EXPERT INSIGHT

Developing IoT Projects with ESP32

Discover the IoT development ecosystem with ESP32 to create production-grade smart devices

Second Edition

Vedat Ozan Oner

packt

Developing IoT Projects with ESP32

Early Access Publication: Developing IoT Projects with ESP32

Early Access Production Reference: B18447

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80323-768-8

www.packt.com

Table of Contents

Developing IoT Projects with ESP32, Second Edition: Discover the IoT development ecosystem with ESP32 to create production-grade smart devices

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: Overview of ESP32 / Introduction to IoT development and the ESP32 platform
2. Chapter 2: Understanding the development tools to create the first application
3. Chapter 3: Using ESP32 peripherals to drive external devices
4. Chapter 4: Employing 3rd-party libraries in ESP32 projects
5. Chapter 5: Project - Audio player
6. Chapter 6: Using WiFi communication for connectivity
7. Chapter 7: Learning ESP32 security features for production-grade devices
8. Chapter 8: Connecting to cloud platforms and using services
9. Chapter 9: Project - Smart home
10. Chapter 10: TensorFlow Basics and TinyML
11. Chapter 11: Developing on EdgeImpulse platform

12. Chapter 12: Project - Baby monitor

1 Introduction to IoT development and the ESP32 platform

Espressif ESP32 is a powerful tool in the toolbox of a developer for many types of **Internet of Things** (**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 the problem, we need to understand the domain, and we need to know the available tools and their features that are important for that specific problem 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.

In this chapter, I will discuss the technology, IoT, 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 helps you to understand the big picture from the 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
- ESP32 family
- Development Platforms & Frameworks
- RTOS options

Technical requirements

In this book, we are going to have 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. However, there are several chapters where Python3 is required to support the subject.

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

- ESP32-S3-BOX-Lite
- ESP32-C3-DevKitM-1
- ESP-Prog
- [todo] The list of other sensors

You can visit the official Espressif website for more information about all the development kits and products, here: https://www.espressif.com/en/products/devkits

The product page also directs to the vendor websites to buy the development kits and other hardware. The total cost of the hardware that you need to have in order to try the examples in this book is around [todo]

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 the (CIA triad).

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 still can apply an organized approach to develop 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 both 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 deeper 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. 10 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.

The following diagram depicts the general structure of IoT solutions:

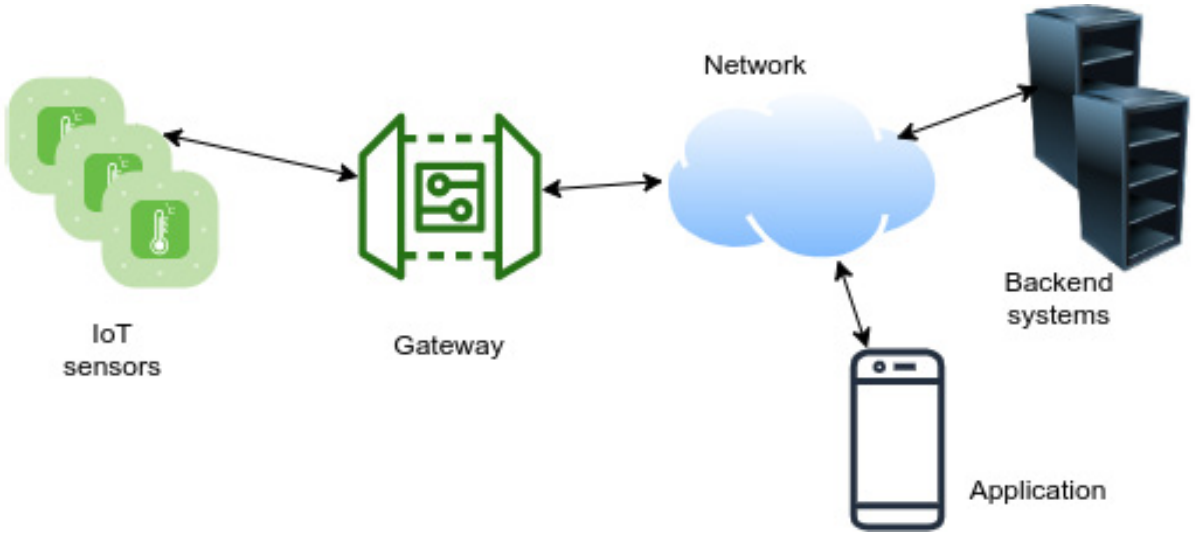

Figure: Basic structure

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

IoT security

One important consideration that remains is security. Actually, it is all about security. I cannot overemphasize its importance whatever I write. IoT devices are connected to the real world and any security incident has the potential for serious damage in the immediate environment, let alone other cybersecurity crimes. Therefore, it should always be in 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 publishing 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.

ESP32 Family

Since the launch of the first ESP32 chip, Espressif Systems has extended the family with new designs for different purposes. They have now 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 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 tick the filter boxes

and the tool lists the hardware matching your requirements: https://products.espressif.com/#/product-selector

The following shows a basic checklist to select 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
- WiFi 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 collect these requirements in many cases. With that, we can look at the different series of SoCs from Espressif Systems.

ESP32 series

This series contains the most common variants of ESP32 chips. Let’s have a quick look at the functional block diagram in its datasheet:

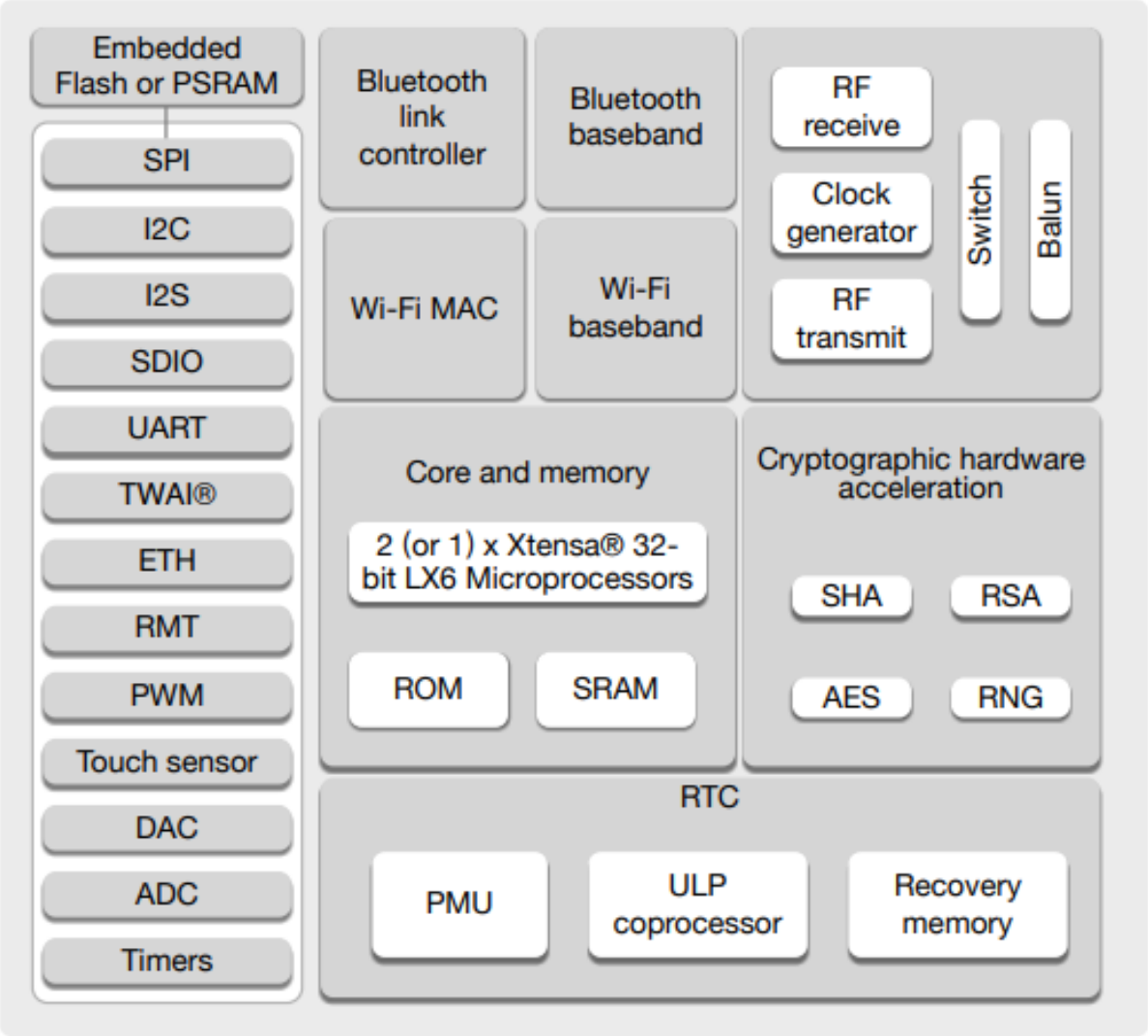

Figure: The functional block diagram of ESP32 series chips. (source: ESP32 Series Datasheet)

The key features are the following:

- It has 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.
- 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)

- Integrated RF components for WiFi (802.11 b/g/n – 2.4GHz) and Bluetooth (BLEv4.2 and BR/EDR) communication
- Cryptographic hardware acceleration
- And low-power management features with real-time clock (RTC) and 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 the 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.

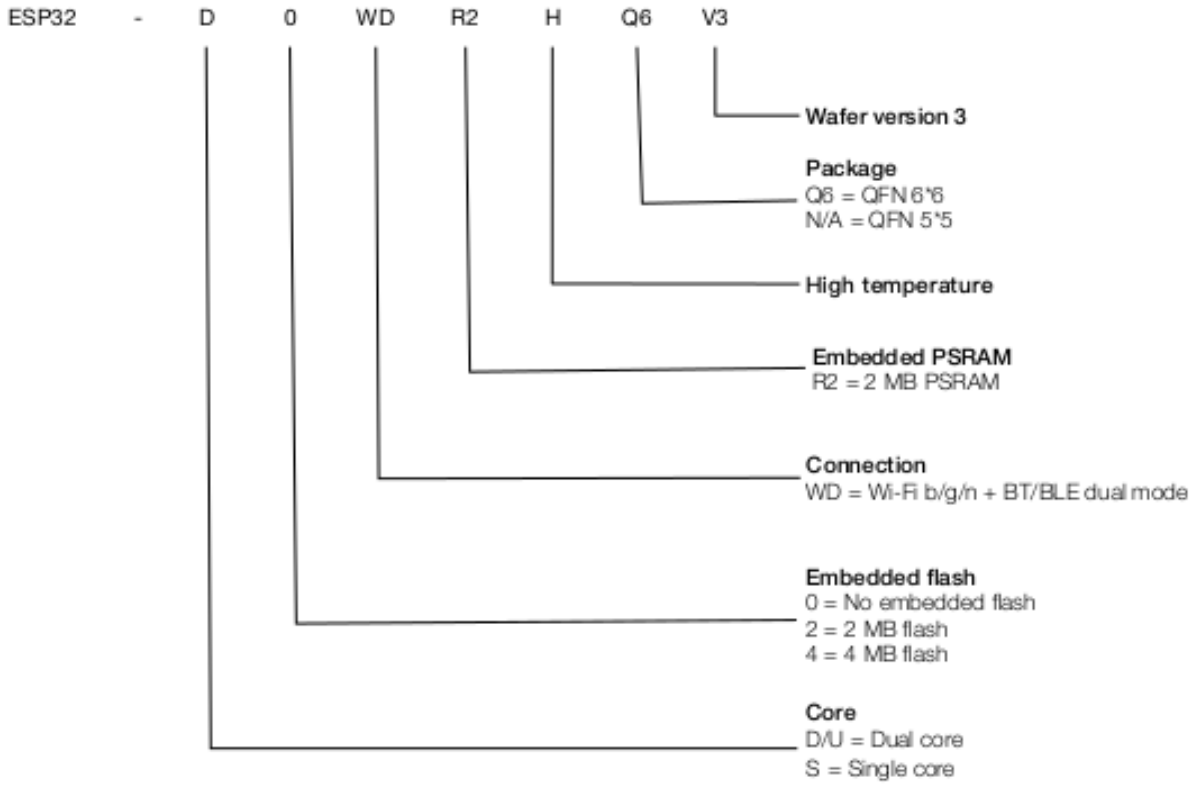

Figure: ESP32 part numbering convention (source: ESP32 datasheet)

The fields in the part number shows the different features of an SoC as you see in the figure. For example, ESP32-D0WDR2-V3 has a dual core, no embedded flash, WiFi and BT/BLE dual mode, 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 there is a newer/better version of this product and Espressif probably will 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 to read the ESP32 series datasheet that is publicly available on the Espressif website to learn more about the SoC features in this series. Especially 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 offer.

Other ESP32 series

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

- ESP32-S2: This series boasts with its 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 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 16MB flash and 8MB 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 WiFi (802.11b/g/n) and BLE5 connectivity with a good enough peripherals for basic IoT applications.
- ESP32-C3: This series shares the same processor with 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 ESP32-C6 series is its connectivity features. In addition to WiFi (802.11b/g/n) and BLE5, it supports WiFi 6 (802.11ax), the new WiFi 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 (WPAN) to be selected as local communication infrastructure in products. Although it is announced, ESP32-C6 series SoCs are not available on the market yet as of writing this book.

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 many hassles 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.

Development Platforms & 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 IDE and install the ESP-IDF extension on it.

> You can find the ESP-IDF 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 to Arduino, you 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, platforms, 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 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 any of them. Rust is another option as programming language to develop IoT applications on ESP32. To be honest, I don’t have much experience with Rust, however, it is getting attention in the embedded world and seems worth trying on ESP32.

RTOS options

Basically, an 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 the 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 officially supported. Amazon FreeRTOS has the necessary libraries to connect to the AWS IoT and other security-related features, such as TLS, OTA updates, secure communication with HTTPS, WebSockets, and MQTT, pretty much everything to develop a secure connected device.

An example of using Amazon FreeRTOS in an ESP32 project is given here: https://docs.aws.amazon.com/freertos/latest/userguide/getting_started_espressif.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 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 is 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.

Summary

In this chapter, we covered all the necessary background information regarding the IoT technology and ESP32 as a hardware platform for developing IoT products.

To add value to our end users, we, as developers, should know the ground. It is not enough to come up with a solution; it has to be the right solution, which requires us to learn more about the technologies and tools available. When it comes to IoT technology, things may become more difficult because an IoT product has several components, starting from

sensor/actuator devices to end user applications, allowing end users to interact with the solution. In this manner, learning ESP32 is an important professional skill to be acquired by an IoT developer.

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.

2 Understanding the development tools

After having a quick overview of Espressif's ESP32 technology in the first chapter, we are now ready for starting with development on 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 acquire the fundamental knowledge and gain 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 are:

- ESP-IDF
- PlatformIO
- The first application and the basics of FreeRTOS
- Debugging
- Unit testing

Let's start with the development framework by Espressif Systems, ESP-IDF.

Technical requirements

Throughout the book, we're going to use **Visual Studio Code** (VS Code) as the integrated development environment (IDE). If you don't have it, you can find it here: https://code.visualstudio.com/.

As hardware, we need both devkits, **ESP32-C3-DevKitM-1** and **ESP32-S3 Box Lite**. I will also share what is required in each example.

The source code in the examples is located in the repository found at this link: [link]

Check out the following video to see the code in action: [link]

ESP-IDF

ESP-IDF is the official framework by Espressif Systems to develop applications on ESP32. It comes with the necessary tools and SDK, thus, it is basically enough to have only ESP-IDF on your machine for ESP32 development. Nonetheless, for the sake of productivity and easy management, it is preferable to use an IDE and install ESP-IDF as an extension in it.

The installation of ESP-IDF differs from platform to platform, so you need to follow the steps as described in the documentation for your target development machine.

> The ESP-IDF documentation provides the best information. You can find the installation steps at this URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/#installation

My personal choice of development environment is Visual Studio Code (version 1.70.2 as of writing this chapter) 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 Eclipse and use ESP-IDF through it.

> If you choose to use VSCode + ESP-IDF Extension, it is explained very well here on GitHub: https://github.com/espressif/vscode-esp-idf-extension

After having the ESP-IDF extension installed, we can start with 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. 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 tools from your development machine)

```
$ lsusb | grep -i espressif
Bus 001 Device 025: ID 303a:1001 Espressif USB JTAG/serial debug unit
```

1. Run VSCode and make sure the ESP-IDF extension is enabled.

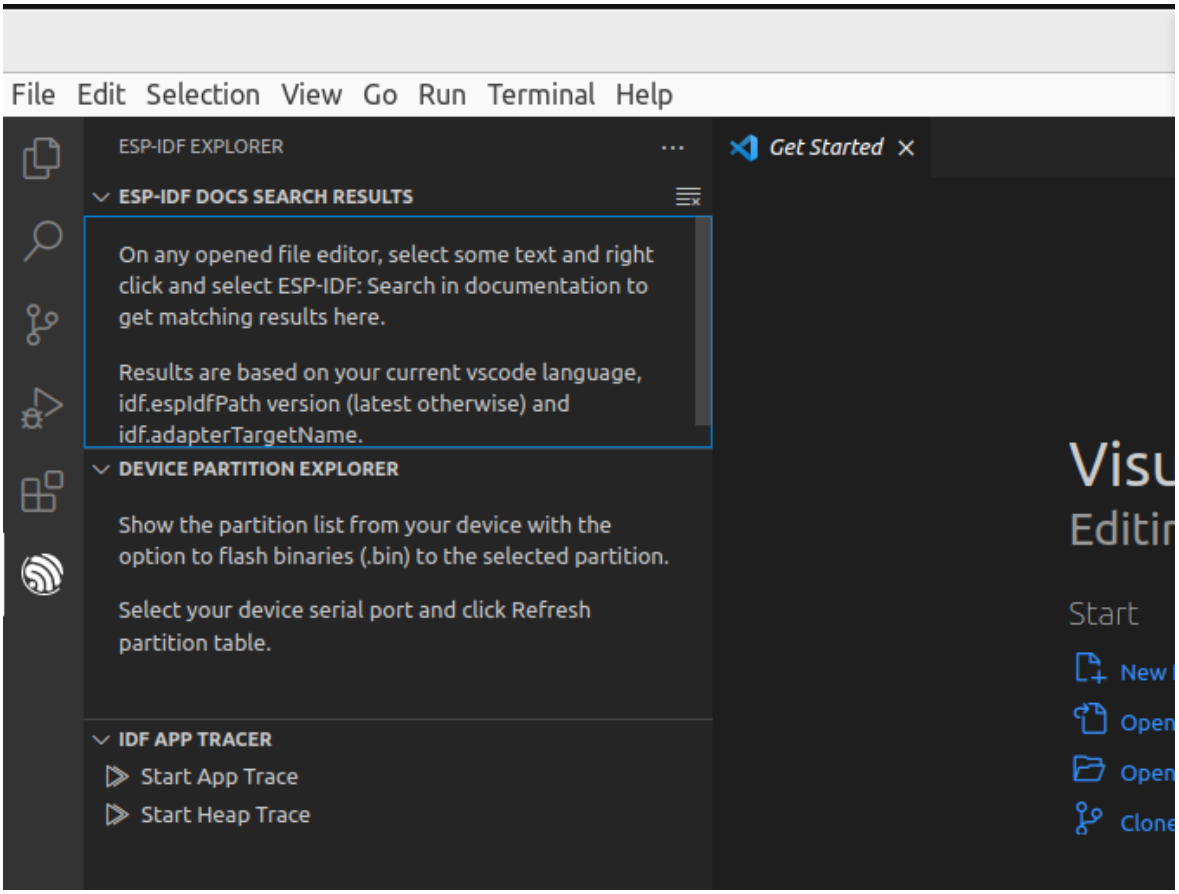

Figure: The ESP-IDF extension is active

2. Create an ESP32 project. There are two ways to do that. You can either select **View/Command Palette/ESP-IDF: New Project** or press the **(Ctrl E) + N** key combination to open the new project screen. Fill the input boxes as the following (Make sure you select ESP32-S3 and the serial port it has connected).

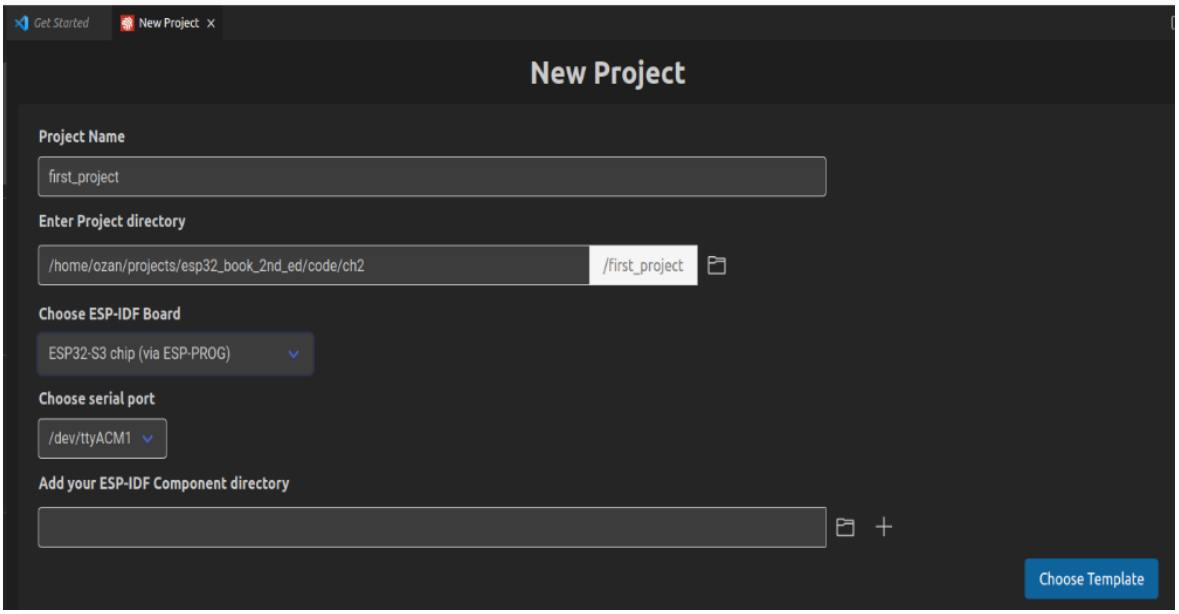

Figure: New project

3. Then click on the **Choose Template** button. In 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.

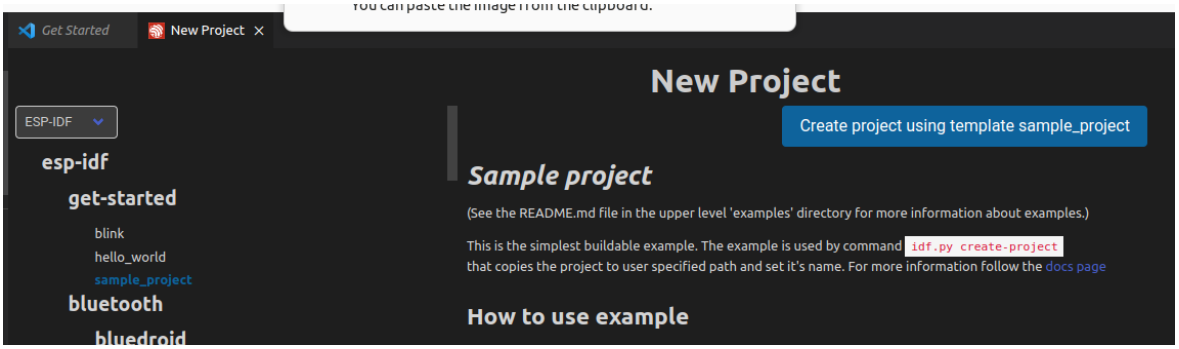

Figure: Select sample_project

4. 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 there will be a new VSCode window with the new project.

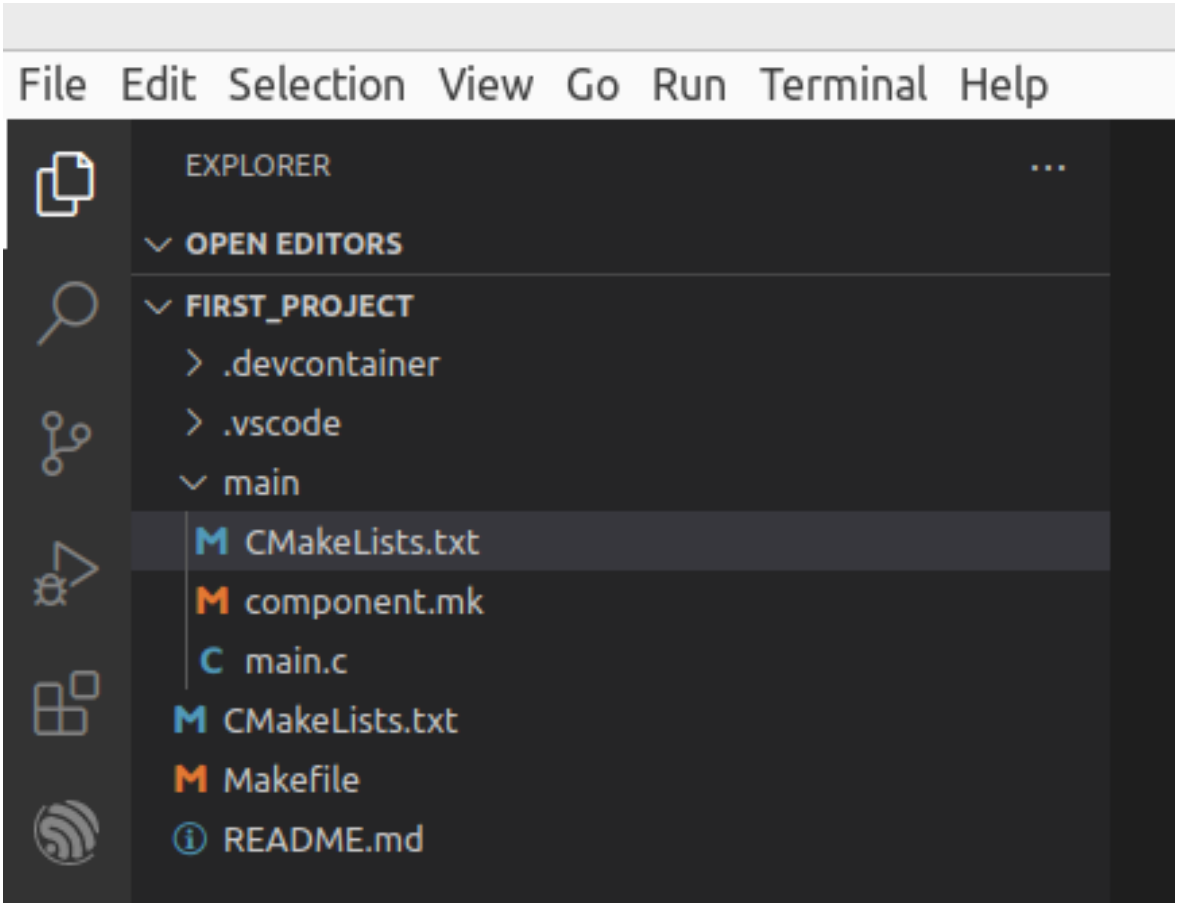

Figure: File explorer of VSCode

5. In the new VSCode window, we now have the environment for the ESP32 project. You can see the file names in the explorer on the left side of the window. 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";
}
```

1. To compile, flash and monitor the application, we can simply press the **(Ctrl E) + D** key combination. The ESP-IDF extension will ask the flash method. There are three options: **JTAG, UART, DFU**. We select **UART**. Then the next step is to select the project – we only have **first_example**. VSCode will open a terminal tab and you can see the compilation output there. If everything goes well, it will connect to the serial monitor and all the logs will be displayed as the following.

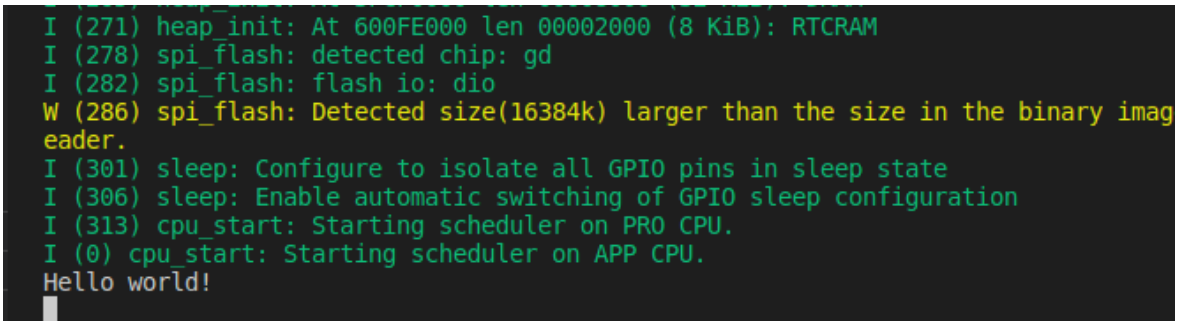

Figure: The serial monitor output

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

Let's go back to the code and discuss it shortly. 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"` says that we will next add some C code and the C++ compiler will not mangle the symbol name coming after. 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";
}
```

This is the entire ESP32 application and we can use it as a blueprint when starting a new 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** to look for any include 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/ The documentation about how to define a new ESP-IDF component is here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html#minimal-component-cmakelists

If you have noticed, when we compiled the application, new additions appeared in the project root.

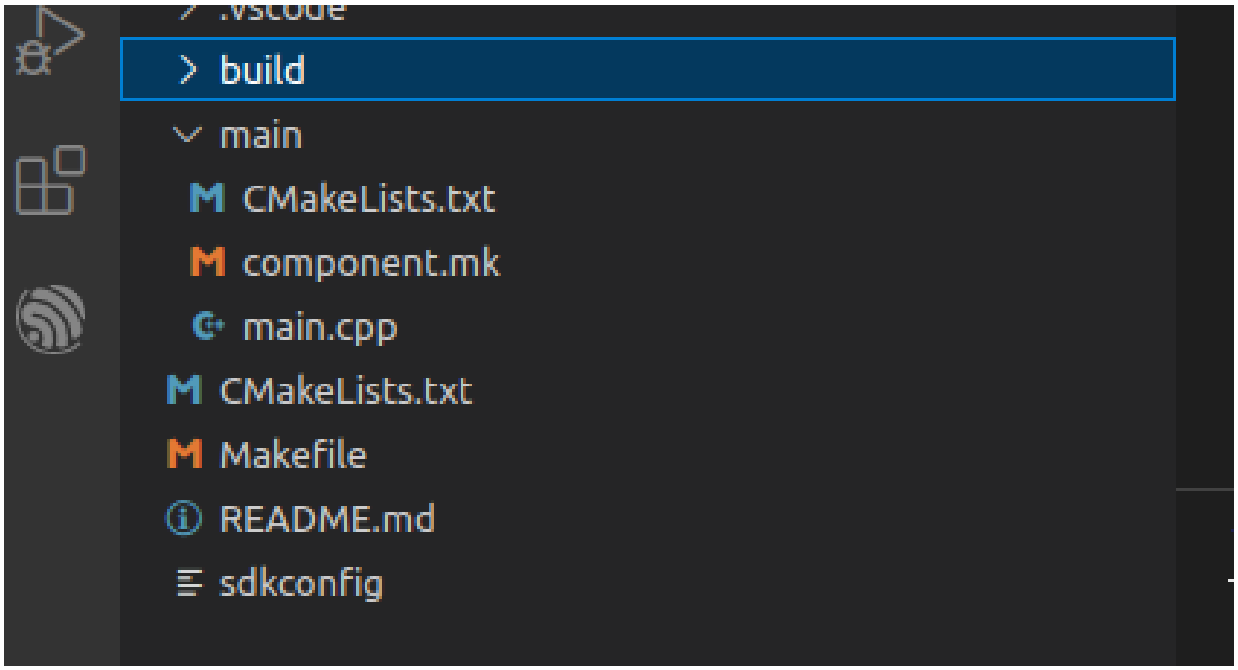

Figure: The files and directories after compilation.

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.

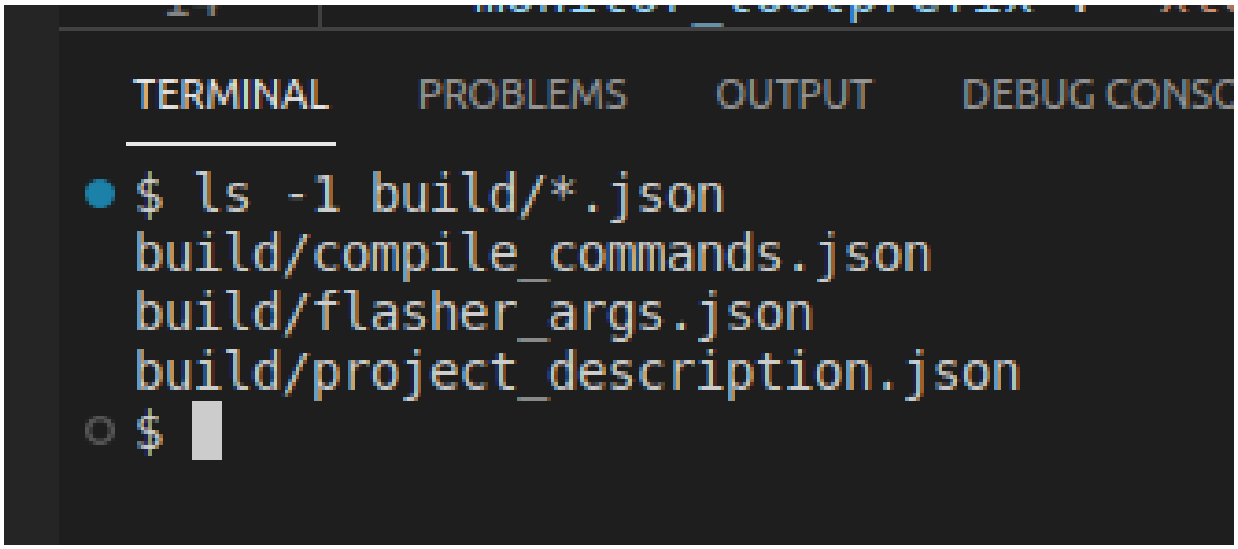

Figure: JSON files in build

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

sdkconfig in the project root is important. It is generated by ESP-IDF automatically during the build if there is none. This file contains the entire configuration of the application and defines the default behavior of the system.

The editor is **menuconfig**, which is accessible from **View/Command Palette/ESP-IDF: SDK Configuration editor (menuconfig)** or simply **(Ctrl+E)+G**.

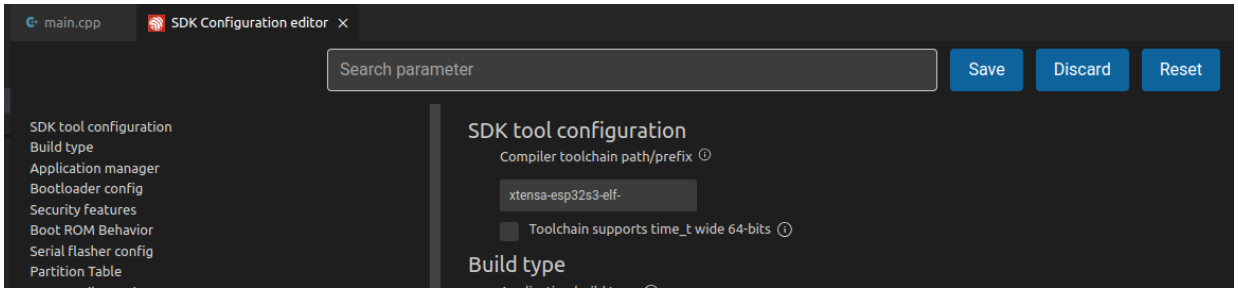

Figure: SDK configuration

We will use **menuconfig** often to configure the projects, and even provide our custom configuration items to be included in **sdkconfig**.

One last thing about the ESP-IDF extension is the **ESP-IDF Terminal**. It provides command-line access to the underlying Python scripts that comes with the ESP-IDF installation. To open a terminal, you can run **View/Command Palette/ESP-IDF: Open ESP-IDF Terminal** or press the key combination of **(Ctrl+E)+T**. It opens an integrated command-line terminal. The two powerful and popular tools are **idf.py** and **esptool.py**. You can manage the entire development environment and build process by only using **idf.py**. 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 /home/ozan/projects/esp32_book_2nd_ed/code/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 de
-- Building ESP-IDF components for target esp32s3
-- Project sdkconfig file /home/ozan/projects/esp32_book_2nd_ed/code/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 compilation files if any), compiles the application, flashes the generated firmware to the devkit and finally starts the serial monitor to see the application output.

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

```
$ esptool.py --chip esp32s3 write_flash -z 0x10000 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…
```

I believe this overview is enough to start using VSCode + ESP-IDF extension together to develop ESP32 projects. The next tool is PlatformIO.

PlatformIO

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

> We'll talk about PlatformIO a lot, but as the ultimate reference, you can find the documentation here: https://docs.platformio.org/en/latest/what-is-platformio.html

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 chip type (not limited to Espressif products), which framework and which version of the 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 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 the first time 8 years ago (roughly – I'm not good at remembering past events), the debug feature was not available in the free version. It was a big disappointment for me. PlatformIO 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.

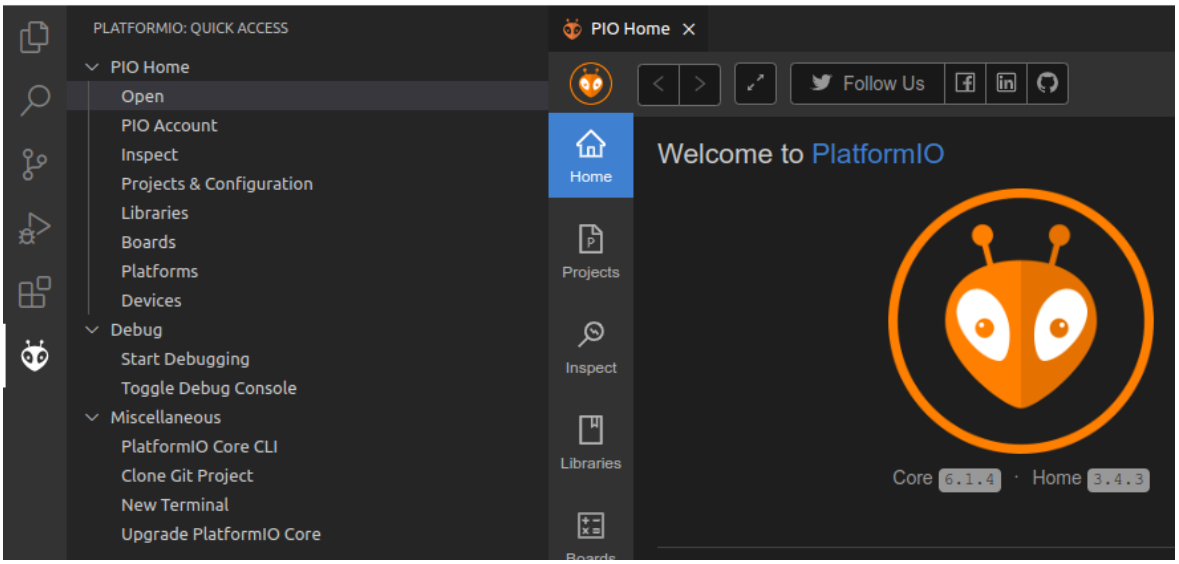

Figure: PlatformIO Home

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

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

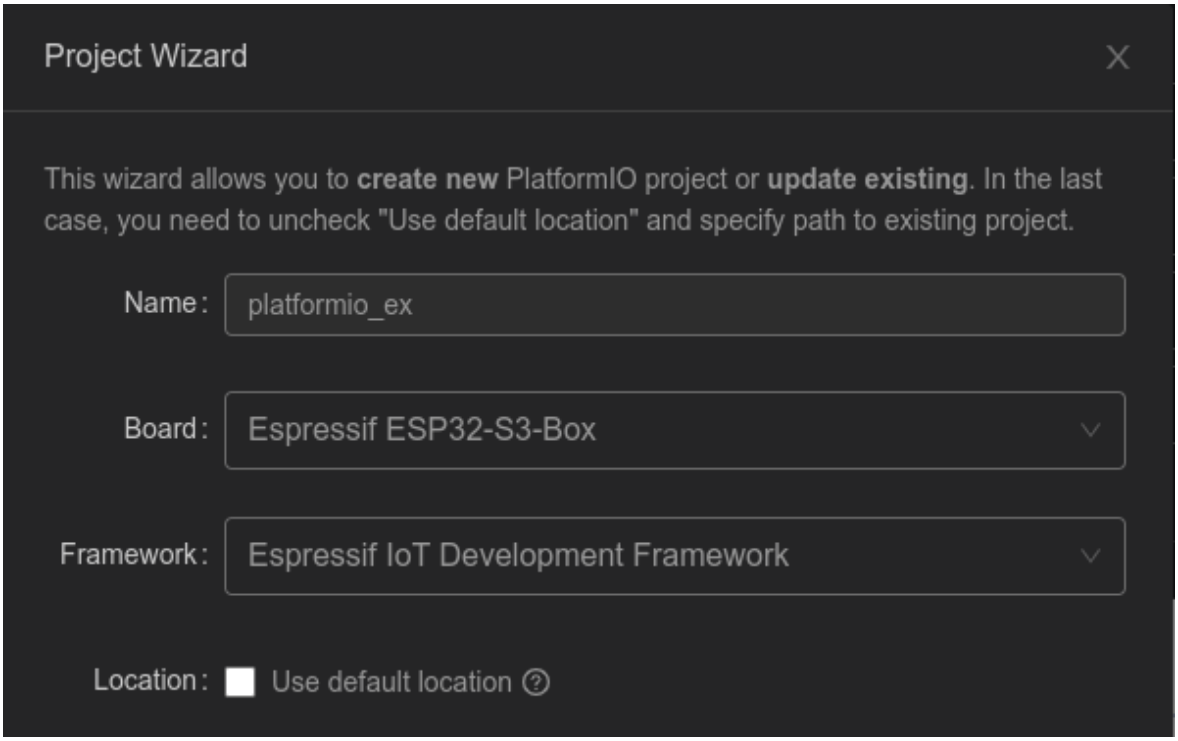

Figure: Project Wizard

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

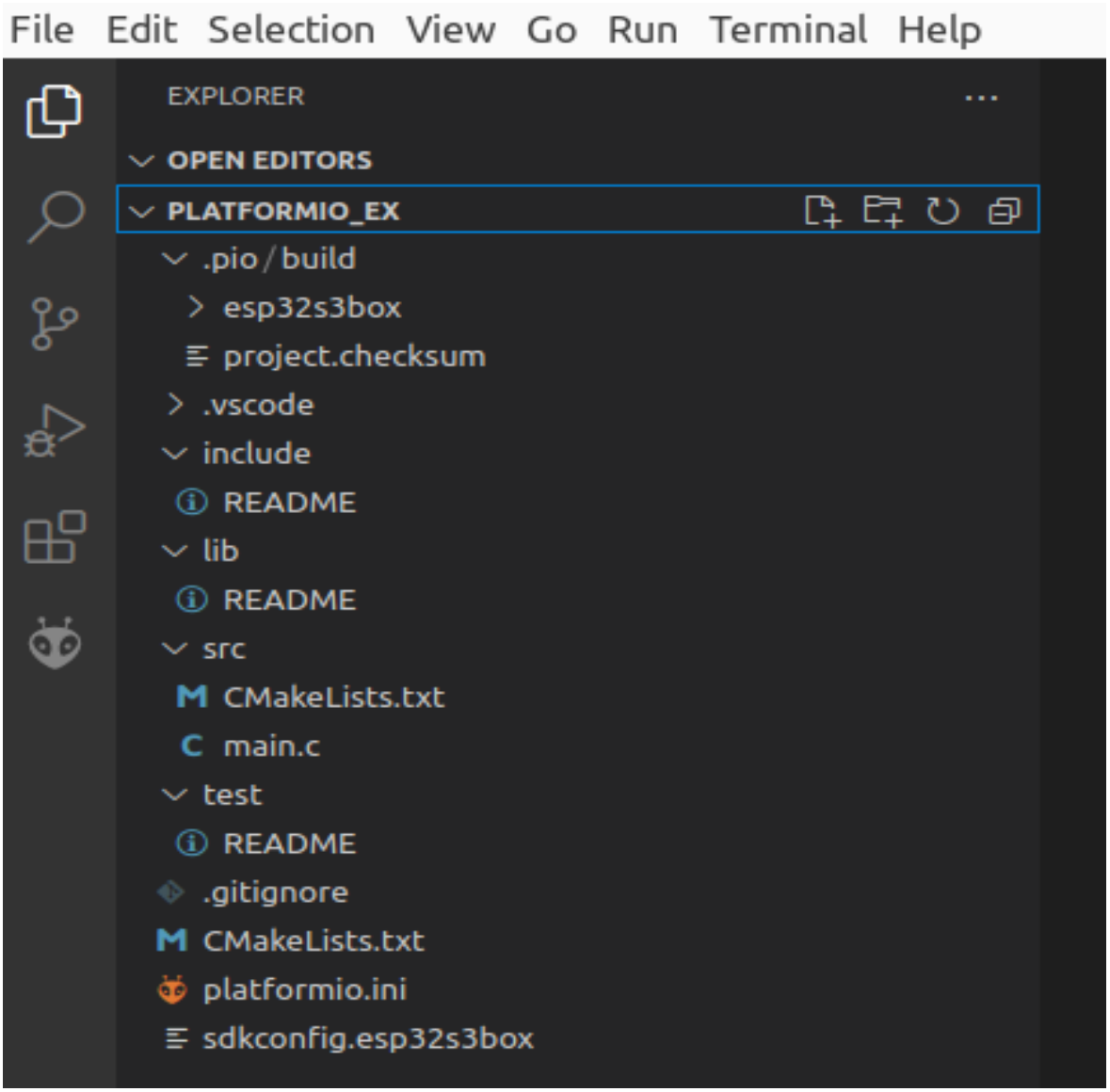

Figure: Project directory structure

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";
}
```

1. 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
```

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

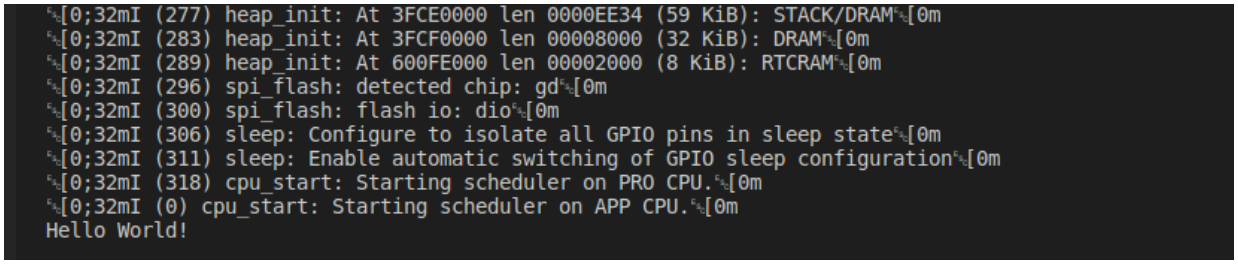

Figure: Application output in the Terminal.

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

```
[env:esp32s3box]
```

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

```
platform = espressif32
```

This sets the platform – `espressif32`. As of writing this chapter, PlatformIO supports 48 different platforms. 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.

```
board = esp32s3box
```

This sets the board – `esp32s3box`. There are 1420 different boards supported by PlatformIO, 162 of them is in the `espressif32` platform.

```
framework = espidf
```

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

These three settings come from the project definition stage. 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 as 115200bps, and RTS and DTR to 0 (zero) 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.

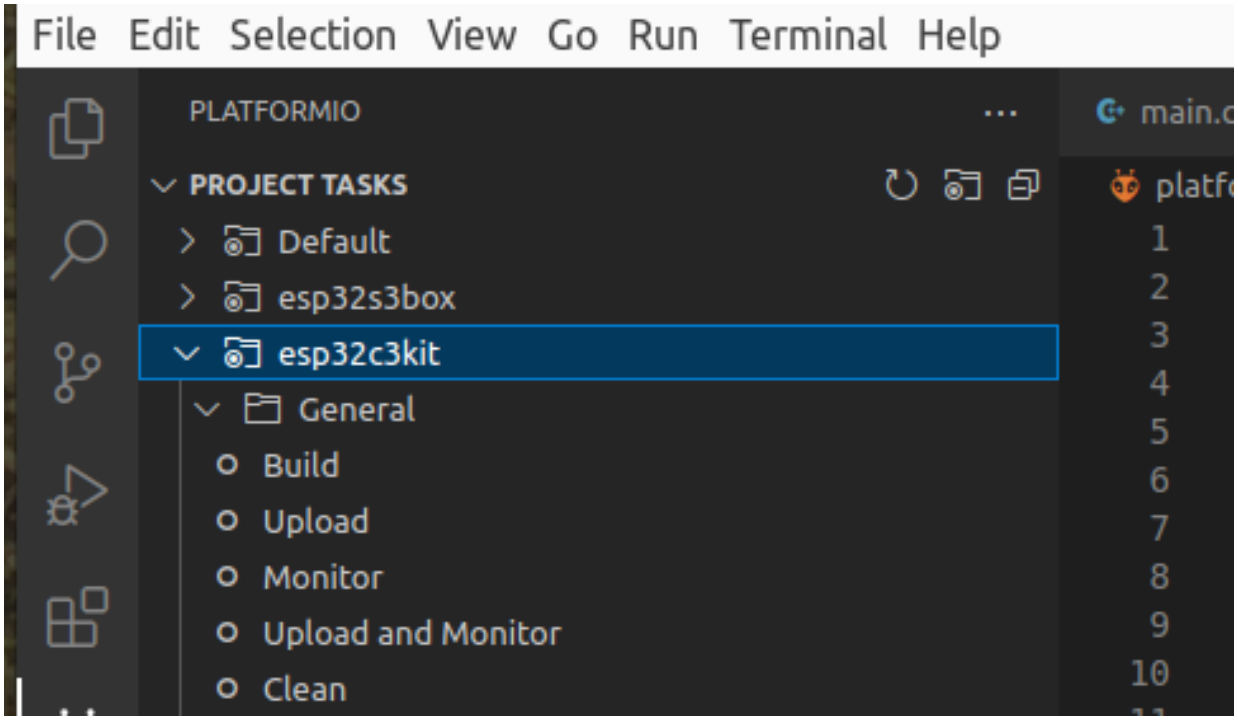

Figure: New environment under Project Tasks

After plugging the new devkit, you can upload and monitor the same application without any other modification in the project. Again, we didn't manually download or install anything for ESP32-C3-DevKitM-1, all handled by PlatformIO. If you wonder 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 a PlatformIO Terminal, you can click on the **PlatformIO: New Terminal** button of the bottom toolbar.

Figure: 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 (labels appear when you hover the mouse pointer over the buttons), it will redirect you to a command-line terminal where you can enter **pio** commands. Write **pio** and press the enter key to display the **pio** options.

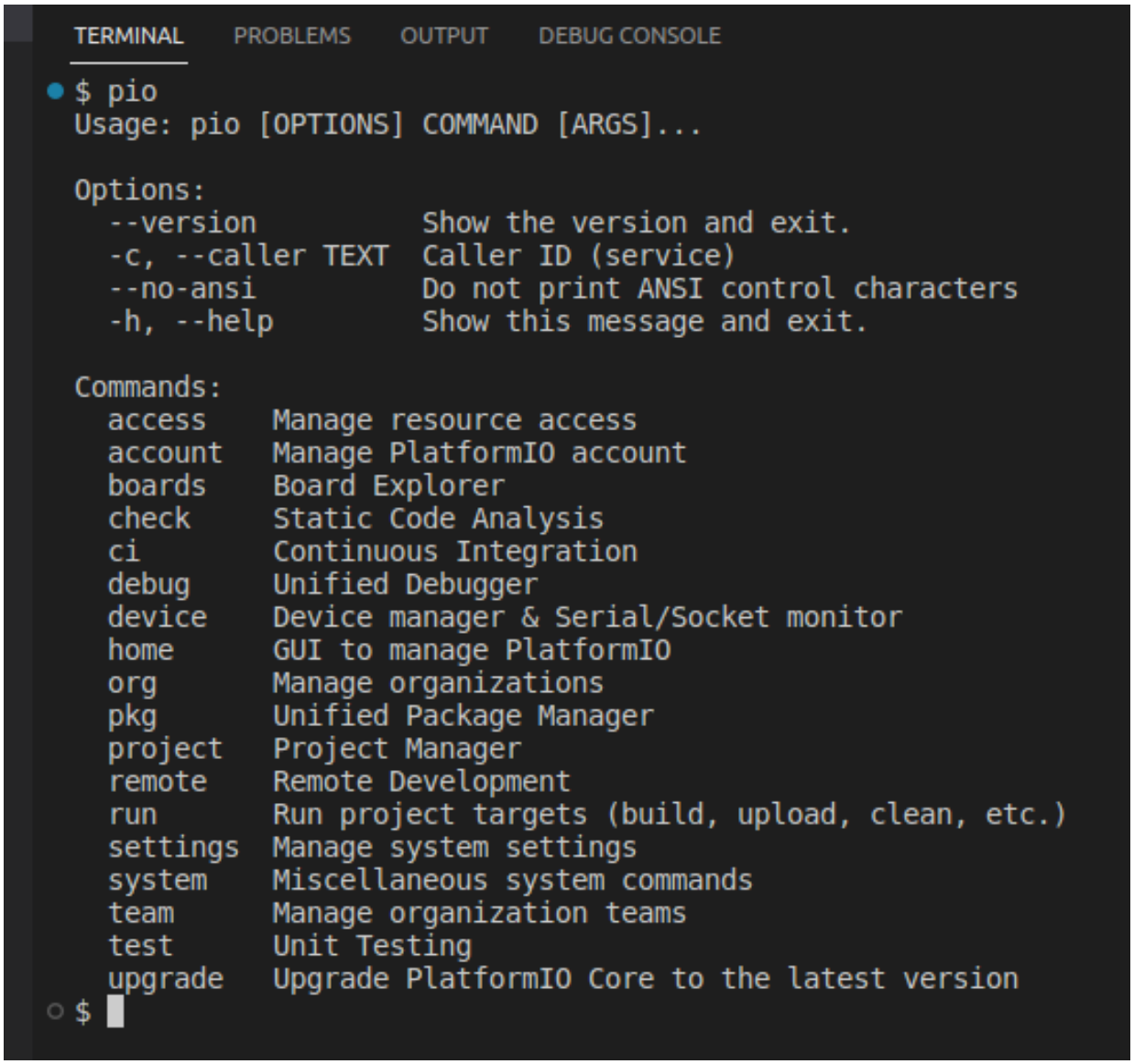

Figure: PlatformIO terminal and the pio command-line tool

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>
[0;32mI (324) cpu_start: Starting scheduler.[0m
Hello World!
```

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

> The online documentation is better. You can see it at this link: 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 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 2-core systems. Most of the differences between the vanilla FreeRTOS and ESP-IDF FreeRTOS stem from this reason. For those who have some experience with FreeRTOS, it would be enough to skim through these differences:

- Creating a new task: We have a new function 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 for protecting 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 `portENTER_CRITICAL_SAFE(mux)` and `portEXIT_CRITICAL_SAFE(mux)` macros for this purpose.

> Dual-core ESP32 names its cores as **PRO_CPU** (cpu0) and **APP_CPU** (cpu1). PRO_CPU starts when ESP32 is first powered and executes all the initialization, including APP_CPU activation. The `app_main` function is called from the main task running on PRO_CPU.

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

Producer – Consumer with ESP-IDF FreeRTOS

In this example, we simply implement the producer-consumer pattern to show some functionality of Espressif FreeRTOS. There will be a single producer and 2 consumer 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. 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 nice and colorful output on the serial monitor).

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

1. Rename **src/main.c** to **src/main.cpp** and edit it to have the following code inside.

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

1. Run **menuconfig** by selecting **PLATFORMIO/PROJECT TASKS/esp32s3box/Platform/Run Menuconfig**. 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 the **(Top) → Component config → FreeRTOS** and check the following options. (The next two is dependent on the one up level, and will be visible when the one before it is enabled.)
    - Enable FreeRTOS trace utility
    - Enable FreeRTOS stats formatting functions
    - Enable display of xCoreID in vTaskList
2. Build the project (**PLATFORMIO/PROJECT TASKS/esp32s3box/General/Build)**.
3. Flash and monitor the application to see the **hi** text on the serial monitor (**PLATFORMIO/PROJECT TASKS/esp32s3box/General/Upload and Monitor**).

So far, so good. Now, we can implement the producer-consumer pattern in the **src/main.cpp** file as the following:

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

All FreeRTOS header files reside in the `freertos` directory and `freertos/FreeRTOS.h` 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. Now 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`. It will be the medium 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 first. 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 prototype:

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

Looking at this prototype, `xTaskCreate` will create a FreeRTOS task which runs the `producer` function that we declared earlier. The task name will be `"producer"` and the stack size is `4096` in bytes. We don't pass any parameter 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 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, tas
```

In addition to the parameters that `xCreateTask` uses, `xTaskCreatePinnedToCore` needs task affinity – 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 all tasks created. 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. We can upload and monitor the application now. Let's discuss the output shortly.

```
<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 the task is created
- Core ID

We can see our tasks in the list in addition to other default tasks. They are (in the order they have been created):

- The **IPC (Inter-processor call)** 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 CoreID 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 comes to 10 as we coded. The rest of the job is only for the consumers to dequeue all numbers remaining in the queue.

You can find the ESP-IDF FreeRTOS API documentation here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html

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

Debugging

All families of ESP32 MCUs support **JTAG (Joint Test Action Group)** 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 application, we use a version of **ESP gdb** (GNU debugger), depending on the architecture of ESP32 that we use in a particular project. The next figure shows a general ESP32 debug setup:

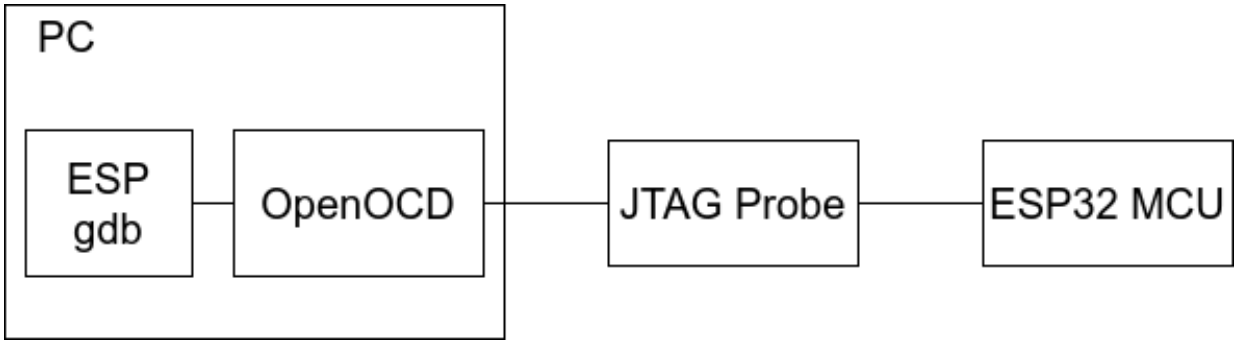

Figure: JTAG debugging

The issue with the JTAG debugging is that it requires at least 4 GPIO pins to carry the JTAG signals, which means 4 GPIO pins less in 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 figure above, the JTAG probe in the middle is not needed for debugging and OpenOCD talks directly to the MCU over USB. The built-in JTAG debugging requires only 2 pins on ESP32, which saves 2 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 have, 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

You can see my simple setup below:

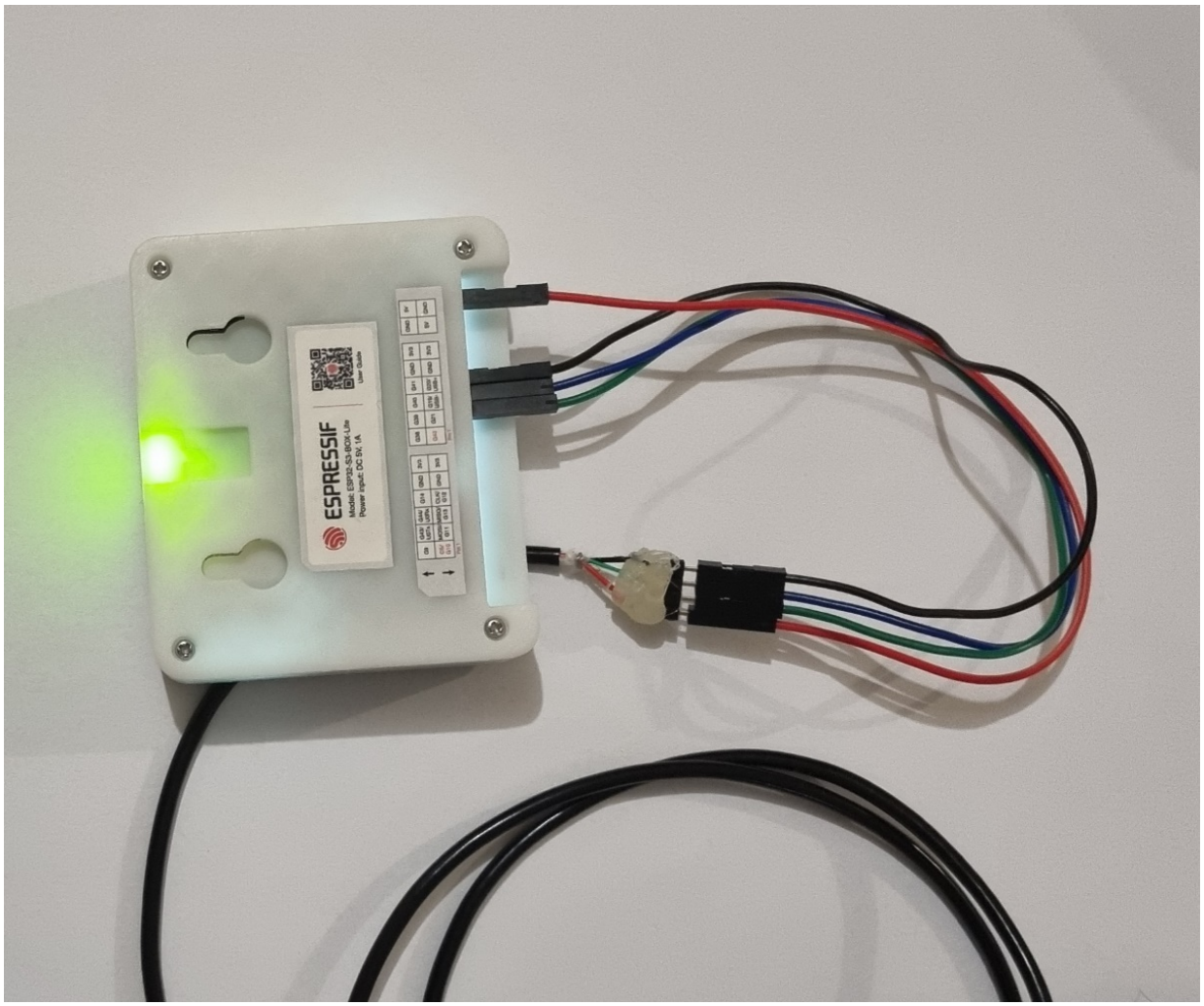

Figure: Built-in JTAG

To make a USB debugging cable, I cut an ordinary micro-USB cable from the middle, removed the micro connector, instead soldered a 4-pin header to the internal signal lines.

Now, it is time to create the project and upload the firmware to see whether our setup works. Let’s do this in steps:

1. Start a new PlatformIO project with the following parameters:
    - Name: **unit_testing_ex**
    - Board: **Espressif ESP32-S3-Box**
    - Framework: **Espressif IoT Development Framework**
2. Open the **platformio.ini** file and set the content as the following.

```
[env:esp32s3box]
platform = espressif32
board = esp32s3box
framework = espidf
upload_protocol = esp-builtin
```

1. Rename **src/main.c** to **src/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(1000 / portTICK_RATE_MS);
        ++i;
        my_func();
    }
}
```

1. Upload the firmware by selecting **PLATFORMIO / PROJECT TASKS / esp32s3box / General / Upload**. If the connection to the devkit over USB is correct, you should see the following log on the terminal.

```
<removed>
CURRENT: upload_protocol = esp-builtin
Uploading .pio/build/esp32s3box/firmware.bin
Open On-Chip Debugger  v0.11.0-esp32-20220706 (2022-07-06-15:48)
<removed>
** Programming Started **
<removed>
** Verify OK **
shutdown command invoked
===== [SUCCESS] Took 114.93 seconds ====
```

Here, we have configured the PlatformIO project with the built-in JTAG USB port to upload the firmware by adding the following line in **platformio.ini**.

```
upload_protocol = esp-builtin
```

The terminal output also shows this configuration. Then, OpenOCD takes control and flashes the devkit with the firmware.

If flashing fails with a port error, just reverse the D+/D- connections. This simple change will probably solve the problem.

The hardware setup is ready. Let’s try debugging the application. To do that, we need to use the **idf.py** tool that comes with ESP-IDF. I wish I could say we can use the integrated PlatformIO debugging here, however, PlatformIO doesn’t fully support ESP32-S3 built-in JTAG debugging as of writing this book, unfortunately. Anyways, we have **idf.py** for this.

The steps to debug the application are:

1. Edit **CMakeLists.txt** in the root of the project with the following content.

```
cmake_minimum_required(VERSION 3.16.0)
set(EXTRA_COMPONENT_DIRS src)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(debugging_ex)
idf_build_set_property(COMPILE_OPTIONS "-O0" "-ggdb3" "-g3" APPEND)
```

1. Open two integrated command-line terminals (you can use the PlatformIO terminal button at the bottom toolbar for this). On one terminal, we will run the OpenOCD server, and the other one is to start the debugger GUI, but first we need to enable **idf.py** by running the export script from the ESP-IDF installation directory.

```
$ ls -1 ~/esp/esp-idf/export.*
~/esp/esp-idf/export.bat
~/esp/esp-idf/export.fish
~/esp/esp-idf/export.ps1
~/esp/esp-idf/export.sh
```

1. Select the correct script for your machine and run it on **both** command-line terminals. My development machine is an Ubuntu Desktop, so I will run **export.sh**.

```
$ source ~/esp/esp-idf/export.sh
<removed>
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:
  idf.py build
```

1. See whether **idf.py** works properly.

```
$ idf.py --version
ESP-IDF v4.4.1
```

1. Set the target ESP32 architecture with the following command on one of the terminals.

```
$ idf.py set-target esp32s3
<removed>
-- Build files have been written to: debugging_ex/build
```

1. Flash the devkit with the application by running the following command.

```
$ idf.py clean flash
<removed>
Leaving...
Hard resetting via RTS pin...
Done
```

1. Then, we can start the OpenOCD server next.

```
$ idf.py openocd --openocd-commands "-f board/esp32s3-builtin.cfg"
Executing action: openocd
OpenOCD started as a background task 106097
Executing action: post_debug
Open On-Chip Debugger  v0.11.0-esp32-20211220 (2021-12-20-15:42)
<removed>
Info : esp32s3.cpu0: Debug controller was reset.
Info : esp32s3.cpu0: Core was reset.
Info : esp32s3.cpu1: Debug controller was reset.
Info : esp32s3.cpu1: Core was reset.
Info : starting gdb server for esp32s3.cpu0 on 3333
Info : Listening on port 3333 for gdb connections
```

1. On the other terminal, start the debugger. The following command will open a browser window and connect to the OpenOCD server.

```
$ idf.py gdbgui
Executing action: gdbgui
<removed>
```

It took many steps but finally we have a web-based GUI to debug the application. Before talking about the GUI, let's review what we have in **CMakeList.txt** of the project root.

The first line specifies the minimum **cmake** version. Each version of **cmake** provides slightly different functionality, so any root **CMakeList.txt** starts with the minimum version.

```
cmake_minimum_required(VERSION 3.16.0)
```

The next line commands ESP-IDF to look for more components in the specified directories. The `app_main` function, the entry point of the application, resides in **src/main.cpp**. Therefore, we add **src** in the `EXTRA_COMPONENT_DIRS` list so that ESP-IDF can find the application source code.

```
set(EXTRA_COMPONENT_DIRS src)
```

Next, the default **cmake** script that comes with ESP-IDF is included and the project is configured with the `project` macro, which is defined in that default script.

```
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(debugging_ex)
```

In the last line, we provide the debugging options to the compiler.

```
idf_build_set_property(COMPILE_OPTIONS "-O0" "-ggdb3" "-g3" APPEND)
```

idf.py makes use of this **cmake** script to configure and build our application.

> I strongly suggest to read the ESP-IDF build system documentation: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html

After this quick explanation about the build process basics, let's debug our application with the web-based GUI. The following screenshot shows this GUI.

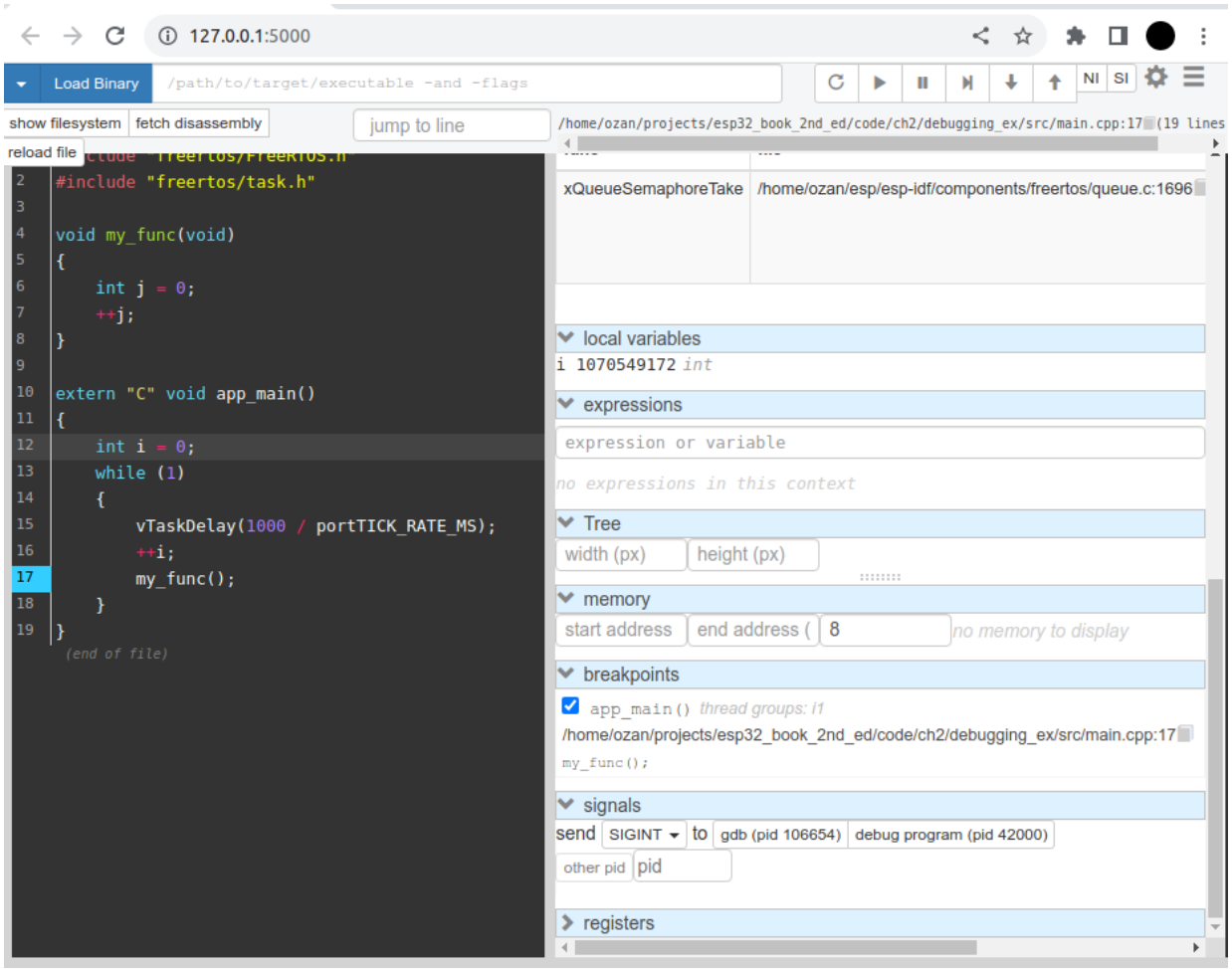

Figure: Web-based debugger

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, 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 the line number 17 to set a breakpoint.
- Press '**c**' (continue) on the keyboard and observe the local variable `i` increases every time the execution hits the breakpoint.
- Try '**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

Either 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 projects. 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. 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 options as unit-test framework. 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 in the **platformio.ini** file of the project.

> The unit testing documentation by PlatformIO is provided here:
> https://docs.platformio.org/en/latest/advanced/unit-testing/index.html

In this example, we will develop a simple class for light control by setting a GPIO pin of ESP32-C3-DevKitM-1 to high/low and test it by using the GoogleTest framework. Here are the steps to achieve this:

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 the following.

```
[env:esp32-c3-devkitm-1]
platform = espressif32@5.1.1
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
monitor_raw = yes
lib_deps = google/googletest@1.12.1
test_framework = googletest
```

1. Build the project.

The project is now configured and ready for the application development. However, before moving on, I want to discuss the library management mechanism of PlatformIO briefly. We have several options to add external libraries in our projects. The easiest one is probably just 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.

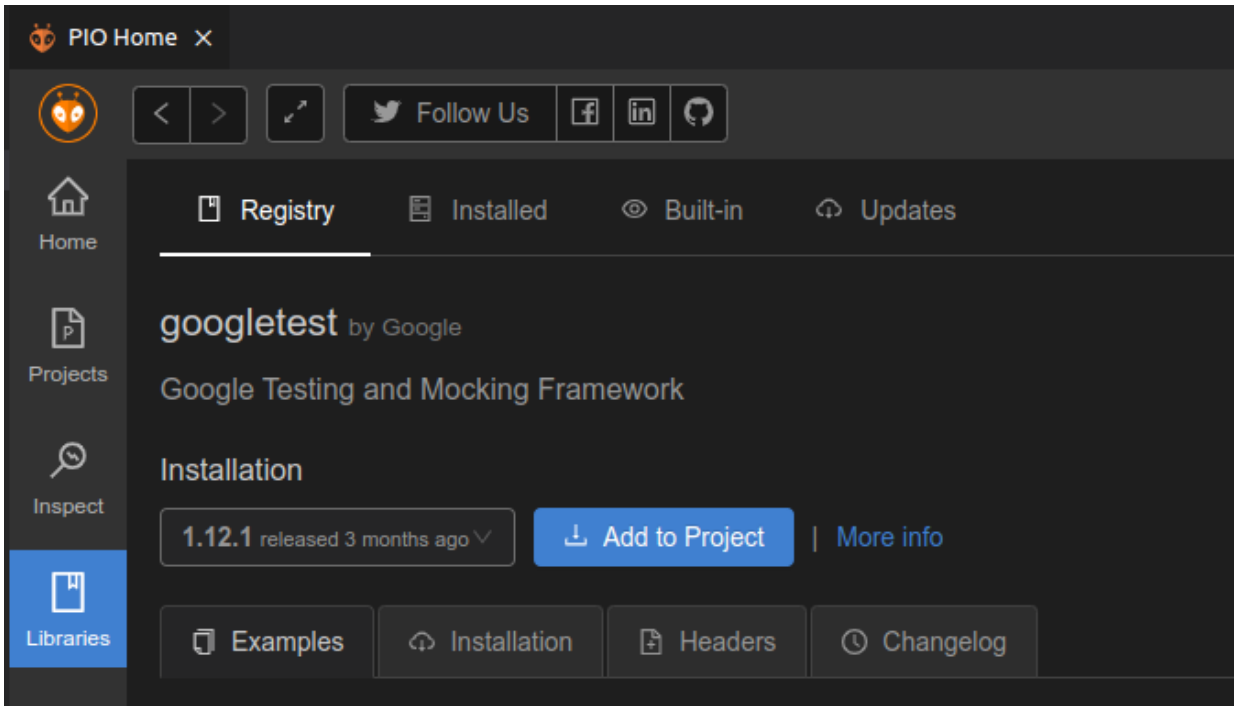

Figure: PlatformIO Registry

In our example, I have just preferred to specify **googletest** directly as in the following configuration line without using the graphical interface.

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

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

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

If you have noticed, we can also set the platform version.

```
platform = espressif32@5.1.1
```

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 information about the library management, we can continue with the application. Let's create 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"
```

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_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 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 as 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` (zero) 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. 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 base class `::testing::Test`. In its protected area, we declare a static `AppLight` object and initialize it in the constructor of the fixture. Having the fixture ready, we can now write a test as the following:

```
 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 if it is really turned on. The `ASSERT_GT` macro checks whether the first parameter is greater than the second one.

Another test would be 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 if it is 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 the entire test cases. This finalizes the test coding. Let's run it on the devkit and see the test results.

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

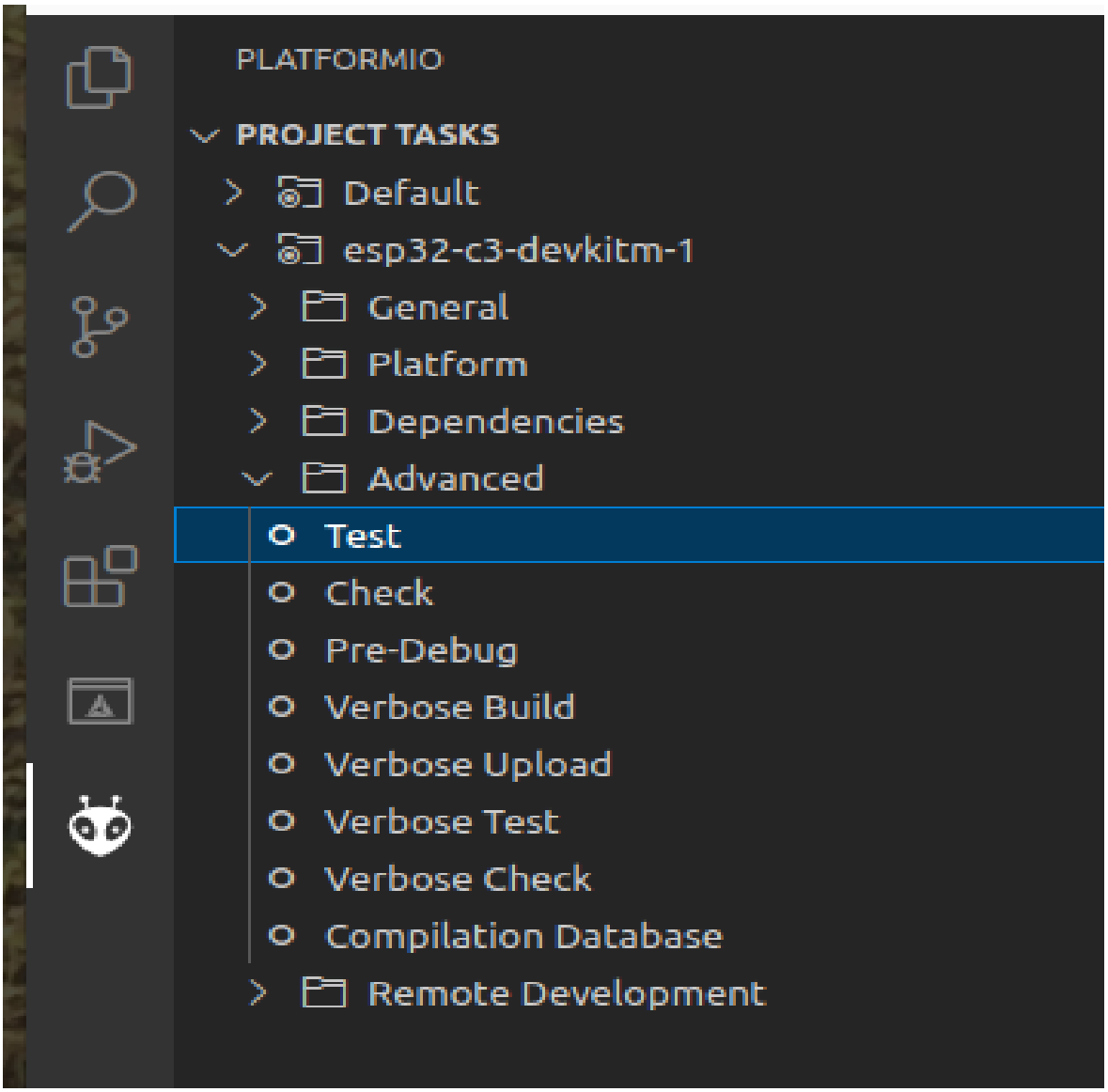

Figure: PlatformIO unit testing

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

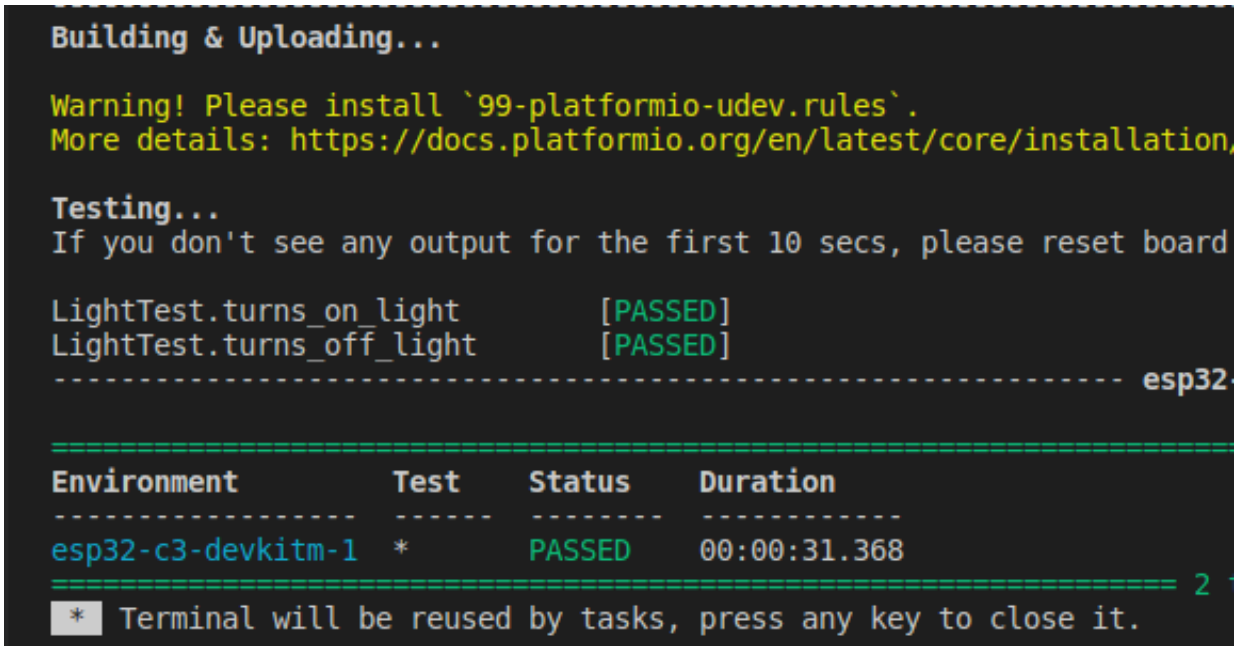

Figure: Terminal output

The terminal output 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 not to limit yourself with 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 to get familiar with their features and tools as much as possible.

Summary

In this chapter, we have learned 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 real IDE features on top of that. With its strong integration with the Visual Studio Code 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 cover all of them in a single chapter, we will learn the common peripherals by 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 as an overview of this chapter:

1. What is the name of the script file that the ESP-IDF build system uses to configure an ESP32 project?
    - CMakeLists.txt
    - platformio.ini
    - main.cpp
    - Makefile
2. What is the name of the project specific configuration file which is usually edited by running **menuconfig**?
    - CMakeLists.txt

    - sdkconfig
    - pio
    - platformio.ini
3. Which one is the most fundamental tool that comes with ESP-IDF to manage an ESP32 project?
    - pio
    - openocd
    - gdb
    - idf.py
4. Which one of the following methods is the easiest to debug an ESP32-S3?
    - JTAG
    - SWD
    - Built-in JTAG/USB
    - UART
5. Which file defines a PlatformIO project?
    - CMakeLists.txt
    - sdkconfig
    - pio
    - platformio.ini

3 Using ESP32 peripherals

In the previous chapter, we discussed the available tools that we can use to develop applications on ESP32. We learned about ESP-IDF, the PlatformIO IDE, debugging, and testing applications. Now, it is time to use the ESP32 peripherals in applications. Peripherals are how MCUs connect to the outer world. The ESP32 series chips provide a wide range of peripherals. Depending on the product vision, each family may have different types of peripherals and different number of channels of peripherals. For example, ESP32-S2/S3 series have more GPIO pins compared to other families and features **USB OTG (On-the-Go)** to act as a USB host, which allows us to connect other USB devices, such as a keyboard or mouse. On the other hand, ESP32-C2/C3 have a reduced set of peripherals to achieve much lower costs.

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

- Driving General-Purpose Input/Output (GPIO)
- Interfacing with sensors over Inter-Integrated Circuit (I^2C)
- Integrating with SD-card over Serial Peripheral Interface (SPI)
- Audio output over Inter-IC Sound (I^2S)
- Developing graphical user interfaces on Liquid-Crystal Display (LCD)

Technical requirements

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

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

- A 5mm LED
- A 220Ω resistor
- A tactile switch
- BME280 – temperature, humidity, pressure sensor
- TSL2561 – ambient light sensor
- An SD-card breakout board
- A micro SD-card
- 4x 10KΩ resistors

The source code in the examples is located in the repository found at this link: [link]

Check out the following video to see the code in action: [link]

Driving General-Purpose Input/Output (GPIO)

Fundamentally, a sensor is any device that generates some sort of output when exposed to a phenomenon—say, temperature, humidity, light, vibration, and so on. In an IoT application, we use sensors as data sources by connecting their output via an interface. **General-Purpose Input/Output (GPIO)** is the simplest form of a communication interface in the list of peripherals. It provides high or low values. For instance, it is quite possible to read the status of a contact sensor by interfacing it with GPIO, since a contact sensor can only be either open or closed. Other more complex sensors may require different types of interfaces, such as, **Inter-Integrated Circuit (I2C)** or **Serial Peripheral Interface (SPI)**.

Actuators are on the output side of IoT solutions. They change their state according to an analog or digital signal coming from the microcontroller and generate output to the environment. Some examples are a buzzer to make

sound, an LED to emit light, a relay to switch on/off, or a motor to create motion. We can use GPIO to drive on/off devices if they implement a GPIO interface for it.

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

Tip

A **schematic** shows the components, interfaces, and internal connections of a circuit. While working on an IoT device, keep the schematic at hand all the time so that you can easily refer to it when you need to find pin connections. You can download the schematic for ESP32-S3 Box Lite here: https://github.com/espressif/esp-box/blob/master/hardware/esp32_s3_box_lite_MB_V1.1/schematic/SCH_ESP32-S3-BOX-Lite_MB_V1.1_20211221.pdf

Turning an LED on/off by using a button

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

The hardware components of this example are:

- ESP32-S3 Box Lite
- A 5mm LED
- A 220Ω resistor
- A tactile switch

And the Fritzing sketch of the circuitry is:

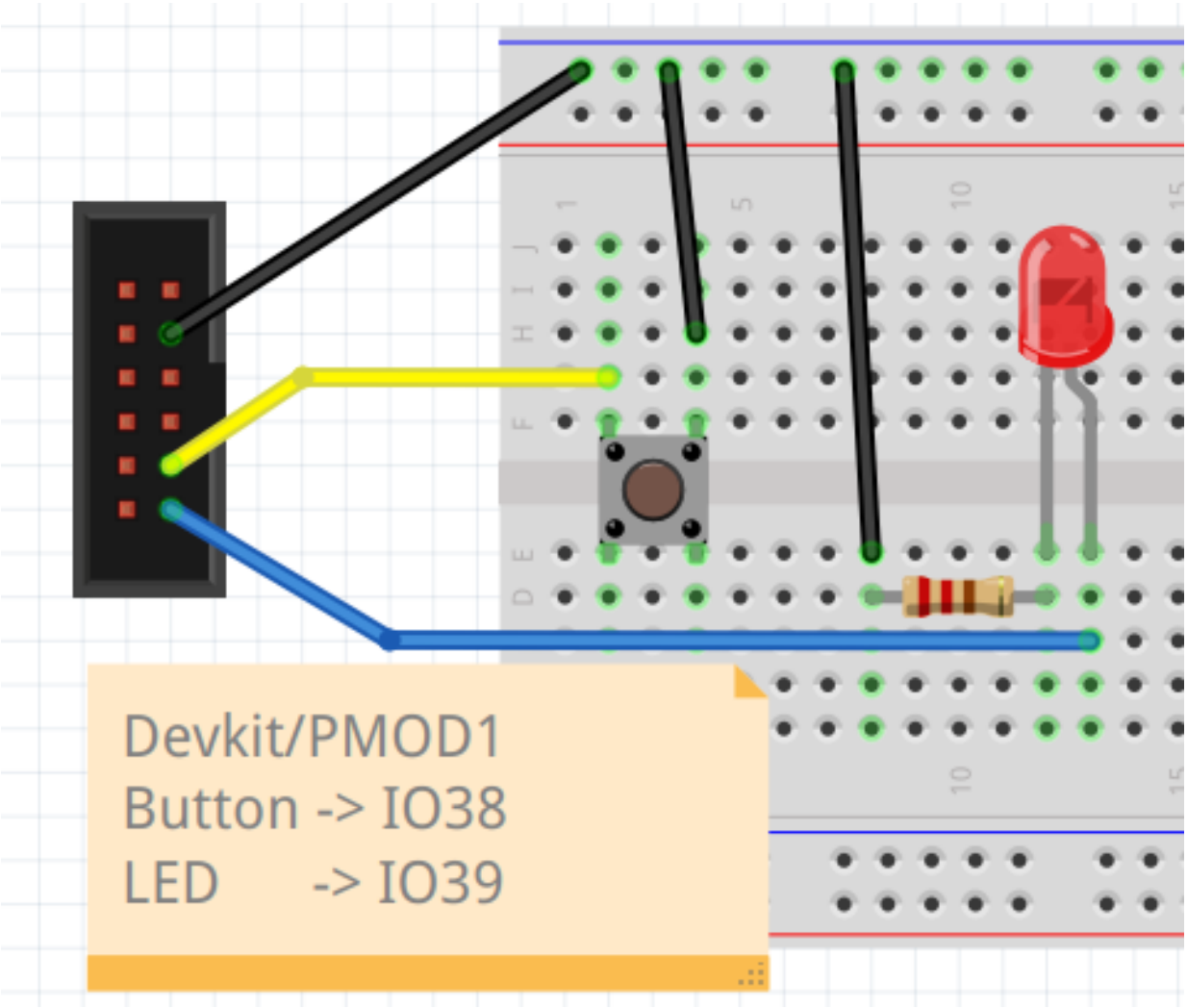

Figure: Fritzing sketch of the project

The terminals of the button are connected to PMOD1/GPIO38 and GND. We will set the internal pull-up resistor of the GPIO38 pin while configuring it so that when the button is pressed, it will read a **LOW** value.

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

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

After having this setup ready, you can create a ESP-IDF project for ESP32-S3 Box Lite as follows:

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

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

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

```
$ idf.py set-target esp32s3
Adding "set-target"'s dependency "fullclean" to list of commands with default set of options.
Executing action: fullclean
<logs removed>
```

1. See that you have the following directory structure

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

1. Rename the C source file to a **cpp** file.

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

1. Update the content of the main/CMakeLists.txt file as the following:

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

1. Start the VS Code IDE.

```
$ code .
```

With these steps, our project is ready to develop and we can continue in the VS Code development environment.

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

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

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

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

```
        void set(bool val)
        {
            gpio_set_level(GPIO_NUM_39, static_cast<uint32_t>(val));
        }
    }; // end of class
} // end of namespace
```

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

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

```
#pragma once
#include <functional>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
namespace
{
    void button_handler(void *arg);
}
```

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

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

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

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

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

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

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

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

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

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

```
namespace
{
    IRAM_ATTR void button_handler(void *arg)
    {
        static volatile TickType_t next = 0;
        TickType_t now = xTaskGetTickCountFromISR();
        if (now > next)
        {
            auto btn = reinterpret_cast<app::AppButton *>(arg);
            btn->toggle();
            next = now + 500 / portTICK_PERIOD_MS;
        }
    }
}
```

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

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

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

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

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

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

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

```
led.init();
button.init();
vTaskSuspend(nullptr);}
```

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

The development is finished and it is time to test. We can compile and upload the firmware by using PlatformIO. When the firmware is uploaded successfully, we can try pressing the button and see how the LED toggles (fingers crossed).

Troubleshooting

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

- Make sure the hardware setup is fine. Double check the button is connected to GPIO38 and the LED is on GPIO39. Use a multimeter if needed.
- Check the GND pin is connected to the correct legs of the LED and button (for the LED, it is the short leg). Make sure the LED is functional. Use a multimeter to test the LED if needed.
- It is good to have a multimeter and use it to ensure the components function properly.
- You can use the `ESP_LOGI` macro in various parts of the code, for example, in the button interrupt handler. If it prints a log when you press the button, then the GPIO configuration is correct.
- To test the `AppLed` class, you can run a simple loop in the `app_main` function, which toggles the LED every second.

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

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

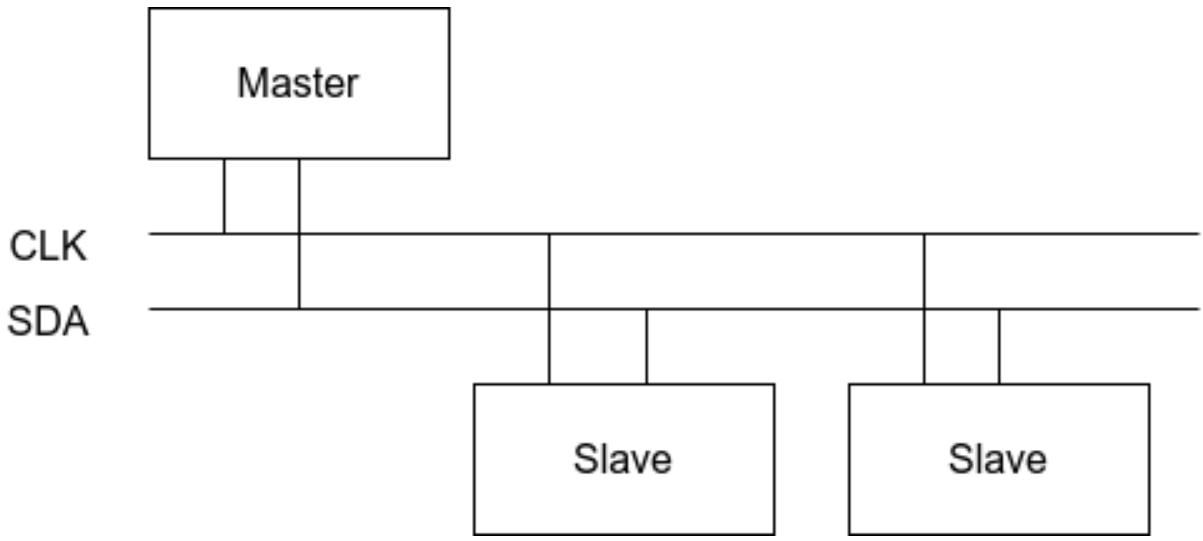

Figure: I 2 C architecture

The only rule here is that each slave has to have its own unique address on the bus. In other words, two sensors with the same I2C address cannot share a common bus. As defined in the protocol, the master and slaves exchange data over the SDA line. The master sends a command and the addressed slave replies to that command. Now, let's see an example of I2C communication on ESP32 by implementing a multisensor application.

Developing a multisensor

The purpose of this example is to develop an application in which we read values from different I2C sensors and display the readings on the serial console. We will use two different sensors for this:

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

BME280 is a Bosch product and you can find many breakout boards on the market with this sensor chip on it. BME280 can have two different addresses by configuring its SDO pin, 0x76 if connected to GND and 0x77 if left floating.

TSL2561 is a luminosity sensor from AMS. It supports three different addresses by connecting its address pin to GND (0x29) or VDD (0x49), or leaving it floating (0x39). As you may have already noticed, the addresses are all less than 0x80 since the I2C addressing scheme uses only 7 bits. The following figure shows a BME280 and TSL2561 breakout board that you can find from any electronics distributor.

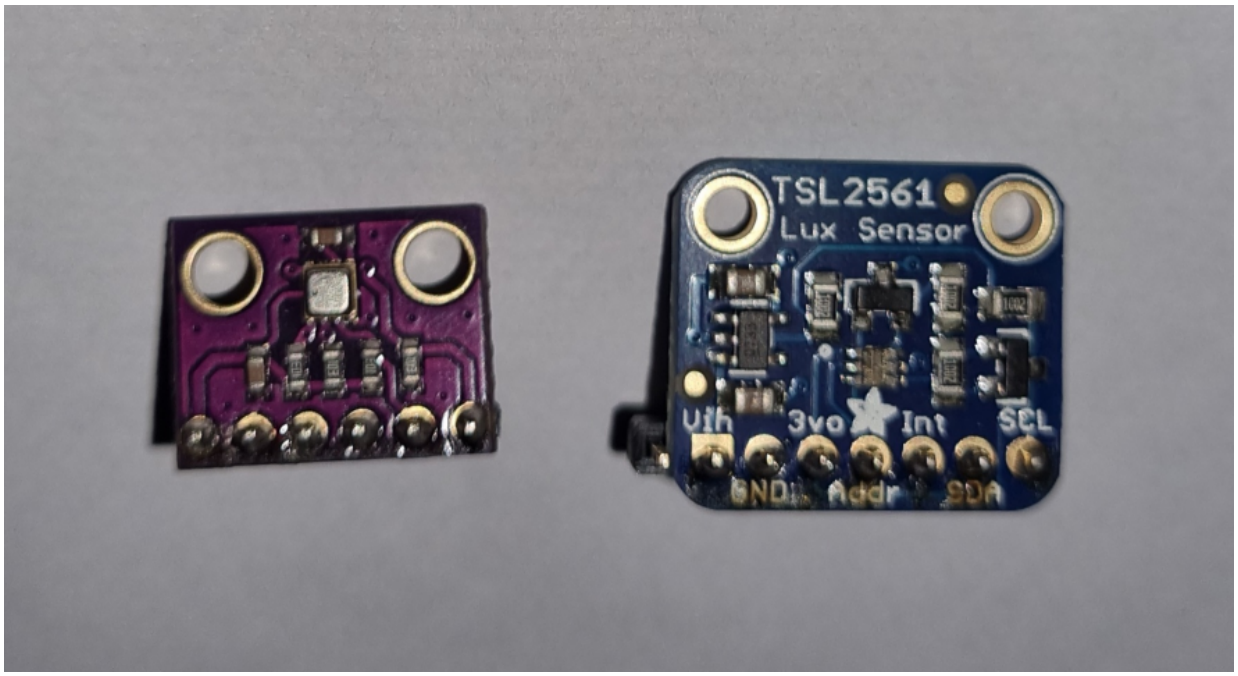

Figure: BME280 (left) and TSL2561 (right)

The hardware components of this example are:

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

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

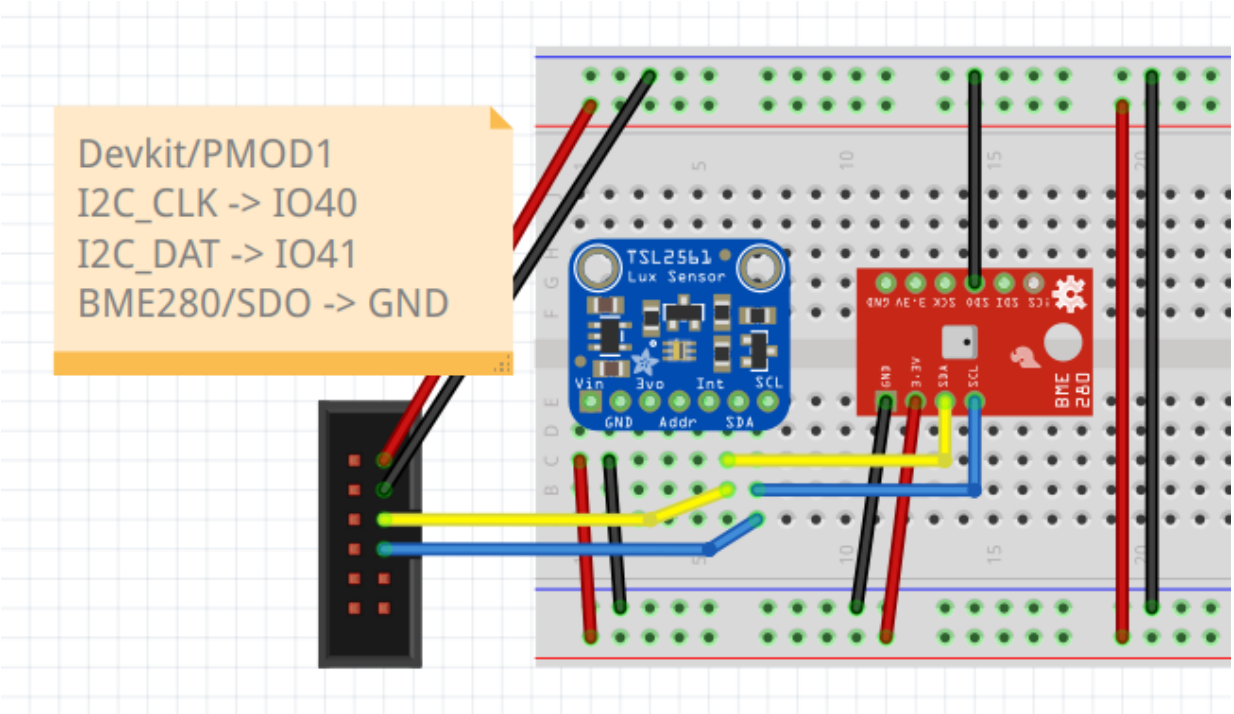

Figure: Fritzing sketch of the project

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

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

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

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

1. We need to use external components in this project and specify the directories that include them in the **CMakeLists.txt** file of the project root. Set its content as the following or just copy-paste the file from the Github repository.

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

1. Please note that the component directories in the previous step are relative to the project directory. This configuration will work without any issue if you have cloned the repository, but since we are configuring a new project together now, you need to copy those directories manually and set the correct paths.
2. Rename the source code file to **main/i2c_ex.cpp** and set the content of the **main/CMakeLists.txt** as the following:

```
idf_component_register(SRCS "i2c_ex.cpp" INCLUDE_DIRS ".")
```

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

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

Before moving on to the application development, I want to talk about the external components of the project shortly. We will discuss some useful 3rd-party libraries in detail in the next chapter, but there is one specific library that we need to include in this project, which is the **ESP-IDF Components library** at this link: https://github.com/UncleRus/esp-idf-lib. It is quite popular among ESP32 developers because of the device drivers it provides. We will communicate with the project sensors, BME280 and TSL2561, with the help of this library. The library currently doesn't support ESP32-S3 fully, therefore I had to modify it a bit and included in the book repository for ease of use.

Now, it is time to develop the application. Let's start with the multisensor class implementation. For this, we create a source code file, **main/AppMultisensor.hpp**.

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

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

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

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

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

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

```
    public:
        void init(void)
        {
            ESP_ERROR_CHECK(i2cdev_init());
            memset(&m_light_sensor, 0, sizeof(tsl2561_t));
            ESP_ERROR_CHECK(tsl2561_init_desc(&m_light_sensor, TSL2561_I2C_ADDR_FLOAT, I2C_NUM_1,
```

```
            ESP_ERROR_CHECK(tsl2561_init(&m_light_sensor));
            ESP_ERROR_CHECK(bmp280_init_default_params(&m_bme280_params));
            ESP_ERROR_CHECK(bmp280_init_desc(&m_temp_sensor, BMP280_I2C_ADDRESS_0, I2C_NUM_1, GP
            ESP_ERROR_CHECK(bmp280_init(&m_temp_sensor, &m_bme280_params));
        }
```

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

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

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

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

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

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

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

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

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

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

```
$ idf.py flash monitor
Executing action: flash
```

```
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<more logs>
```

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

Troubleshooting

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

- Make sure the hardware setup is correct. Check the addressing configuration for each sensor matches the code.
- Check the CLK and SDA signal lines are connected to the right pins of the sensors. You can try swapping the signal lines to see if anything changes. You can use a logic analyzer for further investigation.
- You can comment out the code for one sensor and only test the other one, then add the excluded sensor to see if anything changes.

Integrating with SD-card over Serial Peripheral Interface (SPI)

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

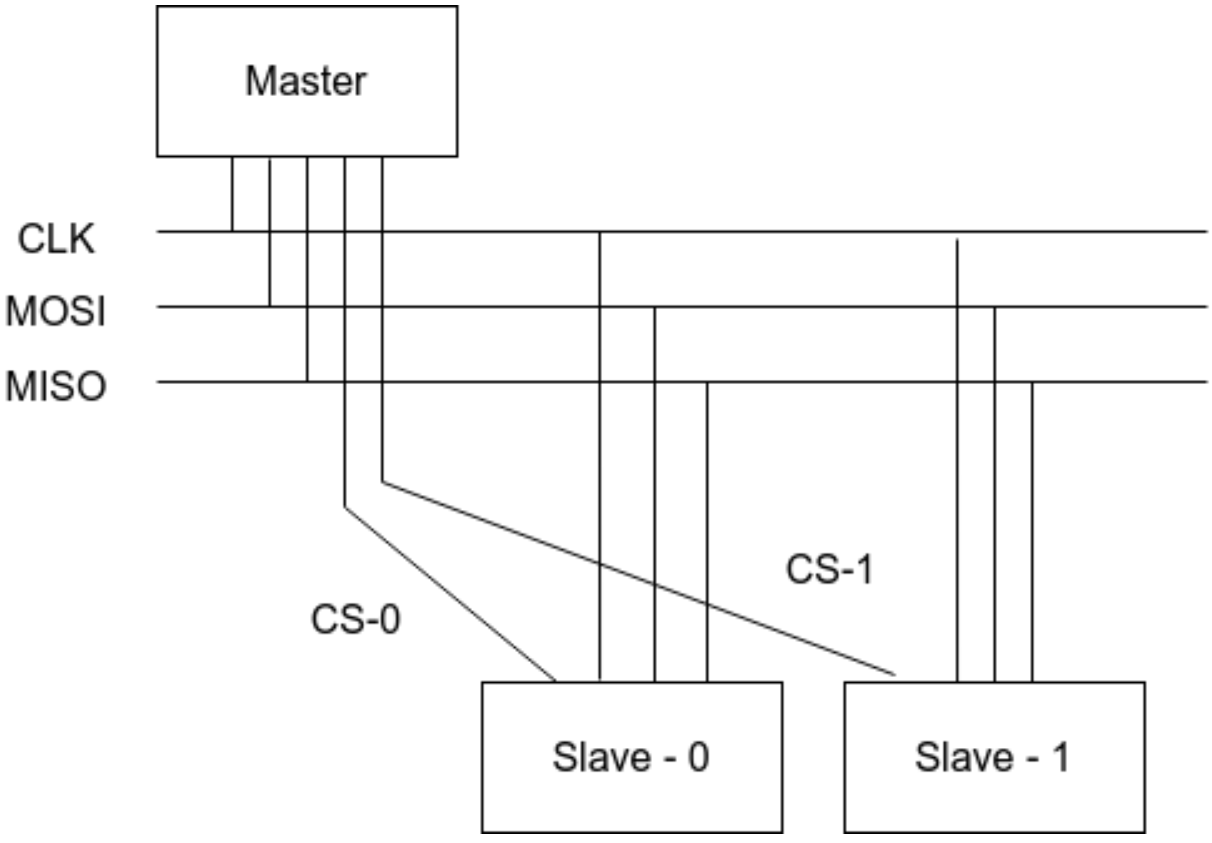

Figure: SPI architecture

In this figure, Master provides clock over the CLK line, uses the Master-Out-Slave-In (MOSI) line to send data, and receives data over the Master-In-Slave-Out (MISO) line. It can communicate with a single slave at a time. To select a slave, it pulls the corresponding Chip-Select (CS) line to low.

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

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

Adding SD-card storage

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

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

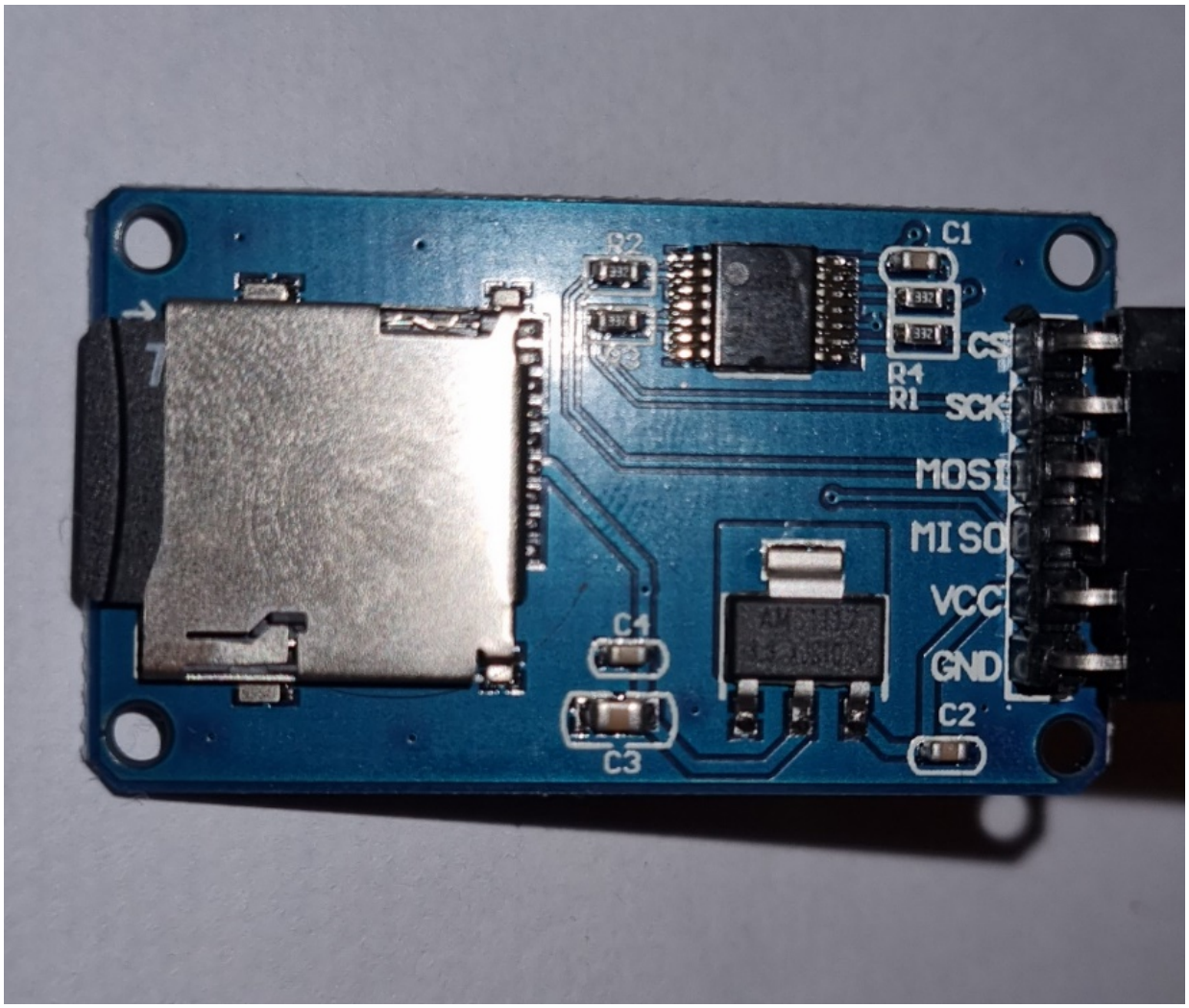

Figure: SD-card breakout board with an SD-card inserted

As you can see on this figure, there are four lines connected to four different GPIO pins of ESP32, CS, SCK, MOSI, and MISO. The following Fritzing sketch shows the connections in this example:

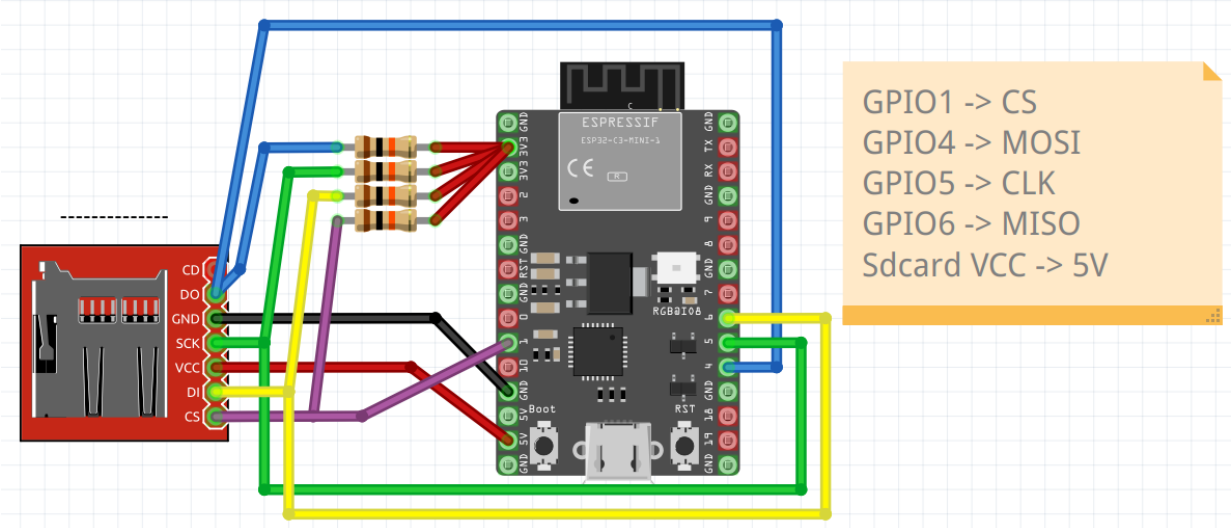

Figure: Fritzing sketch of the project

The hardware components of the project are:

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

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

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

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

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

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

1. We can now remove the **sdkconfig** file to force the **idf.py** tool to generate a new **sdkconfig** next time we run it.

After having the project created, we can move on to coding. Let’s implement a mock class to generate data in a file named **main/AppSensor.hpp**.

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

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

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

The name of the mock class is `AppSensor` . In the private section, we have 3 member variables. The first one holds dummy readings and the next one shows the number of stored readings in the array. `save` is a C++ function object to be called when the array is full. We implement the `read` function in the private section as follows:

```
        static void read(void *d)
        {
            AppSensor *sensor = reinterpret_cast<AppSensor *>(d);
            while (1)
            {
                vTaskDelay(pdMS_TO_TICKS(100));
                SensorData d{20, 50, 1000, 55};
                sensor->readings[sensor->cnt++] = d;
                if (sensor->cnt == 100)
                {
                    sensor->cnt = 0;
                    sensor->save(reinterpret_cast<const uint8_t *>(sensor->readings), 100 * size
                }
            }
        }
```

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

```
    public:
        void init(std::function<void(const uint8_t *, size_t)> fn)
        {
            save = fn;
            if (xTaskCreate(AppSensor::read,
                            "sensor",
                            3072,
                            reinterpret_cast<void *>(this),
                            5,
                            nullptr) == pdPASS)
            {
                ESP_LOGI(__func__, "task created");
            }
        }
    }; // class
} // namespace
```

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

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

There are two new header files here. **esp_vfs_fat.h** provides the interface for the FAT filesystem structures and functions. **sdmmc_cmd.h** is for SD-card access. Both come with ESP-IDF, we don't need any external libraries or framework to be added into the project. After the header files, we can jump into the class implementation.

```
namespace app
{
    class AppStorage
    {
    private:
        constexpr static const gpio_num_t PIN_NUM_MOSI{GPIO_NUM_4};
        constexpr static const gpio_num_t PIN_NUM_MISO{GPIO_NUM_6};
        constexpr static const gpio_num_t PIN_NUM_CS{GPIO_NUM_1};
        constexpr static const gpio_num_t PIN_NUM_CLK{GPIO_NUM_5};
        constexpr static const char *MOUNT_POINT{"/sdcard"};
```

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

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

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

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

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

```
        esp_err_t init(void)
        {
            esp_err_t ret{ESP_OK};
            esp_vfs_fat_mount_config_t mount_config = {
```

```
        .format_if_mount_failed = true,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024};
```

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

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

The `bus_cfg` variable describes the GPIO pin connections and the SPI bus speed with the SD-card. The `host` variable provides abstraction for the application on top of the physical layer. We will use them next in order to initialize the SPI bus and mount the filesystem.

```
    ret = spi_bus_initialize((spi_host_device_t)host.slot, &bus_cfg, SDSPI_DEFAULT_DMA);
    if (ret != ESP_OK)
    {
        return ret;
    }
```

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

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

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

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

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

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

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

```
            std::ofstream file{std::string(MOUNT_POINT) + "/log.bin", std::ios_base::binary | st
```

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

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

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

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

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

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

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

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

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

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

It is time to flash the devkit and test the application.

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C3
<logs removed>
I (311) sdspi_transaction: cmd=52, R1 response: command not supported
I (361) sdspi_transaction: cmd=5, R1 response: command not supported
Name: 00000
Type: SDSC
Speed: 20 MHz
Size: 1875MB
I (371) init: task created
```

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

```
$ xxd LOG.BIN | head -1
00000000: 1400 0000 3200 0000 e803 0000 3700 0000  ....2.......7…
```

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

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

Troubleshooting

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

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

Audio output over Inter-IC Sound (I²S)

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

- Data, Data-In (DIN), or Data-Out (DOUT)
- Clock or bit clock (BCLK)
- Channel select, word select (WS), or left-right clock (LRCLK)

The interface is a standard; however, the naming is not, as we see above. The data line carries stereo audio data for both the left (channel 0) and right (channel 1) channels. The channel select signal level indicates which channel data is currently being transferred: it is low for the left channel and high for the right channel. Finally, the clock line is a common clock for both ends provided by the master, which is usually the sending party in this type of communication.

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

Developing a simple audio player

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

1. Create an ESP-IDF project in any way you prefer. Set its name to **audio_ex** and the chip type of the project as **ESP32S3** since we will use ESP32-S3 Box Lite.
2. There is a long list of default values for **sdkconfig.defaults**. Please copy both the **sdkconfig** and **sdkconfig.defaults** file from the book repository.
3. We will store the MP3 file on the flash. For this, we need to define the flash partitions in a file named **partitions.csv**. Set its contents as the following (we will discuss the fields later):

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

1. We need some external components, so update the content of **CMakeLists.txt** in the project root with the following (Its path is given relative to the project directory. If you clone the book repository, you can directly copy the **components** directory):

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

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

```
idf_component_register( SRCS "audio_ex.cpp" INCLUDE_DIRS ".")
spiffs_create_partition_image(storage ../spiffs FLASH_IN_PROJECT)
```

1. We can try building the project now:

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

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

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

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

The first field is the partition name and we can refer to a partition with its name anywhere in the project. The second and third fields define the partition type together. The offset field shows the offset from the flash beginning. If nothing is specified for this field, it is automatically calculated by adding the given sizes of the previous partitions. The next field denotes the size of the partition, and the final one is to mark the encrypted partitions. The MP3 file will be on the `storage` partition.

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

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

```
spiffs_create_partition_image(storage ../spiffs FLASH_IN_PROJECT)
```

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

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

```
    "flash_settings" : {
        "flash_mode": "dio",
        "flash_size": "16MB",
        "flash_freq": "80m"
    },
```

The `flash_size` field comes from **sdkconfig.defaults**.

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

And, these are the binary images that have been generated during the build. They will be flashed starting from the calculated memory addresses when we run **idf.py flash**. Enough of the talk, let's develop the application.

If you have noticed, we have several other partitions defined in the **partitions.txt** file. One of them is `nvs`, which we can use for storing application settings. In this application, we can store, for instance, the volume level so that every time the application starts, the last volume level is restored to play the MP3 file. Let's write a class for this in **main/AppSettings.hpp**.

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

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

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

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

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

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

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

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

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

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

The `init` function starts with the NVS partition initialization. If it fails, the `init` function just returns since the NVS is not available somehow (you can check the return values to understand the issue if such a thing happens). When the NVS is ready, then we can try to open the NVS namespace to read the settings.

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

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

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

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

This is the entire `AppButton` implementation, thanks to the board support package! In the `init` function of the class, we simply set the button press handlers for the left, middle, and right buttons of the devkit, and that is it. Before moving on to the audio player implementation, it would be a good idea to briefly talk about what happens behind the scenes in the board support package. The following figure is from the devkit schematic:

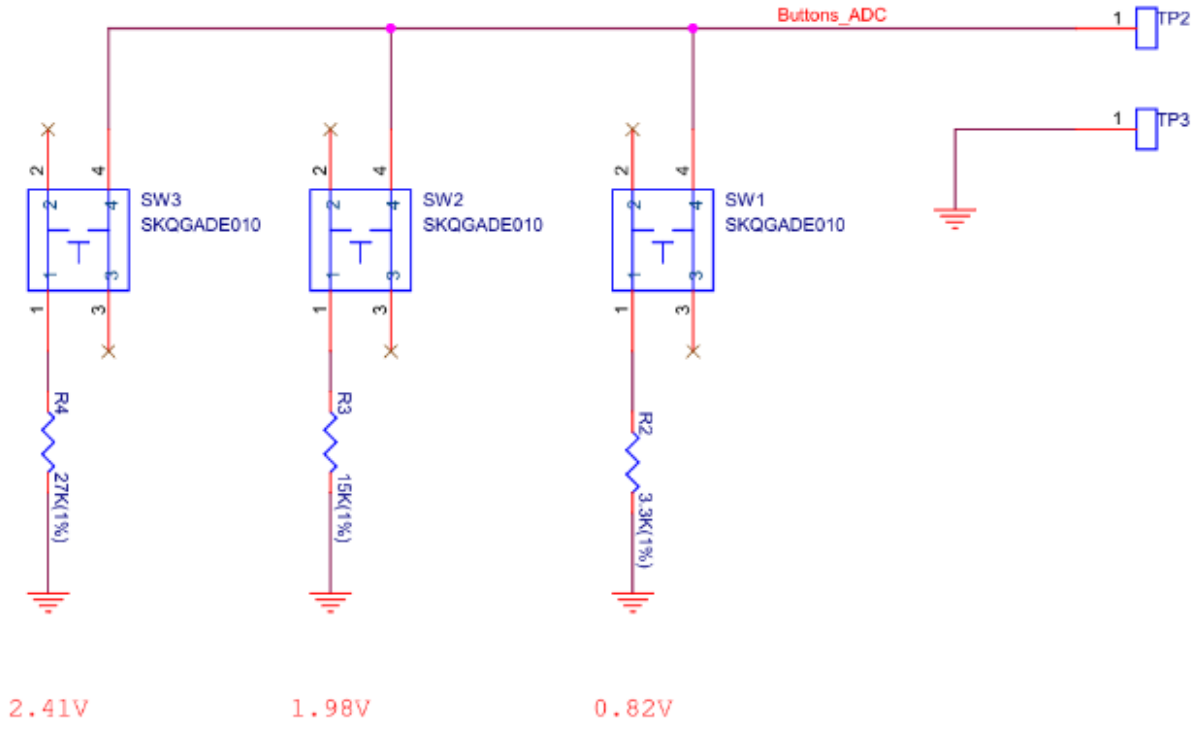

Figure: Buttons schematic

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

Next comes the audio player in **main/AppAudio.hpp**.

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

The devkit has one ADC and one DAC chip integrated on it for audio encode/decode purposes. As we talked about at the beginning of the chapter, ESP32-S3 Box Lite features a ES8156 for the audio output. The `bsp_codec.h` header file provides functionality to drive the audio chips. The `audio_player.h` header file is for playing any audio files (wav or mp3) on the underlying audio system. The `bsp_storage.h` header file provides access to the filesystem on the flash, where the MP3 file resides. With that, we can implement the audio class.

```
namespace app
{
    class AppAudio
    {
    private:
        AppSettings &m_settings;
        bool m_playing;
        FILE *m_fp;
```

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

```
    public:
        AppAudio(AppSettings &settings) : m_settings(settings), m_playing(false), m_fp(nullptr) {
        void init(audio_player_mute_fn fn)
        {
            audio_player_config_t config = {.port = I2S_NUM_0,
                                            .mute_fn = fn,
                                            .priority = 1};
            audio_player_new(config);
        }
```

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

```
        void mute(bool m)
        {
            bsp_codec_set_mute(m);
            if (!m)
            {
                bsp_codec_set_voice_volume(m_settings.getVolume());
            }
        }
```

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

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

The `play` function will open the MP3 file from the filesystem, then the **audio_player_play** function will play it. The flip side is to pause the player by calling the `audio_player_pause` function if the current state is not `m_playing`. The remaining member functions are for volume control.

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

The **volume_up** and **volume_down** are very similar to each other as expected. The former checks if the volume level is less than 100 then increases the volume by 10, the latter checks if the volume level is greater than 0 and decreases the volume by 10. In both functions we first get the current volume from the settings, set the volume by calling the **bsp_codec_set_voice_volume** function and update the settings.

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

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

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

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

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

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

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

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

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

```
    esp_err_t audio_mute_function(AUDIO_PLAYER_MUTE_SETTING setting)
    {
        m_app_audio.mute(setting == AUDIO_PLAYER_MUTE);
        return ESP_OK;
    }
} // end of anonymous namespace
```

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

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<logs removed>
I (572) I2S: DMA Malloc info, datalen=blocksize=640, dma_buf_count=6
I (573) I2S: DMA Malloc info, datalen=blocksize=640, dma_buf_count=6
I (573) I2S: I2S0, MCLK output by GPIO2
I (574) codec: Detected codec at 0x08. Name : ES7243
I (574) codec: Detected codec at 0x10. Name : ES8156
I (580) codec: init ES7243
I (581) codec: init ES8156
I (581) gpio: GPIO[46]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
I (651) bsp_spiffs: Partition size: total: 956561, used: 516809
```

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

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

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

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

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

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

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

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

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

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

A simple graphical user interface (GUI) on ESP32

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

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

- Open-source and free (MIT license)
- Supporting GUI designer (SquareLine Studio – sorry, you need to pay for a license)
- A wide range of visual components (widgets), such as, label, text area, button, slider, list, chart, checkbox, drop-down list, image, etc.
- Containers, such as, canvas, window, tab-view
- Advanced graphics features, such as, animations, anti-aliasing, opacity, etc
- Very little memory footprint for the minimal set of features (64 kB Flash, 16 kB RAM). However, it is worth to note that memory usage increases with more features enabled.
- And a good documentation

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

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

Let's create our first GUI with LVGL with the following steps:

1. Create an ESP-IDF project in any way you prefer. Set its name to **ui_ex** and the chip type of the project as **ESP32S3** since we will use ESP32-S3 Box Lite.
2. There is a long list of default values in **sdkconfig.defaults**. Please copy it from the book repository.
3. We need some external components, so update the content of **CMakeLists.txt** in the project root with the following (Its path is given relative to the project directory. If you clone the book repository, you can directly

copy the **components** directory):

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

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

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

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

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

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

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

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

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

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

```
        void init(btn_handler_f btn_down_handler, btn_handler_f btn_up_handler)
        {
            bsp_btn_register_callback(static_cast<board_btn_id_t>(m_type), BUTTON_PRESS_DOWN, btn
            bsp_btn_register_callback(static_cast<board_btn_id_t>(m_type), BUTTON_PRESS_UP, btn_u
        }
```

In the `init` function, we register for both button down and button up events. Please note that we also pass the `this` pointer to the `bsp_btn_register_callback` function as a parameter to be able to access the class instances from the button callback functions. There is one more function left to be implemented, as it comes next:

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

The `getObject` is a `static` member function, which makes it a class member rather than an object member. The board support package defines all buttons as the type of `button_dev_t` and it passes a `button_dev_t` pointer to the callback functions of the buttons as a means of distinguishing buttons in the callback functions. The same structure has also a `cb_user_data` field which is the `this` pointer to the `AppButton` object that we passed in the

`init` function. Therefore, the `getObject` static function provides access to the `AppButton` object when it is called in the button callbacks.

This completes the `AppButton` class and we can continue with the GUI implementation in **main/AppUi.hpp**.

```
#pragma once
#include <mutex>
#include "bsp_lcd.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
```

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

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

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

```
    public:
        void init(void)
        {
            lv_port_init();
            m_label_title = lv_label_create(lv_scr_act());
            lv_obj_set_style_text_color(m_label_title, lv_color_make(0, 0, 0), LV_STATE_DEFAULT);
            lv_obj_set_style_text_font(m_label_title, &lv_font_montserrat_24, LV_STATE_DEFAULT);
            lv_label_set_text(m_label_title, "Press a button");
            lv_obj_align(m_label_title, LV_ALIGN_CENTER, 0, 0);
            xTaskCreatePinnedToCore(lvglTask, "lvgl", 6 * 1024, nullptr, 3, nullptr, 0);
            bsp_lcd_set_backlight(true);
        }
```

The `init` function initializes the class, hence the name. The first thing is to make the display hardware ready by calling the `lv_port_init` function. Then, we create the label widget with the help of the `lv_label_create` function. Each widget needs a parent container and we pass the active screen by calling the `lv_scr_act` function. After configuring the different features of the label widget, we create the FreeRTOS task for LVGL. Lastly, we

turn on the LCD backlight in the initialization. Next, we will implement the final function in the class to set the label text:

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

In the `setLabelText` function, we create a `std::lock_guard` object before accessing the label. We set the label text by calling the `lv_label_set_text` function and the class implementation ends. It is time to test the application. We flash the devkit and play with the buttons to see how the label text changes with the button press and release events.

```
$ idf.py flash monitor
Executing action: flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<logs removed>
I (724) lv_port: Try allocate two 320 * 20 display buffer, size:25600 Byte
I (727) lv_port: Add KEYPAD input device to LVGL
```

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

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

Summary

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

The next chapter will provide a list of popular IoT libraries with examples. When it comes to IoT applications, 3rd-party libraries are almost inevitable for every project to speed up the product development and reduce the costs. Therefore, it is always better to have an idea about the available options before jumping into development. We will discuss some of those libraries and use cases in the next chapter.

Questions

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

1. What would be the right peripheral to use when you need to drive an LED?
    - ADC
    - GPIO
    - I^2C
    - DAC
2. Which of the following options supports multiple clients on the same bus with addressing?
    - I^2S
    - GPIO
    - I^2C

- SPI

3. Why do SD-cards use SPI communication with MCUs?
   - Less error-prone
   - Less resource hungry (less pins to use)
   - Higher transfer rate
   - Higher memory capacity
4. The I^2S protocol defines a Word-Select (WS) signal:
   - To transfer audio data
   - To clock the bus
   - To select left or right channel
   - To increase the data rate
5. Which of the following is not a display technology generally used in IoT applications?
   - MRI
   - LCD
   - TFT
   - OLED

4 Employing 3rd-party libraries in ESP32 projects

Developing an IoT product usually means that you need help from 3rd-parties because of the practical reasons, such as cost, time, and market needs. It is obvious that every development that we decide to do in-house means more time and more money to burn in order to have a final, working product. However, we can cut some of the costs by using 3rd-party libraries where possible, no need to reinvent the wheel. Market needs can also drive your development decisions. Let's say, if your product has to support a specific type of communication layer, for example, Matter – a popular smart home connectivity protocol, then it would make sense to use an SDK for it for a smooth certification process of your product. In this chapter, we are not going to talk about Matter, but some other popular libraries that you might need in your next IoT project with ESP32.

In this chapter, we will discuss the following free and open-source (FOSS) 3rd-party libraries:

- LittleFS, an alternative to SPIFFS that comes with ESP-IDF
- Nlohmann JSON, a JSON library for modern C++, as advertised
- Miniz, the data compression library included in ESP-IDF
- Flatbuffers, an efficient serialization library by Google
- LVGL, Light and Versatile Embedded Graphics Library
- ESP-IDF Components library (UncleRus)
- The frameworks and libraries by Espressif Systems

Technical requirements

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

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

- Light dependent resistor (LDR)
- A jumper wire (both ends are male)
- A pull-up resistor (10KΩ)

The source code in the examples is located in the repository found at this link: [link]

Check out the following video to see the code in action: [link]

LittleFS

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

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

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

There are different methods to include a 3rd-party library in an ESP-IDF project. One of them is the **IDF Component Manager**. Espressif maintains a list of libraries on its **IDF Component Registry** so that developers can easily access the popular libraries without hassle. They are all compatible with ESP-IDF, as a result, including a library from the IDF Component Registry is only a matter of a single command. We will add LittleFS in our project by using this method.

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

Let's prepare the project in steps:

1. Create an ESP-IDF project

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

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

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

1. This is the step where we include the LittleFS library in the project by using the IDF component manager. The **idf.py** tool provides access to the IDF component manager. The following command will create a file, **main/idf_component.yml**, in the main directory. This file shows the libraries to be included in the project with their versions.

```
$ idf.py add-dependency joltwallet/littlefs==1.5.0
```

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

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

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

```
idf_component_register(SRCS "littlefs_ex.cpp" INCLUDE_DIRS ".")
littlefs_create_partition_image(storage ../files FLASH_IN_PROJECT)
```

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

```
$ mkdir files && echo "this is an example file" > files/file1.txt
```

1. Rename the **main/littlefs_ex.c** source file to **main/littlefs_ex.cpp** and we are ready to develop the project.

As a quick note before moving on to coding, the ESP32 port of the LittleFS library exposes the same API as the official ESP-IDF SPIFFS library. It is enough to replace the `spiffs` prefix with `littlefs` everywhere. Similarly, we use the `littlefs_create_partition_image` function instead of `spiffs_create_partition_image` in **main/CMakeLists.txt**. Now, we can start VSCode and develop the application. Let's edit a new file, **main/AppDoorLogger.hpp**, for the door logger.

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

We include the header files first, as usual. The header file for LittleFS is `esp_littlefs.h`. Then the class implementation comes:

```
namespace app
{
    class AppDoorLogger
    {
    private:
        constexpr static const char *TAG{"door_logger"};
        constexpr static const char *FILENAME{"/files/log.txt"};
```

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

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

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

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

The `getObject` function returns a reference to the class instance so that we can use its member functions. The `doorOpened` function is a button handler that will run when the left button is pressed and `doorClosed` is for the button release. Thus, pressing on the left button means that the door is opened and releasing it marks the door closed. After setting the door states in these functions, we call the `log` member function of the object to log the state change in the `log.txt` file. The implementation of the `log` function follows:

```
        void log(void)
        {
            std::ofstream log_file{FILENAME, std::ios_base::app};
            log_file << "[" << esp_log_system_timestamp() << "]: ";
            log_file << (m_door_state == eDoorState::OPENED ? "opened" : "closed") << "\n";
        }
```

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

In the next static member function, we print the file content, ie. logs collected.

```
        static void print(void *data)
        {
            std::ifstream log_file{FILENAME};
            std::string line1;
            while (!log_file.eof())
            {
                std::getline(log_file, line1);
                ESP_LOGI(TAG, "%s", line1.c_str());
            }
        }
```

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

```
    public:
        void init(void)
        {
            bsp_board_init();
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_DOWN, AppDoorLogger::doorO
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_UP, AppDoorLogger::doorClos
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_PRESS_DOWN, AppDoorLogger::print
```

The `init` function starts with registering the button handlers. We attach the `AppDoorLogger::doorOpened` function to the left button press event, the `AppDoorLogger::doorClosed` function to the left button release event, and the `AppDoorLogger::print` function to the middle button press event. Then we continue with the filesystem initialization:

```
            esp_vfs_littlefs_conf_t conf = {
                .base_path = "/files",
                .partition_label = "storage",
                .format_if_mount_failed = true,
                .dont_mount = false,
            };
            esp_vfs_littlefs_register(&conf);
```

We first define a configuration variable where we specify the partition label to be mounted and the base path for it. The partition label comes from the **partitions.csv** file. The `esp_vfs_littlefs_register` function of the LittleFS library registers the partition given in the configuration variable. Next, we will try to open the log file to see if it really works.

```
            std::ofstream log_file{FILENAME, std::ios_base::trunc};
            if (!log_file.is_open())
            {
                ESP_LOGE(TAG, "file open failed (%s)", FILENAME);
            }
        } // init function end
    }; // class end
}  // namespace end
```

If the partition mount operation fails, then we cannot open the file and print an error message on the serial console. This finalizes the class implementation.

To complete the application, we edit the **main/littlefs_ex.cpp** source file and create an `AppDoorLogger` object in it as the following:

```
#include "AppDoorLogger.hpp"
namespace
{
    app::AppDoorLogger door_logger;
}
extern "C" void app_main(void)
{
```

```
    door_logger.init();
}
```

There is really nothing much to do in the `app_main` function. After creating the `AppDoorLogger` object, named `door_logger`, we only call the `init` function of the object and that is it! Let's test the application by flashing it on the devkit and observe how it responds to the button presses:

```
$ idf.py flash monitor
<logs removed>
I (16556) door_logger: [01:00:16.953]: opened
I (16556) door_logger: [01:00:17.638]: closed
I (16557) door_logger: [01:00:18.434]: opened
I (16557) door_logger: [01:00:19.200]: closed
```

After two presses and releases of the left button, we press the middle button to see the logs on the serial console.

The next library is *Nlohmann-JSON* for JSON processing capabilities in our ESP32 applications.

Nlohmann-JSON

JavaScript Object Notation (JSON) is a common data exchange format that uses human-readable text and Nlohmann-JSON is a popular library that implements the JSON functionality in C++. It is released under the MIT license that allows us to use the library in our projects without any limitations.

> The IDF Component Registry doesn't have Nlohmann-JSON, therefore we can use it in a project by directly downloading the header file from its repository here: https://github.com/nlohmann/json

Nlohmann-JSON provides a single header file as the library, therefore the only thing we need to do is simply to download this header file from its repository and include it in the project.

The goal of the example is to develop a touch logger. We will use ESP32-S3 Box Lite and a simple jumper wire to expose the GPIO9 pin of the devkit on Pmod Header 2 so that we can use it as the touch sensor of the application. When we touch the pin of the jumper wire, it will generate a touch event. All touch events will be collected in the memory. The left button of the devkit will JSON-serialize the touch records. The middle button will navigate through the records and print them on the serial console in JSON format.

Let's create an ESP-IDF application first:

1. Create an ESP-IDF project in any way you prefer.
2. Copy the **sdkconfig.defaults** file from the project directory (**ch4/json_ex**) on Github into the local project directory.
3. We will use the buttons on the devkit. The BSP has the driver and we can include it by specifying in the root **CMakeLists.txt** file. Set the content of this file as the following:

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

1. Download the Nlohmann-JSON header file into the **main** directory.

```
$ cd main && wget https://raw.githubusercontent.com/nlohmann/json/develop/single_include/nlohmann
```

1. Rename the source code file to **main/json_ex.cpp** and set the content of the **main/CMakeLists.txt** file to reflect this change:

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

The blueprint is ready and we can write the `AppTouchLogger` class in the **main/AppTouchLogger.hpp** file.

```
#pragma once
#include <string>
#include <ctime>
#include <vector>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "driver/touch_pad.h"
#include "json.hpp"
```

We begin with including the headers. The `driver/touch_pad.h` header file defines the functions and structures for the touch sensor peripheral of ESP32. The `json.hpp` header file is the one that we have downloaded from the Nlohmann-JSON Github repository. Next comes the structure that defines a touch event.

```
namespace app
{
    struct TouchEvent_t
    {
        uint32_t timestamp;
        uint32_t pad_num;
        uint32_t intr_mask;
    };
```

In this structure, the `timestamp` field denotes the time that the event occurred, `pad_num` is the touch pad number (it is 9 in our project), and `intr_mask` shows the event type. We will only log the touch active and touch inactive events. Then we use a macro from the Nlohmann-JSON library to serialize/deserialize the `TouchEvent_t` structure as the following:

```
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(TouchEvent_t, timestamp, pad_num, intr_mask);
```

The Nlohmann-JSON library provides a means to convert custom structures into JSON objects and vice versa by looking for two designated functions for the custom type: `to_json` and `from_json`. The `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro defines them for us.

The library has a great documentation here: https://json.nlohmann.me/api/adl_serializer/

We can now define the class.

```
    class AppTouchLogger
    {
    private:
        constexpr static const char *TAG{"touch_logger"};
        std::vector<TouchEvent_t> m_touch_list;
```

We will collect all the events in the `m_touch_list` member variable of the class. These events will come from the touch interrupt handler as we define next:

```
        static void touchsensor_interrupt_cb(void *arg)
        {
            TouchEvent_t touch_event{esp_log_timestamp(),
                                     touch_pad_get_current_meas_channel(),
                                     touch_pad_read_intr_status_mask()};
            AppTouchLogger &obj = *(reinterpret_cast<AppTouchLogger *>(arg));
            obj.m_touch_list.push_back(touch_event);
        }
```

The `touchsensor_interrupt_cb` function is a static function in the class definition so that we can provide it as an interrupt handler. We create a touch event first with the help of the ESP-IDF touch pad functions. The `touch_pad_read_intr_status_mask` function shows what happened (active/inactive) and the `touch_pad_get_current_meas_channel` function shows where it happened (only Touch-9 here). We will pass the object address as a parameter when we register the interrupt handler, so the `arg` argument is a pointer to the `AppTouchLogger` object. We append the touch event to the object's touch list. Then, we can develop the initialization function of the class as the following:

```
    public:
        void init(void)
        {
            touch_pad_init();
            touch_pad_config(TOUCH_PAD_NUM9);
```

We initialize the touch peripheral first by calling the `touch_pad_init` and `touch_pad_config` functions. After that, we need to create a touch filter so that the peripheral can generate reliable touch events to the application:

```
            touch_filter_config_t filter_info = {
                .mode = TOUCH_PAD_FILTER_IIR_16,
                .debounce_cnt = 1,
                .noise_thr = 0,
                .jitter_step = 4,
                .smh_lvl = TOUCH_PAD_SMOOTH_IIR_2,
            };
            touch_pad_filter_set_config(&filter_info);
            touch_pad_filter_enable();
            touch_pad_timeout_set(true, SOC_TOUCH_PAD_THRESHOLD_MAX);
            touch_pad_isr_register(touchsensor_interrupt_cb, this, TOUCH_PAD_INTR_MASK_ALL);
            touch_pad_intr_enable(TOUCH_PAD_INTR_MASK_ACTIVE | TOUCH_PAD_INTR_MASK_INACTIVE);
            touch_pad_set_fsm_mode(TOUCH_FSM_MODE_TIMER);
            touch_pad_fsm_start();
            vTaskDelay(pdMS_TO_TICKS(50));
            uint32_t touch_value;
            touch_pad_read_benchmark(TOUCH_PAD_NUM9, &touch_value);
            touch_pad_set_thresh(TOUCH_PAD_NUM9, touch_value * .2);
        }
```

There is a bunch of configuration code in this snippet. For the sake of simplicity, we can just ignore them for now. The only thing to know for this example is that we register the touch sensor interrupt handler by calling the `touch_pad_isr_register` function and enable active/inactive interrupts by calling the `touch_pad_intr_enable` function.

> There is a lot going on here. The capacitive touch sensing and how it works are perfectly described in this application note: https://github.com/espressif/esp-iot-solution/blob/release/v1.0/documents/touch_pad_solution/touch_sensor_design_en.md

The final member function of the class is where we return the internal event vector as a JSON array. Here it is how we can implement it:

```
        nlohmann::json serialize(void)
        {
            return m_touch_list;
        } // function end
    }; // class end
} // namespace end
```

Surprisingly, the `serialize` function has a very short body, thanks to the macro call at the beginning that generates the `to_json` and `from_json` functions for us. The Nlohmann-JSON library knows how to serialize the `TouchEvent_t` type and it also knows how to serialize C++ STL vectors. Therefore, when we return the internal `m_touch_list` member variable, the compiler can implicitly convert it to a `nlohmann::json` object.

The `AppTouchLogger` class is done and now we can implement the application buttons to print the events in JSON format on the serial console. Let's create the **main/AppNavigator.hpp** in the application and edit it.

```
#pragma once
#include "esp_log.h"
#include "bsp_board.h"
#include "bsp_btn.h"
#include "AppTouchLogger.hpp"
#include "json.hpp"
```

We include the header files, including the one that implements the `AppTouchLogger` class, and then we can define the `AppNavigator` class as the following:

```
namespace app
{
    class AppNavigator
    {
    private:
        constexpr static const char *TAG{"nav"};
        AppTouchLogger m_touch_logger;
        nlohmann::json m_touch_list;
        size_t m_list_pos{0};
```

In the private section, we define the member variables first. The `m_touch_logger` member is an instance of the `AppTouchLogger` class that we implemented previously. The touch list, `m_touch_list`, is a JSON object here. After the member variables, we continue with the static functions for the button press handling.

```
        static AppNavigator &getObject(void *btn_ptr)
        {
            button_dev_t *btn_dev = reinterpret_cast<button_dev_t *>(btn_ptr);
            return *(reinterpret_cast<AppNavigator *>(btn_dev->cb_user_data));
        }
```

The `getObject` function extracts the `AppNavigator` object from the `btn_ptr` parameter that comes with the button handler calls. The next static function handles the left button press:

```
        static void countPressed(void *btn_ptr)
        {
            AppNavigator &obj = getObject(btn_ptr);
            obj.m_touch_list = obj.m_touch_logger.serialize();
            obj.m_list_pos = 0;
            ESP_LOGI(TAG, "Touch event count: %u", obj.m_touch_list.size());
        }
```

When we press the left button, we call the `serialize` function of the `AppTouchLogger` object. It returns the touch events as a JSON array and we assign it to the `m_touch_list` member variable. We also set the value of the `m_list_pos` member variable to zero to start the navigation from the beginning of the JSON array.

The middle button is for printing the JSON records of the events on the serial console sequentially. Its handler comes next:

```
        static void nextPressed(void *btn_ptr)
        {
            AppNavigator &obj = getObject(btn_ptr);
            if (obj.m_touch_list.size() <= 0)
            {
                ESP_LOGW(TAG, "no touch detected");
                return;
            }
            ESP_LOGI(TAG, "%s", obj.m_touch_list[obj.m_list_pos].dump().c_str());
            ++obj.m_list_pos;
            obj.m_list_pos %= obj.m_touch_list.size();
        }
```

We can access each event record by index. The index returns another `nlohmann::json` object which is the JSON representation of a touch event. The `dump` function call on the JSON object returns a `std::string` value and we can print it on the serial console by accessing the underlying char array via the `c_str` function. The handlers are ready and we can develop the initialization function of the class in the public section as the following:

```
    public:
        void init(void)
        {
            bsp_board_init();
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_DOWN, AppNavigator::countPr
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_PRESS_DOWN, AppNavigator::nextPr
            m_touch_logger.init();
        } // function end
    }; // class end
} // namespace end
```

In the `init` member function, we register the button handlers and initialize the `m_touch_logger` member. Finally, in the **main/json_ex.cpp** file, we will implement the application entry point, ie. the `app_main` function:

```
#include "AppNavigator.hpp"
namespace
{
    app::AppNavigator nav;
}
extern "C" void app_main(void)
{
    nav.init();
}
```

The `app_main` function is very brief. We only call the `init` function of the `app::AppNavigator` object, and it handles the rest as we have already discussed. Now, we can flash the application and test it by touching the jumper wire pin. After holding and releasing the pin twice, I have got the following output on the serial output when I press the left button first and the middle button several times.

```
$ idf.py flash monitor
I (29374) nav: Touch event count: 4
I (33584) nav: {"intr_mask":2,"pad_num":9,"timestamp":23516}
I (34094) nav: {"intr_mask":4,"pad_num":9,"timestamp":24595}
I (34564) nav: {"intr_mask":2,"pad_num":9,"timestamp":26023}
I (35359) nav: {"intr_mask":4,"pad_num":9,"timestamp":27423}
```

The event count is 4, which is correct for holding and releasing the pin twice since we log both touch pad active/inactive events.

The next library that we will discuss is *Miniz*, the data compression library that comes with ESP-IDF.

Miniz

Miniz is a lossless data compression library that implements RFC 1950 and RFC 1951 for compression/decompression. The library port in ESP-IDF is licensed under the MIT license. ESP-IDF has already imported it for its own purposes but we can also use it freely. There is no need for any library management. Miniz can be especially helpful when you need to transfer a large amount of data. After compressing and sending the data, the receiving side can easily decompress it with any library that implements the same RFCs.

> Unfortunately, the documentation is very poor (in fact, almost none) for Miniz. You can see the examples in this repository: https://github.com/richgel999/miniz

In this example, we will simply compress and decompress a sample text by pressing the buttons on ESP32-S3 Box Lite. There is no other hardware required in this example. Let's start a new ESP-IDF project:

1. Create an ESP-IDF project in any way you prefer.
2. Copy the **sdkconfig.defaults** file from the project directory (**ch4/miniz_ex**) on Github into the local project directory.
3. We will use the buttons on the devkit. The BSP has the driver and we can include it by specifying in the root **CMakeLists.txt** file. Set the content of this file as the following:

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

1. Rename the source code file to **main/miniz_ex.cpp** and set the content of the **main/CMakeLists.txt** file to reflect this change:

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

We can now start the VSCode editor and add a new file for the class that handles the data compression operations. The relative path of the file is **main/AppZip.hpp**.

```
#pragma once
#include <cstring>
#include "esp_log.h"
#include "esp_heap_caps.h"
#include "esp_system.h"
#include "rom/miniz.h"
```

The data compression functions and structures are declared in `rom/miniz.h`. We have another interesting header, `esp_heap_caps.h`, which is for accessing the PSRAM on the ESP32-S3 Box Kit. The devkit has 512KB of internal static RAM (SRAM) and 8MB of external pseudo-RAM (PSRAM). Although the SRAM of the devkit is more than enough for our purposes in this example, we will allocate memory on the PSRAM to learn how to use it.

```
namespace app
{
    class AppZip
    {
    private:
        constexpr static const size_t BUFFERSIZE{1024 * 64};
        char *m_data_buffer;
        char *m_compressed_buffer;
        char *m_decompressed_buffer;
        tdefl_compressor m_comp;
        tinfl_decompressor m_decomp;
```

We define the buffer pointers and the compressor/decompressor member variables. The types, `tdefl_compressor` and `tinfl_decompressor`, come from the Miniz library. The compressor and decompressor will operate on the defined buffers. Let's allocate memory for the buffers on PSRAM:

```
    public:
        void init(void)
        {
            ESP_LOGI(__func__, "Free heap (before alloc): %u", esp_get_free_heap_size());
            m_data_buffer = (char *)heap_caps_malloc(BUFFERSIZE, MALLOC_CAP_SPIRAM | MALLOC_CAP_8
            m_compressed_buffer = (char *)heap_caps_malloc(BUFFERSIZE, MALLOC_CAP_SPIRAM | MALLO(
            m_decompressed_buffer = (char *)heap_caps_malloc(BUFFERSIZE, MALLOC_CAP_SPIRAM | MALI
            ESP_LOGI(__func__, "Free heap (after alloc): %u", esp_get_free_heap_size());
        }
```

In the `init` function, we call the `heap_caps_malloc` function of the heap management library in ESP-IDF. Similar to `malloc`, it allocates memory and returns a pointer to it, but from PSRAM.

> Please see the API documentation for the heap management strategies/capabilities of ESP-IDF here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/mem_alloc.html The subject is important and it would be wise to read the official documentation thoroughly to learn more.

The initialization is completed and we can continue with the `zip` function for data compression.

```
        char *zip(const char *data, size_t &len)
        {
            tdefl_init(&m_comp, NULL, NULL, TDEFL_WRITE_ZLIB_HEADER | 1500);
            memset(m_data_buffer, 0, BUFFERSIZE);
            memcpy(m_data_buffer, data, len);
            size_t inbytes = 0;
            size_t outbytes = 0;
            size_t inpos = 0;
            size_t outpos = 0;
```

The `zip` function takes two arguments, one is the pointer to the data to be compressed and the other one is its length, and it will return a pointer to the compressed data. The `len` parameter is a `size_t` reference and it will show the length of the compressed data upon returning from the function. The `tdefl_init` function, which comes in the Miniz library, initializes the `m_comp` member variable. Then we copy the data to the internal buffer,

`m_data_buffer`, to process it in there. The local variables coming after the copy will be used during the compression. We compress the data in a while loop as the following:

```
            while (inbytes != len)
            {
                outbytes = BUFFERSIZE - outpos;
                inbytes = len - inpos;
                tdefl_compress(&m_comp, &m_data_buffer[inpos], &inbytes, &m_compressed_buffer[ou
                inpos += inbytes;
                outpos += outbytes;
            }
            len = outpos;
            return m_compressed_buffer;
        }
```

In the while loop, we process the input data. The `tdefl_compress` function does the job by reading from `m_data_buffer` and writing to `m_compressed_buffer`. It also updates the local variables to manage the process. When all data is compressed, we update the `len` variable with the compressed data length and return the buffer pointer to the caller. Next, we develop the `unzip` function to decompress data in a very similar manner:

```
        char *unzip(const char *data, size_t &len)
        {
            tinfl_init(&m_decomp);
            if (data != m_compressed_buffer)
            {
                memset(m_compressed_buffer, 0, BUFFERSIZE);
                memcpy(m_compressed_buffer, data, len);
            }
            size_t inbytes = 0;
            size_t outbytes = 0;
            size_t inpos = 0;
            size_t outpos = 0;
```

The `unzip` function takes the compressed data and its length as parameters. We initialize the decompressor by passing it to the Miniz `tinfl_init` function and we define the local variables. Then we will extract the original data from the compressed binary as the following:

```
            while (inbytes != len)
            {
                outbytes = BUFFERSIZE - outpos;
                inbytes = len - inpos;
                tinfl_decompress(&m_decomp, (const mz_uint8 *)&m_compressed_buffer[inpos], &inbyt
                inpos += inbytes;
                outpos += outbytes;
            }
            len = outpos;
            return m_decompressed_buffer;
        } // unzip end
    }; // class end
} // namespace end
```

The function to decompress data is `tinfl_decompress`. It takes the compressed data buffer, the output buffer, and other local variables to manage the process. When all the binary data is processed in the `while` loop, the `unzip` function returns a pointer to the decompressed data with its length updated. The class implementation finishes at this point and we can move on to the implementation of the `AppButton` class to trigger compress/decompress by pressing the buttons on the devkit. We create a new file, **main/AppButton.hpp**, for it and edit the file.

```
#pragma once
#include "bsp_board.h"
#include "bsp_btn.h"
namespace app
{
    using btn_pressed_handler_f = void (*)(void *);
    class AppButton
    {
    public:
        void init(btn_pressed_handler_f l, btn_pressed_handler_f m)
```

```
        {
            bsp_board_init();
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_DOWN, l, nullptr);
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_PRESS_DOWN, m, nullptr);
        }
    };
}
```

The entire `AppButton` implementation is very simple. We only write an initialization function, `init`, where we pass the button handlers as parameters. The left and middle buttons of the devkit are used in this example.

Finally, we can implement the application in the **main/miniz_ex.cpp** source code file.

```
#include <cstring>
#include "esp_log.h"
#include "AppButton.hpp"
#include "AppZip.hpp"
```

We start with including the header files, then, define the application objects and variables in the anonymous namespace as the following:

```
namespace
{
    const char *m_test_str = "this is a repeating text to be compressed. you can try anything\n"
                             "this is a repeating text to be compressed. you can try anything\n"
                             "this is a repeating text to be compressed. you can try anything\n"
                             "this is a repeating text to be compressed. you can try anything";
    app::AppButton m_btn;
    app::AppZip m_zip;
    size_t m_data_len;
    char *m_compressed_data;
    char *m_decompressed_data;
```

We will compress the `m_test_str` string when we press the left button and extract the original text by pressing the middle button. We define the left button handler next:

```
    void zipBtn(void *btn_ptr)
    {
        m_data_len = strlen(m_test_str);
        m_compressed_data = m_zip.zip(m_test_str, m_data_len);
        ESP_LOGI(__func__, "compressed to %u from %u", m_data_len, strlen(m_test_str));
        ESP_LOG_BUFFER_HEX(__func__, m_compressed_data, m_data_len);
    }
```

The `zipBtn` function calls the `zip` function of the `m_zip` object, which is an instance of the `app::AppZip` class. We set the `m_compressed_data` pointer to the returned value from the `zip` function call. The next function is the handler for the middle button:

```
    void unzipBtn(void *btn_ptr)
    {
        m_decompressed_data = m_zip.unzip(m_compressed_data, m_data_len);
        ESP_LOGI(__func__, "%.*s", m_data_len, m_decompressed_data);
    }
}
```

In the `unzipBtn` function, we call the `unzip` function of the `m_zip` object, hence the name. This function call extracts the original text and if everything goes well, we should see the same `m_test_str` text printed on the serial console when we press on the middle button. Let's flash the application and test it by pressing the left button and middle button sequentially. Here is the output of my test.

```
$ idf.py flash monitor
<logs removed>
I (15480) zipBtn: compressed to 71 from 255
I (15480) zipBtn: 78 01 d5 cb c1 0d 80 30 0c 04 c1 3f 55 5c 05 f4
I (15481) zipBtn: 64 82 45 f2 c0 8e ec 43 c2 dd 93 36 90 f6 b9 c3
I (15481) zipBtn: 3e 12 2b 41 e8 54 e1 b0 0b d4 97 a0 e3 50 34 bf
```

```
I (15481) zipBtn: 67 68 a6 9e 3b ca 1f 34 31 30 0a 62 c5 be de 8d
I (15481) zipBtn: ff f6 1f 5d b1 5c f7
I (17509) unzipBtn: this is a repeating text to be compressed. you can try anything
this is a repeating text to be compressed. you can try anything
this is a repeating text to be compressed. you can try anything
this is a repeating text to be compressed. you can try anything
```

The application seems working properly. When I press the left button, it compresses the text to 71 bytes, and the middle button decompresses to the same text.

The next library that we are going to talk about is *Flatbuffers*, which is very useful to pass data between different platforms regardless of their architectures.

Flatbuffers

Flatbuffers is a cross-platform serialization library from Google. The library supports many different programming languages so it is possible to use it on any platform. Another interesting feature of Flatbuffers is that it can directly map binary data to its representation on the platform it runs without parsing or using extra buffers. As a result, the library is quite efficient in terms of both memory usage and processing power.

> The library has a great documentation here: https://google.github.io/flatbuffers/

In the Flatbuffers example, we will collect analog data from a light-dependent resistor (LDR) and convert the data to binary format (serialization) by using the Flatbuffers library. Moreover, we will again employ the library to revert the binary data to the programming structures (deserialization).

The hardware components of the project are:

- ESP32-S3 Box Lite
- An LDR
- A pull-up resistor (10KΩ)

The Fritzing sketch that shows the connections is:

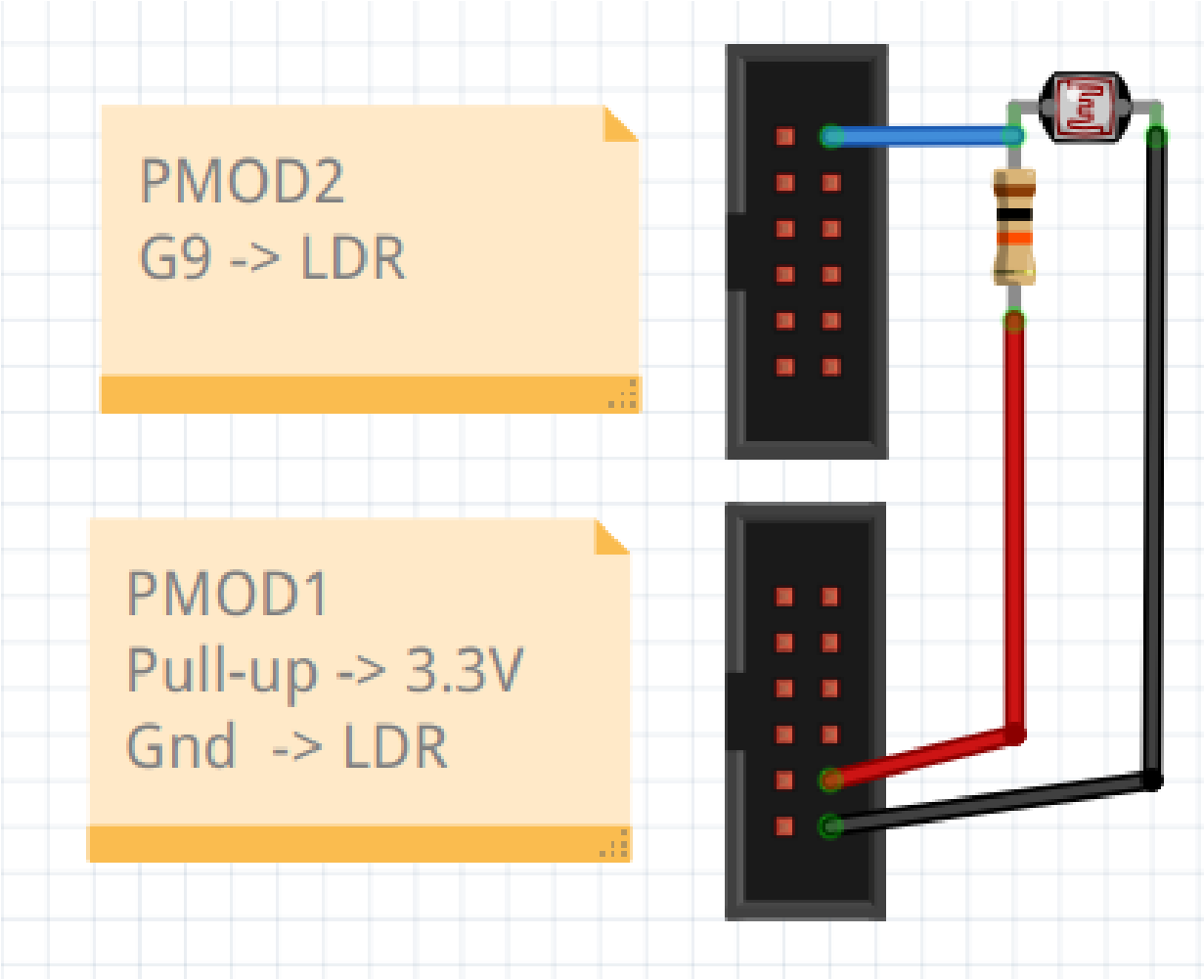

Figure: Fritzing sketch of the project

After having this hardware setup prepared, let's create the application as the following:

1. Create an ESP-IDF project in any way you prefer.
2. Copy the **sdkconfig.defaults** file from the project directory (**ch4/flatbuffers_ex**) on GitHub into the local project directory.
3. We will use the buttons on the devkit. The BSP has the driver and we can include it by specifying in the root **CMakeLists.txt** file. Set the content of this file as the following:

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

1. Clone the Flatbuffers repository into the components directory.

```
$ mkdir components && cd components
$ git clone https://github.com/google/flatbuffers.git
$ rm -rf .git/
```

1. We need to register the Flatbuffers library as an ESP-IDF component in the project. To do that, edit the **components/flatbuffers/CMakeLists.txt** file with the following content:

```
cmake_minimum_required(VERSION 3.10)
set(FlatBuffers_Library_SRCS
  src/idl_parser.cpp
  src/idl_gen_text.cpp
  src/reflection.cpp
  src/util.cpp
)
file(GLOB SOURCES ${FlatBuffers_Library_SRCS})
idf_component_register(SRCS ${SOURCES} INCLUDE_DIRS include)
```

1. We will need the FlatBuffer compiler, **flatc**, to convert a Flatbuffers schema definition into the target programming language. Download it from Github for your development platform:

```
$ mkdir tmp && cd tmp
$ wget https://github.com/google/flatbuffers/releases/download/v22.10.26/Linux.flatc.binary.g++-1
$ unzip Linux.flatc.binary.g++-10.zip
$ ./flatc –version
flatc version 22.10.26
```

1. Rename the source code file to **main/flatbuffers_ex.cpp** and set the content of the **main/CMakeLists.txt** file to reflect this change:

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

We are ready to continue with the development of the project. Let's start VSCode and add the Flatbuffers schema that we are going to use to define our data types. The name of the schema file is **app_data.fbs** in the project root.

```
namespace app;
table ReadingFb {
    timestamp:uint;
    light:ushort;
}
```

Flatbuffers uses its own syntax, **Interface Definition Language** (**IDL**), for this purpose. The first line shows the namespace of all definitions that the Flatbuffers compiler, **flatc**, will generate. Then we define a table for LDR readings. The `ReadingFb` table has two fields, `timestamp` and `light`. When this is converted, it will be a C++ struct.

This document explains the syntax of IDL:
https://google.github.io/flatbuffers/flatbuffers_guide_writing_schema.html

The `ReadingFb` table describes a reading from LDR, but we also want to define a light sensor in the schema as the following:

```
table LightSensorFb {
    location:string;
    readings:[ReadingFb];
}
root_type LightSensorFb;
```

The second table in the schema is `LightSensorFb`, which describes a light sensor. A light sensor has a `location` and an array of `readings`. In the last line, we set the root type as `LightSensorFb` for the code generation. Any Flatbuffers binary for this schema will have a light sensor data at its root. Next we will run the **tmp/flatc** compiler to generate the C++ header file that corresponds to this schema.

```
$ tmp/flatc  -b -t -c --gen-object-api app_data.fbs
$ mv app_data_generated.h main/
```

The output of the flatc compiler is **app_data_generated.h** and we move it to the **main** directory where the source codes of our project reside.

The `--gen-object-api` flag instructs the compiler to generate an object-based API. It is not needed in most projects, but for the sake of simplicity, we will use this API to easily access the sensor data in this example.

Then, we develop the LDR logger class in the **main/AppLdrLogger.hpp** file:

```
#pragma once
#include <vector>
#include <cinttypes>
#include <memory>
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/adc.h"
#include "flatbuffers/idl.h"
#include "flatbuffers/util.h"
#include "app_data_generated.h"
```

As mentioned at the beginning, we will read light values from an LDR. The `driver/adc.h` header file provides us with the analog-digital conversion (ADC) functionality so that we can convert the analog output of the LDR to digital values that we can process in the application. We also include the `app_data_generated.h` header file to be able to serialize the collected light data. We continue with the class definition next:

```
namespace app
{
    class AppLdrLogger
    {
    private:
        LightSensorFbT m_light_sensor;
        const adc1_channel_t m_adc_ch = ADC1_CHANNEL_8;
```

In the private section of the `AppLdrLogger` class, we define the member variables. The first one is an instance of `LightSensorFbT`, a C++ struct that is a part of the generated object API. If you look back to the schema, its fields are `location` and `readings`. The ADC channel is `ADC1_CHANNEL_8` and it is associated with the PMOD2/G9 pin of the devkit.

We can move on to the public section of the class:

```
    public:
        void init(void)
        {
            adc1_config_width(ADC_WIDTH_BIT_12);
            adc1_config_channel_atten(m_adc_ch, ADC_ATTEN_DB_11);
            m_light_sensor.location = "office";
        }
```

In the `init` member function, we first initialize the ADC peripheral. The resolution is 12 bits, which means we can read a value in the range of 0 to 4095 from the LDR. After specifying the ADC channel attenuation, we set the sensor `location` as `office`. The next member function reads from the LDR:

```
        void run(void)
        {
            while (1)
            {
                vTaskDelay(pdMS_TO_TICKS(5000));
```

The `run` function will be a FreeRTOS task. The purpose is to make periodic readings from the sensor with an interval of 5 seconds. After 5 seconds pass, we read from the ADC channel as the following:

```
                uint32_t adc_val = 0;
                for (int i = 0; i < 32; ++i)
                {
                    adc_val += adc1_get_raw(m_adc_ch);
                }
                adc_val /= 32;
```

We get 32 consecutive readings from the ADC channel and take the average as the final value (oversampling). Then we will record this value as we do next:

```
                auto reading = std::unique_ptr<ReadingFbT>(new ReadingFbT());
                reading->timestamp = esp_log_timestamp();
                reading->light = (uint16_t)adc_val;
                m_light_sensor.readings.push_back(std::move(reading));
            }
        }
```

Each record has a `timestamp` and `light` values. We append the record to the `m_light_sensor.readings` vector. Please note that we haven't defined these types manually anywhere in the C++ source code, they are automatically generated and part of the Flatbuffers object-API in this particular project. The last member function serializes the sensor and so the collected data associated with it:

```
        size_t serialize(uint8_t *buffer)
        {
            flatbuffers::FlatBufferBuilder fbb;
            fbb.Finish(LightSensorFb::Pack(fbb, &m_light_sensor));
            memcpy(buffer, fbb.GetBufferPointer(), fbb.GetSize());
            size_t len = fbb.GetSize();
            m_light_sensor.readings.clear();
            return len;
        } // serialize end
    }; // class end
} // namespace end
```

The `serialize` function takes a pointer where the binary output of the serialization will be copied to. In the function, we define a `flatbuffers::FlatBufferBuilder` object, which converts the `m_light_sensor` object to the binary representation when its `Finish` function is called in the next line. After copying the binary data to the given buffer, we clear all the readings from the `m_light_sensor` object until a new call to the `serialize` function of the `AppLdrLogger` class.

The logger class is finished and next we will develop a client class that deserializes the binary data back into a `LightSensorFbT` object and use it for its own purposes. Let's add a new file, **main/AppLdrClient.hpp**, for it.

```
#pragma once
#include <cinttypes>
#include "esp_log.h"
#include "flatbuffers/idl.h"
#include "flatbuffers/util.h"
#include "app_data_generated.h"
```

Again, we include the same generated model for the client class. The definition of the class is as the following:

```
namespace app
{
    class AppLdrClient
    {
    private:
        LightSensorFbT m_light_sensor;
    public:
        void consume(const uint8_t *buffer)
        {
            app::GetLightSensorFb(buffer)->UnPackTo(&m_light_sensor);
            ESP_LOGI(__func__, "location: %s", m_light_sensor.location.c_str());
            for (auto &&rec : m_light_sensor.readings)
            {
                ESP_LOGI(__func__, "ts: %u, light: %d", rec->timestamp, rec->light);
            }
        }
    };
}
```

The implementation of the `AppLdrClient` class is quite simple. We define a member variable, `m_light_sensor`, into which we are going to deserialize binary data. The `consume` function does this deserialization operation. When it is called, the `consume` function receives the binary buffer and reads it into the `m_light_sensor` object by calling the `UnPackTo` function of the object that is returned by the `GetLightSensorFb` function call. The

implementation of these functions comes in the header file generated by the *flatc* compiler. After the deserialization, we can use the `m_light_sensor` object in any way we need.

We will make use of the devkit buttons to prove our Flatbuffers serialization/deserialization implementation works as intended. Pressing the left button will call the logger's `serialize` function and the middle button will call the client's `consume` function. We can develop this button class in the **main/AppButton.hpp** file as the following:

```
#pragma once
#include "bsp_board.h"
#include "bsp_btn.h"
namespace app
{
    using btn_pressed_handler_f = void (*)(void *);
    class AppButton
    {
    public:
        void init(btn_pressed_handler_f l, btn_pressed_handler_f m)
        {
            bsp_board_init();
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_DOWN, l, nullptr);
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_PRESS_DOWN, m, nullptr);
        }
    };
}
```

Again, it is a very brief class to implement. The `init` function of the `AppButton` class takes two function parameters as handlers to be registered for the left and middle buttons.

We have all the necessary classes implemented. The only coding work left is to integrate them in the **main/flatbuffers_ex.cpp** file.

```
#include <cinttypes>
#include "esp_log.h"
#include "esp_heap_caps.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "AppButton.hpp"
#include "AppLdrLogger.hpp"
#include "AppLdrClient.hpp"
```

We include our class implementations and then continue with the anonymous namespace of the application as the following:

```
namespace
{
    constexpr const char *TAG{"app"};
    constexpr const size_t BUFFERSIZE{16u * 1024};
    uint8_t *m_buffer;
```

In the anonymous namespace, we define a buffer pointer. We will allocate memory for the buffer from the PSRAM when the application starts. We define the objects of the application next:

```
    app::AppButton m_btn;
    app::AppLdrLogger m_logger;
    app::AppLdrClient m_client;
```

The objects are the instances of the classes that we have implemented earlier. We will close the anonymous namespace with the `loggerTask` function:

```
    void loggerTask(void *param)
    {
        m_logger.run();
    }
} // namespace end
```

The `loggerTask` function is actually a wrapper for the `run` function of the `m_logger` object and will be a FreeRTOS task in the application. If you remember, the `run` function will collect data from the LDR every 5 seconds. Let's continue with the `app_main` function:

```
extern "C" void app_main(void)
{
    m_buffer = reinterpret_cast<uint8_t *>(heap_caps_malloc(BUFFERSIZE, MALLOC_CAP_SPIRAM));
```

As promised, we allocate memory for `m_buffer` from the PSRAM by calling the `heap_caps_malloc` function. It will be the buffer where we will store the serialized data. A lambda function comes next to do this:

```
    auto serialize = [](void *)
    {
        ESP_LOGI(TAG, "serializing..");
        size_t len = m_logger.serialize(m_buffer);
        ESP_LOG_BUFFER_HEX(TAG, m_buffer, len);
    };
```

The `serialize` lambda function calls the `serialize` function of `m_logger` with `m_buffer` as the parameter. Then we print the content in hexadecimal format on the serial console. We also need to show that the deserialization works. Here comes another lambda function for it:

```
    auto deserialize = [](void *)
    {
        ESP_LOGI(TAG, "deserializing..");
        m_client.consume(m_buffer);
    };
```

This time we call the `consume` function of `m_client` with `m_buffer` as its parameter. It will convert the binary data in the buffer to a sensor object and print the object content on the serial console. These two lambda functions are the press handlers of the button object to be initialized, as we do next:

```
    m_btn.init(serialize, deserialize);
    m_logger.init();
    xTaskCreate(loggerTask, "logger", 3072, nullptr, 5, nullptr);
}
```

We initialize the `m_btn` object, we initialize the `m_logger` object, and start a FreeRTOS task with the `loggerTask` function for data collection before we finish up the `app_main` function.

The application is now ready for testing and we can flash it to see how it works:

```
$ idf.py flash monitor
<logs removed>
I (29453) app: serializing..
I (29454) app: 0c 00 00 00 08 00 0c 00 04 00 08 00 08 00 00 00
I (29454) app: 64 00 00 00 04 00 00 00 05 00 00 00 4c 00 00 00
I (29454) app: 34 00 00 00 24 00 00 00 14 00 00 00 04 00 00 00
I (29454) app: d0 ff ff ff 00 00 15 0e e0 63 00 00 dc ff ff ff
I (29455) app: 00 00 61 01 58 50 00 00 e8 ff ff ff 00 00 73 0c
I (29455) app: d0 3c 00 00 f4 ff ff ff 00 00 97 0c 48 29 00 00
I (29455) app: 08 00 0c 00 08 00 06 00 08 00 00 00 00 00 24 0c
I (29455) app: c0 15 00 00 06 00 00 00 6f 66 66 69 63 65 00 00
I (31398) app: deserializing..
I (31398) consume: location: office
I (31398) consume: ts: 5568, light: 3108
I (31398) consume: ts: 10568, light: 3223
I (31399) consume: ts: 15568, light: 3187
I (31399) consume: ts: 20568, light: 353
I (31399) consume: ts: 25568, light: 3605
```

After waiting half a minute for the light data to be collected, when we press the left button of the devkit, the Flatbuffers serialization occurs. The binary data is displayed on the console as hex numbers. Then, when we pressed the middle button, the actual sensor content is printed this time, which corresponds to the deserialization. You can play with it to see how the readings change when the LDR is exposed to the different levels of light.

Flatbuffers can be a very efficient solution when it is used properly. It is a good way of exchanging data between different platforms.

In the next topic, we will discuss another great library, *Light and Versatile Graphics Library*, or *LVGL* for short.

LVGL

LVGL is one of the most popular graphics libraries for embedded systems. There are many factors that I can count here as the reasons for its popularity:

- It is first and foremost really light-weight compared to the functionality and widgets that comes with it.
- Fully configurable, modern widgets
- Simple API with plain C structures and callbacks
- Extensive documentation with examples
- Great support from the team
- Free and open source with MIT license (The GitHub repository: https://github.com/lvgl/lvgl)

We have already developed an example with LVGL in the previous chapter while talking about displays and GUI development. In this example, we will have more chances to discuss LVGL and its capabilities in detail. We will develop different screens where we can navigate by using the devkit buttons and also interact with the widgets that we add on those screens. Let's start.

One feature that I haven't mentioned in the introduction of LVGL is that it has a GUI designer, named **SquareLine Studio**. Although we can design and develop everything by hand, the GUI designer has some obvious benefits, such as, fast GUI development, easy visualizing and in-place testing. The tool requires a license but it is free for personal use, therefore you can use SquareLine freely while fiddling with the examples of the book.

> You can download SquareLine Studio from this URL: https://squareline.io/downloads The documentation explains the usage well but there are also many learning videos on YouTube: https://www.youtube.com/@squarelinestudio/videos

We won't delve into the GUI design with SquareLine Studio, nonetheless, I will share screenshots from the designer to be able to explain the examples in this book better.

Figure: GUI screens with SquareLine

In this application, we will have four screens with a different widget on each of them. The buttons on the devkit will provide navigation, the left button to go left and the right button for the next screen on the right. The middle button will be for changing the state or property of the widget on the current screen. For example, when we are on **Screen1** with a **Text area**, pressing the middle button will update the text inside this text area. Let's prepare the application for development as in the following steps:

1. Create an ESP-IDF project in any way you like.

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

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

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

1. We need the code for the GUI that we discussed above. You have two options here. You can either use SquareLine to generate the C code by opening the SquareLine project that I have provided in the Github repository under the **design** directory and export for **ESP-BOX**, or copy the generated C code directly from the repository in the **main** directory. There are five files:

```
$ ls main/ui*
main/ui.c  main/ui_events.c  main/ui.h  main/ui_helpers.c  main/ui_helpers.h
```

1. Rename the **main/lvgl_ex.c** source file to **main/lvgl_ex.cpp** and update **main/CMakeLists.txt** to reflect this change. We also need to add the GUI files as well. The content of the **main/CMakeLists.txt** should be:

```
idf_component_register(SRCS "lvgl_ex.cpp" "ui_helpers.c" "ui.c" "ui_events.c" INCLUDE_DIRS ".")
```

The project is ready for further development but if you choose to use SquareLine to generate the GUI C code, then you need to follow the steps below:

1. After starting SquareLine, import the project by selecting File/Open from the main menu and then clicking on the IMPORT PROJECT button of the dialog that is just displayed. You should be able to see the project icon named `lvgl_ex.spj` on the same dialog if the import succeeds. Double-click on it to open the GUI design.
2. From the main menu, select **Export/Export UI Files** and choose the **main** directory in the project root so that the GUI sources are placed in the same directory with the default project source code.

Having the project structure ready, we can start to develop the application, first the application button in **main/AppButton.hpp**:

```
#pragma once
#include "bsp_board.h"
#include "bsp_btn.h"
namespace
{
    template <board_btn_id_t I, button_event_t E>
    void button_event_handler(void *param);
}
```

The button event handler is a template function with a button id and event type as template parameters. We will implement the function body at the end of this file. Next, we continue with some supporting definitions:

```
namespace app
{
    struct sAppButtonEvent
    {
        board_btn_id_t btn_id;
        button_event_t evt_id;
    };
    using fAppButtonCallback = void (*)(sAppButtonEvent &);
```

We will use the `sAppButtonEvent` structure to pack the button press information into a single parameter when passing it to a callback function of type `fAppButtonCallback`. Then comes the class definition:

```
    class AppButton
    {
    private:
        fAppButtonCallback m_btn_cb;
```

In the private section of the `AppButton` class, we keep a member variable as the callback when a button is pressed. It will be the connection point of the class to any client code. The public section of the class is as the following:

```
    public:
        void init(fAppButtonCallback cb)
        {
            m_btn_cb = cb;
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_DOWN, button_event_handler<
            bsp_btn_register_callback(BOARD_BTN_ID_NEXT, BUTTON_PRESS_DOWN, button_event_handler<
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_PRESS_DOWN, button_event_handle
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_PRESS_UP, button_event_handler<B
        }
```

The `init` function takes a callback function as parameter and sets the `m_btn_cb` member variable to it. In the `init` function body, we register four callbacks for all three buttons of the devkit: only for press-down events of the left and right buttons and both press-down and release events for the middle button. In the public section, we have two more functions:

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

The `getObject` function is a static function that returns a reference to the `AppButton` object that registers the button handler. The `btn_ptr` parameter of the function points to that button. The `runCallback` member function simply calls the `m_btn_cb` callback function with the button event. The `AppButton` class is finished. Next we implement the `button_event_handler` function body:

```
namespace
{
    template <board_btn_id_t I, button_event_t E>
    void button_event_handler(void *btn_ptr)
    {
        app::AppButton &app_btn = app::AppButton::getObject(btn_ptr);
        app::sAppButtonEvent e{I, E};
        app_btn.runCallback(e);
    } // function end
} // anonymous namespace end
```

In the `button_event_handler` function, we find the `AppButton` object from the `btn_ptr` pointer and run its callback with the event. Since the `sAppButtonEvent` event structure contains both the button id and the event type, any client code will know what happened to a button when it initializes the `AppButton` object with a callback function.

We have completed the `AppButton` development and can move on to the GUI management by editing a new file, **main/AppUi.hpp**:

```
#pragma once
#include <mutex>
#include <vector>
#include "bsp_lcd.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
#include "ui.h"
#include "AppButton.hpp"
```

There are three header files related to LVGL. The `lvgl/lvgl.h` file encloses the core library functionality, the `lv_port/lv_port.h` header is for linking the devkit hardware and LVGL, and the last one, `ui.h`, has the definitions of the application, ie. the real GUI. We can begin with the class implementation next:

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

In the private section, we define a `mutex` that controls access to the LVGL memory objects. The `lvglTask` function will be a FreeRTOS task that runs periodically and renders all the changes on the active screen. In a while loop in this function, we first acquire the access by creating a lock on the mutex, thus, no other task can change anything while the `lv_task_handler` function is running. Then, we define the remaining private members:

```
        std::vector<lv_obj_t *> m_screens;
        int m_scr_pos;
```

We keep a `vector` of pointers to the LVGL screens that we have designed and the index of the active screen, `m_scr_pos`, on this vector. The left and right buttons will change this index and command LVGL to set the active screen. Let's move on to the public section of the class:

```
    public:
        void init(void)
        {
            lv_port_init();
            ui_init();
            m_scr_pos = 0;
            m_screens.push_back(ui_Screen1);
            m_screens.push_back(ui_Screen2);
            m_screens.push_back(ui_Screen3);
            m_screens.push_back(ui_Screen4);
            xTaskCreatePinnedToCore(lvglTask, "lvgl", 6 * 1024, nullptr, 3, nullptr, 0);
            bsp_lcd_set_backlight(true);
        }
```

The `init` function does several important jobs to start the GUI. First, it calls the `lv_port_init` function to bind the hardware and LVGL. Then, it runs the `ui_init` function to create all LVGL screens. This function is a part of the code generated by SquareLine. Next, we collect all the screen pointers in the `m_screens` vector. In this way, we complete the LVGL initialization and create a FreeRTOS task for periodic screen updates. At the end of the `init` function, we turn the backlight of the LCD on to activate it. The next function that we are going to implement is the button event handler:

```
        void buttonEventHandler(sAppButtonEvent &btn_evt)
        {
            std::lock_guard<std::mutex> lock(m_ui_access);
```

The `buttonEventHandler` function takes an argument of type `sAppButtonEvent`. The first thing we do in the `buttonEventHandler` function is to acquire the access to the LVGL memory objects by creating a lock on the `m_ui_access` member variable. The function argument, `btn_evt`, carries the button event information and we will check it to respond accordingly:

```
        switch (btn_evt.btn_id)
        {
        case board_btn_id_t::BOARD_BTN_ID_PREV:
            m_scr_pos = (m_scr_pos - 1) % m_screens.size();
            lv_scr_load(m_screens[m_scr_pos]);
            break;
        case board_btn_id_t::BOARD_BTN_ID_NEXT:
            m_scr_pos = (m_scr_pos + 1) % m_screens.size();
            lv_scr_load(m_screens[m_scr_pos]);
            break;
```

In a switch statement, we check which button has generated the event. If it is the left button (`BOARD_BTN_ID_PREV`), we set the `m_scr_pos` index one less to point to the previous screen and call the `lv_scr_load` function of the LVGL library to render the newly pointed screen. Remember that it will be actually drawn in the next cycle of the LVGL FreeRTOS task. We do the same thing for the right button but in the opposite direction.

> It is possible to configure LVGL to handle the buttons as an input device, but I preferred to handle the button events manually to show the flow a bit easier. You can refer to the online API document about input devices here: https://docs.lvgl.io/master/overview/indev.html

We also need to respond to the middle button press and release events:

```
        case board_btn_id_t::BOARD_BTN_ID_ENTER:
        {
            switch (m_scr_pos)
            {
            case 0:
                updateTextArea(btn_evt.evt_id);
                break;
            case 1:
                updateLvButtonState(btn_evt.evt_id);
                break;
            case 2:
                toggleSwitch(btn_evt.evt_id);
                break;
            case 3:
                toggleSpinnerVisibility(btn_evt.evt_id);
                break;
            default:
                break;
            }
        }
        break;
        default:
            break;
        } // switch end
    } // function end
```

The handling of the middle button event depends on the active screen. In another switch statement, we check which screen is active and take the appropriate action. For instance, if it is **Screen1**, then we update the text area on it by calling the `updateTextArea` function. Similarly, we call other functions for the widgets that we have on the screens. Let's implement these functions one by one:

```
void updateTextArea(button_event_t btn_evt_id)
{
    if (btn_evt_id == button_event_t::BUTTON_PRESS_DOWN)
    {
        lv_textarea_add_text(ui_Screen1_TextArea1, "button down\n");
    }
    else
    {
        lv_textarea_add_text(ui_Screen1_TextArea1, "button up\n");
    }
}
```

On **Screen1**, we have a text area. We append a text to its end by calling the `lv_textarea_add_text` function of the LVGL library, showing the button event. The next function is for **Screen2**:

```
        void updateLvButtonState(button_event_t btn_evt_id)
        {
            if (btn_evt_id == button_event_t::BUTTON_PRESS_DOWN)
            {
                lv_event_send(ui_Screen2_Button1, LV_EVENT_PRESSED, nullptr);
            }
            else
            {
                lv_event_send(ui_Screen2_Button1, LV_EVENT_RELEASED, nullptr);
            }
        }
```

The widget on **Screen2** is an LVGL button. When we press the middle button of the devkit, we call the `lv_event_send` function with the `LV_EVENT_PRESSED` event so that the GUI button on the screen can also update itself with this state change. The release of the middle button is updated on the GUI as well. We continue with **Screen3**:

```
        void toggleSwitch(button_event_t btn_evt_id)
        {
            static bool checked{false};
            if (btn_evt_id == button_event_t::BUTTON_PRESS_UP)
            {
                checked = !checked;
                if (checked)
                {
                    lv_obj_add_state(ui_Screen3_Switch1, LV_STATE_CHECKED);
                }
                else
                {
                    lv_obj_clear_state(ui_Screen3_Switch1, LV_STATE_CHECKED);
                }
            }
        }
```

There is a switch widget on **Screen3**. Pressing the middle button will toggle its checked state. We set and clear its state by calling the `lv_obj_add_state` and `lv_obj_clear_state` functions respectively. The last function is for **Screen4**:

```
        void toggleSpinnerVisibility(button_event_t btn_evt_id)
        {
            static bool hidden{false};
            if (btn_evt_id == button_event_t::BUTTON_PRESS_UP)
            {
                hidden = !hidden;
                if (hidden)
                {
                    lv_obj_add_flag(ui_Screen4_Spinner1, LV_OBJ_FLAG_HIDDEN);
                }
                else
                {
                    lv_obj_clear_flag(ui_Screen4_Spinner1, LV_OBJ_FLAG_HIDDEN);
                }
            }
        } // function end
    }; // class end
    std::mutex AppUi::m_ui_access;
} // namespace end
```

We have a spinner on **Screen4**. This time, the effect of pressing the middle button is the widget visibility toggle. The visibility of objects in LVGL is maintained as a flag and regardless of the widget type we can change the visibility by calling the `lv_obj_add_flag` and `lv_obj_clear_flag` functions with the `LV_OBJ_FLAG_HIDDEN` parameter on the target object. We apply these functions on the spinner here.

The implementation of the `AppUi` class is finished and we can now integrate the pieces in **main/lvgl_ex.cpp** to complete the application.

```
#include "bsp_board.h"
#include "AppUi.hpp"
#include "AppButton.hpp"
namespace
{
    app::AppUi m_app_ui;
    app::AppButton m_app_btn;
}
```

After including the class headers, we define the `AppUi` and `AppButton` objects in the anonymous namespace. Then the `app_main` function comes next as the entry point of the application:

```
extern "C" void app_main(void)
{
    bsp_board_init();
    auto btn_evt_handler = [](app::sAppButtonEvent &e)
    {
        m_app_ui.buttonEventHandler(e);
    };
    m_app_ui.init();
    m_app_btn.init(btn_evt_handler);
}
```

The critical point here is the `btn_evt_handler` lambda function. It connects the `m_app_ui` and `m_app_btn` objects. We initialize the button object with this lambda function as the button handler and inside it, we pass the event information to the UI object, `m_app_ui`. As a result, when we press any button on the devkit, this event will propagate to the UI and the LCD of the devkit will be updated accordingly. There is no console output worth to note here, so it is the best to flash the app and test it by playing with the devkit buttons:

```
$ idf.py flash
```

This example is only an introduction to the LVGL library and there are many other widgets and features that you might want to try. Please visit the online API documentation and see what LVGL provides more. The next library that we will discuss is the ESP-IDF Components library.

ESP-IDF Components library

It is impossible not to mention the ESP-IDF Components library by UncleRus in this chapter. It is one of the most famous libraries in the community. We have already used a port of this library in *Chapter 3* where we discussed I^2C communication and integrated different sensor breakout boards. The library especially focuses on the I^2C devices and maintains many drivers for sensors from different vendors in a single repository. The driver licenses are all categorized under the FOSS classification but please check for the specific license type before including a sensor driver in your project.

The Github repository of the ESP-IDF Components library is here: https://github.com/UncleRus/esp-idf-lib

The library officially supports ESP32, ESP32-S2, ESP32-C3, and ESP8266. However, as of writing this book, it failed to run I^2C devices on ESP32-S3 because of some kind of timing issue. The book repository contains an updated version of the ESP-IDF Components library to make it compatible with ESP32-S3 series chips.

We won't have an example of this library here since we have already made use of it in the previous chapter. If you need a driver from this library, its repository contains sample applications for each of them.

Espressif Frameworks and Libraries

Besides all these 3rd-party libraries, Espressif Systems powers developers with many other frameworks and libraries. As a quick overview:

- Espressif IoT Solution: this framework brings different hardware drivers together as a working solution to minimize the compatibility issues with ESP-IDF. It contains sensor driver, display controller drivers and LVGL, input devices and buttons, audio output utilities, and more functionality for hassle-free development. (The GitHub repository: https://github.com/espressif/esp-iot-solution)
- Audio Development Framework (ESP-ADF): This is the core framework for audio input/output and processing. It collects all the necessary components under the same roof to develop audio applications, such as, music player/recorder, speech recognition applications, smart speakers, etc. (GitHub: https://github.com/espressif/esp-adf)
- Image Processing Framework (ESP-WHO): It provides image processing capabilities, such as, face detection/recognition, motion detection. You can also use ESP-WHO to develop barcode/QR-code scanners. The Deep Learning library (ESP-DL) powers this framework under the hood. (GitHub: https://github.com/espressif/esp-who)
- ESP RainMaker: This is probably one of the most interesting frameworks by Espressif Systems. Actually, it is better to call it as a platform. ESP RainMaker is composed of many different system components in order to enable developers to develop cloud-based Artificial Intelligence of Things (AIoT) products. These components are: device-agent SDK, the Rainmaker cloud, and the mobile phone apps. (RainMaker home: https://rainmaker.espressif.com/)

For the other framework and libraries by Espressif, you can check the following URLs:

- ESP-IDF Programming Guide: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/libraries-and-frameworks/libs-frameworks.html
- IDF Component Registry: https://components.espressif.com/
- And, of course, the Espressif GitHub repository: https://github.com/espressif

The IoT landscape is extremely dynamic and versatile, therefore, it is practically impossible to provide an exhaustive list when it comes to sharing resources. My personal approach for learning is to select a sample project that I'm interested in, and develop it further with more ideas. The research during the development leads to many other resources and ideas with more learning opportunities. Similarly, you can choose one of the examples in this book as a starting point and find alternative solutions by adding other 3rd-party libraries in your project.

This ends the chapter and we will develop a full-fledged project together in the next chapter, as a practice of what we have learned so far.

Summary

In this chapter, we covered a selection of 3rd-party libraries that we can use in our ESP32 projects. We learned that we can use LittleFS as an alternative to SPIFFS, Nlohmann-JSON as a modern JSON library, Miniz for data compression, and Flatbuffers to share data between different platforms and architectures. LVGL has a special place among them, such that it is the framework if you want to create an amazing graphical interface for the users of your product. ESP-IDF Components library is another popular library for ESP32 developers and it provides many device drivers that we can employ in our projects. Finally, we talked about some important frameworks by Espressif. They provide a head-start and make life much easier when commencing a new ESP32 project.

The next chapter is devoted to a complete project where we will implement an audio player with a GUI. The project will show us how to combine pieces into one application by applying good design and development practices. ESP-ADF and LVGL will be the primary components of the application.

Questions

You can answer the following questions as a review of the topics in this chapter:

1. Which one is the easiest way of adding a library into an ESP32 project? (Assume that all options are available for that library)
   A. IDF Component Registry
   B. Adding as a single header file

C. Cloning from its Github repository
D. Including from ESP-IDF

2. Which one would be a good use case of Flatbuffers?
   A. Serializing data on an external flash
   B. Sending data to a mobile application
   C. Querying an I^2C sensor
   D. Formatting log data into JSON
3. There is a good amount of data to be transferred over WiFi with repeating information. Which method would help most to reduce the data size?
   A. Formatting into JSON by using the Nlohmann-JSON library.
   B. Binary serialization with Flatbuffers
   C. Using Miniz to compress data
   D. Ignoring repeated information.
4. Which one is NOT correct about the use of LVGL in a project?
   A. It is a GUI library so the device should have a display.
   B. LVGL provides API for input devices, such as, keypads or touchscreens
   C. There is no need for a GUI designer.
   D. LVGL comes with the drivers for all display controllers
5. Which framework or library is NOT provided by Espressif Systems?
   A. ESP-IDF Components Library
   B. ESP-ADF
   C. ESP-WHO
   D. ESP RainMaker

5 Project – Audio Player

We have learned a lot so far. We began the journey with the very basics of IoT development on the ESP32 platform, discussed how to connect an ESP32 to the external world by employing its peripherals, and then talked about some popular 3rd-party IoT libraries to speed up software production. Let's cement them by developing a full-fledged project where we can practice our new skills.

In this chapter, we will design and develop an audio player with visual controls as we cover the following topics:

- Feature list of the audio player
- Solution architecture
- Implementation

Technical requirements

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

As hardware, only ESP32-S3 Box Lite will be employed.

You can find the project files here in the repository: [link]

Feature list of the audio player

A basic audio player usually has a playlist control and a volume control. Users can switch between the different recordings and also control the volume. Let's itemize those features for our project:

- Play/pause a recording by pressing a button
- Navigate to the next recording
- Navigate to the previous recording
- Each recording has an associated image. While the user navigates over the playlist, the image is updated on the screen automatically.
- Mute/unmute the volume.
- Increase/decrease the volume
- All the functions have a visual feedback on the LCD display

The requirements are clear enough to begin with, but let's look at the following simple UI design prepared by using *SquareLine Studio* to have a better grasp of what we will do in this project.

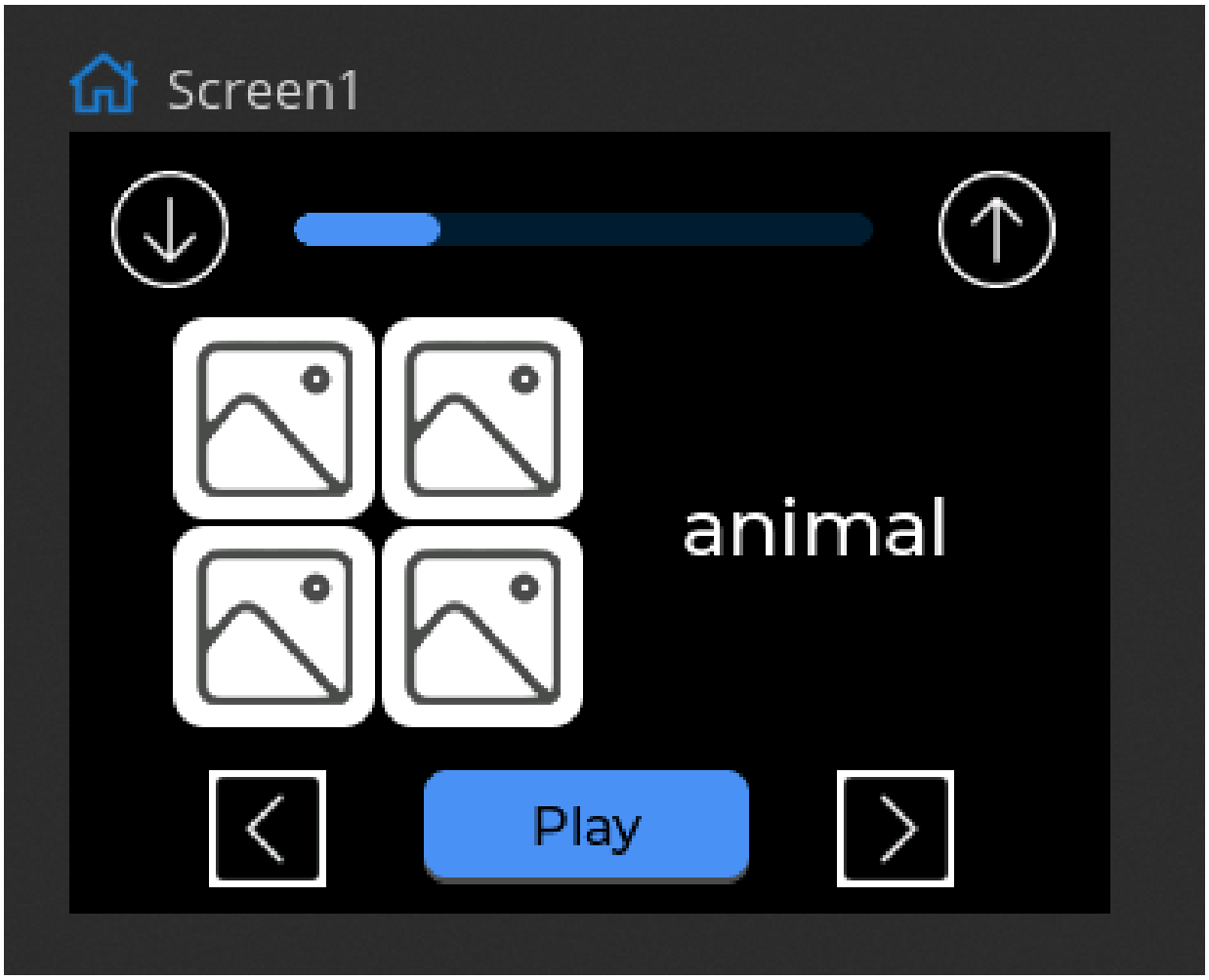

Figure: GUI of the application

Since the devkit has no touchscreen, we will use the physical buttons on it to provide the required functionality. On the top of the screen, we have the volume control. It has two visual buttons for volume up/down. When the user presses the left physical button of the devkit, it will decrease the volume and at the same time, a press event will be sent to the corresponding visual button (top-left) to indicate this event on the screen. The user can increase the volume by pressing the right button of the devkit. The same rule applies for the volume-up button on the top right of the screen. We have a bar in between them. It shows the volume level. The other requirement is to mute/unmute the player. We can use the middle button of the devkit to toggle the mute status.

We will have several animal sounds and images on the flash. The audio player will traverse them. Under the volume control section of the screen, there will be an image container and a label to show the current animal. Both will be updated automatically when the user changes the current animal.

On the bottom of the screen, we will place the playlist control. The idea is very similar to the volume control. The left button of the devkit is for going to the previous animal in the list, the right button is for the next animal. The middle button will play/pause the sound of the selected animal.

We need to switch between the playlist control and the volume control. Let’s assign the middle-button double-click event for this. When the user double-clicks on the middle button of the devkit, the focus toggles between the playlist control and the volume control. To show which one is activated, we can change the color of the play/pause button or the volume-level bar to another color. Let’s say the active control will be red.

Solution architecture

For this list of features, we can have the following classes and relations between them to implement the requirements:

- **AppAudio**: Provides a simple interface for the audio functionality: mute/unmute, play/pause, and volume up/down. It initializes the audio sub-system and delivers audio events to its clients.
- **AppButton**: Handles user press events on the devkit buttons and notifies any client code via an event queue.
- **AppNav**: Keeps track of the animal list. It is the information source for the other components of the application and updates the current animal metadata when the user navigates by using the devkit buttons.
- **AppUi**: Encapsulates the generated UI files and manages the application flow for user interactions by communicating with the other classes mentioned above.

The following class diagram roughly depicts the idea:

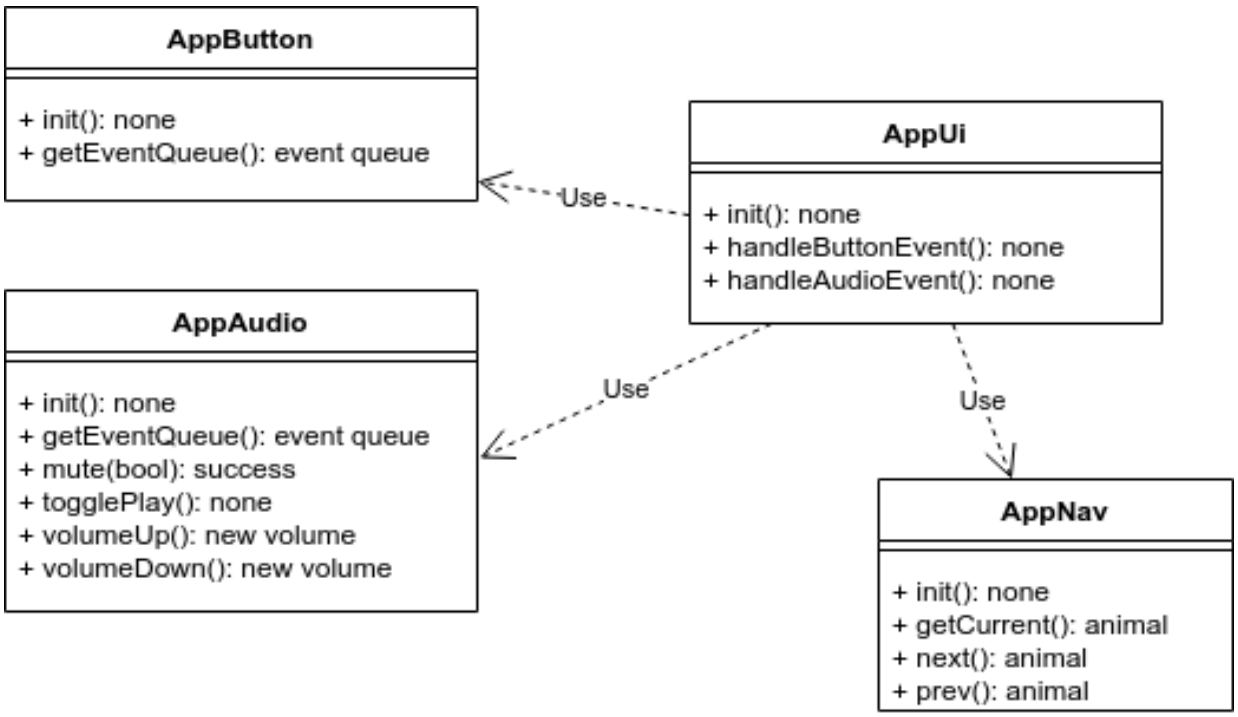

Figure: Class diagram

The implementation will have more supporting fields and methods, for sure, but this diagram shows the classes, their main responsibilities and the relations between them. After the application starts and the initialization of the class instances is done, the **AppUi** instance takes control and begins to receive button events from the **AppButton** instance. According to the active functionality (playlist navigation or volume control), it commands the other class instances (**AppNav** or **AppAudio**). **AppUi** uses the generated UI elements to show the application state to the user. We will store the image files, sounds files and a metadata file on the flash. **AppNav** will read the metadata file to populate the animal list and the multimedia files.

With this, we can implement the application next.

Implementation

There are two different aspects that we need to consider: the GUI design and the application development. The application will have a graphical user interface (GUI) to engage users with visual indicators. We are going to use

LVGL for this purpose, as we have already learned how to use it in the previous chapter. After having the GUI, we can integrate it in the application and move on from there with the implementation of the actual application to react to user inputs.

Let's start with the GUI design.

Graphical User Interface (GUI)

Although you can just copy the generated UI files from the project repository, you can also give a try to design it by yourself by using *SquareLine Studio*. It is impossible to describe every details here, nonetheless, I will list the fundamental steps below:

1. Start *SquareLine Studio* and create a new project for **Espressif** / **ESP-BOX**.

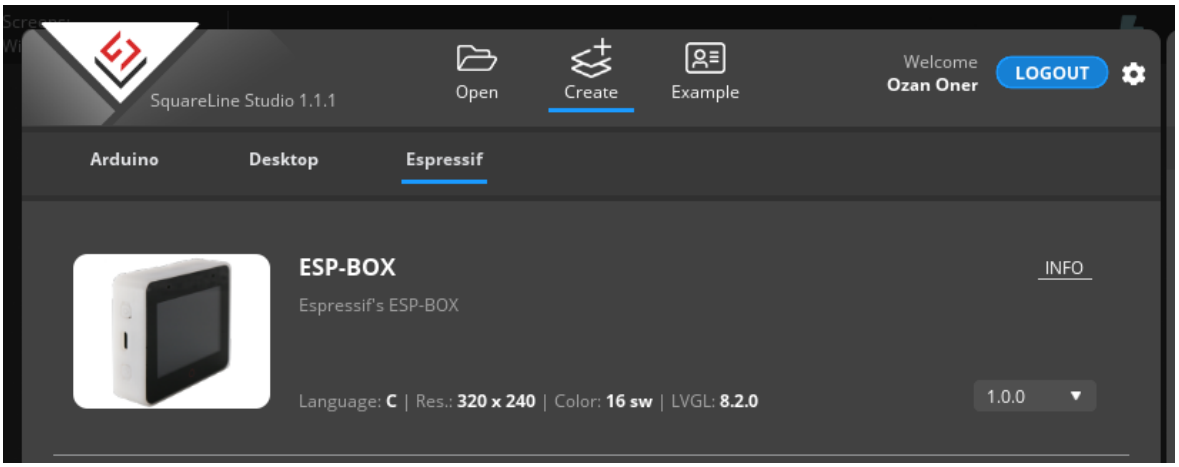

Figure: Create a new SquareLine project.

2. Click on the **Imgbutton** widget of the **Widgets** / **Controller** panel. It will place an image button on **Screen1**. It will be the volume-down button. Drag it to the top-left.

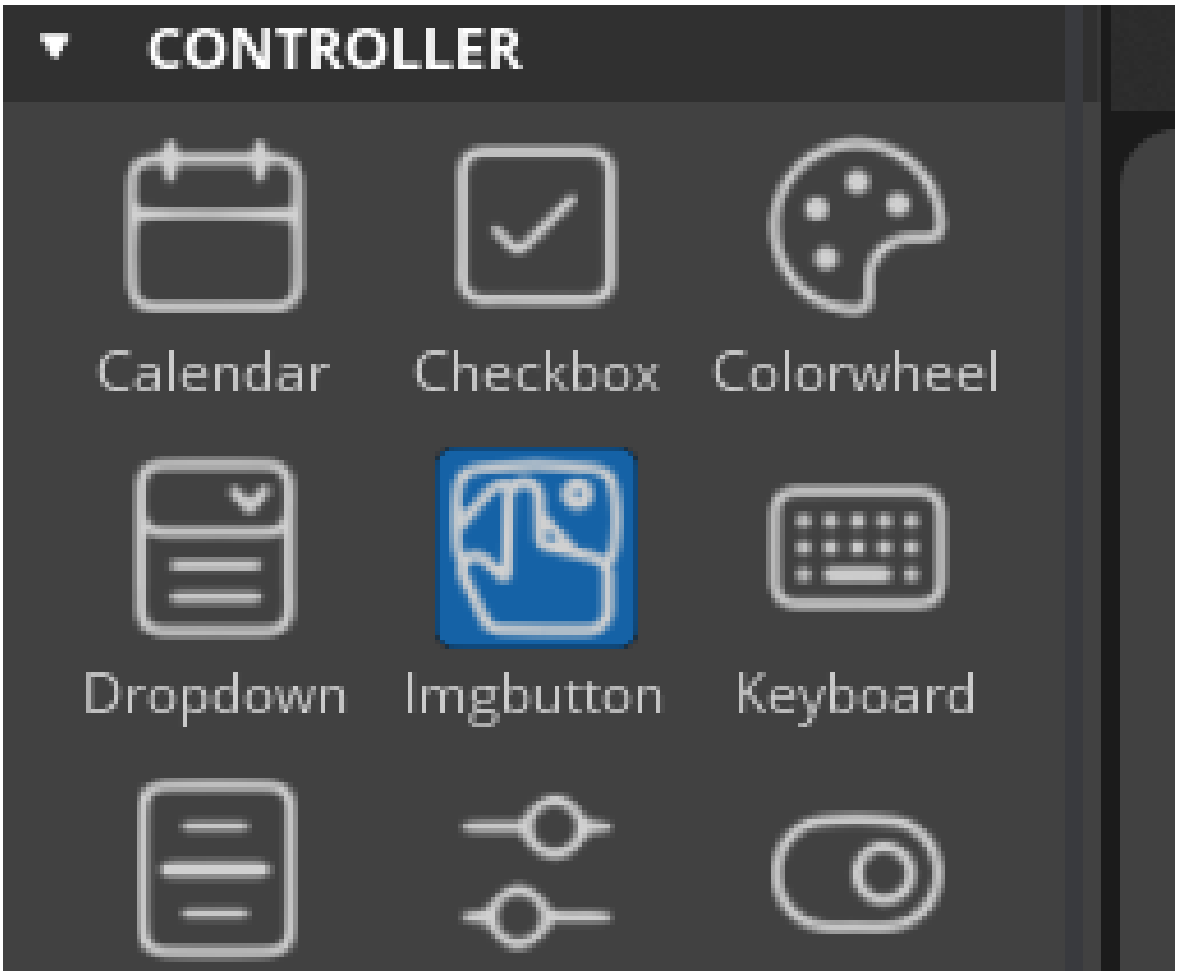

Figure: Image button

3. Download the icons from the project repository (the relative path of the icons is: **ch5/audio_player/gui_design/assets)** and add them into the SquareLine project. They will appear in the **Assets** panel.

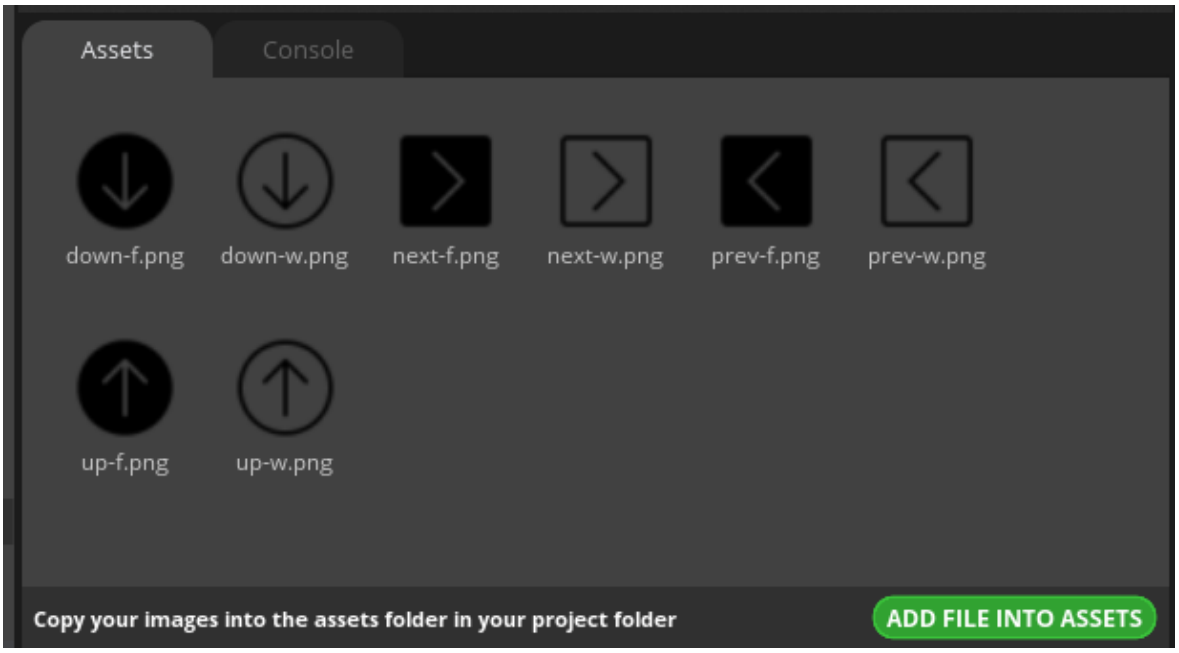

Figure: Icons in the Assets panel

4. Use the **down-f.png** and **down-w.png** files for the pressed and released states of the image button. You can edit the **Imgbutton** widget properties on the **Inspector** panel of the designer.

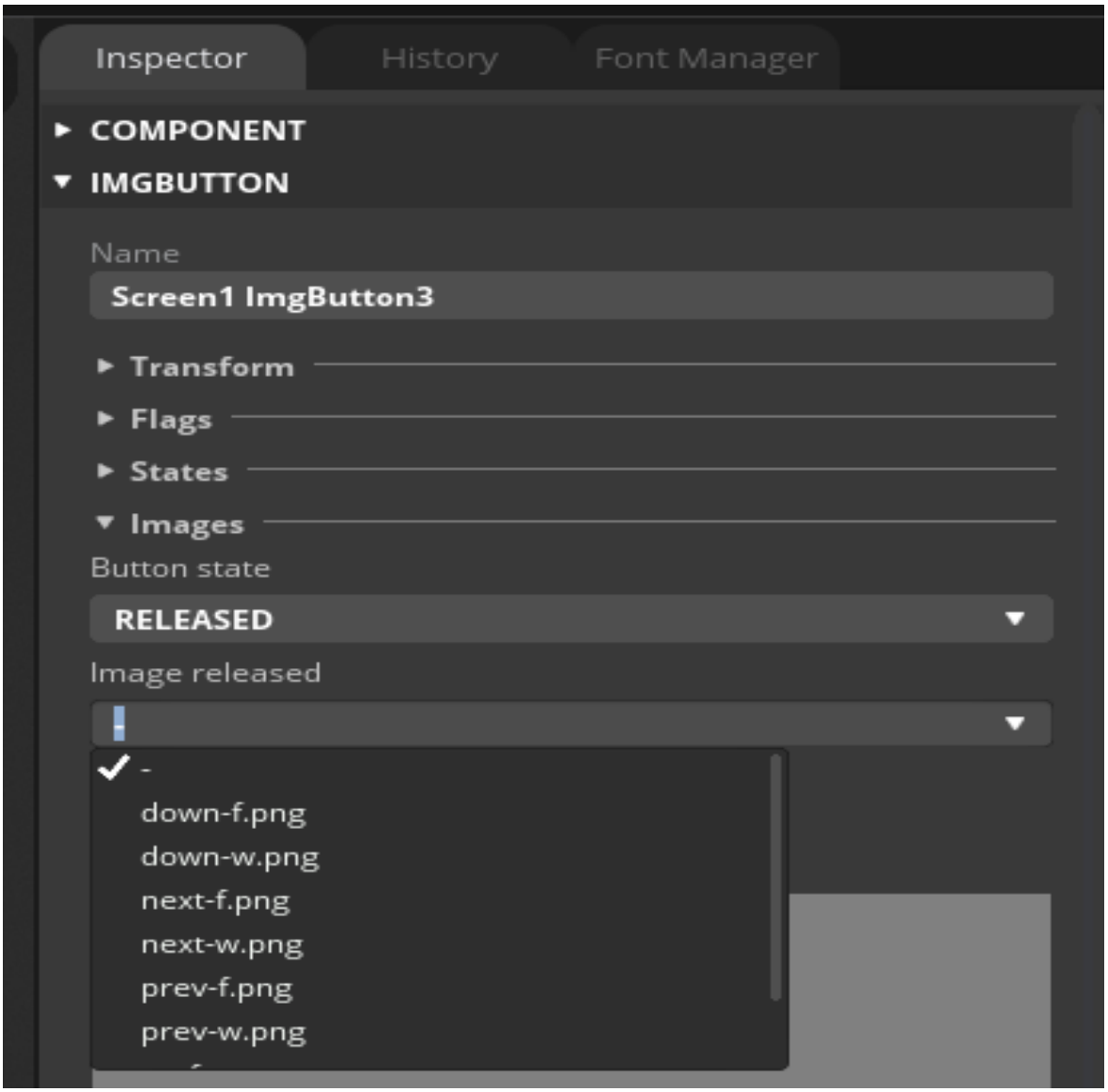

Figure: Image button properties

5. You can enable the play mode to see how the button behaves when pressed and released.

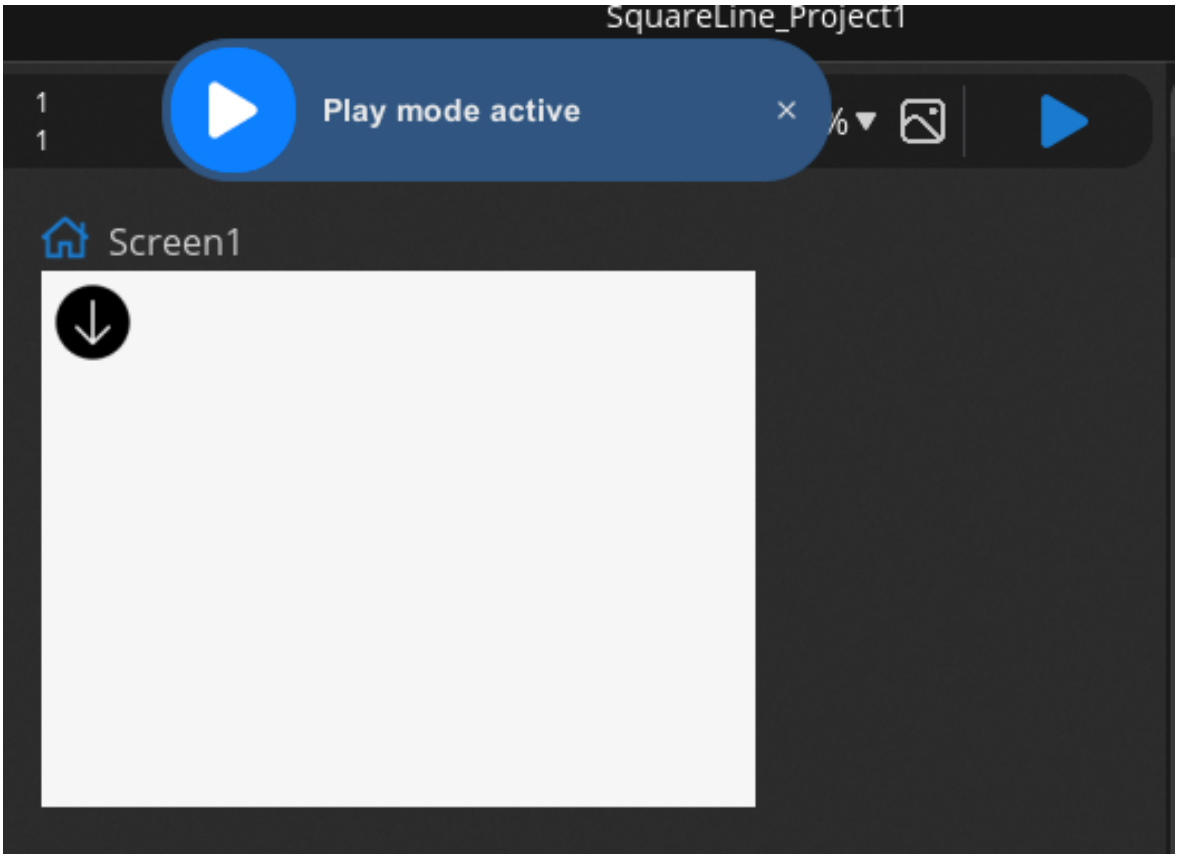

Figure: Play mode

6. Do the same for the remaining three buttons (volume-up, playlist next, playlist previous).

7. Place a **Bar** widget on **Screen1** by selecting from the **Widgets** / **VISUALISER** panel. It will show the volume level.

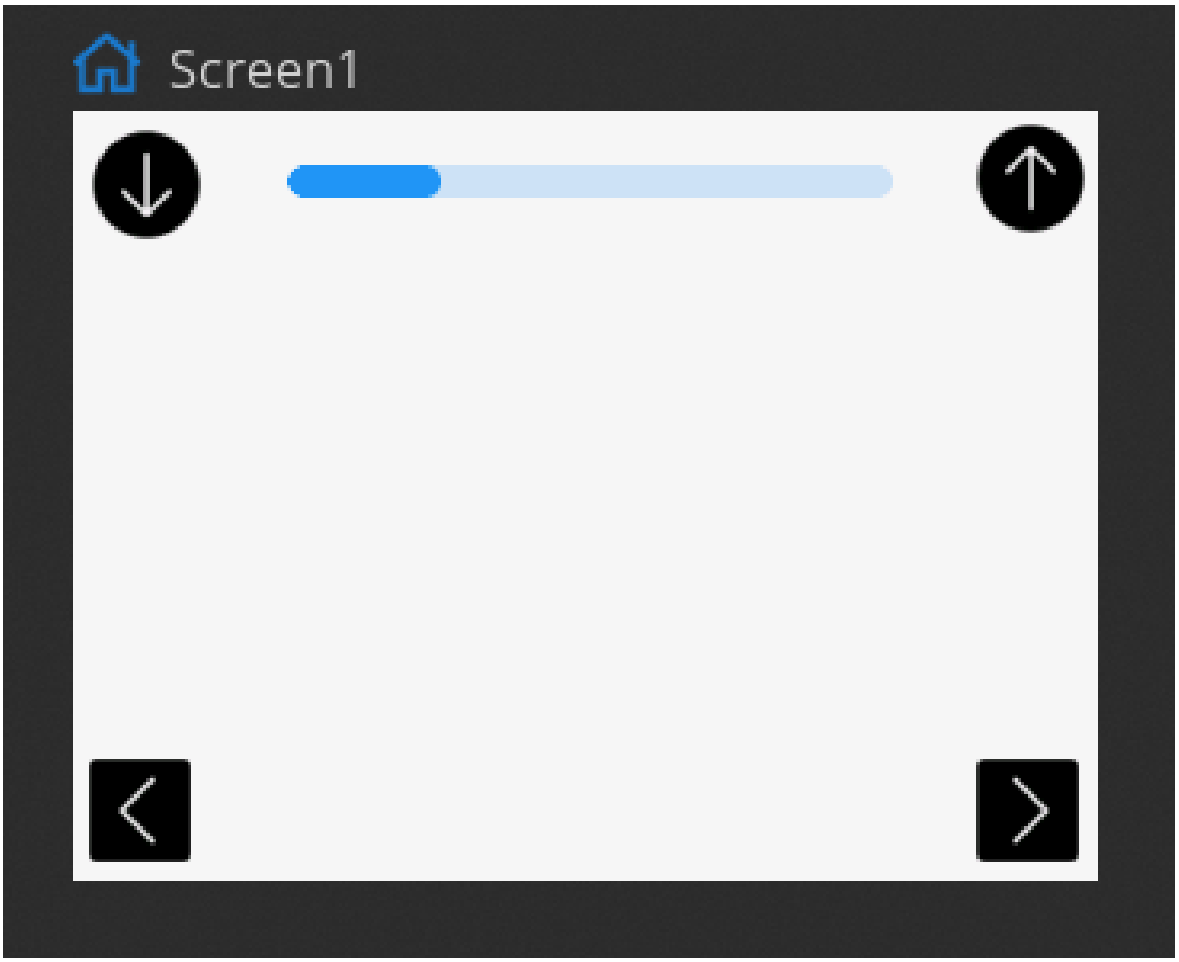

Figure: Bar widget

8. For the play button, we will use two widgets: a button and a label. After placing them on **Screen1**, go to the **Hierarchy** panel and drag and drop the label widget onto the button widget. The button widget will be a container for the label widget.

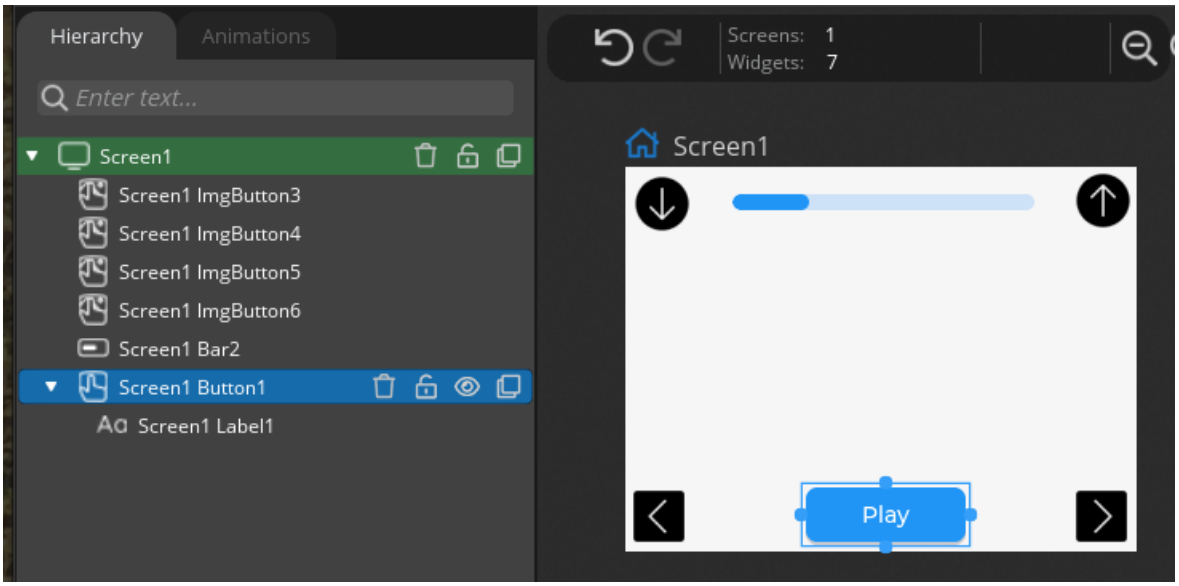

Figure: Button and Label

9. Select an **Image** widget from the **Widgets** panel and set its size to 128px (**Width**) & 128px (**Height**) on the **Inspector** panel. This image widget will display the animal images.

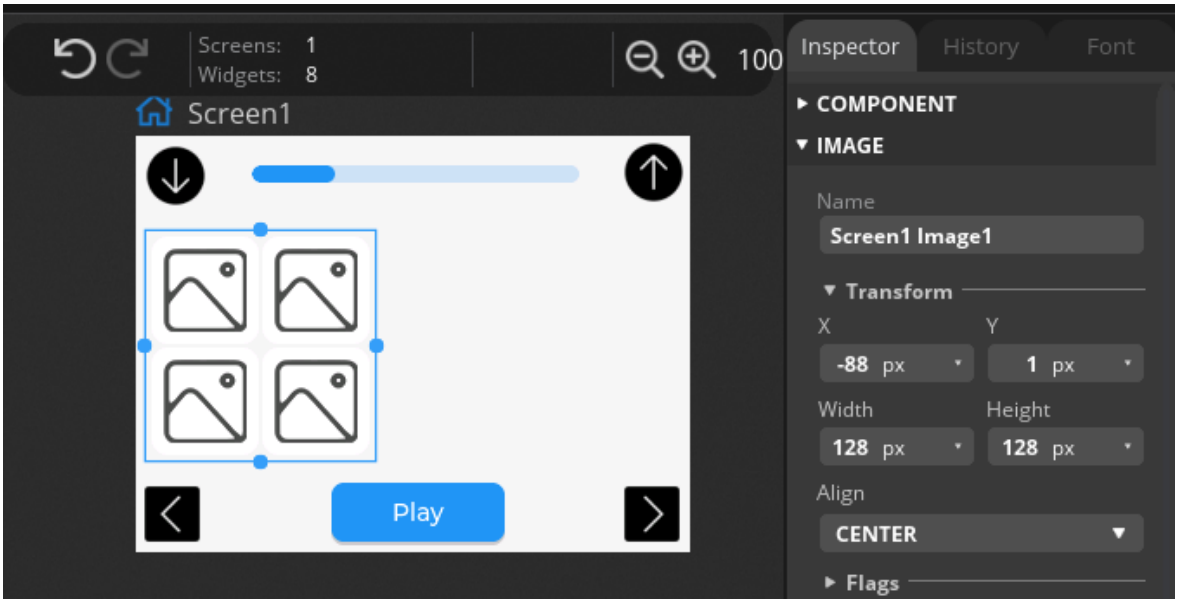

Figure: Image placeholder

10. As the last widget, place a label on the screen. The label will show the animal type. You can change its **Text Font** property from the **Inspector** panel as shown in the following figure.

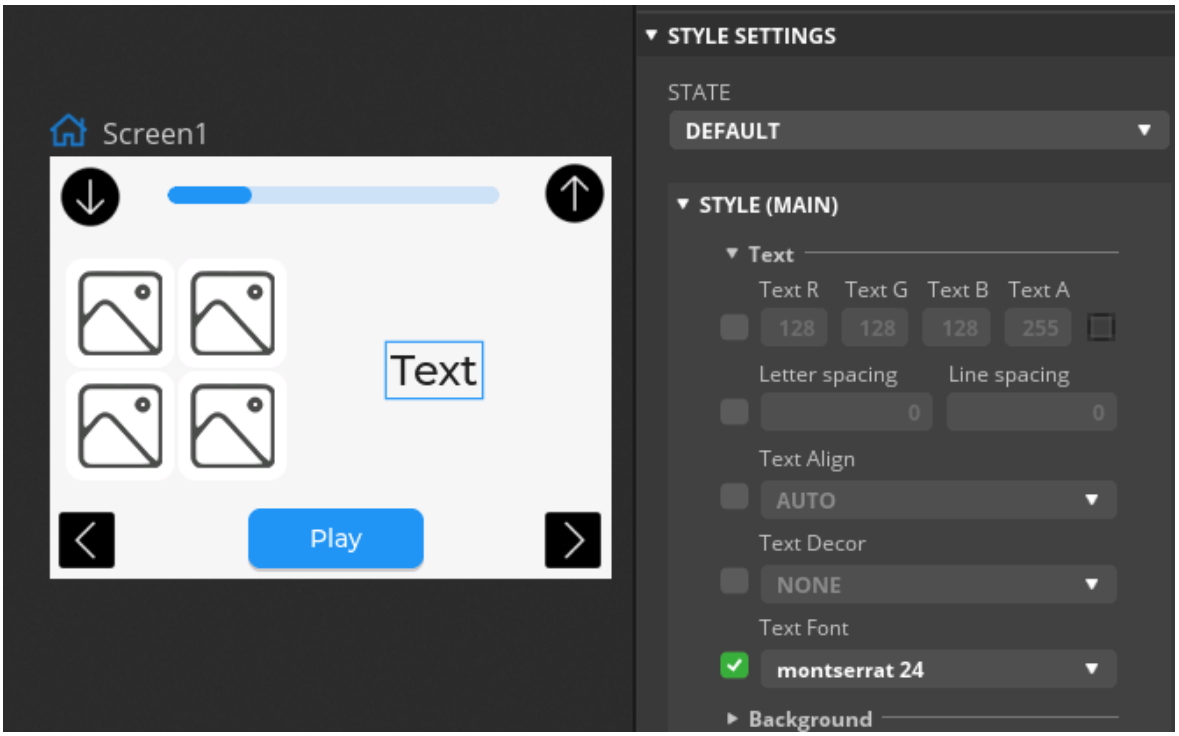

Figure: Setting the text font.

11. We can add function placeholders for the event handlers of different GUI events, such as, click, focus, or lost focus. When the play button or the volume bar gets the focus, the active one will be red in color, and the other one will be back to the default color. Let's configure the play button first. Go to the **Inspector** panel and open the **EVENTS** tab. Add an event and select **CALL FUNCTION** as action from the **Action** combo box. Set the name of the function as **play_focused**. We will implement the body of this function later while coding the application.

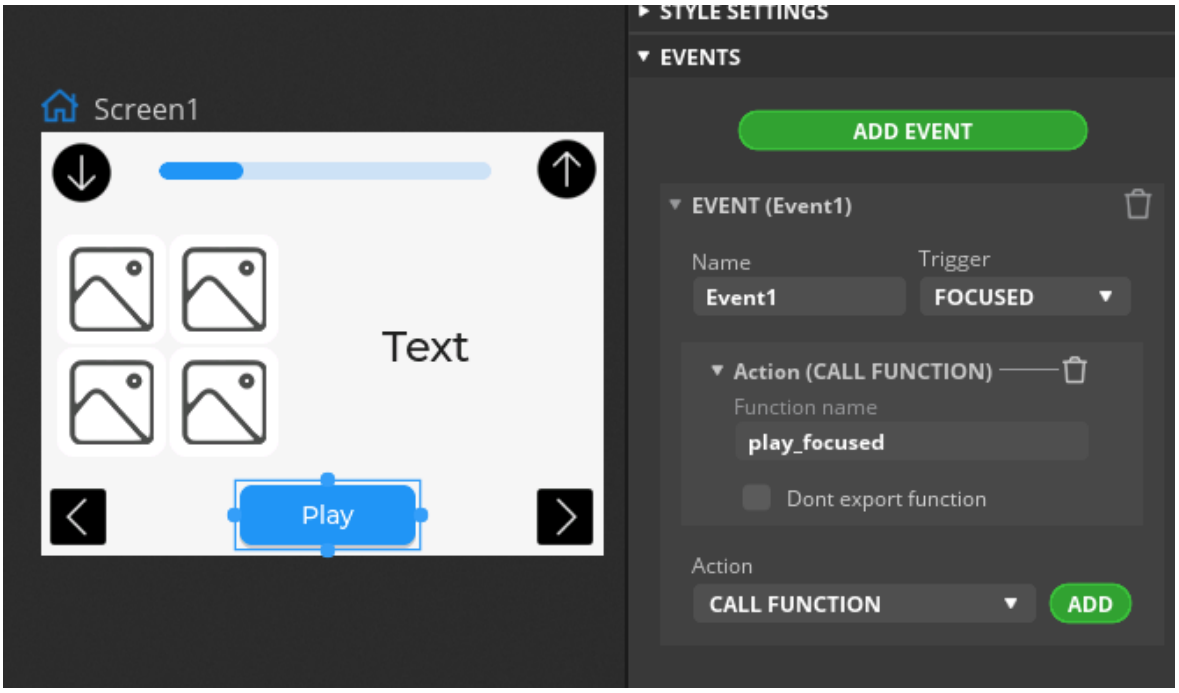

Figure: Adding CALL FUNCTION event

12. Add another event for the volume bar as shown in the following screenshot.

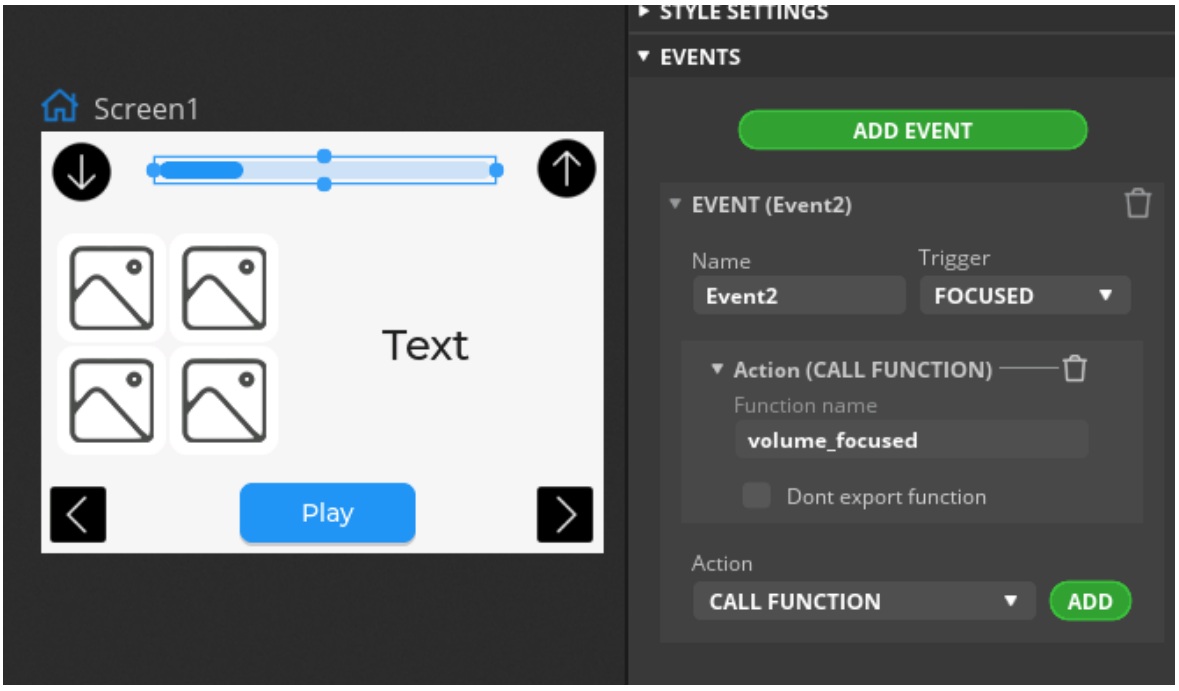

Figure: Focused event for the volume bar

13. The widgets have default names as assigned by the designer. Rename them to something meaningful.

14. Export the project by selecting from the main menu, **Export** / **Export UI Files**. You can move the generated code files next to the other source code files when we create the ESP-IDF project.

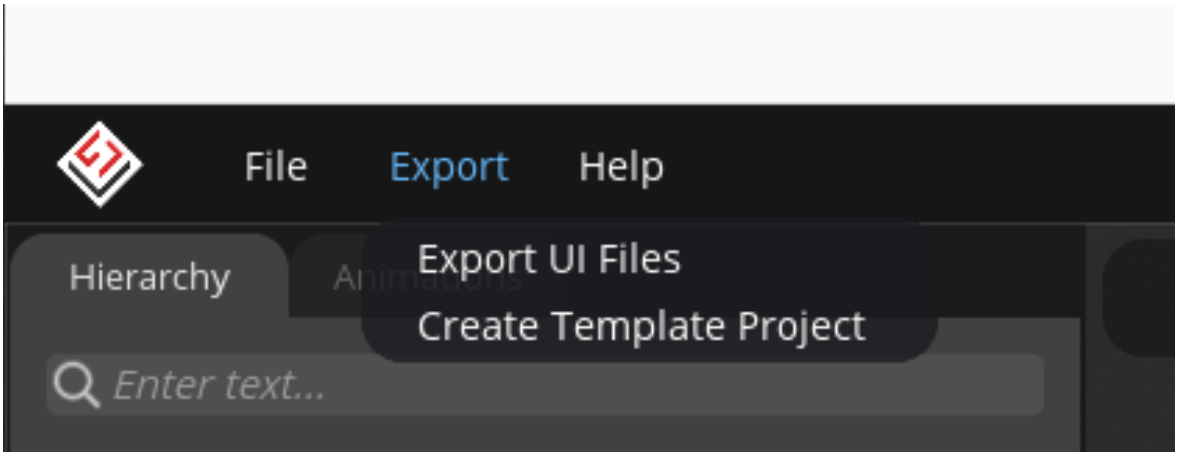

Figure: Export the project

15. See the generate code files in the directory you have chosen while exporting.

```
$ ls -1 ui*
ui.c
ui_events.c
ui.h
ui_helpers.c
ui_helpers.h
ui_img_1258062811.c
ui_img_1258080204.c
ui_img_1310788941.c
ui_img_1310804284.c
ui_img_1552218578.c
ui_img_1552235971.c
ui_img_603081905.c
ui_img_603097248.c
```

The GUI is ready and we can use these files in our project to make LVGL and the underlying hardware drivers render the GUI on the LCD display of the devkit.

> The SquareLine project files are provided in the repository at this link: [link]. You can open the project with the designer and see the final GUI design there. You can also just copy these generated files from the project repository here: [link]

Having the UI files generated, we can continue with the implementation of the classes and integrate them in an ESP-IDF project as we will discuss next.

Application

Let’s prepare the project as in the following steps:

1. Create an ESP-IDF project

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

1. Rename the application source code file to **main/audio_player.cpp**.
2. Copy the **sdkconfig** and **partitions.csv** files from the book repository into the project root. **partitions.csv** defines a partition to store the animal multimedia files on the flash:

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x10000, 0x6000,
phy_init, data, phy,     ,        0x1000,
```

```
factory,  app,  factory, ,        1M,
storage,  data, spiffs,  ,        3M,
```

1. Copy the generated UI files into the **main/** directory.
2. Copy the **spiffs** directory from the project repository into the project root. You can find this directory at this link: [link]. It contains the **mp3** and **png** files for animals. It has also a JSON file, **info.json**, that describes the animals and associated files as metadata. Its content is:

```
[
    {
        "animal": "Dog",
        "audio": "dog.mp3",
        "image": "dog.png"
    },
    {
        "animal": "Donkey",
        "audio": "donkey.mp3",
        "image": "donkey.png"
    },
    {
        "animal": "Goose",
        "audio": "goose.mp3",
        "image": "goose.png"
    },
    {
        "animal": "Sheep",
        "audio": "sheep.mp3",
        "image": "sheep.png"
    }
]
```

1. We will use the *Nlohmann-JSON* library to parse this file. Copy the library's single header file into the **main/** directory (**main/json.hpp**).
2. We need the ESP32-S3 BoxLite BSP in order to enable its buttons. Set the `EXTRA_COMPONENT_DIRS` parameter in the root **CMakeLists.txt** to include the BSP in the project.

```
cmake_minimum_required(VERSION 3.5)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
set(EXTRA_COMPONENT_DIRS ../../components)
add_compile_options(-fdiagnostics-color=always -Wno-write-strings -Wno-unused-variable)
project(audio_player)
```

1. Update the **main/CMakeLists.txt** file to include all the source code files in the project and create the `storage` partition from the `../spiffs` directory. The updated content of the **main/CMakeLists.txt** file is:

```
idf_component_register(SRCS
    audio_player.cpp
    ui.c
    ui_events.c
    ui_helpers.c
    ui_img_1258062811.c
    ui_img_1258080204.c
    ui_img_1310788941.c
    ui_img_1310804284.c
    ui_img_1552218578.c
    ui_img_1552235971.c
    ui_img_603081905.c
    ui_img_603097248.c
    INCLUDE_DIRS ".")
spiffs_create_partition_image(storage ../spiffs FLASH_IN_PROJECT)
```

The project is ready to continue with coding now. Let's develop the button handler in **main/AppButton.hpp**.

```
#pragma once
#include "freertos/FreeRTOS.h"
#include "freertos/queue.h"
#include "bsp_board.h"
#include "bsp_btn.h"
```

```
#include "esp_log.h"
namespace app
{
    enum class eBtnEvent
    {
        L_PRESSED,
        L_RELEASED,
        M_CLICK,
        M_DCLICK,
        R_PRESSED,
        R_RELEASED
    };
}
```

After including the necessary header files, we define an `enum` class, `eBtnEvent`, for the button events that we will generate when the user presses a button. Then, we define the templated function prototype for the press handler callbacks as the following:

```
namespace
{
    template <app::eBtnEvent>
    void button_event_handler(void *param);
}
```

The template parameter is `app::eBtnEvent` so that we don't have to repeat the same code for each type of the button events. Next comes the `AppButton` class:

```
namespace app
{
    class AppButton
    {
    private:
        QueueHandle_t m_event_queue = NULL;
```

In the private section of the `AppButton` class, we define a FreeRTOS queue that we can send button events. The `AppUi` class will later listen to this queue and process incoming events. Then we continue with the `init` function in the public section of the class:

```
    public:
        void init(void)
        {
            m_event_queue = xQueueCreate(10, sizeof(app::eBtnEvent));
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_DOWN, button_event_handler<
            bsp_btn_register_callback(BOARD_BTN_ID_PREV, BUTTON_PRESS_UP, button_event_handler<ap
            bsp_btn_register_callback(BOARD_BTN_ID_NEXT, BUTTON_PRESS_DOWN, button_event_handler<
            bsp_btn_register_callback(BOARD_BTN_ID_NEXT, BUTTON_PRESS_UP, button_event_handler<ap
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_SINGLE_CLICK, button_event_hand
            bsp_btn_register_callback(BOARD_BTN_ID_ENTER, BUTTON_PRESS_REPEAT, button_event_hand
        }
```

The `init` function creates an event queue for ten button events and then registers the callbacks for each of the event types. After the `init` function, we define another function to retrieve the class instance from a pointer of the button structure as the following:

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

`getObject` is a static function that takes a button device pointer as a parameter and returns the class instance reference. It will help when reaching the instance from the button callback. We finish the class implementation with a member function that returns the event queue handle next:

```
        QueueHandle_t getEventQueue(void) const { return m_event_queue; }
    }; // class end
} // namespace end
```

The `getEventQueue` function simply returns the member queue handle in order to allow access the queue from outside. The only remaining coding in this file is the body of the `button_event_handler` function as comes next:

```
namespace
{
    template <app::eBtnEvent E>
    void button_event_handler(void *btn_ptr)
    {
        app::AppButton &app_btn = app::AppButton::getObject(btn_ptr);
        app::eBtnEvent evt{E};
        xQueueSend(app_btn.getEventQueue(), (void *)(&evt), 0);
    }
}
```

In the `button_event_handler` function, we create a local variable, `evt` , for the button event and pass it to the event queue of the `AppButton` instance by calling the `xQueueSend` FreeRTOS function. This completes the implementation of the button handler.

We continue with the implementation of the audio features by creating another source code file, **main/AppAudio.hpp**, and edit it as the following:

```
#pragma once
#include <cstdio>
#include <cinttypes>
#include <string>
#include "freertos/FreeRTOS.h"
#include "freertos/queue.h"
#include "esp_err.h"
#include "bsp_codec.h"
#include "bsp_board.h"
#include "bsp_storage.h"
#include "audio_player.h"
namespace
{
    void audio_player_callback(audio_player_cb_ctx_t *obj);
}
```

As usual, we first include the header files needed in the implementation. Then, we declare the callback prototype for audio events. It will be called when an audio event occurs. We don’t have to use all audio events and can customize them for our purposes. The following `enum` class shows the events that we are going to use in the application:

```
namespace app
{
    enum class eAudioEvent
    {
        PLAYING,
        STOPPED
    };
}
```

We are only interested in the states when the audio player starts playing and stops. We define them in the `eAudioEvent enum` class. Next we begin with the `AppAudio` class implementation:

```
namespace app
{
    class AppAudio
    {
    private:
        FILE *m_fp;
        uint8_t m_vol;
        audio_player_state_t m_player_state;
        QueueHandle_t m_event_queue;
        bool m_muted;
```

The private section of the class contains several member variables. We have a file pointer for the audio files to be passed to the audio player and we keep the volume level. We also keep the internal state of the audio player to be able to toggle between the playing and stopped states. The event queue is for signaling the other parts of the application (in fact, it is only the `AppUi` class) about the audio player state changes. The last variable shows whether the audio player is muted or not. In the public section, we start with the constructor:

```
public:
    AppAudio() : m_fp(nullptr),
                 m_vol{50},
                 m_player_state(AUDIO_PLAYER_STATE_IDLE),
                 m_event_queue(nullptr),
                 m_muted(false) {}
```

The constructor simply sets the initial values of the member variables and nothing more. The next function actually initializes the hardware:

```
    void init(audio_player_mute_fn fn)
    {
        audio_player_config_t config = {.port = I2S_NUM_0,
                                        .mute_fn = fn,
                                        .priority = 1};
        audio_player_new(config);
        audio_player_callback_register(audio_player_callback, this);
        bsp_codec_set_voice_volume(50);
        m_event_queue = xQueueCreate(10, sizeof(app::eBtnEvent));
    }
    QueueHandle_t getEventQueue(void) const { return m_event_queue; }
```

The `init` function takes a mute function as a parameter and we use this callback while constructing the configuration for the audio player. We also register the `audio_player_callback` function that we declared at the beginning in order to track the changes in the audio player state by calling the `audio_player_callback_register` function. The audio event queue is created at the end of the `init` function. The `getEventQueue` function returns the audio event queue for the outside world. Next we implement the `mute` function of the class:

```
    uint8_t mute(bool m, bool toggle = false)
    {
        uint8_t val = 0;
        m_muted = toggle ? !m_muted : m;
        if (m_muted)
        {
            bsp_codec_set_mute(true);
        }
        else
        {
            bsp_codec_set_mute(false);
            bsp_codec_set_voice_volume(m_vol);
            val = m_vol;
        }
        return val;
    }
```

The `mute` member function controls the audio hardware for muting/unmuting. Please note that it is not the callback to be passed to the `init` function, their signatures are different. We will wrap the `mute` member function within another function to make it compatible with the `init` function's parameter. In the body, we set the `m_muted` variable to `true` or `false` or we toggle it, depending on the value of the `toggle` parameter. After checking the value of the `m_muted` variable we call the `bsp_codec_set_mute` function to set the mute state of the audio player. The returned value from the `mute` function is the volume level. Let's implement the play/stop functionality in the next function:

```
    void togglePlay(const std::string &filename)
    {
        switch (m_player_state)
        {
        case AUDIO_PLAYER_STATE_PLAYING:
```

```
                audio_player_pause();
                break;
            case AUDIO_PLAYER_STATE_PAUSE:
                audio_player_resume();
                break;
            default:
                m_fp = fopen(filename.c_str(), "rb");
                audio_player_play(m_fp);
                break;
            }
        }
```

In the `togglePlay` function, we check the internal state of the audio player and take action accordingly. If it is in the playing state, we pause it, else if it is already paused, we resume it. The default action is to open the given audio file and play it by calling the `audio_player_play` function with the file pointer. Next comes the `volumeUp` and `volumeDown` member functions:

```
        uint8_t volumeUp(void)
        {
            if (m_vol < 100)
            {
                m_vol += 10;
                bsp_codec_set_voice_volume(m_vol);
            }
            return m_vol;
        }
        uint8_t volumeDown(void)
        {
            if (m_vol > 0)
            {
                m_vol -= 10;
                bsp_codec_set_voice_volume(m_vol);
            }
            return m_vol;
        }
```

These two functions are very similar to each other. The `volumeUp` function increases the volume by 10 and the `volumeDown` function does the opposite. Both return the final volume level. The last function of the class is `setState` as follows next:

```
        void setState(audio_player_state_t state)
        {
            m_player_state = state;
        }
    }; // class end
} // namespace end
```

The `setState` function only updates the private `m_player_state` member variable with the given value. This function will be used from the audio event callback, `audio_player_callback` , that we have registered in the `init` function. We will implement its body next:

```
namespace
{
    void audio_player_callback(audio_player_cb_ctx_t *param)
    {
        app::AppAudio &app_audio = *(reinterpret_cast<app::AppAudio *>(param->user_ctx));
        audio_player_state_t state = audio_player_get_state();
        app_audio.setState(state);
        app::eAudioEvent evt = state == AUDIO_PLAYER_STATE_PLAYING ? app::eAudioEvent::PLAYING :
        xQueueSend(app_audio.getEventQueue(), (void *)(&evt), 0);
    }
}
```

The `audio_player_callback` function is called when something changes in the state of the audio player. We update the `AppAudio` instance with this change and also send the customized event to the audio event queue of the `AppAudio` instance. The implementation of the audio functionality is done. Next, we will develop the animal navigation in the `main/AppNav.hpp` file:

```
#pragma once
#include <fstream>
#include <string>
#include <vector>
#include "json.hpp"
namespace app
{
    struct Animal_t
    {
        std::string animal;
        std::string audio;
        std::string image;
    };
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(Animal_t, animal, audio, image);
```

Here, we need the *Nlohmann-JSON* library to parse the **/spiffs/info.json** file, which contains metadata about the animal records. Each record has the animal name, the audio file name, and the image file name. We define the `Animal_t` structure to hold this information and also define the JSON conversion functions with the help of the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro that comes with the *Nlohmann-JSON* library. Next, we continue with the `AppNav` class:

```
    class AppNav
    {
    private:
        int m_pos;
        std::vector<Animal_t> m_animals;
```

The private section of the `AppNav` class has only two member variables, the `Animal_t` vector and the record position on this vector. Then we move on to the public section:

```
    public:
        void init()
        {
            std::ifstream fs{"/spiffs/info.json"};
            m_animals = nlohmann::json::parse(fs);
            m_pos = 0;
        }
```

The `init` function parses the metadata information and initializes the member variables. The `parse` function returns a `nlohmann::json` object but this object is implicitly converted to a vector of the `Animal_t` type since we had the `from_json` function by calling the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` macro. The next member function returns the current animal record:

```
        Animal_t &getCurrent(void)
        {
            return m_animals[m_pos];
        }
```

We return the reference of the current animal record from the `getCurrent` function. The remaining two functions are for navigation as comes next:

```
        Animal_t &next(void)
        {
            m_pos = (m_pos + 1) % m_animals.size();
            return m_animals[m_pos];
        }
        Animal_t &prev(void)
        {
            m_pos = (m_pos - 1) % m_animals.size();
            return m_animals[m_pos];
        }
    }; // class end
} // namespace end
```

The `next` member function (yes, its name is `next`) advances the position on the animal vector and returns the reference of the record at that position. Similarly, the `prev` function returns a reference for the previous animal

record in the vector.

We have completed the implementation of the navigation features and now are ready to integrate all of them in the GUI handling. We create the **main/AppUi.hpp** file for it and edit it next:

```
#pragma once
#include <mutex>
#include <string>
#include "bsp_lcd.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lvgl/lvgl.h"
#include "lv_port/lv_port.h"
#include "ui.h"
#include "AppButton.hpp"
#include "AppNav.hpp"
#include "AppAudio.hpp"
namespace
{
    app::AppNav m_nav;
}
```

We start with including the `hpp` header files that we have developed for the button, audio, and navigation. In the anonymous namespace, we define an instance of the `AppNav` class. Next comes the implementation of the `AppUi` class:

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

The `m_ui_access` variable is a static `mutex` that controls the access to the underlying LVGL objects and functions, and the `lvglTask` function is a FreeRTOS task that renders the LVGL objects in a loop with a period of `10` ticks. Since it is a FreeRTOS task, the function won't return. Next, we continue with the definition of the member variables:

```
        bool m_playlist_active;
        app::AppAudio *m_app_audio;
        app::AppButton *m_app_button;
```

If you remember, users can switch between the playlist and the volume control by double-clicking on the middle button of the devkit. The `m_playlist_active` variable denotes whether the playlist is the active feature of the GUI. The `m_app_audio` and `m_app_button` are pointers to the `AppAudio` and `AppButton` class instances respectively. Several function prototypes come next:

```
        static void buttonEventTask(void *param);
        static void audioEventTask(void *param);
        static std::string makeImagePath(const std::string &filename);
        static std::string makeAudioPath(const std::string &filename);
        void update(const Animal_t &an, bool play = true);
        void toggleControl(void);
```

We will implement the function bodies later, but as a quick summary:

- `buttonEventTask` is the FreeRTOS task that monitors the button event queue.
- `audioEventTask` is the FreeRTOS task that monitors the audio event queue.
- `makeImagePath` is the utility function that returns the full path of an image on the filesystem.
- `makeAudioPath` is the utility function that returns the full path of an audio file on the filesystem.
- `update` is a member function to re-render the GUI when the active position changes on the playlist.
- `toggleControl` is a member function to toggle the control between the playlist and volume.

Then we develop the functions in the public section:

```
public:
    void init(app::AppButton *button, app::AppAudio *audio)
    {
        m_app_button = button;
        m_app_audio = audio;
        lv_port_init();
        ui_init();
        m_nav.init();
```

The `init` function takes two pointer parameters to the button and audio instances. We keep them locally in the class by assigning them to their respective member variables. Then, we initialize the LVGL library and the GUI that we designed at the very beginning of the project. We also initialize the `m_nav` object by calling its `init` function. Let's continue with more initialization and the configuration of some UI elements:

```
        m_playlist_active = true;
        lv_event_send(ui_btnPlay, LV_EVENT_FOCUSED, nullptr);
        lv_bar_set_range(ui_barVolume, 0, 100);
        lv_bar_set_value(ui_barVolume, 50, lv_anim_enable_t::LV_ANIM_OFF);
```

The default active feature is the playlist and we set the focus on the UI play button, `ui_btnPlay`, by calling the `lv_event_send` function of the LVGL library. The next two lines above configure the volume bar with a range of 0 to 100 and set its initial value as 50. The `init` function is not finished yet:

```
        update(m_nav.getCurrent(), false);
        bsp_lcd_set_backlight(true);
```

The `update` member function updates the GUI with the first animal in the playlist. After that, we turn on the backlight of the LCD display. Before exiting the `init` function, we start the FreeRTOS tasks as the following:

```
        xTaskCreatePinnedToCore(lvglTask, "lvgl", 6 * 1024, nullptr, 3, nullptr, 0);
        xTaskCreate(buttonEventTask, "btn_evt", 3 * 1024, this, 3, nullptr);
        xTaskCreate(audioEventTask, "audio_evt", 3 * 1024, this, 3, nullptr);
    }
```

We have three FreeRTOS tasks: for LVGL, for button events, and for audio events. The task functions are the ones that we have declared in the private section of the `AppUi` class earlier. Then we define the functions for returning the audio and button event queues:

```
    QueueHandle_t getButtonEventQueue(void) const { return m_app_button->getEventQueue(); }
    QueueHandle_t getAudioEventQueue(void) const { return m_app_audio->getEventQueue(); }
```

The `getButtonEventQueue` and `getAudioEventQueue` functions are simply wrappers for the button and audio objects' `getEventQueue` functions. They will be needed when monitoring the queues in the FreeRTOS tasks. Next comes the heart of the `AppUi` class, the `handleButtonEvent` function:

```
    void handleButtonEvent(app::eBtnEvent &btn_evt)
    {
        std::lock_guard<std::mutex> lock(m_ui_access);
        switch (btn_evt)
        {
        case app::eBtnEvent::M_DCLICK:
            toggleControl();
            break;
```

In the `handleButtonEvent` function, we first lock the `m_ui_access` mutex to prevent any other code from modifying the LVGL objects since we will work on them here. In a `switch` statement we check the incoming button event. If it is a double-click event on the middle button of the devkit, we toggle the control between the playlist and the volume. Next, we will add the case for the single click on the middle button:

```
case app::eBtnEvent::M_CLICK:
    if (m_playlist_active)
    {
        lv_event_send(ui_btnPlay, LV_EVENT_CLICKED, nullptr);
        m_app_audio->togglePlay(makeAudioPath(m_nav.getCurrent().audio));
    }
    else
    {
        lv_bar_set_value(ui_barVolume, m_app_audio->mute(false, true), lv_anim_enable
    }
    break;
```

In case of a single click on the middle button, the behavior of the application changes according to the active feature. If the playlist is active, then we toggle the play status by calling the `togglePlay` function of the `m_app_audio` object. If the volume control is active, this time we toggle the mute status of the volume. The reactions to the button events are reflected on the respective UI elements. The active feature check also applies to all other button click events. Let's handle the left-button pressed-down event next:

```
case app::eBtnEvent::L_PRESSED:
    if (m_playlist_active)
    {
        lv_event_send(ui_btnPrev, LV_EVENT_PRESSED, nullptr);
    }
    else
    {
        lv_event_send(ui_btnVolumeDown, LV_EVENT_PRESSED, nullptr);
    }
    break;
```

In this case, we simply update the UI buttons with the pressed-down visual events. The actual actions will be taken when the button is released as comes next:

```
case app::eBtnEvent::L_RELEASED:
    if (m_playlist_active)
    {
        lv_event_send(ui_btnPrev, LV_EVENT_RELEASED, nullptr);
        update(m_nav.prev());
    }
    else
    {
        lv_event_send(ui_btnVolumeDown, LV_EVENT_RELEASED, nullptr);
        lv_bar_set_value(ui_barVolume, m_app_audio->volumeDown(), lv_anim_enable_t::L
    }
    break;
```

When the left button is released, in addition to the UI element updates, we also call the action functions. If the playlist is the active feature, we navigate to the previous animal on the list, else (the volume control is active) we decrease the volume. The remaining button events are for the right button as the following:

```
case app::eBtnEvent::R_PRESSED:
    if (m_playlist_active)
    {
        lv_event_send(ui_btnNext, LV_EVENT_PRESSED, nullptr);
    }
    else
    {
        lv_event_send(ui_btnVolumeUp, LV_EVENT_PRESSED, nullptr);
    }
    break;
case app::eBtnEvent::R_RELEASED:
    if (m_playlist_active)
    {
```

```
                lv_event_send(ui_btnNext, LV_EVENT_RELEASED, nullptr);
                update(m_nav.next());
            }
            else
            {
                lv_event_send(ui_btnVolumeUp, LV_EVENT_RELEASED, nullptr);
                lv_bar_set_value(ui_barVolume, m_app_audio->volumeUp(), lv_anim_enable_t::LV_
            }
            break;
        default:
            break;
        } // switch end
    } // function end
```

The logic is similar to the left button event handling. Again when released, the right button causes it to pass to the next animal if the playlist is active, or increases the volume level if the volume control is active. The relevant UI elements are all updated during these events. This finishes the `handleButtonEvent` function. Next, we implement the `handleAudioEvent` as the following:

```
    void handleAudioEvent(app::eAudioEvent &evt)
    {
        if (evt == app::eAudioEvent::PLAYING)
        {
            lv_label_set_text(ui_txtPlayButton, "Pause");
        }
        else
        {
            lv_label_set_text(ui_txtPlayButton, "Play");
        }
    } // function end
}; // class end
```

The `handleAudioEvent` function takes the audio event as a parameter. If the audio is playing, then the user can pause it and we indicate this state by changing the label of the `ui_txtPlayButton` UI element to `Pause`. Otherwise, the label becomes `Play`. Let's come to the development of the private functions that we skipped at the beginning of the class implementation. The first one is the `buttonEventTask` function:

```
void AppUi::buttonEventTask(void *param)
{
    AppUi &ui = *(reinterpret_cast<AppUi *>(param));
    app::eBtnEvent evt;
    while (true)
    {
        xQueueReceive(ui.getButtonEventQueue(), &evt, portMAX_DELAY);
        ui.handleButtonEvent(evt);
    }
}
```

In the `buttonEventTask` function, we wait on the button event queue. When an event comes, we pass it to the `AppUi` object to be processed. The next function does the same thing for the audio events:

```
void AppUi::audioEventTask(void *param)
{
    AppUi &ui = *(reinterpret_cast<AppUi *>(param));
    app::eAudioEvent evt;
    while (true)
    {
        xQueueReceive(ui.getAudioEventQueue(), &evt, portMAX_DELAY);
        ui.handleAudioEvent(evt);
    }
}
```

The `audioEventTask` function processes the audio events in a loop by waiting on the audio event queue. When an event appears on the queue, it is passed to the `AppUi` object. The following function creates image path for the LVGL image widget:

```
std::string AppUi::makeImagePath(const std::string &filename)
{
    return std::string("S:/spiffs/") + filename;
}
```

An interesting fact about LVGL is that it expects a drive letter as the prefix of a path. We can set it to `S` in **sdkconfig** by updating the value of `CONFIG_LV_FS_POSIX_LETTER` to `83` (the decimal number **83** corresponds to the character **S** on the ASCII table). You can also run **menuconfig** and go to the LVGL section of the configuration (navigate to **(Top) → LVGL configuration → 3rd Party Libraries**) to update this value. In the next function, we create paths for the audio files:

```
std::string AppUi::makeAudioPath(const std::string &filename)
{
    return std::string("/spiffs/") + filename;
}
```

The `makeAudioPath` function just returns the full path on the filesystem for a given file name. Let's continue with the `update` function of the `AppUi` class:

```
void AppUi::update(const Animal_t &an, bool play)
{
    lv_label_set_text(ui_txtAnimal, an.animal.c_str());
    lv_img_set_src(ui_imgAnimal, makeImagePath(an.image).c_str());
}
```

We call the `update` function when the user changes the current animal. The function updates the `ui_txtAnimal` and `ui_imgAnimal` UI elements with the current record, hence the name. The final function of the class is `toggleControl` as comes next:

```
void AppUi::toggleControl(void)
{
    m_playlist_active = !m_playlist_active;
    if (m_playlist_active)
    {
        lv_event_send(ui_btnPlay, LV_EVENT_FOCUSED, nullptr);
    }
    else
    {
        lv_event_send(ui_barVolume, LV_EVENT_FOCUSED, nullptr);
    }
} // function end
std::mutex AppUi::m_ui_access;
} // namespace end
```

In the `toggleControl` function, we set focus on either the `ui_btnPlay` UI element or the `ui_barVolume` UI element by sending the `LV_EVENT_FOCUSED` event after toggling the `m_playlist_active` member variable. This finalizes the class implementation. However, there is one last requirement left that we need to discuss before moving on to the main application in the **main/audio_player.cpp** file. Setting focus on the play button or the volume bar should change their color to red. For this, we edit the **main/ui_events.c** file which is one of the outputs of the *SquareLine* designer:

```
#define NORMAL_COLOR 0x4C93F4
#define FOCUSED_COLOR 0xFF0000
void play_focused(lv_event_t * e)
{
    lv_obj_set_style_bg_color(ui_barVolume, lv_color_hex(NORMAL_COLOR), LV_PART_INDICATOR | LV_S
    lv_obj_set_style_bg_color(ui_btnPlay, lv_color_hex(FOCUSED_COLOR), LV_PART_MAIN | LV_STATE_DE
}
void volume_focused(lv_event_t * e)
{
    lv_obj_set_style_bg_color(ui_btnPlay, lv_color_hex(NORMAL_COLOR), LV_PART_MAIN | LV_STATE_DEF
    lv_obj_set_style_bg_color(ui_barVolume, lv_color_hex(FOCUSED_COLOR), LV_PART_INDICATOR | LV_S
}
```

If you remember, we named these two event handlers while designing the GUI. They are the functions to be run by the *LVGL* engine when the `ui_btnPlay` or `ui_barVolume` UI elements get focus. The `play_focused` function

sets the color of `ui_btnPlay` as red and the color of `ui_barVolume` as blue, and the `volume_focused` function does the other way around. Now we're done with the class implementations and can edit the **main/audio_player.cpp** file to integrate all that stuff as the final application:

```
#include "bsp_board.h"
#include "bsp_storage.h"
#include "AppUi.hpp"
#include "AppButton.hpp"
#include "AppAudio.hpp"
namespace
{
    app::AppButton m_app_btn;
    app::AppAudio m_app_audio;
    app::AppUi m_app_ui;
    esp_err_t audio_mute_function(AUDIO_PLAYER_MUTE_SETTING setting)
    {
        m_app_audio.mute(setting == AUDIO_PLAYER_MUTE);
        return ESP_OK;
    }
}
```

After including the class headers, we declare the `m_app_btn`, `m_app_audio`, and `m_app_ui` in the anonymous namespace. We also define the `audio_mute_function` function to be passed to the audio initialization. Then we develop the application entry point, the `app_main` function as comes next:

```
extern "C" void app_main(void)
{
    bsp_board_init();
    bsp_board_power_ctrl(POWER_MODULE_AUDIO, true);
    bsp_spiffs_init("storage", "/spiffs", 10);
```

In the `app_main` function, we first initialize the devkit by calling two BSP functions. Then we mount the `storage` partition by calling the `bsp_spiffs_init` function in order to access the files in it. Next comes the object initializations:

```
    m_app_btn.init();
    m_app_audio.init(audio_mute_function);
    m_app_ui.init(&m_app_btn, &m_app_audio);
}
```

We initialize the button and the audio object, and then call the `init` function of the UI object with the pointers of the other two. When the application starts on the devkit, these objects take control and run.

The project is ready for the tests on the devkit now, congratulations for coming so far! Let's begin the fun part. We flash the application as the following:

```
$ idf.py erase-flash clean flash monitor
Executing action: erase-flash
Serial port /dev/ttyACM0
Connecting....
Detecting chip type... ESP32-S3
<more logs>
I (1026) lv_port: Try allocate two 320 * 20 display buffer, size:25600 Byte
I (1028) lv_port: Add KEYPAD input device to LVGL
```

After a successful flash, you should be able to see the GUI on the LCD display:

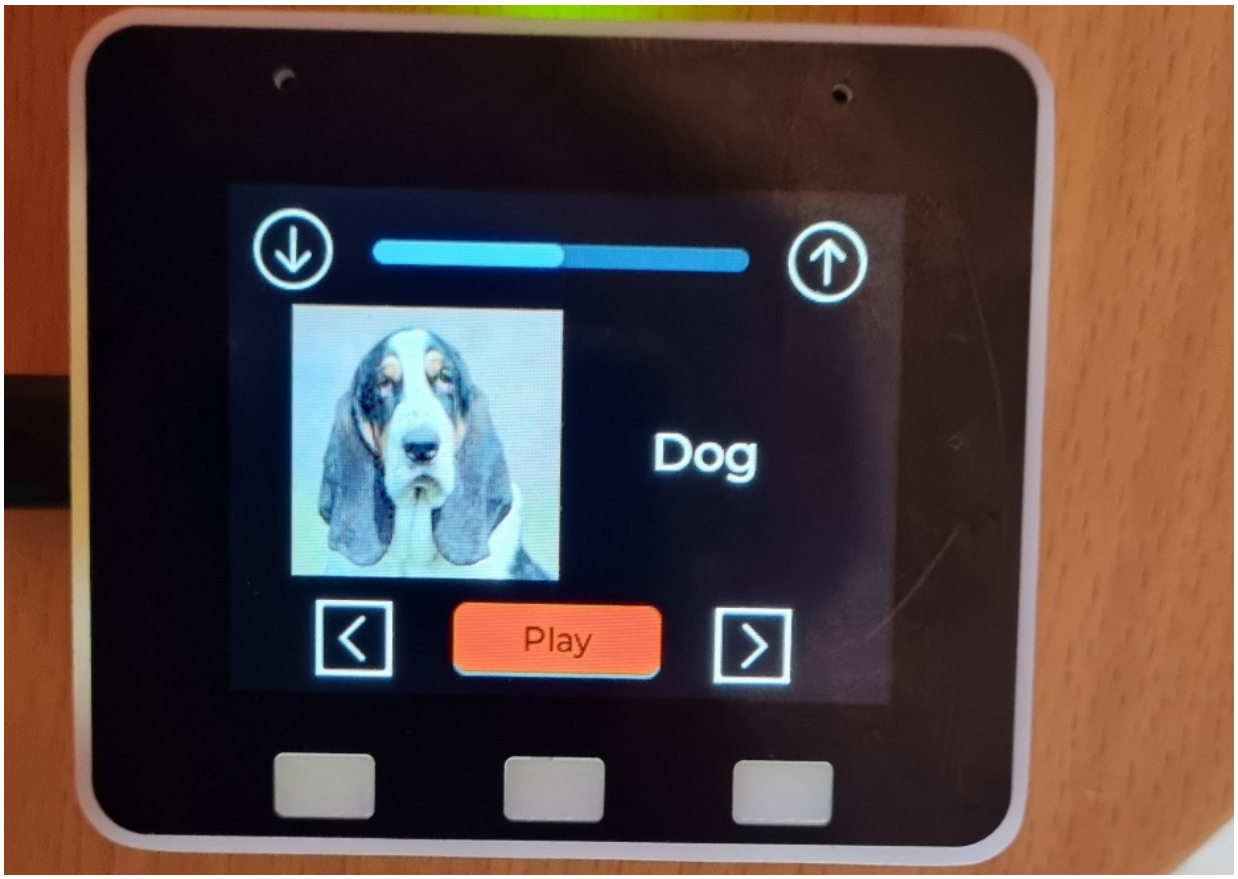

Figure: Application GUI

Here are some test cases:

- Press the middle button to play the dog sound. See it pauses when you press once more while it is playing.
- See the button name changes to **Play** automatically when the audio finishes.
- Press the left and right buttons to navigate over the animal list. See the GUI updates accordingly.
- Double-click on the middle button to change the focus to the volume control. See the color of the volume bar changes to red.
- Increase and decrease the volume.
- Double-click on the middle-button to return to the playlist. When you play the animal sound, check whether the volume is updated.

The project has more rooms for improvement. You can easily apply many other techniques from what we have learned in the first four chapters.

New features

You can improve the project by adding more features as practice. Some ideas are:

- The application employs SPIFSS as the filesystem. Replace it with LittleFS.
- The images load rather slowly with a visible effect on the GUI while rendering. One way to make it faster is to load all of the images to PSRAM when the application starts. Instead of reading from flash, you can make LVGL use them from the PSRAM.
- Create a new LVGL theme for a different level of ambient light. Use LDR to measure the light level and change the theme automatically according to the light level

(https://docs.lvgl.io/latest/en/html/overview/style.html#themes)
- Save/restore the latest status of the player on the **nvs** partition (the latest volume level and the index of the last played item)
- Add two touch sensors to navigate on the playlist with the same functionality of the left and right buttons when the playlist is active.
- Add a beep sound when the user changes the volume level.

Summary

Learning raises on practice and this project was a good opportunity for this. We developed a project from the ground up, starting from the feature list of the audio player, then discussing the solution architecture, and finally the development. The development also has different stages inside. UI/UX design is an important step for a successful product. With a given list of features, a talented UI/UX designer can optimize the GUI and emphasize the unique selling points much better. As developers, we put our efforts to implement the solution architecture by applying our knowledge and skills. Each project moves us forward to be a better developer.

This project marks the end of the first part. In the upcoming chapters, we will learn how to develop connected ESP32 products on cloud systems.