EXPERT INSIGHT

Developing IoT Projects with ESP32

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

Second Edition

Vedat Ozan Oner

packt

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

packt

BIRMINGHAM—MUMBAI

Developing IoT Projects with ESP32

Second Edition

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 Projecís wiíh ESP32, Firsí Ediíion*, published by *Packí* , 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.

Hearí-felí íhanks ío my wife for her releníless supporí and paíience. Her íeas kepí my mind fresh during íhe 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 ío God, ío Eyiíope fior íhe reminders, ío Ibukun, Audrey, and Tinuke fior always being íhere, ío Vedat for the opportunity to be part of this journey, and finally to Manish, Namrata, and the incredible team íhaí worked on íhis 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 ío congraíulaíe Vedaí fior íhe greaí job he did. Wriíing a íechnical book is a big challenge and Vedaí really uíilized his knowledge and experience in real projecís. Also, I would like ío íhank Packí Publishing fior all íhe supporí íhaí íhey gave during íhis 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`

Table of Contents

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.

> "*Acíion wiíhouí síudy is faíal. Síudy wiíhouí acíion is fuíile.*"
>
> *- 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

Chapíer 1, Iníroducíion ío IoT Developmení and íhe ESP32 Plaíform, discusses IoT technology in general and introduces the ESP32 platform in terms of both hardware and software.

Chapíer 2, Undersíanding íhe Developmení 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.

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

Chapíer 4, Employing Third-Paríy Libraries in ESP32 Projecís, talks about different methods of importing third-party libraries with examples. LVGL is one of the libraries discussed in this chapter.
Chapíer 5, Projecí – Audio Player, is the first reference project in the book with audio, graphics, and button interactions to engage its users.

Chapíer 6, Using Wi-Fi Communicaíion fior Conneciiviíy, shows how to communicate over different application layer protocols, such as MQTT and REST, after connecting to a local Wi-Fi network.

ESP32 by giving examples of secure firmware updates and secure communication techniques. ESP
Chapíer 7, ESP32 Securiíy Feaíures fior Producíion-Grade Devices, explores the security features of 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.conf.sample
     /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 Projecís wiíh ESP32, Second Ediíion*, 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:

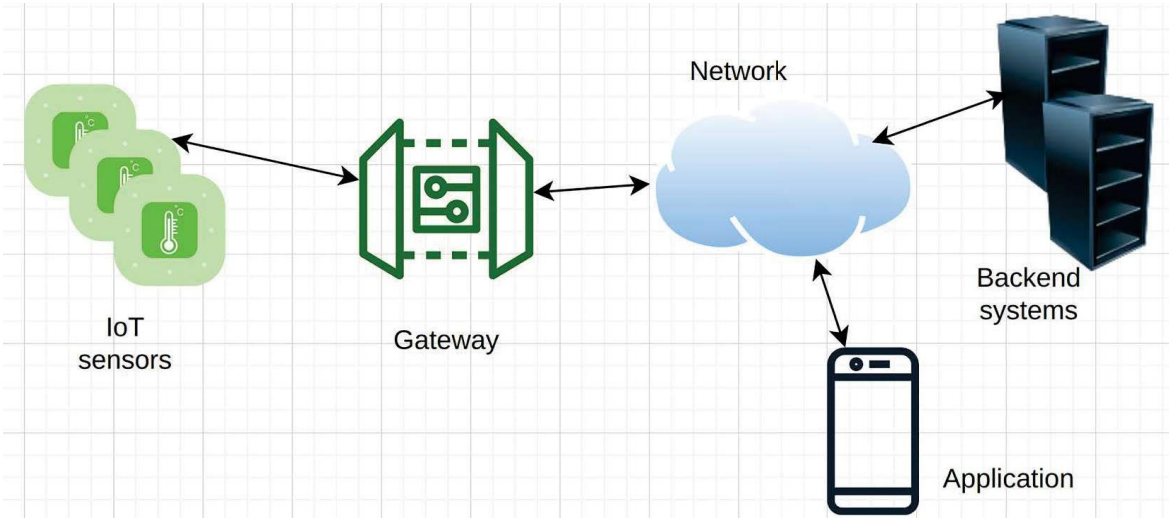

Figure 1.1: Basic structure

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:

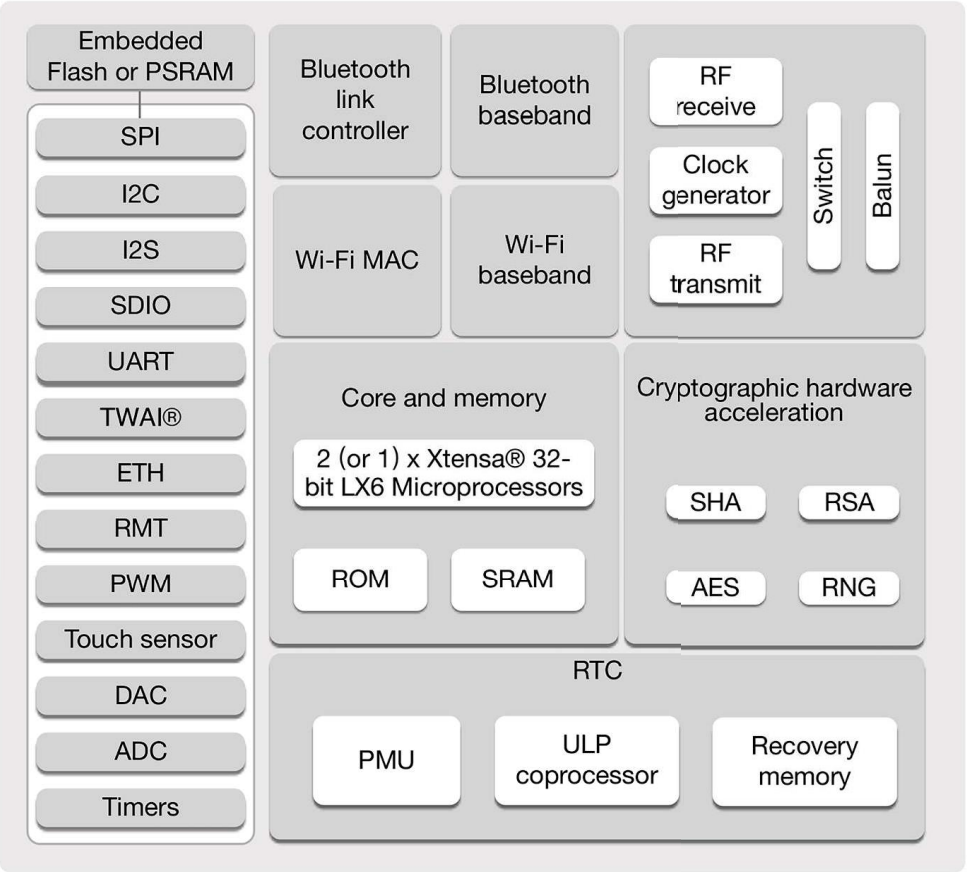

Figure 1.2: The functional block diagram of ESP32 series chips (source: ESP32 Series Datasheet, Version 3.9, Espressif Systems)

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.

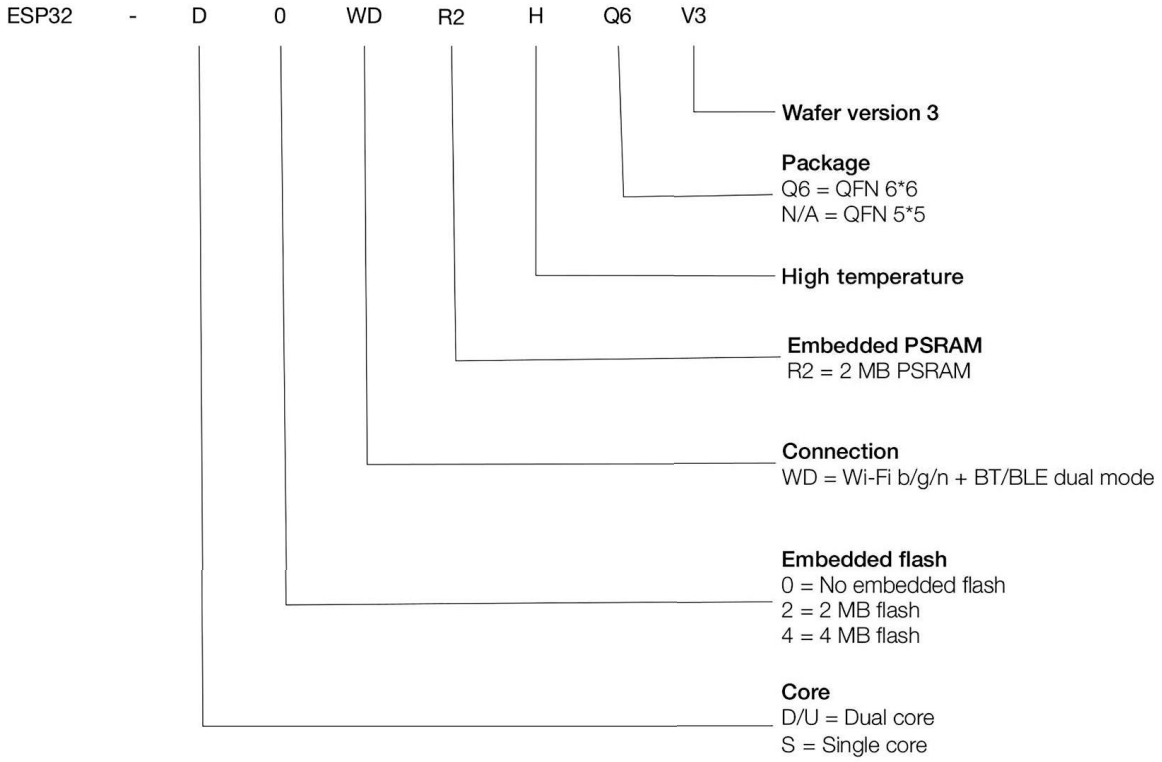

Figure 1.3: ESP32 part numbering convention (source: ESP32 Series Datasheet, Version 3.9, Espressif Systems)

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 JTAG/serial debug
unit
```

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

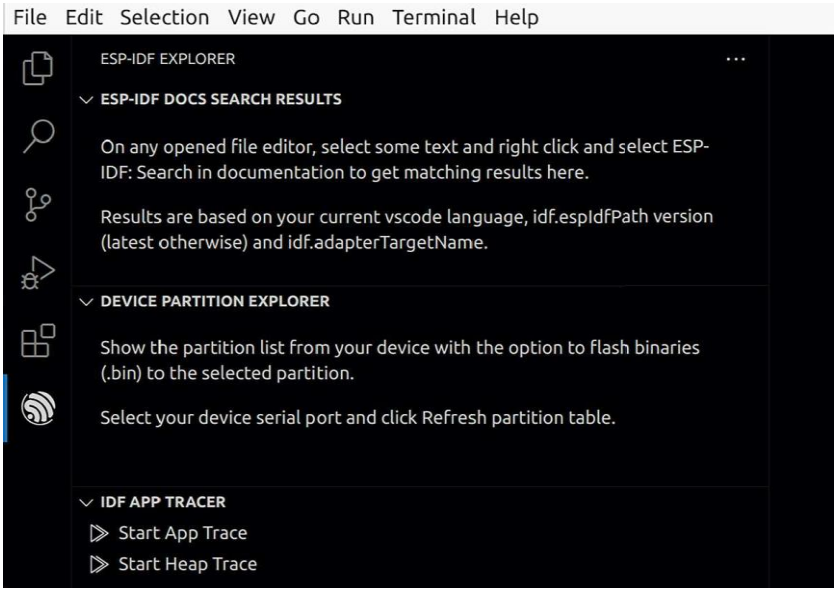

Figure 2.1: Espressif IDF in VSCode

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 *Círl* + *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.

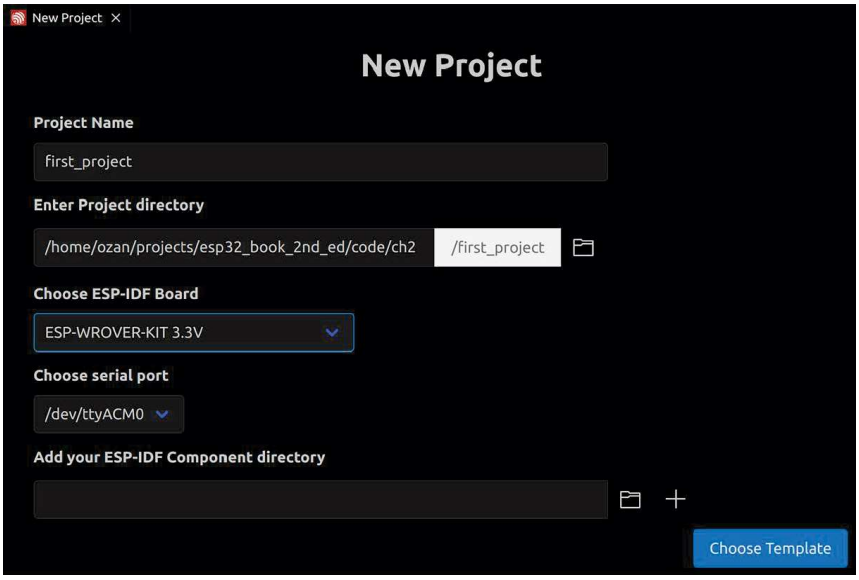

Figure 2.2: New project

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.

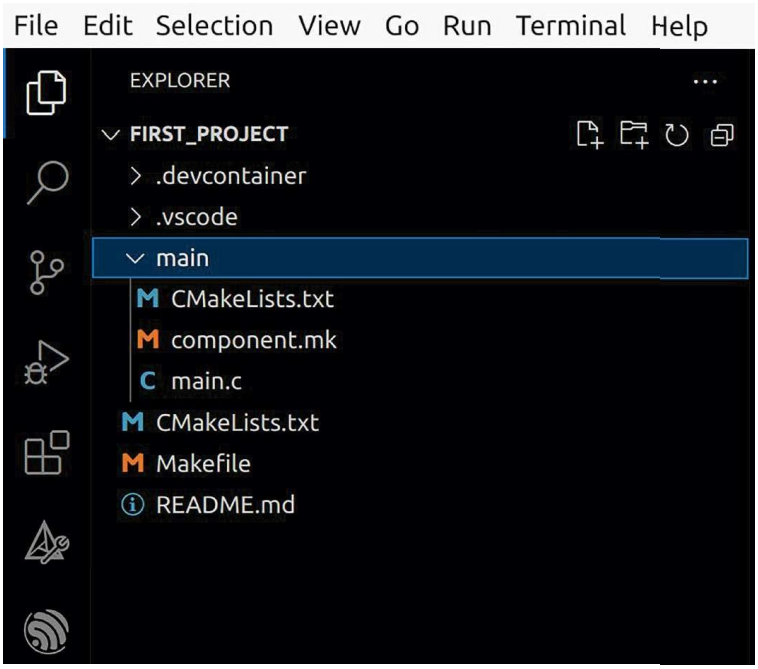

Figure 2.4: Explorer with VSCode files

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 *Círl* + *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.

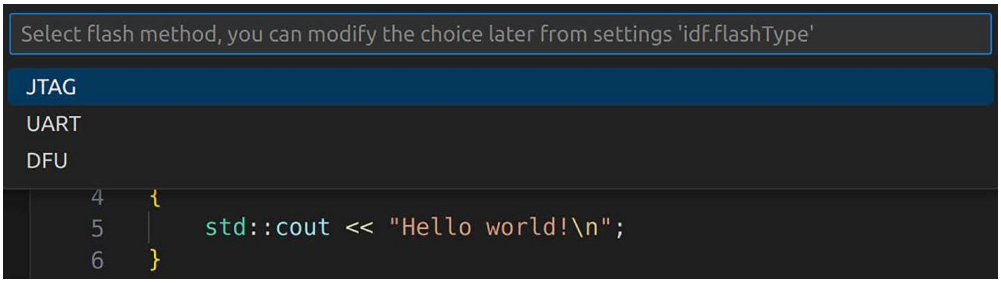

Figure 2.5: Selecting the flash method

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 *Círl +]*.

Let's go back to the application code in *síep 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 *síep 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:

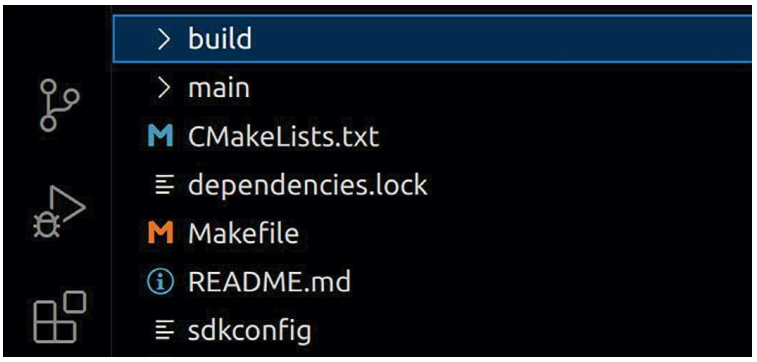

Figure 2.7: The files and directories after compilation

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:

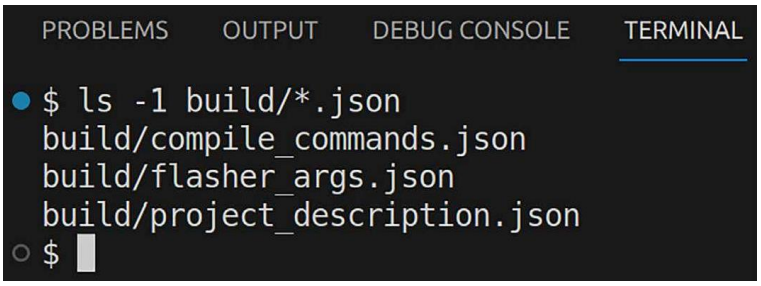

Figure 2.8: JSON files in the build directory

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 *Círl* + *E* + *G*:

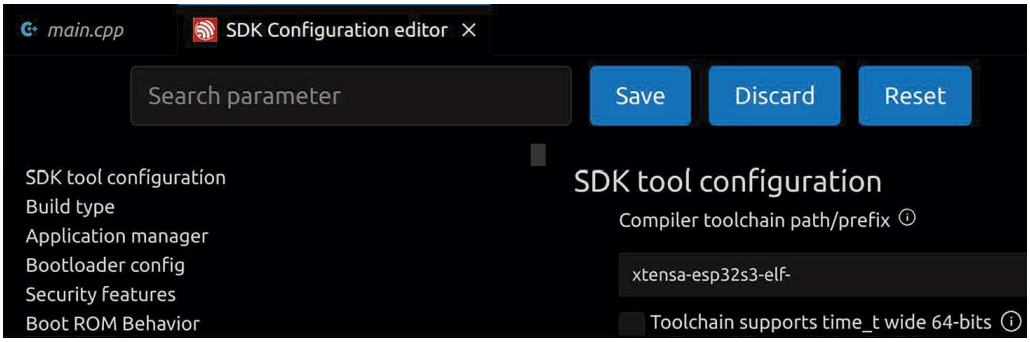

Figure 2.9: SDK configuration (sdkconfig)

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 *Círl* + *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/first_project/build
Executing "ninja clean"...
[0/1] Re-running CMake...
-- Project is not inside a git repository, or git repository has no
commits; will not use 'git describe' to determine PROJECT_VER.
-- Building ESP-IDF components for target esp32s3
-- Project sdkconfig file <your_directory>/ch2/first_project/sdkconfig
-- App "first_project" version: 1
```

```
<The rest of the build and flashing logs. Next comes the application
output.>
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 *Eníer* 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 build/first_project.bin
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 0x00010000 in 4.5 seconds
(effective 769.1 kbit/s)...
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 *PlaíormIO*.

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 (*Círl+Shií+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:

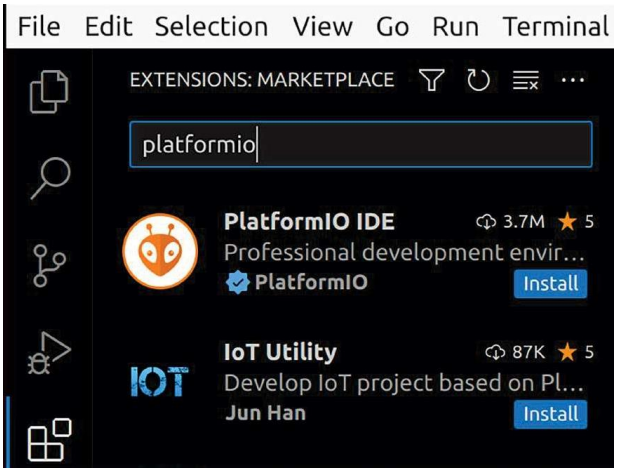

Figure 2.10: Installing PlatformIO

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.

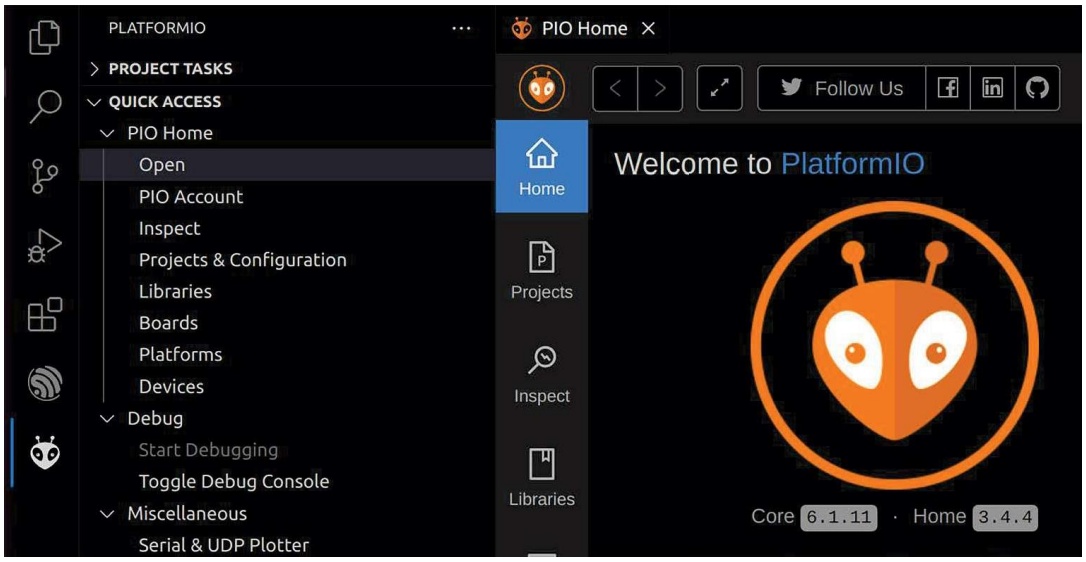

Figure 2.11: PlatformIO Home

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

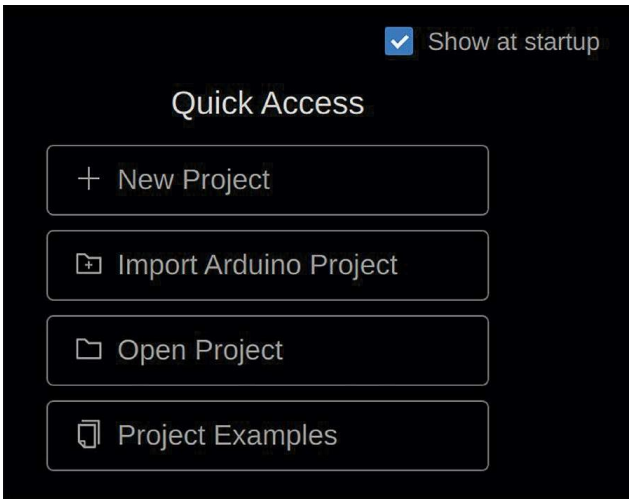

Figure 2.12: Quick access buttons on the PlatformIO Home screen

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.

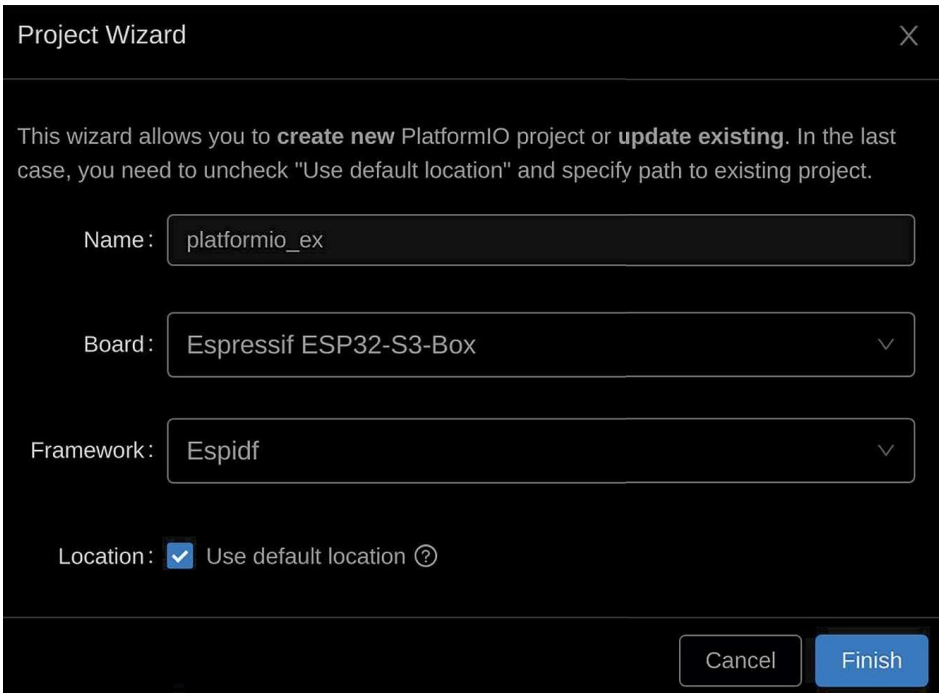

Figure 2.13: PlatformIO Project Wizard

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

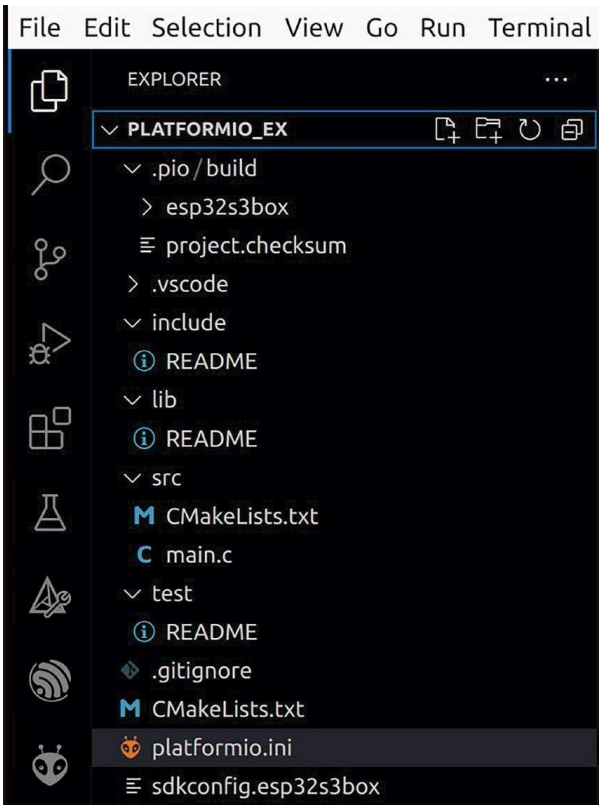

Figure 2.14: Project directory structure

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.

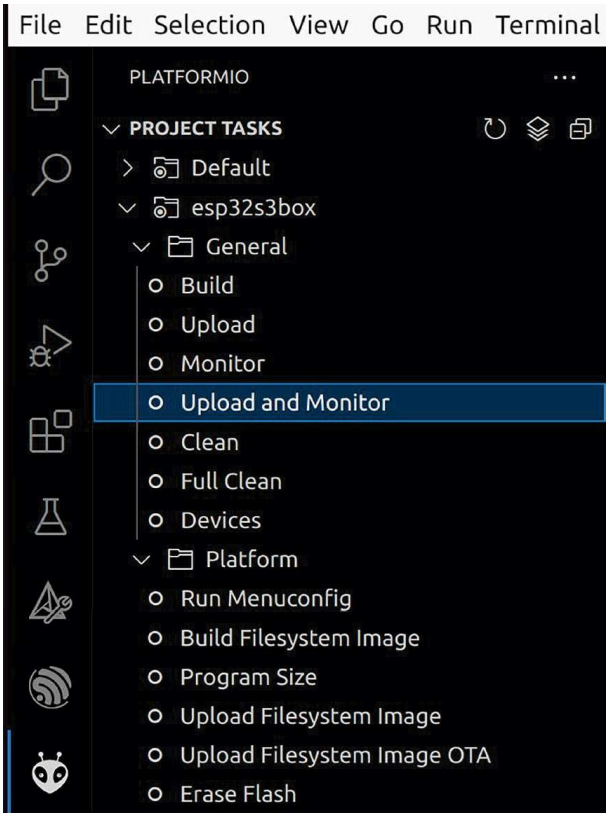

Figure 2.15: PlatformIO project tasks

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:

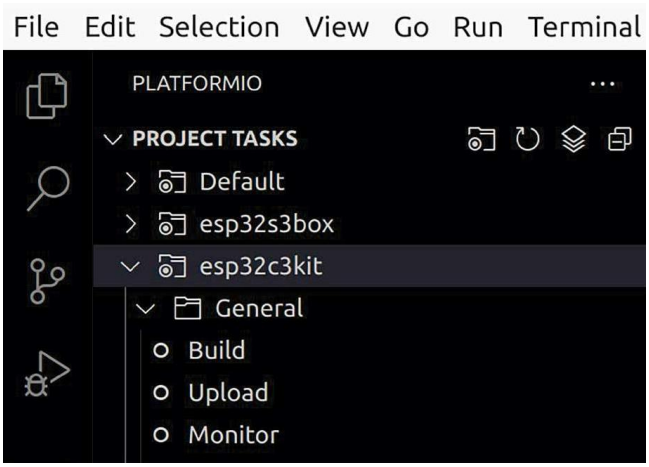

Figure 2.17: New environment under PROJECT TASKS

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 *Eníer* key to display the `pio` options.

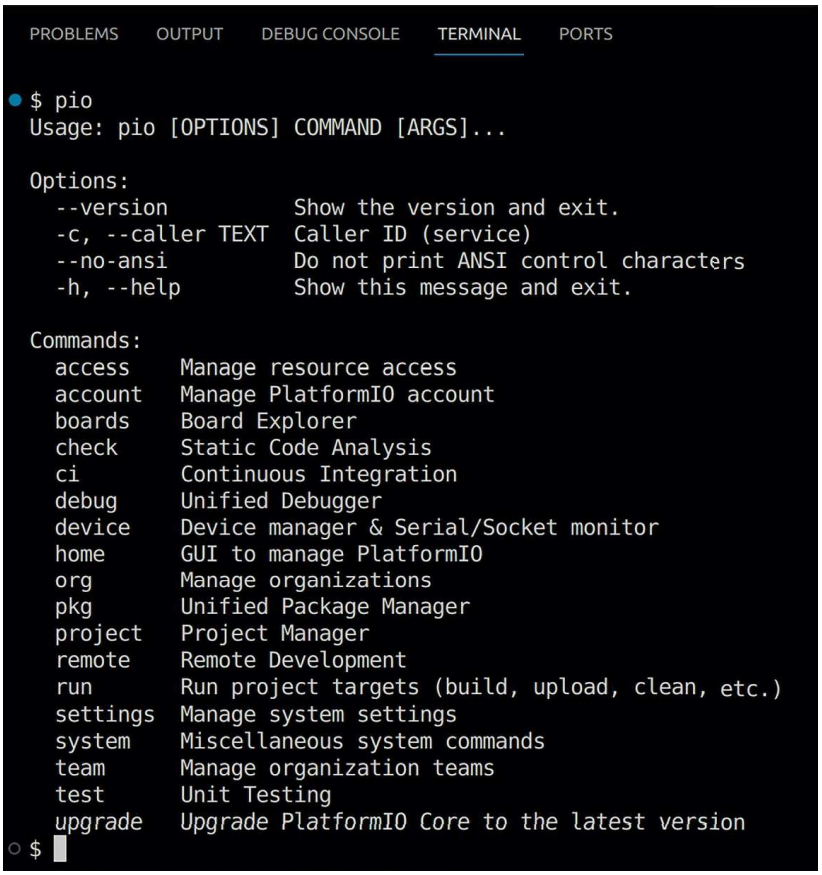

Figure 2.19: PlatformIO Terminal and the pio command-line tool

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

```
$ pio run -t upload -e esp32c3kit
Processing esp32c3kit (platform: espressif32; board: esp32-c3-devkitm-1;
framework: espidf)
-----------------------------------------------------------------------
Verbose mode can be enabled via `-v, --verbose` option
CONFIGURATION: https://docs.platformio.org/page/boards/espressif32/esp32-
c3-devkitm-1.html
PLATFORM: Espressif 32 (5.1.1) > Espressif ESP32-C3-DevKitM-1
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:

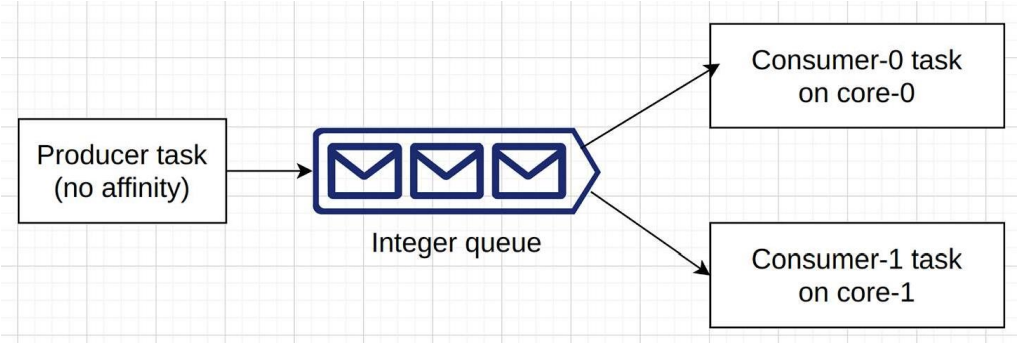

Figure 2.20: Producer-consumer pattern

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**.

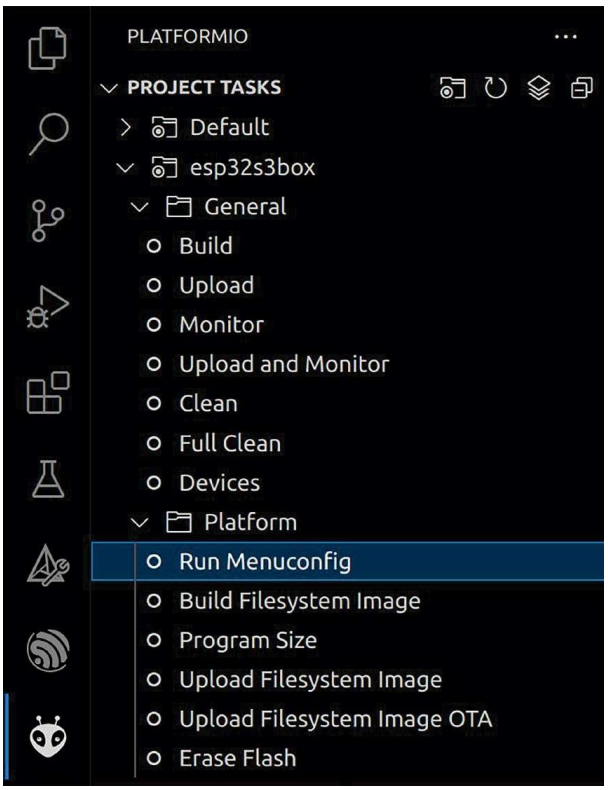

Figure 2.21: Running menuconfig

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`

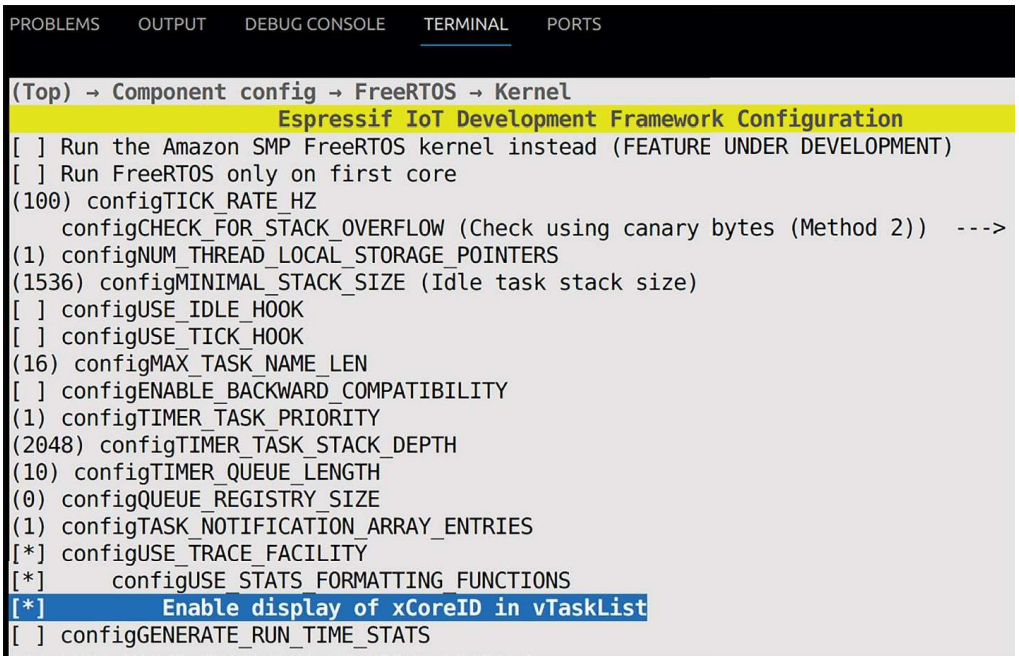

Figure 2.22: Configuring FreeRTOS in menuconfig

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).

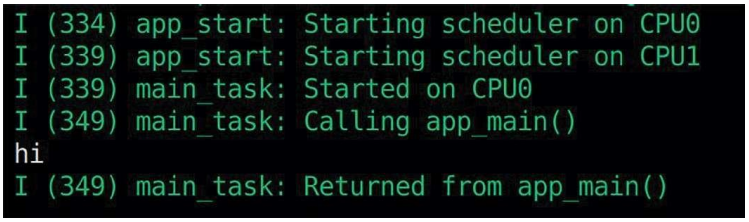

Figure 2.23: The serial monitor output when the application is configured successfully

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, sizeof(int))};
    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, 5, 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_address)
```

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", 4096, (void *)0,
                            5, nullptr, 0);
    xTaskCreatePinnedToCore(consumer, "consumer-1", 4096, (void *)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, stack_depth,
    function_parameters, priority, task_handle_address, task_affinity)
```

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, portMAX_DELAY);
            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, portMAX_DELAY);
            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       0
producer           B   5       3500        7       -1
esp_timer           S   22      3432        3       0
ipc1                B   24      884          2       1
consumer-0      B 5       3412        8       0
ipc0                B   24      892         1       0
```

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:

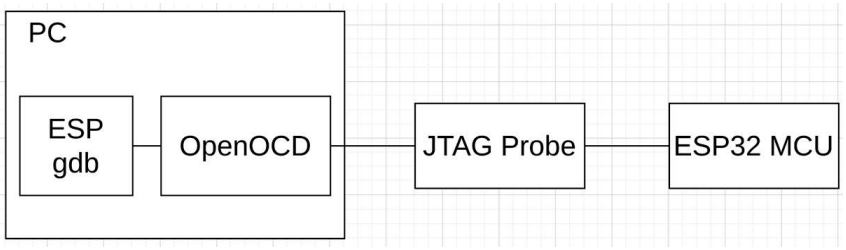

Figure 2.24: JTAG debugging

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:

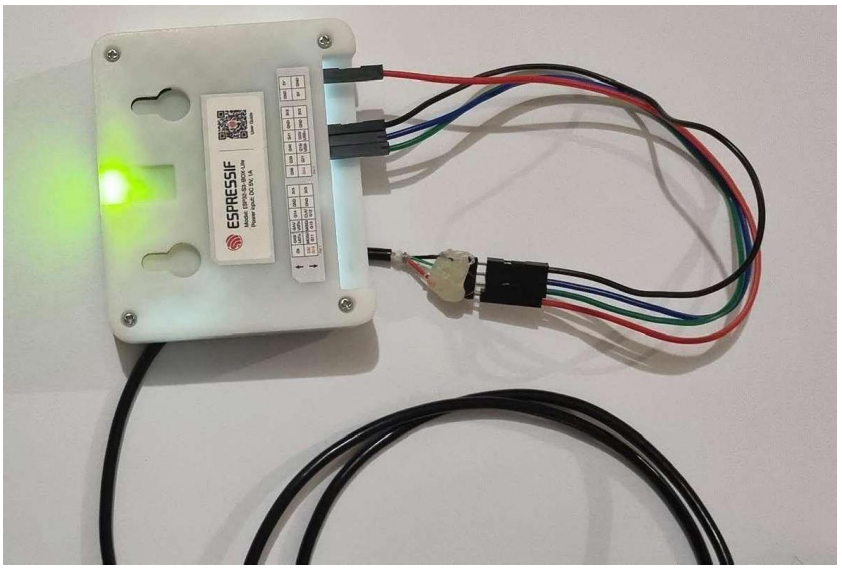

Figure 2.25: Built-in JTAG

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]... [COMMAND2 [ARGS]...]...
ESP-IDF CLI build management tool. For commands that are not known
to idf.py an attempt to execute it as a build system target will be
made.
<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 directory>/debugging_ex
$ 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" INCLUDE_DIRS ".")
```

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" "-ggdb3" "-g3" APPEND)
```

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, 24.1% used)
      .text size:   86403 bytes
   .vectors size:    1027 bytes
Used stat D/IRAM:   13941 bytes ( 158987 remain, 8.1% used)
      .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 padded larger)
```

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/esp32s3-builtin.cfg"
Executing action: openocd
OpenOCD started as a background task 477341
Executing action: post_debug
Open On-Chip Debugger v0.11.0-esp32-20221026 (2022-10-26-14:47)
Licensed under GNU GPL v2
<more logs>
```

15. Start another ESP-IDF command-line terminal as we did in *síep 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.

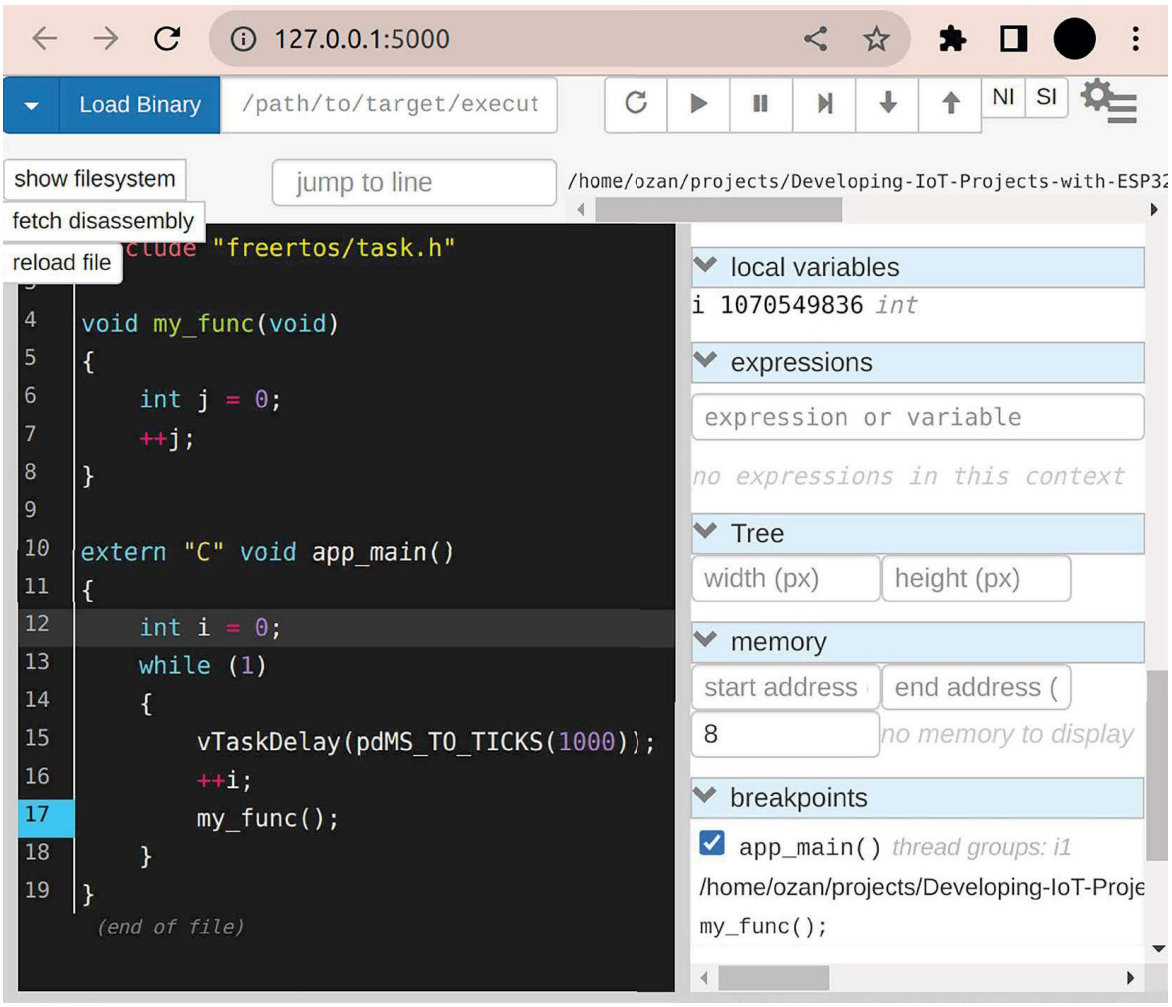

Figure 2.26: Web-based debugger

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.

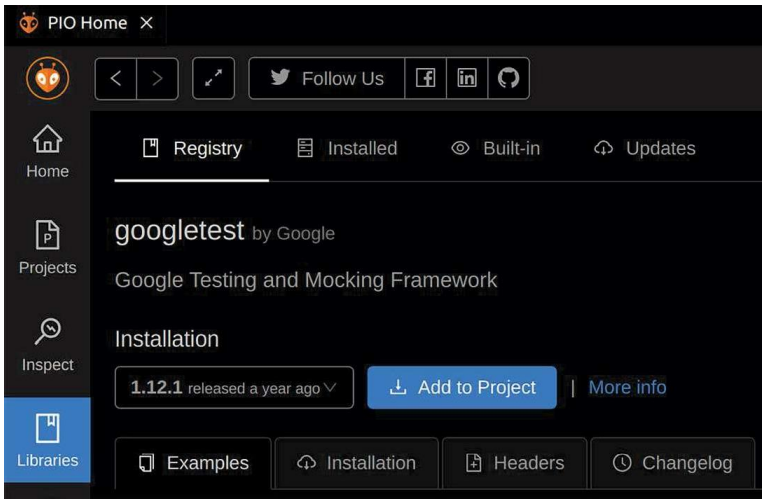

Figure 2.27: PlatformIO Registry

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 *compaíible* 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.

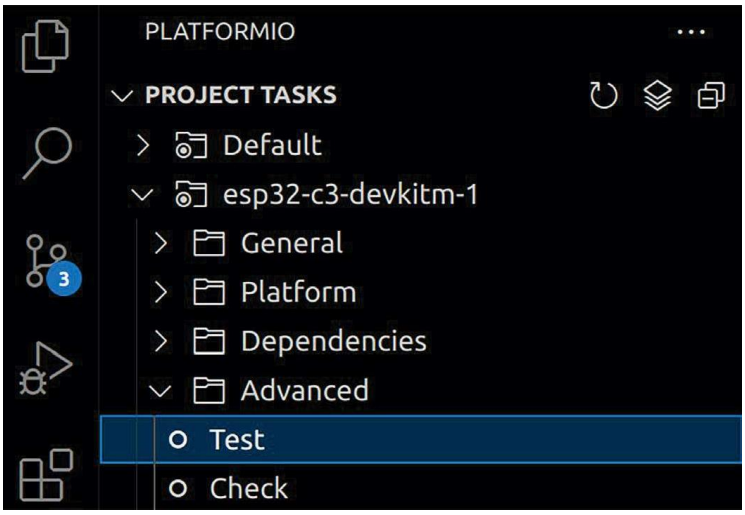

Figure 2.28: PlatformIO unit testing

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.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Processing * in esp32-c3-devkitm-1 environment
--------------------------------------------------------------------------------
Building & Uploading...
Library Manager: Installing google/googletest @ 1.12.1
Unpacking  [####################################]  100%
Library Manager: googletest@1.12.1 has been installed!

Warning! Please install `99-platformio-udev.rules`.
More details: https://docs.platformio.org/en/latest/core/installation/udev-r
ules.html

Testing...
If you don't see any output for the first 10 secs, please reset board (press
 reset button)

LightTest.turns_on_light        [PASSED]
LightTest.turns_off_light       [PASSED]
------------ esp32-c3-devkitm-1:* [PASSED] Took 35.21 seconds ------------

================================ SUMMARY ================================
Environment         Test    Status    Duration
------------------  ------  --------  ------------
esp32-c3-devkitm-1  *       PASSED    00:00:35.205
================ 2 test cases: 2 succeeded in 00:00:35.205 ================
*  Terminal will be reused by tasks, press any key to close it.
```

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. *Chapíer 20* discusses FreeRTOS on multi-core systems such as ESP32.
- *Embedded Sysíems Archiíecíure – Second Ediíion*, 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. *Parí 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`