Chapter 4 Setting Up Development Environment

This chapter focuses on ESP-IDF, the official software development framework for ESP32-C3. We'll explain how to set up the environment on various operating systems, and introduce the project structure and build system of ESP-IDF, as well as the usage of related development tools. Then we'll present the compiling and running process of an example project, while offering a detailed explanation of the output log at each stage.

4.1 ESP-IDF Overview

ESP-IDF (Espressif IoT Development Framework) is a one-stop IoT development framework provided by Espressif Technology. It uses C/C++ as the main development language and supports cross-compilation under mainstream operating systems such as Linux, Mac, and Windows. The example programs included in this book are developed using ESP-IDF, which offers the following features:

- **SoC system-level drivers**. ESP-IDF includes drivers for ESP32, ESP32-S2, ESP32-C3, and other chips. These drivers encompass peripheral low level (LL) library, hardware abstraction layer (HAL) library, RTOS support and upper-layer driver software, etc.
- **Essential components**. ESP-IDF incorporates fundamental components required for IoT development. This includes multiple network protocol stacks such as HTTP and MQTT, a power management framework with dynamic frequency modulation, and features like Flash Encryption and Secure Boot, etc.
- **Development and production tools**. ESP-IDF provides commonly used tools for building, flash, and debugging during development and mass production (see Figure 4.1), such as the building system based on CMake, the cross-compilation tool chain based on GCC, and the JTAG debugging tool based on OpenOCD, etc.

It is worth noting that the ESP-IDF code primarily adheres to the the Apache 2.0 open-source license. Users can develop personal or commercial software without restrictions while complying with the terms of the open-source license. Additionally, users are granted permanent patent licenses free of charge, without the obligation to open-source any modifications made to the source code.

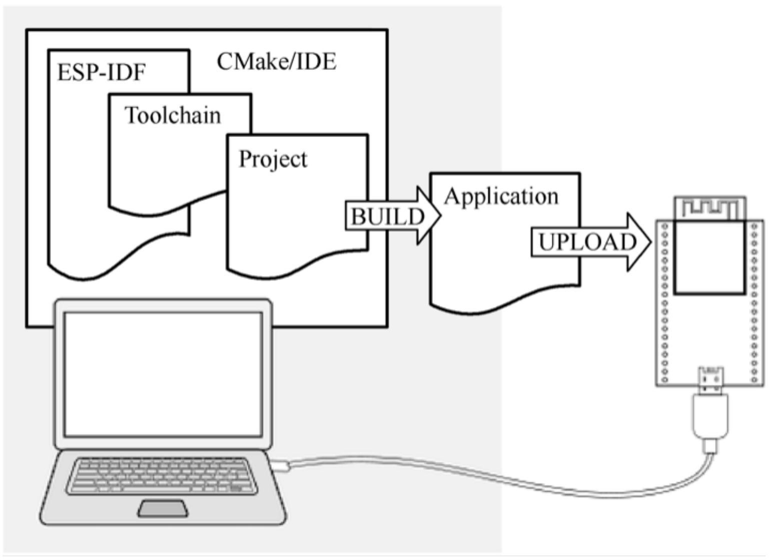


Figure 4.1. Building, flashing, and debugging tools for development and mass production

4.1.1 ESP-IDF Versions

The ESP-IDF code is hosted on GitHub as an open-source project. Currently, there are three major versions available: v3, v4, and v5. Each major version usually contains various sub-versions, such as v4.2, v4.3, and so on. Espressif Systems ensures a 30-month support for bug fixes and security patches for each released sub-version. Therefore, revisions of sub-versions are also released regularly, such as v4.3.1, v4.2.2, etc. Table 4.1 shows the support status of different ESP-IDF versions for Espressif chips, indicating whether they are in a preview stage (offering support for preview versions, which may lack certain features or documentation) or are officially supported.

Table 4.1. Support status of different ESP-IDF versions for Espressif chips

Series	v4.1	v4.2	v4.3	v4.4	v5.0
ESP32	supported	supported	supported	supported	supported
ESP32-S2		supported	supported	supported	supported
ESP32-C3			supported	supported	supported
ESP32-S3				supported	supported
ESP32-C2					supported
ESP32-H2				preview	preview

The iteration of major versions often involves adjustments to the framework structure and updates to the compilation system. For example, the major change from v3.* to v4.* was the gradual migration of the build system from Make to CMake. On the other hand, iteration of minor versions typically entails the addition of new features or support for new chips.

It is important to distinguish and understand the relationship between stable versions and GitHub branches. Versions labeled as v*.* or v*.*.* represent stable versions that have passed complete internal testing by Espressif. Once fixed, the code, tool chain, and release documents for the same version remain unchanged. However, GitHub branches (e.g., the `release/v4.3` branch) undergo frequent code commits, often on a daily basis. Therefore, two code snippets under the same branch may differ, necessitating developers to promptly update their code accordingly.

4.1.2 ESP-IDF Git Workflow

Espressif follows a specific Git workflow for ESP-IDF, outlined as follows:

- New changes are made on the `master` branch, which serves as the main development branch. The ESP-IDF version on the `master` branch always carries a `-dev` tag to indicate that it is currently under development, such as `v4.3-dev`. Changes on the `master` branch will first be reviewed and tested in Espressif's internal repository, and then pushed to GitHub after automated testing is complete.
- Once a new version has completed feature development on the `master` branch and met the criteria for entering beta testing, it transitions to a new branch, such as `release/v4.3`. In addition, this new branch is tagged as a pre-release version, like `v4.3-beta1`. Developers can refer to the GitHub platform to access the complete list of branches and tags for ESP-IDF. It's important to note that the beta version (pre-release version) may still have a significant number of known issues. As the beta version undergoes continuous testing, bug fixes are added to both this version and the `master` branch simultaneously. Meanwhile, the `master` branch may have already begun developing new features for the next version. When testing is nearly complete, a release candidate (`rc`) label is added to the branch, indicating that it is a potential candidate for the official release, such as `v4.3-rc1`. At this stage, the branch remains a pre-release version.
- If no major bugs are discovered or reported, the pre-release version eventually receives a major version label (e.g., v5.0) or a minor version label (e.g., v4.3) and becomes an official release version, which is documented in the release notes page. Subsequently, any bugs identified in this version are fixed on the release branch. After manual testing is completed, the branch is assigned a bug-fix version label (e.g., v4.3.2), which is also reflected on the release notes page.

4.1.3 Choosing a Suitable Version

Since ESP-IDF officially began supporting ESP32-C3 from version v4.3, and v4.4 has not yet been officially released at the time of writing this book, the version used in this book is v4.3.2, which is a revised version of v4.3. However, it is important to note that by the time you read this book, v4.4 or newer versions may already be available. When selecting a version, we recommend the following:

- For **entry-level developers**, it is advisable to choose the stable v4.3 version or its revised version, which aligns with the example version used in this book.
- For **mass production** purposes, it is recommended to use the latest stable version to to benefit from the most up-to-date technical support.
- If you intend to experiment with **new chips** or explore **new product features**, please use the `master` branch. The latest version contains all the latest features, but keep in mind that there may be known or unknown bugs present.
- If the stable version being used does not include the desired new features and you wish to **minimise the risks** associated with the `master` branch, consider using the corresponding release branch, such as the `release/v4.4` branch. Espressif's GitHub repository will first create the `release/v4.4` branch and subsequently release the stable v4.4 version based on a specific historical snapshot of this branch, after completing all feature development and testing.

4.1.4 Overview of ESP-IDF SDK Directory

The ESP-IDF SDK consists of two main directories: `esp-idf` and `.espressif`. The former contains ESP-IDF repository's source code files and compilation scripts, while the latter mainly stores compilation tool chains and other software. Familiarity with these two directories will help developers make better use of available resources and speed up the development process. The directory structure of ESP-IDF is described below:

(1) **ESP-IDF repository code directory** (`~/esp/esp-idf`), as shown in Figure 4.2.

a. **Component directory `components`**

This core directory integrates numerous essential software components of ESP-IDF. No project code can be compiled without relying on the components within this directory. It includes driver support for various Espressif chips. From the LL library and HAL library interfaces for peripherals to the upper-level Driver and Virtual File System (VFS) layer support, developers can choose the appropriate components at different levels for their development needs. ESP-IDF also supports multiple standard network protocol stacks such as TCP/IP, HTTP, MQTT, WebSocket, etc. Developers can utilise familiar interfaces like Socket to build network applications. Components provide comprehen-

```
├── components
│   ├── bootloader
│   ├── driver
│   ├── esp_system
│   ├── esp_wifi
│   ├── freertos
│   └── ...
├── docs
├── examples
│   ├── bluetooth
│   ├── build_system
│   ├── common_components
│   ├── ethernet
│   ├── get-started
│   ├── mesh
│   ├── peripherals
│   ├── protocols
│   ├── provisioning
│   ├── security
│   ├── storage
│   ├── system
│   ├── wifi
│   └── ...
├── tools
│   ├── idf_monitor.py
│   ├── idf.py
│   └── ...
├── export.sh
└── install.sh
```

Figure 4.2. ESP-IDF repository code directory

sive functionality and can be easily integrated into applications, allowing developers to focus solely on the business logic. Some common components include:

- `driver`: This component contains peripheral driver programs for various Espressif chip series, such as GPIO, I2C, SPI, UART, LEDC (PWM), etc. The peripheral driver programs in this component offer chip-independent abstract interfaces. Each peripheral has a common header file (such as `gpio.h`), eliminating the need to deal with different chip-specific support questions.
- `esp_wifi`: Wi-Fi, as a special peripheral, is treated as a separate component. It includes multiple APIs such as initialisation of various Wi-Fi driver modes, parameter configuration, and event processing. Certain functions of this component are provided in the form of static link libraries. ESP-IDF also provides comprehensive driver documentation for ease of use.

- `freertos`: This component contains the complete FreeRTOS code. Apart from providing comprehensive support for this operating system, Espressif has also extended its support to dual-core chips. For dual-core chips like ESP32 and ESP32-S3, users can create tasks on specific cores.

b. Document directory `docs`

This directory contains ESP-IDF related development documents, including the Get Started Guide, API Reference Manual, Development Guide, etc.

NOTE

After being compiled by automated tools, the contents of this directory are deployed at https://docs.espressif.com/projects/esp-idf. Please ensure to switch the document target to ESP32-C3 and select the specified ESP-IDF version.

c. Script tool `tools`

This directory contains commonly used compilation front-end tools such as `idf.py`, and the monitor terminal tool `idf_monitor.py`, etc. The sub-directory `cmake` also contains core script files of the compilation system, serving as the foundation for implementing ESP-IDF compilation rules. When adding the environment variables, the contents within the `tools` directory are added to the system environment variable, allowing `idf.py` to be executed directly under the project path.

d. Example program directory `examples`

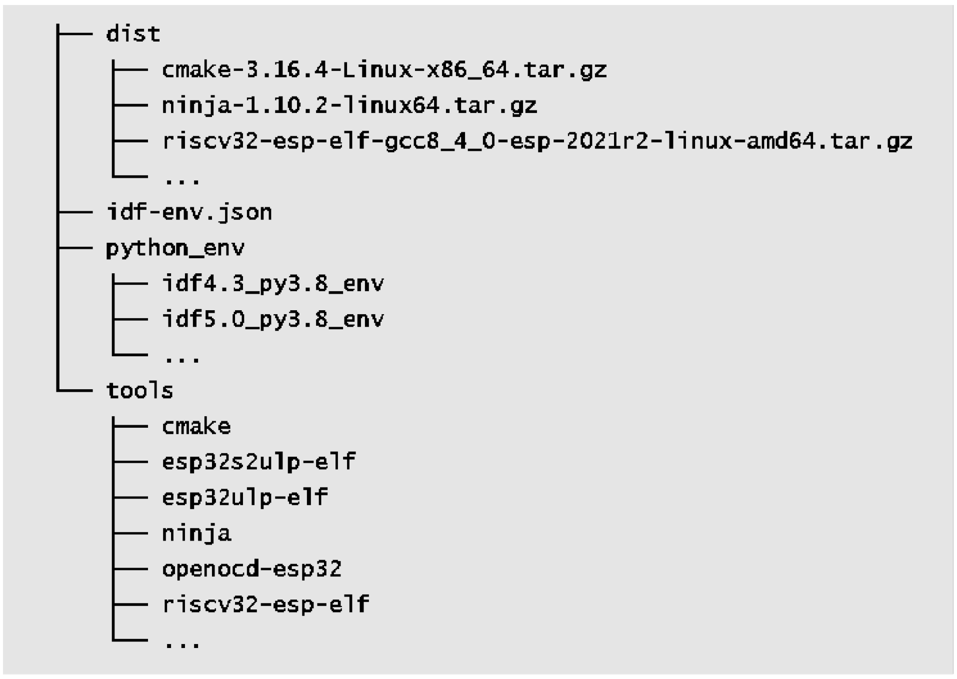
This directory comprises a vast collection of ESP-IDF example programs that demonstrate the usage of component APIs. The examples are organised into various sub-directories based on their categories:

- `get-started`: This sub-directory includes entry-level examples like "hello world" and "blink" to help users grasp the basics.
- `bluetooth`: You can find Bluetooth related examples here, including Bluetooth LE Mesh, Bluetooth LE HID, BluFi, and more.
- `wifi`: This sub-directory focuses on Wi-Fi examples, including basic programs like Wi-Fi SoftAP, Wi-Fi Station, `espnow`, as well as proprietary communication protocol examples from Espressif. It also includes multiple application layer examples based on Wi-Fi, such as Iperf, Sniffer, and Smart Config.
- `peripherals`: This extensive sub-directory is further divided into numerous sub-folders based on peripheral names. It mainly contains peripheral driver examples for Espressif chips, with each example featuring several sub-examples. For instance, the `gpio` sub-directory includes two examples: GPIO and GPIO matrix keyboard. It's important to note that not all examples in this directory are applicable to ESP32-C3.

For example, the examples in `usb/host` are only applicable to peripherals with USB Host hardware (such as ESP32-S3), and ESP32-C3 does not have this peripheral. The compilation system typically provides prompts when setting the target. The README file of each example lists the supported chips.

- `protocols`: This sub-directory contains examples for various communication protocols, including MQTT, HTTP, HTTP Server, PPPoS, Modbus, mDNS, SNTP, covering a wide range of communication protocol examples required for IoT development.
- `provisioning`: Here, you'll find provisioning examples for different methods, such as Wi-Fi provisioning and Bluetooth LE provisioning.
- `system`: This sub-directory includes system debugging examples (e.g., stack tracing, runtime tracing, task monitoring), power management examples (e.g., various sleep modes, co-processors), and examples related to common system components like console terminal, event loop, and system timer.
- `storage`: Within this sub-directory, you'll discover examples of all file systems and storage mechanisms supported by ESP-IDF (such as reading and writing of Flash, SD card and other storage media), as well as examples of non-volatile storage (NVS), FatFS, SPIFFS and other file system operations.
- `security`: This sub-directory contains examples related to flash encryption.

(2) **ESP-IDF compilation tool chain directory** (~/`.espressif`), as shown in Figure 4.3.



```
├── dist
│   ├── cmake-3.16.4-Linux-x86_64.tar.gz
│   ├── ninja-1.10.2-linux64.tar.gz
│   ├── riscv32-esp-elf-gcc8_4_0-esp-2021r2-linux-amd64.tar.gz
│   └── ...
├── idf-env.json
├── python_env
│   ├── idf4.3_py3.8_env
│   ├── idf5.0_py3.8_env
│   └── ...
└── tools
    ├── cmake
    ├── esp32s2ulp-elf
    ├── esp32ulp-elf
    ├── ninja
    ├── openocd-esp32
    ├── riscv32-esp-elf
    └── ...
```

Figure 4.3. ESP-IDF compilation tool chain directory

a. **Software distribution directory `dist`**

   The ESP-IDF tool chain and other software are distributed in the form of compressed packages. During the installation process, the installation tool first downloads the compressed package to the `dist` directory, and then extracts it to the specified directory. Once the installation is complete, the contents in this directory can be safely removed.

b. **Python virtual environment directory `python_env`**

   Different versions of ESP-IDF rely on specific versions of Python packages. Installing these packages directly on the same host can lead to conflicts between package versions. To address this, ESP-IDF utilises Python virtual environments to isolate different package versions. With this mechanism, developers can install multiple versions of ESP-IDF on the same host and easily switch between them by importing different environment variables.

c. **ESP-IDF compilation tool chain directory `tools`**

   This directory mainly contains cross-compilation tools required to compile ESP-IDF projects, such as CMake tools, Ninja build tools, and the gcc tool chain that generates the final executable program. Additionally, this directory houses the standard library of the C/C++ language along with the corresponding header files. If a program references a system header file like `#include <stdio.h>`, the compilation tool chain will locate the `stdio.h` file within this directory.

4.2 Setting Up ESP-IDF Development Environment

The ESP-IDF development environment supports mainstream operating systems such as Windows, Linux, and macOS. This section will introduce how to set up the development environment on each system. It is recommended to develop ESP32-C3 on Linux system, which will be introduced in detail here. Many instructions are applicable across platforms due to the similarity of the development tools. Therefore, it is advised to carefully read the content of this section.

NOTE

You can refer to the online documents available at https://bookc3.espressif.com/esp32c3, which provide the commands mentioned in this section.

4.2.1 Setting up ESP-IDF Development Environment on Linux

The GNU development and debugging tools required for the ESP-IDF development environment are native to the Linux system. Additionally, the command-line terminal in Linux is powerful and user-friendly, making it an ideal choice for ESP32-C3 development. You can select your preferred Linux distribution, but we recommend using Ubuntu or other Debian-based systems. This section provides guidance on setting up the ESP-IDF development environment on Ubuntu 20.04.

1. Install required packages

Open a new terminal and execute the following command to install all necessary packages. The command will automatically skip packages that are already installed.

```
$ sudo apt-get install git wget flex bison gperf python3 python3-pip python3-
setuptools cmake ninja-build ccache libffi-dev libssl-dev dfu-util libusb-1.0-0
```

TIPS

You need to use the administrator account and password for the command above. By default, no information will be displayed when entering the password. Simply press the "Enter" key to continue the procedure.

Git is a key code management tool in ESP-IDF. After successfully setting up the development environment, you can use the `git log` command to view all code changes made since the creation of ESP-IDF. In addition, Git is also used in ESP-IDF to confirm version information, which is necessary for installing the correct tool chain corresponding to specific versions. Along with Git, other important system tools include Python. ESP-IDF incorporates numerous automation scripts written in Python. Tools such as CMake, Ninja-build, and Ccache are widely used in C/C++ projects and serve as the default code compilation and building tools in ESP-IDF. `libusb-1.0-0` and `dfu-util` are the main drivers used for USB serial communication and firmware burning.

Once the software packages are installed, you can use the `apt show <package_name>` command to obtain detailed descriptions of each package. For example, use `apt show git` to print the description information for the Git tool.

Q: What to do if the Python version is not supported?

A: ESP-IDF v4.3 requires a Python version that is not lower than v3.6. For older versions of Ubuntu, please manually download and install a higher version of Python and set Python3 as the default Python environment. You can find detailed instructions by searching for the keyword `update-alternatives python`.

2. Download ESP-IDF repository code

Open a terminal and create a folder named `esp` in your home directory using the `mkdir` command. You can choose a different name for the folder if you prefer. Use the `cd` command to enter the folder.

```
$ mkdir -p ~/esp
$ cd ~/esp
```

Use the `git clone` command to download the ESP-IDF repository code, as shown below:

```
$ git clone -b v4.3.2 --recursive https://github.com/espressif/esp-idf.git
```

In the command above, the parameter `-b v4.3.2` specifies the version to download (in this case, version 4.3.2). The parameter `--recursive` ensures that all sub-repositories of ESP-IDF are downloaded recursively. Information about sub-repositories can be found in the `.gitmodules` file.

3. Install the ESP-IDF development tool chain

Espressif provides an automated script `install.sh` to download and install the tool chain. This script checks the current ESP-IDF version and operating system environment, and then downloads and installs appropriate version of Python tool packages and compilation tool chains. The default installation path for the tool chain is `~/.espressif`. All you need to do is to navigate to the `esp-idf` directory and run `install.sh`.

```
$ cd ~/esp/esp-idf
$ ./install.sh
```

If you install the the tool chain successfully, the terminal will display:

```
All done!
```

At this point, you have successfully set up the ESP-IDF development environment.

4.2.2 Setting up ESP-IDF Development Environment on Windows

1. Download ESP-IDF tools installer

TIPS

It is recommended to set up the ESP-IDF development environment on Windows 10 or above. You can download the installer from https://dl.espressif.com/dl/esp-idf/. The installer is also an open-source software, and its source code can be viewed at https://github.com/espressif/idf-installer.

- **Online ESP-IDF tools installer**

  This installer is relatively small, around 4 MB in size, and other packages and code will be downloaded during the installation process. The advantage of the online installer is that not only can software packages and code be downloaded on demand during the installation process, but also allows the installation of all available releases of ESP-IDF and the latest branch of GitHub code (such as the `master` branch). The disadvantage is that it requires a network connection during the installation process, which may cause installation failure due to network problems.

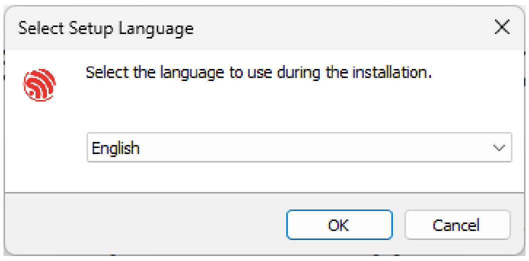
- **Offline ESP-IDF tools installer**

  This installer is larger, about 1 GB in size, and contains all the software packages and code required for environment set up. The main advantage of the offline installer is that it can be used on computers without Internet access, and generally has a higher installation success rate. It should be noted that the offline installer can only install stable releases of ESP-IDF identified by v*.* or v*.*.*.

2. Run the ESP-IDF tools installer

After downloading a suitable version of the installer (take ESP-IDF Tools Offline 4.3.2 for example here), double-click the exe file to launch the ESP-IDF installation interface. The following demonstrates how to install ESP-IDF stable version v4.3.2 using the offline installer.

(1) In the "Select installation language" interface shown in Figure 4.4, select the language to be used from the drop-down list.

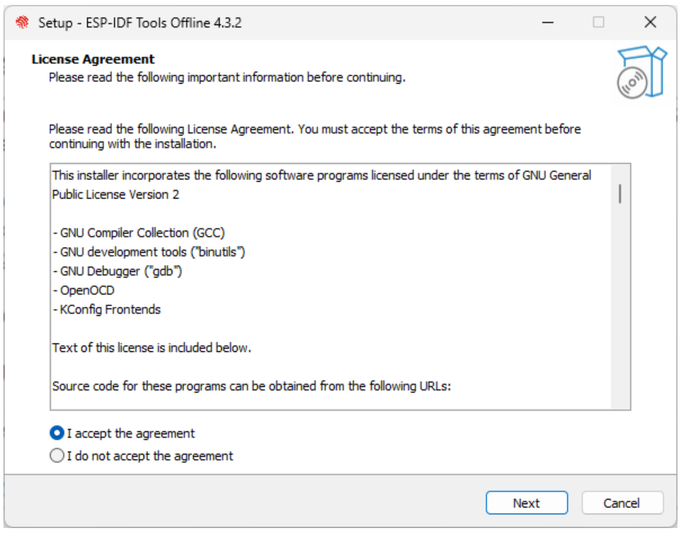

Select Setup Language

Select the language to use during the installation.

English

Figure 4.4. "Select installation language" interface

(2) After selecting the language, click "OK" to pop up the "License agreement" interface (see Figure 4.5). After carefully reading the installation license agreement, select "I accept the agreement" and click "Next".


Setup - ESP-IDF Tools Offline 4.3.2

License Agreement
Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

This installer incorporates the following software programs licensed under the terms of GNU General Public License Version 2

- GNU Compiler Collection (GCC)
- GNU development tools ("binutils")
- GNU Debugger ("gdb")
- OpenOCD
- KConfig Frontends

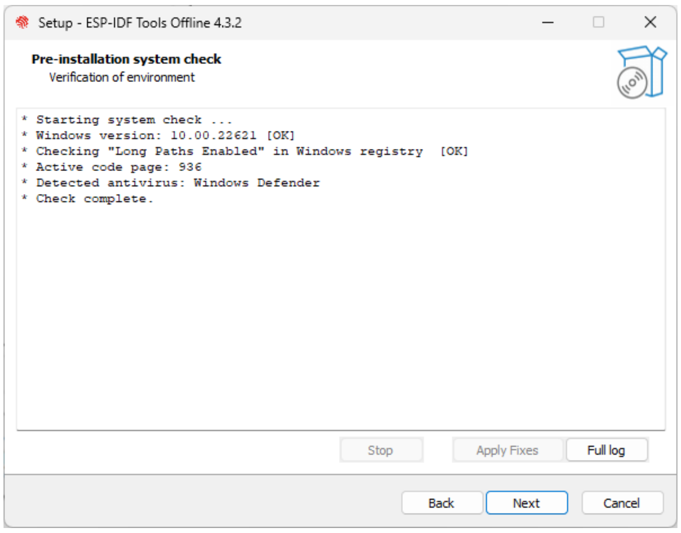
Text of this license is included below.

Source code for these programs can be obtained from the following URLs:

[✓ I accept the agreement]
I do not accept the agreement

Figure 4.5. "License agreement" interface

(3) Review the system configuration in the "Pre-installation system check" interface (see Figure 4.6). Check the Windows version and the installed antivirus software information. Click "Next" if all the configuration items are normal. Otherwise, you can click "Full log" for solutions based on key items.


Setup - ESP-IDF Tools Offline 4.3.2
Pre-installation system check
Verification of environment
Stop
Apply Fixes
Full log
Back
Next
Cancel

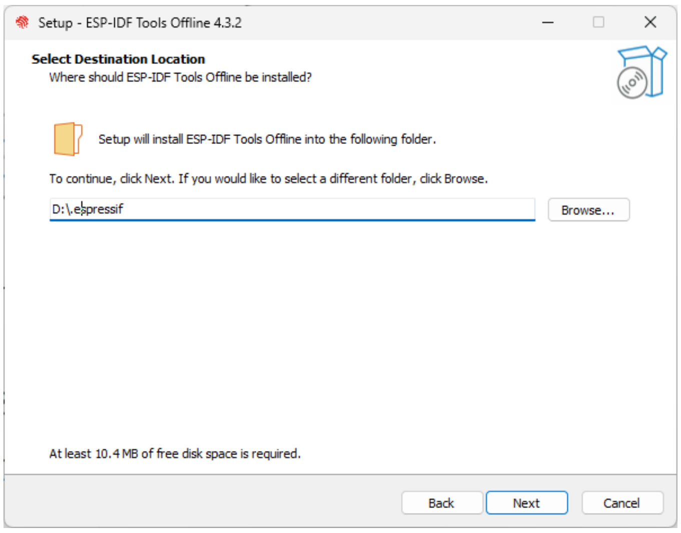
```
* Starting system check ...
* Windows version: 10.00.22621 [OK]
* Checking "Long Paths Enabled" in Windows registry  [OK]
* Active code page: 936
* Detected antivirus: Windows Defender
* Check complete.
```

Figure 4.6. "System check before installation" interface

TIPS

You can submit logs to https://github.com/espressif/idf-installer/issues for help.

(4) Select the ESP-IDF installation directory. Here, select `D:/.espressif`, as shown in Figure 4.7, and click "Next". Please note that `.espressif` here is a hidden directory. After the installation is completed, you can view the specific contents of this directory by opening the file manager and displaying hidden items.


Setup - ESP-IDF Tools Offline 4.3.2
Select Destination Location
Where should ESP-IDF Tools Offline be installed?
Browse...
Back
Next
Cancel

Setup will install ESP-IDF Tools Offline into the following folder.

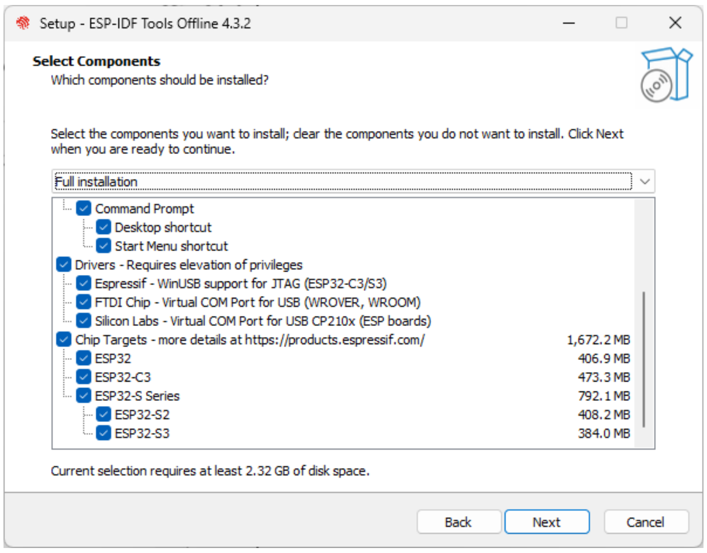
To continue, click Next. If you would like to select a different folder, click Browse.

D:\.espressif

At least 10.4 MB of free disk space is required.

Figure 4.7. Select the ESP-IDF installation directory

(5) Check the components that need to be installed, as shown in Figure 4.8. It is recommended to use the default option, that is, complete installation, and then click "Next".


Setup - ESP-IDF Tools Offline 4.3.2

Select Components
Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

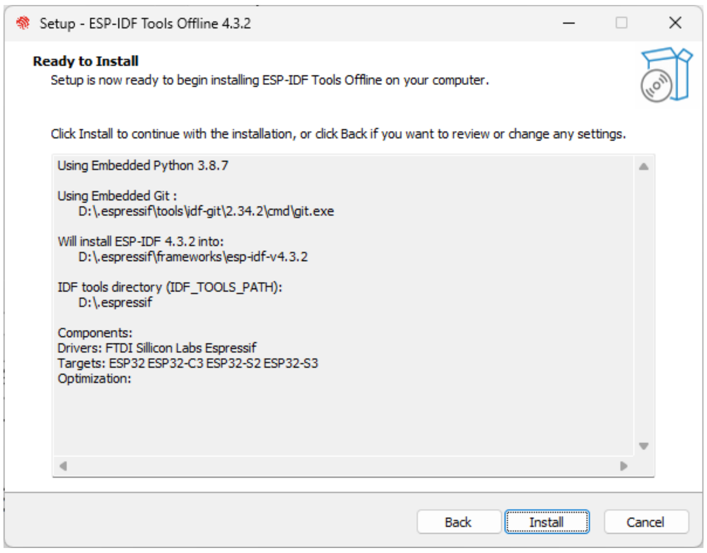
Full installation

- [x] Command Prompt
  - [x] Desktop shortcut
  - [x] Start Menu shortcut
- [x] Drivers - Requires elevation of privileges
  - [x] Espressif - WinUSB support for JTAG (ESP32-C3/S3)
  - [x] FTDI Chip - Virtual COM Port for USB (WROVER, WROOM)
  - [x] Silicon Labs - Virtual COM Port for USB CP210x (ESP boards)
- [x] Chip Targets - more details at https://products.espressif.com/ 1,672.2 MB
  - [x] ESP32 406.9 MB
  - [x] ESP32-C3 473.3 MB
  - [x] ESP32-S Series 792.1 MB
    - [x] ESP32-S2 408.2 MB
    - [x] ESP32-S3 384.0 MB

Current selection requires at least 2.32 GB of disk space.

Figure 4.8. Select the components to install

(6) Confirm the components to be installed and click "Install" to start the automated installation process, as shown in Figure 4.9. The installation process may last tens of minutes and the progress bar of the installation process is shown in Figure 4.10. Please wait patiently.


Setup - ESP-IDF Tools Offline 4.3.2

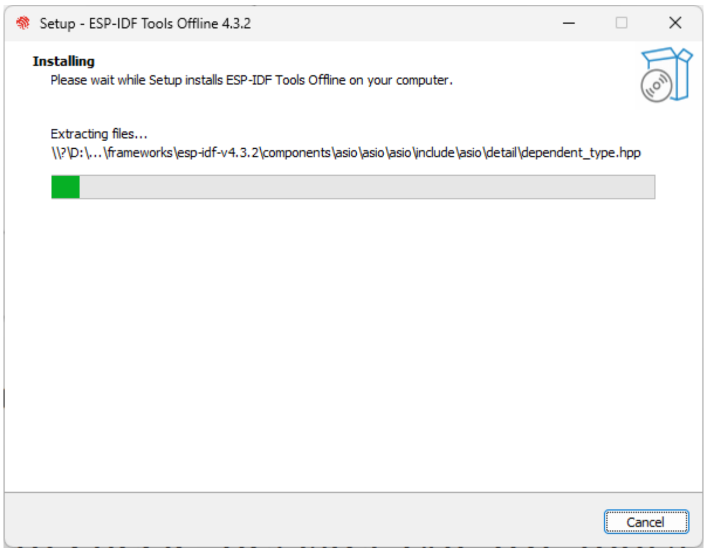
Ready to Install
Setup is now ready to begin installing ESP-IDF Tools Offline on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

```
Using Embedded Python 3.8.7

Using Embedded Git :
    D:\.espressif\tools\idf-git\2.34.2\cmd\git.exe

Will install ESP-IDF 4.3.2 into:
    D:\.espressif\frameworks\esp-idf-v4.3.2

IDF tools directory (IDF_TOOLS_PATH):
    D:\.espressif

Components:
Drivers: FTDI Sillicon Labs Espressif
Targets: ESP32 ESP32-C3 ESP32-S2 ESP32-S3
Optimization:
```

Figure 4.9. Preparing for installation

(7) After the installation is complete, it is recommended to check "Register the ESP-IDF Tools executables as Windows Defender exclusions..." to prevent antivirus software from deleting files. Adding exclusion items can also skip frequent scans by antivirus


Setup - ESP-IDF Tools Offline 4.3.2

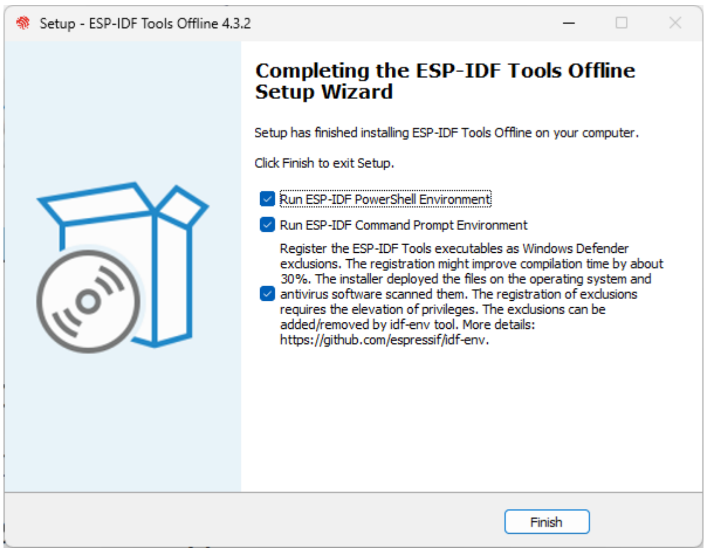
Installing

Please wait while Setup installs ESP-IDF Tools Offline on your computer.

Extracting files...
\\?\D:\...\frameworks\esp-idf-v4.3.2\components\asio\asio\asio\include\asio\detail\dependent_type.hpp

Figure 4.10. Installation progress bar

software, greatly improving the code compilation efficiency of the Windows system. Click “Finish” to complete the installation of the development environment, as shown in Figure 4.11. You can choose to check “Run ESP-IDF PowerShell environment” or “Run ESP-IDF command prompt”. Run the compilation window directly after installation to ensure that the development environment functions normally.


Setup - ESP-IDF Tools Offline 4.3.2

Completing the ESP-IDF Tools Offline Setup Wizard

Setup has finished installing ESP-IDF Tools Offline on your computer.

Click Finish to exit Setup.

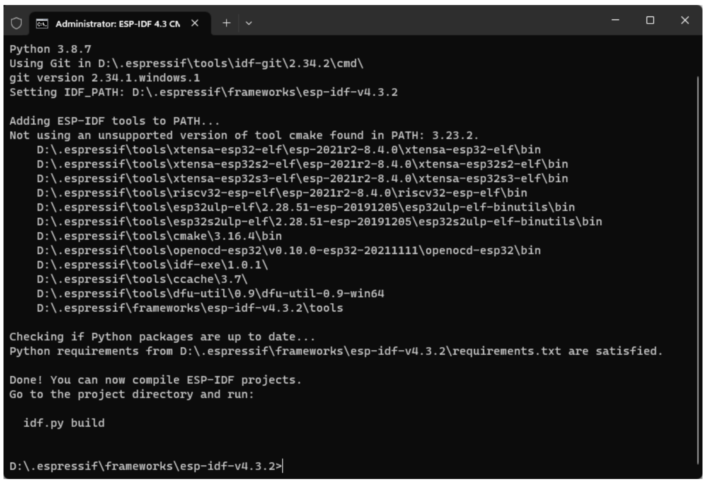
- [✓] Run ESP-IDF PowerShell Environment
- [✓] Run ESP-IDF Command Prompt Environment
- [✓] Register the ESP-IDF Tools executables as Windows Defender exclusions. The registration might improve compilation time by about 30%. The installer deployed the files on the operating system and antivirus software scanned them. The registration of exclusions requires the elevation of privileges. The exclusions can be added/removed by idf-env tool. More details: https://github.com/espressif/idf-env.

Figure 4.11. Installation completed

(8) Open the installed development environment in the program list (either ESP-IDF 4.3 CMD or ESP-IDF 4.3 PowerShell terminal, as shown in Figure 4.12), and the ESP-IDF environment variable will be automatically added when running in the terminal. After that, you can use the `idf.py` command for operations. The opened ESP-IDF 4.3 CMD is shown in Figure 4.13.

ESP-IDF
New
ESP-IDF 4.3 CMD
New
ESP-IDF 4.3 PowerShell
New

Figure 4.12. Development environment installed



```
Python 3.8.7
Using Git in D:\.espressif\tools\idf-git\2.34.2\cmd\
git version 2.34.1.windows.1
Setting IDF_PATH: D:\.espressif\frameworks\esp-idf-v4.3.2

Adding ESP-IDF tools to PATH...
Not using an unsupported version of tool cmake found in PATH: 3.23.2.
    D:\.espressif\tools\xtensa-esp32-elf\esp-2021r2-8.4.0\xtensa-esp32-elf\bin
    D:\.espressif\tools\xtensa-esp32s2-elf\esp-2021r2-8.4.0\xtensa-esp32s2-elf\bin
    D:\.espressif\tools\xtensa-esp32s3-elf\esp-2021r2-8.4.0\xtensa-esp32s3-elf\bin
    D:\.espressif\tools\riscv32-esp-elf\esp-2021r2-8.4.0\riscv32-esp-elf\bin
    D:\.espressif\tools\esp32ulp-elf\2.28.51-esp-20191205\esp32ulp-elf-binutils\bin
    D:\.espressif\tools\esp32s2ulp-elf\2.28.51-esp-20191205\esp32s2ulp-elf-binutils\bin
    D:\.espressif\tools\cmake\3.16.4\bin
    D:\.espressif\tools\openocd-esp32\v0.10.0-esp32-20211111\openocd-esp32\bin
    D:\.espressif\tools\idf-exe\1.0.1\
    D:\.espressif\tools\ccache\3.7\
    D:\.espressif\tools\dfu-util\0.9\dfu-util-0.9-win64
    D:\.espressif\frameworks\esp-idf-v4.3.2\tools

Checking if Python packages are up to date...
Python requirements from D:\.espressif\frameworks\esp-idf-v4.3.2\requirements.txt are satisfied.

Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build


D:\.espressif\frameworks\esp-idf-v4.3.2>
```

Figure 4.13. ESP-IDF 4.3 CMD

4.2.3 Setting up ESP-IDF Development Environment on Mac

The process of installing the ESP-IDF development environment on a Mac system is the same as that on a Linux system. The commands for downloading the repository code and installing the tool chain are exactly the same. Only the commands for installing dependency packages are slightly different.

1. Install dependency packages

Open a terminal, and install pip, the Python package management tool, by running the following command:

```
% sudo easy_install pip
```

Install Homebrew, a package management tool for macOS, by running the following command:

```
% /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
HEAD/install.sh)"
```

Install the required dependency packages by running the following command:

```
% brew python3 install cmake ninja ccache dfu-util
```

2. Download ESP-IDF repository code

Follow the instructions provided in section 4.2.1 to download the ESP-IDF repository code. The steps are the same as for downloading on a Linux system.

3. Install the ESP-IDF development tool chain

Follow the instructions provided in section 4.2.1 to install the ESP-IDF development tool chain. The steps are the same as for installation on a Linux system.

4.2.4 Installing VS Code

By default, the ESP-IDF SDK does not include a code editing tool (though the latest ESP-IDF installer for Windows offers the option to install ESP-IDF Eclipse). You can use any text editing tool of your choice to edit the code and then compile it using terminal commands.

One popular code editing tool is VS Code (Visual Studio Code), which is a free and feature-rich code editor with a user-friendly interface. It offers various plugins that provide functionalities such as code navigation, syntax highlighting, Git version control, and terminal integration. Additionally, Espressif has developed a dedicated plugin called Espressif IDF for VS Code, which simplifies project configuration and debugging.

You can use the `code` command in the terminal to quickly open the current folder in VS Code. Alternatively, you can use the shortcut `Ctrl+~` to open the system's default terminal console within VS Code.

TIPS

It is recommended to use VS Code for ESP32-C3 code development. Download and install the latest version of VS Code at https://code.visualstudio.com/.

4.2.5 Introduction to Third-Party Development Environments

In addition to the official ESP-IDF development environment, which primarily uses the C language, ESP32-C3 also supports other mainstream programming languages and a wide range of third-party development environments. Some notable options include:

Arduino:

an open-source platform for both hardware and software, supporting various microcontrollers, including ESP32-C3.

It uses the C++ language and offers a simplified and standardised API, commonly referred to as the Arduino language. Arduino is widely used in prototype development and educational contexts. It provides an extensible software package and an IDE that allows for easy compilation and flashing.

MicroPython:

a Python 3 language interpreter designed to run on embedded microcontroller platforms.

With a simple script language, it can directly access ESP32-C3's peripheral resources (such as UART, SPI, and I2C) and communication functions (such as Wi-Fi and Bluetooth LE).

This simplifies hardware interaction. MicroPython, combined with Python's extensive mathematical operation library, enables the implementation of complex algorithms on ESP32-C3, facilitating the development of AI-related applications. As a script language, there is no need for repeated compilation; modifications can be made and scripts can be executed directly.

NodeMCU:
an LUA language interpreter developed for ESP series chips.

It supports almost all peripheral functions of ESP chips and is lighter than MicroPython. Similar to MicroPython, NodeMCU uses a script language, eliminating the need for repeated compilation.

Furthermore, ESP32-C3 also supports the NuttX and Zephyr operating systems. NuttX is a real-time operating system that provides POSIX-compatible interfaces, enhancing application portability. Zephyr is a small real-time operating system specifically designed for IoT applications. It includes numerous software libraries required in IoT development, gradually evolving into a comprehensive software ecosystem.

This book does not provide detailed installation instructions for the aforementioned development environments. You can install a development environment based on your requirements by following the respective documentation and instructions.

4.3 ESP-IDF Compilation System

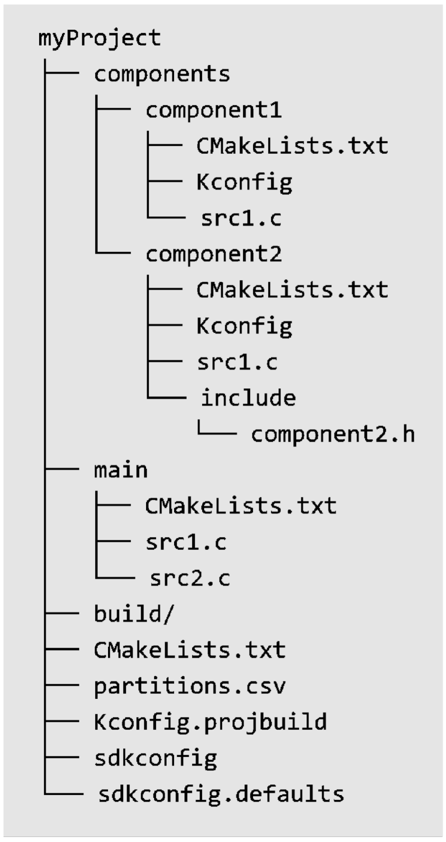
4.3.1 Basic Concepts of Compilation System

An ESP-IDF project is a collection of a main program with an entry function and multiple independent functional components. For example, a project that controls LED switches mainly consists of an entry program `main` and a `driver` component that controls GPIO. If you want to realise the LED remote control, you also need to add Wi-Fi, TCP/IP protocol stack, etc.

The compilation system can compile, link, and generate executable files (.bin) for the code through a set of building rules. The compilation system of ESP-IDF v4.0 and above versions is based on CMake by default, and the compilation script `CMakeLists.txt` can be used to control the compilation behavior of the code. In addition to supporting the basic syntax of CMake, the ESP-IDF compilation system also defines a set of default compilation rules and CMake functions, and you can write the compilation script with simple statements.

4.3.2 Project File Structure

A project is a folder that contains an entry program `main`, user-defined components, and files required to build executable applications, such as compilation scripts, configuration files, partition tables, etc. Projects can be copied and passed on, and the same executable file can be compiled and generated in machines with the same version of ESP-IDF development environment. A typical ESP-IDF project file structure is shown in Figure 4.14.



```
myProject
├── components
│   ├── component1
│   │   ├── CMakeLists.txt
│   │   ├── Kconfig
│   │   └── src1.c
│   └── component2
│       ├── CMakeLists.txt
│       ├── Kconfig
│       ├── src1.c
│       └── include
│           └── component2.h
├── main
│   ├── CMakeLists.txt
│   ├── src1.c
│   └── src2.c
├── build/
├── CMakeLists.txt
├── partitions.csv
├── Kconfig.projbuild
├── sdkconfig
└── sdkconfig.defaults
```

Figure 4.14. Typical ESP-IDF project file structure

Since ESP-IDF supports multiple IoT chips from Espressif, including ESP32, ESP32-S series, ESP32-C series, ESP32-H series, etc., a target needs to be determined before compiling the code. The target is both the hardware device that runs the application program and the build target of the compilation system.

Depending on your needs, you can specify one or more targets for your project. For example, through command `idf.py set-target esp32c3`, you can set the compilation target to ESP32-C3, during which the default parameters and compilation tool chain path for ESP32-C3 will be loaded. After compilation, an executable program can be generated for ESP32-C3. You can also run the command `set-target` again to set a different target, and the compilation system will automatically clean up and reconfigure.

Components

Components in ESP-IDF are modular and independent code units managed within the compilation system. They are organised as folders, with the folder name representing the component name by default. Each component has its own compilation script that specifies its compilation parameters and dependencies. During the compilation process, components are compiled into separate static libraries (.a files) and eventually combined with other components to form the application program.

ESP-IDF provides essential functions, such as the operating system, peripheral drivers, and network protocol stack, in the form of components. These components are stored in the `components` directory located within the ESP-IDF root directory. Developers do not need to copy these components to the `components` directory of `myProject`. Instead, they only need to specify the dependency relationships of these components in the project's `CMakeLists.txt` file using the `REQUIRES` or `PRIV_REQUIRES` directives. The compilation system will automatically locate and compile the required components.

Therefore, the `components` directory under `myProject` is not necessary. It is only used to include some custom components of the project, which can be third-party libraries or user-defined code. Additionally, components can be sourced from any directory other than ESP-IDF or the current project, such as from an open-source project saved in another directory. In this case, you only need to add the path of the component by setting the `EXTRA_COMPONENT_DIRS` variable in the `CMakeLists.txt` under the root directory. This directory will override any ESP-IDF component with the same name, ensuring the correct component is used.

Entry program `main`

The `main` directory within the project follows the same file structure as other components (e.g., `component1`). However, it holds a special significance as it is a mandatory component that must exist in every project. The main directory contains the project's source code and the user program's entry point, typically named `app_main`. By default, the execution of the user program starts from this entry point. The `main` component also differs in that it automatically depends on all components within the search path. Therefore, there is no need to explicitly indicate dependencies using the `REQUIRES` or `PRIV_REQUIRES` directives in the `CMakeLists.txt` file.

Configuration file

The root directory of the project contains a configuration file called `sdkconfig`, which contains the configuration parameters for all the components within the project. The `sdkconfig` file is automatically generated by the compilation system and can be modified and regenerated by the command `idf.py menuconfig`. The menuconfig options mainly originate from the `Kconfig.projbuild` of the project and the `Kconfig` of the components. Component developers generally add configuration items in `Kconfig` to make the component flexible and configurable.

`Build` directory

By default, the `build` directory within the project stores intermediate files and the final executable programs generated by the `idf.py build` command. In general, it is not necessary to directly access the contents of the `build` directory. ESP-IDF provides predefined commands to interact with the directory, such as using the `idf.py flash` command to automatically locate the compiled binary file and flash it to the specified flash address, or using the `idf.py fullclean` command to clean the entire `build` directory.

Partition table (`partitions.csv`)

Each project requires a partition table to divide the space of flash and specify the size and starting address of the executable program and user data space. Command `idf.py flash` or OTA upgrade program will flash the firmware to the corresponding address according to this table. ESP-IDF provides several default partition tables in `components/partition_table`, such as `partitions_singleapp.csv` and `partitions_two_ota.csv`, which can be selected in `menuconfig`.

If the default partition table of the system cannot meet the requirements of the project, a custom `partitions.csv` can be added to the project directory and be selected in `menuconfig`.

4.3.3 Default Build Rules of the Compilation System

Rules for overriding components with the same name

During the component search process, the compilation system follows a specific order. It first searches for internal components of ESP-IDF, then looks for components of the user project, and finally searches for components in `EXTRA_COMPONENT_DIRS`. In cases where multiple directories contain components with the same name, the component found in the last directory will override any previous components with the same name. This rule allows for the customisation of ESP-IDF components within the user project, while keeping the original ESP-IDF code intact.

Rules for including common components by default

As mentioned in section 4.3.2, components need to explicitly specify their dependencies on other components in the `CMakeLists.txt`. However, common components such as `freertos` are automatically included in the build system by default, even if their dependency relationships are not explicitly defined in the compilation script. ESP-IDF common components include `freertos`, `Newlib`, `heap`, `log`, `soc`, `esp_rom`, `esp_common`, `xtensa/riscv`, and `cxx`. Using these common components avoids repetitive work when writing `CMakeLists.txt` and make it more concise.

Rules for overriding configuration items

Developers can add default configuration parameters by adding a default configuration file named `sdkconfig.defaults` to the project. For example, adding `CONFIG_LOG_`

DEFAULT_LEVEL_NONE = y can configure the UART interface to not print log data by default. Furthermore, if specific parameters need to be set for a particular target, a configuration file named sdkconfig.defaults.TARGET_NAME can be added, where TARGET_NAME can be esp32s2, esp32c3, and so on. These configuration files are imported into the sdkconfig during compilation, with the general default configuration file sdkconfig.defaults being imported first, followed by the target-specific configuration file, such as sdkconfig.defaults.esp32c3. In cases where there are configuration items with the same name, the latter configuration file will override the former.

4.3.4 Introduction to the Compilation Script

When developing a project using ESP-IDF, developers not only need to write source code but also need to write CMakeLists.txt for the project and components. CMakeLists.txt is a text file, also known as a compilation script, which defines a series of compilation objects, compilation configuration items, and commands to guide the compilation process of the source code. The compilation system of ESP-IDF v4.3.2 is based on CMake. In addition to supporting native CMake functions and commands, it also defines a series of custom functions, making it much easier to write compilation scripts.

The compilation scripts in ESP-IDF mainly include the project compilation script and the component compilation scripts. The CMakeLists.txt in the root directory of the project is called the project compilation script, which guides the compilation process of the entire project. A basic project compilation script typically includes the following three lines:

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

Among them, the cmake_minimum_required (VERSION 3.5) must be placed on the first line, which is used to indicate the minimum CMake version number required by the project. Newer versions of CMake are generally backward compatible with older versions, so adjust the version number accordingly when using newer CMake commands to ensure compatibility.

include($ENV {IDF_PATH}/tools/cmake/project.cmake) imports pre-defined configuration items and commands of ESP-IDF compilation system, including the default build rules of the compilation system described in Section 4.3.3. project(myProject) creates the project itself and specifies its name. This name will be used as the final output binary file name, i.e., myProject.elf and myProject.bin.

A project can have multiple components, including the main component. The top-level directory of each component contains a CMakeLists.txt file, which is called the component compilation script. Component compilation scripts are mainly used to specify component dependencies, configuration parameters, source code files, and included header files for compilation. With ESP-IDF's custom function `idf_component_register`, the minimum required code for a component compilation script is as follows:

```
idf_component_register(SRCS "src1.c"
                       INCLUDE_DIRS "include"
                       REQUIRES component1)
```

The `SRCS` parameter provides a list of source files in the component, separated by spaces if there are multiple files. The `INCLUDE_DIRS` parameter provides a list of public header file directories for the component, which will be added to the `include` search path for other components that depend on the current component. The `REQUIRES` parameter identifies the public component dependencies for the current component. It is necessary for components to explicitly state which components they depend on, such as `component2` depending on `component1`. However, for the `main` component, which depends on all components by default, the `REQUIRES` parameter can be omitted.

In addition, native CMake commands can also be used in the compilation script. For example, use the command `set` to set variables, such as `set(VARIABLE "VALUE")`.

4.3.5 Introduction to Common Commands

ESP-IDF uses CMake (project configuration tool), Ninja (project building tool) and esptool (flash tool) in the process of code compilation. Each tool plays a different role in the compilation, building, and flash process, and also supports different operating commands. To facilitate user operation, ESP-IDF adds a unified front-end `idf.py` that allows the above commands to be called quickly.

Before using `idf.py`, make sure that:

- The environment variable `IDF_PATH` of ESP-IDF has been added to the current terminal.
- The command execution directory is the root directory of the project, which includes the project compilation script `CMakeLists.txt`.

The common commands of `idf.py` are as follows:

- `idf.py --help`: displaying a list of commands and their usage instructions.
- `idf.py set-target <target>`: setting the compilation taidf.py fullcleanrget, such as replacing `<target>` with `esp32c3`.
- `idf.py menuconfig`: launching `menuconfig`, a terminal graphical configuration tool, which can select or modify configuration options, and the configuration results are saved in the `sdkconfig` file.
- `idf.py build`: initiating code compilation. The intermediate files and the final executable program generated by the compilation will be saved in the `build` directory of the project by default. The compilation process is incremental, which means that if only one source file is modified, only the modified file will be compiled next time.

- `idf.py clean`: cleaning the intermediate files generated by the project compilation. The entire project will be forced to compile in the next compilation. Note that the CMake configuration and the configuration modifications made by `menuconfig` will not be deleted during cleanup.
- `idf.py fullclean`: deleting the entire `build` directory, including all CMake configuration output files. When building the project again, CMake will configure the project from scratch. Please note that this command will recursively delete all files in the build directory, so use it with caution, and the project configuration file will not be deleted.
- `idf.py flash`: flashing the executable program binary file generated by `build` to the target ESP32-C3. The options `-p <port_name>` and `-b <baud_rate>` are used to set the device name of the serial port and the baud rate for flashing, respectively. If these two options are not specified, the serial port will be automatically detected and the default baud rate will be used.
- `idf.py monitor`: displaying the serial port output of the target ESP32-C3. The option `-p` can be used to specify the device name of the host-side serial port. During serial port printing, press the key combination `Ctrl+]` to exit the monitor.

The above commands can also be combined as needed. For example, the command `idf.py build flash monitor` will perform code compilation, flash, and open the serial port monitor in sequence.

You can visit https://bookc3.espressif.com/build-system to know more about ESP-IDF compilation system.

4.4 Practice: Compiling Example Program "Blink"

4.4.1 Example Analysis

This section will take the program Blink as an example to analyse the file structure and coding rules of a real project in detail. The Blink program implements the LED blinking effect, and the project is located in the directory `examples/get-started/blink`, which contains a source file, configuration files, and several compilation scripts.

The smart light project introduced in this book is based on this example program. Functions will be gradually added in later chapters to finally complete it.

Source code

In order to demonstrate the entire development process, the Blink program has been copied to `esp32c3-iot-projects/device_firmware/1_blink`.

The directory structure of the `blink` project files is shown in Figure 4.15.

The `blink` project contains only one `main` directory, which is a special component that

```
blink
├── main
│   ├── blink.c
│   ├── CMakeLists.txt
│   └── Kconfig.projbuild
├── CMakeLists.txt
├── sdkconfig.defaults
└── ...
```

Figure 4.15. File directory structure of the `blink` project

must be included as described in section 4.3.2. The main directory is mainly used to store the implementation of the `app_main()` function, which is the entry point to the user program.The `blink` project does not include the `components` directory, because this example only needs to use the components that come with ESP-IDF and does not require additional components. The `CMakeLists.txt` included in the `blink` project is used to guide the compilation process, while `Kconfig.projbuild` is used to add configuration items for this example program in `menuconfig`. Other unnecessary files will not affect the compilation of the code, so they will not be discussed here. A detailed introduction to the `blink` project files is as follows.

```c
/*blink.c includes the following header files*/
#include <stdio.h>              //Standard C library header file
#include "freertos/freeRTOS.h"  //FreeRTOS main header file
#include "freertos/task.h"      //FreeRTOS Task header file
#include "sdkconfig.h"      //Configuration header file generated by kconfig
#include "driver/gpio.h"        //GPIO driver header file
```

The source file `blink.c` contains a series of header files corresponding to function declarations. ESP-IDF generally follows the order of including standard library header files, FreeRTOS header files, driver header files, other component header files, and project header files. The order in which header files are included may affect the final compilation result, so try to follow the default rules. It should be noted that `sdkconfig.h` is automatically generated by `kconfig` and can only be configured through the command `idf.py menuconfig`. Direct modification of this header file will be overwritten.

```c
/*You can select the GPIO corresponding to the LED in idf.py menuconfig,
and the modification result of menuconfig is that the value of CONFIG_BLINK
_GPIO will be changed. You can also directly modify the macro definition
here, and change CONFIG_BLINK_GPIO to a fixed value.*/
#define BLINK_GPIO CONFIG_BLINK_GPIO
void app_main(void)
{
    /*Configure IO as the GPIO default function, enable pull-up mode, and
    disable input and output modes*/
    gpio_reset_pin(BLINK_GPIO);
    /*Set GPIO to output mode*/
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
    while(1) {
        /*Print log*/
        printf("Turning off the LED\n");
        /*Turn off the LED (output low level)*/
        gpio_set_level(BLINK_GPIO, 0);
        /*Delay (1000 ms)*/
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        printf("Turning on the LED\n");
        /*Turn on the LED (output high level)*/
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

The `app_main()` function in the Blink example program serves as the entry point for user programs. It is a simple function with no parameters and no return value. This function is called after the system has completed initialisation, which includes tasks such as initialising the log serial port, configuring single/dual core, and configuring the watchdog.

The `app_main()` function runs in the context of a task named `main`. The stack size and priority of this task can be adjusted in `menuconfig` → `Componentconfig` → `Common ESP-related`.

For simple tasks like blinking an LED, all the necessary code can be implemented directly in the `app_main()` function. This typically involves initialising the GPIO corresponding to the LED and using a `while(1)` loop to toggle the LED on and off. Alternatively, you can use FreeRTOS API to create a new task that handles the LED blinking. Once the new task is successfully created, you can exit the `app_main()` function.

The content of `main/CMakeLists.txt` file, which guides the compilation process for the main component, is as follows:

```
idf_component_register(SRCS "blink.c" INCLUDE_DIRS "." )
```

Among them, `main/CMakeLists.txt` only calls one compilation system function, that is `idf_component_register`. Similar to the `CMakeLists.txt` for most other components, `blink.c` is added to `SRCS`, and the source files added to `SRCS` will be compiled. At the same time, "`.`", which represents the path where `CMakeLists.txt` is located, should be added to `INCLUDE_DIRS` as the search directories for header files. The content of `CMakeLists.txt` is as follows:

```
#Specify v3.5 as the oldest CMake version supported by the current project
#Versions lower than v3.5 must be upgraded before compilation continues
cmake_minimum_required(VERSION 3.5)
#Include the default CMake configuration of the ESP-IDF compilation system
```

```
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
#Create a project named "blink"
project(myProject)
```

Among them, the `CMakeLists.txt` in the root directory mainly includes `$ENV{IDF_PATH}/tools/cmake/project.cmake`, which is the main CMake configuration file provided by ESP-IDF. It is used to configure the default rules of the ESP-IDF compilation system and define common functions such as `idf_component_register`; `project(blink)` creates a project called `blink`, and the final firmware will be named `blink.bin`.

4.4.2 Compiling the Blink Program

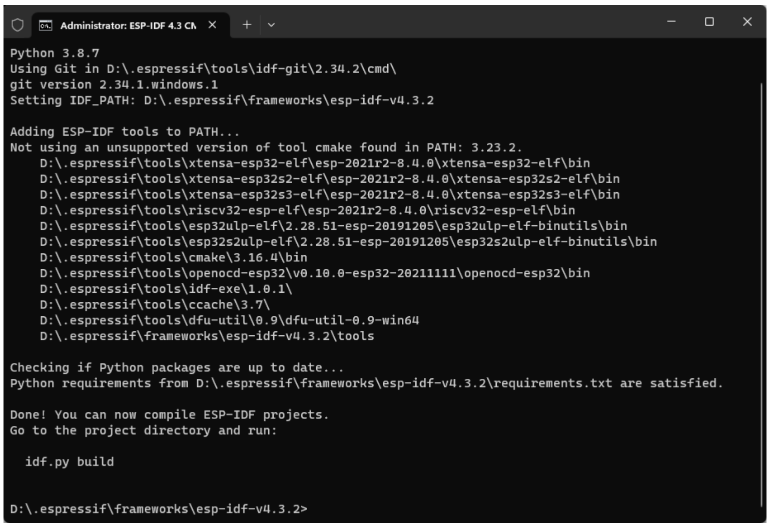
This section takes the Blink program as an example to demonstrate the compilation process of a simple ESP-IDF program. It is important to note that this section uses the high/low level of GPIO to drive the LED. However, the WS2812 indicator light requires a special communication protocol. You can refer to the example program in `esp-idf/examples/peripherals/rmt/led_strip` for more information.

1. Open a new terminal and import the ESP-IDF environment variables

For Linux and Mac systems, use `cd ~/esp/esp-idf` to navigate to the ESP-IDF folder. Then, import the ESP-IDF environment variables using the command `. ./export.sh`. This process also performs a complete integrity check of the development environment.

TIPS

Please note that the **dot** before the space should not be omitted in **`. ./export.sh`**. The dot is equivalent to the `source` directive, which refers to executing the script and changing the environment variables in the current shell.



```
Python 3.8.7
Using Git in D:\.espressif\tools\idf-git\2.34.2\cmd\
git version 2.34.1.windows.1
Setting IDF_PATH: D:\.espressif\frameworks\esp-idf-v4.3.2

Adding ESP-IDF tools to PATH...
Not using an unsupported version of tool cmake found in PATH: 3.23.2.
    D:\.espressif\tools\xtensa-esp32-elf\esp-2021r2-8.4.0\xtensa-esp32-elf\bin
    D:\.espressif\tools\xtensa-esp32s2-elf\esp-2021r2-8.4.0\xtensa-esp32s2-elf\bin
    D:\.espressif\tools\xtensa-esp32s3-elf\esp-2021r2-8.4.0\xtensa-esp32s3-elf\bin
    D:\.espressif\tools\riscv32-esp-elf\esp-2021r2-8.4.0\riscv32-esp-elf\bin
    D:\.espressif\tools\esp32ulp-elf\2.28.51-esp-20191205\esp32ulp-elf-binutils\bin
    D:\.espressif\tools\esp32s2ulp-elf\2.28.51-esp-20191205\esp32s2ulp-elf-binutils\bin
    D:\.espressif\tools\cmake\3.16.4\bin
    D:\.espressif\tools\openocd-esp32\v0.10.0-esp32-20211111\openocd-esp32\bin
    D:\.espressif\tools\idf-exe\1.0.1\
    D:\.espressif\tools\ccache\3.7\
    D:\.espressif\tools\dfu-util\0.9\dfu-util-0.9-win64
    D:\.espressif\frameworks\esp-idf-v4.3.2\tools

Checking if Python packages are up to date...
Python requirements from D:\.espressif\frameworks\esp-idf-v4.3.2\requirements.txt are satisfied.

Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build


D:\.espressif\frameworks\esp-idf-v4.3.2>
```

Figure 4.16. Automatic addition of environment variables in Windows system

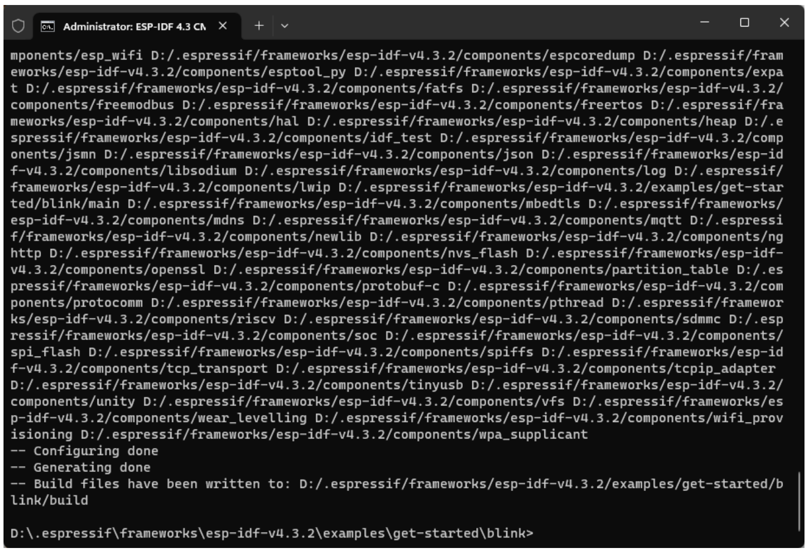
For Windows systems, you can directly find and open ESP-IDF 4.3 CMD or ESP-IDF 4.3 PowerShell in the program list. After the terminal is opened, the environment variables will be automatically added, as shown in Figure 4.16.

2. Navigate to the root directory of the `blink` project

Before compiling the project, navigate to the root directory of the project. To do this, use the command `cd examples/get-started/blink`.

3. Set the compilation target to ESP32-C3

Use the command `idf.py set-target esp32c3` to set the compilation target to ESP32-C3, as shown in Figure 4.17. If this step is skipped, the compilation target defaults to ESP32.



```
mponents/esp_wifi D:/.espressif/frameworks/esp-idf-v4.3.2/components/espcoredump D:/.espressif/fram
eworks/esp-idf-v4.3.2/components/esptool_py D:/.espressif/frameworks/esp-idf-v4.3.2/components/expa
t D:/.espressif/frameworks/esp-idf-v4.3.2/components/fatfs D:/.espressif/frameworks/esp-idf-v4.3.2/
components/freemodbus D:/.espressif/frameworks/esp-idf-v4.3.2/components/freertos D:/.espressif/fra
meworks/esp-idf-v4.3.2/components/hal D:/.espressif/frameworks/esp-idf-v4.3.2/components/heap D:/.e
spressif/frameworks/esp-idf-v4.3.2/components/idf_test D:/.espressif/frameworks/esp-idf-v4.3.2/comp
onents/jsmn D:/.espressif/frameworks/esp-idf-v4.3.2/components/json D:/.espressif/frameworks/esp-id
f-v4.3.2/components/libsodium D:/.espressif/frameworks/esp-idf-v4.3.2/components/log D:/.espressif/
frameworks/esp-idf-v4.3.2/components/lwip D:/.espressif/frameworks/esp-idf-v4.3.2/examples/get-star
ted/blink/main D:/.espressif/frameworks/esp-idf-v4.3.2/components/mbedtls D:/.espressif/frameworks/
esp-idf-v4.3.2/components/mdns D:/.espressif/frameworks/esp-idf-v4.3.2/components/mqtt D:/.espressi
f/frameworks/esp-idf-v4.3.2/components/newlib D:/.espressif/frameworks/esp-idf-v4.3.2/components/ng
http D:/.espressif/frameworks/esp-idf-v4.3.2/components/nvs_flash D:/.espressif/frameworks/esp-idf-
v4.3.2/components/openssl D:/.espressif/frameworks/esp-idf-v4.3.2/components/partition_table D:/.es
pressif/frameworks/esp-idf-v4.3.2/components/protobuf-c D:/.espressif/frameworks/esp-idf-v4.3.2/com
ponents/protocomm D:/.espressif/frameworks/esp-idf-v4.3.2/components/pthread D:/.espressif/framewor
ks/esp-idf-v4.3.2/components/riscv D:/.espressif/frameworks/esp-idf-v4.3.2/components/sdmmc D:/.esp
ressif/frameworks/esp-idf-v4.3.2/components/soc D:/.espressif/frameworks/esp-idf-v4.3.2/components/
spi_flash D:/.espressif/frameworks/esp-idf-v4.3.2/components/spiffs D:/.espressif/frameworks/esp-id
f-v4.3.2/components/tcp_transport D:/.espressif/frameworks/esp-idf-v4.3.2/components/tcpip_adapter
D:/.espressif/frameworks/esp-idf-v4.3.2/components/tinyusb D:/.espressif/frameworks/esp-idf-v4.3.2/
components/unity D:/.espressif/frameworks/esp-idf-v4.3.2/components/vfs D:/.espressif/frameworks/es
p-idf-v4.3.2/components/wear_levelling D:/.espressif/frameworks/esp-idf-v4.3.2/components/wifi_prov
isioning D:/.espressif/frameworks/esp-idf-v4.3.2/components/wpa_supplicant
-- Configuring done
-- Generating done
-- Build files have been written to: D:/.espressif/frameworks/esp-idf-v4.3.2/examples/get-started/b
link/build

D:\.espressif\frameworks\esp-idf-v4.3.2\examples\get-started\blink>
```

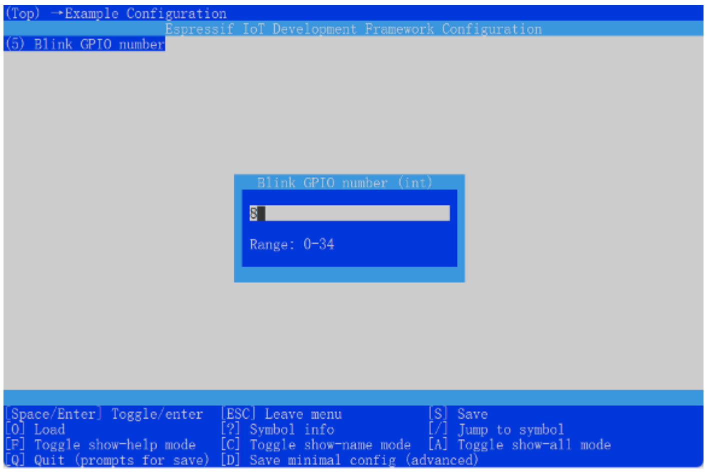
Figure 4.17. Set the compilation target to ESP32-C3

4. Configure GPIOs

Use the command `idf.py menuconfig` to enter the configuration interface. Navigate using the up/down keys and press Enter key to enter the `Example Configuration`. Enter a number to change the GPIO to the specified pin, as shown in Figure 4.18. Save the configuration by following the prompts.

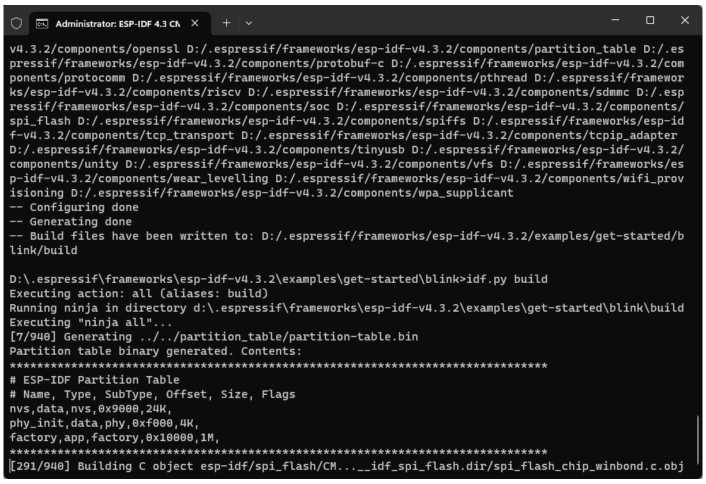
5. Build the code

Use the command `idf.py build` to build the code. The code building process is shown in Figure 4.19. Relevant prompts and flash commands will be printed once the build is complete, as shown in Figure 4.20.



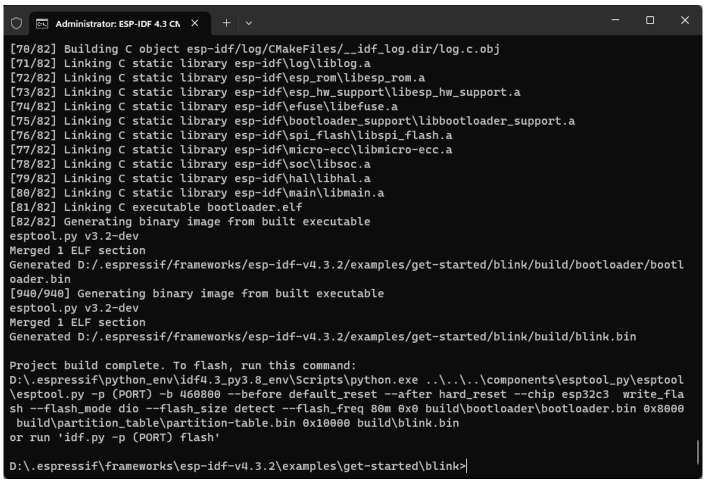
```
(Top) →Example Configuration
                    Espressif IoT Development Framework Configuration
(5) Blink GPIO number

                         Blink GPIO number (int)
                        5
                        Range: 0-34

[Space/Enter] Toggle/enter  [ESC] Leave menu           [S] Save
[O] Load                    [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Figure 4.18. Configure GPIO using `menuconfig`



```
v4.3.2/components/openssl D:/.espressif/frameworks/esp-idf-v4.3.2/components/partition_table D:/.es
pressif/frameworks/esp-idf-v4.3.2/components/protobuf-c D:/.espressif/frameworks/esp-idf-v4.3.2/com
ponents/protocomm D:/.espressif/frameworks/esp-idf-v4.3.2/components/pthread D:/.espressif/framewor
ks/esp-idf-v4.3.2/components/riscv D:/.espressif/frameworks/esp-idf-v4.3.2/components/sdmmc D:/.esp
ressif/frameworks/esp-idf-v4.3.2/components/soc D:/.espressif/frameworks/esp-idf-v4.3.2/components/
spi_flash D:/.espressif/frameworks/esp-idf-v4.3.2/components/spiffs D:/.espressif/frameworks/esp-id
f-v4.3.2/components/tcp_transport D:/.espressif/frameworks/esp-idf-v4.3.2/components/tcpip_adapter
D:/.espressif/frameworks/esp-idf-v4.3.2/components/tinyusb D:/.espressif/frameworks/esp-idf-v4.3.2/
components/unity D:/.espressif/frameworks/esp-idf-v4.3.2/components/vfs D:/.espressif/frameworks/es
p-idf-v4.3.2/components/wear_levelling D:/.espressif/frameworks/esp-idf-v4.3.2/components/wifi_prov
isioning D:/.espressif/frameworks/esp-idf-v4.3.2/components/wpa_supplicant
-- Configuring done
-- Generating done
-- Build files have been written to: D:/.espressif/frameworks/esp-idf-v4.3.2/examples/get-started/b
link/build

D:\.espressif\frameworks\esp-idf-v4.3.2\examples\get-started\blink>idf.py build
Executing action: all (aliases: build)
Running ninja in directory d:\.espressif\frameworks\esp-idf-v4.3.2\examples\get-started\blink\build
Executing "ninja all"...
[7/940] Generating ../../partition_table/partition-table.bin
Partition table binary generated. Contents:
*******************************************************************************
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs,data,nvs,0x9000,24K,
phy_init,data,phy,0xf000,4K,
factory,app,factory,0x10000,1M,
*******************************************************************************
[291/940] Building C object esp-idf/spi_flash/CM...__idf_spi_flash.dir/spi_flash_chip_winbond.c.obj
```

Figure 4.19. Code compilation process



```
[70/82] Building C object esp-idf/log/CMakeFiles/__idf_log.dir/log.c.obj
[71/82] Linking C static library esp-idf\log\liblog.a
[72/82] Linking C static library esp-idf\esp_rom\libesp_rom.a
[73/82] Linking C static library esp-idf\esp_hw_support\libesp_hw_support.a
[74/82] Linking C static library esp-idf\efuse\libefuse.a
[75/82] Linking C static library esp-idf\bootloader_support\libbootloader_support.a
[76/82] Linking C static library esp-idf\spi_flash\libspi_flash.a
[77/82] Linking C static library esp-idf\micro-ecc\libmicro-ecc.a
[78/82] Linking C static library esp-idf\soc\libsoc.a
[79/82] Linking C static library esp-idf\hal\libhal.a
[80/82] Linking C static library esp-idf\main\libmain.a
[81/82] Linking C executable bootloader.elf
[82/82] Generating binary image from built executable
esptool.py v3.2-dev
Merged 1 ELF section
Generated D:/.espressif/frameworks/esp-idf-v4.3.2/examples/get-started/blink/build/bootloader/bootl
oader.bin
[940/940] Generating binary image from built executable
esptool.py v3.2-dev
Merged 1 ELF section
Generated D:/.espressif/frameworks/esp-idf-v4.3.2/examples/get-started/blink/build/blink.bin

Project build complete. To flash, run this command:
D:\.espressif\python_env\idf4.3_py3.8_env\Scripts\python.exe ..\..\..\components\esptool_py\esptool
\esptool.py -p (PORT) -b 460800 --before default_reset --after hard_reset --chip esp32c3  write_fla
sh --flash_mode dio --flash_size detect --flash_freq 80m 0x0 build\bootloader\bootloader.bin 0x8000
 build\partition_table\partition-table.bin 0x10000 build\blink.bin
or run 'idf.py -p (PORT) flash'

D:\.espressif\frameworks\esp-idf-v4.3.2\examples\get-started\blink>
```

Figure 4.20. Prompt after code compilation is complete

4.4.3 Flashing the Blink Program

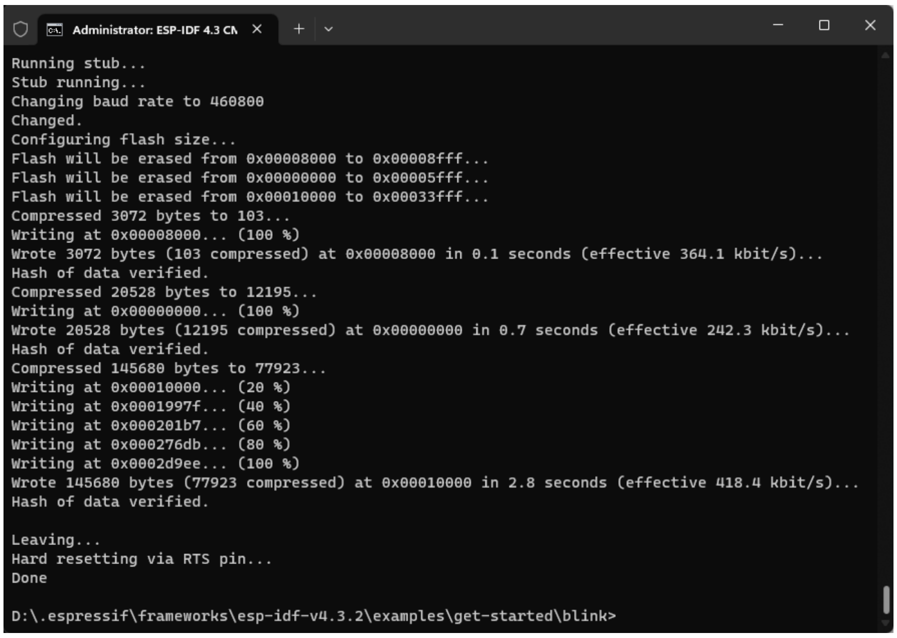
For Linux systems, connect the ESP32-C3 to the computer via USB-UART chip (such as CP2102), and use the command `ls /dev/ttyUSB*` to view the serial port number. If the current serial port number printed is `/dev/ttyUSB0`, use the command `idf.py -p /dev/ttyUSB0 flash` to flash the program onto the ESP32-C3.

For Mac systems, connect the ESP32-C3 to the computer via USB-UART chip (such as CP2102), and use the command `ls /dev/cu.*` to view the serial port number. If the current serial port number printed is `/dev/cu.SLAB_USBtoUART`, use the command `idf.py -p /dev/cu.SLAB_USBtoUART flash` to flash the program onto the ESP32-C3.

For Windows systems, connect the ESP32-C3 to the computer via USB-UART chip (such as CP2102), and view the serial port number through the device manager. If the current serial port number is `COM5`, use the command `idf.py -p COM5 flash` to flash the program onto the ESP32-C3.

After the flashing process is completed, you will see a prompt as shown in Figure 4.21 in the console. When the following log appears, the code will start executing, and the LED on the development board will start flashing.

```
Hard resetting via RTS pin...
Done
```



```
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Flash will be erased from 0x00008000 to 0x00008fff...
Flash will be erased from 0x00000000 to 0x00005fff...
Flash will be erased from 0x00010000 to 0x00033fff...
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.1 seconds (effective 364.1 kbit/s)...
Hash of data verified.
Compressed 20528 bytes to 12195...
Writing at 0x00000000... (100 %)
Wrote 20528 bytes (12195 compressed) at 0x00000000 in 0.7 seconds (effective 242.3 kbit/s)...
Hash of data verified.
Compressed 145680 bytes to 77923...
Writing at 0x00010000... (20 %)
Writing at 0x0001997f... (40 %)
Writing at 0x000201b7... (60 %)
Writing at 0x000276db... (80 %)
Writing at 0x0002d9ee... (100 %)
Wrote 145680 bytes (77923 compressed) at 0x00010000 in 2.8 seconds (effective 418.4 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Done

D:\.espressif\frameworks\esp-idf-v4.3.2\examples\get-started\blink>
```

Figure 4.21. Prompt in the console after flashing is completed

4.4.4 Serial Port Log Analysis of the Blink Program

Once the firmware compilation and download are completed, navigate to the project folder, and run the command `idf.py monitor`. This will open a monitor with coloured font. The monitor will output the serial port log of the target ESP32-C3. The content is divided into three parts by default: **first-level bootloader information**, **second-level bootloader information**, and **user program output**. During the output of log, you can press the `Ctrl+]` key combination to exit the log output.

```
ESP-ROM:esp32c3-api1-20210207
Build:Feb  7 2021
rst:0x1 (POWERON),boot:0xc (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fcd6100,len:0x1798
load:0x403ce000,len:0x8dc
load:0x403d0000,len:0x2984
entry 0x403ce000
```

First-level bootloader information

By default, the first-level bootloader information is output from UART and cannot be turned off through configuration in ESP-IDF version 4.3.2. This information includes the ROM code version information fixed internally in the chip. Different chips in the same series may have different ROM code versions due to ROM repairs and feature expansions. It also includes the reason for the chip restart, such as `rst:0x1` indicating power-on restart of the chip, `rst:0x3` indicating software-triggered restart, `rst:0x4` indicating software exception restart, etc. You can use this information to assess the status of the chip. Additionally, it provides details about the chip's boot mode, such as `boot:0xc` indicating SPI Flash Boot mode (normal operation mode, in which the code in flash is loaded and executed), and `boot:0x4` indicating Flash Download mode, in which the content of flash can be erased and programmed.

Second-level bootloader information

The output of second-level bootloader information can be disabled by setting `menuconfig (Top)` → `Bootloader config` → `Bootloader log verbosity` to `No output`.

This information mainly includes the ESP-IDF version, flash operating mode and speed, system partition and stack allocation, as well as the application name and version.

```
I (30) boot: ESP-IDF v4.3.2-1-g887e7c0c73-dirty 2nd stage bootloader
I (30) boot: compile time 18:27:35
I (30) boot: chip revision: 3
I (34) boot.esp32c3: SPI Speed      : 80MHz
```

```
I (38) boot.esp32c3: SPI Mode       : DIO
I (43) boot.esp32c3: SPI Flash Size : 2MB
I (48) boot: Enabling RNG early entropy source...
I (53) boot: Partition Table:
I (57) boot: ## Label            Usage          Type ST Offset   Length
I (64) boot:  0 nvs              WiFi data      01 02 00009000 00006000
I (72) boot:  1 phy_init         RF data        01 01 0000f000 00001000
I (79) boot:  2 factory          factory app    00 00 00010000 00100000
I (86) boot: End of partition table
I (91) esp_image: segment 0: paddr=00010020 vaddr=3c020020 size=06058h (24664) map
I (103) esp_image: segment 1: paddr=00016080 vaddr=3fc89c00 size=01a88h (6792) load
I (109) esp_image: segment 2: paddr=00017b10 vaddr=40380000 size=08508h (34056) load
I (122) esp_image: segment 3: paddr=00020020 vaddr=42000020 size=15c54h (89172) map
I (138) esp_image: segment 4: paddr=00035c7c vaddr=40388508 size=0157ch (5500) load
I (139) esp_image: segment 5: paddr=00037200 vaddr=50000000 size=00010h (16) load
I (147) boot: Loaded app from partition at offset 0x10000
I (150) boot: Disabling RNG early entropy source...
I (166) cpu_start: Pro cpu up.
I (179) cpu_start: Pro cpu start user code
I (179) cpu_start: cpu freq: 160000000
I (179) cpu_start: Application information:
I (182) cpu_start: Project name:   blink
I (186) cpu_start: App version:    v4.3.2-1-g887e7c0c73-dirty
I (193) cpu_start: Compile time:   Jan 26 2022 18:27:31
I (199) cpu_start: ELF file SHA256:  dadcae8e7bb964ab...
I (205) cpu_start: ESP-IDF:         v4.3.2-1-g887e7c0c73-dirty
I (212) heap_init: Initializing. RAM available for dynamic allocation:
I (219) heap_init: At 3FC8C4D0 len 00033B30 (206 KiB): DRAM
I (225) heap_init: At 3FCC0000 len 0001F060 (124 KiB): STACK/DRAM
I (232) heap_init: At 50000010 len 00001FF0 (7 KiB): RTCRAM
I (238) spi_flash: detected chip: generic
I (243) spi_flash: flash io: dio
W (247) spi_flash: Detected size(4096k) larger than the size in the binary image
header(2048k). Using the size in the binary image header.
I (260) sleep: Configure to isolate all GPIO pins in sleep state
I (267) sleep: Enable automatic switching of GPIO sleep configuration
I (274) cpu_start: Starting scheduler.
```

User program output

The user program output includes all information that is printed using the `printf()` function, which is the standard output function in the C language, or the `ESP_LOG()` function, which is a custom output function provided by ESP-IDF. It is recommended to use `ESP_LOG()` because it allows you to specify the log level for better organisation and filtering of logs.

You can configure which logs above a certain level are output through `menuconfig(Top)`

→ `Component config` → `Log output`. This allows you to control the verbosity of the logs and customise the level of detail that is displayed during runtime.

```
I (278) gpio: GPIO[5]| InputEn: 0| OutfgputEn: 0| OpenDrain: 0| Pullup: 1|
Pulldown: 0| Intr:0
Turning off the LED
Turning on the LED
Turning off the LED
Turning on the LED
```

In addition to log output, `idf.py monitor` can also parse system exceptions and trace software errors. For example, when the application crashes, the following register dump and traceback information will be generated:

```
Guru Meditation Error of type StoreProhibited occurred on core 0. Exception was
unhandled.
Register dump:
PC  : 0x400f360d  PS    : 0x00060330  A0    : 0x800dbf56  A1    : 0x3ffb7e00
A2  : 0x3ffb136c  A3    : 0x00000005  A4    : 0x00000000  A5    : 0x00000000
A6  : 0x00000000  A7    : 0x00000080  A8    : 0x00000000  A9    : 0x3ffb7dd0
A10 : 0x00000003  A11   : 0x00060f23  A12   : 0x00060f20  A13   : 0x3ffba6d0
A14 : 0x00000047  A15   : 0x0000000f  SAR   : 0x00000019  EXCCAUSE  : 0x0000001d
EXCVADDR: 0x00000000  LBEG : 0x4000c46c  LEND : 0x4000c477  LCOUNT  : 0x00000000

Backtrace: 0x400f360d:0x3ffb7e00 0x400dbf56:0x3ffb7e20 0x400dbf5e:0x3ffb7e40
0x400dbf82:0x3ffb7e60 0x400d071d:0x3ffb7e90
```

Based on the register address, the IDF monitor will query the compiled `ELF` file and trace the code call process when the application crashes, outputting the function call information to the monitor:

```
Guru Meditation Error of type StoreProhibited occurred on core 0. Exception was
unhandled.
Register dump:
PC  : 0x400f360d  PS    : 0x00060330  A0    : 0x800dbf56  A1    : 0x3ffb7e00
0x400f360d: do_something_to_crash at /home/gus/esp/32/idf/examples/get-started/
hello_world/main/./hello_world_main.c:57
(inlined by) inner_dont_crash at /home/gus/esp/32/idf/examples/get-started/hello
_world/main/./hello_world_main.c:52
A2  : 0x3ffb136c  A3    : 0x00000005  A4    : 0x00000000  A5    : 0x00000000
A6  : 0x00000000  A7    : 0x00000080  A8    : 0x00000000  A9    : 0x3ffb7dd0
A10 : 0x00000003  A11   : 0x00060f23  A12   : 0x00060f20  A13   : 0x3ffba6d0
A14 : 0x00000047  A15   : 0x0000000f  SAR   : 0x00000019  EXCCAUSE  : 0x0000001d
EXCVADDR: 0x00000000  LBEG : 0x4000c46c  LEND : 0x4000c477  LCOUNT  : 0x00000000

Backtrace: 0x400f360d:0x3ffb7e00 0x400dbf56:0x3ffb7e20 0x400dbf5e:0x3ffb7e40
0x400dbf82:0x3ffb7e60 0x400d071d:0x3ffb7e90
0x400f360d: do_something_to_crash at /home/gus/esp/32/idf/examples/get-started/
hello_world/main/./hello_world_main.c:57
```

```
(inlined by) inner_dont_crash at /home/gus/esp/32/idf/examples/get-started/hello
_world/main/./hello_world_main.c:52
0x400dbf56: still_dont_crash at /home/gus/esp/32/idf/examples/get-started/hello
_world/main/./hello_world_main.c:47
0x400dbf5e: dont_crash at /home/gus/esp/32/idf/examples/get-started/hello_world/
main/./hello_world_main.c:42
0x400dbf82: app_main at /home/gus/esp/32/idf/examples/get-started/hello_world/
main/./hello_world_main.c:33
0x400d071d: main_task at /home/gus/esp/32/idf/components/esp32/./cpu_start.c:254
```

The trace information of the monitor shows that the application crashes in the function `do_something_to_crash()`, which is called by the function `app_main()` → `dont_crash()` → `still_dont_crash()` → `inner_dont_crash()` → `do_something_to_crash()`. Based on this, the input/output parameters of each link can be checked to determine the cause of the crash.

4.5 Summary

In this chapter, we have covered the setup of the official software development environment, ESP-IDF, for ESP32-C3. We have introduced the code resources and file structure of ESP-IDF and provided a demonstration of the ESP-IDF project structure, compilation system, and related development tools using a simple example.

By following the instructions in this chapter, you can start developing with ESP-IDF for simple projects. However, for more specific and advanced compilation requirements, it is recommended to refer to both the ESP-IDF official documentation and the CMake official documentation.