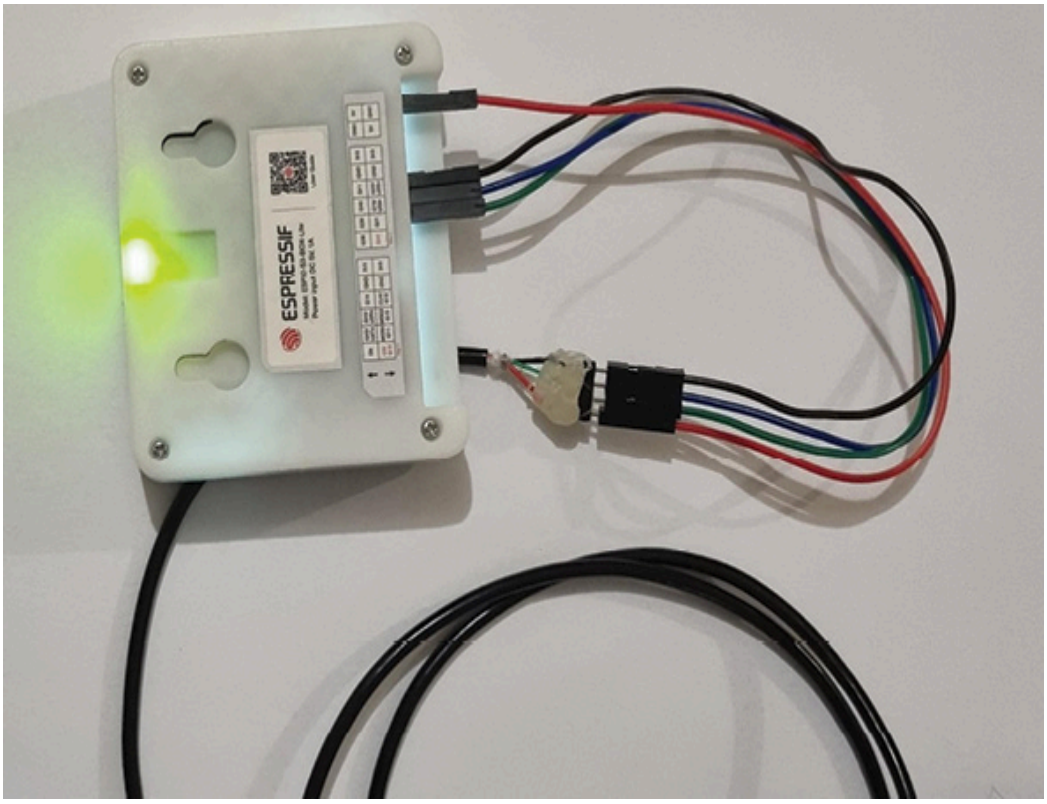


Figure 2.25: Built-in JTAG

We don't need a driver for Linux or macOS to use the built-in JTAG debugging. The Windows driver comes with the **ESP-IDF Tools Installer** (https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/get-started/windows-setup.html#esp-idf-tools-installer).

Now, it is time to create the project and upload the firmware to see whether our setup works. In this example, we will see another way of creating an ESP-IDF project. We will work in the ESP-IDF environment from the command line and use the `idf.py` script to create the project and for other project tasks. Let's do this in steps:

1. If you are a Windows user, run the **ESP-IDF Command Prompt** shortcut from the Windows Start menu. It will open a

command-line terminal with the ESP-IDF environment. If your development platform is Linux or macOS, start a terminal and run the `export.sh` or `export.fish` scripts respectively to have the ESP-IDF environment in the terminal:

```
$ source ~/esp/esp-idf/export.sh
Detecting the Python interpreter
Checking "python" ...
Python 3.10.11
"python" has been detected
Adding ESP-IDF tools to PATH…
<more logs>
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:
idf.py build
```

2. Test the `idf.py` script by running it without any arguments. It will print the help message:

```
$ idf.py
Usage: idf.py [OPTIONS] COMMAND1 [ARGS]... [COMM
ESP-IDF CLI build management tool. For commands
<rest of the help message>
```

3. Go to your project directory and run `idf.py` with a new project name. The script will create an ESP-IDF project with that name in a directory with the same name:

```
$ idf.py create-project debugging_ex
Executing action: create-project
```

```
The project was created in <your project directo
$ ls
debugging_ex
$ cd debugging_ex/
$ tree
.
├── CMakeLists.txt
└── main
    ├──CMakeLists.txt
    └── debugging_ex.c
1 directory, 3 files
```

4. Download the `sdkconfig` file from the book repository into the project directory. It can be found here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/blob/main/ch2/debugging_ex/sdkconfig.
5. Run VSCode and open the `debugging_ex` directory.
6. In the VSCode IDE, rename `main/main.c` to `main/main.cpp` and edit it to have the following code inside:

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
void my_func(void)
{
    int j = 0;
    ++j;
}
extern "C" void app_main()
```

```
{
    int i = 0;
    while (1)
    {
        vTaskDelay(pdMS_TO_TICKS(1000));
        ++i;
        my_func();
    }
}
```

7. Update the `main/CMakeLists.txt` file with the following content:

```
idf_component_register(SRCS "debugging_ex.cpp" I
```

8. After these changes, we should have the following directory structure:

```
$ tree
.
├── CMakeLists.txt
├── main
│   ├── CMakeLists.txt
│   └── debugging_ex.cpp
└── sdkconfig
1 directory, 4 files
```

9. We need to enable the debug options in the root `CMakeLists.txt` file. Edit it and set its content as the following:

```
cmake_minimum_required(VERSION 3.16.0)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(debugging_ex)
idf_build_set_property(COMPILE_OPTIONS "-O0" "-g
```

10. In the terminal, build the application by running `idf.py`:

```
$ idf.py build
Executing action: all (aliases: build)
<more logs>
Creating esp32s3 image...
Merged 2 ELF sections
Successfully created esp32s3 image.
<more logs>
```

11. See that the `idf.py` script has generated the application binary under the build directory:

```
$ ls build/*.bin
build/debugging_ex.bin
```

12. Run the following command to see the basic size information of the application:

```
$ idf.py size
<some logs>
Total sizes:
Used static IRAM:   87430 bytes ( 274810 remain,
      .text size:   86403 bytes
   .vectors size:    1027 bytes
Used stat D/IRAM:   13941 bytes ( 158987 remain,
      .data size:   11389 bytes
      .bss  size:    2552 bytes
Used Flash size :  153979 bytes
      .text     :  113547 bytes
      .rodata   :   40176 bytes
Total image size:  252798 bytes (.bin may be pad
```

13. Flash the application on the devkit (if flashing fails with a port error, just reverse the D+/D- connections of the devkit. This simple change will probably solve the problem):

```
$ idf.py flash
Executing action: flash
Serial port /dev/ttyACM0
Connecting...
Detecting chip type... ESP32-S3
<more logs>
Leaving...
Hard resetting via RTS pin...
Done
```

14. We now have an application running on the devkit, ready for debugging. Run a GDB server with the following command

(`idf.py` will start an `OpenOCD` process for this):

```
$ idf.py openocd --openocd-commands "-f board/es
Executing action: openocd
OpenOCD started as a background task 477341
Executing action: post_debug
Open On-Chip Debugger v0.11.0-esp32-20221026 (20
Licensed under GNU GPL v2
<more logs>
```

15. Start another ESP-IDF command-line terminal as we did in *step 1* and change the current directory to the project root directory.
16. Run the following command to start a GDB client. It will open a web-based GUI in your default browser:

```
$ idf.py gdbgui
Executing action: gdbgui
gdbgui started as a background task 476131
```

The `idf.py` script is the single point of contact to manage an ESP-IDF project. We can create, build, flash, and debug an application by only using this script. It has more features and we will have many chances to learn and practice those features throughout the book. The ESP-IDF build system documentation provides detailed information about the `idf.py` script and how it works with *cmake* to collect all the project components to compile them into an application. Here is the link for the documentation: https://docs.espressif.com/projects/esp-idf/en/v4.4.4/esp32s3/api-guides/build-system.html.

With a debugging session ready on the web GUI, we can now debug the application. The following screenshot shows this GUI.

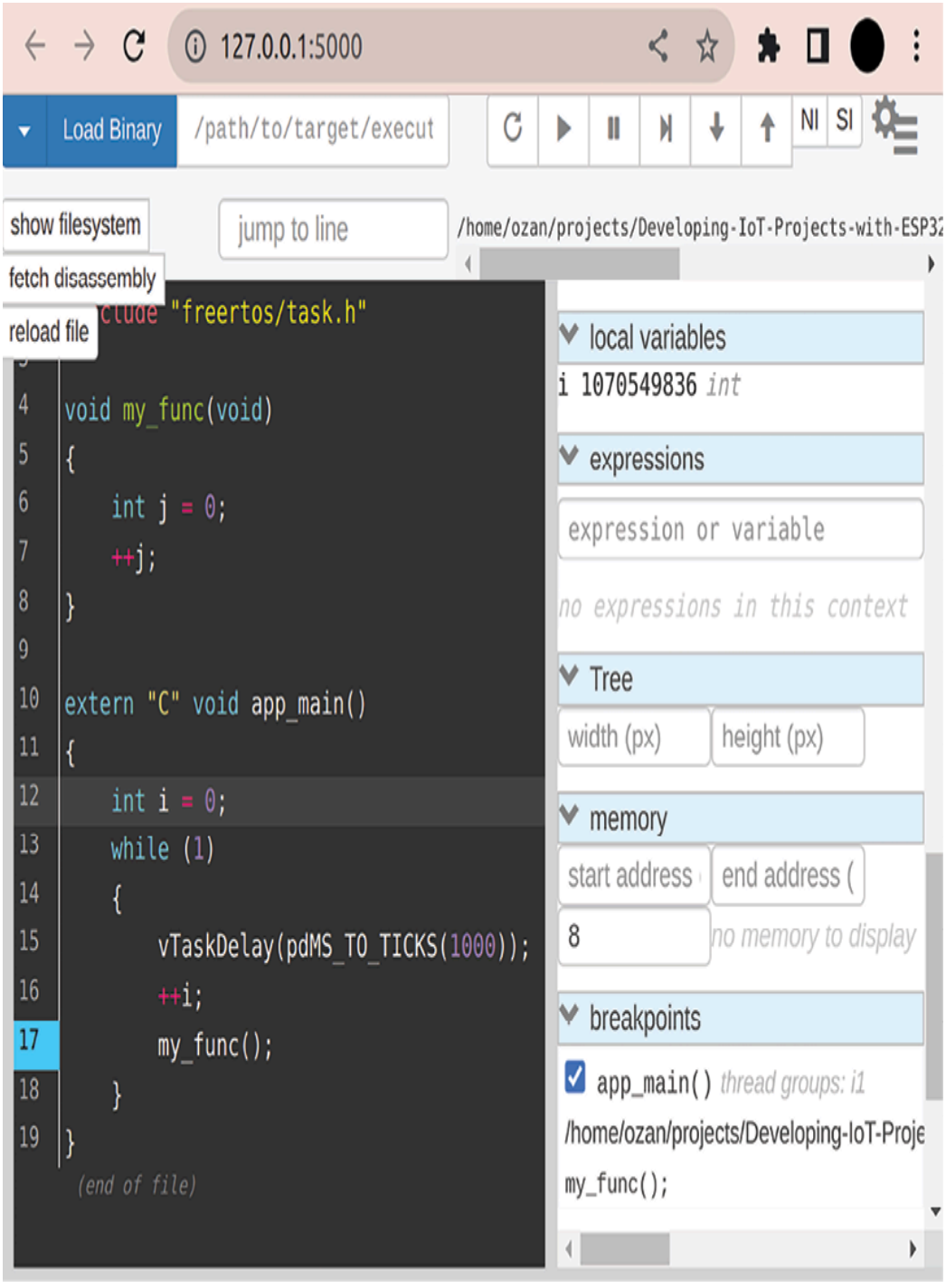


Figure 2.26: Web-based debugger

The left panel shows the source code that is being debugged. We can set/remove breakpoints by clicking on the row numbers. The right panel shows the current status of the application, including threads, variables, memory, etc. On the top right, the buttons for the debug functions (restart, pause, continue, and step in/out/over) are placed. The debug functions also have keyboard shortcuts to ease the debugging process. Try the following debugging tasks on the GUI:

- Click on line number **17** to set a breakpoint.
- Press *C* (continue) on the keyboard and observe that the local variable `i` increases every time the execution hits the breakpoint.
- Try pressing *N* (next) to run each of the lines consecutively.
- When the execution comes to `my_func`, press *S* (step in) to enter the function. You can exit the function by pressing *U* (up).

This GUI is enough for an average debugging session and can be used to observe the behavior of the application when necessary. If you need to access other `gdb` commands, there is also another panel at the bottom where you can type these commands.

Unit testing

Whether you prefer **Test-Driven Development** (**TDD**) or just write unit tests as a safety net against regression, it is always wise to include them in the plans of any type of software project. Although the adopted testing strategy for a project depends on the project type and company policies, well-designed unit tests are the basic safeguard of any serious product. The time and effort you put into

unit tests always pay off in every stage of the product life cycle, from the beginning of the development to the maintenance and upgrades.

For ESP32 projects, we have several unit-testing framework options. ESP-IDF supports the **Unity** framework but we can also use **GoogleTest** in our projects. We can configure PlatformIO to use any of them and run tests on a target device, such as an ESP32 devkit, and/or on the local development machine. Therefore, it is really easy to select different strategies for unit testing. For example, if the library that you are working on doesn't need to use hardware peripherals, then it can be tested on the local machine and you can instruct PlatformIO to do this by simply adding some definitions to the `platformio.ini` file of the project.

We will create a sample project to see how unit testing is done in an ESP32 project next.

Creating a project

Let's assume that we want to develop a simple light control class that sets a GPIO pin of ESP32-C3-DevKitM-1 to high/low in order to turn on and off the light that is connected to the GPIO pin. In this example, we will develop that class and write tests for it with the GoogleTest framework. We will also configure PlatformIO to run the tests on the devkit so that we know that the class works as intended on the real hardware. Let's create the project as in the following steps:

1. Start a new PlatformIO project with the following parameters:
    - Name: `unit_testing_ex`

    - Board: **Espressif ESP32-C3-DevKitM-1**
    - Framework: **Espressif IoT Development Framework**
2. Open the `platformio.ini` file and set its content as follows:

```
[env:esp32-c3-devkitm-1]
platform = espressif32@6.2.0
board = esp32-c3-devkitm-1
framework = espidf
build_flags = -std=gnu++11 -Wno-unused-result
monitor_speed = 115200
monitor_rts = 0
monitor_dtr = 0
monitor_filters = colorize
lib_deps = google/googletest@1.12.1
test_framework = googletest
```

3. Build the project (**PLATFORMIO | PROJECT TASKS | esp32-c3-devkitm-1 | General | Build)**.

The project is now configured and ready for development. However, before moving on, I want to briefly discuss the library management mechanism of PlatformIO. We have several options to add external libraries to our projects. The easiest one is probably just by referring to the PlatformIO registry. You can search the registry by navigating to **PlatformIO Home/Libraries** and typing the name of the library that you are looking for. When the library is listed, you select it and PlatformIO shows its detailed information. At this point, you can click on the **Add to Project** button to include the library in the project.

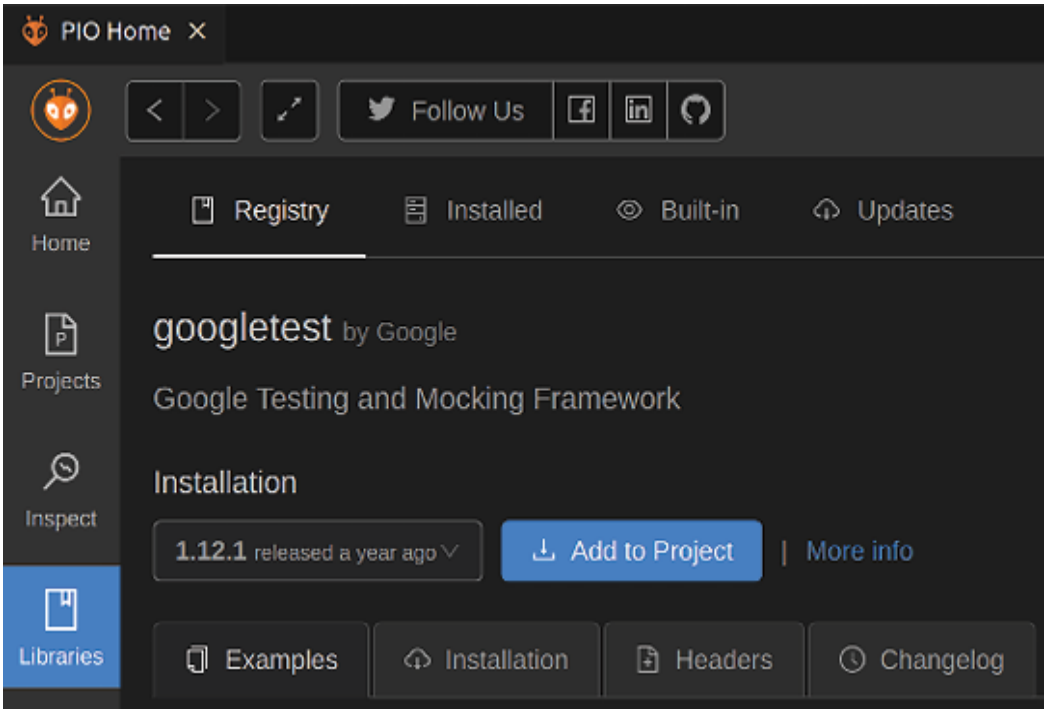


Figure 2.27: PlatformIO Registry

In our example, I preferred to specify `googletest` directly as in the following configuration line without using the graphical interface:

```
lib_deps = google/googletest@1.12.1
```

This line points to the PlatformIO registry. The format is `<provider>/<library>@<version>.` In this way, it is possible to add many other libraries consecutively in a project. With the `lib_deps` configuration parameter, we can also refer to other online repositories by providing their URLs.

The other popular option is to add local directories with `lib_extra_dirs` in `platformio.ini`. Any ESP-IDF-compatible library in these directories can be included in projects. I will talk about what *compatible* means in this context later in the book.

You can learn more about the PlatformIO Library Manager at this link: https://docs.platformio.org/en/latest/librarymanager/index.html.

You may have noticed that we can also set the platform version:

```
platform = espressif32@6.2.0
```

With this configuration, we set the platform version to a fixed value so that no matter when we compile the project, we know that it will compile without any compatibility issues with all other versioned libraries and of course with our code.

After this brief overview of library management, we can continue with the application.

Coding the application

Let's begin with adding a header file, named `src/AppLight.hpp`, for the light control class and add the required header for GPIO control:

```
#pragma once
#include "driver/gpio.h"
#define GPIO_SEL_4 (1<<4)
```

Then we define the class as follows:

```
namespace app
{
    class AppLight
    {
    private:
        bool m_initialized;
```

In the `private` section of the class, we define a member variable, `m_initialized`, which shows if the class is initialized. The `public` section comes next:

```
    public:
        AppLight() : m_initialized(false) {}
        void initialize()
        {
            if (!m_initialized)
            {
                gpio_config_t config_pin4{
                  GPIO_SEL_4,
                  GPIO_MODE_INPUT_OUTPUT,
                  GPIO_PULLUP_DISABLE,
                  GPIO_PULLDOWN_DISABLE,
                  GPIO_INTR_DISABLE
                };
                gpio_config(&config_pin4);
                m_initialized = true;
            }
            off();
        }
```

After the constructor, we implement the `initialize` function. Its job is to configure the GPIO-4 pin of the devkit if it is not initialized yet and set its initial state to off. We use the `gpio_config` function to configure a GPIO pin as defined in the configuration structure that is provided as input. Here, it is `config_pin4`. The `gpio_config` function and the `gpio_config_t` structure are declared in the `driver/gpio.h` header file.

The `off` function is another member function of the class to be implemented next:

```
        void off()
         {
             gpio_set_level(GPIO_NUM_4, 0);
         }
         void on()
         {
             gpio_set_level(GPIO_NUM_4, 1);
         }
    };
} // namespace app
```

In the `off` member function, we call `gpio_set_level` with the parameters of `GPIO_NUM_4` as the pin number and `0` as the pin level. Again, the `gpio_set_level` function is declared in the `driver/gpio.h` header file. Similarly, we add another function, `on`, in order to set the pin level to `1`, or high.

The `AppLight` class is ready and we can write the test code for it next.

Adding unit tests

We create another source file, `test/test_main.cpp`, and add the header files that are needed for the unit tests:

```
#include "gtest/gtest.h"
#include "AppLight.hpp"
#include "driver/gpio.h"
```

For the `AppLight` testing, it would be a good idea to create a test fixture:

```
namespace app
{
    class LightTest : public ::testing::Test
    {
    protected:
        static AppLight light;
        LightTest()
        {
            light.initialize();
        }
    };
    AppLight LightTest::light;
```

The name of the test fixture is `LightTest` and it is derived from the `::testing::Test` base class. In its protected area, we declare a static `AppLight` object and initialize it in the constructor of the fixture. With the fixture ready, we can now write a test as follows:

```
    TEST_F(LightTest, turns_on_light)
    {
        light.on();
        ASSERT_GT(gpio_get_level(GPIO_NUM_4), 0);
    }
```

The `TEST_F` macro defines a test on a test fixture. The first parameter shows the fixture name and the second parameter is the test name. In the test, we turn the light on, and assert whether it is really turned on. The `ASSERT_GT` macro checks whether the first parameter is greater than the second one.

Another test checks whether the `off` function is working properly or not. It is very similar to the previous test:

```
    TEST_F(LightTest, turns_off_light)
    {
        light.off();
        ASSERT_EQ(gpio_get_level(GPIO_NUM_4), 0);
    }
} // namespace app
```

This time, we turn the light off, and check whether it is actually turned off by using the `ASSERT_EQ` macro.

For each new test, a new fixture object will be created. That is why we defined the `light` object as static since we don't want it to be initialized every time a new fixture is created. For more information about

GoogleTest, see its documentation here:https://google.github.io/googletest/primer.html.

We still need an `app_main` function as usual. Here it comes:

```
extern "C" void app_main()
{
    ::testing::InitGoogleTest();
    RUN_ALL_TESTS();
}
```

The two lines in the `app_main` function initialize and run all of the test cases. This finalizes the test coding. Let's run it on the devkit and see the test results.

Running unit tests

We can run the test application on the devkit and see the unit test results as in the following steps:

1. Plug the devkit into one of the USB ports of your development machine.
2. Navigate to **PLATFORMIO | PROJECT TASKS | esp32-c3-devkitm-1 | Advanced** and click on the **Test** option there.

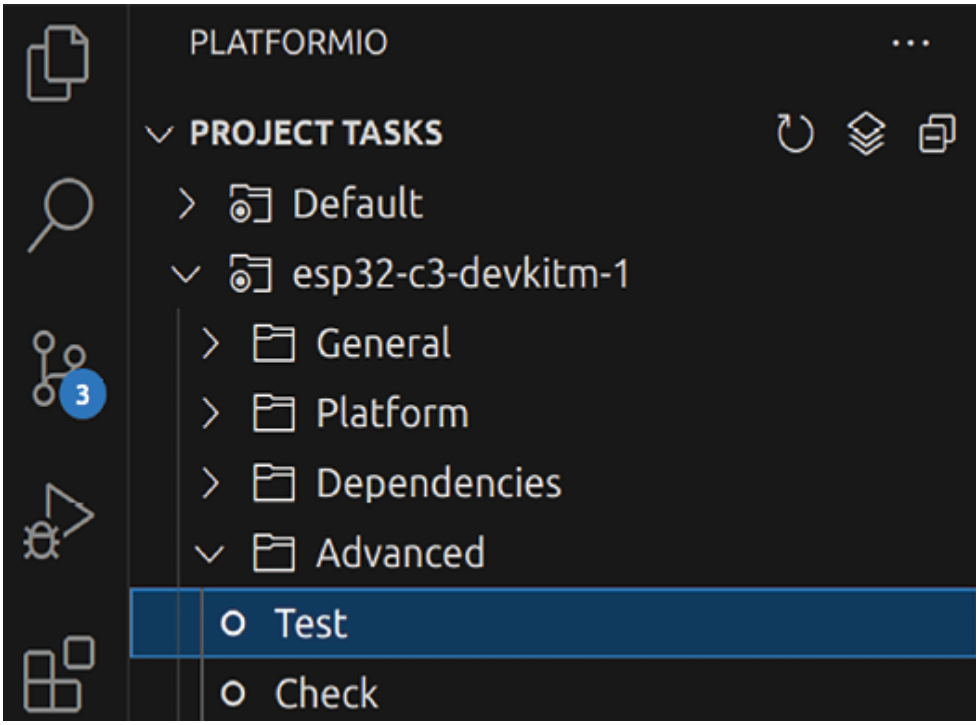


Figure 2.28: PlatformIO unit testing

3. PlatformIO will compile the test application, upload it, and then run the tests. You can see the result on the terminal that popped up when you clicked on the **Test** option.

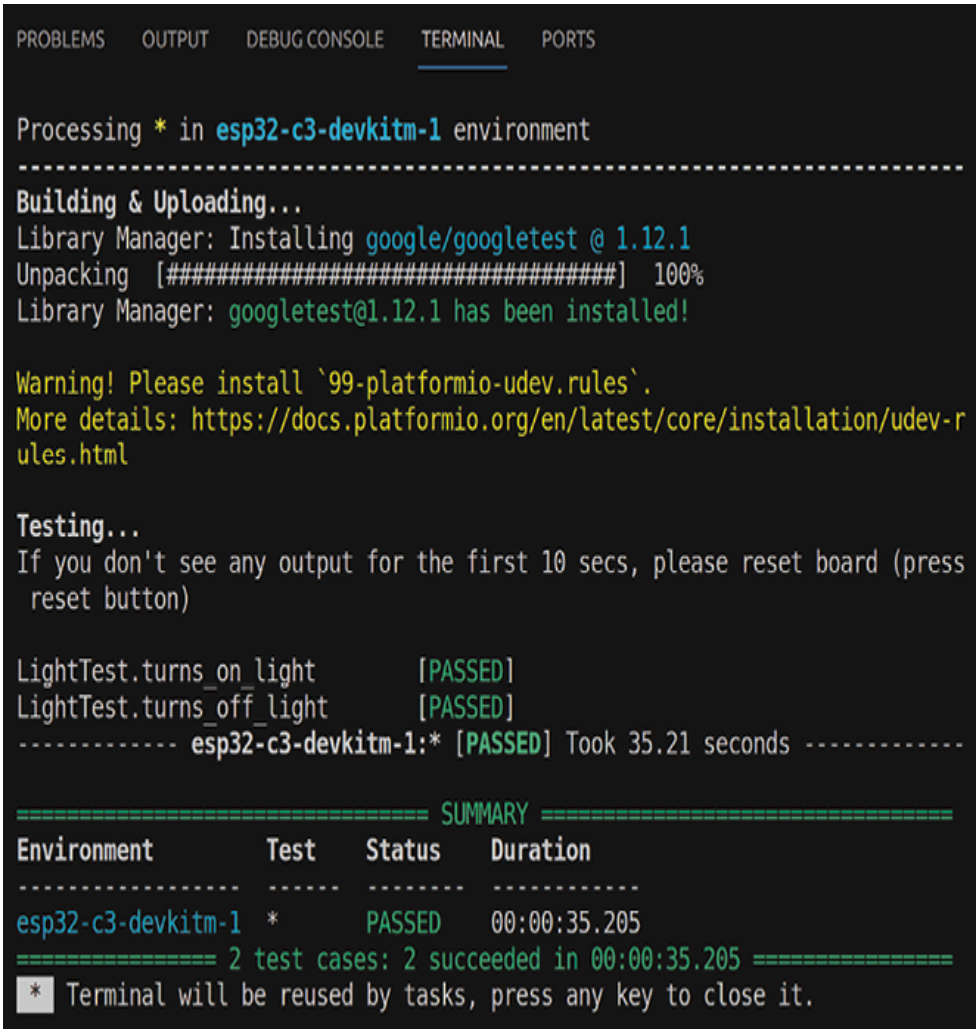


Figure 2.29: Terminal output

The terminal lists the tests and the results. When a test fails, you can go back to the code, debug it, and run the tests again until they all pass.

With this topic, we conclude the chapter. However, I strongly suggest you don't limit yourself to the explanations here and try other tools from both PlatformIO and ESP-IDF. I will continue to talk about them throughout the book and use them within the examples to help you get familiar with the tools and their features as much as possible.

Summary

In this chapter, we have learned about the tools and the basics of ESP32 development. **ESP-IDF** is the official framework to develop applications on any family of ESP32 series microcontrollers, maintained by Espressif Systems. It comes with the entire set of command-line utilities that you would need in your ESP32 projects. **PlatformIO** adds more IDE features on top of that. With its strong integration with the VSCode IDE and declarative project configuration approach, it provides a professional environment for embedded developers.

In the next chapter, we'll discuss the ESP32 peripherals. Although it is impossible to cover all of them in a single chapter, we will learn about the common peripherals using examples so that we can easily carry out the tasks in real projects and the other experiments in the book.

Questions

Let's answer the following questions to test our knowledge of the topics covered in this chapter:

1. What is the name of the script file that the ESP-IDF build system uses to configure an ESP32 project?
    a. `CMakeLists.txt`
    b. `platformio.ini`
    c. `main.cpp`
    d. `Makefile`

2. What is the name of the project-specific configuration file that is usually edited by running `menuconfig`?

    a. `CMakeLists.txt`
    b. `sdkconfig`
    c. `pio`
    d. `platformio.ini`

3. Which of the following is the most fundamental tool that comes with ESP-IDF to manage an ESP32 project?

    a. `pio`
    b. `openocd`
    c. `gdb`
    d. `idf.py`

4. Which of the following methods is the easiest to debug an ESP32-S3 board?

    a. JTAG
    b. SWD
    c. Built-in JTAG/USB
    d. UART

5. Which file defines a PlatformIO project?

    a. `CMakeLists.txt`
    b. `sdkconfig`
    c. `pio`
    d. `platformio.ini`

Further reading

- *Hands-On RTOS with Microcontrollers*, Brian Amos, Packt Publishing (https://www.packtpub.com/product/hands-on-rtos-with-microcontrollers/9781838826734): Although the book uses STM32 in the examples, it is a great resource to learn about FreeRTOS. *Chapter 20* discusses FreeRTOS on multi-core systems such as ESP32.
- *Embedded Systems Architecture – Second Edition*, Daniele Lacamera, Packt Publishing (https://www.packtpub.com/product/embedded-systems-architecture-second-edition/9781803239545): A good reading to learn the embedded systems in general. *Part 4* talks about multithreading.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

3

Using ESP32 Peripherals

In the previous chapter, we discussed the available tools that we can use to develop applications on ESP32. We learned about ESP-IDF, the PlatformIO IDE, and debugging and testing applications. Now, it is time to use the ESP32 peripherals in applications. Peripherals are how MCUs connect to the outer world. The ESP32 series chips provide a wide range of peripherals. Depending on the product vision, each family may have different types of peripherals and different numbers of channels of peripherals. For example, the ESP32-S2/S3 series has more GPIO pins compared to other families and features **USB OTG** (**On-the-Go**) to act as a USB host, which allows us to connect other USB devices, such as a keyboard or mouse. On the other hand, the ESP32-C family has a reduced set of peripherals to achieve much lower costs.

In this chapter, we will learn how to use some popular peripherals of ESP32 in the example applications. The topics are:

- Driving **General-Purpose Input/Output** (**GPIO**)
- Interfacing with sensors over **Inter-Integrated Circuit** (**I^2C**)
- Integrating with SD cards over **Serial Peripheral Interface** (**SPI**)
- Audio output over **Inter-IC Sound** (**I^2S**)

- Developing graphical user interfaces on **Liquid-Crystal Display** (**LCD**)

Technical requirements

We will use Visual Studio Code and ESP-IDF command-line tools to create, develop, flash, and monitor the applications in this chapter.

As hardware, both of the development kits, ESP32-C3-DevKitM-1 and ESP32-S3 Box Lite, will be employed. The sensors and other hardware components used in this chapter are:

- A 55 mm LED
- A 220 Ω resistor
- A tactile switch
- BME280 – temperature, humidity, pressure sensor
- TSL2561 – ambient light sensor
- An SD card breakout board
- A micro-SD card
- 4x 10K Ω resistors

The source code for the examples is located in the repository found at this link: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch3

Driving General-Purpose Input/Output (GPIO)

Fundamentally, a sensor is any device that generates some sort of output when exposed to a phenomenon – say, temperature,

humidity, light, vibration, and so on. In an IoT application, we use sensors as data sources by connecting their output via an interface. **General-Purpose Input/Output** (**GPIO**) is the simplest form of communication interface in the list of peripherals. It provides high or low values. For instance, it is quite possible to read the status of a contact sensor by interfacing it with GPIO, since a contact sensor can only be either open or closed. Other, more complex sensors may require different types of interfaces, such as **Inter-Integrated Circuit** (**I^2C**) or **Serial Peripheral Interface** (**SPI**).

Actuators are on the output side of IoT solutions. They change their state according to an analog or digital signal coming from the microcontroller and generate output to the environment. Some examples are a buzzer to make sound, an LED to emit light, a relay to switch on/off, or a motor to create motion. We can use GPIO to drive on/off devices if they implement a GPIO interface for it.

The most basic skill with any embedded development is using GPIO pins to read from sensors and control actuators. In the next example, we will configure ESP32 pins for I/O and use them.

A schematic shows the components, interfaces, and internal connections of a circuit. While working on an IoT device, keep the schematic at hand all the time so that you can easily refer to it when you need to find pin connections.You can download the schematic for ESP32-S3 Box Lite here: https://github.com/espressif/esp-box/blob/master/hardware/esp32_s3_box_lite_M

```
B_V1.1/schematic/SCH_ESP32-S3-BOX-
Lite_MB_V1.1_20211221.pdf
```

Let's have an example where we use the GPIO pins of ESP32.

Turning an LED on/off by using a button

The goal of this example is to toggle the state of an LED when a button is pressed. Press the button, then the LED is on; another press, the LED is off. Therefore, the button is the sensor in this example and the LED is the actuator.

The hardware components of this example are:

- ESP32-S3 Box Lite
- A 55 mm LED
- A 220 Ω resistor
- A tactile switch

And the Fritzing sketch of the circuitry is:

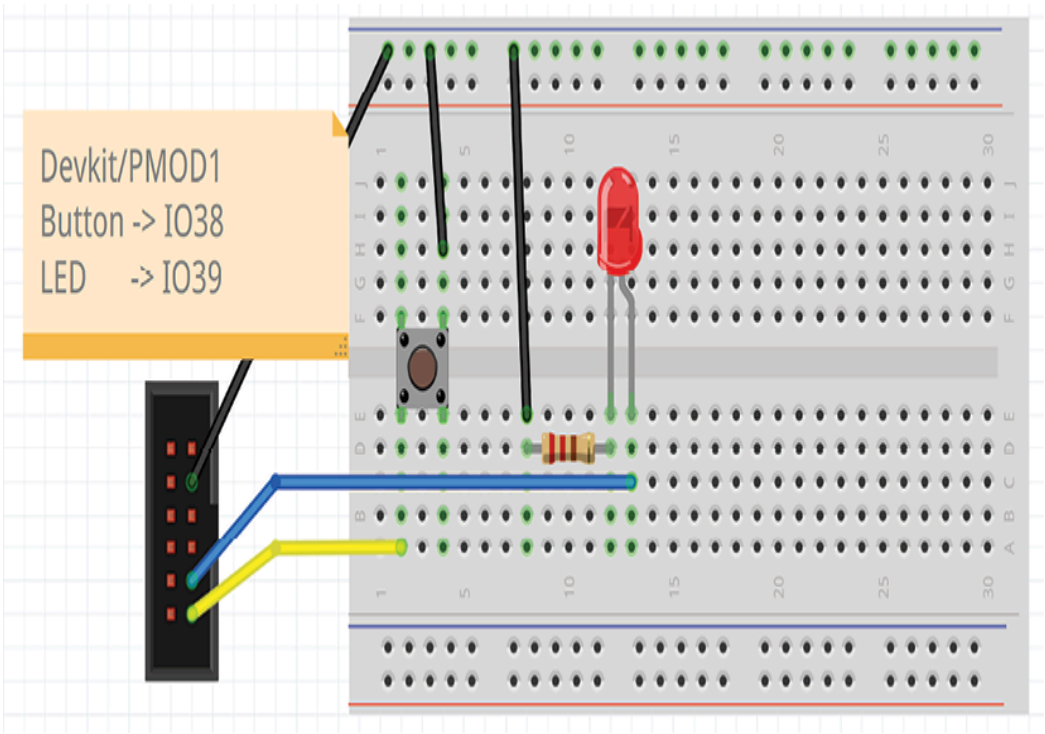


Figure 3.1: Fritzing sketch of the GPIO example

The terminals of the button are connected to PMOD1/GPIO38 and GND. We will set the internal pullup resistor of the GPIO38 pin while configuring it so that when the button is pressed, it will read a **LOW** value (the label on the devkit uses short forms for the GPIO names, e.g., G38 for GPIO38). The following figure shows the label on the back side of the devkit:

Figure 3.2: ESP32-S3 Box Lite headers

The LED is on GPIO39 as an output pin. We use a resistor to limit the current and protect the LED.

Make sure the shorter leg (cathode) of the LED is connected to GND through the resistor and the longer leg (anode) is connected to GPIO39. If it is wired in reverse, it won't work.

After having this setup ready, we can create an ESP-IDF project for ESP32-S3-Box Lite next.

Creating a project

Let's create and configure an ESP-IDF project as follows:

1. Start an ESP-IDF project in any way you wish. My personal preference is to use command-line tools as follows:

```
$ source ~/esp/esp-idf/export.sh
Setting IDF_PATH to '/home/ozan/esp/esp-idf'
Detecting the Python interpreter
Checking "python" …
<logs removed>
$ idf.py create-project led_button_ex
Executing action: create-project
The project was created in led_button_ex
$ cd led_button_ex
```

2. Set the target as ESP32-S3 since we will develop the project on the ESP32-S3-Box Lite kit:

```
$ idf.py set-target esp32s3
Add "set-target's" dependency "fullclean" to the
Executing action: fullclean
<logs removed>
```

3. See that you have the following directory structure:

```
$ tree .
├── build
<more files and directories>
├──CMakeLists.txt
├── main
│   ├── CMakeLists.txt
│   └── led_button_ex.c
└── sdkconfig
529 directories, 146 files
```

4. Rename the C source file to a `.cpp` file. This way, `idf.py` selects the C++ compiler of the toolchain:

```
$ mv main/led_button_ex.c main/led_button_ex.cpp
```

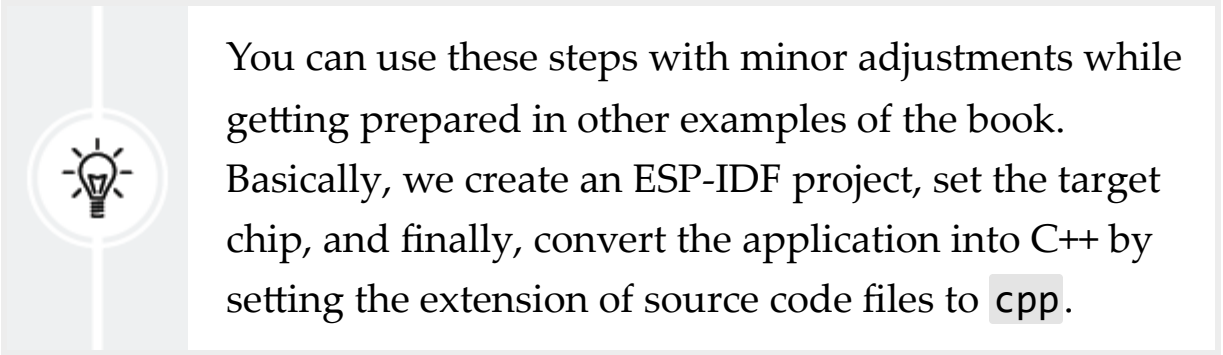
5. Every IDF application must be registered as an IDF component with its source files and include directories. Update the content of the `main/CMakeLists.txt` file as follows:

```
idf_component_register( SRCS "led_button_ex.cpp"
```

6. Start the VS Code IDE:

```
$ code
```

With these steps, our project is ready to develop.



You can use these steps with minor adjustments while getting prepared in other examples of the book. Basically, we create an ESP-IDF project, set the target chip, and finally, convert the application into C++ by setting the extension of source code files to `cpp`.

Let's continue in the VS Code environment to code the application.

Coding the application

We will implement an LED class and a button class and integrate them into the `app_main` function. Let's start with adding a new file in the `main` directory of the project for the LED, name it `AppLed.hpp`, and edit the content as follows:

```
#pragma once
#include "driver/gpio.h"
namespace app
{
    class AppLed
    {
    public:
        void init(void)
        {
            gpio_config_t config{GPIO_SEL_39,
                                 GPIO_MODE_OUTPUT,
                                 GPIO_PULLUP_DISABLE
                                 GPIO_PULLDOWN_DISAB
                                 GPIO_INTR_DISABLE};
            gpio_config(&config);
        }
```

The header file for the GPIO structures and functions is `driver/gpio.h`. After including it, we define the LED class, `AppLed`. The `init` function of `AppLed` initializes pin 39 of the devkit as output. To do that, we define a variable, `config`, of type `gpio_config_t` and pass it to the `gpio_config` function as a parameter. `gpio_config_t` is basically a structure and stores configuration values for a GPIO pin. Here, we configure the GPIO

pin 39 as output by setting the GPIO mode as `GPIO_MODE_OUTPUT`. The `init` function is done and we need one more function to turn the LED on/off next:

```
        void set(bool val)
        {
            gpio_set_level(GPIO_NUM_39, static_cast
        }
    }; // end of class
} // end of namespace
```

The `set` function of `AppLed` is easy. It takes a `bool` parameter and calls the `gpio_set_level` function with this value on `GPIO_NUM_39`. That is it.

Next comes the button handler implementation. For this, we develop a new class in a new file. The path of the file is `main/AppButton.hpp` and the name of the class is `AppButton`. We start by including the header files as usual:

```
#pragma once
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
namespace
{
```

```
    void button_handler(void *arg);
}
```

In addition to `driver/gpio.h`, we also need the FreeRTOS header files for the button implementation. Right after them, we forward declare a regular C function as an interrupt handler. I'll explain this function in detail when we define it. Let's continue with the class implementation now:

```
namespace app
{
    class AppButton
    {
    private:
        bool state;
        std::function<void(bool)> pressedHandler;
    public:
        AppButton(std:function<void(bool)> h) : stat
    pressedHandler(h) {}
```

In the `private` section of the `AppButton` class, we have two member variables. The `state` variable holds the toggle state. Since we use a tactile button, each click on the button will toggle its internal conceptual state. The other member variable is `pressedHandler`, which is of the `std:function<void(bool)>` type. It is the one that will be called when a state toggle occurs. In the `public` section, we define the class constructor. Its parameter is

the function to be set as `pressedHandler`. Having the basics implemented, we can move on to the `init` function of the class:

```
        void init(void)
        {
            gpio_config_t config{GPIO_SEL_38,
                                 GPIO_MODE_INPUT,
                                 GPIO_PULLUP_ENABLE,
                                 GPIO_PULLDOWN_DISAB
                                 GPIO_INTR_POSEDGE};
            gpio_config(&config);
            gpio_install_isr_service(0);
            gpio_isr_handler_add(GPIO_NUM_38, buttor
        }
```

The GPIO initialization for the button is a bit different. Let's start with discussing the configuration. First of all, it is an input; thus, we set its mode as `GPIO_MODE_INPUT`. Then, we configure its pullup as enabled so that when we push and release the button, it generates low and high logic levels sequentially on the GPIO pin. We also enable the interrupt handler for the button to be able to receive logic-level changes. It will trigger on the positive edge, i.e., from low to high, as we instructed by passing the interrupt type as `GPIO_INTR_POSEDGE`. The `gpio_install_isr_service` function enables the interrupt service and, finally, we connect the press event to the interrupt handling function, `button_handler`, by calling the `gpio_isr_handler_add` function. The last parameter of the `gpio_isr_handler_add` function is a pointer to be passed to the

`button_handler` function when it is invoked. We pass the `this` pointer to link the interrupt handler and the button object.

We are not done with the class implementation yet. Let's develop the `toggle` function next:

```
        void toggle(void)
        {
            state = !state;
            pressedHandler(state);
        }
    }; // end of class
} // end of namespace
```

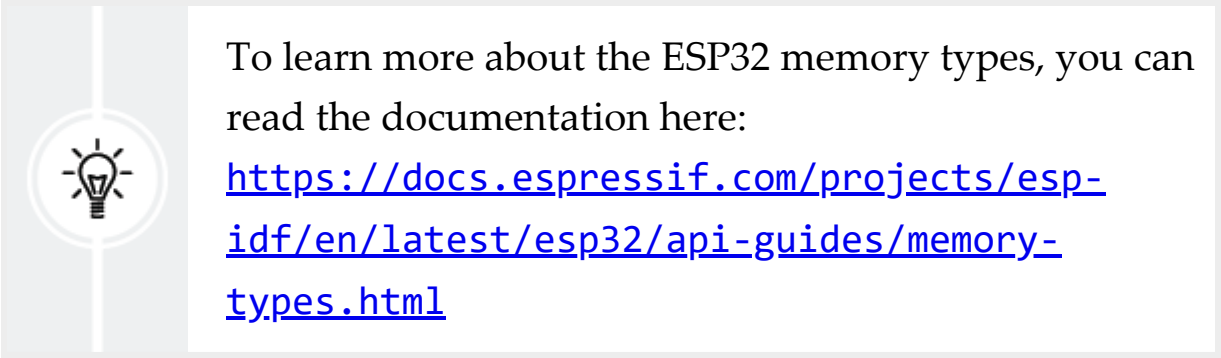
The purpose of the `toggle` function is simply to reverse the internal state and let the outer world know about this change by calling the `pressedHandler` callback.

Lastly, we define the real interrupt handler in the anonymous namespace as follows:

```
namespace
{
    IRAM_ATTR void button_handler(void *arg)
    {
        static volatile TickType_t next = 0;
        TickType_t now = xTaskGetTickCountFromISR();
        if (now > next)
        {
            auto btn = reinterpret_cast<app:AppButt
            btn->toggle();
```

```
                next = now + 500 / portTICK_PERIOD_MS;
            }
        }
    }
```

We mark the interrupt handler as `IRAM_ATTR`. Interrupt handlers must run in **Instruction RAM** (**IRAM**) for performance reasons and this macro instructs the linker for this.



To learn more about the ESP32 memory types, you can read the documentation here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/memory-types.html

In the function body, we get the time in ticks by calling a FreeRTOS function, `xTaskGetTickCountFromISR`. The postfix `FromISR` at the end of the function name shows that this function can be called inside an interrupt handler. It is a FreeRTOS convention to distinguish ISR context and task context. To disregard the electrical noise that may occur when we press the button (called **bouncing**), we check if `now` has passed `next`, and if so, we call the button's `toggle` function. Remember, the `arg` parameter has been provided as the button object pointer when we registered the `button_handler` function as an interrupt handler in the `init` function of the `AppButton` class. The statement `500 / portTICK_PERIOD_MS` converts 500 milliseconds into ticks to

calculate the `next` variable at the end. This finalizes the `AppButton` class implementation. Now we can develop the real application in the `app_main` function in `main/led_button_ex.cpp`:

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "AppLed.hpp"
#include "AppButton.hpp"
```

We include the headers for FreeRTOS and our classes. Then, we continue with the `app_main` function as follows:

```
extern "C" void app_main()
{
    app::AppLed led;
    auto fn = [&led](bool state)
    { led.set(state); };
    app::AppButton button(fn);
```

We define the LED and button objects and link them by using the lambda function, `fn`. Inside the `fn` function, we simply set the LED state with the given `state` value. The same function is passed to the `AppButton` constructor so that it will be called when a button press occurs. When the button is pressed, the state of the LED changes. Next, we will initialize the objects:

```
    led.init();
    button.init();
```

```
    vTaskSuspend(nullptr);
}
```

To initialize the GPIO pins, we call the `init` functions of both `led` and `button`. Then we suspend the `main` task by calling a FreeRTOS function, `vTaskSuspend`. The reason for this is that we don't want to lose the `led` and `button` objects, which provide the whole functionality of the application. The `main` task is suspended but the objects inside are still alive and run in the memory.

The button in the example is a sensor and we configured the GPIO pin that it is connected to as input. When we press the button, its handler function updates the LED state by setting/resetting the LED's GPIO pin, which is configured as digital output. The development is finished and it is time to test. We can compile and upload the firmware by using PlatformIO. When the firmware is uploaded successfully, we can try pressing the button and see how the LED toggles (fingers crossed).

Troubleshooting

Here are some checkpoints if the application fails to work as expected.

- Make sure the hardware setup is fine. Double-check the button is connected to GPIO38 and the LED is on GPIO39. Use a multimeter if needed.
- Check the GND pin is connected to the correct legs of the LED and button (for the LED, it is the short leg). Make sure the LED

is functional. Use a multimeter to test the LED if needed.

- It is good to have a multimeter and use it to ensure the components function properly.
- You can use the `ESP_LOGI` macro in various parts of the code, for example, in the button interrupt handler. If it prints a log when you press the button, then the GPIO configuration is correct.
- To test the `AppLed` class, you can run a simple loop in the `app_main` function, which toggles the LED every second.

In the next topic, we will talk about I^2C, a protocol to communicate with external devices.

Interfacing with sensors over Inter-Integrated Circuit (I^2C)

I^2C is a serial communication bus that supports multiple devices on the same line. Devices on the bus use 7-bit addressing. Two lines are needed for the I^2C interface: **Clock** (**CLK**) and **Serial Data** (**SDA**). The master device provides the clock to the bus. The following figure shows a typical architecture of an I^2C bus:

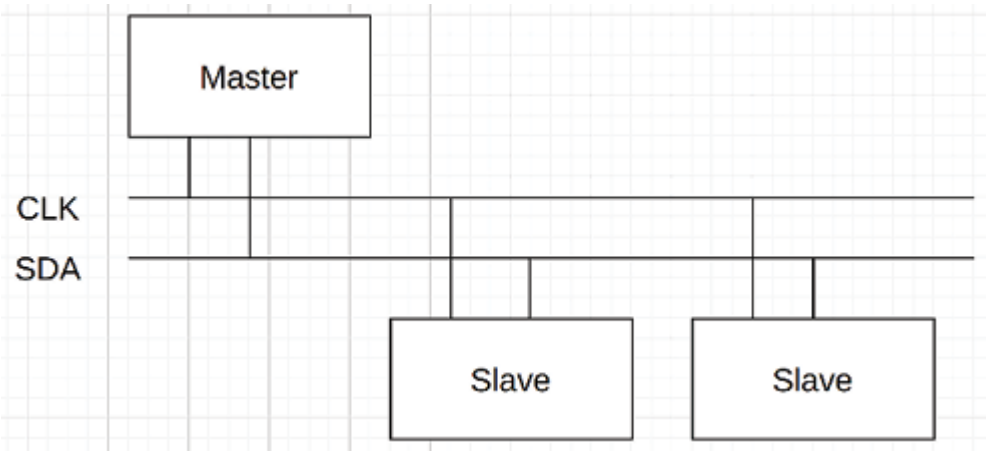


Figure 3.3: I^2C architecture

The only rule here is that each slave has to have its own unique address on the bus. In other words, two sensors with the same I^2C address cannot share a common bus. As defined in the protocol, the master and slaves exchange data over the SDA line. The master sends a command and the addressed slave replies to that command. Now, let's see an example of I^2C communication on ESP32 by implementing a multisensor application.

Developing a multisensor application

The purpose of this example is to develop an application in which we read values from different I^2C sensors and display the readings on the serial console. We will use two different sensors for this:

- BME280 – temperature, humidity, pressure sensor
- TSL2561 – ambient light sensor

BME280 is a Bosch product. With its low-power, small hardware footprint features, BME280 is a strong option in many IoT projects where such a sensor is needed. You can find many breakout boards

on the market with this sensor chip on it. BME280 can have two different addresses by configuring its SDO pin, – 0x76 if connected to GND and 0x77 when connected to Vcc.

TSL2561 is a luminosity sensor from AMS. Again, it is a popular IoT sensor, for light measurements this time. As suggested in its datasheet, it is a general-purpose light sensor, although TSL2561 is particularly designed for LCD screens to extend the battery life of the device that it is attached to. It supports three different addresses by connecting its address pin to GND (0x29) or VDD (0x49), or leaving it floating (0x39). As you may have already noticed, the addresses are all less than 0x80 since the I^2C addressing scheme uses only 7 bits. The following figure shows BME280 and TSL2561 breakout boards, which you can find from any electronics distributor.

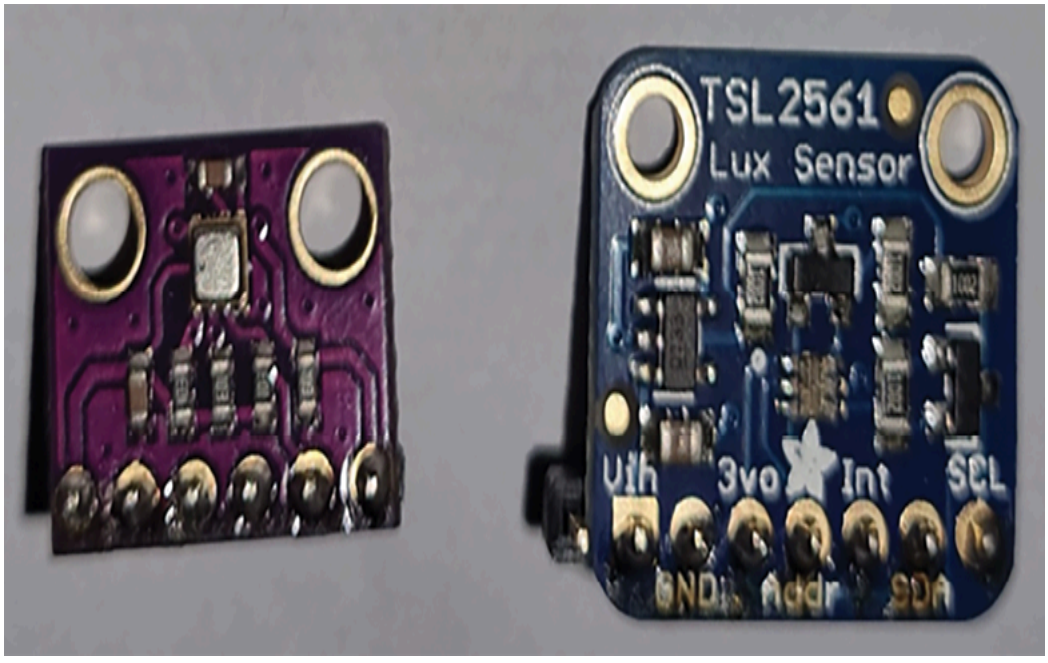


Figure 3.4: BME280 (left) and TSL2561 (right)

The hardware components of this example are:

- The ESP32-S3 Box Lite development kit
- A BME280 breakout board
- A TSL2561 breakout board

The following Fritzing sketch shows the connections of the hardware setup:

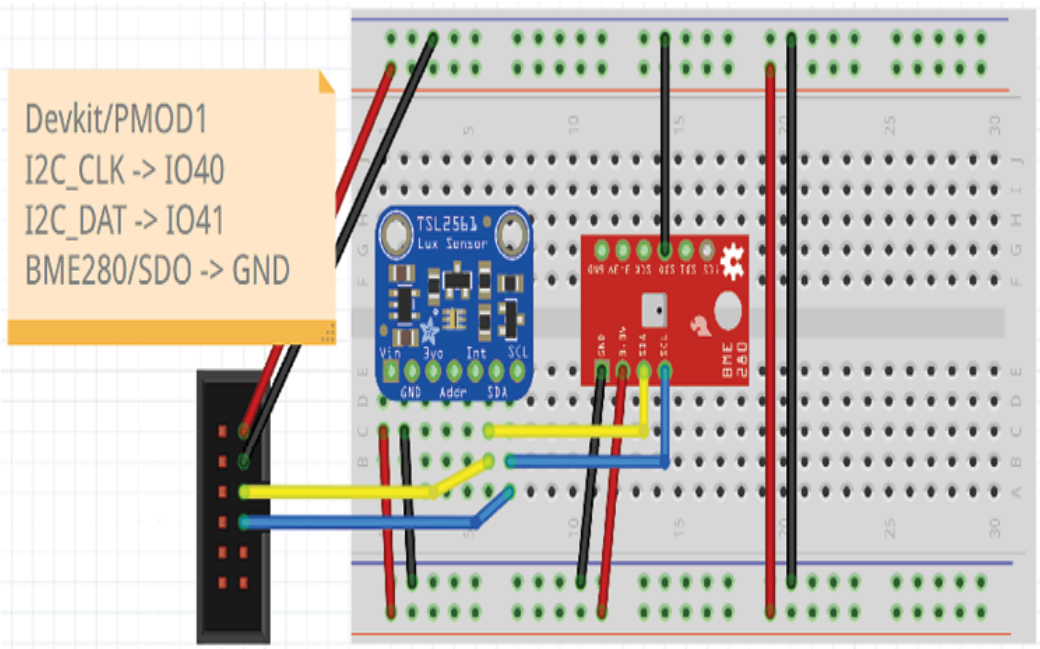


Figure 3.5: Fritzing sketch of the I^2C example

Here, the I^2C clock signal is on IO40 of the devkit and the SDA signal is on IO41. We connect the SDO pin of BME280 to GND so that its bus address becomes 0x76 and leave the TSL2561 address pin floating, making its address 0x39.

After preparing this circuitry, we can continue with creating a new ESP-IDF project.

Creating a project

Let's create and configure the project as follows:

1. Create an ESP-IDF project as we did in the GPIO example. Set its name to `i2c_ex` and the chip type of the project as `ESP32S3` since we will use ESP32-S3-Box Lite.
2. Create a new file in the project root and name it `sdkconfig.defaults`. This is a special file that the `idf.py` tool looks for to create a new `sdkconfig` file when it cannot find the `sdkconfig` file of the project. In this example, we want to do exactly that. Therefore, just remove the existing `sdkconfig` file to make the `idf.py` tool create a new `sdkconfig` with the configuration values in the `sdkconfig.defaults` file at the next build. Set the content of `sdkconfig.defaults` as the following (you can copy-paste the file from the GitHub repository here):

   ```
   CONFIG_ESPTOOLPY_FLASHSIZE_16MB=y
   CONFIG_ESPTOOLPY_FLASHSIZE="16MB"
   CONFIG_ESP_GDBSTUB_ENABLED=y
   CONFIG_ESP_GDBSTUB_SUPPORT_TASKS=y
   CONFIG_ESP_GDBSTUB_MAX_TASKS=32
   CONFIG_ESP_SYSTEM_PANIC_GDBSTUB=y
   CONFIG_GDBSTUB_SUPPORT_TASKS=y
   CONFIG_GDBSTUB_MAX_TASKS=32
   CONFIG_ESP32S2_PANIC_GDBSTUB=y
   ```

3. We need to use external components in this project and specify the directories that include them in the `CMakeLists.txt` file of

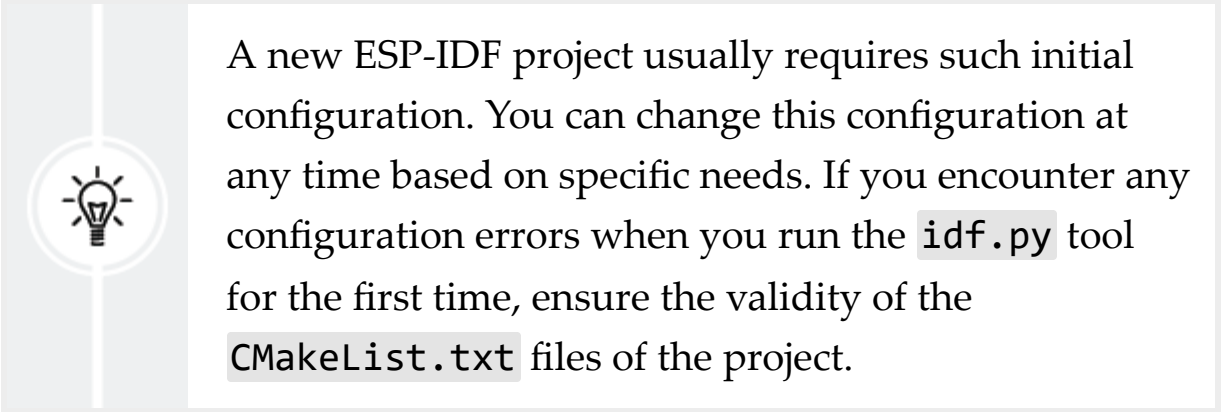
the project root. Set its content as the following or just copy-paste the file from the GitHub repository. Please note that the directories given below are relative to the project directory. This configuration will work without any issue if you have cloned the repository, but since we are now configuring a new project together, you need to copy those directories manually and set the correct paths:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components/esp-id
add_compile_options(-fdiagnostics-color=always -
project(i2c_ex)
```

4. Rename the source code file to `main/i2c_ex.cpp` and set the content of `main/CMakeLists.txt` as the following:

```
idf_component_register(SRCS "i2c_ex.cpp" INCLUDE
```

We can start the VS Code editor in the project directory.



A new ESP-IDF project usually requires such initial configuration. You can change this configuration at any time based on specific needs. If you encounter any configuration errors when you run the `idf.py` tool for the first time, ensure the validity of the `CMakeList.txt` files of the project.

Before moving on to the application development, I want to talk about the external components of the project shortly. We will discuss some useful third-party libraries in detail in the next chapter, but there is one specific library that we need to include in this project, which is the **ESP-IDF Components library** at this link: https://github.com/UncleRus/esp-idf-lib. It is quite popular among ESP32 developers because of the device drivers it provides. We will communicate with the project sensors, BME280 and TSL2561, with the help of this library. The library currently doesn't support ESP32-S3 fully, therefore I had to modify it a bit and included it in the book repository for ease of use.

Now, it is time to develop the application.

Coding the application

Let's start with the multisensor class implementation. For this, we create a source code file, `main/AppMultisensor.hpp`:

```
#pragma once
#include <cstring>
#include "tsl2561.h"
#include "bmp280.h"
namespace app
{
```

We include the header files for the sensors and open the `app` namespace. Then, we define the data structure that holds a multisensor reading:

```
struct SensorReading
{
    float pressure;
    float temperature;
    float humidity;
    uint32_t lux;
};
```

This structure is a plain C structure and has fields for temperature, humidity, pressure, and lux readings. Next, we continue with the class definition:

```
class AppMultisensor
{
private:
    tsl2561_t m_light_sensor;
    bmp280_t m_temp_sensor;
    bmp280_params_t m_bme280_params;
```

In the `private` section of the class, we declare the sensors. They come from the **ESP-IDF Components library** that we included in the project while configuring it. In the `public` section, we define the external functionality of the class as the following:

```
public:
    void init(void)
    {
        ESP_ERROR_CHECK(i2cdev_init());
        memset(&m_light_sensor, 0, sizeof(tsl25
```

```
            ESP_ERROR_CHECK(tsl2561_init_desc(&m_lig
                TSL2561_I2C_ADDR_FLOAT, I2C_NUM_1,
                GPIO_NUM_41, GPIO_NUM_40));
            ESP_ERROR_CHECK(tsl2561_init(& m_light_s
            ESP_ERROR_CHECK(bmp280_init_default_para
                &m_bme280_params));
            ESP_ERROR_CHECK(bmp280_init_desc(&m_temp
                BMP280_I2C_ADDRESS_0, I2C_NUM_1, GPI
                GPIO_NUM_40));
            ESP_ERROR_CHECK(bmp280_init(&m_temp_sens
                &m_bme280_params));
        }
```

In the `init` function, we initialize the sensors, hence the name. We first call the `i2cdev_init` function, which initializes the underlying I^2C structures. The sensor function names have sensor models as postfixes. The `tsl2561_init_desc` function configures TSL2561 with the I^2C address, the ESP32 I^2C port (here, it is `I2C_NUM_1`), and the GPIO pins for the SDA and CLK signals, respectively. ESP32-S3 has two I^2C ports and we associate the GPIO pins with the port `I2C_NUM_1`. BME280 also has a similar function for configuration. The `bmp280_init_desc` function passes the same I^2C parameters except the I^2C address for BME280 since they share the same bus (actually, there is another sensor, named BMP280, for only barometric pressure measurements. – I haven't tried this function on a BMP280 sensor myself but it probably works). As a final note on the `init` member function, we use the `ESP_ERROR_CHECK` macro to check whether everything goes well with the sensor initializations.

In the next member function, we expose an interface to return the sensor readings:

```
        SensorReading read(void)
        {
            SensorReading reading;
            bmp280_read_float(&m_temp_sensor,
                &reading.temperature, &reading.press
                &reading.humidity);
            tsl2561_read_lux(&m_light_sensor, &read
            return reading;
        }
    };
} // namespace app
```

The `read` member function simply calls the `bmp280_read_float` and `tsl2561_read_lux` to read the values from the sensors and returns them in a `SensorReading` structure. This completes the multisensor implementation. Now, we are ready to write the actual application in `main/i2c_ex.cpp`:

```
#include "esp_err.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "AppMultisensor.hpp"
```

We include the header files first, as usual. Then, we can immediately start with the `app_main` function as the entry point of the entire

application:

```
extern "C" void app_main(void)
{
    app::AppMultisensor multisensor;
    multisensor.init();
```

In the `app_main` function, we declare a multisensor and initialize it. The `init` function starts the I^2C communication. Having the multisensor ready for the I^2C communication, we create a loop to get the readings next:

```
    while (true)
    {
        vTaskDelay(10000 / portTICK_PERIOD_MS);
        auto reading = multisensor.read();
        ESP_LOGI(__func__, "pres: %f, temp: %f, hum
            lux: %d", reading.pressure, reading.temp
            reading.humidity, reading.lux);
    }
}
```

In the loop, we wait for 10 seconds and then read from the multisensor. The `ESP_LOGI` macro prints the values on the serial terminal.

The application is ready for testing. We can run it and monitor the serial output by using the `idf.py` tool:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<more logs>
```

The application should print the readings from the sensors. In case of any errors, you can try the options listed in the troubleshooting.

Troubleshooting

Here are some checkpoints if the application fails to work as expected:

- Make sure the hardware setup is correct. Check the addressing configuration for each sensor matches the code.
- Check the CLK and SDA signal lines are connected to the right pins of the sensors. You can try swapping the signal lines to see if anything changes. You can use a logic analyzer for further investigation.
- You can comment out the code for one sensor and only test the other one, then add the excluded sensor to see if anything changes.
- Any compile-time errors about not-found files are an indication of the configuration mistakes in `CMakeLists.txt` files. Make sure the project files and include paths are correct.

In the next topic, we will discuss SPI, an alternative to I^2C communication.

Integrating with SD cards over Serial Peripheral Interface (SPI)

Serial Peripheral Interface (**SPI**) is another serial communication protocol that can be used with devices. The main difference is that SPI requires at least four signal lines and one more each time when adding a new device to the bus.

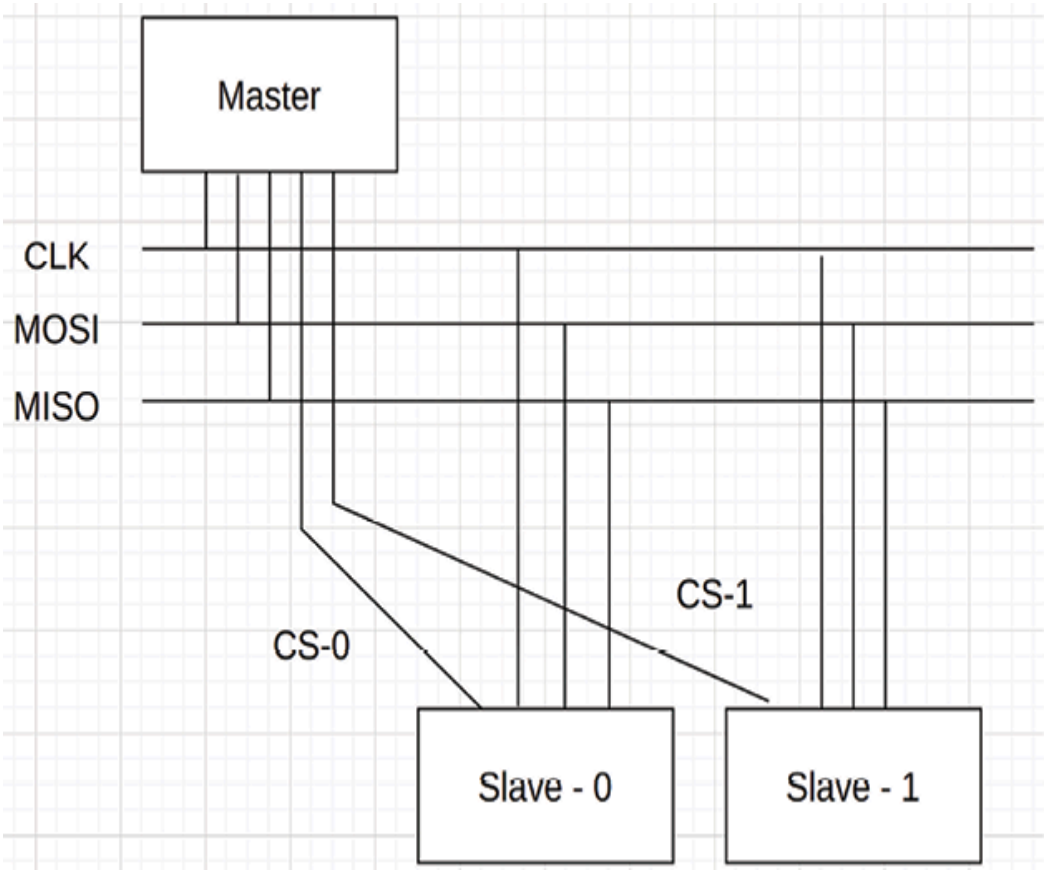


Figure 3.6: SPI architecture

In this figure, the master node provides a clock over the CLK line,uses the **Master-Out-Slave-In** (**MOSI**) line to send data, and receives data over the **Master-In-Slave-Out** (**MISO**) line. It can

communicate with a single slave node at a time. To select a slave node, it pulls the corresponding **Chip-Select** (**CS**) line to low.

Although SPI consumes more pin resources on an MCU, it achieves a higher data transfer rate compared to I^2C. Therefore, in some applications, such as where SD-card integration is needed, it makes sense to prefer the SPI protocol over I^2C.

In the next example, we will add SD card storage to our ESP32-C3-DevKitM-1 development kit.

Adding SD card storage

ESP32-C3-DevKitM-1 already has 4 MB of flash memory on it, but if you want to develop a data logger that needs to run for an extended period of time without intervention and without a WiFi connection, the existing flash memory might not be enough.

In this project, we will integrate an SD card for this purpose. The following figure shows an example of an SD card breakout board with SPI communication support.

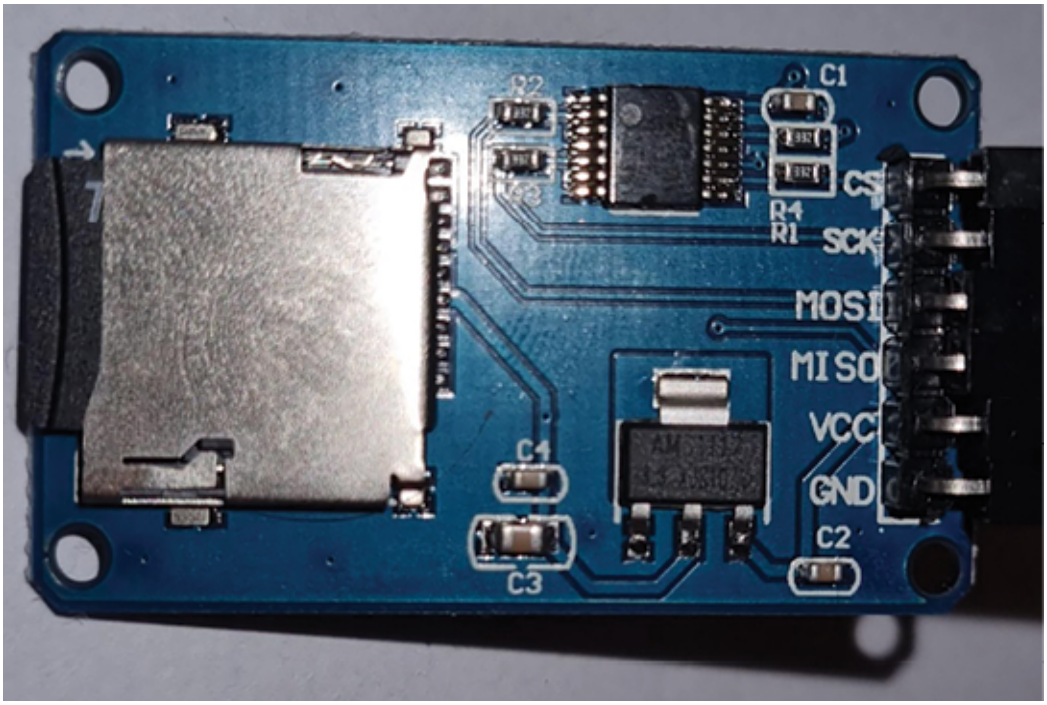


Figure 3.7: SD card breakout board with an SD card inserted

As you can see in this figure, there are four lines connected to four different GPIO pins of ESP32: CS, SCK, MOSI, and MISO. We use pullup resistors for these lines in order not to leave them floating before the SPI initialization. The following Fritzing sketch shows the connections in this example:

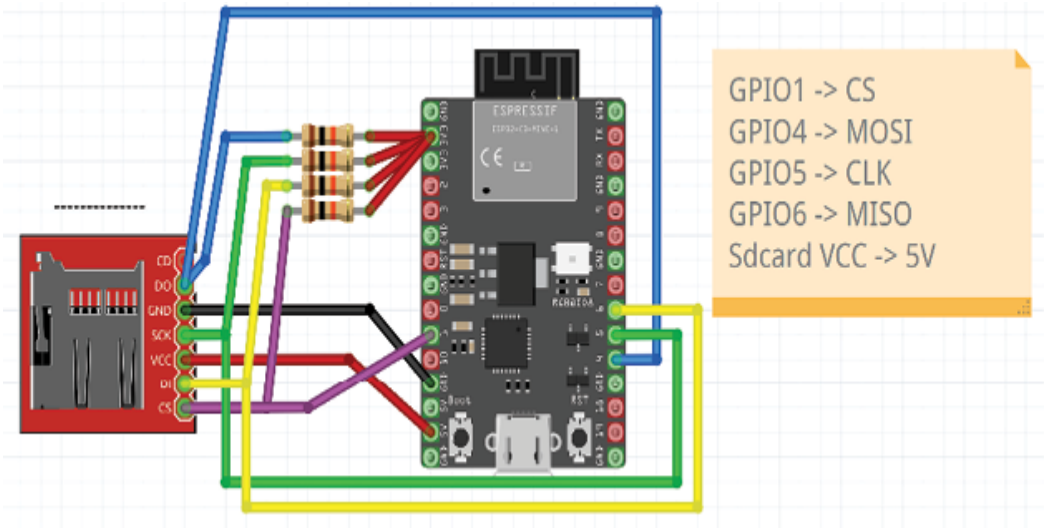


Figure 3.8: Fritzing sketch of the project

The hardware components of the project are:

- ESP32-C3-DevKitM-1
- An SD card breakout board (I have an AZ-Delivery board, but any other standard SD card breakout board on the market should be fine)
- A micro-SD card
- 4x 10K Ω resistors

In this setup, we need to connect the SPI lines to 3.3V with pullup resistors and power the SD card breakout board with 5V. After having this circuitry, we can create an ESP-IDF project.

Creating the project

Let's follow the steps below to create and configure the project:

1. Create an ESP-IDF project in any way you prefer. Set its name to `spi_ex` and the chip type of the project as `ESP32C3` since we will use ESP32-C3-DevKitM-1.
2. Add `sdkconfig.defaults` with the following content (you can copy-paste the file from the GitHub repository here):

```
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=y
CONFIG_ESPTOOLPY_FLASHSIZE="4MB"
CONFIG_ESP_GDBSTUB_ENABLED=y
CONFIG_ESP_GDBSTUB_SUPPORT_TASKS=y
CONFIG_ESP_GDBSTUB_MAX_TASKS=32
CONFIG_ESP_SYSTEM_PANIC_GDBSTUB=y
```

```
CONFIG_GDBSTUB_SUPPORT_TASKS=y
CONFIG_GDBSTUB_MAX_TASKS=32
```

3. Rename the source code file to `main/spi_ex.cpp` and set the content of `main/CMakeLists.txt` as the following:

```
idf_component_register(SRCS "spi_ex.cpp" INCLUDE
```

4. We can now remove the `sdkconfig` file to force the `idf.py` tool to generate `sdkconfig` next time we run it.

Now that we've created the project, we can move on to coding.

Coding the application

Let's implement a mock class first to generate data in a file named `main/AppSensor.hpp`:

```
#pragma once
#include <cinttypes>
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
namespace app
{
    struct SensorData
    {
        int temperature;
```

```
        int humidity;
        int pressure;
        int lux;
    };
```

We first include the required header files and in the `app` namespace, we define a plain C structure for dummy sensor data. Then, we can start with the declaration of the mock class:

```
    class AppSensor
    {
    private:
        SensorData readings[100];
        int cnt = 0;
        std::function<void(const uint8_t *, size_t)
```

The name of the mock class is `AppSensor`. In the `private` section, we have Three member variables. The first one holds dummy readings and the next one shows the number of stored readings in the array. `save` is a C++ function object to be called when the array is full. We implement the `read` function in the `private` section as follows:

```
        static void read(void *arg)
        {
            AppSensor *sensor = reinterpret_cast<Ap
            while (1)
```

```
            {
                vTaskDelay(pdMS_TO_TICKS(100));
                SensorData d{20, 50, 1000, 55};
                sensor->readings[sensor->cnt++] = d;
                if (sensor->cnt == 100)
                {
                    sensor->save(reinterpret_cast<
                      const uint8_t *>(sensor->read
                          * sizeof(SensorData));
                    sensor->cnt = 0;
                }
            }
        }
```

The `read` function is a static function to be run as a FreeRTOS task when we initialize the class. The `arg` function argument will hold the pointer to the class instance so we can just cast it back to an `AppSensor` pointer. In the task loop, we simply create a `SensorData` and add it to the array. When the number of records in the array reaches 100, we call the `save` function object with the `readings` array. Next, we will develop the initialization function of the class in the `public` section of the class implementation:

```
    public:
        void init(std::function<void(const uint8_t *
        {
            save = fn;
            if (xTaskCreate(AppSensor::read,
                            "sensor",
```

```
                                3072,
                                reinterpret_cast<void *>
                                5,
                                nullptr) == pdPASS)
                {
                    ESP_LOGI(__func__, "task created");
                }
            }
        }; // class
    } // namespace
```

The `init` function takes a function object parameter, `fn`. It is the `save` function that is called when the `readings` array is full. When we glue the pieces in the main application, we will call the `init` function with a function parameter that really saves data on the SD card. The `AppSensor` class is done and we will implement another class for the SD card access next. Let's create another C++ header file and name it `main/AppStorage.hpp`:

```
#pragma once
#include <fstream>
#include "esp_err.h"
#include "esp_log.h"
#include "esp_vfs_fat.h"
#include "sdmmc_cmd.h"
```

There are two new header files here. `esp_vfs_fat.h` provides the interface for the **File Allocation Table** (**FAT**) filesystem structures

and functions. `sdmmc_cmd.h` is for SD card access. Both come with ESP-IDF; we don't need any external libraries or frameworks to be added to the project. After the header files, we can jump into the class implementation:

```
namespace app
{
    class AppStorage
    {
    private:
        constexpr static const gpio_num_t PIN_NUM_MO
        constexpr static const gpio_num_t PIN_NUM_MI
        constexpr static const gpio_num_t PIN_NUM_CS
        constexpr static const gpio_num_t PIN_NUM_CL
        constexpr static const char *MOUNT_POINT{"/s
```

In the `private` section of the `AppStorage` class, we first define the GPIO pins of ESP32-C3 to be used for the SPI communication. As you would expect, they are the ones in the Fritzing sketch of the project. Then we define another constant value for the FAT mount point when we initialize the SD card. There are several other private member variables left that we need to define:

```
        sdmmc_card_t *m_card;
        spi_host_device_t m_host_slot;
        bool m_sdready;
```

`sdmmc_card_t` is a structure that keeps SD-card-related information, such as the type of the memory card, and register values of the SD card. `spi_host_device_t` is an enumeration to show the SPI host number. We will retrieve their values when we initialize the SD card device and use them later in the destructor of the class. The `m_sdready` member variable simply shows whether the SD card is initialized correctly and ready to use. Next, we implement the `public` section of the class:

```
    public:
        AppStorage() : m_card(nullptr), m_sdready{fa
        {
        }
        virtual ~AppStorage()
        {
            if (m_sdready)
            {
                esp_vfs_fat_sdcard_unmount(MOUNT_PO
                spi_bus_free(m_host_slot);
            }
        }
```

In the constructor, there is nothing interesting; we only set the initial values of the member variables. In the destructor, if the SD card is ready, we first unmount the FAT filesystem on `m_card`, then free the SPI slot associated with the SD card. The destructor releases all the resources that we allocate in the initialization function for the SD card communication. Next comes this initialization function:

```
esp_err_t init(void)
{
    esp_err_t ret{ESP_OK};
    esp_vfs_fat_mount_config_t mount_config
        .format_if_mount_failed = true,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024};
```

In the `init` function, we start with the local variables. The `mount_config` variable describes how we want to mount the filesystem. For example, here, it describes an instruction for the SD card driver to try to format the SD card with the given `allocation_unit_size` if the mount fails. It also says that the filesystem can have up to 5 files on it. Then we define two other variables for the SPI bus communication:

```
    spi_bus_config_t bus_cfg = {
        .mosi_io_num = PIN_NUM_MOSI,
        .miso_io_num = PIN_NUM_MISO,
        .sclk_io_num = PIN_NUM_CLK,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .max_transfer_sz = 4000,
    };
    sdmmc_host_t host = SDSPI_HOST_DEFAULT()
```

The `bus_cfg` variable describes the GPIO pin connections and the SPI bus speed with the SD card. The `host` variable provides an

abstraction for the application on top of the physical layer. We will use them next in order to initialize the SPI bus and mount the filesystem:

```
            ret = spi_bus_initialize((spi_host_devi
                   &bus_cfg, SDSPI_DEFAULT_DMA);
            if (ret != ESP_OK)
            {
                return ret;
            }
```

We call the `spi_bus_initialize` function first. It prepares the physical SPI bus. The `SDSPI_DEFAULT_DMA` parameter shows that the driver will use the **Direct Memory Access** (**DMA**) controller for high performance to communicate with the SD card:

```
            sdspi_device_config_t slot_config =
               SDSPI_DEVICE_CONFIG_DEFAULT();
            slot_config.gpio_cs = PIN_NUM_CS;
            slot_config.host_id = (spi_host_device_t
            m_host_slot = (spi_host_device_t)host.sl
            ret = esp_vfs_fat_sdspi_mount(MOUNT_POIN
                      &slot_config, &mount_config, &
            if (ret != ESP_OK)
            {
                return ret;
            }
```

The `slot_config` variable is another configuration variable that we need, for mounting the filesystem this time. We call the `esp_vfs_fat_sdspi_mount` function with the `MOUNT_POINT` static member that we defined in the `private` section. The initialization of the SD card seems a bit bulky but in fact, we only set the GPIO pins and decide on a mount point for the filesystem; the rest is only boilerplate. The `init` function finalizes with printing the SD card information as follows:

```
            sdmmc_card_print_info(stdout, m_card);
            m_sdready = true;
            return ret;
        } // end of init
```

The `sdmmc_card_print_info` function will simply print the SD card info on `stdout`. We now continue with the `save` member function of the class to actually store data in a file on the SD card:

```
        void save(const uint8_t *data, size_t len)
        {
            if (!m_sdready)
            {
                ESP_LOGW(__func__, "sdcard is not re
                return;
            }
```

The `save` function starts with a check whether the SD card is ready. If it is ready, we can proceed with the following:

```
            std::ofstream file{std::string(MOUNT_PO
                std::ios_base::binary | std::ios_bas
                std::ios_base:app};
```

We define a `std:ofstream` object, which is the file to be written. Its full path is `/sdcard/log.bin`. Then we need to check whether the file is really ready to append:

```
            if (!file.fail())
            {
                file.write((const char *)data, len);
                if (!file.good())
                {
                    ESP_LOGE(__func__, "file write f
                }
            }
            else
            {
                ESP_LOGE(__func__, "file open failed
            }
        } // save function
    }; // class
} // namespace
```

If the file is ready for the operation, we write the given data into the file.

`std:ofstream` is **Resource Acquisition Is Initialization**, or RAII, so there is no need to close the file explicitly. Here is more about RAII: https://en.cppreference.com/w/cpp/language/raii

We are done with the class implementations. It is time to glue them together in the main application, `main/spi_ex.cpp`:

```
#include <cinttypes>
#include "esp_log.h"
#include "AppStorage.hpp"
#include "AppSensor.hpp"
namespace
{
    app::AppStorage app_storage;
    app::AppSensor app_sensor;
}
```

We include the classes and define the objects in the anonymous namespace. Then, we add the `app_main` function:

```
extern "C" void app_main(void)
{
    if (app_storage.init() == ESP_OK)
    {
        auto fn = [](const uint8_t *data, size_t le
        { app_storage.save(data, len); };
```

```
            app_sensor.init(fn);
        }
        else
        {
            ESP_LOGE(__func__, "app_storage.init failed'
        }
    }
```

In the `app_main` function, we first initialize the `app_storage` global object by calling its `init` function to have an SD card in our application. If it succeeds, we define a lambda function, `fn`, in which we call the `save` function of the same object. Then we initialize the `app_sensor` object by passing the `fn` lambda function. It is the glue between the `app_storage` and `app_sensor` objects. When `app_sensor` generates a record, it will call the `fn` lambda function, resulting in data accumulating in the `log.bin` file on the SD card.

It is time to test the application.

Testing the application

We can run the following command to flash the application on the devkit and monitor it on the serial console:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
```

```
<logs removed>
I (311) sdspi_transaction: cmd=52, R1 response: comm
I (361) sdspi_transaction: cmd=5, R1 response: comma
Name: 00000
Type: SDSC
Speed: 20 MHz
Size: 1875MB
I (371) init: task created
```

The application reports the size of the SD card that I use as `1,875 MB`, which is true. Let's check the content of the binary file on the SD card by directly attaching it to the development machine.

```
$ xxd LOG.BIN | head -1
00000000: 1400 0000 3200 0000 e803 0000 3700 0000
```

Keeping in mind that the ESP32 products are little-endian, the data above corresponds to the integer values of 20, 50, 1000, and 55, exactly as we sent from the `app_sensor` object.

If you encounter a problem, you can check the following troubleshooting section to see if any of the listed items match your case.

Troubleshooting

Here are some points to review if the application fails to work as expected:

- Make sure the hardware setup is correct. Check the connections, especially the resistors for the 3.3V pullup.
- The SD card breakout board requires a 5V power source. ESP32-C3-DevKitM-1 has 5V pins that you can use for this purpose.
- Check whether the SD card is functional by attaching it to your machine and creating files on it.
- Check all the signal lines are connected to the right pins of the devkit. You can try swapping the MOSI/MISO signal lines to see if anything changes. You can use a logic analyzer for further investigation.

The next communication protocol that we will discuss is specifically designed for audio.

Audio output over Inter-IC Sound (I²S)

Inter-IC Sound (**I²S**) is another type of data interface but for audio. Essentially, it has three lines for the following:

- Data, **Data-In** (**DIN**), or **Data-Out** (**DOUT**)
- Clock or bit clock (BCLK)
- Channel select, **Word Select** (**WS**), or **Left-Right Clock** (**LRCLK**)

The interface is standard; however, the naming is not, as we see above. The data line carries stereo audio data for both the left (channel 0) and right (channel 1) channels. The channel select signal level indicates which channel's data is currently being transferred: it is low for the left channel and high for the right channel. Finally, the

clock line is a common clock for both ends provided by the master, which is usually the sending party in this type of communication.

In audio projects, we normally need a **Digital-Analog Converter** (**DAC**) to convert digital audio data to its analog counterpart and an amplifier to forward the analog output to a speaker in order to generate sound. Luckily, ESP32-S3-Box Lite has everything that we need to develop an integrated audio application and we don't have to deal with any hardware assembly in the example of this topic. It has an ES8156 Stereo Audio DAC and integrated speaker inside; thus, we can simply focus on the software development.

Developing a simple audio player

The goal in this example is to develop a basic MP3 player on ESP32-S3-Box Lite with play/pause and volume up/down functionality. We will store an MP3 file on the flash and use the front buttons of the devkit to implement the control functions of the player. As hardware, we only need ESP32-S3 Box Lite, and no other components or breakout boards to attach to the devkit. Let's create an ESP-IDF project as shown in the following steps:

1. Create an ESP-IDF project in any way you prefer. Set its name to `audio_ex` and the chip type of the project as `ESP32S3` since we will use ESP32-S3-Box Lite.
2. There is a long list of default values for `sdkconfig.defaults`. Please copy both the `sdkconfig` and `sdkconfig.defaults` files from the book repository at this link: https://github.com/PacktPublishing/Developing-IoT-

Projects-with-ESP32-2nd-edition/tree/main/ch3/audio_ex

3. We will store the MP3 file on the flash. For this, we need to define the flash partitions in a file named `partitions.csv`. Set its contents as the following (we will discuss the fields later):

```
nvs,          data, nvs,     0x10000, 0x6000,
phy_init,     data, phy,    , 0x1000,
factory,    app,  factory, ,        1M,
storage, data, spiffs,          ,        1M,
```

4. We need some external components, so update the content of `CMakeLists.txt` in the project root with the following (its path is given relative to the project directory. – if you clone the book repository, you can directly copy the `components` directory):

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
add_compile_options(-fdiagnostics-color=always -
project(audio_ex)
```

5. We will play an MP3 audio file from the flash. You can copy it from the book repository as `spiffs/mp3/music.mp3` (or any other MP3 file that you like with a size of less than 1 MB to fit into the flash partition that we defined in *Step 3*).
6. Rename the source code file `main/audio_ex.cpp` and set the content of `main/CMakeLists.txt` as the following:

```
idf_component_register( SRCS "audio_ex.cpp" INCL
spiffs_create_partition_image(storage ../spiffs
```

7. We can try building the project now:

```
$ idf.py set-target esp32s3
Adding "set-target's" dependency "fullclean" to
<logs removed>
$ idf.py build
Executing action: all (aliases: build)
<logs removed>
```

If everything goes well, we should see a success message explaining how to flash the application on the devkit. Before developing the application code, let's discuss several interesting points in the preparation steps.

In *Step 3*, we added a new file, `partitions.txt`, to the project. It really defines the flash partitions and the build system uses the table that comes with this file to generate the partition binary images. The structure of this file is:

```
# Name, Type, SubType, Offset, Size, Flags
```

The first field is the partition name and we can refer to a partition with its name anywhere in the project. The second and third fields define the partition type together. The `Offset` field shows the offset from the flash beginning. If nothing is specified for this field, it is

automatically calculated by adding the given sizes of the previous partitions. The next field denotes the size of the partition, and the final one is to mark the encrypted partitions. The MP3 file will be on the `storage` partition.

For more information about the custom partitions, you can refer to the documentation at this link: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/partition-tables.html

In *Step 6*, we updated the `main/CMakeLists.txt` file, and it has a directive in it for the build system about how to create the `storage` partition binary.

```
spiffs_create_partition_image(storage ../spiffs FLAS
```

This line basically says to use the `../spiffs` directory as the source of the `storage` binary. Just as a spoiler, when we mount the partition in the application, we will see the exact directory structure of the `../spiffs` directory with the files in it.

As a final check before coding, let's see what is in the `build/flasher_args.json` file:

```
    "flash_settings" : {
        "flash_mode": "dio",
        "flash_size": "16MB",
```

```
        "flash_freq": "80m"
    },
```

The `flash_size` field above comes from `sdkconfig.defaults`. The binaries are also listed in this file:

```
    "flash_files" : {
        "0x0" : "bootloader/bootloader.bin",
        "0x20000" : "audio_ex.bin",
        "0x8000" : "partition_table/partition-table
        "0x120000" : "storage.bin"
    }
```

These are the binary images that have been generated during the build. They will be flashed starting from the calculated memory addresses when we run `idf.py flash`.

If you have noticed, we have several other partitions defined in the `partitions.txt` file. One of them is `nvs`, which we can use for storing application settings. In this application, we can store, for instance, the volume level so that every time the application starts, the last volume level is restored to play the MP3 file. Enough talking; let's develop the application.

Coding the application

Let's write a class to handle application settings in `main/AppSettings.hpp`:

```
#pragma once
#include <cinttypes>
#include "esp_err.h"
#include "nvs_flash.h"
#include "nvs.h"
#define NAME_SPACE "app"
#define KEY "settings"
```

We include the header files for **Non-Volatile Storage** (**NVS**) access. ESP-IDF abstracts NVS access for us by implementing them in the framework. The NVS library divides the `nvs` partition into logical units by namespaces. The namespace that we will use for the application settings is `NAME_SPACE`. The library stores key-value pairs in namespaces. The key for the application settings is `KEY`. Next, we define the class:

```
namespace app
{
    class AppSettings
    {
    private:
        uint8_t m_volume;
```

In the `private` section of the `AppSettings` class, we only keep the volume level as the application settings. Next comes the `public` section of the class:

```
    public:
        AppSettings() : m_volume(50) {}
        uint8_t getVolume(void) const { return m_vo
```

The constructor sets the default value of `50` for the volume level and the `getVolume` function returns the current volume. The next function in the `public` section updates the volume level in the settings:

```
        void updateVolume(uint8_t vol)
      {
         if (vol == m_volume)
         {
           return;
         }
            nvs_handle_t my_handle{0};
            if (nvs_open(NAME_SPACE, NVS_READWRITE,
                    ESP_OK)
            {
                m_volume = vol;
                nvs_set_blob(my_handle, KEY, this,
                    sizeof(AppSettings));
                nvs_commit(my_handle);
                nvs_close(my_handle);
            }
        } // end of function
```

In the `updateVolume` function, we first check whether the given `vol` parameter matches the current value. If so, there is nothing to do and the function simply returns. If they are different, we will update the current volume and save it on the `nvs` partition. To achieve that, we open the NVS namespace, `NAME_SPACE`, by running the `nvs_open` function from the NVS library. Then we call the `nvs_set_blob` function to serialize the settings into an intermediary memory area before actually saving it on the flash. The settings here is the current `AppSettings` object. The `nvs_commit` function is the one that saves the data physically on the flash. Finally, we close the NVS access by calling the `nvs_close` function.

The last member function that we need to implement is the `init` function:

```
        void init(void)
        {
            if (nvs_flash_init() != ESP_OK)
            {
                nvs_flash_erase();
                if (nvs_flash_init() != ESP_OK)
                {
                    return;
                }
            }
```

The `init` function starts with the NVS partition initialization. If it fails, the `init` function just returns since the NVS is not available somehow (you can check the return values to understand the issue if

such a thing happens). When the NVS is ready, then we can try to open the NVS namespace to read the settings:

```
            nvs_handle_t my_handle{0};
            if (nvs_open(NAME_SPACE, NVS_READONLY, &
                    ESP_ERR_NVS_NOT_FOUND)
            {
                updateVolume(50);
            }
            else
            {
                size_t len = sizeof(AppSettings);
                if (nvs_get_blob(my_handle, KEY, th
                        ESP_OK)
                {
                    updateVolume(50);
                }
            }
            nvs_close(my_handle);
        } // function end
    }; // class end
} // namespace end
```

If the `nvs_open` function cannot find `NAME_SPACE` on the `nvs` partition, it means it is the first time the application has run; therefore, we need to write the default value, which also creates the NVS namespace.If the namespace exists, we deserialize the binary blob stored on the flash into the current `AppSettings` object by calling the `nvs_get_blob` function.

The next class to be implemented in the project is the button handler class in `main/AppButton.hpp`. The buttons are the user interface to control the audio player:

```
#pragma once
#include "bsp_btn.h"
#include "bsp_board.h"
namespace app
{
    using btn_handler_f = void (*)(void *);
    class AppButton
    {
    public:
        void init(btn_handler_f l, btn_handler_f m,
        {
            bsp_btn_register_callback(BOARD_BTN_ID_
               BUTTON_PRESS_DOWN, l, NULL);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, m, NULL);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, r, NULL);
        }
    }; // class end
} // namespace end
```

This is the entire `AppButton` implementation, thanks to the board support package! In the `init` function of the class, we simply set the button press handlers for the left, middle, and right buttons of the devkit, and that is it. Before moving on to the audio player

implementation, it would be a good idea to briefly talk about what happens behind the scenes in the board support package. The following figure is from the devkit schematic:

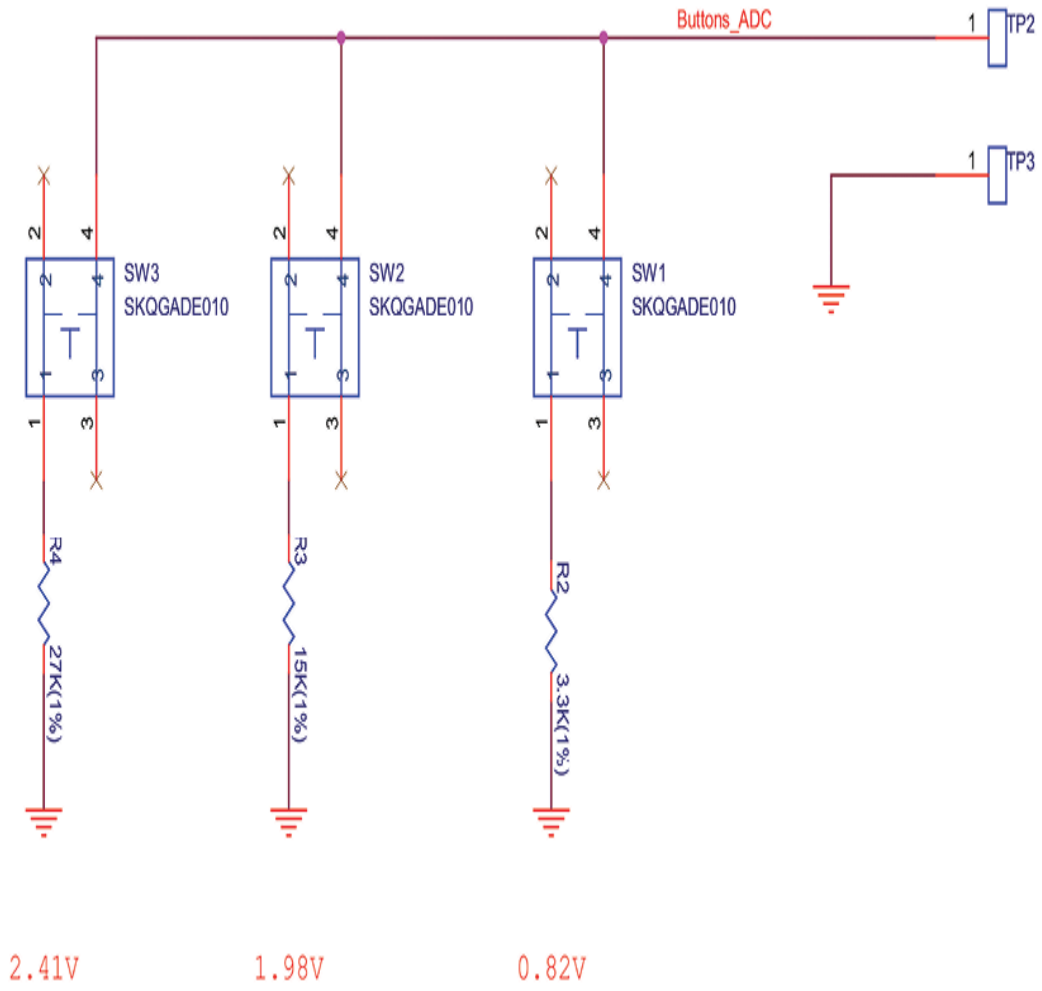


Figure 3.9: Buttons schematic (Source: https://github.com/espressif/esp-box/blob/master/hardware/esp32_s3_box_lite_Button_V1.1/schematic/SCH_ESP32-S3-BOX-Lite_Button_V1.1_20211125.pdf)

As you can see in the figure, the buttons on the devkit are simply a voltage-divider circuit. When a button is pressed, a different voltage value is read by one of the ADC pins on ESP32-S3 and the board support package runs the corresponding button-press handler if

provided. The `AppButton` class is another abstraction on top of the board support package.

Next comes the audio player in `main/AppAudio.hpp`:

```
#pragma once
#include <cstdio>
#include <cinttypes>
#include "esp_err.h"
#include "bsp_codec.h"
#include "bsp_board.h"
#include "bsp_storage.h"
#include "audio_player.h"
#include "AppSettings.hpp"
```

The devkit has one ADC and one DAC chip integrated into it for audio encode/decode purposes. As we talked about at the beginning of the chapter, ESP32-S3 Box Lite features an ES8156 for the audio output. The `bsp_codec.h` header file provides functionality to drive the audio chips. The `audio_player.h` header file is for playing any audio files (WAV or MP3) on the underlying audio system. The `bsp_storage.h` header file provides access to the filesystem on the flash, where the MP3 file resides. With that, we can implement the audio class:

```
namespace app
{
    class AppAudio
    {
    private:
```

```
        AppSettings &m_settings;
        bool m_playing;
        FILE *m_fp;
```

The `AppAudio` class has three member variables in its `private` section. As we discussed earlier, `m_settings` keeps the volume value on the NVS. The `m_fp` variable is a `FILE` pointer to the MP3 file, and the `m_playing` variable denotes whether the MP3 file is currently being played or not:

```
    public:
        AppAudio(AppSettings &settings) : m_settings
            m_playing(false), m_fp(nullptr) {}
        void init(audio_player_mute_fn fn)
        {
            audio_player_config_t config = {.port =
                                            .mute_fn
                                            .priorit

            audio_player_new(config);
        }
```

The `init` function initializes the audio player by calling the `audio_player_new` function. The audio chip is connected to the `I2S_NUM_0` port of ESP32S3. Next, we define the `mute` function:

```
        void mute(bool m)
        {
```

```
            bsp_codec_set_mute(m);
            if (!m)
            {
                bsp_codec_set_voice_volume(m_settir
            }
        }
```

In the `mute` member function, we call `bsp_codec_set_mute` to mute/unmute the device. If it is to be unmuted, then we restore the volume level from the settings. Let's develop another member function to play the MP3 file:

```
        void play(void)
        {
            if (!m_playing)
            {
                m_fp = fopen("/spiffs/mp3/music.mp3'
                audio_player_play(m_fp);
            }
            else
            {
                audio_player_pause();
            }
            m_playing = !m_playing;
        }
```

The `play` function will open the MP3 file from the filesystem, then the `audio_player_play` function will play it. The flip side is to

pause the player by calling the `audio_player_pause` function if the current state is not `m_playing`. The remaining member functions are for volume control:

```
        void volume_up(void)
        {
            uint8_t volume = m_settings.getVolume();
            if (volume < 100)
            {
                volume += 10;
                bsp_codec_set_voice_volume(volume);
                m_settings.updateVolume(volume);
            }
        }
        void volume_down(void)
        {
            uint8_t volume = m_settings.getVolume();
            if (volume > 0)
            {
                volume -= 10;
                bsp_codec_set_voice_volume(volume);
                m_settings.updateVolume(volume);
            }
        }
    }; // class end
} // namespace end
```

The `volume_up` and `volume_down` functions are very similar to each other, as expected. The former checks if the volume level is less than

100, then increases the volume by 10, the latter checks if the volume level is greater than 0 and decreases the volume by 10; In both functions, we first get the current volume from the settings, set the volume by calling the `bsp_codec_set_voice_volume` function, and update the settings.

We have implemented the classes and now we can integrate them to construct the application to operate as a whole in the `main/audio_ex.cpp` file:

```
#include "bsp_board.h"
#include "bsp_storage.h"
#include "AppSettings.hpp"
#include "AppAudio.hpp"
#include "AppButton.hpp"
```

We include the header files and then declare the application variables and functions in the anonymous namespace as follows:

```
namespace
{
    app::AppSettings m_app_settings;
    app::AppAudio m_app_audio(m_app_settings);
    app::AppButton m_app_btn;
    esp_err_t audio_mute_function(AUDIO_PLAYER_MUTE_
    void play_music(void *data);
    void volume_up(void *data);
```

```
    void volume_down(void *data);
}
```

We create the class instances and declare the function prototypes to control the music. These functions will be the glue between the music player and the button controls as we see in the `app_main` function implementation next:

```
extern "C" void app_main(void)
{
    bsp_board_init();
    bsp_board_power_ctrl(POWER_MODULE_AUDIO, true);
    bsp_spiffs_init("storage", "/spiffs", 2);
    m_app_settings.init();
    m_app_audio.init(audio_mute_function);
    m_app_btn.init(play_music, volume_down, volume_u
}
```

In the `app_main` function, we initialize the board and the audio system power and mount the `storage` partition as the `"/spiffs"` root directory. We call the `init` functions of all objects so that they can also initialize their internal states. The button `init` function is interesting because we bind the button presses to the real music player functionality by passing the music player callbacks. Let's see how we can implement these callbacks:

```
namespace
{
    void play_music(void *data)
    {
        m_app_audio.play();
    }
    void volume_up(void *data)
    {
        m_app_audio.volume_up();
    }
    void volume_down(void *data)
    {
        m_app_audio.volume_down();
    }
```

The callback functions couldn't be simpler. The only thing to do is just call the corresponding member function of the audio player. The play button will toggle the play state of the audio player. If the play state is stopped, then the audio player will play when the button is pressed, and vice versa. The volume control buttons similarly call the audio player's volume up and down functions. The last callback function in the application is the mute function that we passed to the audio player when we initialized it:

```
    esp_err_t audio_mute_function(AUDIO_PLAYER_MUTE_
    {
        m_app_audio.mute(setting == AUDIO_PLAYER_MU
        return ESP_OK;
```

```
    }
} // end of anonymous namespace
```

In this callback, we only check whether the input is mute or not and call the audio player object's `mute` function with a Boolean parameter denoting the requested state. If you're wondering why this callback has this signature, the reason is that we pass it directly to the underlying audio player configuration as required by its design. This completes the application coding and we can enjoy the application after flashing it onto the devkit.

Testing the application

We can flash and monitor the application as follows:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<logs removed>
I (572) I2S: DMA Malloc info, datalen=blocksize=640,
I (573) I2S: DMA Malloc info, datalen=blocksize=640,
I (573) I2S: I2S0, MCLK output by GPIO2
I (574) codec: Detected codec at 0x08. Name : ES724
I (574) codec: Detected codec at 0x10. Name : ES815
I (580) codec: init ES7243
I (581) codec: init ES8156
```

```
I (581) gpio: GPIO[46]| InputEn: 0| OutputEn: 1| Ope
I (651) bsp_spiffs: Partition size: total: 956561, u
```

Please check the logs coming from the application; they provide a lot of information about how the underlying framework and libraries do their jobs. We can press the buttons and listen to the music!

There are no hardware connections in this example project; therefore, if anything doesn't work as expected, please review the application code and debug it. You can also add simple log prints to see the application state and button presses.

In the next topic, we will learn how to use graphics on ESP32.

Developing graphical user interfaces on Liquid-Crystal Display (LCD)

There are several types of display technologies on the market for IoT applications, such as **Liquid-Crystal Display** (**LCD**), **Organic Light-Emitting Diode** (**OLED**) displays, **Thin Film Transistor** (**TFT**) displays, and e-paper technologies. They have pros and cons,therefore, it is necessary to select the display technology according to the requirements of the project. Some criteria can be:

- Price tag
- Power consumption

- Hardware resources to drive the display (I^2C vs SPI communication)
- Driver support
- Graphics capabilities, size, and resolution
- Color requirements

For example, TFTs generally show more graphics capabilities and high resolution, but higher energy consumption too. If the project requirements mandate the least amount of energy consumption, a reflective display, such as an e-paper type display, would be the right choice.

In the next example, we will use ESP32-S3-Box Lite, which comes with an integrated LCD. According to its online product brief:

- It has a 2.4-inch LCD with 240x320 resolution and supports RGB color
- The communication interface is SPI with 40 MHz speed
- The driver IC is ST7789V from Sitronix

After this brief introduction to the display technologies, let's have a practical example on the devkit.

A simple graphical user interface (GUI) on ESP32

The aim of this example is to develop a GUI that shows basic information about button presses on the devkit. As hardware, we will only use ESP32-S3 Box Lite.

Although it is quite possible to design and develop a GUI by directly using a driver library for ST7789V, we have a better option, **Light and Versatile Graphics Library** (**LVGL**). LVGL provides a great abstraction for underlying details and we, as IoT developers, only need to focus on the project requirements. Some key features of LVGL are:

- Open source and free (MIT license).
- Supporting GUI designer (SquareLine Studio is a licensed product with a fee but you won't need it in the examples of the book. – I will provide the source code for the GUI designs).
- A wide range of visual components (widgets), such as label, text area, button, slider, list, chart, checkbox, drop-down list, image, etc.
- Containers, such as canvas, window, and tab-view.
- Advanced graphics features, such as animations, anti-aliasing, opacity, etc.
- Very little memory footprint for the minimal set of features (64 KB flash, 16 KB RAM). However, it is worth noting that memory usage increases with more features enabled.
- Good documentation.

We will learn more about LVGL in the next chapter, but as a quick introduction, this example will provide important insights about this life-saving library.

The LVGL documentation can be found here for more information: https://docs.lvgl.io/master/intro/index.html

Let's create and configure the project next.

Creating the project

We can create our first GUI with LVGL with the following steps:

1. Create an ESP-IDF project in any way you prefer. Set its name to `ui_ex` and the chip type of the project as **ESP32S3** since we will use ESP32-S3-Box Lite.
2. There is a long list of default values in `sdkconfig.defaults`. Please copy it from the book repository given at this link: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/blob/main/ch3/ui_ex/sdkconfig.defaults
3. We need some external components, so update the content of `CMakeLists.txt` in the project root with the following (its path is given relative to the project directory. – if you clone the book repository, you can directly copy the `components` directory):

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
add_compile_options(-fdiagnostics-color=always -
project(ui_ex)
```

4. Rename the source code file to `main/main.cpp` and set the content of `main/CMakeLists.txt` as the following:

```
idf_component_register( SRCS "main.cpp" INCLUDE_
```

Now that we have the project configured, let's write the application code next.

Coding the application

We can start with the button class implementation in `main/AppButton.hpp`:

```
#pragma once
#include "bsp_btn.h"
#include "bsp_board.h"
#define APPBTN_LEFT BOARD_BTN_ID_PREV
#define APPBTN_MIDDLE BOARD_BTN_ID_ENTER
#define APPBTN_RIGHT BOARD_BTN_ID_NEXT
```

After including the headers and defining the application buttons, we move on to the `AppButton` class code:

```
namespace app
{
    using btn_handler_f = void (*)(void *);
    class AppButton
    {
    private:
        int m_type;
```

The `btn_handler_f` type defines the callback type for button handlers. The `AppButton` class corresponds to a single button and the `m_type` member variable denotes which button it is from the macro definitions above (`APPBTN_*`) when the class is instantiated. Then comes the `public` section of the class definition:

```
    public:
        AppButton(int type) : m_type(type) {}
        int getType(void) { return m_type; }
```

The constructor takes the button type as a parameter and initializes the private `m_type` member variable with the incoming value. We define the `init` function where we attach the button handlers next:

```
        void init(btn_handler_f btn_down_handler,
            btn_handler_f btn_up_handler){
          bsp_btn_register_callback( static_cast<
              board_btn_id_t>(m_type),
            BUTTON_PRESS_DOWN, btn_down_handler, th
          bsp_btn_register_callback(static_cast<
              board_btn_id_t>(m_type),
            BUTTON_PRESS_UP, btn_up_handler, this);
        }
```

In the `init` function, we register for both button-down and button-up events. Please note that we also pass the `this` pointer to the `bsp_btn_register_callback` function as a parameter to be able to

access the class instances from the button callback functions. There is one more function left to be implemented, as comes next:

```
        static AppButton &getObject(void *btn_ptr)
        {
            button_dev_t *btn_dev = reinterpret_cast
               button_dev_t *>(btn_ptr);
            return *(reinterpret_cast<app:AppButton
                (btn_dev->cb_user_data));
        }
    }; // class end
} // namespace end
```

The `getObject` is a static member function, which makes it a class member rather than an object member. The board support package defines all buttons as the type `button_dev_t` and it passes a `button_dev_t` pointer to the callback functions of the buttons as a means of distinguishing buttons in the callback functions. The same structure also has a `cb_user_data` field, which is the `this` pointer to the `AppButton` object that we passed in the `init` function. Therefore, the `getObject` static function provides access to the `AppButton` object when it is called in the button callbacks. This completes the `AppButton` class and we can continue with the GUI implementation in `main/AppUi.hpp`:

```
#pragma once
#include <mutex>
#include "bsp_lcd.h"
```

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
```

We include the header files as usual. To access the LVGL functions , we need to include both `lvgl/lvgl.h` and `lv_port/lv_port.h`. The latter is specific to hardware and it is a concrete implementation of the abstraction that is reserved by LVGL for hardware. Basically, it contains the hardware (display and any other input devices, touch screen, etc.) initialization and timer access for LVGL, as you might expect. The implementation of the `AppUi` class comes next:

```
namespace app
{
    class AppUi
    {
    private:
        lv_obj_t *m_label_title;
        static std::mutex m_ui_access;
        static void lvglTask(void *param)
        {
            while (true)
            {
                {
                    std::lock_guard<std:mutex> lock(
                    lv_task_handler();
                }
                vTaskDelay(pdMS_TO_TICKS(10));
```

```
            }
        }
```

In the `private` section of the `AppUi` class, we define our first LVGL object, a label. The general design principle in LVGL is to keep a base object pointer for widgets and create a widget by calling its LVGL `create` function, which returns the memory address of the newly created widget. Later, we use this pointer to configure or change the widget. Another design principle is to protect access to the LVGL objects for concurrency. For this, we define a static mutex, `m_ui_access`, in the `private` section.We also need a FreeRTOS task to render the GUI objects on the LCD. The `lvglTask` function does this job. In the loop of the task function, we define a `std:lock_guard` object on `m_ui_access` to protect the LGVL objects' internal states so that the `lv_task_handler` function can render them correctly. With the closing brace, the `lock` object dies, and the FreeRTOS task waits idle for 10 milliseconds. We can now continue with the `public` section as follows:

```
    public:
        void init(void)
        {
            lv_port_init();
            m_label_title = lv_label_create(lv_scr_a
            lv_obj_set_style_text_color(m_label_titl
                lv_color_make(0, 0, 0), LV_STATE_DEF
            lv_obj_set_style_text_font(m_label_title
                &lv_font_montserrat_24, LV_STATE_DEF
```

```
            lv_label_set_text(m_label_title, "Press
            lv_obj_align(m_label_title, LV_ALIGN_CEN
            xTaskCreatePinnedToCore(lvglTask, "lvgl"
               nullptr, 3, nullptr, 0);
            bsp_lcd_set_backlight(true);
        }
```

The `init` function initializes the class, hence the name. The first thing is to make the display hardware ready by calling the `lv_port_init` function. Then, we create the label widget with the help of the `lv_label_create` function. Each widget needs a parent container and we pass the active screen as the label's parent after retrieving the pointer to the active screen by calling the `lv_scr_act` function. After configuring the different features of the label widget, we create the FreeRTOS task for LVGL. Lastly, we turn on the LCD backlight in the initialization. Next, we will implement the final function in the class to set the label text:

```
        void setLabelText(const char *lbl_txt)
        {
            std::lock_guard<std::mutex> lock(m_ui_ac
            lv_label_set_text(m_label_title, lbl_txt
        }
    }; // class end
    std::mutex AppUi::m_ui_access;
} // namespace end
```

In the `setLabelText` function, we create a `std:lock_guard` object before accessing the label. We set the label text by calling the `lv_label_set_text` function and the class implementation ends. It is time to test the application.

Testing the application

We flash the devkit and play with the buttons to see how the label text changes with the button press and release events:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<logs removed>
I (724) lv_port: Try allocate two 320 * 20 display b
I (727) lv_port: Add KEYPAD input device to LVGL
```

There are no hardware connections in this example project; therefore, if anything doesn't work as expected, please review the application code and debug it. You can also add simple log prints to see the application state and button presses.

This was the last example in this chapter. Before moving on to the next chapter, you can answer the end-of-chapter quiz to review what we have learned so far.

Summary

ESP32 has a diverse range of peripherals to be employed in different scenarios. In this chapter, we covered several important peripherals that provide interaction with the outer world in IoT applications. The most basic one is **GPIO**, which can be configured for any digital input/output needs. **I^2C** and **SPI** are prominent in sensor communication. For audio output, we can use **I^2S**. It is similar to I^2C but supports stereo. We have also seen GUI development on LCD with the help of **LVGL**.

The next chapter will provide a list ofpopular IoT libraries with examples. When it comes to IoT applications, the use of third-party libraries is almost inevitable for every project to speed up product development and reduce costs. Therefore, it is always better to have an idea about the available options before jumping into development. We will discuss some of those libraries and use cases in the next chapter.

Questions

Here are some questions to reiterate the topics in this chapter:

1. What would be the right peripheral to use when you need to drive an LED?
    a. ADC
    b. GPIO
    c. I^2C
    d. DAC

2. Which of the following options supports multiple clients on the same bus with addressing?

    a. I^2S
    b. GPIO
    c. I^2C
    d. SPI

3. Why do SD cards use SPI communication with MCUs?

    a. Less error-prone
    b. Less resource-hungry (fewer pins to use)
    c. Higher transfer rate
    d. Higher memory capacity

4. The I^2S protocol defines a **Word-Select** (**WS**) signal:

    a. To transfer audio data
    b. To clock the bus
    c. To select the left or right channel
    d. To increase the data rate

5. Which of the following is not a display technology generally used in IoT applications?

    a. MRI
    b. LCD
    c. TFT
    d. OLED

Further reading

Another resource for more information about embedded systems peripherals is:

- *Embedded Systems Architecture*, Daniele Lacamera, Packt Publishing (https://www.packtpub.com/product/embedded-systems-architecture-second-edition/9781803239545): Teaches embedded systems in more general terms. *Chapter 6* explains general-purpose peripherals, including GPIO. *Chapter 7* is where local bus interfaces, such as I^2S or SPI, are discussed.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

4

Employing Third-Party Libraries in ESP32 Projects

In the previous chapters, we mostly stayed in the ESP-IDF environment and used the components and libraries that come with it. However, developing an IoT product usually means that you need help from third parties for practical reasons, such as cost, time, and market needs. It is obvious that every development that we decide to do in-house means more time and more money to burn in order to have a final, working product. However, we can cut some of the costs by using third-party libraries where possible – no need to reinvent the wheel. Market needs can also drive your development decisions. Let's say your product has to support a specific type of communication layer – for example, Matter, a popular smart home connectivity protocol. Then, it would make sense to use an SDK for it to ensure a smooth certification process for your product. In this chapter, we are not going to talk about Matter, but some other popular libraries that you might need in your next IoT project with ESP32.

There are different methods of integrating third-party libraries in an ESP-IDF project. For instance, we can define a library dependency to instruct the build system to download the library from the **IDF**

Component Registry or the online components registry maintained by Espressif Systems, or we can clone a library from GitHub. Some libraries are provided as a single header file to be copied into the project without any extra steps. We can also find libraries developed as IDF components so that we can directly add them to the extra components list of the build system after cloning. This chapter covers all these different methods with examples.

In this chapter, we will discuss the following **Free and Open Source Software** (**FOSS**) third-party libraries:

- LittleFS, an alternative to SPIFFS that comes with ESP-IDF
- nlohmann/json, a JSON library for modern C++, on its website
- Miniz, the data compression library included in ESP-IDF
- FlatBuffers, an efficient serialization library by Google
- **LVGL**, which stands for **Light and Versatile Embedded Graphics Library**
- ESP-IDF Components library (UncleRus)
- The frameworks and libraries by Espressif Systems

Technical requirements

We will use Visual Studio Code and ESP-IDF command-line tools to create, develop, flash, and monitor the applications during this chapter.

For hardware, only ESP32-S3 Box Lite will be employed. The sensors and other hardware components of this chapter are:

- **A light-dependent resistor** (**LDR**)
- A jumper wire (both ends are male)

- A pull-up resistor (10KΩ)

The source code in the examples is located in the repository found at this link: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch4

LittleFS

The design goal of LittleFS is to provide fail-safe filesystem access for MCUs even in case of power loss. All POSIX operations are atomic, which means that when a function call returns successfully, LittleFS makes sure the result is persistent on the filesystem. If you have hard requirements on fail-safety features of the underlying filesystem in a project, then LittleFS can be a good option. In this example, we are going to use a port of LittleFS for ESP-IDF.

You can find the library on GitHub here: https://github.com/joltwallet/esp_littlefs. It comes with an MIT license.

The goal in this example is to develop a door event logger. It will simply log when the door is opened and closed on the LittleFS filesystem. We will use ESP32-S3 Box Lite as the development kit and simulate a door sensor by clicking on a button of the devkit.

There are different methods to include a third-party library in an ESP-IDF project. One of them is the **IDF Component Manager**. Espressif maintains a list of libraries on its **IDF Component Registry** so that developers can easily access the popular libraries without hassle. They are all compatible with ESP-IDF; as a result, including a

library from the IDF Component Registry is only a matter of a single command. We will add LittleFS to our project by using this method.

The URL of the IDF Component Registry is https://components.espressif.com/. You can browse or search the compatible libraries at this address. You can also see the details about the IDF Component Manager on the official documentation here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/tools/idf-component-manager.html.

We will create an IDF project first to see how LittleFS is integrated and used in a project.

Creating a project

Let's prepare the project by following these steps:

1. Create an ESP-IDF project:

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project littlefs_ex
```

2. Copy the `sdkconfig.defaults` and `partitions.csv` files from the book repository into the project root.
3. We are going to need the board support package to drive the devkit's buttons. We set `EXTRA_COMPONENT_DIRS` to the BSP path in the project root `CMakeLists.txt` for this. The content of the file should be as follows:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
add_compile_options(-fdiagnostics-color=always -
project(littlefs_ex)
```

4. This is the step where we include the LittleFS library in the project by using the IDF Component Manager. The `idf.py` tool provides access to the IDF Component Manager. The following command will create a file, `main/idf_component.yml`, in the `main` folder. This file shows the libraries to be included in the project with their versions:

```
$ idf.py add-dependency joltwallet/littlefs==1.5
```

5. Compile the project and see whether the compilation finishes successfully. The build process will create the `managed_components` directory and download the library code there. Rename the `managed_components` directory to `components` to prevent some issues later:

```
$ idf.py build
$ mv managed_components components
```

6. Update the `main/CMakeLists.txt` file with the following content. We instruct the *CMake* build system to generate a LittleFS partition.

```
idf_component_register(SRCS "littlefs_ex.cpp" IN
littlefs_create_partition_image(storage ../files
```

7. Create the `files` directory in the project root and put a file inside. The build system will generate the partition from this directory as we have instructed in the previous step.

```
$ mkdir files && echo "this is an example file"
```

8. Rename the `main/littlefs_ex.c` source file to `main/littlefs_ex.cpp`; we are now ready to develop the project.

As a quick note before moving on to coding, the ESP32 port of the LittleFS library exposes the same API as the official ESP-IDF SPIFFS library. It is enough to replace the `spiffs` prefix with `littlefs` everywhere. Similarly, we use the `littlefs_create_partition_image` function instead of `spiffs_create_partition_image` in `main/CMakeLists.txt`. Now, we can start VS Code and develop the application.

Coding the application

Let's add a new file, `main/AppDoorLogger.hpp`, for the door logger and include some header files that we are going to need in the code:

```
#pragma once
#include <fstream>
```

```
#include <string>
#include <ctime>
#include "esp_log.h"
#include "bsp_board.h"
#include "bsp_btn.h"
#include "esp_littlefs.h"
```

The interesting header file here is the one for LittleFS, `esp_littlefs.h`. Then the class implementation comes next:

```
namespace app
{
    class AppDoorLogger
    {
    private:
        constexpr static const char *TAG{"door_logge
        constexpr static const char *FILENAME{"/file
```

The absolute path of the logging file is `/files/log.txt`. We continue the definitions in the `private` section:

```
        enum class eDoorState
        {
            OPENED,
            CLOSED
        };
        eDoorState m_door_state;
```

The `eDoorState` enum class shows the door state and the `m_door_state` member variable keeps the current state of the class instance. The next three `static` functions are for handling button presses on the devkit:

```
        static AppDoorLogger &getObject(void *btn_pt
        {
            button_dev_t *btn_dev = reinterpret_cast
                button_dev_t *>(btn_ptr);
            return *(reinterpret_cast<AppDoorLogger
                (btn_dev->cb_user_data));
        }
        static void doorOpened(void *btn_ptr)
        {
            AppDoorLogger &obj = getObject(btn_ptr);
            obj.m_door_state = eDoorState::OPENED;
            obj.log();
        }
        static void doorClosed(void *btn_ptr)
        {
            AppDoorLogger &obj = getObject(btn_ptr);
            obj.m_door_state = eDoorState::CLOSED;
            obj.log();
        }
```

The `getObject` function returns a reference to the class instance so that we can use its member functions. The `doorOpened` function is a button handler that will run when the left button is pressed and `doorClosed` will be called when the left button is released. Thus,

pressing the left button means that the door is opened, and releasing it marks the door as closed. After setting the door states in these functions, we call the `log` member function of the object to log the state change in the `log.txt` file. The implementation of the `log` function follows:

```
        void log(void)
        {
            std::ofstream log_file{FILENAME, std::ic
            log_file << "[" << esp_log_system_timest
            log_file << (m_door_state == eDoorState
                "opened" : "closed") << "\n";
        }
```

We open the log file as an output file stream in append mode and write the door state with a timestamp. Please note that `std::ofstream` is **implements the Resource Acquisition Is Initialization (RAII) technique**, so the file will be automatically closed when the function exits.

In the next `static` member function, we print the file content – i.e., logs collected:

```
        static void print(void *data)
        {
            std::ifstream log_file{FILENAME};
            std::string line1;
            while (!log_file.eof())
            {
```

```
                std::getline(log_file, line1);
                ESP_LOGI(TAG, "%s", line1.c_str());
            }
        }
```

The `print` function is also a button handler but for the middle button presses. We open the file as an input stream this time and print the lines on the serial output. This completes the `private` section of the class implementation. We implement the initialization in the `public` section like this:

```
    public:
        void init(void)
        {
            bsp_board_init();
            bsp_btn_register_callback(
                BOARD_BTN_ID_PREV,
                  BUTTON_PRESS_DOWN,
                  AppDoorLogger::doorOpened,
                  reinterpret_cast<void *>(this));
            bsp_btn_register_callback(
                BOARD_BTN_ID_PREV,
                  BUTTON_PRESS_UP,
                  AppDoorLogger::doorClosed,
                 reinterpret_cast<void *>(this));
            bsp_btn_register_callback(
                BOARD_BTN_ID_ENTER,
```

```
                BUTTON_PRESS_DOWN,
                AppDoorLogger::print, nullptr);
```

The `init` function starts with registering the button handlers. We attach the `AppDoorLogger::doorOpened` function to the left button press event, the `AppDoorLogger::doorClosed` function to the left button release event, and the `AppDoorLogger::print` function to the middle button press event. Then we continue with the filesystem initialization:

```
            esp_vfs_littlefs_conf_t conf = {
                .base_path = "/files",
                .partition_label = "storage",
                .format_if_mount_failed = true,
                .dont_mount = false,
            };
            esp_vfs_littlefs_register(&conf);
```

We first define a configuration variable where we specify the partition label to be mounted and the base path for it. The partition label comes from the `partitions.csv` file. The `esp_vfs_littlefs_register` function of the LittleFS library registers the partition given in the configuration variable. Next, we will try to open the log file to see if it really works:

```
            std::ofstream log_file{FILENAME, std::ic
            if (!log_file.is_open())
```

```
            {
                ESP_LOGE(TAG, "file open failed (%s)
            }
        } // init function end
    }; // class end
}  // namespace end
```

If the partition mount operation fails, then we cannot open the file and print an error message on the serial console. This finalizes the class implementation.

To complete the application, we edit the `main/littlefs_ex.cpp` source file and create an `AppDoorLogger` object in it as follows:

```
#include "AppDoorLogger.hpp"
namespace
{
    app::AppDoorLogger door_logger;
}
extern "C" void app_main(void)
{
    door_logger.init();
}
```

There is really nothing much to do in the `app_main` function. After creating the `AppDoorLogger` object, named `door_logger`, we only call the `init` function of the object and that is it! Let's test the application by flashing it on the devkit and observe how it responds to the button presses.

Testing the application

We can run the `idf.py` tool from the command line to test the application:

```
$ idf.py flash monitor
<logs removed>
I (16556) door_logger: [01:00:16.953]: opened
I (16556) door_logger: [01:00:17.638]: closed
I (16557) door_logger: [01:00:18.434]: opened
I (16557) door_logger: [01:00:19.200]: closed
```

After two presses and releases of the left button, we press the middle button to see the logs on the serial console.

The next library is *nlohmann/json* for JSON processing capabilities in our ESP32 applications.

Nlohmann-JSON

JavaScript Object Notation (**JSON**) is a common data exchange format that uses human-readable text, and nlohmann/json is a popular library that implements the JSON functionality in C++. It is released under the MIT license, which allows us to use the library in our projects without any limitations.

The IDF Component Registry doesn't have nlohmann/json; therefore, we can use it in a project by directly downloading the header file from its repository here: https://github.com/nlohmann/json

Nlohmann-JSON provides a single header file as the library; therefore, the only thing we need to do is simply to download this header file from its repository and include it in the project.

The goal of the example is to develop a touch logger. We will use ESP32-S3 Box Lite and a simple jumper wire to expose the GPIO9 pin of the devkit on Pmod header 2 so that we can use it as the touch sensor of the application. When we touch the pin of the jumper wire, it will generate a touch event. All touch events will be collected in the memory. The left button of the devkit will JSON-serialize the touch records. The middle button will navigate through the records and print them on the serial console in JSON format. Let's create an ESP-IDF project first.

Creating a project

We can create and configure a project as follows:

1. Create an ESP-IDF project in any way you prefer.
2. Copy the `sdkconfig.defaults` file from the project directory (`ch4/json_ex`) on GitHub into the local project directory.
3. We will use the buttons on the devkit. The BSP has the driver and we can include it by specifying the path to its IDF component in the root `CMakeLists.txt` file. Set the content of this file as follows:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
```

```
add_compile_options(-fdiagnostics-color=always -
project(json_ex)
```

4. Download the nlohmann/json header file into the `main` directory:

```
$ cd main && wget https://raw.githubusercontent.
```

5. Rename the source code file to `main/json_ex.cpp` and set the content of the `main/CMakeLists.txt` file to reflect this change:

```
idf_component_register(SRCS "json_ex.cpp" INCLUD
```

The project is ready for development, and we can move on to coding.

Coding the application

We can now write the `AppTouchLogger` class in the `main/AppTouchLogger.hpp` file:

```
#pragma once
#include <string>
#include <ctime>
#include <vector>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
```

```
#include "driver/touch_pad.h"
#include "json.hpp"
```

We begin by including the headers. The `driver/touch_pad.h` header file defines the functions and structures for the touch sensor peripheral of ESP32. The `json.hpp` header file is the one that we have downloaded from the nlohmann/json GitHub repository. Next comes the structure that defines a touch event:

```
namespace app
{
    struct TouchEvent_t
    {
        uint32_t timestamp;
        uint32_t pad_num;
        uint32_t intr_mask;
    };
```

In this structure, the `timestamp` field denotes the time that the event occurred, `pad_num` is the touch pad number (it is 9 in our project), and `intr_mask` shows the event type. We will only log the touch active and touch inactive events. Then we use a macro from the nlohmann/json library to serialize/deserialize the `TouchEvent_t` structure as follows:

```
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(TouchEvent_t,
```

```
            pad_num, intr_mask);
```

The nlohmann/json library provides a means to convert custom structures into JSON objects, and vice versa, by looking for two designated functions for the custom type: `to_json` and `from_json`. The `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro defines them for us.

The library has great documentation here: https://json.nlohmann.me/api/adl_serializer/

We can now define the class:

```
    class AppTouchLogger
    {
    private:
        constexpr static const char *TAG{"touch_logg
        std::vector<TouchEvent_t> m_touch_list;
```

We will collect all the events in the `m_touch_list` member variable of the class. These events will come from the touch interrupt handler, as we define next:

```
        static void touchsensor_interrupt_cb(void *a
        {
            TouchEvent_t touch_event{esp_log_timesta
                touch_pad_get_current_meas_channel()
                touch_pad_read_intr_status_mask()};
```

```
            AppTouchLogger &obj = *(reinterpret_cast
                AppTouchLogger *>(arg));
            obj.m_touch_list.push_back(touch_event);
        }
```

The `touchsensor_interrupt_cb` function is a `static` function in the class definition so we can provide it as an interrupt handler. We create a touch event first with the help of the ESP-IDF touch pad functions. The `touch_pad_read_intr_status_mask` function shows what happened (active/inactive) and the `touch_pad_get_current_meas_channel` function shows where it happened (only `Touch-9` here). We will pass the object address as a parameter when we register the interrupt handler, so the `arg` argument is a pointer to the `AppTouchLogger` object. We append the touch event to the object's touch list. Then, we can develop the initialization function of the class as follows:

```
    public:
        void init(void)
        {
            touch_pad_init();
            touch_pad_config(TOUCH_PAD_NUM9);
```

We initialize the touch peripheral first by calling the `touch_pad_init` and `touch_pad_config` functions. After that, we need to create a touch filter so that the peripheral can generate reliable touch events to the application:

```
            touch_filter_config_t filter_info = {
                .mode = TOUCH_PAD_FILTER_IIR_16,
                .debounce_cnt = 1,
                .noise_thr = 0,
                .jitter_step = 4,
                .smh_lvl = TOUCH_PAD_SMOOTH_IIR_2,
            };
            touch_pad_filter_set_config(&filter_info
            touch_pad_filter_enable();
            touch_pad_timeout_set(true, SOC_TOUCH_PA
            touch_pad_isr_register(touchsensor_inter
                TOUCH_PAD_INTR_MASK_ALL);
            touch_pad_intr_enable(TOUCH_PAD_INTR_MAS
                TOUCH_PAD_INTR_MASK_INACTIVE);
            touch_pad_set_fsm_mode(TOUCH_FSM_MODE_TI
            touch_pad_fsm_start();
            vTaskDelay(pdMS_TO_TICKS(50));
            uint32_t touch_value;
            touch_pad_read_benchmark(TOUCH_PAD_NUM9,
            touch_pad_set_thresh(TOUCH_PAD_NUM9, tou
        }
```

There is a bunch of configuration code in this snippet. For the sake of simplicity, we can just ignore it for now. The only thing to know for this example is that we register the touch sensor interrupt handler by calling the `touch_pad_isr_register` function and enable active/inactive interrupts by calling the `touch_pad_intr_enable` function.

The capacitive touch sensing and how it works are perfectly described in this application note: https://github.com/espressif/esp-iot-solution/blob/release/v1.0/documents/touch_pad_solution/touch_sensor_design_en.md

The final member function of the class is where we return the internal event vector as a JSON array. Here is how we can implement it:

```
        nlohmann::json serialize(void)
        {
            return m_touch_list;
        } // function end
    }; // class end
} // namespace end
```

Surprisingly, the `serialize` function has a very short body, thanks to the macro call at the beginning of the implementation that generates the `to_json` and `from_json` functions for us. The nlohmann/json library knows how to serialize the `TouchEvent_t` type and it also knows how to serialize C++ STL vectors. Therefore, when we return the internal `m_touch_list` member variable, the compiler can implicitly convert it to a `nlohmann::json` object.

The `AppTouchLogger` class is done, and now we can implement the application buttons to print the events in JSON format on the serial console. Let's create the `main/AppNavigator.hpp` file in the application and edit it:

```
#pragma once
#include "esp_log.h"
#include "bsp_board.h"
#include "bsp_btn.h"
#include "AppTouchLogger.hpp"
#include "json.hpp"
```

We include the header files, including the one that implements the `AppTouchLogger` class, and then we can define the `AppNavigator` class as follows:

```
namespace app
{
    class AppNavigator
    {
    private:
        constexpr static const char *TAG{"nav"};
        AppTouchLogger m_touch_logger;
        nlohmann::json m_touch_list;
        size_t m_list_pos{0};
```

In the `private` section, we define the member variables first. The `m_touch_logger` member is an instance of the `AppTouchLogger` class that we implemented previously. The touch list, `m_touch_list`, is a JSON object here. After the member variables, we continue with the static functions for the button press handling like this:

```
static AppNavigator &getObject(void *btn_ptr
{
    button_dev_t *btn_dev = reinterpret_cast
        button_dev_t *>(btn_ptr);
    return *(reinterpret_cast<
        AppNavigator *>(btn_dev->cb_user_dat
}
```

The `getObject` function extracts the `AppNavigator` object from the `btn_ptr` parameter that comes with the button handler calls. The next static function handles the left button press:

```
static void countPressed(void *btn_ptr)
{
    AppNavigator &obj = getObject(btn_ptr);
    obj.m_touch_list = obj.m_touch_logger.se
    obj.m_list_pos = 0;
    ESP_LOGI(TAG, "Touch event count: %u",
        obj.m_touch_list.size());
}
```

When we press the left button, we call the `serialize` function of the `AppTouchLogger` object. It returns the touch events as a JSON array and we assign it to the `m_touch_list` member variable. We also set the value of the `m_list_pos` member variable to 0 to start the navigation from the beginning of the JSON array.

The middle button is for printing the JSON records of the events on the serial console sequentially. Its handler comes next:

```
        static void nextPressed(void *btn_ptr)
        {
            AppNavigator &obj = getObject(btn_ptr);
            if (obj.m_touch_list.size() <= 0)
            {
                ESP_LOGW(TAG, "no touch detected");
                return;
            }
            ESP_LOGI(TAG, "%s",
                obj.m_touch_list[obj.m_list_pos].dur
            ++obj.m_list_pos;
            obj.m_list_pos %= obj.m_touch_list.size(
        }
```

We can access each event record by index. The index returns another `nlohmann::json` object, which is the JSON representation of a touch event. The `dump` function call on the JSON object returns a `std::string` value and we can print it on the serial console by accessing the underlying `char` array via the `c_str` function. The handlers are ready and we can develop the initialization function of the class in the `public` section as follows:

```
    public:
        void init(void)
        {
            bsp_board_init();
```

```
            bsp_btn_register_callback(
                BOARD_BTN_ID_PREV, BUTTON_PRESS_DOWN
                AppNavigator::countPressed,
                reinterpret_cast<void *>(this));
            bsp_btn_register_callback(
                BOARD_BTN_ID_ENTER, BUTTON_PRESS_DOW
                AppNavigator::nextPressed,
                reinterpret_cast<void *>(this));
            m_touch_logger.init();
        } // function end
    }; // class end
} // namespace end
```

In the `init` member function, we register the button handlers and initialize the `m_touch_logger` member. Finally, in the `main/json_ex.cpp` file, we will implement the application entry point – i.e., the `app_main` function:

```
#include "AppNavigator.hpp"
namespace
{
    app::AppNavigator nav;
}
extern "C" void app_main(void)
{
    nav.init();
}
```

The `app_main` function is very brief. We only call the `init` function of the `app::AppNavigator` object, and it handles the rest as we have already discussed. It is time to test the application on the devkit.

Testing the application

Now, we can flash the application and test it by touching the jumper wire pin. After holding and releasing the pin twice, we will have the following output on the serial output when we press the left button first and the middle button several times:

```
$ idf.py flash monitor
I (29374) nav: Touch event count: 4
I (33584) nav: {"intr_mask":2,"pad_num":9,"timestamp
I (34094) nav: {"intr_mask":4,"pad_num":9,"timestamp
I (34564) nav: {"intr_mask":2,"pad_num":9,"timestamp
I (35359) nav: {"intr_mask":4,"pad_num":9,"timestamp
```

The event count is 4, which is correct for holding and releasing the pin twice since we log both touch pad active/inactive events.

The next library that we will discuss is *Miniz*, the data compression library that comes with ESP-IDF.

Miniz

Miniz is a lossless data compression library that implements RFC 1950 and RFC 1951 for compression/decompression. The library port in ESP-IDF is licensed under the MIT license. ESP-IDF has already

imported it for its own purposes, but we can also use it freely. There is no need for library management. *Miniz* can be especially helpful when you need to transfer a large amount of data. After compressing and sending the data, the receiving side can easily decompress it with any library that implements the same RFCs.

Unfortunately, the documentation is very poor (in fact, there is almost none) for Miniz. You can see the examples in this repository: https://github.com/richgel999/miniz

In this example, we will simply compress and decompress a sample text by pressing the buttons on ESP32-S3 Box Lite. There is no other hardware required in this example. We can create a new project to see how to use this library in an application next.

Creating a project

Let's create and configure an ESP-IDF project as follows:

1. Create an ESP-IDF project in any way you prefer.
2. Copy the `sdkconfig.defaults` file from the project directory (`ch4/miniz_ex`) on GitHub into the local project directory.
3. We will use the buttons on the devkit. The BSP has the driver, and we can include it by specifying the path to the component in the root `CMakeLists.txt` file. Set the content of this file as the following:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
```

```
add_compile_options(-fdiagnostics-color=always -
project(miniz_ex)
```

4. Rename the source code file to `main/miniz_ex.cpp` and set the content of the `main/CMakeLists.txt` file to reflect this change:

```
idf_component_register(SRCS "miniz_ex.cpp" INCLU
```

Now that we have the project configured, we can continue with coding the application.

Coding the project

We can now start the VS Code editor and add a new file for the class that handles the data compression operations. The relative path of the file is `main/AppZip.hpp`:

```
#pragma once
#include <cstring>
#include "esp_log.h"
#include "esp_heap_caps.h"
#include "esp_system.h"
#include "rom/miniz.h"
```

The data compression functions and structures are declared in `rom/miniz.h`. We have another interesting header, `esp_heap_caps.h`, which is for accessing the **Pseudo-RAM** (**PSRAM**), the external memory, on the ESP32-S3 Box Kit. The devkit

has 512 KB of internal **Static RAM** (**SRAM**) and 8 MB of external PSRAM. Although the SRAM of the devkit is more than enough for our purposes in this example, we will allocate memory on the PSRAM to learn how to use it. We can define the class next:

```
namespace app
{
    class AppZip
    {
    private:
        constexpr static const size_t BUFFERSIZE{10
        char *m_data_buffer;
        char *m_compressed_buffer;
        char *m_decompressed_buffer;
        tdefl_compressor m_comp;
        tinfl_decompressor m_decomp;
```

We define the buffer pointers and the compressor/decompressor member variables in the `private` section. The types, `tdefl_compressor` and `tinfl_decompressor`, come from the Miniz library. The compressor and decompressor will operate on the defined buffers. Let's allocate memory for the buffers on PSRAM in the `init` function:

```
    public:
        void init(void)
        {
            ESP_LOGI(__func__, "Free heap (before a
                esp_get_free_heap_size());
```

```
        m_data_buffer = (char *)heap_caps_malloc
            BUFFERSIZE, MALLOC_CAP_SPIRAM | MALL
        m_compressed_buffer = (char *)heap_caps_
            BUFFERSIZE, MALLOC_CAP_SPIRAM | MALL
        m_decompressed_buffer = (char *)heap_cap
            BUFFERSIZE, MALLOC_CAP_SPIRAM | MALL
        ESP_LOGI(__func__, "Free heap (after all
            esp_get_free_heap_size());
    }
```

In the `init` function, we call the `heap_caps_malloc` function of the heap management library in ESP-IDF. Similar to `malloc`, it allocates memory and returns a pointer to it, but from PSRAM.

Please see the API documentation for the heap management strategies/capabilities of ESP-IDF here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/mem_alloc.html. The subject is important and it would be wise to read the official documentation thoroughly to learn more.

The initialization is completed, and we can continue with the `zip` function for data compression:

```
    char *zip(const char *data, size_t &len)
    {
        tdefl_init(&m_comp, NULL, NULL,
             TDEFL_WRITE_ZLIB_HEADER | 1500);
        memset(m_data_buffer, 0, BUFFERSIZE);
```

```
            memcpy(m_data_buffer, data, len);
            size_t inbytes = 0;
            size_t outbytes = 0;
            size_t inpos = 0;
            size_t outpos = 0;
```

The `zip` function takes two arguments, one is the pointer to the data to be compressed and the other one is its length, and it will return a pointer to the compressed data. The `len` parameter is a `size_t` reference and it will show the length of the compressed data upon returning from the function. The `tdefl_init` function, which comes in the Miniz library, initializes the `m_comp` member variable. Then we copy the data to the internal buffer, `m_data_buffer`, to process it in there. The local variables coming after the copy will be used during the compression. We compress the data in a `while` loop as follows:

```
            while (inbytes != len)
            {
                outbytes = BUFFERSIZE - outpos;
                inbytes = len - inpos;
                tdefl_compress(&m_comp, &m_data_buf
                      &inbytes, &m_compressed_buffe
                    &outbytes, TDEFL_FINISH);
                inpos += inbytes;
                outpos += outbytes;
            }
            len = outpos;
```

```
        return m_compressed_buffer;
    }
```

In the `while` loop, we process the input data. The `tdefl_compress` function does the job by reading from `m_data_buffer` and writing to `m_compressed_buffer`. It also updates the local variables to manage the process. When all data is compressed, we update the `len` variable with the compressed data length and return the buffer pointer to the caller. Next, we develop the `unzip` function to decompress data in a very similar manner:

```
    char *unzip(const char *data, size_t &len)
    {
        tinfl_init(&m_decomp);
        if (data != m_compressed_buffer)
        {
            memset(m_compressed_buffer, 0, BUFFI
            memcpy(m_compressed_buffer, data, le
        }
        size_t inbytes = 0;
        size_t outbytes = 0;
        size_t inpos = 0;
        size_t outpos = 0;
```

The `unzip` function takes the compressed data and its length as parameters. We initialize the decompressor by passing it to the Miniz `tinfl_init` function and we define the local variables. Then

we will extract the original data from the compressed binary as follows:

```
            while (inbytes != len)
            {
                outbytes = BUFFERSIZE - outpos;
                inbytes = len - inpos;
                tinfl_decompress(&m_decomp, (
                      const mz_uint8 *)&m_compressed
                    &inbytes, (uint8_t *)m_decompres
                    (mz_uint8 *)&m_decompressed_buff
                    &outbytes, TINFL_FLAG_PARSE_ZLIB
                inpos += inbytes;
                outpos += outbytes;
            }
            len = outpos;
            return m_decompressed_buffer;
        } // unzip end
    }; // class end
} // namespace end
```

The function to decompress data is `tinfl_decompress`. It takes the compressed data buffer, the output buffer, and other local variables to manage the process. When all the binary data is processed in the `while` loop, the `unzip` function returns a pointer to the decompressed data with its length updated. The class implementation finishes at this point and we can move on to the implementation of the `AppButton` class to trigger compress/decompress by pressing the buttons on the devkit. We

create a new file for it, `main/AppButton.hpp`, and write the class as follows:

```
#pragma once
#include "bsp_board.h"
#include "bsp_btn.h"
namespace app
{
    using btn_pressed_handler_f = void (*)(void *);
    class AppButton
    {
    public:
        void init(btn_pressed_handler_f l, btn_press
        {
            bsp_board_init();
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, l, nullptr);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, m, nullptr);
        }
    };
}
```

The entire `AppButton` implementation is very simple. We only write an initialization function, `init`, where we pass the button handlers as parameters. The left and middle buttons of the devkit are used in this example.

Finally, we can implement the application in the `main/miniz_ex.cpp` source code file:

```
#include <cstring>
#include "esp_log.h"
#include "AppButton.hpp"
#include "AppZip.hpp"
```

After including the header files, we define the application objects and variables in the anonymous namespace as follows:

```
namespace
{
    const char *m_test_str = "this is a repeating te
                              You can try anything\r
                             "this is a repeating te
                              You can try anything\r
                             "this is a repeating te
                              You can try anything\r
                             "this is a repeating te
                              You can try anything";

    app::AppButton m_btn;
    app::AppZip m_zip;
    size_t m_data_len;
    char *m_compressed_data;
    char *m_decompressed_data;
```

We will compress the `m_test_str` string when we press the left button and extract the original text by pressing the middle button. We define the left button handler next:

```
    void zipBtn(void *btn_ptr)
    {
        m_data_len = strlen(m_test_str);
        m_compressed_data = m_zip.zip(m_test_str, m_
        ESP_LOGI(__func__, "compressed to %u from %u
            m_data_len, strlen(m_test_str));
        ESP_LOG_BUFFER_HEX(__func__, m_compressed_da
            m_data_len);
    }
```

The `zipBtn` function calls the `zip` function of the `m_zip` object, which is an instance of the `app::AppZip` class. We set the `m_compressed_data` pointer to the returned value from the `zip` function call. The next function is the handler for the middle button:

```
    void unzipBtn(void *btn_ptr)
    {
        m_decompressed_data = m_zip.unzip(m_compress
            m_data_len);
        ESP_LOGI(__func__, "%.*s", m_data_len,
            m_decompressed_data);
    }
}
```

In the `unzipBtn` function, we call the `unzip` function of the `m_zip` object, hence the name. This function call extracts the original text and, if everything goes well, we should see the same `m_test_str`

text printed on the serial console when we press the middle button. The application is now ready for testing.

Testing the application

Let's flash the application and test it by pressing the left button and middle button sequentially. Here is the output of my test:

```
$ idf.py flash monitor
<logs removed>
I (15480) zipBtn: compressed to 71 from 255
I (15480) zipBtn: 78 01 d5 cb c1 0d 80 30 0c 04 c1 3
I (15481) zipBtn: 64 82 45 f2 c0 8e ec 43 c2 dd 93 3
I (15481) zipBtn: 3e 12 2b 41 e8 54 e1 b0 0b d4 97 a
I (15481) zipBtn: 67 68 a6 9e 3b ca 1f 34 31 30 0a 6
I (15481) zipBtn: ff f6 1f 5d b1 5c f7
I (17509) unzipBtn: this is a repeating text to be c
this is a repeating text to be compressed. You can t
this is a repeating text to be compressed. You can t
this is a repeating text to be compressed. You can t
```

The application seems to be working properly. When I press the left button, it compresses the text to 71 bytes, and the middle button decompresses to the same text.

The next library that we are going to talk about is *FlatBuffers*, which is very useful for passing data between different platforms regardless of their architectures.

FlatBuffers

FlatBuffers is a cross-platform serialization library from Google. The library supports many different programming languages, so it is possible to use it on any platform. Another interesting feature of *FlatBuffers* is that it can directly map binary data to its representation on the platform it runs without parsing or using extra buffers. As a result, the library is quite efficient in terms of both memory usage and processing power.

The library has great documentation here: https://google.github.io/flatbuffers/

In the FlatBuffers example, we will collect analog data from an LDR and convert the data to binary format (serialization) by using the FlatBuffers library. Moreover, we will again employ the library to revert the binary data to the programming structures (deserialization).

The hardware components of the project are:

- ESP32-S3 Box Lite
- An LDR
- A pull-up resistor (10KΩ)

The following is a Fritzing sketch that shows the connections:

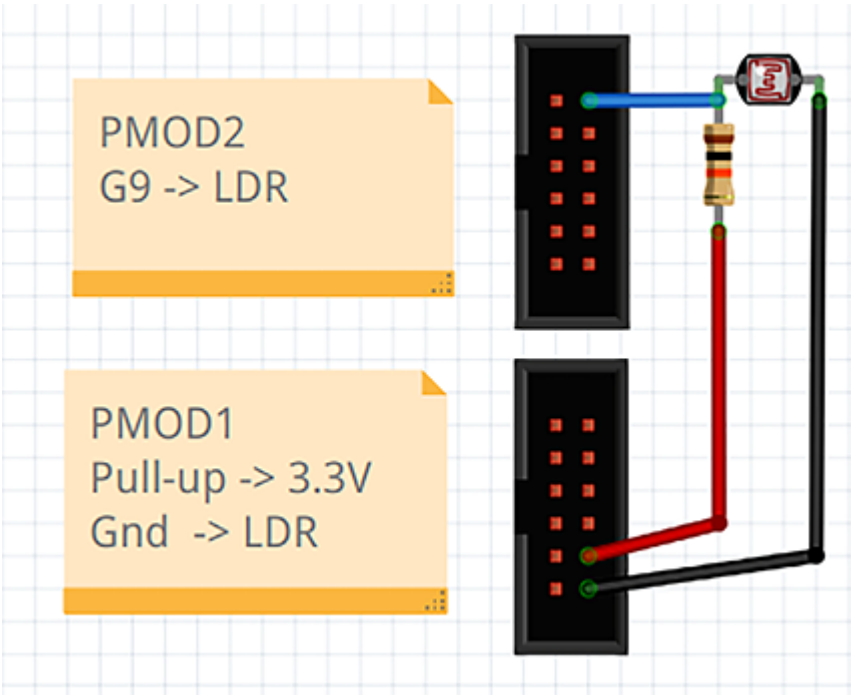


Figure 4.1: Fritzing sketch of the project

After having this hardware setup prepared, we can create an ESP-IDF project next.

Creating a project

We can follow the steps below to create and configure a project:

1. Create an ESP-IDF project in any way you prefer.
2. Copy the `sdkconfig.defaults` file from the project directory (`ch4/flatbuffers_ex`) on GitHub into the local project directory.
3. We will use the buttons on the devkit. The BSP has the driver for them as an IDF component and we can include it by setting `EXTRA_COMPONENT_DIRS` in the root `CMakeLists.txt` file. Set the content of this file as the following:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
add_compile_options(-fdiagnostics-color=always -
project(flatbuffers_ex)
```

4. Clone the FlatBuffers repository into the `components` directory:

```
$ mkdir components && cd components
$ git clone https://github.com/google/flatbuffer
$ rm -rf .git/
```

5. We need to register the FlatBuffers library as an ESP-IDF component in the project. To do that, replace the content of the `components/flatbuffers/CMakeLists.txt` file with the following:

```
cmake_minimum_required(VERSION 3.10)
set(FlatBuffers_Library_SRCS
  src/idl_parser.cpp
  src/idl_gen_text.cpp
  src/reflection.cpp
  src/util.cpp
)
file(GLOB SOURCES ${FlatBuffers_Library_SRCS})
idf_component_register(SRCS ${SOURCES} INCLUDE_D
```

6. We will need the FlatBuffers compiler, `flatc,` to convert a FlatBuffers schema definition into the target programming language. Download it from GitHub for your development platform:

```
$ mkdir tmp && cd tmp
$ wget https://github.com/google/flatbuffers/rel
$ unzip Linux.flatc.binary.g++-10.zip
$ ./flatc -version
flatc version 22.10.26
```

7. Rename the source code file to `main/flatbuffers_ex.cpp` and set the content of the `main/CMakeLists.txt` file to reflect this change:

```
idf_component_register(SRCS "flatbuffers_ex.cpp"
```

The project is now ready for development.

Coding the application

Let's start VS Code and add the `FlatBuffers` schema that we are going to use to define our data types. The name of the schema file is `app_data.fbs` in the project root:

```
namespace app;
table ReadingFb {
    timestamp:uint;
```

```
    light:ushort;
}
```

FlatBuffers uses its own syntax, **Interface Definition Language** (**IDL**), for this purpose. The first line shows the namespace of all definitions that the FlatBuffers compiler, `flatc,` will generate. Then we define a table for LDR readings. The `ReadingFb` table has two fields, `timestamp` and `light`. When this is converted, it will be a C++ struct.

This document explains the syntax of IDL: https://google.github.io/flatbuffers/flatbuffers_guide_writing_schema.html

The `ReadingFb` table describes a reading from the LDR, but we also want to define a light sensor in the schema as follows:

```
table LightSensorFb {
    location:string;
    readings:[ReadingFb];
}
root_type LightSensorFb;
```

The second table in the schema is `LightSensorFb`, which describes a light sensor. A light sensor has a `location` and an array of `readings`. In the last line, we set the root type as `LightSensorFb` for the code generation. Any FlatBuffers binary for this schema will have light sensor data at its root. Next, we will run the `tmp/flatc`

compiler to generate the C++ header file that corresponds to this schema:

```
$ tmp/flatc  -b -t -c --gen-object-api app_data.fbs
$ mv app_data_generated.h main/
```

The output of the `flatc` compiler is `app_data_generated.h` and we move it to the `main` directory where the source code of our project resides.

The `--gen-object-api` flag instructs the compiler to generate an object-based API. It is not needed in most projects, but for the sake of simplicity, we will use this API to easily access the sensor data in this example.

Then, we develop the LDR logger class in the `main/AppLdrLogger.hpp` file:

```
#pragma once
#include <vector>
#include <cinttypes>
#include <memory>
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/adc.h"
#include "flatbuffers/idl.h"
#include "flatbuffers/util.h"
#include "app_data_generated.h"
```

As mentioned in the introduction of the example, we will read light values from an LDR. The `driver/adc.h` header file provides us with the **Analog-to-Digital Conversion** (**ADC**) functionality so that we can convert the analog output of the LDR to digital values that we can process in the application. We also include the `app_data_generated.h` header file to be able to serialize the collected light data. We continue with the class definition next:

```
namespace app
{
    class AppLdrLogger
    {
    private:
        LightSensorFbT m_light_sensor;
        const adc1_channel_t m_adc_ch = ADC1_CHANNEl
```

In the `private` section of the `AppLdrLogger` class, we define the member variables. The first one is an instance of `LightSensorFbT`, a C++ struct that is a part of the generated object API. If you look back at the schema, its fields are `location` and `readings`. The ADC channel is `ADC1_CHANNEL_8` and it is associated with the PMOD2/G9 pin of the devkit.

We can move on to the `public` section of the class and add the `init` function:

```
    public:
        void init(void)
        {
```

```
            adc1_config_width(ADC_WIDTH_BIT_12);
            adc1_config_channel_atten(m_adc_ch, ADC_
            m_light_sensor.location = "office";
        }
```

In the `init` member function, we first initialize the ADC peripheral. The resolution is 12 bits, which means we can read a value in the range of 0 to 4,095 from the LDR. After specifying the ADC channel attenuation, we set the sensor `location` as `office`. The next member function reads from the LDR:

```
        void run(void)
        {
            while (1)
            {
                vTaskDelay(pdMS_TO_TICKS(5000));
```

The `run` function will be a FreeRTOS task. The purpose is to make periodic readings from the sensor with an interval of 5 seconds. After 5 seconds pass, we read from the ADC channel as follows:

```
                uint32_t adc_val = 0;
                for (int i = 0; i < 32; ++i)
                {
                    adc_val += adc1_get_raw(m_adc_ch
```

```
                }
                adc_val /= 32;
```

We get 32 consecutive readings from the ADC channel and take the average as the final value (oversampling). Then we will record this value:

```
                auto reading = std::unique_ptr<
                    ReadingFbT>(new ReadingFbT());
                reading->timestamp = esp_log_timesta
                reading->light = (uint16_t)adc_val;
                m_light_sensor.readings.push_back(
                    std::move(reading));
            }
        }
```

Each record has `timestamp` and `light` values. We append the record to the `m_light_sensor.readings` vector. Please note that we haven't defined these types manually anywhere in the C++ source code; they are automatically generated and part of the FlatBuffers object API in this particular project. The last member function serializes the sensor and so the collected data associated with it:

```
        size_t serialize(uint8_t *buffer)
        {
            flatbuffers::FlatBufferBuilder fbb;
```

```
            fbb.Finish(LightSensorFb::Pack(fbb, &m_l
            memcpy(buffer, fbb.GetBufferPointer(), f
            size_t len = fbb.GetSize();
            m_light_sensor.readings.clear();
            return len;
        } // serialize end
    }; // class end
} // namespace end
```

The `serialize` function takes a pointer to where the binary output of the serialization will be copied. In the function, we define a `flatbuffers::FlatBufferBuilder` object, which converts the `m_light_sensor` object to the binary representation when its `Finish` function is called in the next line. After copying the binary data to the given buffer, we clear all the readings from the `m_light_sensor` object until a new call to the `serialize` function of the `AppLdrLogger` class.

The `logger` class is finished and, next, we will develop a `client` class that deserializes the binary data back into a `LightSensorFbT` object and uses it for its own purposes. Let's add a new file, `main/AppLdrClient.hpp`, for it:

```
#pragma once
#include <cinttypes>
#include "esp_log.h"
#include "flatbuffers/idl.h"
```

```
#include "flatbuffers/util.h"
#include "app_data_generated.h"
```

Again, we include the same generated model for the `client` class. The definition of the class is as follows:

```
namespace app
{
    class AppLdrClient
    {
    private:
        LightSensorFbT m_light_sensor;
    public:
        void consume(const uint8_t *buffer)
        {
            app::GetLightSensorFb(
                buffer)->UnPackTo(&m_light_sensor);
            ESP_LOGI(__func__, "location: %s",
                m_light_sensor.location.c_str());
            for (auto &&rec : m_light_sensor.reading
            {
                ESP_LOGI(__func__, "ts: %u, light: %
                    rec->timestamp, rec->light);
            }
        }
    };
}
```

The implementation of the `AppLdrClient` class is quite simple. We define a member variable, `m_light_sensor`, into which we are going to deserialize binary data. The `consume` function does this deserialization operation. When it is called, the `consume` function receives the binary buffer and reads it into the `m_light_sensor` object by calling the `UnPackTo` function of the object that is returned by the `GetLightSensorFb` function call. The implementation of these functions comes in the header file generated by the `flatc` compiler. After the deserialization, we can use the `m_light_sensor` object in any way we need.

We will make use of the devkit buttons to prove our FlatBuffers serialization/deserialization implementation works as intended. Pressing the left button will call the logger's `serialize` function and the middle button will call the client's `consume` function. We can develop this button class in the `main/AppButton.hpp` file as follows:

```
#pragma once
#include "bsp_board.h"
#include "bsp_btn.h"
namespace app
{
    using btn_pressed_handler_f = void (*)(void *);
    class AppButton
    {
    public:
        void init(btn_pressed_handler_f l,
            btn_pressed_handler_f m)
        {
```

```
            bsp_board_init();
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, l, nullptr);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, m, nullptr);
        }
    };
}
```

Again, it is a very brief class to implement. The `init` function of the `AppButton` class takes two function parameters as handlers to be registered for the left and middle buttons.

We have all the necessary classes implemented. The only coding work left is to integrate them in the `main/flatbuffers_ex.cpp` file:

```
#include <cinttypes>
#include "esp_log.h"
#include "esp_heap_caps.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "AppButton.hpp"
#include "AppLdrLogger.hpp"
#include "AppLdrClient.hpp"
```

We include our class implementations and then continue with the anonymous namespace of the application as follows:

```
namespace
{
    constexpr const char *TAG{"app"};
    constexpr const size_t BUFFERSIZE{16u * 1024};
    uint8_t *m_buffer;
```

In the anonymous namespace, we define a buffer pointer. We will allocate memory for the buffer from the PSRAM when the application starts. We define the objects of the application next:

```
    app::AppButton m_btn;
    app::AppLdrLogger m_logger;
    app::AppLdrClient m_client;
```

The objects are the instances of the classes that we have implemented earlier. We will close the anonymous namespace with the `loggerTask` function:

```
    void loggerTask(void *param)
    {
        m_logger.run();
    }
} // namespace end
```

The `loggerTask` function is actually a wrapper for the `run` function of the `m_logger` object and will be a FreeRTOS task in the application. If you remember, the `run` function will collect data from

the LDR every 5 seconds. Let's continue with the `app_main` function:

```
extern "C" void app_main(void)
{
    m_buffer = reinterpret_cast<
        uint8_t *>(heap_caps_malloc(
        BUFFERSIZE, MALLOC_CAP_SPIRAM));
```

As promised, we allocate memory for `m_buffer` from the PSRAM by calling the `heap_caps_malloc` function. It will be the buffer where we will store the serialized data. A lambda function comes next to do this:

```
    auto serialize = [](void *)
    {
        ESP_LOGI(TAG, "serializing..");
        size_t len = m_logger.serialize(m_buffer);
        ESP_LOG_BUFFER_HEX(TAG, m_buffer, len);
    };
```

The `serialize` lambda function calls the `serialize` function of `m_logger` with `m_buffer` as the parameter. Then we print the content in hexadecimal format on the serial console. We also need to show that the deserialization works. We use another lambda function for it:

```
    auto deserialize = [](void *)
    {
```

```
        ESP_LOGI(TAG, "deserializing..");
        m_client.consume(m_buffer);
    };
```

This time, we call the `consume` function of `m_client` with `m_buffer` as its parameter. It will convert the binary data in the buffer to a sensor object and print the object content on the serial console. These two lambda functions are the press handlers of the button object to be initialized, as we do next:

```
    m_btn.init(serialize, deserialize);
    m_logger.init();
    xTaskCreate(loggerTask, "logger", 3072, nullptr
}
```

We initialize the `m_btn` object and the `m_logger` object and start a FreeRTOS task with the `loggerTask` function for data collection before we finish up the `app_main` function.

The application is now ready for testing, and we can flash it on the devkit to see how it works.

Testing the application

Let's flash and monitor the application as follows:

```
$ idf.py flash monitor
<logs removed>
```

```
I (29453) app: serializing..
I (29454) app: 0c 00 00 00 08 00 0c 00 04 00 08 00 0
I (29454) app: 64 00 00 00 04 00 00 00 05 00 00 00 4
I (29454) app: 34 00 00 00 24 00 00 00 14 00 00 00 0
I (29454) app: d0 ff ff ff 00 00 15 0e e0 63 00 00 0
I (29455) app: 00 00 61 01 58 50 00 00 e8 ff ff ff 0
I (29455) app: d0 3c 00 00 f4 ff ff ff 00 00 97 0c 4
I (29455) app: 08 00 0c 00 08 00 06 00 08 00 00 00 0
I (29455) app: c0 15 00 00 06 00 00 00 6f 66 66 69 0
I (31398) app: deserializing..
I (31398) consume: location: office
I (31398) consume: ts: 5568, light: 3108
I (31398) consume: ts: 10568, light: 3223
I (31399) consume: ts: 15568, light: 3187
I (31399) consume: ts: 20568, light: 353
I (31399) consume: ts: 25568, light: 3605
```

After waiting half a minute for the light data to be collected, when we press the left button of the devkit, the FlatBuffers serialization occurs. The binary data is displayed on the console as hex numbers. Then, when we press the middle button, the actual sensor content is printed this time, which corresponds to the deserialization. You can play with it to see how the readings change when the LDR is exposed to the different levels of light.

FlatBuffers can be a very efficient solution when it is used properly. It is a good way of exchanging data between different platforms.

In the next topic, we will discuss another great library, *LVGL*.

LVGL

Light and Versatile Graphics Library, or **LVGL** for short, is one of the most popular graphics libraries for embedded systems. There are many factors that I can count here as the reasons for its popularity:

- First and foremost, it is really lightweight compared to the functionality and widgets that come with it.
- It has fully configurable, modern widgets.
- It has a simple API with plain C structures and callbacks.
- There is extensive documentation available with examples.
- It has great support from the LVGL team.
- It is free and open source with an MIT license (the GitHub repository is at `https://github.com/lvgl/lvgl`).

We have already developed an example with LVGL in the previous chapter while talking about displays and GUI development. In this example, we will have more chances to discuss LVGL and its capabilities in detail. We will develop different screens that we can navigate by using the devkit buttons and also interact with the widgets that we add on those screens. Let's start.

Designing the GUI

One feature that I didn't mention in the introduction of LVGL is that it has a GUI designer, named **SquareLine Studio**. Although we can design and develop everything by hand, the GUI designer has some obvious benefits, such as fast GUI development, easy visualizing, and in-place testing. The tool requires a license, but it is free for

personal use; therefore, you can use SquareLine freely while following the examples of the book.

You can download SquareLine Studio from this URL: https://squareline.io/downloads. The documentation explains the usage well but there are also many videos for learning on YouTube: https://www.youtube.com/@squarelinestudio/videos.

We won't delve into GUI design with SquareLine Studio; nonetheless, I will share screenshots from the designer to be able to explain the examples in this book better. The following figure shows the screens of this project:

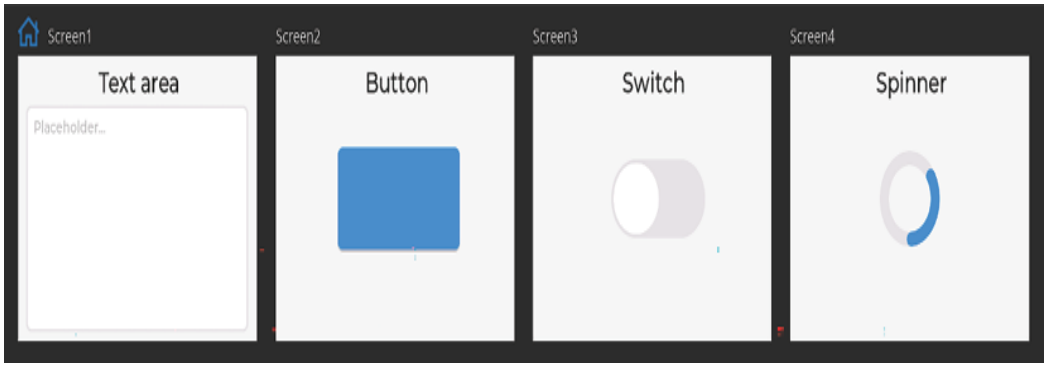


Figure 4.2: GUI screens with SquareLine

In this application, we will have four screens with different widgets on each of them. The buttons on the devkit will provide navigation: the left button to go left and the right button for the next screen on the right. The middle button will be for changing the state or property of the widget on the current screen. For example, when we are on **Screen1** with a **Text area**, pressing the middle button will update the text inside this text area. After having the screens designed with SquareLine, we can create an ESP-IDF project.

Creating a project

Let's prepare the project for development in the following steps:

1. Create an ESP-IDF project in any way you like:

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project lvgl_ex
```

2. Copy the `sdkconfig` file from the book repository into the project root.
3. We are going to need the board support package to drive the devkit's buttons. We set `EXTRA_COMPONENT_DIRS` to the BSP path in the project root `CMakeLists.txt` for this. The content of the file should be as follows:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
add_compile_options(-fdiagnostics-color=always -
project(lvgl_ex)
```

4. We need the code for the GUI that we discussed above. You have two options here. You can either use SquareLine to generate the C code by opening the SquareLine project that I have provided in the GitHub repository under the `design` directory and exporting for `ESP-BOX`, or copy the generated C code directly from the repository in the `main` directory. There are five files:

```
$ ls main/ui*
main/ui.c  main/ui_events.c  main/ui.h  main/ui_
```

5. Rename the `main/lvgl_ex.c` source file to `main/lvgl_ex.cpp` and update `main/CMakeLists.txt` to reflect this change. We also need to add the GUI files as well. The content of the `main/CMakeLists.txt` should be:

```
idf_component_register(SRCS "lvgl_ex.cpp" "ui_he
```

The project is ready for further development, but if you choose to use SquareLine to generate the GUI C code, then you need to follow these steps:

1. After starting SquareLine, import the project by selecting **File/Open** from the main menu and then clicking on the **IMPORT PROJECT** button of the dialog that is just displayed. You should be able to see the project icon named `lvgl_ex.spj` on the same dialog if the import succeeds. Double-click on it to open the GUI design.
2. From the main menu, select **Export/Export UI Files** and choose the `main` directory in the project root so that the GUI sources are placed in the same directory with the default project source code.

We have the project ready for development and we can now move on to coding.

Coding the application

Let's begin with the application button in `main/AppButton.hpp`:

```
#pragma once
#include "bsp_board.h"
#include "bsp_btn.h"
namespace
{
    template <board_btn_id_t I, button_event_t E>
    void button_event_handler(void *param);
}
```

The button event handler is a template function with a button ID and event type as template parameters. We will implement the function body at the end of this file. Next, we continue with some supporting definitions:

```
namespace app
{
    struct sAppButtonEvent
    {
        board_btn_id_t btn_id;
        button_event_t evt_id;
    };
    using fAppButtonCallback = void (*)(sAppButtonEv
```

We will use the `sAppButtonEvent` structure to pack the button press information into a single parameter when passing it to a

callback function of type `fAppButtonCallback`. Next comes the class definition:

```
class AppButton
{
private:
    fAppButtonCallback m_btn_cb;
```

In the `private` section of the `AppButton` class, we keep a member variable as the callback when a button is pressed. It will be the connection point of the class to any client code. The `public` section of the class is as follows:

```
public:
    void init(fAppButtonCallback cb)
    {
        m_btn_cb = cb;
        bsp_btn_register_callback(BOARD_BTN_ID_P
            BUTTON_PRESS_DOWN, button_event_hand
            BOARD_BTN_ID_PREV,BUTTON_PRESS_DOWN);
        bsp_btn_register_callback(BOARD_BTN_ID_N
            BUTTON_PRESS_DOWN, button_event_hand
            BOARD_BTN_ID_NEXT, BUTTON_PRESS_DOWN
        bsp_btn_register_callback(BOARD_BTN_ID_E
            BUTTON_PRESS_DOWN, button_event_hand
            BOARD_BTN_ID_ENTER, BUTTON_PRESS_DOW
        bsp_btn_register_callback(BOARD_BTN_ID_E
            BUTTON_PRESS_UP, button_event_handle
```

```
                    BOARD_BTN_ID_ENTER, BUTTON_PRESS_UP:
        }
```

The `init` function takes a callback function as a parameter and sets the `m_btn_cb` member variable to it. In the `init` function body, we register four callbacks for all three buttons of the devkit: only for press-down events of the left and right buttons and both press-down and release events for the middle button. In the `public` section, we have two more functions:

```
        static AppButton &getObject(void *btn_ptr)
        {
            button_dev_t *btn_dev = reinterpret_cast
                button_dev_t *>(btn_ptr);
            return *(reinterpret_cast<app::AppButtor
                btn_dev->cb_user_data));
        }
        void runCallback(sAppButtonEvent &e)
        {
            m_btn_cb(e);
        } // function end
    }; // class end
} // namespace end
```

The `getObject` function is a static function that returns a reference to the `AppButton` object that registers the button handler. The `btn_ptr` parameter of the function points to that button. The

`runCallback` member function simply calls the `m_btn_cb` callback function with the button event. The `AppButton` class is finished. Next, we implement the `button_event_handler` function body:

```
namespace
{
    template <board_btn_id_t I, button_event_t E>
    void button_event_handler(void *btn_ptr)
    {
        app::AppButton &app_btn = app::AppButton::ge
        app::sAppButtonEvent e{I, E};
        app_btn.runCallback(e);
    } // function end
} // anonymous namespace end
```

In the `button_event_handler` function, we find the `AppButton` object from the `btn_ptr` pointer and run its callback with the event. Since the `sAppButtonEvent` event structure contains both the button ID and the event type, any client code will know what happened to a button when it initializes the `AppButton` object with a callback function.

We have completed the `AppButton` development and can move on to the GUI management by adding a new file, `main/AppUi.hpp`:

```
#pragma once
#include <mutex>
#include <vector>
#include "bsp_lcd.h"
```

```
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
#include "ui.h"
#include "AppButton.hpp"
```

There are three header files related to LVGL. The `lvgl/lvgl.h` file encloses the core library functionality, the `lv_port/lv_port.h` header is for linking the devkit hardware and LVGL, and the last one, `ui.h`, has the definitions of the application – i.e., the real GUI. We can begin with the class implementation next:

```
namespace app
{
    class AppUi
    {
    private:
        static std::mutex m_ui_access;
        static void lvglTask(void *param)
        {
            while (true)
            {
                {
                    std::lock_guard<std::mutex> lock
                    lv_task_handler();
                }
                vTaskDelay(pdMS_TO_TICKS(10));
```

```
            }
        }
```

In the `private` section, we define a `mutex` that controls access to the LVGL memory objects. The `lvglTask` function will be a FreeRTOS task that runs periodically and renders all the changes on the active screen. In a `while` loop in this function, we first acquire access by creating a lock on the `mutex`; thus, no other task can change anything while the `lv_task_handler` function is running. Then, we define the remaining `private` members:

```
        std::vector<lv_obj_t *> m_screens;
        int m_scr_pos;
```

We keep a vector of pointers to the LVGL screens that we have designed and the index of the active screen, `m_scr_pos`, on this vector. The left and right buttons will change this index and command LVGL to set the active screen. Let's move on to the `public` section of the class:

```
    public:
        void init(void)
        {
            lv_port_init();
            ui_init();
            m_scr_pos = 0;
            m_screens.push_back(ui_Screen1);
```

```
        m_screens.push_back(ui_Screen2);
        m_screens.push_back(ui_Screen3);
        m_screens.push_back(ui_Screen4);
        xTaskCreatePinnedToCore(lvglTask, "lvgl'
            nullptr, 3, nullptr, 0);
        bsp_lcd_set_backlight(true);
    }
```

The `init` function does several important jobs to start the GUI. First, it calls the `lv_port_init` function to bind the hardware and LVGL. Then, it runs the `ui_init` function to create all LVGL screens. This function is a part of the code generated by SquareLine. Next, we collect all the screen pointers in the `m_screens` vector. In this way, we complete the LVGL initialization and create a FreeRTOS task for periodic screen updates. At the end of the `init` function, we turn the backlight of the LCD on to activate it. The next function that we are going to implement is the button event handler:

```
    void buttonEventHandler(sAppButtonEvent &btr
    {
        std::lock_guard<std::mutex> lock(m_ui_ac
```

The `buttonEventHandler` function takes an argument of type `sAppButtonEvent`. The first thing we do in the `buttonEventHandler` function is to acquire access to the LVGL memory objects by creating a lock on the `m_ui_access` member

variable. The function argument, `btn_evt`, carries the button event information, and we will check it to respond accordingly:

```
            switch (btn_evt.btn_id)
            {
            case board_btn_id_t::BOARD_BTN_ID_PREV:
                m_scr_pos = (m_scr_pos - 1) % m_scre
                lv_scr_load(m_screens[m_scr_pos]);
                break;
            case board_btn_id_t::BOARD_BTN_ID_NEXT:
                m_scr_pos = (m_scr_pos + 1) % m_scre
                lv_scr_load(m_screens[m_scr_pos]);
                break;
```

In a `switch` statement, we check which button has generated the event. If it is the left button (`BOARD_BTN_ID_PREV`), we set the `m_scr_pos` index one less to point to the previous screen and call the `lv_scr_load` function of the LVGL library to render the newly pointed screen. Remember that it will actually be drawn in the next cycle of the LVGL FreeRTOS task. We do the same thing for the right button but in the opposite direction.

It is possible to configure LVGL to handle the buttons as an input device, but I prefer to handle the button events manually to show the flow a bit more easily. You can refer to the online API document about input devices here:
https://docs.lvgl.io/master/overview/indev.html

We also need to respond to the middle button press and release events:

```
            case board_btn_id_t::BOARD_BTN_ID_ENTER
            {
                switch (m_scr_pos)
                {
                case 0:
                    updateTextArea(btn_evt.evt_id);
                    break;
                case 1:
                    updateLvButtonState(btn_evt.evt_
                    break;
                case 2:
                    toggleSwitch(btn_evt.evt_id);
                    break;
                case 3:
                    toggleSpinnerVisibility(btn_evt
                    break;
                default:
                    break;
                }
            }
            break;
            default:
                break;
            } // switch end
        } // function end
```

The handling of the middle button event depends on the active screen. In another `switch` statement, we check which screen is active and take the appropriate action. For instance, if it is **Screen1**, then we update the text area on it by calling the `updateTextArea` function. Similarly, we call other functions for the widgets that we have on the screens. Let's implement these functions one by one:

```
void updateTextArea(button_event_t btn_evt_
{
    if (btn_evt_id == button_event_t::BUTTO
    {
        lv_textarea_add_text(ui_Screen1_Text
            "button down\n");
    }
    else
    {
        lv_textarea_add_text(ui_Screen1_Text
            "button up\n");
    }
}
```

On **Screen1**, we have a text area. We append text to its end by calling the `lv_textarea_add_text` function of the LVGL library, showing the button event. The next function is for **Screen2**:

```
void updateLvButtonState(button_event_t btn_
{
    if (btn_evt_id == button_event_t::BUTTO
    {
```

```
                lv_event_send(ui_Screen2_Button1,
                    LV_EVENT_PRESSED, nullptr);
            }
            else
            {
                lv_event_send(ui_Screen2_Button1,
                    LV_EVENT_RELEASED, nullptr);
            }
        }
```

The widget on **Screen2** is an LVGL button. When we press the middle button of the devkit, we call the `lv_event_send` function with the `LV_EVENT_PRESSED` event so that the GUI button on the screen can also update itself with this state change. The release of the middle button is updated on the GUI as well. We continue with **Screen3**:

```
        void toggleSwitch(button_event_t btn_evt_id)
        {
            static bool checked{false};
            if (btn_evt_id == button_event_t::BUTTON
            {
                checked = !checked;
                if (checked)
                {
                    lv_obj_add_state(ui_Screen3_Swit
                        LV_STATE_CHECKED);
                }
                else
```

```
                {
                    lv_obj_clear_state(ui_Screen3_Sw
                        LV_STATE_CHECKED);
                }
            }
        }
```

There is a switch widget on **Screen3**. Pressing the middle button will toggle its checked state. We set and clear its state by calling the `lv_obj_add_state` and `lv_obj_clear_state` functions, respectively. The last function is for **Screen4**:

```
        void toggleSpinnerVisibility(button_event_t
        {
            static bool hidden{false};
            if (btn_evt_id == button_event_t::BUTTON
            {
                hidden = !hidden;
                if (hidden)
                {
                    lv_obj_add_flag(ui_Screen4_Spinn
                        LV_OBJ_FLAG_HIDDEN);
                }
                else
                {
                    lv_obj_clear_flag(ui_Screen4_Spi
                        LV_OBJ_FLAG_HIDDEN);
                }
            }
```

```
        } // function end
    }; // class end
    std::mutex AppUi::m_ui_access;
} // namespace end
```

We have a spinner on `Screen4`. This time, the effect of pressing the middle button toggles the widget visibility(press->invisible). The visibility of objects in LVGL is maintained as a flag and, regardless of the widget type, we can change the visibility by calling the `lv_obj_add_flag` and `lv_obj_clear_flag` functions with the `LV_OBJ_FLAG_HIDDEN` parameter on the target object. We apply these functions on the spinner here.

The implementation of the `AppUi` class is finished and we can now integrate the pieces into `main/lvgl_ex.cpp` to complete the application:

```
#include "bsp_board.h"
#include "AppUi.hpp"
#include "AppButton.hpp"
namespace
{
    app::AppUi m_app_ui;
    app::AppButton m_app_btn;
}
```

After including the class headers, we define the `AppUi` and `AppButton` objects in the anonymous namespace. Then the

`app_main` function comes next as the entry point of the application:

```
extern "C" void app_main(void)
{
    bsp_board_init();
    auto btn_evt_handler = [](app::sAppButtonEvent &
    {
        m_app_ui.buttonEventHandler(e);
    };
    m_app_ui.init();
    m_app_btn.init(btn_evt_handler);
}
```

The critical point here is the `btn_evt_handler` lambda function. It connects the `m_app_ui` and `m_app_btn` objects. We initialize the `button` object with this lambda function as the button handler and, inside it, we pass the event information to the UI object, `m_app_ui`. As a result, when we press any button on the devkit, this event will propagate to the UI and the LCD of the devkit will be updated accordingly. The application is ready for testing now.

Testing the application

There is no console output worth noting here, so it is best to flash the app and test it by playing with the devkit buttons:

```
$ idf.py flash
```

After flashing the application, we can press the middle button and observe the text area showing the button-pressed events. For the next screen, we use the right button of the devkit. Each screen has a widget on it, and we can change its state by pressing the middle button.

This example is only an introduction to the LVGL library and there are many other widgets and features that you might want to try. Please visit the online API documentation and see what else LVGL provides. The next library that we will discuss is the **ESP-IDF Components library**.

ESP-IDF Components library

It is impossible not to mention the **ESP-IDF Components library** by UncleRus in this chapter. It is one of the most famous libraries in the community. We have already used a port of this library in *Chapter 3*, *Using ESP32 Peripherals*, where we discussed I^2C communication and integrated different sensor breakout boards. The library especially focuses on I^2C devices and maintains many drivers for sensors from different vendors in a single repository. The driver licenses are all categorized under the FOSS classification but please check for the specific license type before including a sensor driver in your project.

The GitHub repository of the ESP-IDF Components library is here: https://github.com/UncleRus/esp-idf-lib

The library officially supports ESP32, ESP32-S2, ESP32-C3, and ESP8266. However, as of the time of writing this book, it failed to run I^2C devices on ESP32-S3 because of some kind of timing issue.

The book repository contains an updated version of the ESP-IDF Components library to make it compatible with ESP32-S3 series chips.

We won't have an example of this library here since we have already made use of it in the previous chapter. If you need a driver from this library, its repository contains sample applications for each of them.

Espressif frameworks and libraries

Besides all these third-party libraries, Espressif Systems empowers developers with many other frameworks and libraries. As a quick overview, here is a short list of those frameworks from Espressif:

- **ESP-IoT-Solution**: This framework brings different hardware drivers together as a working solution to minimize compatibility issues with ESP-IDF. It contains sensor drivers, display controller drivers and LVGL, input devices and buttons, audio output utilities, and more functionality for hassle-free development (https://github.com/espressif/esp-iot-solution).
- **Audio Development Framework** (**ESP-ADF**): This is the core framework for audio input/output and processing. It collects all the necessary components under the same roof to develop audio applications, such as music players/recorders, speech recognition applications, smart speakers, etc. (https://github.com/espressif/esp-adf).
- **Image Processing Framework** (**ESP-WHO**): This provides image processing capabilities, such as face

detection/recognition, and motion detection. You can also use ESP-WHO to develop barcode/QR code scanners. The **Deep Learning** (**ESP-DL**) library powers this framework under the hood (https://github.com/espressif/esp-who).

- **ESP RainMaker**: This is probably one of the most interesting frameworks by Espressif Systems. Actually, it is better to call it a platform. ESP RainMaker is composed of many different system components in order to enable developers to develop cloud-based **Artificial Intelligence of Things** (**AIoT**) products. These components are the device-agent SDK, the RainMaker cloud, and the mobile phone apps (https://rainmaker.espressif.com/).

For the other frameworks and libraries by Espressif, you can check the following URLs:

- *ESP-IDF Programming Guide*: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/libraries-and-frameworks/libs-frameworks.html
- IDF Component Registry: https://components.espressif.com/
- And, of course, the Espressif GitHub repository: https://github.com/espressif

The IoT landscape is extremely dynamic and versatile; therefore, it is practically impossible to provide an exhaustive list when it comes to sharing resources. My personal approach to learning is to select a sample project that I'm interested in and develop it further with more ideas. The research during the development leads to many

other resources and ideas with more learning opportunities. Similarly, you can choose one of the examples in this book as a starting point and find alternative solutions by adding other third-party libraries to your project.

When it comes to project development and how to choose the right set of libraries, each real-world project has its own unique requirements and constraints; therefore, library selection depends on such restrictions. One thing to remember is that we are working on constrained devices. Although ESP32 has a good hardware specification, it is still a constrained device with a limited memory capacity, and we need to keep this in mind while selecting alternatives. For example, one advantage of using nlohmann/json as a JSON parser is that its implementation respects the modern features of C++, such as custom memory allocation for its STL-like JSON containers. In one of my projects, we could easily switch to the ESP32 external memory by defining a simple memory allocator class when we were at the internal RAM limits. For graphics, the challenge is a bit different. In addition to the memory restrictions, the abstraction level of the graphics library and display driver integration become important factors for decision. In my ESP32 projects, I usually end up with LVGL after comparing the project requirements and the graphics library options on the market. If a project requires a significant amount of data to be delivered to a cloud infrastructure, compressing data before transferring would be a good idea since it could save cloud service costs. In another ESP32 project of mine, we used Miniz to compress data to one-tenth of its original size before sending it to AWS IoT over MQTT, saving a good amount of money in operational costs. The idea here is that we can

almost certainly find a library for a common problem. It is our responsibility to do proper research, evaluate the alternatives, and then import the right one into the project before jumping into coding.

Summary

In this chapter, we covered a selection of third-party libraries that we can use in our ESP32 projects. We learned that we can use **LittleFS** as an alternative to SPIFFS, **nlohmann/json** as a modern JSON library, **Miniz** for data compression, and **FlatBuffers** to share data between different platforms and architectures. **LVGL** has a special place among them, such that it is the framework to use if you want to create an amazing graphical interface for the users of your product. The **ESP-IDF Components library** is another popular library for ESP32 developers and it provides many device drivers that we can employ in our projects. Finally, we talked about some important frameworks by Espressif.

They provide a head-start and make life much easier when commencing a new ESP32 project. The examples in this chapter also showed us different methods of importing third-party libraries in our projects so that we can apply them when we need other libraries in the next project.

The next chapter is devoted to a complete project where we will implement an audio player with a GUI. The project will show us how to combine pieces into one application by applying good design and development practices. ESP-ADF and LVGL will be the primary components of the application.

Questions

You can answer the following questions as a review of the topics in this chapter:

1. Which of the following would be a good use case for FlatBuffers?
    a. Serializing data on an external flash
    b. Sending data to a mobile application
    c. Querying an I2C sensor
    d. Formatting log data into JSON
2. There is a good amount of data to be transferred over WiFi with repeating information. Which method would help most to reduce the data size?
    a. Formatting into JSON by using the nlohmann/json library
    b. Binary serialization with FlatBuffers
    c. Using Miniz to compress data
    d. Ignoring repeated information
3. Which of the following statements is NOT correct about the use of LVGL in a project?
    a. It is a GUI library so the device should have a display.
    b. LVGL provides an API for input devices, such as keypads or touchscreens.
    c. There is no need for a GUI designer.
    d. LVGL comes with the drivers for all display controllers.
4. Which framework or library is NOT provided by Espressif Systems?

a. ESP-IDF Components library
b. ESP-ADF
c. ESP-WHO
d. ESP RainMaker

5. Which of the following frameworks can we use for audio processing?

a. ESP-ADF
b. ESP-WHO
c. ESP RainMaker
d. LVGL

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

5

Project - Audio Player

We have learned a lot so far. We began the journey with the very basics of IoT development on the ESP32 platform, discussed how to connect an ESP32 to the external world by employing its peripherals, and then talked about some popular third-party IoT libraries to speed up software production. Let's cement them by developing a fully-fledged project where we can practice our new skills.

In this chapter, we will design and develop an audio player with visual controls as we cover the following topics:

- The feature list of the audio player
- Solution architecture
- Developing the project
- Testing the project
- New features
- Troubleshooting

Technical requirements

We will use Visual Studio Code and the ESP-IDF command-line tools to create, develop, flash, and monitor the application in this chapter.

As hardware, only ESP32-S3-Box Lite will be employed.

You can find the project files here in the repository: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch5/audio_player.

The feature list of the audio player

A basic audio player usually has a playlist control and a volume control. Users can switch between the different recordings and control the volume. Let's itemize those features for our project:

- Play/pause a recording by pressing a button.
- Navigate to the next recording.
- Navigate to the previous recording.
- Each recording has an associated image. While the user navigates over the playlist, the image is updated on the screen automatically.
- Mute/unmute the volume.
- Increase/decrease the volume.
- All the functions have visual feedback on the LCD display.

The requirements are clear enough to begin with, but let's look at the following simple UI design prepared by *SquareLine Studio* to get a better grasp of what we will do in this project:

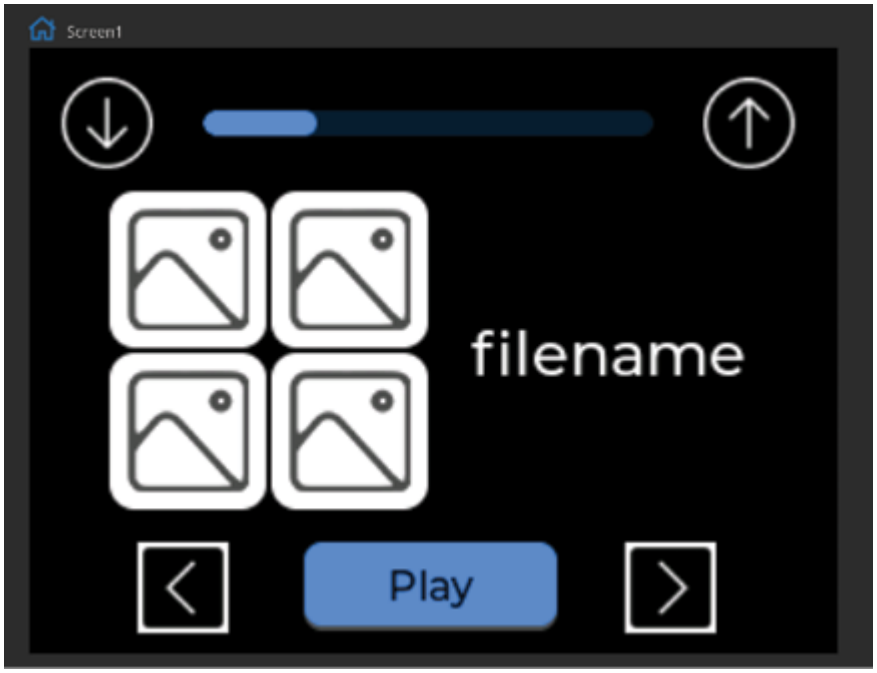


Figure 5.1: GUI of the application

Since the devkit has no touchscreen, we will use the physical buttons on it to provide the required functionality. At the top of the screen, we have the volume control. It has two visual buttons for volume up/down.

When a user presses the left physical button of the devkit, it will decrease the volume, and at the same time, a press event will be sent to the corresponding visual button (top-left) to indicate this event on the screen. The user can increase the volume by pressing the right button of the devkit. The same rule applies to the volume-up button at the top right of the screen. We have a bar in between them. It shows the volume level. The other requirement is to mute/unmute the player. We can use the middle button of the devkit to toggle the mute status. The following figure shows the button functions for volume control on the devkit:

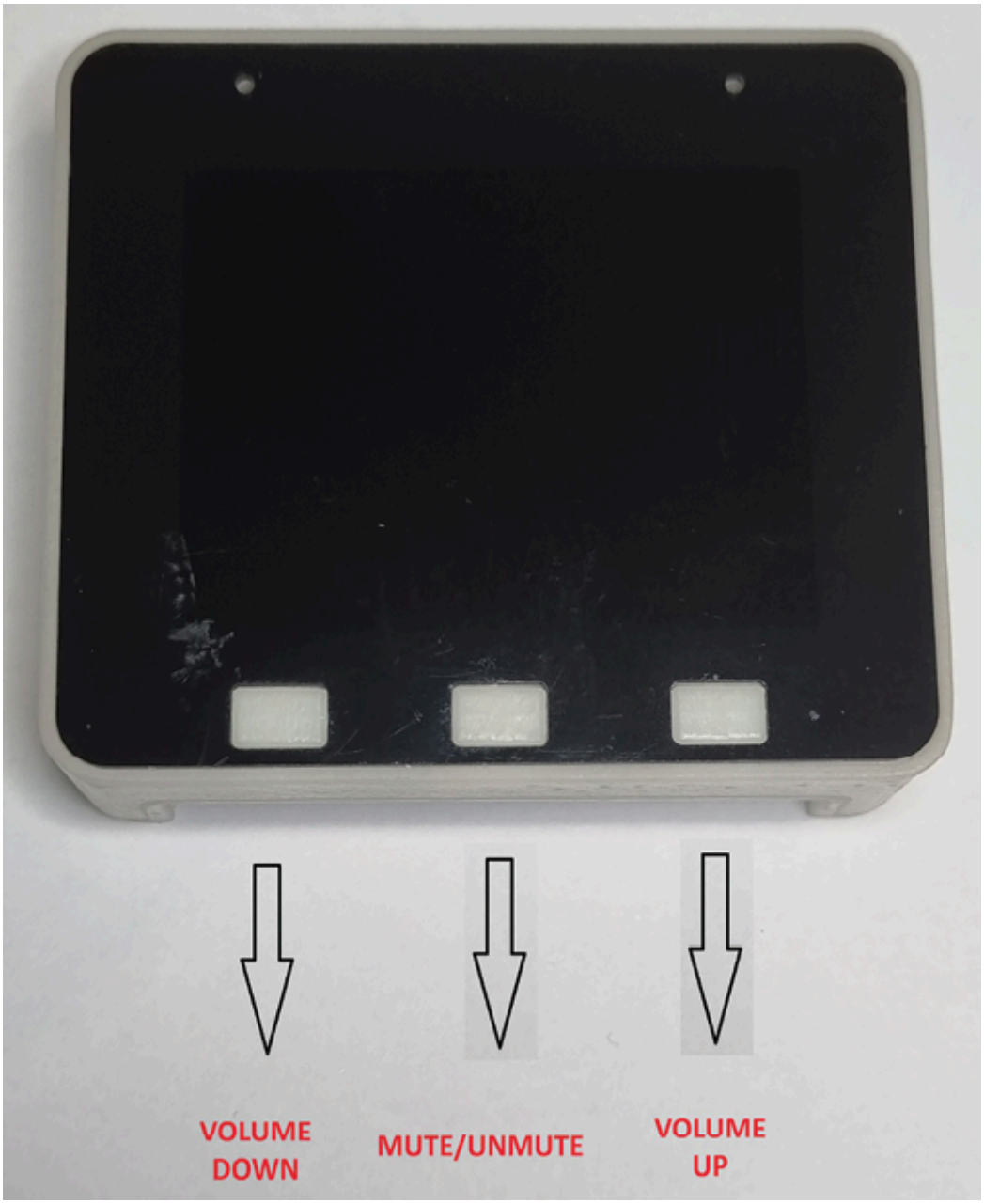


Figure 5.2: Button functions for volume control on ESP32-S3-Box-Lite

We will have several animal sounds and images in the flash. The audio player will traverse them. Under the volume control section of the screen, there will be an image container and a label to show the current animal. Both will be updated automatically when the user changes the current animal.

At the bottom of the screen, we will place the playlist control. The idea is very similar to the volume control. The left button of the devkit is to navigate to the previous animal in the list, and the right button is for the next animal. The middle button will Play/Pause the sound of the selected animal. In the following figure, we can see the button functions for the playlist control:

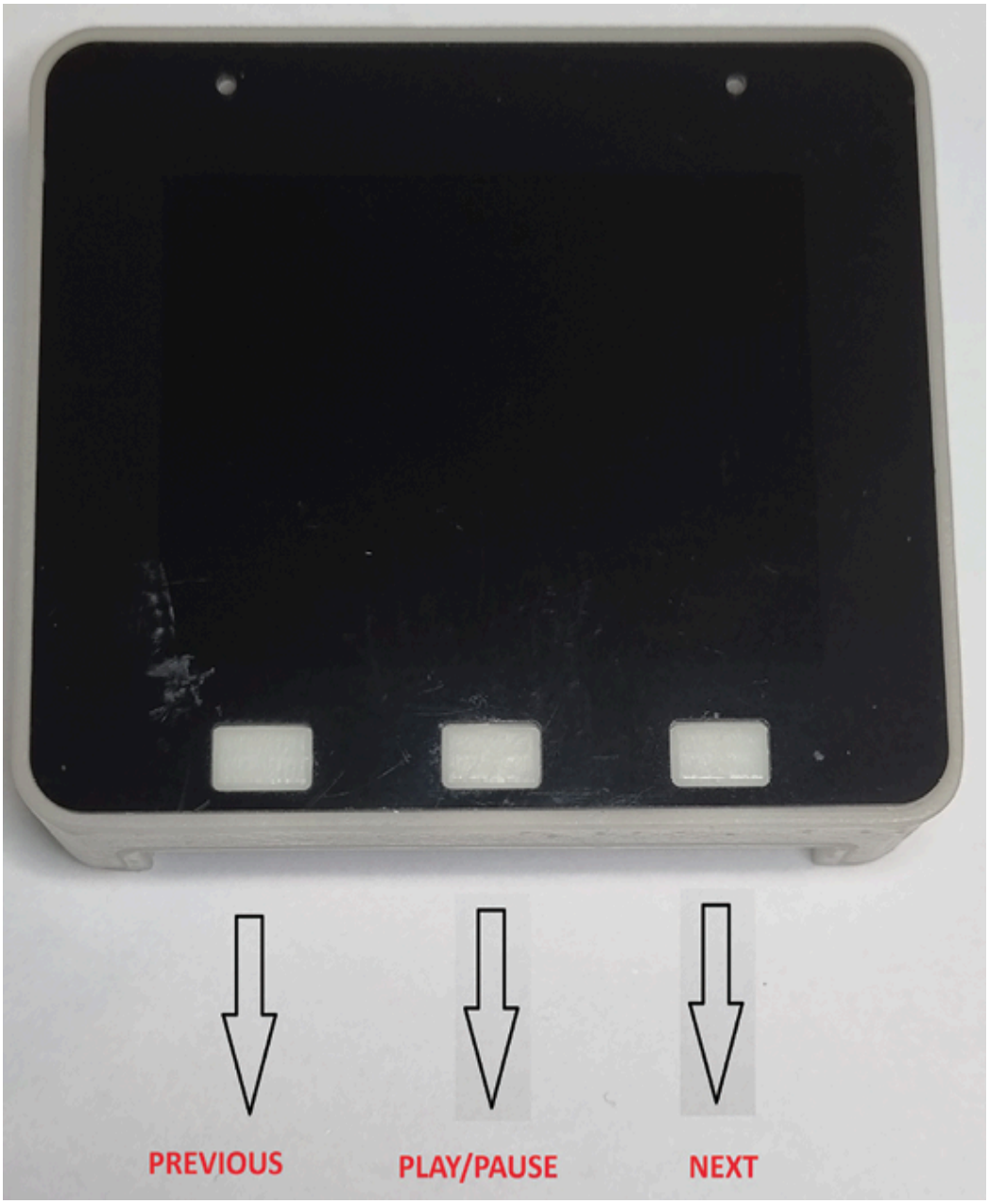


Figure 5.3: Playlist control on ESP32-S3-Box-Lite

We need to switch between the playlist control and the volume control. Let's assign the middle-button double-click event for this. When a user double-clicks on the middle button of the devkit, the focus toggles between the playlist control and the volume control. To show which one is activated, we can change the color of the Play/Pause button or the volume-level bar to another color. Let's say the active control will be red.

In the next topic, let's discuss the solution architecture that we can propose to implement the requirements of the project.

Solution architecture

For this list of features, we can have the following classes and relations between them to implement the requirements:

- `AppAudio`: Provides a simple interface for the audio functionality, consisting of the mute/unmute, Play/Pause, and volume up/down buttons. It initializes the audio sub-system and delivers audio events to its clients.
- `AppButton`: Handles user press events on the devkit buttons and notifies any client code via an event queue.
- `AppNav`: Keeps track of the animal list. It is the information source for the other components of the application, and it updates the current animal metadata when a user navigates by using the devkit buttons.
- `AppUi`: Encapsulates the generated UI files and manages the application flow for user interactions, by communicating with

the other classes mentioned above.

The following class diagram roughly depicts how classes relate to each other:

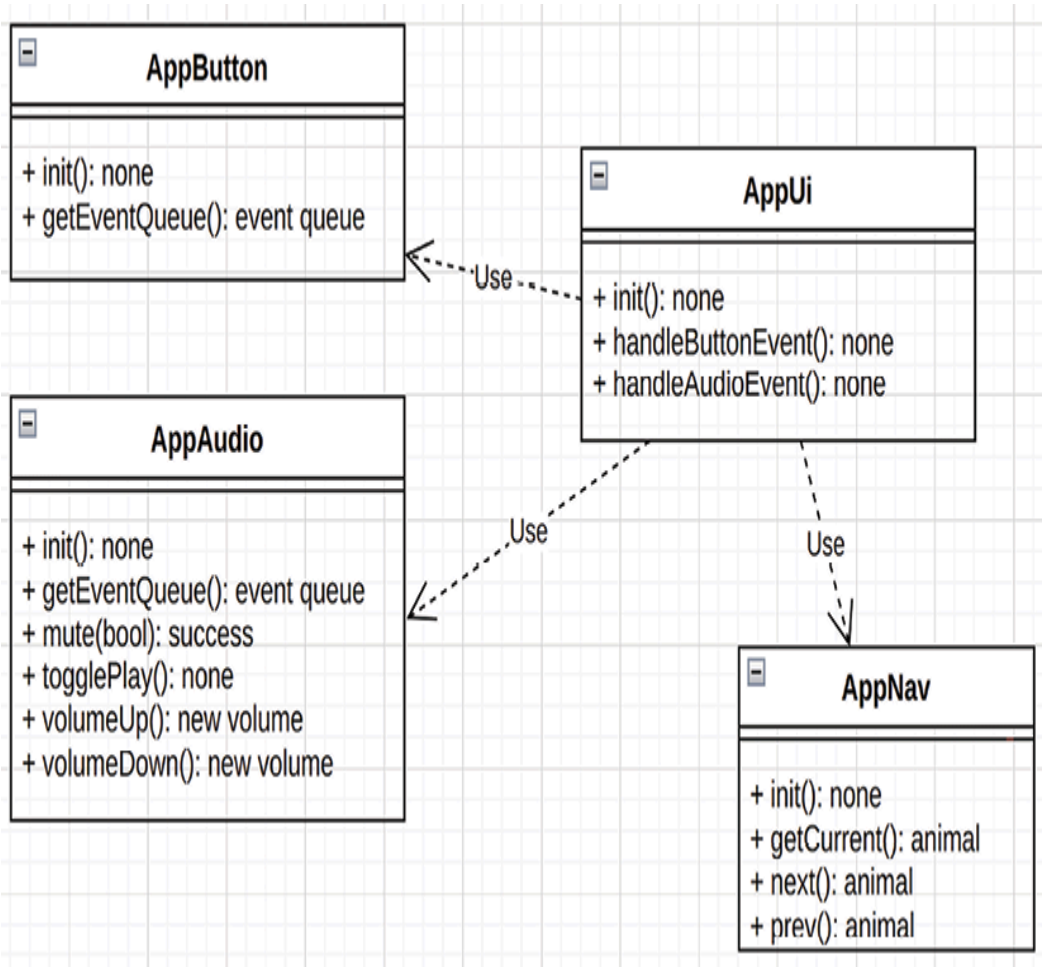


Figure 5.4: Class diagram

The implementation will have more supporting fields and methods, for sure, but this diagram shows the classes, their main responsibilities, and the relationships between them. After the application starts and the initialization of the class instances is done, the `AppUi` instance takes control and begins to receive button events from the `AppButton` instance. According to the active functionality

(playlist navigation or volume control), it commands the other class instances (`AppNav` or `AppAudio`). `AppUi` uses the generated UI elements to show the application state to the user. We will store the image files, sounds files, and a metadata file in the flash. `AppNav` will read the metadata file to populate the animal list and the multimedia files.

Now that we understand the solution architecture, we can implement the application.

Developing the project

There are two different aspects that we need to consider: the GUI design and application development. The application will have a **Graphical User Interface** (**GUI**) to engage users with visual indicators. We are going to use **Light and Versatile Embedded Graphics Library** *(***LVGL***)* for this purpose, as we already learned how to use it in the previous chapter. After having the GUI, we can integrate it into the application and move on from there, with the implementation of the actual application to react to user input.

Let's start with the GUI design.

Designing the GUI

Although you can just copy the generated UI files from the project repository, you can also try to design the GUI yourself by using *SquareLine Studio*. It is impossible to describe every detail here; nonetheless, I will list the fundamental steps below:

1. Start *SquareLine Studio*, and create a new project for **Espressif** / **ESP-BOX**.

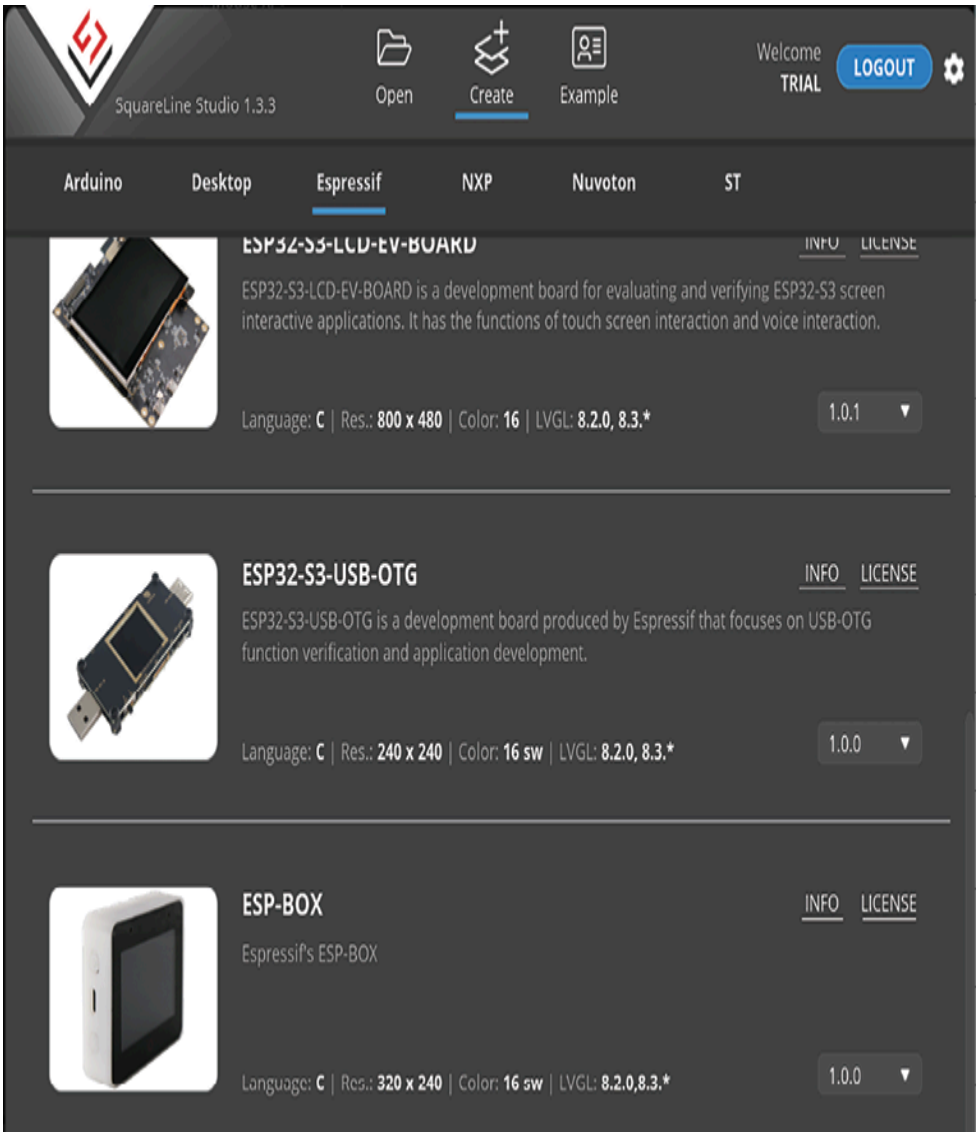


Figure 5.5: Create a new SquareLine project

2. Click on the **Imgbutton** widget in the **Widgets** / **CONTROLLER** panel. It will place an image button on **Screen1**. This will be the volume-down button. Drag it to the top left.

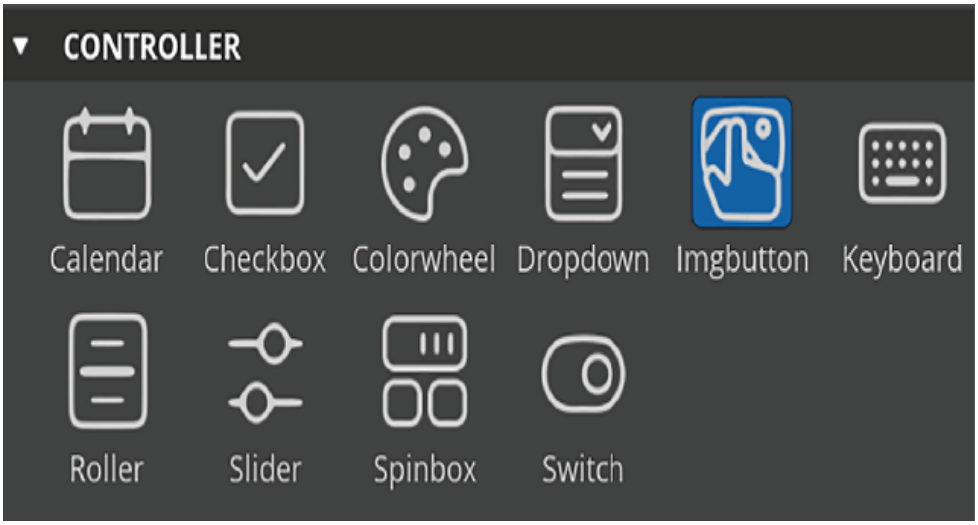


Figure 5.6: Image button

3. Download the icons from the project repository (the relative path of the icons is `ch5/audio_player/gui_design/assets`) and add them to the SquareLine project. They will appear in the **Assets** panel.

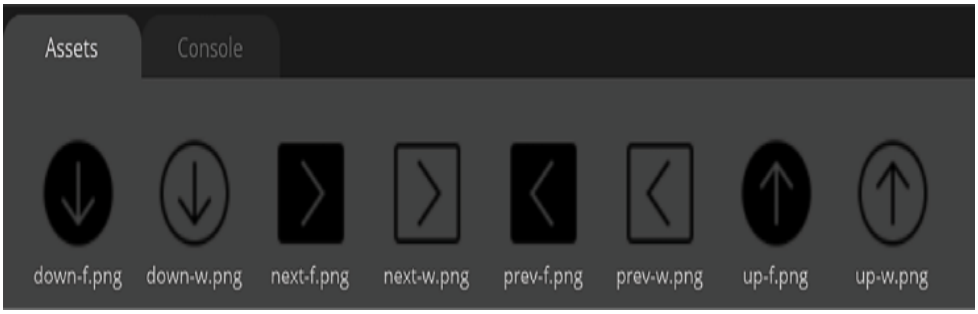


Figure 5.7: Icons in the Assets panel

4. Use the `down-f.png` and `down-w.png` files for the pressed and released states of the image button, respectively. You can edit the **Imgbutton** widget properties on the **Inspector** panel of the designer.

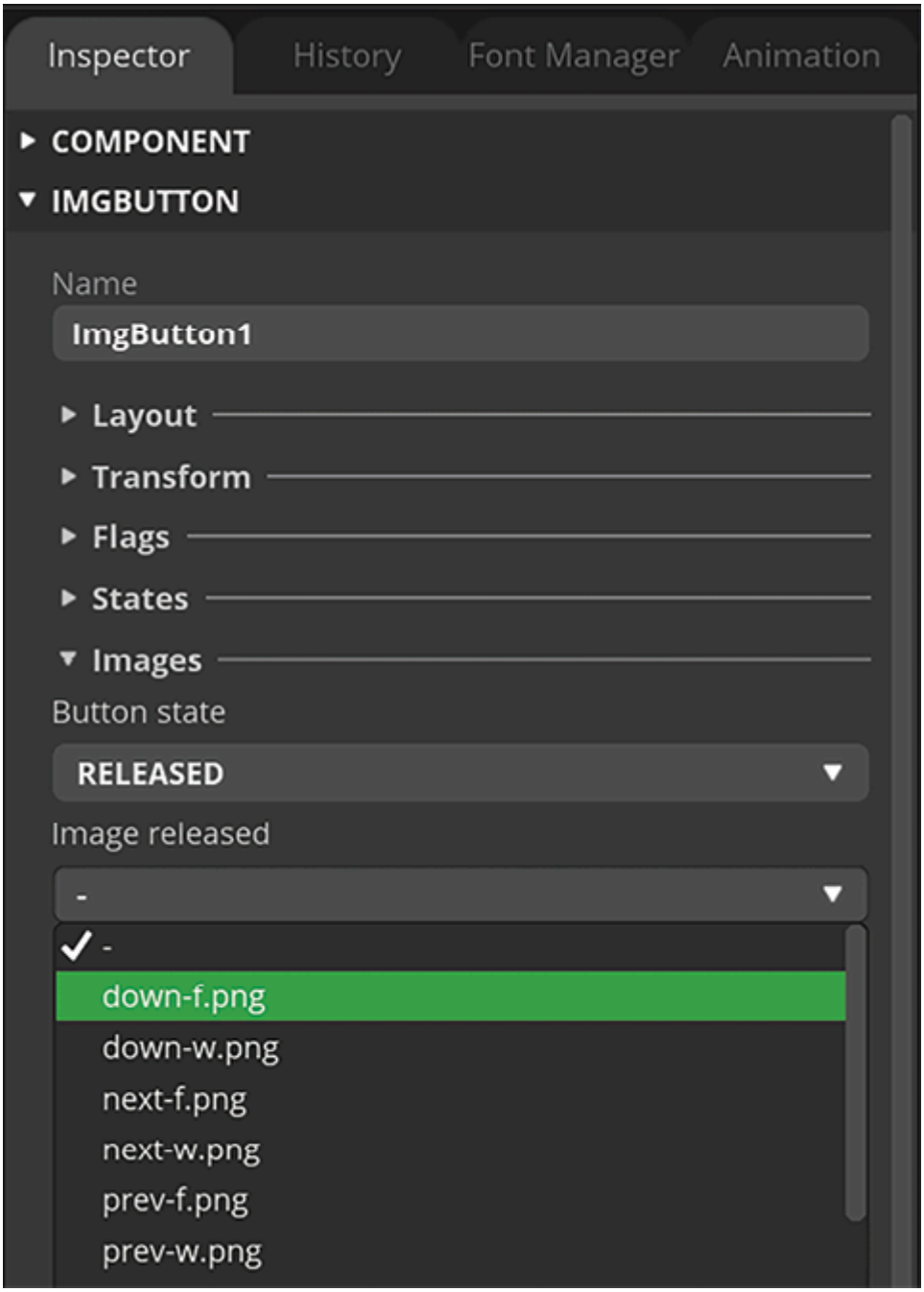


Figure 5.8: Image button properties

5. You can enable Play mode to see how the button behaves when pressed and released.

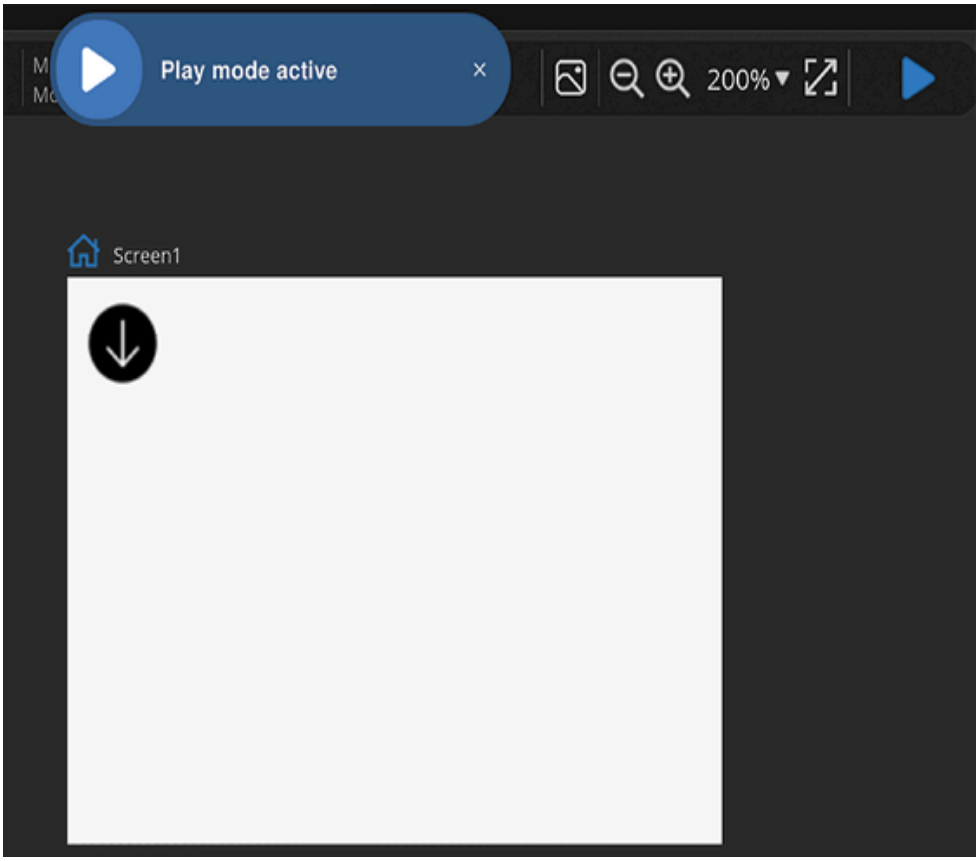


Figure 5.9: Play mode

6. Do the same for the rest three buttons (**volume-up**, **playlist next**, and **playlist previous**).
7. Place a **Bar** widget on **Screen1** by selecting from the **Widgets** / **VISUALIZER** panel. It will show the volume level.

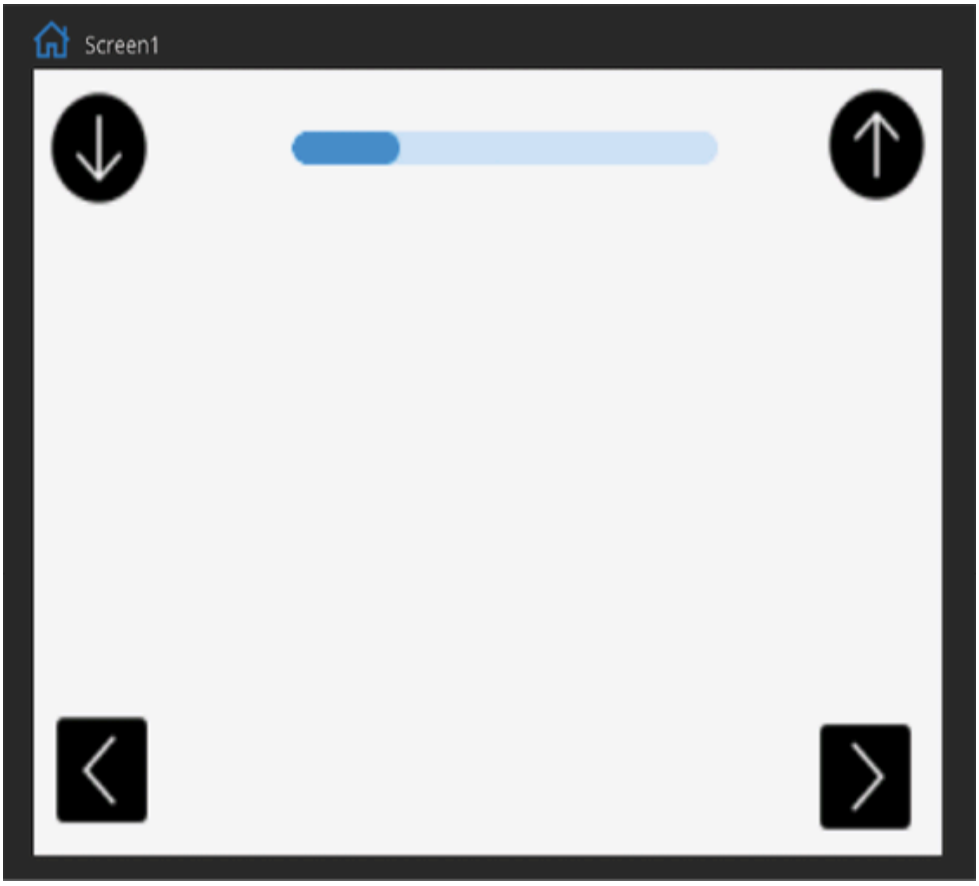


Figure 5.10: Bar widget

8. For the Play button, we will use two widgets: a button and a label. After placing them on **Screen1**, go to the **Hierarchy** panel and drag and drop the label widget onto the button widget. The button widget will be a container for the label widget.

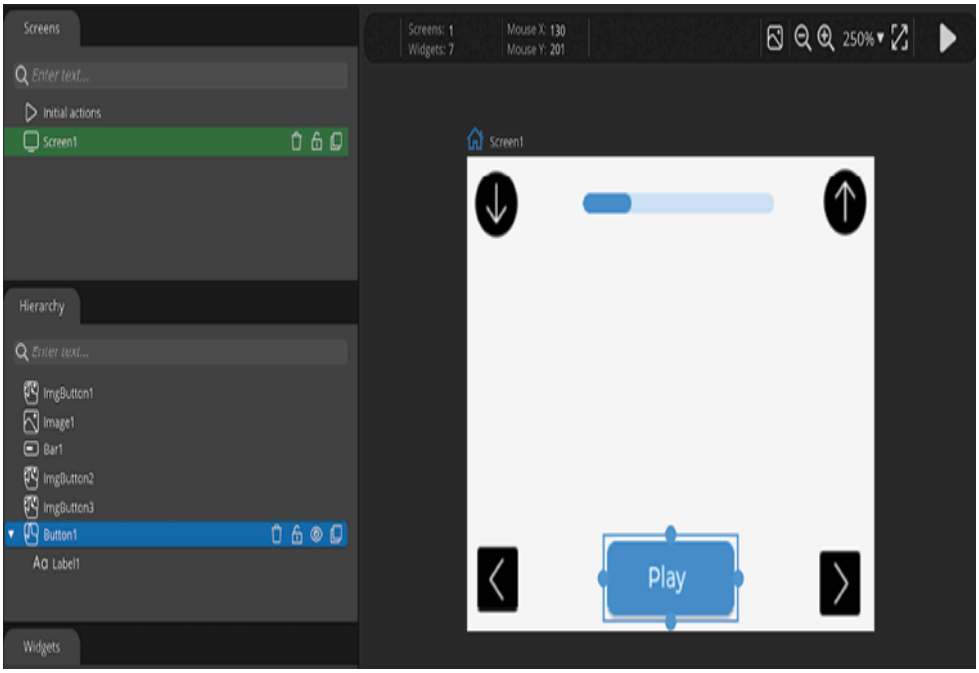


Figure 5.11: Button and label

9. Select an image widget from the **Widgets** panel, and set its size to **128 px** (**Width**) and **128 px** (**Height**) on the **Inspector** panel. This image widget will display the animal images.

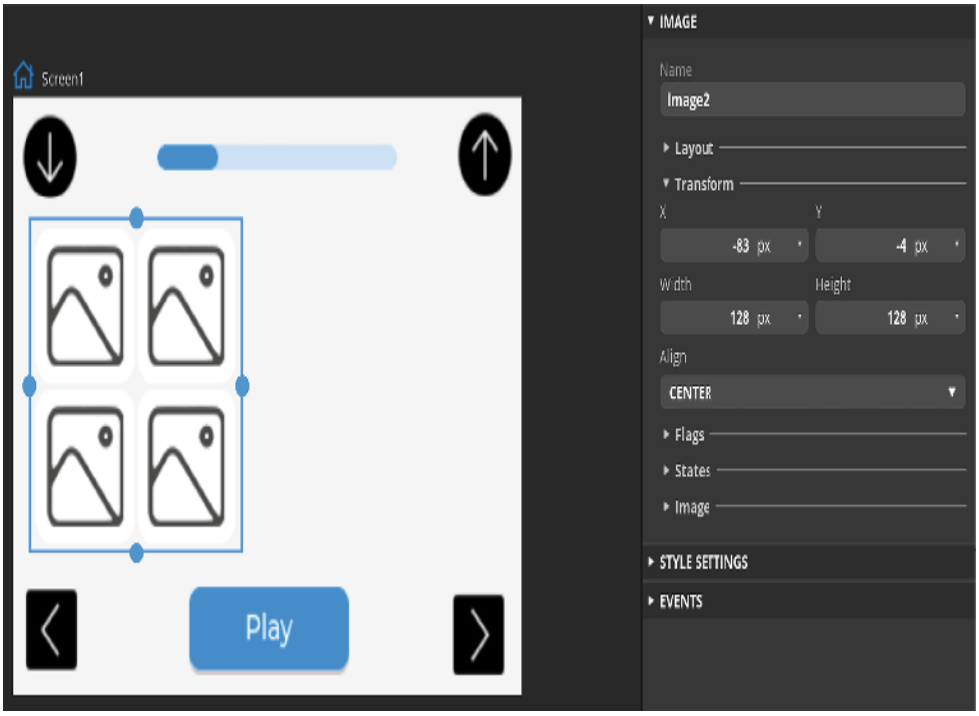


Figure 5.12: Image placeholder

10. For the last widget, place a label on the screen. The label will show the type of an animal being displayed. You can change its **Text Font** property from the **Inspector** panel, as shown in the following figure:

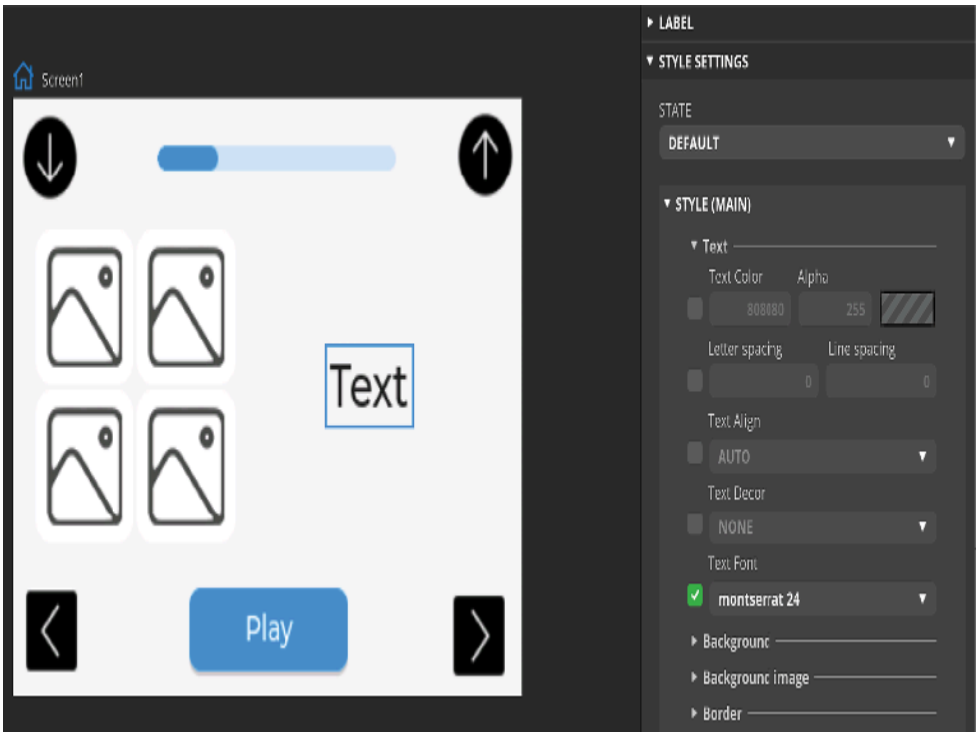


Figure 5.13: Setting the text font

We can add function placeholders for the event handlers of different GUI events, such as click, focus, or lost focus. When the Play button or the volume bar is focused on, the active one will be colored red, and the other one will return to its default color. Let's configure the Play button first.

11. Go to the **Inspector** panel, and open the **EVENTS** tab. Add an event, and select **CALL FUNCTION** as an action from the **Action** combo box.

12. Set the name of the function as `play_focused`. We will implement the body of this function later while coding the application.

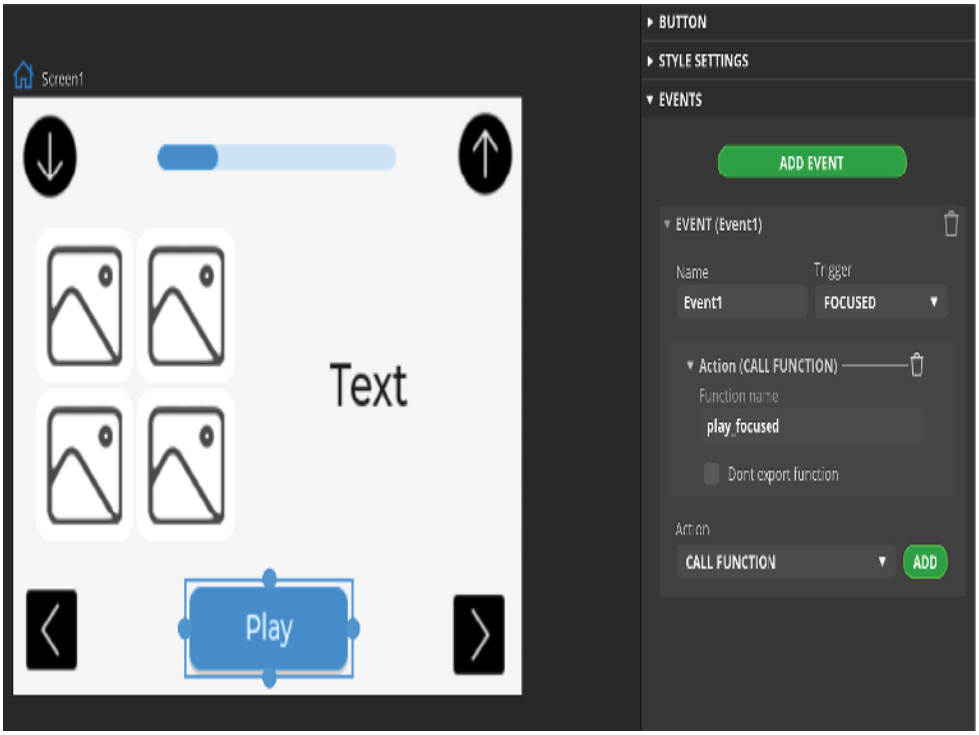


Figure 5.14: Adding a CALL FUNCTION event

13. Add another event for the volume bar, as shown in the following screenshot.

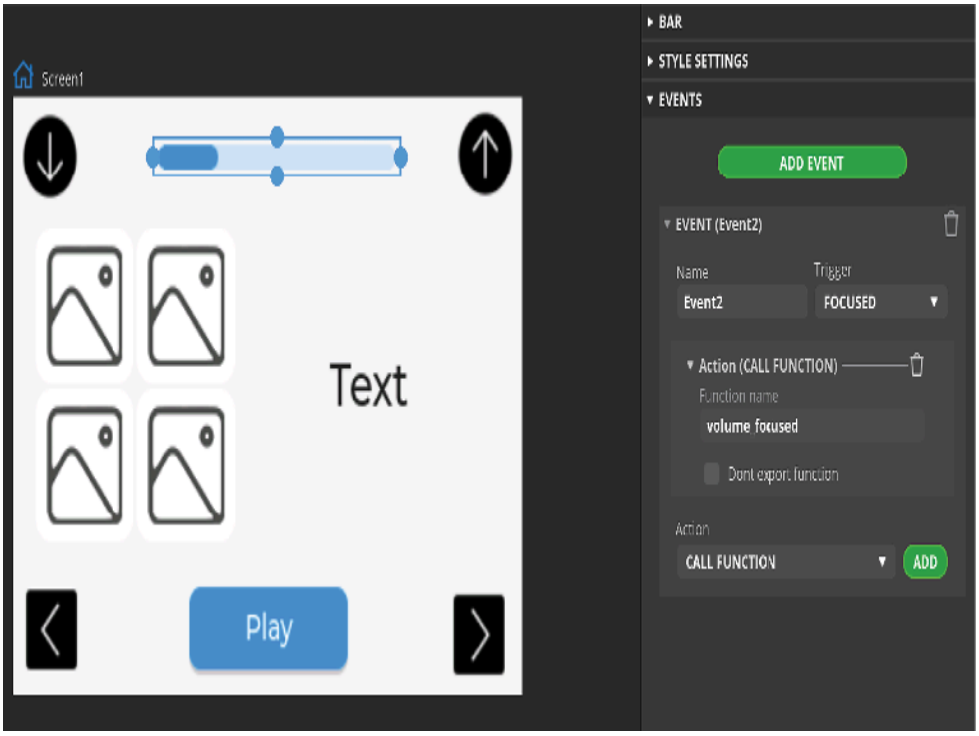


Figure 5.15: Focused event for the volume bar

The widgets have default names as assigned by the designer. Rename them something meaningful.

14. Export the project by selecting from the main menu **Export** / **Export UI Files**. You can move the generated code files next to the other source code files when we create the ESP-IDF project.

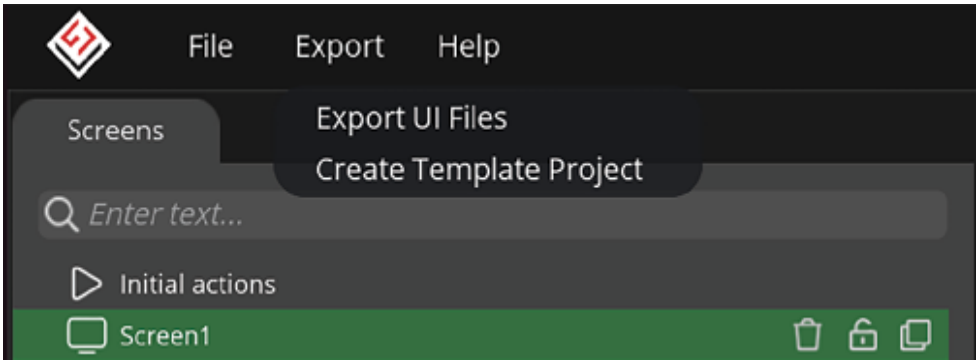


Figure 5.16: Export the project

15. See the generated code files in the directory you chose while exporting the project:

```
$ ls -1 ui*
ui.c
ui_events.c
ui.h
ui_helpers.c
ui_helpers.h
ui_img_1258062811.c
ui_img_1258080204.c
ui_img_1310788941.c
ui_img_1310804284.c
ui_img_1552218578.c
ui_img_1552235971.c
ui_img_603081905.c
ui_img_603097248.c
```

The GUI is ready, and we can use these project files to make LVGL and the underlying hardware drivers render the GUI on the LCD display of the devkit.

The SquareLine project files are provided in the repository at this link: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch5/audio_player/gui_design. You can open the project with the designer and see the final GUI design there. You can also just copy

> these generated files from the project repository here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch5/audio_player/main.

With the UI files generated, we can continue with the implementation of the classes and integrate them into an ESP-IDF project, as we will discuss next.

Creating the IDF project

Let's prepare the project in the following steps:

1. Create an ESP-IDF project:

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project audio_player
```

2. Rename the application source code file to `main/audio_player.cpp`.
3. Copy the `sdkconfig` and `partitions.csv` files from the book repository into the project root. `partitions.csv` defines a partition to store the animal multimedia files on the flash:

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x10000, 0x6000,
phy_init, data, phy,     ,        0x1000,
```

```
factory,  app,  factory, ,        1M,
storage,  data, spiffs,  ,        3M,
```

4. Copy the generated UI files into the `main/` directory.
5. Copy the `spiffs` directory from the project repository into the project root. You can find this directory at this link: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch5/audio_player/spiffs. It contains the `.mp3` and `.png` files for animals. It also has a JSON file, `info.json`, that describes the animals and associated files as metadata. Its content is:

```
[
    {
        "animal": "Dog",
        "audio": "dog.mp3",
        "image": "dog.png"
    },
    {
        "animal": "Donkey",
        "audio": "donkey.mp3",
        "image": "donkey.png"
    },
    {
        "animal": "Goose",
        "audio": "goose.mp3",
        "image": "goose.png"
    },
    {
```

```
        "animal": "Sheep",
        "audio": "sheep.mp3",
        "image": "sheep.png"
    }
]
```

6. We will use the *nlohmann-json* library to parse this file. Copy the library's single header file into the `main/` directory (`main/json.hpp`).
7. We need the ESP32-S3 Box-Lite **Board Support Package** (**BSP**) in order to enable its buttons. Set the `EXTRA_COMPONENT_DIRS` parameter in the root `CMakeLists.txt` to include the BSP in the project. It will look like this:

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
set(EXTRA_COMPONENT_DIRS ../../components)
add_compile_options(-fdiagnostics-color=always -
project(audio_player)
```

8. Update the `main/CMakeLists.txt` file to include all the source code files in the project and create the `storage` partition from the `./spiffs` directory. The updated content of the `main/CMakeLists.txt` file is:

```
idf_component_register(SRCS
    audio_player.cpp
    ui.c
```

```
    ui_events.c
    ui_helpers.c
    ui_img_1258062811.c
    ui_img_1258080204.c
    ui_img_1310788941.c
    ui_img_1310804284.c
    ui_img_1552218578.c
    ui_img_1552235971.c
    ui_img_603081905.c
    ui_img_603097248.c
    INCLUDE_DIRS ".")
spiffs_create_partition_image(storage ../spiffs
```

The IDF project is configured now, and we can code the application next.

Coding the application

As discussed in the solution architecture, we will have four classes: `AppAudio`, `AppButton`, `AppNav`, and `AppUi`. Let's start with the button handler class, `AppButton`, in `main/AppButton.hpp`:

```
#pragma once
#include "freertos/FreeRTOS.h"
#include "freertos/queue.h"
#include "bsp_board.h"
#include "bsp_btn.h"
#include "esp_log.h"
namespace app
```

```
{
    enum class eBtnEvent
    {
        L_PRESSED,
        L_RELEASED,
        M_CLICK,
        M_DCLICK,
        R_PRESSED,
        R_RELEASED
    };
}
```

After including the necessary header files, we will define an `enum` class, `eBtnEvent`, for the button events that we will generate when a user presses a button. Then, we will define the templated function prototype for the press handler callbacks as follows:

```
namespace
{
    template <app::eBtnEvent>
    void button_event_handler(void *param);
}
```

The template parameter is `app::eBtnEvent` so that we don't have to repeat the same code for each type of button event. Next comes the `AppButton` class:

```
namespace app
{
    class AppButton
    {
    private:
        QueueHandle_t m_event_queue = NULL;
```

In the `private` section of the `AppButton` class, we will define a FreeRTOS queue that we can send buttons events to. The `AppUi` class will later listen to this queue and process incoming events. Then, we will continue with the `init` function in the `public` section of the class:

```
    public:
        void init(void)
        {
            m_event_queue = xQueueCreate(10, sizeof(
                app::eBtnEvent));
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, button_event_hand
                app::eBtnEvent::L_PRESSED>, this);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_UP, button_event_handle
                app::eBtnEvent::L_RELEASED>, this);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_DOWN, button_event_hand
                app::eBtnEvent::R_PRESSED>, this);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_UP, button_event_handle
                app::eBtnEvent::R_RELEASED>, this);
```

```
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_SINGLE_CLICK, button_event_ha
                app::eBtnEvent::M_CLICK>, this);
            bsp_btn_register_callback(BOARD_BTN_ID_
                BUTTON_PRESS_REPEAT, button_event_ha
                app::eBtnEvent::M_DCLICK>, this);
        }
```

The `init` function creates an event queue for 10 button events and then registers the callbacks for each of the event types. After the `init` function, we define another function to retrieve the class instance from a pointer of the button structure, as follows:

```
        static AppButton &getObject(void *btn_ptr)
        {
            button_dev_t *btn_dev = reinterpret_cast
                button_dev_t *>(btn_ptr);
            return *(reinterpret_cast<app::AppButton
                btn_dev->cb_user_data));
        }
```

`getObject` is a static function that takes a button device pointer as a parameter and returns the class instance reference. It will help when reaching the instance from the button callback. We will finish the class implementation with a member function that returns the event queue handle next:

```
        QueueHandle_t getEventQueue(void) const {
            return m_event_queue; }
    }; // class end
} // namespace end
```

The `getEventQueue` function simply returns the member queue handle in order to allow access to the queue from outside. The only remaining coding in this file is the body of the `button_event_handler` function, as comes next:

```
namespace
{
    template <app::eBtnEvent E>
    void button_event_handler(void *btn_ptr)
    {
        app::AppButton &app_btn =
            app::AppButton::getObject(btn_ptr);
        app::eBtnEvent evt{E};
        xQueueSend(app_btn.getEventQueue(), (void *)
    }
}
```

In the `button_event_handler` function, we will create a local variable, `evt`, for the button event and pass it to the event queue of the `AppButton` instance by calling the `xQueueSend` FreeRTOS function. This completes the implementation of the button handler.

We will continue with the implementation of the audio features by creating another source code file, `main/AppAudio.hpp`, and editing

it as follows:

```
#pragma once
#include <cstdio>
#include <cinttypes>
#include <string>
#include "freertos/FreeRTOS.h"
#include "freertos/queue.h"
#include "esp_err.h"
#include "bsp_codec.h"
#include "bsp_board.h"
#include "bsp_storage.h"
#include "audio_player.h"
namespace
{
    void audio_player_callback(audio_player_cb_ctx_t
}
```

As usual, we first include the header files needed in the implementation. Then, we declare the callback prototype for audio events. It will be called when an audio event occurs. We don't have to use all audio events and can customize them for our purposes. The following `enum` class shows the events that we will use in the application:

```
namespace app
{
    enum class eAudioEvent
    {
```

```
        PLAYING,
        STOPPED
    };
}
```

We are only interested in the states when the audio player starts playing and stops. We will define them in the `eAudioEvent enum` class. Next, we will begin with the `AppAudio` class implementation:

```
namespace app
{
    class AppAudio
    {
    private:
        FILE *m_fp;
        uint8_t m_vol;
        audio_player_state_t m_player_state;
        QueueHandle_t m_event_queue;
        bool m_muted;
```

The `private` section of the class contains several member variables. We have a file pointer, `m_fp`, for the audio files to be passed to the audio player, and we will keep the volume level in the `m_vol` member variable. We will also keep the internal state of the audio player to be able to toggle between the playing and stopped states. The event queue is to signal to the other parts of the application (in fact, it is only the `AppUi` class) about the audio player state changes.

The last variable shows whether the audio player is muted or not. In the `public` section, we will start with the constructor:

```
    public:
        AppAudio() : m_fp(nullptr),
                     m_vol{50},
                     m_player_state(AUDIO_PLAYER_STA
                     m_event_queue(nullptr),
                     m_muted(false) {}
```

The constructor simply sets the initial values of the member variables and nothing more. The next function actually initializes the hardware:

```
        void init(audio_player_mute_fn fn)
        {
            audio_player_config_t config = {.port =
                                            .mute_fn
                                            .priorit
            audio_player_new(config);
            audio_player_callback_register(
                 audio_player_callback, this);
            bsp_codec_set_voice_volume(50);
            m_event_queue = xQueueCreate(10, sizeof(
                 app::eBtnEvent));
        }
        QueueHandle_t getEventQueue(void) const {
            return m_event_queue;}
```

The `init` function takes a `mute` function as a parameter, and we will use this callback while constructing the configuration for the audio player. We also register the `audio_player_callback` function that we declared at the beginning to track the changes in the audio player state, by calling the `audio_player_callback_register` function. The audio event queue is created at the end of the `init` function. The `getEventQueue` function returns the audio event queue for the outside world. Next, we will implement the `mute` function of the class:

```
        uint8_t mute(bool m, bool toggle = false)
        {
            uint8_t val = 0;
            m_muted = toggle ? !m_muted : m;
            if (m_muted)
            {
                bsp_codec_set_mute(true);
            }
            else
            {
                bsp_codec_set_mute(false);
                bsp_codec_set_voice_volume(m_vol);
                val = m_vol;
            }
            return val;
        }
```

The `mute` member function controls the audio hardware for muting/unmuting. Please note that it is not the callback to be passed to the `init` function; their signatures are different. We will wrap the `mute` member function within another function to make it compatible with the `init` function's parameter. In the body, we will set the `m_muted` variable to `true` or `false`, or we will toggle it, depending on the value of the `toggle` parameter. After checking the value of the `m_muted` variable, we will call the `bsp_codec_set_mute` function to set the mute state of the audio player. The returned value from the `mute` function is the volume level. Let's implement the Play/Stop functionality in the next function:

```
        void togglePlay(const std::string &filename)
        {
            switch (m_player_state)
            {
            case AUDIO_PLAYER_STATE_PLAYING:
                audio_player_pause();
                break;
            case AUDIO_PLAYER_STATE_PAUSE:
                audio_player_resume();
                break;
            default:
                m_fp = fopen(filename.c_str(), "rb");
                audio_player_play(m_fp);
                break;
```

```
        }
    }
```

In the `togglePlay` function, we will check the internal state of the audio player and take action accordingly. If it is in the playing state, we will pause it; if it is already paused, we will resume it. The default action is to open the given audio file and play it, by calling the `audio_player_play` function with the file pointer. Next comes the `volumeUp` and `volumeDown` member functions:

```
    uint8_t volumeUp(void)
    {
        if (m_vol < 100)
        {
            m_vol += 10;
            bsp_codec_set_voice_volume(m_vol);
        }
        return m_vol;
    }
    uint8_t volumeDown(void)
    {
        if (m_vol > 0)
        {
            m_vol -= 10;
            bsp_codec_set_voice_volume(m_vol);
        }
        return m_vol;
    }
```

These two functions are very similar to each other. The `volumeUp` function increases the volume by 10, and the `volumeDown` function does the opposite. Both return the final volume level. The last function of the class is `setState`, as follows:

```
        void setState(audio_player_state_t state)
        {
            m_player_state = state;
        }
    }; // class end
} // namespace end
```

The `setState` function only updates the `private` `m_player_state` member variable with the given value. This function will be used from the audio event callback, `audio_player_callback`, that we registered in the `init` function. We will implement its body next:

```
namespace
{
    void audio_player_callback(audio_player_cb_ctx_t
    {
        app::AppAudio &app_audio = *(reinterpret_cas
            app::AppAudio *>(param->user_ctx));
        audio_player_state_t state = audio_player_ge
        app_audio.setState(state);
        app::eAudioEvent evt = state ==
            AUDIO_PLAYER_STATE_PLAYING ? app::eAudio
            PLAYING : app::eAudioEvent::STOPPED;
```

```
            xQueueSend(app_audio.getEventQueue(), (void
        }
    }
```

The `audio_player_callback` function is called when something changes in the state of the audio player. We will update the `AppAudio` instance with this change and send the customized event to the audio event queue of the `AppAudio` instance. The implementation of the audio functionality is done. Next, we will develop the animal navigation in the `main/AppNav.hpp` file:

```
#pragma once
#include <fstream>
#include <string>
#include <vector>
#include "json.hpp"
namespace app
{
    struct Animal_t
    {
        std::string animal;
        std::string audio;
        std::string image;
    };
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(Animal_t, an
        audio, image);
```

Here, we need the *nlohmann-json* library to parse the `/spiffs/info.json` file, which contains metadata about the animal records. Each record has the animal name, the audio file name, and the image file name. We will define the `Animal_t` structure to hold this information and also define the JSON conversion functions, with the help of the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro that comes with the *nlohmann-json* library. Next, we will continue with the `AppNav` class:

```
    class AppNav
    {
    private:
        int m_pos;
        std::vector<Animal_t> m_animals;
```

The `private` section of the `AppNav` class has only two member variables, the `Animal_t` vector and the record position on this vector. Then, we will move on to the `public` section:

```
    public:
        void init()
        {
            std::ifstream fs{"/spiffs/info.json"};
            m_animals = nlohmann::json::parse(fs);
            m_pos = 0;
        }
```

The `init` function parses the metadata information and initializes the member variables. The `parse` function returns a

`nlohmann::json` object, but this object is implicitly converted to a vector of the `Ani mal_t` type, since we had the `from_json` function by calling the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro. The next member function returns the current animal record:

```
        Animal_t &getCurrent(void)
        {
            return m_animals[m_pos];
        }
```

We will return the reference of the current animal record from the `getCurrent` function. The remaining two functions are for navigation, as shown next:

```
        Animal_t &next(void)
        {
            m_pos = (m_pos + 1) % m_animals.size();
            return m_animals[m_pos];
        }
        Animal_t &prev(void)
        {
            m_pos = (m_pos - 1) % m_animals.size();
            return m_animals[m_pos];
        }
    }; // class end
} // namespace end
```

The `next` member function advances the position on the animal vector and returns the reference of the record at that position.

Similarly, the `prev` function returns a reference for the previous animal record in the vector.

We have completed the implementation of the navigation features and are now ready to integrate all of them in the GUI handling. Let's create the `main/AppUi.hpp` file for it and edit it next:

```
#pragma once
#include <mutex>
#include <string>
#include "bsp_lcd.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
#include "ui.h"
#include "AppButton.hpp"
#include "AppNav.hpp"
#include "AppAudio.hpp"
namespace
{
    app::AppNav m_nav;
}
```

We will start by including the `hpp` header files that we have developed for the button, audio, and navigation. In the anonymous namespace, we will define an instance of the `AppNav` class. Next comes the implementation of the `AppUi` class:

```
namespace app
{
    class AppUi
    {
    private:
        static std::mutex m_ui_access;
        static void lvglTask(void *param)
        {
            while (true)
            {
                {
                    std::lock_guard<std::mutex> lock
                    lv_task_handler();
                }
                vTaskDelay(pdMS_TO_TICKS(10));
            }
        }
```

The `m_ui_access` variable is a static `mutex` that controls the access to the underlying LVGL objects and functions, and the `lvglTask` function is a FreeRTOS task that renders the LVGL objects in a loop with a period of `10` ticks. Since it is a FreeRTOS task, the function won't return. Next, we will continue with the definition of the member variables:

```
        bool m_playlist_active;
        app::AppAudio *m_app_audio;
        app::AppButton *m_app_button;
```

If you remember, users can switch between the playlist and the volume control by double-clicking on the middle button of the devkit. The `m_playlist_active` variable denotes whether the playlist is the active feature of the GUI. The `m_app_audio`'s and `m_app_button`'s are pointers to the `AppAudio` and `AppButton` class instances, respectively. Several function prototypes come next:

```
        static void buttonEventTask(void *param);
        static void audioEventTask(void *param);
        static std::string makeImagePath(const std:
            &filename);
        static std::string makeAudioPath(const std:
            &filename);
        void update(const Animal_t &an, bool play =
        void toggleControl(void);
```

We will implement the function bodies later, but here is a quick summary of the functions' roles:

- `buttonEventTask` is the FreeRTOS task that monitors the button event queue.
- `audioEventTask` is the FreeRTOS task that monitors the audio event queue.
- `makeImagePath` is the utility function that returns the full path of an image on the filesystem.
- `makeAudioPath` is the utility function that returns the full path of an audio file on the filesystem.

- `update` is a member function to re-render the GUI when the active position changes on the playlist.
- `toggleControl` is a member function to toggle the control between the playlist and volume.

Now, we will develop the functions in the `public` section:

```
    public:
        void init(app::AppButton *button, app::AppAu
        {
            m_app_button = button;
            m_app_audio = audio;
            lv_port_init();
            ui_init();
            m_nav.init();
```

The `init` function takes two pointer parameters to the `button` and `audio` instances. We will keep them locally in the class by assigning them to their respective member variables. Then, we will initialize the LVGL library and the GUI that we designed at the very beginning of the project. We will also initialize the `m_nav` object by calling its `init` function.

Let's continue with more initialization and the configuration of some UI elements:

```
            m_playlist_active = true;
            lv_event_send(ui_btnPlay, LV_EVENT_FOCUS
            lv_bar_set_range(ui_barVolume, 0, 100);
```

```
            lv_bar_set_value(ui_barVolume, 50,
                lv_anim_enable_t::LV_ANIM_OFF);
```

The default active feature is the playlist, and we will set the focus on the UI Play button, `ui_btnPlay`, by calling the `lv_event_send` function of the LVGL library. The next two lines in the above code configure the volume bar with a range of 0 to 100, setting its initial value as `50`. The `init` function is not finished yet:

```
            update(m_nav.getCurrent(), false);
            bsp_lcd_set_backlight(true);
```

The `update` member function updates the GUI with the first animal in the playlist. After that, we turn on the backlight of the LCD display. Before exiting the `init` function, we will start the FreeRTOS tasks, as follows:

```
            xTaskCreatePinnedToCore(lvglTask, "lvgl"
                nullptr, 3, nullptr, 0);
            xTaskCreate(buttonEventTask, "btn_evt",
                this, 3, nullptr);
            xTaskCreate(audioEventTask, "audio_evt",
                this, 3, nullptr);
        }
```

We have three FreeRTOS tasks: for LVGL, for button events, and for audio events. The task functions are the ones that we declared in the

`private` section of the `AppUi` class earlier. Now, we will define the functions to return the audio and button event queues:

```
QueueHandle_t getButtonEventQueue(void) cons
    return m_app_button->getEventQueue(); }
QueueHandle_t getAudioEventQueue(void) const
    return m_app_audio->getEventQueue(); }
```

The `getButtonEventQueue` and `getAudioEventQueue` functions are simply wrappers for the button and audio objects' `getEventQueue` functions. They will be needed when monitoring the queues in the FreeRTOS tasks. Next comes the heart of the `AppUi` class, the `handleButtonEvent` function:

```
void handleButtonEvent(app::eBtnEvent &btn_e
{
    std::lock_guard<std::mutex> lock(m_ui_ac
    switch (btn_evt)
    {
    case app::eBtnEvent::M_DCLICK:
        toggleControl();
        break;
```

In the `handleButtonEvent` function, we will first lock the `m_ui_access` mutex to prevent any other code from modifying the LVGL objects, since we will work on them here. In a `switch` statement, we will check the incoming button event. If it is a double-

click event on the middle button of the devkit, we will toggle the control between the playlist and the volume. Next, we will add the case for a single click on the middle button:

```
            case app::eBtnEvent::M_CLICK:
                if (m_playlist_active)
                {
                    lv_event_send(ui_btnPlay, LV_EVE
                        nullptr);
                    m_app_audio->togglePlay(makeAudi
                        m_nav.getCurrent().audio));
                }
                else
                {
                    lv_bar_set_value(ui_barVolume,
                        m_app_audio->mute(false, tru
                        lv_anim_enable_t::LV_ANIM_OF
                }
                break;
```

In the case of a single click on the middle button, the behavior of the application changes according to the active feature. If the playlist is active, then we will toggle the play status by calling the `togglePlay` function of the `m_app_audio` object. If the volume control is active, this time we will toggle the mute status of the volume. The reactions to the button events are reflected in the respective UI elements. The active feature check also applies to all other button-click events. Let's handle the left-button pressed-down event next:

```
            case app::eBtnEvent::L_PRESSED:
                if (m_playlist_active)
                {
                    lv_event_send(ui_btnPrev, LV_EVE
                        nullptr);
                }
                else
                {
                    lv_event_send(ui_btnVolumeDown,
                        LV_EVENT_PRESSED, nullptr);
                }
                break;
```

In this case, we will simply update the UI buttons with the pressed-down visual events. The actual actions will be taken when the button is released, as shown next:

```
            case app::eBtnEvent::L_RELEASED:
                if (m_playlist_active)
                {
                    lv_event_send(ui_btnPrev, LV_EVE
                        nullptr);
                    update(m_nav.prev());
                }
                else
                {
                    lv_event_send(ui_btnVolumeDown,
                        LV_EVENT_RELEASED, nullptr);
                    lv_bar_set_value(ui_barVolume,
```

```
                    m_app_audio->volumeDown(),
                    lv_anim_enable_t::LV_ANIM_OF
            }
            break;
```

When the left button is released, in addition to the UI element updates, we will also call the action functions. If the playlist is the active feature, we will navigate to the previous animal on the list; else (the volume control is active), we will decrease the volume. The remaining button events are for the right button, as follows:

```
        case app::eBtnEvent::R_PRESSED:
            if (m_playlist_active)
            {
                lv_event_send(ui_btnNext,
                    LV_EVENT_PRESSED, nullptr);
            }
            else
            {
                lv_event_send(ui_btnVolumeUp,
                    LV_EVENT_PRESSED, nullptr);
            }
            break;
        case app::eBtnEvent::R_RELEASED:
            if (m_playlist_active)
            {
                lv_event_send(ui_btnNext, LV_EVE
                    nullptr);
                update(m_nav.next());
```

```
                }
                else
                {
                    lv_event_send(ui_btnVolumeUp,
                        LV_EVENT_RELEASED, nullptr);
                    lv_bar_set_value(ui_barVolume,
                        m_app_audio->volumeUp(),
                        lv_anim_enable_t::LV_ANIM_OF
                }
                break;
            default:
                break;
            } // switch end
        } // function end
```

The logic is similar to the left button event handling. Again, when released, the right button causes it to pass to the next animal if the playlist is active, or increases the volume level if the volume control is active. The relevant UI elements are all updated during these events. This finishes the `handleButtonEvent` function. Next, we will implement the `handleAudioEvent`, as follows:

```
        void handleAudioEvent(app::eAudioEvent &evt)
        {
            if (evt == app::eAudioEvent::PLAYING)
            {
                lv_label_set_text(ui_txtPlayButton,
            }
            else
```

```
            {
                lv_label_set_text(ui_txtPlayButton,
            }
        } // function end
    }; // class end
```

The `handleAudioEvent` function takes the audio event as a parameter. If the audio is playing, then the user can pause it, and we will indicate this state by changing the label of the `ui_txtPlayButton` UI element to `Pause`. Otherwise, the label becomes `Play`. Let's focus on the development of the `private` functions that we skipped at the beginning of the class implementation. The first one is the `buttonEventTask` function:

```
    void AppUi::buttonEventTask(void *param)
    {
        AppUi &ui = *(reinterpret_cast<AppUi *>(para
        app::eBtnEvent evt;
        while (true)
        {
            xQueueReceive(ui.getButtonEventQueue(),
                &evt, portMAX_DELAY);
            ui.handleButtonEvent(evt);
        }
    }
```

In the `buttonEventTask` function, we will wait on the button event queue. When an event comes, we will pass it to the `AppUi` object to

be processed. The next function does the same thing for the audio events:

```
void AppUi::audioEventTask(void *param)
{
    AppUi &ui = *(reinterpret_cast<AppUi *>(para
    app::eAudioEvent evt;
    while (true)
    {
        xQueueReceive(ui.getAudioEventQueue(),
            &evt, portMAX_DELAY);
        ui.handleAudioEvent(evt);
    }
}
```

The `audioEventTask` function processes the audio events in a loop by waiting on the audio event queue. When an event appears in the queue, it is passed to the `AppUi` object. The following function creates an image path for the LVGL image widget:

```
std::string AppUi::makeImagePath(const std::str
{
    return std::string("S:/spiffs/") + filename
}
```

An interesting fact about LVGL is that it expects a drive letter as the prefix of a path. We can set it to `S` in `sdkconfig` by updating the value of `CONFIG_LV_FS_POSIX_LETTER` to `83` (the decimal number

83 corresponds to the character **S** on the ASCII table). You can also run `menuconfig` and go to the LVGL section of the configuration (navigate to **(Top)** → **LVGL configuration** → **3rd Party Libraries**) to update this value. In the next function, we will create paths for the audio files:

```
std::string AppUi::makeAudioPath(const std::str
{
    return std::string("/spiffs/") + filename;
}
```

The `makeAudioPath` function just returns the full path on the filesystem for a given file name. Let's continue with the `update` function of the `AppUi` class:

```
void AppUi::update(const Animal_t &an, bool pla
{
    lv_label_set_text(ui_txtAnimal, an.animal.c_
    lv_img_set_src(ui_imgAnimal, makeImagePath(
        an.image).c_str());
}
```

We will call the `update` function when the user changes the current animal. The function updates the `ui_txtAnimal` and `ui_imgAnimal` UI elements with the current record, hence the name. The final function of the class is `toggleControl`, as shown next:

```
    void AppUi::toggleControl(void)
    {
        m_playlist_active = !m_playlist_active;
        if (m_playlist_active)
        {
            lv_event_send(ui_btnPlay, LV_EVENT_FOCUS
        }
        else
        {
            lv_event_send(ui_barVolume, LV_EVENT_FOC
                nullptr);
        }
    } // function end
    std::mutex AppUi::m_ui_access;
} // namespace end
```

In the `toggleControl` function, we will set focus on either the `ui_btnPlay` UI element or the `ui_barVolume` UI element, by sending the `LV_EVENT_FOCUSED` event after toggling the `m_playlist_active` member variable. This finalizes the class implementation.

There is one last requirement left that we need to discuss before moving on to the main application in the `main/audio_player.cpp` file: setting the focus on the Play button or the volume bar should change its color to red. To do this, we will edit the `main/ui_events.c` file, which is one of the outputs of the *SquareLine* designer:

```
#define NORMAL_COLOR 0x4C93F4
#define FOCUSED_COLOR 0xFF0000
void play_focused(lv_event_t * e)
{
    lv_obj_set_style_bg_color(ui_barVolume, lv_colo
        NORMAL_COLOR), LV_PART_INDICATOR | LV_STATE_
    lv_obj_set_style_bg_color(ui_btnPlay, lv_color_h
        FOCUSED_COLOR), LV_PART_MAIN | LV_STATE_DEFA
}
void volume_focused(lv_event_t * e)
{
    lv_obj_set_style_bg_color(ui_btnPlay, lv_color_h
        NORMAL_COLOR), LV_PART_MAIN | LV_STATE_DEFAU
    lv_obj_set_style_bg_color(ui_barVolume, lv_colo
        FOCUSED_COLOR), LV_PART_INDICATOR | LV_STATE
}
```

If you remember, we named these two event handlers while designing the GUI. They are the functions to be run by the *LVGL* engine when the `ui_btnPlay` or `ui_barVolume` UI element gets focus. The `play_focused` function sets the color of `ui_btnPlay` as red and the color of `ui_barVolume` as blue, and the `volume_focused` function swaps the button colors. Now, we're done with the class implementations and can edit the `main/audio_player.cpp` file to integrate all the classes as the final application:

```
#include "bsp_board.h"
#include "bsp_storage.h"
```

```
#include "AppUi.hpp"
#include "AppButton.hpp"
#include "AppAudio.hpp"
namespace
{
    app::AppButton m_app_btn;
    app::AppAudio m_app_audio;
    app::AppUi m_app_ui;
    esp_err_t audio_mute_function(AUDIO_PLAYER_MUTE_
        setting)
    {
        m_app_audio.mute(setting == AUDIO_PLAYER_MU
        return ESP_OK;
    }
}
```

After including the class headers, we declare `m_app_btn`, `m_app_audio`, and `m_app_ui` in the anonymous namespace. We also define the `audio_mute_function` function to be passed to the audio initialization. Then, we develop the application entry point, the `app_main` function, as shown next:

```
extern "C" void app_main(void)
{
    bsp_board_init();
    bsp_board_power_ctrl(POWER_MODULE_AUDIO, true);
    bsp_spiffs_init("storage", "/spiffs", 10);
```

In the `app_main` function, we will first initialize the devkit by calling two BSP functions. Then, we will mount the `storage` partition by calling the `bsp_spiffs_init` function, in order to access the files in it. Next comes the object initializations:

```
    m_app_btn.init();
    m_app_audio.init(audio_mute_function);
    m_app_ui.init(&m_app_btn, &m_app_audio);
}
```

We will initialize the button and the audio object, and then call the `init` function of the UI object with the pointers of the other two. When the application starts on the devkit, these objects take control and run.

The project is ready for the tests on the devkit now. Congratulations on coming so far!

Testing the Project

Let's begin the fun part. We will flash the application, as follows:

```
$ idf.py erase-flash clean flash monitor
Executing action: erase-flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<more logs>
```

```
I (1026) lv_port: Try allocate two 320 * 20 display
I (1028) lv_port: Add KEYPAD input device to LVGL
```

After a successful flash, you should be able to see the GUI on the LCD display:

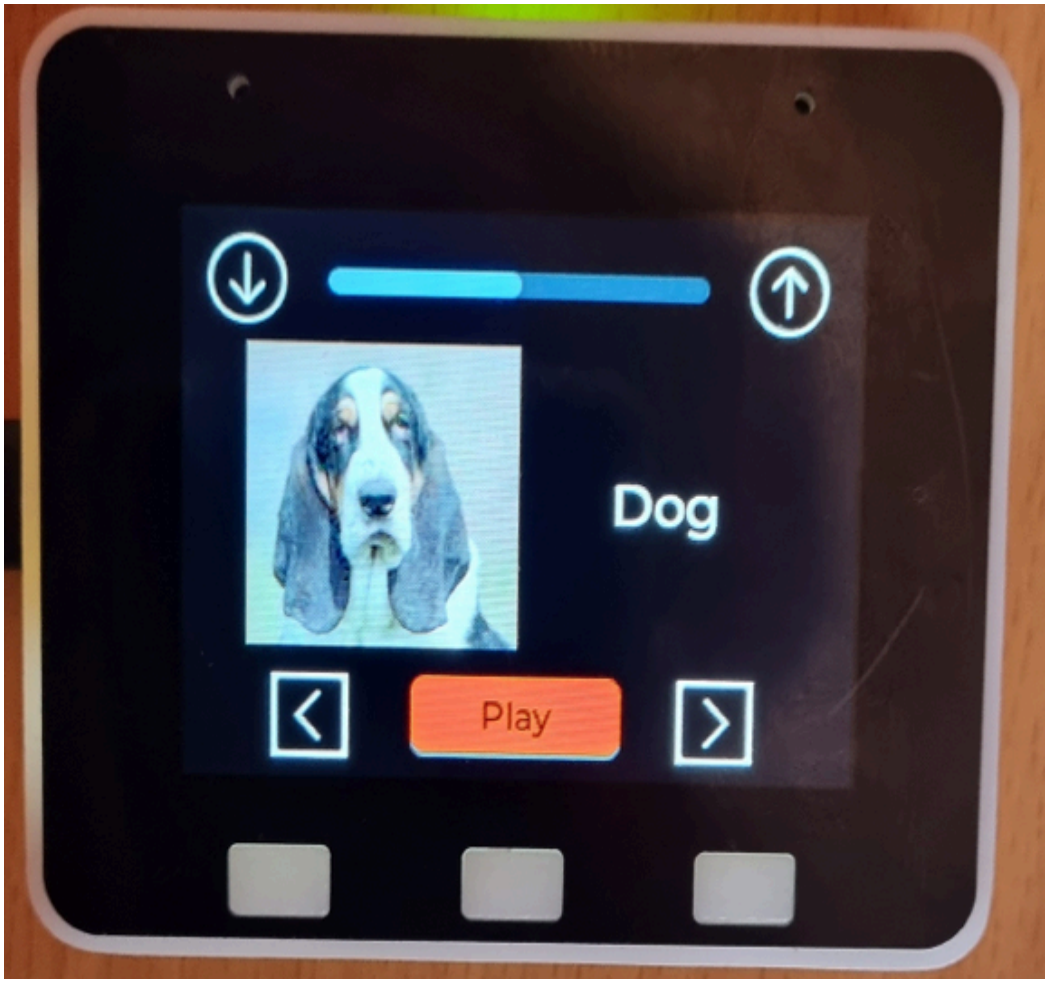


Figure 5.17: Application GUI

Here are some test cases:

- Press the middle button to play the dog sound. See it pause when you press it once more while it is playing.

- See the button name change to **Play** automatically when the audio finishes.
- Press the left and right buttons to navigate over the animal list. See the GUI update accordingly.
- Double-click on the middle button to change the focus to the volume control. See the color of the volume bar change to red.
- Increase and decrease the volume.
- Double-click on the middle button to return to the playlist. When you Play the animal sound, check whether the volume is updated.

The project has room for improvement. You can easily apply many other techniques from what we learned in the first four chapters. In the next topic, you can find some ideas to improve the project as further practice.

New features

You make the project more interesting by adding the following features:

- The application employs SPIFFS as the filesystem. Replace it with LittleFS.
- The images load rather slowly with a visible effect on the GUI while rendering. One way to make it faster is to load all of the images to **Pseudo-RAM** (**PSRAM**) when the application starts. Instead of reading from the flash, you can make LVGL use them from the PSRAM.

- Create a new LVGL theme for a different level of ambient light. Use LDR to measure the light level, and change the theme automatically according to the light level (https://docs.lvgl.io/latest/en/html/overview/style.html#themes).
- Save/restore the latest status of the player on the `nvs` partition (the latest volume level and the index of the last played item).
- Add two touch sensors to navigate the playlist, with the same functionality as the left and right buttons when the playlist is active.
- Add a beep sound when a user changes the volume level.

Troubleshooting

The project has several distinct stages (GUI design, project configuration, and coding) with relatively long source code; thus, it is easy to make mistakes during development. The following list may help if you encounter issues while working on it:

- First of all, make sure the project compiles successfully on your development machine and runs on the devkit, as cloned from the book repository.
- If you choose to develop the project from the ground up yourself, follow the configuration steps exactly as given in the *Creating the IDF project* topic.
- The project uses some IDF components from the book repository; therefore, the root `CMakeLists.txt` file should set the directory that contains those components as `EXTRA_COMPONENT_DIRS`.

- The project has a custom partitions file, `partitions.csv`. Make sure it is included in the project.
- The `sdkconfig` file that comes with the project clone contains the right configuration for the project (flash size/partition definitions, LVGL configuration, SPIRAM, etc.). You can copy it from the book repository, as described in the *Creating the IDF project* topic. The `menuconfig` tool can help if you want to see what is included in the `sdkconfig` file.
- If you change the project configuration files (`CMakeLists.txt`, `sdkconfig`, or `partitions.csv`) and the application doesn't behave as expected, you can try a clean build on the erased flash and check the serial console if any error is printed:

```
$ idf.py erase-flash fullclean flash monitor
```

Summary

Learning is enhanced with practice on practice, and this project was a good opportunity for this. We developed a project from the ground up, starting from the feature list of the audio player, then discussing the solution architecture, and finally, development and testing. The development stage also has different internal stages. UI/UX design is an important step for a successful product. With a given list of features, a talented UI/UX designer can optimize a GUI and emphasize the unique selling points much better. As developers, we put our efforts into implementing solution architecture by applying our knowledge and skills. Each project moves us forward to become better developers.

This project marks the end of the first part of the book. In the upcoming chapters, we will learn how to develop connected ESP32 products on cloud systems.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

6

Using Wi-Fi Communication for Connectivity

Wi-Fi (of the IEEE 802.11 family of standards) is the most prominent wireless standard in the industry, so Espressif has integrated this technology into all of its products except the ESP32-H2 series SoC, which combines IEEE 802.15.4 and Bluetooth LE wireless connectivity on the same hardware. In this chapter, we will learn how to develop applications in Wi-Fi environments. After connecting to a local Wi-Fi network, the huge world of IoT opens, which allows ESP32 to communicate with servers and other connected devices. ESP-IDF provides all the software support needed to develop **Transmission Control Protocol/Internet Protocol** (**TCP/IP**) applications.

This chapter contains practical examples of Wi-Fi connectivity basics as well as popular IoT protocols over TCP/IP, such as **Message Queue Telemetry Transport** (**MQTT**) and **Representational State Transfer** (**REST**) services. In the *Further reading* section, you can find more resources to learn about the basics of modern networking and the TCP/IP family of protocols, including REST and MQTT, if you are not familiar with them.

In this chapter, we're going to cover the following topics:

- Connecting to local Wi-Fi
- Provisioning ESP32 on a Wi-Fi network
- Communicating over an MQTT broker
- Running a RESTful server on ESP32
- Consuming RESTful services

Technical requirements

As hardware, we will only use ESP32-C3-DevKitM-1 in the examples of this chapter. The Devkit has a button and LED on it, so it is enough for our purposes to try bi-directional communication for use cases with input/output. The code is located in the GitHub repository here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch6

There are also several third-party software tools that we need to test the final applications. They are:

- **curl**: A command-line tool for communication over TCP/IP-based protocols. Its website for download and documentation is here: https://curl.se/
- **Mosquitto**: An MQTT broker from the Eclipse Foundation. It also contains other utility tools to publish/subscribe to MQTT topics. You can download the broker and find the documentation here: https://mosquitto.org/
- **ESP SoftAP Provisioning**: A mobile application from Espressif for provisioning. Available for both Android and iOS.

- **Python 3.6.8** and **pyenv**: *pyenv* is a Python tool to create and manage virtual environments. Here is a good article about it: https://realpython.com/intro-to-pyenv/

We'll begin with examples relating to Wi-Fi communication.

Connecting to local Wi-Fi

Nodes in a Wi-Fi network form a star topology, which means a central hub exists and other nodes connect to the central hub to communicate within the Wi-Fi network. They can also talk to the outside world if the hub is connected to an **Internet Service Provider** (ISP) as a router. On a Wi-Fi network, we see two different modes of operation:

- **Access point** (**AP**) mode
- **Station** (**STA**) mode

We can configure ESP32 in both modes. In STA mode, ESP32 can connect to an access point and join a Wi-Fi network as a node. When it is in AP mode, other Wi-Fi-capable devices, such as mobile phones, can connect to the Wi-Fi network that ESP32 starts as the access point. The following figure shows both cases with two different Wi-Fi networks:

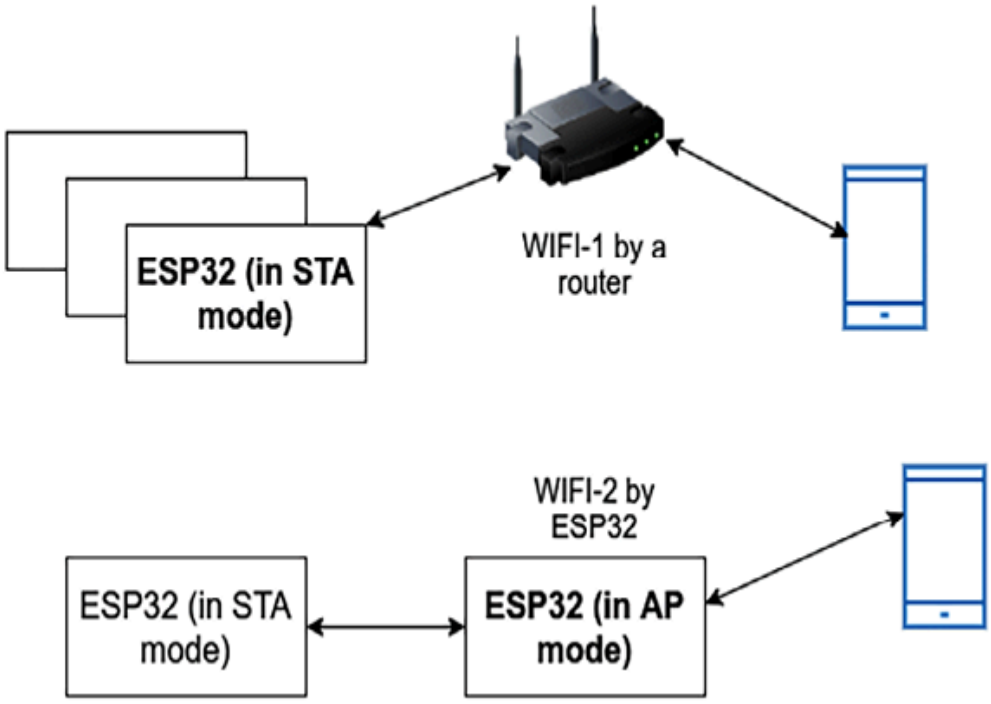


Figure 6.1: ESP32 in STA mode and AP mode

Let's see an example of connecting ESP32 to a Wi-Fi network. In this project, we will develop a class with two callbacks for Wi-Fi-connected and Wi-Fi-disconnected events. The devkit is ESP32-C3-DevKitM-1. Let's create a project next.

Creating a project

We can prepare an ESP-IDF project with the following steps:

1. Create an ESP-IDF project from the command line (you can choose any other way described in *Chapter 2, Understanding the Development Tools*):

```
$ export $HOME/esp/esp-idf/export.sh
```

```
$ idf.py create-project wifi_ex
```

2. Copy the `sdkconfig.defaults` file from the project repository (`ch6/wifi_ex`) into the project root. It contains the hardware definition (chip type and flash size).
3. We want to keep the SSID and password outside the code for security reasons. *cmake* will read them from the environment and pass them to the compiler as preprocessor macros. The content of the `CMakeLists.txt` file in the root is:

```
cmake_minimum_required(VERSION 3.5)
if(DEFINED ENV{WIFI_SSID})
  add_compile_options(-DWIFI_SSID=$ENV{WIFI_SSID
else(DEFINED ENV{WIFI_SSID})
  message(FATAL_ERROR "WiFi SSID is needed")
endif(DEFINED ENV{WIFI_SSID})
if(DEFINED ENV{WIFI_PWD})
  add_compile_options(-DWIFI_PWD=$ENV{WIFI_PWD})
else(DEFINED ENV{WIFI_PWD})
  message(FATAL_ERROR "WiFi password is needed")
endif(DEFINED ENV{WIFI_PWD})
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(wifi_ex)
```

4. Rename the default source code file `main/wifi_ex.cpp` and update the content of the `main/CMakeLists.txt` file as follows:

```
idf_component_register(SRCS "wifi_ex.cpp" INCLUD
```

Now that we have the ESP-IDF project configured and ready to develop, we can move on to coding.

Coding the application

Let's add a new C++ header file to implement the Wi-Fi handling class, named `main/AppWifi.hpp`:

```
#pragma once
#include <functional>
#include <string>
#include <cstring>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_wifi.h"
#include "nvs_flash.h"
namespace app
{
    using OnWifiConnected_f = std::function<void(
        esp_ip4_addr_t *)>;
    using OnWifiDisconnected_f = std::function<void(
```

After the header files, we define the callback types in the `app` namespace for the Wi-Fi events. Then, we implement the `AppWifi` class as follows:

```
class AppWifi
{
private:
    OnWifiConnected_f m_connected;
    OnWifiDisconnected_f m_disconnected;
    static void handleWifiEvent(void *arg, esp_e
```

In the `private` section of the class, we have two function objects as class member variables: `m_connected` and `m_disconnected`. They will be provided by the clients of an `AppWifi` instance while initializing it. When these events occur, the instance will call them. The `handleWifiEvent` function is the callback that we register for the Wi-Fi events when we initialize the ESP-IDF Wi-Fi library. Next, we continue with the `public` section of the class implementation:

```
public:
    void init(OnWifiConnected_f conn, OnWifiDisc
    {
        m_connected = conn;
        m_disconnected = disc;
        esp_err_t err = nvs_flash_init();
        if (err == ESP_ERR_NVS_NO_FREE_PAGES ||
            ESP_ERR_NVS_NEW_VERSION_FOUND)
        {
            nvs_flash_erase();
            nvs_flash_init();
        }
```

The `init` function of the `AppWifi` class takes two parameters as we discussed earlier. We set the `m_connected` and `m_disconnected` member variables to them. Then we call the `nvs_flash_init` function to initialize the NVS partition of the flash. The default configuration of a new ESP-IDF project contains this partition to save the application configuration parameters, such as Wi-Fi SSID and password. The ESP-IDF Wi-Fi library needs the NVS partition for that reason. We aren't done yet with the `init` function:

```
            esp_netif_init();
            esp_event_loop_create_default();
            esp_netif_create_default_wifi_sta();
```

The `esp_netif_init` function initializes the underlying TCP/IP stack. We also call the `esp_event_loop_create_default` function to create a default event loop by which the Wi-Fi events are delivered. To create a Wi-Fi network interface for station mode, we call the `esp_netif_create_default_wifi_sta` function. The Wi-Fi network interface is created but the Wi-Fi infrastructure is not initialized yet. We'll do it next:

```
            wifi_init_config_t cfg = WIFI_INIT_CONF
            esp_wifi_init(&cfg);
            esp_event_handler_register(WIFI_EVENT,
                ESP_EVENT_ANY_ID, handleWifiEvent, t
            esp_event_handler_register(IP_EVENT, IP_
```

```
                handleWifiEvent, this);
        }
```

We initialize the Wi-Fi driver with the default configuration by calling the `esp_wifi_init` function. This allocates the necessary resources for Wi-Fi and starts the Wi-Fi FreeRTOS task. The next two calls to the `esp_event_handler_register` function register our `handleWifiEvent` class member function for the Wi-Fi and IP events. They belong to two different event groups and therefore need to be registered separately. We also pass the `this` object pointer to the `esp_event_handler_register` function. When the `handleWifiEvent` class function is called, it will receive the `AppWifi` object instance as one of its parameters. The initialization is done and we can develop the `connect` function:

```
        void connect(void)
        {
            const char *ssid = WIFI_SSID;
            const char *password = WIFI_PWD;
            wifi_config_t wifi_config;
            memset(&wifi_config, 0, sizeof(wifi_con
            memcpy(wifi_config.sta.ssid, ssid, strle
            memcpy(wifi_config.sta.password, passwor
                strlen(password));
            wifi_config.sta.threshold.authmode =
                WIFI_AUTH_WPA2_PSK;
```

In the `connect` function, we create a configuration variable for Wi-Fi, `wifi_config`. We set the SSID and password fields of `wifi_config` with the values that come from the environment variables and the Wi-Fi authentication mode as **Wi-Fi Protected Access 2/Pre-Shared-Key** (**WPA2/PSK**). Then we complete the `connect` member function by starting Wi-Fi:

```
            esp_wifi_set_mode(WIFI_MODE_STA);
            esp_wifi_set_config(WIFI_IF_STA, &wifi_c
            esp_wifi_start();
        } // function end
    }; // class end
```

We set the Wi-Fi mode to STA and configure the Wi-Fi driver with the `wifi_config` configuration variable. Finally, we call the `esp_wifi_start` function to start the process.

We declared the `handleWifiEvent` function but didn't implement it. Its implementation comes next:

```
    void AppWifi::handleWifiEvent(void *arg, esp_eve
        event_base, int32_t event_id, void *event_da
    {
        AppWifi *obj = reinterpret_cast<AppWifi *>(a
        if (event_base == WIFI_EVENT && event_id ==
            WIFI_EVENT_STA_START)
        {
```

```
            esp_wifi_connect();
        }
```

The first event that we handle in the `handleWifiEvent` function is `WIFI_EVENT_STA_START`. It This is when we run `esp_wifi_start` in the `connect` member function. The call to the `esp_wifi_connect` function here actually tries to connect to the configured Wi-Fi network. The next case that we are going to handle is the Wi-Fi-disconnected case:

```
        else if (event_base == WIFI_EVENT && event_
            WIFI_EVENT_STA_DISCONNECTED)
        {
            obj->m_disconnected();
            vTaskDelay(pdMS_TO_TICKS(3000));
            esp_wifi_connect();
        }
```

When the `WIFI_EVENT_STA_DISCONNECTED` event happens, we first call the `m_disconnected` member of the `AppWifi` instance to let the client code know about this event. After 3 seconds, we try to connect to Wi-Fi again by calling the `esp_wifi_connect` function. The last event is when the devkit connects to the Wi-Fi network and gets an IP, as comes next:

```
        else if (event_base == IP_EVENT && event_id
            IP_EVENT_STA_GOT_IP)
```

```
        {
            ip_event_got_ip_t *event = reinterpret_
            obj->m_connected(&event->ip_info.ip);
        }
    } // function end
} // namespace end
```

The `IP_EVENT_STA_GOT_IP` event marks a successful connection to the configured Wi-Fi network. We call the `m_connected` member of the `AppWifi` instance and pass the IP information to the client code.

The `AppWifi` class is ready. We can use it in any application where a Wi-Fi connection is needed. Let's do this in `main/wifi_ex.cpp` to see how it works:

```
#include <cinttypes>
#include "esp_log.h"
#include "AppWifi.hpp"
#define TAG "app"
namespace
{
    app::AppWifi app_wifi;
}
```

We start by including the `AppWifi.hpp` header file and defining an instance of the `AppWifi` class. Then we move on to the `app_main` function implementation:

```
extern "C" void app_main(void)
{
    auto connected = [](esp_ip4_addr_t* ip)
    {
        ESP_LOGI(TAG, ">>>>>>> connected");
    };
    auto disconnected = []()
    {
        ESP_LOGW(TAG, ">>>>>>> disconnected");
    };
```

In the `app_main` function, we define two lambda functions as callbacks for the `app_wifi` object. We pass them to its `init` member function as follows:

```
    app_wifi.init(connected, disconnected);
    app_wifi.connect();
}
```

We initialize the `app_wifi` object and then call the `connect` member function to start the Wi-Fi connection process. That is it. The rest is handled by the `app_wifi` object, as we implemented earlier.

With that, we've finished coding and the application is ready for testing.

Testing the application

The build system takes the Wi-Fi SSID and password from the environment variables. We provide them before building the project as follows:

```
$ WIFI_SSID=\"your_ssid\" WIFI_PWD=\"your_password\'
$ idf.py erase-flash flash monitor
Executing action: erase-flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<logs removed>
I (1325) esp_netif_handlers: sta ip: 192.168.1.159,
I (1325) app: >>>>>>> connected
```

As we can see, the devkit has got an IP from the router and the callback function is run, printing a log message on the serial console. We can ping the devkit to make sure it is on the network:

```
$ ping 192.168.1.159
PING 192.168.1.159 (192.168.1.159) 56(84) bytes of c
64 bytes from 192.168.1.159: icmp_seq=1 ttl=255 time
64 bytes from 192.168.1.159: icmp_seq=2 ttl=255 time
```

It is truly on the network and replying to the ping packets.

Troubleshooting

If the devkit doesn't connect to the local Wi-Fi network, we can check several points:

- First of all, make sure the SSID and password are defined as environment variables and correct. If you don't define them in the IDF compilation environment, the compiler will give errors, saying that the definitions are not found as needed in the code. If the Wi-Fi credentials are not correct, the application will compile and start successfully. Nonetheless, it will fail to connect to the Wi-Fi network.
- Check whether the router has MAC-filtering enabled. If so, you can disable it temporarily to test the application or add the devkit's MAC address to the white-list of the router. (The Wi-Fi interface MAC is also printed in the serial logs by default.)
- Check whether the router has the WPA2/PSK authentication enabled.

If none of them works, you can reboot the router and then the devkit to see if it makes any difference.

In the next topic, we will discuss Wi-Fi provisioning.

Provisioning ESP32 on a Wi-Fi network

Provisioning is a standard feature in almost every IoT solution. It is usually the first thing to do during the installation of an IoT device, and for IoT devices, the easier the installation, the happier the user. The key issue here is how to share the credentials of a Wi-Fi network

in a secure way. Luckily, ESP-IDF supports several methods for Wi-Fi provisioning:

- **Unified provisioning** from Espressif: With this provisioning method, we can select BLE or Wi-Fi as the transport layer to share the credentials of the Wi-Fi network to be joined. If BLE is selected, the device to be provisioned runs a GATT server to receive the credentials. When Wi-Fi SoftAP is the transport scheme, the device starts a **Soft Access Point** (**softAP**) and HTTP server to receive the credentials. A client application, e.g., a mobile application, connects to the device and shares the credentials.
- **SmartConfig** from Texas Instruments: This provisioning method is also included in ESP-IDF. The solution requires a Wi-Fi-enabled credentials source, such as a mobile phone or a laptop. The device to be provisioned scans all the Wi-Fi channels to find the credentials source and then receives the Wi-Fi credentials from it.
- **Easy Connect** from Wi-Fi Alliance: Also known as a **Device Provisioning Protocol** (**DPP**), Easy Connect works in a similar fashion to **Unified Provisioning**. It requires a mobile application to pass the Wi-Fi credentials to the device by using a QR code or **Near-Field Communication** (**NFC**).

In this example, we will develop several classes to manage the Wi-Fi operations for BLE and Wi-Fi provisioning if the device is not provisioned on a Wi-Fi network. The library that we are going to employ is **Unified Provisioning** from Espressif. We will need a mobile application to pass the Wi-Fi credentials to ESP32. It is **ESP**

SoftAP Provisioning from Espressif, which is available for both Android and iOS in their respective application repositories. The application needs a QR code to connect to a target device, so we will use the QR code library from the *IDF Component Registry*. We will run the example on ESP32-C3-DevKitM-1. Let's create a project for it next.

Creating a project

We can prepare an ESP-IDF project as follows:

1. Create an ESP-IDF project:

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project prov_ex
```

2. Copy the `sdkconfig.defaults` file from the project repository (`ch6/prov_ex`) into the project root. It contains the hardware definition (chip type and flash size) as well as the Bluetooth configuration for BLE provisioning.
3. Add the QR code library to the project:

```
$ idf.py add-dependency "espressif/qrcode==0.1.0
```

4. Add the nlohmann-json library to the project (`main/json.hpp`) by copying it from the repository.
5. We will control the active provisioning method from *Kconfig*. Create `main/Kconfig.projbuild` with the following content:

```
menu "Application settings"
    choice PROV_METHOD
        prompt "Provisioning method"
        default PROV_METHOD_SOFTAP
        help
            Provisioning method to add a WiFi ne
        config PROV_METHOD_SOFTAP
            bool "softap"
        config PROV_METHOD_BLE
            bool "ble"
    endchoice

    config PROV_METHOD
        int
        default 1 if PROV_METHOD_BLE
        default 0
endmenu
```

6. Try building the project and see whether everything is fine. Then you can rename the `managed_components` directory to `components` in the project root. This will prevent some warnings later in the project.
7. Rename the application source code to `main/prov_ex.cpp` and update the `main/CMakeLists.txt` file as follows:

```
idf_component_register(SRCS "prov_ex.cpp" INCLUD
```

We now have the project ready to develop. Let's code next.

Coding the application

We can begin with adding a new file, `main/AppWifi.hpp`, to manage the Wi-Fi events. It is quite similar to the Wi-Fi management class of the previous Wi-Fi example. The difference is that the class in this implementation is an abstract class where the `connect` function is virtual because of the different provisioning methods the derived classes need to implement. Here is the code for this base class:

```
#pragma once
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "nvs_flash.h"
#include "wifi_provisioning/manager.h"
#include "qrcode.h"
#include "json.hpp"
#define PROV_SERVICE_NAME "PROV_ESP32"
#define PROV_POP "abcd1234"
```

The header file for **Unified Provisioning** is `wifi_provisioning/manager.h`. We have two macro definitions that we will need for provisioning. We will discuss them in their specific places. Next come the callback types for the Wi-Fi events in the `app` namespace:

```
namespace app
{
    using OnWifiConnected_f = std::function<void(
        esp_ip4_addr_t *)>;
    using OnWifiDisconnected_f = std::function<void(
```

These are the callback types for the Wi-Fi-connected and Wi-Fi-disconnected events. They will be passed as parameters when initializing a class instance. Then we have the definition of the `AppWifi` class:

```
    class AppWifi
    {
    protected:
        OnWifiConnected_f m_connected;
        OnWifiDisconnected_f m_disconnected;
        static void handleWifiEvent(void *arg, esp_e
            event_base, int32_t event_id, void *even
        static void printQrCode(const char *transpor
```

In the `protected` section, we have two member variables for the Wi-Fi event callbacks of the `AppWifi` class, `m_connected` and `m_disconnected`. The `handleWifiEvent` function is the callback for the underlying Wi-Fi events that come from the ESP-IDF Wi-Fi library. The `printQrCode` function is the interesting one here. We will encode the provisioning data into a QR code to be used by the provisioning mobile application, *ESP SoftAP Provisioning*. Since

ESP32-C3-DevKitM-1 doesn't have a display, we will use the serial console for the QR code output. In real life, you can use an LCD screen if your device has one or print the QR code on the device enclosure for those without a display. The implementation of the function bodies will be after the class definition. The `protected` section is done; we'll continue with the `public` section next:

```
    public:
        void init(OnWifiConnected_f conn,
            OnWifiDisconnected_f disc)
        {
            m_connected = conn;
            m_disconnected = disc;
            esp_err_t err = nvs_flash_init();
            if (err == ESP_ERR_NVS_NO_FREE_PAGES ||
                ESP_ERR_NVS_NEW_VERSION_FOUND)
            {
                nvs_flash_erase();
                nvs_flash_init();
            }
            esp_netif_init();
            esp_event_loop_create_default();
            esp_event_handler_register(WIFI_EVENT,
                ESP_EVENT_ANY_ID, handleWifiEvent, t
            esp_event_handler_register(IP_EVENT,
                IP_EVENT_STA_GOT_IP, handleWifiEvent
            esp_netif_create_default_wifi_sta();
            wifi_init_config_t cfg = WIFI_INIT_CONF
```

```
            esp_wifi_init(&cfg);
        }
```

The `init` function in this implementation is the same as in the previous Wi-Fi example. As a quick recap, we first initialize the flash **Non-Volatile Storage** (**NVS**), create the default event loop to catch the Wi-Fi events, and initialize the Wi-Fi library with the default configuration that comes with ESP-IDF. The Wi-Fi events will be handled by the `handleWifiEvent` member function. We complete the class definition with the `connect` pure virtual function declaration:

```
        virtual void connect(void) = 0;
    };
```

The `connect` function will be implemented by the derived classes with BLE and SoftAP provisioning techniques.

The class definition is finished, but we need to implement two functions that we have declared in the `protected` section of the class. Next comes the `handleWifiEvent` function body:

```
    void AppWifi::handleWifiEvent(void *arg, esp_eve
        event_base, int32_t event_id, void *event_da
    {
        AppWifi *obj = reinterpret_cast<AppWifi *>(a
        if (event_base == WIFI_EVENT && event_id ==
            WIFI_EVENT_STA_START)
```

```
        {
            esp_wifi_connect();
        }
        else if (event_base == WIFI_EVENT && event_i
            WIFI_EVENT_STA_DISCONNECTED)
        {
            obj->m_disconnected();
            vTaskDelay(pdMS_TO_TICKS(3000));
            esp_wifi_connect();
        }
        else if (event_base == IP_EVENT && event_id
            IP_EVENT_STA_GOT_IP)
        {
            ip_event_got_ip_t *event = reinterpret_c
                ip_event_got_ip_t *>(event_data);
            obj->m_connected(&event->ip_info.ip);
        }
    }
```

Again, it is the same as in the previous Wi-Fi-connect example. We try to connect to Wi-Fi by calling the `esp_wifi_connect` function and run the member callbacks according to the event. The last function in this C++ header file is `printQrCode`:

```
    void AppWifi::printQrCode(const char *transport_
    {
        nlohmann::json payload{{"ver", "v1"},
                               {"name", PROV_SERVICE
                               {"pop", PROV_POP},
```

```
                                        {"transport", transpo
        esp_qrcode_config_t cfg = ESP_QRCODE_CONFIG_
        esp_qrcode_generate(&cfg, payload.dump().c_s
    } // end of function
} // end of namespace
```

In the `printQrCode` function, we define the provisioning data as a JSON payload. The service name will be the access-point SSID if the transport method is SoftAP or the GATT service name if the transport method is BLE. **POP** is short for **Proof of Possession**. It authorizes the provisioning session and is used to create a shared key between the provisioning application and the device to protect the Wi-Fi credentials to be passed to the device. Then we generate and print a QR code on the serial console by calling the `esp_qrcode_generate` function. The `printQrCode` function is a utility function for the classes derived from the `AppWifi` class and will be called if the device is not provisioned on a Wi-Fi network.

We have the `AppWifi` base class now and we can develop another two classes for BLE and SoftAP provisioning by deriving from it. Let's implement the SoftAP provisioning first, in the `main/AppWifiSoftAp.hpp` file:

```
#pragma once
#include "AppWifi.hpp"
#include "wifi_provisioning/scheme_softap.h"
#define PROV_TRANSPORT_SOFTAP "softap"
```

The SoftAP provisioning method, or scheme as it's called in ESP-IDF, is available by including `wifi_provisioning/scheme_softap.h`. Then we define the `AppWifiSoftAp` class as follows:

```
namespace app
{
    class AppWifiSoftAp : public AppWifi
    {
    public:
        void connect(void) override
        {
            wifi_prov_mgr_config_t config = {
                .scheme = wifi_prov_scheme_softap,
                .scheme_event_handler = WIFI_PROV_E\
            };
            wifi_prov_mgr_init(config);
```

In the `AppWifiSoftAp` class definition, we only need to implement the `connect` function that is declared as virtual in the base class. We select the SoftAP scheme in the configuration and initialize the provisioning manager with this configuration. After that, we can check whether the device is already provisioned:

```
            bool provisioned = false;
            wifi_prov_mgr_is_provisioned(&provisione
```

We call the `wifi_prov_mgr_is_provisioned` function to get the device provisioning status. A local variable is passed by reference to this function and it is updated with the provisioning status. In the next step, we test this variable and act accordingly:

```
            if (provisioned)
            {
                wifi_prov_mgr_deinit();
                esp_wifi_set_mode(WIFI_MODE_STA);
                esp_wifi_start();
            }
```

If the device is already provisioned, i.e., the NVS partition contains a valid Wi-Fi credentials record, we simply free the provisioning resources by calling the `wifi_prov_mgr_deinit` function and start Wi-Fi in STA mode. We handle the not-provisioned case as follows:

```
            else
            {
                esp_netif_create_default_wifi_ap();
                wifi_prov_mgr_start_provisioning(WI
                printQrCode(PROV_TRANSPORT_SOFTAP);
                wifi_prov_mgr_wait();
                wifi_prov_mgr_deinit();
            } // end of else
        } // end of function
    }; // end of class
} // end of namespace
```

We first create an access point on ESP32 so that other devices, such as a mobile device for provisioning, can connect in STA mode. Then we start the provisioning manager. Since we initialized it with the SoftAP scheme, it will wait on the Wi-Fi interface for any provisioning clients. One important point to mention here is that the first parameter of the `wifi_prov_mgr_start_provisioning` function denotes the security type to communicate with the provisioning clients. ESP-IDF has three options here:

- **Security-0**: No security at all. All packets are exchanged in plain text.
- **Security-1**: The most common security model. In the session setup, a shared key is generated and this key is used for further communication. As extra security, **POP** can be used.
- **Security-2**: The developer needs to provide underlying secrets and set up the session accordingly.

In this example, we use the `WIFI_PROV_SECURITY_1` type of security with a POP string. After starting the provisioning manager, we print the QR code on the console for the SoftAP provisioning. The mobile application will scan this code to find the device and connect to it. During the provisioning, we block the code by calling `wifi_prov_mgr_wait`. Nonetheless, it is quite possible to write event-driven provisioning by providing a callback function to handle the provisioning events. The SoftAP provisioning is ready and we can develop the BLE provisioning in `main/AppWifiBle.hpp`:

```
#pragma once
#include "AppWifi.hpp"
#include "wifi_provisioning/scheme_ble.h"
#define PROV_TRANSPORT_BLE "ble"
```

The header file for BLE provisioning is `wifi_provisioning/scheme_ble.h`. The class implementation is very similar to the SoftAP provisioning, as comes next:

```
namespace app
{
    class AppWifiBle : public AppWifi
    {
    public:
        void connect(void) override
        {
            wifi_prov_mgr_config_t config = {
                .scheme = wifi_prov_scheme_ble,
                .scheme_event_handler =
                    WIFI_PROV_EVENT_HANDLER_NONE,
            };
            wifi_prov_mgr_init(config);
            bool provisioned = false;
            wifi_prov_mgr_is_provisioned(&provisione
            if (provisioned)
            {
                wifi_prov_mgr_deinit();
                esp_wifi_set_mode(WIFI_MODE_STA);
```

```
                esp_wifi_start();
            }
```

This time, we initialize the provisioning manager for BLE and check whether the device is already provisioned. If so, we simply start the Wi-Fi connection to the configured Wi-Fi network. If the device is not provisioned to any Wi-Fi network, we need to start the provisioning process as follows:

```
            else
            {
                uint8_t custom_service_uuid[] = {0xb
                    0x1c, 0x3f, 0x6b, 0xf4, 0xbf, 0x
                    0x82, 0x03, 0x04, 0x90, 0x1a, 0x
                wifi_prov_scheme_ble_set_service_uui
                    custom_service_uuid);
                wifi_prov_mgr_start_provisioning(
                    WIFI_PROV_SECURITY_1, PROV_POP,
                   PROV_SERVICE_NAME, nullptr);
                printQrCode(PROV_TRANSPORT_BLE);
                wifi_prov_mgr_wait();
                wifi_prov_mgr_deinit();
            }
        } // end of function
    }; // end of class
} // end of namespace
```

After setting the provisioning service UUID (basically a BLE GATT service), we start provisioning as we did for the SoftAP case. The QR code printed on the screen is for BLE provisioning and contains information about the BLE service.

The Wi-Fi classes with provisioning are completed and we can use them in the main application, `main/prov_ex.cpp`:

```
#include "esp_log.h"
#include "AppWifiSoftAp.hpp"
#include "AppWifiBle.hpp"
#define TAG "app"
namespace
{
#if CONFIG_PROV_METHOD == 0
    app::AppWifiSoftAp *app_wifi = new app::AppWifiS
#else
    app::AppWifiBle *app_wifi = new app::AppWifiBle(
#endif
}
```

According to the application configuration, we select either `AppWifiSoftAp` or `AppWifiBle` as the Wi-Fi instance of the application. Next, we will use the selected one in the `app_main` function:

```
extern "C" void app_main(void)
{
    auto connected = [](esp_ip4_addr_t *ip)
```

```
    {
        ESP_LOGI(TAG, ">>>>>>> connected");
    };
    auto disconnected = []()
    {
        ESP_LOGW(TAG, ">>>>>>> disconnected");
    };
    app_wifi->init(connected, disconnected);
    app_wifi->connect();
}
```

We initialize the `app_wifi` object with the callbacks and call its `connect` function, as we implemented for different types of provisioning. This completes the coding work of the project and we can test it by flashing the application.

Testing application

We flash and test the application as follows:

```
$ idf.py erase-flash flash monitor
Executing action: erase-flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<logs-removed>
```

Please note that we erase the flash to make sure there is nothing left on the NVS partition so that the provisioning process can start.

At the end of the logs, it prints the QR code for the mobile application that we will use for provisioning:

Figure 6.2: SoftAP provisioning

At this point, we start the *ESP SoftAP Prov* application on a mobile device. When we scan the QR code on the serial console, the mobile application automatically finds the SoftAP that runs on ESP32:

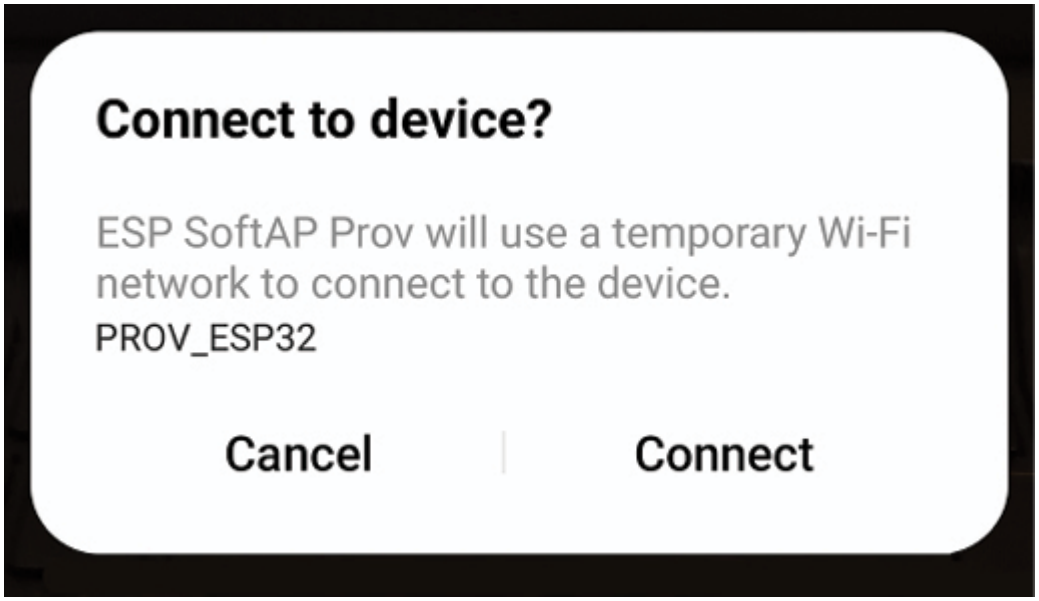


Figure 6.3:ESP32 SoftAP

We press the **Connect** button, then the mobile application asks for the Wi-Fi network that we want to join. We provide the Wi-Fi credentials and a success screen is displayed:

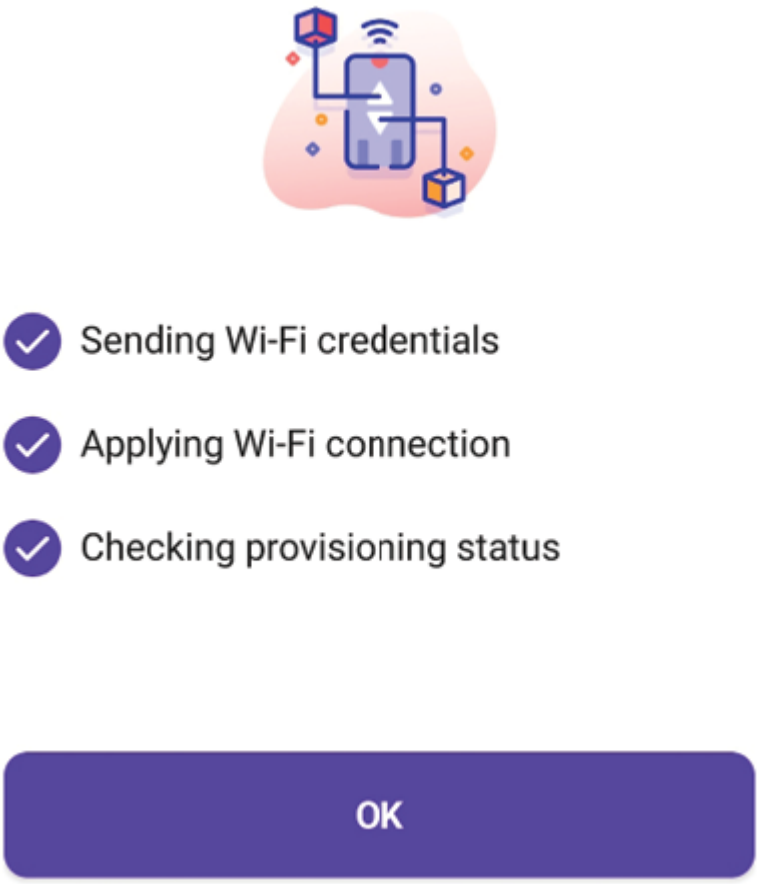


Figure 6.4: Wi-Fi provisioning progress

We also see the success message on the serial console of ESP32:

```
<logs removed>
I (307382) app: >>>>>>> connected
I (307382) esp_netif_handlers: sta ip: 192.168.1.15
I (307382) wifi_prov_mgr: STA Got IP
I (311702) wifi:station: 9e:a8:cb:8d:95:72 leave, A
I (311702) wifi:new:<11,0>, old:<11,2>, ap:<11,2>,
W (311702) wifi:<ba-del>idx
```

```
I (312622) wifi:mode : sta (a0:76:4e:76:f9:0c)
I (312632) wifi_prov_mgr: Provisioning stopped
```

To configure the application for BLE provisioning, we run `menuconfig` and set BLE as the active provisioning method:

```
$ idf.py menuconfig
```

The following figure shows this setting:

Figure 6.5: Setting BLE as the provisioning method

The remaining steps for testing BLE provisioning are the same as the Wi-Fi SoftAP provisioning, thus there is no need to repeat them here. Please remember to erase the flash to force the provisioning process to start.

In production, it is probably best to use your own mobile application for provisioning. Espressif provides the source code of the mobile applications as examples on GitHub; for Android – https://github.com/espressif/esp-idf-provisioning-android and for iOS – https://github.com/espressif/esp-idf-provisioning-ios.

Troubleshooting

In addition to the previous troubleshooting items in the Wi-Fi connection example, we might experience issues with the mobile application connecting to the devkit. If this happens, you can try scanning the QR code a few more times. The logs on the serial console can give more information about the reason for any failure.

After connecting ESP32 to the local Wi-Fi network, we have different options in the application layer to communicate with remote servers and systems. We can develop our own application-layer protocols but there is no need to reinvent the wheel. In the upcoming topics, we will discuss some prominent IoT protocols to exchange data with remote systems.

Communicating over MQTT

Message Queue Telemetry Transport (**MQTT**) is a many-to-many communication protocol with a message broker as the mediator. There are publishers that send messages to the topics on the broker, and there are subscribers that receive messages from the topics that they subscribe to. A node can be a publisher and subscriber at the same time:

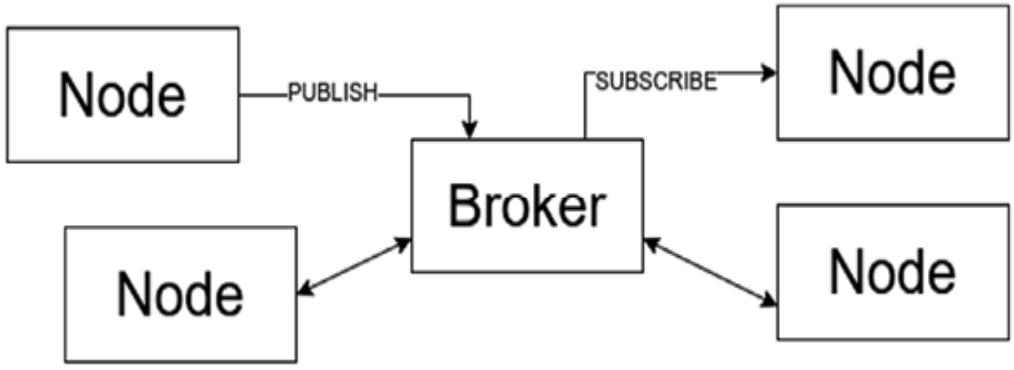


Figure 6.6: MQTT communication model

For MQTT, TCP is the underlying transport protocol and TLS can be configured for communication security.

Let's see how MQTT works with a simple example. The goal in this example is to publish sensor data to an MQTT broker running in the local network and control the sensor features remotely by connecting to the broker with an MQTT client running on another machine. As listed in *Technical requirements*, the broker is **Mosquitto**. We begin with installing the broker.

Installing the MQTT broker

The `mosquitto` installation package comes with an MQTT broker, a publisher client (`mosquitto_pub`), and a subscriber client (`mosquitto_sub`). They are all command-line tools. Please install *Mosquitto* on your development machine as described in its documentation and make sure the broker and clients work properly. We will use the broker for plain MQTT communication (no TLS security) with a username and password to connect to it. The commands to install and configure *Mosquitto* on a local machine

differ a little bit depending on the platform but you can see the following steps for the Ubuntu installation to get an idea:

1. Install *Mosquitto*:

```
$ sudo apt install mosquitto
```

2. Update its configuration file (`default.conf`) as follows:

```
$ sudo vi /etc/mosquitto/conf.d/default.conf
listener 1883
allow_anonymous false
password_file /etc/mosquitto/passwd
```

3. Create a user:

```
$ sudo mosquitto_passwd -c /etc/mosquitto/passwd
```

4. Start the broker service:

```
$ sudo systemctl start mosquitto
```

5. If you are running a firewall, allow the MQTT port for external connections:

```
$ sudo ufw allow 1883
```

6. After the installation, you can test the connection with the following simple command:

```
$ mosquitto_sub -h <broker_ip> -t '#' -u <user_n
```

These commands are for Ubuntu Linux, but the logic applies to other platforms as well.

You can find many interesting tutorials about MQTT and *mosquitto* here: http://www.steves-internet-guide.com/. Configuring the broker with TLS encryption and using client certificates with MQTT are two important topics that you will find tutorials about on this website.

The devkit of this example is again ESP32-C3-DevKitM-1. We will use the button of the devkit as a sensor that toggles a Boolean value (on/off). We will publish this value to a topic on the broker. The devkit also has an integrated RGB LED. We will control it over MQTT.

Creating a project

With the broker and tools installed, we can create the project with the following steps:

1. Create an ESP-IDF project from the command line (you can choose any other way described in *Chapter 2, Understanding the Development Tools*).

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project mqtt_ex
```

2. Copy the `sdkconfig.defaults` file from the project repository (`ch6/mqtt_ex`) into the project root. It contains the hardware definition (chip type and other configuration settings).
3. We need to increase the partition size for the application. Copy the `partitions.csv` file from the repository (`ch6/mqtt_ex`) into the project root.
4. We want to keep the MQTT username and password outside the code for security reasons. *cmake* will read them from the environment and pass them to the compiler as preprocessor macros. The content of the `CMakeLists.txt` file in the project root is:

```
cmake_minimum_required(VERSION 3.5)
set(EXTRA_COMPONENT_DIRS ../common)
if(DEFINED ENV{MQTT_USER})
  add_compile_options(-DMQTT_USER=$ENV{MQTT_USER
else(DEFINED ENV{MQTT_USER})
  message(FATAL_ERROR "MQTT username is needed")
endif(DEFINED ENV{MQTT_USER})
if(DEFINED ENV{MQTT_PWD})
  add_compile_options(-DMQTT_PWD=$ENV{MQTT_PWD})
else(DEFINED ENV{MQTT_USER})
  message(FATAL_ERROR "MQTT password is needed")
endif(DEFINED ENV{MQTT_PWD})
```

```
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(mqtt_ex)
```

5. There a `common` library directory for this chapter in the GitHub repository. Copy this to the one-up directory of the project as well (or anywhere you like, but don't forget to update the root `CMakeLists.txt` file accordingly).
6. Define the environment variables for the MQTT username and password (different OSs/shells have their own syntax):

```
$ export MQTT_USER=\"<username>\"
$ export MQTT_PWD=\"<password>\"
```

7. We want the MQTT broker IP, port, and MQTT client identifier to be configurable. Copy the `main/Kconfig.projbuild` file from the repository (`ch6/mqtt_ex`) into the project.
8. Compile the application to make sure there are no errors up to this point:

```
$ idf.py build
```

9. Rename the default source code file to `main/mqtt_ex.cpp` and update the content of the `main/CMakeLists.txt` file as follows:

```
idf_component_register(SRCS "mqtt_ex.cpp" INCLUD
```

The project is configured now and we can move on to coding the application.

Coding the application

Let's start with a base class, `main/AppSensor.hpp`, for our sensor. This base class will define some basic features of any sensor that we can develop with the devkit:

```
#pragma once
#include <string>
#include "json.hpp"
namespace app
{
    class AppSensor
    {
    protected:
        std::string m_sensor_name;
        bool m_enabled;
        nlohmann::json m_state;
        virtual void handleNewState(void) = 0;
```

A sensor that derives from the `AppSensor` class will have a name and a property that shows whether the sensor is enabled or not. It also has a JSON member variable, `m_state`, to show all properties in a single variable (including its name and enabled state). This will be published on a topic of the broker if the sensor is enabled. There is also a pure virtual function, `handleNewState`, to be implemented by derived classes. When the state of the sensor is changed by a

message to a topic of the broker, this function will be called to allow the derived class to update its internal variables. We continue with the `public` section:

```
    public:
        AppSensor(std::string name) : m_sensor_name(
            m_enabled(false)
        {
            m_state["enabled"] = false;
        }
        virtual ~AppSensor() {}
        virtual void init(void) = 0;
```

After the constructor and destructor, there is another virtual function, `init`, to be implemented by derived classes. This function is designated to allow a derived class to initialize the hardware and any other member variables. The next two functions are the getters of the member variables in `AppSensor`:

```
        std::string getName(void) const { return m_s
        bool isEnabled(void) const { return m_enable
```

The getter functions, `getName` and `isEnabled`, simply return the `m_sensor_name` and `m_enabled` member variables of the base class respectively. We finalize the class as follows:

```
        std::string getState(void) const { return m_

        void setState(std::string new_state)
        {
            nlohmann::json val = nlohmann::json::par
                nullptr, false);
            if (!val.is_discarded())
            {
                m_state.merge_patch(val);
                m_enabled = m_state["enabled"].get<b
                handleNewState();
            }
        } // function end
    }; // class end
} // namespace end
```

The `getState` function returns a string representation of the `m_state` JSON value. The `setState` function updates `m_state` with the parameter value. This function will be called when a new target state is received from the broker. In this way, we can enable/disable the sensor. It also calls the `handleNewState` function that is to be implemented by derived classes.

Let's add a new class in the `main/AppSensorDevkit.hpp` file to implement a concrete sensor:

```
#pragma once
#include "AppSensor.hpp"
#include "AppEsp32C3Bsp.hpp"
```

```
#include "sdkconfig.h"
namespace app
{
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(BspLed_t, sta
        saturation, brightness);
```

We first include the header files. The `AppEsp32C3Bsp.hpp` header file defines some structures and a class to help with our examples. It abstracts the details of the original devkit BSP. We will discuss its definitions in place along with the code. The `BspLed_t` type is one of them. It describes the RGB LED on the devkit. With the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro, we have the JSON conversion functions from/to the `BspLed_t` type. Next, we define the sensor class:

```
    class AppSensorDevkit : public AppSensor
    {
    private:
        AppEsp32C3Bsp m_bsp;
        bool m_toggle_btn;
```

The `AppSensorDevkit` class derives from `AppSensor`. Inside the `private` section, we define an instance of `AppEsp32C3Bsp`, which comes from the `AppEsp32C3Bsp.hpp` header file. It provides access to the underlying hardware. We also define a member variable for the conceptual toggle button and we will toggle its value every time

the physical button of the devkit is pressed. The next `static` function does this:

```
        static void handleBtn(void *arg)
        {
            AppSensorDevkit *obj = reinterpret_cast<
                AppSensorDevkit *>(arg);
            obj->m_toggle_btn = !obj->m_toggle_btn;
            obj->m_state["button"] = obj->m_toggle_b
                "off";
        }
```

The `handleBtn` function is a callback function for the button-pressed event. It will be called when the physical button of the devkit is pressed. The `m_state` member variable of the instance, which is defined in the base class, is also updated. We continue with the `protected` section of the class:

```
    protected:
        void handleNewState(void) override
        {
            m_state["button"] = m_toggle_btn ? "on"
            m_bsp.setLed(m_state["led"]);
            m_state["led"] = m_bsp.getLed();
        }
```

The `protected` section has a single function, which is the implementation of the `handleNewState` virtual function. The button

is a sensor and we don't want its value to be overwritten by an MQTT message, so we ensure that it shows the real state of the toggle button. However, the MQTT message can update the LED. It can turn it on or off, or change the color values. We pass the incoming value of the LED to the `m_bsp` object by calling its `setLed` function. The only thing to be careful about is the value ranges of the LED. The `setLed` function respects the value ranges and sets the internal values accordingly. That is why we update `m_state["led"]` by calling the `getLed` function. Please note that the conversion between JSON and `BspLed_t` is done automatically in both ways, thanks to the macro that we added before the class definition. The `public` section comes next:

```
    public:
        AppSensorDevkit() : AppSensor(
            CONFIG_MQTT_CLIENT_IDENTIFIER), m_toggle
        {
            m_state = {{"button", "off"},
                {"led", m_bsp.getLed()}};
        }
```

The constructor initializes the member variables with the default values. Then we implement another virtual function of the base class:

```
        void init(void) override
        {
            m_bsp.init();
```

```
            auto fn = [](void *arg)
            {
                handleBtn(arg);
            };
            m_bsp.addButtonHandler(button_cb_type_t
                BUTTON_CB_RELEASE, fn, this);
        } // function end
    }; // class end
} // namespace end
```

In the `init` function, we first initialize the hardware and then attach the button callback to the `m_bsp` object. This completes the concrete sensor implementation and we can move on to the MQTT handling in `main/AppMqtt.hpp`:

```
#pragma once
#include <functional>
#include <string>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "mqtt_client.h"
#include "sdkconfig.h"
#include "AppSensor.hpp"
```

We start with the header files as usual. The interesting one here is `mqtt_client.h`, which comes with ESP-IDF and has the MQTT client definitions, hence the name. We will use it to communicate

with the MQTT broker. Next, we will define our own MQTT event structure to expose some of the MQTT events outside the class:

```
namespace app
{
    typedef struct
    {
        esp_mqtt_event_id_t id;
        void *data;
    } MqttEventData_t;
    using MqttEventCb_f = std::function<void(MqttEve
```

The `MqttEventData_t` structure shows the event and has a pointer for the event data. Callbacks for the MQTT events take this structure as a parameter. Then we continue with the MQTT class implementation:

```
    class AppMqtt
    {
    private:
        AppSensor *m_sensor;
        esp_mqtt_client_handle_t m_client = nullptr;
        std::string m_sensor_topic;
        TaskHandle_t m_publish_handle = nullptr;
        MqttEventCb_f m_event_cb;
```

The `AppMqtt` class will behave as a bridge between the sensor and the MQTT broker. Therefore, we keep a pointer to the sensor and an

MQTT client handle of the `esp_mqtt_client_handle_t` type. The `m_sensor_topic` member variable is an MQTT topic that we need to subscribe to in order to receive data from the MQTT broker. At the same time, we will use this topic to publish the state of the sensor. We will publish periodic data to the MQTT broker and create a FreeRTOS task for this purpose. The last member variable is the callback function to be run when an MQTT event occurs. Then we have private functions:

```
        void handleMqttData(esp_mqtt_event_handle_t
        static void mqttEventHandler(void *arg,
                                     esp_event_base_
                                     int32_t event_i
                                     void *event_dat
        static void publishSensorState(void *param);
```

The `handleMqttData` function will be called by the `mqttEventHandler` function when data comes to the sensor topic. The latter is the event callback of the MQTT clients. The `publishSensorState` function is the FreeRTOS task function that will publish the sensor data periodically. Next comes the `public` section of the class implementation:

```
    public:
        AppMqtt(AppSensor *sensor) : m_sensor(sensor
            m_sensor_topic(sensor->getName() + "/sta
```

```
{
}
```

The constructor takes a sensor pointer as its parameter and sets the private `m_sensor` member variable with this value. It also initializes the sensor topic name by concatenating the sensor name and `"/state"`. We will implement the `init` function where we set up the MQTT client and FreeRTOS task next:

```
void init(MqttEventCb_f cb)
{
    m_event_cb = cb;
    esp_mqtt_client_config_t mqtt_cfg = {
        .host = CONFIG_MQTT_BROKER_IP,
        .port = CONFIG_MQTT_PORT,
        .client_id = m_sensor->getName().c_s
        .username = MQTT_USER,
        .password = MQTT_PWD};
```

The `init` function takes a function as its parameter for exposing MQTT events to the external code. We also define a variable for MQTT configuration. In this configuration, we set the MQTT IP and port, client identifier, and username and password to connect to the MQTT broker. This will be needed when initializing the MQTT client as we do next:

```
            m_client = esp_mqtt_client_init(&mqtt_c
            esp_mqtt_client_register_event(m_client,
                mqttEventHandler, this);
```

The `esp_mqtt_client_init` function takes the configuration parameters as its input and returns the MQTT client. We will access the MQTT communication functionality over `m_client`. We also attach the MQTT event handler to the MQTT client by calling `esp_mqtt_client_register_event`. We will complete the `init` function by creating the FreeRTOS task:

```
            xTaskCreate(publishSensorState, "publish
                5, &m_publish_handle);
            vTaskSuspend(m_publish_handle);
        }
```

We create a FreeRTOS task with the `publishSensorState` function and suspend the task immediately. It will be resumed when the MQTT broker is connected. We have one last function before closing the class definition:

```
        void start(void)
        {
            esp_mqtt_client_start(m_client);
        } // function end
    }; // class end
```

The `start` function will be used as the Wi-Fi-connected event handler and start the MQTT client. The call to the `esp_mqtt_client_start` function will trigger the MQTT connection process. The class definition has ended, but we are not done with the implementation in this file yet. Next comes the `mqttEventHandler` function body:

```
void AppMqtt::mqttEventHandler(void *arg,
                               esp_event_base_t
                               int32_t event_id,
                               void *event_data)
{
    AppMqtt *obj = reinterpret_cast<AppMqtt *>(a
    switch (static_cast<esp_mqtt_event_id_t>(eve
    {
    case MQTT_EVENT_CONNECTED:
        esp_mqtt_client_subscribe(obj->m_client,
            obj->m_sensor_topic.c_str(), 1);
        vTaskResume(obj->m_publish_handle);
        break;
```

In the `mqttEventHandler` function, we handle MQTT events, hence the name. The first event is when the client connects to the MQTT broker. When connected, we subscribe to the sensor topic to be able to receive the incoming messages and also resume the FreeRTOS task to send periodic sensor data to the broker. The next event is `MQTT_EVENT_DATA`:

```
        case MQTT_EVENT_DATA:
            obj->handleMqttData(reinterpret_cast<
                esp_mqtt_event_handle_t>(event_data)
            break;
```

When new data comes over MQTT, we call the `handleMqttData` of the `AppMqtt` instance. The last case handles the MQTT broker disconnected event:

```
        case MQTT_EVENT_DISCONNECTED:
            vTaskSuspend(obj->m_publish_handle);
            break;
        default:
            break;
        }
        obj->m_event_cb( {static_cast<esp_mqtt_event
            event_id), event_data});
    }
```

When the broker is disconnected, we suspend the FreeRTOS task to prevent it from publishing data since there is no active connection anymore. Before exiting the `mqttEventHandler` function, we call the registered callback function of the instance during initialization. We continue with the handling of the incoming data as implemented by the `handleMqttData` function:

```
    void AppMqtt::handleMqttData(esp_mqtt_event_hand
    {
        std::string data{event->data, (size_t)event·
        m_sensor->setState(data);
    }
```

In the `handleMqttData` function, we extract the data from the `event` parameter and send it to the sensor for processing. The final function is for the FreeRTOS task, as follows:

```
    void AppMqtt::publishSensorState(void *param)
    {
        AppMqtt *obj = reinterpret_cast<AppMqtt *>(p
        while (true)
        {
            vTaskDelay(pdMS_TO_TICKS(3000));
            if (obj->m_sensor->isEnabled())
            {
                std::string state_text = obj->
                    m_sensor->getState();
                esp_mqtt_client_publish(obj->m_clien
                    m_sensor_topic.c_str(), state_te
                    state_text.length(), 1, 0);
            }
        }
    } // function end
} // namespace end
```

The `publishSensorState` function has a loop inside it where it publishes the sensor data every three seconds. Since it is a FreeRTOS task function, it won't return. The task will be paused or resumed depending on the MQTT broker connection state. To publish data to an MQTT topic, we use the `esp_mqtt_client_publish` function. The last two parameters of this function show the QoS level and the `retain` flag of the MQTT message. Here, the QoS level is 1, which guarantees the delivery of the message. The `retain` flag is 0 – false. Therefore, the broker won't store the last message for any newly connected clients.

We have all the components necessary to write the application. Now it is time to integrate them in the main application source code, `main/mqtt_ex.cpp`:

```
#include "esp_log.h"
#include "AppWifiSoftAp.hpp"
#include "AppMqtt.hpp"
#include "AppSensorDevkit.hpp"
#define TAG "app"
namespace
{
    app::AppWifiSoftAp app_wifi;
    app::AppSensorDevkit app_sensor;
    app::AppMqtt app_mqtt{&app_sensor};
}
```

After including the header files, we instantiate the classes that we implemented earlier. We have a Wi-Fi client, a sensor object, and an MQTT client. We pass the address of the sensor to the MQTT client

constructor. Next, we implement the application entry point, the `app_main` function:

```
extern "C" void app_main(void)
{
    auto wifi_connected = [](esp_ip4_addr_t *ip)
    {
        ESP_LOGI(TAG, "wifi connected");
        app_mqtt.start();
    };
    auto wifi_disconnected = []()
    {
        ESP_LOGW(TAG, "wifi disconnected");
    };
```

The first two lambda functions in `app_main` are for Wi-Fi events. When the Wi-Fi is connected, we call the `start` function of the `app_mqtt` object. We will have another lambda function for the MQTT event callback:

```
    auto mqtt_cb = [](app::MqttEventData_t event)
    {
        switch (event.id)
        {
        case MQTT_EVENT_ERROR:
            ESP_LOGW(TAG, "mqtt error");
            break;
        case MQTT_EVENT_CONNECTED:
            ESP_LOGI(TAG, "mqtt connected");
            break;
```

```
        case MQTT_EVENT_DISCONNECTED:
            ESP_LOGW(TAG, "mqtt disconnected");
            break;
        default:
            break;
        }
    };
```

We do nothing special in this lambda function but only print what happened to the MQTT connection.

These lambda functions glue the application logic between the objects. In the last lines of the source code, we simply initialize the application objects:

```
    app_sensor.init();
    app_mqtt.init(mqtt_cb);
    app_wifi.init(wifi_connected, wifi_disconnected);
    app_wifi.connect();
} // app_main end
```

We call the `init` functions of the objects and pass the relevant lambda functions as arguments. When we call the `connect` function of the `app_wifi` object, the underlying Wi-Fi library tries to connect to the local Wi-Fi or starts the SoftAP provisioning if Wi-Fi is not configured.

The application is finished and we can test it now after flashing it to the devkit.

Testing the application

Let's test the application in steps:

1. We flash the application:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<logs removed>
I (1403) app: wifi connected
I (1403) esp_netif_handlers: sta ip: 192.168.153
I (1493) app: mqtt connected
```

2. Then, we start an MQTT client to receive from all topics:

```
$ mosquitto_sub -h <broker-ip> -t '#' -u <userna
```

3. When we enable the sensor, it will start to publish data on the topic. Let's enable it by publishing the new state to its topic (`sensor-0/state`):

```
$ mosquitto_pub -h 192.168.153.199 -t sensor-0/s
```

4. As soon as we enable it, it starts to publish its state on the topic and we can see it on the MQTT client that we started in *step 2*:

```
sensor-0/state {"button":"off","enabled":true,"l
"saturation":0,"state":false}}
```

5. We can toggle the button state by pressing the **Boot** button of the devkit and we see it updated on the MQTT client:

```
sensor-0/state {"button":"on","enabled":true,"le
"saturation":0,"state":false}}
```

6. We can also control the LED remotely by publishing on the topic. When we change the LED values, the LED on the devkit will light accordingly:

```
$ mosquitto_pub -h <broker-ip> -t sensor-0/state
"saturation":100,"state":true}}'
```

You can turn on/off the LED and change the color and brightness. You can also try enabling/disabling the sensor to see the effect.

Troubleshooting

Since we are running an external service, the MQTT broker, on another machine, there are more points that we need to visit in case of any issues. They are:

- First and foremost, you need to check whether the MQTT broker is up and running. How you would do this depends on the platform that the broker is running on. There is a good

article about the subject here: http://www.steves-internet-guide.com/mosquitto-broker/

- If you are also running a firewall on the same machine that the MQTT broker is started on, then make sure the MQTT port is open on the firewall.
- Make sure you have configured the correct broker IP and port in the *Kconfig* configuration.
- Double-check whether the correct username and password pair is passed to the environment while compiling the application.

MQTT is quite common as a data exchange protocol in many IoT applications. Therefore, I strongly advise you spend more time on it trying out different scenarios to get more familiar with it. For example, you can connect several clients to the same broker and subscribe to others' topics to receive data from them. In the next topic, we will discuss another popular communication method, REST.

Running a RESTful server on ESP32

Representational State Transfer (**REST**) is basically the client-server architecture for the web. It defines how a client and server communicate over a resource that the server exposes. In fact, we know it from the HTTP protocol as it implements the entire **World Wide Web** (**WWW**) document exchange services. A RESTful server publishes a REST API and clients consume it. A client sends a message, such as *GET*, *POST*, *PUT*, or *DELETE*, to the server for a resource, and the server replies to the client with an HTTP status code, such as *200 OK*, *201 Created*, or *404 Not Found*. There exist

many RESTful services on the internet; therefore, REST communication occupies an important place in IoT development. Mozilla provides many articles about HTTP here: https://developer.mozilla.org/en-US/docs/Web/HTTP

We can employ ESP32 as either a RESTful server or a client. When it is a server, we run an HTTP server on ESP32, define resources, and provide handlers for the HTTP methods on those resources that we design. A client can connect to the ESP32 RESTful service and make requests on the resources. When ESP32 is a client, it sits on the other side of the table and makes requests on the resources that are exposed by a server. We will see examples of both cases.

In this first example of REST, we will implement a server. The goal is to implement a sensor server as a RESTful service. The server will have two resources: for configuration and data. Clients can enable/disable the sensor by updating the configuration resource and get its state by querying the data resource. The devkit is ESP32-C3-DevKitM-1. Let's create an ESP-IDF project and configure it for development.

Creating the project

We can create the project as in the following steps:

1. Create an ESP-IDF project from the command line (you can choose any other way described in *Chapter 2, Understanding the Development Tools*).

```
$ export $HOME/esp/esp-idf/export.sh
```

```
$ idf.py create-project rest_server_ex
```

2. Copy the `sdkconfig.defaults` file from the project repository (`ch6/rest_server_ex`) into the project root. It contains the hardware definition (chip type and flash size).
3. Edit the root `CMakeLists.txt` file with the following content to include the libraries:

```
cmake_minimum_required(VERSION 3.5)
set(EXTRA_COMPONENT_DIRS ../common)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(rest_server_ex)
```

4. Rename the source code file to `main/rest_server_ex.cpp` and update the `main/CMakeLists.txt` file for it:

```
idf_component_register(SRCS "rest_server_ex.cpp"
                    INCLUDE_DIRS ".")
```

With that, we can start to write the code.

Coding the application

Let's begin by creating a REST server in a C++ header file, `main/AppRestServer.hpp`:

```
#pragma once
#include <string>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "json.hpp"
#include "AppEsp32C3Bsp.hpp"
#include "esp_http_server.h"
```

The header file for HTTP servers in ESP-IDF is `esp_http_server.h`. After the header files, we define the resource structures as follows:

```
namespace app
{
    struct sSensorConfig
    {
        std::string name;
        bool enabled;
    };
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(sSensorConfi
        enabled);
    struct sSensorData
    {
        std::string name;
        std::string button;
    };
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(sSensorData,
```

We have two resources. The `sSensorConfig` C structure defines the configuration resource and `sSensorData` is for the data resource of the server. We also add their JSON representations for easy conversion. Then the class definition comes next:

```
class AppRestServer
{
private:
    AppEsp32C3Bsp m_bsp;
    bool m_toggle_btn;
    sSensorConfig m_config;
    sSensorData m_data;
    httpd_handle_t m_server = nullptr;
```

In the `private` section, we define the member variables for the resources and a handle for the RESTful server. The prototypes of the handler functions follow in the `private` section:

```
    static esp_err_t handleDataGet(httpd_req_t *
    static esp_err_t handleConfigPut(httpd_req_t
    static void handleBtn(void *arg);
```

The `handleDataGet` function is for the GET requests to the data resource. Similarly, the `handleConfigPut` function handles the PUT requests to the configuration resource. The last function, `handleBtn`, is the callback for the button press events. We continue with the `init` function in the `public` section of the class:

```
    public:
        void init(void)
        {
            m_config.enabled = true;
            m_config.name = "sensor-0";
            m_toggle_btn = false;
            m_data.name = m_config.name;
            m_data.button = m_toggle_btn ? "on" : "
```

We start with the initialization of the member variables and then initialize the ESP32-C3 BSP as follows:

```
            m_bsp.init();
            auto fn = [](void *arg)
            {
                handleBtn(arg);
            };
            m_bsp.addButtonHandler(button_cb_type_t
                BUTTON_CB_RELEASE, fn, this);
        }
```

After calling the `init` function of the BSP, we attach the button handler function to the BSP so that it can pass the button-released event to the application. The next function is for starting the RESTful server:

```
        void start(void)
        {
```

```
            httpd_config_t config = HTTPD_DEFAULT_C(
            config.stack_size = 6000;
            httpd_start(&m_server, &config);
```

We will start the server when the Wi-Fi is connected by calling the `start` function. In the function body, we first define a default HTTP configuration and update the stack size to `6000`. This value is assigned to the FreeRTOS stack size for the HTTP service behind the scenes. Then we call the `httpd_start` function to actually start the server. Next, we configure the resources:

```
            httpd_uri_t data_ep = {
                .uri = "/data",
                .method = HTTP_GET,
                .handler = handleDataGet,
                .user_ctx = this};
            httpd_register_uri_handler(m_server, &da
```

This setting binds the `AppRestServer` class and the HTTP server. Whenever an `HTTP_GET` request comes to the `/data` URI, the HTTP server will call the `handleDataGet` function of the class. We also pass the `this` pointer so that the `handleDataGet` function can access the runtime instance of the class. We do the same for the configuration resource, as follows:

```
            httpd_uri_t config_ep = {
                .uri = "/config",
```

```
                .method = HTTP_PUT,
                .handler = handleConfigPut,
                .user_ctx = this};
            httpd_register_uri_handler(m_server, &c
        }
```

This time, we handle the PUT requests to the `/config` URI with the `handleConfigPut` function. This completes the `start` function of `AppRestServer`. Next, we define the `stop` function:

```
        void stop(void)
        {
            httpd_stop(m_server);
            m_server = nullptr;
        } // function end
    }; // class end
```

The `stop` function is quite easy. We simply call the `httpd_stop` function to stop the server. This will be called when the Wi-Fi is disconnected. Let's continue with the `handleDataGet` implementation:

```
    esp_err_t AppRestServer::handleDataGet(httpd_re
    {
        AppRestServer *obj = (AppRestServer *)(req-
        if (obj->m_config.enabled)
        {
            obj->m_data.button = obj->m_toggle_btn
```

```
            nlohmann::json data_json = obj->m_data;
            std::string data_str = data_json.dump();
            httpd_resp_send(req, data_str.c_str(),
                 data_str.length());
        }
```

As we registered in the `start` function of the class, the `handleDataGet` function will handle the GET request to the `/data` URI. We access the request details with the help of the `httpd_req_t` type. This function takes a pointer of this type as an argument and the underlying HTTP server calls the handler function with such a pointer. After converting the object's `m_data` member to a JSON string, we pass it to the requesting client by calling the `httpd_resp_send` function. This happens if the sensor is enabled. The `else` part comes next:

```
        else
        {
            httpd_resp_send(req, nullptr, 0);
        }
        return ESP_OK;
    } // function end
```

If the sensor is not enabled, we still have to send a reply to the connected client. However, it will only be an empty reply since the sensor is disabled. The default HTTP status code that is sent with the `httpd_resp_send` function is `HTTP-200, OK`.

We implement the `handleConfigPut` function body next:

```
esp_err_t AppRestServer::handleConfigPut(httpd_r
{
    AppRestServer *obj = (AppRestServer *)(req->
    char buffer[256] = {0};
    httpd_req_recv(req, buffer, sizeof(buffer));
    nlohmann::json req_json = nlohmann::json::pa
        buffer, nullptr, false);
```

The `handleConfigPut` function handles the PUT requests to the `/config` resource as we configured earlier. The client has to send data to update the value of the associated resource. We allocate a buffer for this data and read it into the buffer by calling the `httpd_req_recv` function. The expected format is JSON, therefore we try to parse the data in the buffer into a JSON variable. Next, we check whether the parse has failed:

```
    if (req_json.is_discarded())
    {
        httpd_resp_send_err(req, HTTPD_400_BAD_R
            "not a JSON");
    }
```

If the data is not a JSON string, we reply to the request with `HTTPD_400_BAD_REQUEST`. If it is a JSON string, we handle it in the next code snippet:

```
        else
        {
            nlohmann::json config_json = obj->m_con
            config_json.merge_patch(req_json);
            obj->m_config = config_json;
            obj->m_data.name = obj->m_config.name;
            httpd_resp_send(req, NULL, 0);
        }
        return ESP_OK;
    } // function end
```

We update the sensor configuration with the incoming data. The data resource also contains the sensor name and we update it as well. We call the `httpd_resp_send` function to inform the client that its request has been handled successfully. The last implementation in this file is the button handler body, as comes next:

```
    void AppRestServer::handleBtn(void *arg)
    {
        AppRestServer *obj = reinterpret_cast<
            AppRestServer *>(arg);
        obj->m_toggle_btn = !obj->m_toggle_btn;
    } // function end
} // namespace end
```

In the `handleBtn` function, we simply toggle the button state. The `AppRestServer` implementation is done and we can edit the main application in `main/rest_server_ex.cpp`:

```
#include "esp_log.h"
#include "AppWifiSoftAp.hpp"
#include "AppRestServer.hpp"
#define TAG "app"
namespace
{
    app::AppWifiSoftAp app_wifi;
    app::AppRestServer rest_server;
}
```

After including the necessary header files, we define the objects of `AppWifiSoftAp` and `AppRestServer`. Then we continue with the `app_main` function:

```
extern "C" void app_main(void)
{
    auto connected = [](esp_ip4_addr_t *ip)
    {
        ESP_LOGI(TAG, "wifi connected");
        rest_server.start();
    };
```

The `connected` lambda function handles the Wi-Fi-connected event and calls the RESTful server's `start` function. We also need a handler for the Wi-Fi-disconnected event, as comes next:

```
    auto disconnected = []()
    {
        ESP_LOGW(TAG, "wifi disconnected");
```

```
        rest_server.stop();
    };
```

As promised, the Wi-Fi-disconnected event handler stops the RESTful server. Next, we initialize the objects and start the Wi-Fi connection:

```
    rest_server.init();
    app_wifi.init(connected, disconnected);
    app_wifi.connect();
} // function end
```

This completes the application. We are ready to flash and test it.

Testing the application

Let's test the application as follows:

1. We flash the application. You can use the `erase-flash` option for a clean flash operation. It will also wipe out the NVS partition and force Wi-Fi provisioning when it boots up:

```
$ idf.py erase-flash flash monitor
Executing action: flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<logs removed>
```

```
I (1377) app: wifi connected
I (1377) esp_netif_handlers: sta ip: 192.168.153
```

2. The devkit has the IP `192.168.153.223` after provisioning. We will use the *curl* command-line tool to make requests. First, we enable the sensor. After sending the command, we see the `HTTP-200` reply, which means the server handled the request successfully and enabled the sensor:

```
$ curl -w "%{http_code}" 192.168.153.223/config
200
```

3. We can send a GET request to the data resource on the server. This time, we see the sensor data and `HTTP-200` response code:

```
$ curl -w "%{http_code}" 192.168.153.223/data -X
{"button":"off","name":"sensor-0"}200
```

You can try other test cases, such as pressing the **Boot** button of the devkit and then sending another GET request to see whether it is updated or not.

In the next example, we will develop a client application to connect to a RESTful server on another machine.

Consuming RESTful services

In this example, our sensor will read its configuration from a RESTful server by connecting it as a client and publishing its state on the same server. As a server, we will run a simple Flask application in a virtual Python environment. Let's prepare the server in steps:

1. Copy the `ch6/rest_client_ex/server` directory from the GitHub repository and switch to this directory. Install the Python requirements in a virtual environment:

```
$ python --version
Python 3.6.8
$ pyenv virtualenv 3.6.8 apptest
$ pyenv local apptest
(apptest) $ pip install -r requirements.txt
```

2. Set the Flask application:

```
(apptest) $ export FLASK_APP=./rest_server.py
```

3. Start the server. The `-h 0.0.0.0` option makes Flask serve on all network interfaces of the machine:

```
(apptest) $ flask run -h 0.0.0.0
* Serving Flask app './rest_server.py' (lazy loa
<logs removed>
* Running on http://10.8.0.2:5000/ (Press CTRL+C
```

4. The default development server runs on port `5000`. Allow the port on the firewall of the machine where you run the Flask

application:

```
$ sudo ufw allow 5000
```

5. Query the `/config` resource on another terminal. The output shows the current configuration for the `sensor-0` sensor:

```
$ curl http://192.168.1.95:5000/config
{"enabled":true,"name":"sensor-0"}
```

6. Query the `/data` resource. Since nothing has been published yet, the server returns empty JSON data:

```
$ curl http://192.168.1.95:5000/data
{}
```

You can run all these commands as they are listed above, except the one for the firewall. You need to allow port `5000` on whatever firewall you run on the same machine. As a note, the RESTful server also exposes *HTTP PUT* methods for both resources so that we can manipulate the configuration and the sensor can upload its state. You can check the Python code to see how it is implemented. Let's move on to the ESP32 application.

Creating the project

First, we need to configure a new ESP-IDF project with the following steps:

1. Create an ESP-IDF project from the command line (you can choose any other way described in *Chapter 2, Understanding the Development Tools*).

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project rest_client_ex
```

2. Copy the `sdkconfig.defaults` file from the project repository (`ch6/rest_client_ex`) into the project root. It contains the hardware definition (chip type and flash size).
3. Copy the `main/Kconfig.projbuild` file from the repository into the project. It contains the Flask server IP and port configuration.
4. Edit the root `CMakeLists.txt` file with the following content to include the libraries:

```
cmake_minimum_required(VERSION 3.5)
set(EXTRA_COMPONENT_DIRS ../common)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(rest_client_ex)
```

5. Rename the source code file to `main/rest_client_ex.cpp` and update the `main/CMakeLists.txt` file for it:

```
idf_component_register(SRCS "rest_client_ex.cpp"
                       INCLUDE_DIRS ".")
```

Now that we have the project configured, we can continue to code the application.

Coding the application

The first source code file that we add in the project is `main/AppRestClient.hpp` for the client implementation:

```
#pragma once
#include <string>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "json.hpp"
#include "AppEsp32C3Bsp.hpp"
#include "esp_http_client.h"
```

ESP-IDF provides the HTTP client functionality in `esp_http_client.h`. We will use the functions and structures in it to connect to the RESTful server. Next, we define the C structures that correspond to the resources on the server:

```
namespace app
{
    struct sSensorConfig
    {
        std::string name;
        bool enabled;
    };
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(sSensorConfi
        enabled);
```

```
struct sSensorData
{
    std::string name;
    std::string button;
};
NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(sSensorData,
```

In fact, they are the same structures as the ones in the previous RESTful server example. The `sSensorConfig` structure is for the sensor configuration and `sSensorData` is for data, along with their JSON representations. Messages to be exchanged with the server will be in JSON format. Next, we continue with the class implementation:

```
class AppRestClient
{
private:
    AppEsp32C3Bsp m_bsp;
    bool m_toggle_btn;
    TaskHandle_t m_publish_task;
```

In the `private` section of the class, we define a member variable for the FreeRTOS publish task. It will periodically update the server. Then we define the handler functions:

```
        static esp_err_t handleHttpEvents(esp_http_
            *evt);
```

```
        static void publishSensorState(void *arg);
        static void handleBtn(void *arg);
```

The `handleHttpEvents` function is the callback for the incoming data after we query the server. The `publishSensorState` function is the FreeRTOS task function that will do the periodic update on the server. The last function, `handleBtn`, is the handler for button press events. The `public` section of the implementation comes next:

```
    public:
        void init(void)
        {
            m_bsp.init();
            auto fn = [](void *arg)
            {
                handleBtn(arg);
            };
            m_bsp.addButtonHandler(button_cb_type_t
                BUTTON_CB_RELEASE, fn, this);
            xTaskCreate(publishSensorState, "publish
                5, &m_publish_task);
            vTaskSuspend(m_publish_task);
        }
```

In the `init` function of the class, we start with initializing the BSP and attaching the button handler to it. After that, we create the FreeRTOS task and pause it to wait for the Wi-Fi connection. The next two functions are to control the task for Wi-Fi events:

```
        void start(void)
        {
            vTaskResume(m_publish_task);
        }
        void pause(void)
        {
            vTaskSuspend(m_publish_task);
        }
    }; // class end
```

When the state of the Wi-Fi connection is changed, we need to react to such events by starting or pausing the data-publish task accordingly. The `start` and `pause` functions do this job. The class definition is done. We will implement the function bodies that we declared in the `private` section of the class next:

```
    void AppRestClient::publishSensorState(void *arg
    {
        AppRestClient *obj = reinterpret_cast<
            AppRestClient *>(arg);
        std::string host{std::string("http://"
          CONFIG_REST_SERVER_IP ":") + std::to_strin
              CONFIG_REST_SERVER_PORT)};
        std::string config_url{host + "/config"};
        std::string data_url{host + "/data"};
        static char buffer[1024];
```

The `publishSensorState` function is the FreeRTOS task to publish periodic data to the server. We construct the server URL by

concatenating its IP and port. Then we obtain the resource URLs by appending `/config` and `/data` at the end of the `host` variable. The `buffer` variable here will hold the incoming data for further processing when we query the configuration resource of the RESTful server. Next, we start the main loop of the task as follows:

```
        while (true)
        {
            vTaskDelay(pdMS_TO_TICKS(3000));
            esp_http_client_config_t client_config =
                .url = config_url.c_str(),
                .method = HTTP_METHOD_GET,
                .event_handler = handleHttpEvents,
                .transport_type = HTTP_TRANSPORT_OVE
                .user_data = buffer};
```

In the loop, after waiting for three seconds, we define a variable for the HTTP client. We need to query the sensor configuration first, so we will configure the HTTP client accordingly. In the `client_config` variable, we set the URL, HTTP method, event handler function, and user data pointer. The transport type is `HTTP_TRANSPORT_OVER_TCP`. This would be `HTTP_TRANSPORT_OVER_SSL` if we started the RESTful server with an SSL certificate to encrypt the data exchange. With this configuration, we can query the server for the sensor configuration as follows next:

```
            esp_http_client_handle_t client =
                esp_http_client_init(&client_config)
            esp_http_client_perform(client);
```

We define and initialize an HTTP client, and then call the `esp_http_client_perform` function to send an HTTP GET request to the server as configured. The next thing to do is to push the sensor data to the server:

```
            sSensorConfig sensor_config =
                nlohmann::json::parse(buffer);
            if (sensor_config.enabled)
            {
                esp_http_client_set_url(client, data
                esp_http_client_set_method(client,
                    HTTP_METHOD_PUT);
                esp_http_client_set_header(client, "
                    "application/json");
```

We parse the incoming data and create a sensor configuration variable with it. If the sensor is enabled, then we configure the HTTP client to send the latest state of the sensor. We set the URL to `data_url` and the HTTP method to PUT. We also set the request `Content-Type` header as `application/json` since this is what we will be sending:

```
                sSensorData data{.name = sensor_cont
                    .button = obj->m_toggle_btn ? "on
                nlohmann::json data_json = data;
                std::string data_str = data_json.dum
                esp_http_client_set_post_field(clien
                    data_str.c_str(), data_str.lengt
                esp_http_client_perform(client);
            }
```

After we have prepared the data to be sent, we update the request with it by calling the `esp_http_client_set_post_field` function and call the `esp_http_client_perform` function to actually run the request. Before the loop ends, we free the resources that are acquired by the HTTP client:

```
            esp_http_client_cleanup(client);
        } // loop end
    } // function end
```

A call to the `esp_http_client_cleanup` function frees the client resources. Each time the loop runs, we create a new HTTP client and free it at the end of the loop. This finishes the implementation of the `publishSensorState` function. The next one is `handleHttpEvents`:

```
    esp_err_t AppRestClient::handleHttpEvents(
        esp_http_client_event_t *evt)
    {
```

```
        switch (evt->event_id)
        {
        case HTTP_EVENT_ON_DATA:
            memcpy(evt->user_data, evt->data, evt->
        default:
            break;
        }
        return ESP_OK;
    }
```

In the `handleHttpEvents` function, we only handle incoming data events. We copy the incoming data to the memory area marked by the `user_data` pointer, which is the buffer that we set when we initialize the HTTP client. The last function in this file is the callback for the button press events:

```
    void AppRestClient::handleBtn(void *arg)
    {
        AppRestClient *obj = reinterpret_cast<
            AppRestClient *>(arg);
        obj->m_toggle_btn = !obj->m_toggle_btn;
    } // function end
} // namespace end
```

The `handleBtn` function handles the button presses and toggles the `m_toggle_btn` member variable.

We move on to editing the `main/rest_client_ex.cpp` application source code to employ the HTTP client as follows:

```
#include "esp_log.h"
#include "AppWifiSoftAp.hpp"
#include "AppRestClient.hpp"
#define TAG "app"
namespace
{
    app::AppWifiSoftAp app_wifi;
    app::AppRestClient rest_client;
}
```

After including the headers, we define the `app_wifi` and `rest_client` objects in the anonymous namespace. Then we develop the `app_main` function as comes next:

```
extern "C" void app_main(void)
{
    auto connected = [](esp_ip4_addr_t *ip)
    {
        ESP_LOGI(TAG, "wifi connected");
        rest_client.start();
    };
    auto disconnected = []()
    {
        ESP_LOGW(TAG, "wifi disconnected");
        rest_client.pause();
    };
```

We start with defining two lambda functions for Wi-Fi-connected and disconnected events. When Wi-Fi is connected, we resume the

publish task. We pause it when the Wi-Fi drops. The only thing left in the application is to initialize the objects and start the Wi-Fi client:

```
    rest_client.init();
    app_wifi.init(connected, disconnected);
    app_wifi.connect();
}
```

We initialize `rest_client` and `app_wifi`, then finally call the `connect` function of the `app_wifi` object to start the Wi-Fi connection.

The application is finished and ready to build and test.

Testing the application

Let's test the application as in the following steps:

1. Set the server IP and the port in `menuconfig` before building the application:

```
$ idf.py menuconfig
```

2. Assuming that the server is already running, we can query the current sensor configuration and sensor data (the server IP for me is `192.168.153.199` and the port is `5000`):

```
$ curl http://192.168.153.199:5000/config
{"enabled":true,"name":"sensor-0"}
```

```
$ curl http://192.168.153.199:5000/data
{}
```

3. Flash the application:

```
$ idf.py flash monitor
```

4. Observe that the devkit has started to send data:

```
$ curl http://192.168.153.199:5000/data
{"button":"off","name":"sensor-0"}
```

5. Press the **Boot** button of the devkit to toggle the button state:

```
$ curl http://192.168.153.199:5000/data
{"button":"on","name":"sensor-0"}
```

If you face any issues in any of the steps above, please take a look at the *Troubleshooting* section below to see the common reasons for problems with possible solutions.

Troubleshooting

If the application doesn't connect to the server, you can try the following things to resolve the issue:

- Make sure the Flask server is starting with the `-h 0.0.0.0` option. This option makes Flask serve on all the possible network interfaces of your machine.

- If you are running a firewall, allow port `5000` for incoming traffic.
- Check the application configuration by running `menuconfig` and see that the server IP and port are correct.

This was the last example of the chapter. Connectivity is one of the most important subjects in IoT and the selection of the communication technology stack can make a difference in the product's success or failure. Therefore, for us, it is of utmost criticality to be knowledgeable about them and their advantages and disadvantages when analyzing the project requirements and designing a solution. Although the protocols that we discussed in this chapter are very common in the IoT industry, this is by no means a comprehensive list. In *Further reading*, you can find more books about other IoT protocols.

Summary

In this chapter, we have seen examples of some popular IoT connectivity protocols. To connect a device to the internet, we need a networking infrastructure in the first place. Wi-Fi is the one for ESP32. We learned how to connect ESP32 to a Wi-Fi network and how to provision it if there is no Wi-Fi network configured. ESP-IDF provides several different ways to do this. After having a Wi-Fi connection, we can communicate with other devices on the network. **MQTT** is a many-to-many IoT communication protocol with a central broker that connects clients.

We installed *Mosquitto* as the MQTT broker on a machine in the local network and developed an application on ESP32 to subscribe and

publish to the MQTT topics. **REST** is another common method for IoT communication. We can run a RESTful server on ESP32 or use ESP32 as a client that consumes a RESTful service. We developed applications for both use cases.

The upcoming chapter will explain how to develop secure applications on ESP32. An IoT device is a connected device and can be as susceptible to cyber-attacks as any other machine on the internet. As developers, it is our responsibility to take the necessary precautions to protect our products from any malicious or unauthorized access to protect our devices and hence their users. In the next chapter, we will learn how to do that with ESP32.

Questions

Here are some questions to review what we have learned in this chapter:

1. Which one of the following statements is FALSE?
    a. ESP32 can start in Wi-Fi STA mode
    b. ESP32 can start in Wi-Fi AP mode
    c. In STA mode, a static IP is required
    d. Wi-Fi events are delivered over an event loop
2. Which of the following is not a method provided by ESP-IDF?
    a. NFC provisioning
    b. Unified provisioning
    c. Smart Config
    d. Easy Connect

3. Which protocol makes use of topics to publish and subscribe?

    a. HTTP
    b. WebSocket
    c. CoAP
    d. MQTT

4. Which of the following is the correct HTTP method to request a resource from a RESTful server?

    a. PUT
    b. DELETE
    c. POST
    d. GET

5. Which HTTP status code can we expect from a RESTful service when we try to create a resource by sending a POST request?

    a. 201 Created
    b. 400 Bad Request
    c. 500 Internal Server Error
    d. All of the above

Further reading

- *Developing IoT Projects with ESP32, 1st Edition*, by Vedat Ozan Oner (https://www.packtpub.com/product/developing-iot-projects-with-esp32/9781838641160): *Chapter 6, A Good Old Friend – Wi-Fi*, *Chapter 8, I Can Speak BLE*, *Chapter 10, No Cloud No IoT*. In the *1st Edition*, several other IP-based protocols and BLE are discussed with examples.

- *IoT and Edge Computing for Architects, 2nd Edition*, by Perry Lea (https://www.packtpub.com/product/iot-and-edge-computing-for-architects-second-edition/9781839214806): *Chapter 10, Edge to Cloud Protocols*.
- *Networking Fundamentals*, by Gordon Davies (https://www.packtpub.com/product/networking-fundamentals/9781838643508): *Chapter 4, Understanding Wireless Networking*, explains all the basics of Wi-Fi starting from the physical layer, including the network topology. TCP/IP protocols are discussed in *Chapter 10, Understanding TCP/IP*.
- *Hands-On Network Programming with C*, by Lewis Van Winkle (https://www.packtpub.com/product/hands-on-network-programming-with-c/9781789349863): I strongly recommend this book if you want to learn more about network programming. Sockets are discussed in *Chapter 2, Getting to Grips with Socket APIs*. Application layer protocols, such as HTTP, are explained in *Section 2* of the book. *Chapter 14, Web Programming for the Internet of Things*, is dedicated to IoT connectivity, which is discussed in broad terms to give you an overall picture.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

7

ESP32 Security Features for Production-Grade Devices

Any internet-facing solution must be designed with “security-first” in mind; otherwise, it would be vulnerable to cyber-attacks. This especially applies to IoT products since they are usually introduced in batches to the market and delivered to end users who often don’t have a basic understanding of IoT security. When it comes to security, ESP32 provides a good level of hardware support for developers with its cryptography subsystem. ESP-IDF also integrates industry-standard encryption libraries and provides a good abstraction when a custom security solution is needed.

In this chapter, we will discuss the essentials of the ESP32 platform when developing production-grade IoT devices and see examples of secure communication protocols to understand how to utilize them in our projects. The RainMaker platform by Espressif Systems will support us in the examples of the chapter to understand the bare minimum before launching an IoT product on the market.

This chapter requires some background knowledge about the fundamentals of cryptography to follow the examples easily. If you don’t feel comfortable with the security fundamentals, you can find

some resources to help you in the *Further reading* section at the end of the chapter.

In this chapter, we're going to cover the following topics:

- ESP32 security features
- **Over-the-air** (**OTA**) updates
- Sharing data over secure MQTT

Technical requirements

The hardware requirements of the chapter are:

- ESP32-C3 DevkitM-1
- A **Light-Dependent Resistor** (**LDR**) or photosensor
- A pull-up resistor (10KΩ)
- Jumper wires

On the software side, we will use the RainMaker library. The book repository contains it as a sub-module, but you can find it here as well: https://github.com/espressif/esp-rainmaker.

The mobile applications from Espressif Systems are listed below. They are available for both Android and iOS mobile devices:

- **ESP SoftAP Provisioning**: The application for joining a WiFi network.
- **ESP RainMaker**: The companion application that comes with the RainMaker platform to add devices and manage them in the platform.

Other software tools and libraries that we need in the examples are:

- **openssl**: An industry-standard tool to generate certificates and run web servers. You can find the binaries here: https://wiki.openssl.org/index.php/Binaries.
- **Flask**: A Python module to develop web servers. Its documentation is available here: https://flask.palletsprojects.com/en/2.2.x/.
- **curl**: A utility to interact with TCP/IP applications: https://curl.se/.

The examples in the chapter are in the GitHub repository here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch7.

ESP32 security features

Espressif Systems has shown constant progress in security features with every new family of ESP32. To be honest, the initial versions of ESP32 had some hardware design flaws that could enable an attacker to access the system. Espressif not only corrected these flaws but also added many other important security features in the new designs. The Espressif team definitely knows the needs of IoT product developers and reflects them on their product roadmaps. ESP-IDF has also been regularly updated to support all the ESP32 chips, enabling IoT developers to provide their customers with devices and products that fulfill modern security requirements. Let's look at an overview of those features.

Secure Boot v1

Secure Boot creates a chain of trust, from boot to the application firmware, by authenticating the running software at each step. The whole process may sound a bit confusing, but in simple terms, it works as follows:

- Authenticate the bootloader by using the symmetric key (AES-256) in eFuse block 2: ESP32 has 1,024-bit **One-Time Programmable** (**OTP**) memory to keep system settings and secrets. This OTP memory is composed of four blocks of eFuses. The Secure Boot key is kept in block 2. After enabling Secure Boot, it is not possible to read the key using software due to the read/write protection. ESP32 verifies the bootloader image by using the Secure Boot key.
- Authenticate the application by using the asymmetric key pair generated by the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**): Each application is signed by the private part of a key pair and the bootloader has the public key. Before loading the application in the memory, the bootloader verifies the signature by using the public key it has.

For more information about Secure Boot v1, the official documentation is a perfect resource, at this link: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v1.html.

After enabling Secure Boot, the second layer of protection is flash encryption. The flash encryption key (AES-256) is stored in eFuse block 1. Again, the key is not accessible using software. When flash encryption is enabled, ESP32 encrypts the bootloader, partition table, and application partition. The Espressif documentation explains all

the details about flash encryption here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html.

Secure Boot v2

Secure Boot v2 is available from ESP32 Rev3 and for all other ESP32 families. The main difference from Secure Boot v1 is that bootloader authentication is also done by asymmetric key signature verification, using **Rivest–Shamir–Adleman-Probabilistic Signature Scheme** (**RSA-PSS**). This time, eFuse block 2 keeps the SHA-256 of the public key, which is embedded in the bootloader. ESP32 first validates the public key of the bootloader and then verifies the signature of the bootloader by using this public key. After the bootloader is authenticated, control is passed to the bootloader to continue with the application authentication. If any signatures in these steps don't match, ESP32 simply won't run the application. The advantage of this scheme is that there is no private key on ESP32; therefore, it is impossible to run any malicious code on the device if the private key is kept in a secure location. For more information about how to enable Secure Boot v2, you can refer to the official documentation here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v2.html.

Secure Boot and flash encryption are not reversible. Therefore, only apply them to production devices.

There are several key points to remember before going to mass production:

- We must ensure the authenticity of the firmware – that is, the firmware running on our ESP32 device must be our firmware. The way to do that is to enable Secure Boot. The critical issues here are the generation of cryptographic keys and the security of private keys. We want to keep the private keys private, and we should only share the public keys with third parties, such as production and assembly factories.
- Secure Boot guarantees authenticity, but it doesn't encrypt firmware, which means that it is flashed as cleartext. Anyone can read the firmware and use it somewhere else. Moreover, if we have to embed some sensitive data in it, the implications of this vulnerability can reach beyond the device itself. Flash encryption is used to prevent attacks. However, firmware is still open to exploitation at the assembly factory if we share the application firmware. Therefore, it is preferable to use two different firmware binaries: the first one for the assembly factory that validates the hardware, and the second one being the real application, where we flash it in a trusted factory area with flash encryption.

Digital Signature (DS)

For IoT devices, Secure Boot and flash encryption are only a part of the story. In scenarios where a private key is needed to be kept on the device (for instance, mutual authentication with a secure server), we want to have more security on that key. Espressif introduced the

Digital Signature peripheral on newer ESP32 families for this purpose.

The common way of keeping private keys private is to integrate a secure element in hardware design. For example, the ESP32-WROOM-32SE module has Microchip's **ATECC608A** chip integrated inside. A secure element stores the private keys of an application and, when a cryptographic operation is needed, such as sign or verify, the secure element runs the underlying security algorithm with the associated private key and returns the result to the application. There is no way to retrieve a private key from a secure element. Although the digital signature peripheral works a bit differently, it serves the same purpose: keeping a private key secure.

The DS peripheral is only available on ESP32-S2, ESP-S3, ESP32-C3, and ESP-C6, and it provides hardware acceleration for RSA-based operations. The technique that DS uses for private key security is a combination of several other cryptographic methods. The basic idea is to employ a secret key to encrypt the application's private key so that the only way to make use of the application's private key is to have the initial secret key. The good thing about this secret key is that it is only accessible by DS when DS is enabled in the application.

We have two stages to utilize the DS peripheral in an application:

1. **Setup phase**: In the setup phase, a 256-bit secret key and a 256-bit **initialization vector** (**IV**) are generated. With these parameters, the application's private key is encrypted (symmetric encryption). The encrypted private key and the IV are stored on the flash. However, the secret key is stored in one of the eFuse blocks of ESP32, and that block is locked to prevent

any further access. From this point, it becomes available only to hardware – i.e., the DS peripheral.

2. **Operation**: During operation, DS takes data to be signed, calculates the signature, and returns the result to the application. Since the secret key, which is on the eFuse block to decrypt and use the encrypted private key, is only available to DS, the application has to call the DS functions to generate valid signatures. It is usually enough for an application to initialize the ESP-TLS library with the parameters from the setup phase for TLS communication. ESP-TLS works with the DS peripheral and the MbedTLS library to accomplish encryption tasks. With DS configured, even if an attacker somehow had access to the application binary, what they could see would be only the encrypted private key, which is useless.

You can refer to the online documentation for more information about the DS peripheral here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/api-reference/peripherals/ds.html.

ESP Privilege Separation

ESP Privilege Separation is an interesting framework by Espressif Systems. It is quite new and under active development but what it offers can be very useful in some specific projects.

An application that we develop on ESP32 has access to almost every resource (memory, peripherals, flash, etc.) and uses them freely; there is no restriction on how the application uses the underlying system resources. The driving idea behind this framework is to

divide the application into two parts: privileged environment and user application.

The application running in the privileged environment (protected application) has control of the entire system and provides a controlled interface for the user application. The user application can access the resources only as much as allowed by the protected application. Moreover, whatever is done in the user application is isolated from the rest of the system, protecting the underlying privileged environment from unauthorized access and invalid operations – for example, writing to a memory address that is not granted. Basically, this framework is an implementation of the **Trusted Execution Environment** (**TEE**) architecture.

With such a framework, it is much easier to develop a specific type of IoT platform where end-users can easily develop a user application on ESP32 for their own business logic. The hard work (such as provisioning, secure connection to the backend, logging, monitoring, and OTA firmware upgrade) can be handled by the protected application as a component of the IoT platform.

If you would like to give it a try, the repository of the framework is here: https://github.com/espressif/esp-privilege-separation.

In the next section, we will discuss another indispensable feature for production-grade IoT devices – over-the-air updates.

Over-the-air updates

After deploying an IoT device, it doesn't mean the development is finished forever. On the contrary, this is the most important stage in the life cycle of an IoT product and still requires active development. We might want to add new features as a response to the users' needs or it might be a necessity to improve the product security after discovering a potential risk. In either case, we need to have a means to update the firmware remotely without physically touching the deployed IoT products. **Over-the-air** (**OTA**) **update** techniques provide this capability.

The basic OTA update mechanism in ESP-IDF works as follows:

1. We configure the flash to have two different partitions, `ota_0` and `ota_1`, to accommodate the running firmware and a new firmware.
2. When we upload a new firmware, ESP-IDF chooses the free partition to save the incoming bytes. ESP-IDF marks it as the candidate active partition as soon as all the bytes are transferred.
3. If the validation of the application on the candidate partition passes, then the bootloader will mark the partition as the active one and run the application on it after a reboot.

We have several options for how we can design the OTA update mechanism for our applications:

- Secure update with signed applications.
- Enable/disable Secure Boot.
- Application rollback. When this feature is enabled, we can restore the previous version of the application if needed – for example, if a faulty firmware is downloaded. However,

disabling it can also be a good idea to prevent any mistaken update operations.

As a remote firmware source, you can choose anything in your design. It can be an HTTP server or a cloud-based MQTT broker to transfer firmware from file storage. We will have two examples of OTA updates with different designs in this chapter.

Upgrading firmware from an HTTPS server

In the first example of the OTA update, we will start a file server in the local network and download new firmware from this server to upgrade the device. The file server is a simple Python Flask application that runs on our development machine. The device firmware will poll the firmware information from the server periodically and if it shows the firmware version on the server is different from the version that runs on the device, then it will download the new firmware from the server for an upgrade.

One of the important points in OTA updates is that we want a secure communication channel between the server and devices to prevent any unwanted ears from listening to the network traffic and intercepting the new firmware while transferring to the devices. Thus, we will configure the server with TLS encryption in this example. The Flask application will use a TLS certificate to encrypt the HTTP traffic between the parties. On the ESP32 application side, we will embed the public key of the HTTPS server and configure the HTTP client in the application to use it while talking to the server.

The devkit in this example is *ESP32-C3 DevkitM-1*. Let's prepare the HTTPS server first.

Preparing the server

We can start the HTTPS server on our development machines as in the following steps:

1. Copy the `ch7/ota_http_ex/server` directory from the GitHub repository and switch to this directory. Install the Python requirements in a virtual environment:

```
$ pyenv virtualenv 3.6.8 apptest
$ pyenv local apptest-ota
(apptest) $ pip install -r requirements.txt
```

2. Create a TLS certificate for the secure HTTP server:

```
(apptest) $ openssl req -x509 -newkey rsa:4096 -
```

3. Set the Flask application:

```
(apptest) $ export FLASK_APP=./file_server.py
```

4. Start the server. The `-h 0.0.0.0` option makes Flask serve on all network interfaces of the machine:

```
(apptest) $ flask run --cert cert.pem --key key.
 * Serving Flask app './file_server.py'
```

```
 * Debug mode: off
WARNING: This is a development server. Do not us
 * Running on all addresses (0.0.0.0)
 * Running on https://127.0.0.1:5000
 * Running on https://192.168.1.95:5000
```

5. The default development server runs on port 5000. Allow the port on the firewall of the machine where you run the Flask application. On an Ubuntu machine, the command is:

```
$ sudo ufw allow 5000
```

6. The server exposes an endpoint to query the firmware information. Query the `/info` resource on another terminal to see that it returns the firmware filename:

```
$ curl https://192.168.1.95:5000/info -k
ota_http_ex.bin
```

Having the secure file server ready, we can create an ESP-IDF project next.

Creating a project

Let's follow the steps below to have a new project for the example:

1. Create an ESP-IDF project from the command line (you can choose any other way, as described in *Chapter 2, Understanding the Development Tools*):

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project ota_http_ex
```

2. Create a root `CMakeLists.txt` file, which sets the extra component directories:

```
cmake_minimum_required(VERSION 3.5)
set(EXTRA_COMPONENT_DIRS ../common)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(ota_http_ex)
```

3. Add `sdkconfig.defaults` with the following content. Please note that we want a partition table with two OTA partitions:

```
CONFIG_IDF_TARGET="esp32c3"
CONFIG_IDF_TARGET_ESP32C3=y
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=y
CONFIG_FREERTOS_TIMER_TASK_STACK_DEPTH=3120
CONFIG_BT_ENABLED=y
CONFIG_BTDM_CTRL_MODE_BLE_ONLY=y
CONFIG_BT_NIMBLE_ENABLED=y
CONFIG_PARTITION_TABLE_TWO_OTA=y
```

4. Create another file in the project root and name it `version.txt`. ESP-IDF includes the content of this file automatically as the firmware version. It has a single line that shows the current version:

```
v1.0
```

5. Rename the application source code to `main/ota_http_ex.cpp` and update the `main/CMakeLists.txt` file with the following content. We also command the build system to embed the server certificate in the project binary so that we can connect to the file server over HTTPS.

```
idf_component_register(SRCS "ota_http_ex.cpp" IN
```

6. We can add a file, `main/Kconfig.projbuild`, to make the file server information configurable. Its content is:

```
menu "Application settings"
    config FILE_SERVER_IP
        string "IP of the file server"
        default "10.1.1.10"
        help
            The IP of the file server
    config FILE_SERVER_PORT
        string "Port of the server in use"
        default "5000"
        help
            The port of the file server.
endmenu
```

We have configured the application, and it is ready for development.

Coding the application

Let's add a C++ header file, `main/AppOtaClient.hpp`, to implement the OTA client, which queries the file server for new firmware. We start by including the required header files:

```
#pragma once
#include <string>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "AppEsp32C3Bsp.hpp"
#include "esp_http_client.h"
#include "sdkconfig.h"
#include "esp_ota_ops.h"
#include "esp_https_ota.h"
#include "esp_log.h"
```

We have two important header files here: `esp_http_client.h` contains definitions for connecting to the file server securely by using the TLS certificate of the server, and `esp_https_ota.h` helps us to connect to the file server, download any new firmware, and manage the entire OTA process. Next, we add some definitions for the server:

```
#define FILE_SERVER_URL "https://" CONFIG_FILE_SERVE
extern const uint8_t server_cert_pem_start[] asm(
    "_binary_cert_pem_start");
```

```
extern const uint8_t server_cert_pem_end[] asm(
    "_binary_cert_pem_end");
```

The `FILE_SERVER_URL` macro combines IP and port values that come from `sdkconfig.h` to construct the server URL. As we instructed the build system, the TLS certificate of the file server is embedded in the firmware binary and the `server_cert_pem_start` constant marks its address in the memory. Then, we can continue with the OTA update class:

```
namespace app
{
    class AppOtaClient
    {
    private:
        TaskHandle_t m_ota_task;
        static void otaTaskFunc(void *arg);
```

In the `private` section of the `AppOtaClient` class, we define a member variable to hold the underlying FreeRTOS tasks. Its job is to poll the server periodically and install any valid firmware if found. We'll provide the `otaTaskFunc` function for that task. We define the endpoint URLs as the following:

```
        static constexpr char *INFO_ENDPOINT{
    FILE_SERVER_URL "/info"};
```

```
        static constexpr char *FILE_SERVER_STATIC{
    FILE_SERVER_URL "/static/"};
```

The file server has two endpoints: one is to query the firmware information, and the other one is the location of the binary firmware file. When we query the firmware information by sending an `HTTP GET` request to `INFO_ENDPOINT`, it will return the firmware filename. We will construct the download URL by concatenating the `FILE_SERVER_STATIC` value and the filename that comes with the query, and then start the download. We complete the `private` section of the class definition by adding functions to accomplish these tasks, as comes next:

```
        std::string getInfo(char *buffer, size_t bu
        void doOtaUpdate(char *buffer, size_t buffe
            const std::string &filename);
        static esp_err_t handleHttpEvents(
            esp_http_client_event_t *evt);
        bool m_ota_done;
```

It is the `getInfo` function that queries the server for the firmware filename. The `doOtaUpdate` function takes this filename as a parameter and performs the OTA update if the firmware on the server is a newer version. The `handleHttpEvents` function is the handler for HTTP events. The last one, `m_ota_done`, is a member variable that shows if the downloaded firmware is valid. Next, we develop the functions in the `public` section of the class:

```
    public:
        void init(void)
        {
            m_ota_done = false;
            xTaskCreate(otaTaskFunc, "ota", 8192, th
                5, &m_ota_task);
            vTaskSuspend(m_ota_task);
        }
```

In the `public` section, the first function is `init`. It basically creates the FreeRTOS task that monitors the server with the `otaTaskFunc` function and suspends it immediately since we need a WiFi connection first. Please note that we pass the `this` pointer to `xTaskCreate` as a parameter, so that we can have access to the class instance at runtime when the task starts. When the WiFi connection is established, this task can resume operating. The next two functions provide the external world with a mechanism to control the state of the FreeRTOS task according to the WiFi connection status:

```
        void start(void)
        {
            vTaskResume(m_ota_task);
        }
        void pause(void)
        {
```

```
            vTaskSuspend(m_ota_task);
        }
```

The `start` function resumes the OTA update FreeRTOS task and `pause` suspends it. The last function of the `public` section indicates whether an OTA download is finished:

```
        bool isOtaDone(void) const
        {
            return m_ota_done;
        }
    }; // class end
```

When new firmware is ready on the flash, we need to inform the main task of the application. As a simple method, the main task calls the `isOtaDone` function periodically to see whether the OTA client has found new firmware to deploy. The class definition is done but we haven't implemented the `private` functions yet. Let's start with the `otaTaskFunc` function:

```
    void AppOtaClient::otaTaskFunc(void *arg)
    {
        AppOtaClient *obj = reinterpret_cast<AppOta(
        static char buffer[8192];
```

`otaTaskFunc` is a static function for the FreeRTOS task. However, we can access the `AppOtaClient` object, which is created at runtime,

by casting back the `arg` argument to an object pointer. The `buffer` variable here is a placeholder for the incoming data from the server. Then, we add the loop of the task:

```
        while (true)
        {
            vTaskDelay(pdMS_TO_TICKS(10000));
            std::string filename = obj->getInfo(buf
                sizeof(buffer));
            if (!filename.empty())
            {
                obj->doOtaUpdate(buffer, sizeof(buf
                    filename);
            }
        }
    } // function end
```

The task loop calls the `getInfo` function every 10 seconds to receive the firmware filename, then it passes this information to the `doOtaUpdate` function of the `AppOtaClient` instance. We can implement the `getInfo` function as follows:

```
    std::string AppOtaClient::getInfo(char *buffer,
        buffer_len)
    {
        memset(buffer, 0, buffer_len);
        esp_http_client_config_t client_config = {
            .url = INFO_ENDPOINT,
            .cert_pem = (const char *)server_cert_pe
```

```
            .method = HTTP_METHOD_GET,
            .event_handler = handleHttpEvents,
            .transport_type = HTTP_TRANSPORT_OVER_SS
            .user_data = buffer,
            .skip_cert_common_name_check = true};
```

In the `getInfo` function body, we first clear the buffer that we use for storing incoming data from the server. Then we define the configuration variable, which describes how we connect to the file server securely. The configuration contains the usual information about the connection, such as the URL and HTTP method, as well as the buffer to store data. It also shows that the transport protocol is TLS and has a field for the server certificate, which will be used to establish a secure HTTP connection to the server. With this configuration, we initialize the HTTP client next:

```
        esp_http_client_handle_t client =
            esp_http_client_init(&client_config);
        std::string filename{""};
        if (esp_http_client_perform(client) == ESP_(
        {
            filename = std::string(buffer);
        }
        esp_http_client_cleanup(client);
        return filename;
    }
```

We call the `esp_http_client_init` function of the HTTP client library of ESP-IDF to initialize the HTTP client. Then, we query the server as described in the configuration by calling `esp_http_client_perform`. The buffer should contain the firmware filename after this query. We return the filename as a string from `getInfo`. Let's develop a short function, `handleHttpEvents`, for HTTP event handling next:

```
esp_err_t AppOtaClient::handleHttpEvents(
    esp_http_client_event_t *evt)
{
    switch (evt->event_id)
    {
    case HTTP_EVENT_ON_DATA:
        memcpy(evt->user_data, evt->data, evt->
    default:
        break;
    }
    return ESP_OK;
}
```

The `switch` statement of the `handleHttpEvents` function only handles the `HTTP_EVENT_ON_DATA` case. We copy the event data, which is the firmware filename, to the buffer as pointed by the `evt->user_data` field. We continue with the `doOtaUpdate` implementation, where we use this information:

```
void AppOtaClient::doOtaUpdate(char *buffer, si
    buffer_len, const std::string &filename)
{
    std::string file_url{std::string(FILE_SERVE
        filename};
```

The `doOtaUpdate` function takes `filename` as an argument and uses it to construct the file URL. Then we define a variable for the secure HTTP connection as we did in the `getInfo` function:

```
        esp_http_client_config_t client_config = {
            .url = file_url.c_str(),
            .cert_pem = (const char *)server_cert_pe
            .method = HTTP_METHOD_GET,
            .event_handler = nullptr,
            .transport_type = HTTP_TRANSPORT_OVER_SS
            .user_data = buffer,
            .skip_cert_common_name_check = true};
```

In fact, it is almost the same configuration, with just a slight difference. The `url` field shows the firmware file and we don't need an HTTP event handler this time since the data will be handled by the OTA update library of ESP-IDF. Let's define the configuration for the OTA update:

```
        esp_https_ota_config_t ota_config = {
            .http_config = &client_config,
```

```
            .partial_http_download = true,
            .max_http_request_size = (int)buffer_le
```

The OTA configuration has a field to keep a pointer to the HTTP client configuration that we have just defined. The library will create an HTTP client with it. We also set `partial_http_download` to `true` since the firmware is too big to download in a single HTTP `GET` request. We also set the download chunk size as `buffer_len`. In the next code snippet, we find the version of the current running firmware:

```
        const esp_partition_t *running =
            esp_ota_get_running_partition();
        esp_app_desc_t running_app_info;
        esp_ota_get_partition_description(running,
            &running_app_info);
```

We have two partitions for firmware, as we configured at the very beginning of the example while creating the project. One partition holds the current running firmware and the other one is reserved for the OTA update. We call `esp_ota_get_running_partition` to get a pointer to the running partition. Then we run the `esp_ota_get_partition_description` function to get its description. This description contains the version of the running firmware. Next, we initialize the OTA library:

```
        esp_https_ota_handle_t https_ota_handle{nul
        esp_https_ota_begin(&ota_config, &https_ota_
```

We define a `handle` variable for the OTA update and then call the `esp_https_ota_begin` function to create the handle with the OTA configuration. Then, we retrieve the new firmware description from the server:

```
        esp_app_desc_t app_desc;
        if (esp_https_ota_get_img_desc(https_ota_har
            != ESP_OK || memcmp(app_desc.version,
             running_app_info.version,
             sizeof(app_desc.version)) == 0)
        {
            esp_https_ota_abort(https_ota_handle);
            return;
        }
```

Every ESP32 application has description data at the beginning of its firmware file. The `esp_https_ota_get_img_desc` function exploits this property of firmware files by only requesting the header part of the firmware from the remote server. After this partial data comes from the file server, it extracts the firmware description and stores it in a variable, which is `app_desc` in our code. We compare the running firmware version and the remote firmware version. If they are the same, then we abort the process by calling `esp_https_ota_abort` and return from the `doOtaUpdate` function

without doing anything else. If the firmware description indicates new firmware, we need to download it, as comes next:

```
        int img_size = esp_https_ota_get_image_size(
            https_ota_handle);
        while (esp_https_ota_perform(https_ota_handl
            ESP_ERR_HTTPS_OTA_IN_PROGRESS)
        {
            int read_size = esp_https_ota_get_image_
                https_ota_handle);
            ESP_LOGI(__func__, "%%%0.1f (bytes %d/%d
                ((float)read_size) * 100 / img_size,
                read_size, img_size);
        }
```

In a loop, we call the `esp_https_ota_perform` function to download the firmware from the server. When it is finished, we check the validity of the downloaded data as follows:

```
        if (esp_https_ota_is_complete_data_received(
            https_ota_handle) != true)
        {
            esp_https_ota_abort(https_ota_handle);
        }
```

The `esp_https_ota_is_complete_data_received` function shows whether the downloaded data is valid firmware. If it

somehow finds an error, we abort the process. The `else` part of the `if` clause completes the OTA update process:

```
        else
        {
            if (esp_https_ota_finish(https_ota_hand
            {
                m_ota_done = true;
                vTaskDelete(nullptr);
            }
            else
            {
                esp_https_ota_abort(https_ota_handle
            }
        } // outer else
    } // function end
} // namespace end
```

We end the process by calling the `esp_https_ota_finish` function. If it also shows a successful OTA update, we set the `m_ota_done` member variable to `true` and delete the FreeRTOS task since we don't need it anymore. When the main task of the application checks the OTA status by calling the `isOtaDone` function of the `AppOtaClient` instance, it will reboot the device to activate the new firmware.

The class implementation was a bit longer than the average in our examples, but it encapsulates an entire secure OTA process. The next step is to edit the `main/ota_http_ex.cpp` application source to

instantiate this class and use the object for the OTA firmware upgrade:

```
#include "esp_log.h"
#include "AppWifiSoftAp.hpp"
#include "AppOtaClient.hpp"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#define TAG "app"
```

We first include the header files. Then, we need to create instances of the WiFi client and OTA client as follows:

```
namespace
{
    app::AppWifiSoftAp app_wifi;
    app::AppOtaClient ota_client;
}
```

We will integrate the `app_wifi` and `ota_client` objects by connecting them in callback functions. Let's do it in the `app_main` function of the application:

```
extern "C" void app_main(void)
{
    auto connected = [](esp_ip4_addr_t *ip)
    {
        ESP_LOGI(TAG, "wifi connected");
```

```
        ota_client.start();
    };
```

The first callback function is `connected`. It is called when WiFi is connected, hence the name, and starts the OTA client. We also have another callback for the WiFi disconnected event:

```
    auto disconnected = []()
    {
        ESP_LOGW(TAG, "wifi disconnected");
        if (!ota_client.isOtaDone())
        {
            ota_client.pause();
        }
    };
```

In the `disconnected` lambda function, we first check whether an OTA update has just finished. If this is the case, we shouldn't call the `pause` function of `ota_client` because there will be nothing to pause since it has deleted the underlying FreeRTOS task. The next step is to initialize all the objects and connect to the WiFi:

```
    ota_client.init();
    app_wifi.init(connected, disconnected);
    app_wifi.connect();
```

We pass the lambda functions to the `app_wifi` object so that we can connect to the WiFi by calling the `connect` function. We complete the application implementation by adding the main loop:

```
    while (1)
    {
        vTaskDelay(pdMS_TO_TICKS(2000));
        if (ota_client.isOtaDone())
        {
            esp_restart();
        }
    } // loop
} // app_main end
```

In the `while` loop, we check whether there is new firmware ready to boot. If so, we reboot the device by calling the `esp_restart` function. When booted, the bootloader finds this new firmware and activates it. Great! Let's test it now.

Testing the application

The following steps show how to test the OTA update in our setup:

1. First, check whether the file server is up and running in a shell window.
2. Run `menuconfig` and set the file server IP and port at (**Top**) → **Application settings**. You can see this information on the Flask application console (for me, it says `Running on https://192.168.142.199:5000`):

```
$ idf.py menuconfig
```

3. Update the content of `version.txt` to `v1.1`, compile, and move the `build/ota_http_ex.bin` file to the `server/static` folder:

```
$ echo v1.1 > version.txt && idf.py build && mv
```

4. Revert the version to `v1.0` and flash the devkit with this version. When it connects to the file server, it will see that it has a different version from the one on the server and upgrade itself from the server. If the devkit is not configured for any WiFi network, you can use the *ESP SoftAP Prov* mobile application to provision it.

```
$ echo v1.0 > version.txt && idf.py flash monito
```

5. You can observe that it connects to the file server and starts downloading the firmware.
6. When the download finishes, it applies the new firmware and reboots. The serial log shows the new version:

```
I (313) cpu_start: Pro cpu up.
I (321) cpu_start: Pro cpu start user code
I (321) cpu_start: cpu freq: 160000000
I (321) cpu_start: Application information:
I (324) cpu_start: Project name:     ota_http_ex
I (329) cpu_start: App version:      v1.1
```

```
I (334) cpu_start: Compile time:     Feb 11 2023
I (340) cpu_start: ELF file SHA256:  261f1697949
I (346) cpu_start: ESP-IDF:          v4.4.2
```

This example was interesting in several aspects. We started a secure file server with a self-signed certificate. We configured the application to have two partitions: one holds the running active firmware and the other one is reserved for OTA updates. When started, it connected to the local WiFi and then to the HTTPS server. Since we embedded the server certificate in the firmware, the application was able to connect to the file server by using its certificate. The next step was to query the server to get the filename. We constructed the file URL and then the OTA update started. After downloading the new firmware, the devkit rebooted and upgraded to the new version of the application.

Of course, HTTPS is not the only method for OTA updates. We can customize the process according to the product design. For instance, if we have a backend system that runs an MQTT broker for IoT devices, it is quite possible to transfer a new firmware over MQTT and use it from there. In the next example, we will see how to do an OTA update on Espressif's **RainMaker** platform.

Troubleshooting

If you get stuck at some point while trying the example, you can check the following list for help:

- During WiFi provisioning, if the mobile application gives an error about pairing with the devkit, just retry by scanning the QR code again.
- Check the Flask version. The older versions don't support command-line certificate parameters:

```
(apptest) $ flask --version
Python 3.8.0
Flask 2.2.2
Werkzeug 2.2.2
```

- Make sure you have configured the firmware with the correct server IP and port by running `menuconfig`. Check the local firewall for that port.
- You can sometimes see that the devkit fails to connect to the Flask server. If this happens, wait for a few rounds. If it still fails, you can try restarting the server.
- Another useful command-line tool is **openssl**. You can use **openssl** to validate the server as well. If it succeeds in connecting to the HTTPS server, the following command will show its certificate with a success message:

```
$ openssl s_client -showcerts -connect 192.168.1
CONNECTED(00000003)
<more logs and certificate>
```

Let's see how to do OTA update by using the RainMaker platform next.

Utilizing RainMaker for OTA updates

RainMaker is a cloud platform by Espressif Systems that you can connect to your devices and manage remotely. The underlying cloud infrastructure is provided by **Amazon Web Services** (**AWS**) and RainMaker runs on top of AWS. RainMaker integrates many important features that we can expect from an IoT platform, such as user management, device management, scheduling for automation, monitoring and data analysis, and diagnostics. Espressif provides an instance of the RainMaker platform free for learning and testing purposes, but it is also available on the AWS Marketplace if you want to build your product around this platform (https://aws.amazon.com/marketplace/pp/prodview-sre2djwuggnyw). You can find more information about RainMaker on its website here: https://rainmaker.espressif.com/.

Although we will extensively discuss how to develop ESP32 applications in a cloud environment in the upcoming chapter, working on a RainMaker application provides us with a good opportunity to learn about the common features of a secure IoT product in general. In this example, we will use the RainMaker platform to upgrade the firmware on *ESP32-C3 DevKitM-1*. The RainMaker library documentation is here as a reference during the development of this example: https://docs.espressif.com/projects/esp-rainmaker/en/latest/index.html.

First, let's get prepared for development. We need to clone the RainMaker repository and install the mobile application to access our devices.

Configuring RainMaker

Here are the steps to configure RainMaker in our example:

1. Clone the RainMaker repository from GitHub (if you have cloned the book repository, RainMaker comes with it as a sub-module in `ch7/common`):

```
$ git clone --recursive https://github.com/espre
```

2. It has a command-line utility developed in Python under the `cli` directory. Create a virtual environment for it and install the Python modules required by the tool.

```
$ cd esp-rainmaker/cli
$ pyenv virtualenv  3.8.0 rmaker_cli && pyenv lc
$ pip install --upgrade pip && pip install -r re
```

3. The utility needs to know the ESP-IDF path and looks for it in the environment variables. You can run it after setting the `IDF_PATH` environment variable to see the options.

```
$ export IDF_PATH=$HOME/esp/esp-idf
```

```
$ python ./rainmaker.py
```

4. Run the utility with the `signup` option to create a RainMaker account. Then, log in with that account.

```
$ python ./rainmaker.py signup <email>
$ python ./rainmaker.py login --email <email>
```

The RainMaker account is now ready. It is time to install the mobile application, **ESP RainMaker**, on your mobile device. We will use this application to provision a new node in the local WiFi network and associate it with our account. It is available for both Android and iOS devices in their respective application stores. After the installation, log in to the application with the same credentials that you created in *step 4*.

Next, we can create an ESP-IDF project.

Creating a project

After installing the tools, we can move on to application development. Let's create the project as follows:

1. Create an ESP-IDF project from the command line (you can choose any other way, as described in *Chapter 2, Understanding the Development Tools*). You will also need to set the RainMaker path as an environment variable if you have cloned the RainMaker repository yourself.

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project rmaker_ota_ex
$ export RMAKER_PATH=<rmaker_path>
```

2. We need a project `CMakeLists.txt` file that sets the extra component directories and firmware version as `PROJECT_VER`:

```
cmake_minimum_required(VERSION 3.5)
if(DEFINED ENV{RMAKER_PATH})
  set(RMAKER_PATH $ENV{RMAKER_PATH})
else()
  set(RMAKER_PATH ${CMAKE_CURRENT_LIST_DIR}/../c
endif(DEFINED ENV{RMAKER_PATH})
set(EXTRA_COMPONENT_DIRS ${RMAKER_PATH}/componen
set(PROJECT_VER "1.0")
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(rmaker_ota_ex)
```

3. We have a long list of default configuration options. Copy `sdkconfig.defaults` from the repository into your project. There are some interesting options in it, which we will discuss in a while.
4. Create a `partitions.csv` file with the following content. It has two partitions for the OTA upgrade. It also contains a partition, `fctry`, for cryptographic data (keys and certificates).

```
nvs,      data, nvs,     0x10000,   0x6000,
otadata,  data, ota,     ,          0x2000
phy_init, data, phy,     ,          0x1000,
```

```
ota_0,    app,  ota_0,   0x20000,   1600K,
ota_1,    app,  ota_1,   ,          1600K,
fctry,    data, nvs,     0x340000,  0x6000
```

5. Rename the application source code to `main/rmaker_ota_ex.cpp` and update the `main/CMakeLists.txt` file accordingly:

```
idf_component_register(SRCS ./rmaker_ota_ex.cpp
```

Now that we have the ESP-IDF configured, we can get to coding.

Coding the application

The first C++ header file that we are going to add is responsible for the WiFi initialization. We create a new file, rename it to `main/AppDriver.hpp`, and develop the code as follows:

```
#pragma once
#include <esp_log.h>
#include <nvs_flash.h>
#include <app_wifi.h>
```

First, we add the header files that we need in this source code. `app_wifi.h` comes with RainMaker to ease the WiFi provisioning in a very similar way as we did in the previous chapters. Then, we can continue with the `AppDriver` class:

```
namespace app
{
    class AppDriver
    {
    public:
        void init()
        {
            esp_err_t err = nvs_flash_init();
            if (err == ESP_ERR_NVS_NO_FREE_PAGES ||
                == ESP_ERR_NVS_NEW_VERSION_FOUND)
            {
                nvs_flash_erase();
                nvs_flash_init();
            }
            app_wifi_init();
        }
```

In the `init` function of the class, we begin with initializing the `NVS` partition, which has the WiFi credentials on it if configured earlier. Then, we call the `app_wifi_init` function of the WiFi library of RainMaker to initialize the WiFi stack. Next, we will add another public function to start the WiFi:

```
        void start()
        {
            app_wifi_start(POP_TYPE_RANDOM);
        } // function end
```

```
    }; // class end
} // namespace end
```

The `start` function is quite simple; we only call `app_wifi_start` of RainMaker. It checks the provisioning status of the device in a WiFi network and, if it is not provisioned, it shows a QR code to be used with the ESP RainMaker mobile application to pass the local WiFi credentials. This finishes the implementation of the `AppDriver` class.

The next class that we will develop is `AppNode` in the `main/AppNode.hpp` file. It is the class where we implement the actual RainMaker functionality and OTA service. We include the header files first as usual:

```
#pragma once
#include <cinttypes>
#include <esp_log.h>
#include <esp_event.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <esp_rmaker_core.h>
#include <esp_rmaker_standard_types.h>
#include <esp_rmaker_standard_params.h>
#include <esp_rmaker_standard_devices.h>
#include <esp_rmaker_ota.h>
#include <esp_rmaker_common_events.h>
```

The names of the header files are quite self-explanatory, but as a quick overview, `esp_rmaker_core.h` contains the functions to define a RainMaker device that can communicate with the RainMaker platform. To define the device specifics, such as a temperature sensor or a plug, we use the definitions and functions in the `esp_rmaker_standard_*` header files. The `esp_rmaker_ota.h` header exposes the OTA functionality of the library, hence the name. Let's define the class and start with its `private` section:

```
namespace app
{
    class AppNode
    {
    private:
        esp_rmaker_node_t *m_rmaker_node;
        esp_rmaker_device_t *m_device;
        static void event_handler(void *arg,
            esp_event_base_t event_base, int32_t eve
            void *event_data);
```

Node and **device** are two important building blocks of a RainMaker application. A device defines the interface for users, such as a temperature sensor or a plug. A node defines a single node on the RainMaker platform and is a container for devices – that is, a node can contain one or more devices. Therefore, we can define combined devices in a single node. After declaring the pointers to a node and device, we have a static function to handle RainMaker events. We

will see its body after the class definition. Next comes the `public` section of `AppNode`:

```
    public:
        void init()
        {
            esp_event_handler_register(RMAKER_EVENT
                ESP_EVENT_ANY_ID, &event_handler, nu
            esp_event_handler_register(RMAKER_OTA_EV
                ESP_EVENT_ANY_ID, &event_handler, nu
```

In the `init` function, we begin with registering two events: RainMaker events and OTA events. These events allow us to know about the changes in the RainMaker state. Then, we can create the node:

```
            esp_rmaker_config_t rainmaker_cfg = {
                .enable_time_sync = false,
            };
            m_rmaker_node = esp_rmaker_node_init(&ra
                "A node", "OtaNode");
```

We create the node by calling the `esp_rmaker_node_init` function with a configuration variable. The function also takes the node name and type as parameters. These properties are used in the internal representation of a node and listed when we query the RainMaker platform with the CLI tool. We then define the device as the following:

```
            m_device = esp_rmaker_device_create("Ota
                ESP_RMAKER_DEVICE_OTHER, nullptr);
            esp_rmaker_device_add_param(m_device,
                esp_rmaker_name_param_create(
                    ESP_RMAKER_DEF_NAME_PARAM, "My d
            esp_rmaker_node_add_device(m_rmaker_node
```

We create the device by calling `esp_rmaker_device_create`. It creates the device with an internal name (`"OtaDevice"`), type (`ESP_RMAKER_DEVICE_OTHER`), and a pointer to custom data attached to it, which is `nullptr` in our case. RainMaker has a predefined list of devices but, basically, you can define anything as a device type. Then, we attach a parameter to the device with the help of the `esp_rmaker_name_param_create` and `esp_rmaker_device_add_param` functions. It is the `name` property of the device that is displayed on the mobile application for end-users. A user can change this parameter via the GUI to customize the device name. Finally, we call the `esp_rmaker_node_add_device` function to link the device and the node. The next step is to define the OTA service:

```
            esp_rmaker_ota_config_t ota_config = {
                .server_cert = ESP_RMAKER_OTA_DEFAUL
            };
            esp_rmaker_ota_enable(&ota_config, OTA_U
        } // function end
```

The function to enable the OTA service of the application is `esp_rmaker_ota_enable`. The configuration includes the certificate of the OTA server running on the RainMaker platform. This certificate comes with the RainMaker library and is embedded into the flash image at compile time. We've finished the `init` function and it will provide us with a fully-fledged RainMaker node with the OTA update capability. The next function of the class starts the RainMaker task:

```
        void start()
        {
            esp_rmaker_start();
        } // function end
    }; // class end
```

In the `start` function, we call `esp_rmaker_start` of RainMaker to enable it to process all the messages and commands. It also generates events to deliver its internal state in the application. The class definition is completed but the implementation is not finished yet. We will implement the `event_handler` function next:

```
    void AppNode::event_handler(void *arg,
        esp_event_base_t event_base, int32_t event_
        void *event_data)
    {
        if (event_base == RMAKER_EVENT)
        {
            switch (event_id)
            {
```

```
                case RMAKER_EVENT_INIT_DONE:
                    ESP_LOGI(__func__, "RainMaker Initi
```

In the event handler, we simply print the events to see them on the serial console. We had registered the `RMAKER_EVENT` and `RMAKER_OTA_EVENT` events in the `init` function of the class, thus we will receive them through this function. The function is a bit lengthy, so please refer to the repository to see the entire function body. It doesn't do anything special but only prints event descriptions. Its output will help us to understand the inner workings of the underlying RainMaker task.

The last development in the project is the `app_main` function of the application where we integrate the classes, in `main/rmaker_ota_ex.cpp`:

```
#include "AppDriver.hpp"
#include "AppNode.hpp"
namespace
{
    app::AppDriver app_driver;
    app::AppNode app_node;
}
```

After including the header files, we instantiate the classes in an anonymous namespace. Then we can just use them in the `app_main` function as follows:

```
extern "C" void app_main()
{
    app_driver.init();
    app_node.init();
    app_node.start();
    app_driver.start();
}
```

We initialize the objects and then call the `start` functions of them. The order here is important. The call to the `init` function of the `app_driver` object initializes the underlying flash/NVS access and WiFi, and then `app_node`'s `init` configures RainMaker accordingly. We then call the `start` function of `app_node` to allow the RainMaker task to monitor WiFi events and react to them. After that, we start WiFi by running `app_driver`'s `start`.

The coding is done. Let's flash the application and see how it works.

Testing the application

We erase the flash first, to remove any WiFi credentials left on the flash memory, and then flash the application:

```
$ idf.py erase-flash flash monitor
Executing action: erase-flash
Serial port /dev/ttyUSB0
Connecting....
```

```
Detecting chip type... ESP32-C3
<more logs>
```

We see some interesting logs after flashing the devkit successfully:

```
I (586) esp_claim: Initialising Self Claiming. This
W (606) esp_claim: Generating the private key. This
I (7906) event_handler: RainMaker Initialised.
I (7906) esp_rmaker_node: Node ID ----- A0764E76F90(
```

In RainMaker terms, **claiming** means generating a certificate for a device signed by the RainMaker platform so that it can connect to the platform securely. For that, a private and public key pair needs to be generated and a sign request is prepared for the public key. Then ask the RainMaker platform with the **Certificate Signing Request** (**CSR**) to create a TLS certificate and receive the certificate in response to complete the claiming process.

RainMaker defines three types of claiming methods:

- **Self-claiming**: The device itself talks to the RainMaker platform to get a certificate. Only ESP32-C3, ESP32-C6, ESP32-S2, and ESP32-S3 can do that.
- **Assisted claiming**: The ESP RainMaker mobile application helps the device get a certificate from the RainMaker platform. The device creates the private and public key pair, but the mobile application tells the platform to create the certificate.

- **Host-driven claiming**: This corresponds to manual claiming by generating the keys and the device certificate manually on a PC by using the CLI application.

ESP32-C3 (and others listed above) can do claiming itself because it has a shared key with the platform on one of its eFuses to set up an initial secure communication channel. That shared key is generated at manufacturing and burned to the chip. ESP32 has other eFuse memory blocks reserved for our use and we can utilize them in a similar manner when needed. You can see the API documentation here for eFuse management: https://docs.espressif.com/projects/esp-idf/en/v5.0/esp32c3/api-reference/system/efuse.html.

The logs above also show a node ID. It is the unique ID used to access the RainMaker node over the platform when it is registered and associated with a user. For my devkit, the node ID is `A0764E76F90C.`

At the end of the logs, we see a QR code for the mobile application:

Figure 7.1: Provisioning a new RainMaker node

Then, we run the ESP RainMaker mobile application and scan the QR code to continue with the provisioning:

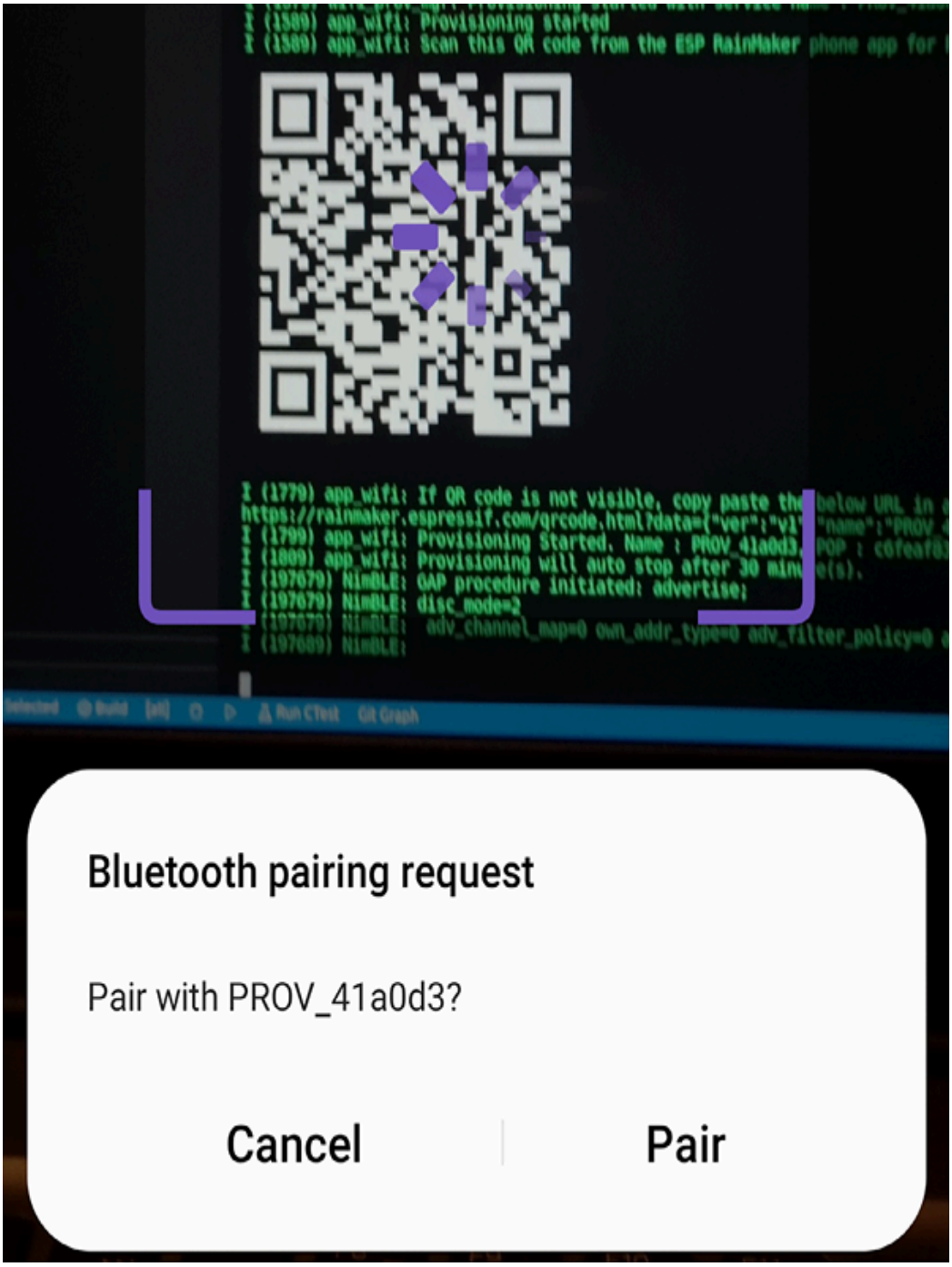


Figure 7.2: Provisioning on the mobile application

We pass the WiFi credentials to the devkit and it uses this information to connect to the local WiFi. At this point, **user-node mapping** happens. Our accounts also have a unique ID on the

platform and a secret key. The mobile application shares them with the devkit and the devkit communicates further with the platform to associate itself with our account. Now, the entire provisioning process is complete and we can see the devkit on the mobile application under our account with the name **My device**, as we configured it in the firmware.

It is all good, but what happened to all these WiFi credentials, private keys, and certificates? They're saved on the flash. As long as the flash is physically intact, the devkit will be able to connect to the RainMaker platform even after going through a full power cycle. The `nvs` partition keeps the WiFi credentials as usual but the private key, the TLS certificate signed by RainMaker, and the user mapping record are saved in the `fctry` partition that we defined in the `partitions.csv` file of the project.

Another question at this point is: which library provides the security functionality? It is **MbedTLS** by default in ESP-IDF. When we check the `sdkconfig.defaults` file of the project, we can see some configuration entries for it:

```
CONFIG_MBEDTLS_DYNAMIC_BUFFER=y
CONFIG_MBEDTLS_DYNAMIC_FREE_PEER_CERT=y
CONFIG_MBEDTLS_DYNAMIC_FREE_CONFIG_DATA=y
CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_CMN=y
```

The first three entries in `sdkconfig.defaults` configure some memory management parameters of MbedTLS. The last one commands MbedTLS to include the common certificate bundle. To

verify a TLS server, **X509 root certificates** are needed. A root certificate helps a TLS client verify that the server it wants to connect to has a valid certificate registered by a **Certificate Authority** (**CA**). This bundle contains the certificates of the common CAs that come from **Mozilla's NSS root certificate store**.

ESP-IDF has an abstraction for the TLS cryptographic functionality in its **ESP-TLS** library so that we can have other cryptographic libraries, such as **wolfSSL**, easily integrated into our applications. (For more information about ESP-TLS: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_tls.html)

We have our devkit connected to the RainMaker platform. It means that we can now test the OTA service in the following steps:

1. Create a new version of the application by simply setting a new version number in the root `CMakeLists.txt` file of the project:

```
set(PROJECT_VER "1.1")
```

2. Compile the application to generate a new binary:

```
$ idf.py build
```

3. In another terminal window, run the RainMaker CLI tool to verify the node is connected:

```
(rmaker_cli) $ python ./rainmaker.py getnodes
A0764E76F90C
(rmaker_cli) $ python ./rainmaker.py getnodeconf
{
  "fw_version": "1.0",
  "model": "rmaker_ota_ex",
  "name": "A node",
  "platform": "esp32c3",
  "project_name": "rmaker_ota_ex",
  "type": "OtaNode"
}
```

4. Run the CLI tool to upgrade the firmware to version 1.1:

```
(rmaker_cli) $ python ./rainmaker.py otaupgrade
Checking esp.service.ota in node config...
Uploading OTA Firmware Image...This may take tim
Setting the OTA URL parameter...
OTA Upgrade Started. This may take time.
Getting OTA Status...
[04:08:13] in-progress : Downloading Firmware Im
[04:08:54] in-progress : Rebooting into new firm
[04:09:11] success : OTA Upgrade finished and ve
```

5. At the same time, we observe that the devkit receives the new firmware and reboots automatically to activate it:

```
I (483570) event_handler: Starting OTA.
I (483580) esp_rmaker_ota: Reporting in-progress
```

```
I (483580) esp_rmaker_param: Reporting params: {
I (483590) esp_rmaker_param: Reporting params: {
I (483600) event_handler: OTA is in progress.
W (483600) esp_rmaker_ota: Starting OTA. This ma
I (484400) esp-x509-crt-bundle: Certificate vali
I (485530) esp_https_ota: Starting OTA...
I (485530) esp_https_ota: Writing to partition s
I (485530) wifi:Set ps type: 0
I (485550) esp_rmaker_ota: Reporting in-progress
I (485550) esp_rmaker_param: Reporting params: {
I (485560) esp_rmaker_param: Reporting params: {
I (485560) event_handler: OTA is in progress.
I (487150) esp_rmaker_ota: Image bytes read: 504
<more logs, then reboots>
I (392) cpu_start: Application information:
I (395) cpu_start: Project name:     rmaker_ota_
I (400) cpu_start: App version:      1.1
I (405) cpu_start: Compile time:     Feb 20 2023
I (411) cpu_start: ELF file SHA256:  10fbd644f90
I (417) cpu_start: ESP-IDF:          v4.4.2
<more logs>
I (4458) esp_rmaker_ota: Reporting success: OTA
I (4528) event_handler: OTA successful.
```

6. When we query it over the RainMaker platform via the CLI tool, it reports its version as `1.1`.

```
(rmaker_cli) $ python ./rainmaker.py getnodeconf
```

```
"1.1"
```

That is it! We have the most basic RainMaker node with the OTA service enabled on it. You can try other scenarios, such as sending the same firmware or broken firmware, to see how the application behaves in such circumstances.

The OTA upgrade is one of the most important security features of any IoT device before launching a product, so it is better to make sure it works and it works securely. It is the method to patch any issues or add new features remotely without the need for physical access, which saves time, money, reputation, and eventually, the product itself.

Troubleshooting

The application should work without any problem if the RainMaker configuration is done correctly. The only thing is that during WiFi provisioning, if the mobile application gives an error about pairing with the devkit, just retry by scanning the QR code again.

In the next example, we will dig further into RainMaker to learn how we can exchange data with the RainMaker platform over a secure MQTT connection by defining a new RainMaker node in the application.

Sharing data over secure MQTT

A paramount concept in cyber security is how to secure data in transit and at rest. Data at rest means any information that is stored on a non-volatile memory, such as a flash or hard drive. Data can be structured data, for example, a SQL database, or any type of file. Data in transit means bytes that are being transferred over a medium, such as a wireless network. These definitions perfectly apply to IoT. Data is at every step of an IoT product, starting from collecting environmental data via sensors and transferring them to a backend or cloud service for further processing and storage. It doesn't stop there; we would need to share it with other endpoints, such as mobile applications. The nature of data affects the product design decisions at every single step. Below is a list of items along with questions that we can ask during the design phase to help reveal the nature of the data:

- **The type, frequency, and volume of data collected by sensors**: Is there any need for local storage to provide a specific feature at the edge (data at rest)?
- **The type**, **frequency, and volume of data to be transferred to the backend system**: What type of medium should be used to transfer data (wireless, wired, or even the storage type of the transfer medium in some cases)? Which transport layer and application layer protocols (HTTP, MQTT, CoAP, WebSocket, etc.) should you select in order to meet the requirements?
- **The type and volume of data to be processed and stored in the backend system**: What are the performance requirements to access data? What are the authentication and authorization considerations?

- **The type, frequency, and volume of data to be transferred to the endpoints**: What are the application layer protocols?
- **The type and volume of data to be processed at the endpoints, such as mobile applications**: Are there any local storage requirements?

Although these items are only a brief list of issues while collecting data requirements, they reveal a common shared goal: to prevent any unauthorized access to data *everywhere*, both in transit and at rest.

In this example, we will develop an application on *ESP32-C3 DevKitM-1* for passing light readings from an LDR to the RainMaker backend over secure MQTT. Of course, RainMaker is not an ultimate solution for every IoT product (in fact, it only uses the smart home scenarios as a showcase); nonetheless, it is quite helpful to attain a general understanding of the requirements for real-world projects.

As we discussed earlier, ESP RainMaker runs on top of AWS IoT Core and utilizes its features to provide a secure and reliable backend system. In the application, we will use the default RainMaker settings for communication (or data in transit) security, which is MQTT with the X.509 client certificate authentication on port 443. With this option, the RainMaker library uses mutual authentication over TLS between a node and the backend. The client key and certificate are generated during **node claiming** (this concept is explained in more detail in the previous section, *Utilizing RainMaker for OTA updates*).

Let's start to discuss the example project. If you don't have the RainMaker library and/or a RainMaker account, please refer to the

previous topic to prepare for this example.

The hardware components of the project are:

- ESP32-C3 DevkitM-1
- An LDR
- A pull-up resistor (10KΩ)

The Fritzing sketch that shows the connections is:

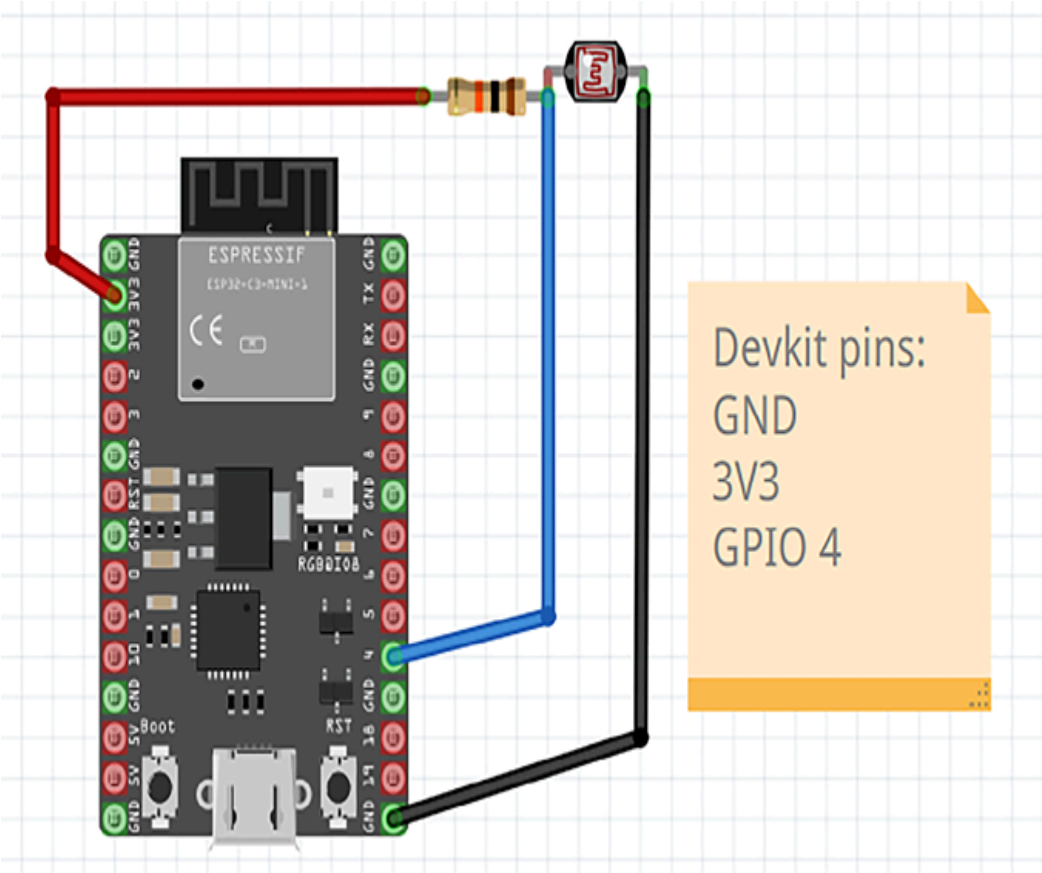


Figure 7.3: Fritzing sketch of the project

After having this hardware set up, we can create an ESP-IDF project.

Creating a project

Let's create and configure the project as follows:

1. Create an ESP-IDF project from the command line (you can choose any other way, as described in *Chapter 2, Understanding the Development Tools*). You will also need to set the RainMaker path as an environment variable if you have cloned the RainMaker repository yourself.

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project rmaker_ota_ex
$ export RMAKER_PATH=<rmaker_path>
```

2. Copy `sdkconfig.defaults`, `CMakeLists.txt`, and `partitions.cvs` from the project repository into the root folder of the project.
3. Rename the application source code to `main/rmaker_mqtt_ex.cpp` and update the `main/CMakeLists.txt` file accordingly:

```
idf_component_register(SRCS ./rmaker_mqtt_ex.cpp
```

The ESP-IDF project is ready to develop the application next.

Coding the application

Now, it is time for coding. Let's add a C++ header file, `main/AppDriver.hpp`, for NVS and WiFi:

```
#pragma once
#include <esp_log.h>
#include <nvs_flash.h>
#include <app_wifi.h>
namespace app
{
    class AppDriver
    {
    public:
        void init()
        {
            esp_err_t err = nvs_flash_init();
            if (err == ESP_ERR_NVS_NO_FREE_PAGES ||
                ESP_ERR_NVS_NEW_VERSION_FOUND)
            {
                nvs_flash_erase();
                nvs_flash_init();
            }
            app_wifi_init();
        }
        void start()
        {
            app_wifi_start(POP_TYPE_RANDOM);
        }
    };
}
```

In fact, this is the same class that we developed in the RainMaker OTA example. In the `init` function, we initialize the NVS and WiFi.

The `start` function simply starts WiFi or manages provisioning and RainMaker claiming when the application runs for the first time.

The second class that we are going to add defines the sensor client interface. The sensor object will notify its clients via this interface. Let's add and edit `main/AppSensorClient.hpp`:

```
#pragma once
#include <cinttypes>
namespace app
{
    class AppSensorClient
    {
    public:
        virtual void update(uint32_t light_level) =
    };
} // namespace end
```

Any class that implements this interface must define the `update` function. A sensor object will pass light readings by calling the `update` function of its client.

We can now define the `AppSensor` class in `main/AppSensor.hpp`:

```
#pragma once
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include "esp_log.h"
#include "driver/adc.h"
#include "AppSensorClient.hpp"
```

```
namespace app
{
    class AppSensor
    {
    private:
        AppSensorClient *m_client;
        const adc1_channel_t m_adc_ch{ADC1_CHANNEL_4
```

The `AppSensor` class defines a member variable, `m_adc_ch`, for the ADC channel at `ADC1_CHANNEL_4`, which corresponds to the GPIO-4 pin of the devkit. It will read from the LDR connected to GPIO-4, and pass the reading to the sensor client as pointed to by `m_client`. Next are the function declarations for sensor readings:

```
        static void readTask(void *arg);
        uint32_t readLight(void);
```

The `readTask` function is for the FreeRTOS task. We will do periodic readings inside it. The `readLight` function reads values from the LDR. We can now move on to the `public` member functions:

```
    public:
        void init(app::AppSensorClient *cl)
        {
            m_client = cl;
            adc1_config_width(ADC_WIDTH_BIT_12);
```

```
            adc1_config_channel_atten(m_adc_ch, ADC_
        }
```

In the `init` function, we initialize the ADC channel width to 12 bits and set the channel attenuation to 11 dB in order to increase the measurable voltage range as a basic signal conditioning before sampling. The 12 bits of channel width means that the readings coming from the ADC channel will be in the range of 0 and 2^12-1 (4095). Next, we define the `start` function of the class where we create the FreeRTOS task:

```
        void start(void)
        {
            xTaskCreate(readTask, "sensor", 2048, th
                nullptr);
        }
    }; // class end
```

The FreeRTOS task takes the `readTask` function that we have declared in the `private` section of the class as a parameter. We also pass the `this` pointer to be able to access the `AppSensor` instance at runtime. We can continue with the `readTask` function body next:

```
    void AppSensor::readTask(void *arg)
    {
        AppSensor *obj = reinterpret_cast<AppSensor
        while (1)
```

```
        {
            vTaskDelay(5000 / portTICK_PERIOD_MS);
            obj->m_client->update(obj->readLight());
        }
    }
```

In the `readTask` function, we take a reading by calling the `readLight` function of the current `AppSensor` instance every 5 seconds and pass the reading to the sensor client by calling its `update` function. The last implementation in this source code file is the `readLight` function body, as follows:

```
    uint32_t AppSensor::readLight(void)
    {
        uint32_t adc_val{0};
        for (int i = 0; i < 32; ++i)
        {
            adc_val += adc1_get_raw(m_adc_ch);
        }
        adc_val /= 32;
        return adc_val;
    }
} // namespace end
```

The `readLight` function has a `for` loop to read from the ADC channel. It calls the `adc1_get_raw` function 32 times in the loop and takes the average to have a better approximation of the light value coming from the LDR. This completes the `AppSensor` class

implementation, and we can develop the RainMaker node in `main/AppNode.hpp`:

```
#pragma once
#include <cinttypes>
#include <map>
#include <string>
#include <esp_log.h>
#include <esp_event.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <esp_rmaker_core.h>
#include <esp_rmaker_standard_types.h>
#include <esp_rmaker_standard_params.h>
#include <esp_rmaker_standard_devices.h>
#include <esp_rmaker_common_events.h>
#include <esp_rmaker_utils.h>
#include <app_insights.h>
#include "sdkconfig.h"
#include "AppSensorClient.hpp"
```

We include a bunch of header files. The `esp_rmaker*` files have the structures and declarations that allow us to develop a RainMaker node as in the RainMaker OTA example of the previous topic. The interesting header here is `app_insights.h`. ESP RainMaker has a service named **ESP Insights** and, when enabled, the `Insights` agent running in the background of a node collects information about the node's health and sends this information to the RainMaker backend. This is clearly an important feature for an IoT backend service. We have to know about our IoT devices in the field so that we can

decide the next step in the product life cycle. ESP Insights allows us to see what is going on in the field *after* the product is installed. If we see crashes or unexpected behavior of the RainMaker nodes when they are running in the wild, we can upgrade them with new firmware to make corrections (an extreme example is the Mars rovers, which you can read about here: https://www.computerworld.com/article/2563630/mars-rovers-get-long-distance-os-updates.html). We will discuss what ESP Insights can provide in detail while testing this example application. Let's develop the `AppNode` class next:

```
namespace app
{
    class AppNode : public AppSensorClient
    {
    private:
        esp_rmaker_node_t *m_rmaker_node;
        esp_rmaker_device_t *m_device;
        esp_rmaker_param_t *m_light_param;
```

The `AppNode` class derives from the `AppSensorClient` abstract class to implement its interface. The `AppSensor` instance will pass light readings to `AppNode` via this interface. In the `private` section of the class, we define pointers for the RainMaker node, device, and light parameter. The hierarchy of a RainMaker node is that a node definition can include multiple device definitions (for instance, a node can integrate a light sensor and a bulb on the same hardware) and a device definition can include multiple parameter definitions (for instance, a multisensor device with temperature, humidity, and

light parameters). In our RainMaker node, we will have a single device and a single parameter. We finish the private section with the following:

```
        bool m_connected{false};
        static void event_handler(void *arg,
            esp_event_base_t event_base, int32_t eve
            void *event_data);
```

The `m_connected` member variable shows whether the node is connected to the RainMaker MQTT service. By checking this flag, we decide if we can publish the light readings or not. The `event_handler` function is for the RainMaker events. We will handle MQTT `connected` and `disconnected` events to set the value of the `m_connected` member variable.

We will develop the `init` function of the class in the `public` section next:

```
    public:
        void init()
        {
            esp_event_handler_register(RMAKER_COMMON
                ESP_EVENT_ANY_ID, &event_handler, th
```

The first thing that we do in the `init` function is register the event handler. We also pass the `this` pointer as a parameter so that we

can access the `m_connected` member variable of the class instance. Then, we can create a RainMaker node as follows:

```
            esp_rmaker_time_set_timezone (
                CONFIG_ESP_RMAKER_DEF_TIMEZONE);
            esp_rmaker_config_t rainmaker_cfg = {
                .enable_time_sync = true,
            };
            m_rmaker_node = esp_rmaker_node_init(&ra
                "A light-sensor node", "LightSensorN
```

We can associate readings with a timestamp. To do that, we need to set a time zone and enable the time synchronization of ESP32. The time zone comes from `sdkconfig` and you can update it to your time zone by running `menuconfig` (the time zones are listed here: https://rainmaker.espressif.com/docs/time-service.html). When the time synchronization is enabled, the RainMaker library connects to the configured SNTP server to get the time. You can see it in `menuconfig` at **(Top) | Component config |ESP RainMaker Common**. We call the `esp_rmaker_node_init` function to create the RainMaker code with this configuration.

We will create the light device next:

```
            m_device = esp_rmaker_device_create("Lig
                ESP_RMAKER_DEVICE_LIGHT, nullptr);
            esp_rmaker_device_add_param(m_device,
```

```
                esp_rmaker_name_param_create(
                  ESP_RMAKER_DEF_NAME_PARAM, "Light
```

We create the device by calling the `esp_rmaker_device_create` function and attaching a `name` parameter to the device. The value of the `name` parameter is displayed on the mobile application for the device. We also need a `light` parameter, which we will do next:

```
            m_light_param = esp_rmaker_param_create(
                "app.lightlevel", esp_rmaker_int(0),
                 PROP_FLAG_READ | PROP_FLAG_TIME_SER
```

The `esp_rmaker_param_create` function is the one to create a custom parameter. There is no predefined RainMaker parameter for a light sensor, thus we create it manually. The first argument to the function is the parameter name, the second one is the type, then comes the initial value of the parameter with its value type, and the last argument shows the properties of the parameter as a combination of flags. `LightParam` is a read-only parameter and its values will be stored as time-series data on the RainMaker platform. We are not done with the light parameter yet:

```
            esp_rmaker_param_add_ui_type(m_light_par
                ESP_RMAKER_UI_TEXT);
            esp_rmaker_device_add_param(m_device, m_
```

```
            esp_rmaker_device_assign_primary_param(r
                m_light_param);
```

We need to define how to render the parameter value on the mobile application. For that, we call the `esp_rmaker_param_add_ui_type` function with the `ESP_RMAKER_UI_TEXT` argument, which means that the value will be shown as simple text on the UI. After that, we associate the light parameter with the light device with the help of the `esp_rmaker_device_add_param` function. We also assign the light parameter as the primary parameter of the device. This also helps the mobile application render the device view on the GUI. We finish the `init` function with the following:

```
            esp_rmaker_node_add_device(m_rmaker_node
            app_insights_enable();
        } // function end
```

We linked the parameter and the device but not the device and the node. We call `esp_rmaker_node_add_device` for that. This makes our node a proper RainMaker node to interact with over the RainMaker platform. Before the function ends, we also enable the Insights agent by calling the `app_insights_enable` function so that the node can report its status to the backend. The next member function of the class is `start`:

```
        void start()
        {
            esp_rmaker_start();
        }
```

In the `start` function, we simply call `esp_rmaker_start` to enable the node so that it can start its operation. In the last function of the class, we implement the `AppSensorClient` interface as follows:

```
        void update(uint32_t light_level) override
        {
            if (m_connected)
            {
                esp_rmaker_param_update_and_report(r
                    esp_rmaker_int(light_level));
            }
        } // function end
    }; // class end
```

The `AppSensorClient` base class only requires the implementation of the `update` function. It takes the light level as its only argument and we call the `esp_rmaker_param_update_and_report` to pass the light level to the RainMaker platform if the node is connected. Now that the class definition is done, we can develop the `event_handler` function that helps us to listen to the RainMaker events:

```
    void AppNode::event_handler(void *arg,
        esp_event_base_t event_base, int32_t event_
        void *event_data)
{
    AppNode *obj = reinterpret_cast<AppNode *>(a
    switch (event_id)
    {
    case RMAKER_MQTT_EVENT_CONNECTED:
        obj->m_connected = true;
        break;
    case RMAKER_MQTT_EVENT_DISCONNECTED:
        obj->m_connected = false;
        break;
    default:
        break;
    } // switch end
} // function end
} // namespace end
```

The `event_handle` function has a `switch` statement where we look for MQTT connected and disconnected events. We update the `m_connected` member variable of the `AppNode` instance accordingly.

All the classes of the application are ready to be integrated in `main/rmaker_mqtt_ex.cpp`:

```
#include "AppDriver.hpp"
#include "AppNode.hpp"
#include "AppSensor.hpp"
```

```
#include "AppSensorClient.hpp"
namespace
{
    app::AppDriver app_driver;
    app::AppNode app_node;
    app::AppSensor app_sensor;
}
```

We include the class headers and create instances of them in the anonymous namespace. Then, we implement the `app_main` function as follows:

```
extern "C" void app_main()
{
    app_driver.init();
    app_node.init();
    app_sensor.init(dynamic_cast<app::AppSensorClie
    app_sensor.start();
    app_node.start();
    app_driver.start();
}
```

We call the `init` function of the `driver`, `node`, and `sensor` instances. Then, we call the `start` functions of them in reverse order. Please note that the order is important to catch the events and data properly. The application is ready to test.

Testing the application

Let's flash the application to the devkit and see how it works:

1. First, you need to remove the node from the RainMaker platform if you have already provisioned it in the previous example (*Utilizing RainMaker for OTA updates*). Run the CLI tool for that:

```
(rmaker_cli) $ python ./rainmaker.py removenode
```

2. Flash the application with the `erase-flash` option:

```
$ idf.py erase-flash flash monitor
Executing action: erase-flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<more logs>
```

3. Provision the node as described in the previous example with the help of the ESP RainMaker mobile application.

After provisioning, we will see many logs flowing in the serial console. Let's see what they are:

```
I (67742) esp_mqtt_glue: MQTT Connected
I (67742) esp_rmaker_node_config: Reporting Node Cor
I (67752) esp_rmaker_param: Params MQTT Init done.
I (67752) esp_rmaker_param: Reporting params (init)
```

```
E (67762) esp_rmaker_param: Current time not yet ava
I (67772) esp_rmaker_cmd_resp: Enabling Command-Resp
```

When the MQTT service of the RainMaker platform is connected, the node will try to publish data, but it will fail due to the time synchronization. It takes some time to synchronize with the SNTP server and the node cannot publish its data until the synchronization is done.

We also see some logs from the Insights agent:

```
I (67782) esp_insights: Scheduling Insights timer fo
I (67782) heap_metrics: free:0x2a76c lfb:0x1e800 min
I (67792) wifi_metrics: rssi:-30 min_rssi_ever:-50
```

It runs every 60 seconds to collect metrics about the node. As we see from the logs, memory and WiFi metrics are collected periodically.

When the time synchronization is completed, we see that it starts to publish light data successfully:

```
I (90372) esp_rmaker_time: SNTP Synchronised.
I (90372) esp_rmaker_time: The current time is: Sat
I (91602) esp_rmaker_param: Reporting Time Series Da
I (91602) esp_rmaker_param: Reporting params: {"Ligh
```

Let’s check the mobile application to see the device and its data:

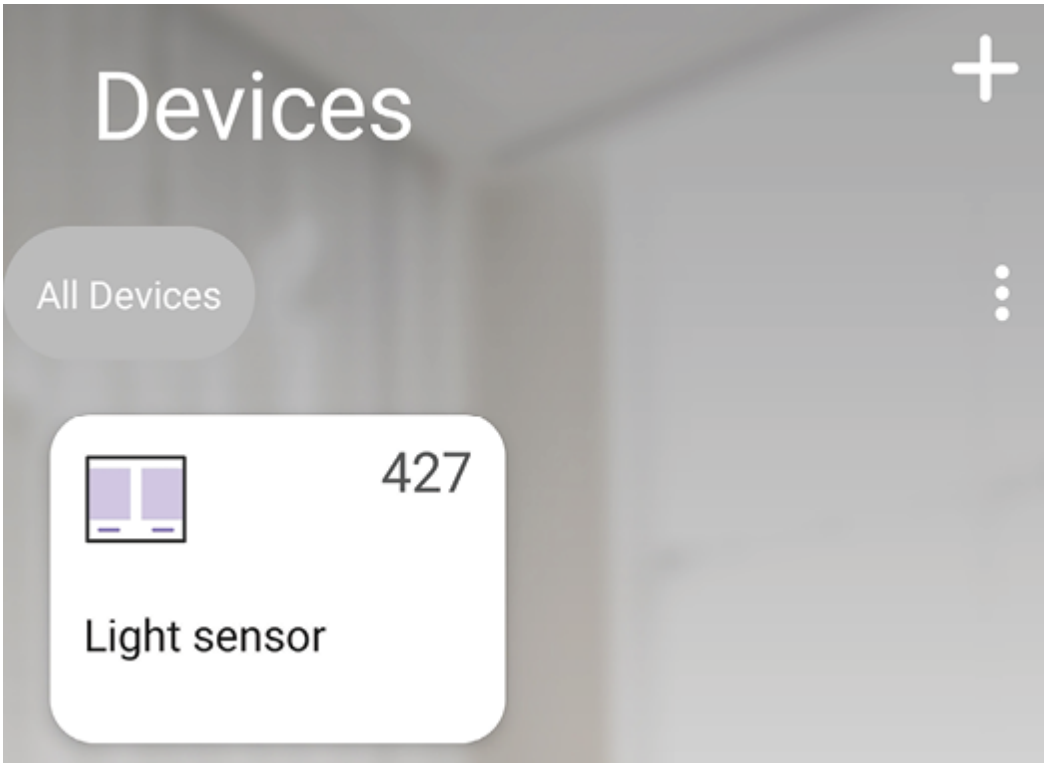


Figure 7.4: Light sensor on the mobile application

The mobile application shows the node as **Light sensor** and the latest light value at the top right of the node icon. When we select the node, we see more details about it:

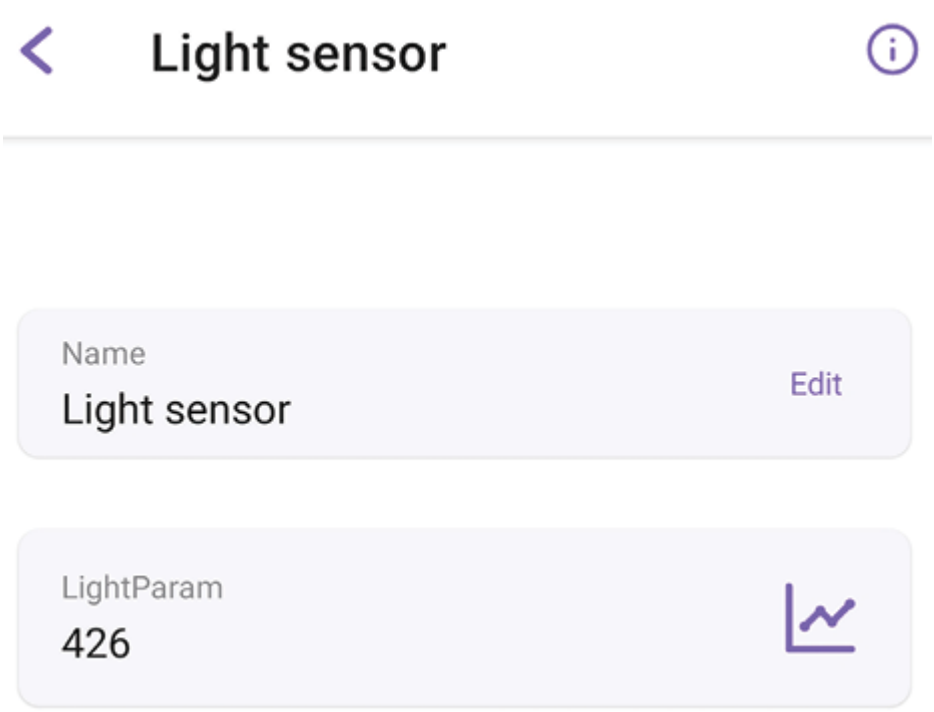


Figure 7.5: The details of the node on the mobile application

We have two parameters on the **Light sensor** as we intended: **Name** and **LightParam**. The **LightParam** parameter is associated with time-series data. The icon to the right of it navigates to its data.

The RainMaker platform also provides a web interface to see the details of our installed nodes at https://dashboard.rainmaker.espressif.com/home/nodes/. After logging in, we can see the list of nodes:

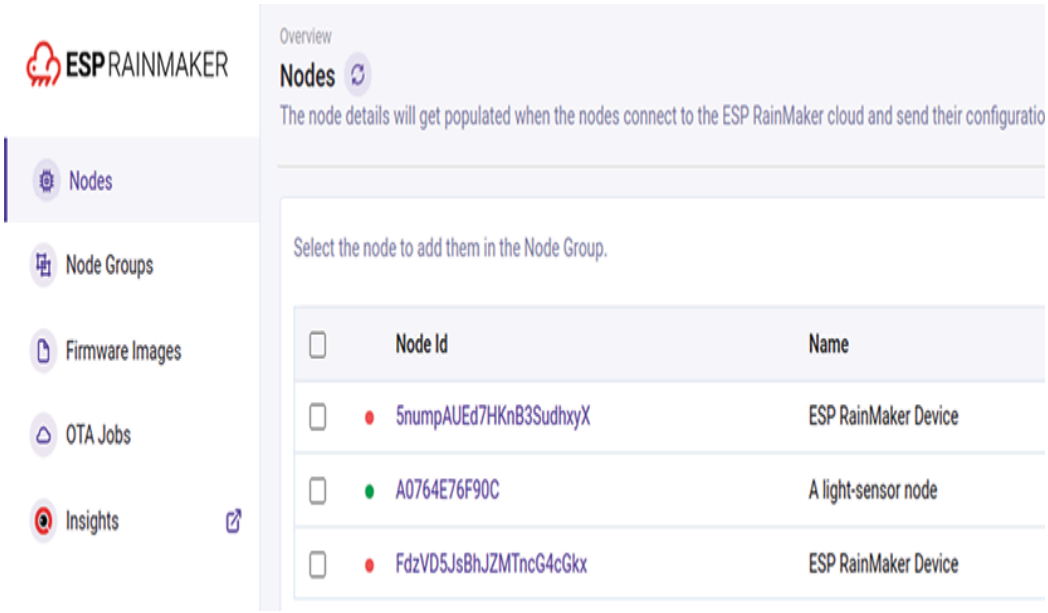


Figure 7.6: The list of nodes

When we select the node from the list, it navigates to **ESP Insights** and shows the node dashboard:

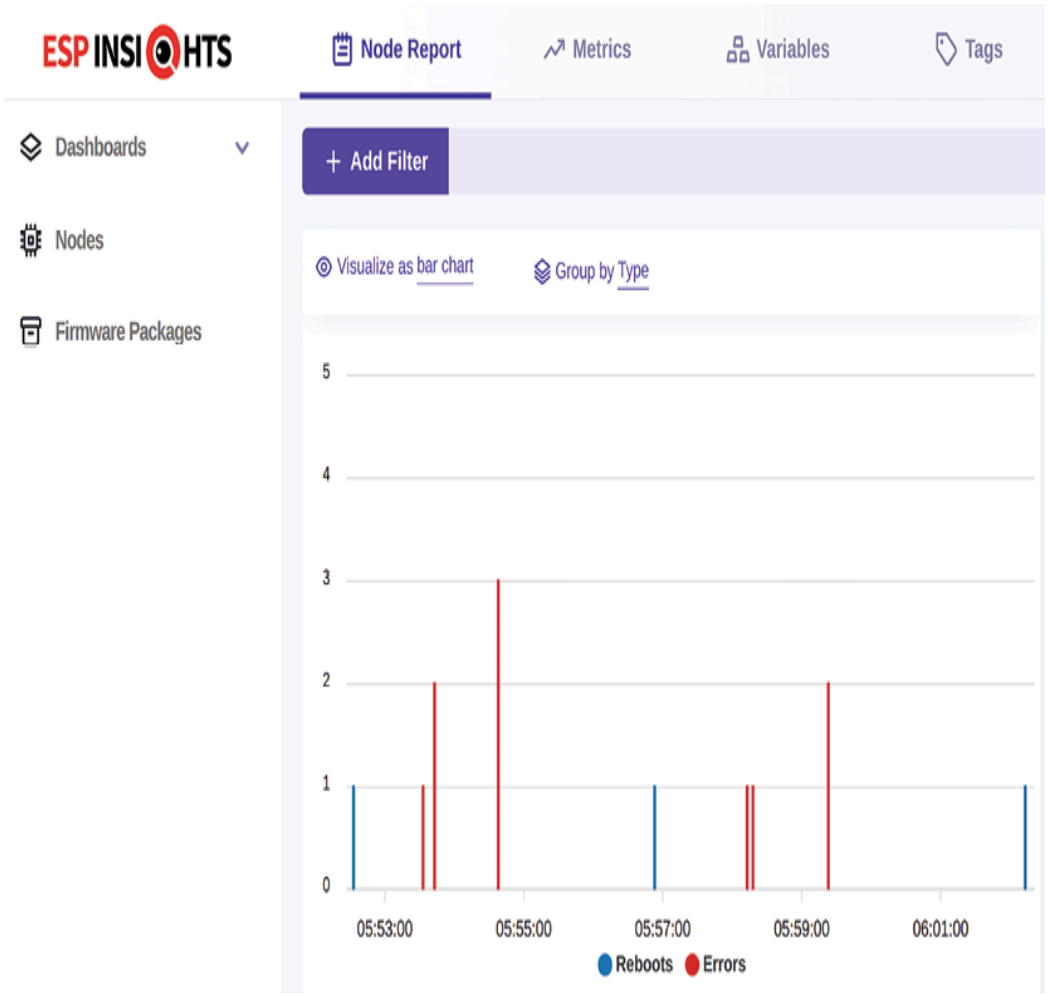


Figure 7.7: Node dashboard on ESP Insights

The dashboard shows the metrics and data shared by the Insights agent running on the node. We can see all the details of the node here, such as *errors*, *crashes*, *reboots*, etc., and also the metrics, such as WiFi and memory statistics:

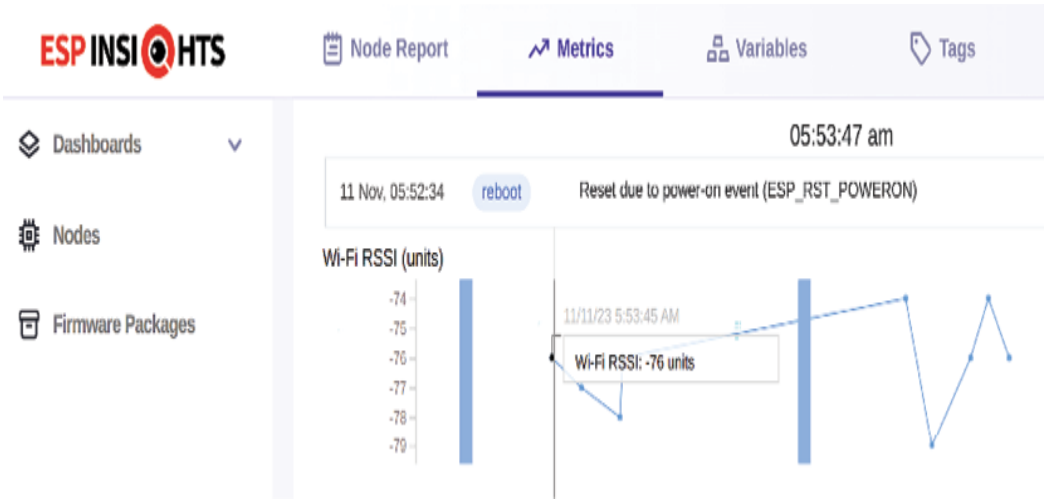


Figure 7.8: Node metrics

You can discover other features of ESP Insights by following the links and documents. It provides us with valuable data to monitor the nodes registered on the RainMaker platform. The takeaway here is that our job doesn't finish when we develop the project and install the IoT devices in the field. We need to have some means to monitor them for any type of maintenance purposes and ESP Insights is a good example of how to do that. More information about ESP Insights is here: https://github.com/espressif/esp-insights/blob/main/FEATURES.md.

Troubleshooting

If you see any unexpected behavior during the provisioning and claiming, take a step back and repeat the process. If it doesn't solve, reboot the devkit and the mobile application and start again. ESP RainMaker is great, but not perfect.

Summary

An IoT product to be launched on the market requires different security approaches combined in the same solution. A solution usually has different components, including connected devices, a backend service, and client applications to interact with the product. When we talk about IoT security, it covers the entire solution with all of these components. In this chapter, we discussed what the ESP32 platform, hardware, and software together provide us to ensure the utmost cybersecurity for our products. We talked about the **Secure Boot** and **flash encryption** features to secure the firmware on ESP32 devices. The newer ESP32 families, such as ESP32-C3, come with a **digital signature** peripheral to secure the application's private keys with a smart technique. **OTA update** is one of the most valuable features of any IoT product.

We have seen two different examples of OTA updates. We learned that we need to monitor our devices in the field where possible. **ESP RainMaker** is a well-designed platform that shows us what an IoT platform should provide to create real-world IoT projects. We developed a light sensor node that runs on RainMaker and discussed **ESP Insights** as an example of a monitoring solution.

Questions

Let's practice what we have learned in this chapter by answering the following questions:

1. When we want to protect the device firmware from any external access, which is the right technique to do that?

a. Flash encryption
b. Secure Boot
c. The digital signature peripheral
d. ESP Privilege Separation

2. Which of the following is false about the OTA update for ESP32?

a. Two partitions are needed.
b. There are many techniques for OTA updates, such as over HTTP, over MQTT, or running a TCP server on the device itself.
c. A secure channel must be used.
d. Application rollback is possible.

3. Which of the following is not true for the RainMaker platform?

a. It runs on top of AWS.
b. It is available on the AWS Marketplace for private installation and customization.
c. It supports OTA updates.
d. It doesn't require mutual authentication for devices.

4. What is the monitoring tool of ESP RainMaker?

a. ESP Privilege Separation
b. ESP Insights
c. ESP-TLS
d. Amazon CloudWatch

5. Which of the following components of an IoT product is the least important when discussing security requirements?

a. Device

b. Backend service
c. Client applications
d. All of them are important

Further reading

- *Practical Internet of Things Security – 2nd Edition*, Brian Russell, Drew Van Duren, Packt Publishing (https://www.packtpub.com/product/practical-internet-of-things-security-second-edition/9781788625821): This book covers all security considerations for IoT products. *Chapter 6*, *Cryptographic Fundamentals for IoT Security Engineering*, explains the fundamentals of cryptography, such as random number generation, symmetric and asymmetric encryption, digital signatures, hashes, and cipher suites.
- *Developing IoT Projects with ESP32, 1st Edition*, Vedat Ozan Oner, Packt Publishing (https://www.packtpub.com/product/developing-iot-projects-with-esp32/9781838641160): *Chapter 6*, *Security First!*, explains how to integrate ESP32 with another secure element by giving an example for Optiga TrustX Security Shield2Go.
- *IoT Security Foundation Guidelines* (https://www.iotsecurityfoundation.org/best-practice-guidelines/): The foundation publishes many guidelines and best practices for IoT security.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

8

Connecting to Cloud Platforms and Using Services

One of the enablers of the **Internet of Things** (**IoT**) is cloud computing. We can do all kinds of magic with ESP32 on a local network. We can collect data, share it between nodes, interact with users via the physical switches and displays of the devices, and add more interesting features based on collective sensor data from the local device network. However, the missing part here is cloud connectivity. We should be able to access our devices remotely from anywhere in the world and analyze device data to gain insights into our product more fruitfully. As a matter of fact, in some cases, the analysis of IoT data and any insight resulting from this analysis can provide more benefits than the direct use of the devices themselves. Cloud technologies make all these benefits available in IoT products.

This chapter explains how to integrate ESP32 into **AWS IoT Core** to benefit from cloud connectivity. After having IoT data on the AWS cloud, we have a whole bunch of other cloud services, such as visualization and analytics, or we can expose an API for end-user applications or other third-party services. We will see an example of how to visualize IoT data with **Grafana**. Voice assistants also come in handy in some projects, and we will learn how to enable **Alexa**

Voice Services (**AVS**) in our ESP32 products as another practical example in the chapter.

In this chapter, we're going to cover the following topics:

- Developing on AWS IoT
- Visualizing with Grafana
- Integrating with Amazon Alexa

Technical requirements

The hardware requirements of the chapter are:

- ESP32-C3 DevkitM-1
- A **Light-Dependent Resistor** (**LDR**), or photo sensor
- A pull-up resistor (10KΩ)
- ESP32-S3 Box Lite
- A DS18B20 temperature sensor (breakout board)
- Jumper wires

On the software side, we will use the **AWS IoT Device SDK** library. The book repository contains it as a sub-module, but you can find it here too: https://github.com/espressif/esp-aws-iot

We will also need some other software tools in the examples. They are:

- **ESP SoftAP Provisioning**: The mobile application by Espressif Systems to provision ESP32 in a WiFi network. It is available for both Android and iOS.
- **curl**: A command-line utility to interact with TCP/IP applications: https://curl.se/

- **Eclipse Mosquitto** client tools: They are useful to test connections to an MQTT broker. The binaries are available here: https://mosquitto.org/download/
- **Amazon Alexa**: The mobile application by Amazon to control and monitor Alexa-enabled devices. It is available for both Android and iOS.

The examples of the chapter are in the GitHub repository here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch8

Developing on AWS IoT

We have many options on the market for cloud connectivity. How to choose the right IoT platform for a project mostly depends on the specific technology requirements, such as how easy it is to develop on the platform, deployment capabilities, operations after deployment, and the ecosystem to extend the product beyond the platform functionality. Based on these criteria, Amazon Web Services provides one of the most prominent IoT platforms with an entire set of products and services. Here is a basic overview of the AWS services and products related to IoT:

- **AWS IoT Device SDK** to connect an IoT device to the AWS cloud: The SDK supports C/C++, Python, Java, and JavaScript. Espressif has forked its version for ESP32 devices. The SDK contains the libraries for the common standards and protocols, such as coreMQTT, coreJSON, and coreHTTP. It also contains other libraries, allowing developers to easily connect their

devices to AWS and manage them, such as AWS IoT Device Shadow, AWS IoT OTA update, and many others. **Amazon FreeRTOS** includes all these libraries and the FreeRTOS kernel to enable developers with RTOS functionality.

- **AWS IoT Core** for MQTT, HTTPS, and LoraWAN communication: On top of the cloud connectivity, IoT Core defines **device shadows** to manage individual device states. **IoT rules** route device data to other AWS services and external HTTP endpoints. IoT Core also allows developers to update device firmware remotely (an OTA upgrade) and run remote **jobs**. On the security side, **AWS IoT Device Defender** provides fleet security by auditing device configurations and detecting anomalies. **AWS IoT Greengrass** brings some of the cloud functionality to the edge devices for in-place processing of data. This way, it is possible to develop ML applications on the edge. **AWS IoT ExpressLink** helps to develop connectivity modules. An ExpressLink module is a ready-to-use cloud connectivity hardware component that can be integrated with the application SoC in IoT devices. For example, we can easily configure ESP32-C3 as an ExpressLink module, enabling the application to run on a host SoC with AWS cloud connectivity.
- **IoT visualization**: For time-series data, **Amazon Managed Grafana** is a handy tool to visualize IoT data. It is capable of filtering and correlating data from different sources, with a broad range of dashboard displays in the same workspace. **QuickSight** is the unified **business intelligence** (**BI**) solution in the cloud.

After this quick introduction of AWS IoT products and services, let's have a practical example where we connect ESP32 to AWS IoT Core and pass sensor data to the cloud.

Hardware setup

In this example, we will use ESP32-C3 DevkitM-1 with an LDR to collect light data, publish the readings to a topic on the MQTT broker, and see them in the **MQTT test client** of AWS IoT Core by subscribing to the same topic. The following Fritzing sketch shows the hardware setup:

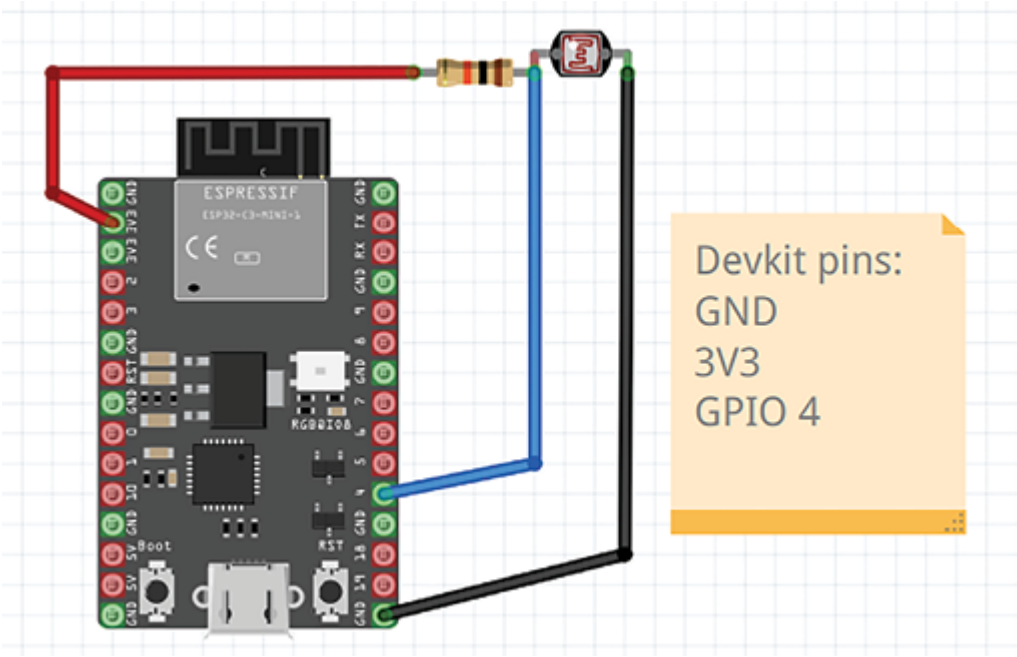


Figure 8.1: Fritzing sketch of the project

Before moving on to the code, we need to create an **AWS IoT thing** from the AWS console.

Creating an AWS IoT thing

IoT things are how actual IoT devices are represented on the AWS cloud. Let's do this in the following steps:

1. Log in to the AWS console by browsing to https://aws.amazon.com/console/
2. Navigate to the *IoT Core* service in your favorite region.
3. Click the Connect device button on the IoT Core homepage to define an AWS IoT thing.

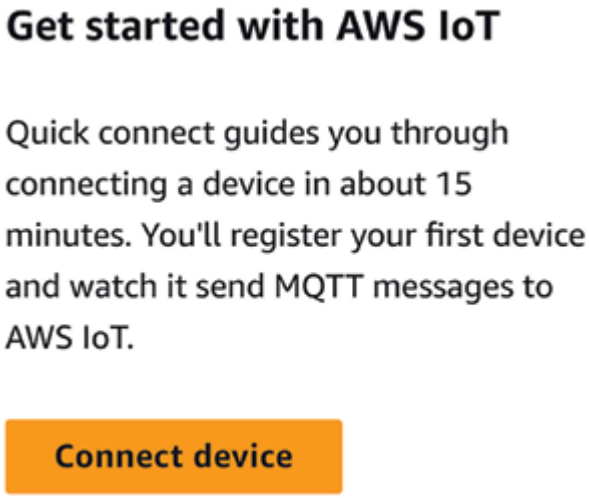


Figure 8.2: The Connect device wizard

4. Enter a name for your device and click **Next**:

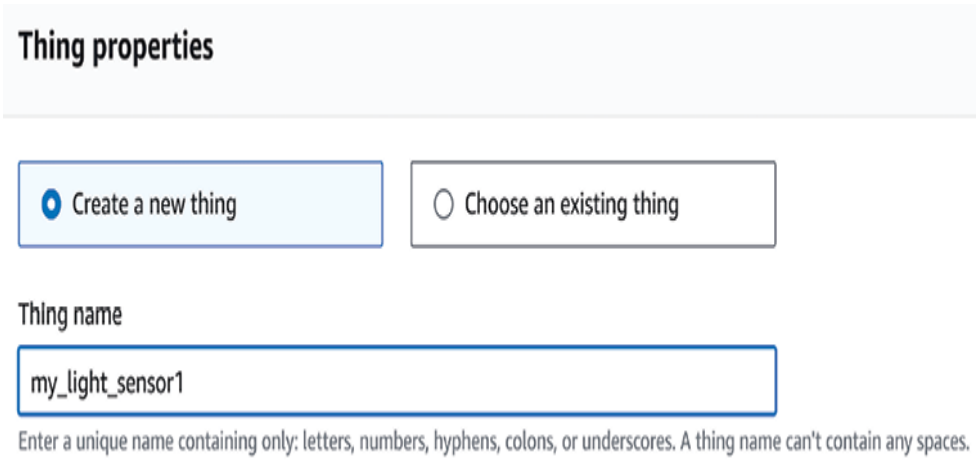


Figure 8.3: Enter a thing name

5. Select the platform and device SDK. You can select anything; we are only interested in creating a thing and its credentials. Then, click **Next**.
6. Download the device kit by clicking on the **Download connection kit** button. It contains all the information and data to connect to AWS IoT Core. Click **Next**.
7. Click **Continue** on the next step of the wizard. When the wizard finishes, you will see a **View thing** button at the end. You can browse the IoT Core thing to see its features.
8. We need to change the thing policy a bit. Navigate to the thing policy that you have just created, and click on the **Edit active version** button.

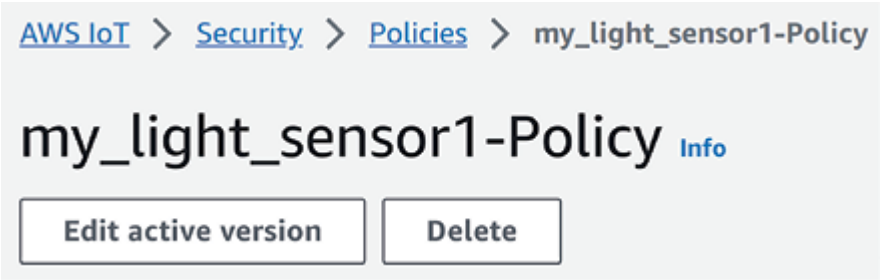


Figure 8.4: Editing the active policy

9. Set the policy JSON document as the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Receive",
        "iot:Subscribe",
        "iot:Connect"
      ],
      "Resource": "arn:aws:iot:*:*:*"
    }
  ]
}
```

10. Mark it as the active policy and save it.

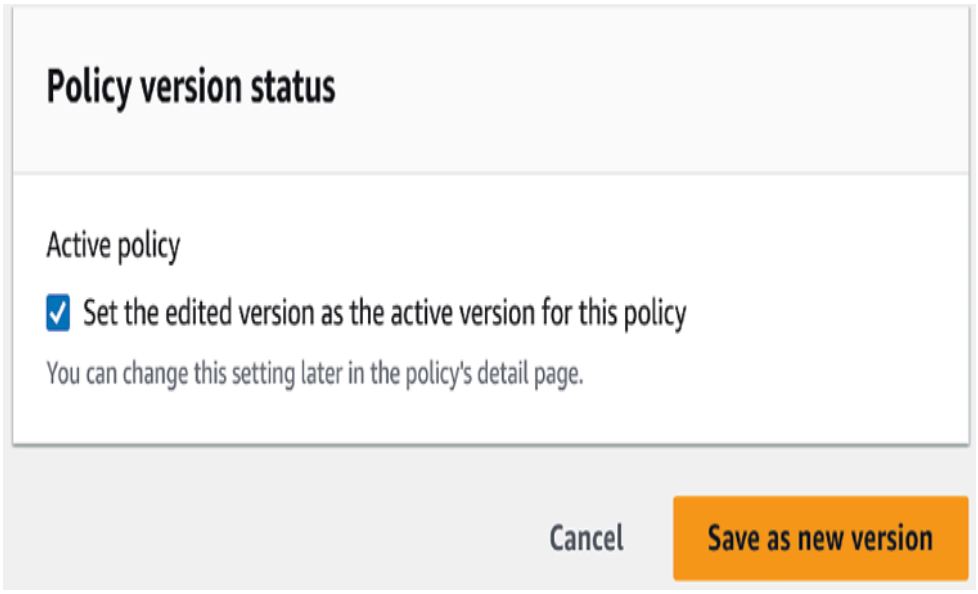


Figure 8.5: Save the policy as the active one

11. When it is done, you should see the following list as the policy.

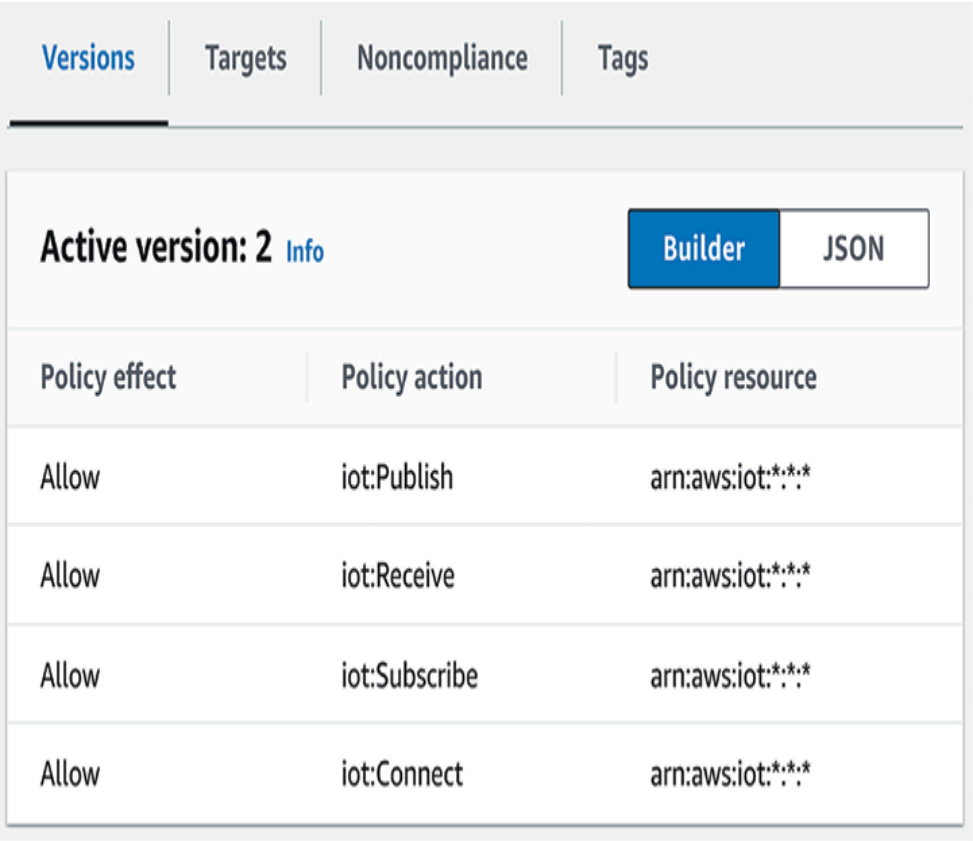


Figure 8.6: Active policy with permissions

Now, we have a light sensor defined on the AWS cloud. We will use its cryptographic credentials to connect our devkit to AWS IoT Core. We can continue with creating and configuring a new IDF project.

Configuring a project

We can create and configure an IDF project as follows:

1. Create an ESP-IDF project from the command line (you can choose any other way, as described in *Chapter 2, Understanding the Development Tools*).

```
$ export $HOME/esp/esp-idf/export.sh
$ idf.py create-project aws_ex
```

2. Copy `sdkconfig.defaults`, `CMakeLists.txt`, and `partitions.cvs` from the project repository into the root folder of the project.
3. Rename the application source code `main/aws_ex.cpp` and update the `main/CMakeLists.txt` file accordingly:

```
idf_component_register(SRCS ./aws_ex.cpp INCLUDE
```

4. We need Espressif's fork of *AWS IoT Device SDK*. The book repository contains it as a sub-module, and you can use it directly. However, if you want to clone it yourself, here is the URL for it: https://github.com/espressif/esp-aws-iot. Please use the **release/202210.01-LTS** branch as described in the documentation. Then, you need to set an environment variable, showing the path to the library:

```
$ export AWSIOT_PATH=<your_path>
```

5. Create a new directory, `tmp`, in the project root, extract the downloaded AWS device kit into this directory, and rename the files as follows:

```
$ mkdir tmp && cd tmp
$ unzip connect_device_package.zip
```

```
$ mv connect_device_package/my_light_sensor.cert
$ mv connect_device_package/my_light_sensor.priv
```

6. We also need the Amazon root certificate in the same directory as the device files:

```
$ curl https://www.amazontrust.com/repository/Am
```

7. The `start.sh` file that comes with the device kit contains the **AWS IoT endpoint**. Use the IoT endpoint address in this file to set the environment variable for it:

```
$ export AWS_ENDPOINT="\"<your_endpoint>-ats.iot
```

8. Make sure that you have the chapter's `common` directory downloaded from the book repository:

```
$ ls -1 <book_repo_clone>/ch8/common/
esp-aws-iot
espressif__qrcode
sensor
wifi
```

9. Double-check whether the `CMakeLists.txt` file in the project root matches your configuration. You can now run `idf.py` to compile the project:

```
$ idf.py build
```

The project is configured and ready for development, which comes next.

Coding the application

Let's begin with the implementation of the AWS client by adding a header file, `main/AppAwsClient.hpp`, for it:

```
#pragma once
#include <string>
#include <sstream>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "core_mqtt.h"
#include "network_transport.h"
#include "clock.h"
#include "sdkconfig.h"
#include "AppSensorClient.hpp"
```

The `core_mqtt.h`, `network_transport.h`, and `clock.h` headers come with the AWS IoT Device SDK. The purposes of the headers are implied by their names, but in short, `network_transport.h` provides the functionality for the TLS-level connection to the AWS cloud infrastructure, by using the device's cryptographic data. `core_mqtt.h` defines the functions and structures for MQTT connectivity. It is also a part of *Amazon FreeRTOS*. `clock.h` contains

the time helper functions for the other components of the SDK. Then, we declare the variables to get access to the cryptographic data that we embedded in the firmware:

```
extern "C"
{
    extern const char root_cert_auth_start[] asm(
        "_binary_root_cert_auth_crt_start");
    extern const char root_cert_auth_end[] asm(
        "_binary_root_cert_auth_crt_end");
    extern const char client_cert_start[] asm(
        "_binary_client_crt_start");
    extern const char client_cert_end[] asm(
        "_binary_client_crt_end");
    extern const char client_key_start[] asm(
        "_binary_client_key_start");
    extern const char client_key_end[] asm(
        "_binary_client_key_end");
}
```

We define the AWS root certificate, the device certificate, and the device private key. They will be needed to create a TLS connection to the AWS IoT endpoint. Next, we define the AWS client class:

```
namespace app
{
    class AppAwsClient : public AppSensorClient
    {
    private:
```

```
        bool m_aws_connected{false};
        TaskHandle_t m_process_task{nullptr};
```

The `m_aws_connected` member variable shows whether AWS IoT Core is connected or not. `m_process_task` is the handle for the task where we call the MQTT loop function to let the MQTT library do its internal processing. We continue with more member variables:

```
        std::string m_aws_endpoint{AWS_ENDPOINT};
        std::string m_thing_name{"my_light_sensor"};
        std::string m_user_name{"any_user_name"};
        std::string m_topic_name{"my_light_sensor/re
        uint8_t m_local_buffer[1024];
```

These member variables are defined to configure the MQTT library. We will publish the light data to the `m_topic_name` topic, and we provide a buffer, `m_local_buffer`, for the MQTT library for any incoming messages. Next comes the member variables for the TLS connection:

```
        NetworkContext_t m_network_context;
        TransportInterface_t m_transport;
```

The `m_network_context` variable holds the TLS connection configuration, such as the AWS IoT endpoint address and port in addition to the cryptographic data. The `m_transport` variable is the

TLS interface for the MQTT connection. Their types are defined in the `network_transport.h` header file as you might expect. Then we define the MQTT variables:

```
MQTTConnectInfo_t m_connect_info;
MQTTFixedBuffer_t m_network_buffer;
MQTTPublishInfo_t m_publish_info;
MQTTContext_t m_mqtt_context;
```

The `m_connect_info` member variable has the MQTT-related configuration values, such as username and client identifier. `m_network_buffer` will be configured to point to the `m_local_buffer` variable for incoming data, and `m_publish_info` defines the topic where we publish light readings. The `m_mqtt_context` member variable is the reference for the underlying MQTT connection, and all MQTT library functions need it to operate. Next, we have a static function for the FreeRTOS task of the MQTT loop:

```
static void processMqtt(void *arg)
{
    AppAwsClient *obj = reinterpret_cast<
        AppAwsClient *>(arg);
    while (1)
    {
        vTaskDelay(10 / portTICK_PERIOD_MS);
        MQTT_ProcessLoop(&obj->m_mqtt_contex
```

```
            }
        }
```

In the `processMqtt` function, we run a loop for a period of 10 milliseconds and call `MQTT_ProcessLoop` of the MQTT library, with the context variable of the client object. The next member function is the MQTT event handler:

```
        static void eventCallback(MQTTContext_t *pM
            MQTTPacketInfo_t *pPacketInfo,
            MQTTDeserializedInfo_t *pDeserializedIn
        {}
```

We don't have any code in the `eventCallback` function in this example, but this is the place where we can handle incoming MQTT messages and data if we subscribe to an MQTT topic. The `MQTTPacketInfo_t` type provides information about the MQTT message, such as its type, and we access incoming data through the `MQTTDeserializedInfo_t` type. The `private` section of the class implementation is finished, and we add the `public` section next:

```
    public:
        void init(void)
        {
            xTaskCreate(processMqtt, "pmqtt", 2048,
```

```
            5, &m_process_task);
        vTaskSuspend(m_process_task);
```

The first function in the `public` section is `init`. In the `init` function, we create a FreeRTOS task for the MQTT loop processing. We immediately suspend it because there is no AWS connection yet. Next, we initialize the `m_network_context` variable for a TLS connection:

```
        m_network_context.pcHostname = m_aws_enc
        m_network_context.xPort = 8883;
        m_network_context.pxTls = NULL;
        m_network_context.xTlsContextSemaphore =
            xSemaphoreCreateMutex();
        m_network_context.disableSni = 0;
        m_network_context.pcServerRootCA =
            root_cert_auth_start;
        m_network_context.pcServerRootCASize =
            root_cert_auth_end - root_cert_auth_
        m_network_context.pcClientCert = client_
        m_network_context.pcClientCertSize = cli
            client_cert_start;
        m_network_context.pcClientKey = client_k
        m_network_context.pcClientKeySize = clie
            client_key_start;
        m_network_context.pAlpnProtos = NULL;
```

The `m_network_context` variable shows how to connect to AWS IoT Core over TLS by specifying the endpoint address and port, the device's cryptographic credentials, and several other fields for internal processing. Then, we set the fields of the TLS interface for the MQTT context:

```
            m_transport.pNetworkContext = &m_network
            m_transport.send = espTlsTransportSend;
            m_transport.recv = espTlsTransportRecv;
            m_transport.writev = nullptr;
```

The `m_transport` variable has a field for the `m_network_context` variable, since it is the contact point for the TLS communication. When we try to set up a connection to the server, it uses the connection information to start a TLS session with the server. The `espTlsTransportSend` and `espTlsTransportRecv` functions are the Espressif implementation in the AWS IoT Device SDK, and they use the ESP-IDF TLS infrastructure to talk to the server over TLS. The `writev` is not required and not necessary, so we simply set it to `nullptr`. Let's configure the buffer next:

```
            m_network_buffer.pBuffer = m_local_buffe
            m_network_buffer.size = sizeof(m_local_b
```

The buffer configuration requires only the actual physical buffer and its size. This buffer is used by the MQTT library for incoming data. Next, we specify the MQTT connection configuration:

```
            m_connect_info.cleanSession = true;
            m_connect_info.pClientIdentifier =
                m_thing_name.c_str();
            m_connect_info.clientIdentifierLength =
                m_thing_name.length();
            m_connect_info.keepAliveSeconds = 60;
            m_connect_info.pUserName = m_user_name.
            m_connect_info.userNameLength = m_user_
```

For the MQTT connection, we set the client identifier, MQTT username, and the period of the keep-alive messages. The client identifier must be unique per MQTT client on the broker. The final configuration is for publishing data on the MQTT broker:

```
            m_publish_info.qos = MQTTQoS0;
            m_publish_info.pTopicName = m_topic_name
            m_publish_info.topicNameLength =
                m_topic_name.length();
```

The `m_publish_info` member variable shows the MQTT topic to publish and the **Quality of Service** (**QoS**) level. It is `MQTTQoS0` in our example. The MQTT protocol defines three levels of QoS:

- **QoS-0 means at most once**: A single MQTT message is sent to the broker without the expectation of an acknowledgment message. Therefore, there is no delivery guarantee at this level of QoS.

- **QoS-1 means at least once**: The client will continue to send the message to the broker until it receives an acknowledge message, which means that the broker may receive the same message more than once.
- **QoS-2 means exactly once**: There is a handshake mechanism between the broker and the client in this type of QoS. This guarantees the delivery of the message exactly once.

The most resource-efficient type of service is QoS-0, since the client sends a message only once and waits for no reply. This is the most efficient way of communicating when the delivery of the message doesn't have much importance; for example, if one message is not delivered, the second one can do the same job.

We finish the `init` function by initializing the MQTT library as follows:

```
            MQTT_Init(&m_mqtt_context, &m_transport,
                Clock_GetTimeMs, eventCallback,
                &m_network_buffer);
        }
```

The `MQTT_Init` function takes the `m_mqtt_context`, `m_transport`, and `m_network_buffer` configuration variables as parameters. In addition to them, we pass the event handler and the `Clock_GetTimeMs` function for the MQTT library to calculate the timings of messages. MQTT initialization is quite long, but we now have all the configurations ready to connect and send MQTT

messages to AWS IoT Core. The next function is to handle the WiFi connection state changes:

```
void setWiFiStatus(bool connected)
{
    if (connected)
    {
        if (xTlsConnect(&m_network_context)
            TLS_TRANSPORT_SUCCESS)
        {
            vTaskResume(m_process_task);
            bool sess_present;
            m_aws_connected = MQTT_Connect(&
              &m_connect_info, nullptr, 1000
              &sess_present) == MQTTSuccess;
            if (!m_aws_connected)
            {
                vTaskSuspend(m_process_task);
            }
        }
    }
```

If the local WiFi is connected, we try to connect to AWS IoT Core over TLS by calling the `xTlsConnect` function of the MQTT library. Again, this function comes with the Espressif port and is specific to the ESP32 devices. If the TLS connection succeeds, we try to set up the MQTT connection by calling the `MQTT_Connect` function. The `MQTT_Connect` function blocks for `1000` milliseconds and returns `MQTTSuccess` if everything goes well. If it fails, we suspend the

MQTT processing task, since there is no active MQTT connection. We handle the case when the TLS connection fails as follows:

```
            else
            {
                m_aws_connected = false;
                vTaskSuspend(m_process_task);
            }
        } // function end
```

If the TLS connection fails (possibly a configuration error), there is nothing more to do, and we simply suspend the MQTT processing task. The next function implements the `update` virtual function of the `AppSensorClient` interface:

```
        void update(uint32_t light_level) override
        {
            if (!m_aws_connected)
            {
                return;
            }
```

At the beginning of the `update` function, we check whether there is an active MQTT connection, and if not, we simply return from the function. When we have an active connection, we can prepare a message to publish as follows:

```
            std::stringstream ss_mqtt_message;
            ss_mqtt_message << "{\"light_level\":"
```

```
                << light_level << "}";
            std::string payload = ss_mqtt_message.st
            m_publish_info.pPayload = payload.c_str(
            m_publish_info.payloadLength = payload.l
```

We create a JSON message for the light reading and update the `m_publish_info` variable's payload with it. The only remaining thing to do is publish it to the broker, which comes next:

```
            uint16_t packet_id = MQTT_GetPacketId(&m
            MQTT_Publish(&m_mqtt_context, &m_publish
                packet_id);
        } // function end
    }; // class end
} // namespace end
```

The MQTT library needs a unique identifier for a message to be published. We get this identifier by calling the `MQTT_GetPacketId` function of the library. Then, we call the `MQTT_Publish` function to send the message to AWS IoT Core. This completes the implementation of the `AppAwsClient` class. Now, we can use it in the application and integrate it with the other components. Let's edit `main/aws_ex.cpp` for this:

```
#include <functional>
#include "AppWifiSoftAp.hpp"
#include "AppAwsClient.hpp"
```

```
#include "AppSensor.hpp"
namespace
{
    app::AppWifiSoftAp app_wifi;
    app::AppAwsClient app_client;
    app::AppSensor app_sensor;
}
```

After including the headers for our application, we define the objects in the anonymous namespace. The `app_client` object is an instance of `AppAwsClient` that we have just implemented. We developed the `AppWifiSoftAp` and `AppSensor` classes in the previous chapters (*Chapter 6, Using Wi-Fi Communication for Connectivity*, and *Chapter 7, ESP32 Security Features for Production-Grade Devices*, respectively), so we are just using them here again to create one instance for each. Then, we define the `app_main` function:

```
extern "C" void app_main()
{
    auto wifi_connected = [](esp_ip4_addr_t *addr_i
    {
        app_client.setWiFiStatus(true);
        app_sensor.start();
    };
```

The `wifi_connected` local variable is the `lambda` function to be called by the `app_wifi` object when WiFi is connected. It updates

the `app_client` object with this event and starts the sensor readings. We also need another `lambda` function for the WiFi-disconnected state:

```
    auto wifi_disconnected = [](void)
    {
        app_client.setWiFiStatus(false);
    };
```

Again, we update the `app_client` object when WiFi is disconnected so that it can change its internal state accordingly. Next, we link all the pieces:

```
    app_wifi.init(wifi_connected, wifi_disconnected)
    app_client.init();
    app_sensor.init(&app_client, 5);
```

We first call the `init` function of the `app_wifi` object with the callback functions. Then, we initialize the `app_client` object. Lastly, we pass `app_client`'s address to the `init` function of `app_sensor`. The sensor will read the light level every `5` seconds and inform `app_client` with the last reading, which, in turn, publishes it to the MQTT broker of AWS IoT Core. We can connect to the local WiFi as the last statement of the application:

```
    app_wifi.connect();
```

```
} // app_main end
```

Testing the application

The application is ready to test. Let's flash it on our devkits and observe the light readings on AWS IoT Core in the following steps:

1. Use `idf.py` tool to flash and monitor the application. If you have WiFi configured on your devkit, it will connect automatically. If not, the application will show a QR code that you can scan and configure the WiFi connection by using the Espressif mobile application, *ESP SoftAP Provisioning*:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<more logs>
I (2913) coreMQTT: MQTT connection established w
I (2913) esp_netif_handlers: sta ip: 192.168.1.1
```

2. There is an online MQTT client that you can access via the AWS console. Log in to your account and navigate to IoT Core. The MQTT client is listed on the left menu.

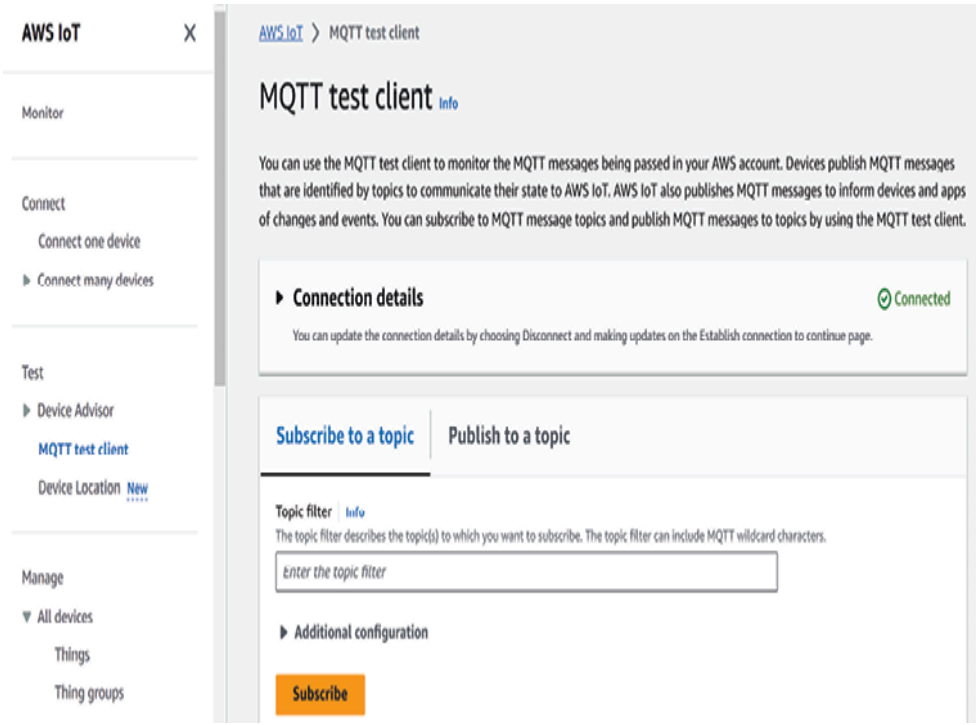


Figure 8.7: MQTT test client on the AWS console

3. Subscribe to the device topic by entering `my_light_sensor/reading` in the textbox and clicking on the **Subscribe** button.
4. You will see the light data published on the screen:

Figure 8.8: Light readings printed on the MQTT client screen

Nice! We can now collect light data from our sensor device and deliver it to the AWS cloud over MQTT.

Troubleshooting

If you encounter some difficulties along the way, take a look at the following list, which may help with them:

- If the application fails to connect to the WiFi, you can erase the flash and try a new configuration by using the mobile application. Erasing the flash will clear the NVS and force the application to start in the provisioning mode:

```
$ idf.py erase-flash flash monitor
```

- If the application cannot connect to the AWS server (because the TLS fails), make sure the cryptographic keys are correct. If the

application connects but the MQTT communication fails, check the device permissions in the AWS policy. You can test the cryptographic keys and device permissions by using an MQTT client, such as `mosquitto_pub`. If it succeeds, it means the device configuration is correct on AWS IoT Core, and you can see the message on the *MQTT test client* of AWS IoT Core:

```
$ mosquitto_pub --cafile tmp/root_cert_auth.crt
Client (null) sending CONNECT
Client (null) received CONNACK (0)
Client (null) sending PUBLISH (d0, q0, r0, m1, '
Client (null) sending DISCONNECT
```

In the next section, we will show the light data on a dashboard of **AWS Managed Grafana**.

Visualizing with Grafana

Grafana is a powerful tool for IoT data visualization. It is an open-source project available to everyone, but we also have a cloud option with *AWS Managed Grafana*, which frees us from any server management overhead. Some key features of Grafana are:

- Workspaces for developers to design dashboards for end users
- Easy integration with multiple data sources with plugins
- Easy manipulation and transformation of data with SQL
- A wide variety of customizable graph types for different needs

In this example, we will utilize AWS Managed Grafana to visualize the light data of the previous example. Here is an overview of what

we will do in this example:

- We will create an **Amazon Timestream** database and a table in it for the light data. Amazon Timestream is a managed time-series database that comes with all the advantages of using the AWS cloud, such as no server to manage, auto-scaling, auto-encryption, etc.
- We will have an **IoT rule** in AWS IoT Core to forward the light data to the Timestream database. IoT rules are used to route data from MQTT topics to other AWS services by using SQL queries.
- We will create a **Grafana workspace** and a dashboard inside it. The dashboard will have a panel to show the light data that resides in the Timestream table.

Let's get prepared by creating the database first.

Creating a Timestream database

We will create a table in a Timestream database and forward MQTT data into the newly created table, as follows:

1. Navigate to **Amazon Timestream** in your selected region and click on the **Create database** button on the screen:

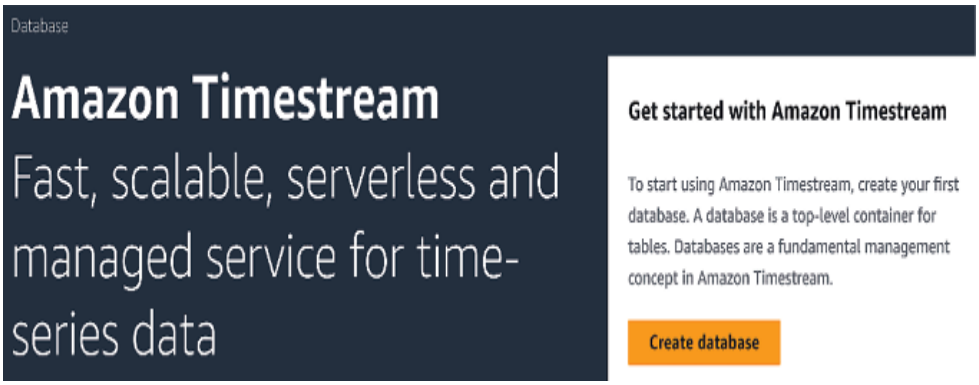


Figure 8.9: Creating a Timestream database

2. Select **Standard database** as the configuration, set its name to `ch8_db`, and continue by clicking on the **Create database** button at the bottom of the screen.

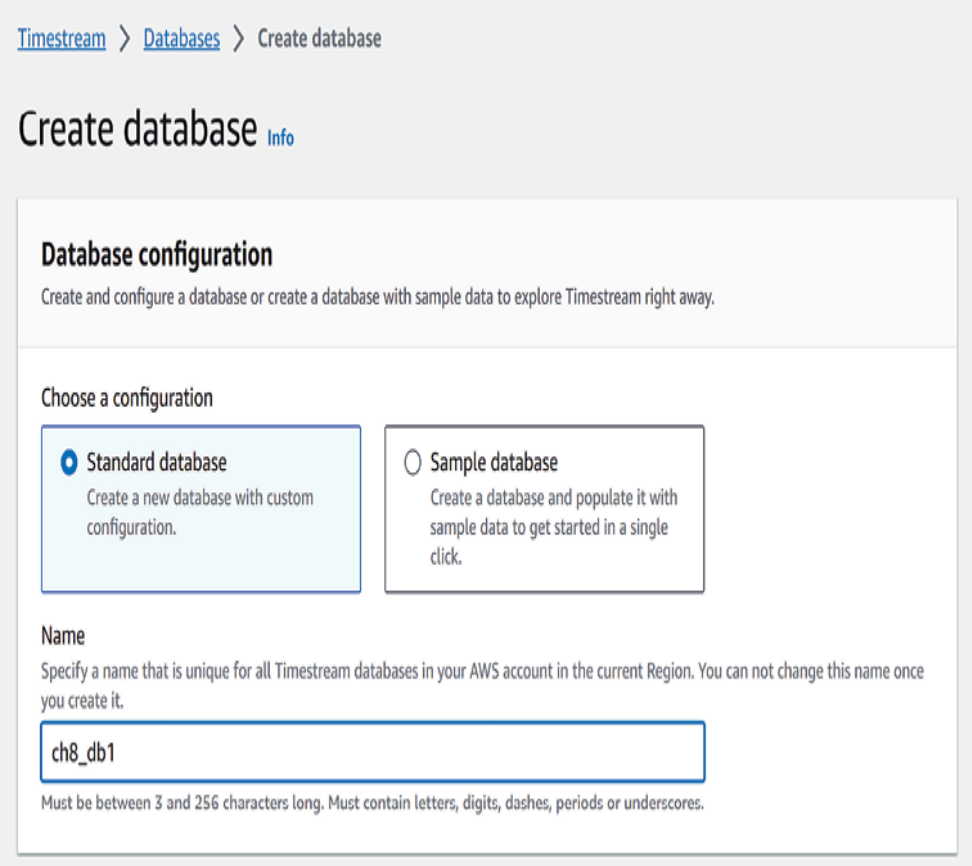


Figure 8.10: Creating a standard database

3. Select the database from the list, go to the **Tables** tab, and click on the **Create table** button:

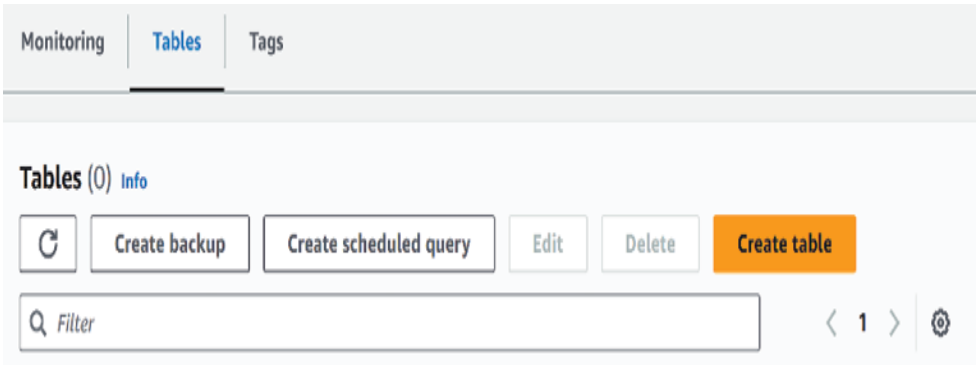


Figure 8.11: Create a table

4. Set **Table name** as `light_data`. Set minimum values for **Data retention**. We don't need **Magnetic Storage Writes**, so uncheck this option. Click on the **Create table** button at the bottom of the page:

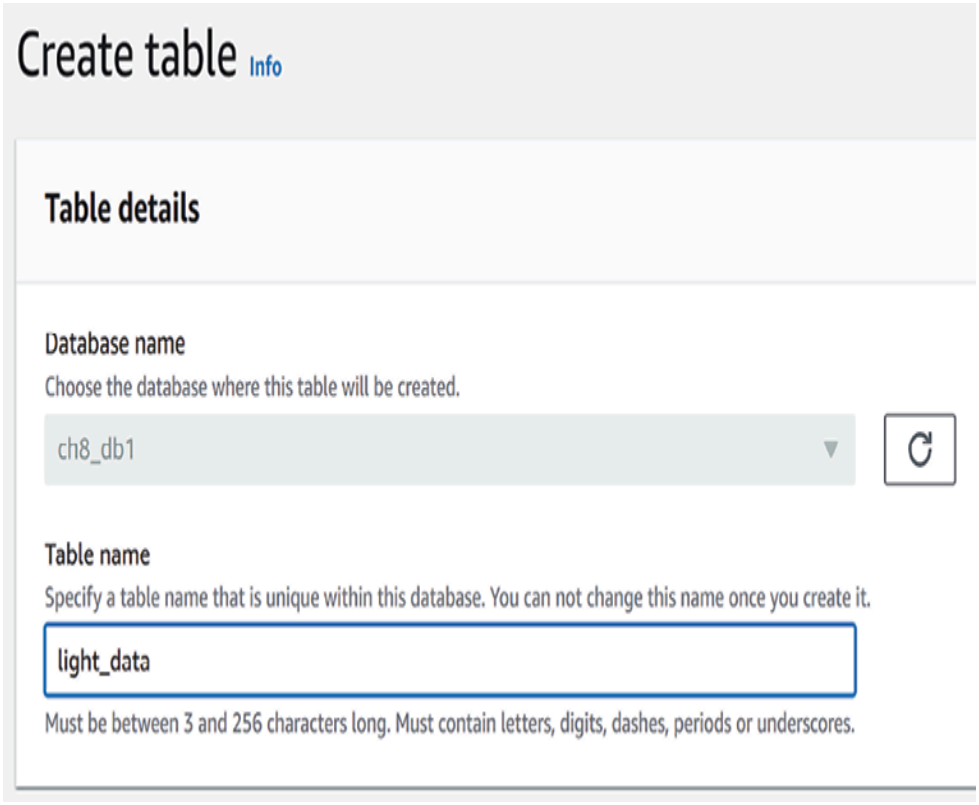


Figure 8.12: Create a table

We have a table to store light data. The next thing we will do is create an IoT rule to route data into it:

1. Navigate to IoT Core. From the left menu, select **Manage/Message routing/Rules** and click on the **Create rule** button:

Figure 8.13: Create a rule

2. Creating a rule is a four-step process. We first set **Rule name** as `light_rule`:

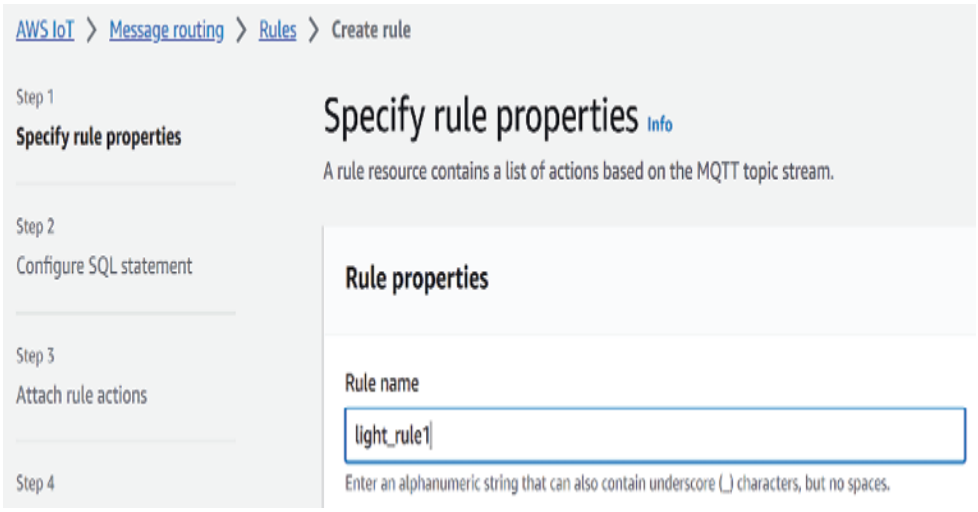


Figure 8.14: Step 1 of the wizard, setting the rule name

3. Set the SQL statement to `SELECT * FROM 'my_light_sensor/reading'`. We select all the data coming from the device topic.

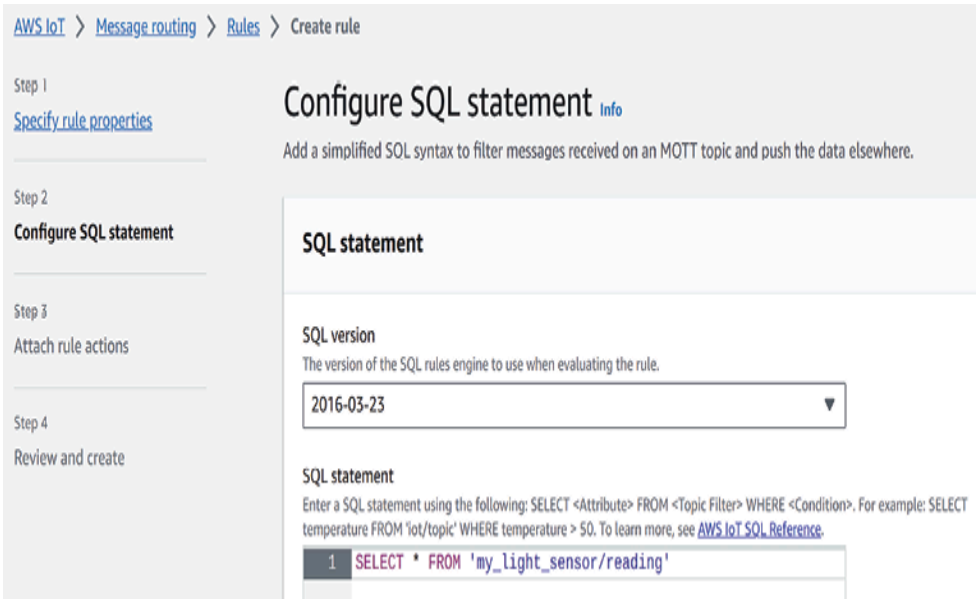


Figure 8.15: Step 3, SQL statement

4. In the next step, we specify the Timestream database and table by selecting them from the list. We also add a table dimension for data. Basically, a dimension is a database field to describe data. We simply add a dummy value there, since we have only one sensor, but we could also assign a field from the MQTT topic data. We will leave the **Timestamp** value empty, meaning that the Timestream database engine will assign it as data comes in. One last thing that we need before the final step is an **IAM role** for the rule. We simply click on **Create new role** and let the wizard set the necessary permissions for the role. We complete the step by clicking on the **Next** button at the bottom of the screen.

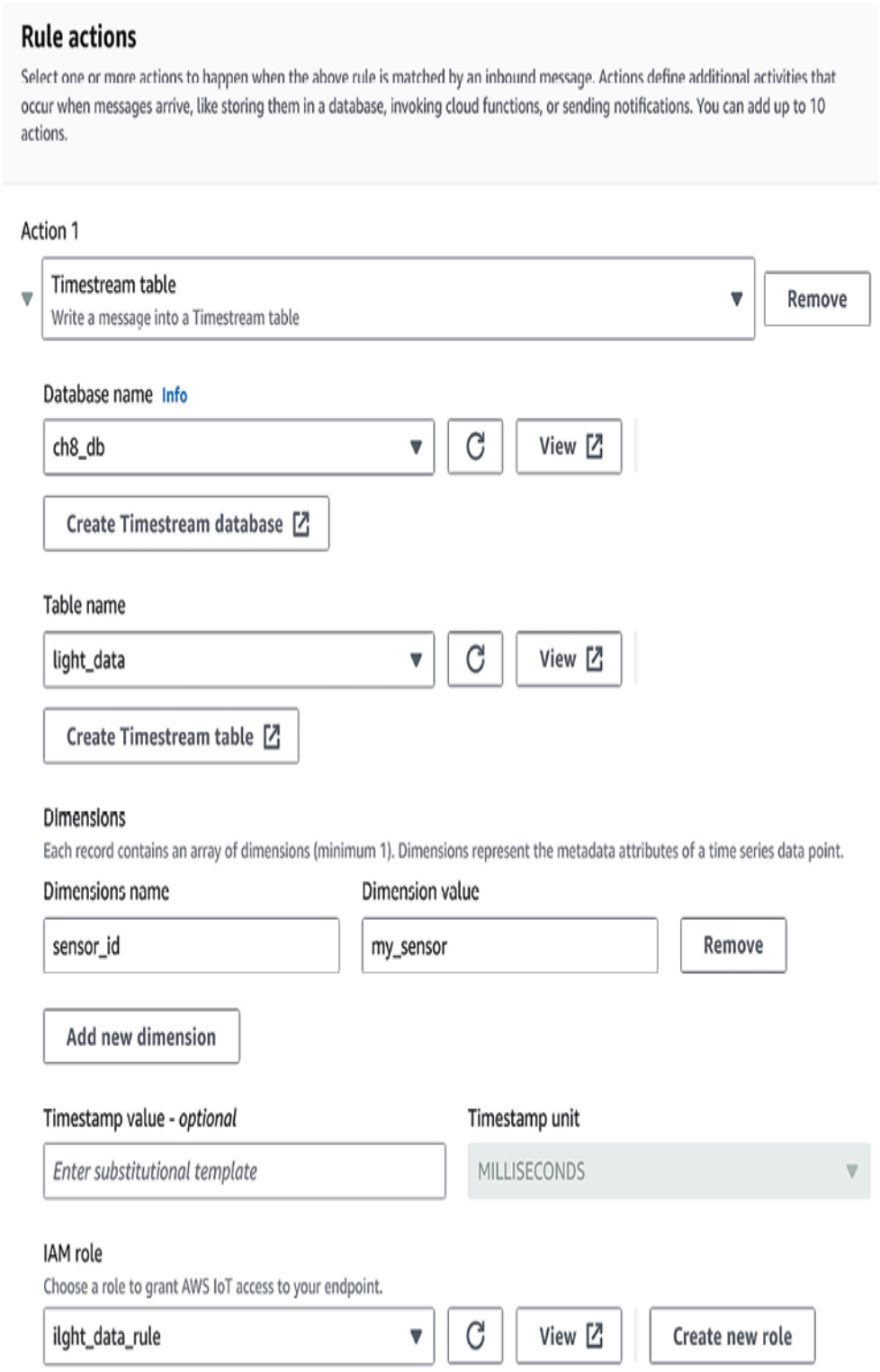


Figure 8.16: Step 3, adding a rule action

5. The wizard shows a summary. We can review the values for the IoT rule and click on the **Create** button to finalize the wizard.

We now have the IoT rule to forward data into the Timestream table. Let's check if data really accumulates in the table:

1. Go back to the Timestream service and select **Management tools/Query editor** from the left menu. In **Query editor**, we can run a `select` statement on the table to see the light data:

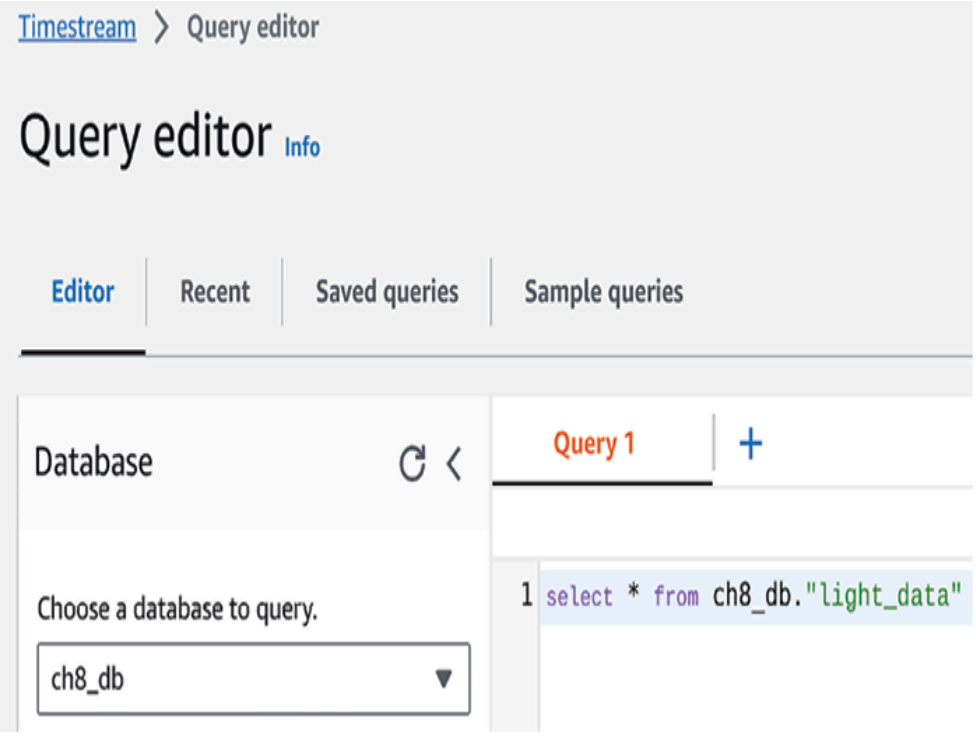


Figure 8.17: Selecting data from Timestream

2. The result is similar to the following figure. The table has four attributes, or columns. The first one is **sensor_id**, as we set in the rule configuration. **measure_name** and **measure_value** come from the MQTT topic. The **time** attribute is attached to the table by the Timestream engine as the light readings enter the table.

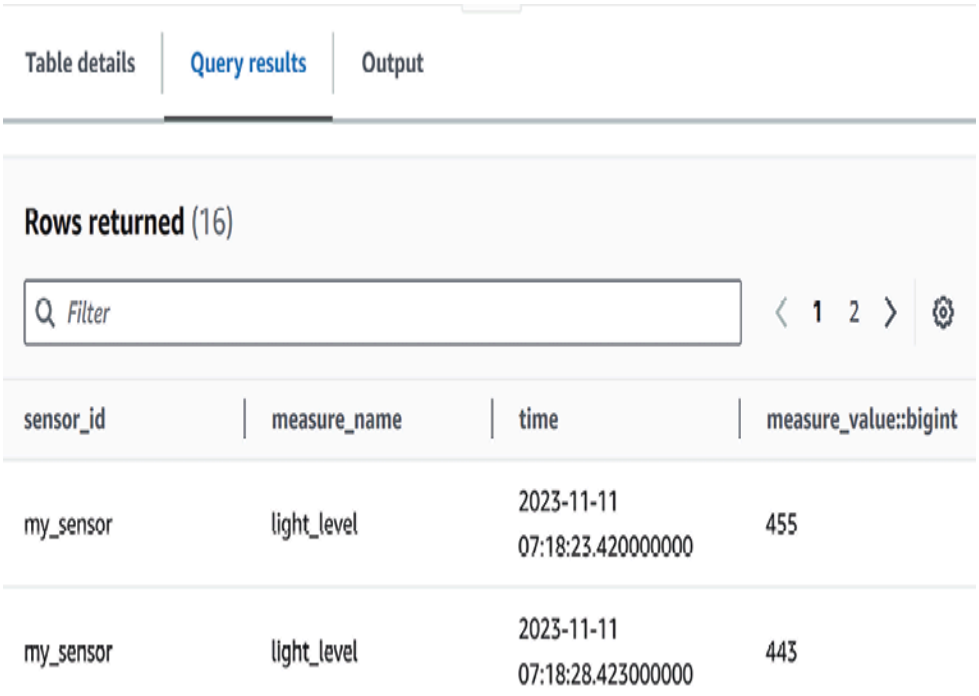


Figure 8.18: Light data in the Timestream table

We can see the light data flowing into the Timestream table with the help of the IoT rule, which means we can continue with Grafana to visualize them on a dashboard next.

Creating a Grafana workspace

Here are the steps to create a Grafana workspace:

1. Navigate to AWS Managed Grafana.
2. Start by clicking the **Create workspace** button. A four-step wizard will open. Give a name to the new Grafana workspace:

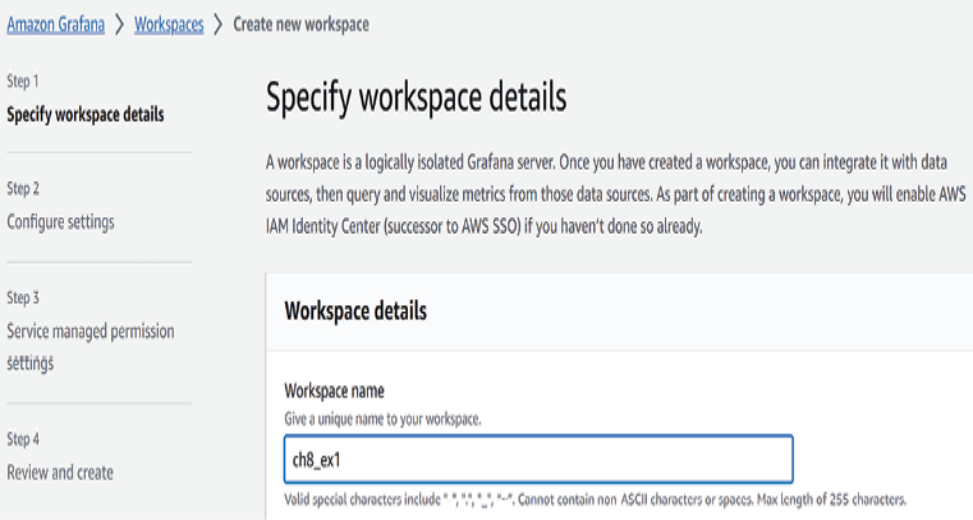


Figure 8.19: Setting up a Grafana workspace

3. Provide the configuration for the workspace. We specify the authentication method to connect to the Grafana service. It is the **AWS IAM Identity Center** for our example. Leave the other options with their default values.

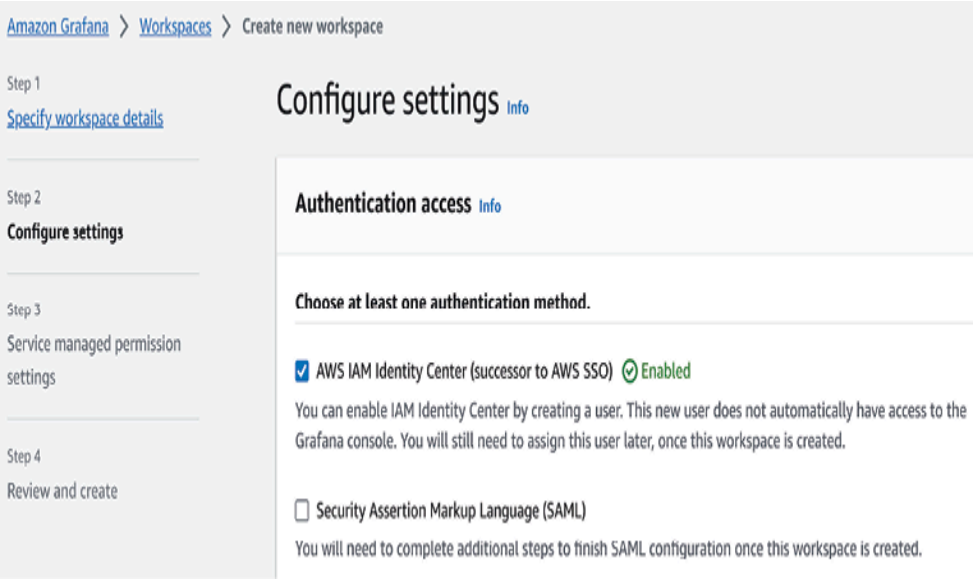


Figure 8.20: Selecting the authentication method

4. Select which data sources Grafana can access in our account. It is only **Amazon TimeStream** in our example.

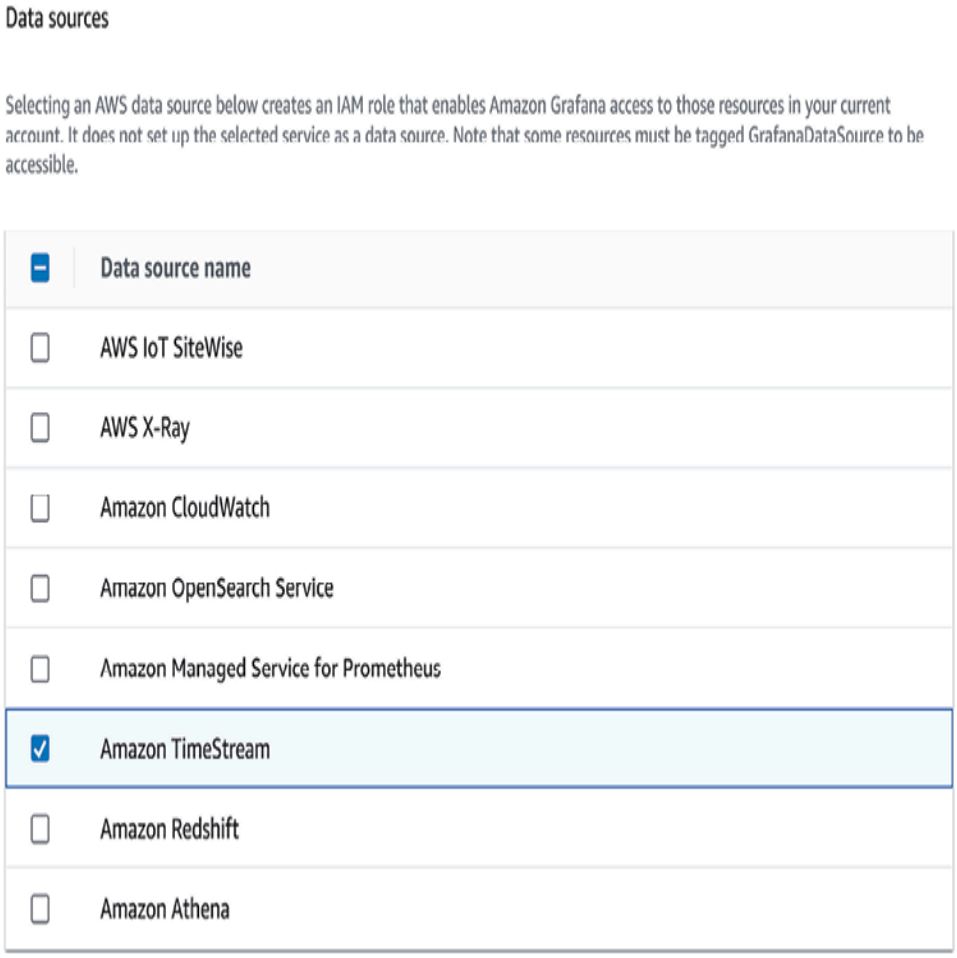


Figure 8.21: Selecting data sources

5. In the last step of the wizard, we review the workspace configuration and create it by clicking the **Create workspace** button at the bottom of the screen. It will take some time to get the workspace ready.
6. Although we have an active workspace, we are not done with the configuration. We need to assign a user from **AWS IAM Identity Center**. Click on the **Assign new user or group** button for this.

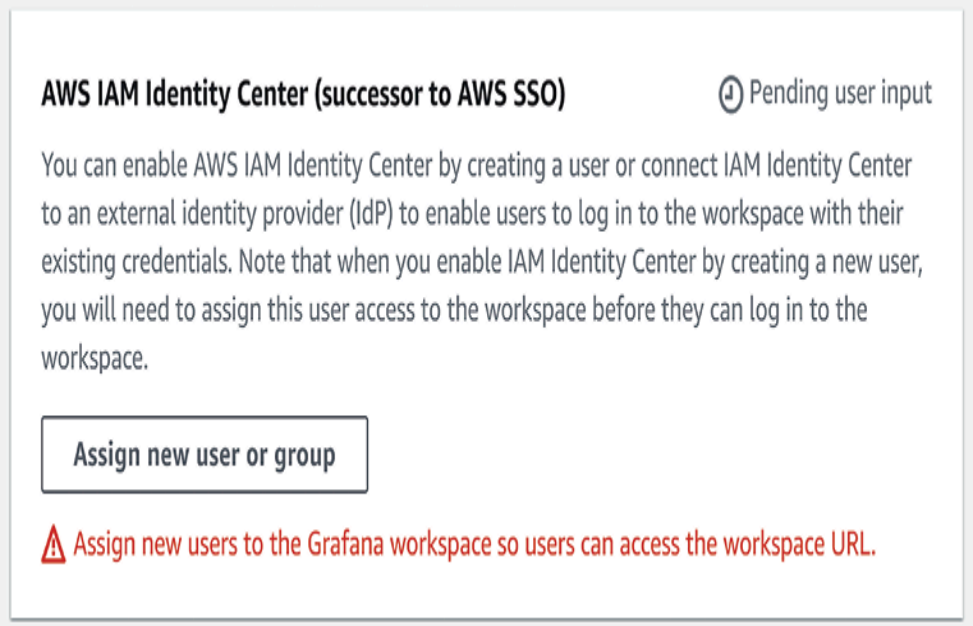


Figure 8.22: Assigning a new user to the workspace

7. If you already have an IAM identity, you can select it from the list. If you don't have one, you need to create one for this purpose. It requires some more steps in IAM Identity Center. Navigate to the service in your preferred AWS region, select **Users** from the left menu, and click on the **Add user** button.

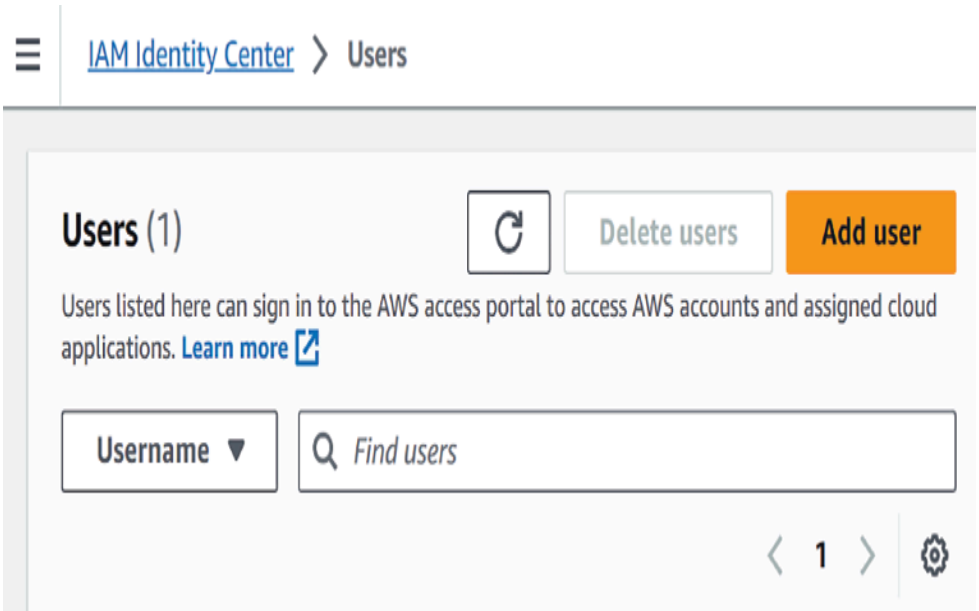


Figure 8.23: Adding a new IAM Identity user

8. It will open a new wizard to create an IAM Identity user. Follow the steps to add a user. When you finish the steps, you can use that user with Grafana. Don't forget to assign the user as **Admin**; otherwise, you cannot work in the Grafana workspace.

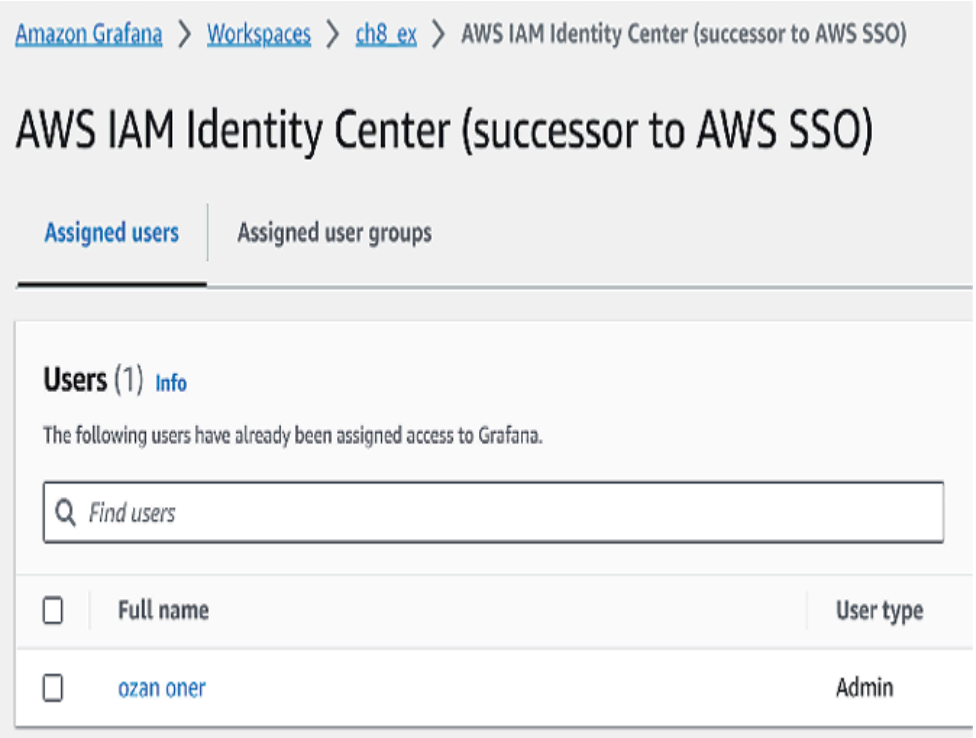


Figure 8.24: Assigning the user as Admin

Now that we have a Grafana workspace, we can create a dashboard in the workspace to display IoT data.

Creating a Grafana dashboard

We have created a workspace and assigned an admin user. We can log in with this user to create a Grafana dashboard, as shown in the following steps:

1. Follow the link in *step 6* of the previous topic to open the Grafana workspace. Enter the username and password to log in. You will see the workspace landing page.

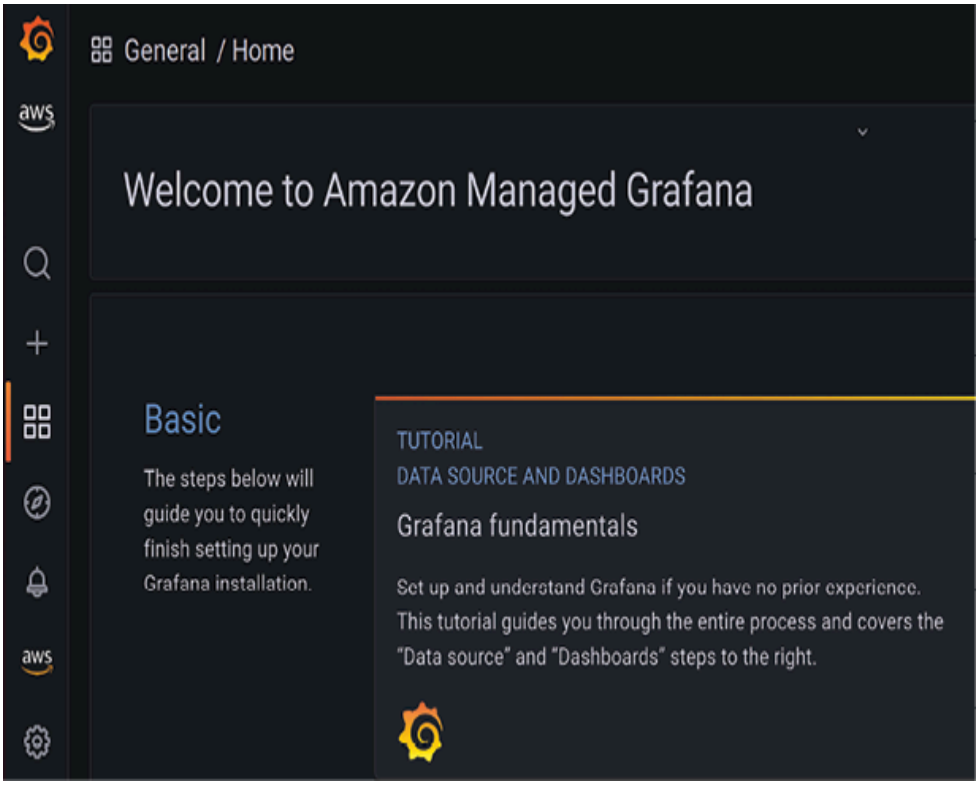


Figure 8.25: Grafana workspace

2. Grafana has great documentation at https://grafana.com/docs/grafana/latest/getting-started/, showing what can be done next, but the home page of the workspace also provides quick links to create a data source and dashboard. Let's follow these links:

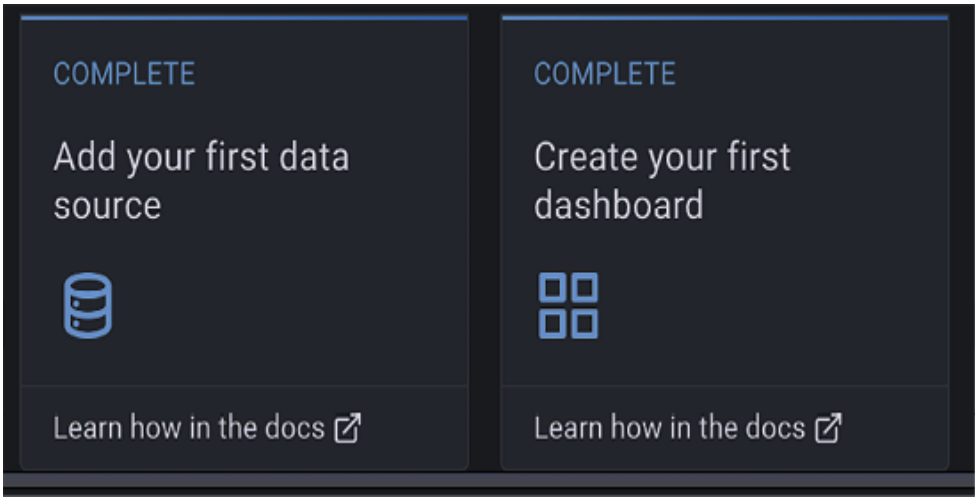


Figure 8.26: Quick links

3. Create a data source by clicking on the **Add your first data source** link on the screen. Use the filter on the displayed page to select **Amazon Timestream**:

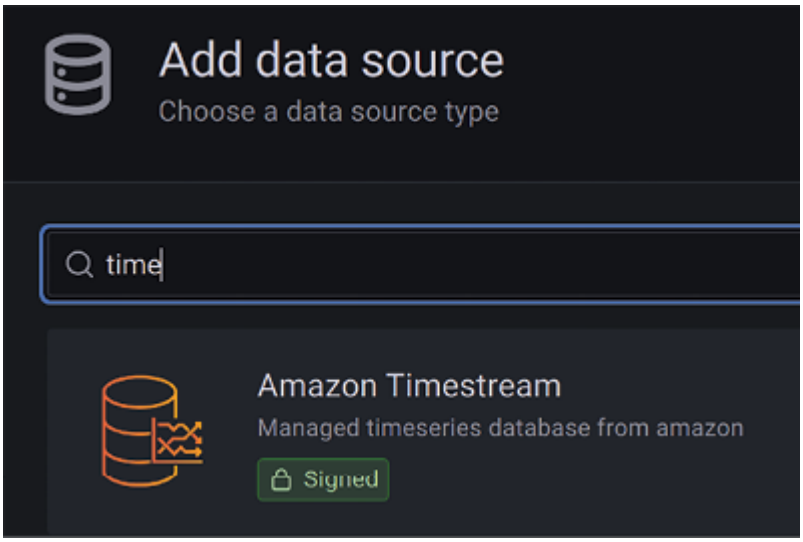


Figure 8.27: Setting up a data source

4. On the next page, we configure the data source by selecting from the combo boxes. Select **Default Region**, **Database**, **Table**, and **Measure**. The default region is the region where you have

configured your Timestream database. Then, click the **Save & test** button at the bottom of the screen.

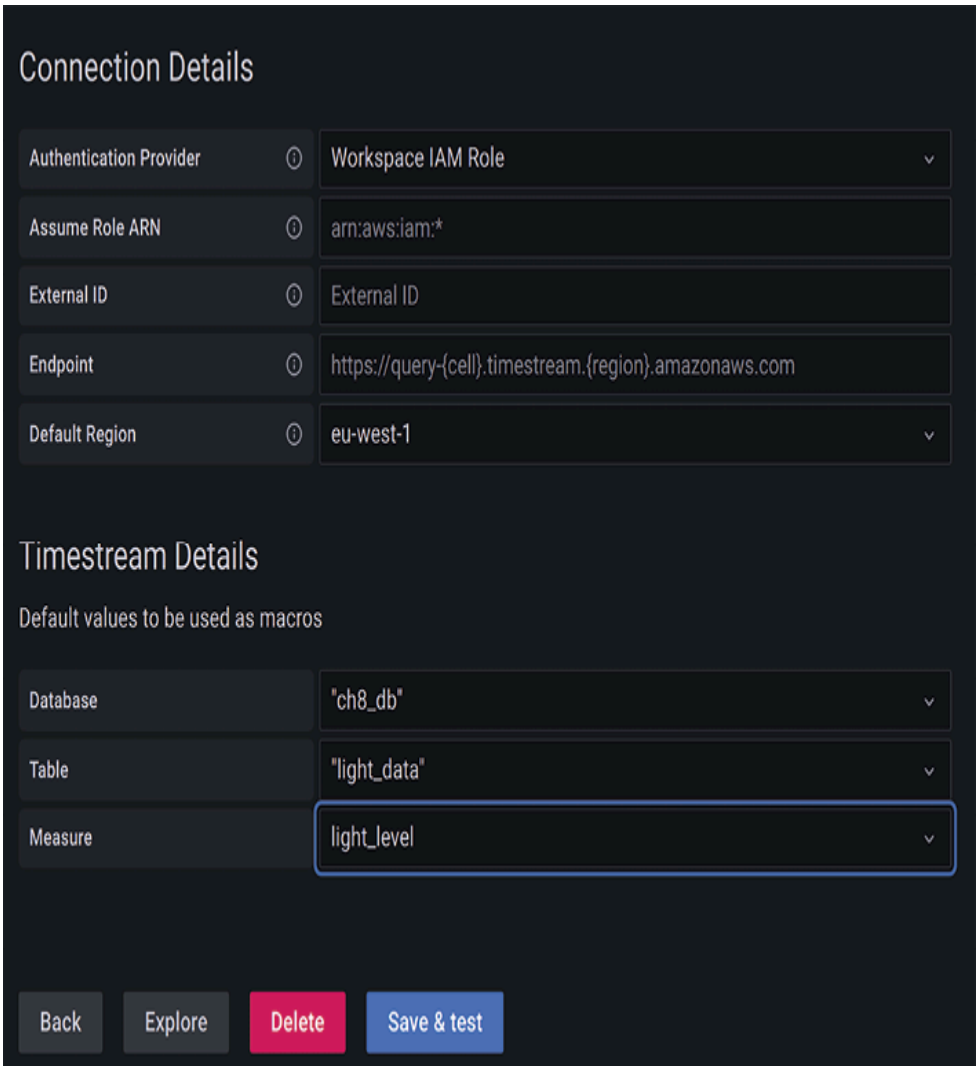


Figure 8.28: Data source configuration

5. We have the data source ready. Go back to the workspace home page to create a dashboard. Click on the **Create your first dashboard** link for this. The link will lead you to the dashboard designer. Select **Add a new panel** there.

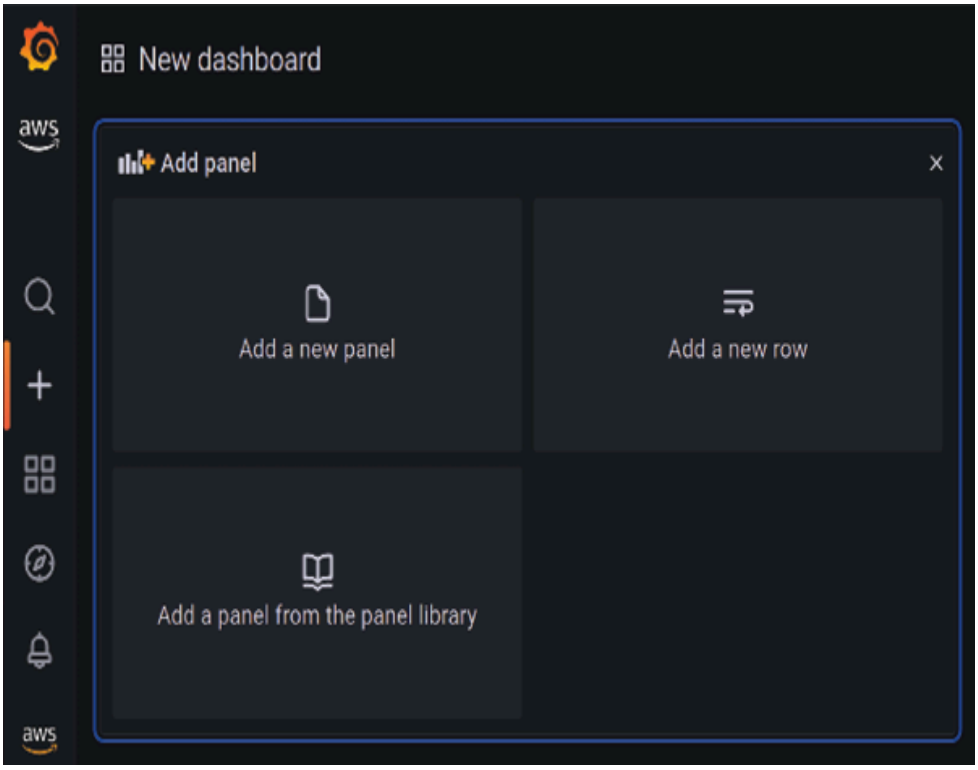


Figure 8.29: Adding a new panel

6. In the panel designer, we provide the query to be run on the data source. The interface provides some sample queries that we can select from a drop-down menu. The result of the query will be shown as a **time series** graph.

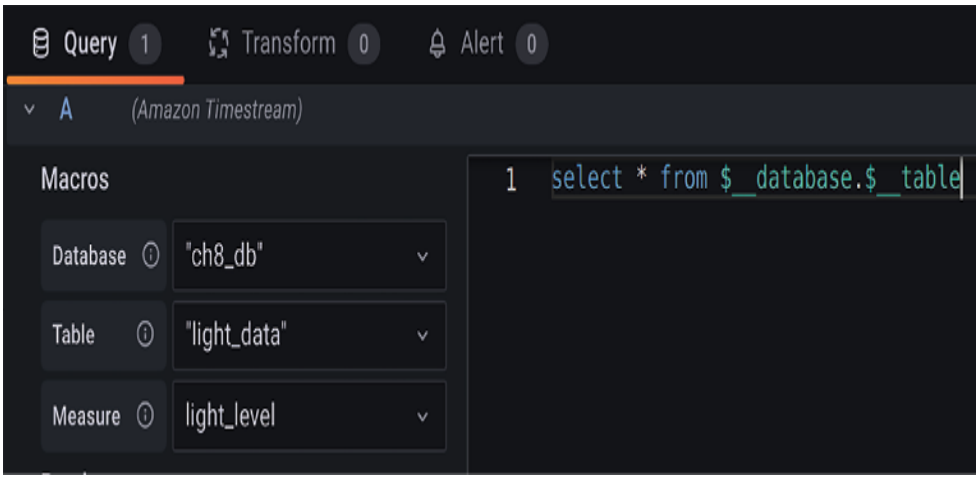


Figure 8.30: Selecting data to show in a time series graph

7. Click the **Apply** button to save the panel on the dashboard.

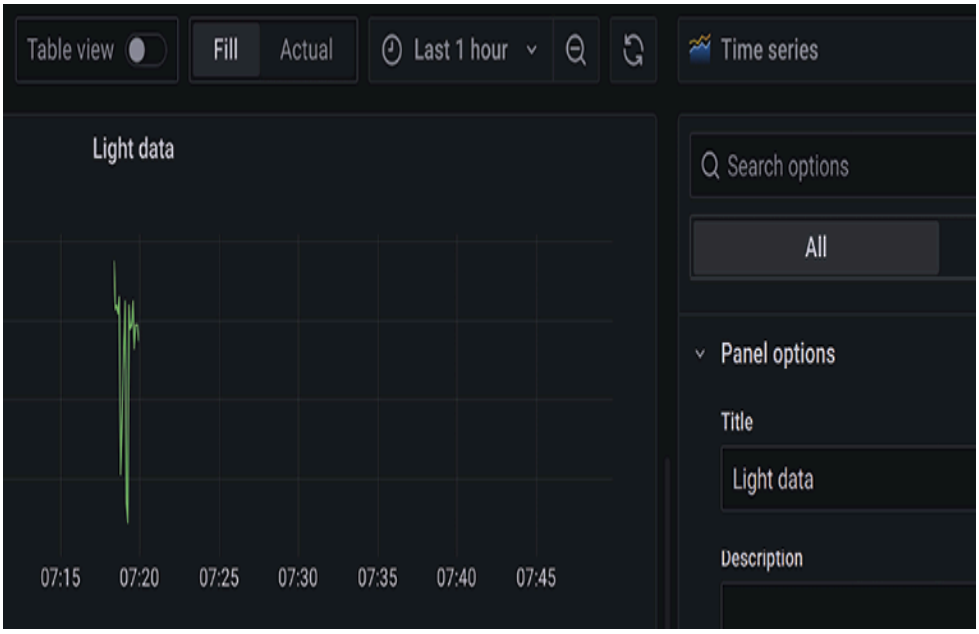


Figure 8.31: Adding a panel to the dashboard

8. Save the dashboard by clicking on the **Save** button – the one with a floppy disk icon at the top of the screen:

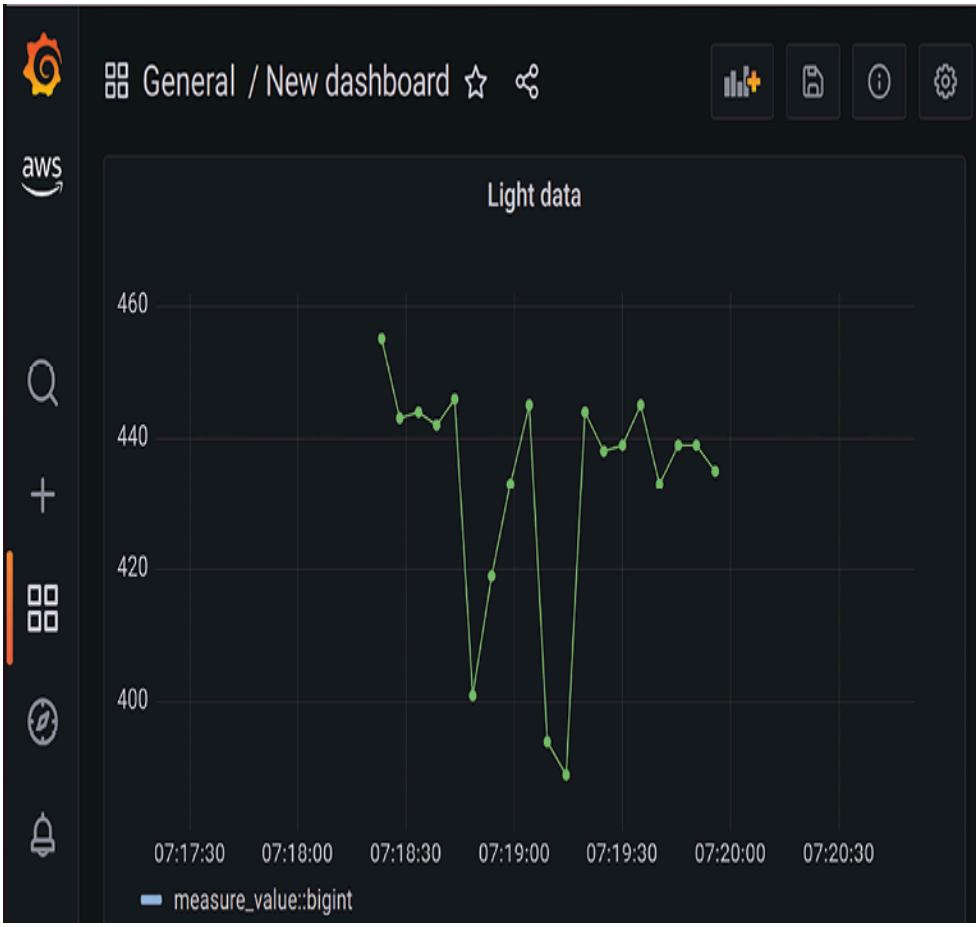


Figure 8.32: Saving the dashboard

9. Now, we have an interactive Grafana dashboard. There are buttons and drop-down boxes on the top to access the panel functionality, such as filtering data or automatically refreshing of the view. You can discover the dashboard functionality by playing with those UI elements.

Grafana has many other features that we cannot cover here. Please take your time to try other types of graphs to represent the data best in your specific case. Another great feature of Grafana is **Alerts**. It allows us to define thresholds on data to generate events and share them with other services, such as **Amazon Simple Notification Service** (**SNS**).

Troubleshooting

There are many steps, hence many places to make mistakes. Making mistakes can be a great learning opportunity. However, if you need some guidance, check out the following links:

- If you suspect the IoT rule doesn't do its job right, you can configure IoT Core logging on *CloudWatch*. Here is how to diagnose IoT rule issues on *CloudWatch*: https://docs.aws.amazon.com/iot/latest/developerguide/diagnosing-rules.html
- If anything goes wrong with permissions in general, you can check the corresponding IAM policies. IAM policies define the permissions on AWS resources. Please see the AWS documentation for more about policies: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html
- Grafana is quite straightforward to configure and has good documentation in general. For example, if you want to learn more about the Timestream plugin, here is some help: https://grafana.com/grafana/plugins/grafana-timestream-datasource/

In the next section, we will see how to integrate an ESP32 device with Amazon Alexa.

Integrating an ESP32 device with Amazon Alexa

Voice assistants provide another type of user interface in human-machine interaction, which is called a **Voice User Interface** (**VUI**). A user gives a voice command and a machine replies with a spoken response, in addition to any associated action if configured. Although there are many other enabling technologies, modern voice assistant systems basically make use of **speech recognition**, **Natural Language Processing** (**NLP**), and **speech synthesis**.

Amazon's voice assistant solution is **Alexa Voice Service** (**AVS**), and in this example, we will integrate ESP32 with AVS in a smart home **skill**. An Alexa skill is basically a voice application. The **Amazon Alexa mobile application** is the platform to enable skills and use Alexa-enabled devices in our Alexa accounts. If you don't already have this application on your mobile device, it is time to install it for this example.

The blueprint architecture for a smart home skill looks like this:

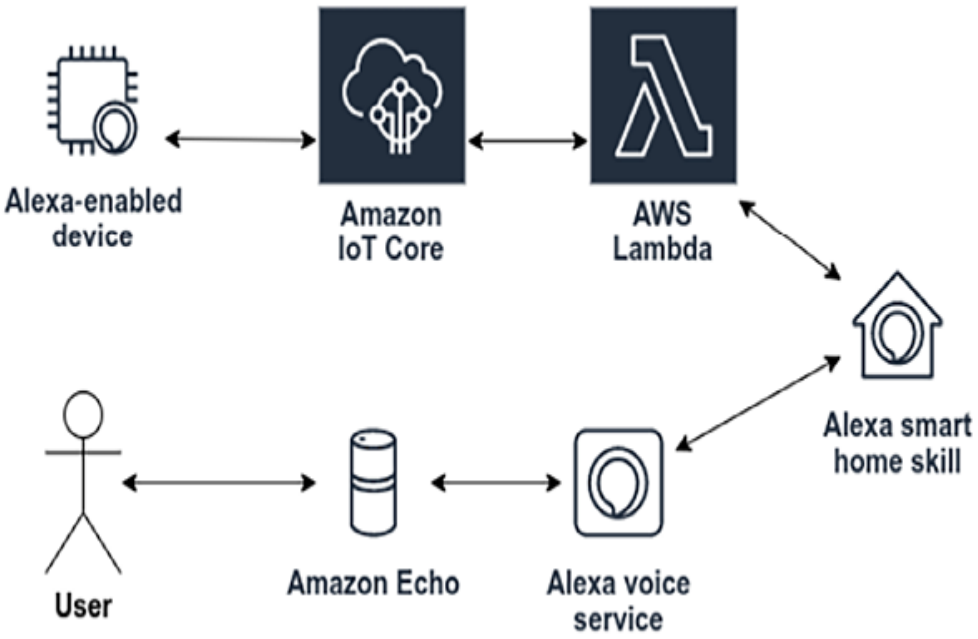


Figure 8.33: A smart home application architecture

Here is an overview of what we will do in this example:

1. We will develop a temperature sensor by using ESP32-S3 Box Lite and a DS18B20 temperature sensor. The application will update the **device shadow** of our sensor. A device shadow is a persistent representation of an IoT device on AWS.
2. We will develop a **smart home skill**, which is the voice application on AVS.
3. We will add a **lambda function** to handle requests from the smart home skill. We will handle three types of requests in this lambda function: **AcceptGrant**, **Discovery**, and **State** requests. The reply to a state request will return the latest light reading from the thing shadow.

AVS is the core service. It does the conversion between data and speech. When a user activates an Alexa speaker, such as Amazon Echo, with its **wake word**, for example by saying "*Alexa*," the speaker collects voice data coming after the wake word and sends it to AVS. AVS processes the voice data to understand the command, or **utterance**. Then, AVS extracts the **intent** from the utterance and delivers the intent to the associated skill. In the reverse direction, when a reply comes from the skill, it converts the reply data to speech to be voiced by the Alexa speaker.

In our example, a possible utterance would be:

"Alexa, what is the temperature inside?"

After the round is completed as explained above, we expect the current temperature from AVS and hear the response from the Alexa

developer console. You can use an Alexa speaker in this setup, but the Alexa Developer console has all the functionality to develop and test a skill on our development machines.

Let's start with the thing shadow.

Updating the thing shadow

AWS IoT Core automatically defines a device shadow for each thing that we create on it. Basically, it is a persistent JSON document for an IoT device in the registry and shows the device state. There are also special MQTT topics defined on the MQTT broker. For example, if you have a temperature sensor thing with the name `home_temp_sensor`, when you subscribe to the `$aws/things/home_temp_sensor/shadow/get/accepted` topic and publish a request to the `$aws/things/home_temp_sensor/shadow/get` topic, you will see the latest sensor shadow. Similarly, we publish to the `$aws/things/home_temp_sensor/shadow/update` topic in order to update its shadow. You can see the entire MQTT topic structure and message formats at this link: https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-mqtt.html

This time, I will not list the ESP32 application code, since it is very similar to the light sensor of the first example in the chapter, but instead, we will prepare the sensor hardware, flash the application from the book repository, and create a new IoT thing together to make it ready for Alexa integration:

1. Let's prepare the hardware first. We will use our ESP32-S3 Box Lite and a DS18B20 breakout board. This sensor is quite common and available in many electronics shops. The following figure shows the connection:

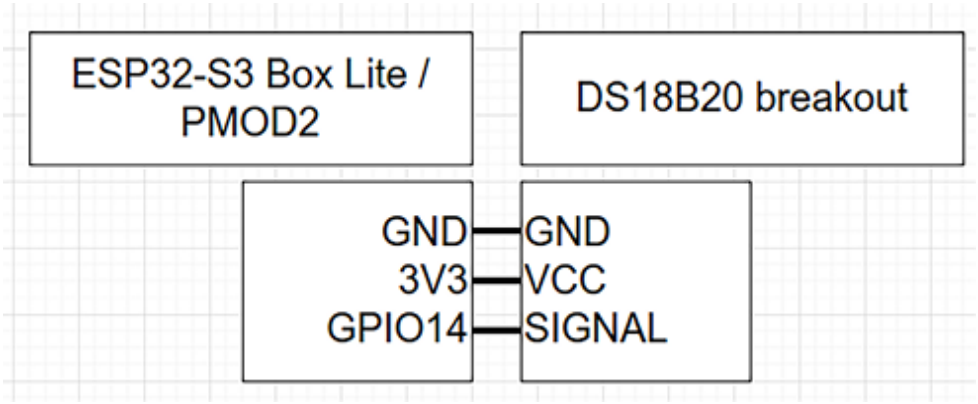


Figure 8.34: Hardware connections

2. In your local copy of the book repository, switch to the `ch8/alexa_ex` directory and create another directory inside it, with the name `tmp`:

```
$ cd ch8/alexa_ex
$ mkdir tmp
$ cd tmp
```

3. Create an IoT thing on AWS IoT Core, as we did before in the first example of the chapter. Set its name to `home_temp_sensor`. Download the connection kit into the `ch8/alexa_ex/tmp` directory and extract it there:

```
$ unzip connect_device_package.zip
$ ls connect_device_package
home_temp_sensor.cert.pem
```

```
home_temp_sensor.private.key
home_temp_sensor-Policy
home_temp_sensor.public.key
start.sh
```

4. We need the certificate and private key of the sensor. Copy them to the `tmp` directory as follows:

```
$ cp connect_device_package/home_temp_sensor.cer
$ cp connect_device_package/home_temp_sensor.pri
```

5. We also need the Amazon root CA certificate:

```
$ curl https://www.amazontrust.com/repository/Am
```

6. Now, we should have the following files under the `tmp` directory:

```
$ pwd
<book_repo_clone>/ch8/alexa_ex/tmp
$ ls -1
client.crt
client.key
connect_device_package
connect_device_package.zip
root_cert_auth.crt
```

7. We will need the **AWS IoT endpoint** to configure the application. It is written in the `connect_device_package/start.sh` script. Use its value to set the AWS endpoint environment variable:

```
$ grep endpoint connect_device_package/start.sh
node aws-iot-device-sdk-js-v2/samples/node/pub_s
$ export AWS_ENDPOINT="\"XXXX-ats.iot.eu-west-1.
```

8. We are ready to flash and monitor the application. Make your *ESP SoftAP Prov* mobile application ready if the devkit wasn't provisioned in your WiFi network earlier. If everything goes well, we should be able to connect our device to AWS IoT Core. It will start to publish temperature readings on the device shadow:

```
$ cd .. && pwd
<book_repo_clone>/ch8/alexa_ex
$ source ~/esp/esp-idf/export.sh
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting...
Detecting chip type... ESP32-S3
<more logs>
I (3226) coreMQTT: MQTT connection established w
```

9. Let's subscribe to the `$aws/things/home_temp_sensor/shadow/update/accepted` topic and see what is being published:

```
$ mosquitto_sub --cafile tmp/root_cert_auth.crt
<logs>
Client (null) received PUBLISH (d0, q0, r0, m0,
{"state":{"reported":{"temperature":16.6875}},"m
{"temperature":{"timestamp":1680891612}}},
"version":146,"timestamp":1680891612}
```

The sensor hardware is ready and successfully publishing temperature readings to the MQTT broker. When we go to the AWS console in a browser and navigate to the device shadow, we can also observe the same value there:

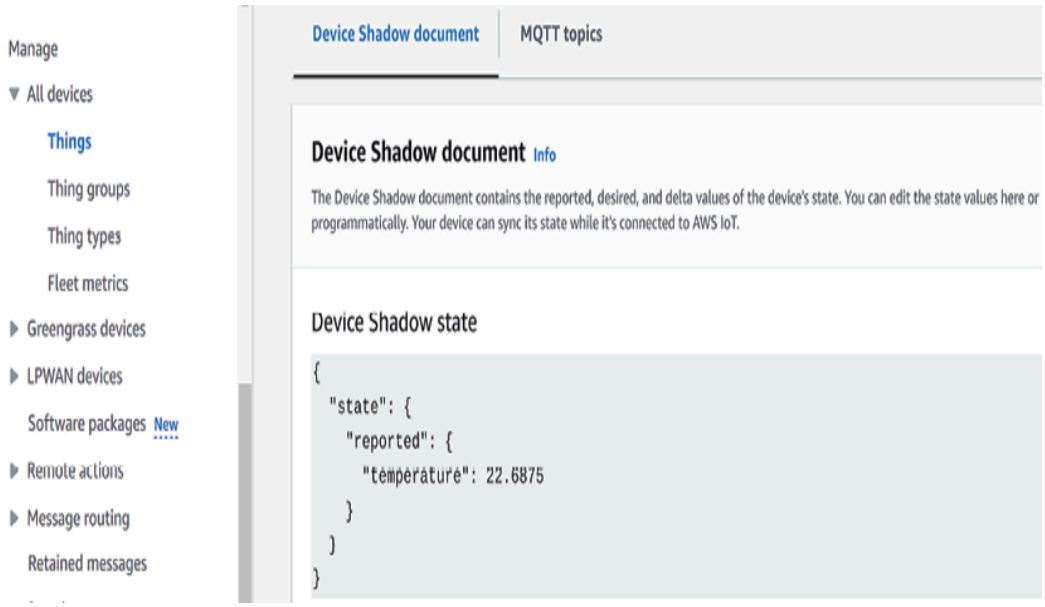


Figure 8.35: Device shadow

The active device shadow will be the data source for the AWS lambda function, which we will create, configure, and develop next.

Creating the lambda handler

We will use a lambda function as a bridge between the smart home skill and the device shadow. The skill will ask the device state, and the lambda function will retrieve the device shadow from AWS IoT Core in order to return what the skill needs.

Let's create the lambda function and configure it as follows:

1. In the AWS console, go to **Lambda functions** and add a new function with the name **alexa_temp_sensor**. The runtime is Python 3.9. Click on the **Create function** button at the bottom.

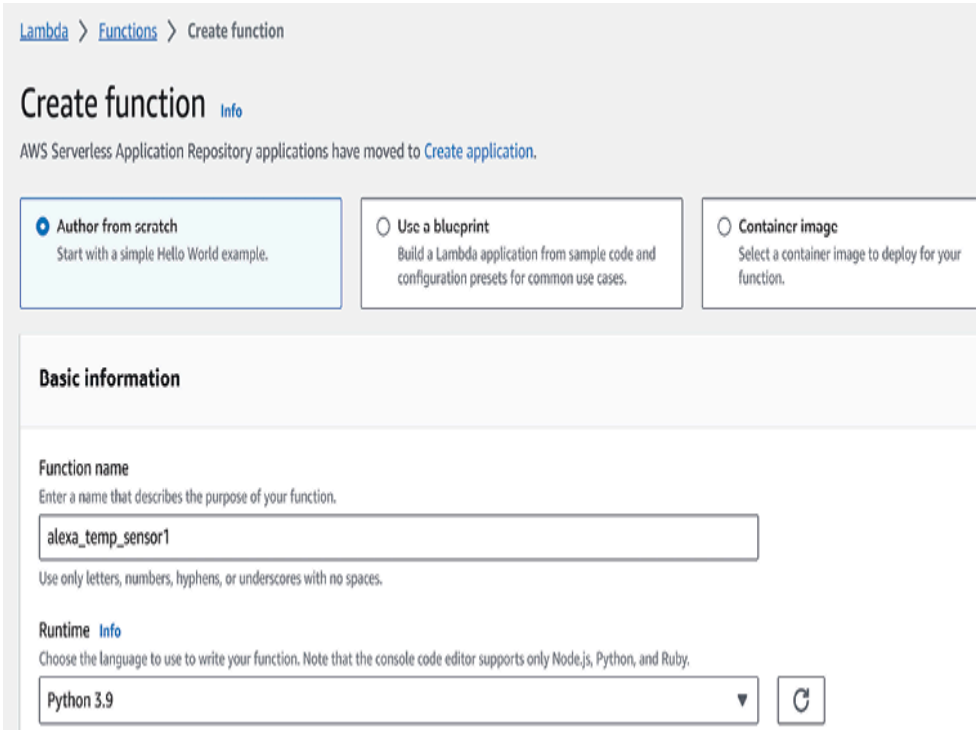


Figure 8.36: Lambda function for Alexa

2. With the default policy, a lambda function can write *CloudWatch* logs. In addition to that, we need IoT Core access to be able to read from the device shadow. For this, go to the **Configuration** tab of the lambda function and click on the role name. It will open the lambda role in another window.

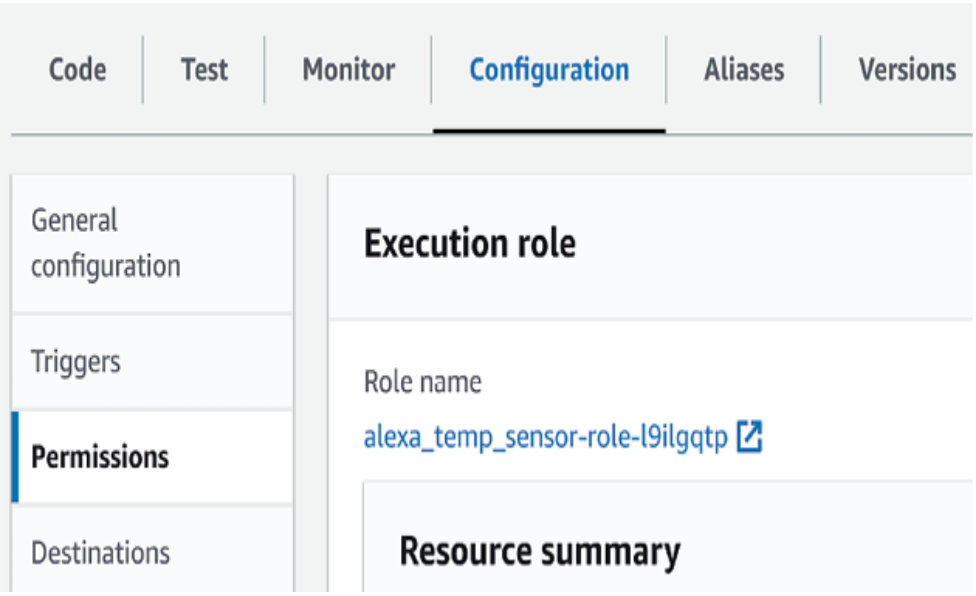


Figure 8.37: Adding IoT core permission

3. On the **Configuration/Permissions** tab, we will add an inline policy by selecting **Create inline policy** from the **Add permissions** drop-down menu.

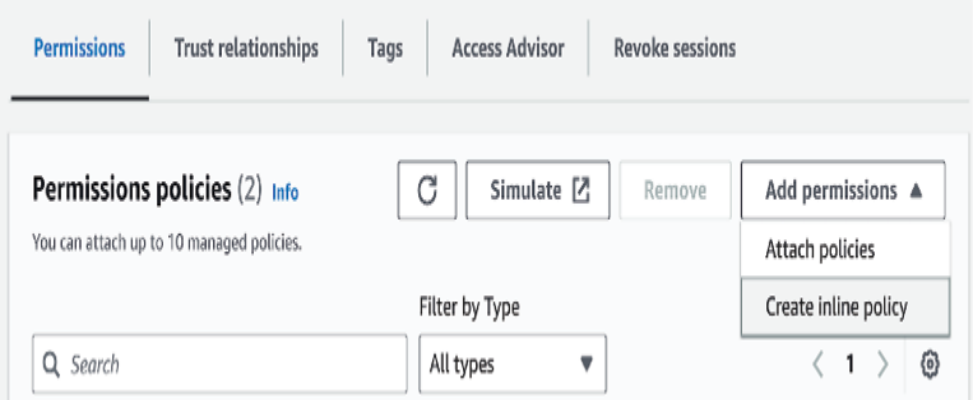


Figure 8.38: Creating an inline policy

4. Go to the **JSON** tab and paste the following JSON policy there. Then, click on the **Review policy** button. It will show a summary just before creating the inline policy. We can set the policy name as **temp_shadow_access** and create the policy by clicking on the **Create policy** button:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect",
                "iot:Receive",
                "iot:UpdateThingShadow",
                "iot:GetThingShadow"
            ],
            "Resource": "*"
        }
    ]
}
```

5. We now have two policies in the role for the lambda function. We can close this window.

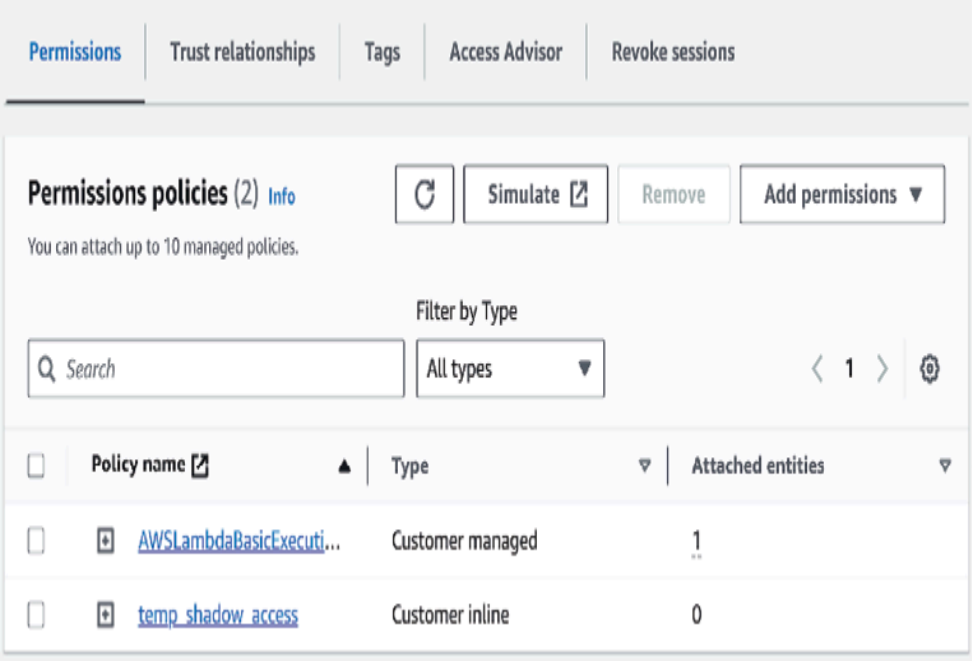


Figure 8.39: Policies in the role

6. We are basically done with the lambda configuration. Let's switch to the **Code** tab of the lambda function.

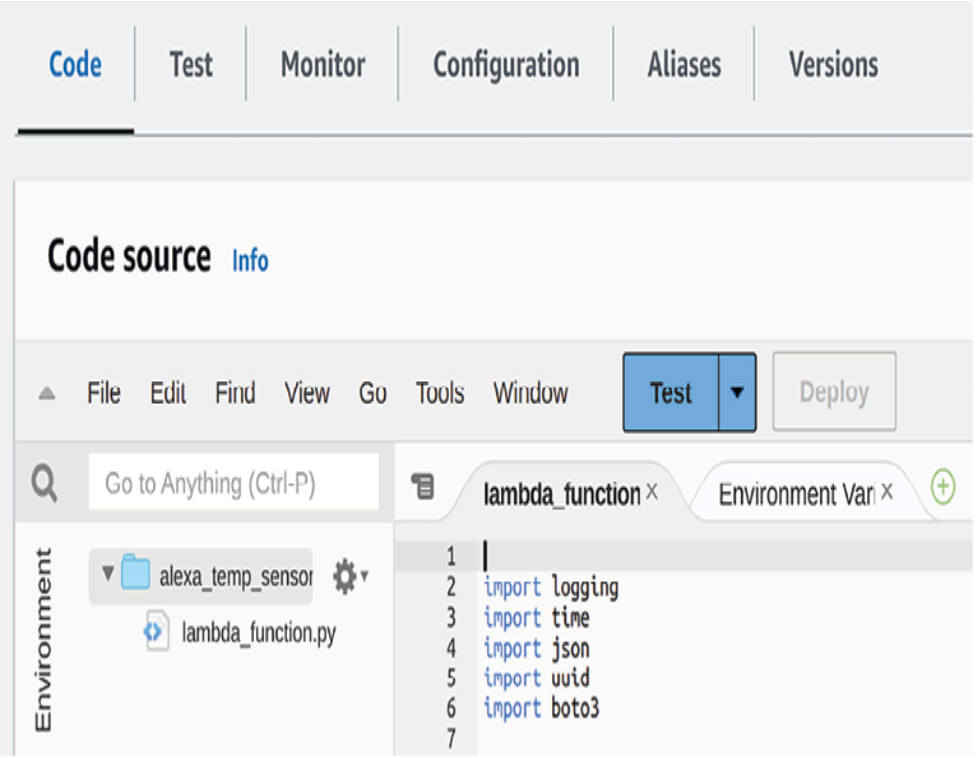


Figure 8.40: Code editor

The lambda handler is configured and ready for coding, which comes next.

Coding the lambda handler

We will develop the Python code to handle the AVS requests. Let's clear the default handler code in the editor and import some Python modules:

```
import logging
import time
import json
import uuid
import boto3
```

The module names imply their purposes. The most interesting one is probably `boto3`. It provides the functionality to interact with the AWS services, including IoT Core. Then comes some global definitions:

```
endpoint_id = "home_temp_sensor"
logger = logging.getLogger()
logger.setLevel(logging.INFO)
client = boto3.client('iot-data')
```

We set `endpoint_id` as the thing name, `home_temp_sensor`. We also define a `logger` and `client` objects. We define them outside the lambda handler in the execution environment because the

lambda environment is kept across function invocations, which provides performance gain after the first invocation, since they won't be initialized later every time the lambda function is triggered. The next thing to do is to define the response blueprints for AVS requests.

```
discovery_response = {…}
state_report = {…}
accept_grant_response = {…}
```

The response blueprints are a bit lengthy, so I couldn't list them here. However, as we discussed briefly in the introduction, we basically need three types of responses: for `discovery` requests, `AcceptGrant` requests, and device state reports. We will use these blueprints to reply to the requests from AVS. You can find the complete blueprints in the book repository under `ch8/alexa_ex/aws`.

We can add the lambda handler next:

```
def lambda_handler(request, context):
    """Lambda handler for the smart home skill
    """
    try:
        request_namespace =
            request["directive"]["header"]["namespac
        request_name = request["directive"]["header'
```

Each request from AVS has `namespace` and `name` fields in the request body. We use these two fields to understand the request type. We reply to an incoming request as follows:

```
        if request_namespace == "Alexa.Discovery" ar
            request_name == "Discover":
            response = gen_response(
                repsonse_blueprint=discovery_respons
```

The first request type that we handle is discovery. We create the response by calling the `gen_response` function, which we will define after the lambda handler. Then, we handle the state request:

```
        elif request_namespace == "Alexa" and reques
            "ReportState":
            response = gen_report_state(request
                ["directive"]["header"]["correlatior
            )
```

For the device state requests, AVS expects that the token it passes in the request is included in the reply to be able to match them. Thus, the `gen_report_state` function takes that token as its parameter. The last request type that we handle is `AcceptGrant`:

```
        elif request_namespace == "Alexa.Authorizat
            request_name == "AcceptGrant":
            response = gen_response(
```

```
                repsonse_blueprint=accept_grant_resp
            )
```

Again, we use the `gen_response` function to generate a response from the `AcceptGrant` response blueprint. We also add an `else` statement just in case an unexpected request occurs:

```
        else:
            logger.error("unexpected request")
            return response
```

After handling all the request types, we return the response to the request as follows:

```
            return response
    except ValueError as error:
            logger.error(error)
            raise
```

In addition to returning the response, we handle any exceptions that may occur during the execution. We log it, and the `lambda_handler` function completes with that. Next, we have some utility functions:

```
def get_uuid():
    return str(uuid.uuid4())
def get_utc_timestamp(seconds=None):
    return time.strftime(
```

```
            "%Y-%m-%dT%H:%M:%S.00Z",time.gmtime(seconds)
        )
```

The `get_uuid` function generates a unique identifier to be used in the replies. Similarly, `get_utc_timestamp` creates timestamp strings for the replies. In the next function, we retrieve the device shadow from IoT Core:

```
def read_temp_thing():
    response = client.get_thing_shadow(thingName=enc
    streamingBody = response["payload"].read().decod
    jsonState = json.loads(streamingBody)
    return jsonState["state"]["reported"]["temperatu
```

The `get_thing_shadow` function of the `client` object returns the device shadow as a JSON string. We extract the `temperature` value from it and return it to the caller. The `client` object has several other member functions, as documented here: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/iot-data.html. The next function we define is `gen_report_state`, which creates the response for a state request:

```
def gen_report_state(tkn):
    response = state_report
```

```
    response["event"]["header"]["messageId"] = get_u
    response["event"]["header"]["correlationToken"]
```

We first create a local copy of the report blueprint and set its `messageId` and `correlationToken` fields. Then, we continue with the other fields:

```
    ts_str = get_utc_timestamp()
    response["context"]["properties"][0]["timeOfSamp
    response["context"]["properties"][0]["value"]["v
        read_temp_thing()
    response["context"]["properties"][1]["timeOfSamp
    return response
```

The `["context"]["properties"][0]` array element corresponds to the shadow state. We set the timestamp and temperature values in it. The other array element shows the endpoint health, and we set its timestamp, too. The final function in the handler code generates the responses for the remaining two request types:

```
def gen_response(repsonse_blueprint):
    response = repsonse_blueprint
    response["event"]["header"]["messageId"] = get_u
    return response
```

The `gen_response` function is for the discovery and `AcceptGrant` requests. It simply copies the response blueprint and sets its unique identifier, before returning it to the caller.

We have completed the lambda handler for AVS. Now, we can deploy and test the function with some sample requests, imitating AVS. After deploying the function by clicking on the **Deploy** button, use the **Test** drop-down menu and select **Configure test event**.

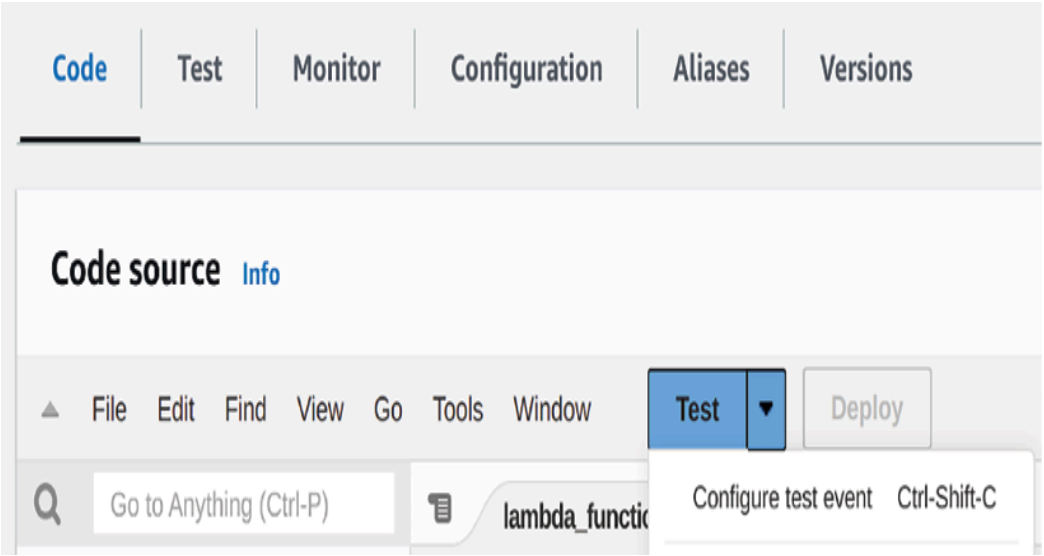


Figure 8.41: Testing the lambda

A pop-up dialog will provide a textbox for the sample request. Enter the following JSON in the **Event JSON** textbox, and give a name to the test event:

```
{
  "directive": {
    "header": {
      "namespace": "Alexa.Discovery",
      "name": "Discover",
```

```
        "payloadVersion": "3",
        "messageId": "1bd5d003-31b9-476f-ad03-71d47192
      },
      "payload": {
        "scope": {
          "type": "BearerToken",
          "token": "access-token-from-skill"
        }
      }
    }
}
```

When we click the **Test** button this time, we can see how our lambda handler runs and returns the response for the discovery request:

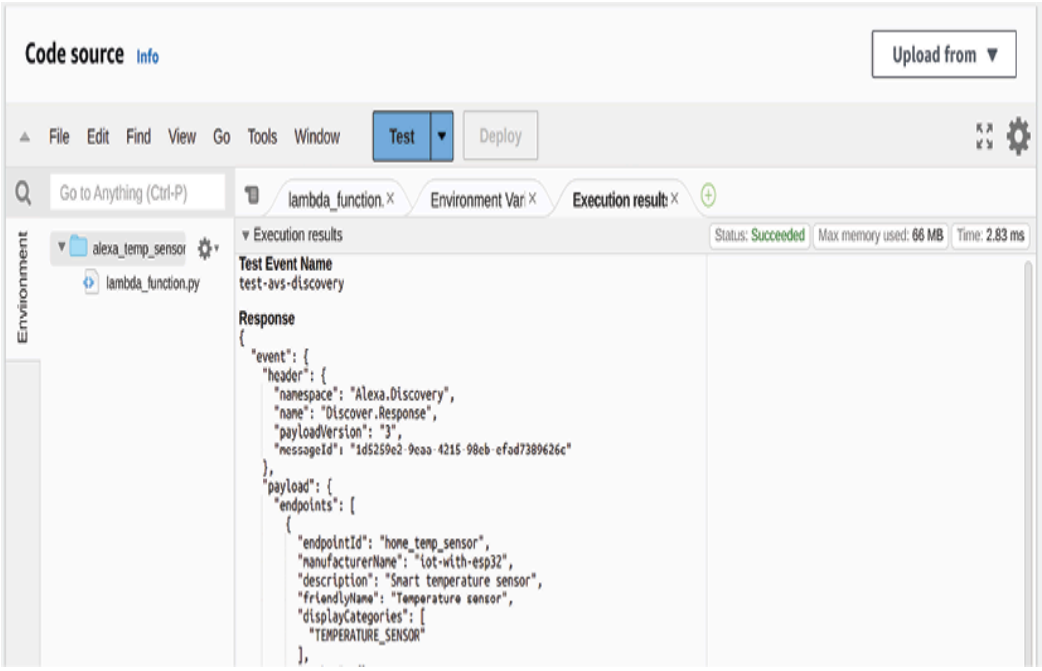


Figure 8.42: Test result

There are also other sample requests and responses in the `ch8/alexa_ex/aws` directory of the book repository. The entire list of AVS requests is provided in the **Alexa Developer Documentation** here: https://developer.amazon.com/en-US/docs/alexa/device-apis/alexa-foundational-apis.html

Now, we can move on to the last part of the Alexa integration: the smart home skill.

Creating the smart home skill

In this final part of the integration, we will link all the pieces together. To do that, let's follow the steps below:

1. Navigate to the Alexa developer console at the following URL and click on the **Create Skill** button: https://developer.amazon.com/alexa/console/ask
2. It will start a simple wizard to help us create a skill. In the first step, we provide a skill name, for instance, `home_temp_sensor_skill`. Click on the **Next** button at the top after entering the skill name.
3. The wizard asks for the type of the experience. Choose **Smart home** from the list and click the **Next** button again.
4. The last step of the wizard is the review. Continue with the **Create Skill** button.
5. The Alexa developer console shows the skill configuration page, since we have a smart home skill now. At this point, we can link the skill and the lambda handler by providing **Amazon Resource Names** (**ARNs**) to each of them. First, copy the **Skill**

ID from the **Build** tab. You can see it under the **2. Smart Home service endpoint** section.

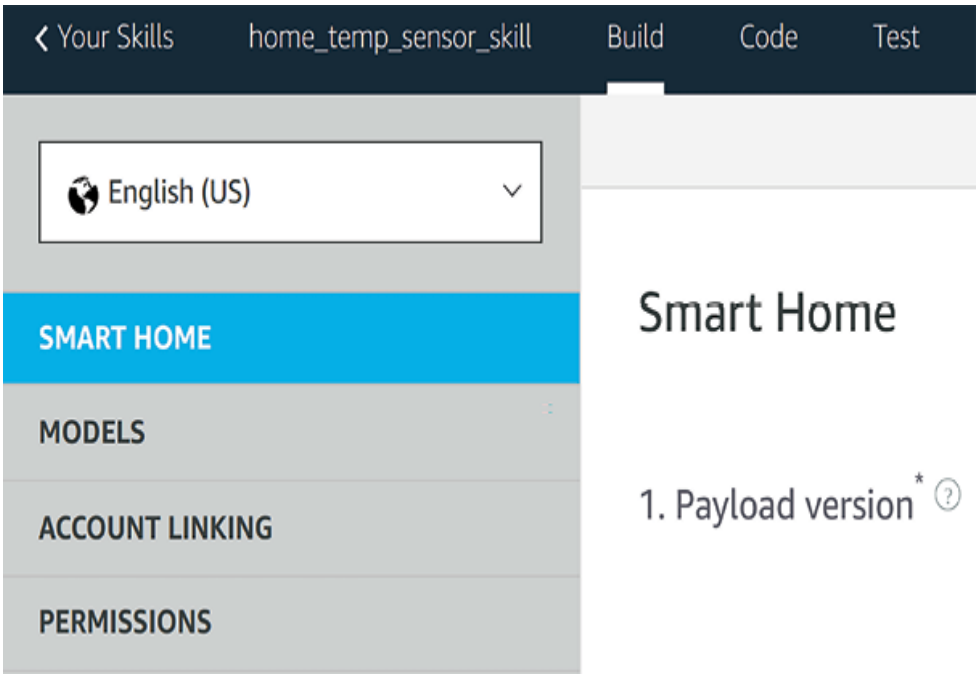


Figure 8.43: Skill Build tab

6. Go back to the lambda handler on the AWS console and click on the **Add trigger** button.

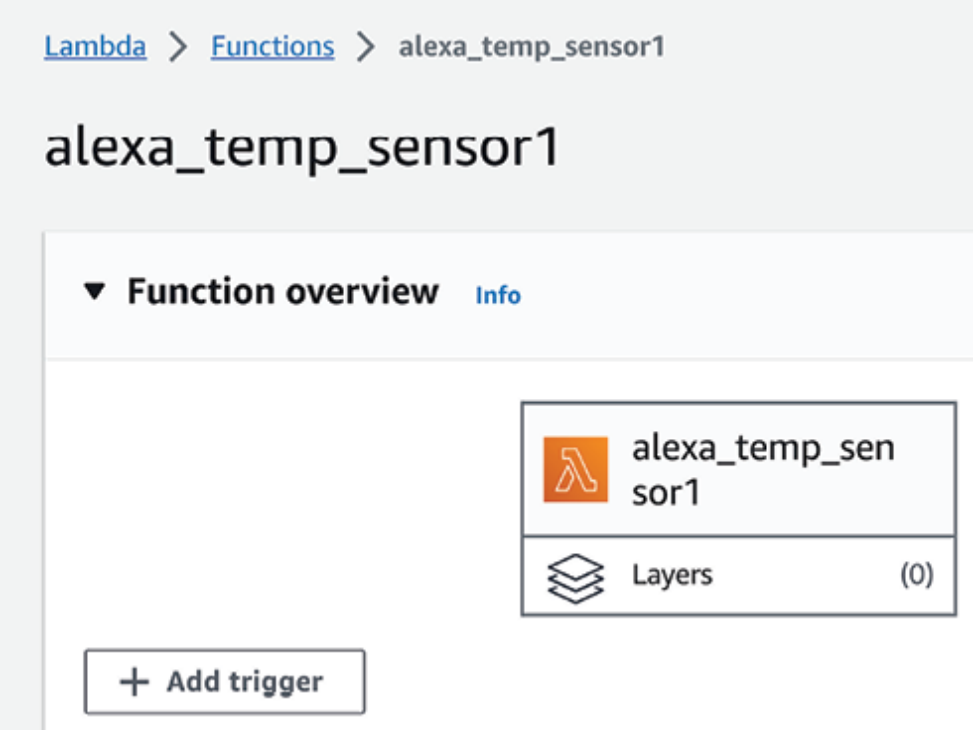


Figure 8.44: Add trigger to the lambda handler

7. Find the **Alexa** skill from the drop-down list, choose the **Alexa Smart Home** option, paste the skill ID into the box provided, and click on the **Add** button.

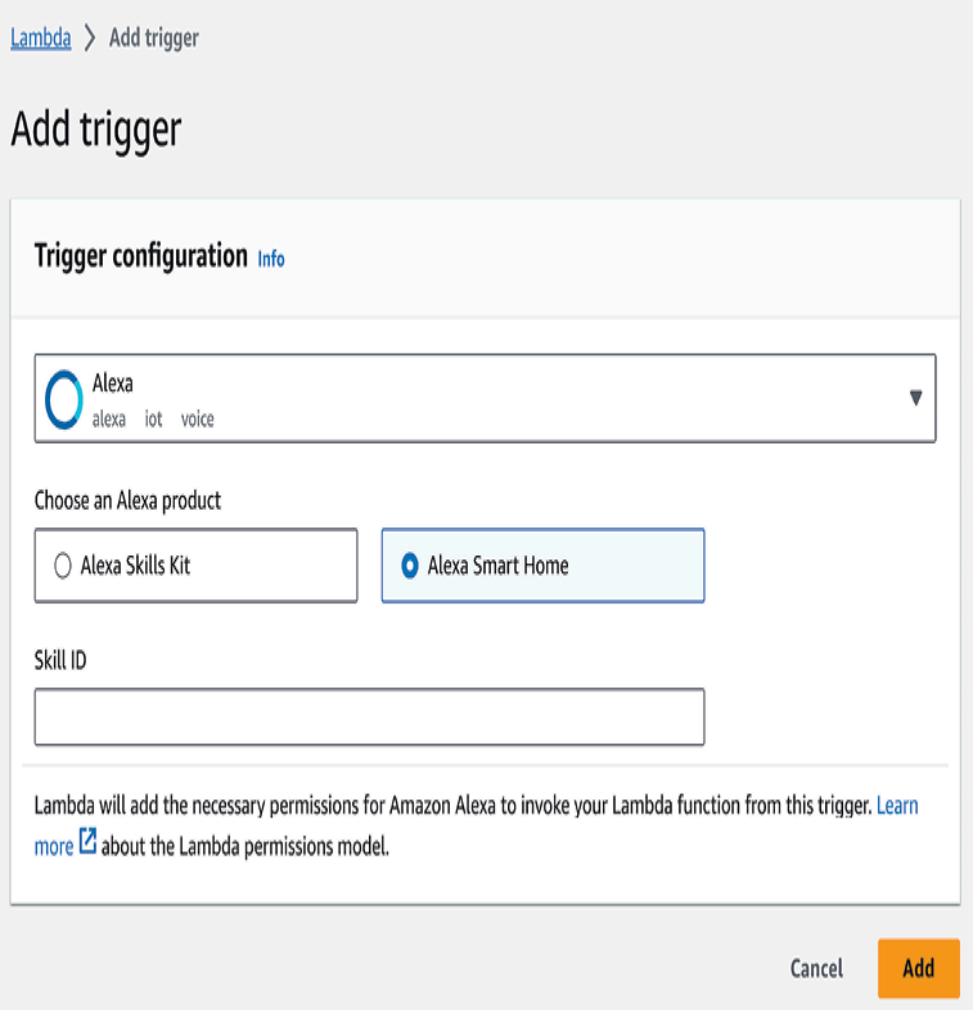


Figure 8.45: Passing the skill ID to the lambda handler

8. Copy the function ARN from the returned page and switch to the Alexa developer console. Paste the function ARN there into the **Default endpoint** box and click on the **Save** button.
9. We then need to set up account linking. This means linking the skill and our Amazon account. Click on the **Setup Account Linking** button at the bottom of the page.

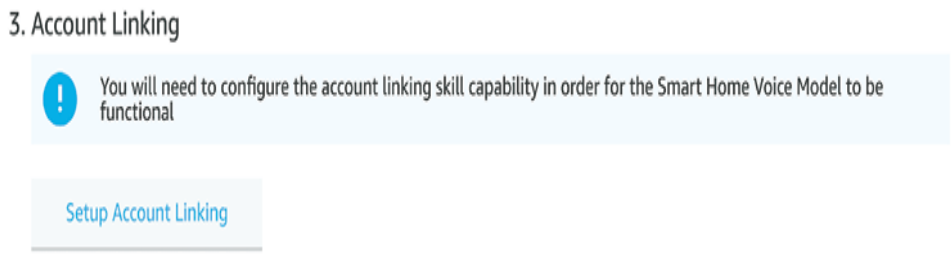


Figure 8.46: Setting up account linking

10. The opened page asks about some settings and Amazon account credentials to be linked. We also see the **Alexa Redirect URLs** at the end of the page. We will need this information when we create a security profile to be linked with the skill. For a security profile, navigate to `https://developer.amazon.com/dashboard` in another browser window. Click on the **Login with Amazon** link at the top of the page.
11. The next page shows us a user interface to create a new security profile or select from the existing ones. Let's create a new one by clicking on the **Create a New Security Profile** button.
12. Fill in the form as follows (you can set **Consent Privacy Notice URL** to anything you want).

Figure 8.47: Creating a new security profile

13. We have a client ID and client secret for our skill now. We will enter these credentials in the smart home skill later.

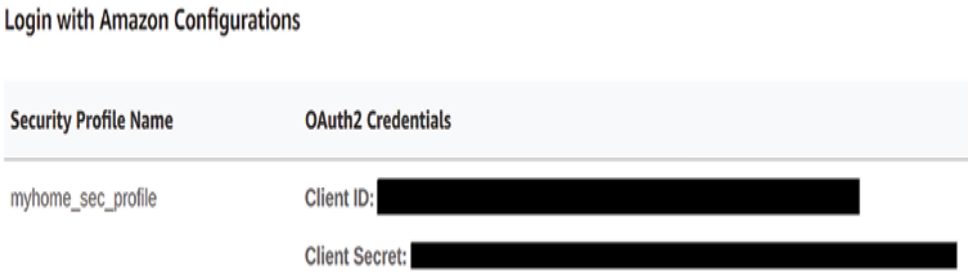


Figure 8.48: OAuth2 credentials

14. Click on the manage button (on the same row as the gear icon) of the security profile and select **Web Settings**. Enter the **Alexa Redirect URLs** in the **Allowed Return URLs** list and click on the **Save** button to persist the settings, as shown in the following figure.

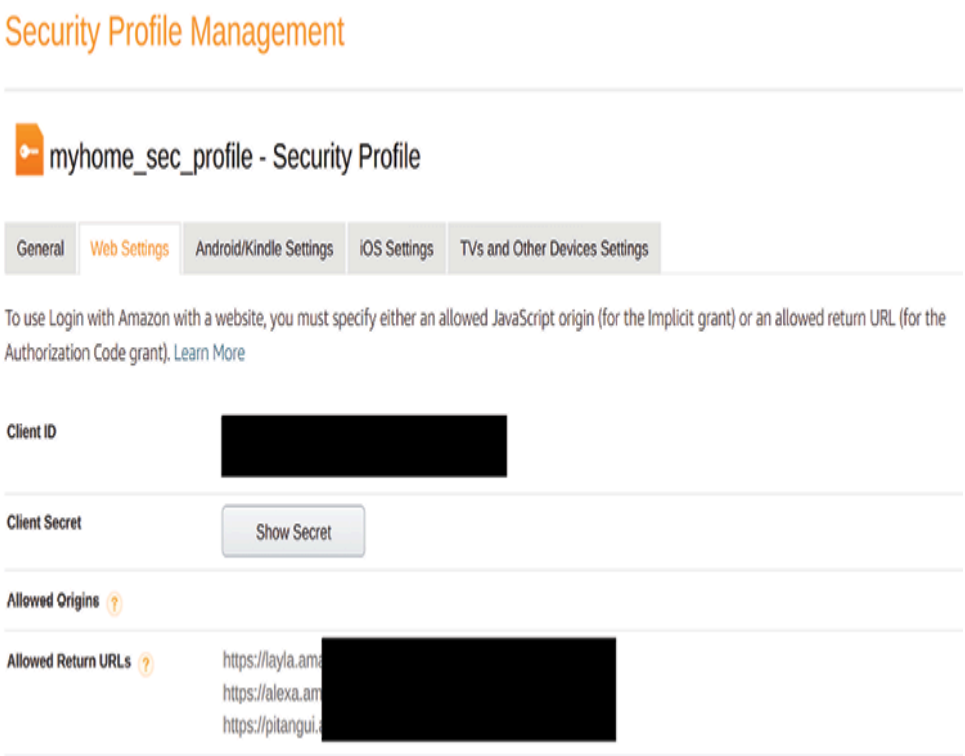


Figure 8.49: Security profile web settings

15. We are done with the security profile. We can now go back to the Alexa developer console to continue with the account linking. Fill in the form there with the following values:
    a. **Your Web Authorization URI**: `https://www. amazon. com/ap/oa`
    b. **Access Token URI**: `https: //api. amazon. com/auth/o2/token`
    c. **Your Client ID**: The client ID from the security profile
    d. **Your Secret**: The client secret from the security profile
    e. **Your Authentication Scheme**: HTTP basic
    f. **Scope**: `profile:user_id`
16. Click on the **Save** button to complete the account linking.

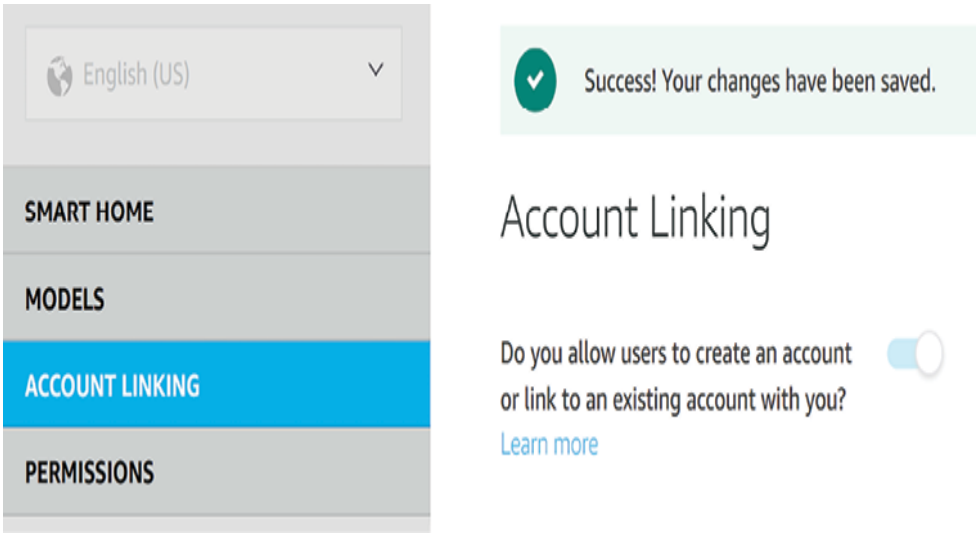


Figure 8.50: Finalizing account linking

17. One last thing: we need to enable the skill by using the **Amazon Alexa** mobile application. Open it on your mobile phone and go to the **Your Skills** tab of **Skills & Games**. You will see the temperature skill listed there. Select it to continue:

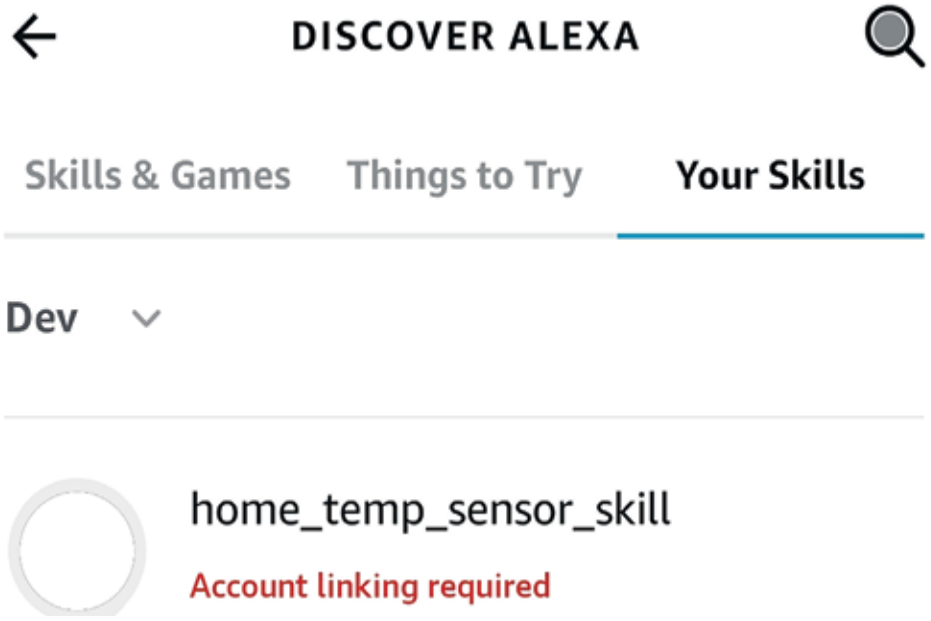


Figure 8.51: The skill on the Amazon Alexa application

18. We start the process by clicking on the **Enable** button. Then, follow the wizard to complete the skill enablement. When it is finished, we see the following message on the mobile application. To continue with the device discovery, click on the **Next** button at the bottom of the screen.

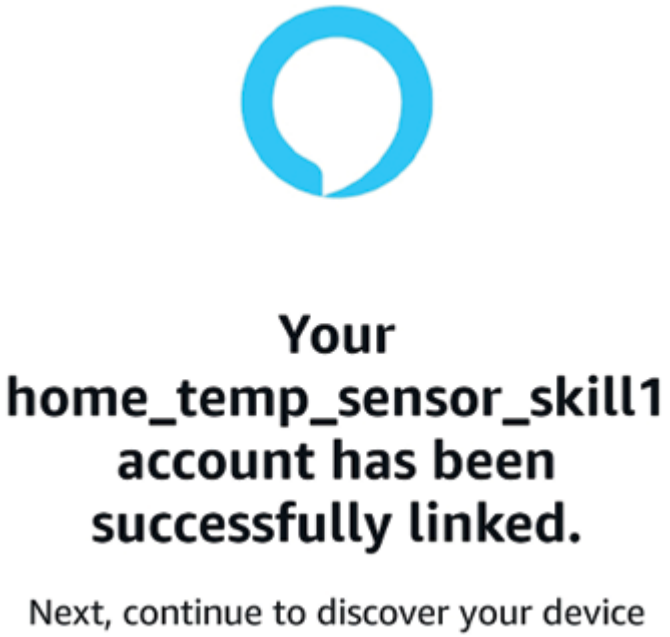


Figure 8.52: The skill is enabled

19. During the discovery phase, the lambda handler will be invoked and share the device information. The following message shows up when the discovery finishes:

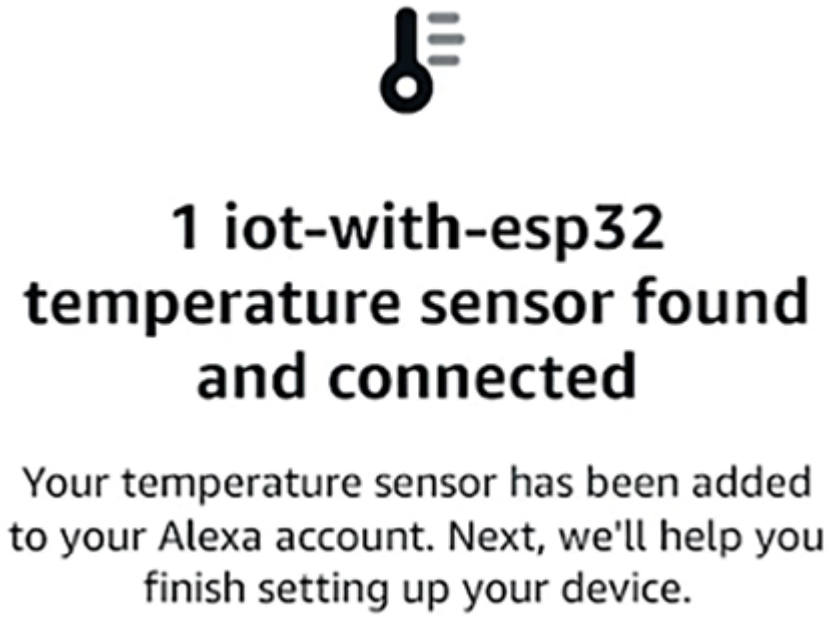


Figure 8.53: Device discovery

20. Our sensor is now listed in the **All devices** list of the application. Select **Temperature sensor** from the list.

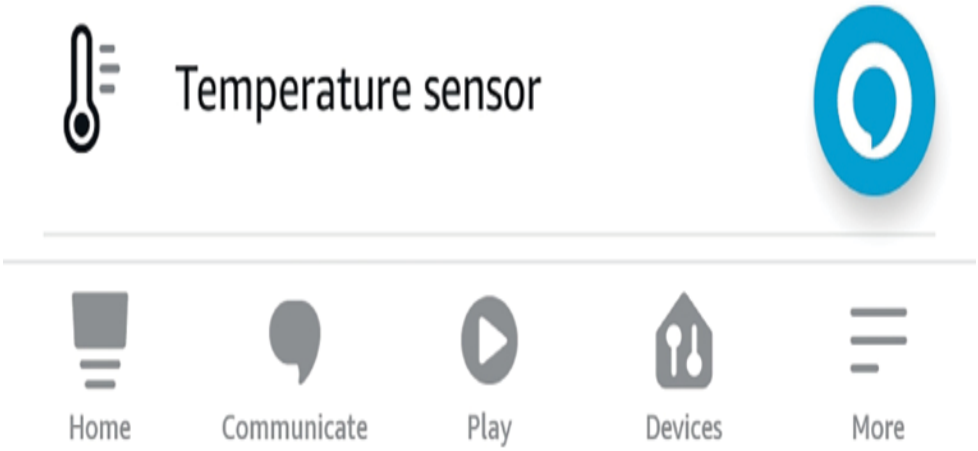


Figure 8.54: The sensor is listed in the application

21. It shows the temperature that comes from our real ESP32 temperature sensor device.

Figure 8.55: The current temperature

22. At this point, you can try an Alexa speaker if you have one. Another option is to use the Alexa developer console. Go to the **Test** tab of the skill. Hold the mic there and say "*the temperature sensor*." It will reply with the temperature value in Fahrenheit, since the default region is **English (US)**.

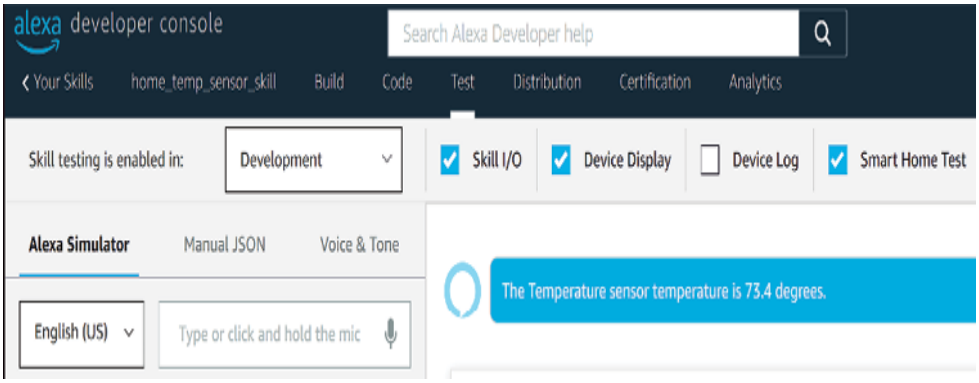


Figure 8.56: The Test page on the Alexa developer console

Finally, the Alexa integration is done and working as intended. I have to admit that there are many steps to integrate Alexa, and it requires firm focus to complete them successfully. Nonetheless, they are necessary if you need to support Alexa voice features in your product.

Troubleshooting

This example has several stages, and each stage contains many steps. Therefore, if you encounter unexpected behavior, make sure each component works properly, independent of each other.

We have temperature sensor hardware. If you suspect there is an issue with it, check the serial console logs and see the thing shadow on AWS IoT Core. The lambda handler is quite easy to test, and you can see the logs on the execution output when you invoke the function manually. The most challenging part of the example is probably the smart home skill configuration. It consists of more than 20 steps and also includes manual copy-pastes between different windows. The steps are mostly managed by wizards, but the entire integration still requires attention. Here are some tips that might help you in case of issues:

- You really need to be careful with the copy-pastes between windows in those steps. Use the **copy** buttons wherever available. If you receive an error in any of the steps, read the error message carefully. They are helpful more than anything.
- Policies are easy to make mistakes with. Remember to check the *CloudWatch logs* to see if anything is logged about permissions.
- The Alexa developer documentation is the best resource to learn about the subject in depth. Before trying something new, it might be a good idea to check the documentation to see if it provides any guidance. Here is the root link for the smart home skills: https://developer.amazon.com/en-US/docs/alexa/smarthome/understand-the-smart-home-skill-api.html

Summary

It is almost impossible to think of IoT without cloud platforms. In this chapter, we talked about **Amazon Web Services** as an example and learned how to integrate ESP32 with the AWS cloud. The first example in the chapter was an AWS-connected light sensor. We created an **IoT thing** on **AWS IoT Core**, and we configured the ESP32 application with its credentials. The light sensor sent light readings from an LDR to IoT Core over MQTT. **IoT visualization** was another subject of the chapter. We created an **IoT rule** on AWS IoT Core to forward light readings to a managed **Timestream** database. **Grafana** was the tool to visualize time-series light data. We configured a Grafana panel to read from the Timestream table and show the light values, with associated timestamps.

Voice assistants are also popular in the IoT world. Consumer products usually come with voice service integrations as an alternative method of user interface. In the last example, we discussed how to integrate an IoT sensor with the **Amazon Alexa** voice service by creating a **smart home skill**. A **lambda function** was the bridge between the **device shadow** of the sensor and the skill.

With this chapter, we learned how to develop an IoT product end to end. We can read from sensors, such as a light sensor or temperature sensor, attached to ESP32, connect ESP32 to the local WiFi network, and then send data to the AWS cloud. In the next chapter, we will practice all these skills in a project. It will be a smart home product with sensors and actuators, where the ESP32 devices will talk to each other to accomplish tasks and communicate events to the cloud.

Questions

Try to answer these questions as an overview of the chapter:

1. Which of the following is used as the SDK to connect ESP32 to the AWS cloud?
    a. AWS IoT Core SDK
    b. AWS IoT Device SDK
    c. AWS IoT Device Defender
    d. AWS IoT Greengrass
2. Which is not required when we develop an ESP32 application to publish MQTT messages on AWS IoT Core?
    a. Creating an IoT thing
    b. Embedding the credentials in the application
    c. Defining an IoT rule to pass messages
    d. Defining the right policies to publish messages
3. Which one is not correct about Grafana?
    a. It is a tool for IoT visualization.
    b. It can integrate data from different sources on the same panel.
    c. You need to manage an EC2 instance to have Grafana
    d. It has the ability to generate alerts over data
4. Assume that you are integrating an ESP32 device with AVS. You see that the device shadow is updated, but the Alexa speaker doesn't reply with the correct value. Which component would be the first one to check?

a. The ESP32 application
b. The lambda handler
c. The IoT rule
d. The device policies

5. If the Amazon Alexa mobile application cannot find a device at the discovery stage, which approach would likely lead to a solution faster?

a. Checking ESP32 serial logs
b. Checking CloudWatch logs
c. Checking account linking
d. Testing the lambda handler with a state request

Further reading

- *Building IoT Visualizations using Grafana*, Rodrigo Juan Hernández, Packt Publishing (https://www.packtpub.com/product/building-iot-visualizations-using-grafana/9781803236124): The book discusses Grafana in depth, including both management and usage. You can find practical examples of dashboard design and integration with other data sources.
- *Hands-On Chatbot Development with Alexa Skills and Amazon Lex*, Sam Williams, Packt Publishing (https://www.packtpub.com/product/hands-on-chatbot-development-with-alexa-skills-and-amazon-lex/9781788993487): Although the main subject of the book is chatbot development, it explains the basics of the Alexa skill

development well, especially in *Chapter 2*, *Getting Started with AWS and Amazon CLI*, and *Chapter 3*, *Creating Your First Alexa Skill*. The book also covers several other Amazon services, such as Amazon S3 and Amazon DynamoDB, to help develop complete cloud applications.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

9

Project – Smart Home

Developing an IoT product requires many hardware and software components to work together flawlessly for the best customer experience. As IoT developers, we might not be experts in all of them, but we still need to know the components and how to integrate them into a project. All that is needed is practice to gain experience with the options. A smart home is a good example because it has all the main IoT components operating together. It is a well-known field in the IoT world: a smart home solution incorporates sensors and actuators to help people monitor and control their homes and entertainment systems, which are integrated in the solution for a complete experience. The devices in a smart home usually run in the same local network. The solution implements a backend infrastructure as the backbone of the entire system, and web or mobile applications are included for end users to use their devices remotely. This chapter is a good opportunity to learn how to build a full-fledged smart home solution without developing all of the system components but integrating them to work as a whole.

In this chapter, we will discuss the following topics in relation to developing a smart home solution:

- The feature list of the smart home solution
- Solution architecture

- Implementation
- Testing project
- Troubleshooting
- New features

Technical requirements

The hardware requirements of the chapter are:

- ESP32-C3 DevkitM-1
- A relay module (any 3.3V 1-channel relay with optocoupler isolation, available in online stores)
- ESP32-S3 Box Lite
- A BME280 breakout
- A TSL2561 ambient light sensor breakout
- Jumper wires

We will also need two mobile applications to test the project. They are:

- **ESP RainMaker**: The companion application that comes with the RainMaker platform to add devices and manage them in the platform. It is available for both Android and iOS.
- **Amazon Alexa**: The mobile application by Amazon to control and monitor Alexa-enabled devices. It is available for both Android and iOS.

It is not a mandatory tool since I will provide the generated code, but the GUI in the sensor application is designed by using **SquareLine Studio** (available at https://squareline.io/ for all mainstream platforms).

You can find the project code in the GitHub repository here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch9

The feature list of the smart home solution

The smart home solution will have the following features:

- A **plug** that will be able to power small home appliances.
- A **mobile application** to turn the plug on or off.
- The plug firmware can be **updated over-the-air remotely**.
- A **multisensor** that will show temperature and ambient light intensity on its display.
- Temperature and light intensity values can be monitored remotely via the mobile application.
- The multisensor firmware can be updated over the air remotely.
- The solution will support **automation**. Users will be able to define rules such as if the light intensity is over 20, turn off the plug.
- The solution will integrate **Amazon Alexa** as the voice interface.

Let's discuss a possible solution architecture for these features.

Solution architecture

The solution will integrate common components that a typical smart home product can usually have. There will be smart devices, a plug and a sensor, applications, a cloud system that provides control and

monitoring facilities, and a mobile application for end user to access the devices remotely. We will employ **ESP RainMaker** (https://rainmaker.espressif.com/) as the cloud system that the devices connect to. ESP RainMaker also comes with a mobile application for remote access to the devices. The following figure shows this architecture:

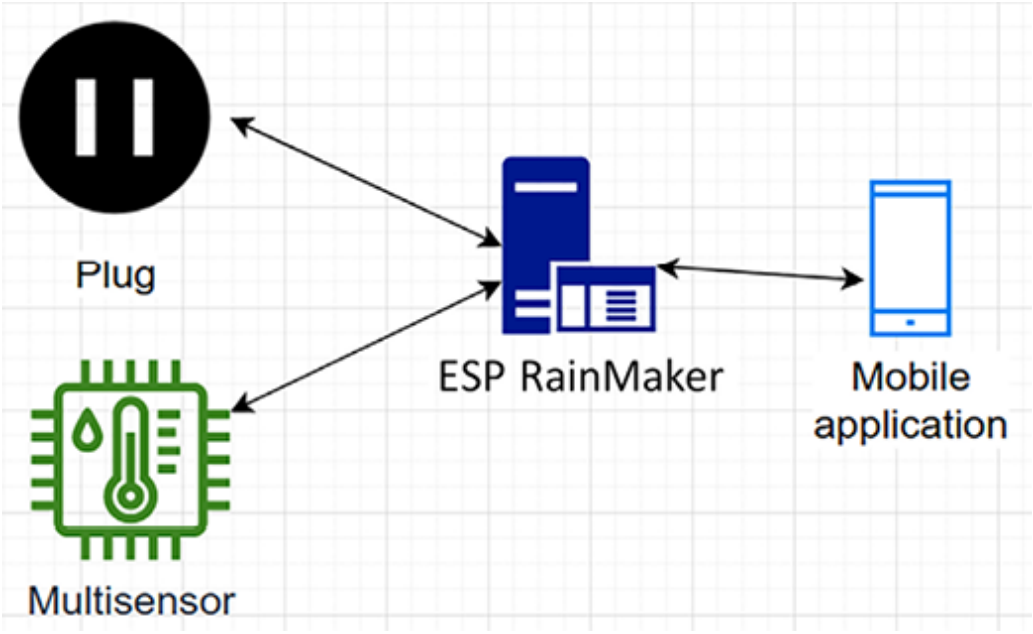


Figure 9.1: Solution architecture

ESP RainMaker provides all the backend functionality that we need in order to implement this project. By integrating with the ESP RainMaker infrastructure, we will have the following capabilities:

- Device provisioning, that is, adding/removing devices via the mobile application
- Remote monitor and control
- OTA device firmware upgrade
- Alexa Voice Services integration

Although we have ESP RainMaker as cloud infrastructure and use it in the solution as it is, we still need to design and develop the plug and multisensor devices and integrate them with ESP RainMaker. Let's prepare the devices by using our devkits.

Setting up plug hardware

We can use ESP32-C3 DevkitM-1 as the plug hardware by connecting a relay to it. It has a button to control the relay module and also, we can use its LED as a feedback mechanism about its state. The following Fritzing sketch shows the connection between the devkit and the relay:

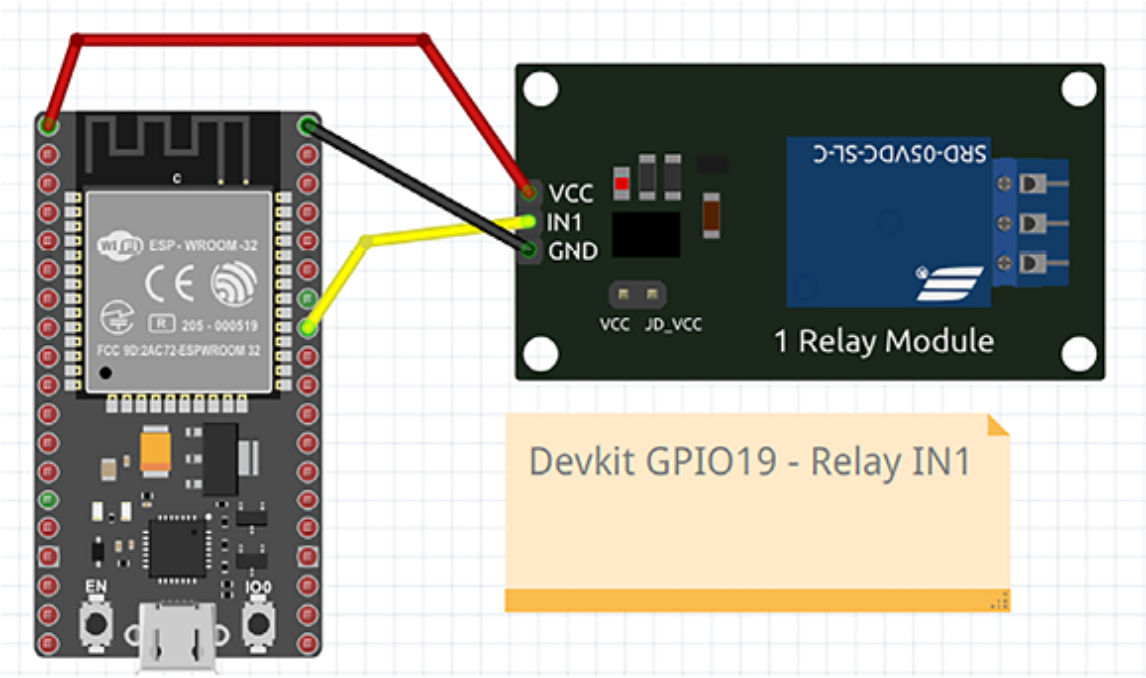


Figure 9.2: Plug connections

Let's see the multisensor hardware next.

Setting up multisensor hardware

ESP32-S3 Box Lite will be the multisensor in the project. We will connect a TSL2451 breakout to the devkit as the source for the ambient light measurements and a BME280 breakout for the temperature readings. The connections are shown in the following Fritzing sketch:

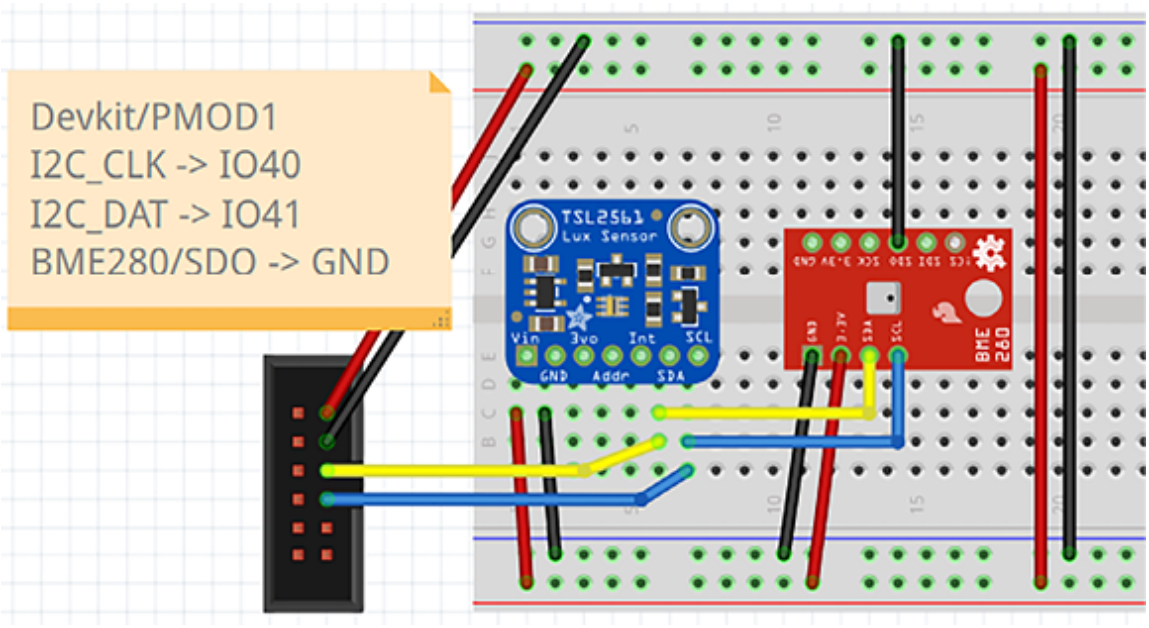


Figure 9.3: Multisensor connections

The connections of the multisensor are the same as the one that we discussed in *Chapter 3, Using ESP32 Peripherals,* specifically the *Interfacing with sensors over Inter-Integrated Circuit (I^2C)* section. We can attach multiple sensors on the same I^2C bus, thus TLS2561 and BME280 can share the same pins of the devkit.

Before moving on to the implementation, let's have a quick overview of the software architecture to understand what we will develop.

Software architecture

There will be some commonalities between the plug code and the multisensor code, such as Wi-Fi connectivity and RainMaker definitions. We can encapsulate them in the following classes:

- `AppDriver`: This will initialize the NVS and Wi-Fi peripherals.
- `AppNode`: This is the base class for a RainMaker node. We will derive the plug node and the multisensor node from it. It will implement the common functionality for the derived classes, such as RainMaker event handling and system services.

The plug application will be relatively easier with a single class definition:

- `AppPlug`: This will derive from the `AppNode` class to define a plug node on RainMaker. It will also make use of the ESP32-C3 DevkitM-1 **Board Support Package** (**BSP**) implementation (`AppEsp32C3Bsp`) of *Chapter 7*, *ESP32 Security Features for Production-Grade Devices*, for driving the button and LED of the devkit.

The multisensor application has more requirements. The following classes are part of the multisensor implementation:

- `AppSensorNode`: Similar to `AppPlug`, this class will define a RainMaker node for the multisensor application. It will also drive BME280 and TSL2561.
- `AppUi`: We will show the light and temperature readings on the ESP32-S3 Box Lite display. This class will initialize the underlying display drivers and use LVGL to show the readings on the display.

The next class diagram depicts the responsibilities and interactions between the classes:

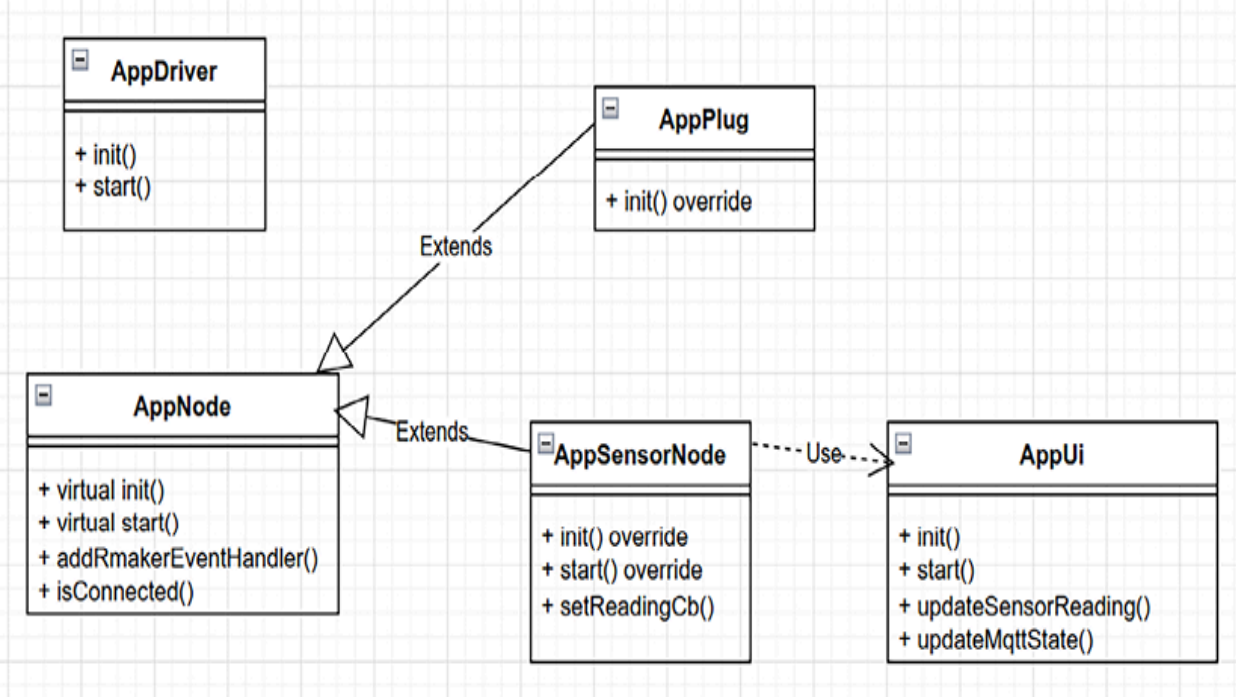


Figure 9.4: Class diagram of the device applications

Now that we understand the solution architecture, we can now implement the project based on this design.

Implementation

The solution architecture defines three different software packages:

- **Common libraries and components**: We can reuse the third-party libraries and the components that we developed in the previous examples. After all, it is the whole point of modular software design.
- **Plug application**: The plug application will have the `AppPlug` class on top of the common libraries to satisfy the project

requirements.

- **Multisensor application**: We will develop the `AppSensorNode` and `AppUi` classes in the multisensor application. The application will integrate all the classes as a whole as described in the requirements.

Let's start with the common libraries.

Preparing common libraries

The requirements and the design reveal some of the third-party libraries that we are going to need to develop the project. They are:

- ESP RainMaker for cloud connectivity (https://github.com/espressif/esp-rainmaker)
- A modified version of the ESP-IDF library to integrate TLS2561 and BME280 (https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/components/esp-idf-lib)
- LVGL for ESP32 (https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/components/lvgl)
- Board support packages (https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/components/bsp)

You can definitely integrate them into the project by yourself, but some of the libraries require modifications to work with our hardware setup, so I suggest using the book repository as it is to get to the point fast.

After having all libraries available to the project, we will create an ESP-IDF component to be used with the plug and multisensor applications.

Creating IDF component

The steps to create the IDF component are the following:

1. Create a directory for the component with the name `ch9_common`.
2. Create the *cmake* build configuration script, `CMakeLists.txt`, for the component under `ch9_common`. Its content is:

```
idf_component_register(INCLUDE_DIRS "include")
```

3. Create another directory for the header files with the name `ch9_common/include`.
4. Under the `ch9_common/include` directory, add two files for the common classes: `AppDriver.hpp` and `AppNode.hpp`.
5. When all is done, we should have the following directory structure:

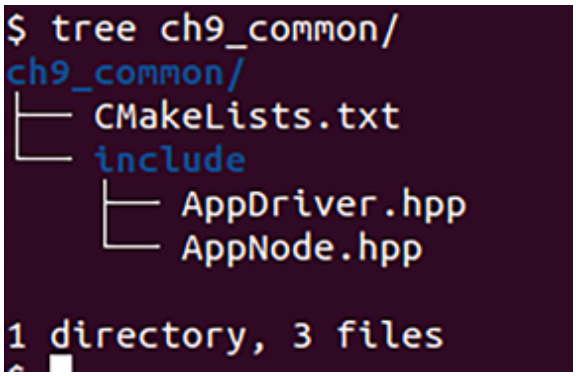


Figure 9.5: Directory structure of the component

We have the directory structure for the component and can continue to code the classes in it.

Coding IDF component

We can begin with the `AddDriver` class implementation in `AddDriver.hpp`.

```
#pragma once
#include <esp_log.h>
#include <nvs_flash.h>
#include <app_wifi.h>
```

We include the header files first. We will initialize the NVS and Wi-Fi hardware in `AddDriver`. Therefore, we add the headers for them. The `app_wifi.h` header file comes with the ESP RainMaker library. Next, we will define the class and its `init` function:

```
namespace app
{
    class AppDriver
```

```
    {
    public:
        void init()
        {
            esp_err_t err = nvs_flash_init();
            if (err == ESP_ERR_NVS_NO_FREE_PAGES ||
                ESP_ERR_NVS_NEW_VERSION_FOUND)
            {
                nvs_flash_erase();
                nvs_flash_init();
            }
            app_wifi_init();
        }
```

The `init` function starts with the NVS initialization because the Wi-Fi library needs access to the NVS to read stored Wi-Fi credentials. The `app_wifi_init` function initializes the Wi-Fi peripheral of ESP32. Then we define the `start` function of the class:

```
        void start()
        {
            app_wifi_start(POP_TYPE_RANDOM);
        }
    }; // end of class
} // end of namespace
```

The `start` function calls the `app_wifi_start` function of the ESP RainMaker library, which checks the Wi-Fi credentials. If there are no stored Wi-Fi credentials, then the `app_wifi_start` function will

start the provisioning process for ESP RainMaker. We will need the ESP RainMaker mobile application when the provisioning process starts on the device.

The `AppDriver` class is done, and we move on to the implementation of the `AppNode` class by editing the `AppNode.hpp` file:

```
#pragma once
#include <cinttypes>
#include <vector>
#include <string>
#include <functional>
#include <cstring>
#include <esp_log.h>
#include <esp_err.h>
#include <esp_event.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <esp_rmaker_core.h>
#include <esp_rmaker_standard_types.h>
#include <esp_rmaker_standard_params.h>
#include <esp_rmaker_standard_devices.h>
#include <esp_rmaker_common_events.h>
#include <esp_rmaker_utils.h>
#include <app_insights.h>
#include <esp_rmaker_ota.h>
#include <esp_rmaker_schedule.h>
#include <esp_rmaker_mqtt.h>
#include <esp_rmaker_scenes.h>
```

We first include a bunch of header files. The first group contains the header files from the standard library. The second group is for the functionality that comes with ESP-IDF. The third group allows us to use the ESP RainMaker functions and structures. Then we will add the type definitions and aliases to be shared with client codes:

```
namespace app
{
    using InitDevice_f = std::function<void(esp_rmak
        void *)>;
    using RmakerEventHandler_f = std::function<void(
        esp_event_base_t, int32_t, void *)>;
```

We have two function aliases as callbacks to be used in the `AppNode` class. The derived classes (`AppPlug` and `AppSensorNode`) will provide `AppNode` with an `InitDevice_f` function to insert their specific RainMaker definitions. The `RmakerEventHandler_f` alias shows the prototype for handling RainMaker events by any client code. Then we have a type definition for the `AppNode` construction:

```
    typedef struct
    {
        std::string node_name;
        std::string node_type;
        std::string device_name;
        std::string device_type;
        InitDevice_f init_device;
```

```
        void *init_device_arg;
    } AppNodeParams_t;
```

The `AppNodeParams_t` type definition creates an initialization package for the `AppNode` construction. The `AppNode` instance will store a parameter of this type and use it while creating the RainMaker node. The class definition comes next:

```
    class AppNode
    {
    private:
        static void eventHandler(void *arg, esp_even
           event_base, int32_t event_id, void *event
```

The first function in the `AppNode` class is the static `eventHandler` function in the private section. We will register it as the RainMaker event handler. Then we define a vector of event handlers for the clients of the `AppNode` class:

```
        using RmakerEventHandlerItem_t = struct
        {
            RmakerEventHandler_f fn;
            void *arg;
        };
        std::vector<RmakerEventHandlerItem_t> event_
```

The `event_handlers` vector will keep a list of RainMaker event handler functions to be run when a RainMaker event occurs. A client

code will register its handler function to the `AppNode` instance and the `AppNode` instance will forward the event to the client code by calling its registered function. This mechanism will be clearer when we implement the static `eventHandler` function that we declared previously. Let's move on to the protected section of the `AppNode` base class:

```
    protected:
        esp_rmaker_node_t *m_rmaker_node;
        esp_rmaker_device_t *m_device;
        bool m_connected;
        app::AppNodeParams_t m_init_params;
```

In the protected section, we only define several member variables of the `AppNode` class to be shared with derived classes. Derived classes will have access to the underlying node definition (`m_rmaker_node`), device definition (`m_device`) and the initialization parameters (`m_init_params`). There is also another member variable, `m_connected`, which indicates whether the node is connected to the MQTT broker of RainMaker. We continue with the public section of the class:

```
    public:
        AppNode() : m_rmaker_node(nullptr),
                    m_device(nullptr),
                    m_connected(false),
                    m_init_params({"", "", "", "", n
                        nullptr})
```

```
        {
        }
```

The first public function is the default constructor. It only sets the initial values of the member variables that we defined in the protected section. Next comes another constructor:

```
        AppNode(app::AppNodeParams_t p) : m_rmaker_n
                                          m_device(n
                                          m_connecte
        {
            m_init_params = p;
        }
```

This version of the constructor takes an `AppNode` initialization argument and simply sets the `m_init_params` member variable with that argument. The next function we define is `init`:

```
        virtual void init()
        {
            esp_event_handler_register(RMAKER_COMMON
                ESP_EVENT_ANY_ID, &eventHandler, thi
            esp_event_handler_register(RMAKER_OTA_EV
                ESP_EVENT_ANY_ID, &eventHandler, thi
```

The `init` function is the one that creates the underlying RainMaker node. It is a virtual function so a derived class can override it with its

own implementation. In the function body, we first register the RainMaker events by passing the `eventHandler` function of the class. Then we create the actual RainMaker node as the following:

```
            esp_rmaker_time_set_timezone(
                  CONFIG_ESP_RMAKER_DEF_TIMEZONE);
            esp_rmaker_config_t rainmaker_cfg = {
                .enable_time_sync = true,
            };
            m_rmaker_node = esp_rmaker_node_init(&ra
                m_init_params.node_name.c_str(),
                m_init_params.node_type.c_str());
```

After setting the time zone and enabling **Simple Network Time Protocol** (**SNTP**) time synchronization in a configuration variable, we call the `esp_rmaker_node_init` function of the RainMaker library. It will create the underlying RainMaker node with the node name and node type as given in the `m_init_params` member variable. Next, we define the RainMaker device:

```
            m_device = esp_rmaker_device_create(
                m_init_params.device_name.c_str(),
                m_init_params.device_type.c_str(), (
            esp_rmaker_device_add_param(m_device,
                esp_rmaker_name_param_create(
                  ESP_RMAKER_DEF_NAME_PARAM,
                  m_init_params.device_name.c_str())
```

We define the RainMaker device by calling the `esp_rmaker_device_create` function from the RainMaker library. We also add a standard name parameter to the device as the display name on the mobile application by calling `esp_rmaker_device_add_param`. As we discussed in *Chapter 7*, *ESP32 Security Features for Production-Grade Devices*, specifically the *Sharing data over secure MQTT* section, a RainMaker node can have one or more RainMaker devices, and a RainMaker device definition can contain one or more RainMaker parameters. The name parameter is the first parameter of the RainMaker device. Next, we will add the RainMaker parameters that are specific to the derived class:

```
            if (m_init_params.init_device != nullptr
            {
                m_init_params.init_device(m_device,
                    m_init_params.init_device_arg);
            }
            esp_rmaker_node_add_device(m_rmaker_node
```

If a derived class provides a `m_init_params.init_device` callback function, we run it here. This is the place where the `AppNode` base class specializes the RainMaker device definition for the concrete derived class. Then we finalize the basic node definition by associating the device and node with the help of the `esp_rmaker_node_add_device` function. We are not done yet with the `init` member function of `AppNode`. Let's define the node services as follows:

```
            esp_rmaker_ota_config_t ota_config;
            ota_config.server_cert =
                ESP_RMAKER_OTA_DEFAULT_SERVER_CERT;
            esp_rmaker_ota_enable(&ota_config, OTA_U
```

The first service is the OTA service. We enable the OTA service by calling the `esp_rmaker_ota_enable` function so that the device's firmware can be upgraded remotely. Then we enable the system service:

```
            esp_rmaker_system_serv_config_t system_s
                .flags = RMAKER_EVENT_REBOOT |
                     RMAKER_EVENT_WIFI_RESET,
                .reboot_seconds = 2,
                .reset_seconds = 2,
                .reset_reboot_seconds = 2};
            esp_rmaker_system_service_enable(&system
```

We configure the system service for remote reboot and Wi-Fi reset. This configuration enables us to use the mobile application to reset the device remotely. We finalize the `init` function with the following services:

```
            app_insights_enable();
            esp_rmaker_schedule_enable();
            esp_rmaker_scenes_enable();
        }
```

As we discussed in *Chapter 7, ESP32 Security Features for Production-Grade Devices*, the `app_insights_enable` function is for diagnostics. It sends the device metrics, such as errors, crashes, or Wi-Fi signal level, to RainMaker so that we can monitor the devices in the field remotely by using the RainMaker dashboard (https://dashboard.rainmaker.espressif.com/). The `esp_rmaker_schedule_enable` function makes it possible to create time-scheduled actions from the mobile application. For instance, we can define a specific date/time to turn the plug on or off. The scenes service is enabled by calling the `esp_rmaker_scenes_enable` function. With that, a user can define a scene and create a group of devices with their actions that run when the scene is activated with a single click on the mobile application. We will test and discuss all these features in detail when we complete the development.

The next member function of the `AppNode` class is `start`:

```
        virtual void start()
        {
            esp_rmaker_start();
        }
```

The `start` function simply calls the `esp_rmaker_start` function of the RainMaker library to enable the underlying task that is responsible for communication with the RainMaker cloud. It is virtual, so a derived class can change it according to its specific needs. Next comes another public member function:

```
        void addRmakerEventHandler(RmakerEventHandle
            void *arg)
        {
            if (fn != nullptr)
            {
                event_handlers.push_back({fn, arg});
            }
        }
```

The `addRmakerEventHandler` function collects the event handlers from client codes. For example, in the main function of the sensor application, we can call this function to pass a callback, which enables us to process RainMaker events outside the `AppNode` class. We finish the `AppNode` class definition with the following function:

```
        bool isConnected() const { return m_connecte
    }; // class end
```

The `isConnected` function is a getter function that returns the value of the `m_connected` member variable. We have completed the class definition but not implemented the body of the `eventHandler` function. It comes next:

```
    void AppNode::eventHandler(void *arg, esp_event_
        event_base, int32_t event_id, void *event_da
```

```
    {
        AppNode *obj = reinterpret_cast<AppNode *>(a
```

We registered the `eventHandler` function in the `init` function of the `AppNode` class to handle the RainMaker events. The `eventHandler` function will be called when a RainMaker event occurs. We capture the MQTT connected and disconnected events as follows:

```
        if (event_base == RMAKER_COMMON_EVENT)
        {
            switch (event_id)
            {
            case RMAKER_MQTT_EVENT_CONNECTED:
                ESP_LOGI(obj->m_init_params.device_n
                    "mqtt connected");
                obj->m_connected = true;
                break;
            case RMAKER_MQTT_EVENT_DISCONNECTED:
                obj->m_connected = false;
                break;
            default:
                break;
            }
        }
```

MQTT events come in the `RMAKER_COMMON_EVENT` group. We set the `m_connected` member variable of the `AppNode` instance according to the raised MQTT connected/disconnected events. In the last lines of

the function body, we run the event callback functions registered by the client codes:

```
        for (auto &h : obj->event_handlers)
        {
            h.fn(h.arg, event_base, event_id, event_
        }
    } // function end
} // namespace end
```

We run a `for` loop to traverse all callback functions that have been provided in the initialization phase of the class. We call them with the event information so that the client codes can take actions accordingly.

The `AppNode` class is the base class for any RainMaker node in our solution. We will inherit the `AppPlug` class from it in the next section, where we will develop the plug application.

Developing plug

The plug hardware uses ESP32-C3 DevkitM-1 and has a relay attached to the GPIO-19 pin of the devkit. Let's prepare the project as follows:

1. Create an ESP-IDF project:

```
$ source ~/esp/esp-idf/export.sh
Detecting the Python interpreter:
Checking "python" ...
```

```
Python 3.10.11
"python" has been detected
Adding ESP-IDF tools to PATH…
<more logs>
$ idf.py create-project plug
```

2. Copy the `sdkconfig.defaults`, `partitions.csv`, and `CMakeLists.txt` files from the project repository into the project directory.
3. The `CMakeLists.txt` file contains the *cmake* build configuration. Edit it to update the paths in it according to your directory structure on your development machine.
4. Rename the `main/plug.c` file to `main/plug.cpp`.
5. Update the `main/CMakeLists.txt` file content as follows:

```
idf_component_register(SRCS "plug.cpp" INCLUDE_DI
```

The ESP-IDF project is ready for development and we will code the plug application next.

Adding plug node

Let's add a new file, `main/AppPlug.hpp`, for the plug node implementation:

```
#pragma once
#include "AppNode.hpp"
#include "driver/gpio.h"
#include "AppEsp32C3Bsp.hpp"
```

We include the header files for the plug node implementation. The plug node will derive from the `AppNode` and `AppEsp32C3Bsp` classes, so we include their respective header files. We also need the `driver/gpio.h` header file of ESP-IDF to drive the GPIO pin of the relay. Then we continue with the class definition:

```
namespace app
{
    class AppPlug : public AppNode, private AppEsp32
    {
    private:
        esp_rmaker_param_t *m_power_param;
        bool m_state;
```

The `AppPlug` class inherits from `AppNode` and exposes its public functions to outside of the class. In the `private` section, we define two member variables. The `m_power_param` variable is the RainMaker parameter for the plug. The `m_state` variable shows the relay state whether it is on or off.

RainMaker defines a set of standard device types and their parameters. When we define a standard device from that list, it can show specific icons for the device on the mobile application and also map to the voice service types. You can see the entire list here: https://rainmaker.espressif.com/docs/standard-types.html.

Then we define the prototypes for the private member functions:

```
        void update(bool val);
        static void buttonHandler(void *arg);
        static void definePlug(esp_rmaker_device_t *
            void *arg);
        static esp_err_t requestHandler(const esp_rm
            *device, const esp_rmaker_param_t *param
            const esp_rmaker_param_val_t val, void *
            esp_rmaker_write_ctx_t *ctx);
```

The `update` function updates the internals when a request comes to toggle the relay state. The request can come from the physical button on the devkit or from the mobile application via the RainMaker cloud. The `buttonHandler` function handles the physical button presses as the name implies. The `definePlug` function is the one that we will pass as a parameter to the `AppNode` code constructor in order to define the plug as a RainMaker node. Lastly, the `requestHandler` function is the callback to handle the RainMaker requests on the device. Then we move on to the `public` section of the class:

```
    public:
        AppPlug() : AppNode({"Plug Node", "esp.node.
            "esp.device.plug", definePlug, this}), m
        {
        }
```

The `AppPlug` constructor calls the `AppNode` constructor with a set of parameters, as we defined earlier when we implemented the `AppNode` base class. As a recap, the first field shows the node name and the second one is the node type. The third and fourth fields are the device name and device type respectively. The device type is a standard type from the RainMaker list.

The fifth field of the parameter is the `definePlug` callback function. We also pass the `this` pointer of the `AppPlug` instance. The initial state of the relay is off. Next comes the `init` function of the class:

```
        void init() override
        {
            gpio_config_t config{GPIO_SEL_19,
                                 GPIO_MODE_OUTPUT,
                                 GPIO_PULLUP_DISABLE
                                 GPIO_PULLDOWN_DISAB
                                 GPIO_INTR_DISABLE};
            gpio_config(&config);
```

We start with the GPIO initialization in the `init` function. We will drive the relay via the GPIO-19 pin of the devkit, so we configure the pin in output mode. Then we add other initialization codes:

```
            AppEsp32C3Bsp::init();
            addButtonHandler(button_cb_type_t::BUTTO
                buttonHandler, this);
            AppNode::init();
```

```
        } // end of function
    }; // end of class
```

We call the `init` function of the `AppEsp32C3Bsp` base class. It will initialize the button and LED of the devkit. The `addButtonHandler` function comes with the AppEsp`32C3Bsp` base class, and we call it to register the `buttonHandler` function of `AppPlug`. Lastly, we call the `AppNode::init` function, and this call makes the `AppPlug` class ready for the RainMaker communication. The class definition is complete, but we still need to implement the class private function bodies as comes next:

```
    void AppPlug::update(bool val)
    {
        m_state = val;
        gpio_set_level(gpio_num_t::GPIO_NUM_19, val
        esp_rmaker_param_update_and_report(m_power_p
            esp_rmaker_bool(val));
        setLed({val, 360, 50, 50});
    }
```

The `update` function is a short one, but in fact it implements the heart of the plug functionality. We set the `m_state` member variable to keep track of the plug state and then call the `gpio_set_level` function to actually set the relay pin. We report this state change to RainMaker by calling the `esp_rmaker_param_update_and_report` function. As immediate feedback to the user, we also set the LED state of the devkit. The next function is the button handler:

```
void AppPlug::buttonHandler(void *arg)
{
    AppPlug *obj = reinterpret_cast<AppPlug *>(a
    obj->update(!obj->m_state);
}
```

In the `buttonHandler` function, we simply call the `update` function of the instance with the inverted value of the current relay state so it is turned off if it is currently on and vice versa. We implement the `requestHandler` function as follows:

```
esp_err_t AppPlug::requestHandler(const esp_rmak
    *device, const esp_rmaker_param_t *param,
    const esp_rmaker_param_val_t val, void *priv
    esp_rmaker_write_ctx_t *ctx)
{
    AppPlug *obj = reinterpret_cast<AppPlug *>(p
    obj->update(val.val.b);
    return ESP_OK;
}
```

The `requestHandler` function is the callback for RainMaker requests. As we can see in the argument list, it takes a pointer to the RainMaker device, a pointer to the RainMaker parameter, and the value of the RainMaker parameter. Since we have only a single device and a single parameter in the `AppPlug` class, we don't need to check the pointers to find the RainMaker parameter that owns the incoming value. It has to be the new state of the relay. Therefore, we directly

call the `update` function with the new state. The last function of the class is `definePlug`:

```
void AppPlug::definePlug(esp_rmaker_device_t *de
    void *arg)
{
    AppPlug *obj = reinterpret_cast<AppPlug *>(a
    obj->m_power_param = esp_rmaker_power_param_
        false);
```

As we discussed earlier, the `definePlug` function implements the device-specific features. As noted in the RainMaker documentation, it has to have a **power** parameter. We create this parameter by calling the `esp_rmaker_power_param_create` function. Next, we associate it with the RainMaker device:

```
    esp_rmaker_device_add_param(device, obj->m_p
    esp_rmaker_device_assign_primary_param(devic
        obj->m_power_param);
```

We call the `esp_rmaker_device_add_param` function to add the power parameter into the device and assign it as the primary parameter of the device. Assigning it as `primary` makes it the default parameter of the device on the mobile application so that the interaction with the device on the GUI directly refers to this parameter. We will see how it works when we test the application. Finally, we register the `requestHandler` function with the device:

```
        esp_rmaker_device_add_cb(device, requestHand
    } // end of function
} // end of namespace
```

We defined the `requestHandler` function earlier, but we also need to specify it as the handler for the RainMaker write requests. We call the `esp_rmaker_device_add_cb` function for this. This finishes the implementation of the `AppPlug` class. We will code the plug application next.

Coding application

We have the `AppPlug` class, which implements a RainMaker node. We will use it in the plug application source code, `main/plug.cpp`:

```
#include "AppDriver.hpp"
#include "AppPlug.hpp"
namespace
{
    app::AppDriver app_driver;
    app::AppPlug app_plug;
}
```

After including the header files, we define two objects, one for the `AppDriver` class, and one for the `AppPlug` class that we have just developed. Next, we will use them in the `app_main` function of the application:

```
extern "C" void app_main(void)
{
    app_driver.init();
    app_plug.init();
    app_plug.start();
    app_driver.start();
}
```

The `app_main` function body is quite simple. We initialize the objects and call the `start` functions of the objects to enable them to operate.

Let's build the application to see if it compiles successfully:

```
$ idf.py build
Executing action: all (aliases: build)
<more logs>
Project build complete. To flash, run this command:
<removed>
or run 'idf.py -p (PORT) flash'
```

Now that the plug application is ready, let's develop the multisensor application.

Developing multisensor

Our target hardware is ESP32-S3 Box Lite for the multisensor device. The process of creating the ESP-IDF project is similar to what we did for the plug:

1. Create an ESP-IDF project:

```
$ source ~/esp/esp-idf/export.sh
Detecting the Python interpreter
Checking "python" ...
Python 3.10.11
"python" has been detected
Adding ESP-IDF tools to PATH…
<more logs>
$ idf.py create-project sensor
```

2. Copy the `sdkconfig.defaults`, `partitions.csv`, and `CMakeLists.txt` files from the project repository into the project directory.
3. The `CMakeLists.txt` file contains the *cmake* build configuration. Edit it to update the paths in it according to your directory structure on your development machine.
4. Rename the `main/sensor.c` file to `main/sensor.cpp`.
5. There is a GUI design to show the sensor values on the devkit display. These values are implemented in four files: `main/ui.c`, `main/ui.h`, `main/ui_helpers.c`, `main/ui_helpers.h`. Copy them from the book repository into the project `main` directory, too.
6. Update the `main/CMakeLists.txt` file content as follows:

```
idf_component_register(SRCS "sensor.cpp" "ui.c"
    "ui_helpers.c" INCLUDE_DIRS "." )
```

Now that we have the project created, we can move on to coding.

Adding sensor node

We can start coding by adding a header file, `main/AppCommon.hpp`, for the `type` definition common to the application components:

```
#pragma once
#include <cstdint>
namespace app
{
    typedef struct
    {
        uint8_t light_intensity;
        float temperature;
    } SensorReading_t;
}
```

The `SensorReading_t` type defines the fields for the light intensity and temperature values that come from the sensors connected to the devkit. This type is available to the classes in the project to share data between them.

Next, we add another file, `main/AppSensorNode.hpp`, for the sensor node implementation:

```
#pragma once
#include <cstring>
#include <cstdint>
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "sdkconfig.h"
```

```
#include "tsl2561.h"
#include "bmp280.h"
#include "AppNode.hpp"
#include "AppCommon.hpp"
```

We include some standard header files, then FreeRTOS headers for task management, and the C headers to drive the sensors. We also include `AppNode.hpp` for the `AppNode` base class. Then comes a simple macro:

```
#define LIGHT_INTENSITY(x) (uint8_t)((x) > 2048 ? 10
    (int)((x)*100 / 2048))
```

The TSL2561 ambient light sensor has a 16-bit resolution, but the RainMaker parameter for light measurements accepts values between 0 and 100, thus we map the sensor readings into the 0-100 range with this macro. After that, we define the `AppSensorNode` class as follows:

```
namespace app
{
    using SensorReadingCb_f = std::function<void(
        const SensorReading_t &)>;
    class AppSensorNode : public AppNode
    {
    private:
        SensorReadingCb_f m_reading_cb;
        tsl2561_t m_light_sensor;
        bmp280_t m_temp_sensor;
```

The `AppSensorNode` class derives from `AppNode`, as we discussed in the *Solution architecture* section. In the `private` section of the class, we define a callback function object, `m_reading_cb`, to be run with each sensor reading. We also define the sensor devices as the class members. Next comes the RainMaker parameters specific to `AppSensorNode`:

```
        esp_rmaker_param_t *m_light_rmaker_param;
        esp_rmaker_param_t *m_temp_rmaker_param;
```

We have two RainMaker parameters as expected, one for light readings and one for temperature. We will use them to pass sensor readings to the RainMaker cloud. Lastly, we define two class functions in the `private` section:

```
        static void defineSensor(esp_rmaker_device_t
            void *arg);
        static void readSensor(void *arg);
```

The `defineSensor` function will be passed to the `AppNode` base class when creating the RainMaker node as we did for the plug. The `readSensor` is a FreeRTOS task function to read from the connected sensors periodically. Nothing is left in the `private` section, so we can move on to the `public` section of the `AppSensorNode` class:

```
    public:
        AppSensorNode() : AppNode({"Sensor Node",
            "esp.node.sensor", "Sensor", "esp.device
```

```
            defineSensor, this}), m_reading_cb(nullpt
        {
        }
```

In the `public` section, we first define the `AppSensorNode` constructor. It passes the node and device information to the `AppNode` base class, as well as the `defineSensor` member function, to add the `AppSensorNode` RainMaker parameters in the RainMaker device definition. Then we will override the `init` function as follows:

```
        void init() override
        {
            static bmp280_params_t bme280_sensor_par
            memset(&m_light_sensor, 0, sizeof(tsl256
            ESP_ERROR_CHECK(i2cdev_init());

            ESP_ERROR_CHECK(tsl2561_init_desc(&m_lig
                TSL2561_I2C_ADDR_FLOAT, I2C_NUM_1,
                gpio_num_t::GPIO_NUM_41, gpio_num_t:
            ESP_ERROR_CHECK(tsl2561_init(&m_light_se
            ESP_ERROR_CHECK(bmp280_init_default_para
                &bme280_sensor_params));
            ESP_ERROR_CHECK(bmp280_init_desc(&m_temp
                BMP280_I2C_ADDRESS_0, I2C_NUM_1,
                gpio_num_t::GPIO_NUM_41, gpio_num_t:
            ESP_ERROR_CHECK(bmp280_init(&m_temp_sens
                &bme280_sensor_params));
```

In the first part of the `init` function, we initialize the I^2C bus and the sensors that are connected to this bus. If the physical connections are correct, they are ready to send ambient measurements from the immediate environment. Next, we initialize the underlying base class, `AppNode`:

```
        AppNode::init();
    } // end of function
```

The call to the `AppNode::init` function will create the RainMaker node and make the class ready for the RainMaker communication. The next function in the `public` section in the implementation is `start`, as follows:

```
    void start() override
    {
        AppNode::start();
        xTaskCreate(readSensor, "sensor", 4096,
            nullptr);
    }
```

The `start` function overrides the base class definition. After the initialization of the underlying `AppNode` base class, we create a FreeRTOS task to read from the sensors by passing the `readSensor` function to `xTaskCreate`. Then we define the last function of the `AppSensorNode` class:

```
        void setReadingCb(SensorReadingCb_f cb)
        {
            m_reading_cb = cb;
        } // end of function
    }; // end of class
```

The `setReadingCb` function just sets the `m_reading_cb` member variable. The provided callback will be used with each new reading from the multisensor to pass the reading to client codes. After the class definition, we can develop the private functions of the class:

```
    void AppSensorNode::defineSensor(esp_rmaker_devi
        void *arg)
    {
        AppSensorNode *obj = reinterpret_cast<
            AppSensorNode *>(arg);
        obj->m_light_rmaker_param =
            esp_rmaker_intensity_param_create("light
        esp_rmaker_device_add_param(device,
            obj->m_light_rmaker_param);
        esp_rmaker_device_assign_primary_param(devic
            obj→m_light_rmaker_param);
```

In the `defineSensor` function, we add the RainMaker parameters. The first one is `m_light_rmaker_param`. We first create it, then associate it with the RainMaker device. We also set it as the primary parameter of the device so that the mobile application will show its value on the device UI element. Then we add the temperature parameter as follows:

```
        obj->m_temp_rmaker_param =
            esp_rmaker_temperature_param_create("tem
        esp_rmaker_device_add_param(device,
            obj->m_temp_rmaker_param);
    } // end of function
```

The logic for the `m_temp_rmaker_param` variable is the same. We create it and then associate it with the RainMaker device. The next private function implementation is `readSensor`:

```
    void AppSensorNode::readSensor(void *arg)
    {
        AppSensorNode *obj = reinterpret_cast<
            AppSensorNode *>(arg);
        float pressure;
        float temperature;
        float humidity;
        uint32_t lux;
```

The `readSensor` function is a FreeRTOS task function that reads from BME280 and TSL2561 periodically. We define the local variables first. Then, we will develop the task loop as follows:

```
        while (true)
        {
            vTaskDelay(pdMS_TO_TICKS(10000));
            bmp280_read_float(&obj->m_temp_sensor, &
                &pressure, &humidity);
            tsl2561_read_lux(&obj->m_light_sensor, &
```

```
            SensorReading_t reading{LIGHT_INTENSITY(
                temperature};
```

The task period is 10 seconds. After 10 seconds have passed, we read values from both sensors by calling their respective functions, `bmp280_read_float` and `tsl2561_read_lux`, and create the `reading` variable from the sensor measurements. The next thing to do is to pass the values to the RainMaker cloud:

```
            if (obj->m_connected)
            {
                esp_rmaker_param_update_and_report(
                    obj->m_temp_rmaker_param,
                    esp_rmaker_float(reading.tempera
                esp_rmaker_param_update_and_report(
                    obj->m_light_rmaker_param,
                    esp_rmaker_int(reading.light_int
            }
```

If the node is connected to the RainMaker MQTT broker, we call the `esp_rmaker_param_update_and_report` function of the RainMaker library in order to pass the values to RainMaker. We also call the reading callback to share the reading with the client codes as follows:

```
            if (obj->m_reading_cb != nullptr)
            {
                obj->m_reading_cb(reading);
            } // end of if
```

```
        } // end of loop
    } // end of function
} // end of namespace
```

If the `m_reading_cb` member variable is set, we call it with `reading` as its argument. With this, we have finished the implementation of the `AppSensorNode` class.

Adding a GUI

The next task is to implement the GUI on the devkit display. However, before moving on to coding, let's see what we will have as the GUI in the following figure:

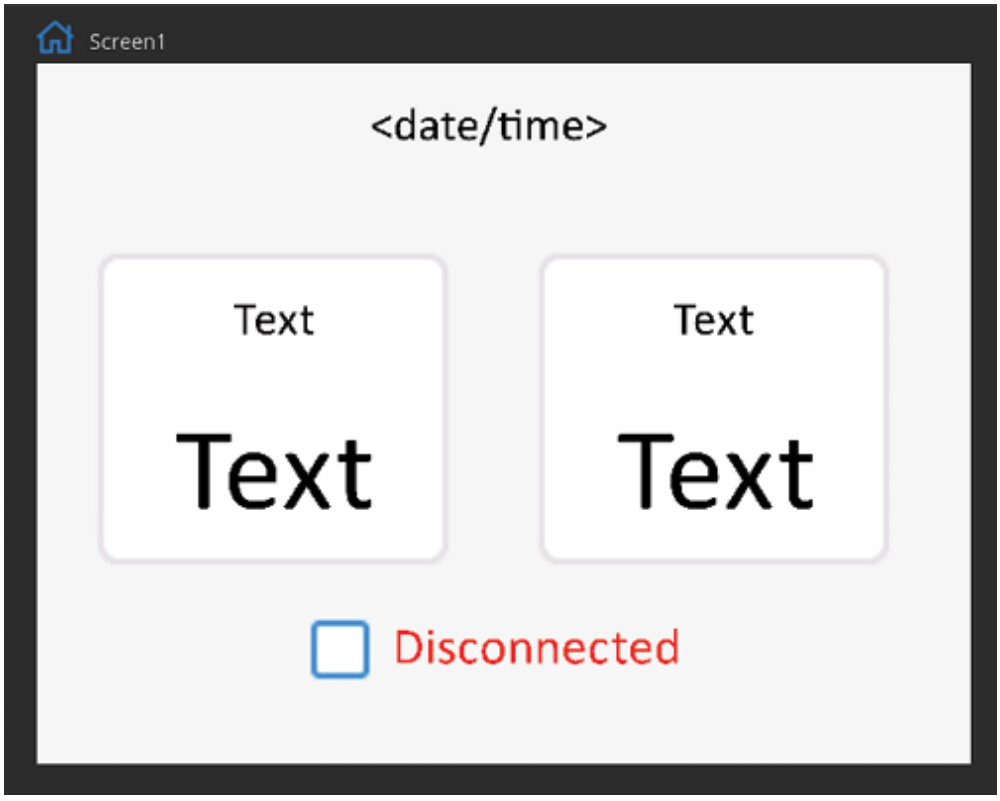


Figure 9.6: Multisensor GUI design in SquareLine Studio

This figure is a screenshot from SquareLine Studio. At the top, there is a label element that shows the current date/time. The two panels below it are for the light and temperature values. The checkbox at the bottom indicates the MQTT connection status. We will have only this screen in the application.

Now, we can add a new C++ header file, `main/AppUi.hpp`, for implementing the GUI:

```
#pragma once
#include <mutex>
#include <vector>
#include <ctime>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/timers.h"
#include "freertos/queue.h"
#include "esp_log.h"
#include "bsp_board.h"
#include "bsp_lcd.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
#include "ui.h"
#include "AppCommon.hpp"
```

We add three groups of header files. In the first group, there are standard headers that we need in this implementation. The second group includes the FreeRTOS headers. The third group is for driving the display and application-level functionality. The `ui.h` file is generated by SquareLine Studio, the GUI designer for LVGL.

Although we can definitely develop the GUI manually, this tool makes the job much easier and leaves us with spare time that we can spend on the application. Then, we begin with the class definition as follows:

```
namespace app
{
    class AppUi
    {
    private:
        static std::mutex m_ui_access;
        QueueHandle_t m_sensor_reading_queue;
```

In the `private` section of the `AppUi` class definition, we have two member variables. The `m_ui_access` variable is the mutex that controls the access to the LVGL elements. We don't want to change LVGL's internal state by calling its functions while the LVGL task is updating the active screen on the display. Using a mutex helps us synchronize access. The `m_sensor_reading_queue` variable is a FreeRTOS queue that we employ to pass sensor reading data from the sensor task that we developed in the `AppSensorNode` class. In this class, we will receive the sensor readings via this queue. Next, we add the prototypes of the private member functions:

```
        static void lvglTask(void *arg);
        void setSensorValues(const SensorReading_t &
        static void updateDatetime(void *arg);
```

The `lvglTask` function is the FreeRTOS task function to update the display by using LVGL. The `setSensorValues` function will update the UI elements for a given sensor reading. We also have a date/time label on the GUI, and `updateDatetime` is the function that refreshes the date/time text on the screen every second. We don't have any other declaration in the `private` section and we can continue with the `public` section of the class:

```
    public:
        void init(void)
        {
            m_sensor_reading_queue = xQueueCreate(1,
                sizeof(SensorReading_t));
```

The `init` function prepares the `AppUi` class instance for execution. We first create the queue to receive sensor readings. It has only one slot and keeps only one reading at a time. This is enough for our purposes because the GUI refresh frequency is much higher than the reading generation frequency, and therefore there will never be a need for a second slot in the queue. Next, we add more initialization code:

```
            bsp_board_init();
            lv_port_init();
            ui_init();
            xTaskCreatePinnedToCore(lvglTask, "lvgl"
```

```
                this, 3, nullptr, 0);
            bsp_lcd_set_backlight(true);
```

To prepare for the GUI operations, we first call the `bsp_board_init` function, which initializes the display driver inside. Then we call the `lv_port_init` function for the LVGL initialization. The `ui_init` function creates the UI elements after the LVGL initialization. When we create the LVGL task with the `lvglTask` function and turn on the backlight, the UI infrastructure becomes ready for operation.

As a side note, the functions that start with `lv_` come from the LVGL library, and functions starting with `ui_` are generated by SquareLine Studio.

The last initialization in this function is the time zone:

```
            setenv("TZ", "GMT", 1);
            tzset();
        }
```

We set the `TZ` environment variable of the execution context to `GMT` and update the time zone with this value by calling the `tzset` function. The next public function of the class is `updateSensorReading`:

```
        void updateSensorReading(const SensorReading
        {
```

```
            xQueueSendToBack(m_sensor_reading_queue,
        }
```

In the `updateSensorReading` function, we simply push the incoming data into the queue. This is how the queue is fed by new sensor readings from external code. The next function, `updateMqttState`, updates the checkbox on the GUI when the MQTT connection status changes:

```
        void updateMqttState(bool connected)
        {
            std::lock_guard<std::mutex> lock(m_ui_ac
            if (connected)
            {
                lv_obj_add_state(ui_chkConnected,
                    LV_STATE_CHECKED);
                lv_checkbox_set_text(ui_chkConnected
                    "Connected");
            }
            else
            {
                lv_checkbox_set_text(ui_chkConnected
                    "Disconnected");
                lv_obj_clear_state(ui_chkConnected,
                    LV_STATE_CHECKED);
            }
        }
```

Since it is an update to a GUI element, we start the `updateMqttState` function by acquiring the `m_ui_access` mutex. After that, we update the text and state of the checkbox according to the MQTT connection status.

We implement the last public function of the class, `start`, as follows:

```
        void start(void)
        {
            static TimerHandle_t timer{nullptr};
            if (timer == nullptr)
            {
                timer = xTimerCreate("datetime",
                    pdMS_TO_TICKS(1000), pdTRUE, nul
                    updateDatetime);
                xTimerStart(timer, 0);
            }
        } // end of function
    }; // end of class
```

The `start` function defines a software timer that calls the `updateDatetime` function every second so that the date/time label on the GUI is updated with the current time. The class definition is finished, and we will develop the functions that we declared in the `private` section next:

```
    std::mutex AppUi::m_ui_access;
    void AppUi::setSensorValues(const SensorReading_
    {
        lv_label_set_text(ui_txtLight,
```

```
            std::to_string(r.light_intensity).c_str(
        lv_label_set_text(ui_txtTemp,
            std::to_string(r.temperature).substr(0,
    }
```

The purpose of the `setSensorValues` function is to update the UI with the latest sensor readings. We call the `lv_label_set_text` function of the LVGL to update the light and temperature elements of the GUI. It will be called in the LVGL task if there is a new reading. It doesn't use the mutex for the UI access because the LVGL task will do this. The next function definition is `updateDatetime`:

```
    void AppUi::updateDatetime(void *arg)
    {
        std::lock_guard<std::mutex> lock(m_ui_access
        time_t now;
        char strftime_buf[64]{0};
        struct tm timeinfo;
        time(&now);
        localtime_r(&now, &timeinfo);
        strftime(strftime_buf, sizeof(strftime_buf),
            &timeinfo);
        lv_label_set_text(ui_txtTime, strftime_buf);
    }
```

The `updateDatetime` function needs to use the mutex to guard the LVGL access because it is called from the software timer context that was created in the `start` function earlier. After getting the system

time, we convert it to the local time and format the current time value into a buffer. We use the content of this buffer to update the date/time UI element, `ui_txtTime`, by calling the `lv_label_set_text` function. The last function in the UI implementation is `lvglTask`:

```
void AppUi::lvglTask(void *arg)
{
    AppUi *obj = reinterpret_cast<AppUi *>(arg);
    while (true)
    {
        {
            std::lock_guard<std::mutex> lock(m_u
            SensorReading_t r;
            if (xQueueReceive(obj->m_sensor_read
                0) == pdTRUE)
            {
                obj->setSensorValues(r);
            }
            lv_task_handler();
        } // end of mutex scope
```

The `lvglTask` function is the FreeRTOS task function that renders the GUI. In the infinite loop of the task, we create an inner scope in which we use the mutex to protect the LVGL operations. After locking the mutex, we check the sensor reading queue to see if any new data is pushed. If so, we update the UI with the new values. Then we call the `lv_task_handler` function to render the latest updates on the GUI. When the inner scope with the mutex ends, the

mutex becomes available for other LVGL calls. We finalize the function as follows:

```
            vTaskDelay(pdMS_TO_TICKS(10));
        } // end of loop
    } // end of function
} // end of namespace
```

The 10-ms delay in the task loop allows other FreeRTOS tasks to run other GUI updates.

Coding the application

Now that the `AppUi` class has been implemented, we can integrate all the pieces in the main application that we will develop in the `main/sensor.cpp` file:

```
#include "AppDriver.hpp"
#include "AppSensorNode.hpp"
#include "AppUi.hpp"
namespace
{
    app::AppDriver app_driver;
    app::AppSensorNode app_sensor_node;
    app::AppUi app_ui;
}
```

We include the header files for the application objects first. Then we define the objects in an anonymous namespace to be used in the

application. Next, we will add the application's entry point, the `app_main` function:

```
extern "C" void app_main(void)
{
    app_driver.init();
    app_ui.init();
    app_sensor_node.init();
```

The first thing to do in the `app_main` function is to initialize all the objects that we have defined in the anonymous namespace. The order here is important. The `app_driver` object initializes the NVS and the Wi-Fi peripheral. The `app_ui` initialization activates the display drivers and LVGL. Then we initialize the sensor hardware by calling `app_sensor_node.init`. Then we start to link them together as follows:

```
    auto sensor_reading_handler = [](const app::Sens
        -> void
    {
        app_ui.updateSensorReading(r);
    };
    app_sensor_node.setReadingCb(sensor_reading_hand
```

We define a lambda function, `sensor_reading_handler`, to link the `app_ui` and `app_sensor_node` objects. It is the callback for `app_sensor_node` when a new reading is available. Inside the `sensor_reading_handler` lambda function, we call

`app_ui.updateSensorReading` to pass the sensor reading to be displayed on the devkit screen. Next is another lambda function for the MQTT events:

```
    auto mqtt_state_handler = [](void *arg, esp_even
        event_base, int32_t event_id, void *event_da
    {
        if (event_base != RMAKER_COMMON_EVENT)
        {
            return;
        }
```

We will use the `mqtt_state_handler` lambda function to update the GUI with the MQTT connection information. In the function, we first check whether the event base is `RMAKER_COMMON_EVENT` since the RainMaker MQTT events come in this group. After that, we add a `switch` statement to handle the MQTT events:

```
        switch (event_id)
        {
        case RMAKER_MQTT_EVENT_CONNECTED:
            app_ui.updateMqttState(true);
            break;
        case RMAKER_MQTT_EVENT_DISCONNECTED:
            app_ui.updateMqttState(false);
            break;
        default:
            break;
        }
```

```
        };
        app_sensor_node.addRmakerEventHandler(mqtt_state
            nullptr);
```

In the `switch` statement, we call the `app_ui.updateMqttState` function for the MQTT connected and disconnected events. The `updateMqttState` function updates the checkbox on the GUI, as we implemented earlier in the `AppUi` class. We pass the `mqtt_state_handler` lambda function to the `app_sensor_node` object to be called when a connection status change occurs. We have linked the objects via the lambda functions, and they are ready to operate together now:

```
        app_ui.start();
        app_sensor_node.start();
        app_driver.start();
    } // end of app_main
```

We call the `start` functions of the application objects and that is it! We will test the project by flashing the applications on both devkits in the next section.

Testing project

We have two devices in the solution. Let's begin with the plug. We will flash the application on the devkit, then provision it to ESP RainMaker and see how the plug application behaves.

Testing plug

We can test the plug application as follows:

1. Make sure the plug hardware is ready with ESP32-C3 DevkitM-1 and the relay module connected to it.
2. Go to the plug ESP-IDF project directory and flash the application on the devkit:

```
$ idf.py erase-flash flash monitor
Executing action: erase-flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<more logs>
I (9447) app_wifi: Scan this QR code from the ESP
<QR-code is here>
```

3. Start the RainMaker application on your mobile device and add the plug. The application runs a wizard for provisioning. When the provisioning is completed, tap on the **Done** button and return to the main screen.

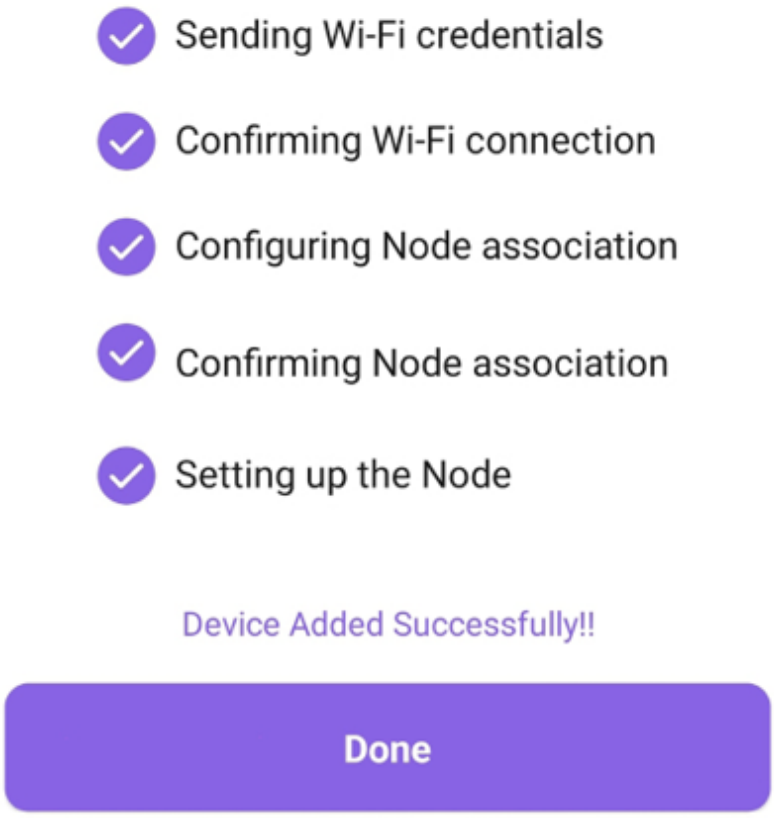


Figure 9.7: Adding the plug to the RainMaker mobile application

4. When the plug is added, it will appear on the main screen. Tap on the **On/Off** icon in the top left of the plug icon to test whether the relay state changes. The LED on the devkit will also change its state accordingly.

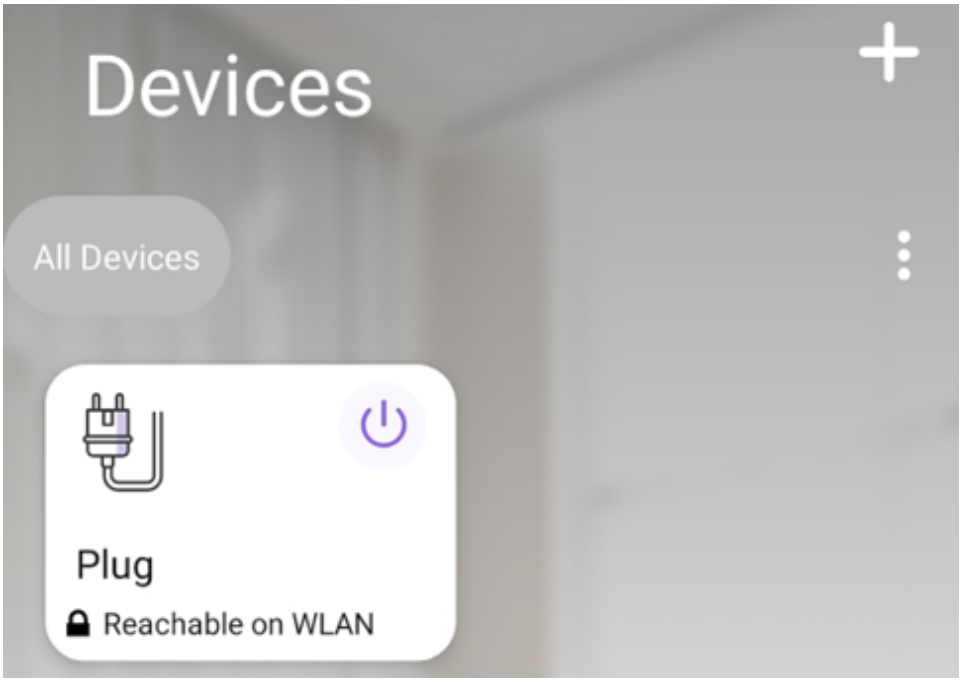


Figure 9.8: The plug on the main screen of the RainMaker application

5. Press the button on the devkit to change the relay state. We can see that the plug's state has been updated on the mobile application:

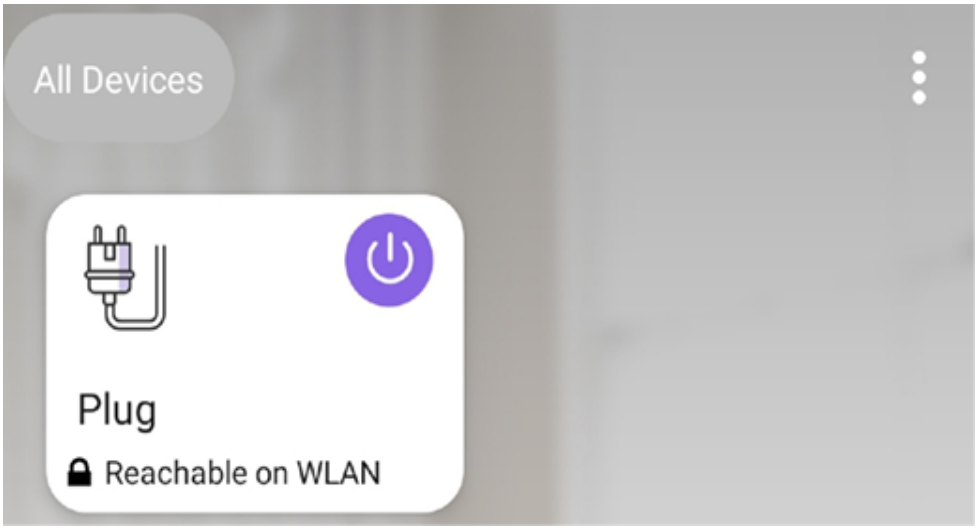


Figure 9.9: The plug is on in the RainMaker application

Nice! We can move on to flashing and testing the multisensor application.

Testing the multisensor application

As with the plug testing, we can follow the steps below to test the multisensor application:

1. Make sure the multisensor hardware is ready, with ESP32-S3 Box Lite and the BME280 and TSL2561 sensors connected to it.
2. Go to the multisensor ESP-IDF project directory and flash the application on the devkit:

```
$ idf.py erase-flash flash monitor -p /dev/ttyACM
<logs removed>
I (9345) app_wifi: Scan this QR code from the ESP
<QR-code is here>
```

3. Tap on the **+** icon on the main screen of the mobile application to add a new device and follow the instructions.

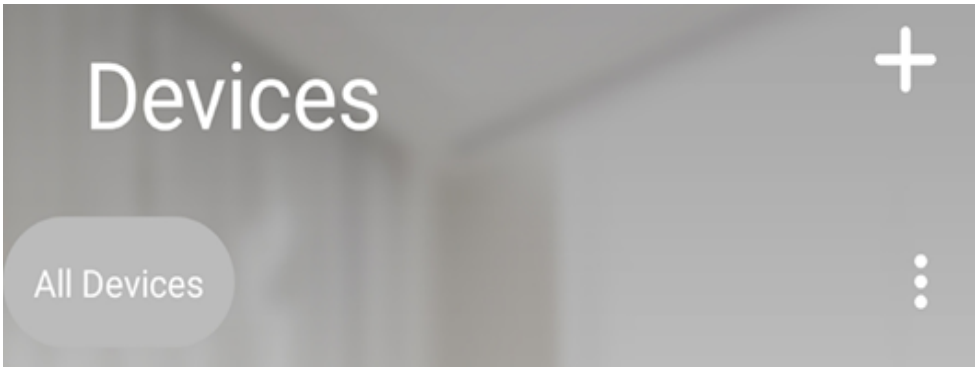


Figure 9.10: Adding more devices to the RainMaker mobile app

4. When the multisensor application has been added, its icon will appear on the main screen next to the plug icon:

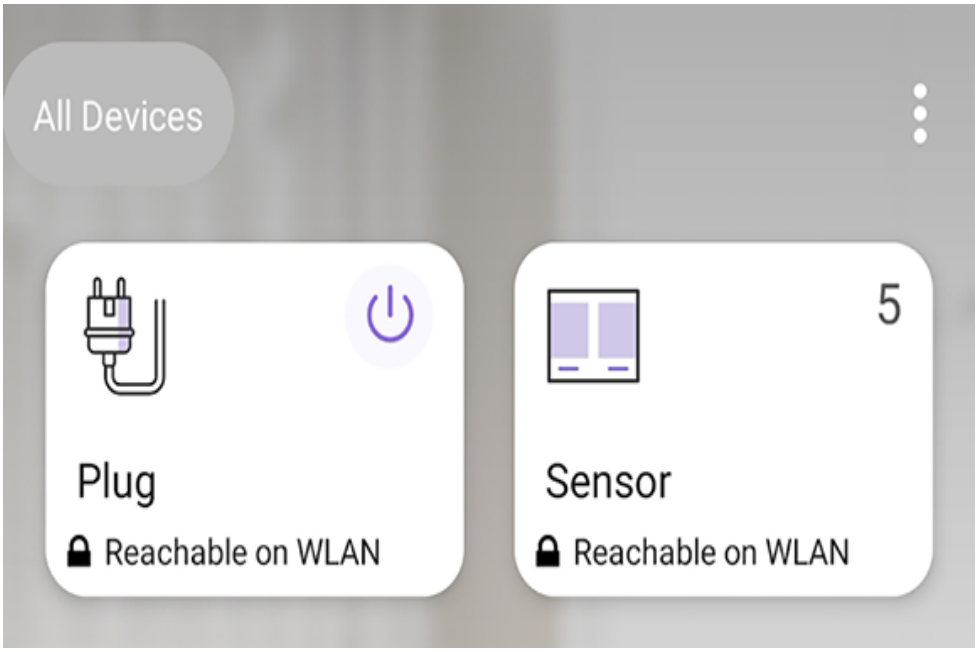


Figure 9.11: Multisensor app added to the RainMaker mobile app

5. We can see the light intensity value in the top right of the multisensor icon because it is the default parameter of the device. Turn on the torch of your mobile device to increase the light level on TSL2561 and observe that both the mobile application and the multisensor display are updated with the new light level.

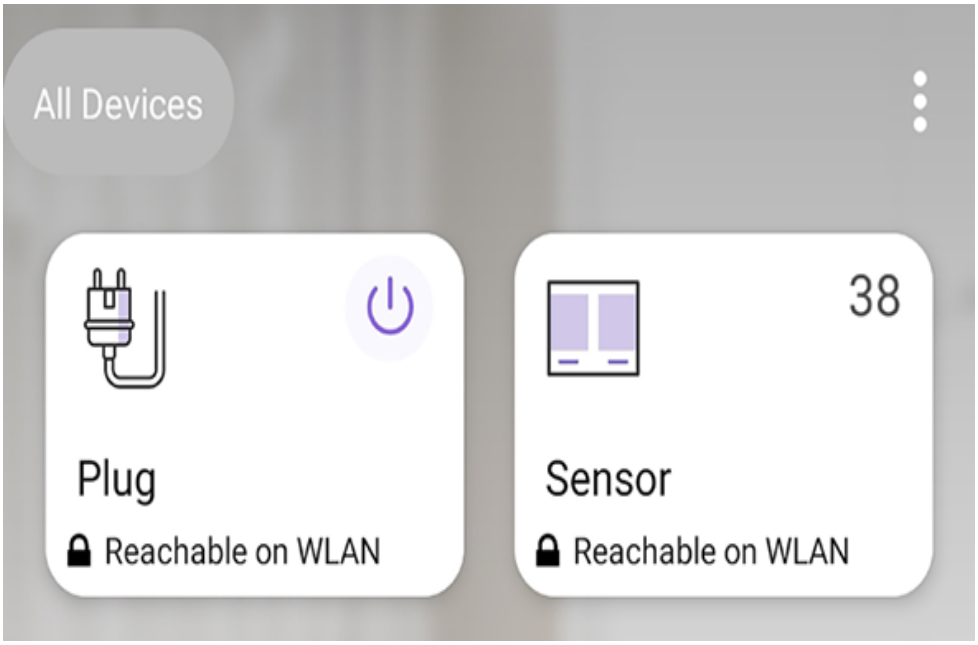


Figure 9.12: Light intensity value increased on the sensor in the RainMaker app

We have both devices up and running. Let's discover other features of the RainMaker mobile application.

Using smart home features

A smart home product usually has more features than simple device controls. We should be able to define automated or scheduled actions, as well as integration with voice services. The RainMaker mobile application provides all these features. In the following steps, we will test them and see how we can use our devices in such scenarios:

1. With schedules, we can set a time to change the plug's state. At the bottom of the main screen, there is a list of icons. Tap on the **Schedules** icon and then the **Add Schedule** button.

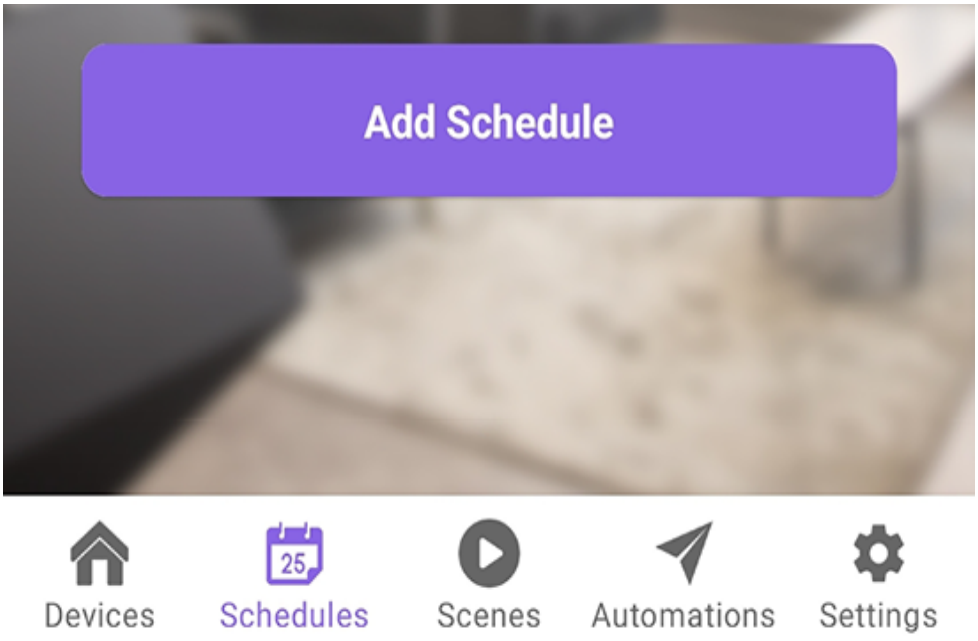


Figure 9.13: Adding a new schedule to the RainMaker app

2. Give the schedule a name, set its time 2 minutes later than the current time, and turn the plug **ON** from the **Actions** list.

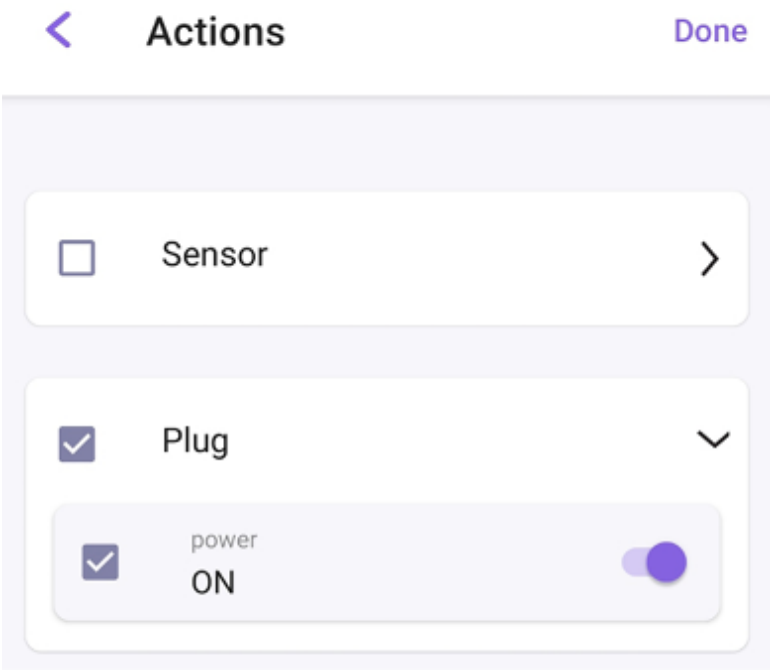


Figure 9.14: Setting an action in the RainMaker app

3. After saving the schedule, it will appear on the schedules list. Wait for 2 minutes to see that the plug's state turns on automatically.

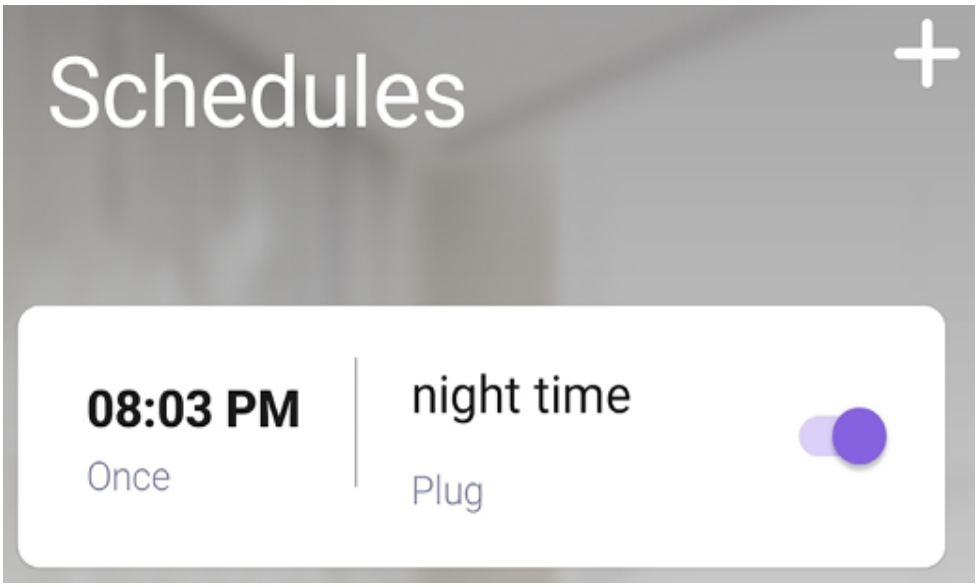


Figure 9.15: A schedule in the RainMaker app

4. The RainMaker platform provides another feature, **scenes**. Scenes helps us to group devices to change their states at the same time with a single tap. For example, in a hypothetical scenario where we have a smart blind, a smart light bulb, and a connected music player, we could group them into a scene to close the blind, set the light bulb level to 50%, and start the music to create a relaxing environment. When we activate the scene, they change their states as specified in the scene definition. In our setup, we don't have those devices, just a plug. Tap on the **Scenes** button at the bottom then the **Add Scene** button to start the wizard.

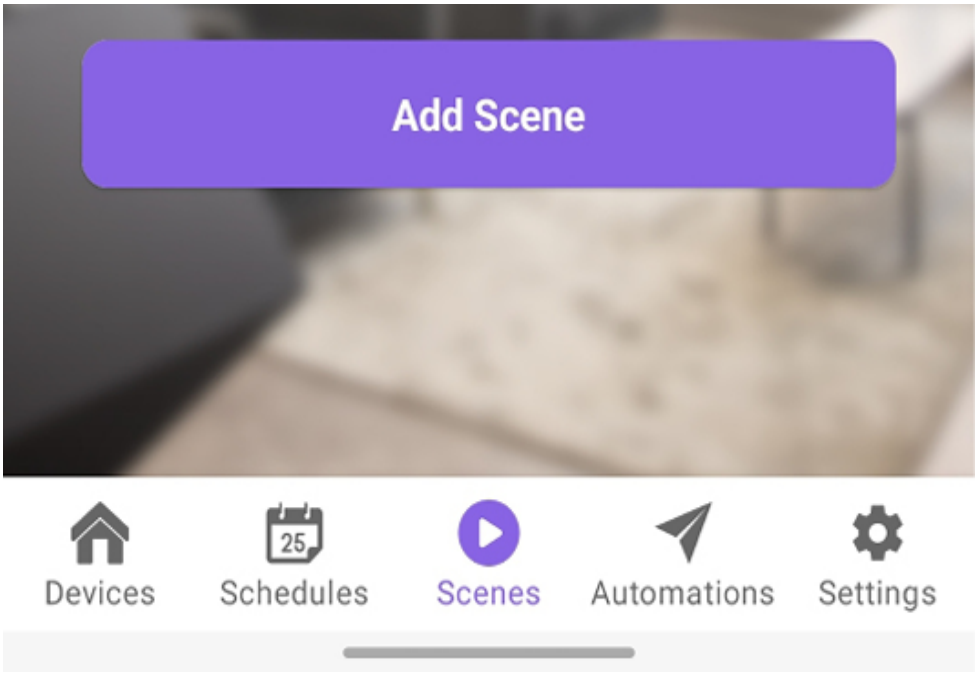


Figure 9.16: Adding a new scene to the RainMaker app

5. After completing the wizard, we have a new scene. Activate the scene to see it in action.

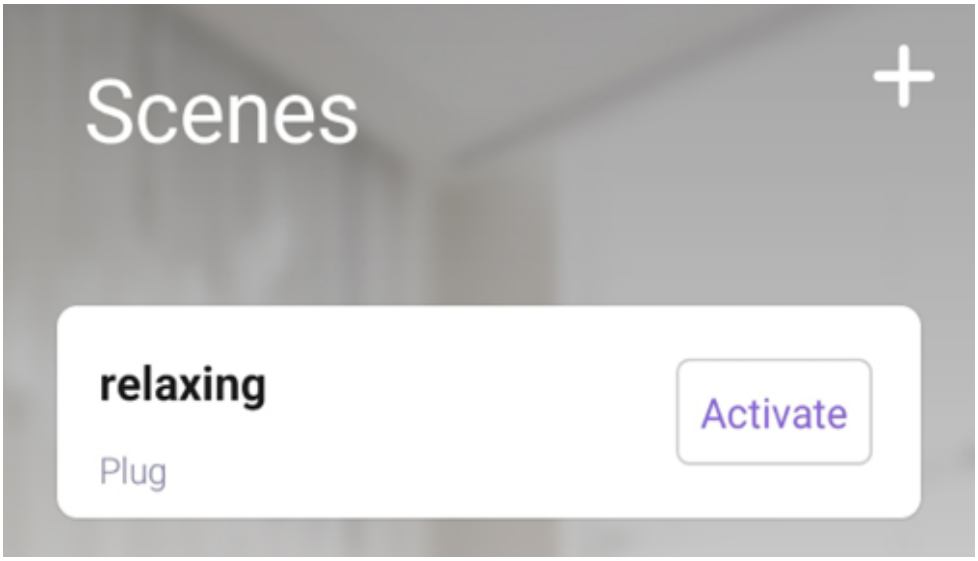


Figure 9.17: Activating a scene in the RainMaker app

6. Another feature of the RainMaker platform is automation. With automation, we define a condition to trigger an action. To create

an automation, tap on the **Automations** button from the bottom list and then the **Add Automation** button.

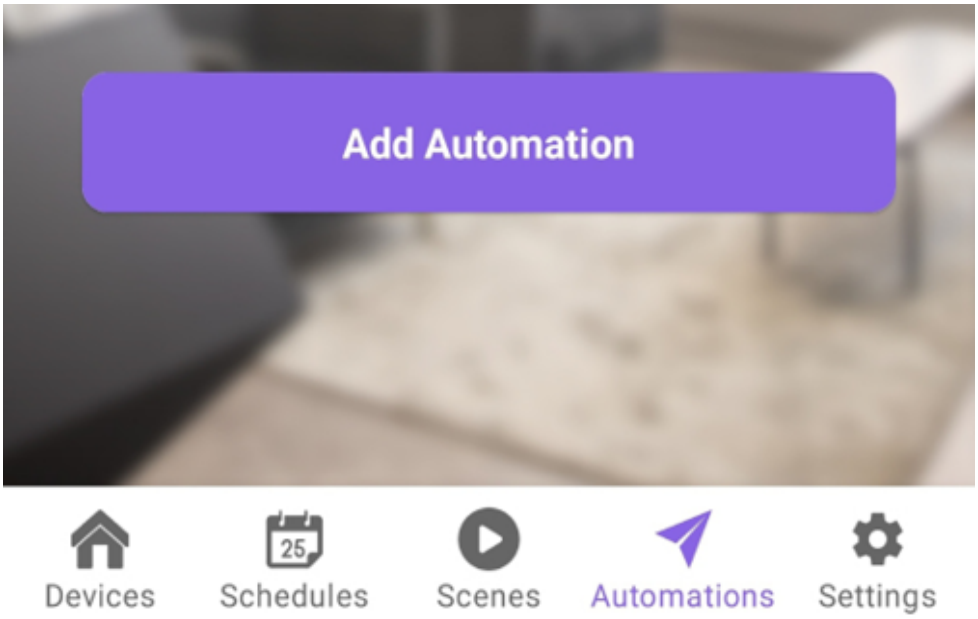


Figure 9.18: Creating an automation in the RainMaker app

7. After giving it a name, select **Sensor** and set the light intensity to a value that's higher than its current value as the event.

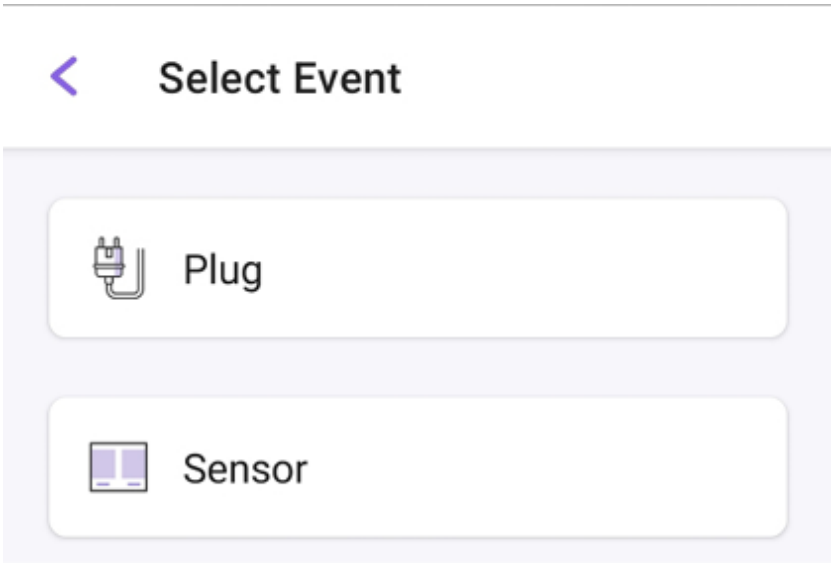


Figure 9.19: Selecting an event in the RainMaker app

8. As the action, set **Plug OFF**.

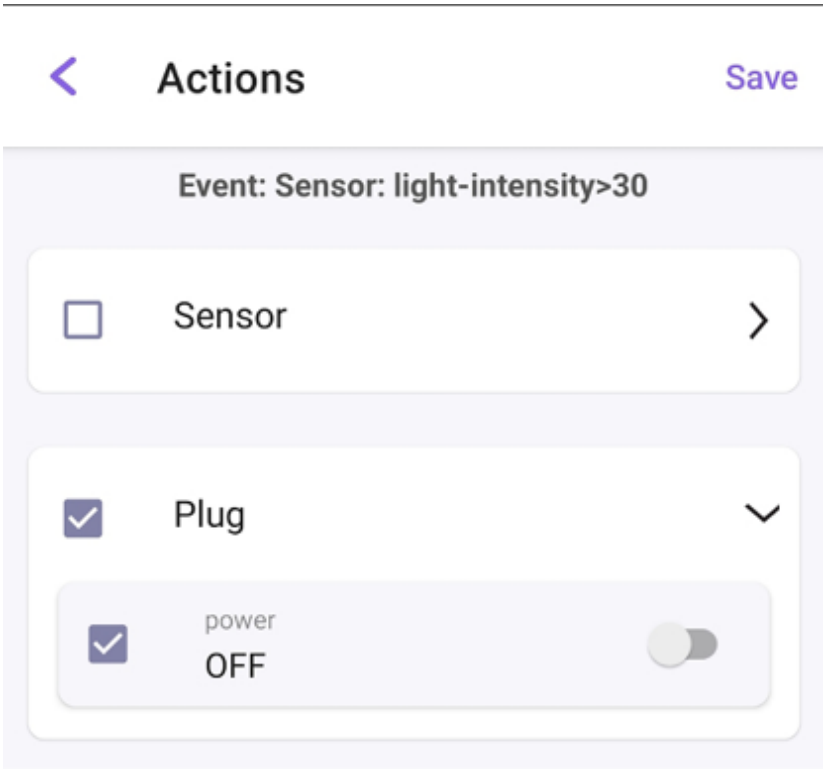


Figure 9.20: Setting an action in the RainMaker app

9. After saving the automation, we see it in the list.

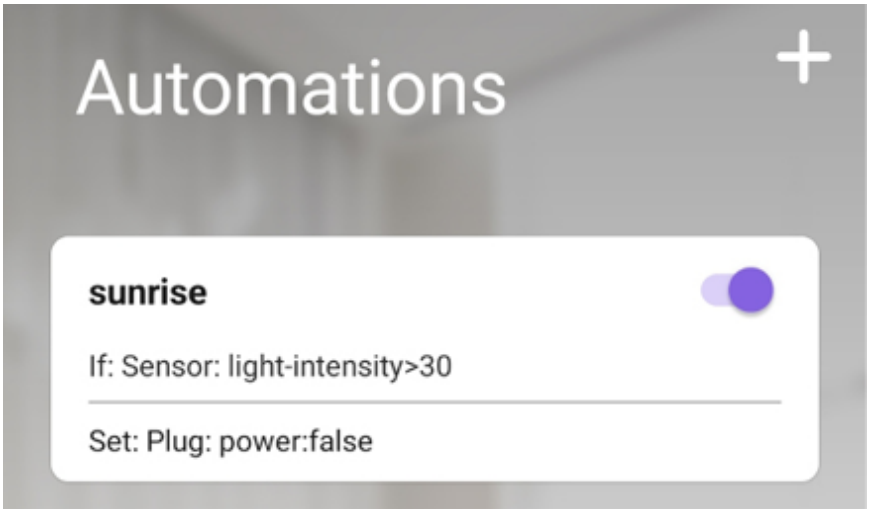


Figure 9.21: An automation in the RainMaker app

10. Use your mobile phone's torch to set the light intensity higher than the threshold value of the event and observe the plug changing its state to off if its state is currently on.
11. The RainMaker platform supports Alexa and Google Assistant voice services. Let's try integrating with Alexa. Select **Settings** from the bottom list and tap **Voice Services** in the settings.

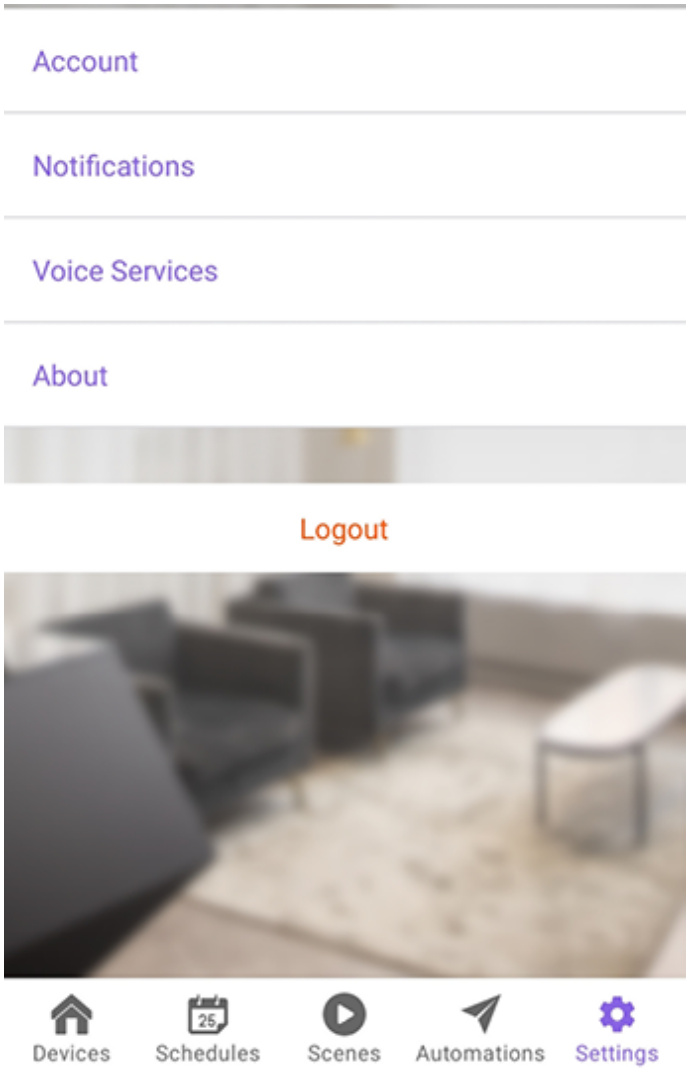


Figure 9.22: Settings in the RainMaker app

12. Select **Amazon Alexa** from the list.

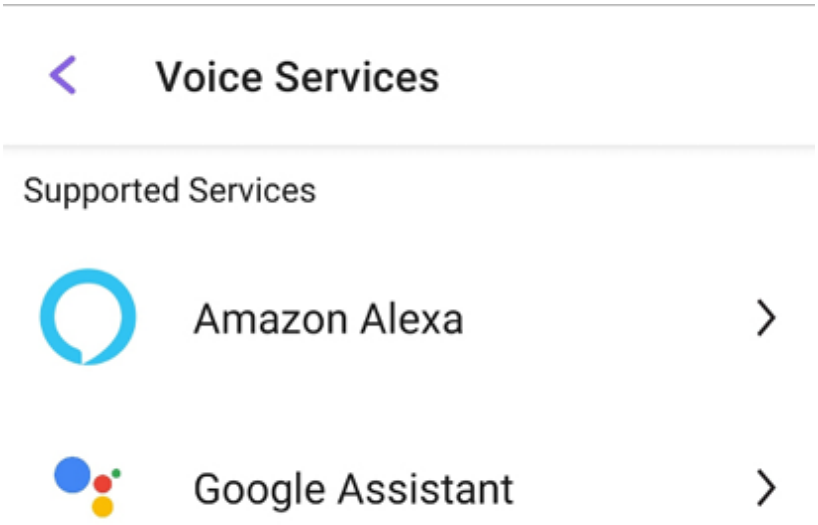


Figure 9.23: Voice Services in the RainMaker app

13. We need to do account linking with Alexa. Follow the wizard to complete this step.
14. Run the Alexa application on your mobile device and go to the list of all devices. Observe that **Plug** and **Sensor** are listed there. Select **Plug**.

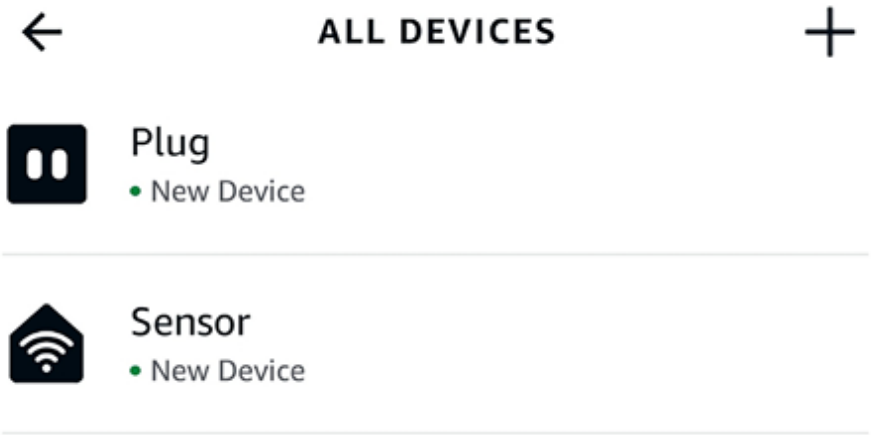


Figure 9.24: RainMaker devices in the Alexa app

15. Try turning the plug on and off by tapping on the on/off button in the middle.

Figure 9.25: Plug power is on in the Alexa app

16. Try voice commands by tapping on the **Alexa** voice icon. Say "*Turn off plug*" when it is listening.

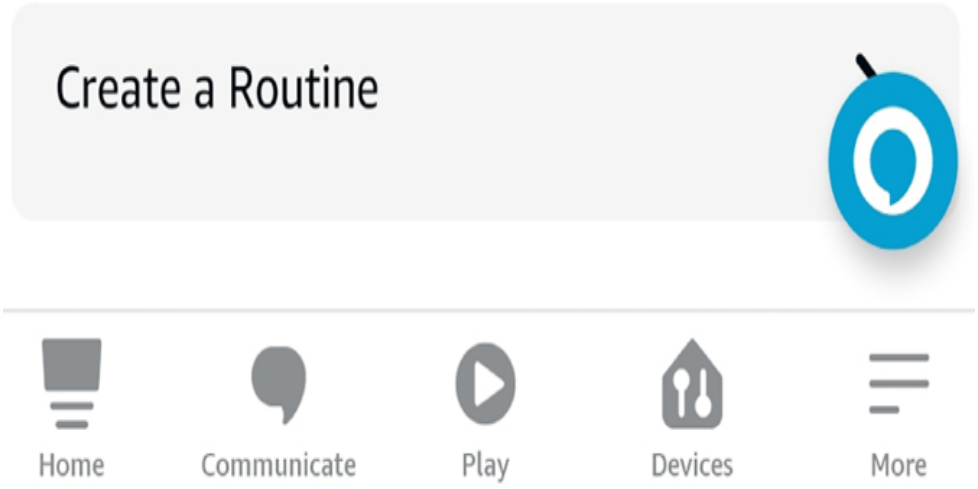


Figure 9.26: Alexa voice commands in the Alexa app

Congratulations! You have developed a fully fledged smart home product on the RainMaker cloud. RainMaker is a great platform for testing ideas quickly. Moreover, we can customize the RainMaker platform and deploy real products in our AWS accounts. For more information, visit https://rainmaker.espressif.com/.

The project has many different components, so there are many places to make a mistake. Check out the following troubleshooting section to get help if you encounter an error or unexpected behavior.

Troubleshooting

Each project comes with its own specific challenges. If you encounter some issues while developing the project, the following list may help:

- First and foremost, make sure the paths in the root `CMakeLists.txt` file are correct if you configure the project yourself. The GitHub clone should compile without an issue since it is already configured with the relative paths in the repository, but when you collect the components and libraries yourself, you need to update the paths according to your directory structure.
- The `idf.py` tool detects the connected devkit automatically without the serial port argument if there is only one. However, we need two devkits to be connected at the same time in this project. Therefore, pass `-p <port>` to `idf.py` when you flash or monitor the second one.
- The default transport method is `BLE` when provisioning the RainMaker nodes in this project. You can run `menuconfig` and set the provisioning method to `SoftAP` in the **(Top) → ESP RainMaker App Wi-Fi Provisioning → Provisioning Transport method** menu. If you stick with the default option, i.e., `BLE`, make sure Bluetooth is enabled on your mobile phone before provisioning the nodes.

- The default RainMaker configuration enforces the `MQTT budget` to prevent buggy code from bombarding the MQTT broker with excessive load. You can change the MQTT budget settings from `menuconfig` from **(Top)** → **ESP RainMaker Config**. When we hit the budget limit, we see the following message on the serial console:

  E (5283739) `esp_rmaker_mqtt`: Out of MQTT Budget. Dropping publish message.

- When provisioning a new device, the mobile application sometimes fails. If this happens, reboot the devkit and start the process from the beginning.
- You cannot add two devices to the RainMaker platform with the same identifiers. If you have already added your devkits, remove them by using the mobile application before adding them again in the scope of this project.

You can improve the project by adding more features as practice. We will discuss some of them in the next section.

New features

Here are some ideas for new features that you can add to the project:

- ESP32-C3 DevkitM-1 doesn't have a display, but it would be nice to have a feedback mechanism for the plug connection status. Use the devkit's LED as an indicator to show the disconnected and connected events.
- We enabled the **Local Control Service** on the devices (https://rainmaker.espressif.com/docs/local-control-

service.html). It runs a web server and mDNS on the device. Update the plug application to define an endpoint (https://docs.espressif.com/projects/esp-idf/en/v4.4.4/esp32c3/api-reference/protocols/esp_http_server.html) on the web server and handle the state change requests. You can use the following `curl` commands to test:

```
$ curl -X PUT http://<ip>/state?value=ON
$ curl -X PUT http://<ip>/state?value=OFF
$ curl -X PUT http://<ip>/state?value=TOGGLE
```

- Update the multisensor application to discover the plug in the network by scanning mDNS packets (https://docs.espressif.com/projects/esp-idf/en/v4.4.4/esp32/api-reference/protocols/mdns.html) and utilize the physical buttons of ESP32-S3 Box Lite to send ON/OFF/TOGGLE commands over a REST API (https://docs.espressif.com/projects/esp-idf/en/v4.4.4/esp32s3/api-reference/protocols/esp_http_client.html). You can use the following Linux commands (or `Bonjour` for other platforms, available at https://developer.apple.com/bonjour/) to discover the plug in the network:

```
$ avahi-browse -a
+ wlp2s0 IPv4 Local Control Service  _esp_local_c
$ avahi-browse -rt _esp_local_ctrl._tcp
```

```
+ wlp2s0 IPv4 Local Control Service _esp_local_ct
= wlp2s0 IPv4 Local Control Service _esp_local_ct
   hostname = [<hostname>]
   address = [<ip>]
   port = [8080]
$ ping <ip>
```

RainMaker is a great platform for developing a smart home application, but it is not the only way. We can employ local mesh networks in such projects. Here are some options:

- ESP-NOW (https://www.espressif.com/en/solutions/low-power-solutions/esp-now): A low-power point-to-point Wi-Fi protocol developed by Espressif. Suitable for devices with batteries.
- ESP-WI-FI-MESH (https://www.espressif.com/en/products/sdks/esp-wifi-mesh/overview): A Wi-Fi mesh protocol by Espressif. Nodes run SoftAP-STA modes at the same time to create a mesh network.
- ESP-BLE-MESH (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/esp-ble-mesh/ble-mesh-index.html): The Espressif implementation of the BLE mesh networking.
- OpenThread (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/openthread.html): The Espressif implementation of the OpenThread networking.

You can try these networking protocols in your plug and multisensor applications to set up a local network and implement the application logic on top of them. To close the IoT loop, we still need a cloud service and a mobile application to communicate with them over the internet.

Summary

In this chapter, we designed and developed a smart home solution with two different devices, a plug and a multisensor, for a given set of requirements. We integrated them with the **RainMaker platform**, which provides cloud connectivity for our devices. We used the **mobile application** that comes with the RainMaker platform in order to provision and manage the devices. The platform also supports **Alexa** and **Google Assistant** voice services. We tested our devices with the Alexa mobile application after the account linking between RainMaker and Alexa.

IoT products are great; they make people's lives easier in the most basic terms. However, they become even more effective when they are powered by artificial intelligence. In the next chapter, we will discuss how to run **Machine Learning** (**ML**) applications on ESP32.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

10

Machine Learning with ESP32

Machine learning (**ML**) is a huge topic covering different disciplines and knowledge areas. It requires data science, math, algorithm development, signal processing, and subject-matter knowledge about the specific problem that we want to solve. Obviously, our goal in this chapter cannot be to discuss all these underlying subjects, but to learn how to apply ML techniques in IoT projects where possible and beneficial.

Developing an ML application is not an easy task, but developing IoT applications with ML capabilities is extra challenging due to the hardware that we work on. IoT devices are constrained in terms of processing capabilities, memory, and power. As a result, a framework that targets IoT devices has to provide the right tools and libraries that take all these constraints into account. **tinyML** is the term to describe the field of machine learning that relates to constrained devices.

The topics that we are going to cover in this chapter are:

- Learning the ML basics
- Running inference on ESP32
- Developing a speech recognition application

Technical requirements

The hardware requirements of the chapter are:

- ESP32-S3 Box Lite
- An RGB LED
- A 440Ω resistor

The libraries and tools to run the examples of the chapter are included in the book repository, therefore there is no need to clone an external repository or download any other third-party tools.

You can find the examples of the chapter here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch10

Learning the ML basics

Before developing tinyML applications on ESP32, we will talk about some ML basics and terminology that will help us to better understand the examples of the chapter and ML concepts in general. As alluded to in the introduction, ML is not for all types of computing problems. We will discuss how ML differs from traditional programming and the types of ML approaches to solve different classes of problems. Then we will have an overview of the tinyML pipeline to understand the overall process of developing an ML application on a constraint device.

Let's start with what ML is and what we can do with it.

ML approaches to solve computing problems

In traditional software development, we apply a set of rules (the program) to data (input) to get a result (output). Since we know the requirements, we write a specific program to generate a distinct output from data. We can see the flow in traditional programming in the following figure:

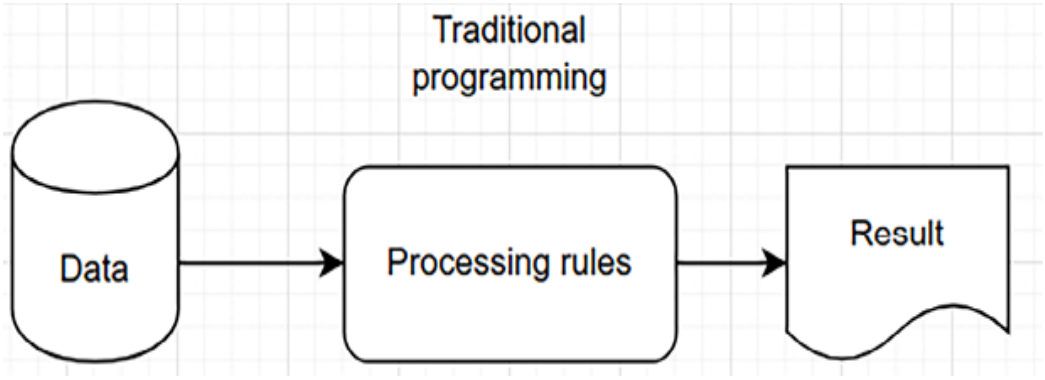


Figure 10.1: Traditional programming approach

The goal of ML is to find the rules that generate the listed output for the given set of data, and then use those rules in an ML application to estimate the result when new data of the same nature is given to the system as an input. The following figure shows the ML approach:

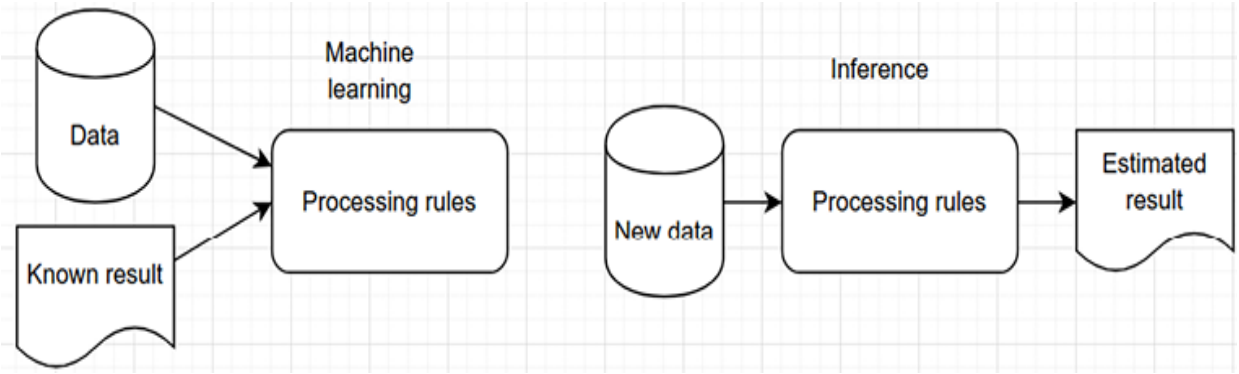


Figure 10.2: Machine learning approach

The traditional approach generates exact results; there is no room for possibilities. We write the rules and the system runs according to those rules. In very general terms, the machine learning approach is a two-step process. First, we use an ML algorithm to find the rules. Then, we import those rules into the real application that processes new data and generates an estimated result.

Let's explore the three different ML approaches.

Supervised learning

In supervised learning, we provide the ML algorithms with data and what data represents. A classic example of that is image classification. For instance, we can collect different animal images and label each of them with the animal species it shows. Then the ML algorithm generates the rules to differentiate animal species. When an ML application applies the same rules to a new set of animal images, it can name the animals as learned in the previous step. In ML terminology, we first **train** an **ML model** to learn which image shows which animal and use this model for **inference** in production. The user of the application provides a new image as input and receives an output that indicates which animal it could be. Of course, the output may not be accurate since the new image can be a bit blurred, or show the animal from another angle, hiding some of its distinct features. Thus, the result comes with a probability. Another example would be training a model with sound recordings where different people say "yes" and "no", and use that model for yes/no inference.

Unsupervised learning

The algorithms in **unsupervised learning** target different types of problems from supervised learning and they don't need data labels during training to achieve their goals. They try to organize data into groups during training to generate models and mark the outliers at runtime by looking at this model. Anomaly detection is one of the problems that can be solved with unsupervised learning. For instance, we can collect data from wind turbines when they are operating normally and use that data to train a model to detect deviations from the normal. In this way, it would be possible to understand when to maintain a turbine before it is broken.

Reinforced learning

The learning method in this group is trial and error. The algorithm (agent) has a goal to reach and tries different options to reach its goal. Another party (environment) scores its trials and the ML algorithm searches the best combination of actions according to the given scores. Training a chatbot is one example where reinforced learning can be applied. By scoring how successful a chatbot is at the end of a conversation session, we help it for the next sessions.

Whatever the approach is, the most important factor for the success of ML algorithms is the quality of data. An algorithm can learn only what it is provided with. If there is one golden rule for ML, it is "garbage in, garbage out"

Now that we have learned about the different ML approaches, we can talk about the tinyML pipeline, which describes the overall ML development process for IoT devices.

TinyML pipeline

The **tinyML pipeline** defines the steps to develop a tinyML application, starting from data collection to deployment on the constrained device. They are:

- Data collection and preprocessing
- Designing and training a model
- Optimizing and preparing the model for deployment
- Running inference on an IoT device

Let's expand on these steps.

Data collection and preprocessing

Data collection and preprocessing have a paramount effect on the quality and success of any ML application. As said, garbage in, garbage out. **Bias** happens if some part of possible data samples is not included in the training data. For example, if we train our animal recognition model with all tabby cats but no white cats, we may end up with a model that calls a white cat a polar bear.

Designing and training a model

In the designing and training a model step, the curated data coming from the previous step is used to train a model. This step requires experience to have the optimal model at the end. For supervised learning, the available data is usually split into three groups: **train**, **validation**, and **test datasets**. By leveraging the train and validation sets, we generate a model that we can test by using the test set.

The test set ensures that our model has not specialized on the training data (**overfitting**). If our model gives great results with the validation data (**high accuracy**) but worse on the test set, then it means it is overfitted to the training data. Thinking about our animal recognition model, an overfitted model would recognize the cats in the training set, but not other new cats. **Hyperparameter tuning** (changing the parameters that control the learning process) helps with overfitting.

Optimizing and preparing the model for deployment

The model that we have after training can take gigabytes of memory. Obviously, we don't have this luxury with constrained devices. The next step in the tinyML pipeline is to optimize the model for MCUs and take the model size from gigabytes down to tens of kilobytes. **Quantization** is one of the techniques that can be applied at this step, where the values of **model parameters** are converted from float to the int8 type for MCUs. Making calculations in int8 is the trick that makes running ML applications possible on MCUs.

Running inference on an IoT device

The model is now ready for deployment on a constrained device. The **inference engine** running in the embedded application inputs data from the environment via sensors and generates a result, or **prediction**, using the model. We can also sample runtime data to adapt our model further to the environment it runs in the next loop of model training.

An ML platform covers all these steps with tools and libraries. **TensorFlow** is one of the most popular platforms for developing ML applications. On the TensorFlow platform, we can normalize the collected raw data to make it ready to train a model, and then design and train the model with the normalized data for our purposes. **TensorFlow Lite for Microcontrollers** (**TFLM**) is a framework in the TensorFlow platform. We use it in the optimization and inference steps of the tinyML pipeline. The MCU platforms supported by TFLM are listed on its website: https://www.tensorflow.org/lite/microcontrollers.

Discussing the entire pipeline is a whole book in itself and won't be covered in this book. Instead, we will see different inference examples on ESP32 to learn the capabilities of ML and ESP32 in this context.

Running inference on ESP32

Deep learning is a supervised learning method in ML. Similar to the human brain, it contains neurons, ie. computational units. **Neural networks** (**NNs**) are implementations of the deep learning method. A neural network has several layers of neurons and each layer can have a different number of neurons. In the last layer, the neural network generates its prediction. There are different types of layers, such as a fully connected layer, convolutional layer, pooling layer, etc. In a **fully connected layer**, all neurons are connected to every neuron in the next layer so that they can pass their calculations to the next layer fully. The following figure shows an NN with fully connected layers:

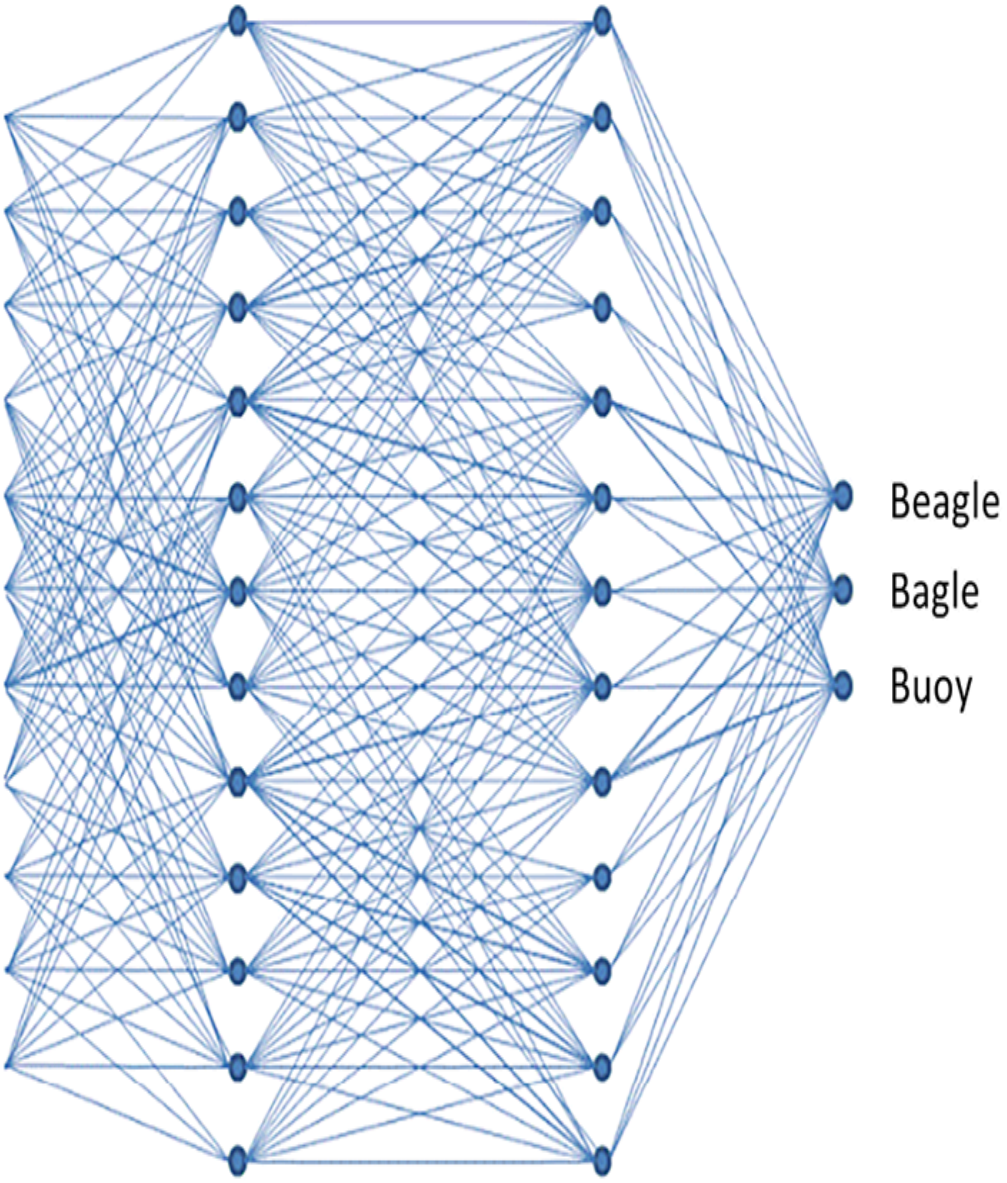


Figure 10.3: A fully-connected NN (Source: Wikimedia Commons)

An NN utilizes number arrays, or **tensors**, as a data structure. For example, a vector is a **One-Dimensional** (**1D**) tensor, and a matrix is a 2D tensor. A scalar, a single number, is a 0D tensor. Data is represented in the form of tensors. An image can be represented as a 2D tensor where each pixel can be accessed via indices.

In this example, we will be developing an application that uses a pre-trained sine model. This model is trained to predict sine values between 0 and 2∏. We provide a random value between 0 and 2∏ as input and the inference engine outputs a prediction for it. The model is an NN with three fully connected layers. The first two layers have 16 neurons each and the third one is the output layer with a single neuron.

If you want to train the model by yourself, it comes from TFLM's hello-world example here, where the steps to train the model are also explained: https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/hello_world

The devkit that we will use in the example is ESP32-S3-BOX-Lite. In the application, we will feed the inference engine with random values and draw the output on a simple coordinate system to visualize the sine wave on the devkit's TFT screen by utilizing LVGL.

Creating the project

Let's configure the project as shown in the following steps:

1. The application requires several other ESP-IDF components to work together. They are all provided in the book repository, integrated and tested, therefore it is best to use them as they are by cloning the repository fully.

2. Change to the directory where you want to create an ESP-IDF project and activate the IDF environment:

```
$ source ~/esp/esp-idf/export.sh
$ idf.py --version
ESP-IDF v4.4.4-1-g6727a4f775
```

3. Create an IDF project and change to its directory:

```
$ idf.py create-project tflite_ex && cd tflite_ex
```

4. Copy the `sdkconfig` file from the book repository into your project directory:

```
$ cp <book_repository>/ch10/tflite_ex/sdkconfig
```

5. Edit the root `CMakeLists.txt` file with the following content:

```
cmake_minimum_required(VERSION 3.5)
set(EXTRA_COMPONENT_DIRS
<book_repository>/components/   <book_repository>
<book_repository>/ch10/components/tflite-micro-es
<book_repository>/ch10/components/tflite-micro-es
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(tflite_ex)
```

6. Copy the application UI sources and the TFLM model sources into the **main** directory of your project:

```
$ cp <book_repository>/ch10/tflite_ex/main/{ui.h,
```

7. In the project, you should have the following files:

```
$ ls *
CMakeLists.txt sdkconfig
main:
CMakeLists.txt model.cc model.h tfl_ex.c ui.c ui
```

8. Rename the application C source to a CPP file:

```
$ mv main/tflite_ex.c main/tflite_ex.cpp
```

Coding the application

Now, we are ready to develop the application. Let's add the `main/AppUi.hpp` C++ header file for the GUI integration:

```
#pragma once
#include <mutex>
#include <vector>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "esp_log.h"
```

```
#include "bsp_board.h"
#include "bsp_lcd.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
#include "ui.h"
```

We include the BSP and LVGL header files to have access to the devkit screen. The GUI elements of the application are defined in the `ui.h` file. We continue with the definition of the `AppUi` class:

```
namespace app
{
    class AppUi
    {
    private:
        std::mutex m_ui_access;
        static void lvglTask(void *arg);
```

The `m_ui_access` member variable is a mutex to synchronize the access to the underlying LVGL functions in a thread-safe manner. The `lvglTask` class function redraws the screen for any updates. The function body implementation will come after the class definition. We can move on to the `public` section functions of the class:

```
    public:
        void init(void)
        {
            bsp_board_init();
```

```
            lv_port_init();
            ui_init();
```

In the `init` public member function, we initialize the hardware, LVGL, and the application GUI. Then we create a FreeRTOS task for LVGL as follows:

```
            xTaskCreate(lvglTask, "lvgl", 6 * 1024,
            bsp_lcd_set_backlight(true);
        }
```

FreeRTOS runs the `lvglTask` class function as a task with the help of the `xTaskCreate` function. Then we call the `bsp_lcd_set_backlight` function to turn on the backlight of the screen and make the UI components visible. The next member function will draw a point on the screen:

```
        void drawSinePoint(float x_val, float y_val)
        {
            std::lock_guard<std::mutex> lock(m_ui_ac
            lv_coord_t x_coord = static_cast<lv_coor
                x_val * 30) + 50;
            lv_coord_t y_coord = static_cast<lv_coor
                y_val * -50) + 100;
            ESP_LOGI("lvgl", "x=%.5f y=%.5f (%d,%d)"
                y_val, x_coord, y_coord);
            ui_setPoint( x_coord, y_coord );
```

```
        } // function end
    }; // class end
```

The `drawSinePoint` function will display a dot on the screen for a given *(x,y)* value pair. The inputs are of the `float` type. We scale and convert them into the `lv_coord_t` type to be able to pass the coordinates to the `ui_setPoint` function. The class definition is done, but we need to implement the `lvglTask` function body as it comes next:

```
    void AppUi::lvglTask(void *arg)
    {
        AppUi *obj = reinterpret_cast<AppUi *>(arg);
        while (true)
        {
            {
                std::lock_guard<std::mutex> lock(
                    obj->m_ui_access);
                lv_task_handler();
            }
            vTaskDelay(pdMS_TO_TICKS(10));
        } // while end
    } // function end
} // namespace end
```

In the `lvglTask` function, we run the task loop. The loop calls the `lv_task_handler` function of LVGL to refresh the screen every 10ms periodically. This completes the UI implementation.

Next, we add the `main/AppSine.hpp` header file where we run the TFLM inference engine to generate sine predictions:

```
#pragma once
#include <cstdint>
#include <limits>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "bootloader_random.h"
#include "esp_random.h"
#include "tensorflow/lite/micro/micro_mutable_op_res
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "model.h"
#include "AppUi.hpp"
```

We begin by including numerous header files. Notable ones are, of course, the TensorFlow headers. They define the classes and functionality to load the model and run the inference engine. The `model.h` file has the global variable declaration for the sine model. Next comes the `AppSine` class definition:

```
namespace app
{
    class AppSine
    {
    private:
```

```
        const tflite::Model *m_model{nullptr};
        tflite::MicroInterpreter *m_interpreter{null
```

We define the pointers to the model and the inference engine in the private section of the class. TFLM names the inference engine as the interpreter. They will be available to the application after the initialization. We continue with other variables:

```
        uint8_t m_tensor_arena[2000];
        static constexpr float INPUT_RANGE = 2.f * 3
        AppUi *m_ui;
```

The `m_tensor_arena` array is the memory area for the inference engine to be used at runtime. The `INPUT_RANGE` constant defines the model input range. The random values that we provide as input must stay in this range. The `AppSine` class also needs the UI for sending inference outputs. Next comes the `public` section of the definition:

```
    public:
        void init(AppUi *ui)
        {
            m_ui = ui;
```

The first function that we define in the `public` section is `init`. It takes a pointer to the UI object and stores it in the `m_ui` member variable. Then we initialize the TFLM variables:

```
            m_model = tflite::GetModel(g_model);
```

First, we get a pointer to the model by calling the `tflite::GetModel` function. It takes `g_model` as an argument and creates a `tflite::Model` pointer from it. `g_model` is the raw model bytes that reside in the `model.cc` source code file of the project. In the next lines, we create a TFLM op-resolver:

```
            static tflite::MicroMutableOpResolver<1>
            resolver.AddFullyConnected();
```

An op-resolver is used for defining the operations in a model. The sine NN model uses only fully connected layers, as we discussed earlier, and the `static resolver` variable contains the operations for the fully connected layers after calling its `AddFullyConnected` function. The interpreter will need this op-resolver to convert an input value to an output. In the remaining part of the `init` function, we initialize the interpreter:

```
            static tflite::MicroInterpreter
                static_interpreter(m_model, resolver
                m_tensor_arena, sizeof(m_tensor_aren
            m_interpreter = &static_interpreter;
            m_interpreter->AllocateTensors();
        }
```

The interpreter constructor takes the model, op-resolver, and the memory area for tensors as its arguments and creates an inference engine from them. The `AllocateTensors` function of the interpreter defines the internal tensor structures for the model in the tensor arena. This completes the initialization of the inference engine and we develop the function to run inference next:

```
        void run(void)
        {
            TfLiteTensor *input_tensor{m_interpreter
            TfLiteTensor *output_tensor{m_interprete
            bootloader_random_enable();
```

In the `run` function, we will run inference. For that, we need to access the input tensor and output tensor of the interpreter. We will feed the local `input_tensor` variable with random values and read the output from `output_tensor` in the next loop:

```
            while (1)
            {
                vTaskDelay(pdMS_TO_TICKS(500));
                float random_val = static_cast<float
                                        static_cast<f
                                        std::numeric_
                                        ::max());
                float x = random_val * INPUT_RANGE;
```

In the loop, we generate a random value every 500ms. The random value is a floating point number between 0 and 1 and we scale it to the input range. The result is again a floating point number. However, we can't use it directly as input to the interpreter since it is optimized for *int8* operations. In the following lines, we convert the input value to *int8*:

```
                int8_t x_quantized = x /
                    input_tensor->params.scale +
                    input_tensor->params.zero_point;
                input_tensor->data.int8[0] = x_quant
```

The `input_tensor` variable has information about the quantization offset and scale. By using the offset (`input_tensor->params.zero_point`) and scale (`input_tensor->params.scale`) values, we quantize the floating point input to *int8* and then we can update the input tensor data with the quantized value. Having the input ready, the interpreter can run inference:

```
                m_interpreter->Invoke();
```

Running inference is as simple as making a single call to the `Invoke` function of the interpreter. The interpreter fills the output tensor with its prediction. We can take this value from the output tensor and use the *(x,y)* pair to draw a point on the screen, as comes next:

```
                int8_t y_quantized = output_tensor->
                float y = (y_quantized - output_tens
```

```
                        params.zero_point) * output_tens
                        params.scale;
                    m_ui->drawSinePoint(x, y);
                } // loop end
            } // function end
        }; // class end
    } // namespace end
```

The output tensor has the quantized prediction at `output_tensor→data.int8[0]`. We reverse the quantization operation for the output and convert it to a floating point value. Then, we pass the *(x,y)* pair to the UI object to be drawn on the screen. This is the end of the sine model interpreter integration.

Having all the pieces implemented, we can develop the `app_main` function of the application in `main/tflite_ex.cpp`:

```
#include "AppSine.hpp"
#include "AppUi.hpp"
namespace
{
    app::AppSine app_sine;
    app::AppUi app_ui;
}
```

We first include the header files that we have just developed and instantiate the classes in an anonymous namespace. We will connect the objects in the `app_main` function as follows:

```
extern "C" void app_main(void)
{
    app_ui.init();
    app_sine.init(&app_ui);
    app_sine.run();
}
```

We first initialize the UI object and pass its address to the `app_sine` initialization. When we call the `run` function of the `app_sine` object, the application starts the ML inference and shows the output on the screen. Before compiling the application, we need to edit the `main/CmakeLists.txt` file to specify the new source code files. Here is its content:

```
idf_component_register(SRCS tflite_ex.cpp model.cc u
```

The development is done, and the application is ready for testing. Let's do it next.

Testing the application

We can build the application and see the output on both the serial logs and the screen, as in the following:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting....
```

```
Detecting chip type... ESP32-S3
<logs removed>
I (583) lv_port: Try allocate two 320 * 20 display b
I (593) lv_port: Add KEYPAD input device to LVGL
I (1103) lvgl: x=1.63838 y=1.04206 (99,48)
I (1603) lvgl: x=3.59154 y=-0.44055 (157,122)
I (2103) lvgl: x=0.02657 y=0.05930 (50,98)
I (2603) lvgl: x=1.54455 y=1.02512 (96,49)
```

After a while, we can see a sine wave-like output on the screen, as in the following figure:

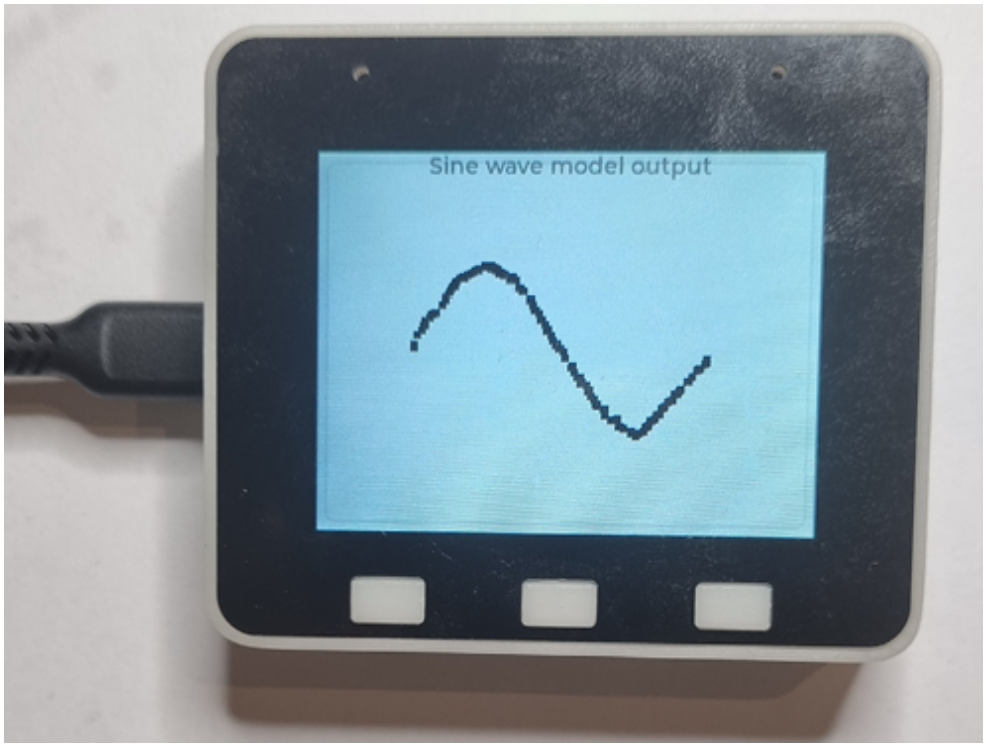


Figure 10.4: The output of the sine wave inferences

This example is, of course, a very simple experiment to see how TFLM inference works. The TFLM interpreter in this example uses

only a single scalar as input and outputs another scalar. However, almost any other tinyML application uses the same logic to employ the inference engine and generate predictions based on the model it uses.

In the next topic, we will develop a voice application on ESP32-S3-BOX-Lite.

Developing a speech recognition application

Speech recognition is another field where ML algorithms can do a very good job and Espressif invests a lot in this field in terms of both hardware and software. The ESP32-S3 series of MCUs is a response to the advancements in ML. When we look at the ESP32-S3 technical reference manual, we see that ESP32-S3 has an extended instruction set to support 128-bit vector operations with an additional eight 128-bit general-purpose registers. Having a **Single Instruction**, **Multiple Data** (**SIMD**) paradigm in mind, the ESP32-S3 **Arithmetic Logic Unit** (**ALU**) is capable of, for example, processing 16 8-bit vectors in a single instruction with this extended instruction set, or **Processor Instruction Extensions** (**PIEs**), as they're called in the ESP32-S3 technical reference.

Espressif supports this hardware with advanced frameworks and libraries. Here is a short summary of them:

- **ESP-DSP**: **The Digital Signal Processing** (**DSP**) library for vector operations, matrix operations, Fast Fourier Transform

(FFT), etc. Available at https://github.com/espressif/esp-dsp

- **ESP-SR**: **The Speech Recognition** (**SR**) framework. It integrates the Audio Front-End library, wake-word engine, and speech command recognition models. Available at https://github.com/espressif/esp-sr
- **ESP-Skainet**: Integrates different Espressif development boards with ESP-SR to show voice assistant features. Available at https://github.com/espressif/esp-skainet
- **ESP-ADF**: **The Audio Development Framework** (**ADF**) covers low-level hardware drivers as well as audio codes and protocols with many other supporting libraries. Available at https://github.com/espressif/esp-adf
- **ESP-DL**: Espressif's implementation of the deep learning algorithm. Provides APIs for NN, layers, and models. Available at https://github.com/espressif/esp-dl
- **ESP-WHO**: Image processing framework that employs ESP-DL and adds hardware support on top of it to be used with the Espressif development boards, such as ESP-Eye. Available at https://github.com/espressif/esp-who

In this example, we will run wake-word detection and voice command recognition on ESP32-S3-BOX-Lite. We will connect an RGB LED to the devkit and control its color with voice commands. The application will activate when it detects *"Hi, ESP"* being said and then wait for a command to change the color. The following figure shows the connections between ESP32-S3-BOX-Lite and the RGB LED:

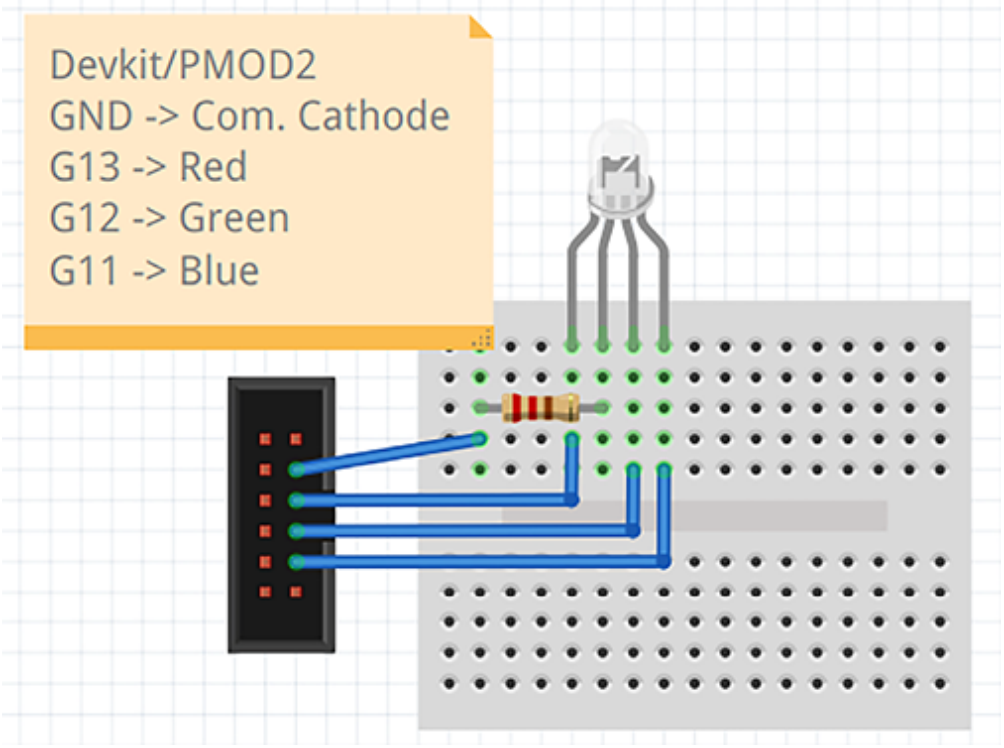


Figure 10.5: Fritzing sketch of the project

If you have the RGB LED module with the devkit, you can just plug it into the devkit/PMOD2 connector's corresponding pins.

Creating the project

With this hardware setup, we can configure the project as follows:

1. Clone the book repository if you haven't done so already. The book repository has all the required libraries of the project integrated and tested.
2. Change to the directory where you want to create an ESP-IDF project and activate the IDF environment:

```
$ source ~/esp/esp-idf/export.sh
$ idf.py --version
```

```
ESP-IDF v4.4.4-1-g6727a4f775
```

3. Create an IDF project and change to its directory:

```
$ idf.py create-project speech_ex && cd speech_ex
```

4. Copy the `sdkconfig` and `partitions.csv` files from the book repository into your project directory:

```
$ cp <book_repository>/ch10/speech_ex/{sdkconfig,
```

5. Replace the content of the root `CMakeLists.txt` file with the following:

```
cmake_minimum_required(VERSION 3.15)
set(EXTRA_COMPONENT_DIRS
<book_repository>/ch10/components/esp-skainet/cor
<book_repository>/ch10/components/esp-skainet/cor
<book_repository>/ch10/components/esp-skainet/cor
<book_repository>/ch10/components/esp-skainet/cor
<book_repository>/ch10/components/esp-skainet/cor
<book_repository>/ch10/components/esp-skainet/cor
<book_repository>/ch10/components/esp-skainet/cor
<book_repository>/ch10/components/esp-skainet/cor
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(speech_ex)
```

6. Copy the LED driver sources into the `main` directory of your project:

```
$ cp <book_repository>/ch10/speech_ex/{app_led.h,
```

7. You should have the following files now in the project:

```
$ ls *
CMakeLists.txt partitions.csv sdkconfig
main:
app_led.c app_led.h CMakeLists.txt speech_ex.c
```

8. Rename the application C source to a CPP file:

```
$ mv main/speech_ex.c main/speech_ex.cpp
```

As you might have noticed, we will develop the application on top of *ESP-Skainet*. The original *ESP-Skainet* doesn't support ESP32-S3-BOX-Lite but the version in the book repository does, so we can easily develop the application on this devkit. Another interesting point about *ESP-Skainet* is that it imports the *ESP-DSP* and *ESP-SR* frameworks as components and takes advantage of the ESP32-S3 hardware for tinyML applications.

The project is ready for coding the application, which we will do next.

Coding the application

Let's start developing the application by adding a new C++ header file, `main/AppLed.hpp`, that integrates the LED driver in the application:

```
#pragma once
#include <cstdint>
#include "app_led.h"
```

We first include the `app_led.h` header file to access the LED driver functionality. Then we define the GPIO pins that the RGB LED connected:

```
#define APP_RED_PIN 13
#define APP_GREEN_PIN 12
#define APP_BLUE_PIN 11
```

Looking at the Fritzing sketch in *Figure 10.5*, you can see that the red leg, the green leg, and the blue leg are connected to GPIO 13, GPIO 12, and GPIO 11 on PMOD2 of the devkit, respectively. Next come three macros to extract red, green, and blue values from an integer value:

```
#define RED(x) (uint8_t)(((x)&0xff0000) >> 16)
#define GREEN(x) (uint8_t)(((x)&0x00ff00) >> 8)
#define BLUE(x) (uint8_t)((x)&0x0000ff)
```

They are simple macros that first mask and then shift the integer input to extract a color value from it. Let's define several colors, as

follows:

```
namespace app
{
    enum eColor : uint32_t
    {
        White = 0xffffff,
        Red = 0xff0000,
        Lime = 0x00ff00,
        Blue = 0x0000ff,
        Yellow = 0xffff00,
        Cyan = 0x00ffff,
        Magenta = 0xff00ff,
        Green = 0x008000,
        Purple = 0x800080,
        Navy = 0x000080
    };
```

In the `app` namespace, we define an enumeration that contains color RGB values. We will use this enumeration to pass color information between the objects of the application. Then, we define the class that encapsulates the LED access:

```
    class AppLed
    {
    public:
        void setColor(eColor color)
        {
            app_pwm_led_set_all(RED(color), GREEN(co
```

```
                BLUE(color));
        }
```

The first member function of the `AppLed` class is `setColor`. It takes an `eColor` value as its only argument and calls the `app_pwm_led_set_all` function that comes with the `app_led.h` header file to change the LED color to the input value. The other member functions are as follows:

```
        void on()
        {
            app_pwm_led_set_power(true);
        }
        void off()
        {
            app_pwm_led_set_power(false);
        }
```

The client codes of an `AppLed` instance can call the `on` and `off` functions to turn on or off the RGB LED. The final member function is for initializing `AppLed`:

```
        void init(void)
        {
            app_pwm_led_init(APP_RED_PIN, APP_GREEN_
            on();
            setColor(eColor::White);
        } // function end
```

```
    }; // class end
} // namespace end
```

In the `init` function, we first initialize the LED driver by calling the `app_pwm_led_init` function with the GPIO pins that the RGB LED is connected to. Then, we turn the LED on and set its color to white. That is all for the `AppLed` class. Now, we can implement the `AppSpeech` class, which detects voice commands, in `main/AppSpeech.hpp`:

```
#pragma once
#include <cstdlib>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_wn_iface.h"
#include "esp_wn_models.h"
#include "esp_afe_sr_iface.h"
#include "esp_afe_sr_models.h"
#include "esp_mn_iface.h"
#include "esp_mn_models.h"
#include "esp_board_init.h"
#include "model_path.h"
#include "esp_process_sdkconfig.h"
#include "AppLed.hpp"
```

The application uses the models from Espressif. Espressif has already developed some models for us to detect wake-words and voice commands in offline applications. Therefore, an IDF application

doesn't have to be connected to a backend service in order to understand the given verbal commands. **WakeNet** is the model for wake-words and **MultiNet** is the model for voice commands. The header files above enable an application for speech recognition with ready-to-use models. *WakeNet* and *MultiNet* only support English and Chinese for now. We define the indexes for the voice commands next:

```
#define CMD_NONE -1
#define CMD_POWER_ON 0   // power on
#define CMD_POWER_OFF 1  // power off
#define CMD_RED 2        // set color red
#define CMD_GREEN 3      // set color green
#define CMD_BLUE 4       // set color blue
#define LISTEN_CMD_TIMEOUT 6000
```

We will have five voice commands in the application. They are defined in `sdkconfig` and we will discuss how to configure them after the application codes. The `LISTEN_CMD_TIMEOUT` macro shows the duration (6 seconds) to listen for the voice commands after the wake-word (*Hi ESP*) is detected. Next comes the `AppSpeech` class definition:

```
namespace app
{
    class AppSpeech
    {
    private:
        esp_afe_sr_iface_t *m_afe_handle{nullptr};
        esp_afe_sr_data_t *m_afe_data{nullptr};
```

```
        srmodel_list_t *m_models{nullptr};
        static constexpr const char *TAG{"speech"};
        AppLed m_led;
```

In the private section of the class, we start by defining the member variables. **Audio Front-End for Speech Recognition** (**AFE/SR**) separates data and functionality. The `m_afe_handle` variable provides access to the AFE/SR functionality and the `m_afe_data` variable is the access point for static (models) and dynamic data (runtime audio data). The `m_models` variable is the abstraction to load raw model data from the flash. We also have several function declarations in the private section:

```
        static afe_config_t defaultAfeConfig();
        static void feedTask(void *arg);
        static void detectTask(void *arg);
        void handleCommand(esp_mn_iface_t *multinet,
            model_iface_data_t *model_data);
```

`defaultAfeConfig` is a static function that returns a default configuration for AFE/SR. The `feedTask` and `detectTask` functions constitute the core speech functionality as FreeRTOS tasks. `feedTask` provides audio data from the devkit's mics. `detectTask` processes audio data to detect voice commands. The `handleCommand` function runs the corresponding action after a voice command is detected. We continue with the `public` section of the implementation:

```
    public:
        void start(void)
        {
            esp_board_init(AUDIO_HAL_16K_SAMPLES, 1,
            m_led.init();
```

The `start` function is the only function needed to start speech recognition in the application. We first initialize the devkit by calling `esp_board_init`. After that, we initialize the `m_led` object. The next step is to read the models from the flash:

```
            m_models = esp_srmodel_init("model");
            m_afe_handle = const_cast<esp_afe_sr_ifa
               (&ESP_AFE_SR_HANDLE);
```

The `esp_srmodel_init` function reads the models' data from the `model` partition of the flash. The partition name is specified in the `partitions.txt` file of the project. We will have both *WakeNet* and *MultiNet* models on the flash memory after flashing the compiled firmware. Then we initialize the `m_afe_handle` member variable from the `ESP_AFE_SR_HANDLE` definition in the `esp_afe_sr_models.h` header file. It is the access point to the AFE/SR functions. We also need the model data structure, which we initialize next:

```
            afe_config_t afe_config = defaultAfeConf
            afe_config.wakenet_model_name =
```

```
                esp_srmodel_filter(m_models, ESP_WN_
                    nullptr);
            m_afe_data = m_afe_handle->create_from_c
                &afe_config);
```

We first create a default AFE configuration and update its *WakeNet* model name from the `m_models` list by running `esp_srmodel_filter`. Then, we call the interface function, `create_from_config`, of `m_afe_handle` to initialize the `m_afe_data` pointer. When we need static or dynamic AFE/SR data, we will use the `m_afe_data` member variable. We complete the `start` function by creating the FreeRTOS tasks:

```
            xTaskCreatePinnedToCore(detectTask, "det
                8 * 1024, this, 5, nullptr, 1);
            xTaskCreatePinnedToCore(feedTask, "feed"
                this, 5, nullptr, 0);
        } // function end
    }; // class end
```

We assign the `detectTask` task to CPU-1 of ESP32-S3 and the `feedTask` task to CPU-0 to take advantage of having two cores. The class definition is done but we still have functions to implement. Let's start with the `feedTask` function, as comes next:

```
    void AppSpeech::feedTask(void *arg)
    {
        AppSpeech *obj = static_cast<AppSpeech *>(ar
```

```
        size_t buff_size = obj->m_afe_handle->
            get_feed_chunksize(obj->m_afe_data) *
            esp_get_feed_channel()
            * sizeof(int16_t);
        int16_t *audio_buffer = static_cast<int16_t
            malloc(buff_size));
```

We first reserve a buffer for the audio feed data. Its size is calculated by using the audio chunk size and channel count. The audio encoding is 16-bit mono in the models, therefore we use `int16_t` as the buffer type. Next comes the task loop:

```
        while (1)
        {
            esp_get_feed_data(false, audio_buffer, b
            obj->m_afe_handle->feed(obj->m_afe_data,
                audio_buffer);
        }
    } // function end
```

In the loop, we simply fill the audio buffer by calling the `esp_get_feed_data` function and feed it into SR by calling `m_afe_handle→feed.` The next task is for command detection:

```
    void AppSpeech::detectTask(void *arg)
    {
        AppSpeech *obj = static_cast<AppSpeech *>(ar
        bool listen_command = false;
        char *mn_name = esp_srmodel_filter(
```

```
        obj->m_models, ESP_MN_PREFIX, ESP_MN_ENG
    esp_mn_iface_t *multinet =
        esp_mn_handle_from_name(mn_name);
    model_iface_data_t *model_data = multinet->c
        mn_name, LISTEN_CMD_TIMEOUT);
    esp_mn_commands_update_from_sdkconfig(
        multinet, model_data);
```

Similar to *WakeNet*, we need the interface functions and data structure for *MultiNet*. The `multinet` local variable allows us to access the *MultiNet* functions, and the `model_data` local variable is for the data structures. The voice commands that we will catch in the application are defined in `sdkconfig`, thus we call the `esp_mn_commands_update_from_sdkconfig` function to load them into `model_data`. We implement the task loop as follows:

```
    ESP_LOGI(TAG, "waiting wake-word");
    while (1)
    {
        afe_fetch_result_t *res = obj->m_afe_han
            fetch(obj->m_afe_data);
```

In the task loop, we first check whether anything comes from AFE/SR by calling the `fetch` function of the `m_afe_handle` member variable with `m_afe_data` as its parameter. Then, we check if it is the wake-word:

```
 switch (res->wakeup_state)
{
case WAKENET_DETECTED:
    multinet->clean(model_data);
    break;
case WAKENET_CHANNEL_VERIFIED:
    listen_command = true;
    ESP_LOGI(TAG, "waiting command");
default:
    break;
}
```

If the fetch result shows `WAKENET_DETECTED`, i.e. the "*Hi ESP*" wake-word is uttered, we make the `model_data` variable ready for a new voice command by calling the `clean` function of `multinet`. However, the `WAKENET_CHANNEL_VERIFIED` state actually enables listening for a command, which we handle next:

```
if (listen_command)
{
    switch (multinet->detect(model_data,
    {
    case ESP_MN_STATE_TIMEOUT:
        obj->m_afe_handle->enable_wakene
            obj->m_afe_data);
        listen_command = false;
        ESP_LOGI(TAG, "waiting wake-word
        break;
```

If the application is in the listening-command state, the `multinet->detect` call returns the state of the *MultiNet* inference engine. As we configured at the beginning, if a timeout occurs (no known voice command is detected for 6 seconds), we simply enable the *WakeNet* inference engine to wait for the wake-word again. The next state corresponds to a valid voice command detection:

```
                case ESP_MN_STATE_DETECTED:
                    obj->handleCommand(multinet, mod
                default:
                    break;
                } // switch end
            } // if end
        } // loop end
    } // function end
```

When a voice command is detected, we call the `handleCommand` function of the `AppSpeech` instance to react to the command. Here is the `handleCommand` member function:

```
    void AppSpeech::handleCommand(esp_mn_iface_t *mu
        model_iface_data_t *model_data)
    {
        int cmd = CMD_NONE;
        esp_mn_results_t *mn_result = multinet->
            get_results(model_data);
        ESP_LOGI(TAG, "detection results:");
        for (int i = 0; i < mn_result->num; i++)
        {
```

```
            ESP_LOGI(TAG, "- cmd:%d prob:%f", mn_res
                command_id[i], mn_result->prob[i]);
            if (mn_result->prob[i] > 0.5)
            {
                cmd = mn_result->command_id[i];
                break;
            }
        }
```

In the `handleCommand` function, we first retrieve the inference results by calling `multinet->get_results` over `model_data`. The *MultiNet* inference can decide several matches, so we select the one that has a probability of over 50%. Next, we just do what the voice command says:

```
        switch (cmd)
        {
        case CMD_POWER_ON:
            m_led.on();
            break;
        case CMD_POWER_OFF:
            m_led.off();
            break;
        case CMD_RED:
            m_led.setColor(eColor::Red);
            break;
        case CMD_GREEN:
            m_led.setColor(eColor::Green);
            break;
        case CMD_BLUE:
```

```
            m_led.setColor(eColor::Blue);
            break;
        default:
            break;
        } // switch end
    } // function end
```

In a `switch` statement, we select the voice command and use the `m_led` object to fulfill the command. The only remaining function is `defaultAfeConfig`, which simply returns the default configuration for AFE/SR:

```
    afe_config_t AppSpeech::defaultAfeConfig()
    {
        return {
            .aec_init = false,
            .se_init = true,
            .vad_init = true,
            .wakenet_init = true,
            .voice_communication_init = false,
            .voice_communication_agc_init = false,
            .voice_communication_agc_gain = 15,
            .vad_mode = VAD_MODE_3,
            .wakenet_model_name = nullptr,
            .wakenet_mode = DET_MODE_2CH_90,
            .afe_mode = SR_MODE_LOW_COST,
            .afe_perferred_core = 0,
            .afe_perferred_priority = 5,
            .afe_ringbuf_size = 50,
            .memory_alloc_mode = AFE_MEMORY_ALLOC_MO
            .agc_mode = AFE_MN_PEAK_AGC_MODE_2,
```

```
            .pcm_config = {3, 2, 1, 16000},
            .debug_init = false,
            .debug_hook = {{AFE_DEBUG_HOOK_MASE_TASK
                {AFE_DEBUG_HOOK_FETCH_TASK_IN, nullp
        };
    } // function end
} // namespace end
```

There are a lot of fields in the `afe_config_t` structure and most of them are self-explanatory. For example, AFE features **Voice Activity Detection** (**VAD**) to detect the presence of human speech, **Automatic Gain Control** (**AGC**) for the peak audio amplitude control, and **Pulse-Code Modulation** (**PCM**) configuration to define the audio sampling. We have the `AppSpeech` class ready and we can instantiate it in `main/speech_ex.cpp`:

```
#include "AppSpeech.hpp"
namespace
{
    app::AppSpeech m_app_sp;
}
extern "C" void app_main()
{
    m_app_sp.start();
}
```

Basically, there is nothing much to do in the `app_main` function. We only call the `start` function of the `m_app_sp` object to start voice

command detection. The application is ready to test. Let's flash the devkit and see how it works.

Testing the application

It is time to test the application. As usual, we will use the `idf.py` tool for that, as follows:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting...
Detecting chip type... ESP32-S3
<logs deleted>
I (423) es7243e_set_gain: gain:0.000000
I (423) Adev_Codec: Open codec device OK
I (423) es7243e_set_gain: gain:30.000000
I (423) gpio: GPIO[46]| InputEn: 0| OutputEn: 1| Ope
I (423) Adev_Codec: Open codec device OK
I (433) MODEL_LOADER: The storage free size is 15744
I (433) MODEL_LOADER: The partition size is 5168 KB
I (433) MODEL_LOADER: Successfully map model partiti
I (433) AFE_SR: afe interface for speech recognition
I (433) AFE_SR: AFE version: SR_V220727
I (433) AFE_SR: Initial auido front-end, total chann
I (433) AFE_SR: aec_init: 0, se_init: 1, vad_init: 1
I (433) AFE_SR: wakenet_init: 1
MC Quantized wakenet9: wakeNet9_v1h24_hiesp_3_0.63_0
I (633) AFE_SR: wake num: 3, mode: 0, (Jun 2 2023 11
Quantized8 Multinet5: MN5Q8_v2_english_8_0.9_0.90, b
I (833) MN_COMMAND: ----------------------SPEECH COMM
```

```
I (833) MN_COMMAND: Command ID0, phrase ID0: Ptk nN
I (833) MN_COMMAND: Command ID1, phrase ID1: Ptk eF
I (833) MN_COMMAND: Command ID2, phrase ID2: SfT KcL
I (833) MN_COMMAND: Command ID3, phrase ID3: SfT KcL
I (833) MN_COMMAND: Command ID4, phrase ID4: SfT KcL
I (833) MN_COMMAND: Command ID5, phrase ID5: hicST V
I (833) MN_COMMAND: Command ID6, phrase ID6: LbcST V
<logs deleted>
I (843) speech: waiting wake-word
I (4243) speech: waiting command          <--------
I (7853) speech: detection results:
I (7853) speech: - cmd:2 prob:0.281997  <-------- No
I (10923) speech: detection results:
I (10923) speech: - cmd:2 prob:0.337729
I (13993) speech: detection results:
I (13993) speech: - cmd:4 prob:0.794267 <-------- CM
I (28593) speech: detection results:
I (28593) speech: - cmd:1 prob:0.555508 <-------- CM
I (34663) speech: waiting wake-word       <--------
I (41423) speech: waiting command         <--------
I (44973) speech: detection results:
I (44973) speech: - cmd:3 prob:0.603908
```

The following figure shows the devkit when the “set color blue” `(CMD_BLUE)` voice command is detected:

Figure 10.6: ESP32-S3-BOX-Lite with RGB LED

The application detects the wake-word and then the voice commands. As we mentioned at the beginning of the `AppSpeech` class implementation, they are defined in `sdkconfig` and ready to be used directly in the application, but let's see the SR configuration in `sdkconfig`:

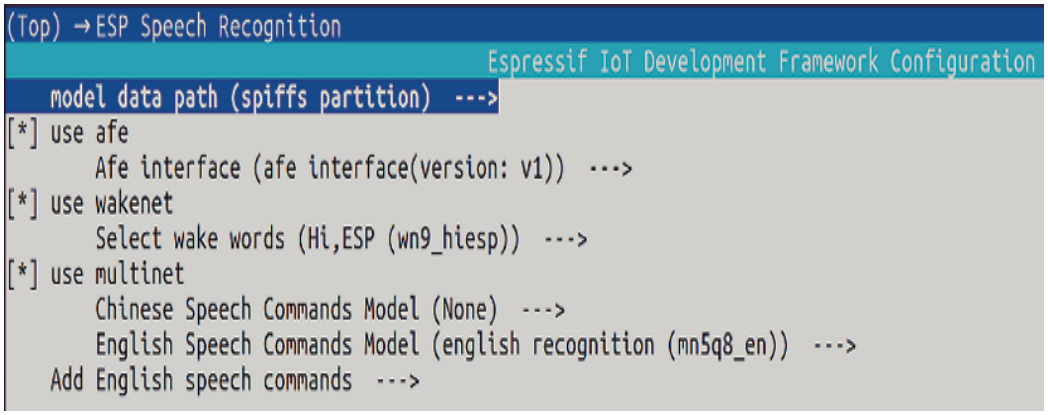


Figure 10.7: SR configuration

The *WakeNet* and *MultiNet* models are selected in the configuration. The interesting option is **Add English speech commands**, as seen in the following figure:

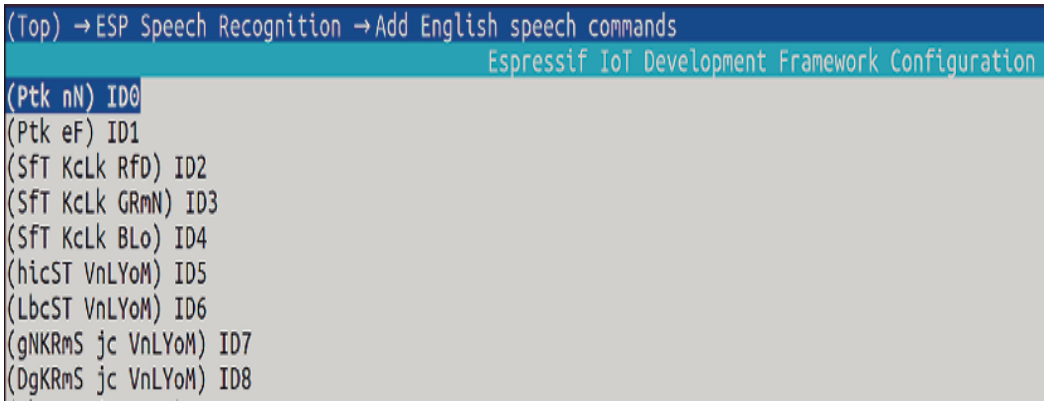


Figure 10.8: Speech commands

We handle the first five commands (**ID0** to **ID4**) from this list in the application. The entries show the phonemes of the voice commands. A phoneme is used to distinguish one word from another. When we want to add new commands, we update this list. The tool to generate phonemes comes with the ESP-SR framework. The following example shows its usage:

```
$ cd <book_clone>/ch10/components/esp-skainet/compon
# Install the Python modules
$ pip install pandas g2p_en
# Generate new phonemes
$ python ./multinet_g2p.py -t "set color yellow;set
in: set color yellow;set color purple
out: SfT KcLk YfLb;SfT KcLk PkPcL;
```

In the above example, we generated the phonemes of new set-color commands for yellow and purple. We can add these new commands in `sdkconfig` and update the application to handle them.

Developing voice applications with Espressif's frameworks is fairly easy. When we have a custom board, the only thing to do is to develop the **board support package** (**BSP**) for it and integrate the BSP with the AFE-SR framework.

Troubleshooting

Here are some tips that can help during development and tests:

- The project's `sdkconfig` file already has the correct configuration values for ESP32-S3-BOX-Lite. However, if you want to run the example on another Espressif board, you need to configure it:

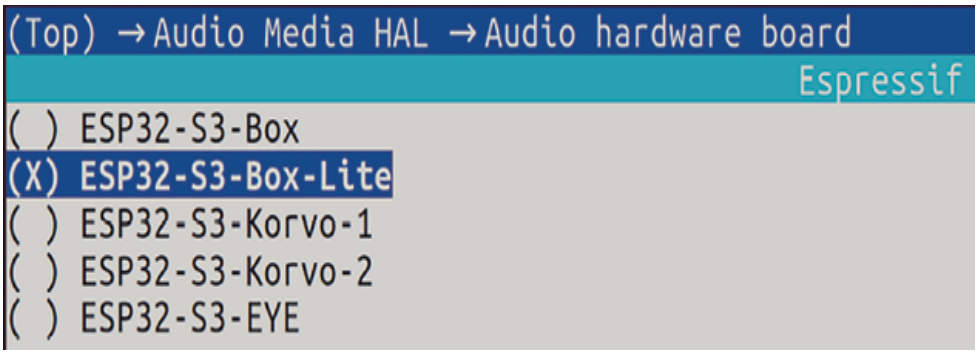


Figure 10.9: Selecting a board

- If you encounter a serial port error while flashing the application, erasing the flash memory may help. One way to do that is to attach the devkit over USB with its USB+/- pins on PMOD1 and use `openocd`:

```
$ openocd -f board/esp32s3-builtin.cfg -c "init;
```

- The configured wake-word is "*Hi ESP*". The WakeNet engine detects this wake-word more easily when the P at the end is emphasized. While testing the voice commands, you can check the detection probabilities on the serial output and adapt the utterance accordingly.

Summary

Advances in MCUs and AI/ML algorithms have made it possible to have models small enough to be run on IoT devices. **tinyML** enables us to develop ML applications that can make inferences on data coming from sensors right in the field without the need for a round-trip to a backend system or cloud. There are several frameworks that

we can employ for this purpose. **TensorFlow** is a platform that provides all the necessary tools and libraries to develop ML applications.

Data collection, ML model design and optimization, and inference are the stages in the **tinyML pipeline**. **TensorFlow Lite for Microcontrollers** (**TFLM**) is the framework in TensorFlow to optimize and use ML models on constrained IoT devices. We developed a simple tinyML application that uses a sine-wave model to make inferences and draws its predictions on the screen. Espressif empowers us with two models to develop voice applications: **WakeNet** and **MultiNet**. We learned how to use these models in an application that activates with a wake-word and then detects voice commands.

In the next chapter, we will learn about another important ML platform, Edge Impulse. It specializes in IoT devices and provides **ML operations** (**MLOps**) features to manage models in production.

Questions

Here are some questions to review what we have learned in the chapter:

1. Which one of the following is not true about machine learning?
    a. Supervised learning needs labeled data to train models.
    b. Unsupervised learning tries to find outliers in data.
    c. The agent in reinforced learning interacts with the environment to learn.

d. Reinforced learning is superior to others at detecting patterns in data.

2. Which one of the following is not a step in the tinyML pipeline?

    a. Data collection and pre-processing
    b. Training the model on an IoT device
    c. Optimizing the model for deployment
    d. Running inference on an IoT device

3. Which technique makes an ML model small enough to fit into the memory of an IoT device?

    a. Training
    b. Quantization
    c. Overfitting
    d. Validation

4. With TFLM, we can:

    a. Optimize a TensorFlow model for IoT devices
    b. Train a TensorFlow model
    c. Pre-process data before model training
    d. Manage data versions

5. Which Espressif ML model can we use to detect voice commands?

    a. MNIST
    b. GPT-4
    c. WakeNet
    d. MultiNet

Further reading

ML is a huge subject to learn about. The following books can be helpful on this journey:

- *TinyML Cookbook*, Gian Marco Iodice, Packt Publishing (https://www.packtpub.com/product/tinyml-cookbook/9781801814973): Discusses tinyML on Arduino and Raspberry Pi with examples. *Chapter 3* shows the complete ML pipeline in an example, starting from preparing a dataset, to training a model, quantization, and inference.
- *Deep Learning with TensorFlow and Keras*, Amita Kapoor, Antonio Gulli, Sujit Pal (https://www.packtpub.com/product/deep-learning-with-tensorflow-and-keras-third-edition/9781803232911): Explains deep learning. A must-read book to design effective NN models.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

11

Developing on Edge Impulse

As we talked about in the previous chapter, developing a TinyML application requires many steps, such as collecting requirements and data, pre-processing data, model development and optimization, deployment, performance tracking, and maintenance. In a traditional software project, we have DevOps to manage the software life cycle. When it comes to machine learning, we define **Machine Learning Operations** or **MLOps**. An **MLOps platform** provides us with the tools and resources to design, develop, and maintain our machine learning applications. What makes an MLOps platform different from a DevOps platform is that it also manages data and the resulting models in its versioning subsystem.

Edge Impulse is the leading MLOps platform for TinyML. It helps at every step of the ML application development process. Some important features of Edge Impulse are the following:

- A web-based development environment, **Edge Impulse Studio**, to manage the entire ML life cycle, starting from data acquisition to designing and testing ML models, including versioning.
- A RESTful API to access the Edge Impulse Studio functionality and projects remotely.

- Device SDKs for data acquisition from devices.
- Inference SDK to run models on IoT devices.
- A command-line interface to manage devices that don't have a direct internet connection.

In this chapter, we will see how to use Edge Impulse Studio to clone an existing ML project, generate the model from it, and run the model on ESP32. The topics of the chapter are:

- An overview of Edge Impulse
- Cloning an Edge Impulse project
- Using the ML model on ESP32
- Next steps for TinyML development

Technical requirements

The hardware requirements of the chapter are:

- ESP32-S3 Box Lite
- An RGB LED
- A 440Ω resistor

An account on the Edge Impulse platform is needed. You can create your account at this link: https://studio.edgeimpulse.com/login.

You can find the example of the chapter here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch11.

Let's have a quick overview of the Edge Impulse platform first.

An overview of Edge Impulse

The Edge Impulse platform has distinct features to support TinyML. First of all, we have an option to collect data directly from a sensor device. **Data Forwarder** is a tool that comes with the platform. With suitable firmware on the device, Data Forwarder retrieves data from the sensor over a serial connection and sends it to the platform. It is important because the quality of data can change between different brands of sensors, which might eventually affect the accuracy of the model. If we collect data from the device that we are going to use in the product, this can improve the final ML model.

After retrieving training data, we can design the model as an **Impulse** on the platform. An Impulse consists of different blocks: an input block, a processing block, and a learning block. The input block defines the nature of data. The processing block extracts the features that we need to train the model on. Finally, the learning block applies the machine learning algorithms to the features to create an ML model. In Edge Impulse, we also have the option of importing a pre-trained model. In this way, we can easily use a TensorFlow model or an **Open Neural Network Exchange** (**ONNX**) model that is trained outside Edge Impulse for our project.

Designing an ML model from scratch can be a challenge for someone who has less experience with ML algorithms and hyperparameter tuning, i.e., the optimization of the model to match the project requirements. An **AutoML** tool helps at this point, which aims to find the best model for a given constraint set. **EON Tuner** is the AutoML tool in Edge Impulse.

We provide EON Tuner with the data, select the target MCU, and EON Tuner returns the alternative models that we can run on our device. The alternatives are listed with their accuracy and performance indicators, such as the inference time, RAM used, and flash needed to store the model. Based on the project requirements and constraints, we can select one of the models as primary in our project.

In the deployment stage, Edge Impulse again provides many options. We can choose a binary for given target hardware, or we can download the model as a generic, portable C++ library.

In this chapter, we will develop a keyword-spotting application on ESP32-S3 Box Lite with a model that comes from Edge Impulse.

Cloning an Edge Impulse project

The Edge Impulse platform provides community projects available online to everybody. We can clone them into our accounts and develop new projects on top of them. We will use one of the public projects in this chapter as an example, which is *Edge Impulse Inc. / TinyML Summit 2021 Keywords*. Its goal is to detect the *tinyml* utterance in a model. Let's clone this example and see what Edge Impulse provides us with as an MLOps platform in the following steps:

1. Navigate to `https://www.edgeimpulse.com/projects` and use the search bar to find the **TinyML Summit 2021 Keywords** project by name. Select it from the list.

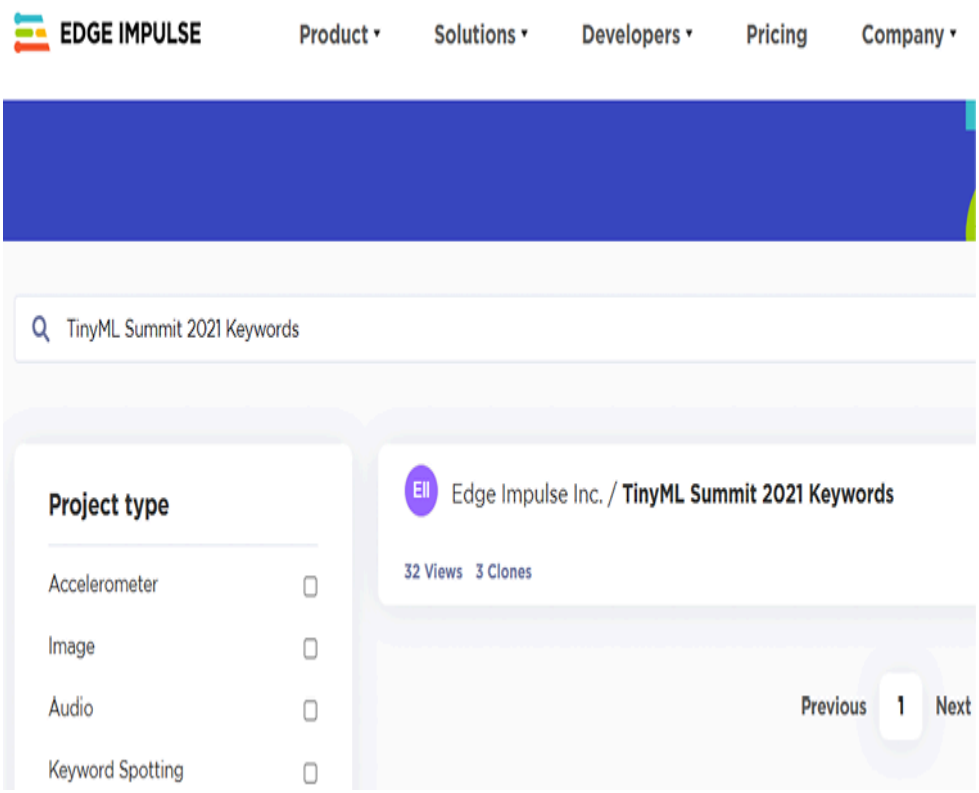


Figure 11.1: Searching the keyword-spotting example

2. Clone the project into your account by clicking on the **Clone this project** button at the top right. You can use the same name or give a new name to the project in the pop-up dialog. Then, the platform will create a duplicate of the project in your account with the given name.

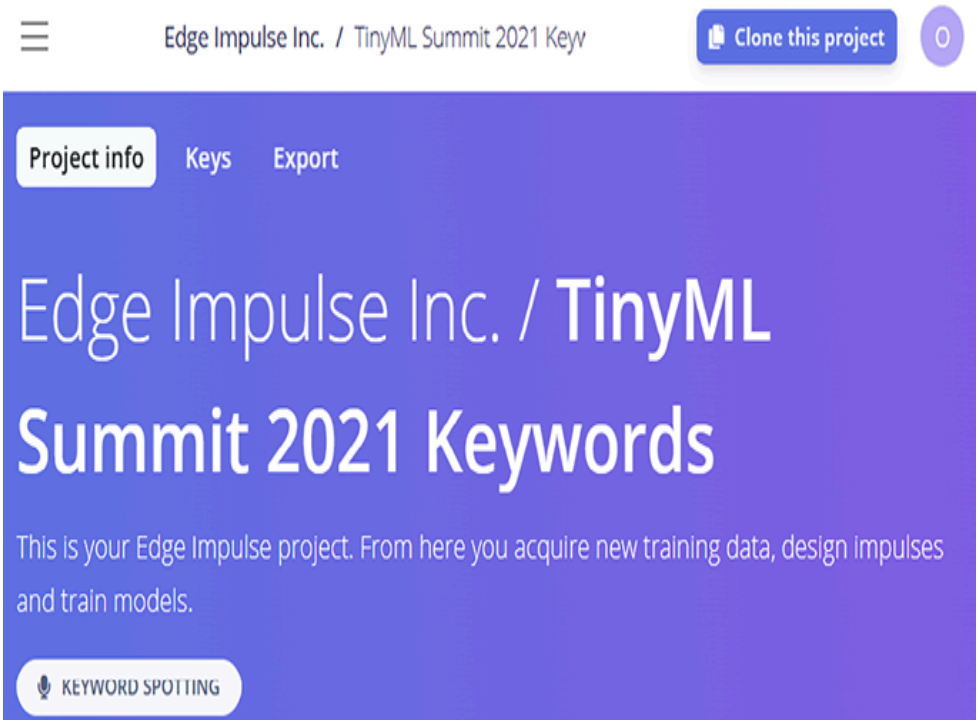


Figure 11.2: Cloning the project

3. After cloning, we can discover the project in our accounts. The dashboard shows the information about the project along with some shortcuts to basic functionality, such as collecting new data or running the model in a browser window. You can immediately test the model in your browser by clicking on the **Launch in browser** button. The browser window looks like this:

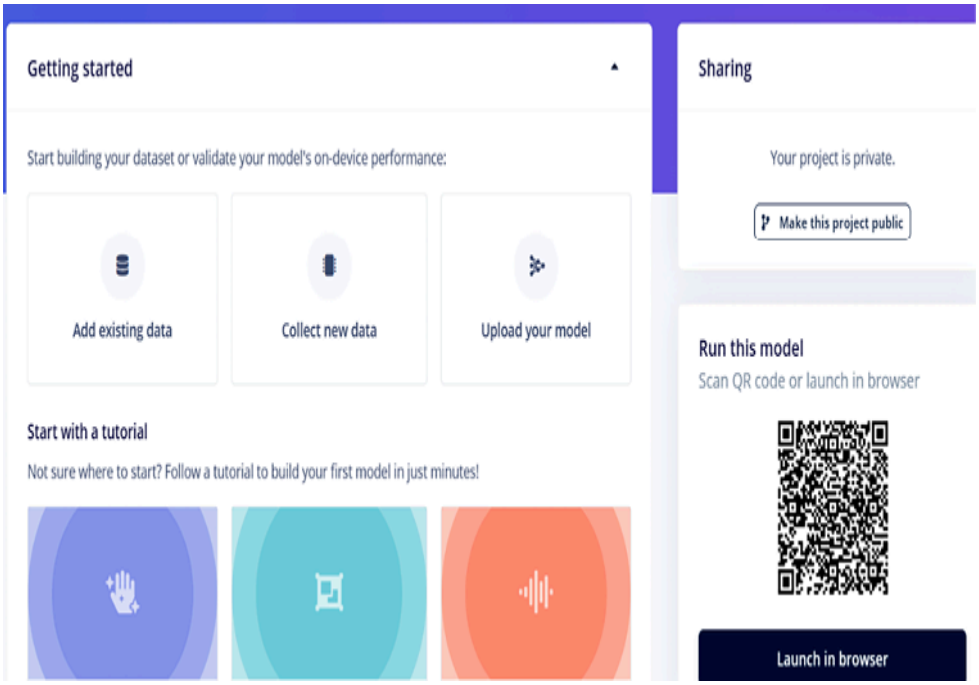


Figure 11.3: Running the model in the browser

4. Click on the **Data acquisition** option on the left menu. It shows the entire dataset that is used to train the model. We can see the total duration of the samples, the train/test split ratio, the individual samples, and their labels.

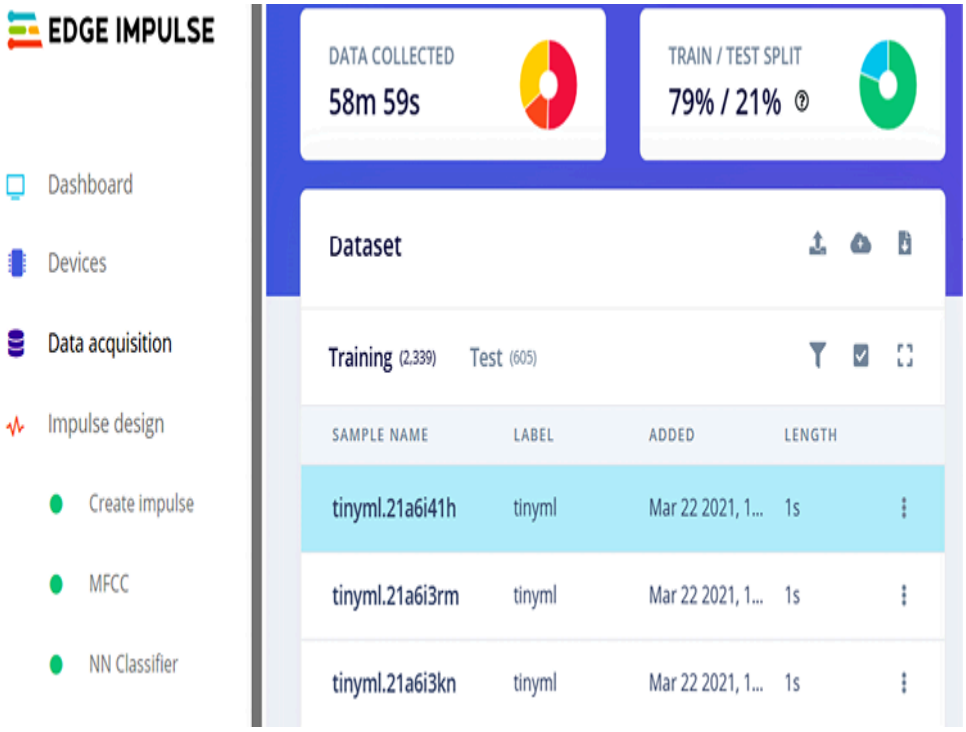


Figure 11.4: Dataset

5. We can see the impulse blocks by clicking on **Impulse design** on the left. Each sample is 16KHz audio data with a 1-second window size. In the pre-processing block, the **Mel-Frequency Cepstral Coefficient** (**MFCC**) technique is applied to extract the features to be learned. The learning block takes the features as input and classifies them into three groups – *Noise*, *Unknown*, and *tinyml*, as can be seen in the following figure:

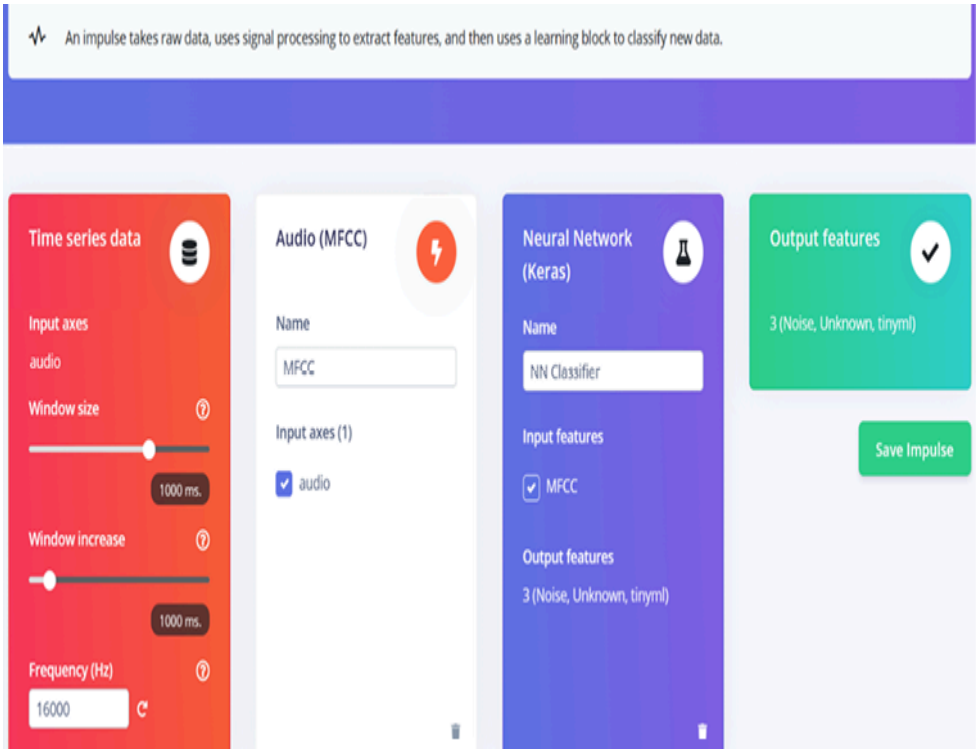


Figure 11.5: Impulse design

6. Go to **Model testing** from the left menu to observe the test results of the model on the test data. On the page, we can see the accuracy of the model, false positives, false negatives, and associated samples with them. According to the table (**confusion matrix**) in the following screenshot, the model has classified 93.9% of the *tinyml* audio samples in the *tinyml* group correctly.

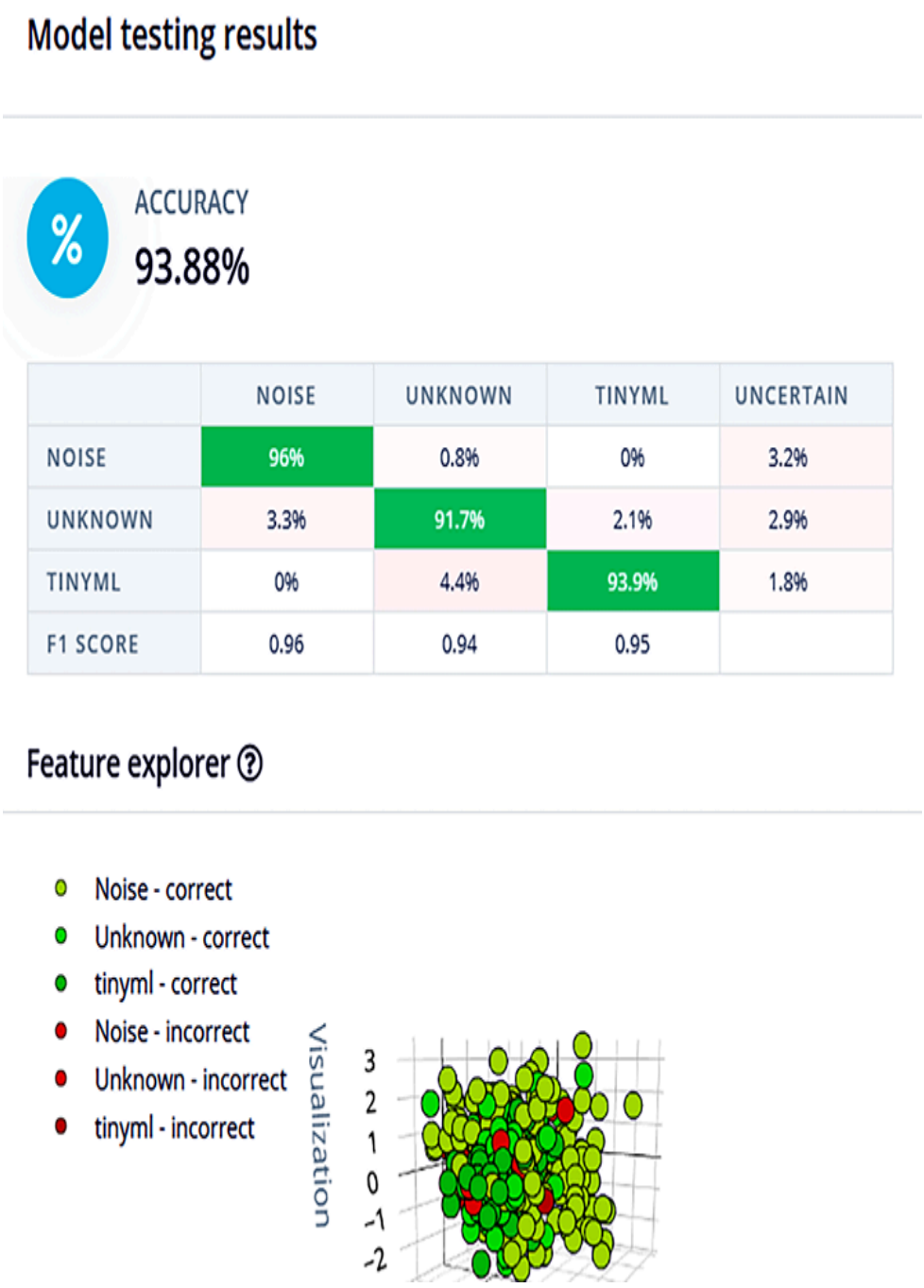


Figure 11.6: Test results

7. Click on the **Deployment** link from the left menu. On the displayed page, we configure the Edge Impulse project output according to our needs. We can choose **C++ library** to include in the application code or direct binaries from the deployment list

that Edge Impulse supports. Different from the TinyML binary models that are run in the inference engine of an ML application, the Edge Impulse C++ model library implements the model as pure C/C++ statements with the help of their **Edge Optimized Neural** (**EON**) **Compiler** when enabled on this page.

This results in more performance gains compared to any interpreted model opcodes in an inference engine. We also select the **Quantized (int8)** model to make it even smaller and faster. Before building the project, we update the target as `ESP-EYE`. This sets some of the C++ compiler options for ESP-IDF in the output, although we will still need some manual updates later to make the model library compatible with ESP32-S3. When we click on the **Build** button, the platform generates the library and downloads it as a ZIP file in the browser.

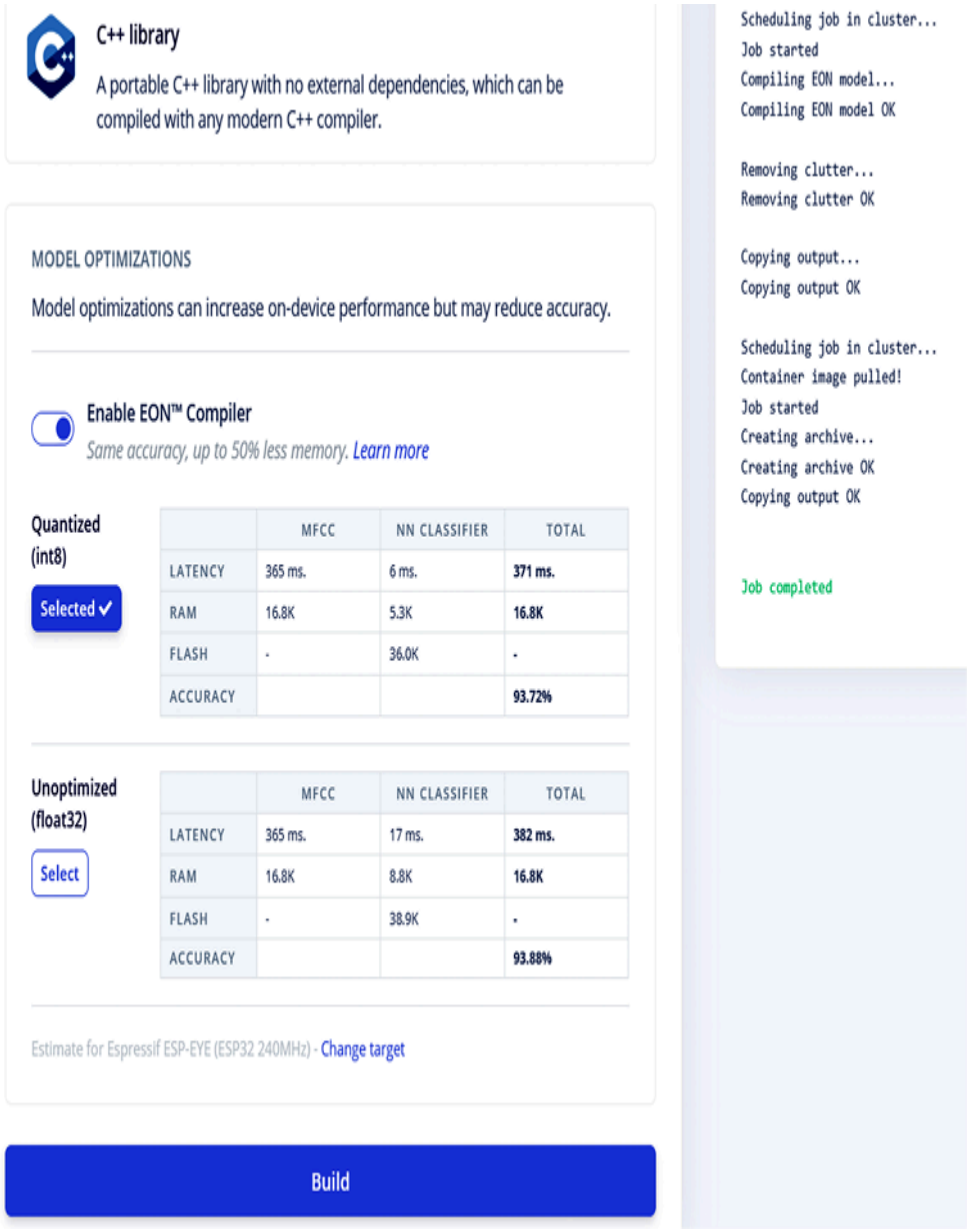


Figure 11.7: The deployment page

These are the basic steps that we take before downloading the model as a C++ library. The platform has many other features to discover. If you want to learn more about them, you can visit the online documentation at this link: https://docs.edgeimpulse.com/docs/

We can now continue with the application development in the next section.

Using the ML model on ESP32

The goal of the example is to understand how to use the downloaded C++ model to develop an ESP-IDF project and run it on ESP32-S3 Box Lite to detect the keyword *tinyml*. We will connect an RGB LED to the devkit so that the application can visually indicate the inference state: green for keyword detection, blue for no sound/background noise, and red for any sound other than the keyword.

Before moving on to the code, let's attach an RGB LED to the GP13 (red), GP12 (green), and GP11 (blue) pins of the devkit. The following Fritzing sketch shows the connections:

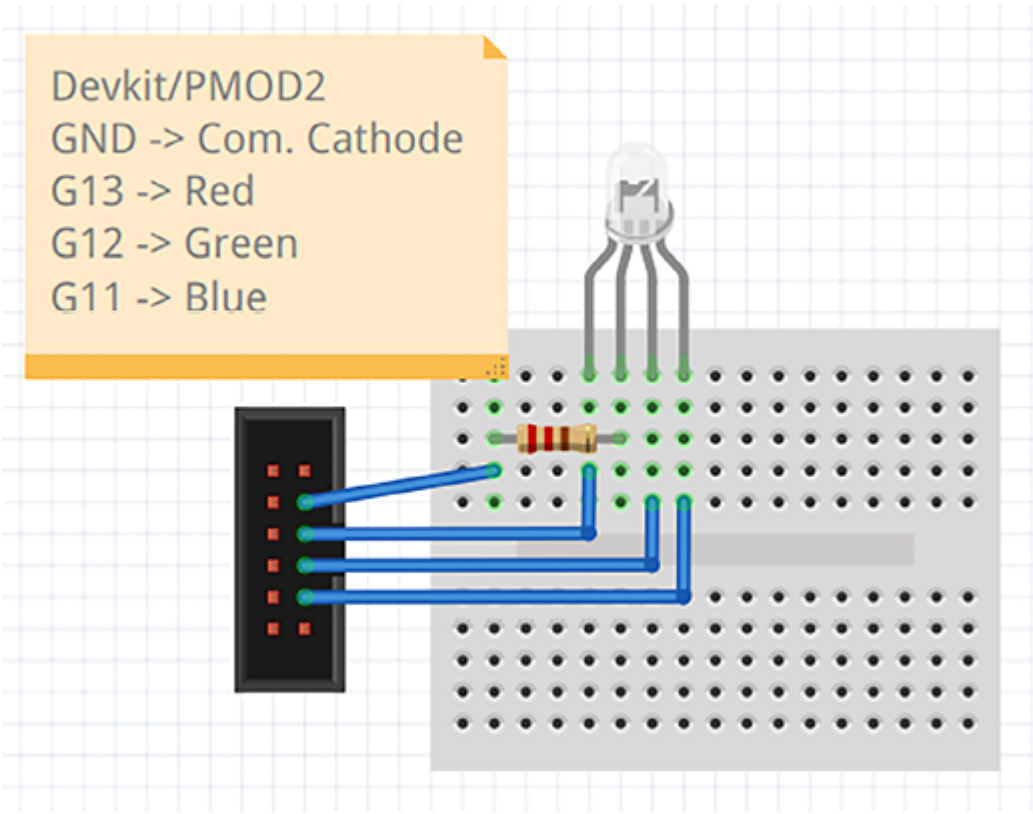


Figure 11.8: The connections between the devkit and the RGB LED

Unfortunately, our devkit is not officially supported by the Edge Impulse platform, so we should do the integration manually. However, we will skip all the integration work by using the application code from the book repository as it is in order to keep the focus on the main subject. We will only discuss the code and see how the Edge Impulse model is integrated into the IDF project, as comes next.

The model library

First, let's see what comes with the Edge Impulse C++ model library:

```
$ ls edge-impulse-sdk/ model-parameters/ tflite-mode
edge-impulse-sdk/:
classifier  cmake  CMSIS  dsp  LICENSE  LICENSE-apac
model-parameters/:
model_metadata.h  model_variables.h
tflite-model/:
tflite_learn_5_compiled.cpp  tflite_learn_5_compiled
```

There are three directories in the ZIP file that we downloaded from the Edge Impulse platform. The `edge-impulse-sdk` directory contains the SDK, hence the name. The SDK imports TensorFlow Lite and other third-party libraries as well as platform-specific code. The Espressif port uses ESP-NN (optimized neural network functions for ESP chips) under the hood. The original Edge Impulse SDK doesn't contain the ESP-NN commit that supports the ESP32-S3. Luckily, we have the latest stable branch in the book repository, which is capable of invoking ESP32-S3-specific instructions for optimized vector operations. The `model-parameters` directory has the definitions in the header files for our TinyML model. Lastly, the `tflite-model` directory keeps the source code files, which are the glue between the Edge Impulse SDK and the TensorFlow Lite library in the project. Together with the header files in the `model-parameters` directory, these two directories have the source code files that define the model in pure C/C++ statements, instead of a model binary to be run in an inference engine.

The application code

The `CMakeLists.txt` file of the project root defines the other components of the project:

```
cmake_minimum_required(VERSION 3.15)
set(EXTRA_COMPONENT_DIRS
../../ch10/components/esp-skainet/components/esp_coc
../../ch10/components/esp-skainet/components/esp-dsp
../../ch10/components/esp-skainet/components/esp-sr
../../ch10/components/esp-skainet/components/hardwar
../../ch10/components/esp-skainet/components/led_str
../../ch10/components/esp-skainet/components/perf_te
../../ch10/components/esp-skainet/components/player
../../ch10/components/esp-skainet/components/sr_ring
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(edgeimpulse_ex)
idf_build_set_property(COMPILE_OPTIONS "-fdiagnostic
idf_build_set_property(COMPILE_OPTIONS "-Wno-unused-
idf_build_set_property(COMPILE_OPTIONS "-Wno-maybe-u
idf_build_set_property(COMPILE_OPTIONS "-Wno-error=f
idf_build_set_property(COMPILE_OPTIONS "-Wno-unused-
idf_build_set_property(COMPILE_OPTIONS "-Wno-error=r
idf_build_set_property(COMPILE_OPTIONS "-Wno-error=s
idf_build_set_property(COMPILE_OPTIONS "-DEI_PORTING
```

The `EXTRA_COMPONENT_DIRS` variable points to the components of the `esp-skainet` library that we used in *Chapter 10, Machine Learning with ESP32*. We need them because it implements the **board support package** (**BSP**) for ESP32-S3 Box Lite, which allows us to

use the microphone of the devkit. The compiler directive in the last line enables the Espressif port in the Edge Impulse SDK.

In the project directory, we now have these files:

```
$ ls
CMakeLists.txt  edge-impulse-sdk  main  model-parame
```

The `sdkconfig.defaults` file defines ESP32-S3 as the target chipset and enables the devkit BSP for *esp-skainet*. In the `main` directory, we have the application source code files as usual:

```
$ ls main
app_led.c  app_led.h  AppLed.hpp  AppSpeech.hpp  CMa
```

The `app_led.c`, `app_led.h`, and `AppLed.hpp` files are again copied from *Chapter 10*. They provide the abstraction for driving the RGB LED, as explained in the *Chapter 10*, *Detecting voice commands* section example, and there's no need to repeat it here. Let's go to the heart of the application and look at what we have in the `main/AppSpeech.hpp` C++ header:

```
#pragma once
#include <cstring>
#include <mutex>
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
```

```
#include "sdkconfig.h"
#include "edge-impulse-sdk/classifier/ei_run_classi
#include "esp_wn_iface.h"
#include "esp_wn_models.h"
#include "dl_lib_coefgetter_if.h"
#include "esp_afe_sr_models.h"
#include "esp_mn_iface.h"
#include "esp_mn_models.h"
#include "esp_board_init.h"
#include "model_path.h"
```

We start with including a bunch of header files. The most notable one is probably `edge-impulse-sdk/classifier/ei_run_classifier.h`. It is the entry point to the Edge Impulse SDK and contains all definitions necessary to run keyword spotting inference. Then we define the indexes for the classifier categories:

```
#define NOISE_IDX 0
#define UNKNOWN_IDX 1
#define TINYML_IDX 2
```

We have three categories, which are defined as a string array in the `model-parameters/model_variables.h` file, as it comes with the generated model. The Edge Impulse SDK refers to this array when returning an inference result. Similarly, we will refer to this array with the indexes defined above. We continue with a C structure definition in the `app` namespace:

```
namespace app
{
    struct AppSpeechParam
    {
        std::function<void(void)> noise_fn;
        std::function<void(void)> unknown_fn;
        std::function<void(void)> tinyml_fn;
    };
```

We will use the `AppSpeechParam` struct when we initialize the `AppSpeech` class. There are three callback functions to be passed to the initialization function and the class will call one of them after running inference according to the result. Next comes the `class` definition:

```
    class AppSpeech
    {
    private:
        static constexpr int AUDIO_BUFFER_SIZE{16000
        int m_audio_buffer_ptr{0};
        float m_audio_buffer[AUDIO_BUFFER_SIZE];
        float m_features[AUDIO_BUFFER_SIZE];
        std::mutex m_features_mutex;
```

In the `private` section of the class, we start with the buffers. We have two buffers: one for collecting data from the mic (`m_audio_buffer_ptr`) and another one for inference (`m_features`). Both of them have a size of `16000`, compatible with

the model. The `m_features_mutex` variable is a mutex to protect the `m_audio_buffer_ptr` buffer from concurrent access by different FreeRTOS tasks in the application. There are several other private member variables, as follows:

```
        esp_afe_sr_iface_t *m_afe_handle{nullptr};
        esp_afe_sr_data_t *m_afe_data{nullptr};
        AppSpeechParam m_callbacks;
```

The `m_afe_handle` variable is the pointer to access Espressif's **audio frontend** (**AFE**) interface and the `m_afe_data` variable is for accessing audio data that comes from the AFE. `m_callbacks` holds the callback functions provided by the clients of the `AppSpeech` class. We have some private member functions not listed here, but before them, let's jump to the `public` section and see what `AppSpeech` provides its clients as an interface:

```
    public:
        void init(AppSpeechParam cbs)
        {
            m_callbacks = cbs;
            esp_board_init(AUDIO_HAL_16K_SAMPLES, 1
```

The `init` function takes a parameter for the callbacks and stores it in the `m_callbacks` member variable. We initialize the devkit by calling the `esp_board_init` function. The first parameter indicates that the audio HAL is configured for 16KHz sampling, again

compatible with the model configuration. We will initialize other member variables next:

```
            m_afe_handle = const_cast<esp_afe_sr_ifa
                *>(&ESP_AFE_VC_HANDLE);
            afe_config_t afe_config = defaultAfeCon
            m_afe_data = m_afe_handle->create_from_
                &afe_config);
        }
```

The `m_afe_handle` pointer points to the `ESP_AFE_VC_HANDLE` handle since we will only use the voice communication functionality of the AFE. The other option would be `ESP_AFE_SR_HANDLE` if we were to employ the speech recognition framework of ESP-Skainet, as we did in *Chapter 10, Machine Learning with ESP32*, but this time, it is Edge Impulse that we are going to use for inference. Then, we create an AFE configuration and underlying data structure with a pointer to it, `m_afe_data`. The next public function is `start`:

```
        void start(void)
        {
            xTaskCreatePinnedToCore(feedTask, "feed"
                this, 5, nullptr, 0);
            xTaskCreatePinnedToCore(detectTask, "det
                8 * 1024, this, 5, nullptr, 1);
            xTaskCreatePinnedToCore(actionTask, "act
                8 * 1024, this, 5, nullptr, 1);
        }
```

```
    }; // end of class
} // end of namespace
```

We need three FreeRTOS tasks, which we create in the `start` function. The `feedTask` function is a private member function of the `AppSpeech` class and generates audio samples from the mic. The `detectTask` function retrieves the raw audio data from the AFE and copies it to the local buffers, and finally, the `actionTask` function uses the data in the local buffers to run inference on it with the help of the Edge Impulse SDK. Please note that we pass the `this` pointer to the task functions as a parameter so that we can access the class instance inside them. Let's look at those task functions in the private section of the class next:

```
        static void feedTask(void *arg)
        {
            AppSpeech *obj{static_cast<AppSpeech *>(
            esp_afe_sr_data_t *afe_data{obj->m_afe_d
            esp_afe_sr_iface_t *afe_handle{obj->m_af
```

In the `feedTask` function, we first obtain the AFE pointers from the class instance. Then we create a local buffer to store audio samples, as follows:

```
            int audio_chunksize = afe_handle->
                get_feed_chunksize(afe_data);
            int feed_channel = esp_get_feed_channel(
```

```
            int16_t *i2s_buff = static_cast<
                int16_t *>(malloc(audio_chunksize *
                  sizeof(int16_t) * feed_channel));
```

The audio sample type is `int16_t`, meaning that each sample is represented as a 16-bit signed integer. We create a local buffer, `i2s_buff`, to store the data coming from the mic. Next comes the task loop:

```
            while(true)
            {
                esp_get_feed_data(false, i2s_buff, a
                  * sizeof(int16_t) * feed_channel);
                afe_handle->feed(afe_data, i2s_buff)
            }
        }
```

In the task loop, we store the audio data in the local buffer and then call the `feed` function of the AFE to update the AFE internals with the new data. The next private function is `detectTask`:

```
        static void detectTask(void *arg)
        {
            AppSpeech *obj{static_cast<AppSpeech *>(
            esp_afe_sr_data_t *afe_data{obj->m_afe_d
            esp_afe_sr_iface_t *afe_handle{obj->m_af
            int afe_chunksize = afe_handle-
                >get_fetch_chunksize(afe_data);
```

```
int16_t *buff = static_cast<int16_t *>(
    malloc(afe_chunksize * sizeof(int16_
```

The `detectTask` function again starts with getting the AFE pointers from the class instance and creating a local buffer to store the pre-processed audio data from the AFE. The audio data is captured by the two microphones of the devkit. The AFE library applies some intelligent algorithms to cancel out the echo and flatten the audio data coming from the underlying audio channels. In the task loop, we will use this clear audio data after the AFE pre-processing:

```
while (true)
{
    afe_fetch_result_t *res = afe_handle
        fetch(afe_data);
    if (res == nullptr || res->ret_value
    {
        continue;
    }
```

We first fetch the AFE result in the task loop. If there is an error, we simply skip the current run of the loop. The next thing to do is to copy the audio data from the internal AFE structures into the class buffer:

```
memcpy(buff, res->data, afe_chunksiz
    sizeof(int16_t));
```

```
                for (int i = 0; i < afe_chunksize; -
                {
                    obj->m_audio_buffer_ptr %= AUDI(
                    obj->m_audio_buffer[obj->
                        m_audio_buffer_ptr++] = buf
                }
```

The audio data resides in `res->data`. We copy it to the local buffer, and then to the class buffer, `m_audio_buffer`. The `m_audio_buffer` array is basically a ring buffer to keep the last 16,000 audio samples:

```
                {
                    std::lock_guard<std::mutex> guar
                        obj->m_features_mutex);
                    for (int i = 0; i < AUDIO_BUFFE
                    {
                        obj->m_features[i] = obj->
                          m_audio_buffer[(obj->
                          m_audio_buffer_ptr + i) %
                          AUDIO_BUFFER_SIZE];
                    }
                } // end of scope
            } // end of loop
        } // end of function
```

We have another class buffer, `m_features`. The purpose of this buffer is to flatten the ring buffer, `m_audio_buffer`, into the time-

ordered data points that can be used as input to the inference function. Since we have two different tasks that access the `m_features` buffer, we protect it with a mutex.

This completes the `detectTask` function, and we can move on to the `actionTask` function, where we run inference:

```
        static void actionTask(void *arg)
        {
            AppSpeech *obj{static_cast<AppSpeech *>(
            ei_impulse_result_t result = {nullptr};
```

We start the `actionTask` function implementation by defining the local variables. The `ei_impulse_result_t` type holds information about the result of an inference. Next, we define a lambda function:

```
            auto get_data_fn = [&obj](size_t offset,
                size_t length, float *out_ptr) -> in
            {
                memcpy(out_ptr, obj->m_features + of
                    length * sizeof(float));
                return 0;
            };
```

The `get_data_fn` function object will be a part of another structure to be passed to the inference function. Its purpose is to provide data to the inference function. Then we continue with the task loop:

```
            while (true)
            {
                signal_t features_signal{get_data_fr
                    AUDIO_BUFFER_SIZE};
                int max_idx = NOISE_IDX;
```

The `features_signal` variable keeps information about how to pass the audio data to the inference function. The inference function will use the information to retrieve data from the application. We have another variable, `max_idx`, which will show the top match after the inference – one of the three categories that come with the model. Next, we run the Edge Impulse classifier – that is, the inference operation:

```
                {
                    std::lock_guard<std::mutex> guar
                        obj->m_features_mutex);
                    if (run_classifier(&features_sig
                        == EI_IMPULSE_OK)
                    {
                        for (uint16_t i = 0; i <
                            EI_CLASSIFIER_LABEL_COUN
                        {
                            if (result.classificatio
                               result.classificatior
                                .value)
                            {
                                max_idx = i;
                            }
```

```
                    }
                }
            }
```

The inference function of the Edge Impulse SDK is `run_classifier`. It uses the lambda function to read the audio data and returns the result in the `result` variable. If there is no error after the function call, we find the maximum match value in the `result` variable and set the `max_idx` variable accordingly. We will run a callback function for the match as follows:

```
            switch (max_idx)
            {
            case NOISE_IDX:
                obj->m_callbacks.noise_fn();
                break;
            case UNKNOWN_IDX:
                obj->m_callbacks.unknown_fn();
                break;
            case TINYML_IDX:
                obj->m_callbacks.tinyml_fn();
                break;
            default:
                break;
            } // end of switch
            vTaskDelay(pdMS_TO_TICKS(1000));
        } // end of loop
    } // end of function
```

There is a simple `switch` statement to select the right callback. We run the callback function and wait for a second for the next loop of inference. The last private function is `defaultAfeConfig`, as follows:

```
        static afe_config_t defaultAfeConfig()
        {
            return {
                .aec_init = false,
                .se_init = true,
                .vad_init = false,
                .wakenet_init = false,
                .voice_communication_init = true,
                .voice_communication_agc_init = fals
                .voice_communication_agc_gain = 15,
                .vad_mode = VAD_MODE_3,
                .wakenet_model_name = nullptr,
                .wakenet_mode = DET_MODE_2CH_90,
                .afe_mode = SR_MODE_LOW_COST,
                .afe_perferred_core = 0,
                .afe_perferred_priority = 5,
                .afe_ringbuf_size = 50,
                .memory_alloc_mode = AFE_MEMORY_ALL
                .agc_mode = AFE_MN_PEAK_AGC_MODE_2,
                .pcm_config = {3, 2, 1, 16000},
                .debug_init = false,
                .debug_hook = {{AFE_DEBUG_HOOK_MASE_
                    nullptr}, {AFE_DEBUG_HOOK_FETCH
                    nullptr}},
```

```
            };
        }
```

The `defaultAfeConfig` function simply returns the configuration for the AFE. We don't use Espressif WakeNet but only the voice communication subset of the framework. The `pcm_config` field shows the number of channels and sampling rate. This completes the `AppSpeech` class, and we can integrate the pieces in the application's `app_main` function, which is implemented in `main/main.cpp`. Let's look at it:

```
#include "AppLed.hpp"
#include "AppSpeech.hpp"
namespace
{
    app::AppLed app_led;
    app::AppSpeech app_speech;
}
```

We include the class headers and create the instances of `AppLed` and `AppSpeech`. Next, we develop the `app_main` function of the application:

```
extern "C" void app_main()
{
    app_speech.init({[]()
                    { app_led.setColor(app::eColor
                    []()
```

```
                        { app_led.setColor(app::eColor
                        []()
                        { app_led.setColor(app::eColor
    app_led.init();
    app_speech.start();
}
```

We initialize the `app_speech` object by passing lambda functions for each audio classification. For *noise*, we will set the RGB LED color to blue, for *unknowns*, it will be red, and lastly, a match to the *tinyml* utterance will be green.

Testing the application

The application is ready to be built and run on the devkit as follows:

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<logs removed>
I (745) AFE_VC: afe interface for voice communicatic
I (755) AFE_VC: AFE version: VC_V220727
I (755) AFE_VC: Initial auido front-end, total chanr
I (765) AFE_VC: aec_init: 0, se_init: 1, vad_init: (
```

```
I (765) AFE_VC: wakenet_init: 0, voice_communicatior
I (875) AFE_VC: mode: 0, (Jun  2 2023 11:39:44)
```

After flashing the devkit, we can see that the AFE starts successfully. It prints the interface info as voice communication and channels information on the serial output. At this point, we can test the application by saying "TinyML" and other words, and see how it responds to utterances.

Troubleshooting

If you encounter a problem while working on the project, the following checkpoints may help you to find the problem:

- Make sure your hardware setup is correct. Test your RGB LED with a multimeter and make sure it works properly. After attaching it to the devkit, you can run a simple application (maybe by integrating it with the devkit buttons) to see if the application can drive the RGB LED.
- If you decide to develop the application yourself, remember to use the *ESP-Skainet* library that comes with the book repository as it supports the devkit, ESP32-S3 Box Lite. The original library from Espressif doesn't support this devkit at the time of writing.
- The Edge Impulse SDK needs ESP-NN for the neural network optimizations and imports it into its repository. Nonetheless, the version of ESP-NN in the SDK doesn't support ESP32-S3; therefore, it needs to be updated from Espressif's repository on GitHub here: https://github.com/espressif/esp-nn. If you

are using the example from the book repository, the correct version is already included there.

We had an example of utilizing Edge Impulse to generate a TinyML model and using it in an ESP32 application. Nevertheless, this is not all there is to TinyML development. The next section gives an overview of a complete TinyML development process.

Next steps for TinyML development

In the scope of this book, we only discussed how to run inference on ESP32 by using different TinyML frameworks. However, in real-world scenarios, we need to do more. Let's review the ML development stages once more and have a short discussion of them in terms of the engineering work needed:

- **Project requirements**: A project starts with a need and requirements that list what to do in response to that need. A machine learning project is no exception for that. The requirements of an ML project usually reveal a lot about the nature of data in the project. With a requirement analysis, we can understand what data we need to collect, the sources of data, how we can collect it, any option to import external data, data versioning requirements, etc. In addition, a requirements document can have information about the performance of the output model, such as the accuracy, response time, and memory limitations. Project requirements have a direct impact on the technical design of an ML solution.
- **Data collection**: As discussed in *Chapter 10*, *Machine Learning with ESP32*, the quality of data that we use to train the model

decides the success of the entire project. Garbage in, garbage out. If we can collect data from the environment with the same sensor that the final product will have, the success rate of the model in production will be higher. Therefore, we might need to develop a separate application only for data collection from the installation environment when possible. **Data bias** is one of the biggest risks in model training. If our model doesn't learn about a type of valid data that it may encounter in the field, then it will fail to recognize it when the time comes. Therefore, data should cover all scenarios as much as possible; this is called **data diversity**. When collecting enough data from the field is not possible, we can use some **data augmentation** techniques, such as adding artificial noise in the input signal (not necessarily audio; this can be any type of data) to generate more data. ML platforms usually help with that by providing automated tools. Although this doesn't always mean improvement in the accuracy of the model, such techniques can help with data diversity. In some cases, we can use publicly available open-source datasets – for instance, **Kaggle datasets** (https://www.kaggle.com/). After logging in, you can search thousands of datasets and use one in your project.

- **Data preprocessing**: Raw data is mostly not usable directly in model training. At a minimum, we have to normalize the data samples to make them comparable to each other. For example, it doesn't help to use 16KHz and 41KHz audio data in the same set to train a model, or PNG and JPEG images with different resolutions in the same training set for image classification. Therefore, it is important to know some popular Python

libraries for such tasks, such as **NumPy**, `pandas`, `scikit-learn`, and **Matplotlib**. The R language is another option for data preprocessing. After preprocessing data for model training, we can also apply **data visualization** techniques to gain more insight and see the relations between data points.

- **Model design, training, and transfer learning**: We talked a bit about different machine problems and approaches in *Chapter 10*, *Machine Learning with ESP32*. The ML frameworks, such as **TensorFlow**, **Keras**, or **PyTorch**, provide us with the capability to design and train a model for a given problem. The ML platforms again support us in model design and training, as Edge Impulse does. We can also reuse a pre-trained model and make it more specialized for a specific task by training more with our dataset. This is called **transfer learning**. The source model in transfer learning is usually trained with a larger set of data – just enough to specialize for different purposes. We import it into our project and modify it according to the project requirements. In this way, we save time and resources. Designing and training a model is not a trivial task. It requires a deep knowledge of the problem domain and ML algorithms that can be applied. **AutoML** tools make our job much easier when employed properly. By providing some input about the nature of data and the problem, in addition to the performance constraints, they can do a very good job and generate models successfully.
- **Iterations and versioning**: One of the key areas in which we need **MLOps** is data and model versioning. We need to track our product's performance and it is impossible to do that

without versioning its components. ML platforms are again a great help at this point.

Machine learning is an exciting field in which to learn and develop. However, it covers a vast number of subjects and requires expertise to build successful products. Let's continue with some other resources that can be useful for TinyML application development.

The Netron app

This is an online model analysis tool. It shows the components in a TFLite model. For example, when we go to the Edge Impulse project that we developed in this chapter and select **Impulse design** / **NN Classifier**, we can see the neural network architecture, as in the following:

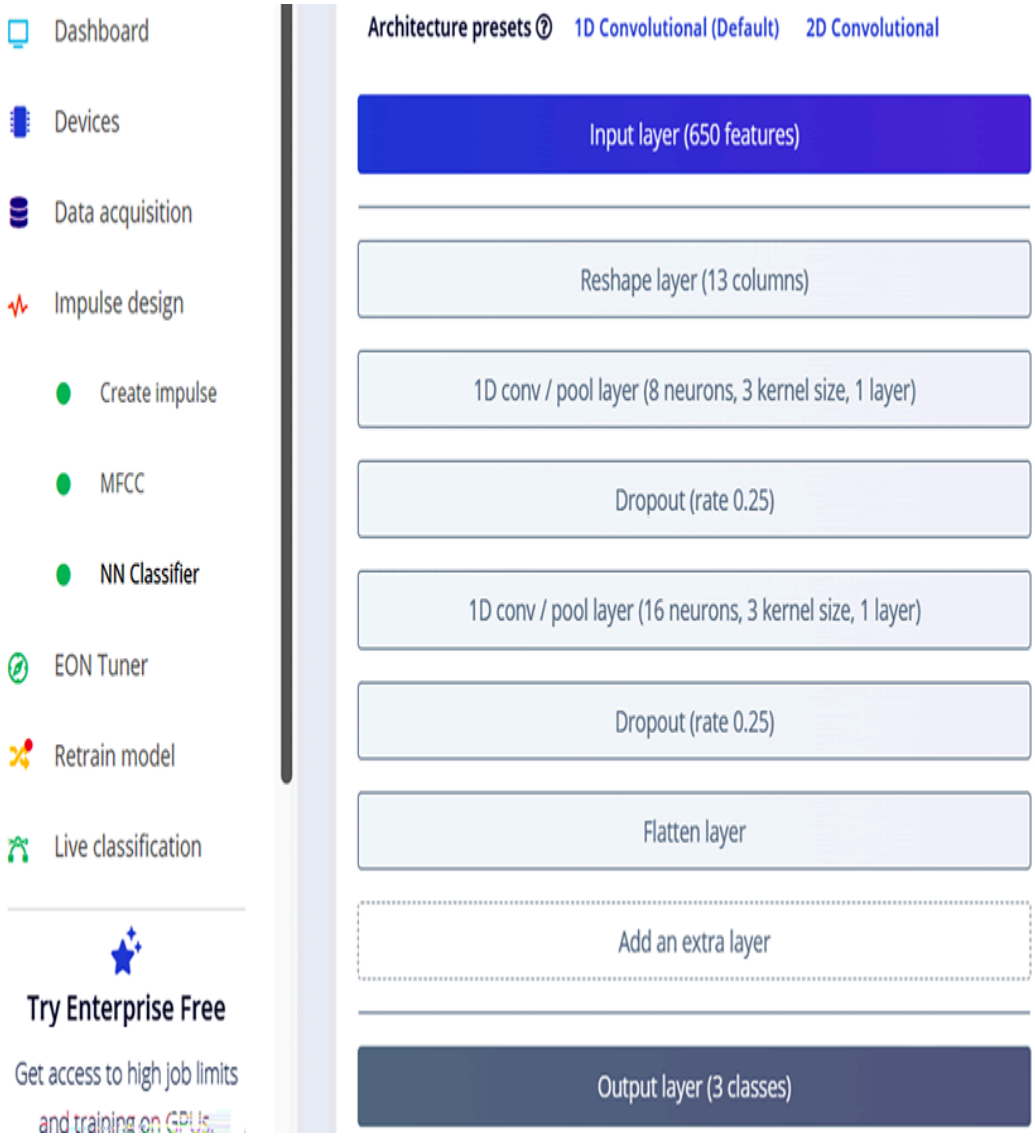


Figure 11.9: NN architecture of the example

The Netron app generates a similar analysis on a TFLite model file. Let's download the model file from the Edge Impulse project and use it for analysis with the Netron app in steps:

1. Go to the project dashboard on the Edge Impulse platform and scroll down to the **Download block output** section. Download the last option, **TensorFlow SavedModel**:

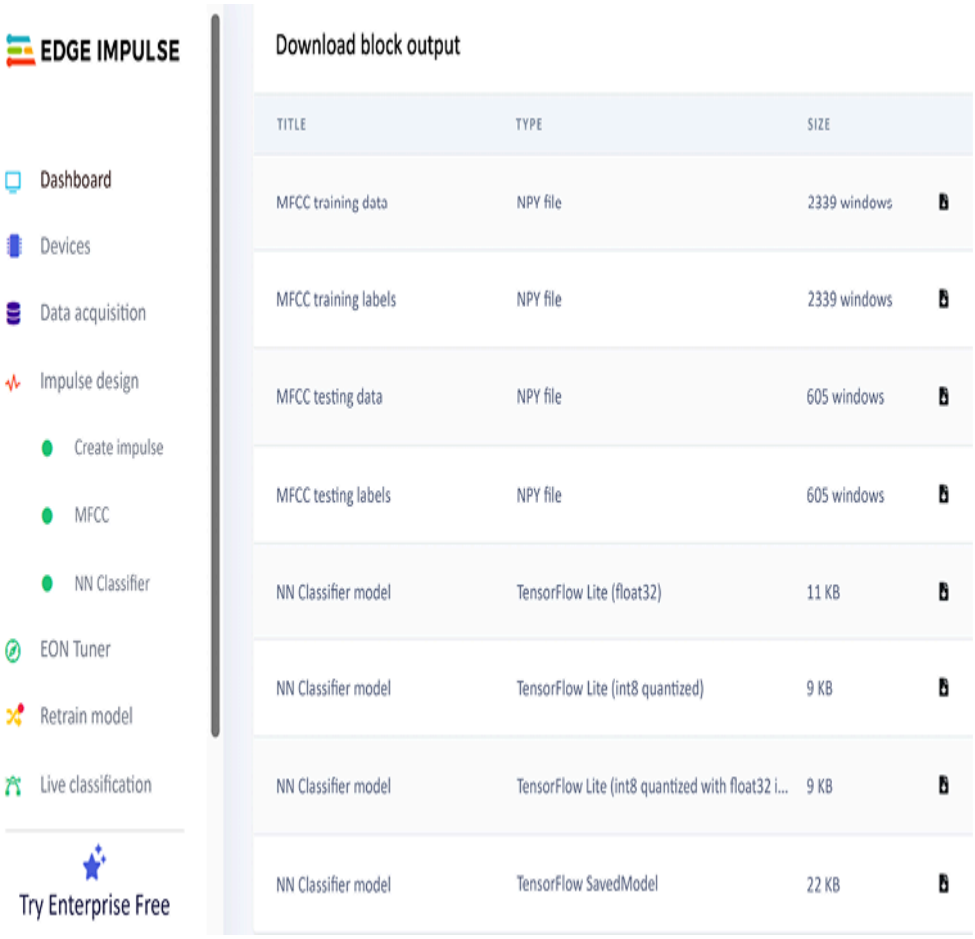


Figure 11.10: Downloading the model

2. Open a new browser window and navigate to https://netron.app/. Click on the **Open Model** button and select the ZIP file that you have just downloaded in the previous step.

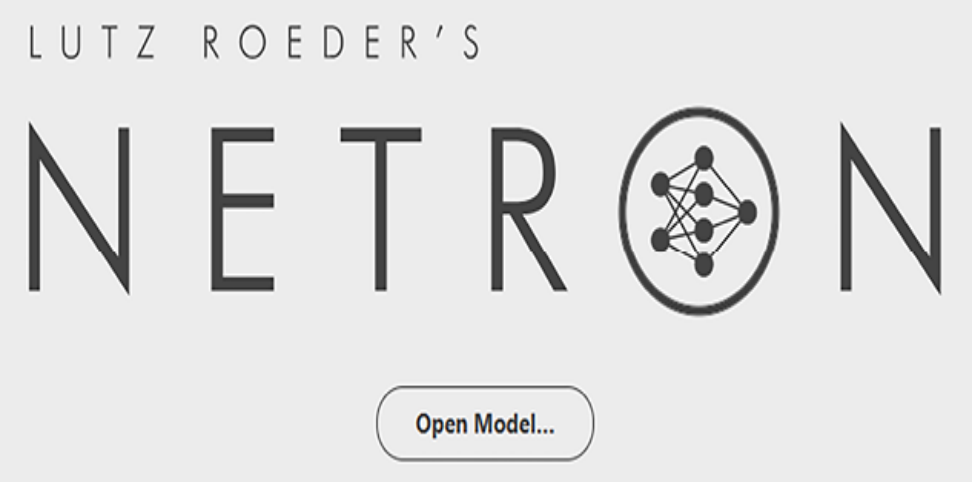


Figure 11.11: Netron app

3. The Netron app will show the entire network with the relations between the components and layers:

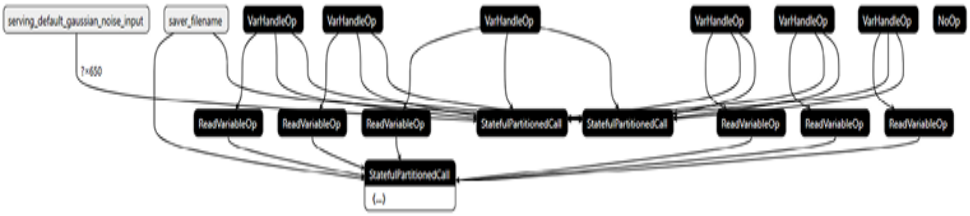


Figure 11.12: The NN graph in the Netron app

4. Click on the menu icon at the top left and select **Show Attributes**. It will add more details in the nodes:

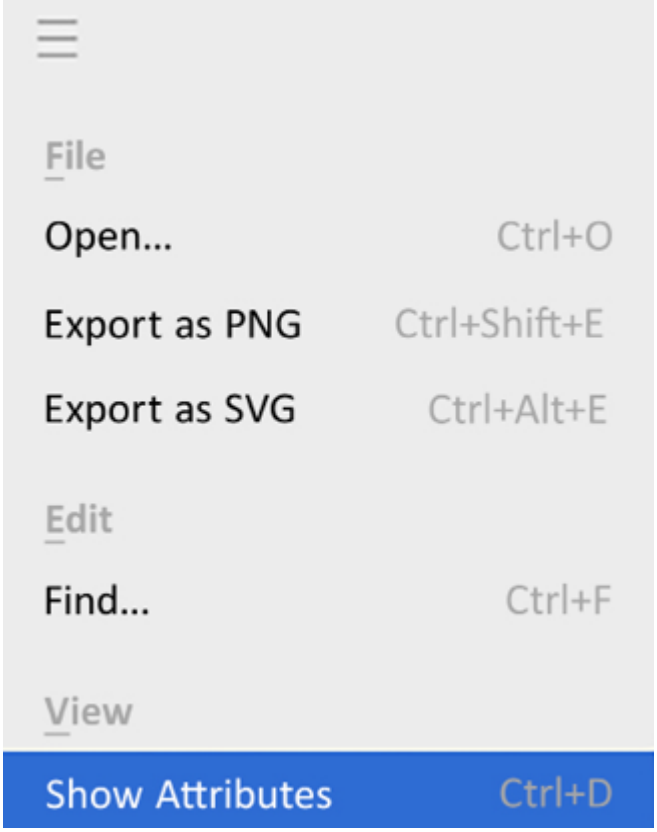


Figure 11.13: Netron app menu

5. See the first **VarHandleOp** node right after the `saver_filename` node. It corresponds to the first **Conv 1D / Pool layer** of the Impulse design on the Edge Impulse platform (*Figure 11.9*):

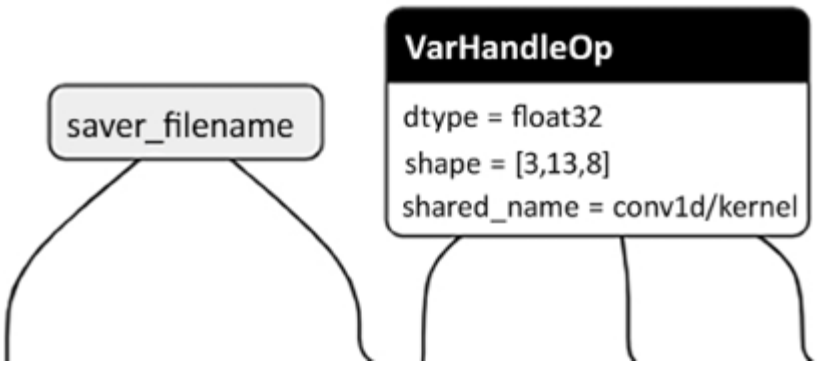


Figure 11.14: Attributes of a node

If you wonder what is going on in a TFLite model, this online tool helps with that. Ultimately, a saved model is an open-format binary file and can be visualized with such tools to investigate it more.

TinyML Foundation

tinyML Foundation is an organization that supports and brings the TinyML community together with different events. It doesn't promote any specific product or tool on the market, but all companies and people work together to increase awareness about the potential of TinyML. If you want to discover other TinyML products and services, the tinyML Foundation website is a great resource: https://www.tinyml.org/.

For example, a publication on its website (dated June 10, 2022) shows a list of AutoML products on the market. In addition to Edge Impulse/EON Tuner, it lists Neuton TinyML (https://neuton.ai/), SensiML Analytics (https://sensiml.com/services/toolkit/), Imagimob (https://www.imagimob.com/), and Qeexo AutoML (https://qeexo.tdk.com/), just to name a few. You can find the entire document here: https://www.tinyml.org/static/3eccc61369f52aaeb2d8eea64e13e6e7/State-of-the-tinyAutoML-Market.pdf.

ONNX format

ONNX itself is not a tool or platform but an open format built to represent machine learning models. The supporting partners include

almost all major technology companies. Both the Edge Impulse platform and various Espressif frameworks can use ONNX model files in ML projects. The home page of the project is here: https://onnx.ai/.

You can also find many open ONNX models available at Model Zoo here: https://github.com/onnx/models.

The best way to improve TinyML skills is to practice by developing different types of ML problems. You can find some ideas to try in the next section.

Project ideas

In this book, we focused on the audio-based TinyML projects by using the mic of ESP32-S3 Box Lite as a sensor. However, virtually any type of sensor can be used in different ML projects. Let's have a look at some project ideas that you can try with other types of sensors.

Image processing with ESP32-S3-EYE

ESP32-S3-EYE is a development board from Espressif with an OV2640 camera integrated into it. Having the ESP32-S3-WROOM-1 module at its core, it is perfectly capable of running image processing applications. More information is available on the GitHub product page: https://github.com/espressif/esp-who/blob/master/docs/en/get-started/ESP32-S3-EYE_Getting_Started_Guide.md.

Espressif also supports developers with advanced ML libraries and frameworks. ESP-DL is the deep learning library by Espressif. The following block diagram shows its architecture in very general terms:

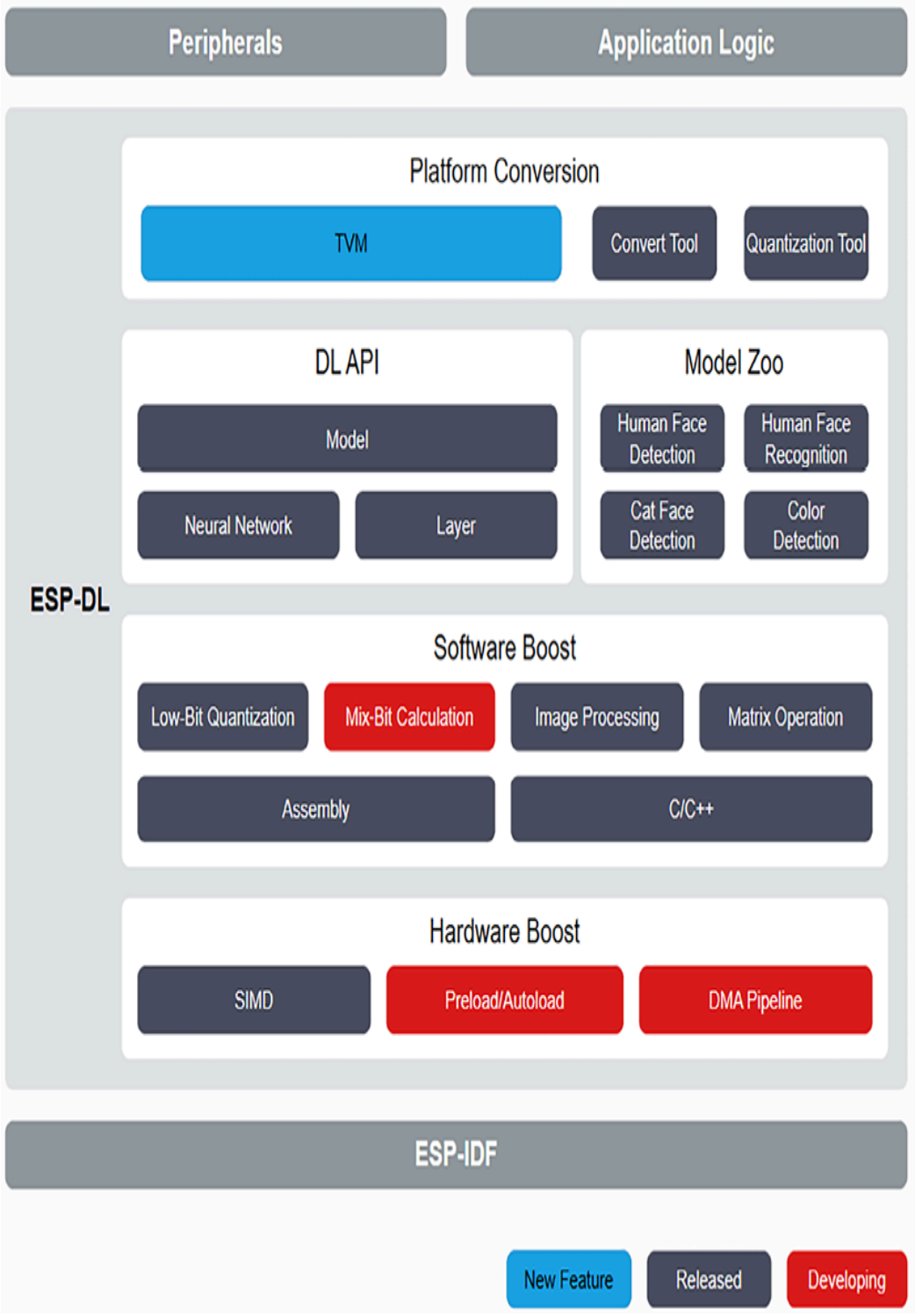


Figure 11. 15: ESP-DL (source: the official documentation)

You can find the official documentation at this link: https://docs.espressif.com/projects/esp-dl/en/latest/esp32s3/introduction.html.

The **ESP-WHO** GitHub repository has great examples for face detection and recognition here: https://github.com/espressif/esp-who.

Apart from those examples, you can try cloning and running the **Harvard University/Person Detection** project on Edge Impulse (https://studio.edgeimpulse.com/public/37001/latest). It imports the *MobileNetV1 96x96 0.25* model (transfer learning) and trains a new model with images tagged as `PERSON` or `NON_PERSON` on top of it. If you use ESP32-S3-EYE to run this example, you will need to update the downloaded SDK for ESP32-S3-EYE since it is not on the list of supported devices.

Anomaly detection

One of the problem domains that can be solved with ML is anomaly detection. It can be anything from wind turbines to a home appliance that gives a physical warning we can measure before breaking down. This means that the user can call for help from the maintenance service before the device breaks down completely, thus saving time and money.

Edge Impulse has an example for it, too. The name of the project is **Jenny Plunkett/Fan Monitoring – Advanced Anomaly Detection**. What makes it different from other examples is that it has another learning block, *anomaly detection* (*K-means*), in addition to the

classifier. The anomaly detection block gives a score to each inference so we can mark an inference result as an anomaly according to that score if it is above a threshold. The following screenshot shows the Impulse design on the Edge Impulse platform:

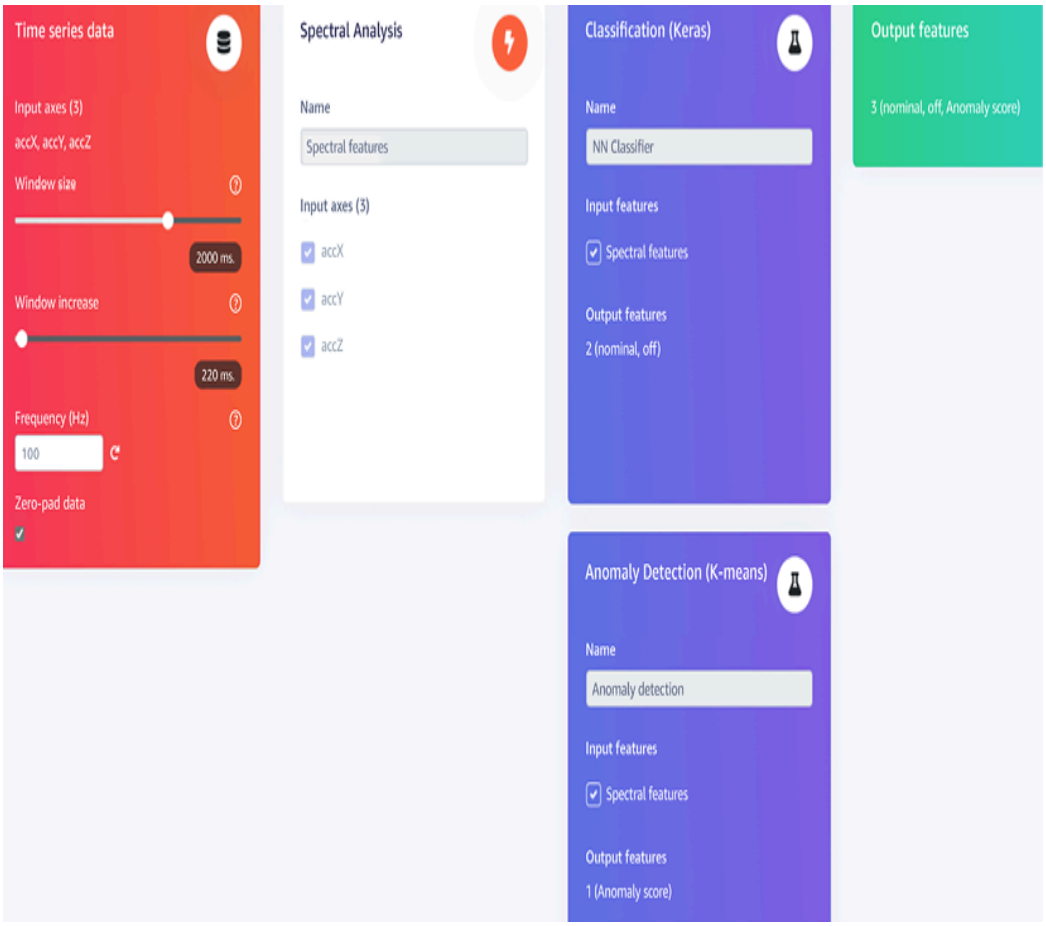


Figure 11. 16: Impulse design for anomaly detection

You can use the ESP32-S3-EYE devkit in this project as well since it has a 3-axis accelerometer on it, although the model might need to be retrained with data coming from the devkit.

There are many other open projects on the Edge Impulse platform. Experimenting with them and discussing the results on the

community forums would be an effective way of learning and gaining experience in this challenging field of IoT.

Summary

MLOps is a group of activities and tools to manage an ML product life cycle. In this chapter, we discussed **Edge Impulse** as an MLOps platform for TinyML. After cloning an open project from the Edge Impulse project repository, we built it in **Edge Impulse Studio**, the online tool to manage projects. We downloaded the **Edge Impulse C++ library** that contains the ML model and integrated that library with the devkit in an ESP-IDF project. We discussed the application in detail to see how to use the model to make inferences on the audio data coming from the mic of the devkit.

In the final chapter of the book, we will design and develop another project to cement what we have learned about TinyML so far.

Questions

Try to answer the following questions to test your understanding of the chapter:

1. Which one of the following is NOT true for an MLOps platform?
    a. It supports data versioning for traceability.
    b. It provides tools and utilities for data import and pre processing.
    c. It always comes with an AutoML tool.

d. Model training is a part of the solution.

2. Which of the following data collection methods is NOT supported by Edge Impulse?

    a. Connect to a mobile device.
    b. Connect to a remote database.
    c. Import a data file.
    d. Read from a sensor device.

3. Which of the following statements about transfer learning is false?

    a. It saves time.
    b. It saves resources.
    c. It refers to a specific version of its source dataset.
    d. It doesn't allow model architecture changes.

4. Which of the following ESP frameworks can be used to develop an ML application?

    a. ESP-BSP
    b. ESP-NN
    c. ESP RainMaker
    d. ESP-CSI

5. Which of the following is a model serialization format?

    a. PDF
    b. CSV
    c. ODX
    d. ONNX

Further reading

ML is a huge subject to learn about. The following books can be helpful in this journey:

- *TinyML Cookbook – Second Edition*, Gian Marco Iodice, Packt Publishing (https://www.packtpub.com/product/tinyml-cookbook-second-edition/9781837637362): Discusses TinyML on Arduino and Raspberry Pi with examples. The example in *Chapter 4* teaches how to develop on Edge Impulse.
- *Deep Learning with TensorFlow and Keras*, Amita Kapoor, Antonio Gulli, and Sujit Pal (https://www.packtpub.com/product/deep-learning-with-tensorflow-and-keras-third-edition/9781803232911): Teaches about deep learning. A must-read book to design effective NN models.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

12

Project – Baby Monitor

Audio analysis and classification is one of the fields where **machine learning** (**ML**) yields very successful results. In this final chapter of the book, we are going to develop an ML application on ESP32-S3 to detect baby cries in the environment. Edge Impulse will be the ML platform of the project since it provides all the necessary tools and functionality that we need for this project. The application will also connect to the RainMaker platform and pass information to the cloud to notify users via the RainMaker mobile application. A typical **Internet of Things** (**IoT**) solution usually requires one or more third-party systems/platforms to be integrated into the same solution. This project provides us with a good opportunity to gain hands-on experience with several integrations and see what problems can arise during development.

In this chapter, we will discuss the following topics to develop a baby monitoring solution:

- The feature list of the baby monitor
- Solution architecture
- Implementation
- Testing project
- Troubleshooting

- New features

Technical requirements

The only hardware requirement for this chapter is the ESP32-S3 Box Lite. We will develop the entire project on this devkit without any additional sensor hardware since the devkit already has two microphones integrated into the box.

On the software side, we will use mobile applications and online platforms to implement the project. They are:

- **ESP RainMaker platform**: The cloud platform of the project.
- **ESP RainMaker mobile application**: The companion application that comes with the RainMaker platform to add devices and manage them in the platform. It is available for both Android and iOS.
- **Amazon Alexa**: The mobile application by Amazon to control and monitor Alexa-enabled devices. It is available for both Android and iOS.
- **Edge Impulse**: The platform to generate the ML model for detecting baby cries.

You can find the project codes in the GitHub repository here: https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch12/baby_monitor.

The feature list of the baby monitor

The baby monitor device will have the following features:

- It will detect when a baby cries.
- It will define a node on ESP RainMaker.
- When a baby-crying event is detected, it will be reported to ESP RainMaker.
- The mobile application will show a notification on the mobile device for the event.
- It will be integrated with Amazon Alexa to provide input to Alexa routines.

The project may only have a very brief feature list; nonetheless, it introduces some interesting challenges. Let's begin with discussing the solution architecture in the next topic.

Solution architecture

As discussed, the goal of the project is to develop a smart device to detect baby cries. When it detects the event, it will pass this information to the ESP RainMaker platform and generate a notification on the mobile application. The solution will also integrate with Amazon Alexa and baby-crying events will be reflected on Amazon Alexa as well. The following figure shows this basic architecture:

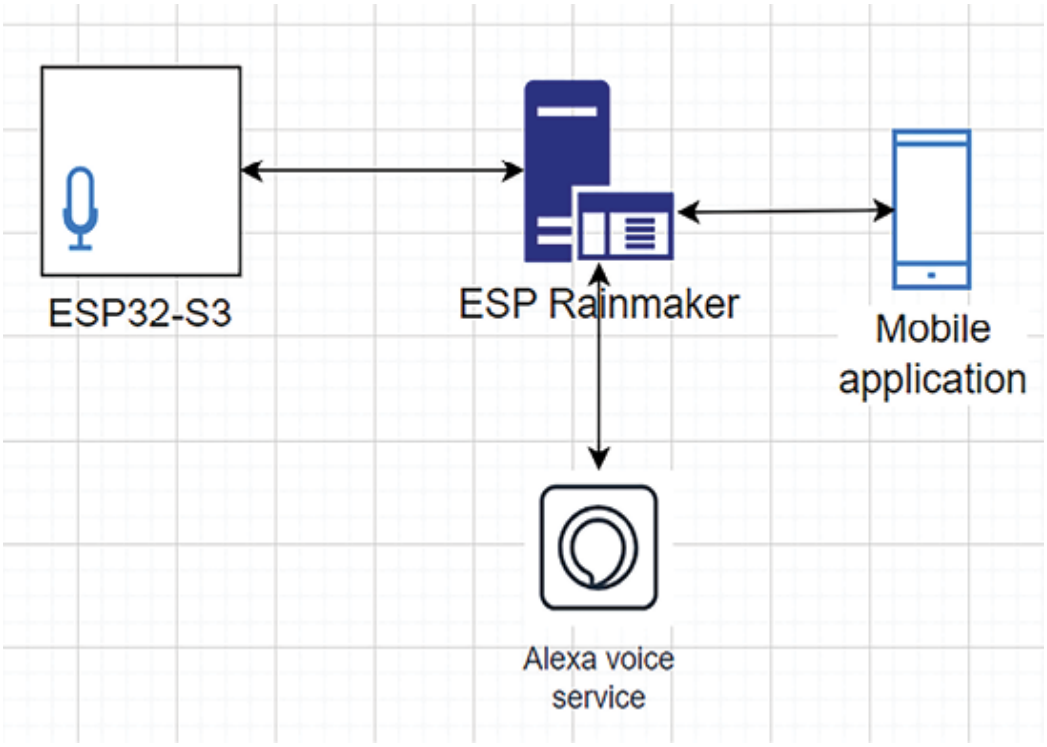


Figure 12.1: ESP RainMaker Solution components

The ESP RainMaker platform (https://rainmaker.espressif.com/) will provide the following functionality:

- Device provisioning – adding/removing devices via the mobile application
- Remote monitoring
- Mobile notifications
- Alexa Voice Service integration

The other platform that we are going to employ in this project is Edge Impulse. We need to have an ML model to analyze audio data and make inferences to decide whether a baby's cry is detected. We don't need to reinvent the wheel; we will only clone one of the open-

source projects on the Edge Impulse platform for the same purpose and download the model for ESP32.

We will develop the following classes and integrate them into the application:

- **AppAudio**: This class retrieves data from the audio subsystem and makes inferences to detect baby-crying events. *ESP-SkaiNet* is the framework that provides the abstraction for the underlying audio hardware and interface to access its functionality. The audio data is then passed to the ML inference engine, which is part of the *Edge Impulse* SDK.
- **AppRmaker**: This is the class to connect to the *RainMaker* platform. It defines a node, device, and parameter to define a baby monitor device on RainMaker.
- **AppMem**: This monitors the internal and external memory usage at runtime.

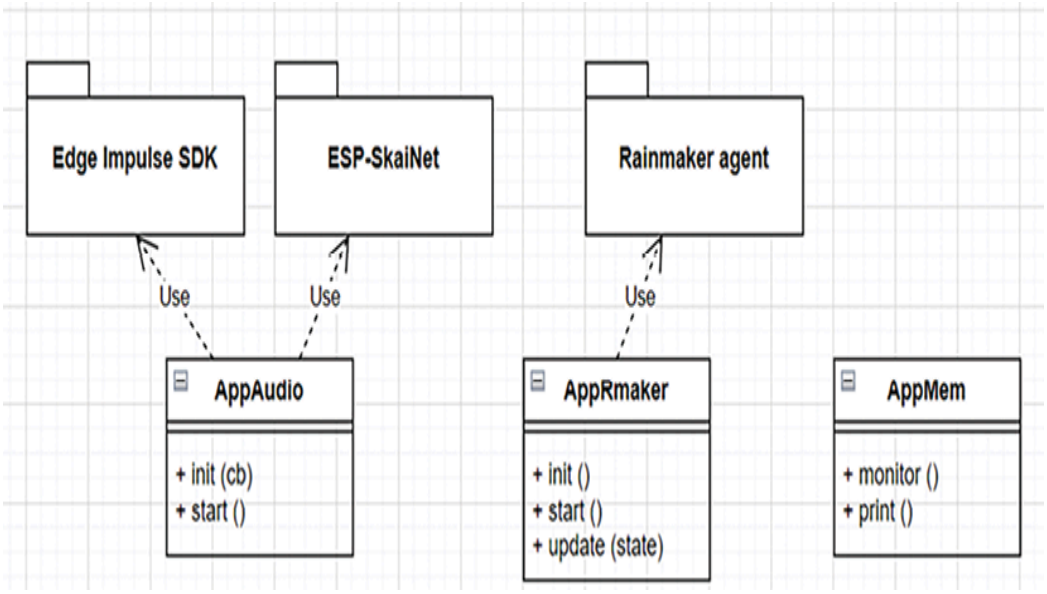


Figure 12.2: External packages and classes in the baby monitor project

After configuring the project by integrating with the frameworks and libraries listed in the above class diagram, we will develop the classes and instantiate them in an IDF application to achieve the project goals. Let's roll up our sleeves and start the implementation next.

Implementation

The implementation will have only one application running on the ESP32-S3 Box Lite. It sounds relatively easy to develop, but each project comes with its own challenges. The challenge of this project is memory management. We will discuss memory management and optimization in detail throughout the implementation. The first step is to generate the ML model.

Generating the ML model

As we mentioned earlier, we will find a baby-crying detection model from the Edge Impulse project repository and download the C++ SDK with the model in it. The following steps show how to do that:

1. Go to https://edgeimpulse.com/projects/ and type `baby-cry-detector` in the search box. Select the first project from the results:

Figure 12.3: Baby-cry-detector project on Edge Impulse

2. Clone the project by pressing the **Clone this project** button at the top right:

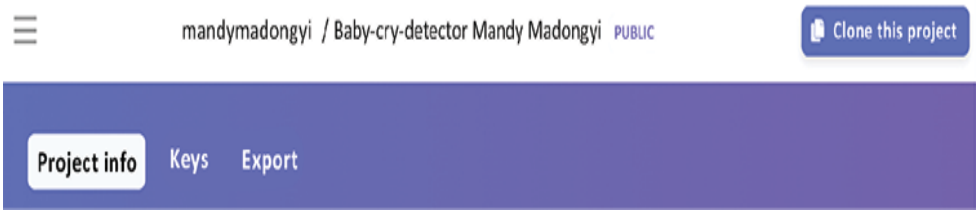


Figure 12.4: Cloning the Edge Impulse project

3. Log in to your Edge Impulse account. A pop-up window will be displayed after logging in. Press the **Clone project** button:

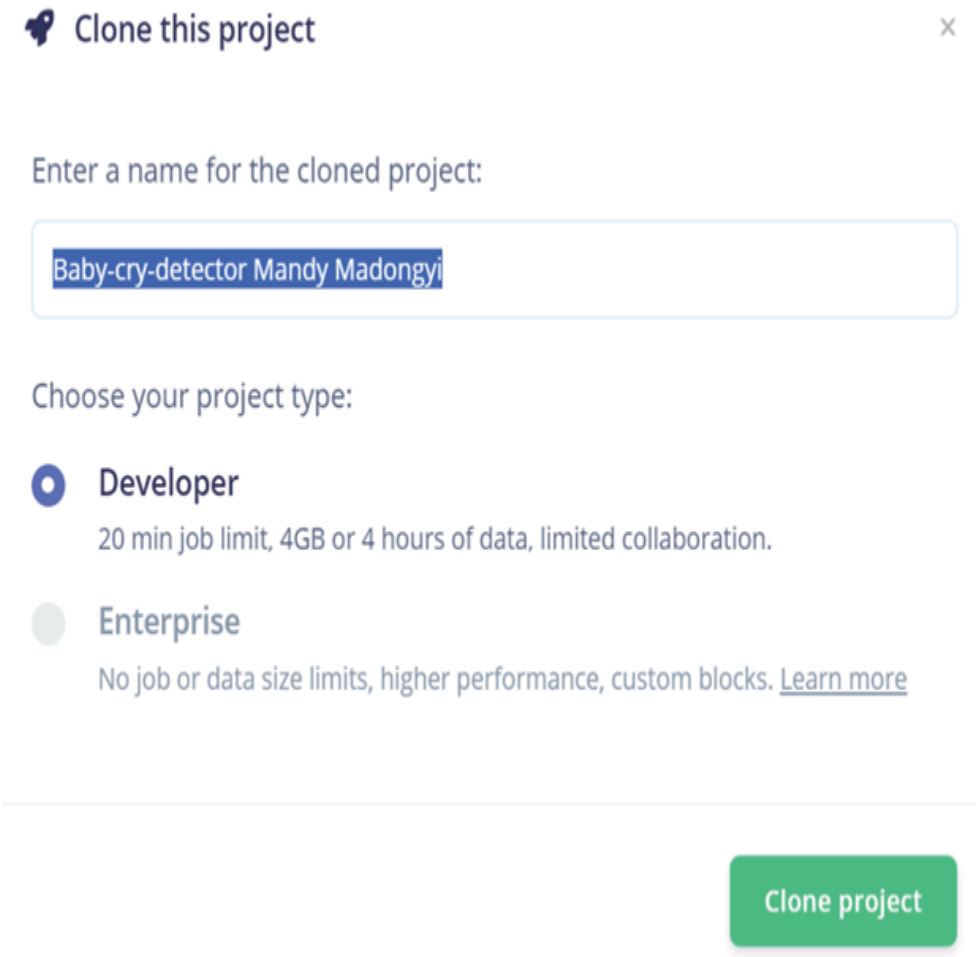


Figure 12.5: The dialog box to clone the project on Edge Impulse

4. After cloning the project, find the **Project info** section of the dashboard. Select `Espressif ESP-EYE` from the list of target devices:

Figure 12.6: Selecting the target device on the Edge Impulse project dashboard

5. Go to the deployment page of the project by selecting from the left menu. Make sure **SELECTED DEPLOYMENT** is **C++ library**:

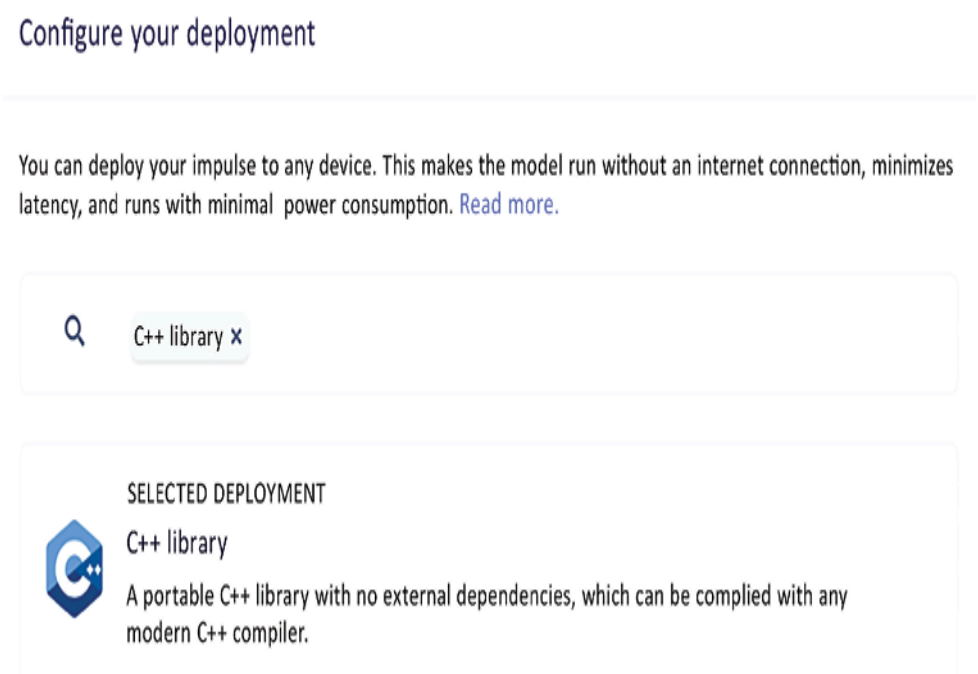
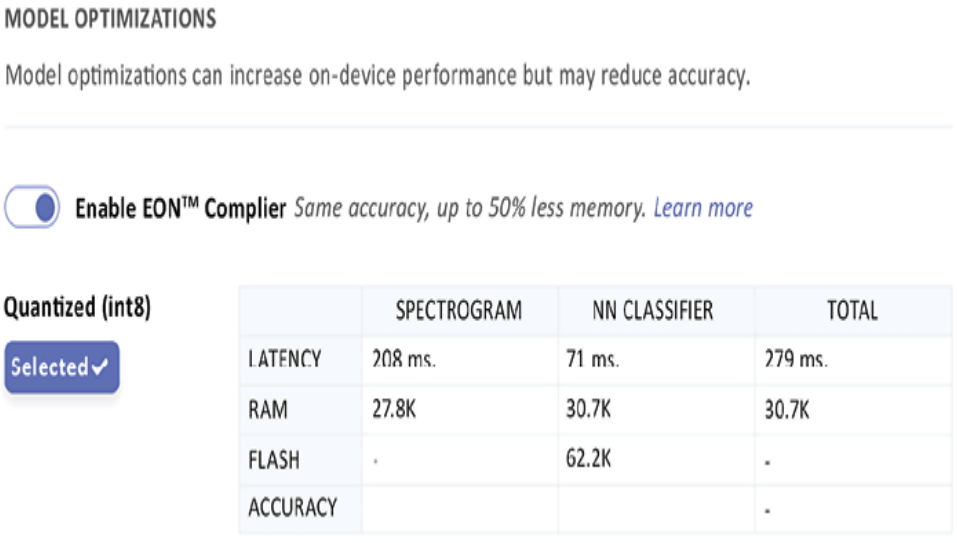


Figure 12.7: The selected deployment is C++ library on the Edge Impulse deployment page

6. In **MODEL OPTIMIZATIONS**, check whether **Enable EON$_{TM}$ Compiler** and **Quantized (int8)** are selected. If not, enable them.



	SPECTROGRAM	NN CLASSIFIER	TOTAL
LATENCY	208 ms.	71 ms.	279 ms.
RAM	27.8K	30.7K	30.7K
FLASH	-	62.2K	-
ACCURACY			-

Figure 12.8: Deployment optimizations on the Edge Impulse deployment page

7. Click on the **Build** button at the end:

Figure 12.9: Building the Edge Impulse project

8. Wait for the build process to finish. You can observe the progress in the **Build output** section of the page on the right. When it's finished, there will be a success pop-up message. Click anywhere to close it:

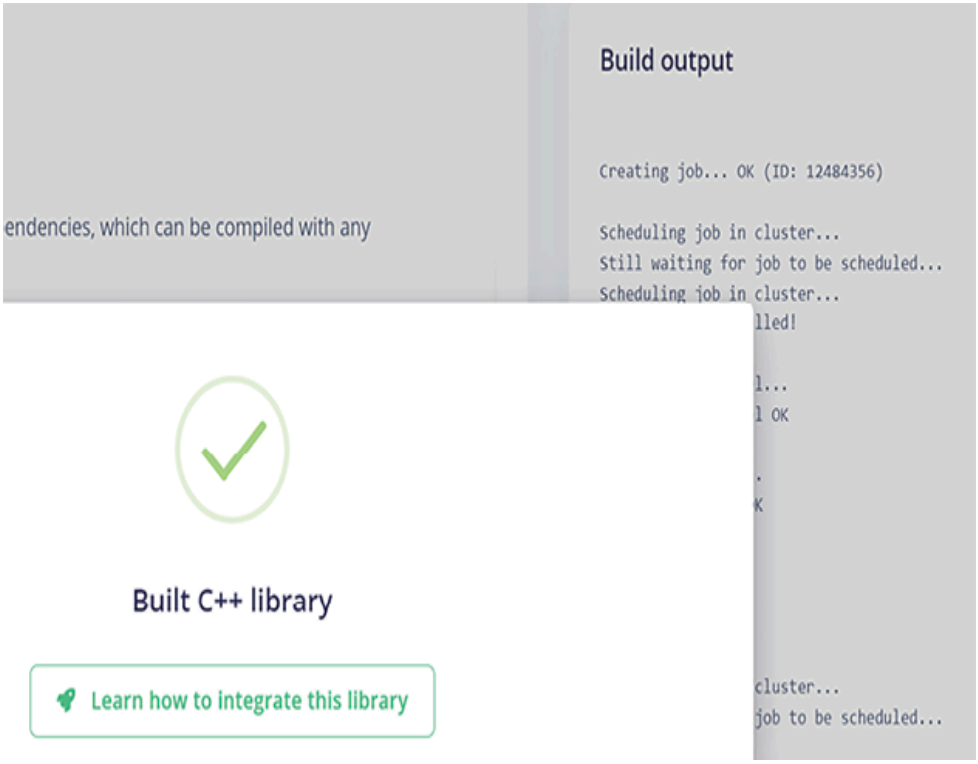


Figure 12.10: Successful build on the Edge Impulse deployment page

9. Download the build output by clicking on the link provided in the **Latest build** section of the page:

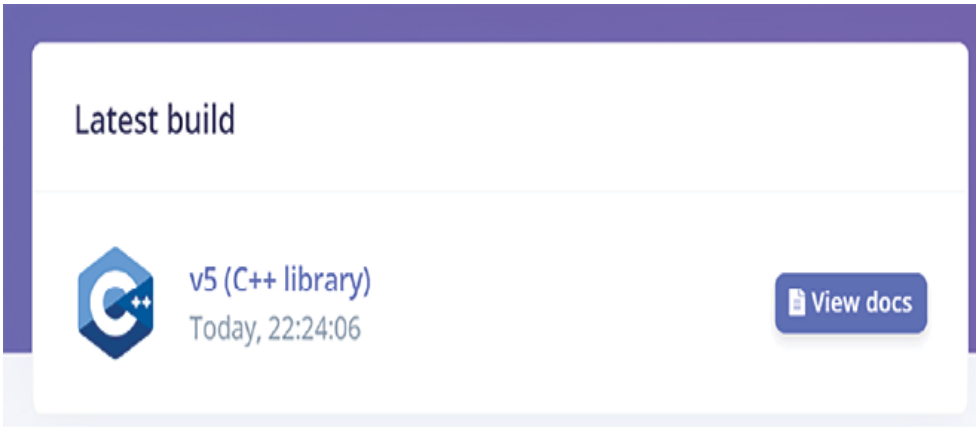


Figure 12.11: Downloading the C++ library that is generated after building the project

10. The downloaded ZIP file contains the model and the Edge Impulse SDK. Note that it doesn't contain the driver for our devkit. We will replace the ESP-EYE driver later with the one for the ESP32-S3 Box Lite while integrating the Edge Impulse SDK in the IDF project. The following screenshot shows the content of the ZIP file:

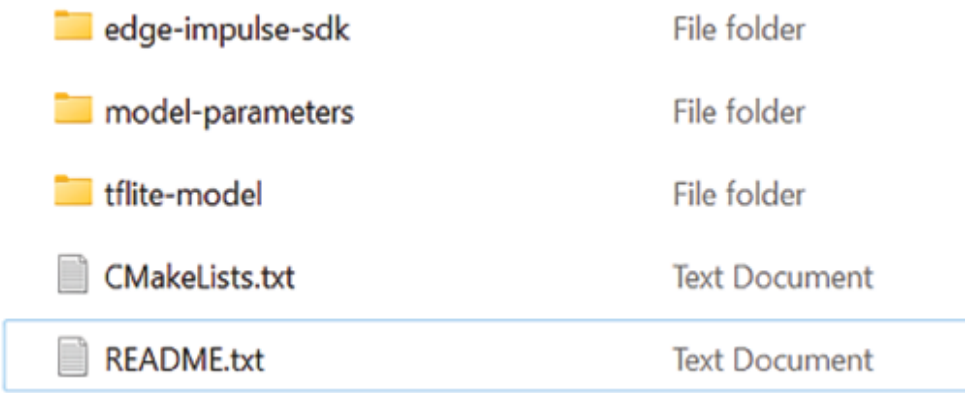


Figure 12.12: The content of the downloaded ZIP file from the Edge Impulse platform

Having the Edge Impulse SDK downloaded, we can continue with creating a new IDF project and configure it for the application.

Creating an IDF project

If you have already cloned the book repository, you can change the directory to `ch12/baby_monitor` and investigate the configuration there, but if you want to configure the project yourself, here are the steps:

1. Create a new IDF project from the command line or any other way you prefer (e.g., VS Code IDF extension):

```
$ source ~/esp/esp-idf/export.sh && idf.py creat
```

2. We need to include the *ESP-SkaiNet* and *ESP-RainMaker* frameworks in the project. Update the content of `CMakeLists.txt` at the project root as the following (set `BOOK_REPO` for your own local configuration):

```
cmake_minimum_required(VERSION 3.15)
set(BOOK_REPO ../..)
set(RMAKER_PATH ${BOOK_REPO}/ch7/common/esp-rain
set(SKAINET_PATH ${BOOK_REPO}/ch10/components/es
set(EXTRA_COMPONENT_DIRS
  ${SKAINET_PATH}/esp_codec_dev
  ${SKAINET_PATH}/esp-dsp
  ${SKAINET_PATH}/esp-sr
  ${SKAINET_PATH}/hardware_driver
  ${SKAINET_PATH}/sr_ringbuf
  ${BOOK_REPO}/ch9/common/ch9_common
      ${RMAKER_PATH}/components/esp-insights/com
      ${RMAKER_PATH}/components
```

```
      ${RMAKER_PATH}/examples/common)
include($ENV{IDF_PATH}/tools/cmake/project.cmake
project(baby_monitor)
idf_build_set_property(COMPILE_OPTIONS "-fdiagno
idf_build_set_property(COMPILE_OPTIONS "-Wno-unu
idf_build_set_property(COMPILE_OPTIONS "-Wno-may
idf_build_set_property(COMPILE_OPTIONS "-Wno-err
idf_build_set_property(COMPILE_OPTIONS "-Wno-unu
idf_build_set_property(COMPILE_OPTIONS "-Wno-err
idf_build_set_property(COMPILE_OPTIONS "-Wno-err
idf_build_set_property(COMPILE_OPTIONS "-DEI_POR
```

3. The application will not fit into the default application partition. Add a new partition configuration file, `partitions.csv`, with the following content, setting the size of the `app` partition to 2M bytes:

```
nvs,data,nvs,0x9000,24K,
phy_init,data,phy,0xf000,4K,
factory,app,factory,0x10000,2M,
fctry,data,nvs,,0x6000
```

4. Copy the `sdkconfig` file from the book repository at https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/blob/main/ch12/baby_monitor/sdkconfig. The `sdkconfig` file has some important configuration items, which we will discuss later while developing the code.

5. Extract the Edge Impulse SDK that we downloaded earlier into the project root. Be careful that the current `CMakeLists.txt` file is not overwritten:

```
$ unzip ~/Downloads/baby-cry-detector-mandy-mado
```

6. Remove the directory, `edge-impulse-sdk/porting/espressif`, which is the default ESP32 port in the Edge Impulse SDK. Replace it with the one in the book repository at https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/tree/main/ch12/baby_monitor/edge-impulse-sdk/porting/espressif.
7. Overwrite the `main/CMakeLists.txt` file from the book repository at https://github.com/PacktPublishing/Developing-IoT-Projects-with-ESP32-2nd-edition/blob/main/ch12/baby_monitor/main/CMakeLists.txt. It contains the script to include Edge Impulse SDK in the project.
8. Rename the main source code file `main/main.cpp`:

```
$ mv main/baby_monitor.c main/main.cpp
```

9. Edit `main/main.cpp` with the following content:

```
extern "C" void app_main(void) { }
```

10. Test the project configuration by building it:

```
$ idf.py build
```

The project should build without error and generate the firmware binaries at this point. We are ready for development now.

Developing the application

As discussed in the architecture, we have three classes to be implemented: `AppAudio`, `AppRmaker`, and `AppMem`. The `AppAudio` class will do the baby-cry detection. It is basically not very different from the `AppSpeech` class implementation in *Chapter 11*, *Developing on Edge Impulse*, except for one important point – external RAM usage. Combining audio processing and ML inference with ESP RainMaker in a single application requires a lot of memory, which cannot fit into the internal RAM of ESP32. Therefore, we need to move some of the buffers and task stacks into the SPIRAM of the ESP32-S3 module of the devkit. Let's see how we can implement the `AppAudio` class in the `main/AppAudio.hpp` header file:

```
#pragma once
#include <cstring>
#include <mutex>
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
```

```
#include "sdkconfig.h"
#include "edge-impulse-sdk/classifier/ei_run_classi
#include "dl_lib_coefgetter_if.h"
#include "esp_afe_sr_models.h"
#include "esp_board_init.h"
#include "model_path.h"
```

The header files that we need in this implementation cover the FreeRTOS functionality, audio processing functions and structures that come in ESP-SkaiNet, and the Edge Impulse SDK for ML inference. Having all necessary header files included, we can define the class:

```
namespace app
{
    class AppAudio
    {
    private:
        static constexpr int CRYING_IDX{0};
        static constexpr int NOISE_IDX{1};
```

In the `private` section of the `AppAudio` class, we start with defining two indexes for the inference results. The model defines two categories as `crying` and `noise` in the `model-parameters/model_variables.h` file and the inference function returns the index values for them. We represent them as `CRYING_IDX` and `NOISE_IDX` in the class. Next comes the buffers for audio processing:

```
        static constexpr int AUDIO_BUFFER_SIZE{16000
        float *m_audio_buffer;
        float *m_features;
        int m_audio_buffer_ptr{0};
        std::mutex m_features_mutex;
```

We have two buffers for audio processing: `m_audio_buffer` to hold the audio data that comes from the **audio front-end** (**AFE**) and `m_features` as a secondary buffer to be passed to the inference engine. They both require `16000 * sizeof(float)` bytes in the memory, so they must go to the external RAM. We will allocate memory for them from the SPIRAM during initialization. We continue with the FreeRTOS task definitions and stacks:

```
        static constexpr int DETECT_TASK_BUFF_SIZE{4
        inline static uint8_t *m_detect_task_buffer;
        inline static StaticTask_t m_detect_task_dat
        static constexpr int ACTION_TASK_BUFF_SIZE{4
        inline static uint8_t *m_action_task_buffer;
        inline static StaticTask_t m_action_task_dat
```

One of the places that we can save the internal memory is FreeRTOS tasks. We usually create a FreeRTOS task via the `xTaskCreate` function, but it uses the internal heap for the task structure data and task stack. The alternative to this is to use static tasks with external memory. To do that, we need to enable an option in `sdkconfig`. The

following figure is a snapshot from `menuconfig` showing this option enabled:

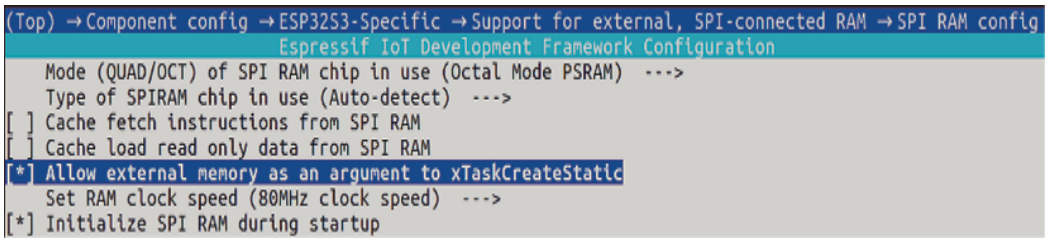
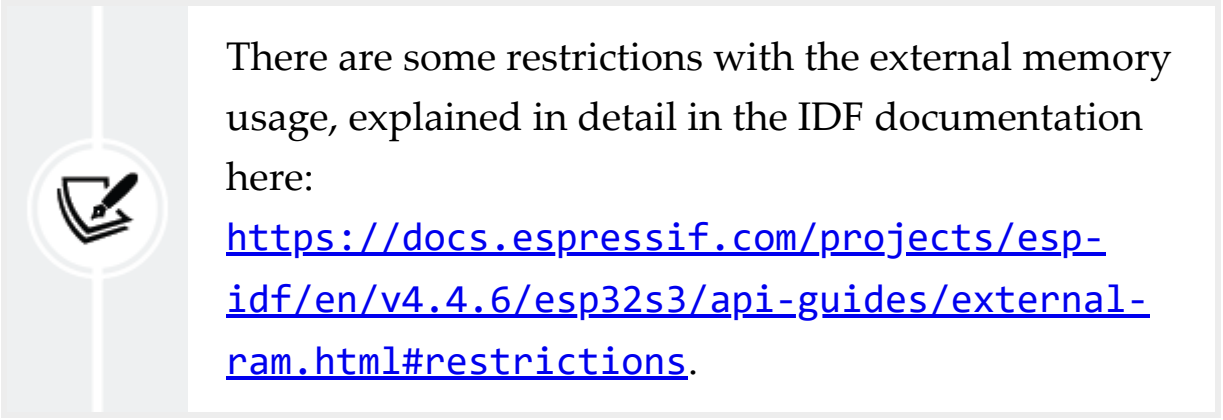


Figure 12.13: The option in sdkconfig for using external memory with FreeRTOS tasks

With **Allow external memory as an argument to xTaskCreateStatic** enabled, we can allocate `m_action_task_buffer` and `m_detect_task_buffer` from the SPIRAM and pass it to the `xTaskCreateStatic` function to be used as the stacks of the tasks. This makes 8K bytes gain on the internal heap. The other task definition is for the audio data feeding:

```
        static constexpr int FEED_TASK_BUFF_SIZE{8 *
        inline static uint8_t m_feed_task_buffer[
            FEED_TASK_BUFF_SIZE];
        inline static StaticTask_t m_feed_task_data;
```

We define the feed task as a static task, too. However, its stack must be on the internal RAM this time. The reason for that is the AFE functions called in the task, which acquire data from the audio subsystem, and this causes crashes when the external memory is used as the task stack.



There are some restrictions with the external memory usage, explained in detail in the IDF documentation here: https://docs.espressif.com/projects/esp-idf/en/v4.4.6/esp32s3/api-guides/external-ram.html#restrictions.

Next, we define the AFE-related handles:

```
esp_afe_sr_iface_t *m_afe_handle{nullptr};
esp_afe_sr_data_t *m_afe_data{nullptr};
```

The first one is the handle for the AFE interface and the second one is to access AFE data. We have two more `private` member variables in the class:

```
std::function<void(bool)> m_crying_fn;
bool m_crying;
```

The `m_crying_fn` function is the callback function object for class clients when a new crying event is detected. A client registers its event handling function with the `AppAudio` class, so it gets informed when the event happens. The `m_crying` variable shows whether the last result returned by the ML inference is a crying event. The remaining `private` members of the class are all functions:

```
        static void feedTask(void *arg);
        static void detectTask(void *arg);
        static void actionTask(void *arg);
        static afe_config_t defaultAfeConfig();
```

The `feedTask`, `detectTask`, and `actionTask` functions are the FreeRTOS tasks to retrieve audio data from the AFE, process it, and act after ML inference, respectively. The `defaultAfeConfig` function returns the configuration to initialize the AFE. We will discuss these functions in detail while developing the function bodies. There is nothing left in the `private` section of the class, so we can move on to the `public` section. The first implementation in the `public` section is the `init` function:

```
    public:
        void init(std::function<void(bool)> f)
        {
            m_crying_fn = f;
            m_crying = false;
            m_audio_buffer = new float[AUDIO_BUFFER_
            m_features = new float[AUDIO_BUFFER_SIZE
```

The `init` function takes a single argument, `f`, which is the callback for baby-crying events. We store it in the private member, `m_crying_fn`. Then we allocate memory for the `m_audio_buffer` and `m_features` buffers. Something interesting happens at this point. The total size of the allocation is `AUDIO_BUFFER_SIZE * sizeof(float)`, which makes 64KB of memory. The `new` operator

of C++ calls `malloc` behind the scenes and the default behavior of the `malloc` function is to allocate memory from the internal RAM. However, we want this memory area to be on external RAM. There is another option for this in `sdkconfig`, as shown in the following screenshot:

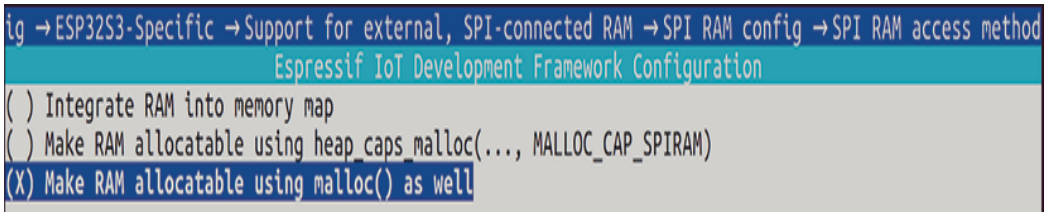


Figure 12.14: SPIRAM option for malloc in sdkconfig

When enabled, the **Make RAM allocatable using malloc() as well** option configures the `malloc` function to get memory from the external RAM as well. In addition to this option, we can configure when `malloc` allocates from the internal RAM and when from the external RAM, with the following option:

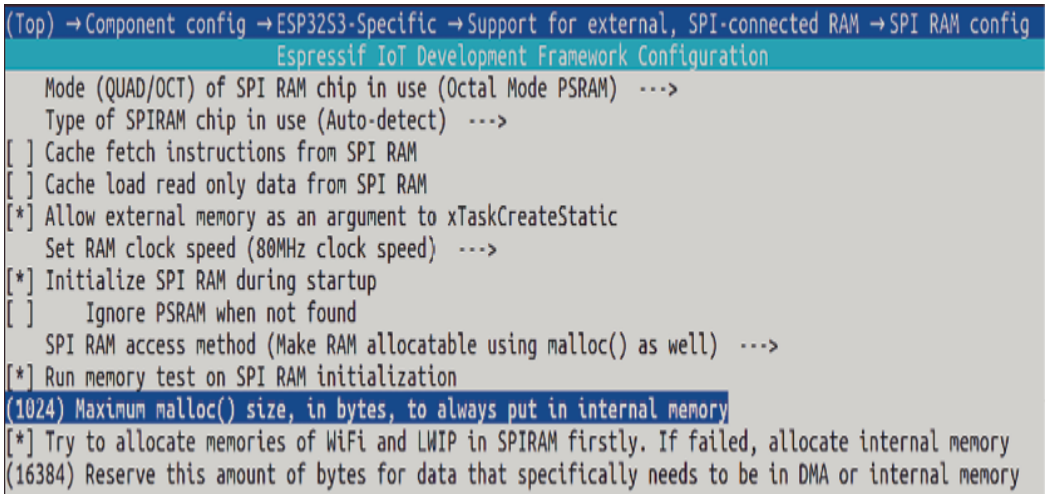


Figure 12.15: Internal memory threshold for malloc in sdkconfig

We set the internal memory size threshold for the `malloc` function in the **Maximum malloc() size, in bytes, to always put in internal memory** option. The limit here is `1024` and anything over 1024 bytes will be allocated from the external memory. Going back to our code, therefore, the `m_audio_buffer` and `m_features` buffers will be on the external RAM, saving 128K bytes of internal heap! We continue with the task stack allocations:

```
            m_detect_task_buffer = new uint8_t[
                DETECT_TASK_BUFF_SIZE];
            m_action_task_buffer = new uint8_t[
                ACTION_TASK_BUFF_SIZE];
```

Very similar to the audio data buffers, the stacks for the detection and action tasks also go into the external RAM since both are over the 1KB threshold. The `init` function ends with the initialization of the devkit and AFE:

```
            esp_board_init(AUDIO_HAL_16K_SAMPLES, 1
            m_afe_handle = const_cast<esp_afe_sr_ifa
                &ESP_AFE_VC_HANDLE);
            afe_config_t afe_config = defaultAfeCon
            m_afe_data = m_afe_handle->create_from_
                &afe_config);
        }
```

We initialize the devkit by calling `esp_board_init`. Then, we set the AFE handle from the `ESP_AFE_VC_HANDLE` declaration in the

AFE framework. We define a new AFE configuration by calling the `defaultAfeConfig` function and use this configuration to create the handle for accessing the underlying AFE data.

The AFE framework defines two handles for its interface: `ESP_AFE_SR_HANDLE` is for the speech recognition functionality and `ESP_AFE_VC_HANDLE` is for the voice communication. We only need the voice communication interface in this project. For more usage scenarios, you can see the documentation here: https://docs.espressif.com/projects/esp-sr/en/latest/esp32s3/audio_front_end/README.html.

The other `public` function of the `AppAudio` class is `start`:

```
        void start(void)
        {
            xTaskCreateStaticPinnedToCore(feedTask,
                FEED_TASK_BUFF_SIZE, this, 5, m_feed
                &m_feed_task_data, 0);
            xTaskCreateStaticPinnedToCore(detectTask
                DETECT_TASK_BUFF_SIZE, this, 5,
                m_detect_task_buffer, &m_detect_task
            xTaskCreateStaticPinnedToCore(actionTask
                ACTION_TASK_BUFF_SIZE, this, 5,
                m_action_task_buffer, &m_action_task
```

```
        } // function end
    }; // class end
```

In the `start` function, we only create the tasks. For that, we call the `xTaskCreateStaticPinnedToCore` function of ESP-IDF FreeRTOS. This function enables us to create tasks on a specific processor – i.e., we can set the CPU that the task will run on explicitly. The feed task has its stack on the internal RAM and the other two have on the external RAM as we allocated in the `init` function of the class. We also place the feed task on CPU-0 and others on CPU-1. The class definition is done but not all its functions are implemented. Let's code the `feedTask` function body next:

```
    void AppAudio::feedTask(void *arg)
    {
        AppAudio *obj{static_cast<AppAudio *>(arg)};
        int audio_chunksize = obj->m_afe_handle->
            get_feed_chunksize(obj->m_afe_data);
        int feed_channel = esp_get_feed_channel();
        int16_t *i2s_buff = new int16_t[audio_chunks
            feed_channel];
```

The purpose of the `feedTask` function is to feed the AFE with raw audio data. We reserve a buffer, `i2s_buff`, to hold audio data. Each audio sample is represented with a 16-bit signed integer and the size of the buffer is calculated by multiplying the number of channels

and audio chunk size – i.e., the number of samples in each sampling cycle. Then we run the task loop:

```
        while (true)
        {
            esp_get_feed_data(false, i2s_buff, audio
                sizeof(int16_t) * feed_channel);
            obj->m_afe_handle->feed(obj->m_afe_data,
        }
    } // end of function
```

In the task loop, we call the `esp_get_feed_data` function to get audio data from the audio subsystem of the devkit. It is raw data, and we pass it to the AFE by calling the `feed` function over the AFE handle. The AFE filters the noise and merges the channels' data into normalized, single-channel data. The next function is `detectTask`:

```
    void AppAudio::detectTask(void *arg)
    {
        AppAudio *obj{static_cast<AppAudio *>(arg)};
        int afe_chunksize{obj->m_afe_handle->
            get_fetch_chunksize(obj->m_afe_data)};
        while (true)
        {
            afe_fetch_result_t *res = obj->m_afe_han
                fetch(obj->m_afe_data);
            if (res == nullptr || res->ret_value ==
            {
                continue;
```

```
            }
            for (int i = 0; i < afe_chunksize; ++i)
            {
                obj->m_audio_buffer_ptr %= AUDIO_BUF
                obj->m_audio_buffer[obj->m_audio_buf
                    res->data[i];
            }
```

The `detectTask` function retrieves the pre-processed, single-channel data from the AFE. In the loop, the `fetch` function of the AFE handle is used to retrieve data. We simply skip the loop cycle if the `fetch` function fails. If it succeeds, we copy the audio data into the class instance buffer, `m_audio_buffer`. It is a circular buffer, so we copy the incoming bytes starting from the latest position in the buffer. It rewinds when the current position reaches the end. We complete the function body as follows:

```
            {
                std::lock_guard<std::mutex> guard(
                    obj->m_features_mutex);
                for (int i = 0; i < AUDIO_BUFFER_SIZ
                {
                    obj->m_features[i] = obj->m_audio
                        obj->m_audio_buffer_ptr + i)
                        %AUDIO_BUFFER_SIZE];
                }
            } // scope end
```

```
        } // loop end
    } // function end
```

The last duty of the `detectTask` function is to flatten the circular buffer, `m_audio_buffer`, into time-ordered data. The `m_features` member variable points to a memory space for that purpose. We copy the audio samples from `m_audio_buffer` into `m_features` starting from the oldest data in `m_audio_buffer`. The memory area pointed by `m_features` is accessed from different FreeRTOS tasks, so we must protect it via mutex.

After having the time-ordered audio data in `m_features`, we can use it in another FreeRTOS task to run ML inference and see if it contains baby-crying audio. The next function does that:

```
    void AppAudio::actionTask(void *arg)
    {
        AppAudio *obj{static_cast<AppAudio *>(arg)};
        ei_impulse_result_t result = {nullptr};
        auto get_data_fn = [&obj](size_t offset, siz
            float *out_ptr) -> int
        {
            memcpy(out_ptr, obj->m_features + offset
                sizeof(float));
            return 0;
        }; // lambda function end
```

We start the `actionTask` implementation with the local variables. The `result` variable will hold inference results and the `get_data_fn` lambda function is a helper for the inference engine to copy audio samples into its own internal buffer. The task loop comes next:

```
        while (true)
        {
            signal_t features_signal{get_data_fn,
                AUDIO_BUFFER_SIZE};
            int result_idx{NOISE_IDX};
            {
                std::lock_guard<std::mutex> guard(
                    obj->m_features_mutex);
                if (run_classifier(&features_signal,
                    EI_IMPULSE_OK)
                {
                    for (uint16_t i = 0; i <
                        EI_CLASSIFIER_LABEL_COUNT; +
                    {
                        if (result.classification[i]
                            result.classification[
                            result_idx].value)
                        {
                            result_idx = i;
                        }
                    } // for end
                } // if end
            } // mutex scope end
```

In each run of the loop, we call the `run_classifier` function of the Edge Impulse SDK, which does the ML inference on data. If `run_classifier` returns successfully, we iterate over the `result.classification` values to find the label with the highest probability. In our application, the label is either `crying` or `noise`, as comes in the Edge Impulse model variables located in the `model-parameters/model_variables.h` header file. The final value of the `result_idx` variable shows the inference result with the highest probability. The `actionTask` function ends with calling the client callback:

```
            switch (result_idx)
            {
            case CRYING_IDX:
            {
                if (!obj->m_crying)
                {
                    obj->m_crying_fn(true);
                    obj->m_crying = true;
                }
            }
            break;
            case NOISE_IDX:
            {
                if (obj->m_crying)
                {
                    obj->m_crying_fn(false);
                    obj->m_crying = false;
                }
            }
```

```
            break;
            default:
                break;
            } // switch end
            vTaskDelay(pdMS_TO_TICKS(1000));
        } // loop end
    } // function end
```

In a `switch` statement, we just call the `obj->m_crying_fn` function with `true` if the inference result is `crying` or `false` if the result is `noise`. We wait one second before the next loop runs so that the AFE can refresh the buffer with new data. The last function of the class is `defaultAfeConfig`:

```
    afe_config_t AppAudio::defaultAfeConfig()
    {
        return {
            .aec_init = false,
            .se_init = true,
            .vad_init = false,
            .wakenet_init = false,
            .voice_communication_init = true,
            .voice_communication_agc_init = false,
            .voice_communication_agc_gain = 15,
            .vad_mode = VAD_MODE_3,
            .wakenet_model_name = nullptr,
            .wakenet_mode = DET_MODE_2CH_90,
            .afe_mode = SR_MODE_LOW_COST,
            .afe_perferred_core = 0,
```

```
            .afe_perferred_priority = 5,
            .afe_ringbuf_size = 50,
            .memory_alloc_mode = AFE_MEMORY_ALLOC_MO
            .agc_mode = AFE_MN_PEAK_AGC_MODE_2,
            .pcm_config = {3, 2, 1, 16000},
            .debug_init = false,
        };
    } // function end
} // namespace end
```

The highlights of the `defaultAfeConfig` function are that we don't use `wakenet`; we only need the voice communication interface. The configuration also has an option for memory usage, so we set it to `AFE_MEMORY_ALLOC_MORE_PSRAM` to save more internal RAM. The `pcm_config` field shows the number of channels (2 mics and a reference channel) and the sampling rate of 16KHz.

We completed the implementation of the `AppAudio` class and can move on to the `AppRmaker` class in `main/AppRmaker.hpp`:

```
#pragma once
#include <cstring>
#include <cstdint>
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "sdkconfig.h"
#include <esp_rmaker_core.h>
#include <esp_rmaker_standard_types.h>
```

```
#include <esp_rmaker_standard_params.h>
#include <esp_rmaker_standard_devices.h>
#include <esp_rmaker_common_events.h>
#include <esp_rmaker_utils.h>
#include <esp_rmaker_mqtt.h>
```

We include the necessary header files to integrate with the ESP RainMaker platform and start the class definition as follows:

```
namespace app
{
    class AppRmaker
    {
    private:
        esp_rmaker_node_t *m_rmaker_node;
        esp_rmaker_device_t *m_device;
        esp_rmaker_param_t *m_cry_param;
```

The RainMaker integration requires the definition of the RainMaker node, a RainMaker device in the node, and a RainMaker parameter in the device. We define the member variables for them in the `private` section of the class definition. Next comes the `init` function of the `AppRmaker` class in the `public` section:

```
    public:
        void init()
        {
            esp_rmaker_time_set_timezone(
```

```
        CONFIG_ESP_RMAKER_DEF_TIMEZONE);
    esp_rmaker_config_t rainmaker_cfg = {
        .enable_time_sync = true,
    };
    m_rmaker_node = esp_rmaker_node_init(&ra
        "Baby node", "esp.node.sensor");
```

In the `init` function, we first set the time zone and enable the time synchronization in the configuration variable. We create the RainMaker node with this configuration. Then we create the RainMaker device:

```
    m_device = esp_rmaker_device_create("Cry
        "esp.device.sensor", (void *)this);
    esp_rmaker_device_add_param(m_device,
        esp_rmaker_name_param_create(
          ESP_RMAKER_DEF_NAME_PARAM, "Cry se
```

We call the `esp_rmaker_device_create` function to create the RainMaker device and attach a name parameter to it to be displayed on the RainMaker mobile application. The next step in the initialization is to define the parameter for the baby-crying state:

```
    m_cry_param = esp_rmaker_param_create("E
        "esp.param.toggle", esp_rmaker_bool(
        PROP_FLAG_READ);
    esp_rmaker_param_add_ui_type(m_cry_parar
```

```
                ESP_RMAKER_UI_TOGGLE);
            esp_rmaker_device_add_param(m_device, m_
            esp_rmaker_device_assign_primary_param(r
                m_cry_param);
```

The `baby-crying` parameter is a read-only Boolean value and will be displayed as a toggle switch on the mobile application GUI. After assigning this parameter to the RainMaker device, we finish the initialization as follows:

```
            esp_rmaker_node_add_device(m_rmaker_node
        }// function end
```

We attach the device definition to the RainMaker node by calling the `esp_rmaker_node_add_device` function and this completes the `init` function. The next function of the `AppRmaker` class is `start`:

```
        void start()
        {
            esp_rmaker_start();
        }
```

The `start` function is quite simple. We only start the RainMaker agent in the application by calling `esp_rmaker_start`. We need one last function in the class to update the baby-crying state:

```
        void update(bool state)
        {
            esp_rmaker_param_update_and_report(m_cry
                esp_rmaker_bool(state));
            if (state)
            {
                esp_rmaker_raise_alert("crying");
            }
        } // function end
    }; // class end
} // namespace end
```

The `update` function takes a Boolean argument for the baby-crying state. In the function body, we call the `esp_rmaker_param_update_and_report` function to update the state on the RainMaker platform. If the argument shows `baby-crying`, we also trigger a notification on the mobile device by calling the `esp_rmaker_raise_alert` function. With this, the RainMaker integration is also done. The last class that we are going to develop in the application is `AppMem` in the `main/AppMem.hpp` header file:

```
#pragma once
#include "esp_timer.h"
#include "esp_log.h"
#include "esp_err.h"
#include "esp_heap_caps.h"
```

The purpose of the `AppMem` class is to monitor the internal and external heaps at runtime so that we can understand how the application uses the available memory while running on ESP32. After including the header files, we continue with the class implementation:

```
namespace app
{
    class AppMem
    {
    private:
        constexpr static const char *TAG{"app-mem"};
        esp_timer_handle_t m_periodic_timer;
```

In the `private` section, we define two members. `TAG` is the class member to be used with logging, and `m_periodic_timer` is the instance member variable for periodic monitoring. We also define a `private` section function for `m_periodic_timer` as its callback:

```
        static void periodic_timer_callback(void *ar
        {
            ESP_LOGI(TAG, "------- mem stats ------
            ESP_LOGI(TAG, "internal\t: %10u (free) /
                heap_caps_get_free_size(
                MALLOC_CAP_INTERNAL), heap_caps_get_
                MALLOC_CAP_INTERNAL));
            ESP_LOGI(TAG, "spiram\t: %10u (free) / %
                heap_caps_get_free_size(MALLOC_CAP_S
```

```
                heap_caps_get_total_size(MALLOC_CAP_
        }
```

In the `periodic_timer_callback` function, we print `free` internal heap vs `total` internal heap and `free` external heap vs `total` external heap. In this way, we can see the heap usage live printed on the serial output. The public `monitor` function will create the periodic timer with `periodic_timer_callback` as we are going to code next:

```
    public:
        void monitor(void)
        {
            const esp_timer_create_args_t periodic_t
                periodic_timer_callback,
                this};
            ESP_ERROR_CHECK(esp_timer_create(&perioc
                &m_periodic_timer));
            ESP_ERROR_CHECK(
                esp_timer_start_periodic(m_periodic_
                  5000000u));
        }
```

In the `monitor` function, we first define the configuration for the periodic timer and call `esp_timer_create` to create the timer. The `esp_timer_start_periodic` function starts the timer with a period

of 5 seconds. The last function to be implemented in the class is `print`:

```
        void print(void)
        {
            periodic_timer_callback(nullptr);
        } // function end
    }; // class end
} // namespace end
```

The purpose of the `print` function is to be able to print the heap usage on the serial output, independent of the periodic timer. We can use it in the application when we want to see the heap usage at some point.

All the classes of the application are ready to be employed in the `app_main` function. Let's edit the `main/main.cpp` source code file of the application:

```
#include "AppAudio.hpp"
#include "AppDriver.hpp"
#include "AppRmaker.hpp"
#include "AppMem.hpp"
namespace
{
    app::AppAudio app_audio;
    app::AppDriver app_driver;
    app::AppRmaker app_rmaker;
```

```
    app::AppMem app_mem;
}
```

We include the class header files and declare the class instances in an anonymous namespace. The `app_main` function comes after them:

```
extern "C" void app_main()
{
    app_mem.print();
    app_driver.init();
    app_rmaker.init();
    app_audio.init([](bool crying)
                   { app_rmaker.update(crying); });
```

We start the `app_main` function by calling `app_mem.print` so that we can see the heap usage before everything. Then, we initialize the instances. The `app_audio` object takes a lambda function as its argument to update the `app_rmaker` object with the baby-crying state. Next, we call the `start` functions of the instances:

```
    app_mem.print();
    app_rmaker.start();
    app_driver.start();
    app_audio.start();
```

Again, we call `app_mem.print` just before starting the instances to operate. The difference with the earlier call to the `print` function

shows how much heap is consumed by the initialization. After starting all the instances, we monitor the heap usage during the operation:

```
    app_mem.monitor();
} // end of app_main
```

We have all the classes implemented, the `app_main` function is finished, and finally, the application is ready for testing.

Testing the project

In the test, we will connect the devkit to the RainMaker platform, as we did earlier in *Chapter 7*, *ESP32 Security Features For Production-Grade Devices*, then provide baby-crying sounds to see whether the event is generated and passed to the RainMaker platform. Let's do this in steps:

1. Make sure the devkit is not registered as another device in the RainMaker platform by checking the RainMaker mobile application. If registered, remove it from the mobile application.

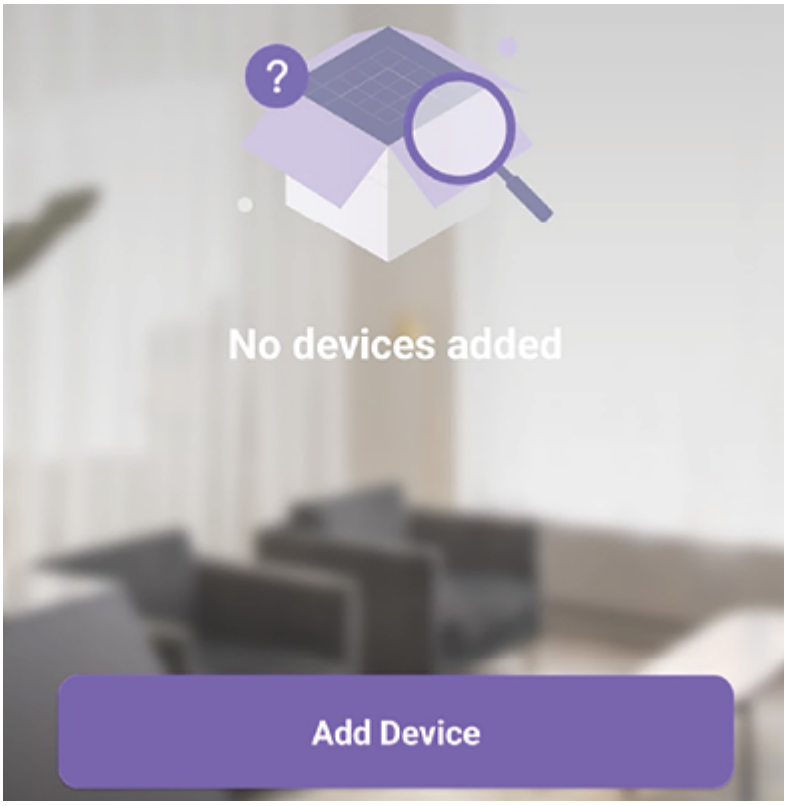


Figure 12.16: The RainMaker mobile application with no device registered

2. Flash the application on the devkit:

```
$ idf.py erase-flash flash monitor
```

3. A QR code will be printed on the serial console. Scan it with the RainMaker mobile application by pressing the **Add Device** button that is shown in *step 1*.

Figure 12.17: QR code to provision the device to the local WiFi, then the RainMaker

4. Confirm that the device is provisioned and listed as **Cry sensor** on the mobile application:

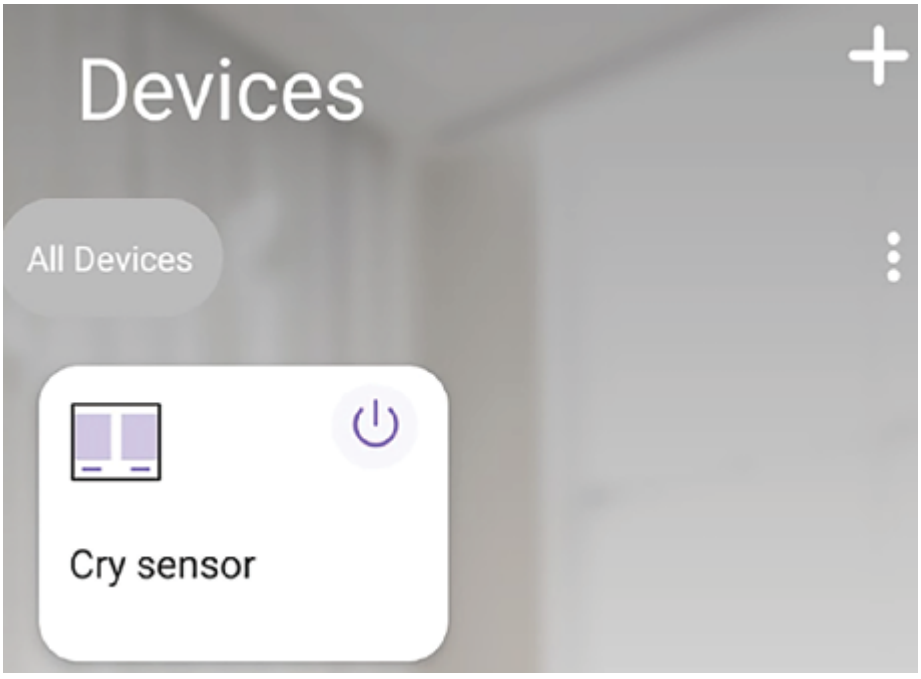


Figure 12.18: Cry sensor on the RainMaker mobile application

5. Try playing a baby-crying sound from the internet to test the application. When detected, a notification will be seen on the mobile device coming from the RainMaker application and listed on the **Settings/Notifications** screen of the RainMaker application as well.

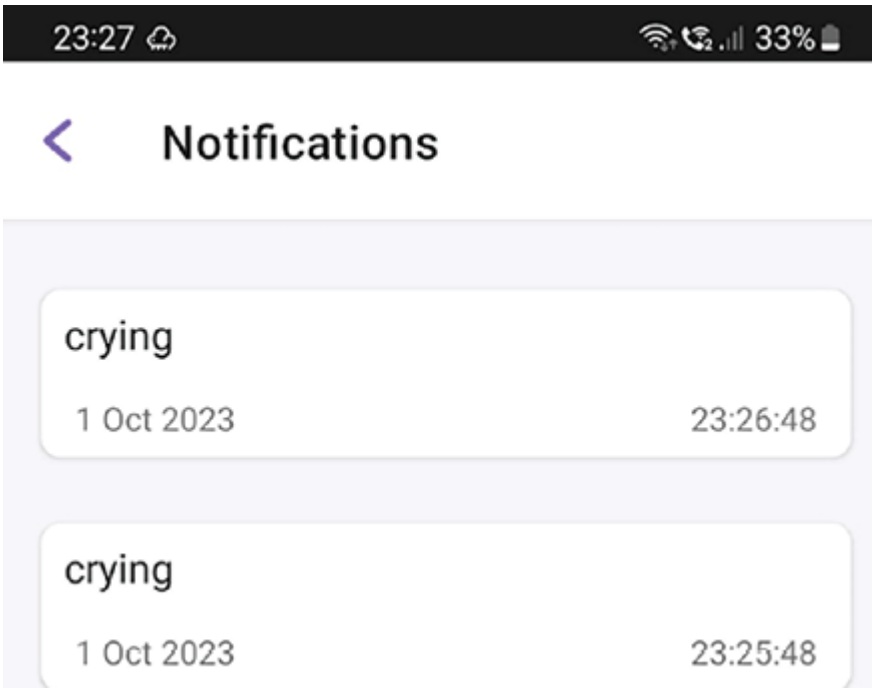


Figure 12.19: Baby-crying notifications on the mobile device

6. Check the serial console of the devkit to see the event there, too:

```
I (625425) esp_rmaker_param: Reporting params: {"Cry sensor":{"Baby crying":true}}
I (625435) esp_rmaker_param: Reporting alert: {"esp.alert.str":"crying"}
I (626705) esp_rmaker_param: Reporting params: {"Cry sensor":{"Baby crying":false}}
```

Figure 12.20: The baby-crying event logs on the device serial console

7. The serial console also shows the heap usage every 5 seconds:

```
I (644395) app-mem: ------- mem stats -------
I (644395) app-mem: internal    :     158307 (free) /     306147 (total)
I (644395) app-mem: spiram      :    8062555 (free) /    8388607 (total)
```

Figure 12.21: Heap usage displayed on the serial console

Let's investigate how the heap usage has changed since the application was started. The following logs are from the serial console:

```
I (805) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
I (825) spiram: Reserving pool of 16K of internal me
I (825) app-mem: ------- mem stats -------
I (835) app-mem: internal       :     265135 (free)
I (845) app-mem: spiram :    8386307 (free) /    838
<removed>
I (1095) AFE_VC: wakenet_init: 0, voice_communicatio
I (1205) AFE_VC: mode: 0, (Jun  2 2023 11:39:44)
I (1205) app-mem: ------- mem stats -------
I (1205) app-mem: internal      :     171159 (free)
I (1205) app-mem: spiram        :    8106951 (free)
<removed>
I (4965) esp_rmaker_time: The current time is: Mon O
I (7375) app-mem: ------- mem stats -------
I (7375) app-mem: internal      :     161619 (free)
I (7375) app-mem: spiram        :    8062803 (free)
I (12375) app-mem: ------- mem stats -------
I (12375) app-mem: internal     :     161619 (free)
I (12375) app-mem: spiram       :    8062803 (free)
```

At the beginning of the application, the log shows **265,135** bytes of internal heap were free. It drops to **171,159** after calling the `init` functions of the objects. When all the objects started to operate, the

free internal heap size became **161,619** bytes. This makes more than 100K bytes of heap allocated in the application. On the external heap side, **8,386,307** bytes were available at the beginning and dropped to **8,062,803**, meaning that an additional 320K bytes are used by the objects and underlying frameworks. This is a lot for an embedded application, but luckily, we have ESP32!

Although we moved the main buffers to the external RAM, the application is still far from being optimized for memory. The task stack sizes could be reduced by monitoring their maximum heap usage and some other runtime buffers could also be moved to the external RAM. ESP-IDF documentation shows more possibilities to minimize RAM usage here: https://docs.espressif.com/projects/esp-idf/en/v4.4.6/esp32s3/api-guides/performance/ram-usage.html.

A market-worthy product always needs to be investigated for optimal RAM and flash use. The `idf.py` tool provides a good starting point for this task. Let's run it on our application to see the memory statistics:

```
$ idf.py size
Total sizes:
Used stat D/IRAM:  139839 bytes ( 206017 remain, 40
      .data size:   22373 bytes
      .bss  size:   28680 bytes
      .text size:   87759 bytes
   .vectors size:    1027 bytes
Used Flash size : 1166081 bytes
```

```
        .text     :  894283 bytes
        .rodata   :  271542 bytes
 Total image size: 1277240 bytes (.bin may be padded
```

The output of `idf.py` shows a summary of how RAM and flash are used in the application. In our case, 40.4% of the available RAM is occupied by the static data and code, and 206K bytes are left for the heap – i.e., dynamic allocations. Note that it is a bit different from what we see in the serial logs of the application. The summary gives some idea about how we should plan the heap usage, but monitoring memory at runtime is still a valuable approach to seeing the real behavior of the system.

Troubleshooting

There can be several pitfalls in this project. The following list can help you in such cases:

- Integration is probably the most difficult part of the project if you decide to configure it yourself. There are several frameworks (ESP-RainMaker, ESP-SkaiNet, Edge Impulse SDK, and BSP of the devkit) that need to work together on the ESP32-S3 Box Lite. Make sure each of them compiles on your development machine and works on the devkit separately before the integration. You can create different IDF projects for this purpose.
- As we mentioned at the beginning of the project, the `sdkconfig` file keeps key configuration items. Use the

`menuconfig` tool (`idf.py menuconfig`) to check whether `sdkconfig` contains the correct configuration values.

- While integrating with the ESP RainMaker platform, make sure everything is in a clean state – that is, there is no other device provisioned on RainMaker. You can also erase the flash memory of the devkit and use a clean build by running the following `idf.py` command:

```
$ idf.py erase-flash fullclean flash monitor
```

- If you are using `idf.py monitor` to investigate the serial output, you can see the source code files and line numbers printed on the serial console if a crash occurs. The `idf.py monitor` tool calls another tool behind the scenes, named **xtensa-esp32s3-elf-addr2line**. It comes with the ESP-IDF installation and is available when an ESP-IDF development environment is started. See this link for more information about the monitoring tool capabilities: https://docs.espressif.com/projects/esp-idf/en/v4.4.6/esp32s3/api-guides/tools/idf-monitor.html.
- GDB is a great tool for debugging when used properly. The documentation about using GDB in ESP-IDF is here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/jtag-debugging/using-debugger.html.

New features

The project has a lot of room to improve. Here are some ideas for practice:

- `RainMaker OTA update` is not enabled on the device. It is especially critical for ML applications to update the device firmware with the new version of the improved model. You can enable OTA update in the application and try upgrading device firmware over the RainMaker platform.
- You can develop another RainMaker node application on ESP32-C3-DevKitM-1 as a companion device that can turn its LED on when a baby-crying event is detected. RainMaker automations would be an easy way to make the devices work together.
- There is an LCD on the ESP32-S3 Box Lite. You can add some graphics features to the application by using *LVGL* and *SquareLine Studio*, such as displaying provisioning QR codes on the screen, RainMaker connection status, and baby-crying events in operation.
- RainMaker supports **Alexa Voice Service** (**AVS**) integration. Nonetheless, the use of the device on AVS is limited by the RainMaker platform capabilities. You can develop a standalone *smart home skill* and integrate the baby-crying sensor directly within AVS to reveal the full power of the voice assistant. An *Alexa routine* with the baby-crying sensor as a trigger for it would be an interesting experiment.
- The ML model that we downloaded from the Edge Impulse project repository is not perfect. You can try training it with

more data and see if it makes any changes, reducing false positives.

Summary

Developing an IoT product usually means a lot of integration with third-party systems and platforms. In this last chapter of the book, we had an example of this by developing a connected machine-learning application on ESP32-S3. We downloaded an ML model from the **Edge Impulse** platform and updated the Edge Impulse SDK for our devkit. On the cloud side, we employed **ESP RainMaker**. We defined a RainMaker node, device, and parameter in the application to exchange data with the RainMaker platform. The challenge of the project was memory usage. The internal memory of ESP32-S3 was not enough to accommodate all the functionality as specified in the requirements; therefore, we enabled and used **SPIRAM** to keep the buffers on it. When we look at real-world IoT projects, they also usually need to address such issues, and we had a hands-on experience by working on the project of this chapter.

For the next steps, I recommend you try other chips from Espressif Systems. Espressif Systems is known for its WiFi products but **ESP32-H2** is the one without WiFi. It has BLE and IEEE 802.15.4 subsystems, helping developers to implement cost-efficient wireless mesh networks, such as **Thread** or **Zigbee**. **Matter** is a new connectivity standard for smart home solutions that can run on top of wireless mesh networks. They are all supported by Espressif Systems in terms of both hardware and software. IoT is a vast field

that touches many technologies. It is a challenge for us, as IoT developers, to learn about these technologies and apply them to our projects. I really enjoy solving real-world problems by using ESP32 in my projects, and since you have this book and joined me in the examples of the book, I can safely assume that you enjoy it, too! It would be an honor if this book helped you in any way in your ESP32 journey.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/3Q9egBjWVZ

Answers

Chapter 02:

1. `CMakeLists.txt`. ESP-IDF integrates cmake as its build system.
2. `sdkconfig. menuconfig` is the tool to edit the project configuration parameters in `sdkconfig`.
3. `idf.py`. This Python tool comes with ESP-IDF installation and is used to manage ESP-IDF projects.
4. Built-in JTAG/USB. ESP32-S3 doesn't require an extra hardware tool for JTAG access since it has a built-in JTAG circuitry.
5. `platformio.ini`. This file is added with each new PlatformIO project and defines the project configuration.

Chapter 03:

1. An LED needs a GPIO connection to drive as a digital output.
2. I^2C requires two lines, clock and data, to communicate with other devices. The protocol supports multiple slaves on the same bus with addressing.
3. Higher transfer rate is the reason for choosing SPI communication for SD cards.
4. WS selects the right or left channel in I2S communication.
5. MRI is not a display technology for IoT.

Chapter 04:

1. FlatBuffers is especially useful when we need to pass data between different platforms.
2. Miniz is a library to compress data – it works especially well on data with repeating patterns.
3. LVGL is a graphics library, but it doesn't include display drivers.
4. The ESP-IDF Components library is not provided by Espressif Systems.
5. ESP-ADF is the framework for audio processing.

Chapter 06:

1. In STA mode, ESP32 can also retrieve a dynamic IP from the Wi-Fi router it connects to and use it to communicate on the network.
2. Although **Near-Field Communication** (**NFC**) can be used as a method for provisioning, it is not supported by ESP-IDF directly.
3. MQTT defines topics to publish and subscribe
4. HTTP GET is the method to request a resource
5. All of the above 2XX codes show a successful request, 4XX codes mean that the client didn't send an expected request to the server, and 5XX codes are returned when an error occurs while processing the request

Chapter 07:

1. Flash encryption. When it is enabled, the firmware on the flash cannot be reverse-engineered.
2. A secure channel is not required for OTA updates, but it creates a security hole to be exploited by intercepting the network traffic to obtain the new firmware. The best practice is to do OTA updates over a secure channel.
3. The RainMaker platform does require mutual authentication with devices.
4. ESP Insights is a tool to monitor ESP32 devices in the field remotely.
5. All of them are important. Each component in an IoT solution creates an attack surface for hackers.

Chapter 08:

1. AWS IoT Device SDK helps us to connect ESP32 to the AWS cloud.
2. IoT rules are used to pass messages to the other AWS services from IoT Core.
3. AWS provides Grafana as a managed service, so there is no need to install Grafana on an EC2 instance.
4. Check the lambda handler first. It is easy to test and provides logs.
5. Using the lambda test tool and sending a discovery message to the lambda function would lead to the solution faster.

Chapter 10:

1. Each category of ML algorithms responds to different classes of problems in nature and they are not comparable in terms of superiority at detecting patterns.
2. Training an ML model requires computational resources that are not available on IoT devices.
3. Quantization is a technique to shrink ML models to fit into IoT devices.
4. TFLM comes with the tools to optimize ML models for IoT devices.
5. MultiNet is the model by Espressif for detecting voice commands offline.

Chapter 11:

1. MLOps platforms help with data and model management and sometimes come with an integrated AutoML tool, but this is not always the case.
2. Edge Impulse doesn't connect to remote databases to retrieve data.
3. With transfer learning, we can add new layers on top of the source model to fit our purposes.
4. ESP-NN provides the functionality for neural network operations.
5. ONNX is an open format for ML models.

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

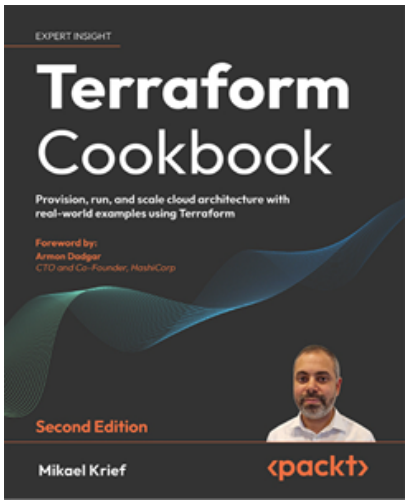
Architectural Patterns and Techniques for Developing IoT Solutions

Jasbir Singh Dhaliwal

ISBN: 9781803245492

- Get to grips with the essentials of different architectural patterns and anti-patterns
- Discover the underlying commonalities in diverse IoT applications

- Combine patterns from physical and virtual realms to develop innovative applications
- Choose the right set of sensors and actuators for your solution
- Explore analytics-related tools and techniques such as TinyML and sensor fusion
- Overcome the challenges faced in securing IoT systems
- Leverage use cases based on edge computing and emerging technologies such as 3D printing, 5G, generative AI, and LLMs



Terraform Cookbook – Second Edition

Mikael Krief

ISBN: 9781804616420

- Use Terraform to build and run cloud and Kubernetes infrastructure using IaC best practices
- Adapt the Terraform command line adapted to appropriate use cases
- Automate the deployment of Terraform confi guration with CI/CD
- Discover manipulation of the Terraform state by adding or removing resources
- Explore Terraform for Docker and Kubernetes deployment, advanced topics on GitOps practices, and Cloud Development Kit (CDK)
- Add and apply test code and compliance security in Terraform configuration

- Debug and troubleshoot common Terraform errors

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Developing IoT Projects with ESP32, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

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

Index

A

Access point (AP) mode 206

adl_serializer

reference link 123

Alexa Developer Documentation

documentation link 367

Alexa Voice Service (AVS) 354, 529

Amazon Alexa

integrating, with thing shadow 354-359

lambda handler, coding 363-367

lambda handler, creating 359-362

smart home skill, creating 368-375

troubleshooting 375

Amazon Alexa mobile application 354

Amazon FreeRTOS 13, 14, 34, 323, 329

Amazon Managed Grafana 323

Amazon Resource Names (ARNs) 368

Amazon Simple Notification Service (SNS) 353

Amazon Web Services (AWS) 13, 288

Analog-to-Digital Conversion (ADC) 140

AppAudio 169, 499

AppButton instance 182, 170

AppMem 500

AppNav 169, 170

AppRmaker 499

AppUi 159, 170, 180, 181

Arduino Core for ESP32 12

Arithmetic Logic Unit (ALU) 444

Artificial Intelligence of Things (AIoT) 1, 10

Audio Development Framework (ADF) 445

Audio Development Framework (ESP-ADF) 161

audio frontend (AFE) 476, 508

Audio Front-End for Speech Recognition (AFE/SR) 450

audio output, over I²S

application, coding 93-102

application, testing 103

audio player

application, coding 180-201

developing 170

feature list 166-168

features 203

Graphical user interface (GUI), designing 170-178

IDF project, creating 178-180

solution architecture 169, 170

test cases 202, 253
testing 201
troubleshooting 203, 204
AutoML tools 486
AWS IoT
application, coding 328-337
application, testing 337, 338
developing 322, 323
hardware setup 324
IDF project, configuring 327, 328
troubleshooting 339
AWS IoT Core 321, 323
AWS IoT Device Defender 323
AWS IoT Device SDK 323
AWS IoT endpoint 328, 358
AWS IoT ExpressLink 323
AWS IoT Greengrass 323, 376
AWS IoT thing
creating 324-326

B

baby monitor
application, developing 507-523
features 498, 528

IDF project, creating 504-506
implementation 500
ML model, generating 500-504
solution architecture 498, 499
testing 523-527
troubleshooting 528
bias 432
BME280 72
Board Support Package (BSP) 179, 383, 461, 474
bouncing 69

C

C/C++ 12
Certificate Signing Request (CSR) 296
claiming methods, RainMaker
assisted claiming 296
host-driven claiming 296
self-claiming 296
cmake tool
reference link 22
communicating, over MQTT 228
application, coding 232-242
application, testing 242, 243
MQTT broker, installing 229, 230

project, creating 230, 231
troubleshooting 244
confusion matrix 470
coreHTTP 34, 323
coreJSON 34
coreMQTT 34
custom partitions
documentation link 92

D

data augmentation 485
data bias 485
data diversity 485
Data Forwarder 466
datasets groups
test 432
train 432
validation 432
data, sharing over secure MQTT 302-304
application, coding 305-314
application, testing 314-318
project, creating 304, 305
troubleshooting 318
data visualization 339, 486

debugging
applications 44-50
Deep Learning (ESP-DL) 161
development platforms and frameworks, ESP32
Arduino Core for ESP32 12
ESP-IDF 11
PlatformIO 12
programming language 12
device provisioning protocol (DPP) 215
Digital-Analog Converter (DAC) 90
Digital Signal Processing (DSP) 444
Digital Signature (DS) 270, 271
Direct Memory Access (DMA) 86
DS peripheral
operation phase 271
setup phase 271

E

Edge Impulse
overview 466
Edge Impulse project
cloning 467-471
Edge Optimized Neural (EON) Compiler 470
Elliptic Curve Digital Signature Algorithm (ECDSA) 269

EON Tuner 466

ESP32 1

development platform and framework 11, 12

features 8, 9

ML model, using on 472

provisioning, on Wi-Fi network 215

real-time operating system (RTOS) 13

RESTful server, running on 244

ESP32-C3-DevKitM-1 11

ESP32 product family 1, 2, 6, 7, 8, 14

ESP32-C2 10

ESP32-C3 11

ESP32-C6 11

ESP32-H2 11

ESP32-S2 10

ESP32-S3 10

ESP32 series 8-10

ESP32 project

application, coding 53, 54

creating 51-53

unit tests, adding 55, 56

unit tests, running 56-58

ESP32, provisioning on Wi-Fi network 215

ESP-IDF application, coding 217-225

ESP-IDF application, testing 225-228
ESP-IDF project, creating 215, 216
troubleshooting 228
ESP32-S3-BOX-Lite 10
ESP32 security features 268
Digital Signature (DS) 270, 271
ESP Privilege Separation 271, 272
Secure Boot v1 269
Secure Boot v2 269, 270
ESP-ADF 445
ESP-DL 445
ESP-DSP 444
espefuse.py tool 25
ESP-IDF 16, 17
ESP32 application 17-23
ESP-IDF Terminal 23-25
installation 16
ESP-IDF Components library 74, 76, 114, 160-163
ESP-IDF methods, for Wi-Fi provisioning
Easy Connect 215
SmartConfig 215
unified provisioning 215
ESP-IDF Programming Guide
reference link 161

ESP-IDF Tools Installer 45

ESP Privilege Separation 271, 272

ESP-Prog debugger 44

ESP RainMaker 161, 268, 289, 291, 296, 298

reference link 161, 381

ESP RainMaker platform

reference link 499

Espressif

documentation link 269

frameworks 161, 162

libraries 161, 162

Espressif IoT Development Framework (ESP-IDF) 11

Espressif IoT Solution 161

Espruino 12

espsecure.py tool 25

ESP SoftAP Provisioning 215

ESP-SR 445

esptool.py tool 25

ESP-WHO 445, 491, 492

GitHub repository 492

F

File Allocation Table (FAT) 84

firmware, upgrading from HTTPS server 273

application, testing 287
project, coding 276-286
project, creating 274, 276
server, preparing 273, 274
troubleshooting 287
FlatBuffers 136, 137
application, coding 138-146
application, testing 147
project, creating 137, 138
reference link 139
flatc 138, 139
FreeRTOS 13, 34
producer - consumer project, coding 38-41
producer-consumer project, creating 34-37
producer- consumer project, running 41-43
reference link 13

G

General-Purpose Input/Output (GPIO)
application, coding 66-70
application, troubleshooting 71
driving 62
LED, turning on/off with button 63, 64
project, creating 64, 65

GoogleTest 51

reference link 56

Grafana

visualizing with 339

Grafana dashboard

creating 349-353

Grafana workspace

creating 346-348

graphical user interface (GUI) 170

designing 148

developing, on ESP32 104, 105

graphical user interfaces, developing on LCD

application, coding 106-110

application, testing 110

project, creating 105

H

Heap Memory Allocation

reference link 131

hyperparameter tuning 433

I

IDF component

coding 386-394

creating 385
IDF Component Manager 115, 116
IDF Component Registry 113, 115
reference link 115, 161
URL 115
Image Processing Framework (ESP-WHO) 161
Imagimob
reference link 490
Impulse 466
inference 431
running, on ESP32 434, 435
running, on IoT devices 433
inference, running on ESP32 434, 435
application, coding 436-443
application, testing 443, 444
project, creating 435, 436
initialization vector (IV) 271
input device
reference link 156
Instruction RAM (IRAM) 69
Interface Definition Language (IDL) 139
Inter-IC Sound (I²S)
audio player, developing 91-93
audio output 90

Inter-Integrated Circuit (I2C)
sensors, interfacing over 71
Internet of Things (IoT) 1
inter-processor call (IPC) tasks 42
IoT Security Foundation 6
URL 6
IoT solutions
autonomous operation 3
backend system 4
communication 4
connectivity 3
device firmware 4
device hardware 4
end user applications 4
identification 3
interoperability 3
scalability 3
security 3, 5
structure 4, 5

J

JavaScript Object Notation (JSON) 121
Joint Test Action Group (JTAG) debugging 44-50
JTAG probe/adapter 44

K

Kaggle datasets

reference link 485

L

Light and Versatile Graphics Library (LVGL) 104, 148

application, coding 150-160

application, testing 160

GUI, designing 148, 149

project, creating 149, 150

Liquid-Crystal Display (LCD)

graphical user interfaces, developing on 104

LittleFS 114

application, coding 116-120

application, testing 121

project, creating 115, 116

local Wi-Fi

connecting to 206, 207

ESP-IDF application, coding 208-213

ESP-IDF application, testing 213, 214

ESP-IDF project, creating 207, 208

troubleshooting 214

low-rate wireless personal area network (LR-WPAN) 11

M

Machine Learning (ML) 430

approaches, to solving computing problems 430, 431

reinforced learning 432

supervised learning 431

unsupervised learning 431

Master-In-Slave-Out (MISO) 79

Master-Out-Slave-In (MOSI) 79

Matplotlib 486

MbedTLS 299

Mel-Frequency Cepstral Coefficient (MFCC) 469

Message Queue Telemetry Transport (MQTT) 228

Microcontroller Unit (MCU) 4

MicroPython 12

Miniz 129

application, testing 136

project, coding 130-135

project, creating 129, 130

ML model, using on ESP32 472

application code 473-483

application, testing 484

model library 472, 473

troubleshooting 484, 485

MLOps 486

model parameters 433

Mongoose OS 13

Mosquitto 229

reference link 230

mos tool 13

MQTT broker

installing 229, 230

MultiNet 450

multisensor development 400

application, coding 413-415

GUI, adding 407-413

sensor node, adding 401-407

multisensor hardware

setting up 382, 383

N

near-field communication (NFC) 215

Nlohmann-JSON 121, 179, 188, 216

application, coding 122-128

application, testing 128

project, creating 121, 122

node claiming 303

Non-Volatile Storage (NVS) access 94

NumPy 486

NuttX 13

O

One-Time Programmable (OTP) 269

Open Neural Network Exchange (ONNX) 466

openssl tool 287

Organic Light-Emitting Diode (OLED) 104

overfitting 433

over-the-air (OTA) update techniques 272, 273

- firmware, upgrading from HTTPS server 273
- RainMaker, utilizing for OTA updates 288

P

pio tool 33

PlatformIO 12, 25, 26

- Hello world application 26-31
- PlatformIO Terminal 32, 33

PlatformIO Library Manager

- reference link 53

PlatformIO registry

- reference link 30

plug

- testing 415-417

plug development

application, coding 399
plug node, adding 395-399
plug hardware
setting up 382
prediction 433
Processor Instruction Extensions (PIEs) 444
proof-of-possession (POP) 222
Pseudo-RAM (PSRAM) 130
pseudostatic-RAM (PSRAM) 8

Q

Qeexo AutoML
reference link 490
quantization 433

R

Rainmaker
configuring 288, 289
platform 287
Rainmaker mobile application
using 419-425
RainMaker platform 267, 268, 287, 288, 292-294
RainMaker, utilizing for OTA updates 288
application, coding 290-295

application, testing 296-301
configuration 288, 289
project, creating 289, 290
troubleshooting 302
real-time operating system (RTOS) 13
reinforced learning 432
Representational State Transfer (REST) 206, 244
Resource Acquisition Is Initialization (RAII) 118
RESTful server, running on ESP32 244, 245
application, coding 247-252
application, testing 253
project, creating 245
RESTful services
application, coding 255-261
application, testing 261, 262
consuming 253, 254
project, creating 254
troubleshooting 262
Rivest-Shamir-Adleman-Probabilistic Signature Scheme (RSA-PSS) 269
RTOS options, for ESP32
Amazon FreeRTOS 13
FreeRTOS 13
Mongoose OS 13
NuttX 13

Zephyr 13

Rust 12

S

SD card, integrating over SPI 78

application, coding 81-88

application, testing 89

application, troubleshooting 90

project, creating 81

storage, adding 79, 80

Secure Boot

v1 269

v2 269, 270

SensiML Analytics

reference link 490

sensors, interfacing with I2C 71

application, coding 75-78

application, troubleshooting 78

multisensor, developing 72, 73

project, creating 73, 74

Serial Peripheral Interface (SPI) 61, 62, 78

SD card, integrating over 78

Simple Network Time Protocol (SNTP) 391

Single Instruction, Multiple Data (SIMD) 444

smart home solution

Amazon Alexa, integrating 380

automation 380

common libraries, preparing 384

feature list 380

features, adding 426, 427

implementing 384

mobile application 380

multisensor 380

multisensor application, testing 417, 418

multisensor, developing 400

plug 380

plug, developing 394

plug, testing 415-417

Rainmaker mobile application, using 419-425

software packages 384

solution architecture 381

testing 415

troubleshooting 425

SoC/module selection

checklist 7

soft access point (softAP) 215

solution architecture, smart home solution 381, 382

multisensor hardware, setting up 382, 383

software architecture 383

speech recognition application

coding 447- 457

developing 444, 445

project, creating 446, 447

testing 457-461

troubleshooting 461

Speech Recognition (SR) 445

SquareLine Studio 148

download link 148

Static RAM (SRAM) 130

Station (STA) mode 207

supervised learning 431

System-On-Chip (SoC) 4

T

TensorFlow 433

TensorFlow Lite for Microcontrollers (TFLM) 433

tensors 434

Test-Driven Development (TDD) 51

Thin Film Transistor (TFT) 104

Thread 529

Timestream database

creating 340-346

TinyML development

anomaly detection 492, 493

image processing, with ESP32-S3-EYE 491, 492

Netron App, using 487-490

ONNX format 490

stages, reviewing 485, 486

tinyML Foundation 490

TinyML pipeline 430, 432, 433, 462

data collection and preprocessing 432

inference, running on IoT device 433

model, designing and training 432

model, optimizing 433

model, preparing for deployment 433

transfer learning 486

Trusted Execution Environment (TEE) 272

TSL2561 72

U

Unified Provisioning 215, 217, 263

unit tests 51

adding 55, 56

running 56-58

Unity framework 51

unsupervised learning 431

user-node mapping 298

V

visualizing, with Grafana 339

dashboard, creating 349-353

Timestream database, creating 340-346

troubleshooting 354

workspace, creating 346-348

voice user interface (VUI) 354

W

wake word 355, 445, 450, 454

Wi-Fi Protected Access 2/Pre-Shared-Key (WPA2/PSK) 211

wireless personal area networks (WPANs) 11

wolfSSL 299

X

X509 root certificates 299

Z

Zephyr 13

Zigbee. Matter 529

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/9781803237688

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