CrowPanel Advanced 5inch ESP32-P4 HMI

www.elecrow.com

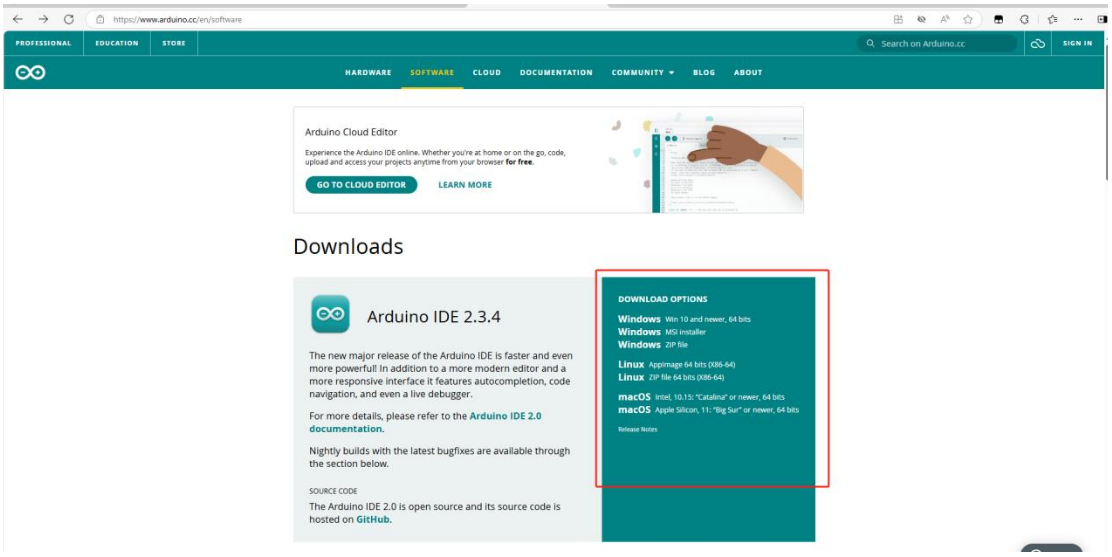
Preface

Installing Arduino IDE

1. Open the official Arduino website

https://www.arduino.cc/en/software

2. Select and install the appropriate version according to your operating system.



Arduino Cloud Editor

Experience the Arduino IDE online. Whether you're at home or on the go, code, upload and access your projects anytime from your browser **for free**.

Downloads

Arduino IDE 2.3.4

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger.

For more details, please refer to the **Arduino IDE 2.0 documentation**.

Nightly builds with the latest bugfixes are available through the section below.

SOURCE CODE

The Arduino IDE 2.0 is open source and its source code is hosted on **GitHub**.

DOWNLOAD OPTIONS

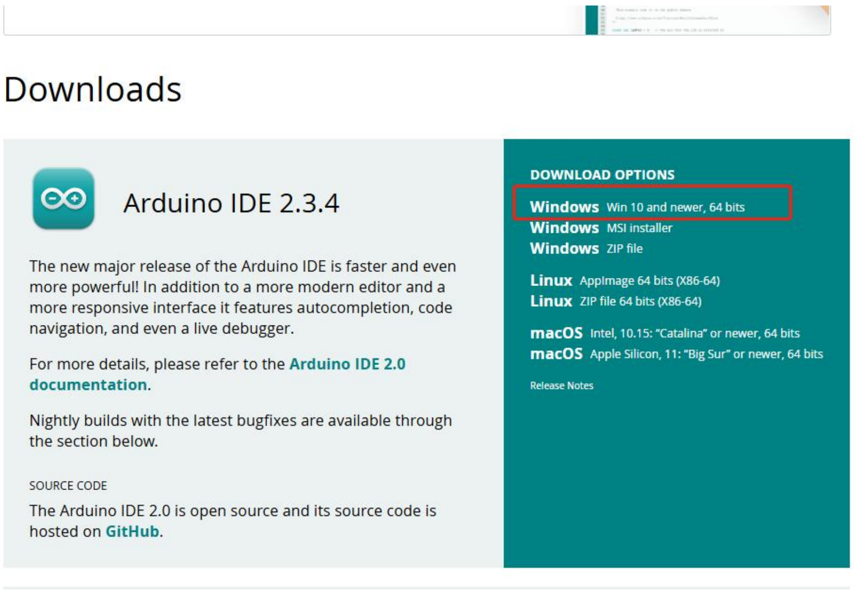
Windows Win 10 and newer, 64 bits
Windows MSI installer
Windows ZIP file

Linux AppImage 64 bits (X86-64)
Linux ZIP file 64 bits (X86-64)

macOS Intel, 10.15: "Catalina" or newer, 64 bits
macOS Apple Silicon, 11: "Big Sur" or newer, 64 bits

Release Notes

2.1 Here I choose the first one.


Downloads

Arduino IDE 2.3.4

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger.

For more details, please refer to the **Arduino IDE 2.0 documentation**.

Nightly builds with the latest bugfixes are available through the section below.

SOURCE CODE

The Arduino IDE 2.0 is open source and its source code is hosted on **GitHub**.

DOWNLOAD OPTIONS

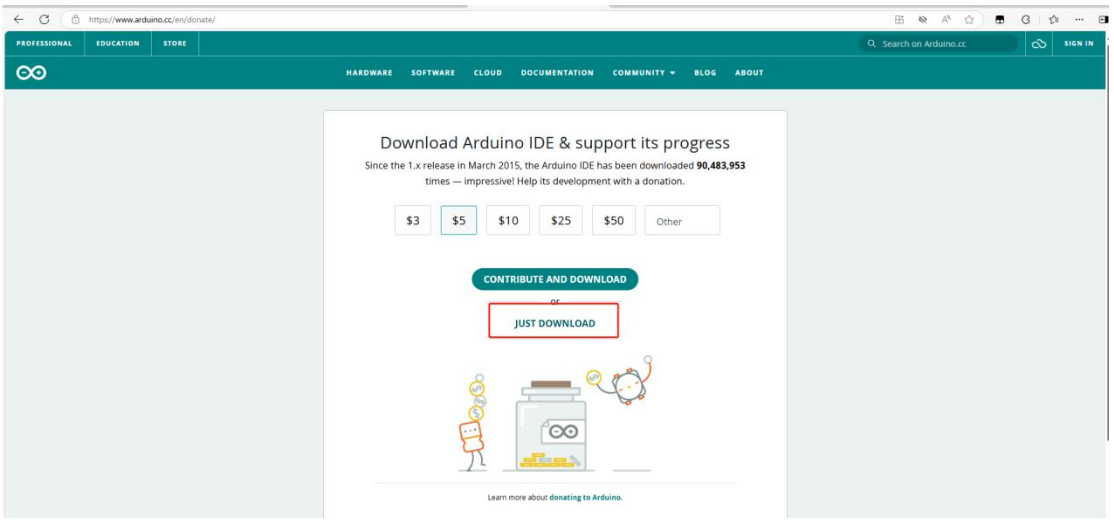
Windows Win 10 and newer, 64 bits
Windows MSI installer
Windows ZIP file

Linux AppImage 64 bits (X86-64)
Linux ZIP file 64 bits (X86-64)

macOS Intel, 10.15: "Catalina" or newer, 64 bits
macOS Apple Silicon, 11: "Big Sur" or newer, 64 bits

Release Notes

3. Select "Download Only" if you wish.

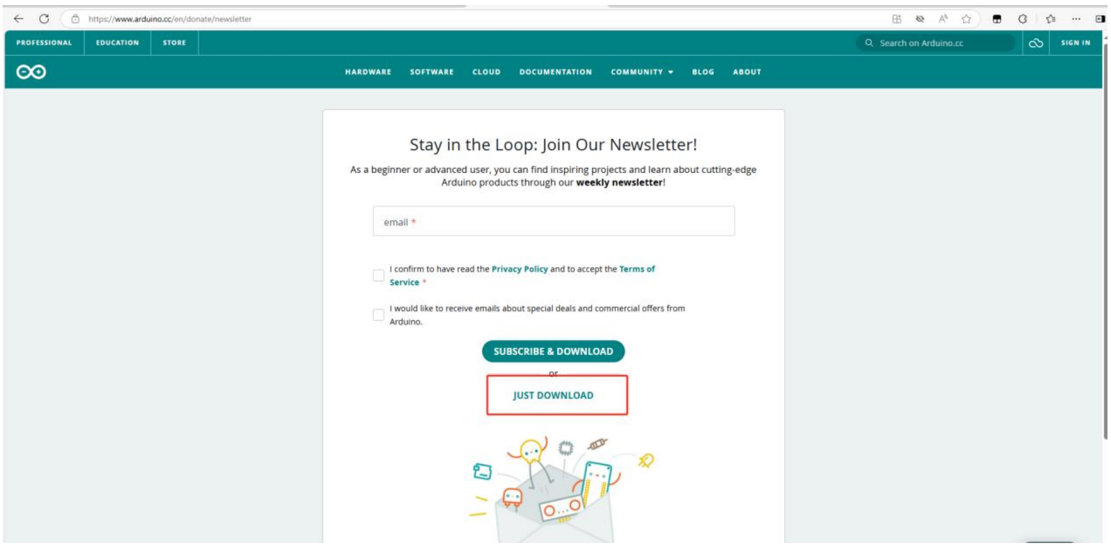

https://www.arduino.cc/en/donate/

Download Arduino IDE & support its progress

Since the 1.x release in March 2015, the Arduino IDE has been downloaded **90,483,953** times — impressive! Help its development with a donation.

Learn more about donating to Arduino.

3.3 The same as above.


https://www.arduino.cc/en/donate/newsletter

Stay in the Loop: Join Our Newsletter!

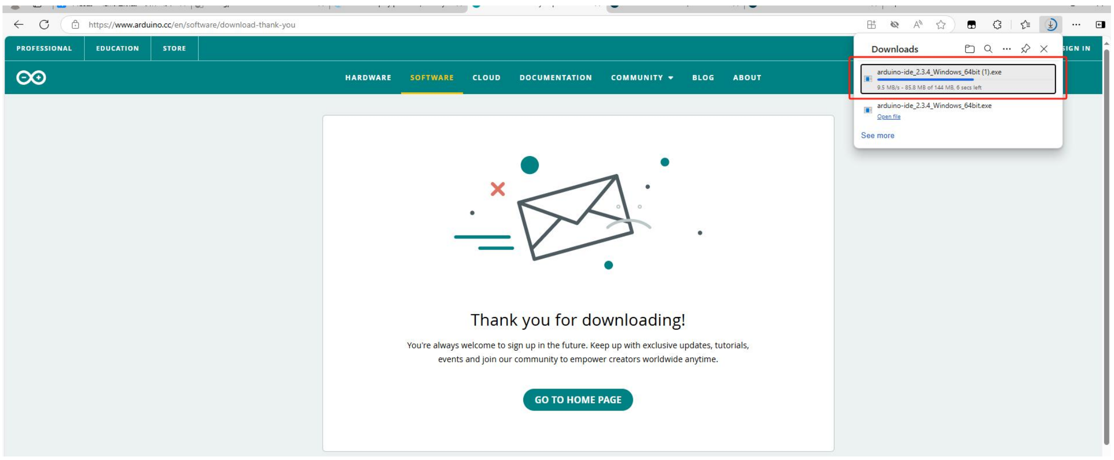
As a beginner or advanced user, you can find inspiring projects and learn about cutting-edge Arduino products through our **weekly newsletter**!

email *

I confirm to have read the Privacy Policy and to accept the Terms of Service *

I would like to receive emails about special deals and commercial offers from Arduino.

4. Wait for the download to complete


https://www.arduino.cc/en/software/download-thank-you

Downloads

arduino-ide_2.3.4_Windows_64bit (1).exe

9.5 MB/s - 85.8 MB of 144 MB, 6 secs left

arduino-ide_2.3.4_Windows_64bit.exe

Open file

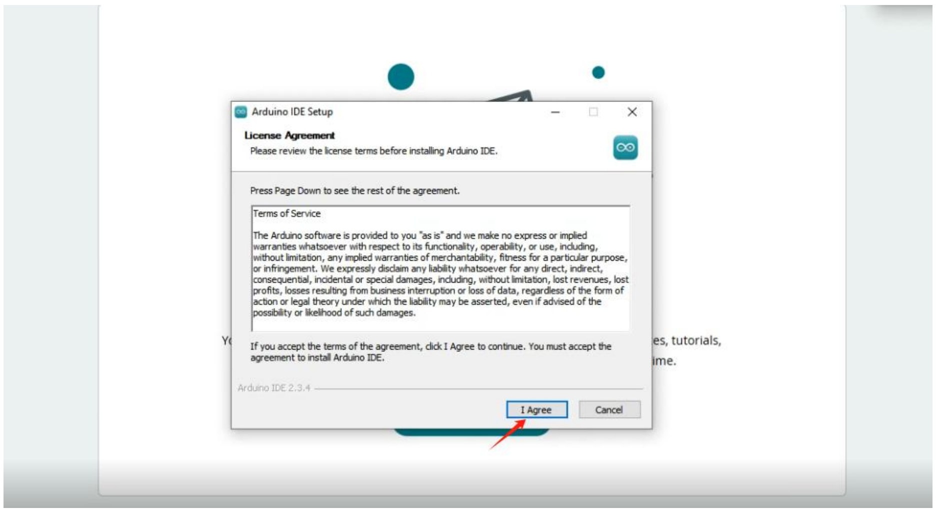
See more

Thank you for downloading!

You're always welcome to sign up in the future. Keep up with exclusive updates, tutorials, events and join our community to empower creators worldwide anytime.

5. Confirm the download process

5.1 Accept the options


Arduino IDE Setup

License Agreement

Please review the license terms before installing Arduino IDE.

Press Page Down to see the rest of the agreement.

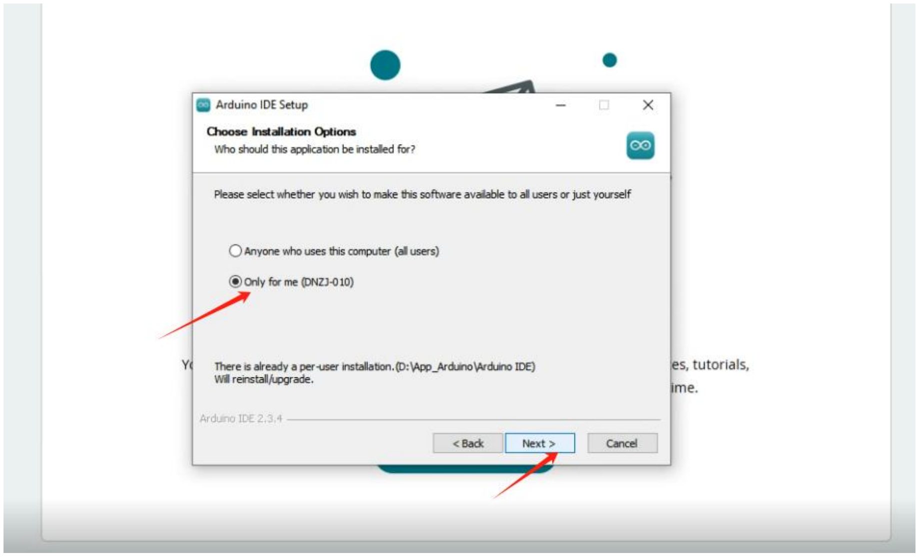
Terms of Service

The Arduino software is provided to you "as is" and we make no express or implied warranties whatsoever with respect to its functionality, operability, or use, including, without limitation, any implied warranties of merchantability, fitness for a particular purpose, or infringement. We expressly disclaim any liability whatsoever for any direct, indirect, consequential, incidental or special damages, including, without limitation, lost revenues, lost profits, losses resulting from business interruption or loss of data, regardless of the form of action or legal theory under which the liability may be asserted, even if advised of the possibility or likelihood of such damages.

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Arduino IDE.

Arduino IDE 2.3.4

5.2 As shown in the figure, determine


Arduino IDE Setup

Choose Installation Options

Who should this application be installed for?

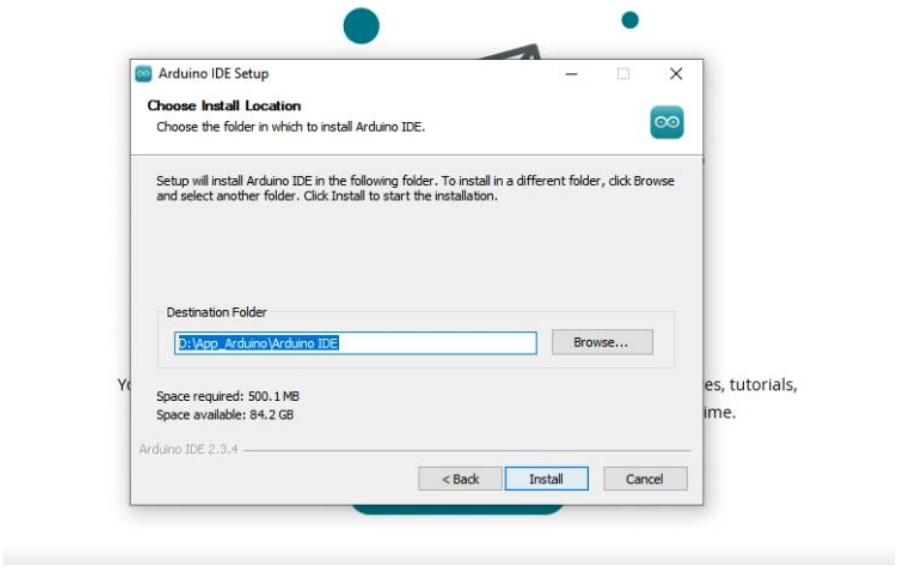
Please select whether you wish to make this software available to all users or just yourself

- ○ Anyone who uses this computer (all users)
- ◉ Only for me (DNZJ-010)

There is already a per-user installation.(D:\App_Arduino\Arduino IDE)
Will reinstall/upgrade.

Arduino IDE 2.3.4

5.3 Customize your path and click "Install"



Arduino IDE Setup

Choose Install Location

Choose the folder in which to install Arduino IDE.

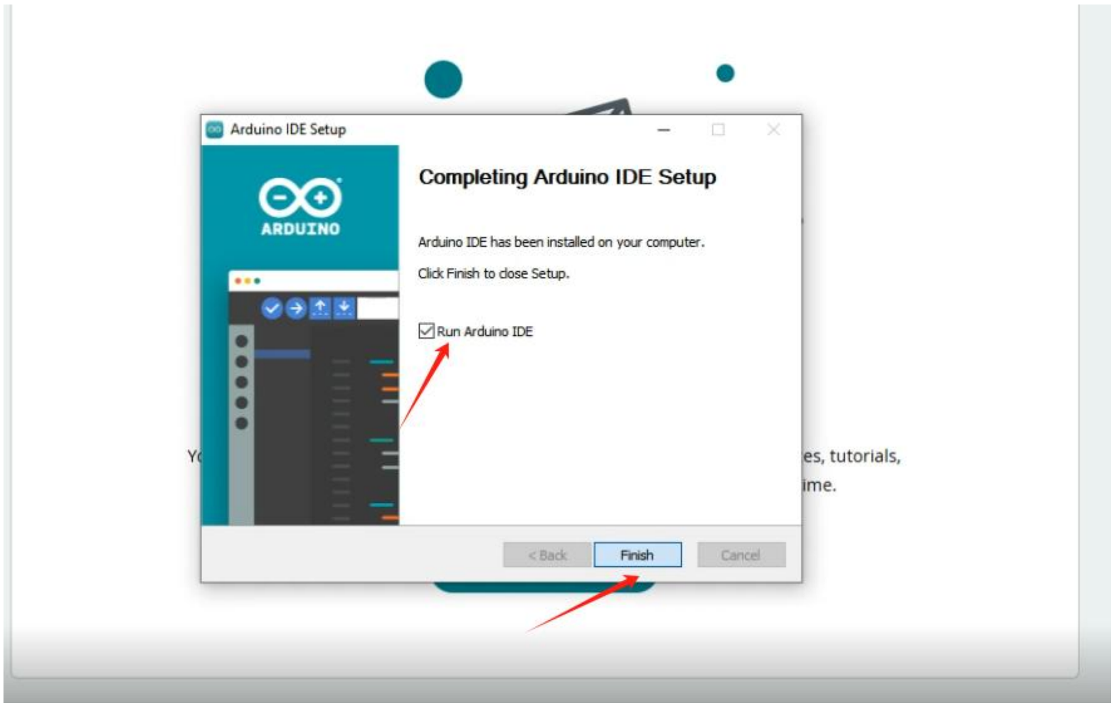
Setup will install Arduino IDE in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder: D:\App_Arduino\Arduino IDE

Space required: 500.1 MB
Space available: 84.2 GB

Arduino IDE 2.3.4

5.4 After the installation is completed, you can start using the Arduino IDE.



Arduino IDE Setup

Completing Arduino IDE Setup

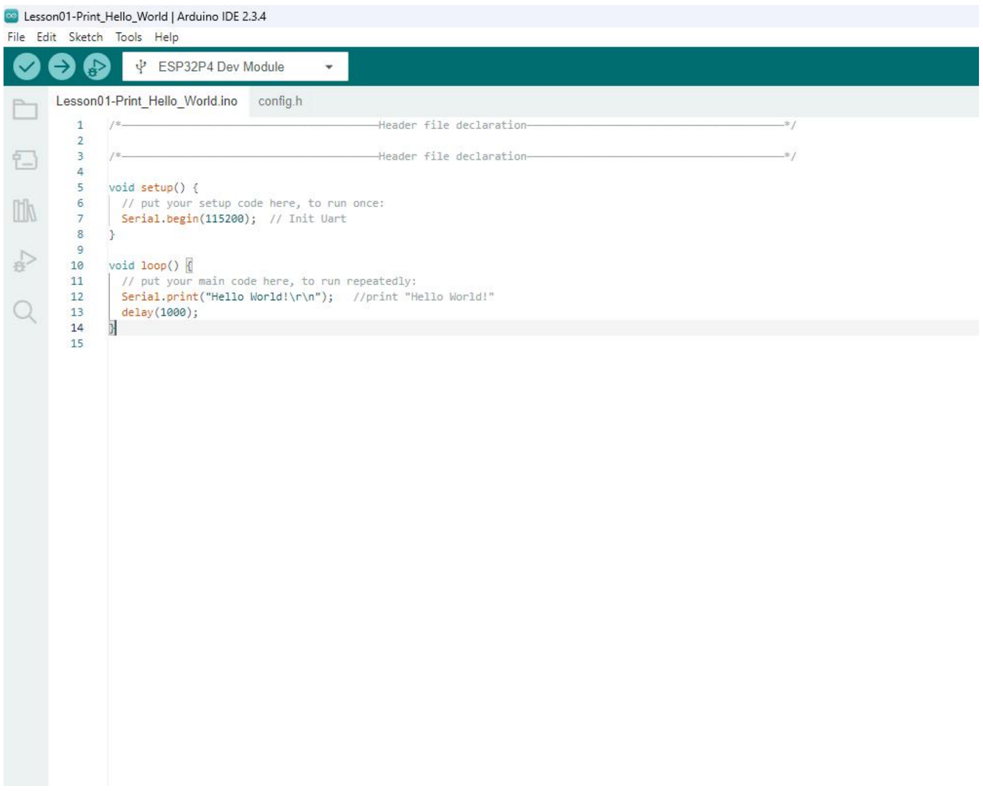
Arduino IDE has been installed on your computer.

Click Finish to close Setup.

☑ Run Arduino IDE

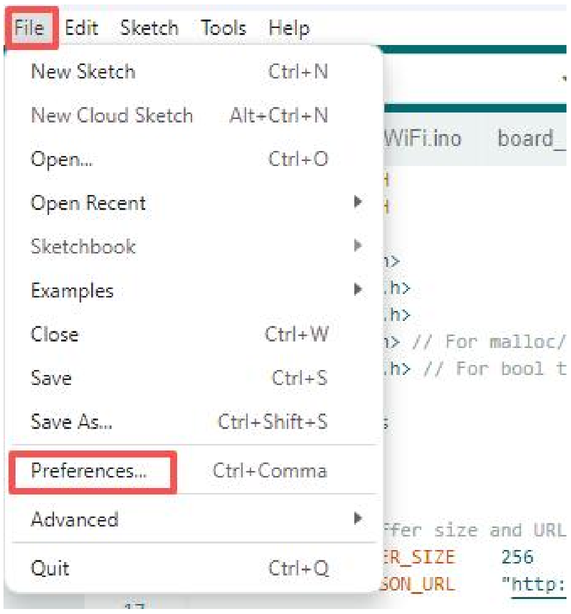
Installing the ESP32-P4 Development Board

After the installation is completed, open the Arduino IDE.
First, randomly open a project.

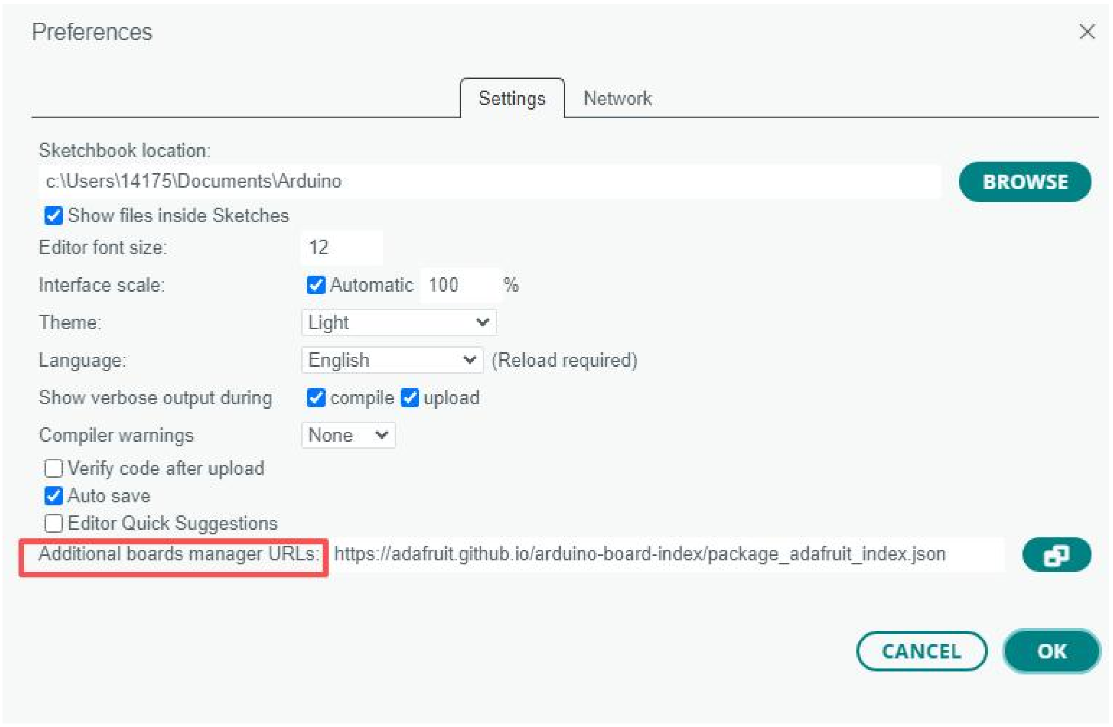

Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson01-Print_Hello_World.ino
config.h

```
/*————————————————————————————Header file declaration————————————————————————————*/

/*————————————————————————————Header file declaration————————————————————————————*/

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);  // Init Uart
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.print("Hello World!\r\n");   //print "Hello World!"
  delay(1000);
}
```

Go to "Preferences".



Configure "Additional Boards Manager URLs"



Preferences

Settings | Network

Sketchbook location:
c:\Users\14175\Documents\Arduino

☑ Show files inside Sketches

Editor font size: 12

Interface scale: ☑ Automatic 100 %

Theme: Light

Language: English (Reload required)

Show verbose output during ☑ compile ☑ upload

Compiler warnings None

☐ Verify code after upload

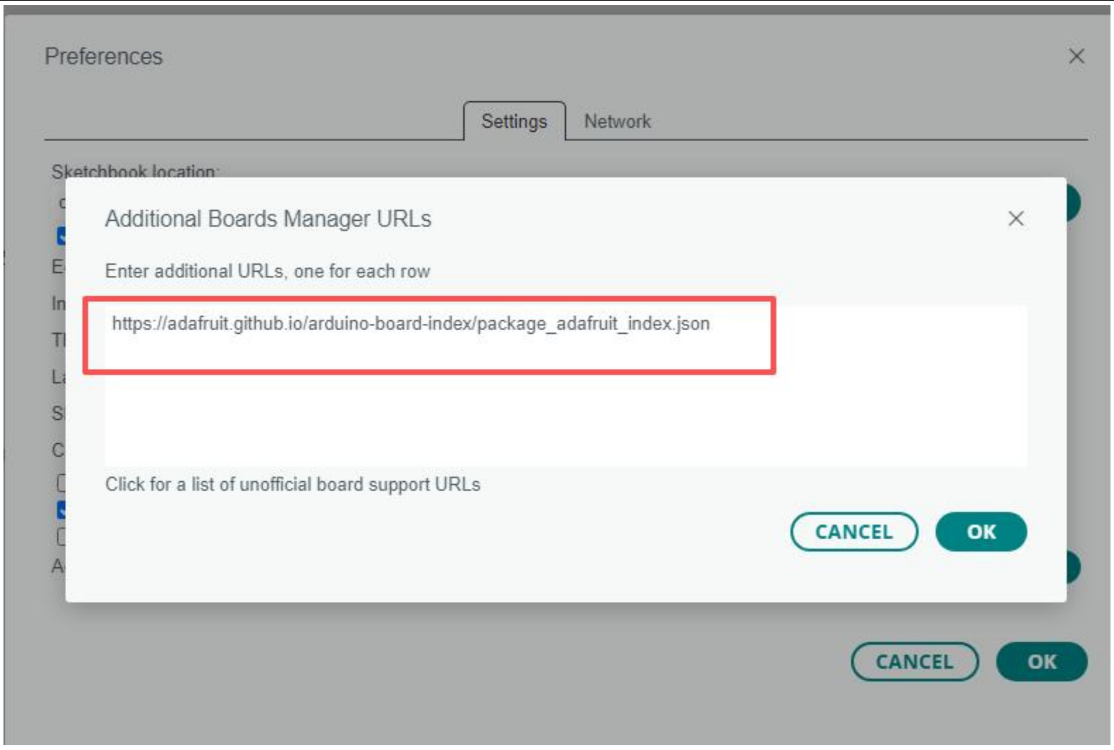
☑ Auto save

☐ Editor Quick Suggestions

Additional boards manager URLs: https://adafruit.github.io/arduino-board-index/package_adafruit_index.json

Add this in.

https://adafruit.github.io/arduino-board-index/package_adafruit_index.json



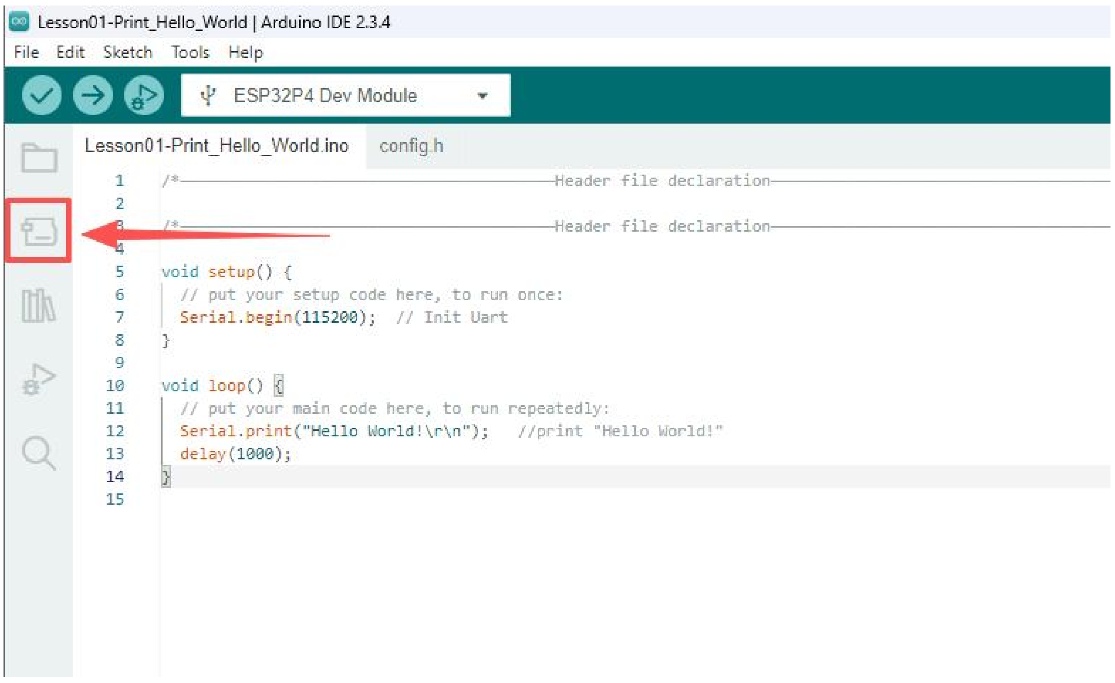
Additional Boards Manager URLs

Enter additional URLs, one for each row

https://adafruit.github.io/arduino-board-index/package_adafruit_index.json

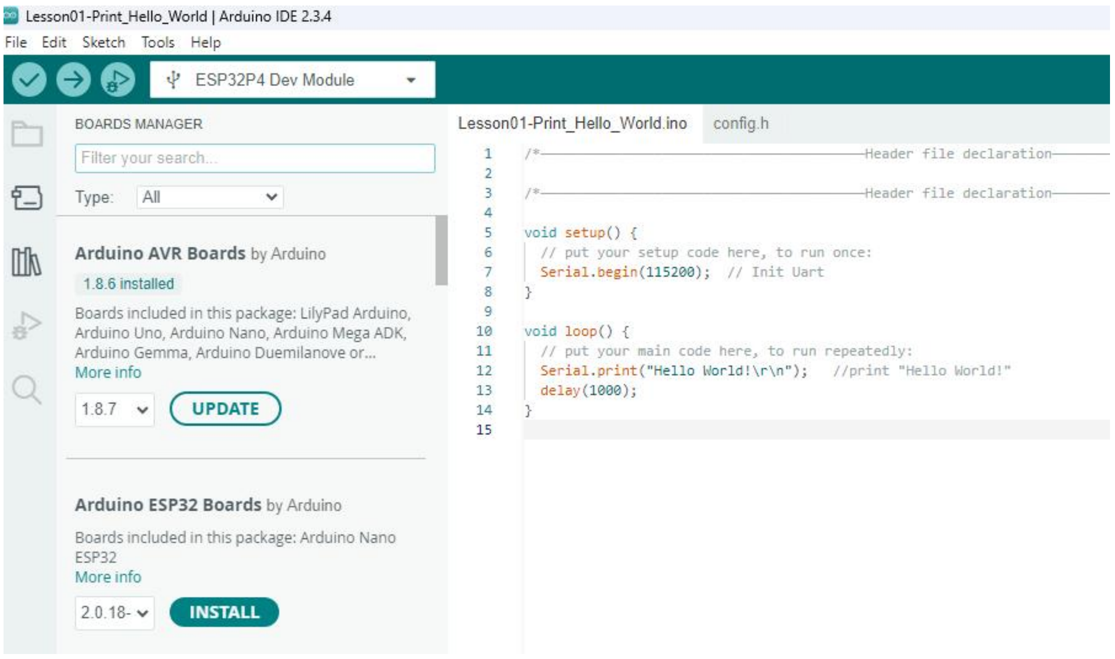
Click for a list of unofficial board support URLs

First, click on "board manage" on the left side.

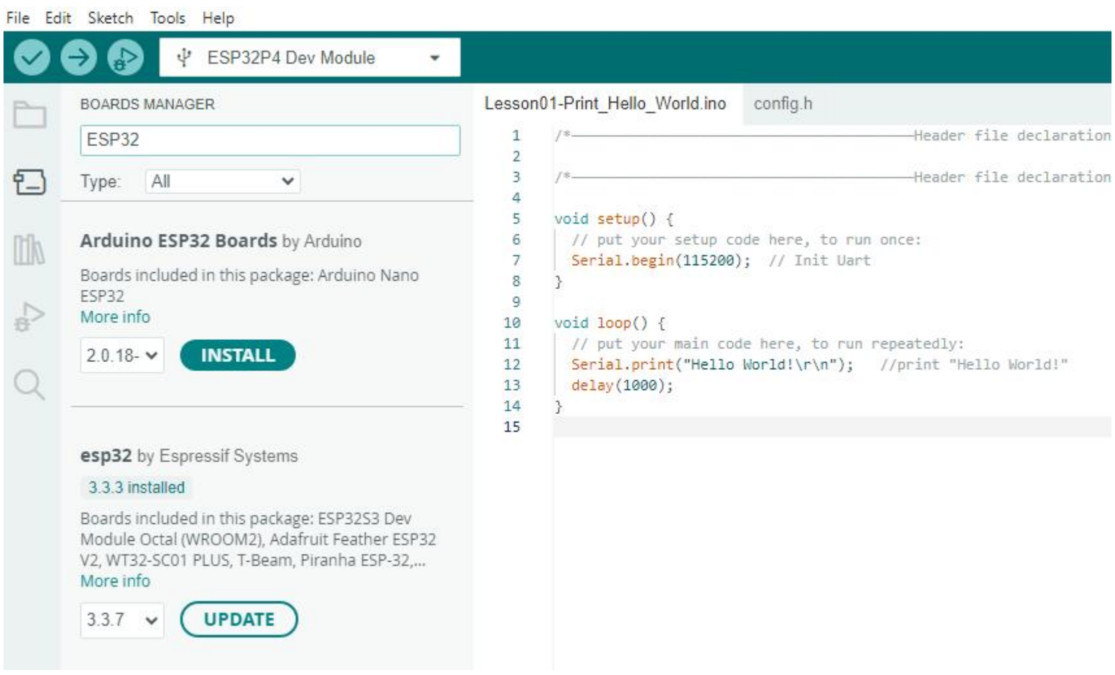

Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson01-Print_Hello_World.ino
config.h

```
/*————————————————————————Header file declaration————————————————

/*————————————————————————Header file declaration————————————————

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);  // Init Uart
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.print("Hello World!\r\n");    //print "Hello World!"
  delay(1000);
}
```

After opening

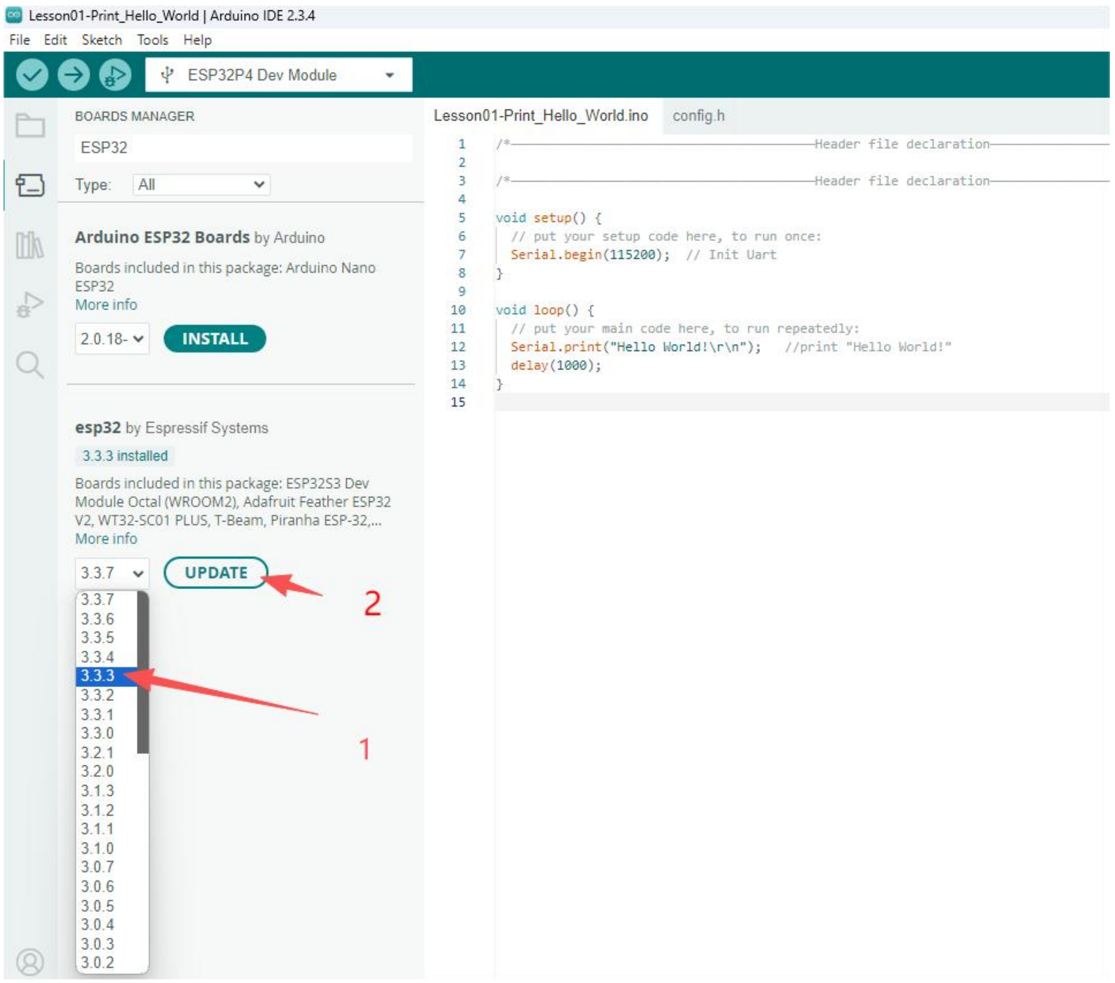

Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
BOARDS MANAGER
Filter your search...
Type: All
Arduino AVR Boards by Arduino
1.8.6 installed
Boards included in this package: LilyPad Arduino, Arduino Uno, Arduino Nano, Arduino Mega ADK, Arduino Gemma, Arduino Duemilanove or...
More info
1.8.7
UPDATE
Arduino ESP32 Boards by Arduino
Boards included in this package: Arduino Nano ESP32
More info
2.0.18-
INSTALL
Lesson01-Print_Hello_World.ino
config.h

Search for ESP32 here


File Edit Sketch Tools Help
ESP32P4 Dev Module
BOARDS MANAGER
ESP32
Type: All
Arduino ESP32 Boards by Arduino
Boards included in this package: Arduino Nano ESP32
More info
2.0.18-
INSTALL
esp32 by Espressif Systems
3.3.3 installed
Boards included in this package: ESP32S3 Dev Module Octal (WROOM2), Adafruit Feather ESP32 V2, WT32-SC01 PLUS, T-Beam, Piranha ESP-32,...
More info
3.3.7
UPDATE
Lesson01-Print_Hello_World.ino
config.h

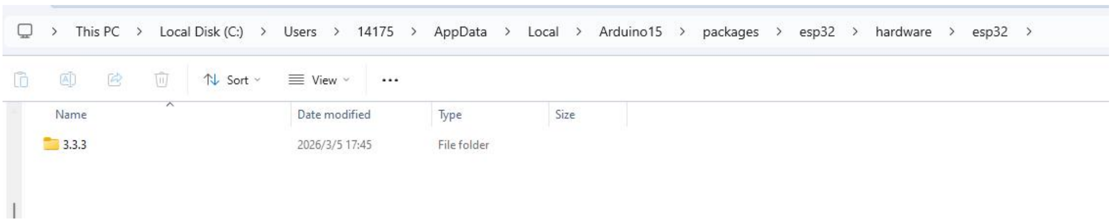
```
/*———————————————————————Header file declaration

/*———————————————————————Header file declaration

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);  // Init Uart
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.print("Hello World!\r\n");   //print "Hello World!"
  delay(1000);
}
```

Select version 3.3.3 here and proceed with the installation.


Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
BOARDS MANAGER
ESP32
Type: All
Arduino ESP32 Boards by Arduino
Boards included in this package: Arduino Nano ESP32
More info
2.0.18-
INSTALL
esp32 by Espressif Systems
3.3.3 installed
Boards included in this package: ESP32S3 Dev Module Octal (WROOM2), Adafruit Feather ESP32 V2, WT32-SC01 PLUS, T-Beam, Piranha ESP-32,...
More info
3.3.7
UPDATE
3.3.7
3.3.6
3.3.5
3.3.4
3.3.3
3.3.2
3.3.1
3.3.0
3.2.1
3.2.0
3.1.3
3.1.2
3.1.1
3.1.0
3.0.7
3.0.6
3.0.5
3.0.4
3.0.3
3.0.2
2
1
Lesson01-Print_Hello_World.ino
config.h

Note that all subsequent codes are developed based on the ESP32 V3.3.3 version. Make sure to keep the version number consistent with ours.

After the download is completed, you will be able to see the downloaded ESP32 version in this path of the file system.


This PC > Local Disk (C:) > Users > 14175 > AppData > Local > Arduino15 > packages > esp32 > hardware > esp32 >

Name	Date modified	Type	Size
3.3.3	2026/3/5 17:45	File folder	

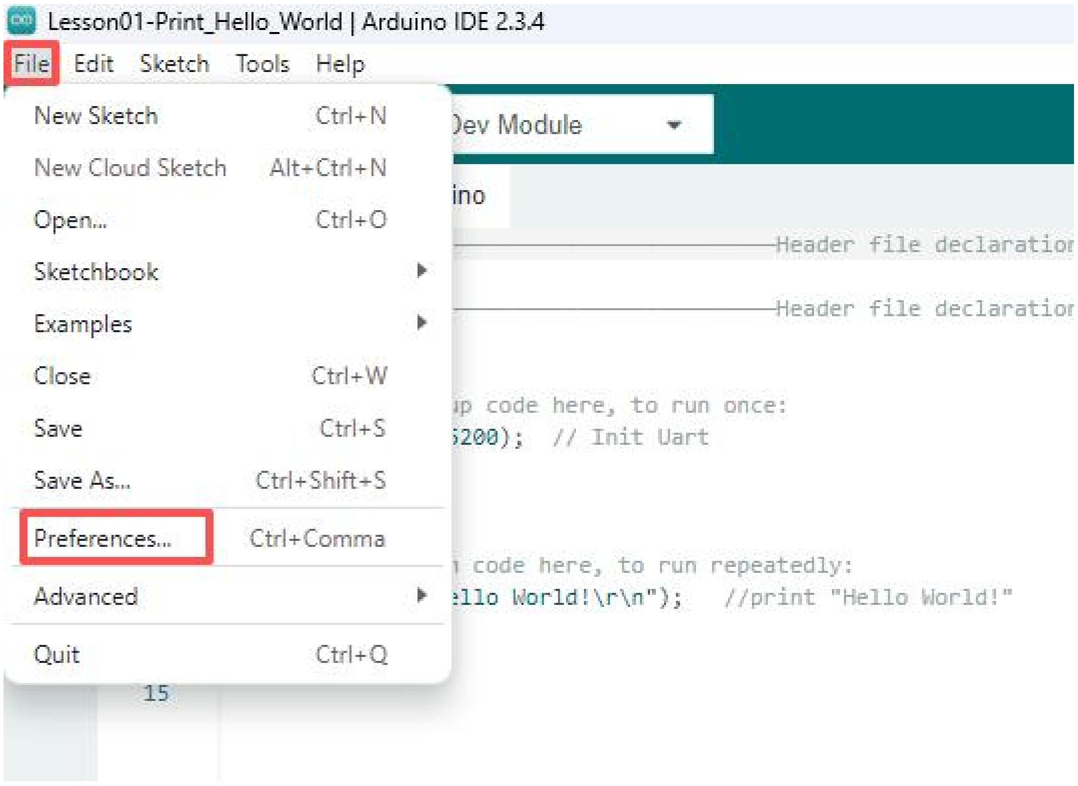
In this way, you have completed the necessary preparations for compiling the code in the Arduino IDE. Next, all you need to do is write your appropriate code and use the corresponding library files to run it.

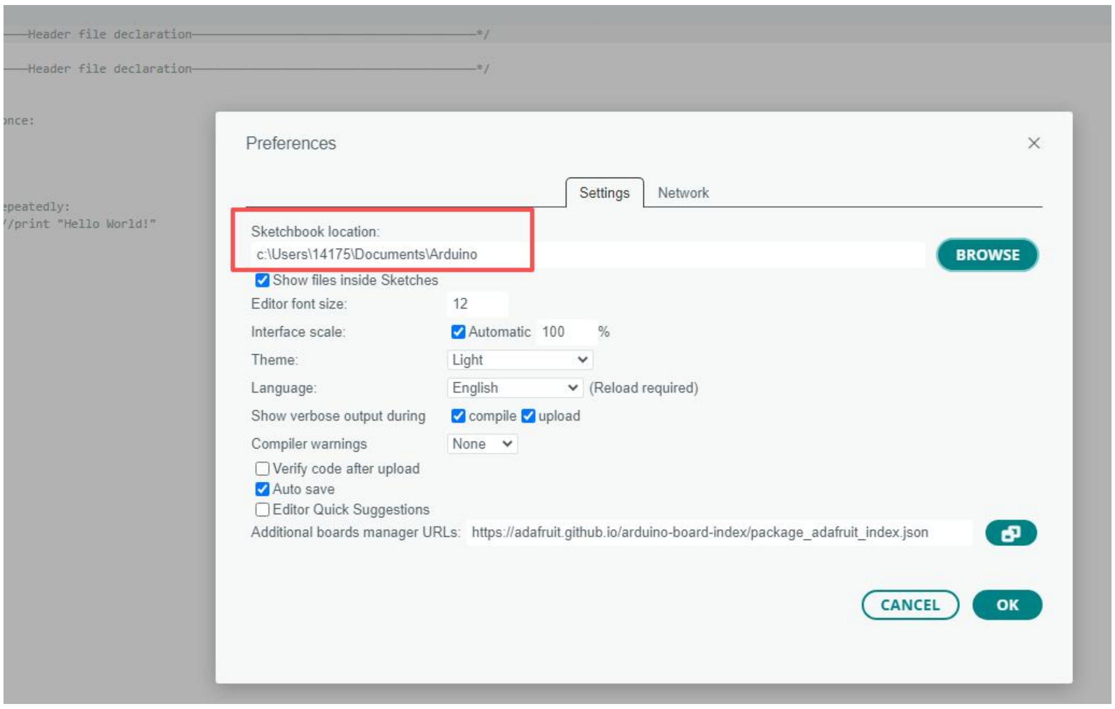
Importing Library Files (Important)

Before starting to use it, we need to first download the library files required for the subsequent functions of the development board to the local machine and place them in the corresponding "/Arduino/libraries" directory on your computer.

Download link:
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/libraries

Enter this path and check the location where the library files are stored.





Preferences

Settings | Network

Sketchbook location:
c:\Users\14175\Documents\Arduino

BROWSE

☑ Show files inside Sketches

Editor font size: 12

Interface scale: ☑ Automatic 100 %

Theme: Light

Language: English (Reload required)

Show verbose output during ☑ compile ☑ upload

Compiler warnings None

☐ Verify code after upload

☑ Auto save

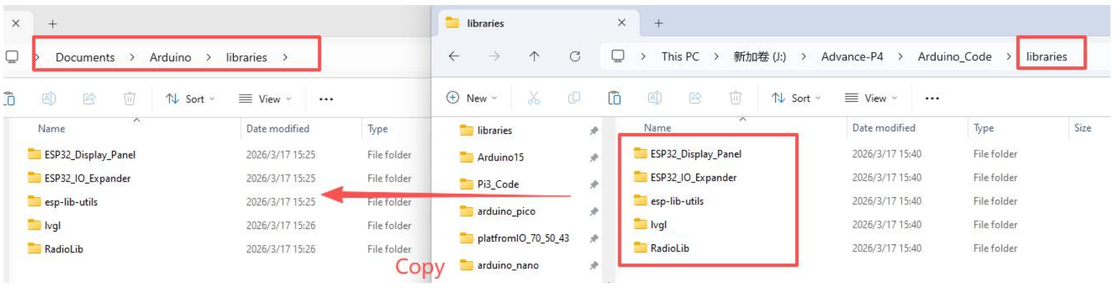
☐ Editor Quick Suggestions

Additional boards manager URLs: https://adafruit.github.io/arduino-board-index/package_adafruit_index.json

CANCEL OK

After determining the location where the library files are stored.

Copy all the library files contained in the downloaded "libraries" file to the "C:\Users\Username\Documents\Arduino\libraries" folder on your local computer.



Documents > Arduino > libraries >

Name	Date modified	Type
ESP32_Display_Panel	2026/3/17 15:25	File folder
ESP32_IO_Expander	2026/3/17 15:25	File folder
esp-lib-utils	2026/3/17 15:25	File folder
lvgl	2026/3/17 15:26	File folder
RadioLib	2026/3/17 15:26	File folder

This PC > 新加卷 (J:) > Advance-P4 > Arduino_Code > libraries

Name	Date modified	Type	Size
ESP32_Display_Panel	2026/3/17 15:40	File folder	
ESP32_IO_Expander	2026/3/17 15:40	File folder	
esp-lib-utils	2026/3/17 15:40	File folder	
lvgl	2026/3/17 15:40	File folder	
RadioLib	2026/3/17 15:40	File folder	

Note: If there is no "libraries" folder in your path, please create one yourself.

Lesson01--- Print "Hello World"

Introduction

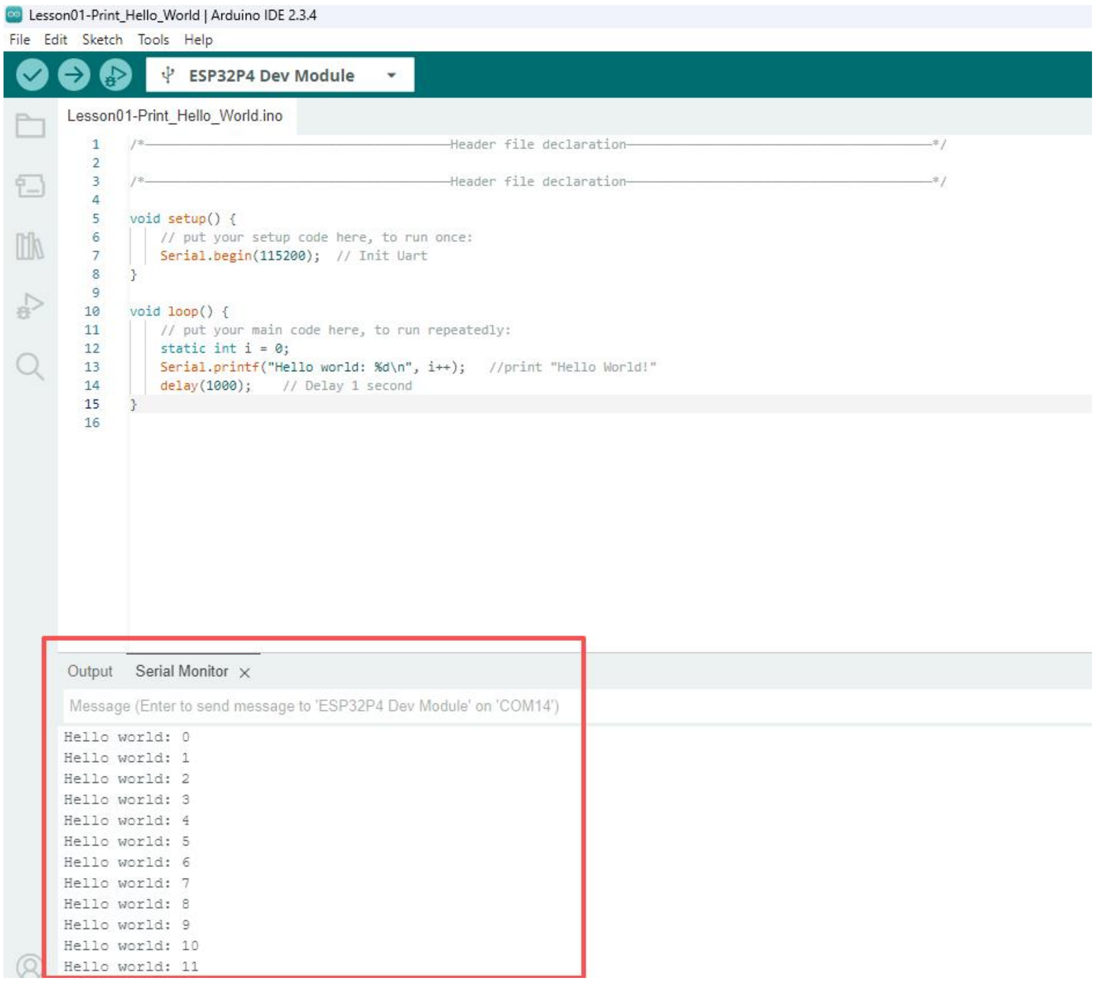
In this lesson, we will officially begin writing code in the Arduino-IDE environment to drive the Advance-P4 development board. Subsequent lessons will follow a gradual "from simple to complex" design, helping you progressively master the Arduino-IDE development framework and the usage logic of the ESP32-P4 chip, while building a clear technical understanding.

Learning Goals

1. Get familiar with the Arduino IDE and complete your first program upload to the ESP32-P4
2. Implement a "Hello World" serial print example on the ESP32-P4 development board

Preview of the Result

When running on the ESP32-P4, the serial terminal will output "Hello world" every 1 second, and the number will increase by 1 every second.


Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson01-Print_Hello_World.ino
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
/*——————————————————Header file declaration——————————————————*/

/*——————————————————Header file declaration——————————————————*/

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);  // Init Uart
}

void loop() {
  // put your main code here, to run repeatedly:
  static int i = 0;
  Serial.printf("Hello world: %d\n", i++);   //print "Hello World!"
  delay(1000);    // Delay 1 second
}

```

```
Hello world: 0
Hello world: 1
Hello world: 2
Hello world: 3
Hello world: 4
Hello world: 5
Hello world: 6
Hello world: 7
Hello world: 8
Hello world: 9
Hello world: 10
Hello world: 11
```

Hardware Used in This Lesson

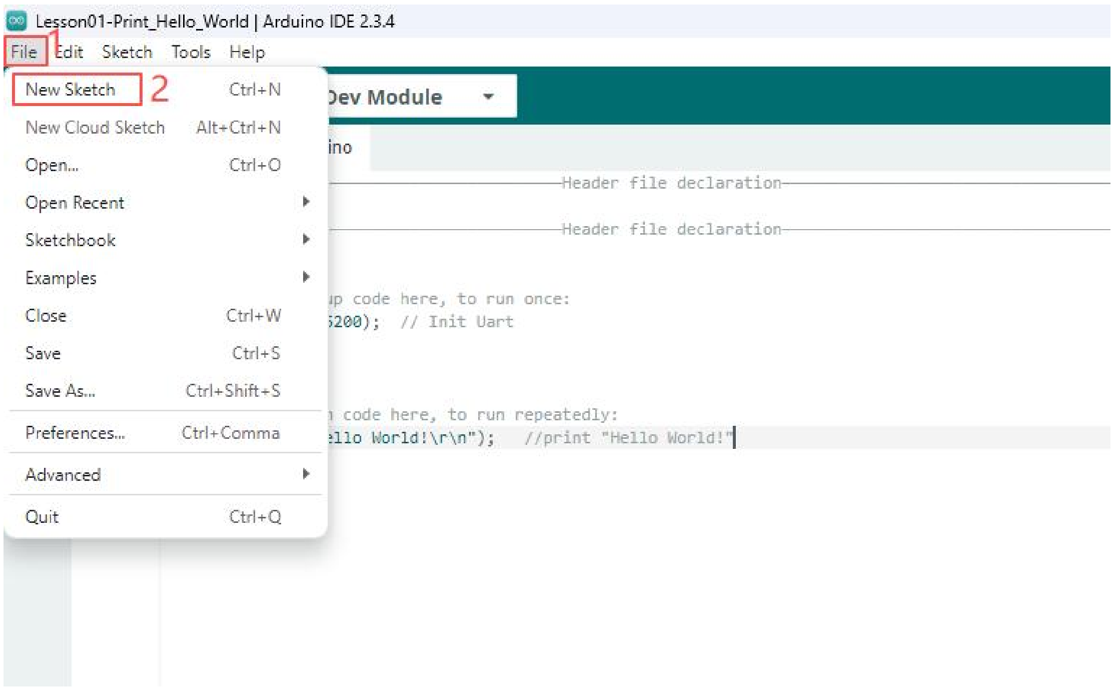
This lesson involves no hardware usage; it solely teaches you how to create a new project and how to upload code to the ESP32-P4 chip using the Arduino-IDE.

Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson01-Print_Hello_World

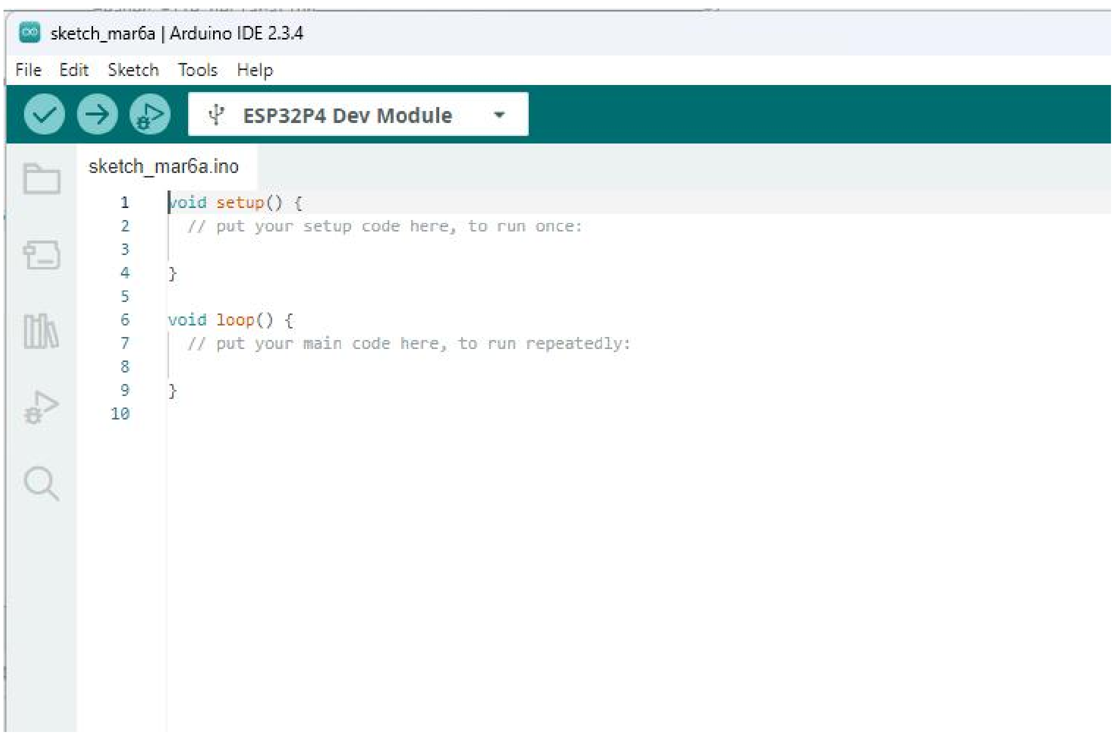
Key Explanations

First, let's walk through how to create a new project in the already-installed Arduino-IDE.



```
Header file declaration
Header file declaration
up code here, to run once:
5200);   // Init Uart
 code here, to run repeatedly:
ello World!\r\n");    //print "Hello World!"
```

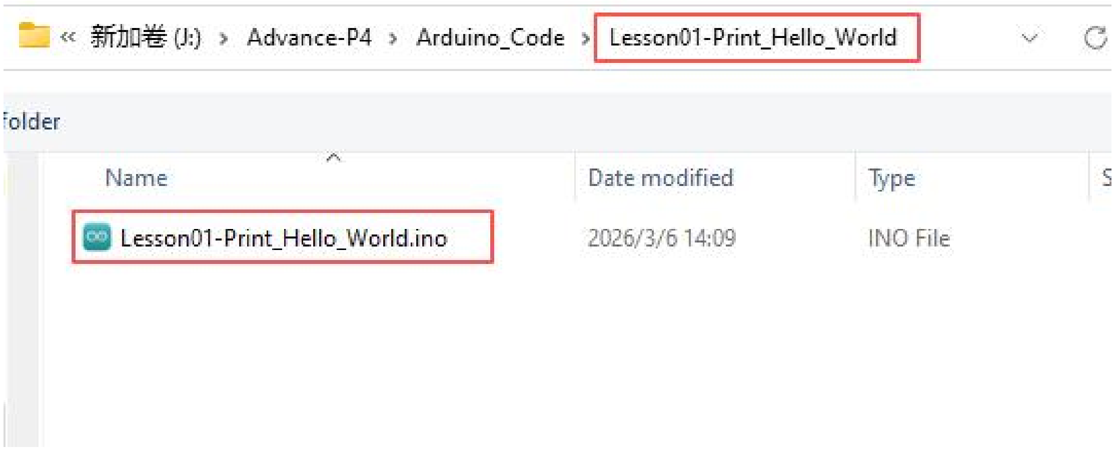
And just like that, a new project has been created.



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

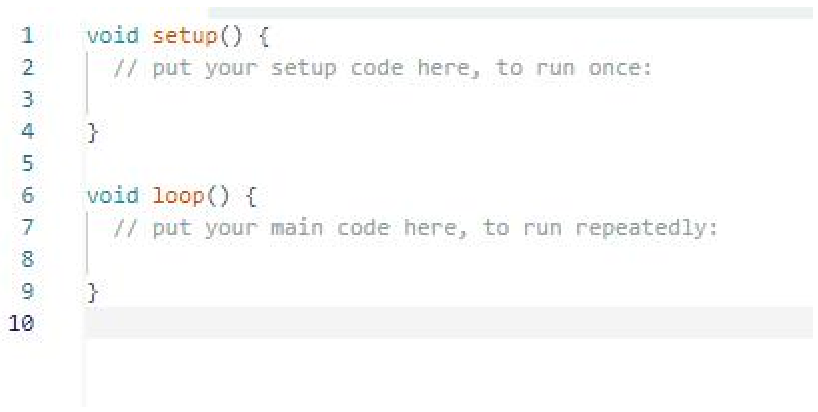
Press and hold Ctrl + S on your keyboard to save the project first.

Name the folder and file identically—this ensures the project opens correctly.


« 新加卷 (J:) › Advance-P4 › Arduino_Code › Lesson01-Print_Hello_World
folder

Name	Date modified	Type
Lesson01-Print_Hello_World.ino	2026/3/6 14:09	INO File

Once saved, you can start writing your code right here.



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

The setup() function runs only once when the Arduino powers on (boots up) or resets, specifically handling "preparation work"—just like putting on your shoes and grabbing your keys before leaving the house.

What belongs in setup():

- Pin mode configuration: Use pinMode() to define pins as INPUT or OUTPUT. For example, set pins to OUTPUT for controlling LEDs, and INPUT for reading buttons.
- Serial initialization: Use Serial.begin(9600) to enable serial communication (essential for debugging).
- Sensor/module initialization: Initial setup commands for LCD screens, servos, Bluetooth modules, etc., required before first use.
- Variable initial assignments: One-time variable assignments (such as setting initial brightness or threshold values).

The loop() function, after setup() completes, executes its code infinitely in a loop. It's the core logic zone of your program, responsible for processing sensor data, controlling actuators, and responding to external events.

What belongs in loop():

- Sensor data reading: Use digitalRead()/analogRead() to read buttons, temperature/humidity, photoresistors, etc.
- Actuator control: Use digitalWrite()/analogWrite() to control LED on/off states, motor speeds, buzzer sounds, etc.
- Logic judgments: if/else, for/while loops and conditional checks (e.g., "if button press detected, turn on LED").
- Delays/waits: Use delay() to set wait times (e.g., intervals for LED blinking).
- Data printing/transmission: Serial printing of sensor data, sending commands to Bluetooth modules, etc. (operations that need to run repeatedly).

In summary:

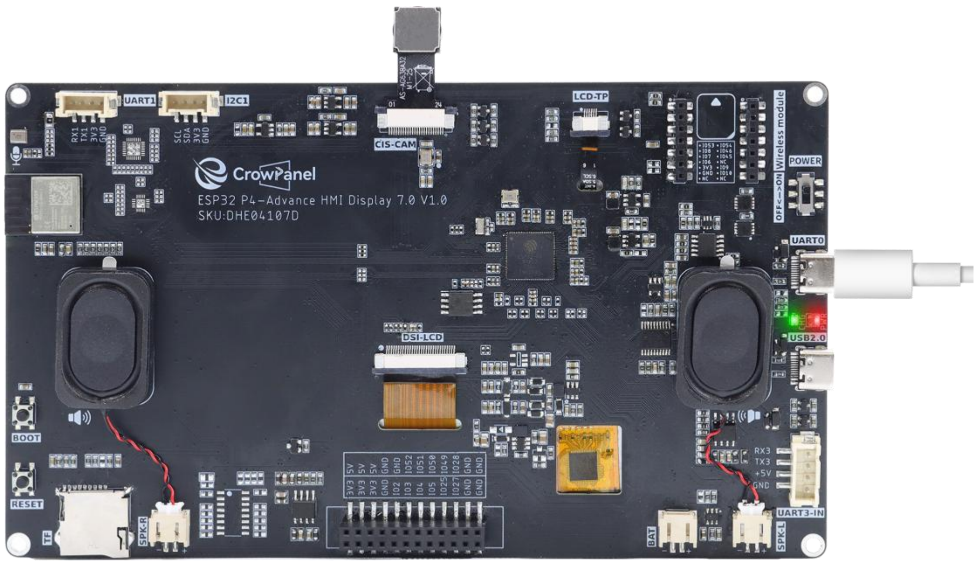
- setup() runs once—focus on initialization configurations (pins, serial, module setup, etc.). This is the "preparation phase" of your program.
- loop() runs infinitely—focus on business logic (reading data, controlling hardware, making judgments, etc.). This is the "runtime phase" of your program.

Remember the core principle: "Things that only need to happen once go in setup; things that need to happen repeatedly go in loop()."

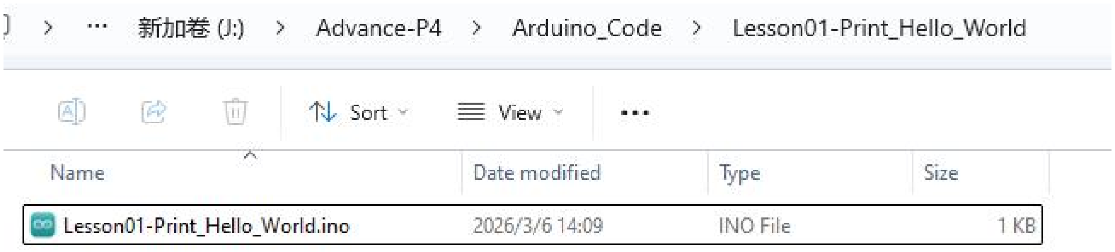
Programming Steps

Now that the code is ready, we need to flash the ESP32-P4 to see the results in action.

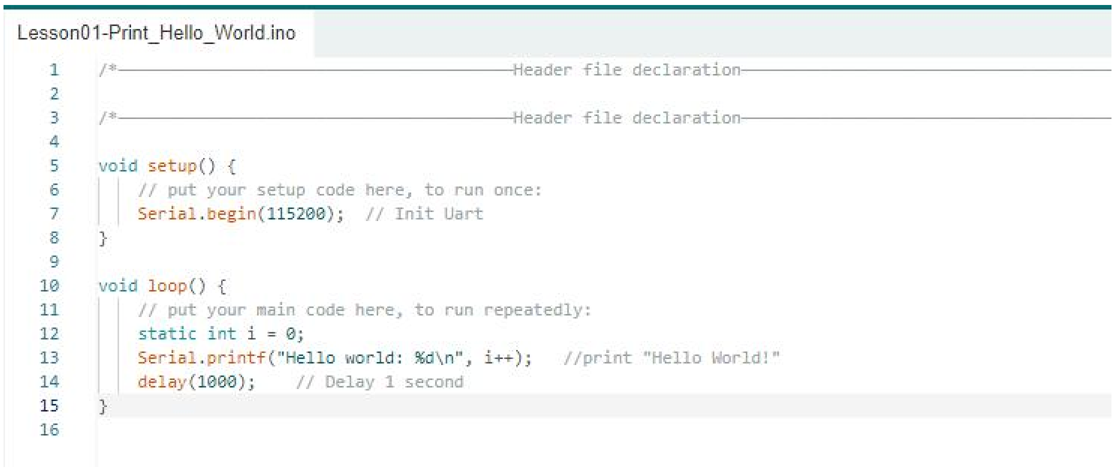
First, connect the Advance-P4 device to your computer host via a USB cable. (Connect to UART0)



Open the code for this lesson from the GitHub repository link provided above.

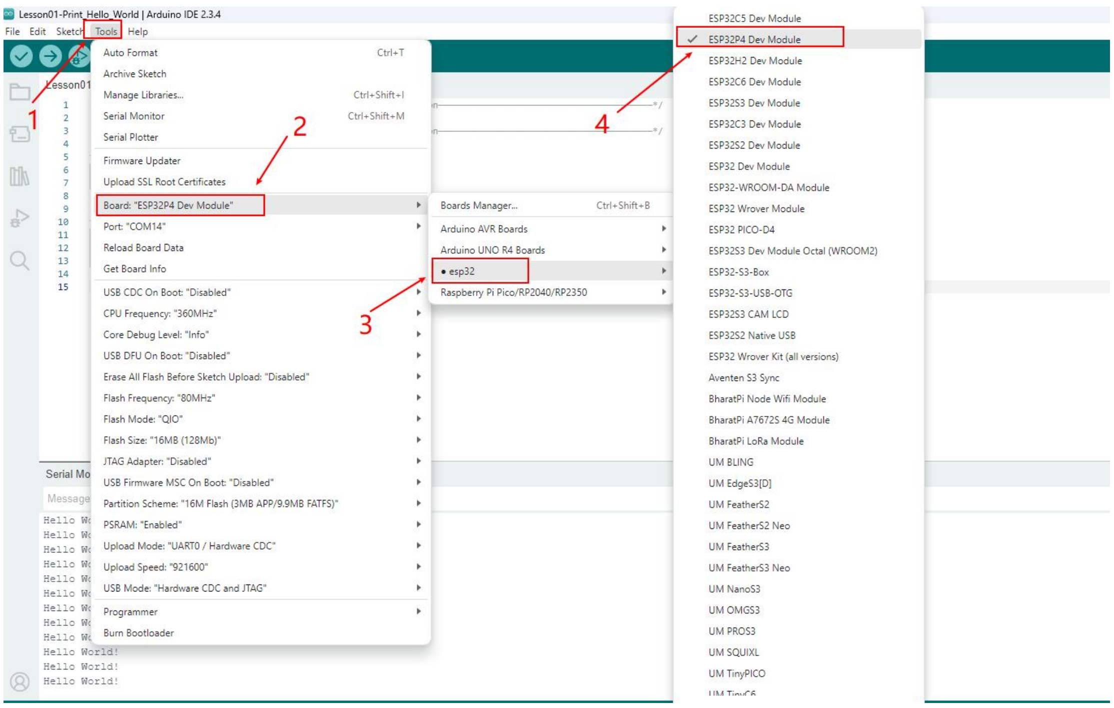

新加卷 (J:) > Advance-P4 > Arduino_Code > Lesson01-Print_Hello_World
Sort
View

Name	Date modified	Type	Size
Lesson01-Print_Hello_World.ino	2026/3/6 14:09	INO File	1 KB



```
/*-----------------------------Header file declaration-----------------------------

/*-----------------------------Header file declaration-----------------------------

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);  // Init Uart
}

void loop() {
  // put your main code here, to run repeatedly:
  static int i = 0;
  Serial.printf("Hello world: %d\n", i++);   //print "Hello World!"
  delay(1000);    // Delay 1 second
}

```

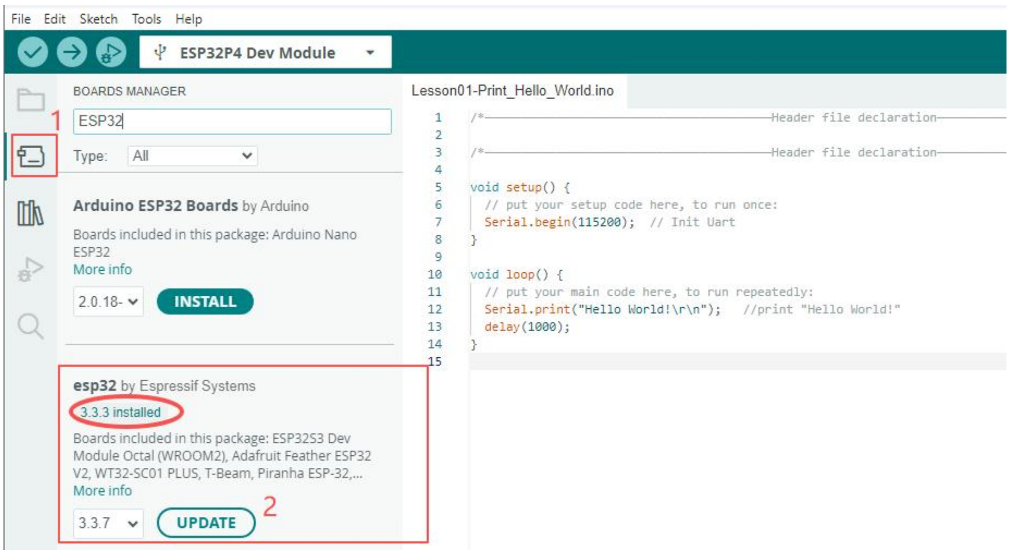
Now let's configure the settings for uploading the code. Click Tools, select Board, and choose **"ESP32P4 Dev Module"** from the list.



Board: "ESP32P4 Dev Module"
Port: "COM14"
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"

```
Hello World!
Hello World!
Hello World!
```

(If you don't see ESP32 in the list here, please follow the instructions from the previous lesson to install ESP32 version 3.3.3.)



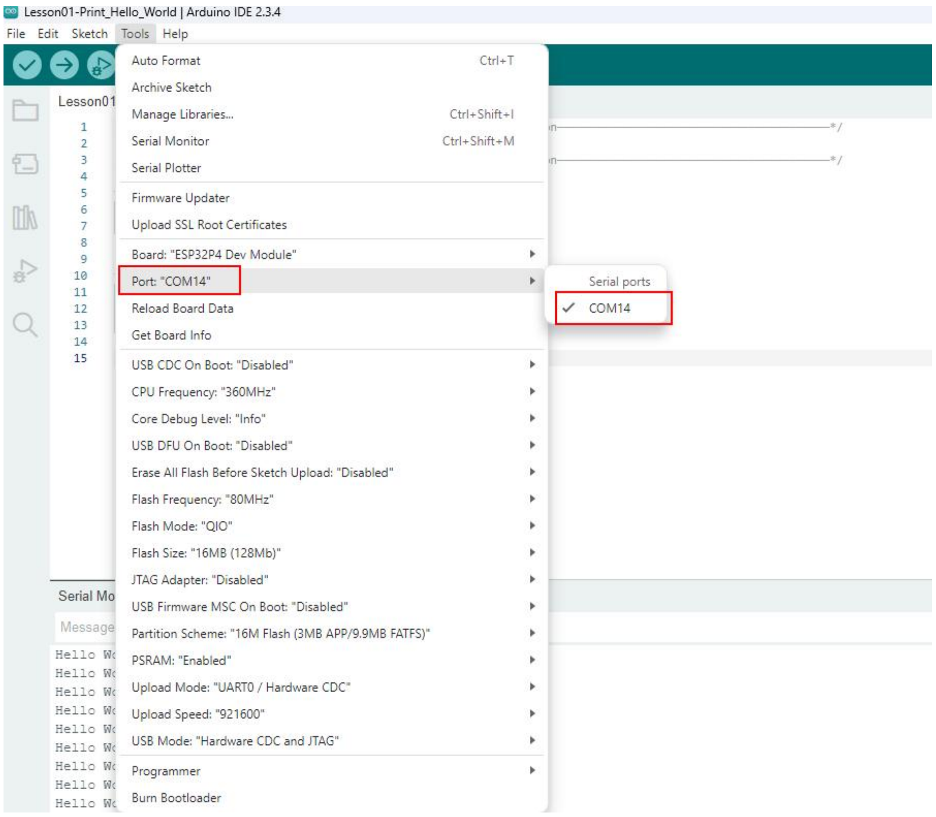
Boards included in this package: Arduino Nano ESP32

Boards included in this package: ESP32S3 Dev Module Octal (WROOM2), Adafruit Feather ESP32 V2, WT32-SC01 PLUS, T-Beam, Piranha ESP-32,...

```
/*————————————————————Header file declaration————
/*————————————————————Header file declaration————

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);  // Init Uart
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.print("Hello World!\r\n");   //print "Hello World!"
  delay(1000);
}
```

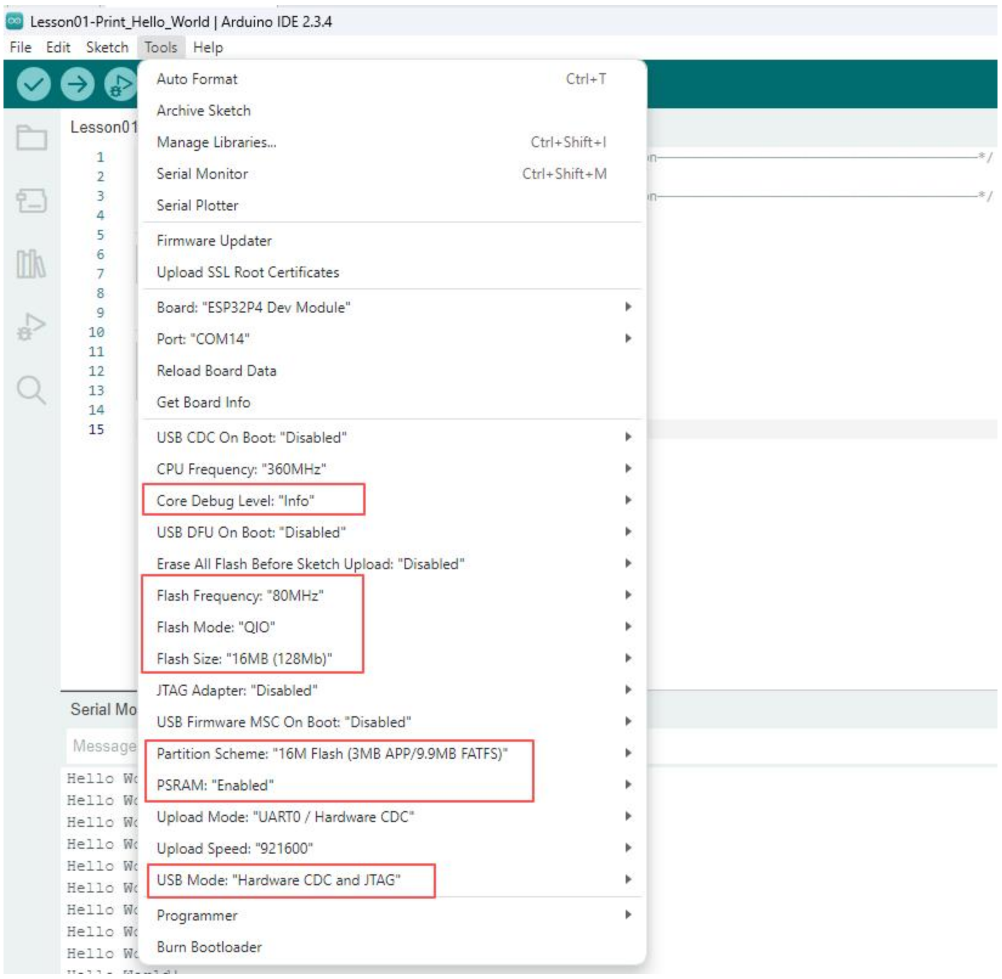
Then select Tools again and verify your COM port number.

(This assumes you have already connected your CrowPanel Advanced ESP32-P4 HMI AI Display [5-inch] to your computer.)


Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Reload Board Data
Get Board Info
Programmer
Burn Bootloader
Serial ports
✓ COM14

Board: "ESP32P4 Dev Module"
Port: "COM14"
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"

Finally, double-check that your options match my selections exactly.


Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Reload Board Data
Get Board Info
Programmer
Burn Bootloader

Board: "ESP32P4 Dev Module"
Port: "COM14"
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"

1. Core Debug Level: "Info"

Purpose: Controls the verbosity of system log output. "Info" is the default level, outputting boot messages and key status logs.
Why set this: During development, you need to see system runtime status (e.g., boot flow, WiFi/Bluetooth initialization logs) for troubleshooting. Higher levels (like "Debug") output excessive logs affecting performance; lower levels (like "Error") lose critical information hindering debugging. For production environments, set to "None" to disable logs and save resources.

2. Flash Configuration Summary（Flash Frequency / Flash Mode / Flash Size）

These three parameters collectively determine external Flash read/write performance and capacity recognition:
Flash Frequency: "80MHz":
The ESP32-P4 supports up to 80MHz Flash clock—a balanced choice for performance and stability. Higher frequencies improve program loading and data read/write speeds; lower frequencies slow system response.
Flash Mode: "QIO":
QIO (Quad I/O) is a four-wire parallel read/write mode, faster than traditional DIO (dual-line), fully utilizing high-speed Flash bandwidth. The high-speed Flash paired with ESP32-P4 generally supports QIO—this is the performance-prioritized standard configuration.
Flash Size: "16MB (128Mb)":
Must match your hardware's actual Flash capacity. Common ESP32-P4 development boards feature 16MB Flash. Incorrect settings cause: capacity recognition errors preventing complete program flashing; partition table mismatches leading to system crashes or data loss.

3. Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"

Purpose: Divides 16MB Flash into functional zones: 3MB APP for your application code; 9.9MB FATFS for file system partition (storing configurations, logs, web files, etc.); remaining space for system boot, OTA upgrades, etc.
Why set this: The ESP32-P4 lacks built-in large-capacity storage, requiring Flash partitioning to manage code and data. This scheme offers a general balance: ensuring sufficient program space while reserving ample storage for files (e.g., web servers, data logging). If your application is small, consider a smaller APP partition; if more file storage is needed, adjust the ratio accordingly.

4. PSRAM: "Enabled"

Purpose: Enables external PSRAM (Pseudo-Static Random Access Memory), providing the ESP32-P4 with additional large-capacity memory (typically 8MB/16MB).
Why set this: The ESP32-P4 has limited built-in RAM; complex applications (AI

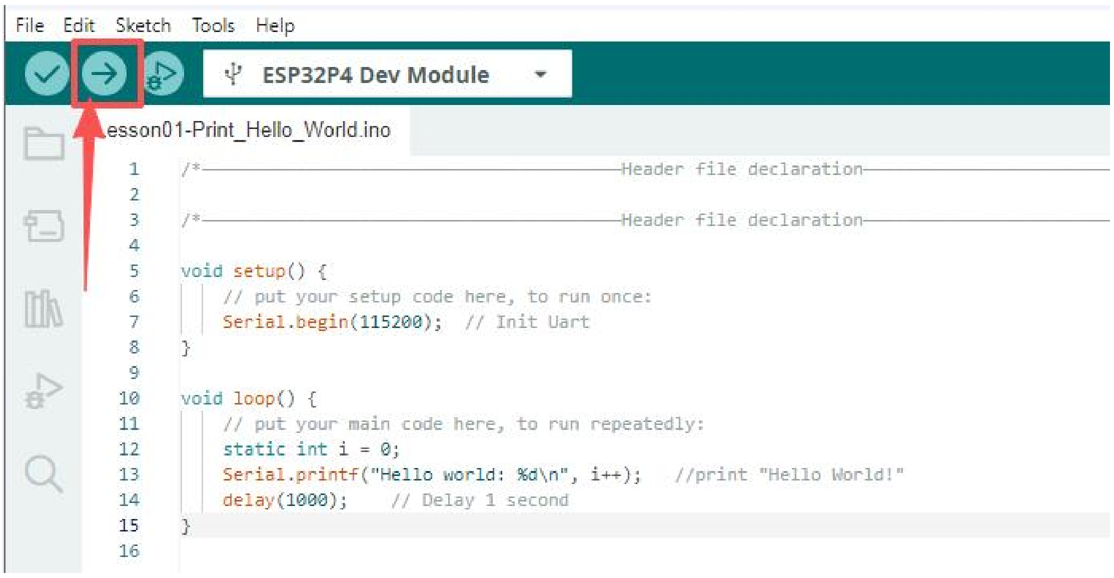
models, multimedia, large buffers) may run out of memory. Enabling PRAM allows the system to place heap memory, large variables, and caches in PSRAM, significantly enhancing multitasking and complex program execution capabilities. If your development board lacks PSRAM, you must disable this—otherwise the system will crash attempting to access non-existent memory.

5. USB Mode: "Hardware CDC and JTAG"

Purpose: Configures USB port functionality: Hardware CDC—hardware-level USB virtual serial port for serial communication (log printing, debugging), more stable and faster than software CDC; JTAG—hardware debugging interface for online debugging, breakpoints, and register inspection.
Why set this: The ESP32-P4 supports USB Hardware CDC, providing high-speed, stable serial connections (ideal for extensive log output). Simultaneously enabling JTAG facilitates low-level debugging (e.g., troubleshooting crashes, performance bottlenecks). If only serial is needed, select "Hardware CDC" only; if debugging isn't required, disable JTAG to save resources.

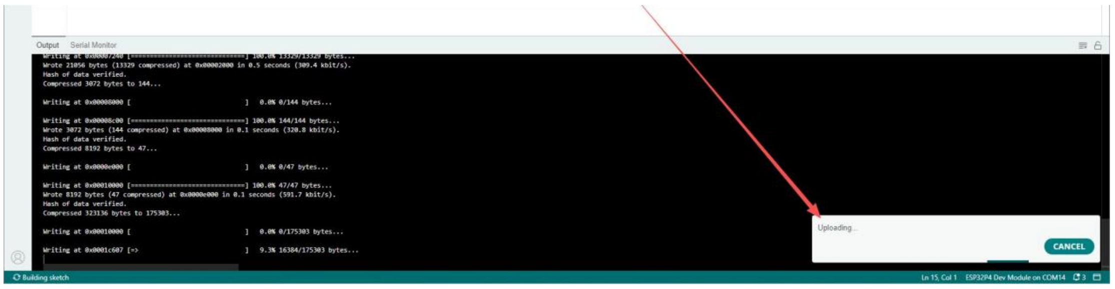
These configurations remain identical for all subsequent project code uploads.

Once configured, upload the code.



```
/*————————————————————————————————————Header file declaration————————————————————

/*————————————————————————————————————Header file declaration————————————————————

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);  // Init Uart
}

void loop() {
  // put your main code here, to run repeatedly:
  static int i = 0;
  Serial.printf("Hello world: %d\n", i++);   //print "Hello World!"
  delay(1000);    // Delay 1 second
}
```

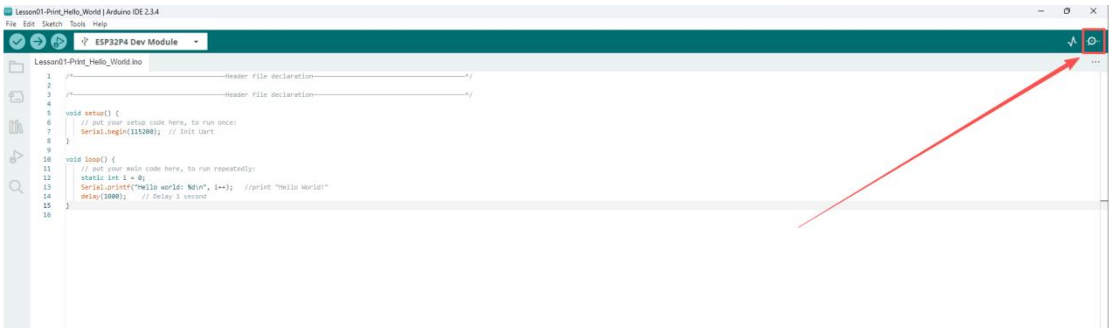
After waiting a moment, you will see the message indicating successful code upload in the output console.



```
Writing at 0x00007240 [==============================] 100.0% 13329/13329 bytes...
Wrote 21056 bytes (13329 compressed) at 0x00002000 in 0.5 seconds (309.4 kbit/s).
Hash of data verified.
Compressed 3072 bytes to 144...

Writing at 0x00008000 [                              ]   0.0% 0/144 bytes...

Writing at 0x00008c00 [==============================] 100.0% 144/144 bytes...
Wrote 3072 bytes (144 compressed) at 0x00008000 in 0.1 seconds (320.8 kbit/s).
Hash of data verified.
Compressed 8192 bytes to 47...

Writing at 0x0000e000 [                              ]   0.0% 0/47 bytes...

Writing at 0x00010000 [==============================] 100.0% 47/47 bytes...
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (591.7 kbit/s).
Hash of data verified.
Compressed 323136 bytes to 175303...

Writing at 0x00010000 [                              ]   0.0% 0/175303 bytes...

Writing at 0x0001c607 [=>                            ]   9.3% 16384/175303 bytes...
```

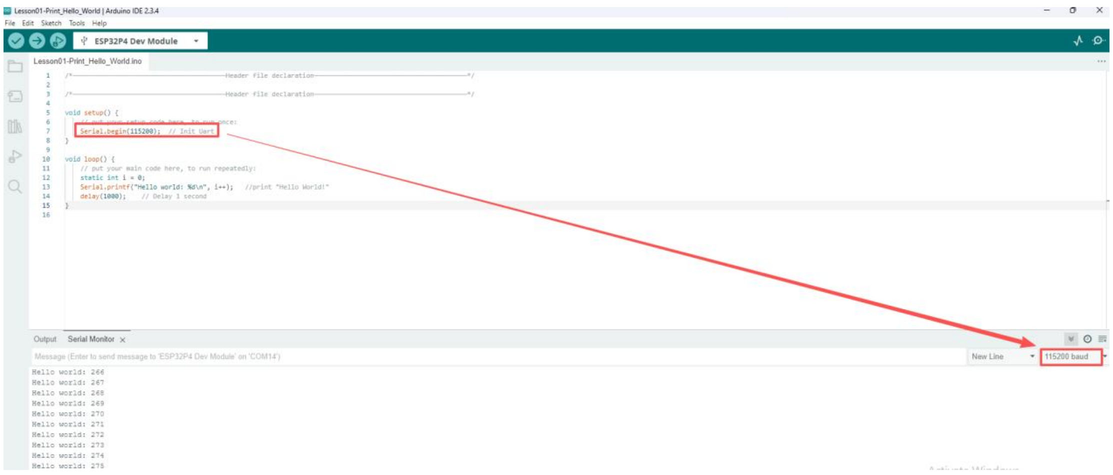
Next, simply open the Serial Monitor and you'll see "Hello World" being printed.

Click the Serial Monitor icon in the top-right corner of the interface to open it.


Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson01-Print_Hello_World.ino

```
/*————————————————Header file declaration————————————————*/

/*————————————————Header file declaration————————————————*/

void setup() {
    // put your setup code here, to run once:
    Serial.begin(115200);  // Init Uart
}

void loop() {
    // put your main code here, to run repeatedly:
    static int i = 0;
    Serial.printf("Hello world: %d\n", i++);   //print "Hello World!"
    delay(1000);    // Delay 1 second
}
```

Then select the baud rate matching your code settings here—this ensures proper output display; otherwise, you'll see garbled text or no output at all.


Lesson01-Print_Hello_World | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson01-Print_Hello_World.ino
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')
New Line
115200 baud

```
/*————————————————Header file declaration————————————————*/

/*————————————————Header file declaration————————————————*/

void setup() {
    // put your setup code here, to run once:
    Serial.begin(115200);  // Init Uart
}

void loop() {
    // put your main code here, to run repeatedly:
    static int i = 0;
    Serial.printf("Hello world: %d\n", i++);   //print "Hello World!"
    delay(1000);    // Delay 1 second
}
```

```
Hello world: 266
Hello world: 267
Hello world: 268
Hello world: 269
Hello world: 270
Hello world: 271
Hello world: 272
Hello world: 273
Hello world: 274
Hello world: 275
```

And that concludes all the content for this lesson. In the next lesson, we will gradually increase the difficulty and show you how to add library files and how to control more hardware and peripherals.

Lesson02---Turn on the LED

Introduction

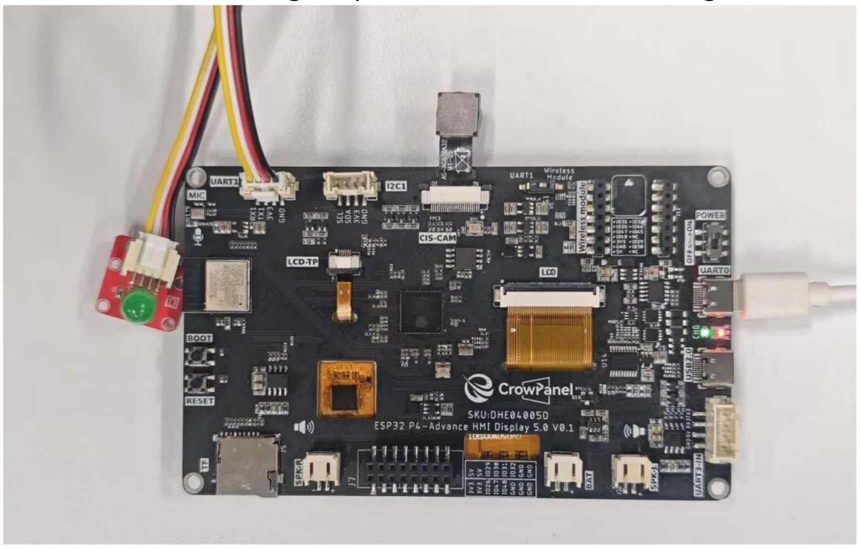
In this lesson, we will begin exploring the simplest control logic in the Arduino-IDE to perform level control on the UART1 interface of the Advance-P4. This will make the LED connected to the UART1 interface turn on for one second and off for one second in a continuous cycle.

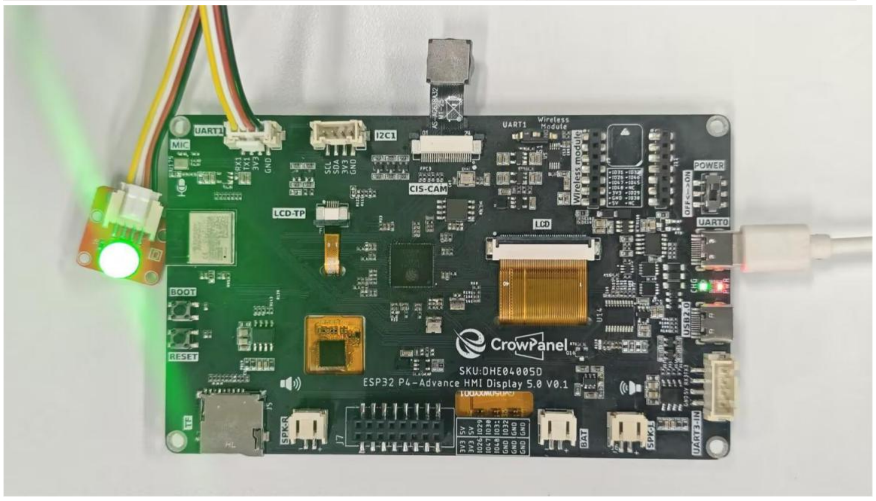
Learning Goals

1. Understand the principles of GPIO and LEDs
2. Learn the syntax of "pinMode" and "digitalWrite"
3. Light up the LED connected to the UART1 interface

Preview of the Result

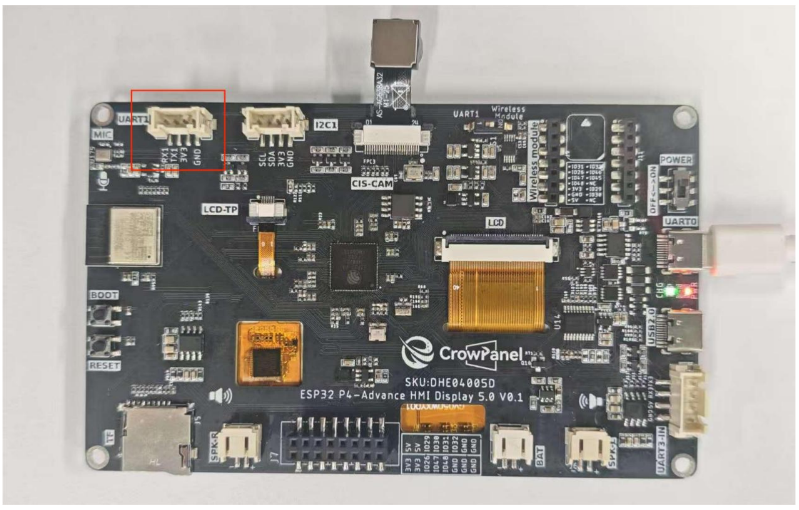
After running the code, you will be able to observe that the LED connected to the UART1 interface will light up for one second and then go off for one second.





Hardware Used in This Lesson

Introduction to the UART1 Interface on the Advance-P4



On our Advance-P4 board, the UART1 interface is identified by the name "UART". We should look for an interface that can be used for serial communication.

Moreover, during the initial design phase, this UART1 interface can also be used as a regular GPIO port. That is, we can treat the RX and TX pins on this interface as two regular GPIO ports.

Introduction to GPIO

The ESP32-P4 chip offers 55 general-purpose input/output (GPIO) functions, providing flexibility and adaptability for a wide range of applications.

The key features of these GPIOs include:

① Multi-functionality: Each GPIO pin can not only be used as an input or output, but can also be configured as various roles through IO MUX (refer to Chapter 2 for details), such as PWM, ADC, I2C, SPI, etc. This enables the ESP32-P4 to adapt to various peripheral connections.
② High current output: The GPIO pins of ESP32-P4 support up to 40mA of current output, allowing direct driving of low-power loads such as LEDs. This reduces the complexity of external driver circuits.
③ Programmability: Through the ESP-IDF (SDK) development framework, users can flexibly configure the input/output mode, pull-up/pull-down parameters, and other settings of each GPIO to meet specific application requirements.

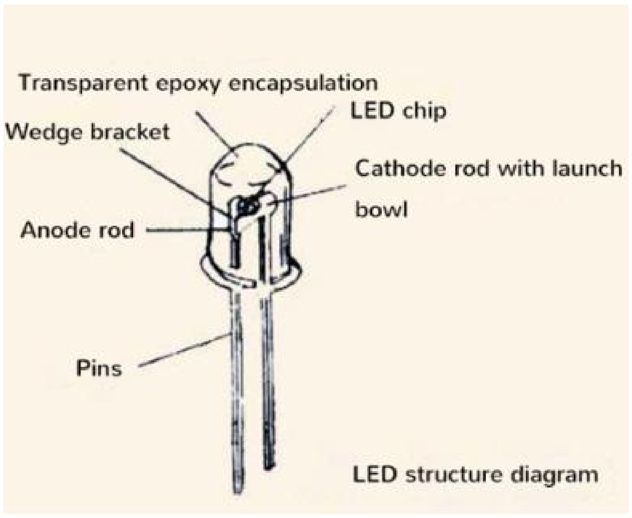
④ Interrupt support: GPIO pins support interrupt functionality, which can trigger interrupts when the signal changes. This is suitable for real-time response applications such as button detection and sensor triggering.
⑤ Status indication: GPIO pins can be used as LED indicators, achieving status visualization through simple high/low level switching. This helps users debug and monitor system operation.

The GPIO functions of ESP32-P4 provide powerful hardware support for developers. In this chapter, we will delve into the application and configuration of GPIO through an example of lighting an LED.

Introduction to LED

LED is a highly efficient and durable miniature semiconductor device that emits light when an electric current passes through it. It has the advantages of high energy conversion efficiency, low heat generation, and environmental friendliness. They are commonly used in indicator lights, display screens, and lighting equipment. LEDs have fast response times and a wide range of color options, making them widely used in electronic products. In the ESP32-P4 lighting demonstration, GPIO control simplifies and makes it intuitive to switch the LEDs, helping users better understand their practical applications.

① The principle of LED light emission
LED devices are light-emitting components based on solid-state semiconductor technology. When a forward current is applied to a semiconductor material with a PN junction, the recombination of charge carriers within the semiconductor releases energy in the form of photons, thereby generating light. Therefore, LEDs are cold light sources, unlike lighting based on filament, which generates heat and thus avoids problems such as burning out. The following chart illustrates the operating principle of LED devices.



In the above chart, the PN junction of the semiconductor exhibits the characteristics of forward conduction, reverse blocking, and breakdown. When there is no external bias and the junction is in a thermal equilibrium state, no carrier recombination occurs within the PN junction, and thus no light emission is produced. However, when a forward bias is applied, the light emission process of the PN junction can be divided into three stages:

Firstly, carriers are injected under forward bias;

Secondly, electrons and holes recombine within the P region, releasing energy;

Finally, the energy released during the recombination process is radiated outward in the form of light. In summary, when current passes through the PN junction, electrons are driven to the P region by the electric field. There, they combine with holes, releasing excess energy and generating photons, thereby achieving the light-emitting function of the PN junction.

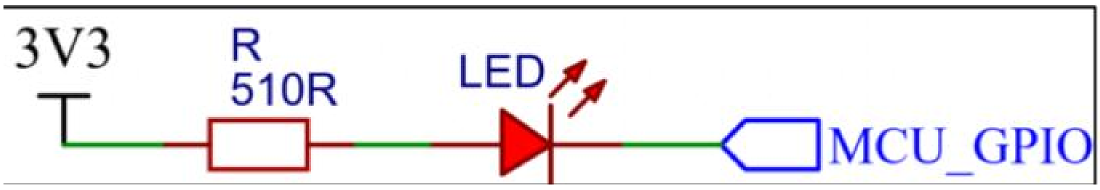
Note: The color of the light emitted by an LED is determined by the band gap width of the semiconductor material used. Different materials will produce light of different wavelengths, thus being able to generate light output of various colors. This efficient light-emitting mechanism has made light-emitting diodes widely adopted in lighting and indication applications.

② Principle of LED Lighting Driver

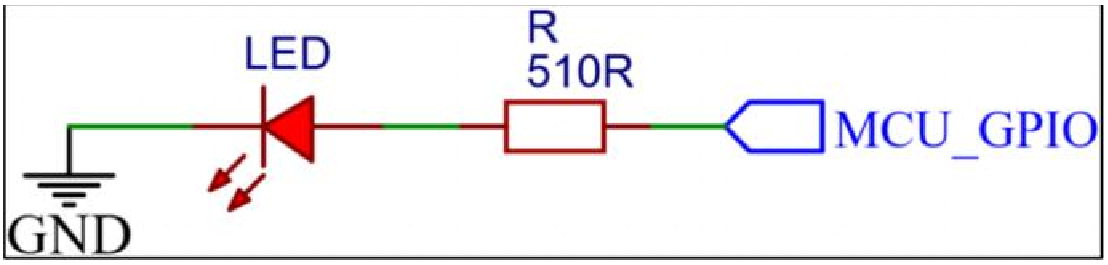
LED driving refers to providing appropriate current and voltage to LEDs through a stable power supply to ensure their normal lighting. The main driving methods for LEDs are constant current driving and constant voltage driving, among which constant current driving is more favored as it can limit the current. Due to the fact that LED lights are very sensitive to current fluctuations, exceeding their rated current may cause damage. Therefore, constant current driving ensures the operation of LEDs by maintaining a stable current flow. Next, we will study these two LED driving methods.

1) Current injection connection. This refers to the working current of the LED being provided externally, and the current is injected into our microcontroller.

The risk here is that the fluctuations of the external power supply can easily cause the microcontroller pins to burn out.



2) Power current configuration. This refers to the voltage and current provided by the microcontroller, and the current output will be applied to the LED. If the LED is driven directly by the GPIO of the microcontroller, its driving capability is relatively weak and may not be able to provide sufficient current for driving the LED.



The LED circuit on the ESP32-P4 development board adopts the "current receiving" configuration. This approach avoids the microcontroller directly powering and supplying current to the LED, thereby effectively reducing the load on the microcontroller. This enables the microcontroller to focus more on performing other core tasks, thereby enhancing the performance and stability of the entire system.

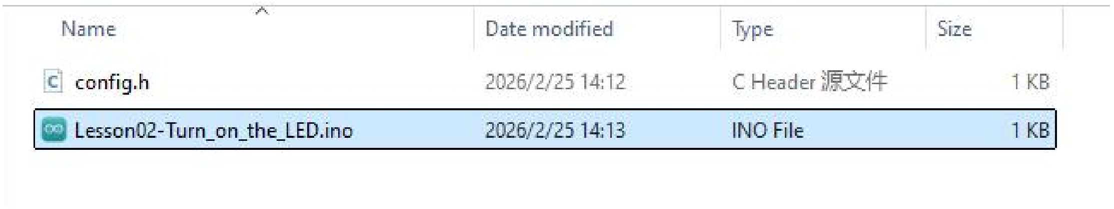
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson02-Turn_on_the_LED

Key Explanations

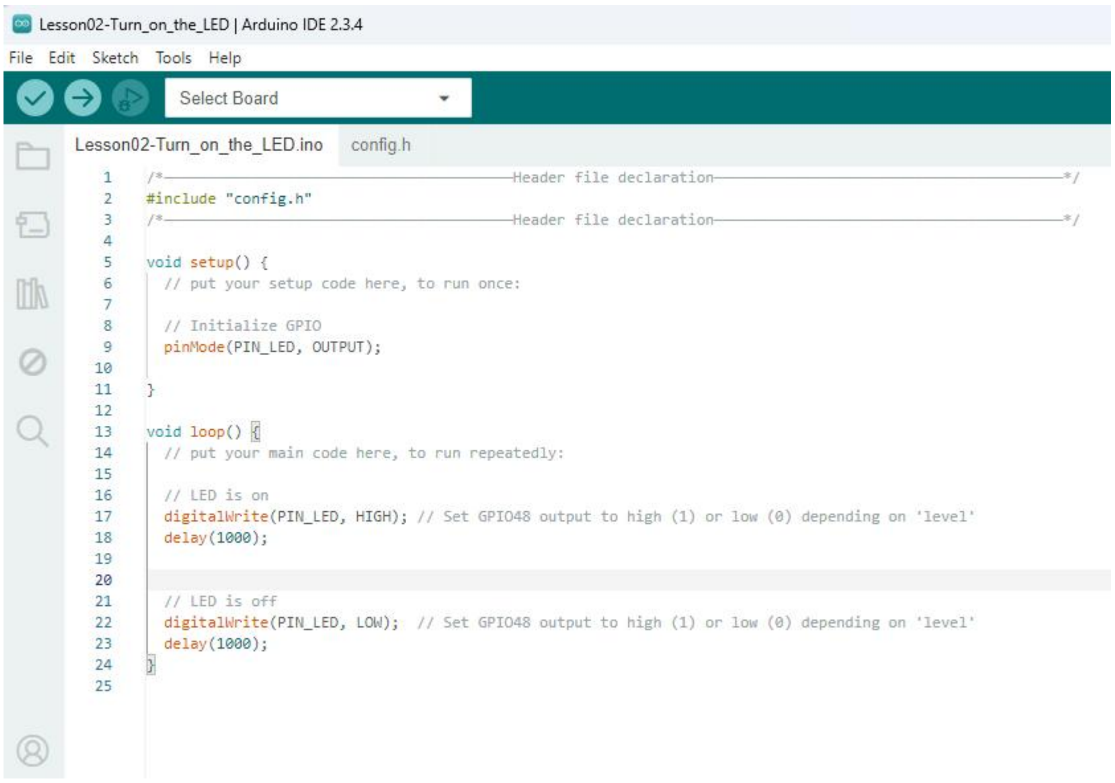
Next, let's explore how the control logic works in this lesson's code. We'll investigate this question together.

Double-click to open this lesson's code (the .ino file).



Name	Date modified	Type	Size
config.h	2026/2/25 14:12	C Header 源文件	1 KB
Lesson02-Turn_on_the_LED.ino	2026/2/25 14:13	INO File	1 KB

After opening, you can see the project's code and the configuration file "config.h".

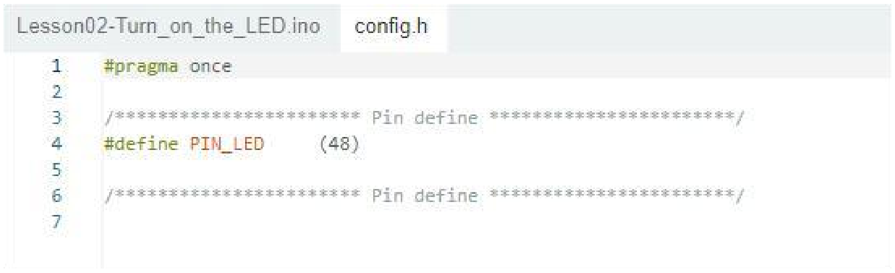

Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4
File Edit Sketch Tools Help
Select Board
Lesson02-Turn_on_the_LED.ino
config.h

```
/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include "config.h"
/*————————————————————————————————————Header file declaration————————————————————————————————————*/

void setup() {
  // put your setup code here, to run once:

  // Initialize GPIO
  pinMode(PIN_LED, OUTPUT);

}

void loop() {
  // put your main code here, to run repeatedly:

  // LED is on
  digitalWrite(PIN_LED, HIGH); // Set GPIO48 output to high (1) or low (0) depending on 'level'
  delay(1000);


  // LED is off
  digitalWrite(PIN_LED, LOW);  // Set GPIO48 output to high (1) or low (0) depending on 'level'
  delay(1000);
}

```

In subsequent code, the ".ino" file will primarily implement our functionality.

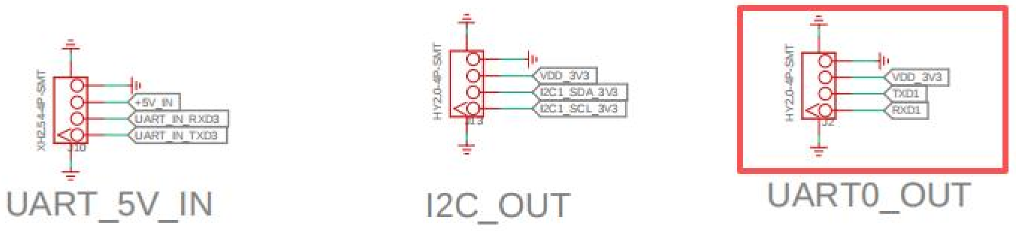
The "config.h" file, meanwhile, houses required pin definitions and other related configurations, enabling code separation and decoupling for easier management.

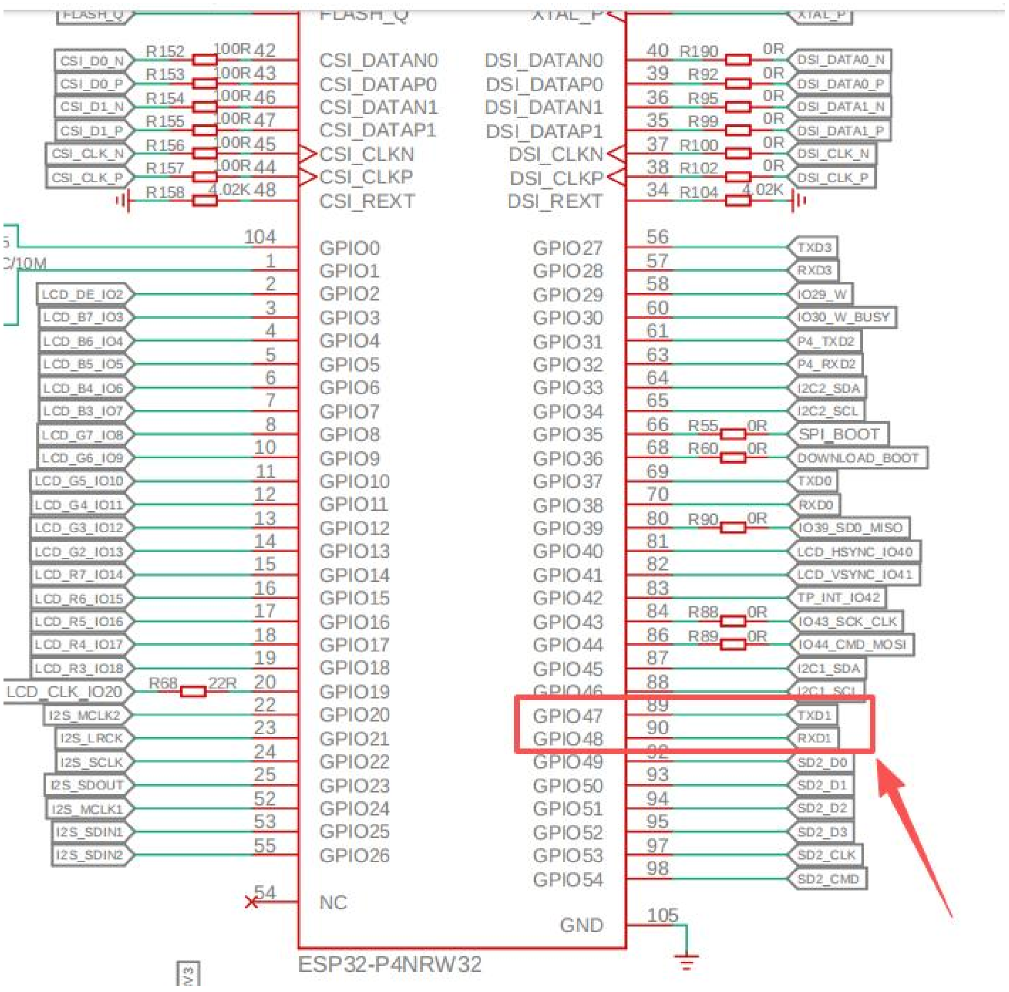
Why this approach: Centralizing pin numbers, parameters, and other constants in a header file allows you to modify only the header file later without touching the main program. This adheres to the programming principle of "high cohesion, low coupling."



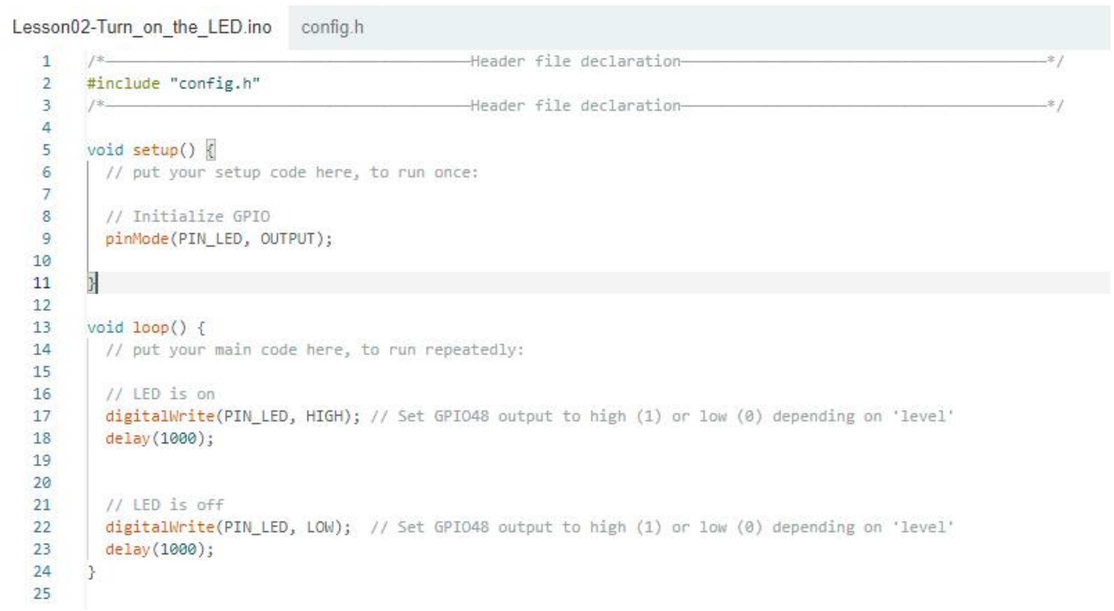
```
#pragma once

/*********************** Pin define ***********************/
#define PIN_LED     (48)

/*********************** Pin define ***********************/

```

In this lesson, we are primarily controlling the LED on the UART1 interface, specifically pin 48 at the UART1 interface.





Now let's look back at the code and walk through it line by line.


Lesson02-Turn_on_the_LED.ino   config.h

```
/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include "config.h"
/*————————————————————————————————————Header file declaration————————————————————————————————————*/

void setup() {
  // put your setup code here, to run once:

  // Initialize GPIO
  pinMode(PIN_LED, OUTPUT);

}

void loop() {
  // put your main code here, to run repeatedly:

  // LED is on
  digitalWrite(PIN_LED, HIGH); // Set GPIO48 output to high (1) or low (0) depending on 'level'
  delay(1000);


  // LED is off
  digitalWrite(PIN_LED, LOW);  // Set GPIO48 output to high (1) or low (0) depending on 'level'
  delay(1000);
}

```

pinMode(PIN_LED, OUTPUT);： Core Arduino function that sets the GPIO pin's operating mode.

Parameter 1（PIN_LED）： The pin number to configure (defined in config.h, e.g., 48).

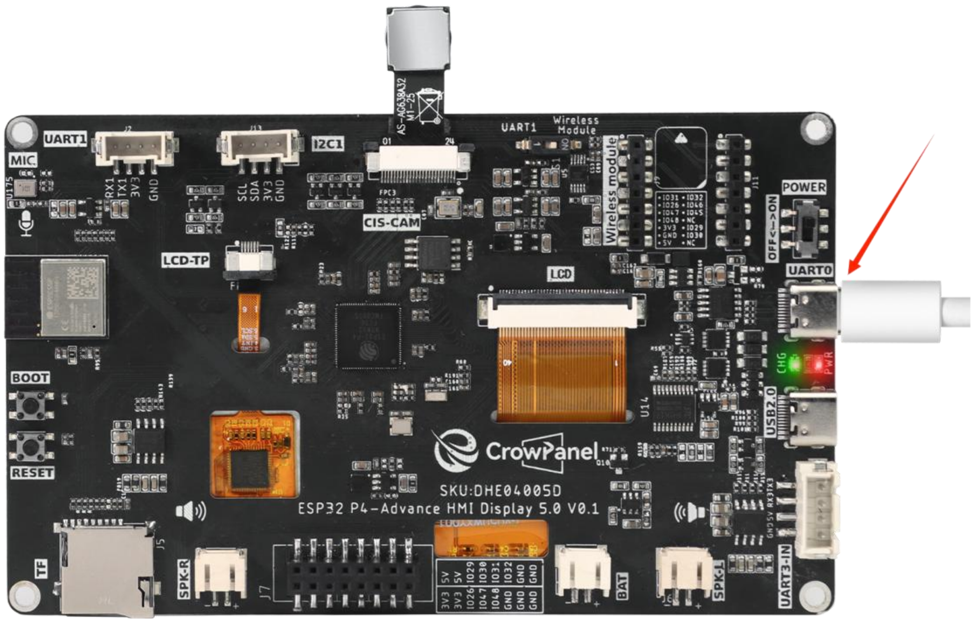
Parameter 2（OUTPUT）: Pin mode; OUTPUT means "output mode" (ESP32-P4 GPIO supports INPUT/OUTPUT/INPUT_PULLUP, etc.).
Underlying logic: This function configures the ESP32-P4 registers, setting the corresponding GPIO pin's hardware circuit to a state "capable of outputting high/low levels externally."

digitalWrite(PIN_LED, HIGH);: Core Arduino function that outputs high or low levels to a specified GPIO pin.
Parameter 1（PIN_LED）: The pin number to control.
Parameter 2（HIGH）: Output level; HIGH represents high level (typically 3.3V on ESP32-P4), LOW represents low level (0V).
delay(1000);: Core Arduino function that pauses program execution for a specified number of milliseconds.
Parameter 1000: Pauses for 1000 milliseconds (i.e., 1 second).
Note: delay() is "blocking delay"—during the delay period, the ESP32-P4 suspends other operations until the delay completes.

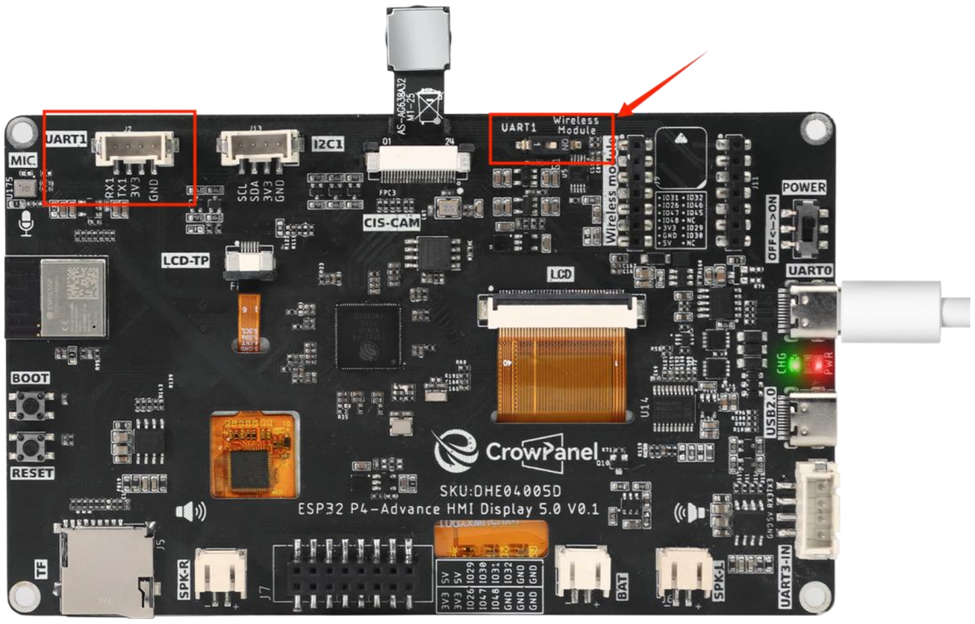
digitalWrite(PIN_LED, LOW);: Sets the PIN_LED pin to output low level, turning the LED off (assuming the LED hardware wiring is "high-level activated"; if it's low-level activated, the logic reverses).
delay(1000);: Pauses for another 1 second, maintaining the LED off state for 1 second.

Programming Steps

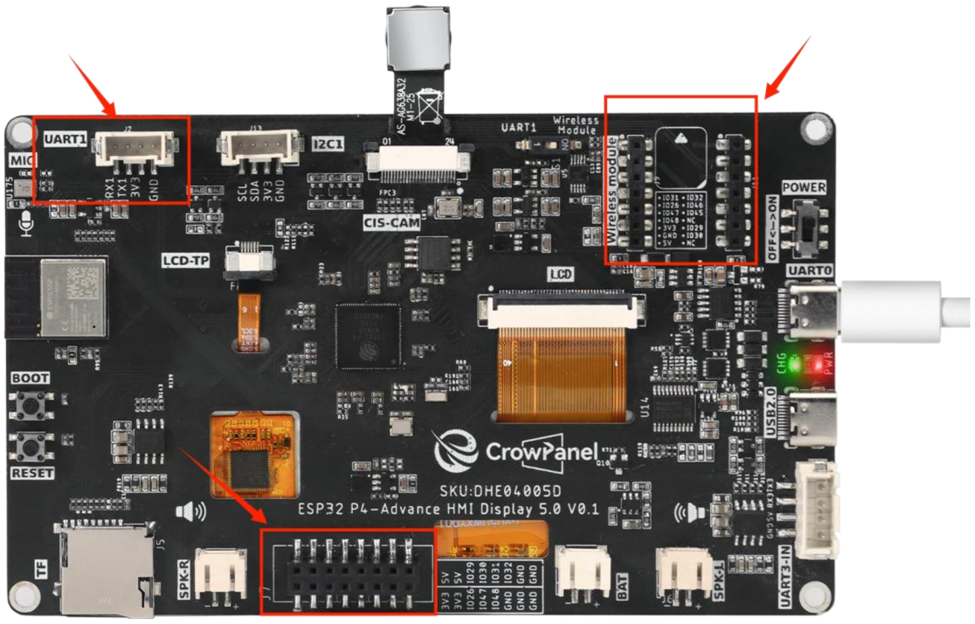
Now that the code is ready, we need to flash the ESP32-P4 to see the results in action. First, connect the Advance-P4 device to your computer host via a USB cable.



Then, switch the toggle switch on the 5-inch Advance-P4 to the UART1 position. Only in this way can the UART1 interface be used.



This is the design on the hardware side.



Switch to UART1 port:

Among the three interfaces shown in the figure, only the UART1 interface can be used at this time.
Alternatively, the expansion header at the bottom can also be used.
That is, either the UART1 interface or the expansion header can be used, but not both.

Switch to Wireless Module port:

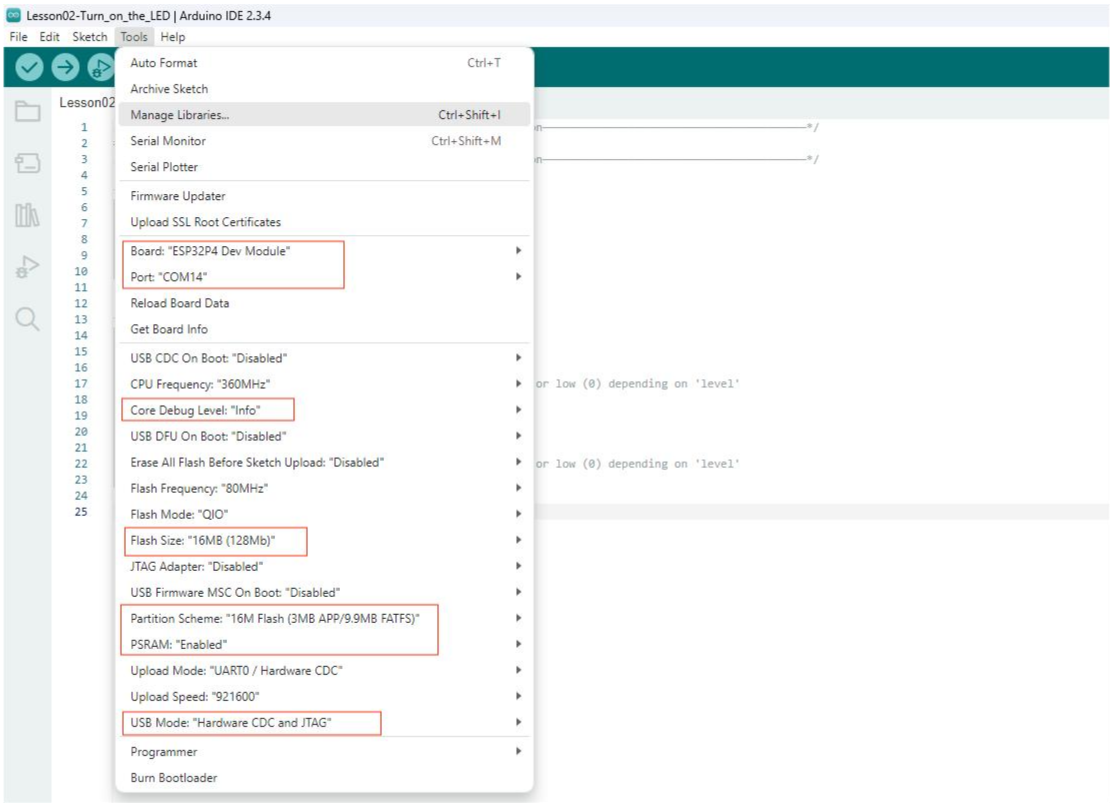
Among the three interfaces shown in the figure, only the wireless module can be used at this time.

Alternatively, the expansion header at the bottom can also be used.
That is, either the wireless module or the expansion header can be used, but not both.

Summary:
The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.
The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

Note: The H2 and C6 wireless modules can be used simultaneously with UART1.
The Lora, 2.4GHz, and WiFi-Halow wireless modules can be used with UART1, but not simultaneously.

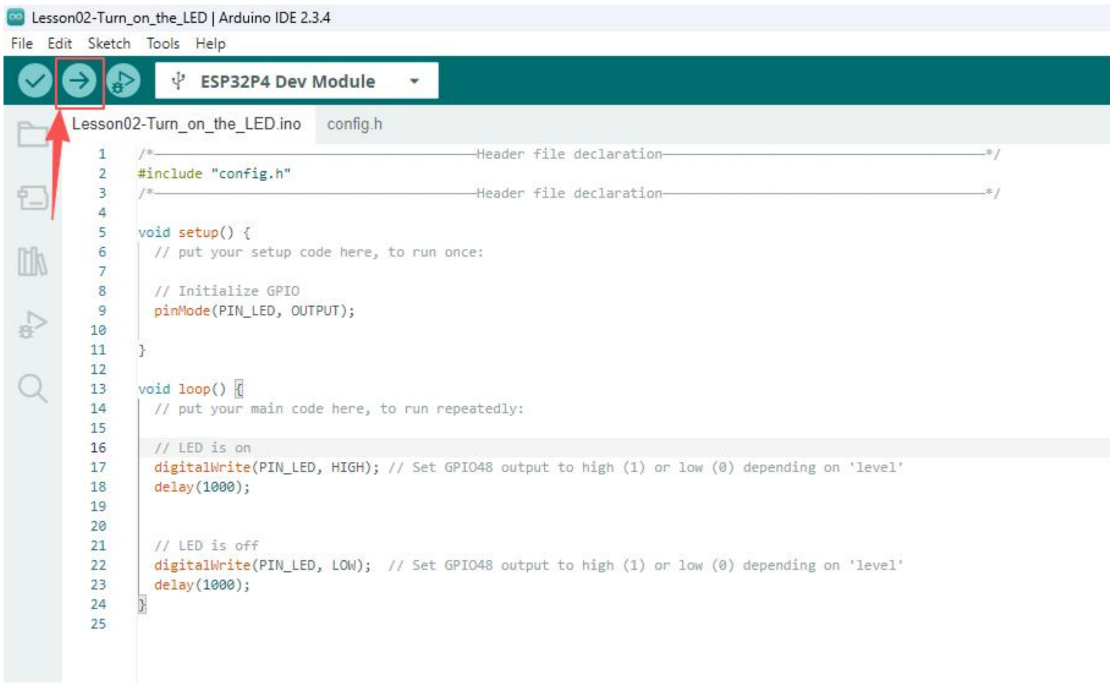
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

File Edit Sketch Tools Help

Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader

Then we compile and upload the code.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson02-Turn_on_the_LED.ino
config.h

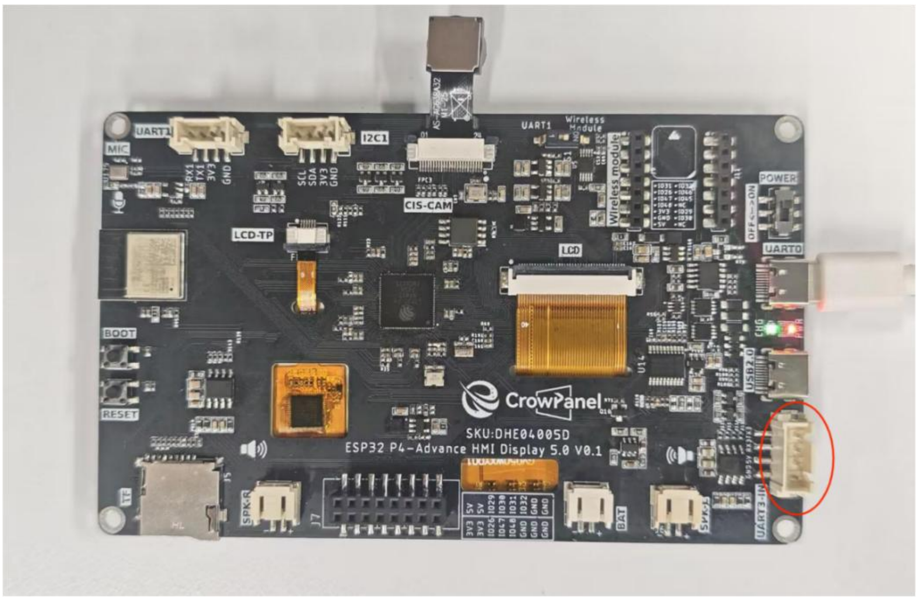
```
/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include "config.h"
/*————————————————————————————————————Header file declaration————————————————————————————————————*/

void setup() {
  // put your setup code here, to run once:

  // Initialize GPIO
  pinMode(PIN_LED, OUTPUT);

}

void loop() {
  // put your main code here, to run repeatedly:

  // LED is on
  digitalWrite(PIN_LED, HIGH); // Set GPIO48 output to high (1) or low (0) depending on 'level'
  delay(1000);


  // LED is off
  digitalWrite(PIN_LED, LOW);  // Set GPIO48 output to high (1) or low (0) depending on 'level'
  delay(1000);
}
```

After waiting a moment, you will see the LED connected to UART1 on your Advance-P4 blinking on for one second and off for one second, repeating this cycle continuously.

Lesson03---UART3-IN interface (external power supply)

Introduction

In this class, we will introduce the UART3-IN interface. There will be no code in this class. Based on the code from the previous class (which turned on the LED), we will explain to you what uses this UART3-IN interface has.

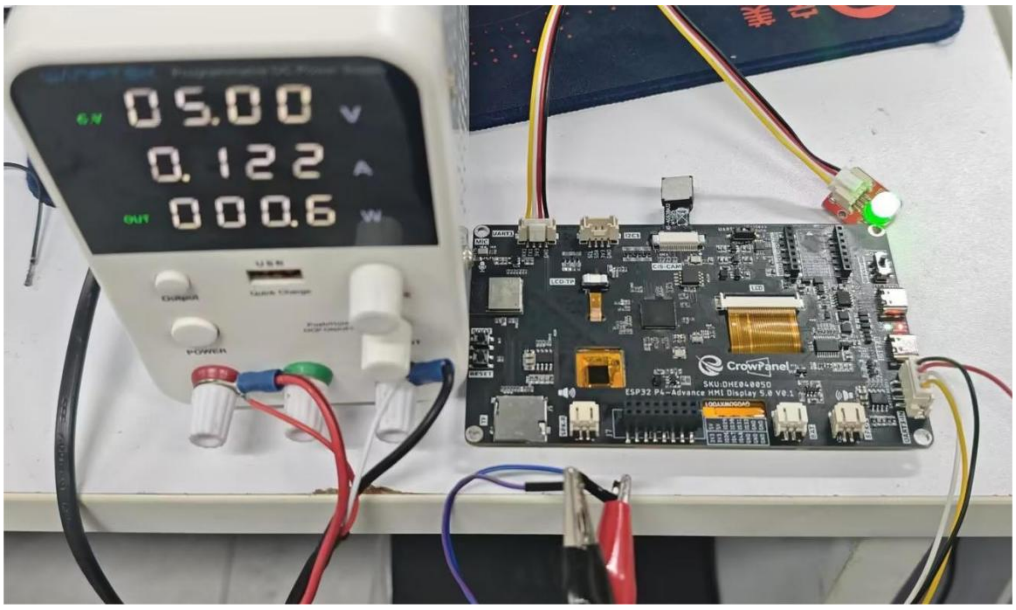


Learning Goals

1. Understand the function of the UART3-IN interface

Preview of the Result

Instead of powering the Advance-P4 through the UART0 and USB 2.0 interfaces, power it solely through the UART3-IN interface, and you will see the LED connected to the UART1 interface light up.



Hardware Used in This Lesson

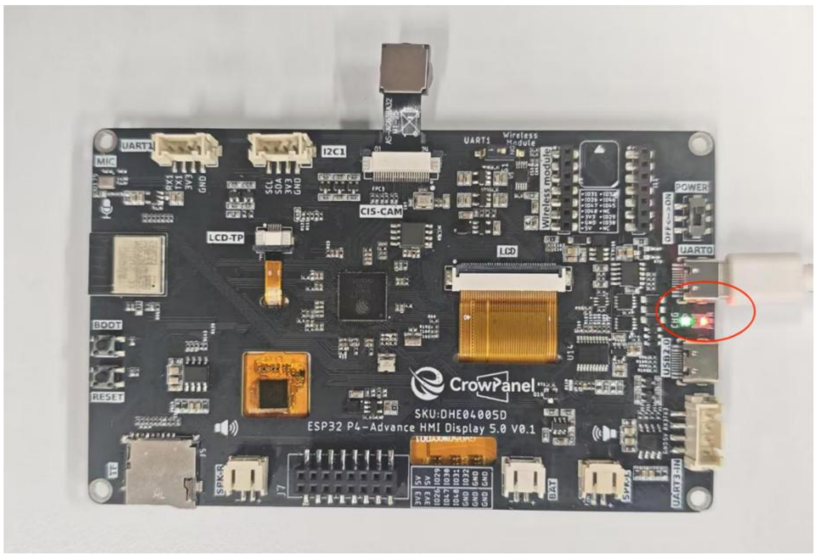
This lesson involves no hardware usage; it solely explains the function of the UART3-IN interface on the Advance-P4 development board.

Complete Code

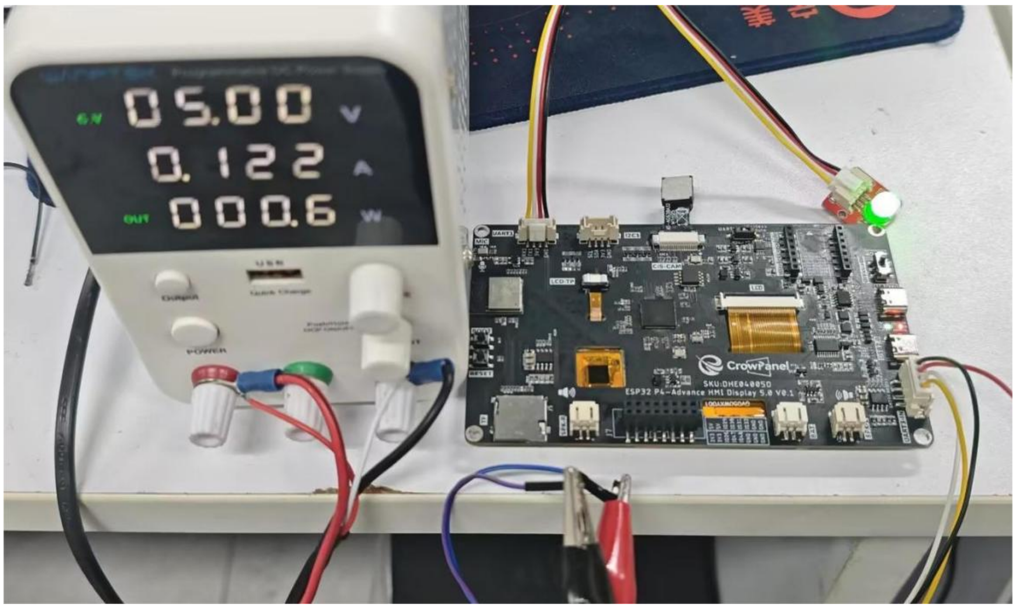
This lesson has no code.

Key Explanations

At this moment, everyone can see that the UART3-IN and UART0 interfaces. In the previous lesson, when we were burning the code, we learned that the UART0 pin is used for uploading the code. At the same time, you can also see that after connecting the UART0 interface, the power indicator next to it lights up, indicating that power supply is still available.



Then we come back to the UART3-IN interface. This interface is similar in function to the UART0 interface we just discussed. It can supply power, but it cannot upload code.
The UART0 interface is connected to the serial port burning chip, making code burning relatively convenient.
However, the UART3-IN interface does not have a serial port burning chip. It can only be used for power supply and serial port operations.
So, here we will explain how the UART3-IN interface can be used as a power supply function.
You need to prepare a power supply, along with two Dupont wires. One wire connects the VCC pin of UART3-IN to the positive terminal of the power supply, and the other wire connects the GND pin of UART3-IN to the negative terminal of the power supply.



Note: The voltage and current used here are provided by a programmable power supply. You only need to ensure that the externally supplied voltage is 5V and the current is 2A, then connect them to the corresponding VCC pin and GND pin on UART3-IN (connect the positive terminal to VCC and the negative terminal to GND).

Make sure your wires are connected correctly, then turn on the power switch to supply power.

At this point, you will be able to see the LED light we turned on in the last lesson. It is also blinking now, indicating that the power supply has been successful.
Of course, in addition to serving as an input power interface, USRT3-IN can also be used as a normal serial port. However, it should be noted that when connecting UART3-IN, since UART3-IN cannot provide power externally, the side connected to UART3-IN needs to be able to supply power itself.

Lesson04---Serial port usage

Introduction

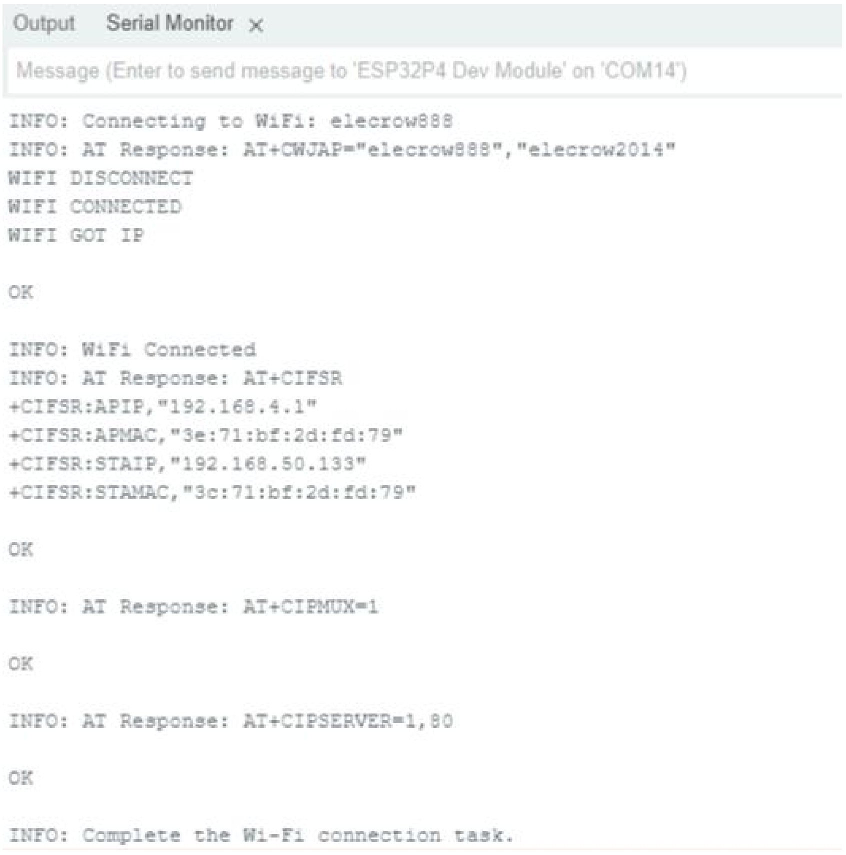
In this class, we will start teaching you how to use the serial port component. We will communicate with the Wi-Fi serial module through the UART1 interface on the Advance-P4.
The Advance-P4 connects to the Wi-Fi module via the serial port. After sending the AT command to the Wi-Fi module, it enables the Wi-Fi module to connect to the Wi-Fi network.

Learning Goals

1. Use the UART1 interface as a normal serial port for communication by connecting a serial communication module.
2. Understand the essence of serial communication.
3. Learn how to implement serial communication through code.

Preview of the Result

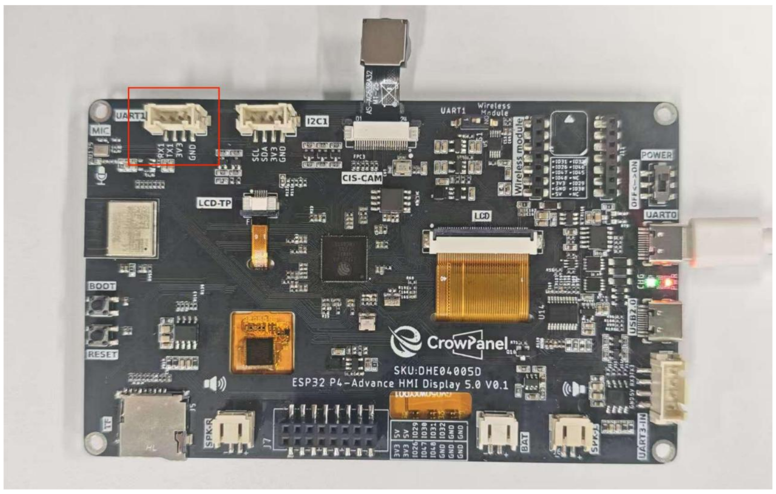
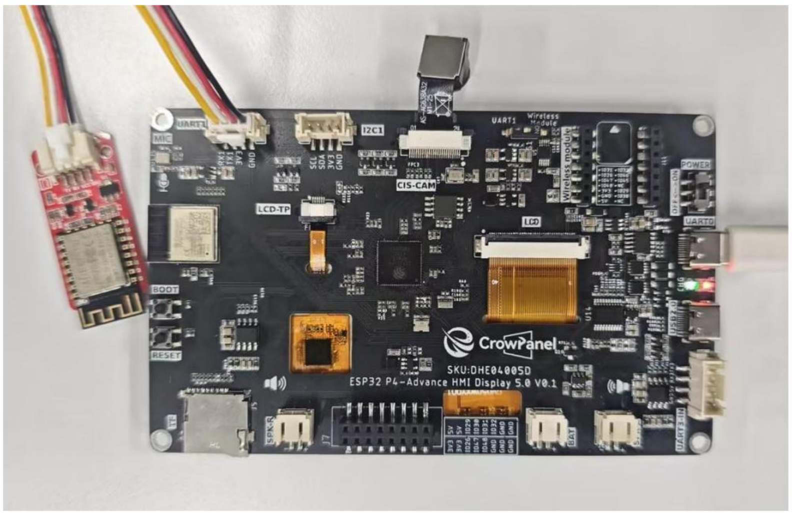
After running the code, you will be able to see the AT commands you sent on the monitor of Arduino-IDE, as well as the responses returned to you by the Wi-Fi module via the serial port.


Output
Serial Monitor ×
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
INFO: Connecting to WiFi: elecrow888
INFO: AT Response: AT+CWJAP="elecrow888","elecrow2014"
WIFI DISCONNECT
WIFI CONNECTED
WIFI GOT IP

OK

INFO: WiFi Connected
INFO: AT Response: AT+CIFSR
+CIFSR:APIP,"192.168.4.1"
+CIFSR:APMAC,"3e:71:bf:2d:fd:79"
+CIFSR:STAIP,"192.168.50.133"
+CIFSR:STAMAC,"3c:71:bf:2d:fd:79"

OK

INFO: AT Response: AT+CIPMUX=1

OK

INFO: AT Response: AT+CIPSERVER=1,80

OK

INFO: Complete the Wi-Fi connection task.
```

Hardware Used in This Lesson

The UART1 interface on the Advance-P4

Crowtail- Serial Wifi Description

The serial wifi module is based on ESP-12, which is an ultra-low power UART-WiFi module. It has excellent dimensions and ULP technology compared to other similar modules. The module is specially designed for mobile devices and the Internet of Things. Once the firmware is upgraded to the appropriate version, a compatible Android device can run the IOT.APK to do the following: control the PWM, I/O pin, or Serial communication. For example, you can use this module to transmit data with its serial port. It is easy to communicate with other device.

If you want to purchase, you can click the link below to learn more about this module.

Purchase link:https://www.elecrow.com/crowtail-serial-wifi-p-1265.html

Complete Code

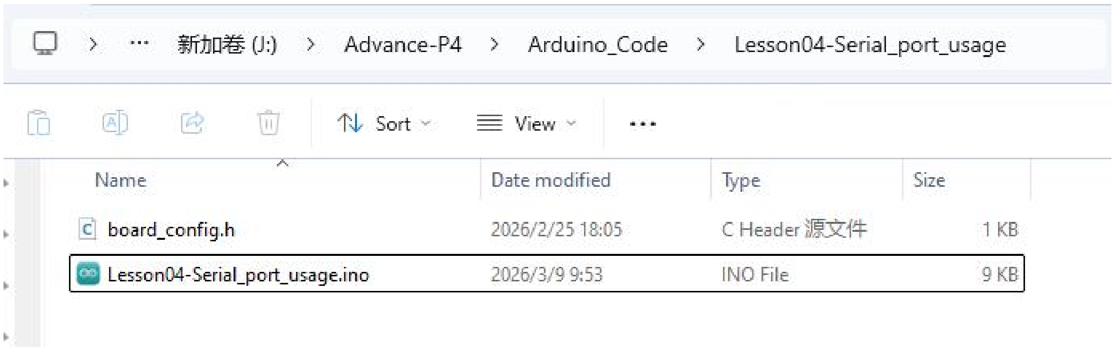
Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson04-Serial_port_usage

Key Explanations

The focus of this lesson is on how to use the serial port. Send commands to the ESP32-P4 chip via the serial port, and after the ESP32-P4 receives the commands, it controls the WiFi module connected to UART1 to establish a WiFi connection.
Once the connection is successful, it sends relevant status messages back to the PC via the serial port, enabling normal serial communication.

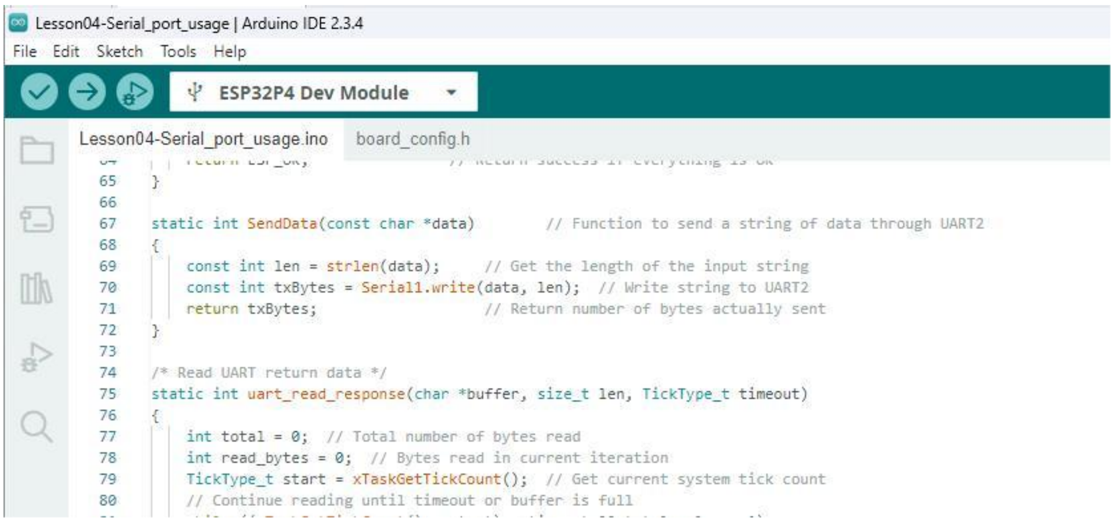
Next, let's understand how serial communication actually works and what conditions must be met to perform serial communication.

Double-click to open this lesson's code (the .ino file).


› ··· 新加卷 (J:) › Advance-P4 › Arduino_Code › Lesson04-Serial_port_usage
Sort
View
···

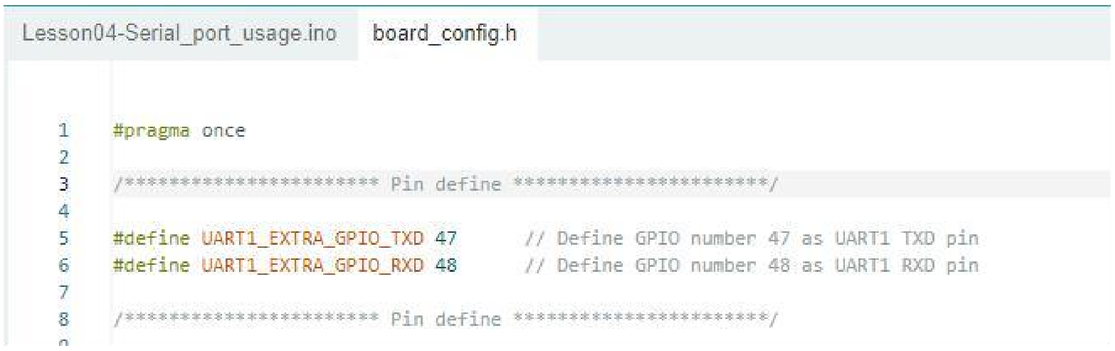
Name	Date modified	Type	Size
board_config.h	2026/2/25 18:05	C Header 源文件	1 KB
Lesson04-Serial_port_usage.ino	2026/3/9 9:53	INO File	9 KB

After opening, you can see the project's code and the configuration file board_config.h.


Lesson04-Serial_port_usage | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson04-Serial_port_usage.ino
board_config.h

```
}

static int SendData(const char *data)        // Function to send a string of data through UART2
{
    const int len = strlen(data);     // Get the length of the input string
    const int txBytes = Serial1.write(data, len);  // Write string to UART2
    return txBytes;                   // Return number of bytes actually sent
}

/* Read UART return data */
static int uart_read_response(char *buffer, size_t len, TickType_t timeout)
{
    int total = 0;  // Total number of bytes read
    int read_bytes = 0;  // Bytes read in current iteration
    TickType_t start = xTaskGetTickCount();  // Get current system tick count
    // Continue reading until timeout or buffer is full
```

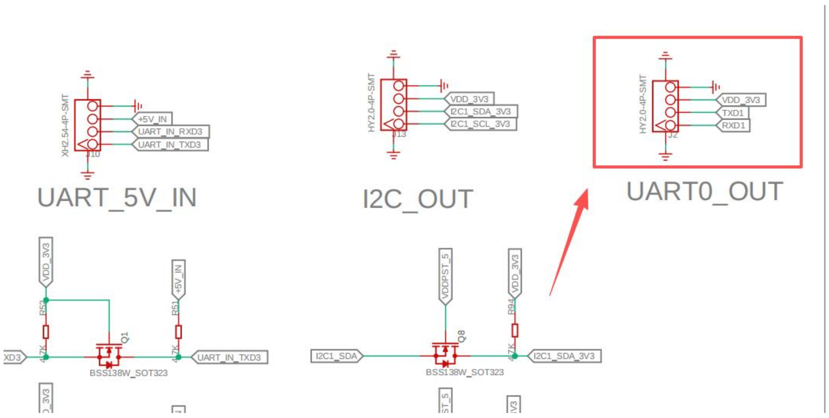
Let's first take a look at the board_config.h file.

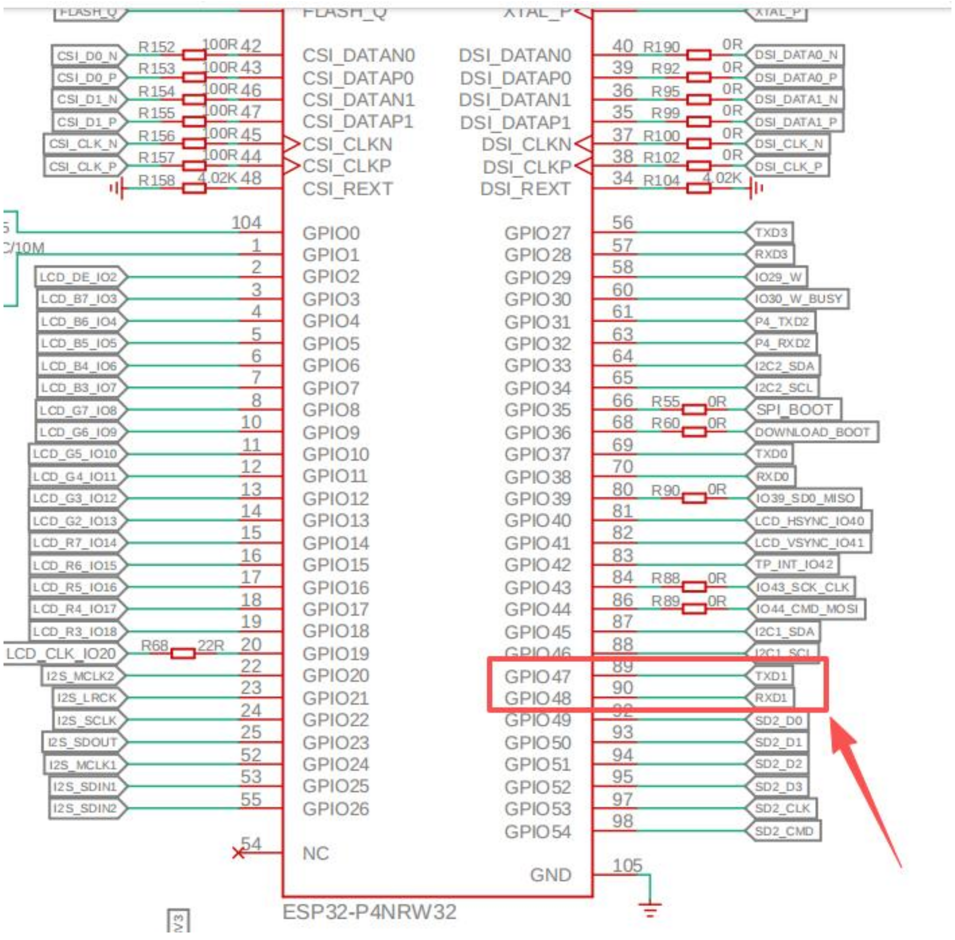


Lesson04-Serial_port_usage.ino board_config.h

```
#pragma once

/************************** Pin define ************************/

#define UART1_EXTRA_GPIO_TXD 47      // Define GPIO number 47 as UART1 TXD pin
#define UART1_EXTRA_GPIO_RXD 48      // Define GPIO number 48 as UART1 RXD pin

/************************** Pin define ************************/
```

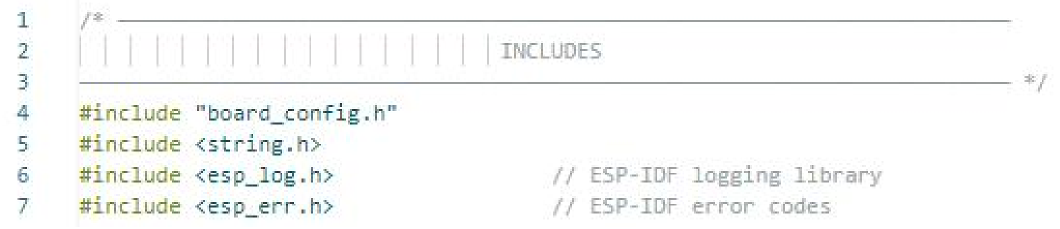
In this lesson, we are primarily controlling the WiFi module on the UART1 interface, specifically pins 47 and 48 at the UART1 interface.





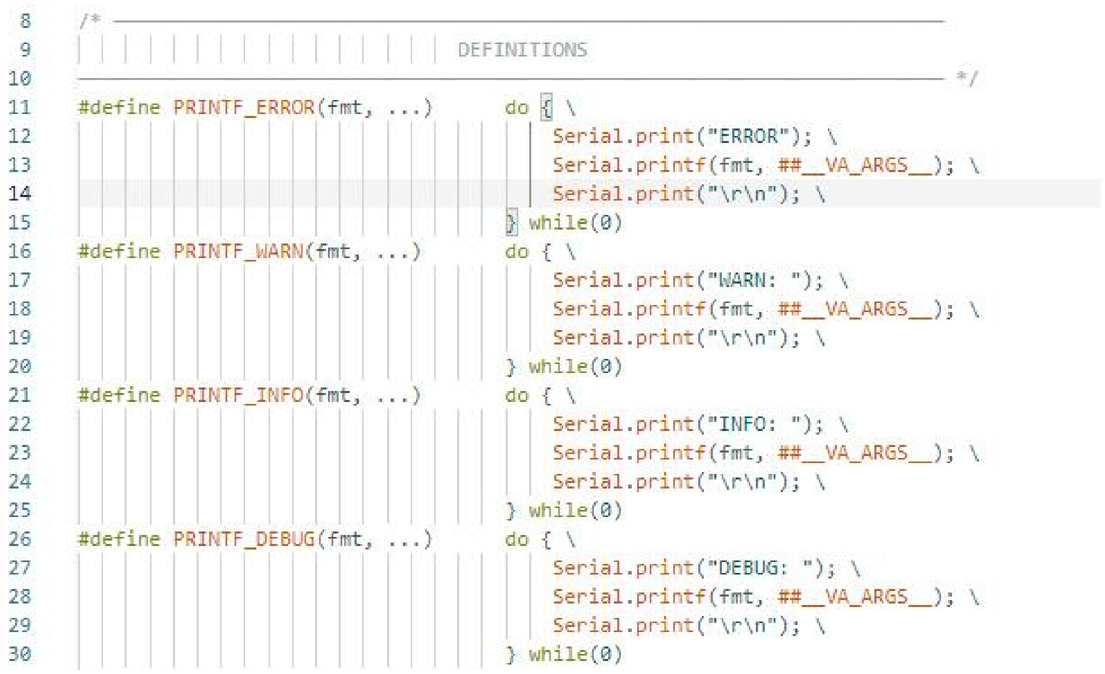
Now let's look back at the code and walk through it.

1. Header Files and Macro Definitions Section



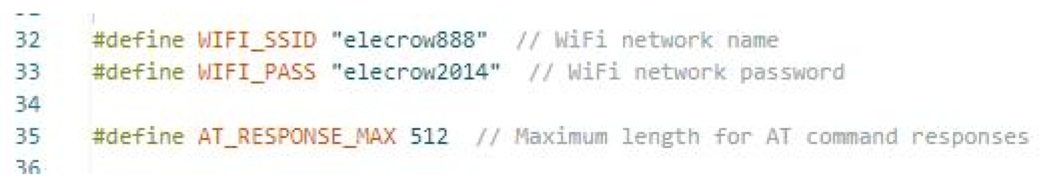
```
/* ────────────────────────────────────────────────────────
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | INCLUDES
─────────────────────────────────────────────────────────── */
#include "board_config.h"
#include <string.h>
#include <esp_log.h>                  // ESP-IDF logging library
#include <esp_err.h>                  // ESP-IDF error codes
```

board_config.h: Custom hardware configuration header file, primarily defines UART1 pins (such as UART1_EXTRA_GPIO_RXD/UART1_EXTRA_GPIO_TXD), must ensure consistency with hardware wiring;
string.h: C standard string library, provides string manipulation functions like strlen()/strstr()/snprintf();
esp_log.h/esp_err.h: ESP-IDF core libraries, provide logging output and error code definitions (such as ESP_OK) respectively.



```
/* ────────────────────────────────────────────────────────
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | DEFINITIONS
─────────────────────────────────────────────────────────── */
#define PRINTF_ERROR(fmt, ...)      do { \
                                        Serial.print("ERROR"); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
#define PRINTF_WARN(fmt, ...)       do { \
                                        Serial.print("WARN: "); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
#define PRINTF_INFO(fmt, ...)       do { \
                                        Serial.print("INFO: "); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
#define PRINTF_DEBUG(fmt, ...)      do { \
                                        Serial.print("DEBUG: "); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
```

Custom logging macros: wrap Serial.print() to implement different levels of log output (ERROR/WARN/INFO/DEBUG), more concise and unified than writing Serial.print directly;
do { ... } while(0): classic C macro definition syntax, ensures the macro executes as a "single statement" in any context (such as after an if statement), avoiding syntax errors;
##__VA_ARGS__: supports variable arguments, allowing the macro to receive multiple parameters like printf (e.g., PRINTF_INFO("Connecting to WiFi: %s", WIFI_SSID)).

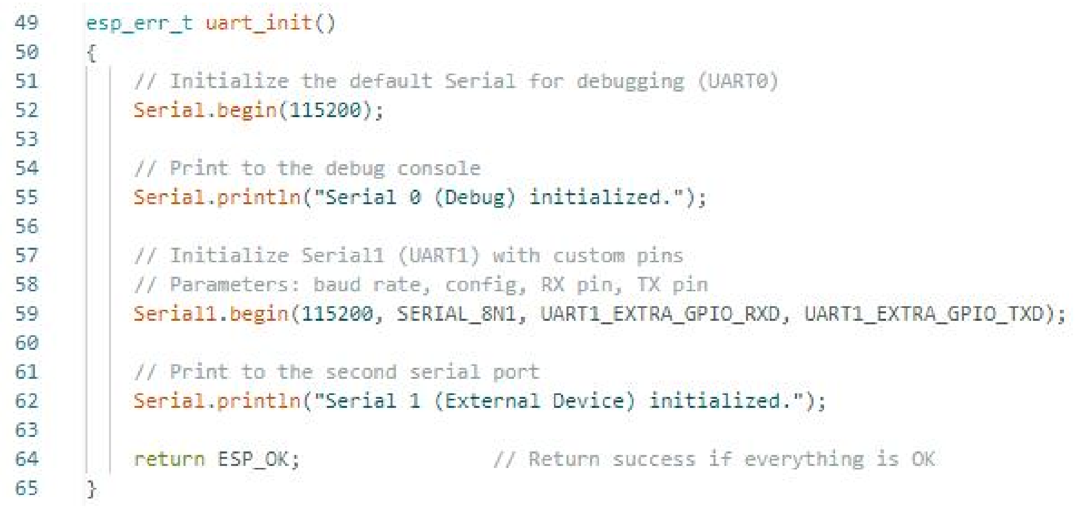


```
#define WIFI_SSID "elecrow888"   // WiFi network name
#define WIFI_PASS "elecrow2014"   // WiFi network password

#define AT_RESPONSE_MAX 512  // Maximum length for AT command responses
```

Constant definitions: centrally manage WiFi credentials and maximum AT response buffer length, no need to modify core logic for subsequent changes.

2. Core Function Area

2.1 UART Initialization Function uart_init()

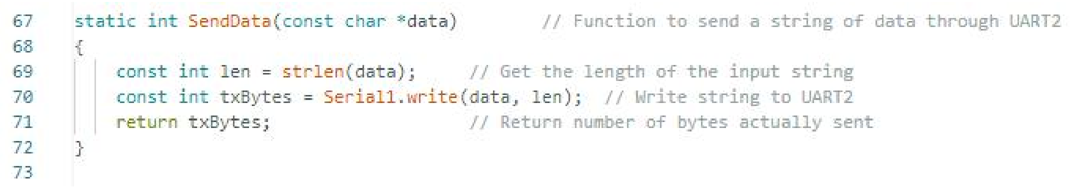


```
esp_err_t uart_init()
{
    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    // Print to the debug console
    Serial.println("Serial 0 (Debug) initialized.");

    // Initialize Serial1 (UART1) with custom pins
    // Parameters: baud rate, config, RX pin, TX pin
    Serial1.begin(115200, SERIAL_8N1, UART1_EXTRA_GPIO_RXD, UART1_EXTRA_GPIO_TXD);

    // Print to the second serial port
    Serial.println("Serial 1 (External Device) initialized.");

    return ESP_OK;                // Return success if everything is OK
}
```

Serial.begin(115200): initializes ESP32-P4 hardware UART0 (default serial port) with baud rate 115200, used for outputting debug logs;

Serial1.begin(...): initializes UART1, parameter explanation:
115200: baud rate (must match AT command baud rate of WiFi module);
SERIAL_8N1: serial format (8 data bits, no parity bit, 1 stop bit, default format for AT modules);
UART1_EXTRA_GPIO_RXD/UART1_EXTRA_GPIO_TXD: custom RX/TX pins (from board_config.h);
Returns ESP_OK: conforms to ESP-IDF error code specification, indicating successful initialization.

2.2 UART Send Function SendData()



```
static int SendData(const char *data)         // Function to send a string of data through UART2
{
    const int len = strlen(data);      // Get the length of the input string
    const int txBytes = Serial1.write(data, len);  // Write string to UART2
    return txBytes;                    // Return number of bytes actually sent
}
```

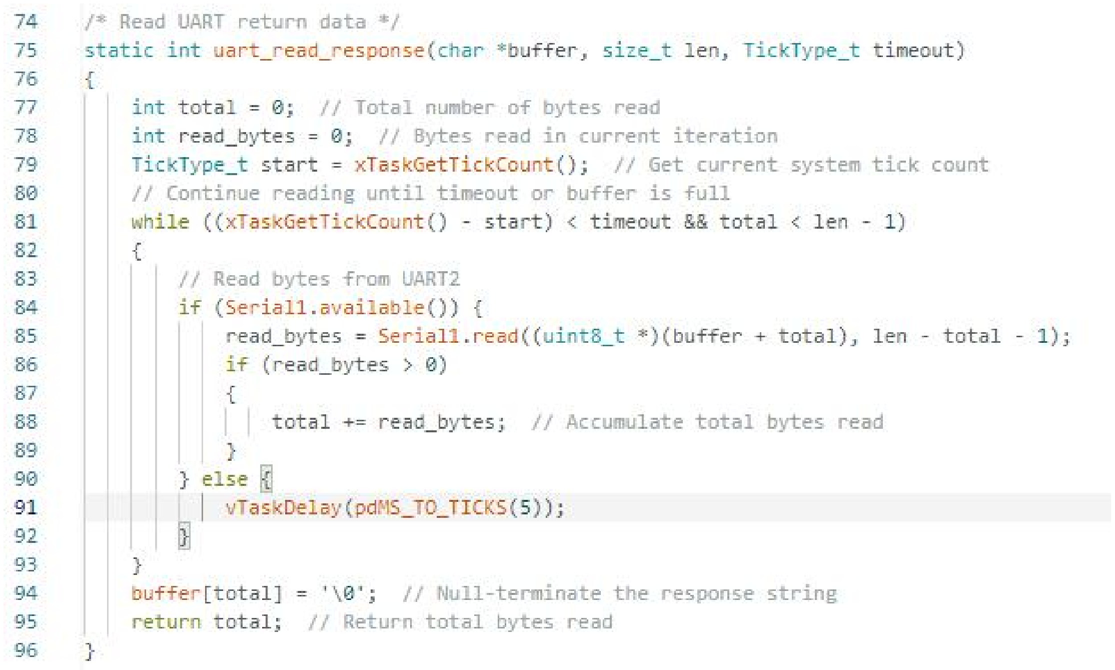
static: function is only visible within the current file, avoiding global naming conflicts;

strlen(data): calculates the length of the string to be sent (excluding the null terminator \0);

Serial1.write(data, len): writes specified length of byte data to UART1 (different from print, write is more suitable for binary/pure character transmission);

Return value: actual number of bytes sent, can be used to verify successful transmission (e.g., return value < len indicates transmission failure).

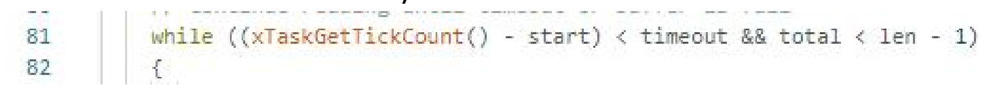
2.3 UART Receive Function uart_read_response()



```
/* Read UART return data */
static int uart_read_response(char *buffer, size_t len, TickType_t timeout)
{
    int total = 0;  // Total number of bytes read
    int read_bytes = 0;  // Bytes read in current iteration
    TickType_t start = xTaskGetTickCount();  // Get current system tick count
    // Continue reading until timeout or buffer is full
    while ((xTaskGetTickCount() - start) < timeout && total < len - 1)
    {
        // Read bytes from UART2
        if (Serial1.available()) {
            read_bytes = Serial1.read((uint8_t *)(buffer + total), len - total - 1);
            if (read_bytes > 0)
            {
                total += read_bytes;  // Accumulate total bytes read
            }
        } else {
            vTaskDelay(pdMS_TO_TICKS(5));
        }
    }
    buffer[total] = '\0';  // Null-terminate the response string
    return total;  // Return total bytes read
}
```

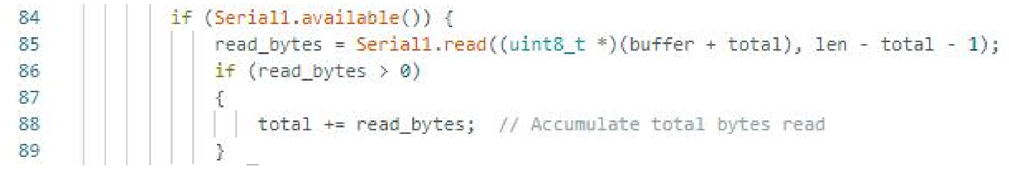
The function uart_read_response() serves to: continuously read data returned by the device from the UART serial port (Serial1) within a specified timeout period, store the read content into the buffer, and finally return the total number of bytes read.

The function first records the current system tick time as the start time, then enters a loop that continues attempting to read serial port data as long as timeout has not occurred and the buffer is not yet full.



```
    while ((xTaskGetTickCount() - start) < timeout && total < len - 1)
    {
```

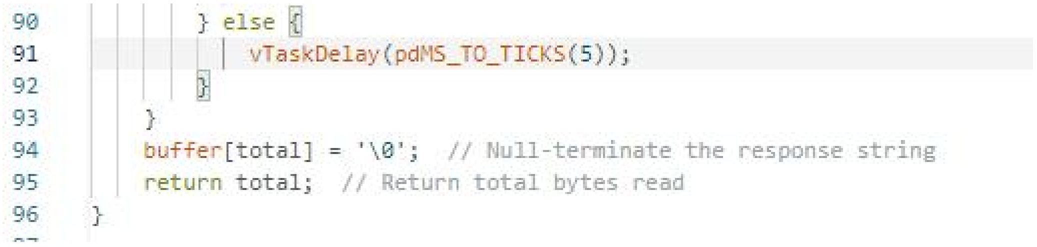
If there is data in the serial port buffer (Serial1.available() returns true), it calls Serial1.read() to read data into the position of buffer + total, thereby appending new data after existing data, and uses total to accumulate the number of bytes already read.



```
        if (Serial1.available()) {
            read_bytes = Serial1.read((uint8_t *)(buffer + total), len - total - 1);
            if (read_bytes > 0)
            {
                total += read_bytes;  // Accumulate total bytes read
            }
```

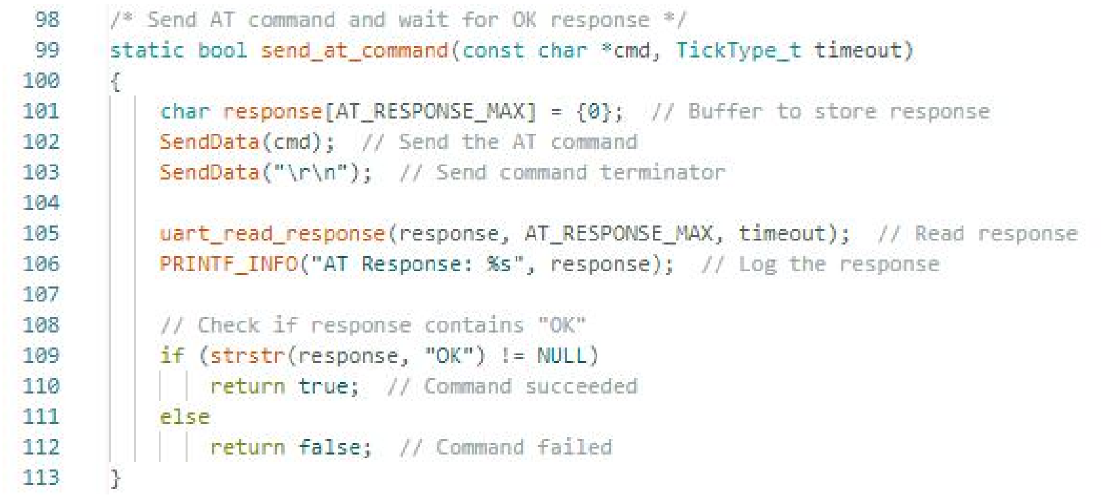
If no data is currently available to read, it pauses the task for a short period via vTaskDelay(5ms) to avoid occupying CPU resources.

After the loop ends, it appends the string terminator '\0' at the end of the data, making the buffer a valid C string. Finally, the function returns total, indicating how many bytes were read in total.



```c
            } else {
                vTaskDelay(pdMS_TO_TICKS(5));
            }
        }
        buffer[total] = '\0';  // Null-terminate the response string
        return total;  // Return total bytes read
    }
```

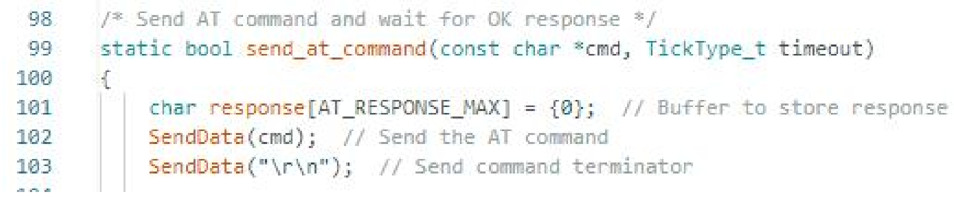
2.4 AT Command Send Function send_at_command()



```c
/* Send AT command and wait for OK response */
static bool send_at_command(const char *cmd, TickType_t timeout)
{
    char response[AT_RESPONSE_MAX] = {0};  // Buffer to store response
    SendData(cmd);  // Send the AT command
    SendData("\r\n");  // Send command terminator

    uart_read_response(response, AT_RESPONSE_MAX, timeout);  // Read response
    PRINTF_INFO("AT Response: %s", response);  // Log the response

    // Check if response contains "OK"
    if (strstr(response, "OK") != NULL)
        return true;  // Command succeeded
    else
        return false;  // Command failed
}
```

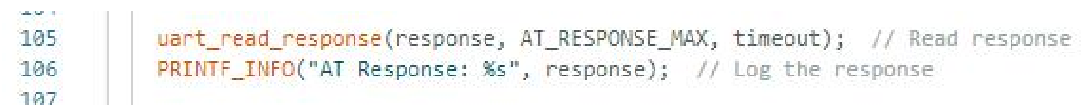
The function send_at_command() serves to: send an AT command to the serial port device, then read the data returned by the device within a specified timeout period, and determine whether the command executed successfully by checking if the returned content contains "OK".

The function first creates a response buffer to save the data returned by the module, then calls SendData(cmd) to send the AT command string, and additionally sends "\r\n" as the AT command terminator (most AT devices require carriage return and line feed as the command ending).



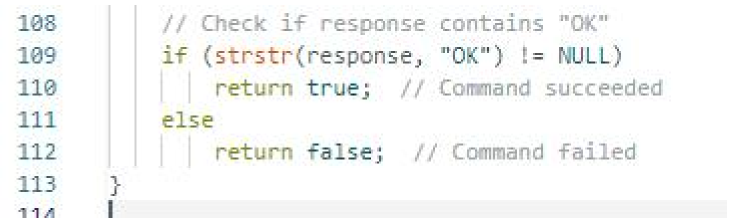
```c
/* Send AT command and wait for OK response */
static bool send_at_command(const char *cmd, TickType_t timeout)
{
    char response[AT_RESPONSE_MAX] = {0};  // Buffer to store response
    SendData(cmd);  // Send the AT command
    SendData("\r\n");  // Send command terminator
```

Then it calls the previously explained uart_read_response() function to read data returned by the module from the serial port within the specified timeout period and store it into response. After reading completes, it prints the entire response content via PRINTF_INFO for convenient debugging to see what the module actually returned.



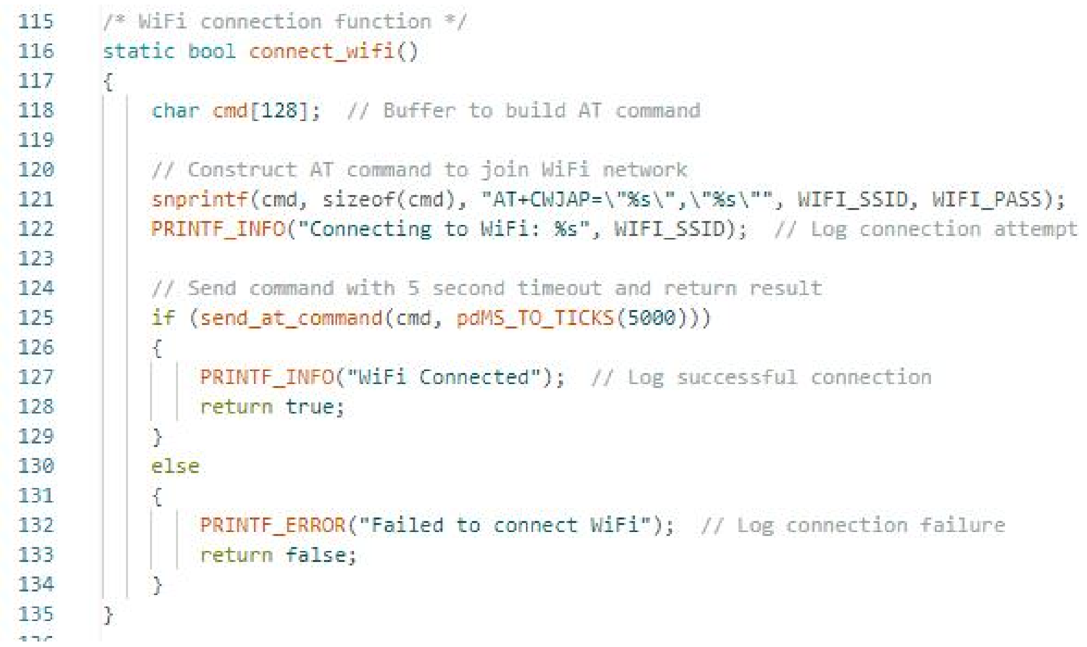
```c
    uart_read_response(response, AT_RESPONSE_MAX, timeout);  // Read response
    PRINTF_INFO("AT Response: %s", response);  // Log the response
```

Finally, the function uses strstr(response, "OK") to search for the "OK" substring in the returned string. If found, it indicates successful module execution and the function returns true; if not found, the command execution is considered failed and returns false.



```
        // Check if response contains "OK"
        if (strstr(response, "OK") != NULL)
            return true;  // Command succeeded
        else
            return false;  // Command failed
    }
```

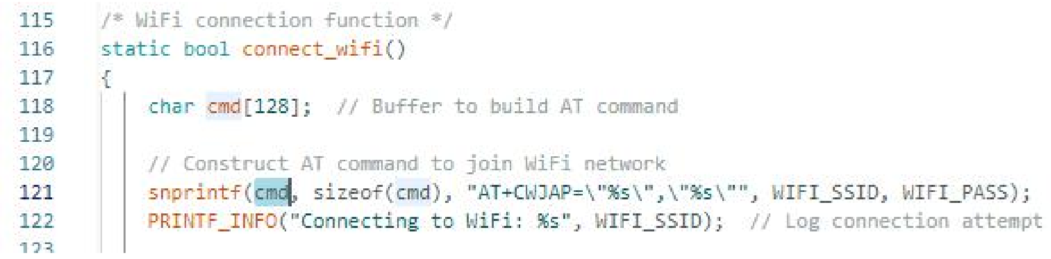
2.5 WiFi Connection Function connect_wifi()



```
/* WiFi connection function */
static bool connect_wifi()
{
    char cmd[128];  // Buffer to build AT command

    // Construct AT command to join WiFi network
    snprintf(cmd, sizeof(cmd), "AT+CWJAP=\"%s\",\"%s\"", WIFI_SSID, WIFI_PASS);
    PRINTF_INFO("Connecting to WiFi: %s", WIFI_SSID);  // Log connection attempt

    // Send command with 5 second timeout and return result
    if (send_at_command(cmd, pdMS_TO_TICKS(5000)))
    {
        PRINTF_INFO("WiFi Connected");  // Log successful connection
        return true;
    }
    else
    {
        PRINTF_ERROR("Failed to connect WiFi");  // Log connection failure
        return false;
    }
}
```

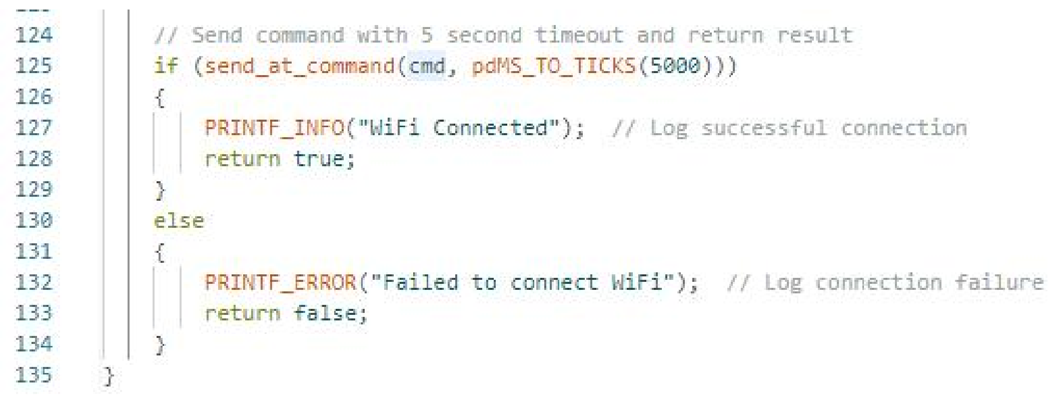
The connect_wifi() function serves to: use AT commands to make the WiFi module connect to a specified WiFi network, and determine whether the connection is successful based on the module's return result.

The function first defines a cmd character array to store the AT command to be sent, then uses snprintf() to generate the complete WiFi connection command AT+CWJAP="SSID","PASSWORD", where WIFI_SSID and WIFI_PASS are the WiFi name and password to connect to.



```
/* WiFi connection function */
static bool connect_wifi()
{
    char cmd[128];  // Buffer to build AT command

    // Construct AT command to join WiFi network
    snprintf(cmd, sizeof(cmd), "AT+CWJAP=\"%s\",\"%s\"", WIFI_SSID, WIFI_PASS);
    PRINTF_INFO("Connecting to WiFi: %s", WIFI_SSID);  // Log connection attempt
```

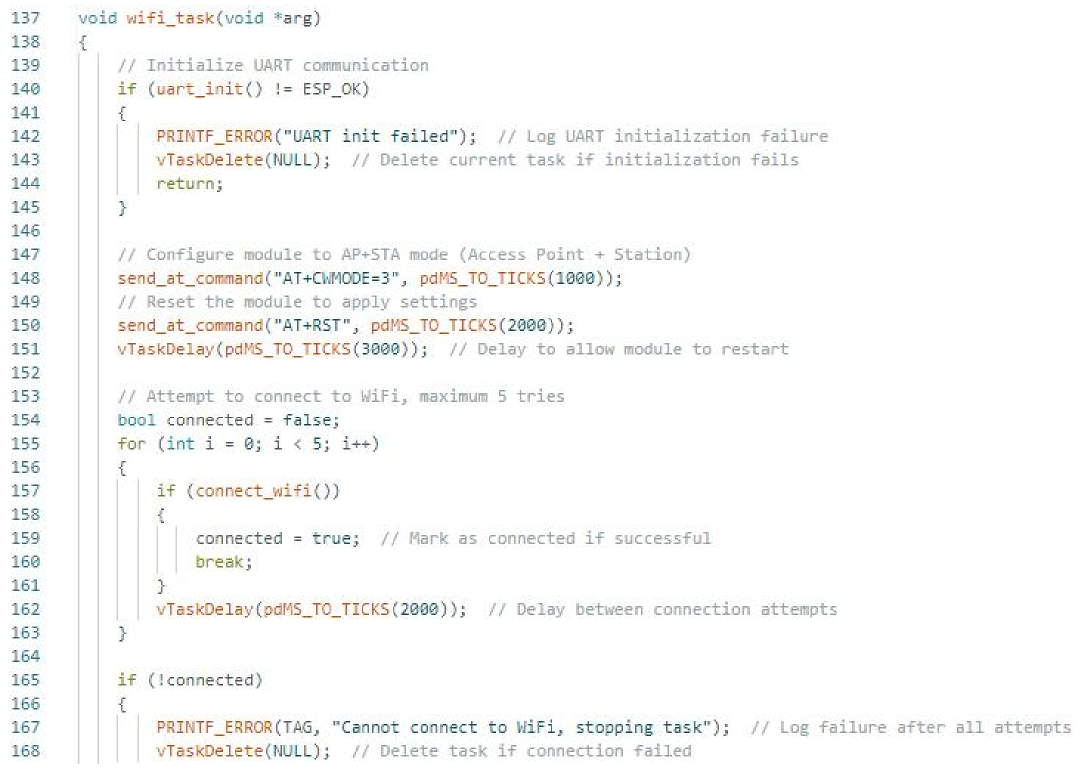
Then it prints a log via PRINTF_INFO, indicating which WiFi it is attempting to connect to. Next, it calls the previously encapsulated send_at_command() function to send this AT command, with a 5-second timeout set (pdMS_TO_TICKS(5000) converts 5000 milliseconds to FreeRTOS tick units).



```
        // Send command with 5 second timeout and return result
        if (send_at_command(cmd, pdMS_TO_TICKS(5000)))
        {
            PRINTF_INFO("WiFi Connected");  // Log successful connection
            return true;
        }
        else
        {
            PRINTF_ERROR("Failed to connect WiFi");  // Log connection failure
            return false;
        }
    }
```

If the module's returned data contains "OK", send_at_command() returns true, and the function prints "WiFi Connected" and returns true to indicate successful connection; if "OK" is not returned, it prints the error log "Failed to connect WiFi" and returns false to indicate connection failure.

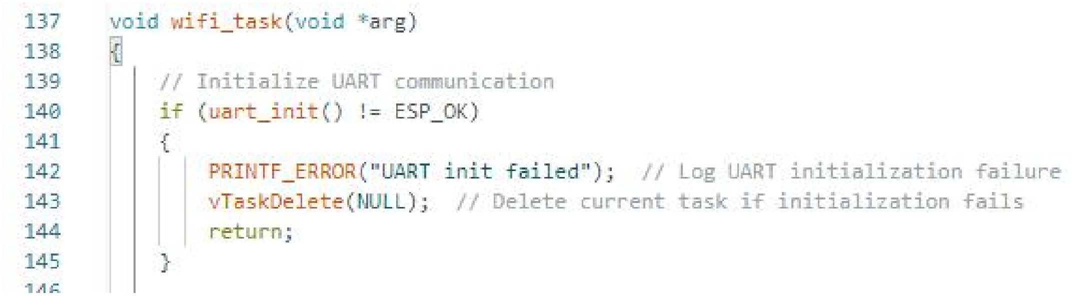
2.6 Task Function wifi_task()



```
void wifi_task(void *arg)
{
    // Initialize UART communication
    if (uart_init() != ESP_OK)
    {
        PRINTF_ERROR("UART init failed");  // Log UART initialization failure
        vTaskDelete(NULL);  // Delete current task if initialization fails
        return;
    }

    // Configure module to AP+STA mode (Access Point + Station)
    send_at_command("AT+CWMODE=3", pdMS_TO_TICKS(1000));
    // Reset the module to apply settings
    send_at_command("AT+RST", pdMS_TO_TICKS(2000));
    vTaskDelay(pdMS_TO_TICKS(3000));  // Delay to allow module to restart

    // Attempt to connect to WiFi, maximum 5 tries
    bool connected = false;
    for (int i = 0; i < 5; i++)
    {
        if (connect_wifi())
        {
            connected = true;  // Mark as connected if successful
            break;
        }
        vTaskDelay(pdMS_TO_TICKS(2000));  // Delay between connection attempts
    }

    if (!connected)
    {
        PRINTF_ERROR(TAG, "Cannot connect to WiFi, stopping task");  // Log failure after all attempts
        vTaskDelete(NULL);  // Delete task if connection failed
```

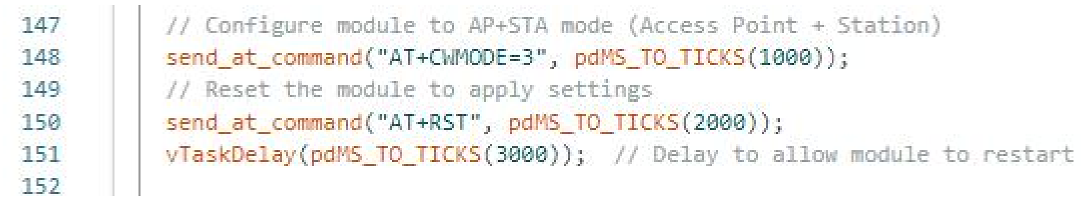
The wifi_task() function is a FreeRTOS task function whose main purpose is to: initialize the serial port, configure the WiFi module, connect to WiFi, and start a TCP server for network communication.

At the beginning of the function, it calls uart_init() to initialize serial communication. If initialization fails, it prints an error log and uses vTaskDelete(NULL) to delete the current task, preventing the task from continuing to run.



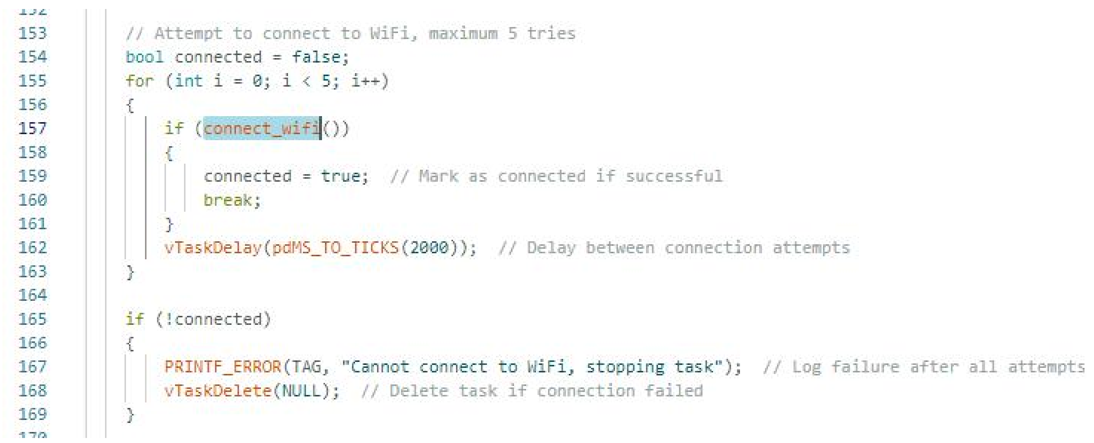
```
void wifi_task(void *arg)
{
    // Initialize UART communication
    if (uart_init() != ESP_OK)
    {
        PRINTF_ERROR("UART init failed");  // Log UART initialization failure
        vTaskDelete(NULL);  // Delete current task if initialization fails
        return;
    }
```

Then it uses send_at_command("AT+CWMODE=3") to set the WiFi module to AP+STA mode (can act as both hotspot and connect to router), then sends AT+RST to restart the module and delays 3 seconds to wait for the module to reboot.



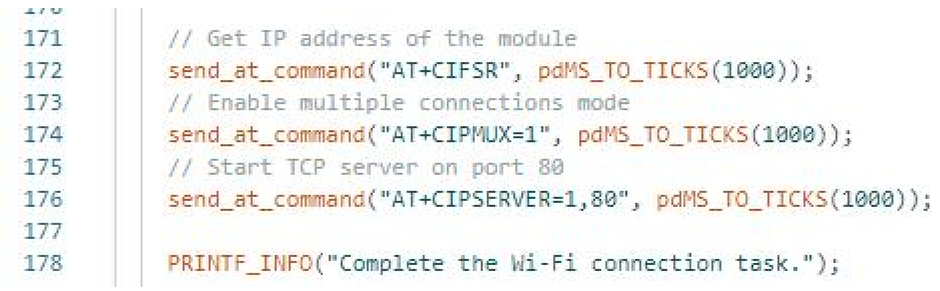
```
    // Configure module to AP+STA mode (Access Point + Station)
    send_at_command("AT+CWMODE=3", pdMS_TO_TICKS(1000));
    // Reset the module to apply settings
    send_at_command("AT+RST", pdMS_TO_TICKS(2000));
    vTaskDelay(pdMS_TO_TICKS(3000));  // Delay to allow module to restart

```

Once the module is ready, the program enters a loop of up to 5 attempts to call connect_wifi() to connect to the specified WiFi. Each failure waits 2 seconds before retrying. If all 5 attempts fail, it prints an error message and deletes the task; if the connection succeeds, it continues with subsequent network configuration.



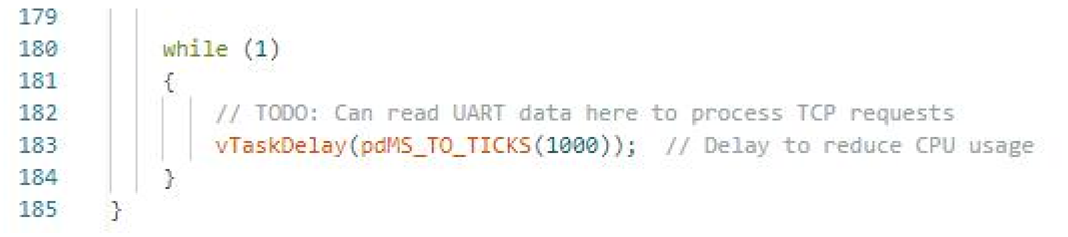
```
    // Attempt to connect to WiFi, maximum 5 tries
    bool connected = false;
    for (int i = 0; i < 5; i++)
    {
        if (connect_wifi())
        {
            connected = true;  // Mark as connected if successful
            break;
        }
        vTaskDelay(pdMS_TO_TICKS(2000));  // Delay between connection attempts
    }

    if (!connected)
    {
        PRINTF_ERROR(TAG, "Cannot connect to WiFi, stopping task");  // Log failure after all attempts
        vTaskDelete(NULL);  // Delete task if connection failed
    }
```

Then the program sends AT+CIFSR to obtain the module's IP address, sends AT+CIPMUX=1 to enable multiple connection mode, and finally sends AT+CIPSERVER=1,80 to start a TCP server on port 80 (similar to a simple network service endpoint).



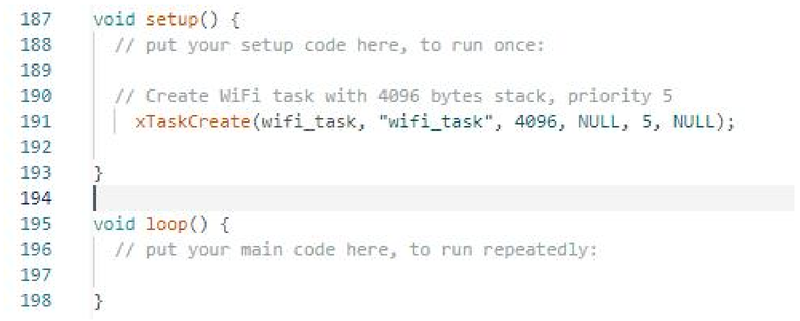
```
    // Get IP address of the module
    send_at_command("AT+CIFSR", pdMS_TO_TICKS(1000));
    // Enable multiple connections mode
    send_at_command("AT+CIPMUX=1", pdMS_TO_TICKS(1000));
    // Start TCP server on port 80
    send_at_command("AT+CIPSERVER=1,80", pdMS_TO_TICKS(1000));

    PRINTF_INFO("Complete the Wi-Fi connection task.");
```

After completing these steps, it prints a message indicating the WiFi connection task is finished, then enters an infinite loop while(1) with a one-second delay per iteration to reduce CPU usage. This loop is typically used for subsequently reading UART data and processing requests from TCP clients.



```
    while (1)
    {
        // TODO: Can read UART data here to process TCP requests
        vTaskDelay(pdMS_TO_TICKS(1000));  // Delay to reduce CPU usage
    }
}
```

2.7 Arduino Framework Entry Functions



```
void setup() {
  // put your setup code here, to run once:

  // Create WiFi task with 4096 bytes stack, priority 5
  xTaskCreate(wifi_task, "wifi_task", 4096, NULL, 5, NULL);

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

This code is the entry point of the Arduino program. Its purpose is to create a FreeRTOS task to run WiFi functionality when the system starts, while the main loop() no longer executes any logic.

In the setup() function, the program calls xTaskCreate() to create a task named "wifi_task". The actual function this task runs is the wifi_task() explained earlier.

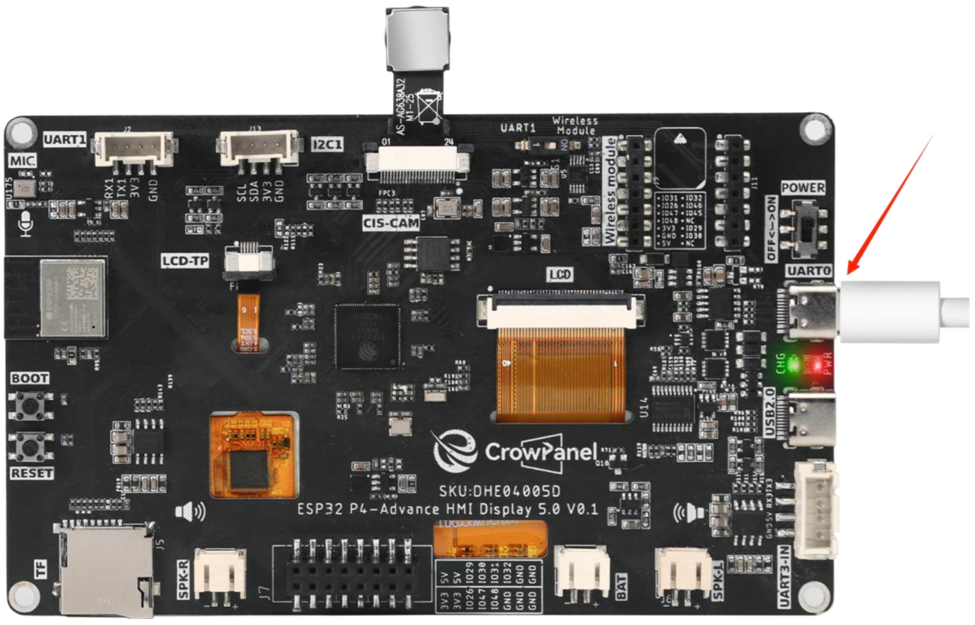
It allocates 4096 bytes of stack space for this task, passes NULL as the parameter, sets task priority to 5, and the last parameter is NULL indicating no need to save the task handle.

This way, after system startup, the FreeRTOS scheduler will run wifi_task, which completes UART initialization, WiFi connection, TCP server startup, and other operations.

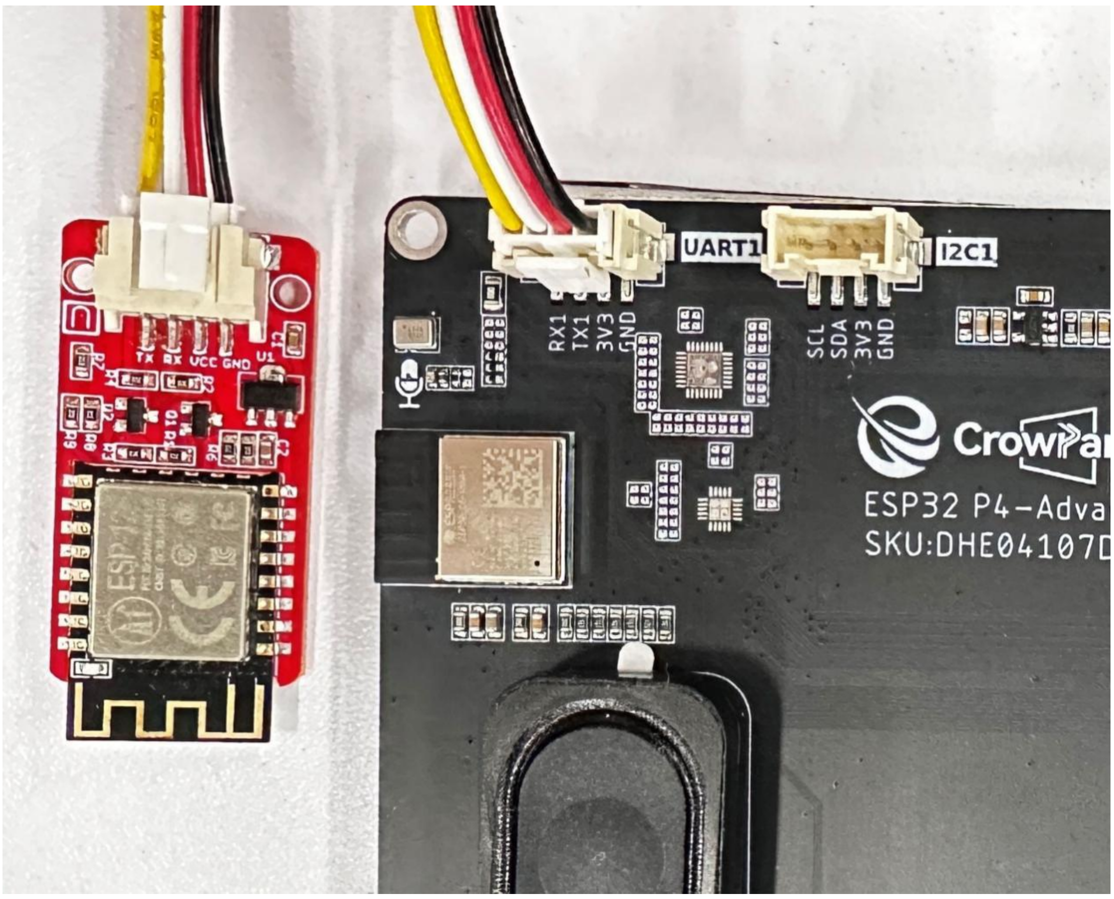
The loop() function remains empty because when using FreeRTOS, main logic is typically placed in individual tasks rather than inside Arduino's loop() cycle.

Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.

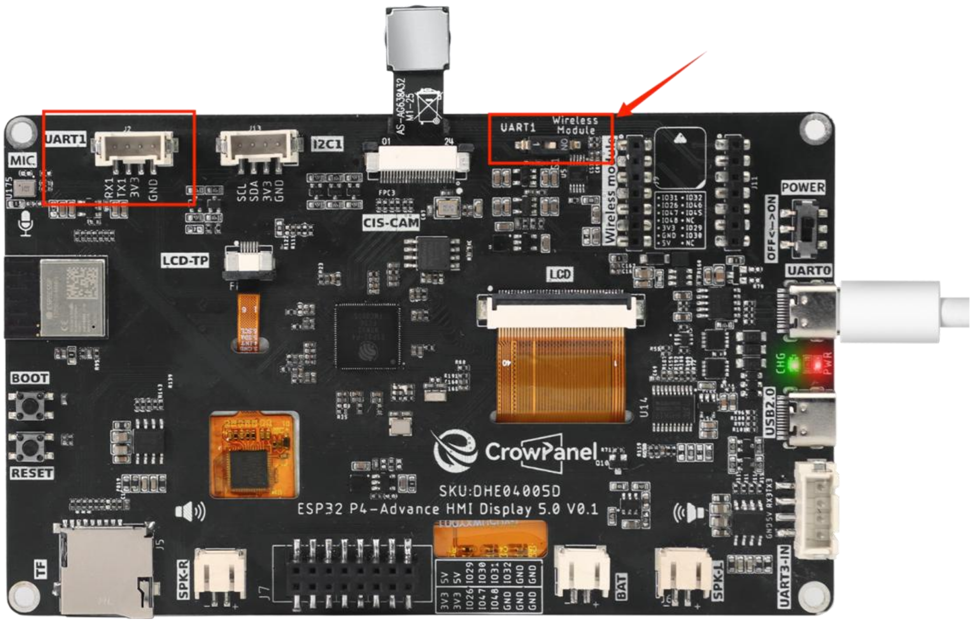


Then, connect an ESP8266 wifi module to the UART1 interface.
(Connect the VCC of UART1 interface to the VCC pin of the wifi module)
(Connect the GND of UART1 interface to the GND pin of the wifi module)
(Turn the TX of UART1 interface to the RX pin of the wifi module) (Cross connection)
(Turn the RX of UART1 interface to the TX pin of the wifi module) (Cross connection)

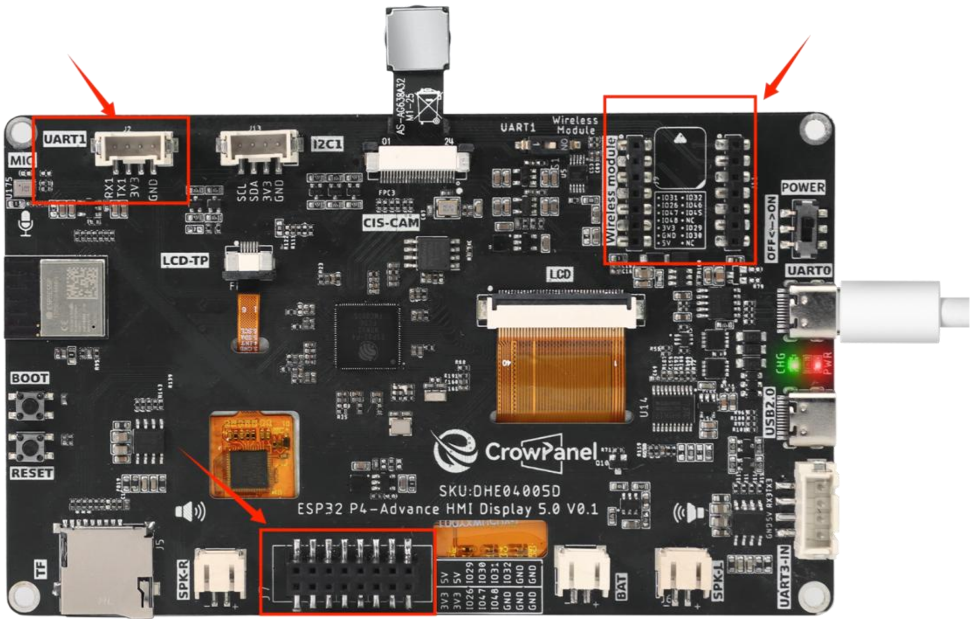


Then, switch the toggle switch on the 5-inch Advance-P4 to the UART1 position.

Only in this way can the UART1 interface be used.



This is the design on the hardware side.



Switch to UART1 port:
Among the three interfaces shown in the figure, only the UART1 interface can be used at this time.
Alternatively, the expansion header at the bottom can also be used.
That is, either the UART1 interface or the expansion header can be used, but not both.

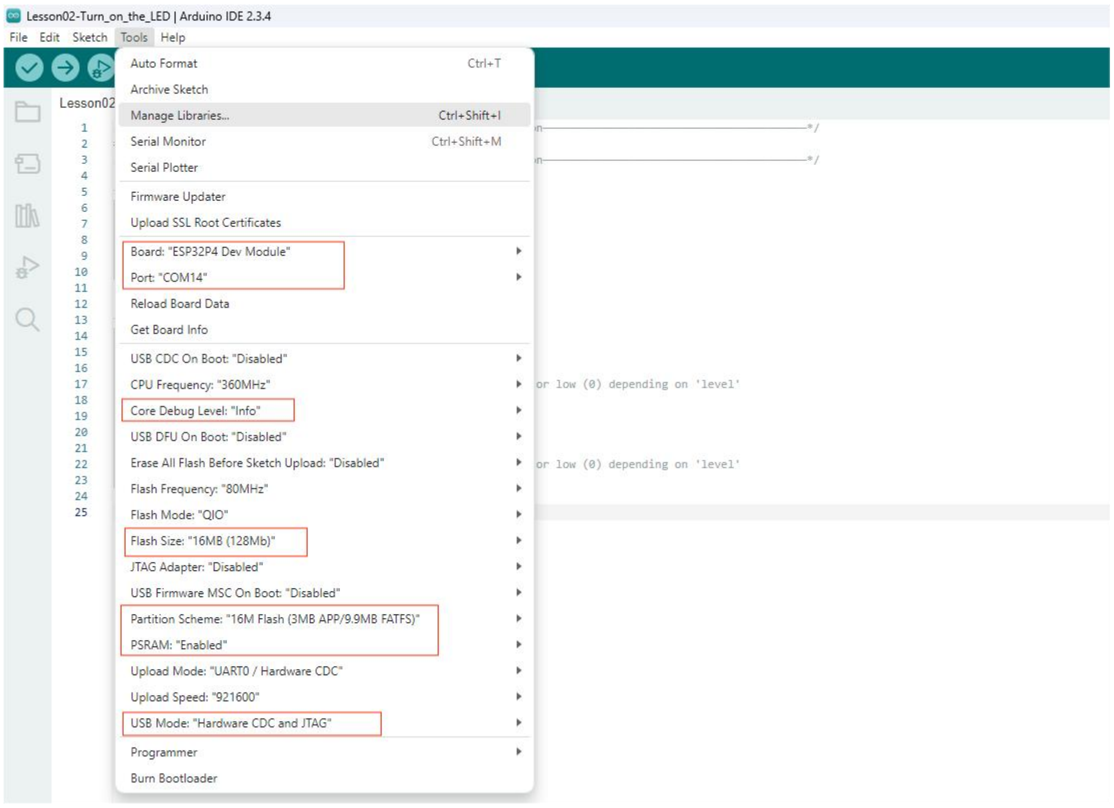
Switch to Wireless Module port:
Among the three interfaces shown in the figure, only the wireless module can be used at this time.
Alternatively, the expansion header at the bottom can also be used.

That is, either the wireless module or the expansion header can be used, but not both.

Summary:
The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.
The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

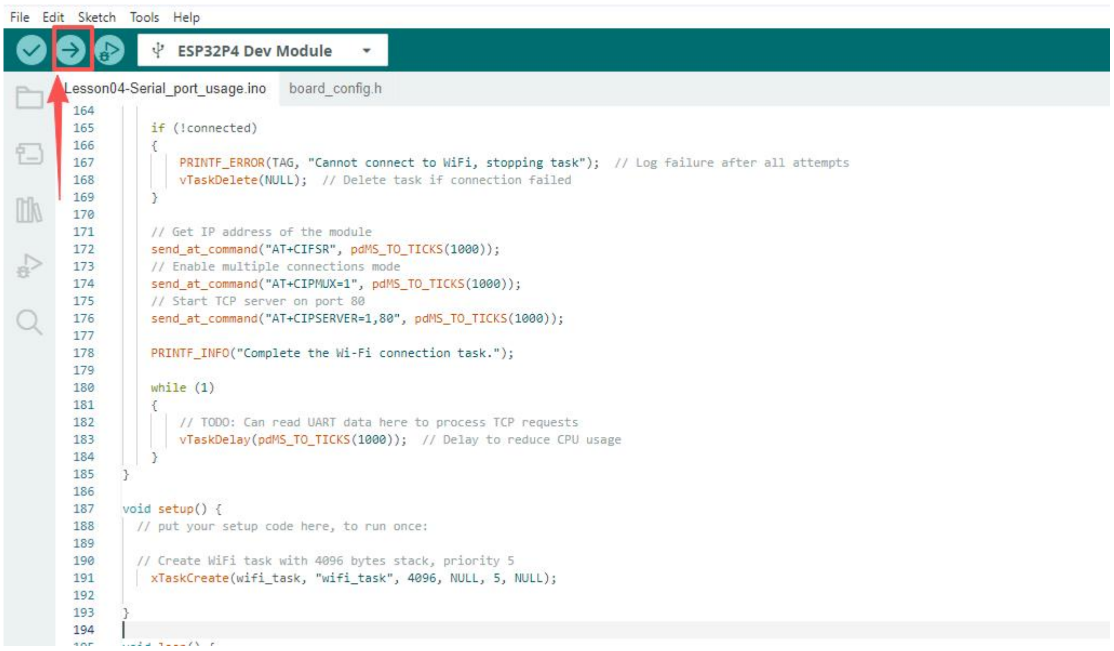
Note: The H2 and C6 wireless modules can be used simultaneously with UART1.
The Lora, 2.4GHz, and WiFi-Halow wireless modules can be used with UART1, but not simultaneously.

Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4
File Edit Sketch Tools Help
Lesson02
Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Reload Board Data
Get Board Info
Programmer
Burn Bootloader

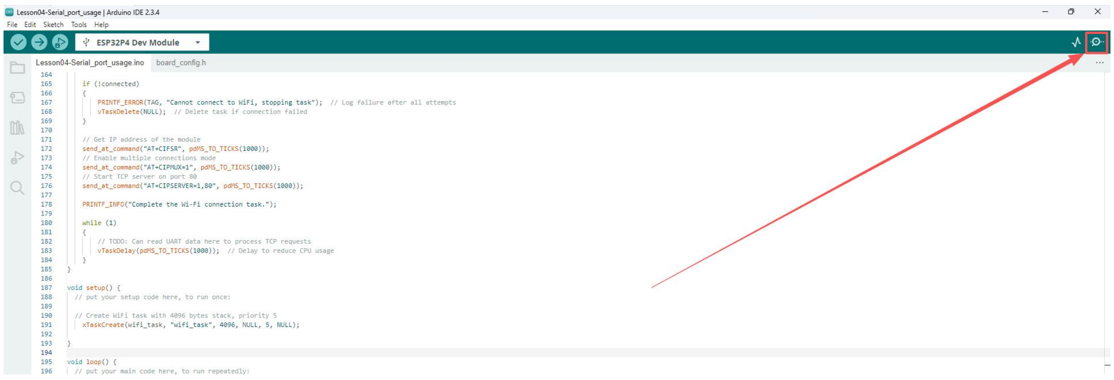
Board: "ESP32P4 Dev Module"
Port: "COM14"
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"

Then we compile and upload the code.



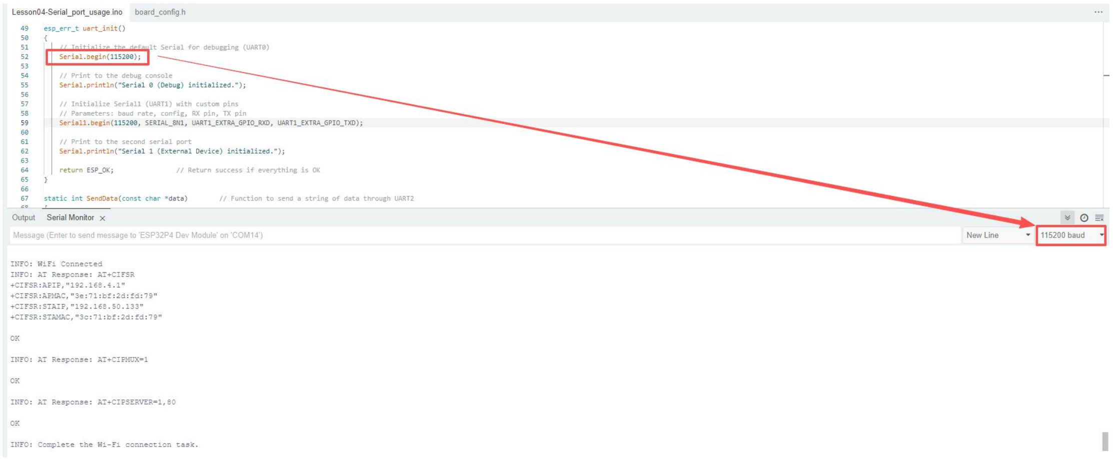
```
    if (!connected)
    {
        PRINTF_ERROR(TAG, "Cannot connect to WiFi, stopping task");  // Log failure after all attempts
        vTaskDelete(NULL);  // Delete task if connection failed
    }

    // Get IP address of the module
    send_at_command("AT+CIFSR", pdMS_TO_TICKS(1000));
    // Enable multiple connections mode
    send_at_command("AT+CIPMUX=1", pdMS_TO_TICKS(1000));
    // Start TCP server on port 80
    send_at_command("AT+CIPSERVER=1,80", pdMS_TO_TICKS(1000));

    PRINTF_INFO("Complete the Wi-Fi connection task.");

    while (1)
    {
        // TODO: Can read UART data here to process TCP requests
        vTaskDelay(pdMS_TO_TICKS(1000));  // Delay to reduce CPU usage
    }
}

void setup() {
  // put your setup code here, to run once:

  // Create WiFi task with 4096 bytes stack, priority 5
  xTaskCreate(wifi_task, "wifi_task", 4096, NULL, 5, NULL);

}
```

After the code upload completes, you can open the Serial Monitor built into the Arduino-IDE.


Lesson04-Serial_port_usage | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson04-Serial_port_usage.ino
board_config.h

```
    if (!connected)
    {
        PRINTF_ERROR(TAG, "Cannot connect to WiFi, stopping task");  // Log failure after all attempts
        vTaskDelete(NULL);  // Delete task if connection failed
    }

    // Get IP address of the module
    send_at_command("AT+CIFSR", pdMS_TO_TICKS(1000));
    // Enable multiple connections mode
    send_at_command("AT+CIPMUX=1", pdMS_TO_TICKS(1000));
    // Start TCP server on port 80
    send_at_command("AT+CIPSERVER=1,80", pdMS_TO_TICKS(1000));

    PRINTF_INFO("Complete the Wi-Fi connection task.");

    while (1)
    {
        // TODO: Can read UART data here to process TCP requests
        vTaskDelay(pdMS_TO_TICKS(1000));  // Delay to reduce CPU usage
    }
}

void setup() {
  // put your setup code here, to run once:

  // Create WiFi task with 4096 bytes stack, priority 5
  xTaskCreate(wifi_task, "wifi_task", 4096, NULL, 5, NULL);

}

void loop() {
  // put your main code here, to run repeatedly:
```

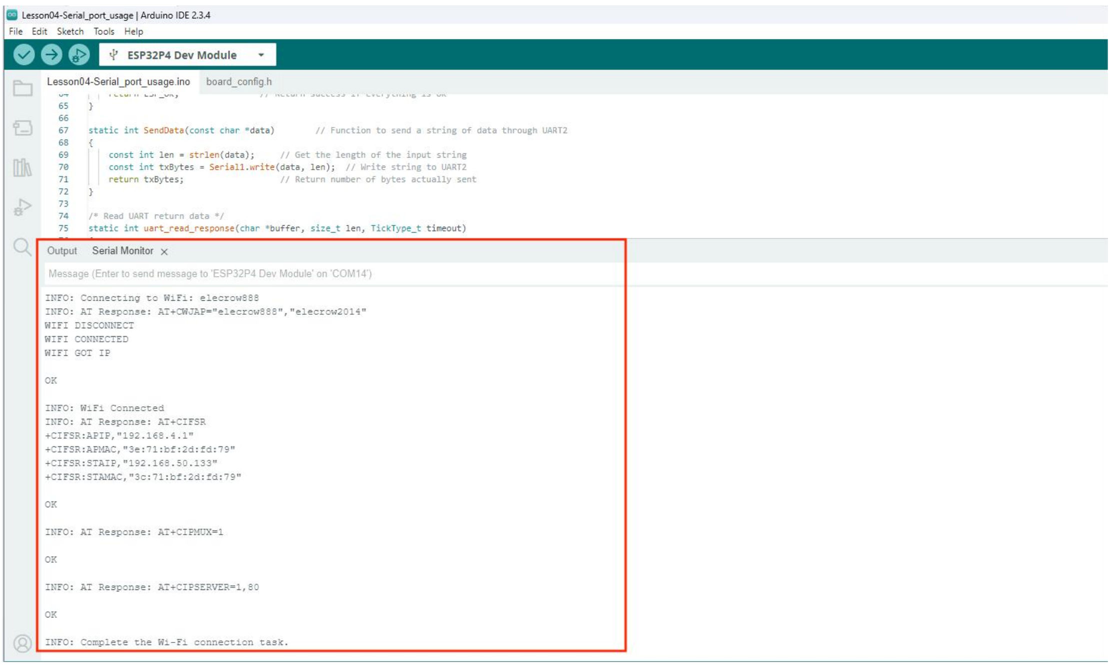
Then set the baud rate required by the serial port, keeping it consistent with what you set in your code.


Lesson04-Serial_port_usage.ino
board_config.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')
New Line
115200 baud

```
esp_err_t uart_init()
{
    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    // Print to the debug console
    Serial.println("Serial 0 (Debug) initialized.");

    // Initialize Serial1 (UART1) with custom pins
    // Parameters: baud rate, config, RX pin, TX pin
    Serial1.begin(115200, SERIAL_8N1, UART1_EXTRA_GPIO_RXD, UART1_EXTRA_GPIO_TXD);

    // Print to the second serial port
    Serial.println("Serial 1 (External Device) initialized.");

    return ESP_OK;              // Return success if everything is OK
}

static int SendData(const char *data)       // Function to send a string of data through UART2
```

```
INFO: WiFi Connected
INFO: AT Response: AT+CIFSR
+CIFSR:APIP,"192.168.4.1"
+CIFSR:APMAC,"3e:71:bf:2d:fd:79"
+CIFSR:STAIP,"192.168.50.133"
+CIFSR:STAMAC,"3c:71:bf:2d:fd:79"

OK

INFO: AT Response: AT+CIPMUX=1

OK

INFO: AT Response: AT+CIPSERVER=1,80

OK

INFO: Complete the Wi-Fi connection task.
```

This way you can see the communication with the WiFi module in progress, thereby connecting to the WiFi module.


Lesson04-Serial_port_usage | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson04-Serial_port_usage.ino
board_config.h
Output
Serial Monitor

```
}

static int SendData(const char *data)        // Function to send a string of data through UART2
{
    const int len = strlen(data);     // Get the length of the input string
    const int txBytes = Serial1.write(data, len);  // Write string to UART2
    return txBytes;                   // Return number of bytes actually sent
}

/* Read UART return data */
static int uart_read_response(char *buffer, size_t len, TickType_t timeout)
```

Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
INFO: Connecting to WiFi: elecrow888
INFO: AT Response: AT+CWJAP="elecrow888","elecrow2014"
WIFI DISCONNECT
WIFI CONNECTED
WIFI GOT IP

OK

INFO: WiFi Connected
INFO: AT Response: AT+CIFSR
+CIFSR:APIP,"192.168.4.1"
+CIFSR:APMAC,"3e:71:bf:2d:fd:79"
+CIFSR:STAIP,"192.168.50.133"
+CIFSR:STAMAC,"3c:71:bf:2d:fd:79"

OK

INFO: AT Response: AT+CIPMUX=1

OK

INFO: AT Response: AT+CIPSERVER=1,80

OK

INFO: Complete the Wi-Fi connection task.
```

Lesson05--- Touchscreen

Introduction

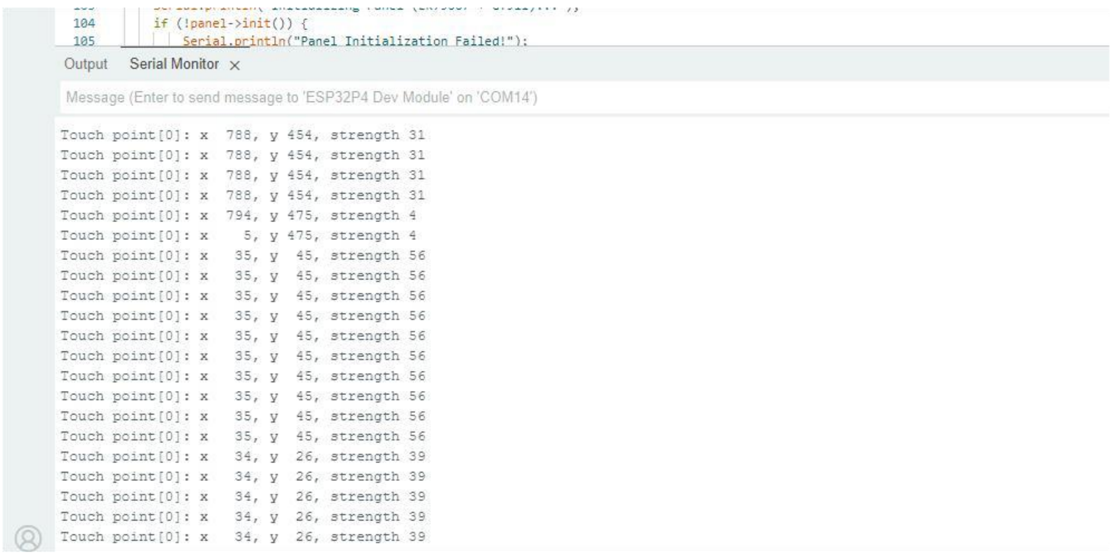
In this lesson, we will learn how to use the touch screen functionality of the CrowPanel Advanced ESP32-P4 HMI AI Display. When you tap the screen, you will see the touch coordinates displayed in the Serial Monitor.

Learning Goals

1. Understand the fundamental principles of touch screens
2. How to install library files in the Arduino-IDE
3. Implement touch functionality using code

Preview of the Result

After running the code, you will be able to see the coordinates returned by the ESP32-P4 to you through the monitor on the Arduino-IDE at the moment when you touched the screen.



```
    if (!panel->init()) {
        Serial.println("Panel Initialization Failed!");
```

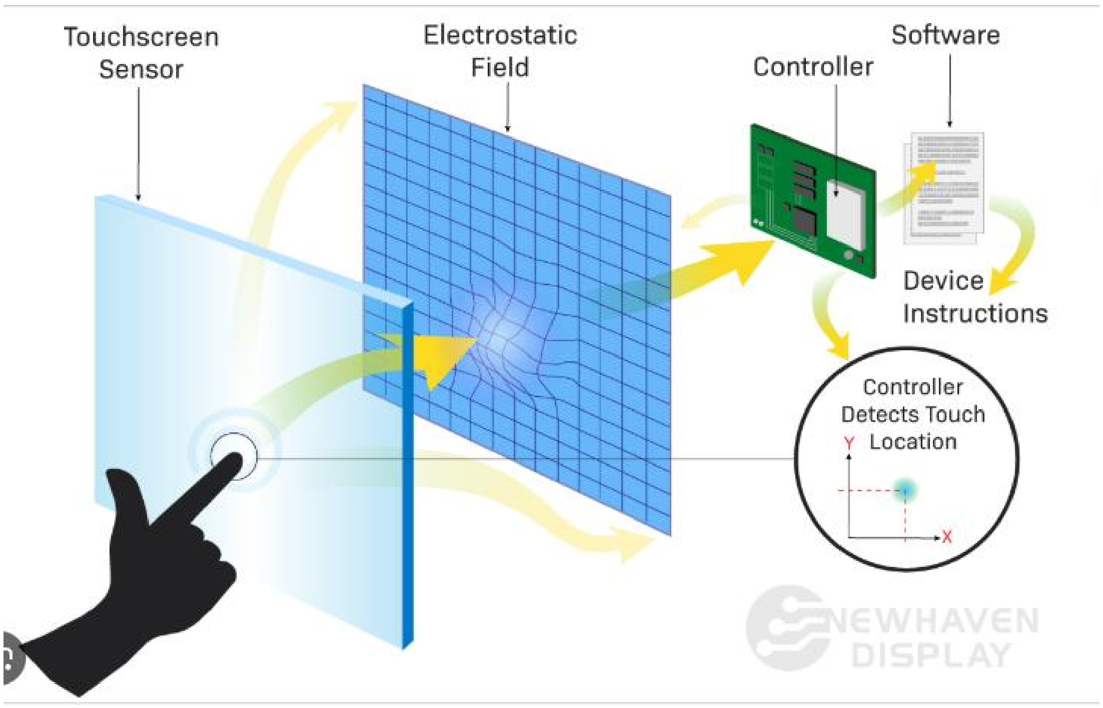
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  794, y 475, strength 4
Touch point[0]: x    5, y 475, strength 4
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
```

Hardware Used in This Lesson

The touchscreen on the Advance-P4

Touchscreen schematic diagram



First, let's look at the Touchscreen Sensor and Electrostatic Field sections. Inside the touchscreen sensor, there is a grid-like electrode structure composed of conductive layers. These electrodes interact with each other, forming a uniform electrostatic field in the screen area. When a finger touches the screen, since the human body is conductive, the finger will form a new capacitance with the conductive layer on the screen. The appearance of this capacitance will interfere with the originally uniform electrostatic field, causing a significant distortion in the distribution of the electrostatic field in the area near the touch point, and subsequently resulting in changes in the capacitance value of the electrodes in that area.

Then, we come to the core function of the Controller. The GT911 takes on this role as the controller. It continuously scans all the electrodes on the touchscreen and precisely detects the changes in the capacitance of each electrode. Based on the detected data of the different capacitances of the electrodes, the GT911 runs a specific algorithm internally, analyzing these data to calculate the X and Y coordinates of the touch point on the screen, which is the coordinate detection process illustrated in the diagram as "Controller Detects Touch Location".
After that, the GT911 sends the calculated touch point coordinate information to the connected main processor (such as an ESP32 microcontroller) according to the pre-set communication protocol (such as I2C, SPI, etc.).

Finally, the main processor receives the coordinate data and further processes and parses these data using software.
At the same time, in combination with the "Device Instructions" (device instruction logic), the software maps and correlates the touch coordinates with specific elements in the device interface (such as buttons, sliders, etc.). Thus, when the user touches the screen, the device can accurately identify whether it is clicking a button, sliding the screen, or other operations, and make corresponding interaction responses, thereby achieving smooth touch interaction functionality.

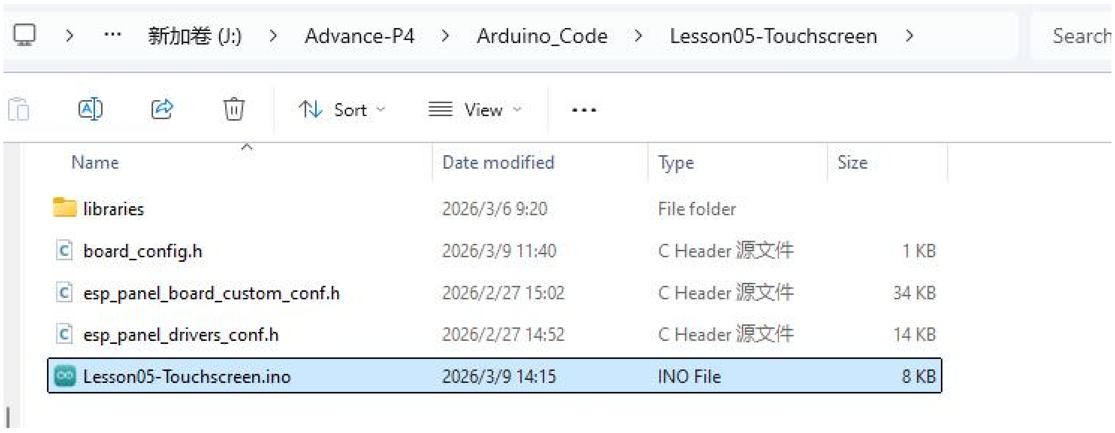
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson05-Touchscreen

Key Explanations

How can we make the touch screen coordinates accurate, so that wherever you tap, it displays the coordinates of that exact location on the screen? Let's dive into how the code achieves this together.

Double-click to open this lesson's code (the .ino file).


新加卷 (J:) > Advance-P4 > Arduino_Code > Lesson05-Touchscreen >
Search
Sort
View

Name	Date modified	Type	Size
libraries	2026/3/6 9:20	File folder	
board_config.h	2026/3/9 11:40	C Header 源文件	1 KB
esp_panel_board_custom_conf.h	2026/2/27 15:02	C Header 源文件	34 KB
esp_panel_drivers_conf.h	2026/2/27 14:52	C Header 源文件	14 KB
Lesson05-Touchscreen.ino	2026/3/9 14:15	INO File	8 KB

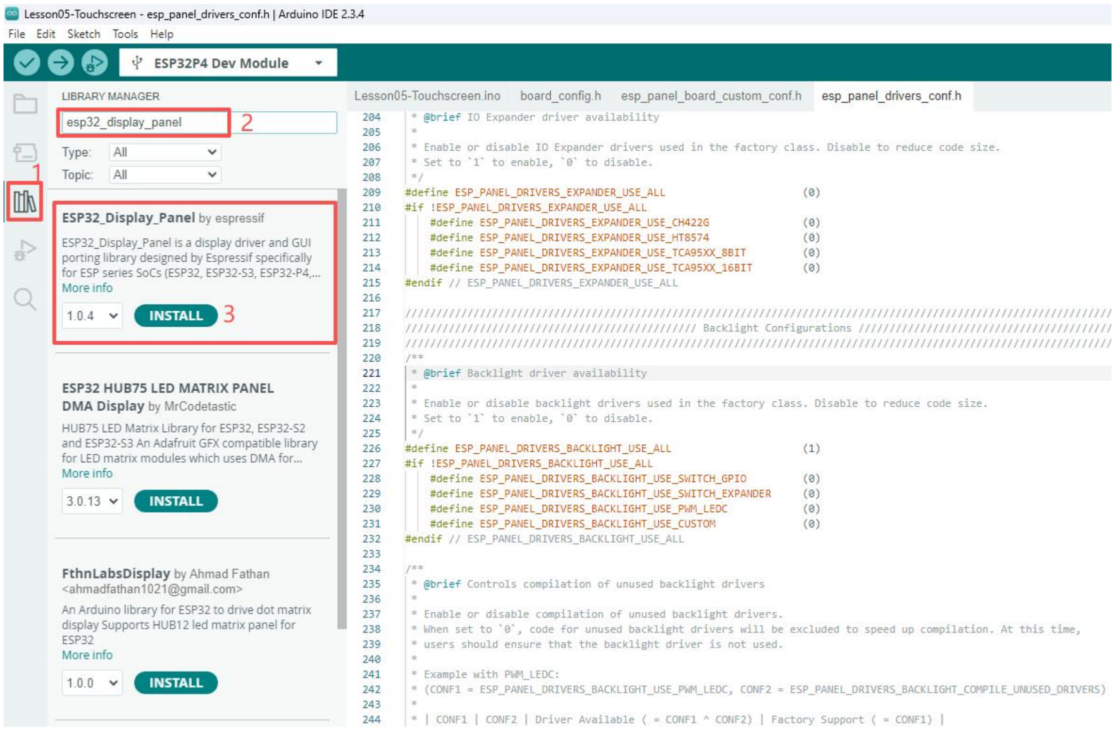
After opening, you can see the project's code, configuration file board_config.h, and other related screen driver files. Let's understand the project architecture for this lesson.

Among them:

The libraries folder:

It contains three files: ESP32_Display_Panel, ESP32_IO_Expander, and esp-lib-utils. Where do these three files come from?

Search for esp32_display_panel in the Library Manager, select version 1.0.4 and install it.


Lesson05-Touchscreen - esp_panel_drivers_conf.h | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
LIBRARY MANAGER
1
2
3
Type: All
Topic: All
Lesson05-Touchscreen.ino
board_config.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h

esp32_display_panel

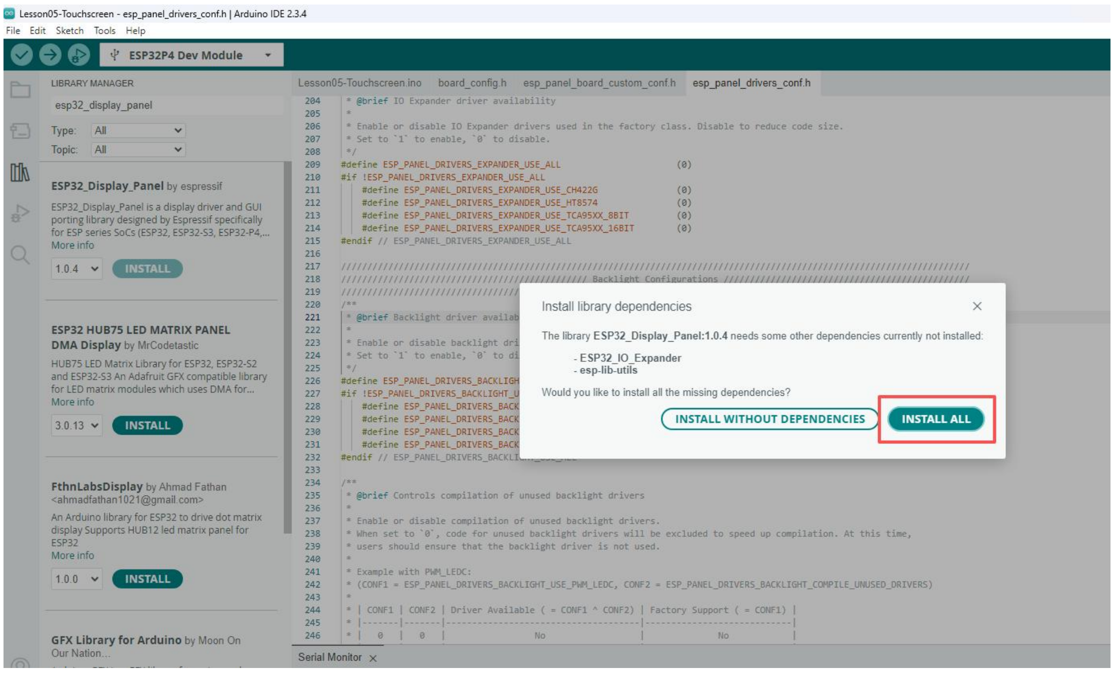
ESP32_Display_Panel by espressif
ESP32_Display_Panel is a display driver and GUI porting library designed by Espressif specifically for ESP series SoCs (ESP32, ESP32-S3, ESP32-P4,...
More info
1.0.4 INSTALL

ESP32 HUB75 LED MATRIX PANEL DMA Display by MrCodetastic
HUB75 LED Matrix Library for ESP32, ESP32-S2 and ESP32-S3 An Adafruit GFX compatible library for LED matrix modules which uses DMA for...
More info
3.0.13 INSTALL

FthnLabsDisplay by Ahmad Fathan <ahmadfathan1021@gmail.com>
An Arduino library for ESP32 to drive dot matrix display Supports HUB12 led matrix panel for ESP32
More info
1.0.0 INSTALL

```
 * @brief IO Expander driver availability
 *
 * Enable or disable IO Expander drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_EXPANDER_USE_ALL                  (0)
#if !ESP_PANEL_DRIVERS_EXPANDER_USE_ALL
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_CH422G           (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_HT8574           (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_8BIT     (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_16BIT    (0)
#endif // ESP_PANEL_DRIVERS_EXPANDER_USE_ALL

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////// Backlight Configurations //////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/**
 * @brief Backlight driver availability
 *
 * Enable or disable backlight drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_BACKLIGHT_USE_ALL                 (1)
#if !ESP_PANEL_DRIVERS_BACKLIGHT_USE_ALL
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_SWITCH_GPIO     (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_SWITCH_EXPANDER (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_PWM_LEDC        (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_CUSTOM          (0)
#endif // ESP_PANEL_DRIVERS_BACKLIGHT_USE_ALL

/**
 * @brief Controls compilation of unused backlight drivers
 *
 * Enable or disable compilation of unused backlight drivers.
 * When set to `0`, code for unused backlight drivers will be excluded to speed up compilation. At this time,
 * users should ensure that the backlight driver is not used.
 *
 * Example with PWM_LEDC:
 * (CONF1 = ESP_PANEL_DRIVERS_BACKLIGHT_USE_PWM_LEDC, CONF2 = ESP_PANEL_DRIVERS_BACKLIGHT_COMPILE_UNUSED_DRIVERS)
 *
 * | CONF1 | CONF2 | Driver Available ( = CONF1 ^ CONF2) | Factory Support ( = CONF1) |
```

It will prompt you whether to install the related dependencies for ESP32_Display_Panel. We select Install All, as we need these dependencies to use it properly.



Lesson05-Touchscreen - esp_panel_drivers_conf.h | Arduino IDE 2.3.4

LIBRARY MANAGER

esp32_display_panel

Type: All
Topic: All

ESP32_Display_Panel by espressif
ESP32_Display_Panel is a display driver and GUI porting library designed by Espressif specifically for ESP series SoCs (ESP32, ESP32-S3, ESP32-P4,...
More info
1.0.4 INSTALL

ESP32 HUB75 LED MATRIX PANEL DMA Display by MrCodetastic
HUB75 LED Matrix Library for ESP32, ESP32-S2 and ESP32-S3 An Adafruit GFX compatible library for LED matrix modules which uses DMA for...
More info
3.0.13 INSTALL

FthnLabsDisplay by Ahmad Fathan <ahmadfathan1021@gmail.com>
An Arduino library for ESP32 to drive dot matrix display Supports HUB12 led matrix panel for ESP32
More info
1.0.0 INSTALL

GFX Library for Arduino by Moon On Our Nation...

```
 * @brief IO Expander driver availability
 *
 * Enable or disable IO Expander drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_EXPANDER_USE_ALL                  (0)
#if !ESP_PANEL_DRIVERS_EXPANDER_USE_ALL
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_CH422G           (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_HT8574           (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_8BIT     (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_16BIT    (0)
#endif // ESP_PANEL_DRIVERS_EXPANDER_USE_ALL

/**
 * @brief Controls compilation of unused backlight drivers
 *
 * Enable or disable compilation of unused backlight drivers.
 * When set to `0`, code for unused backlight drivers will be excluded to speed up compilation. At this time,
 * users should ensure that the backlight driver is not used.
 *
 * Example with PWM_LEDC:
 * (CONF1 = ESP_PANEL_DRIVERS_BACKLIGHT_USE_PWM_LEDC, CONF2 = ESP_PANEL_DRIVERS_BACKLIGHT_COMPILE_UNUSED_DRIVERS)
 *
 * | CONF1 | CONF2 | Driver Available ( = CONF1 ^ CONF2) | Factory Support ( = CONF1) |
 * |-------|-------|-------------------------------------|----------------------------|
 * |   0   |   0   |                 No                  |            No              |
```

Install library dependencies

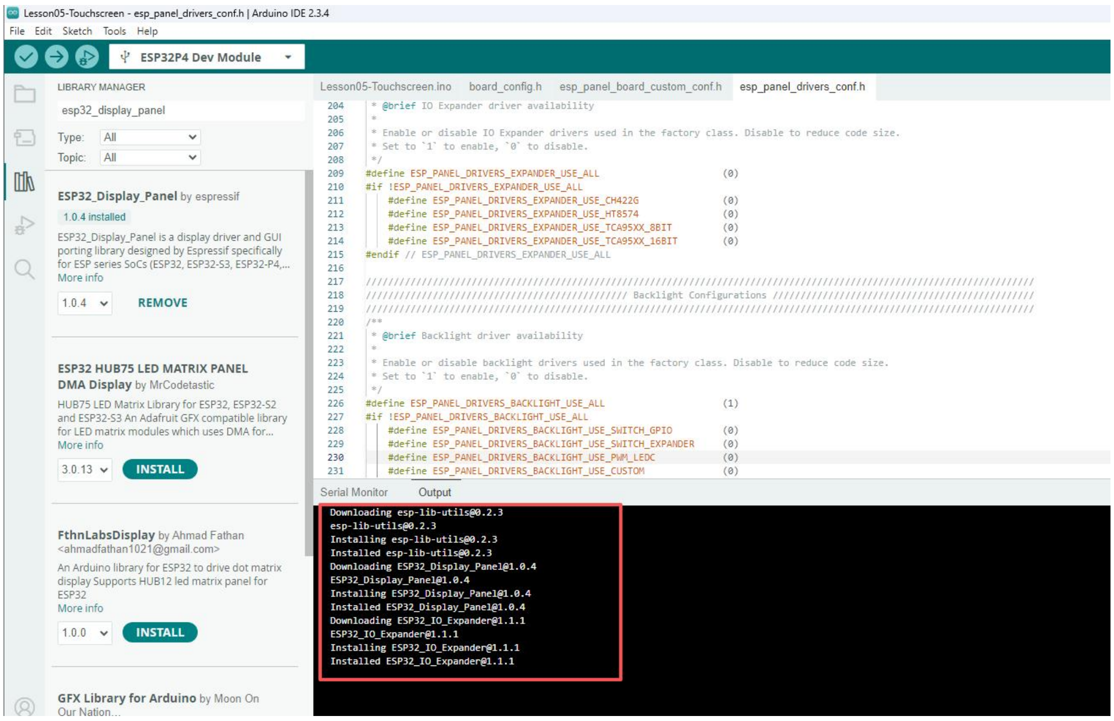
The library **ESP32_Display_Panel:1.0.4** needs some other dependencies currently not installed:

- **ESP32_IO_Expander**
- **esp-lib-utils**

Would you like to install all the missing dependencies?

INSTALL WITHOUT DEPENDENCIES　INSTALL ALL

Installation completed successfully.



Lesson05-Touchscreen - esp_panel_drivers_conf.h | Arduino IDE 2.3.4

LIBRARY MANAGER

esp32_display_panel

Type: All
Topic: All

ESP32_Display_Panel by espressif
1.0.4 installed
ESP32_Display_Panel is a display driver and GUI porting library designed by Espressif specifically for ESP series SoCs (ESP32, ESP32-S3, ESP32-P4,...
More info
1.0.4 REMOVE

ESP32 HUB75 LED MATRIX PANEL DMA Display by MrCodetastic
HUB75 LED Matrix Library for ESP32, ESP32-S2 and ESP32-S3 An Adafruit GFX compatible library for LED matrix modules which uses DMA for...
More info
3.0.13 INSTALL

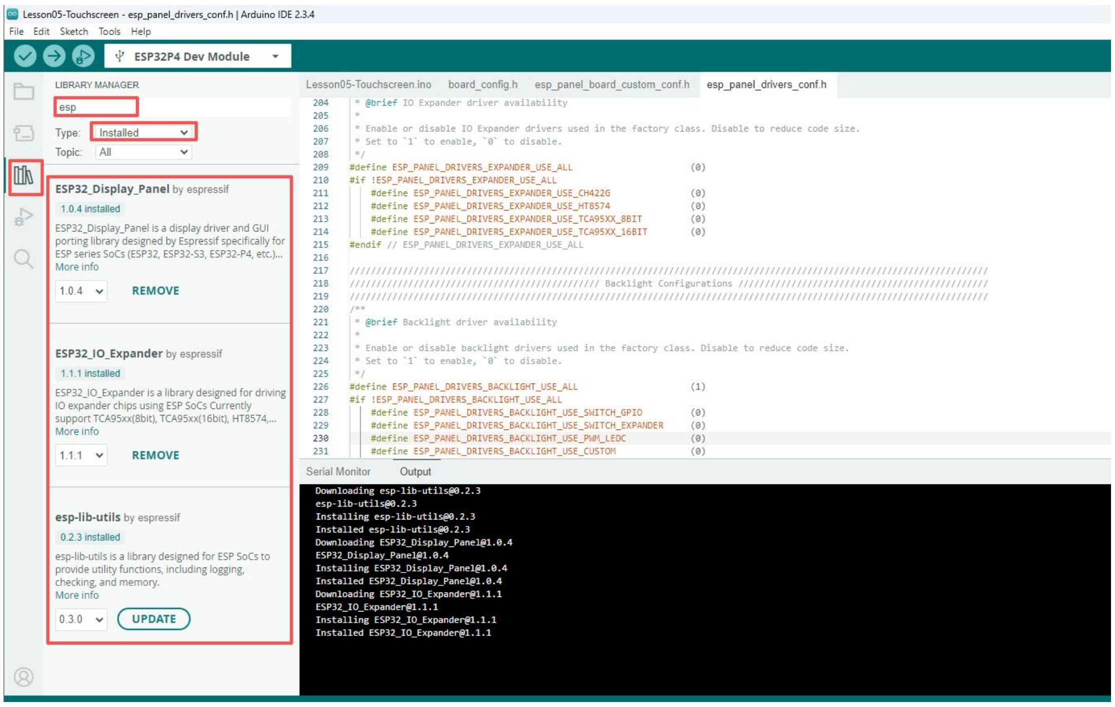
FthnLabsDisplay by Ahmad Fathan <ahmadfathan1021@gmail.com>
An Arduino library for ESP32 to drive dot matrix display Supports HUB12 led matrix panel for ESP32
More info
1.0.0 INSTALL

GFX Library for Arduino by Moon On Our Nation...

```
 * @brief IO Expander driver availability
 *
 * Enable or disable IO Expander drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_EXPANDER_USE_ALL                  (0)
#if !ESP_PANEL_DRIVERS_EXPANDER_USE_ALL
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_CH422G           (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_HT8574           (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_8BIT     (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_16BIT    (0)
#endif // ESP_PANEL_DRIVERS_EXPANDER_USE_ALL

//////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////// Backlight Configurations ///////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////////////////////
/**
 * @brief Backlight driver availability
 *
 * Enable or disable backlight drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_BACKLIGHT_USE_ALL                 (1)
#if !ESP_PANEL_DRIVERS_BACKLIGHT_USE_ALL
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_SWITCH_GPIO     (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_SWITCH_EXPANDER (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_PWM_LEDC        (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_CUSTOM          (0)
```

```
Downloading esp-lib-utils@0.2.3
esp-lib-utils@0.2.3
Installing esp-lib-utils@0.2.3
Installed esp-lib-utils@0.2.3
Downloading ESP32_Display_Panel@1.0.4
ESP32_Display_Panel@1.0.4
Installing ESP32_Display_Panel@1.0.4
Installed ESP32_Display_Panel@1.0.4
Downloading ESP32_IO_Expander@1.1.1
ESP32_IO_Expander@1.1.1
Installing ESP32_IO_Expander@1.1.1
Installed ESP32_IO_Expander@1.1.1
```

You can also see the installed libraries in the left sidebar.


Lesson05-Touchscreen - esp_panel_drivers_conf.h | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
LIBRARY MANAGER
Lesson05-Touchscreen.ino
board_config.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h
Serial Monitor
Output

esp

Type: Installed

Topic: All

ESP32_Display_Panel by espressif
1.0.4 installed
ESP32_Display_Panel is a display driver and GUI porting library designed by Espressif specifically for ESP series SoCs (ESP32, ESP32-S3, ESP32-P4, etc.)...
More info
1.0.4 REMOVE

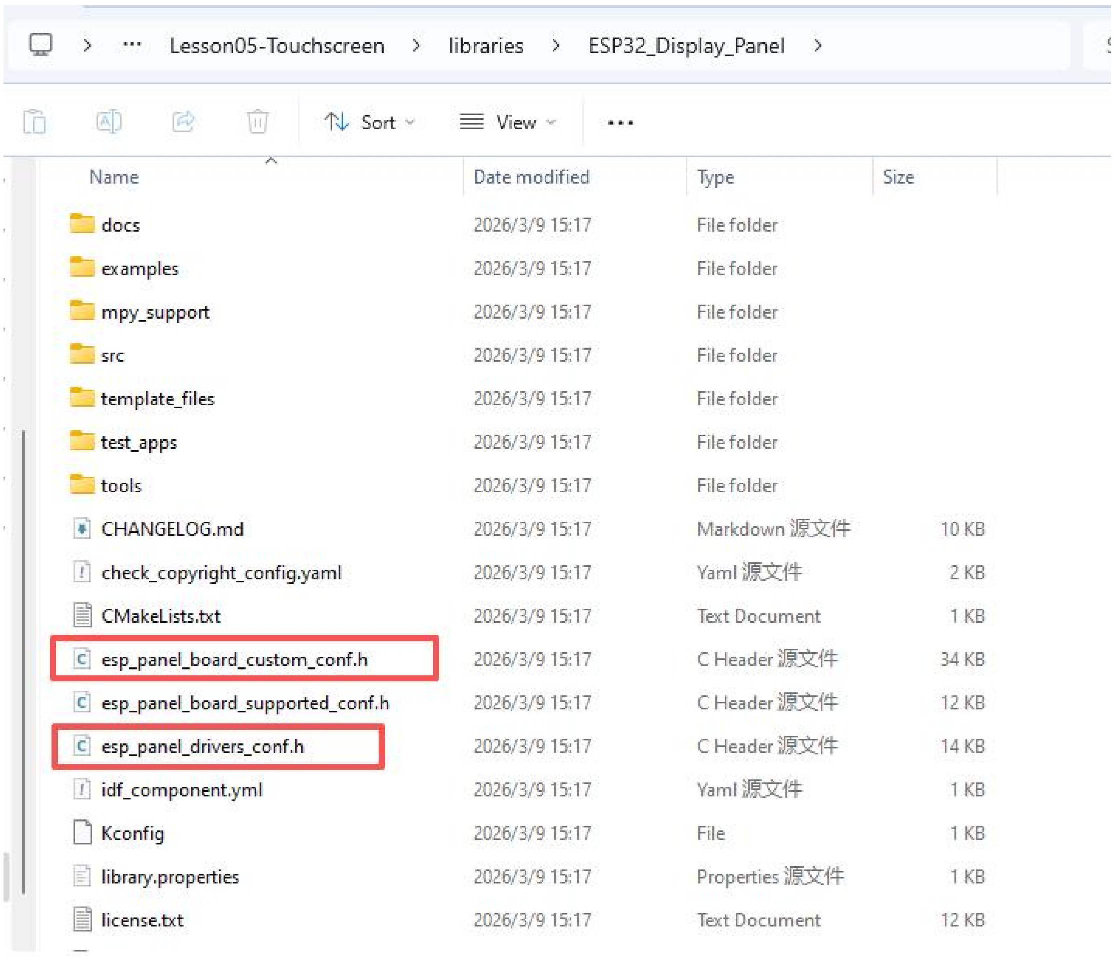
ESP32_IO_Expander by espressif
1.1.1 installed
ESP32_IO_Expander is a library designed for driving IO expander chips using ESP SoCs Currently support TCA95xx(8bit), TCA95xx(16bit), HT8574,...
More info
1.1.1 REMOVE

esp-lib-utils by espressif
0.2.3 installed
esp-lib-utils is a library designed for ESP SoCs to provide utility functions, including logging, checking, and memory.
More info
0.3.0 UPDATE

```
 * @brief IO Expander driver availability
 *
 * Enable or disable IO Expander drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_EXPANDER_USE_ALL                (0)
#if !ESP_PANEL_DRIVERS_EXPANDER_USE_ALL
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_CH422G         (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_HT8574         (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_8BIT   (0)
    #define ESP_PANEL_DRIVERS_EXPANDER_USE_TCA95XX_16BIT  (0)
#endif // ESP_PANEL_DRIVERS_EXPANDER_USE_ALL

///////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////// Backlight Configurations //////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////////////////////////////////
/**
 * @brief Backlight driver availability
 *
 * Enable or disable backlight drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_BACKLIGHT_USE_ALL               (1)
#if !ESP_PANEL_DRIVERS_BACKLIGHT_USE_ALL
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_SWITCH_GPIO       (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_SWITCH_EXPANDER   (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_PWM_LEDC          (0)
    #define ESP_PANEL_DRIVERS_BACKLIGHT_USE_CUSTOM            (0)
```

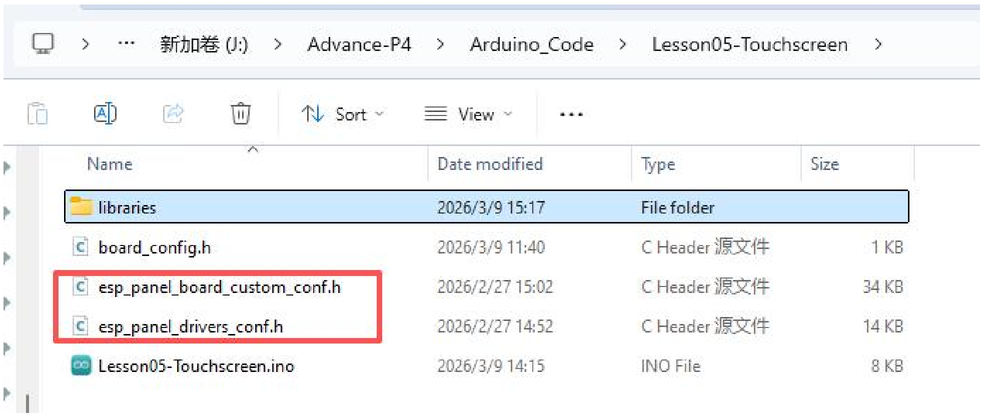
```
Downloading esp-lib-utils@0.2.3
esp-lib-utils@0.2.3
Installing esp-lib-utils@0.2.3
Installed esp-lib-utils@0.2.3
Downloading ESP32_Display_Panel@1.0.4
ESP32_Display_Panel@1.0.4
Installing ESP32_Display_Panel@1.0.4
Installed ESP32_Display_Panel@1.0.4
Downloading ESP32_IO_Expander@1.1.1
ESP32_IO_Expander@1.1.1
Installing ESP32_IO_Expander@1.1.1
Installed ESP32_IO_Expander@1.1.1
```

After downloading, we open the ESP32_Display_Panel library and copy out the esp_panel_board_custom_conf.h and esp_panel_drivers_conf.h files.



Name	Date modified	Type	Size
docs	2026/3/9 15:17	File folder	
examples	2026/3/9 15:17	File folder	
mpy_support	2026/3/9 15:17	File folder	
src	2026/3/9 15:17	File folder	
template_files	2026/3/9 15:17	File folder	
test_apps	2026/3/9 15:17	File folder	
tools	2026/3/9 15:17	File folder	
CHANGELOG.md	2026/3/9 15:17	Markdown 源文件	10 KB
check_copyright_config.yaml	2026/3/9 15:17	Yaml 源文件	2 KB
CMakeLists.txt	2026/3/9 15:17	Text Document	1 KB
esp_panel_board_custom_conf.h	2026/3/9 15:17	C Header 源文件	34 KB
esp_panel_board_supported_conf.h	2026/3/9 15:17	C Header 源文件	12 KB
esp_panel_drivers_conf.h	2026/3/9 15:17	C Header 源文件	14 KB
idf_component.yml	2026/3/9 15:17	Yaml 源文件	1 KB
Kconfig	2026/3/9 15:17	File	1 KB
library.properties	2026/3/9 15:17	Properties 源文件	1 KB
license.txt	2026/3/9 15:17	Text Document	12 KB

Paste them in the same directory path as the .ino file.



新加卷 (J:) > Advance-P4 > Arduino_Code > Lesson05-Touchscreen >

Name	Date modified	Type	Size
libraries	2026/3/9 15:17	File folder	
board_config.h	2026/3/9 11:40	C Header 源文件	1 KB
esp_panel_board_custom_conf.h	2026/2/27 15:02	C Header 源文件	34 KB
esp_panel_drivers_conf.h	2026/2/27 14:52	C Header 源文件	14 KB
Lesson05-Touchscreen.ino	2026/3/9 14:15	INO File	8 KB

Then modify the parameters and configurations within them to adapt to our own product, the CrowPanel Advanced ESP32-P4 HMI AI Display.

(Note: You do not need to modify the esp_panel_board_custom_conf.h and esp_panel_drivers_conf.h files; we have already configured them for you, so you can use them directly.)

esp_panel_board_custom_conf.h

The esp_panel_board_custom_conf.h file is a core configuration file in the ESP32_Display_Panel library for customizing the hardware configuration of the development board. Its main role is to uniformly configure parameters such as the display screen, touch screen, backlight, power supply, bus interface, and extended hardware of the user's development board through a large number of macro definitions, enabling the library to correctly match the specific hardware and automatically complete driver configuration during initialization.

This configuration file is equivalent to a "hardware description file" for the entire display system. It centrally describes all hardware parameters of the screen, touch, backlight, and related interfaces on the development board, allowing the library to automatically complete the initialization and driver loading of display and touch devices based on these configurations when the program calls Board::init() and Board::begin().

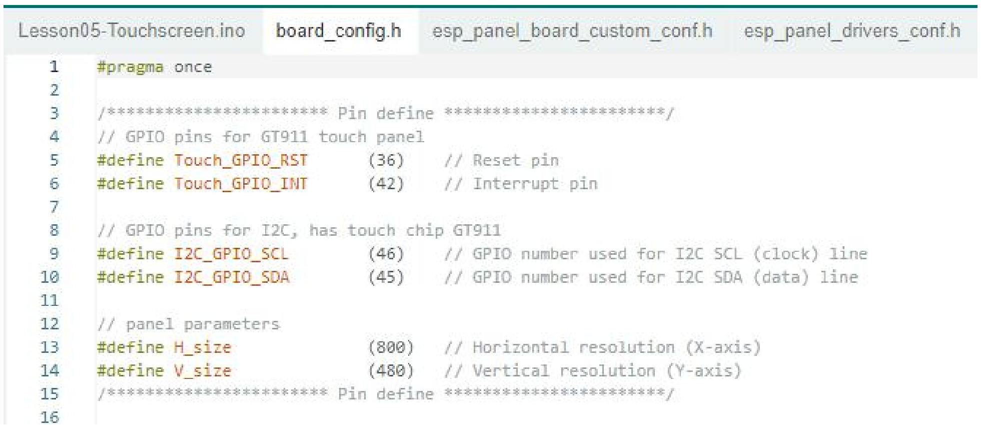
esp_panel_drivers_conf.h

The esp_panel_drivers_conf.h file is a driver function configuration file in the ESP32_Display_Panel library. Its main role is to control whether the driver modules related to the display system are compiled and used through a series of macro definitions, thereby determining which bus drivers, LCD drivers, touch drivers, IO expander drivers, and backlight drivers are ultimately included in the program. By pruning unused drivers, it reduces firmware size, improves compilation speed, and optimizes system resource usage.

The core role of this file can be understood as the "function pruning and driver selection center" of the ESP32 display driver system. Developers can determine which display buses, screen controllers, touch chips, and backlight modes the system supports by modifying the macro definitions here, so that the driver library only includes the functional modules actually needed by the current project during the compilation phase.

board_config.h

It configures the touch pins used in our product, as well as the I2C pins required by the touch chip.



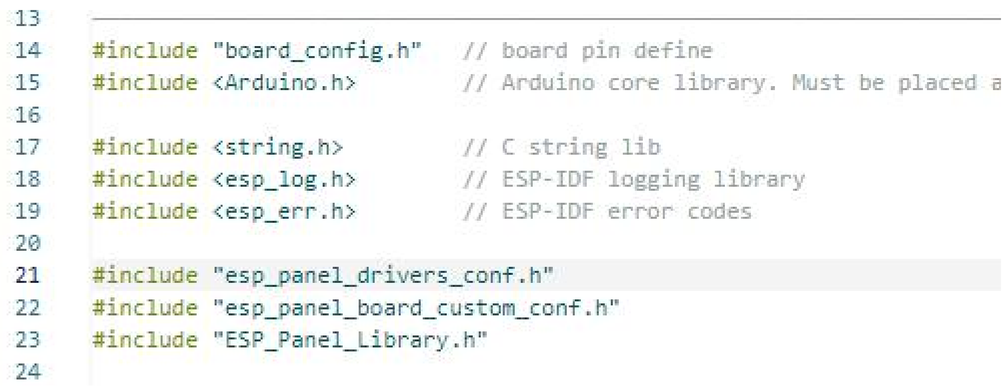
Lesson05-Touchscreen.ino | board_config.h | esp_panel_board_custom_conf.h | esp_panel_drivers_conf.h

```
#pragma once

/*********************** Pin define ***********************/
// GPIO pins for GT911 touch panel
#define Touch_GPIO_RST      (36)    // Reset pin
#define Touch_GPIO_INT      (42)    // Interrupt pin

// GPIO pins for I2C, has touch chip GT911
#define I2C_GPIO_SCL        (46)    // GPIO number used for I2C SCL (clock) line
#define I2C_GPIO_SDA        (45)    // GPIO number used for I2C SDA (data) line

// panel parameters
#define H_size              (800)   // Horizontal resolution (X-axis)
#define V_size              (480)   // Vertical resolution (Y-axis)
/*********************** Pin define ***********************/

```

The touch screen resolution is specified via the macro definitions #define V_size 480 and #define H_size 800, where V_size indicates the vertical (Y-axis) resolution of the screen is 480 pixels, and H_size indicates the horizontal (X-axis) resolution is 800 pixels. This corresponds to the actual display screen resolution of 800×480, and can be used as a reference size when processing touch coordinates or interface layouts later.

Now that we have learned about these configuration files, let's take a look at our main code file with the .ino extension.This code fully implements the initialization of the display screen and capacitive touch screen, power supply configuration, as well as real-time reading of touch data and debugging output on the ESP32-P4 platform.Next, we will go through the code section by section to clearly understand the role of each section.

1. Header file



```

#include "board_config.h"    // board pin define
#include <Arduino.h>         // Arduino core library. Must be placed a

#include <string.h>          // C string lib
#include <esp_log.h>         // ESP-IDF logging library
#include <esp_err.h>         // ESP-IDF error codes

#include "esp_panel_drivers_conf.h"
#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

```

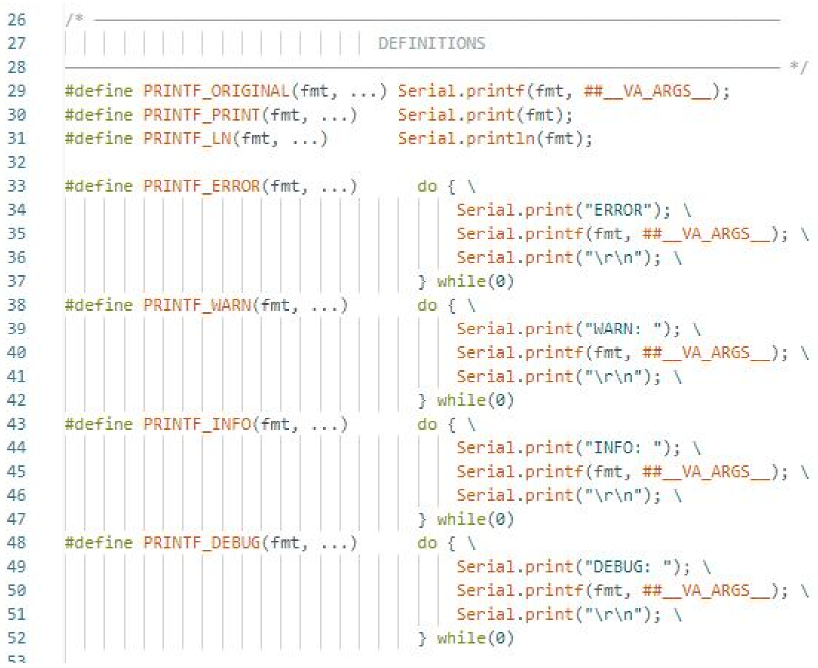
The role of this code is to include various header files required for the program to run, providing basic support for the subsequent implementation of functions such as screen display, touch reading, power management, and log debugging.

Among them, "#include "board_config.h"" is used to load the pin configuration and hardware definitions of the development board;
"#include <Arduino.h>" is the core Arduino library, which provides Arduino APIs such as "Serial", "delay()", "setup()", and "loop()", so it usually needs to be placed at the top to ensure the compiler correctly recognizes these functions.

Next, "string.h" provides C language string processing functions (such as "strlen", "strcpy", etc.), and "esp_log.h" and "esp_err.h" are the log and error code libraries in the ESP-IDF framework, used to print debugging information and determine whether function execution is successful;

The last three files "esp_panel_drivers_conf.h", "esp_panel_board_custom_conf.h", and "ESP_Panel_Library.h" belong to the configuration and driver interfaces related to the ESP32 Display Panel Library (ESP Panel Library). They are used to specify the display driver chip (e.g., EK79007), touch chip (e.g., GT911), and hardware configuration of the panel used, and provide a unified API to initialize the screen, control display, and read touch data.

2. Macro Definitions



```
/* ----------------------------------------------------------------------
                                DEFINITIONS
---------------------------------------------------------------------- */
#define PRINTF_ORIGINAL(fmt, ...) Serial.printf(fmt, ##__VA_ARGS__);
#define PRINTF_PRINT(fmt, ...)    Serial.print(fmt);
#define PRINTF_LN(fmt, ...)       Serial.println(fmt);

#define PRINTF_ERROR(fmt, ...)      do { \
                                        Serial.print("ERROR"); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
#define PRINTF_WARN(fmt, ...)       do { \
                                        Serial.print("WARN: "); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
#define PRINTF_INFO(fmt, ...)       do { \
                                        Serial.print("INFO: "); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
#define PRINTF_DEBUG(fmt, ...)      do { \
                                        Serial.print("DEBUG: "); \
                                        Serial.printf(fmt, ##__VA_ARGS__); \
                                        Serial.print("\r\n"); \
                                    } while(0)
```

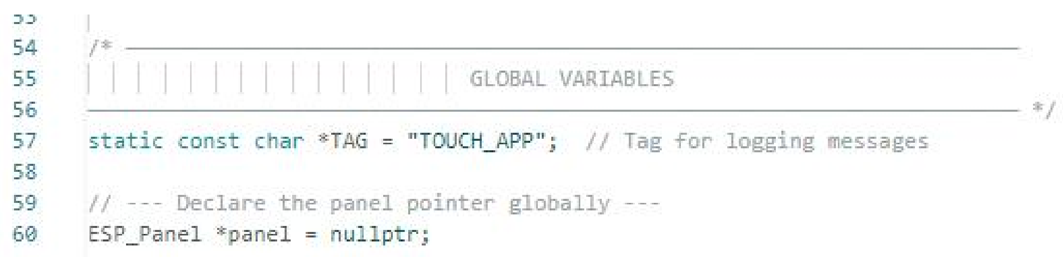
This code mainly defines a set of macros for serial port log output, aiming to unify the printing method of the program's debugging information and make the code more standardized and clear during debugging.

At the very beginning, "PRINTF_ORIGINAL", "PRINTF_PRINT", and "PRINTF_LN" are simple encapsulations of the Arduino serial port functions "Serial.printf()", "Serial.print()", and "Serial.println()", used to directly output strings or formatted content;

Whereas "PRINTF_ERROR", "PRINTF_WARN", "PRINTF_INFO", and "PRINTF_DEBUG" that follow automatically add different log level prefixes (such as "ERROR", "WARN", "INFO", "DEBUG") when printing information, enabling quick differentiation between error messages, warning messages, general information, or debugging information in the serial monitor. For example, when calling "PRINTF_INFO("WiFi Connected");", the serial port will output "INFO: WiFi Connected".

Inside these macros, "Serial.print" and "Serial.printf" are combined to output content, and "\r\n" is manually added at the end to achieve line breaks; meanwhile, multiple statements are encapsulated into a single unit using the "do { ... } while(0)" syntax, making the macro syntactically behave like a normal function call and avoiding logical errors when used in statements such as "if".

3. Variable Definitions



```
|
/* ──────────────────────────────────────────────────────────
| | | | | | | | | | | | | | |  GLOBAL VARIABLES
──────────────────────────────────────────────────────────── */
static const char *TAG = "TOUCH_APP";  // Tag for logging messages

// --- Declare the panel pointer globally ---
ESP_Panel *panel = nullptr;
```

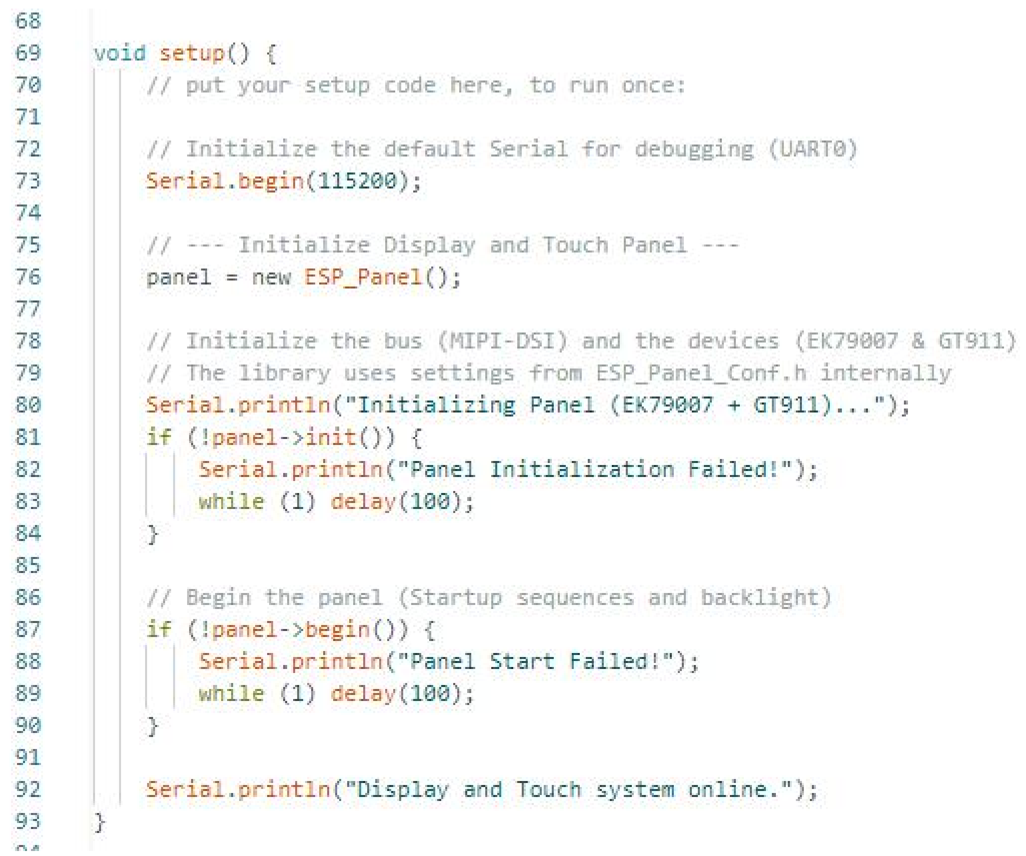
This code mainly completes the declaration of global variables, providing basic parameters and control objects for subsequent display and touch functions.

Next, static const char *TAG = "TOUCH_APP"; is defined, which is a log tag string. It is usually used to identify the source of the current module when printing log information—for example, indicating that the log comes from the touch application module, making it easy to quickly locate problems during debugging.

The last line ESP_Panel *panel = nullptr; declares a pointer variable of type ESP_Panel and initializes it to nullptr (meaning it does not point to any object at present). This pointer will point to an ESP_Panel object during program operation to uniformly manage the display screen and touch device, such as initializing the screen and reading touch data. Since it is defined as a global variable, this panel control object can be directly accessed in setup(), loop(), or other functions.

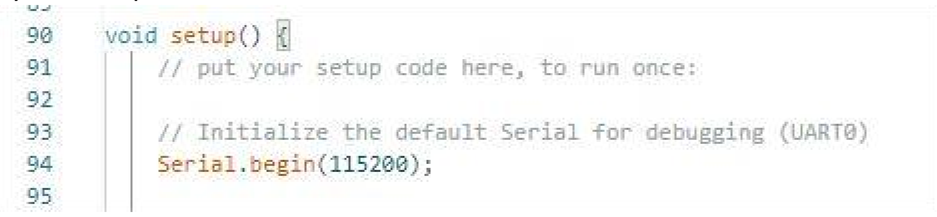
4. setup

This setup() function serves as the system initialization entry point for the entire program. It runs only once when the device is powered on or reset, and is primarily responsible for initializing serial port debugging, the power supply system, as well as the display screen and touch panel hardware.



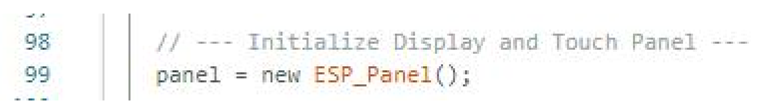
```
void setup() {
  // put your setup code here, to run once:

  // Initialize the default Serial for debugging (UART0)
  Serial.begin(115200);

  // --- Initialize Display and Touch Panel ---
  panel = new ESP_Panel();

  // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
  // The library uses settings from ESP_Panel_Conf.h internally
  Serial.println("Initializing Panel (EK79007 + GT911)...");
  if (!panel->init()) {
    Serial.println("Panel Initialization Failed!");
    while (1) delay(100);
  }

  // Begin the panel (Startup sequences and backlight)
  if (!panel->begin()) {
    Serial.println("Panel Start Failed!");
    while (1) delay(100);
  }

  Serial.println("Display and Touch system online.");
}
```

At the start of the function, Serial.begin(115200) is called to initiate serial communication (UART0) with a baud rate set to 115200. This is used to output debugging information in the serial monitor, enabling developers to monitor the system's operational status.



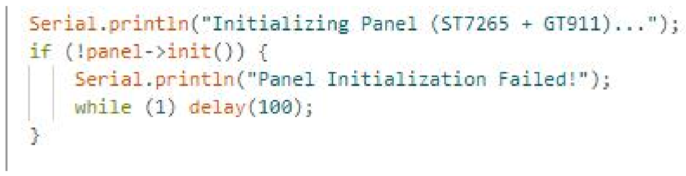
```
void setup() {
  // put your setup code here, to run once:

  // Initialize the default Serial for debugging (UART0)
  Serial.begin(115200);

```

Next, the program creates an ESP_Panel object with panel = new ESP_Panel();. This object is part of the ESP_Panel_Library and is used to uniformly manage hardware devices such as the display screen, communication bus, and touch controller.



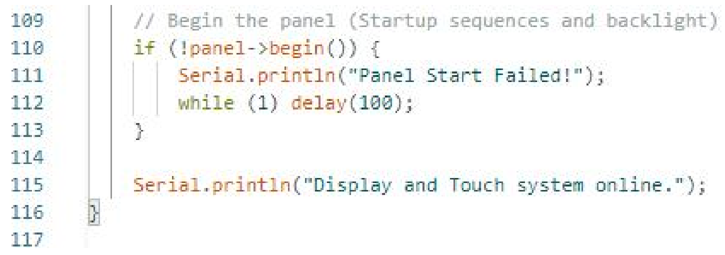
```
  // --- Initialize Display and Touch Panel ---
  panel = new ESP_Panel();
```

Then, call panel->init() to initialize the bus, display driver chip, and GT911 touch controller (the relevant configurations are from the ESP_Panel_Conf.h file in the library). If the initialization fails, the program will enter an infinite loop and stop running.



```
Serial.println("Initializing Panel (ST7265 + GT911)...");
if (!panel->init()) {
    Serial.println("Panel Initialization Failed!");
    while (1) delay(100);
}
```

After successful initialization, the program proceeds to call panel->begin() to start the display panel. This step typically executes operations such as the screen's startup sequence, driver configuration, and backlight activation. If the startup fails, it will similarly print "Panel Start Failed!" and enter an infinite loop.

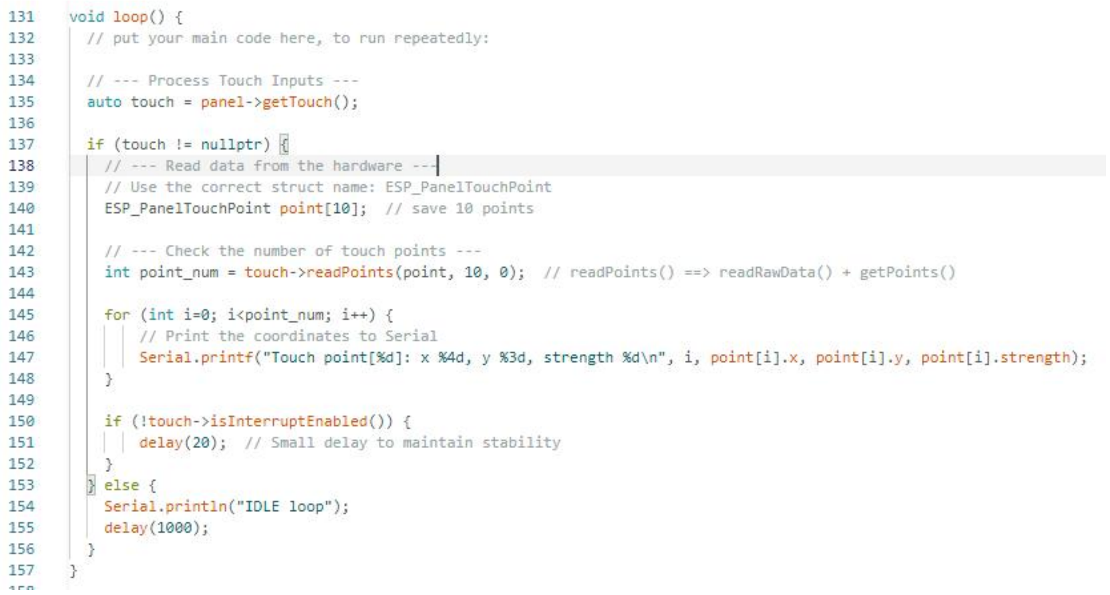


```
    // Begin the panel (Startup sequences and backlight)
    if (!panel->begin()) {
        Serial.println("Panel Start Failed!");
        while (1) delay(100);
    }

    Serial.println("Display and Touch system online.");
}
```

Finally, when all initialization steps are successfully completed, the program prints "Display and Touch system online.", indicating that the display and touch system have been successfully started and are ready for normal operation. At this point, the system is prepared to enter the subsequent main loop logic.

5. loop

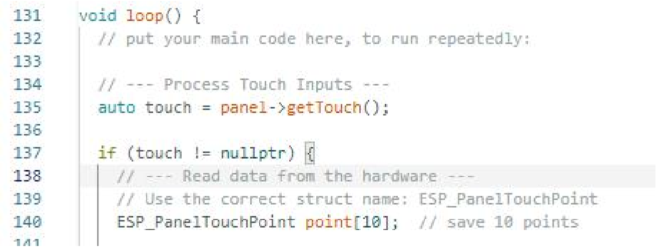
This loop() function code is designed to continuously read touch data from the touch screen during program operation and print the coordinates and touch intensity information of touch points via the serial port.



```
void loop() {
  // put your main code here, to run repeatedly:

  // --- Process Touch Inputs ---
  auto touch = panel->getTouch();

  if (touch != nullptr) {
    // --- Read data from the hardware ---
    // Use the correct struct name: ESP_PanelTouchPoint
    ESP_PanelTouchPoint point[10];  // save 10 points

    // --- Check the number of touch points ---
    int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()

    for (int i=0; i<point_num; i++) {
      // Print the coordinates to Serial
      Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\n", i, point[i].x, point[i].y, point[i].strength);
    }

    if (!touch->isInterruptEnabled()) {
      delay(20);  // Small delay to maintain stability
    }
  } else {
    Serial.println("IDLE loop");
    delay(1000);
  }
}
```

The program first retrieves the touch control object of the current panel via "panel->getTouch()" and stores it in the "touch" variable;

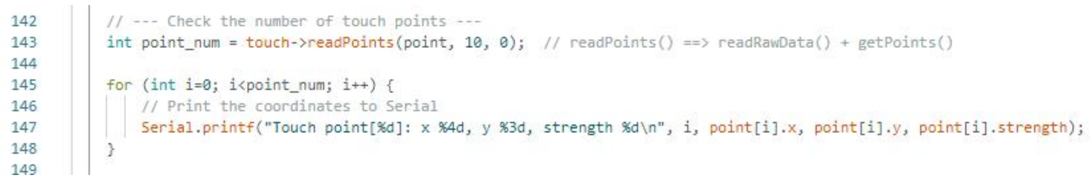
if the object is not null (indicating that the touch device has been successfully initialized), the program creates an array "ESP_PanelTouchPoint point[10]" to store touch data. Up to 10 touch points can be stored here because touch chips such as the GT911 support multi-touch functionality.



```
void loop() {
  // put your main code here, to run repeatedly:

  // --- Process Touch Inputs ---
  auto touch = panel->getTouch();

  if (touch != nullptr) {
    // --- Read data from the hardware ---
    // Use the correct struct name: ESP_PanelTouchPoint
    ESP_PanelTouchPoint point[10];  // save 10 points
```

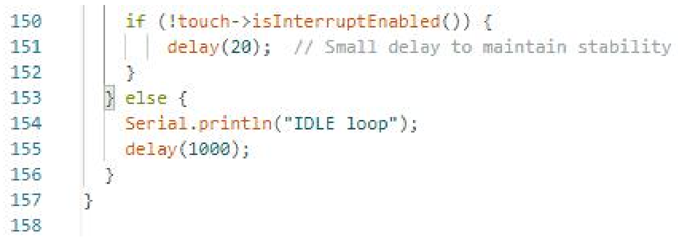
It then calls "touch->readPoints(point, 10, 0)" to read data from the touch hardware. This function reads the raw touch data, parses it into coordinate information, and returns the number of currently detected touch points ("point_num").

The program then iterates through each touch point using a "for" loop and prints the X coordinate, Y coordinate, and touch intensity (strength) of the touch point with "Serial.printf()", allowing the touch position to be viewed in real time in the serial monitor.



```
    // --- Check the number of touch points ---
    int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()

    for (int i=0; i<point_num; i++) {
      // Print the coordinates to Serial
      Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\n", i, point[i].x, point[i].y, point[i].strength);
    }

```

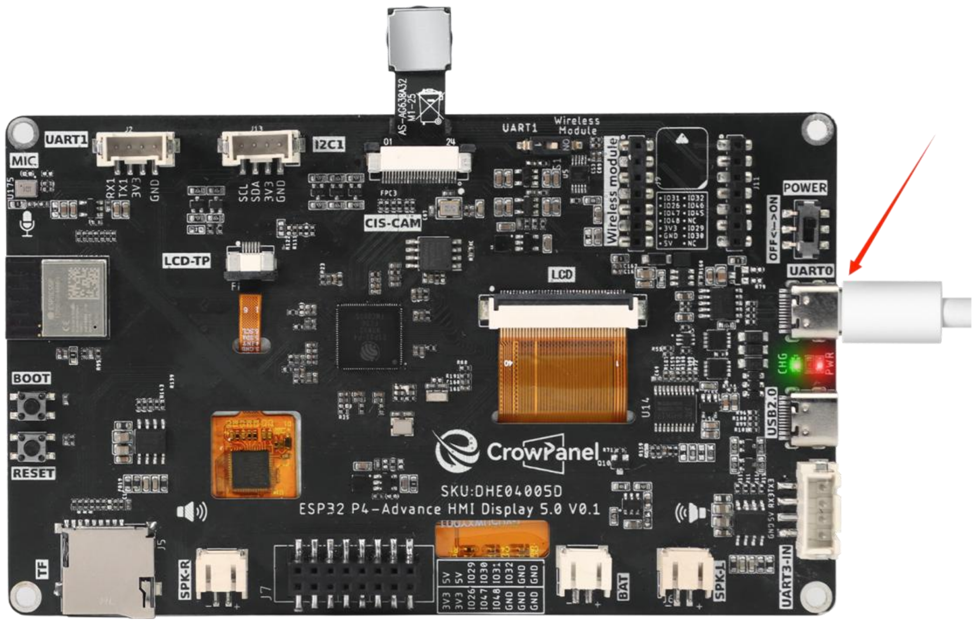
If the touch controller does not have interrupt mode enabled (i.e., "isInterruptEnabled()" returns "false"), the program executes "delay(20)" to introduce a short delay, which reduces the reading frequency and maintains system stability;
if the "touch" object is null (indicating the touch device is not properly initialized or does not exist), the program prints "IDLE loop" once per second to indicate that no touch device is currently in operation.



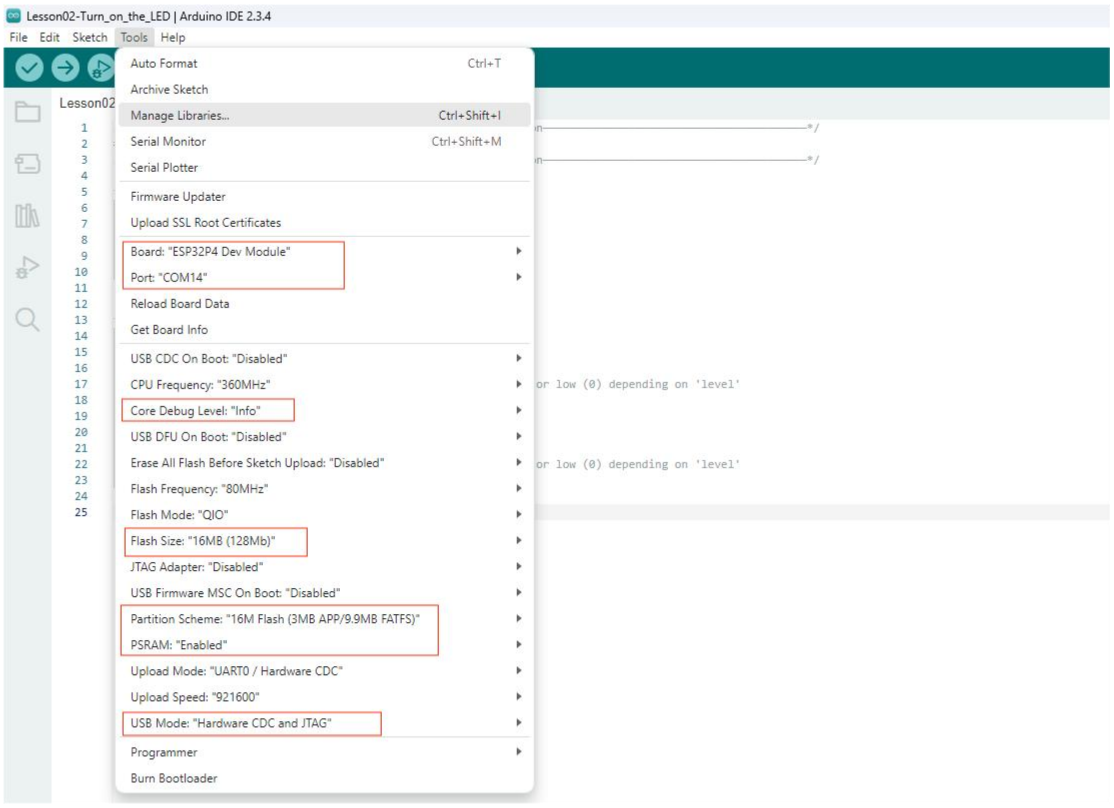
```
    if (!touch->isInterruptEnabled()) {
      delay(20);  // Small delay to maintain stability
    }
  } else {
    Serial.println("IDLE loop");
    delay(1000);
  }
}

```

Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



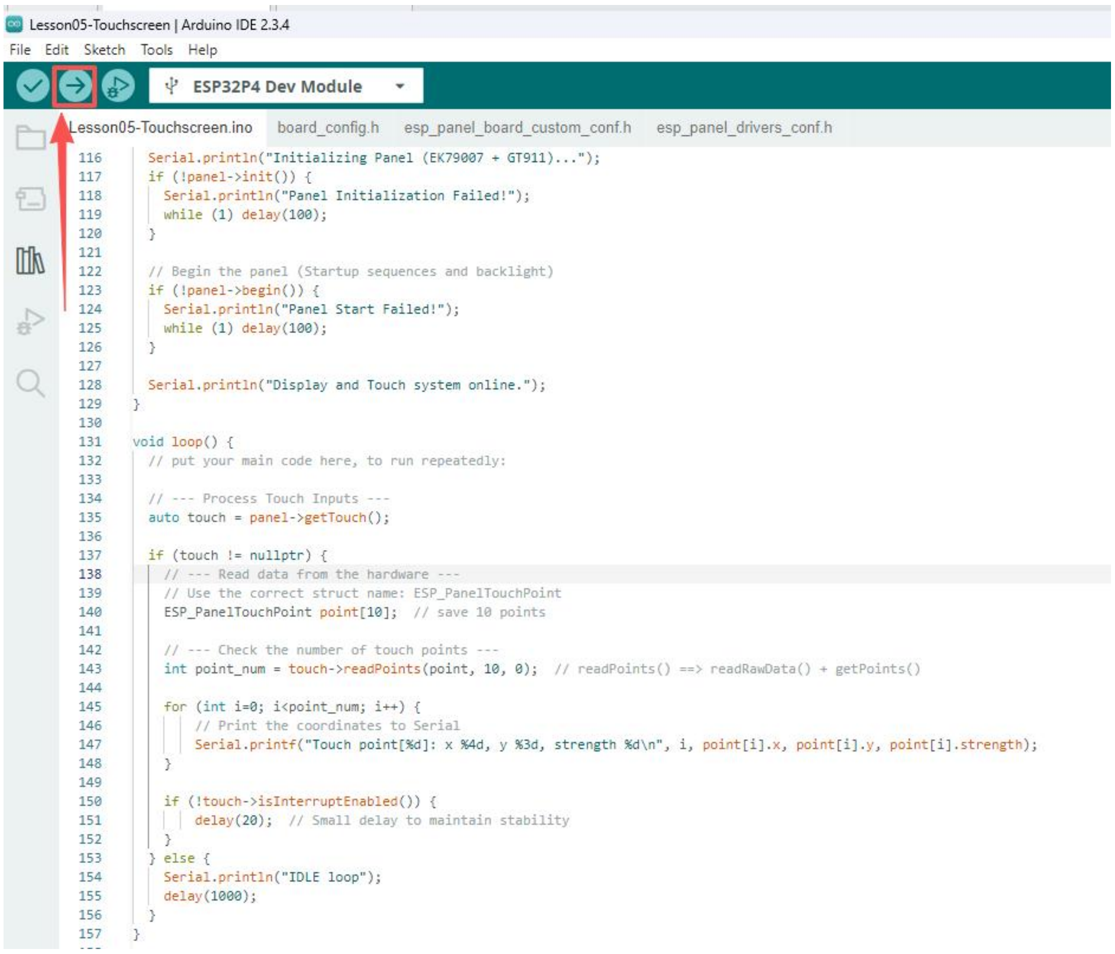
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

File Edit Sketch Tools Help

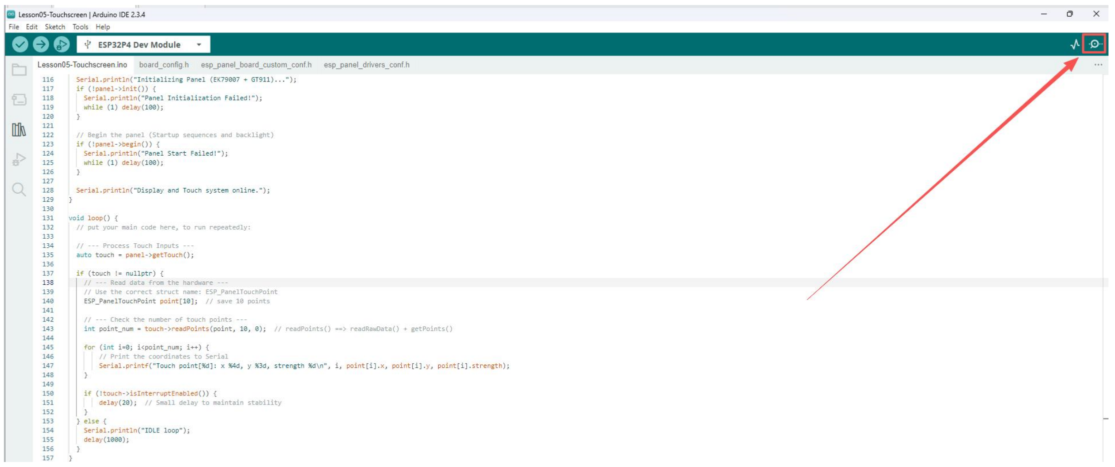
Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader

Then we compile and upload the code.


Lesson05-Touchscreen | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson05-Touchscreen.ino
board_config.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h

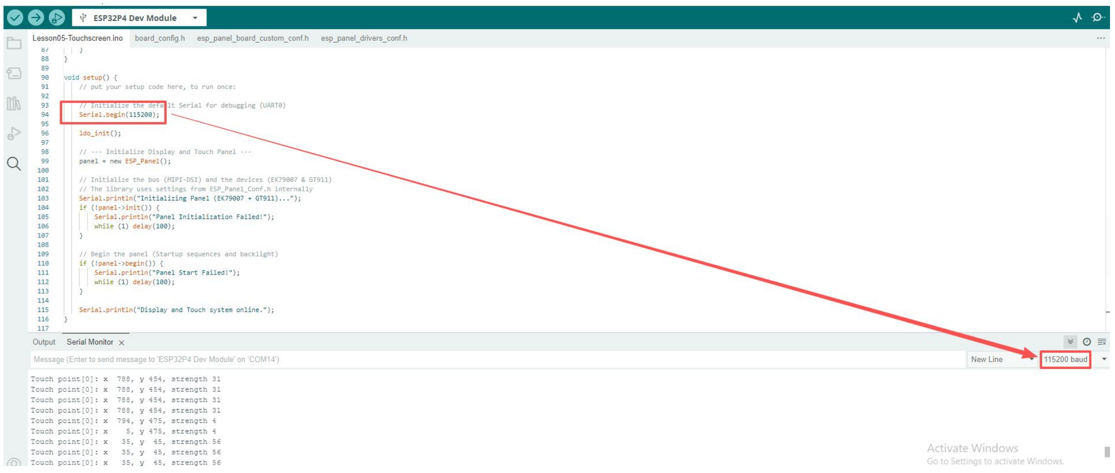
```
  Serial.println("Initializing Panel (EK79007 + GT911)...");
  if (!panel->init()) {
    Serial.println("Panel Initialization Failed!");
    while (1) delay(100);
  }

  // Begin the panel (Startup sequences and backlight)
  if (!panel->begin()) {
    Serial.println("Panel Start Failed!");
    while (1) delay(100);
  }

  Serial.println("Display and Touch system online.");
}

void loop() {
  // put your main code here, to run repeatedly:

  // --- Process Touch Inputs ---
  auto touch = panel->getTouch();

  if (touch != nullptr) {
    // --- Read data from the hardware ---
    // Use the correct struct name: ESP_PanelTouchPoint
    ESP_PanelTouchPoint point[10];  // save 10 points

    // --- Check the number of touch points ---
    int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()

    for (int i=0; i<point_num; i++) {
      // Print the coordinates to Serial
      Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\n", i, point[i].x, point[i].y, point[i].strength);
    }

    if (!touch->isInterruptEnabled()) {
      delay(20);  // Small delay to maintain stability
    }
  } else {
    Serial.println("IDLE loop");
    delay(1000);
  }
}
```

After the code upload completes, you can open the Serial Monitor built into the Arduino-IDE.


Lesson05-Touchscreen | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson05-Touchscreen.ino
board_config.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h

```
  Serial.println("Initializing Panel (EK79007 + GT911)...");
  if (!panel->init()) {
    Serial.println("Panel Initialization Failed!");
    while (1) delay(100);
  }

  // Begin the panel (Startup sequences and backlight)
  if (!panel->begin()) {
    Serial.println("Panel Start Failed!");
    while (1) delay(100);
  }

  Serial.println("Display and Touch system online.");
}

void loop() {
  // put your main code here, to run repeatedly:

  // --- Process Touch Inputs ---
  auto touch = panel->getTouch();

  if (touch != nullptr) {
    // --- Read data from the hardware ---
    // Use the correct struct name: ESP_PanelTouchPoint
    ESP_PanelTouchPoint point[10];  // save 10 points

    // --- Check the number of touch points ---
    int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()

    for (int i=0; i<point_num; i++) {
      // Print the coordinates to Serial
      Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\n", i, point[i].x, point[i].y, point[i].strength);
    }

    if (!touch->isInterruptEnabled()) {
      delay(20);  // Small delay to maintain stability
    }
  } else {
    Serial.println("IDLE loop");
    delay(1000);
  }
}
```

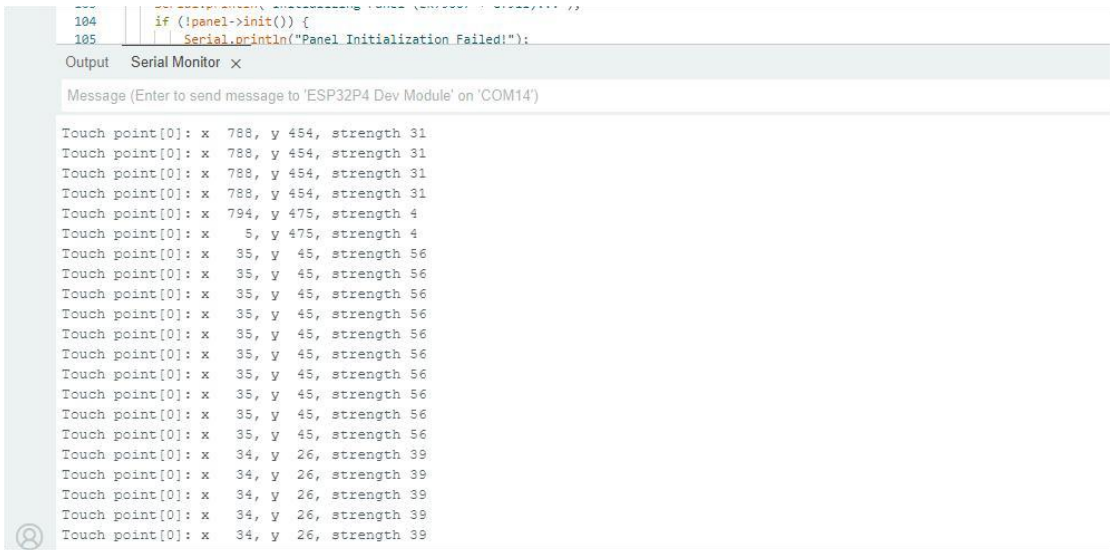
Then set the baud rate required by the serial port, keeping it consistent with what you set in your code.



```
    }
}

void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    ldo_init();

    // --- Initialize Display and Touch Panel ---
    panel = new ESP_Panel();

    // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
    // The library uses settings from ESP_Panel_Conf.h internally
    Serial.println("Initializing Panel (EK79007 + GT911)...");
    if (!panel->init()) {
        Serial.println("Panel Initialization Failed!");
        while (1) delay(100);
    }

    // Begin the panel (Startup sequences and backlight)
    if (!panel->begin()) {
        Serial.println("Panel Start Failed!");
        while (1) delay(100);
    }

    Serial.println("Display and Touch system online.");
}
```

```
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  794, y 475, strength 4
Touch point[0]: x    5, y 475, strength 4
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
```

Next, when you touch the screen, you will be able to see the relevant coordinate information printed in the Serial Monitor.


Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
    if (!panel->init()) {
        Serial.println("Panel Initialization Failed!");
```

```
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  788, y 454, strength 31
Touch point[0]: x  794, y 475, strength 4
Touch point[0]: x    5, y 475, strength 4
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   35, y  45, strength 56
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
Touch point[0]: x   34, y  26, strength 39
```

Lesson06---USB2.0

Introduction

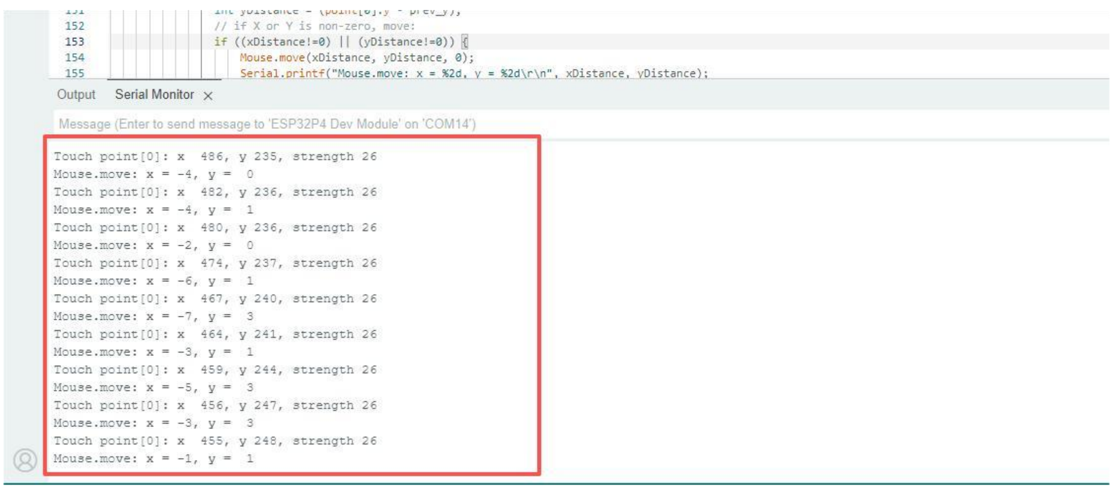
In this lesson, we will build upon what we learned in the previous lesson. Before studying this lesson, please ensure you have understood how the touch functionality was implemented in the previous lesson. This will be very helpful for learning this lesson.
This will enable us to use the USB2.0 interface on the Advance-P4 as a mouse. When you swipe on the Advance-P4 screen, you will find that the mouse on your computer moves accordingly.

Learning Goals

1. Understand USB2.0
2. Implement USB2.0 functionality based on the touch functionality from the previous lesson

Preview of the Result

After running the code, you will be able to see that when you slide the screen on the Advance-P4, the mouse on your computer also moves accordingly, and at the same time, you can see the relevant coordinates printed on the monitor.



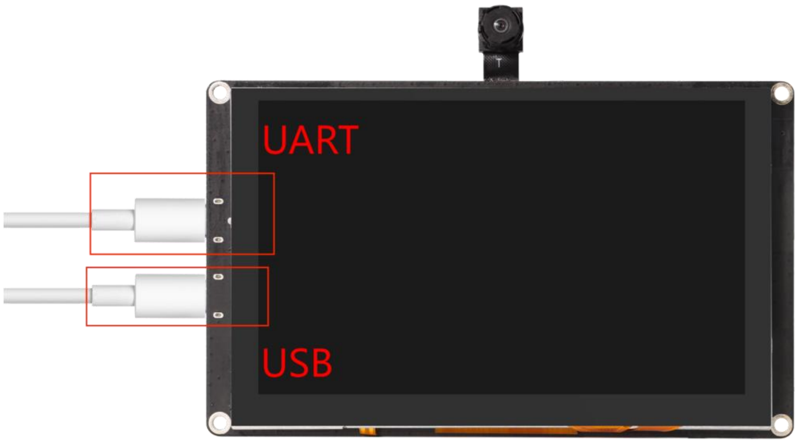
```
        // if X or Y is non-zero, move:
        if ((xDistance!=0) || (yDistance!=0)) {
          Mouse.move(xDistance, yDistance, 0);
          Serial.printf("Mouse.move: x = %2d, y = %2d\r\n", xDistance, yDistance);
```

Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
Touch point[0]: x  486, y 235, strength 26
Mouse.move: x = -4, y =  0
Touch point[0]: x  482, y 236, strength 26
Mouse.move: x = -4, y =  1
Touch point[0]: x  480, y 236, strength 26
Mouse.move: x = -2, y =  0
Touch point[0]: x  474, y 237, strength 26
Mouse.move: x = -6, y =  1
Touch point[0]: x  467, y 240, strength 26
Mouse.move: x = -7, y =  3
Touch point[0]: x  464, y 241, strength 26
Mouse.move: x = -3, y =  1
Touch point[0]: x  459, y 244, strength 26
Mouse.move: x = -5, y =  3
Touch point[0]: x  456, y 247, strength 26
Mouse.move: x = -3, y =  3
Touch point[0]: x  455, y 248, strength 26
Mouse.move: x = -1, y =  1
```

Hardware Used in This Lesson

USB 2.0 on the Advance-P4



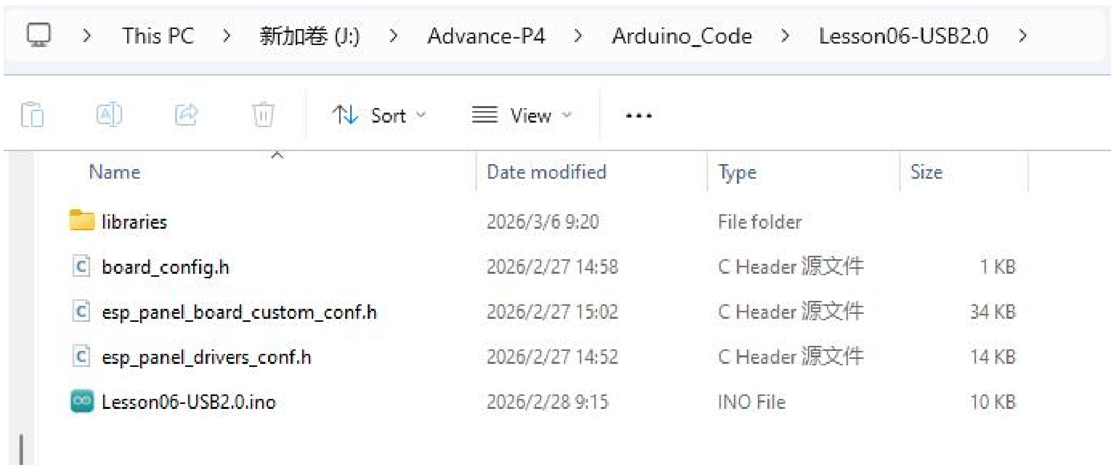
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson06-USB2.0

Key Explanations

Building on the touch screen knowledge from the previous lesson, let's now understand how to connect touch screen input to USB2.0, thereby controlling the mouse on the PC side via USB2.0.

Double-click to open this lesson's code (the .ino file).


This PC > 新加卷 (J:) > Advance-P4 > Arduino_Code > Lesson06-USB2.0 >
Sort
View

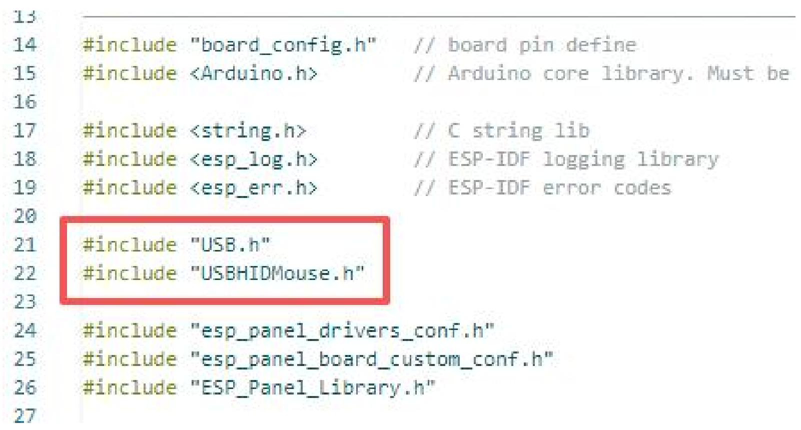
Name	Date modified	Type	Size
libraries	2026/3/6 9:20	File folder	
board_config.h	2026/2/27 14:58	C Header 源文件	1 KB
esp_panel_board_custom_conf.h	2026/2/27 15:02	C Header 源文件	34 KB
esp_panel_drivers_conf.h	2026/2/27 14:52	C Header 源文件	14 KB
Lesson06-USB2.0.ino	2026/2/28 9:15	INO File	10 KB

After opening, you can see the project's code, configuration file board_config.h, and other related screen driver files.
The meanings of these files were explained in detail in the previous lesson.

Next, let's understand how the USB2.0 functionality is implemented in the code.

This program actually does something quite interesting: it turns the ESP32-P4's touch screen into a USB mouse touchpad for the computer. When you slide your finger on the screen, the ESP32 reads the touch coordinates, calculates the finger movement distance, and sends it to the computer via the USB HID protocol. The computer then interprets this as mouse movement.

1. Header Files



```
#include "board_config.h"    // board pin define
#include <Arduino.h>         // Arduino core library. Must be

#include <string.h>          // C string lib
#include <esp_log.h>         // ESP-IDF logging library
#include <esp_err.h>         // ESP-IDF error codes

#include "USB.h"
#include "USBHIDMouse.h"

#include "esp_panel_drivers_conf.h"
#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"
```

(The functions of other library files were explained in detail in the previous lesson.)

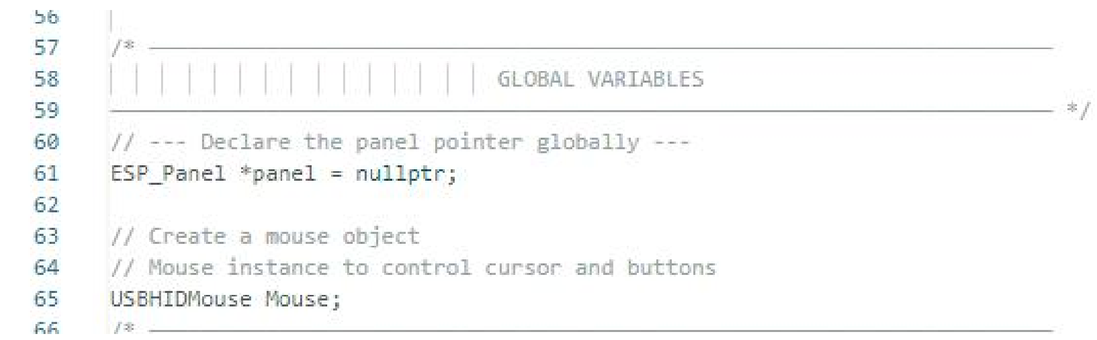
The purpose of these two lines of code is to enable the ESP32 with USB HID mouse device functionality.

USB.h is the core library of the ESP32 USB device framework, used to initialize and manage the USB interface, allowing the chip to communicate with the computer via USB.

USBHIDMouse.h is a HID (Human Interface Device) mouse class library built on top of the USB framework. It encapsulates common mouse operation interfaces, such as Mouse.move() (move mouse pointer), Mouse.press() (press mouse button), Mouse.release() (release button), and Mouse.isPressed() (check button status), etc.

After the program runs and calls USB.begin() and Mouse.begin(), the ESP32 will be recognized on the computer side as a standard USB mouse device. Subsequently, the program only needs to call these functions to send HID reports such as mouse movement, clicks, and scrolling to the computer, thereby achieving control of the computer cursor like a real mouse (for example, in this program, mouse movement is controlled through touch screen coordinate changes).

2. Variable Definitions



```
/* ────────────────────────────────────────────────
 | | | | | | | | | | | | | | | GLOBAL VARIABLES
──────────────────────────────────────────────── */

// --- Declare the panel pointer globally ---
ESP_Panel *panel = nullptr;

// Create a mouse object
// Mouse instance to control cursor and buttons
USBHIDMouse Mouse;
/* ────────────────────────────────────────────────
```

This code mainly completes the declaration of global objects, preparing for subsequent touch reading and USB mouse control.

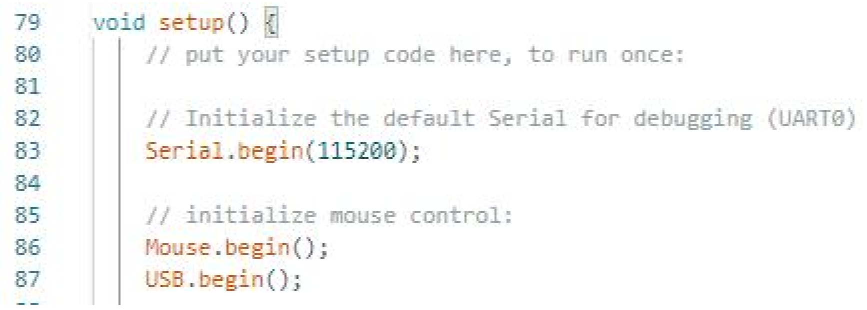
Subsequently, "ESP_Panel *panel = nullptr;" declares a pointer variable of the "ESP_Panel" class and initializes it as a null pointer. It is used to create and manage the display screen and touch controller (such as MIPI-DSI screen and GT911 touch chip) during program operation. Since it is defined as a global variable, this panel object can be accessed in functions such as "setup()" and "loop()".

Finally, "USBHIDMouse Mouse;" creates an instance of a USB HID mouse object. This object encapsulates the operation interfaces of the mouse device, and the program can use it to send mouse movement, click and other commands to the computer—for example, "Mouse.move()" to control cursor movement, "Mouse.press()" and "Mouse.release()" to control mouse buttons—thus realizing the function of operating the computer mouse with the touch screen.

3. setup Section

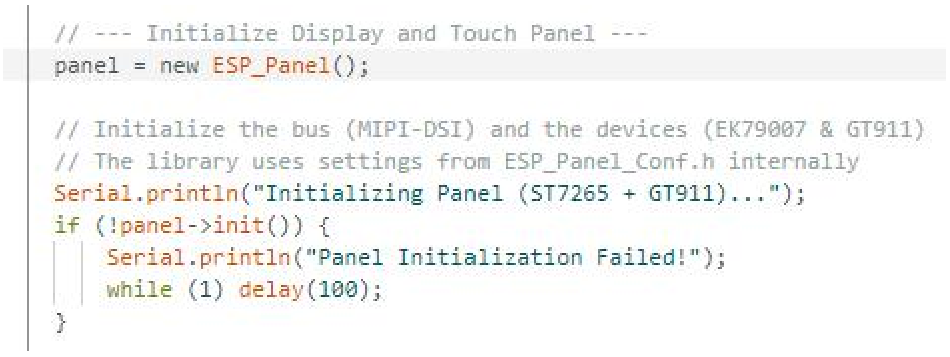
The setup() function code mainly completes system initialization, power configuration, USB mouse function startup, and display and touch screen hardware initialization. It only executes once when the device starts up.

First, serial communication is initialized via Serial.begin(115200) for debugging information output. Then Mouse.begin() and USB.begin() are called to start the USB HID mouse function and USB device protocol, enabling the ESP32 to be recognized as a USB mouse device when connected to a computer.



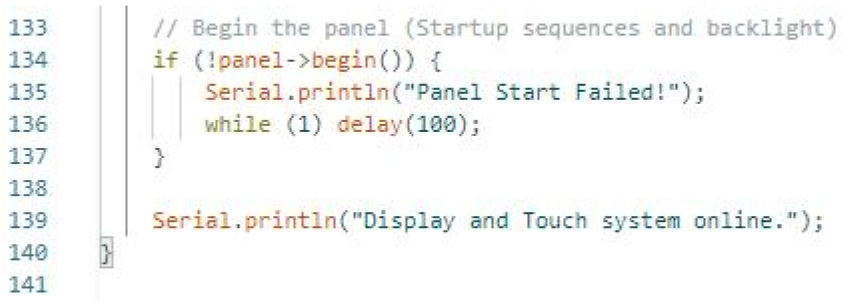
```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    // initialize mouse control:
    Mouse.begin();
    USB.begin();
```

The program creates an ESP_Panel object named "panel = new ESP_Panel()" to manage the display and touchscreen hardware. Then, it calls "panel->init()" to initialize the bus, display driver chip, and GT911 touch controller (the relevant configurations are from the ESP_Panel_Conf.h file in the library). If the initialization fails, the program will enter an infinite loop and stop running.



```
// --- Initialize Display and Touch Panel ---
panel = new ESP_Panel();

// Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
// The library uses settings from ESP_Panel_Conf.h internally
Serial.println("Initializing Panel (ST7265 + GT911)...");
if (!panel->init()) {
    Serial.println("Panel Initialization Failed!");
    while (1) delay(100);
}
```

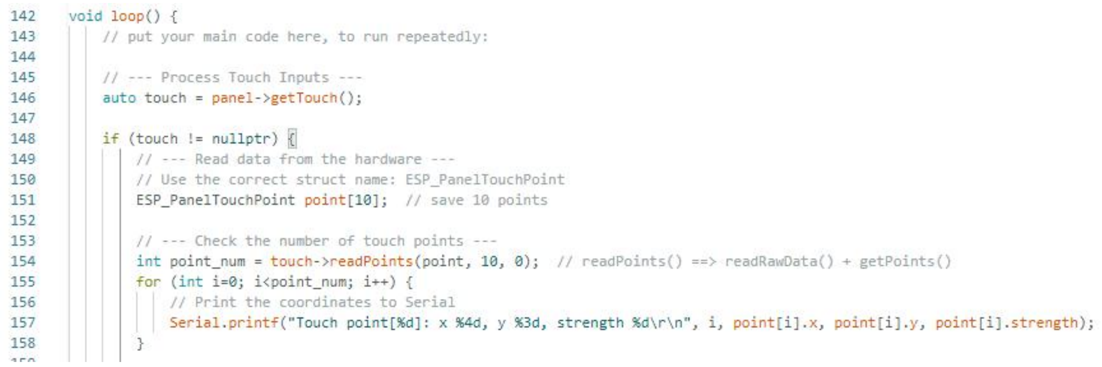
Then call panel->begin() to start the display panel, including executing the display startup sequence and turning on the backlight. If the operation fails, the program will also stop. Finally, print "Display and Touch system online." on the serial port, indicating that the screen display and touch system has successfully started and can enter the main loop to handle touch input and control the mouse.



```
    // Begin the panel (Startup sequences and backlight)
    if (!panel->begin()) {
        Serial.println("Panel Start Failed!");
        while (1) delay(100);
    }

    Serial.println("Display and Touch system online.");
}

```

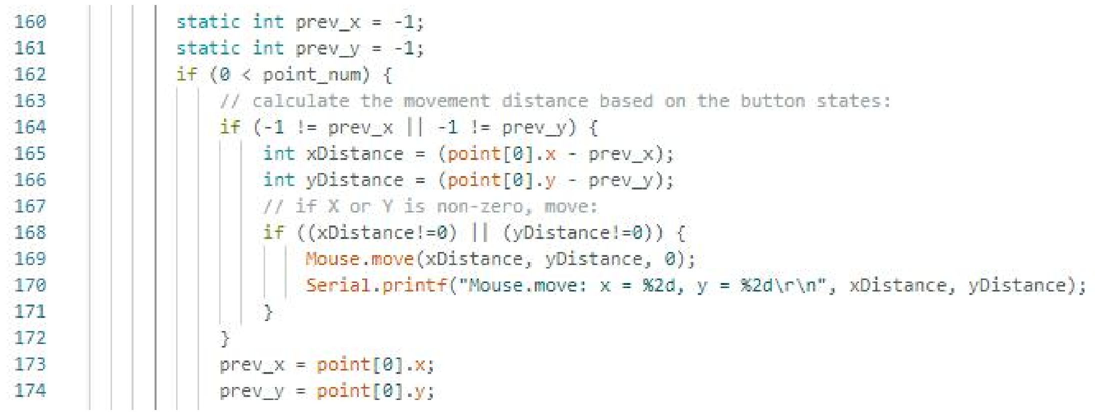
4. Loop Section

This loop() function code implements the core logic for continuously reading touch screen data and converting touch operations into USB mouse control signals.
The program first obtains the touch controller object through panel->getTouch(). If the object exists, it indicates that the touch function is available. Then, an ESP_PanelTouchPoint point[10] array is created to store up to 10 touch point data. The function touch->readPoints(point, 10, 0) is called to read the current number of touch points and their coordinate information from the touch chip (such as GT911). The program outputs the X and Y coordinates of each touch point as well as the touch intensity (pressure/strength) to the serial port through a for loop, facilitating debugging.



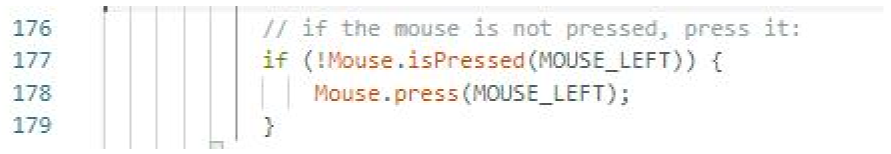
```
void loop() {
    // put your main code here, to run repeatedly:

    // --- Process Touch Inputs ---
    auto touch = panel->getTouch();

    if (touch != nullptr) {
        // --- Read data from the hardware ---
        // Use the correct struct name: ESP_PanelTouchPoint
        ESP_PanelTouchPoint point[10];  // save 10 points

        // --- Check the number of touch points ---
        int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()
        for (int i=0; i<point_num; i++) {
            // Print the coordinates to Serial
            Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\r\n", i, point[i].x, point[i].y, point[i].strength);
        }
```

Then, two static variables, prev_x and prev_y, are used to record the position of the last touch. When at least one touch point is detected (point_num > 0), the program will calculate the displacement difference xDistance and yDistance between the current touch point and the previous touch point. If the displacement is not zero, the Mouse.move(xDistance, yDistance, 0) function is called to send a mouse movement command to the computer, allowing the mouse cursor to move in the direction of the finger's sliding; subsequently, prev_x and prev_y are updated to the current coordinates.



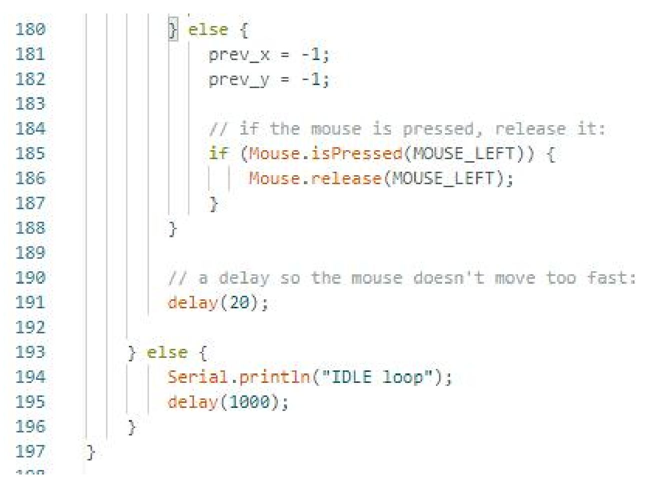
```
        static int prev_x = -1;
        static int prev_y = -1;
        if (0 < point_num) {
            // calculate the movement distance based on the button states:
            if (-1 != prev_x || -1 != prev_y) {
                int xDistance = (point[0].x - prev_x);
                int yDistance = (point[0].y - prev_y);
                // if X or Y is non-zero, move:
                if ((xDistance!=0) || (yDistance!=0)) {
                    Mouse.move(xDistance, yDistance, 0);
                    Serial.printf("Mouse.move: x = %2d, y = %2d\r\n", xDistance, yDistance);
                }
            }
            prev_x = point[0].x;
            prev_y = point[0].y;
```

At the same time, the program simulates the logic of mouse button operations: when a touch is detected, if the left mouse button is not already pressed, then execute Mouse.press(MOUSE_LEFT), which is equivalent to holding down the left mouse button.



```
            // if the mouse is not pressed, press it:
            if (!Mouse.isPressed(MOUSE_LEFT)) {
                Mouse.press(MOUSE_LEFT);
            }
```

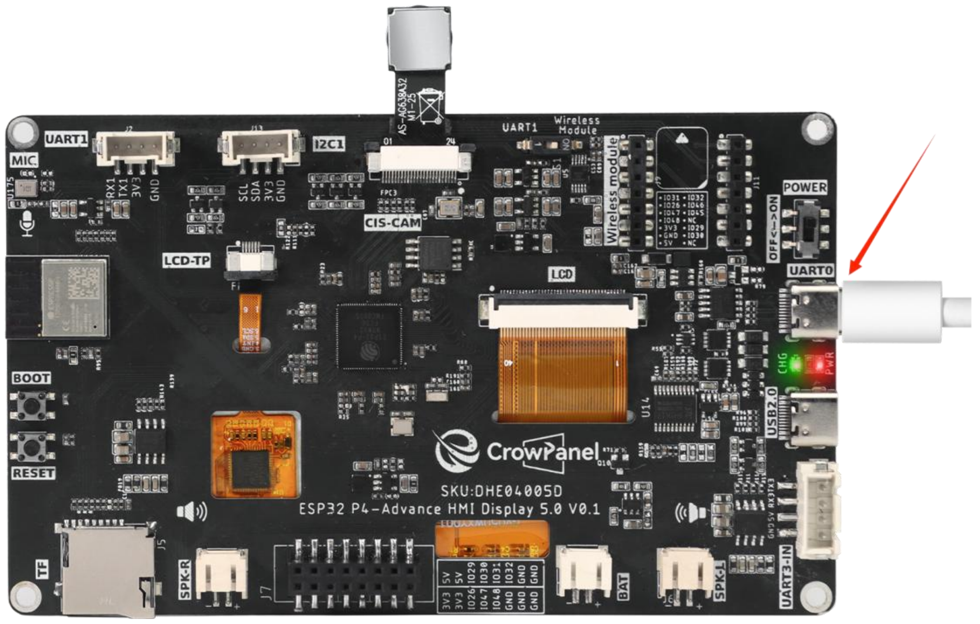
When there is no touch point (when the finger leaves the screen), the program will reset prev_x and prev_y to -1 and call Mouse.release(MOUSE_LEFT) to release the left mouse button, thereby achieving a mouse operation effect similar to "holding and dragging". Finally, through delay(20), the loop execution speed is limited to avoid the mouse moving too fast; if the touch object does not exist, the program outputs "IDLE loop" and checks the status every second.



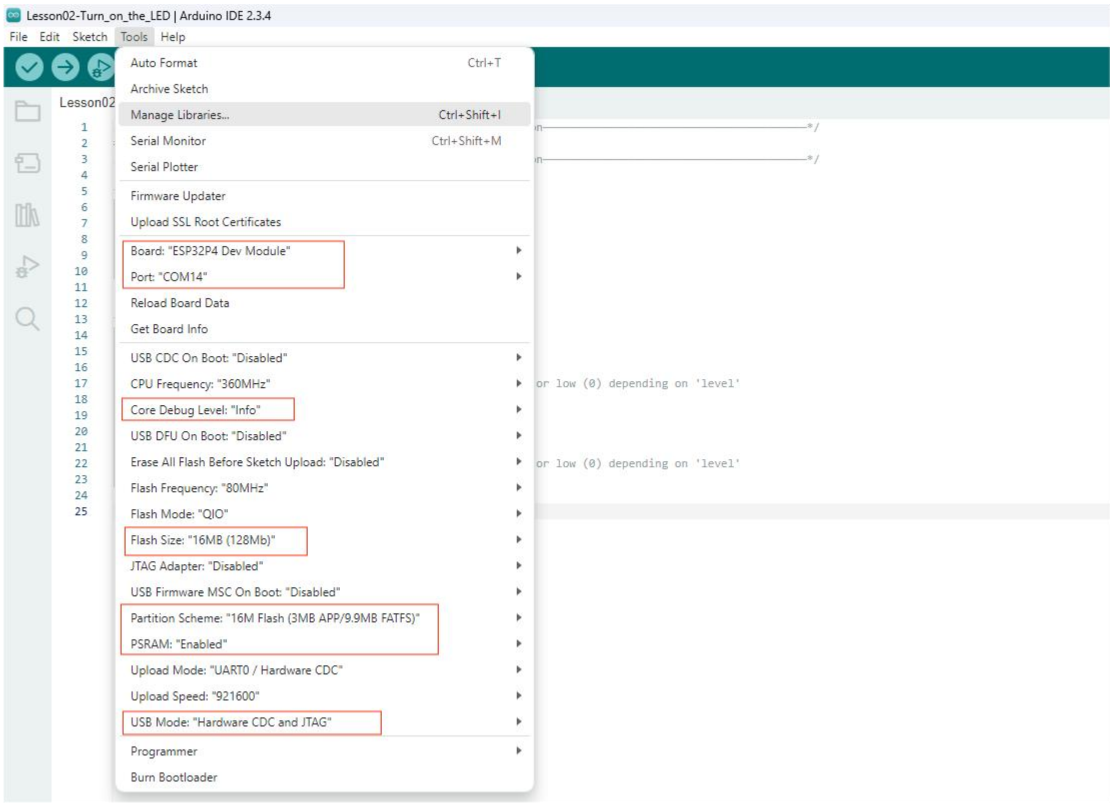
```
        } else {
            prev_x = -1;
            prev_y = -1;

            // if the mouse is pressed, release it:
            if (Mouse.isPressed(MOUSE_LEFT)) {
                Mouse.release(MOUSE_LEFT);
            }
        }

        // a delay so the mouse doesn't move too fast:
        delay(20);

    } else {
        Serial.println("IDLE loop");
        delay(1000);
    }
}
```

Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.

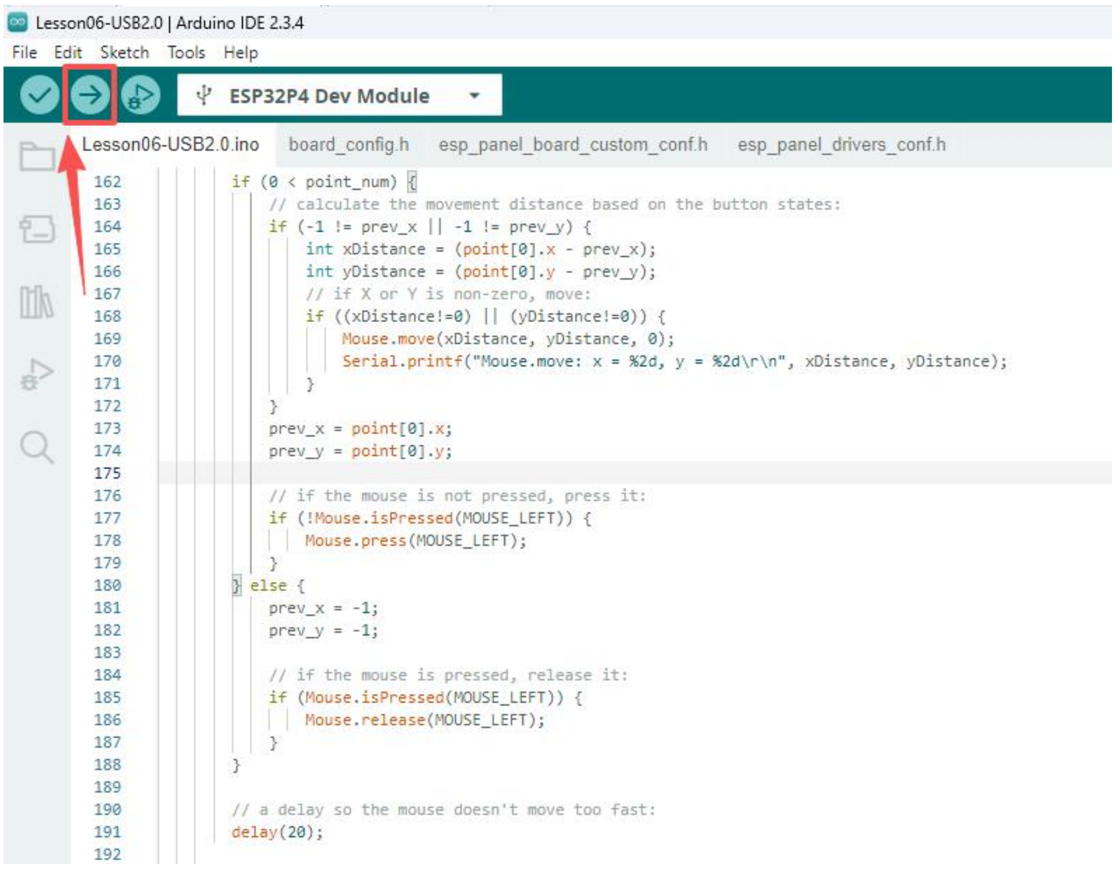


Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

```
Auto Format                                         Ctrl+T
Archive Sketch
Manage Libraries...                                 Ctrl+Shift+I
Serial Monitor                                      Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader
```

Then we compile and upload the code.


Lesson06-USB2.0 | Arduino IDE 2.3.4

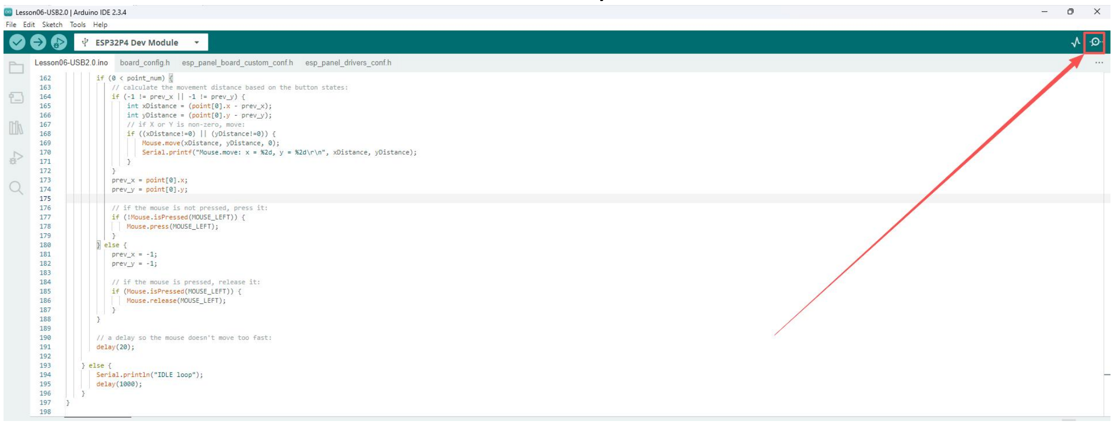
ESP32P4 Dev Module

Lesson06-USB2.0.ino   board_config.h   esp_panel_board_custom_conf.h   esp_panel_drivers_conf.h

```cpp
if (0 < point_num) {
    // calculate the movement distance based on the button states:
    if (-1 != prev_x || -1 != prev_y) {
        int xDistance = (point[0].x - prev_x);
        int yDistance = (point[0].y - prev_y);
        // if X or Y is non-zero, move:
        if ((xDistance!=0) || (yDistance!=0)) {
            Mouse.move(xDistance, yDistance, 0);
            Serial.printf("Mouse.move: x = %2d, y = %2d\r\n", xDistance, yDistance);
        }
    }
    prev_x = point[0].x;
    prev_y = point[0].y;

    // if the mouse is not pressed, press it:
    if (!Mouse.isPressed(MOUSE_LEFT)) {
        Mouse.press(MOUSE_LEFT);
    }
} else {
    prev_x = -1;
    prev_y = -1;

    // if the mouse is pressed, release it:
    if (Mouse.isPressed(MOUSE_LEFT)) {
        Mouse.release(MOUSE_LEFT);
    }
}

// a delay so the mouse doesn't move too fast:
delay(20);
```

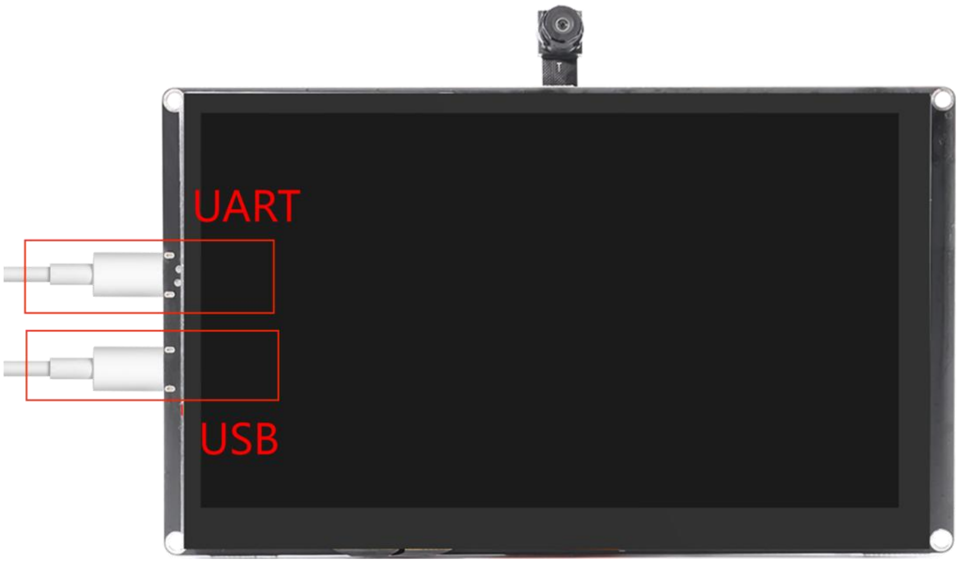
After the code upload completes, you can open the Serial Monitor built into the

Arduino-IDE and set the same baud rate as in your code.

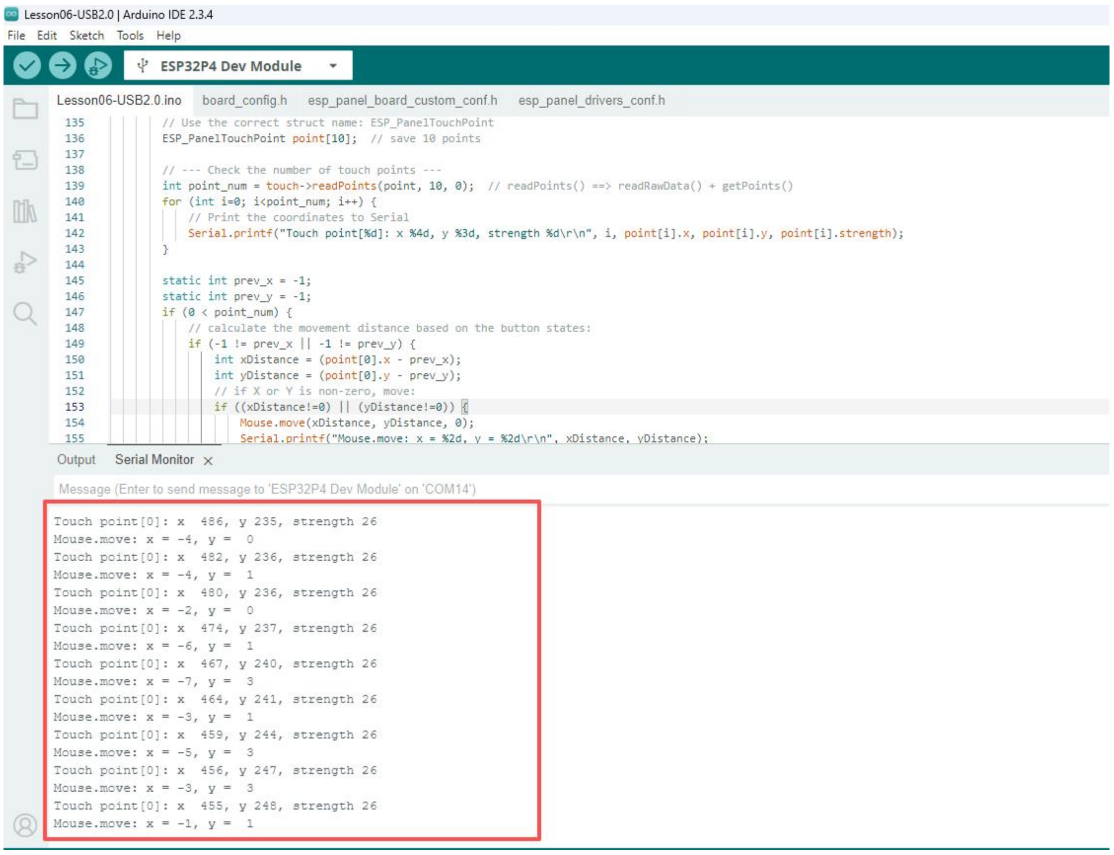

Lesson06-USB2.0 | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson06-USB2.0.ino
board_config.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h

```
            if (0 < point_num) {
                // calculate the movement distance based on the button states:
                if (-1 != prev_x || -1 != prev_y) {
                    int xDistance = (point[0].x - prev_x);
                    int yDistance = (point[0].y - prev_y);
                    // if X or Y is non-zero, move:
                    if ((xDistance!=0) || (yDistance!=0)) {
                        Mouse.move(xDistance, yDistance, 0);
                        Serial.printf("Mouse.move: x = %2d, y = %2d\r\n", xDistance, yDistance);
                    }
                }
                prev_x = point[0].x;
                prev_y = point[0].y;

                // if the mouse is not pressed, press it:
                if (!Mouse.isPressed(MOUSE_LEFT)) {
                    Mouse.press(MOUSE_LEFT);
                }
            } else {
                prev_x = -1;
                prev_y = -1;

                // if the mouse is pressed, release it:
                if (Mouse.isPressed(MOUSE_LEFT)) {
                    Mouse.release(MOUSE_LEFT);
                }
            }

            // a delay so the mouse doesn't move too fast:
            delay(20);

        } else {
            Serial.println("IDLE loop");
            delay(1000);
        }
}
```

At this point, please make sure to connect your Advance-P4 to the computer via a separate Type-C data cable using the USB2.0 interface. Only in this way can communication be carried out using the USB2.0 protocol.



When you slide the screen of the Advance-P4, the mouse on your computer also moves along. At this moment, your Advance-P4 becomes your new mouse. Meanwhile, you can also see the corresponding coordinates printed on the monitor when you turn it on.


Lesson06-USB2.0 | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson06-USB2.0.ino
board_config.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
        // Use the correct struct name: ESP_PanelTouchPoint
        ESP_PanelTouchPoint point[10];  // save 10 points

        // --- Check the number of touch points ---
        int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()
        for (int i=0; i<point_num; i++) {
          // Print the coordinates to Serial
          Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\r\n", i, point[i].x, point[i].y, point[i].strength);
        }

        static int prev_x = -1;
        static int prev_y = -1;
        if (0 < point_num) {
          // calculate the movement distance based on the button states:
          if (-1 != prev_x || -1 != prev_y) {
            int xDistance = (point[0].x - prev_x);
            int yDistance = (point[0].y - prev_y);
            // if X or Y is non-zero, move:
            if ((xDistance!=0) || (yDistance!=0)) {
              Mouse.move(xDistance, yDistance, 0);
              Serial.printf("Mouse.move: x = %2d, y = %2d\r\n", xDistance, yDistance);
```

```
Touch point[0]: x  486, y 235, strength 26
Mouse.move: x = -4, y =  0
Touch point[0]: x  482, y 236, strength 26
Mouse.move: x = -4, y =  1
Touch point[0]: x  480, y 236, strength 26
Mouse.move: x = -2, y =  0
Touch point[0]: x  474, y 237, strength 26
Mouse.move: x = -6, y =  1
Touch point[0]: x  467, y 240, strength 26
Mouse.move: x = -7, y =  3
Touch point[0]: x  464, y 241, strength 26
Mouse.move: x = -3, y =  1
Touch point[0]: x  459, y 244, strength 26
Mouse.move: x = -5, y =  3
Touch point[0]: x  456, y 247, strength 26
Mouse.move: x = -3, y =  3
Touch point[0]: x  455, y 248, strength 26
Mouse.move: x = -1, y =  1
```

Lesson07---Turn on the screen

Introduction

In this class, we will start by teaching you how to turn on the screen. Then, while turning on the screen backlight, we will display "Hellow Elecrow" on the screen. Of course, you can replace it with whatever you want.
The main focus of this class is to teach you how to turn on the screen backlight and turn on the screen, in preparation for the subsequent courses.

Learning Goals

1. Understand the display principles of the screen.
2. Understand the role of LVGL in displaying information.
3. Implement the screen text display function by utilizing libraries.

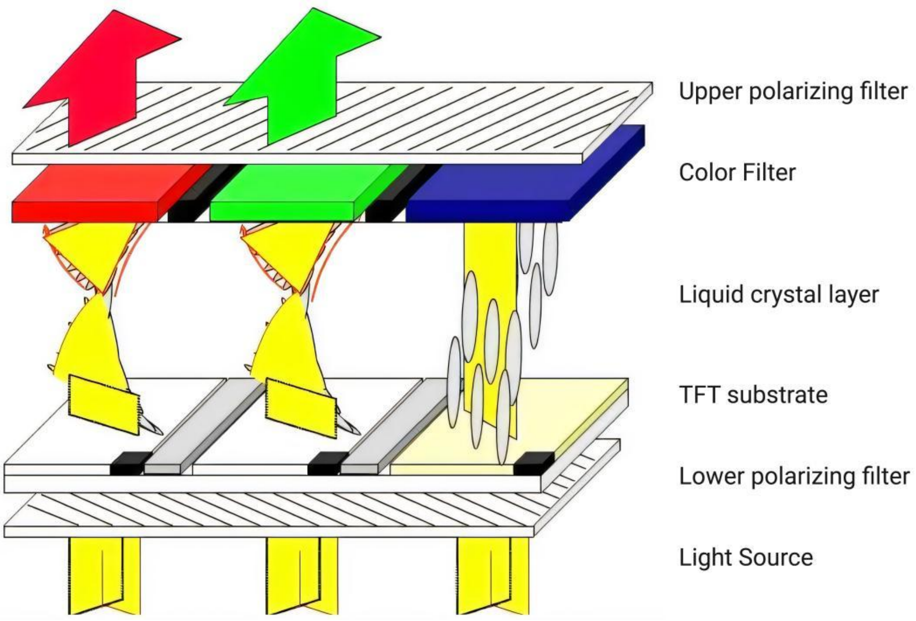
Preview of the Result

After running the code, you will be able to visually see that "Hello Elecrow" is displayed on the screen of the Advance-P4.

Hardware Used in This Lesson

The screen on the Advance-P4

Display Screen CXM090IPS-D27 Schematic Diagram



Firstly, the backlight (usually an LED array) emits a white surface light source, providing the basic light for display.

Then, the lower polarizer polarizes and filters the light from the backlight, allowing only light of a specific polarization direction (such as horizontal) to pass through, forming linearly polarized light. Next, the light reaches the TFT substrate, where the thin-film transistors (TFTs) on the substrate act as switching devices, controlling the electrical state of the liquid crystal molecules in the corresponding pixel area based on the applied voltage, thereby changing the alignment direction of the liquid crystal molecules.

Liquid crystal molecules have optical anisotropy and electric field response characteristics. The change in their alignment direction modulates the polarization state of the passing polarized light. Subsequently, the light enters the color filter, which is composed of red, green, and blue primary color filter units.

Only light corresponding to the color of the filter units (for example, only red light can pass through the red filter unit) can pass through, generating primary color light.

Finally, the upper polarizer (whose polarization direction is perpendicular to that of the lower polarizer, such as horizontal for the lower polarizer and vertical for the upper polarizer) filters the light that has passed through the color filter again.

Only light with a polarization direction consistent with the allowed direction of the upper polarizer can pass through.

Through the precise control of the liquid crystal molecules in each pixel by the TFT substrate, the polarization state of the polarized light is adjusted. Combined with the color filtering of the color filter and the polarization selection of the upper and lower polarizers, different pixels present different brightness and colors, ultimately forming a visible color image.

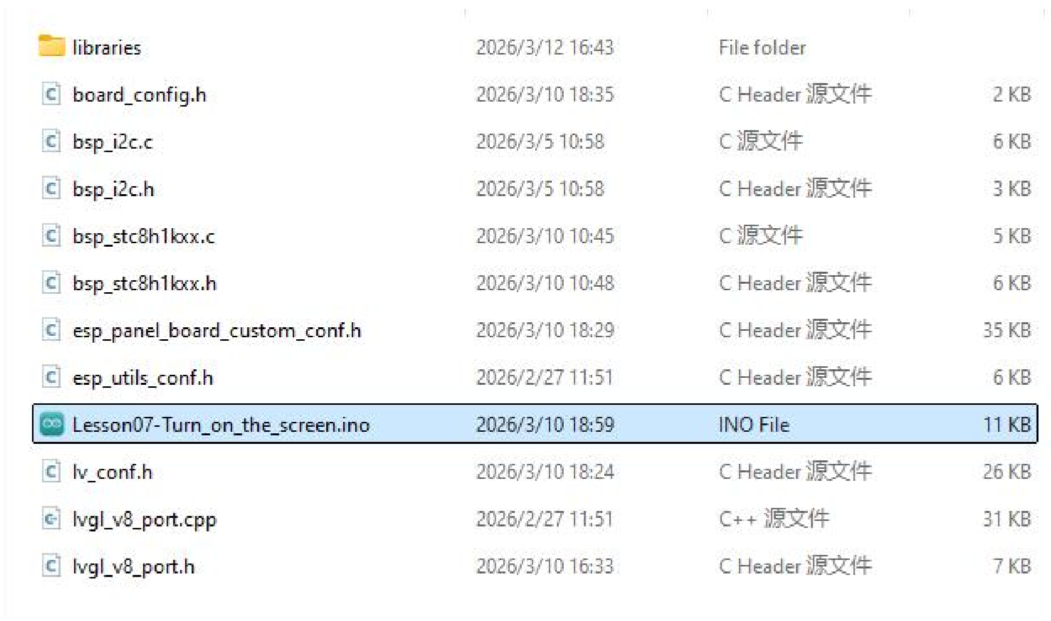
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson07-Turn_on_the_screen

Key Explanations

How do we display text on the screen of the CrowPanel Advanced ESP32-P4 HMI AI Display? Next, let's dive into the specifics together.

Double-click to open this lesson's code (the .ino file).



libraries	2026/3/12 16:43	File folder	
board_config.h	2026/3/10 18:35	C Header 源文件	2 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/10 10:48	C Header 源文件	6 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/2/27 11:51	C Header 源文件	6 KB
Lesson07-Turn_on_the_screen.ino	2026/3/10 18:59	INO File	11 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

After opening, you can see the project's code, configuration file board_config.h, and other related screen driver files. Let's understand the project architecture for this lesson.

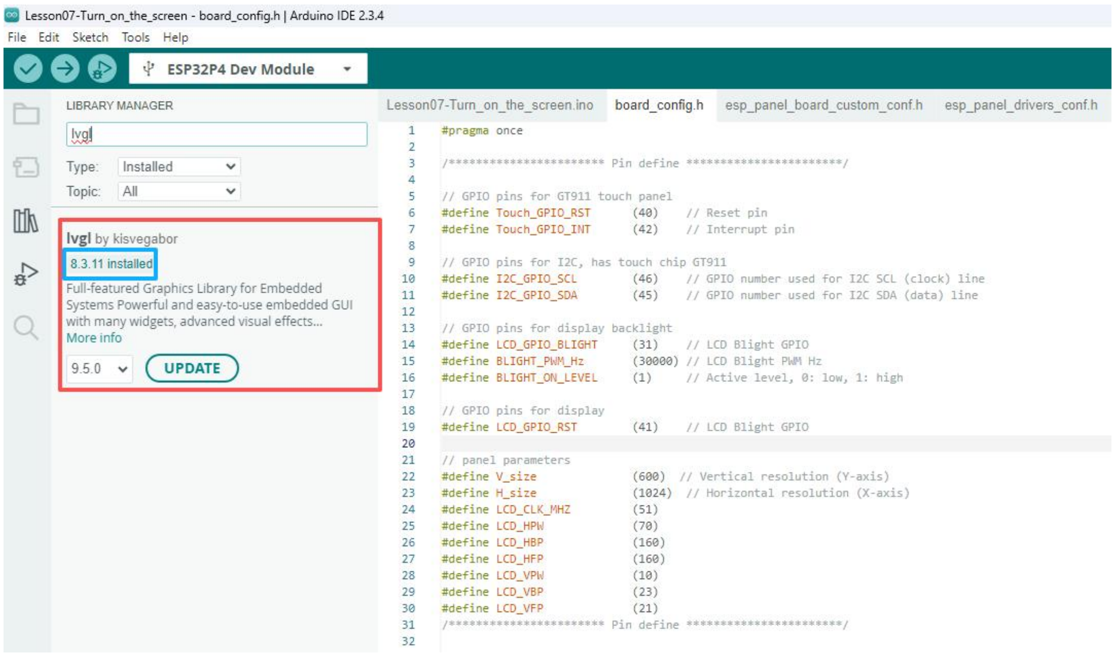
Among them:
1. The libraries folder is the project's dependency library storage directory, storing core third-party libraries such as ESP32_Display_Panel and LVGL v8 according to Arduino library management standards.

In the preface, we have already imported all the library files into the Arduino's "libraries" folder. If you haven't done so, please follow the instructions in the preface to import all the library files into the "libraries" folder of Arduino.

This directory contains all library source code and header files required for compiling the project, ensuring the Arduino IDE can correctly locate and link the ESP32_Display_Panel library, LVGL library, and their dependencies. It forms the library dependency foundation for the project to compile and run normally, avoiding version inconsistency issues from manual library installation.

The ESP32_Display_Panel library is consistent with what we covered in Lesson 5, and the version is also 1.0.4.

The LVGL library is version 8.3.11, please keep this consistent.

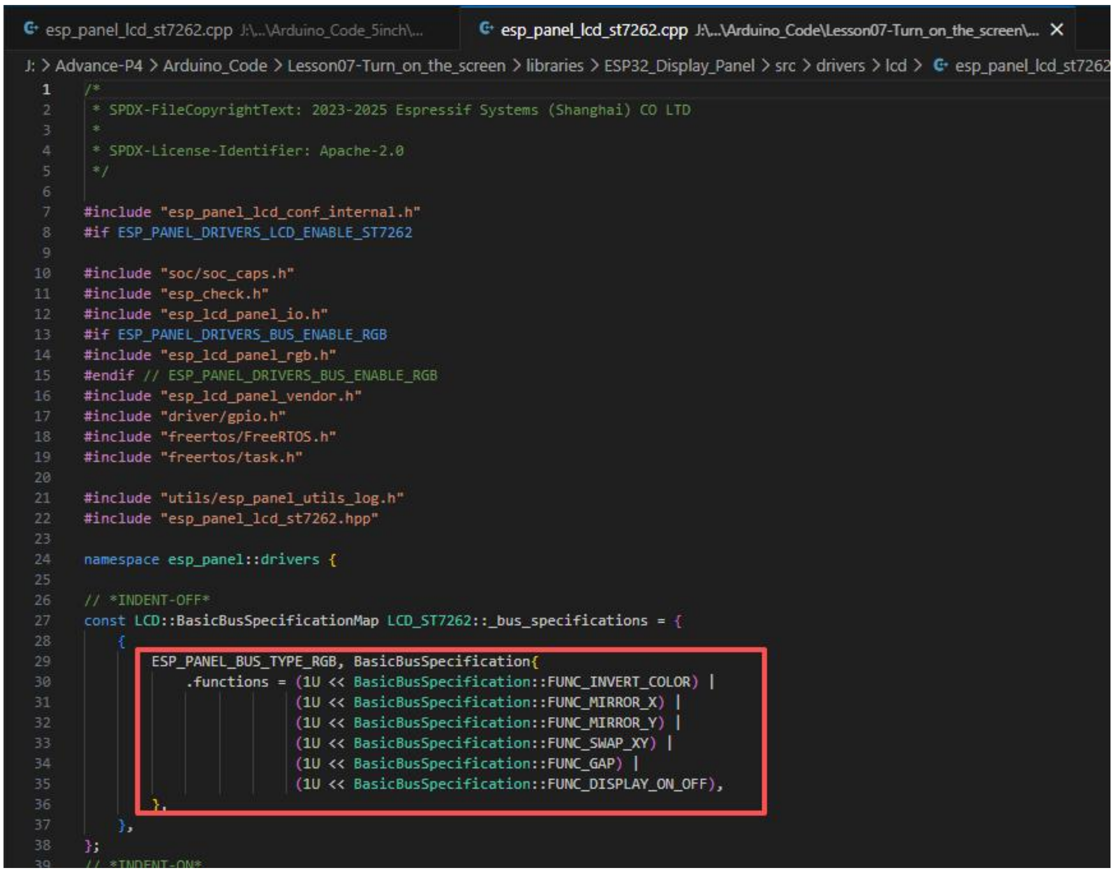

Lesson07-Turn_on_the_screen - board_config.h | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
LIBRARY MANAGER
lvgl
Type: Installed
Topic: All
lvgl by kisvegabor
8.3.11 installed
Full-featured Graphics Library for Embedded Systems Powerful and easy-to-use embedded GUI with many widgets, advanced visual effects...
More info
9.5.0
UPDATE
Lesson07-Turn_on_the_screen.ino
board_config.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h

```
#pragma once

/*********************** Pin define **********************/

// GPIO pins for GT911 touch panel
#define Touch_GPIO_RST      (40)    // Reset pin
#define Touch_GPIO_INT      (42)    // Interrupt pin

// GPIO pins for I2C, has touch chip GT911
#define I2C_GPIO_SCL        (46)    // GPIO number used for I2C SCL (clock) line
#define I2C_GPIO_SDA        (45)    // GPIO number used for I2C SDA (data) line

// GPIO pins for display backlight
#define LCD_GPIO_BLIGHT     (31)    // LCD Blight GPIO
#define BLIGHT_PWM_Hz       (30000) // LCD Blight PWM Hz
#define BLIGHT_ON_LEVEL     (1)     // Active level, 0: low, 1: high

// GPIO pins for display
#define LCD_GPIO_RST        (41)    // LCD Blight GPIO

// panel parameters
#define V_size              (600)  // Vertical resolution (Y-axis)
#define H_size              (1024)  // Horizontal resolution (X-axis)
#define LCD_CLK_MHZ         (51)
#define LCD_HPW             (70)
#define LCD_HBP             (160)
#define LCD_HFP             (160)
#define LCD_VPW             (10)
#define LCD_VBP             (23)
#define LCD_VFP             (21)
/*********************** Pin define **********************/

```

However, please note here, it is extremely important!!

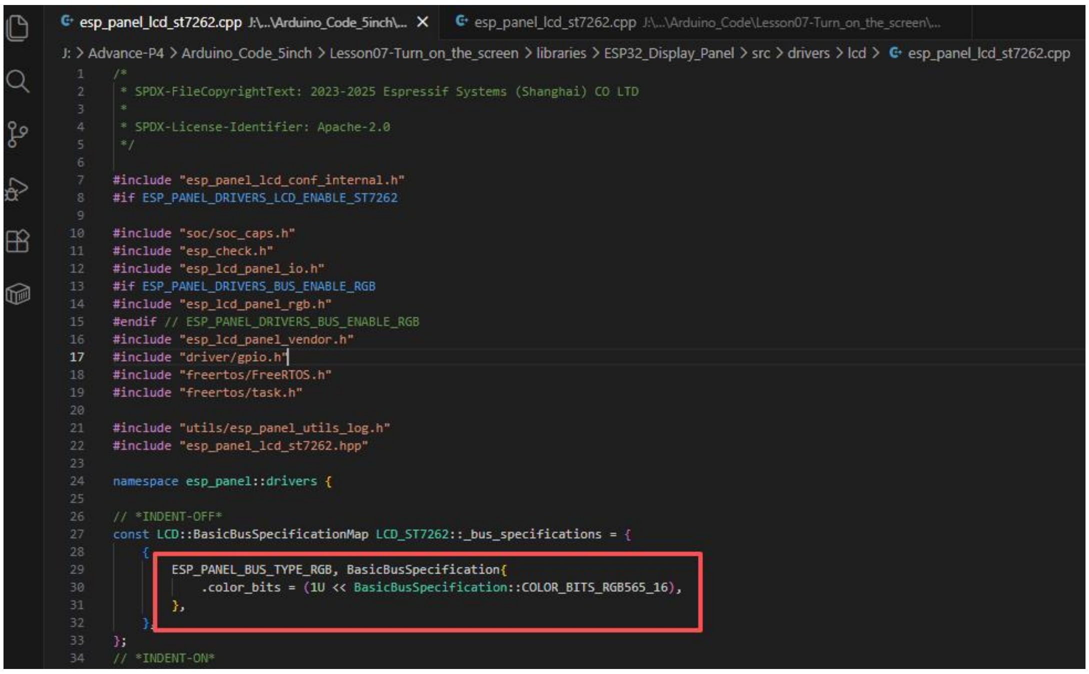
Please directly use the library files provided by us. Because for the 5-inch Advance-P4, we need to modify the library files to adapt to the 5-inch ST7265 display we are using, so that the code will not report errors and the function can display normally.

before modification:


esp_panel_lcd_st7262.cpp J:\...\Arduino_Code_5inch\...
esp_panel_lcd_st7262.cpp J:\...\Arduino_Code\Lesson07-Turn_on_the_screen\...
J: > Advance-P4 > Arduino_Code > Lesson07-Turn_on_the_screen > libraries > ESP32_Display_Panel > src > drivers > lcd > esp_panel_lcd_st7262

```
/*
 * SPDX-FileCopyrightText: 2023-2025 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include "esp_panel_lcd_conf_internal.h"
#if ESP_PANEL_DRIVERS_LCD_ENABLE_ST7262

#include "soc/soc_caps.h"
#include "esp_check.h"
#include "esp_lcd_panel_io.h"
#if ESP_PANEL_DRIVERS_BUS_ENABLE_RGB
#include "esp_lcd_panel_rgb.h"
#endif // ESP_PANEL_DRIVERS_BUS_ENABLE_RGB
#include "esp_lcd_panel_vendor.h"
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#include "utils/esp_panel_utils_log.h"
#include "esp_panel_lcd_st7262.hpp"

namespace esp_panel::drivers {

// *INDENT-OFF*
const LCD::BasicBusSpecificationMap LCD_ST7262::_bus_specifications = {
    {
        ESP_PANEL_BUS_TYPE_RGB, BasicBusSpecification{
            .functions = (1U << BasicBusSpecification::FUNC_INVERT_COLOR) |
                         (1U << BasicBusSpecification::FUNC_MIRROR_X) |
                         (1U << BasicBusSpecification::FUNC_MIRROR_Y) |
                         (1U << BasicBusSpecification::FUNC_SWAP_XY) |
                         (1U << BasicBusSpecification::FUNC_GAP) |
                         (1U << BasicBusSpecification::FUNC_DISPLAY_ON_OFF),
        },
    },
};
// *INDENT-ON*
```

after modification:



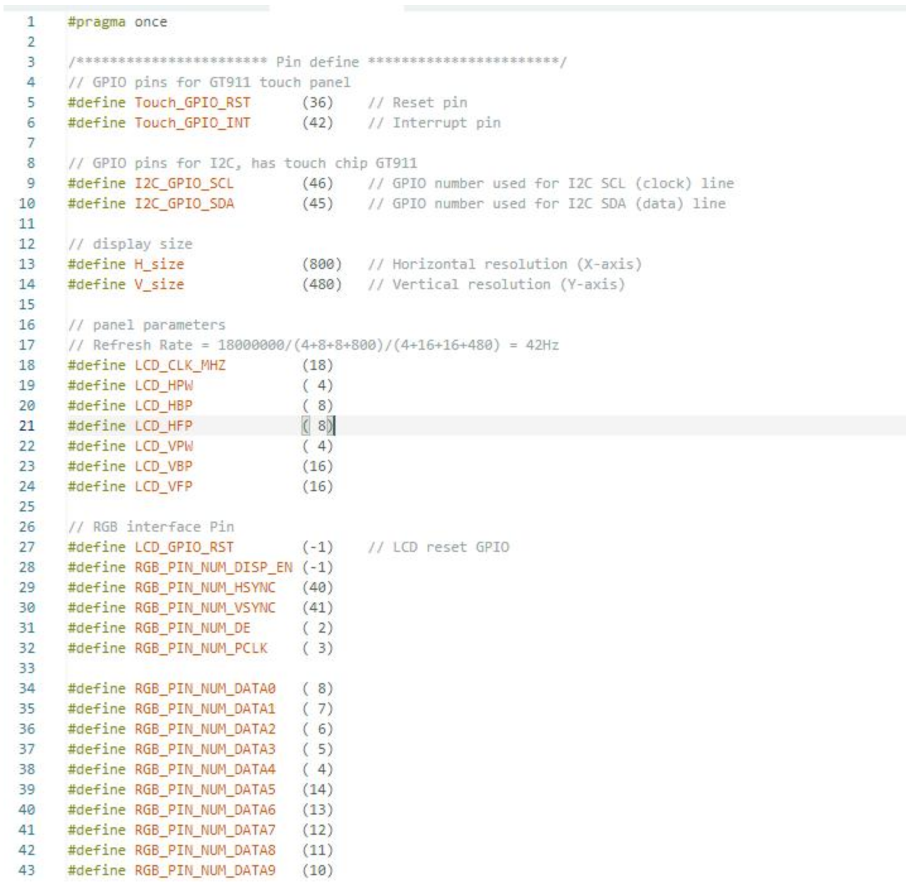
esp_panel_lcd_st7262.cpp J:\...\Arduino_Code_5inch\... × | esp_panel_lcd_st7262.cpp J:\...\Arduino_Code\Lesson07-Turn_on_the_screen\...

J: > Advance-P4 > Arduino_Code_5inch > Lesson07-Turn_on_the_screen > libraries > ESP32_Display_Panel > src > drivers > lcd > esp_panel_lcd_st7262.cpp

```cpp
/*
 * SPDX-FileCopyrightText: 2023-2025 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include "esp_panel_lcd_conf_internal.h"
#if ESP_PANEL_DRIVERS_LCD_ENABLE_ST7262

#include "soc/soc_caps.h"
#include "esp_check.h"
#include "esp_lcd_panel_io.h"
#if ESP_PANEL_DRIVERS_BUS_ENABLE_RGB
#include "esp_lcd_panel_rgb.h"
#endif // ESP_PANEL_DRIVERS_BUS_ENABLE_RGB
#include "esp_lcd_panel_vendor.h"
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#include "utils/esp_panel_utils_log.h"
#include "esp_panel_lcd_st7262.hpp"

namespace esp_panel::drivers {

// *INDENT-OFF*
const LCD::BasicBusSpecificationMap LCD_ST7262::_bus_specifications = {
    {
        ESP_PANEL_BUS_TYPE_RGB, BasicBusSpecification{
            .color_bits = (1U << BasicBusSpecification::COLOR_BITS_RGB565_16),
        },
    }
};
// *INDENT-ON*
```

2. board_config.h：Hardware pins and panel parameter configuration file

This code is a development board hardware configuration header file named board_config.h. Its main function is to centrally define the GPIO pins used by the display, touch screen, and communication interface, as well as the screen parameters, so that they can be uniformly called during the compilation of the entire project.



```c
#pragma once

/*********************** Pin define ***********************/
// GPIO pins for GT911 touch panel
#define Touch_GPIO_RST        (36)    // Reset pin
#define Touch_GPIO_INT        (42)    // Interrupt pin

// GPIO pins for I2C, has touch chip GT911
#define I2C_GPIO_SCL          (46)    // GPIO number used for I2C SCL (clock) line
#define I2C_GPIO_SDA          (45)    // GPIO number used for I2C SDA (data) line

// display size
#define H_size                (800)   // Horizontal resolution (X-axis)
#define V_size                (480)   // Vertical resolution (Y-axis)

// panel parameters
// Refresh Rate = 18000000/(4+8+8+800)/(4+16+16+480) = 42Hz
#define LCD_CLK_MHZ           (18)
#define LCD_HPW               ( 4)
#define LCD_HBP               ( 8)
#define LCD_HFP               ( 8)
#define LCD_VPW               ( 4)
#define LCD_VBP               (16)
#define LCD_VFP               (16)

// RGB interface Pin
#define LCD_GPIO_RST          (-1)    // LCD reset GPIO
#define RGB_PIN_NUM_DISP_EN (-1)
#define RGB_PIN_NUM_HSYNC     (40)
#define RGB_PIN_NUM_VSYNC     (41)
#define RGB_PIN_NUM_DE        ( 2)
#define RGB_PIN_NUM_PCLK      ( 3)

#define RGB_PIN_NUM_DATA0     ( 8)
#define RGB_PIN_NUM_DATA1     ( 7)
#define RGB_PIN_NUM_DATA2     ( 6)
#define RGB_PIN_NUM_DATA3     ( 5)
#define RGB_PIN_NUM_DATA4     ( 4)
#define RGB_PIN_NUM_DATA5     (14)
#define RGB_PIN_NUM_DATA6     (13)
#define RGB_PIN_NUM_DATA7     (12)
#define RGB_PIN_NUM_DATA8     (11)
#define RGB_PIN_NUM_DATA9     (10)
```

The beginning of the file uses #pragma once to prevent the header file from being included multiple times. Then, it first defines the touch screen-related pins: Touch_GPIO_RST and Touch_GPIO_INT respectively represent the reset pin and interrupt pin of the touch controller GT911, which are used to control the startup of the touch chip and notify touch events; subsequently, it defines the I2C bus pins I2C_GPIO_SCL and I2C_GPIO_SDA, which are used for data communication between the MCU and the touch chip.

Then the code defines the screen resolution parameters. Here, H_size represents the horizontal resolution of 800 pixels, and V_size represents the vertical resolution of 480 pixels, indicating that this screen is a display panel of 800×480. Next, the display timing parameters are configured, including pixel clock LCD_CLK_MHZ (18 MHz), as well as horizontal synchronization pulse width LCD_HPW, horizontal trailing edge LCD_HBP, horizontal leading edge LCD_HFP, vertical synchronization pulse width LCD_VPW, vertical trailing edge LCD_VBP, and vertical leading edge LCD_VFP, etc. These parameters are used to control the refresh timing of the screen and are calculated according to the formula to determine the screen refresh rate to be approximately 42Hz.

The last part defines the GPIO pin mapping of the RGB parallel display interface, including the synchronization signal pins HSYNC, VSYNC, DE (Data Enable), and the pixel clock PCLK. It also defines 16 data lines RGB_PIN_NUM_DATA0 to RGB_PIN_NUM_DATA15 for transmitting pixel color data; among them, LCD_GPIO_RST and RGB_PIN_NUM_DISP_EN are set to -1, indicating that the corresponding GPIO control is not used by the current hardware.

Overall, this document serves as a hardware connection manual for the entire development board display system. It clearly stipulates the relationships between the touch controller, I2C communication, screen resolution, refresh timing, and the RGB data line and each GPIO pin of the ESP32-P4, enabling the display driver library to correctly configure the hardware interface and drive the 800×480 RGB display to operate normally during screen initialization.

3. esp_panel_board_custom_conf.h ： ESP display panel custom board-level configuration file

The esp_panel_board_custom_conf.h file is the core configuration file in the ESP32_Display_Panel library used for customizing the hardware configuration of the development board. Its main function is to uniformly configure parameters such as the display screen, touch screen, backlight, power supply, bus interface, and expansion hardware of the user's development board through a large number of macro definitions. This way, the library can correctly match the specific hardware during initialization and automatically complete the driver configuration.

This configuration file is equivalent to the "hardware description file" of the entire display system. It centrally describes all the hardware parameters of the screen, touch, backlight, and related interfaces on the development board, enabling the program to automatically complete the initialization and driver loading of the display and touch devices when calling Board::init() and Board::begin().

4. esp_panel_drivers_conf.h： ESP display panel driver selection configuration file
esp_panel_drivers_conf.h is a driver function configuration file in the ESP32_Display_Panel library. Its main function is to control whether the display system-related driver modules are compiled and used through a series of macro definitions, thereby determining which bus drivers, LCD drivers, touch drivers, IO expanders drivers, and backlight drivers the program ultimately includes, and by trimming unused drivers, reducing the firmware size, improving compilation speed, and optimizing system resource usage.
The core function of this file can be understood as the "function trimming and driver selection center" of the ESP32 display driver system. Developers can decide which display buses, screen controllers, touch chips, and backlight methods the system supports by modifying the macros in this file, so that the driver library only contains the functional modules that the current project truly needs during the compilation stage.

5. esp_utils_conf.h： ESP tool library common configuration file
esp_utils_conf.h is the common configuration file of the esp_utils tool library that the project relies on. It controls the basic behaviors of the tool library: including handling methods for null pointers / memory allocation errors (printing error logs), log output levels (INFO level), memory allocation strategies (default uses standard library malloc/free, can be switched to ESP-IDF exclusive memory allocation) and so on. It provides unified log, memory management and error handling specifications for the entire project, and is the basic tool configuration for ensuring the robustness and maintainability of the project code.

6. lv_conf.h： The core configuration file of the LVGL graphics library
The file lv_conf.h is the core compilation configuration file for the LVGL v8.3.11 graphics library. Through macro definitions, it comprehensively controls the functionality range and resource usage of LVGL: the color depth is set to RGB565 (16 bits), the memory allocation strategy is system malloc/free, the hardware abstraction layer parameters (refresh rate 30ms, input device read cycle 30ms), while enabling complex drawing engines (shadows, gradients, rounded corners, etc.), full set of commonly used controls (buttons, sliders, labels, etc.), the entire series of Montserrat fonts, as well as Demo functions such as control demonstration and performance benchmark testing. It is the "master switch" of the LVGL graphics system, determining which UI capabilities LVGL can provide, how much memory it consumes, and is the key to adapting to the hardware resources of ESP32-P4.

7. lvgl_v8_port.h： LVGL Porting Layer Interface and Configuration Header File
lvgl_v8_port.h is the porting layer interface file that connects the native LVGL library with the ESP32-P4 hardware. On one hand, it defines the key parameters for LVGL to adapt to the ESP32: the LVGL clock tick period (2ms), the type of memory allocation in the buffer (default SRAM, can be switched to PSRAM), anti-aliasing mode (double buffering + full refresh), screen rotation angle (0°), etc.; on the other hand, it declares the core interfaces of the porting layer (lvgl_port_init/deinit, lvgl_port_lock/unlock), providing standardized entry points for LVGL initialization, deinitialization, and thread-safe lock operations for upper-layer applications, serving as the "interface contract" between the upper-layer business and the underlying hardware adaptation.

8. lvgl_v8_port.cpp： The hardware adaptation implementation file of the LVGL transplantation layer
The file lvgl_v8_port.cpp is the core implementation file for transplanting LVGL to the ESP32-P4 platform. It fully implements the adaptation logic between LVGL and the hardware: for MIPI-DSI screens, it implements anti-tearing optimization (synchronizing frame buffer switching through VSYNC interrupt), for 16-bit color depth screens, it optimizes the rotation algorithm (block copying improves performance by 10-20 times), completes the initialization of LCD display buffer, touch interrupt synchronization and coordinate mapping, and at the same time creates an independent LVGL task to handle timer events. It is the core carrier of LVGL transforming from a general graphics library to an executable graphics system for ESP32-P4.

(The above library files can be used directly by everyone. We have already made the necessary modifications for you, so you can easily utilize the screen display functions and LVGL functions on the CrowPanel Advanced ESP32-P4 HMI AI Display.)

9. bsp_stc8h1kxx.h
This code is a header file (BSP driver interface file) for the STC8H1KXX peripheral control module. It is mainly used in the ESP32-P4 system to control and read the battery management, GPIO control, and PWM output functions provided by the STC8H1KXX chip through I2C communication.

The beginning of the file uses the #ifndef / #define / #endif structure to prevent the header file from being included multiple times, and includes several necessary libraries, such as the FreeRTOS task library, the ESP-IDF logging system esp_log.h, error codes esp_err.h, UART driver, GPIO driver, and the custom I2C driver bsp_i2c.h, providing a foundation for subsequent communication with the STC chip.

```
/*-----------------------------------------
#include <string.h>
#include <stdint.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/uart.h"
#include "bsp_i2c.h"
#include "driver/gpio.h"
```

Subsequently, the code defines a set of log macros (STC8H1KXX_INFO, STC8H1KXX_DEBUG, STC8H1KXX_ERROR) for outputting debug information with the "STC8H1KXX" label during system operation. These macros also control module functionality switches through multiple CONFIG_BSP_STC8H1KXX_* macros, such as whether to enable battery detection, GPIO control, or PWM control functions.

```
/*----------------------------------------Variable declaration----------------------
#define STC8H1KXX_TAG "STC8H1KXX"
#define STC8H1KXX_INFO(fmt, ...)        ESP_LOGI(STC8H1KXX_TAG, fmt, ##__VA_ARGS__)
#define STC8H1KXX_DEBUG(fmt, ...)       ESP_LOGD(STC8H1KXX_TAG, fmt, ##__VA_ARGS__)
#define STC8H1KXX_ERROR(fmt, ...)       ESP_LOGE(STC8H1KXX_TAG, fmt, ##__VA_ARGS__)

#define CONFIG_BSP_STC8H1KXX_ENABLED
#define CONFIG_BSP_STC8H1KXX_BATTERY_ENABLED
#define CONFIG_BSP_STC8H1KXX_GPIO_ENABLED
#define CONFIG_BSP_STC8H1KXX_PWM_ENABLED
```

In the basic type definition section, the code defines u8, u16, and u32 as uint8_t, uint16_t, and uint32_t respectively, in order to simplify the writing of subsequent data structures;

```
#ifdef CONFIG_BSP_STC8H1KXX_ENABLED
typedef uint8_t    u8;
typedef uint16_t   u16;
typedef uint32_t   u32;
```

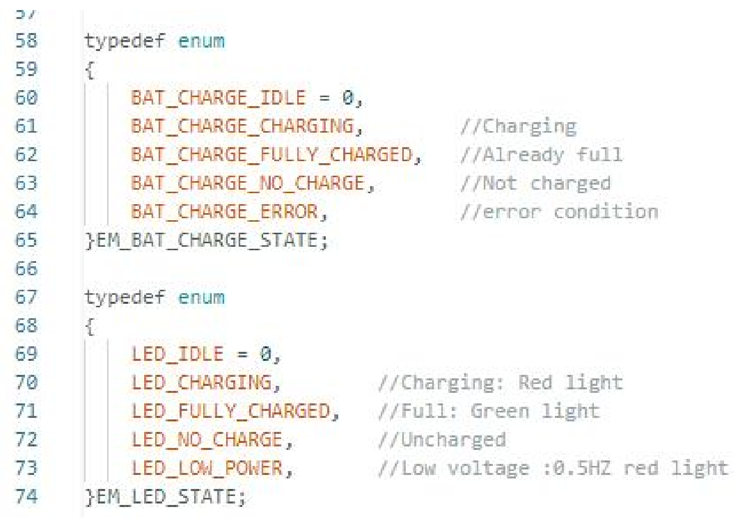
Then, in the battery function module, the I2C slave device address of the STC chip was defined as 0x2F. Multiple register addresses were defined through the EM_STC8_REG_ADDR enumeration for operations such as reading battery information, obtaining GPIO input, setting GPIO output, and setting PWM duty cycle.

```
typedef enum
{
    STC8_REG_ADDR_BATTERY    = 0x00, // Get battery information
    STC8_REG_ADDR_GET_GPIO   = 0x10, // Get GPIO input level
    STC8_REG_ADDR_SET_GPIO   = 0x18, // Set GPIO output level
    STC8_REG_ADDR_SET_PWM    = 0x20, // Set PWM duty cycle
}EM_STC8_REG_ADDR;
```

The code also defines the Battery_info_t structure, which is used to store the battery information read from the STC chip, including the ADC sampling voltage, the actual battery voltage, the percentage of charge, the battery charging status, and the LED indicator status.

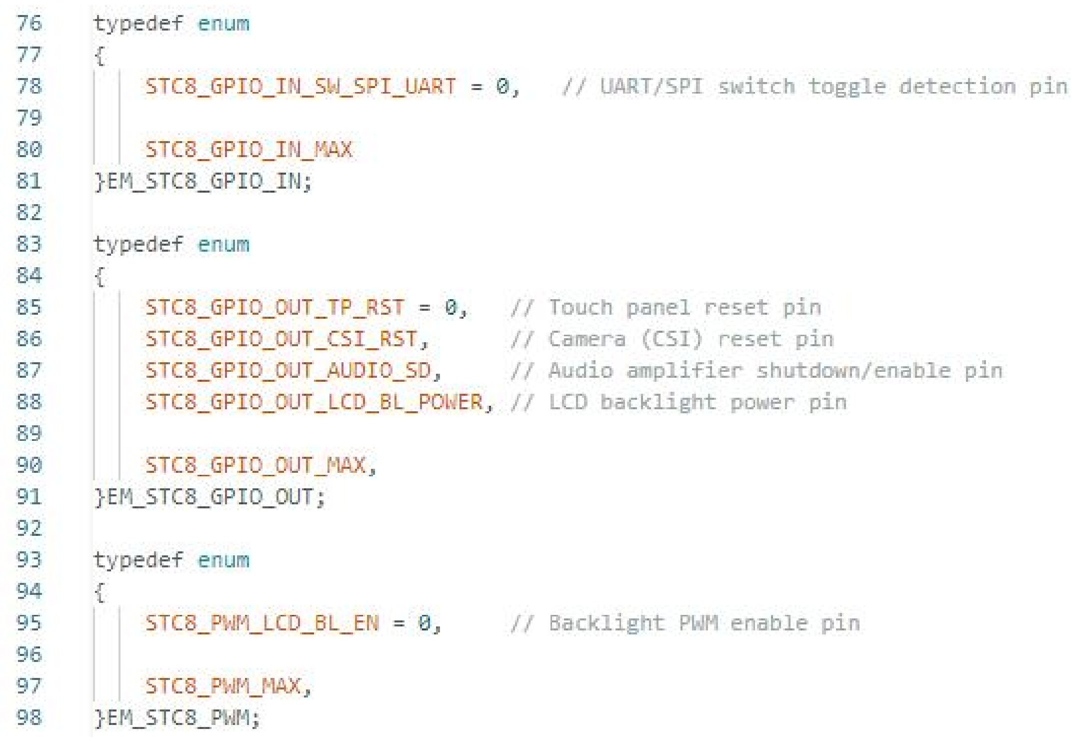
```
typedef enum
{
    BAT_CHARGE_IDLE = 0,
    BAT_CHARGE_CHARGING,        //Charging
    BAT_CHARGE_FULLY_CHARGED,   //Already full
    BAT_CHARGE_NO_CHARGE,       //Not charged
    BAT_CHARGE_ERROR,           //error condition
}EM_BAT_CHARGE_STATE;
```

At the same time, the battery charging status enumeration EM_BAT_CHARGE_STATE and the LED indicator status enumeration EM_LED_STATE were also defined to describe different charging or battery level states.



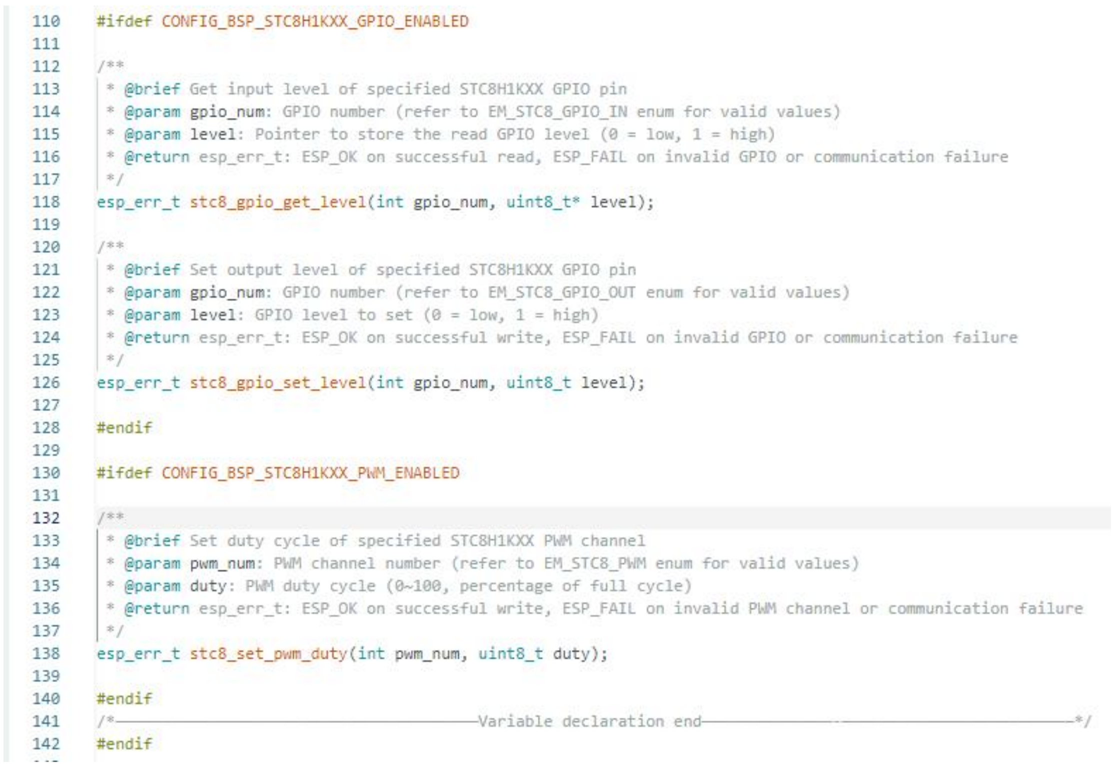
```
typedef enum
{
    BAT_CHARGE_IDLE = 0,
    BAT_CHARGE_CHARGING,          //Charging
    BAT_CHARGE_FULLY_CHARGED,     //Already full
    BAT_CHARGE_NO_CHARGE,         //Not charged
    BAT_CHARGE_ERROR,             //error condition
}EM_BAT_CHARGE_STATE;

typedef enum
{
    LED_IDLE = 0,
    LED_CHARGING,         //Charging: Red light
    LED_FULLY_CHARGED,    //Full: Green light
    LED_NO_CHARGE,        //Uncharged
    LED_LOW_POWER,        //Low voltage :0.5HZ red light
}EM_LED_STATE;
```

In addition, the code defines the available input and output GPIO functions of the STC chip through the EM_STC8_GPIO_IN and EM_STC8_GPIO_OUT enumerations. For example, it defines the detection pins for UART/SPI mode switching, the touch reset pin, the camera reset pin, the audio amplifier enable pin, and the LCD backlight power control pin, etc. Moreover, the EM_STC8_PWM enumeration is defined to identify the PWM control channels (such as the LCD backlight PWM).



```
typedef enum
{
    STC8_GPIO_IN_SW_SPI_UART = 0,    // UART/SPI switch toggle detection pin

    STC8_GPIO_IN_MAX
}EM_STC8_GPIO_IN;

typedef enum
{
    STC8_GPIO_OUT_TP_RST = 0,    // Touch panel reset pin
    STC8_GPIO_OUT_CSI_RST,       // Camera (CSI) reset pin
    STC8_GPIO_OUT_AUDIO_SD,      // Audio amplifier shutdown/enable pin
    STC8_GPIO_OUT_LCD_BL_POWER,  // LCD backlight power pin

    STC8_GPIO_OUT_MAX,
}EM_STC8_GPIO_OUT;

typedef enum
{
    STC8_PWM_LCD_BL_EN = 0,      // Backlight PWM enable pin

    STC8_PWM_MAX,
}EM_STC8_PWM;
```

The final file declares multiple external interface functions, including stc8_battery_info_get() which is used to read battery information via I2C, stc8_gpio_get_level() and stc8_gpio_set_level() which are used to read or set the GPIO level of the STC chip, and stc8_set_pwm_duty() which is used to set the duty cycle of the PWM output, thereby enabling the control of external devices (such as backlight, audio module, camera, etc.).

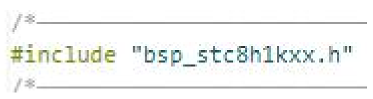


```
#ifdef CONFIG_BSP_STC8H1KXX_GPIO_ENABLED

/**
 * @brief Get input level of specified STC8H1KXX GPIO pin
 * @param gpio_num: GPIO number (refer to EM_STC8_GPIO_IN enum for valid values)
 * @param level: Pointer to store the read GPIO level (0 = low, 1 = high)
 * @return esp_err_t: ESP_OK on successful read, ESP_FAIL on invalid GPIO or communication failure
 */
esp_err_t stc8_gpio_get_level(int gpio_num, uint8_t* level);

/**
 * @brief Set output level of specified STC8H1KXX GPIO pin
 * @param gpio_num: GPIO number (refer to EM_STC8_GPIO_OUT enum for valid values)
 * @param level: GPIO level to set (0 = low, 1 = high)
 * @return esp_err_t: ESP_OK on successful write, ESP_FAIL on invalid GPIO or communication failure
 */
esp_err_t stc8_gpio_set_level(int gpio_num, uint8_t level);

#endif

#ifdef CONFIG_BSP_STC8H1KXX_PWM_ENABLED

/**
 * @brief Set duty cycle of specified STC8H1KXX PWM channel
 * @param pwm_num: PWM channel number (refer to EM_STC8_PWM enum for valid values)
 * @param duty: PWM duty cycle (0~100, percentage of full cycle)
 * @return esp_err_t: ESP_OK on successful write, ESP_FAIL on invalid PWM channel or communication failure
 */
esp_err_t stc8_set_pwm_duty(int pwm_num, uint8_t duty);

#endif
/*------------------------------Variable declaration end------------------------------*/
#endif
```

10. bsp_stc8h1kxx.cpp

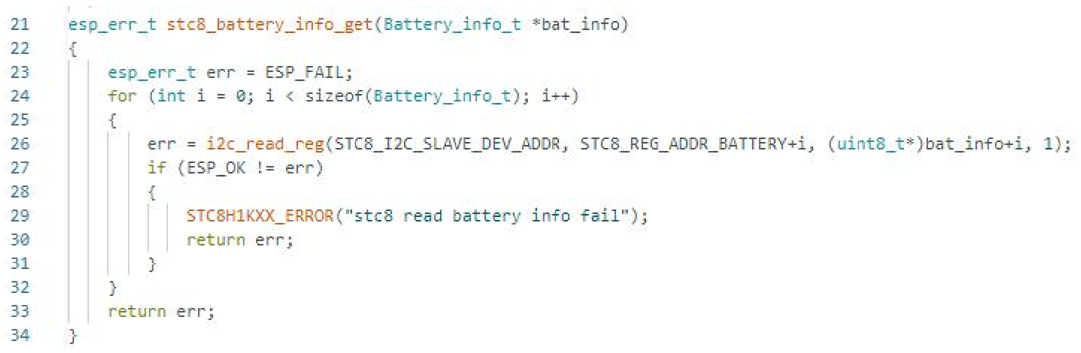
This code is the source file implementation part of the STC8H1KXX driver module. Its main function is to communicate with the STC8H1KXX co-processing chip through the I2C bus on the main control ESP32-P4, and to realize functions such as battery information reading, GPIO control, and PWM output control.

The file begins by importing the previously defined interfaces and data structures through the statement "#include "bsp_stc8h1kxx.h" ", and compiles the related functional modules under the condition of the macro "CONFIG_BSP_STC8H1KXX_ENABLED".



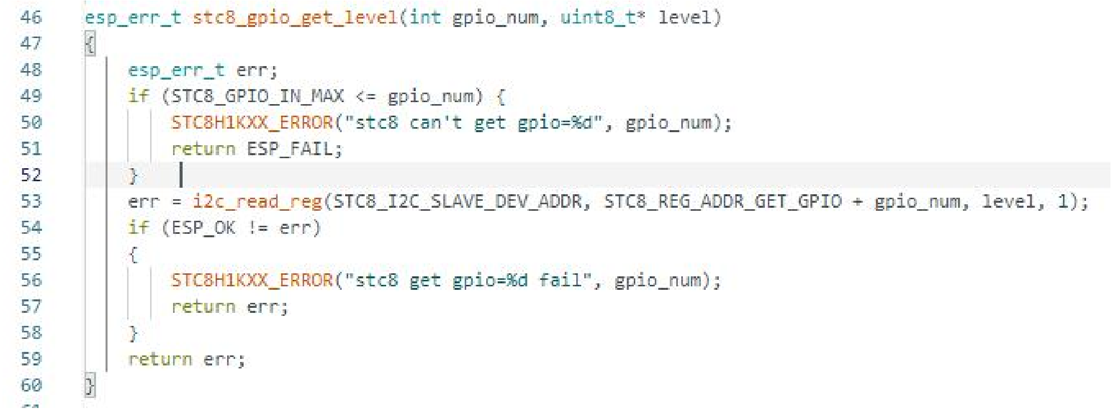
```
/*------------------------
#include "bsp_stc8h1kxx.h"
/*------------------------
```

Firstly, the stc8_battery_info_get() function was implemented in the battery function section. This function reads the battery data in the I2C registers of the STC chip byte by byte in a loop: it uses i2c_read_reg() to continuously read data starting from the STC8_REG_ADDR_BATTERY register of the STC device address STC8_I2C_SLAVE_DEV_ADDR, and writes the data sequentially into the Battery_info_t structure, thereby obtaining information such as the battery ADC voltage, battery voltage, percentage of charge, and charging status. If the reading fails, it outputs the error message through the log macro and returns the error code.



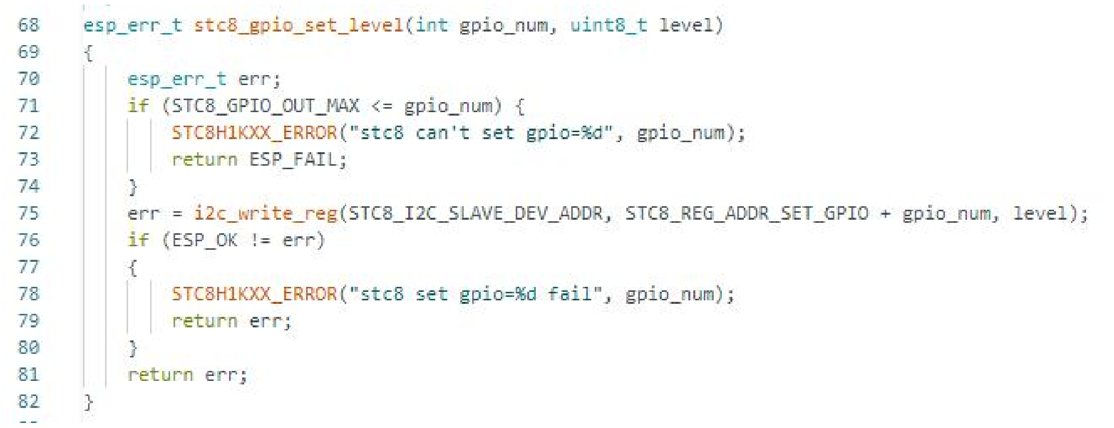
```
esp_err_t stc8_battery_info_get(Battery_info_t *bat_info)
{
    esp_err_t err = ESP_FAIL;
    for (int i = 0; i < sizeof(Battery_info_t); i++)
    {
        err = i2c_read_reg(STC8_I2C_SLAVE_DEV_ADDR, STC8_REG_ADDR_BATTERY+i, (uint8_t*)bat_info+i, 1);
        if (ESP_OK != err)
        {
            STC8H1KXX_ERROR("stc8 read battery info fail");
            return err;
        }
    }
    return err;
}
```

Then, in the GPIO control section, two functions, stc8_gpio_get_level() and stc8_gpio_set_level(), were implemented. Among them, stc8_gpio_get_level() is used to read the level status of the specified GPIO input pin of the STC chip. It first checks whether the GPIO number is valid, and then reads 1 byte of data from the corresponding register through i2c_read_reg() and returns the level value.



```
esp_err_t stc8_gpio_get_level(int gpio_num, uint8_t* level)
{
    esp_err_t err;
    if (STC8_GPIO_IN_MAX <= gpio_num) {
        STC8H1KXX_ERROR("stc8 can't get gpio=%d", gpio_num);
        return ESP_FAIL;
    }
    err = i2c_read_reg(STC8_I2C_SLAVE_DEV_ADDR, STC8_REG_ADDR_GET_GPIO + gpio_num, level, 1);
    if (ESP_OK != err)
    {
        STC8H1KXX_ERROR("stc8 get gpio=%d fail", gpio_num);
        return err;
    }
    return err;
}
```

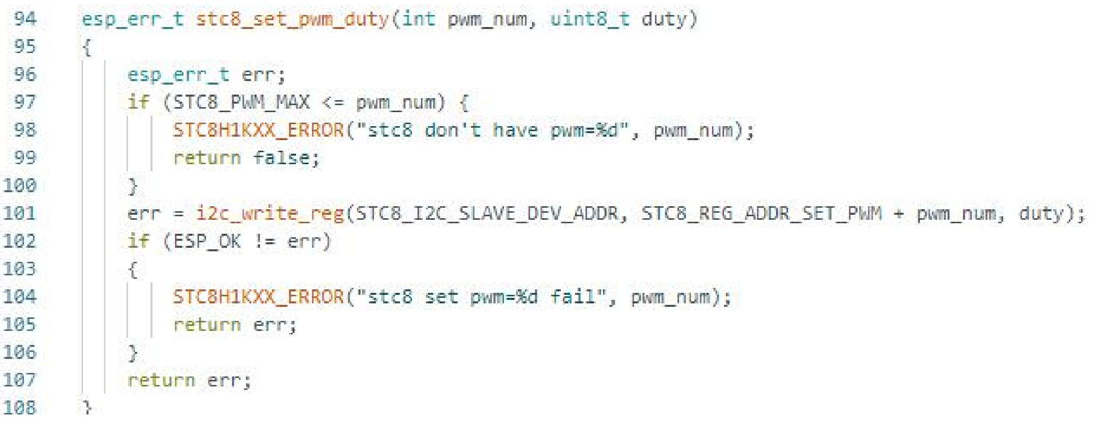
The function stc8_gpio_set_level() is used to control the output state of the GPIO pins of the STC chip. For instance, it can control touch reset, camera reset, audio amplifier enablement, or LCD backlight power supply, etc. It first checks whether the GPIO number is within the valid range, and then calls i2c_write_reg() to write a high or low level into the corresponding register, thereby changing the state of the peripheral pins.



```
esp_err_t stc8_gpio_set_level(int gpio_num, uint8_t level)
{
    esp_err_t err;
    if (STC8_GPIO_OUT_MAX <= gpio_num) {
        STC8H1KXX_ERROR("stc8 can't set gpio=%d", gpio_num);
        return ESP_FAIL;
    }
    err = i2c_write_reg(STC8_I2C_SLAVE_DEV_ADDR, STC8_REG_ADDR_SET_GPIO + gpio_num, level);
    if (ESP_OK != err)
    {
        STC8H1KXX_ERROR("stc8 set gpio=%d fail", gpio_num);
        return err;
    }
    return err;
}
```

Finally, in the PWM function module, the stc8_set_pwm_duty() function was implemented. This function is used to set the duty cycle of the PWM channel of the STC chip. By writing the duty cycle value ranging from 0 to 100 to the

STC8_REG_ADDR_SET_PWM register via I2C, it realizes the brightness adjustment for external devices (such as LCD backlight); the function also checks whether the PWM channel number is valid first, and outputs an error log in case of communication failure.

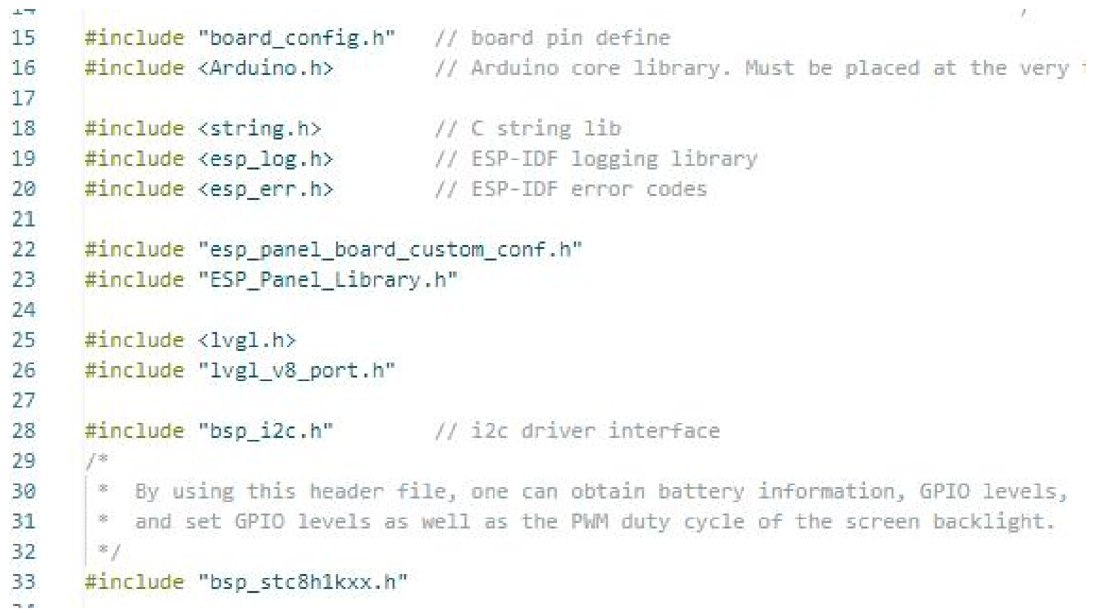


```
esp_err_t stc8_set_pwm_duty(int pwm_num, uint8_t duty)
{
    esp_err_t err;
    if (STC8_PWM_MAX <= pwm_num) {
        STC8H1KXX_ERROR("stc8 don't have pwm=%d", pwm_num);
        return false;
    }
    err = i2c_write_reg(STC8_I2C_SLAVE_DEV_ADDR, STC8_REG_ADDR_SET_PWM + pwm_num, duty);
    if (ESP_OK != err)
    {
        STC8H1KXX_ERROR("stc8 set pwm=%d fail", pwm_num);
        return err;
    }
    return err;
}
```

OK, now let's take a look at the code content in the main code file named "ino".

1. Header file

This code is mainly used to introduce various library files required for the program's operation, providing support for subsequent hardware drivers, system functions, and graphical interface development.



```
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

#include <lvgl.h>
#include "lvgl_v8_port.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"
```

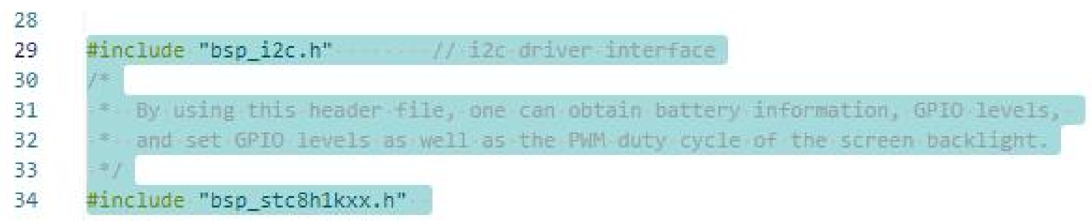
Firstly, "#include "board_config.h" is used to load the hardware configuration file of the development board, which usually contains definitions of GPIO pins, display interface configurations, and other board-level information;

#include <Arduino.h> is the core library of Arduino, which provides basic functions of Arduino, such as setup(), loop(), Serial serial communication, and delay() functions. Therefore, it must be placed at a relatively earlier position to ensure that these APIs can be correctly recognized.

Next, #include <string.h> imports the standard C language string processing library, which is used for string copying, comparison, and other operations; #include <esp_log.h> and #include <esp_err.h> are system libraries provided by ESP-IDF, respectively used for log output and error code management, which can help developers perform system debugging and error handling;

Subsequently, esp_panel_drivers_conf.h, esp_panel_board_custom_conf.h and ESP_Panel_Library.h are introduced. These files belong to the ESP32_Display_Panel driver library and are used for configuring and driving the display and touch screen hardware (such as the EK79007 display driver and the GT911 touch chip).

#include <lvgl.h> and #include "lvgl_v8_port.h" introduce the LVGL graphical interface library and its hardware adaptation layer. Among them, lvgl.h provides all LVGL UI controls and graphic functions, while lvgl_v8_port.h is responsible for connecting LVGL with the underlying display driver and touch driver, enabling developers to create graphical interfaces on the screen and respond to touch inputs.


```

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"
```

The functions of these two lines of code are to import the driver interface files related to I2C communication and STC8H1KXX peripheral control, so as to call the corresponding hardware control functions in the program.

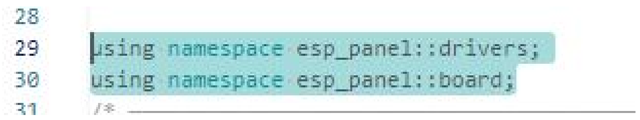
The first line, #include "bsp_i2c.h", is used to include the I2C driver interface header file. This file provides basic communication functions such as I2C bus initialization and register reading/writing (for example, i2c_read_reg() and i2c_write_reg()), enabling the main control ESP32-P4 to interact with external devices through the I2C bus;

The second line, #include "bsp_stc8h1kxx.h", introduces the BSP driver interface of the STC8H1KXX coprocessing chip. This header file encapsulates a series of functions based on the I2C driver at the lower level, such as reading battery information stc8_battery_info_get(), getting or setting GPIO levels stc8_gpio_get_level() / stc8_gpio_set_level(), and setting PWM duty ratio stc8_set_pwm_duty() (usually used to control LCD backlight brightness).

Therefore, the overall function of these two lines of code is to introduce the I2C communication capability and the peripheral control interface of STC8H1KXX into the current program, enabling the main controller to interact with the coprocessor MCU through I2C, thereby achieving functions such as battery status reading, GPIO control, and screen backlight PWM adjustment.

2. Namespace

These two lines of code are used to import the contents of the specified namespace, allowing the program to omit the namespace prefix when using the classes, functions, or variables in these libraries, thereby simplifying the code writing.

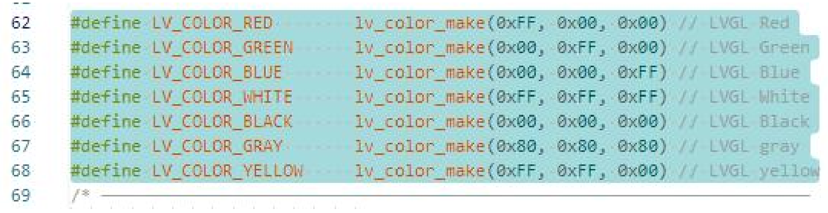


```
using namespace esp_panel::drivers;
using namespace esp_panel::board;
/*
```

using namespace esp_panel::drivers; It indicates that all the members in the drivers namespace of the esp_panel library (such as LCD drivers, touch drivers, and related classes and interfaces) are directly imported into the current file scope. Thus, when using these driver classes, there is no need to write the complete class name of esp_panel::drivers:: anymore.
And "using namespace esp_panel::board;" is to introduce the contents from the "board" namespace. This namespace usually contains encapsulated classes related to the development board, such as the "Board" class, which is responsible for uniformly managing hardware resources like displays, touch screens, and bus interfaces. Therefore, in the code, you can directly write "Board *board = new Board();", instead of writing "esp_panel::board::Board".

3. Macro Definitions

This code uses the #define macro definition to create a set of shortcut names for common colors in the LVGL graphical interface library, making it convenient to directly use these colors when writing interface code.

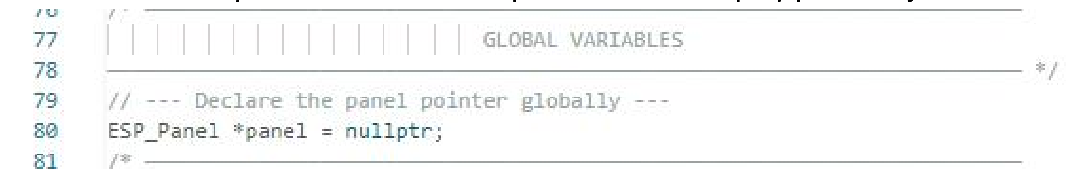


```
#define LV_COLOR_RED        lv_color_make(0xFF, 0x00, 0x00) // LVGL Red
#define LV_COLOR_GREEN      lv_color_make(0x00, 0xFF, 0x00) // LVGL Green
#define LV_COLOR_BLUE       lv_color_make(0x00, 0x00, 0xFF) // LVGL Blue
#define LV_COLOR_WHITE      lv_color_make(0xFF, 0xFF, 0xFF) // LVGL White
#define LV_COLOR_BLACK      lv_color_make(0x00, 0x00, 0x00) // LVGL Black
#define LV_COLOR_GRAY       lv_color_make(0x80, 0x80, 0x80) // LVGL gray
#define LV_COLOR_YELLOW     lv_color_make(0xFF, 0xFF, 0x00) // LVGL yellow
/*
```

Each macro calls the function lv_color_make(R, G, B), which is a color constructor provided by LVGL. It is used to generate an LVGL-recognized color type lv_color_t based on the RGB primary color values (red, green, blue).
For example, LV_COLOR_RED is defined as lv_color_make(0xFF, 0x00, 0x00), which means the red component is 255, the green component is 0, and the blue component is 0, thereby generating pure red.
Similarly, LV_COLOR_GREEN, LV_COLOR_BLUE, LV_COLOR_WHITE, LV_COLOR_BLACK, LV_COLOR_GRAY, and LV_COLOR_YELLOW respectively define green, blue, white, black, gray, and yellow.
The purpose of these macros is to enhance code readability and simplify UI development. When developers set the style or background color of LVGL controls, they only need to write LV_COLOR_RED or LV_COLOR_WHITE instead of repeatedly writing the RGB parameters of lv_color_make(…), thus making the interface code more clear and readable.

4. Global Variables

This code is mainly used to declare the pointer of the display panel object.

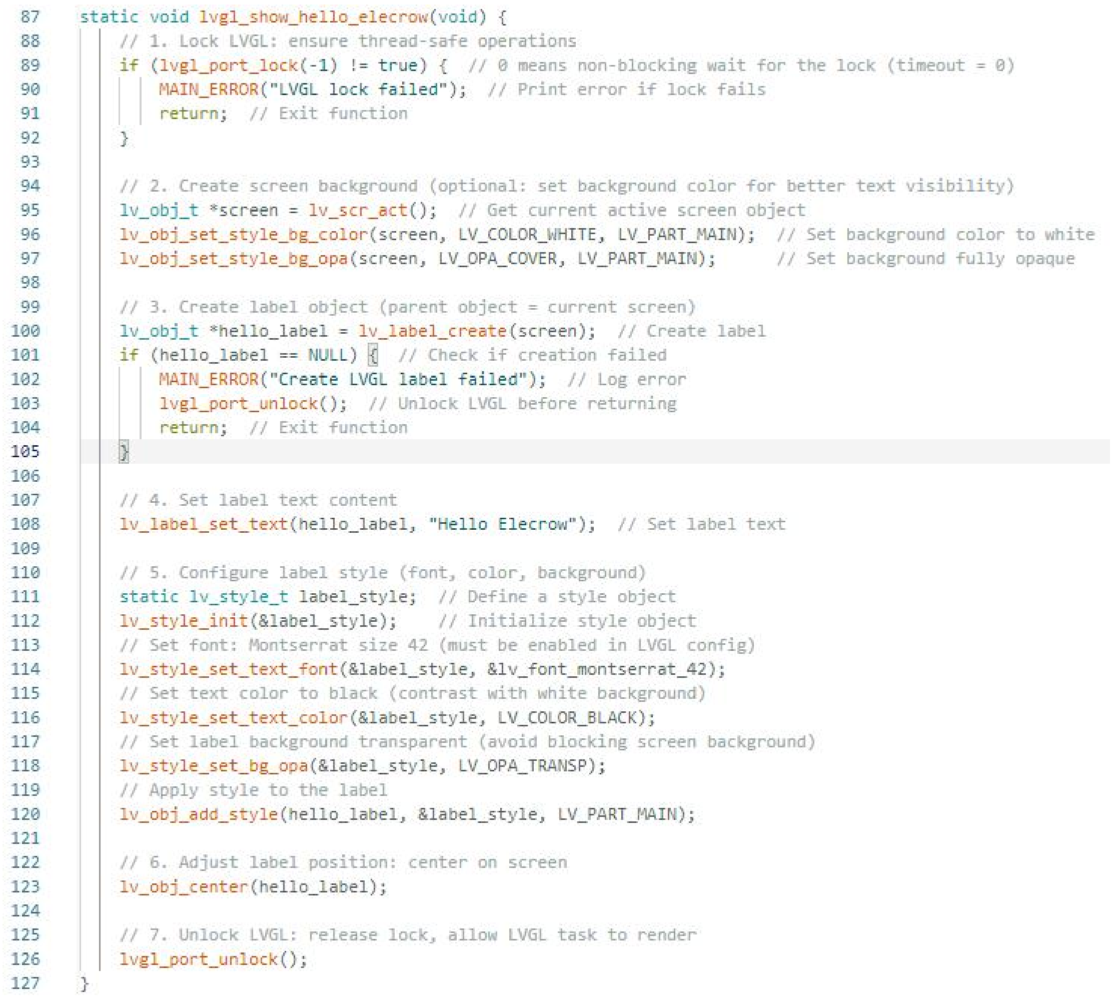


```
/* -------------------------------------------------------------------
| | | | | | | | | | | | | | |  GLOBAL VARIABLES
------------------------------------------------------------------- */
// --- Declare the panel pointer globally ---
ESP_Panel *panel = nullptr;
/* -------------------------------------------------------------------
```

ESP_Panel *panel = nullptr; A pointer variable named "panel" of the ESP_Panel class is declared, and it is initialized as a null pointer "nullptr". This object is mainly used to manage the display and touch screen hardware (such as initializing the display interface, reading touch data, etc.). Since it is defined as a global variable, it can be shared and accessed by various functions in the program, such as setup(), loop(), and other functions. Usually, an actual object of this panel is created during the program initialization stage and the control of the display and touch devices is completed.

5. LVGL text display

This code defines a function "lvgl_show_hello_elecrow()", whose purpose is to use the LVGL graphics library to create and display a "Hello Elecrow" text interface in the center of the screen.



```
static void lvgl_show_hello_elecrow(void) {
    // 1. Lock LVGL: ensure thread-safe operations
    if (lvgl_port_lock(-1) != true) {  // 0 means non-blocking wait for the lock (timeout = 0)
        MAIN_ERROR("LVGL lock failed");  // Print error if lock fails
        return;  // Exit function
    }

    // 2. Create screen background (optional: set background color for better text visibility)
    lv_obj_t *screen = lv_scr_act();  // Get current active screen object
    lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);  // Set background color to white
    lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);      // Set background fully opaque

    // 3. Create label object (parent object = current screen)
    lv_obj_t *hello_label = lv_label_create(screen);  // Create label
    if (hello_label == NULL) {  // Check if creation failed
        MAIN_ERROR("Create LVGL label failed");  // Log error
        lvgl_port_unlock();  // Unlock LVGL before returning
        return;  // Exit function
    }

    // 4. Set label text content
    lv_label_set_text(hello_label, "Hello Elecrow");  // Set label text

    // 5. Configure label style (font, color, background)
    static lv_style_t label_style;  // Define a style object
    lv_style_init(&label_style);    // Initialize style object
    // Set font: Montserrat size 42 (must be enabled in LVGL config)
    lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
    // Set text color to black (contrast with white background)
    lv_style_set_text_color(&label_style, LV_COLOR_BLACK);
    // Set label background transparent (avoid blocking screen background)
    lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
    // Apply style to the label
    lv_obj_add_style(hello_label, &label_style, LV_PART_MAIN);

    // 6. Adjust label position: center on screen
    lv_obj_center(hello_label);

    // 7. Unlock LVGL: release lock, allow LVGL task to render
    lvgl_port_unlock();
}
```

The function first locks LVGL via lvgl_port_lock(-1) to ensure thread-safe operation on UI objects in multi-task or multi-thread environments. If acquiring the lock fails, it outputs an error message and returns directly.

```c
static void lvgl_show_hello_elecrow(void) {
    // 1. Lock LVGL: ensure thread-safe operations
    if (lvgl_port_lock(-1) != true) {  // 0 means non-blocking wait for the lock (timeout = 0)
        MAIN_ERROR("LVGL lock failed");  // Print error if lock fails
        return;  // Exit function
    }
```

Then it calls "lv_scr_act()" to get the current active screen object, and uses "lv_obj_set_style_bg_color()" and "lv_obj_set_style_bg_opa()" to set the screen background to a fully opaque white background, improving text readability.

```c
    // 2. Create screen background (optional: set background color for better text visibility)
    lv_obj_t *screen = lv_scr_act();  // Get current active screen object
    lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);  // Set background color to white
    lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);      // Set background fully opaque
```

Then the program creates a label text widget on the current screen via "lv_label_create(screen)". If creation fails, it prints an error log and releases the LVGL lock.

```c
    // 3. Create label object (parent object = current screen)
    lv_obj_t *hello_label = lv_label_create(screen);  // Create label
    if (hello_label == NULL) {  // Check if creation failed
        MAIN_ERROR("Create LVGL label failed");  // Log error
        lvgl_port_unlock();  // Unlock LVGL before returning
        return;  // Exit function
    }
```

After the label is successfully created, "lv_label_set_text()" is used to set its text content to "Hello Elecrow".

```c
    // 4. Set label text content
    lv_label_set_text(hello_label, "Hello Elecrow");  // Set label text
```

To beautify the display effect, the program then creates a lv_style_t style object and initializes it. It then sets the Montserrat 42pt font, black text color, and transparent background for the label, and applies this style to the label object via lv_obj_add_style().

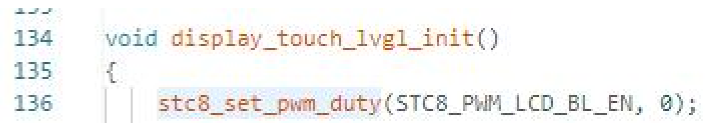
```c
    // 5. Configure label style (font, color, background)
    static lv_style_t label_style;  // Define a style object
    lv_style_init(&label_style);    // Initialize style object
    // Set font: Montserrat size 42 (must be enabled in LVGL config)
    lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
    // Set text color to black (contrast with white background)
    lv_style_set_text_color(&label_style, LV_COLOR_BLACK);
    // Set label background transparent (avoid blocking screen background)
    lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
    // Apply style to the label
    lv_obj_add_style(hello_label, &label_style, LV_PART_MAIN);
```

Finally, it calls "lv_obj_center()" to center the text label on the screen. After completing the interface layout, it uses "lvgl_port_unlock()" to release the LVGL lock, allowing LVGL's background task to refresh and render the interface.

```c
    // 6. Adjust label position: center on screen
    lv_obj_center(hello_label);

    // 7. Unlock LVGL: release lock, allow LVGL task to render
    lvgl_port_unlock();
}
```

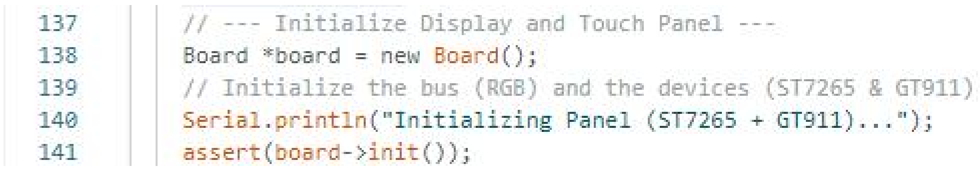
6. display_touch_lvgl_init

The main function of this display_touch_lvgl_init() function is to initialize the screen backlight, RGB display screen, touch screen, and the LVGL graphical interface system, thereby enabling the device to have complete graphic display and touch input capabilities.
The function begins by calling stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0) to set the LCD backlight PWM duty cycle to 0%, which means turning off the screen backlight first. This step is usually used to prevent screen flickering or abnormal brightness during the display initialization process. The PWM control is handled by the external coprocessor STC8H1KXX.



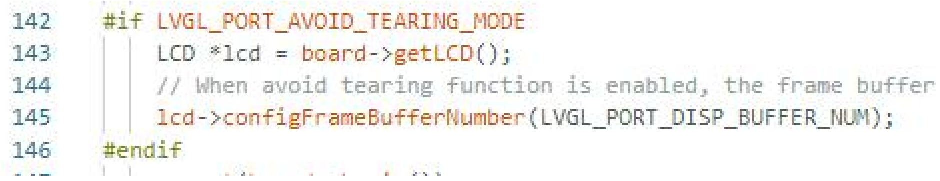
```
void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);
```

Then the program creates a Board object named "board" using the "ESP32_Display_Panel" library, which is used to uniformly manage the display bus, LCD controller, and touch device. Subsequently, it prints "Initializing Panel (ST7265 + GT911)..." indicating that the system is initializing the display and touch hardware. The LCD driver chip is ST7265 and the touch controller is GT911.
The program initializes the display bus (RGB interface) and related devices through "board->init()". If the initialization fails, the assert() function will trigger the program to stop.



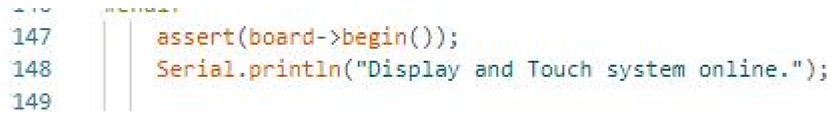
```
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
```

Next, under the conditional compilation #if LVGL_PORT_AVOID_TEARING_MODE, if the tearing avoidance mode is enabled, the program will obtain the LCD object board->getLCD() and call configFrameBufferNumber() to set the number of frame buffers, in order to reduce the problem of image tearing that occurs during screen refresh.



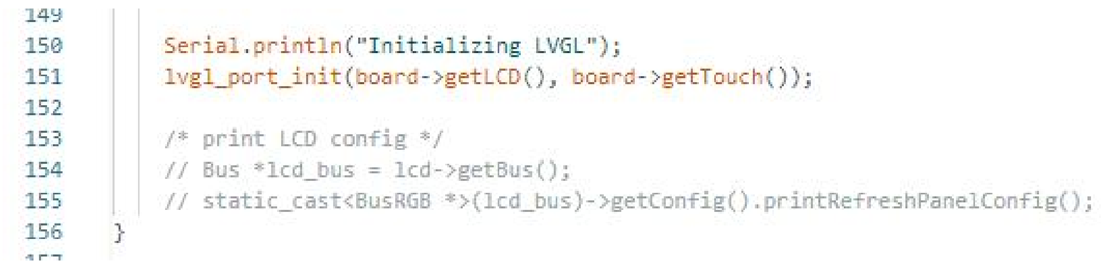
```
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
```

Then, call board->begin() to start the display panel. This process usually involves LCD startup timing, driver configuration, and display interface startup. Once it is successful, print "Display and Touch system online." to indicate that the display and touch system is running normally.



```
    assert(board->begin());
    Serial.println("Display and Touch system online.");

```

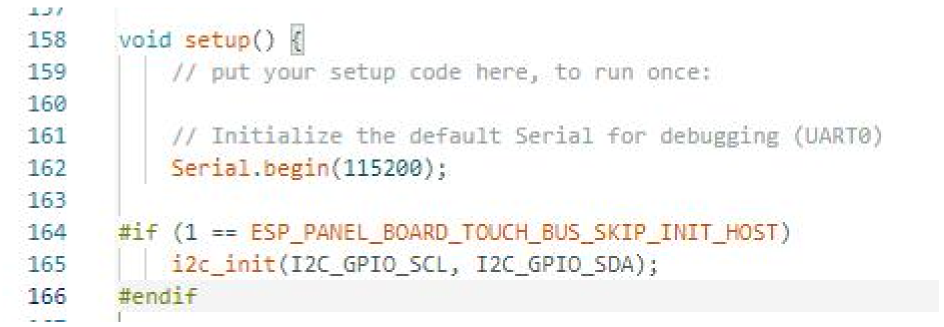
Finally, the program initializes the LVGL graphics library. By calling lvgl_port_init(board->getLCD(), board->getTouch()), the LCD display device and touch input device are registered in LVGL, enabling LVGL to handle screen drawing and touch event processing. The function also retains a section of commented-out debugging code at the end, which is used to obtain the RGB bus configuration and print the LCD refresh parameters.



```
    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());

    /* print LCD config */
    // Bus *lcd_bus = lcd->getBus();
    // static_cast<BusRGB *>(lcd_bus)->getConfig().printRefreshPanelConfig();
}
```

7. setup

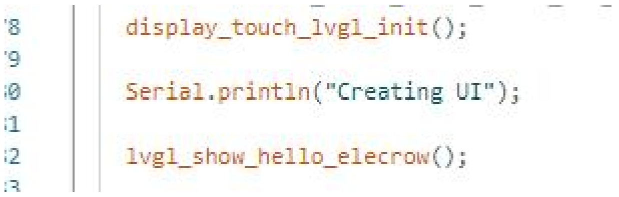
This setup() function is the system initialization entry point of the program. It is executed only once after the device is powered on or reset. Its main function is to initialize serial port debugging, I²C communication, display and touch system, as well as LVGL graphical interface, and finally turn on the screen backlight display to show the content.
The function first initiates serial communication (UART0) through Serial.begin(115200), setting the baud rate to 115200, which is used to output debugging information;
Then, through conditional compilation #if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST), it determines whether manual initialization of the I²C bus is required. If the macro ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST is set to 1, it calls i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) to initialize the I²C interface using the specified SCL and SDA pins. In this way, the main control ESP32-P4 can communicate with external devices.



```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif
```

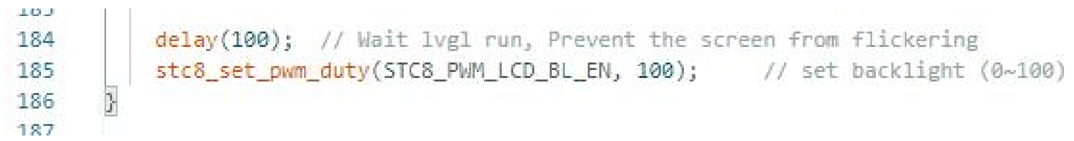
The comments in the code state that by default, ESP32_Display_Panel will automatically initialize the I²C bus used by the touch device. Therefore, manual initialization is usually not necessary. It is only required to enable this macro when using other drivers (such as Wire) or when it is necessary to manage I²C yourself.
Then, the program calls the display_touch_lvgl_init() function to complete the initialization of the display screen, touch screen, and LVGL graphics system, including

the configuration of the RGB screen driver, touch controller, and LVGL display interface;
After initialization, by using Serial.println("Creating UI") to prompt the system to start creating the user interface, then calls lvgl_show_hello_elecrow() to create and display an LVGL label object, and displays the text "Hello Elecrow" on the screen.



```
display_touch_lvgl_init();

Serial.println("Creating UI");

lvgl_show_hello_elecrow();
```

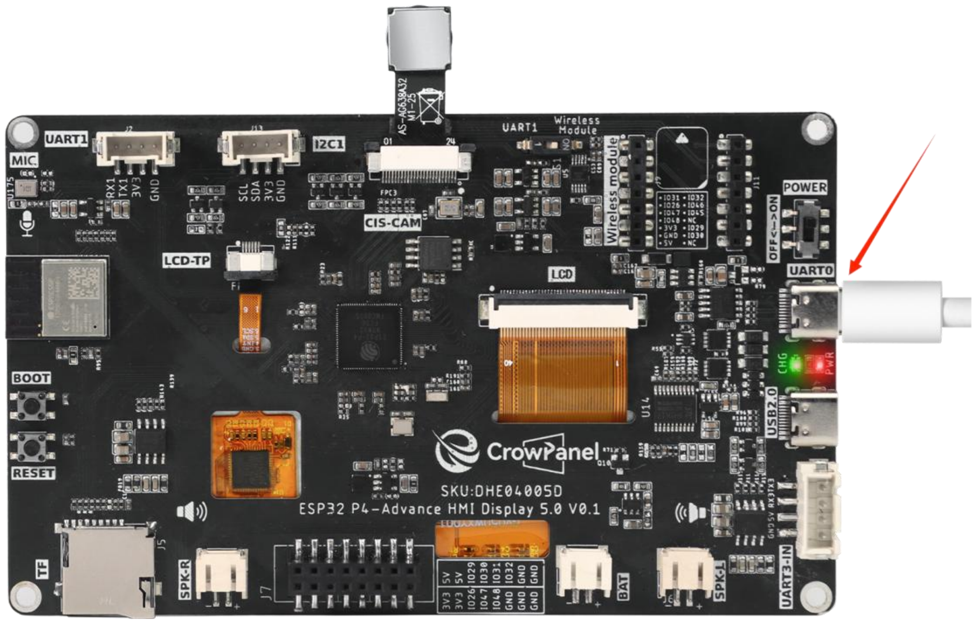
Then, the program executes delay(100) to wait for approximately 100ms, allowing LVGL sufficient time to complete the interface rendering. This is to prevent screen flickering caused by immediately turning on the backlight after the screen initialization is completed. Finally, it calls stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100) to set the LCD backlight PWM duty cycle to 100%. By controlling the backlight brightness through STC8H1KXX, the screen is lit up and the already drawn interface is displayed completely.



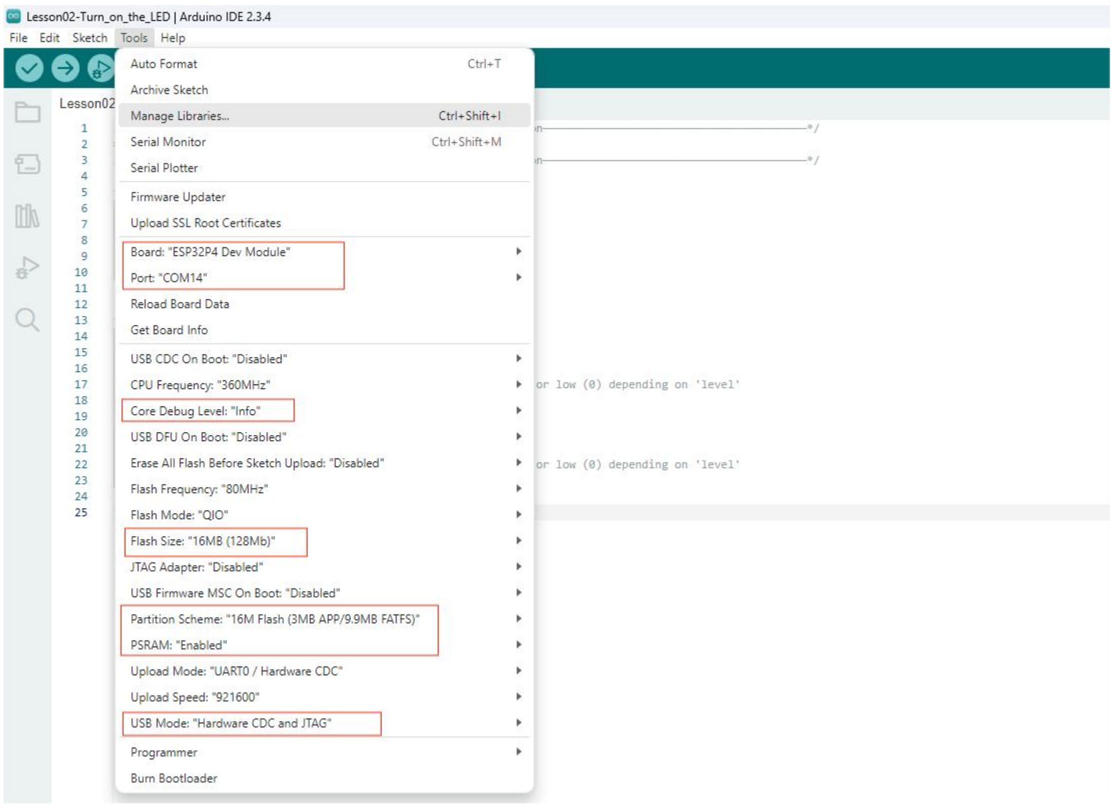
```
    delay(100);  // Wait lvgl run, Prevent the screen from flickering
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);     // set backlight (0~100)
}
```

Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.

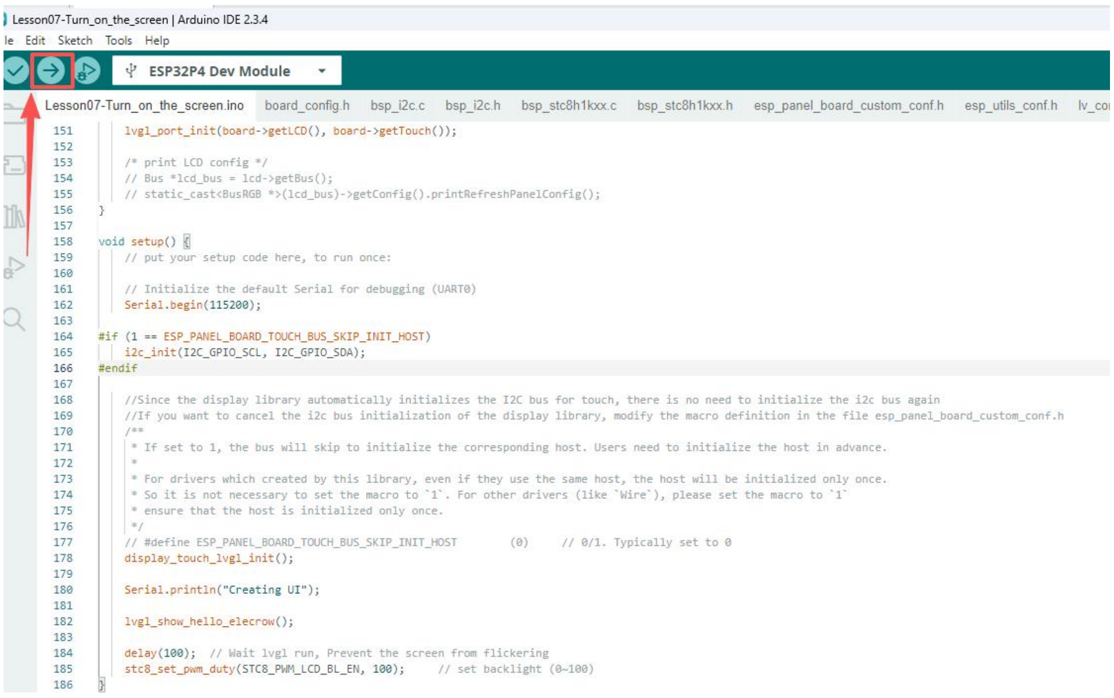


Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

```
Auto Format                                          Ctrl+T
Archive Sketch
Manage Libraries...                                  Ctrl+Shift+I
Serial Monitor                                       Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader
```

Then we compile and upload the code.


Lesson07-Turn_on_the_screen | Arduino IDE 2.3.4

```
    lvgl_port_init(board->getLCD(), board->getTouch());

    /* print LCD config */
    // Bus *lcd_bus = lcd->getBus();
    // static_cast<BusRGB *>(lcd_bus)->getConfig().printRefreshPanelConfig();
}

void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif

    //Since the display library automatically initializes the I2C bus for touch, there is no need to initialize the i2c bus again
    //If you want to cancel the i2c bus initialization of the display library, modify the macro definition in the file esp_panel_board_custom_conf.h
    /**
     * If set to 1, the bus will skip to initialize the corresponding host. Users need to initialize the host in advance.
     *
     * For drivers which created by this library, even if they use the same host, the host will be initialized only once.
     * So it is not necessary to set the macro to `1`. For other drivers (like `Wire`), please set the macro to `1`
     * ensure that the host is initialized only once.
     */
    // #define ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST        (0)     // 0/1. Typically set to 0
    display_touch_lvgl_init();

    Serial.println("Creating UI");

    lvgl_show_hello_elecrow();

    delay(100);  // Wait lvgl run, Prevent the screen from flickering
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);     // set backlight (0~100)
}
```

After the burning process is completed. You will be able to see that your Advance-P4 screen lights up, and the message "Hello Elecrow" appears in the center of the screen.

Hello Elecrow

Lesson08---SD Card File Reading

Introduction

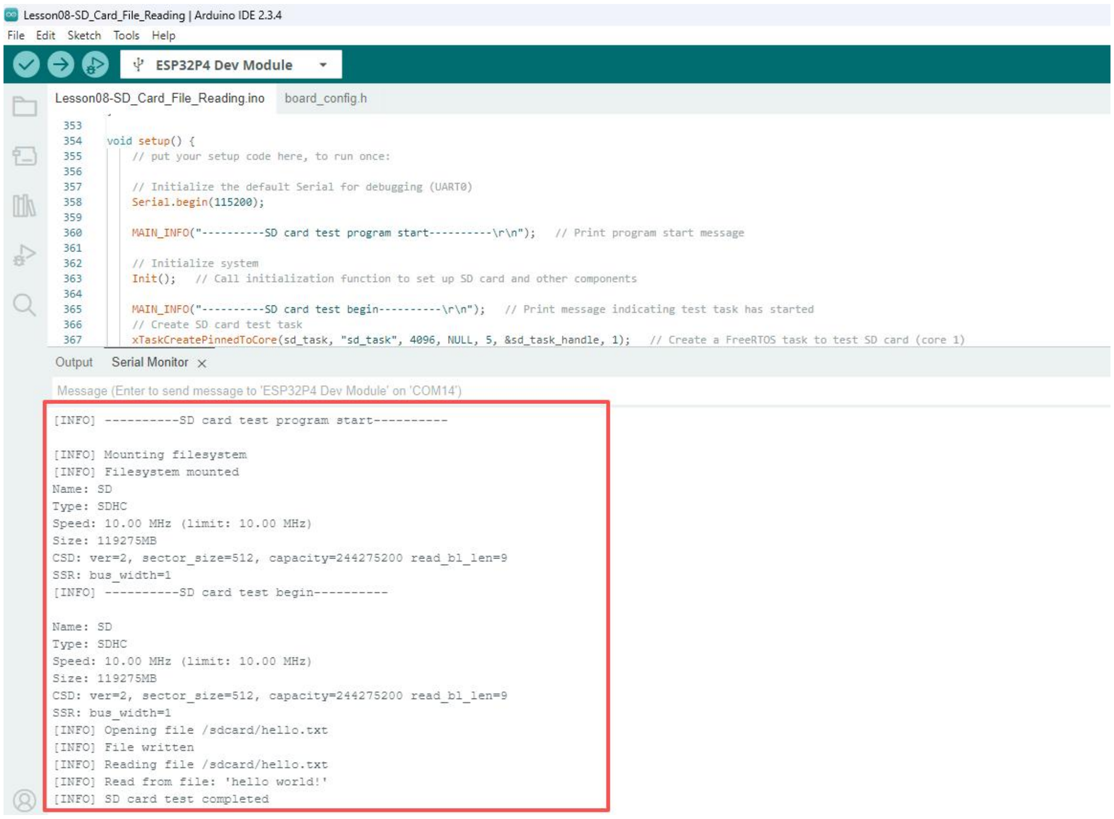
In this lesson, we will start teaching you how to use the SD card on the Advance-P4 development board to perform read and write operations on files stored in the SD card.

Learning Goals

1. Understand the SD_MMC/SPI hardware interface principles and pin configuration methods for the SD card on the ESP32-P4 development board.
2. Master the mounting and initialization process of the SD card FAT file system.
3. Master basic file operations for SD card text/binary files, including reading, writing, positioning, and copying.

Preview of the Result

After running the code, you will be able to visually see that a file named "hello.txt" appears in the SD card, with the content "hello world!" already written in it.

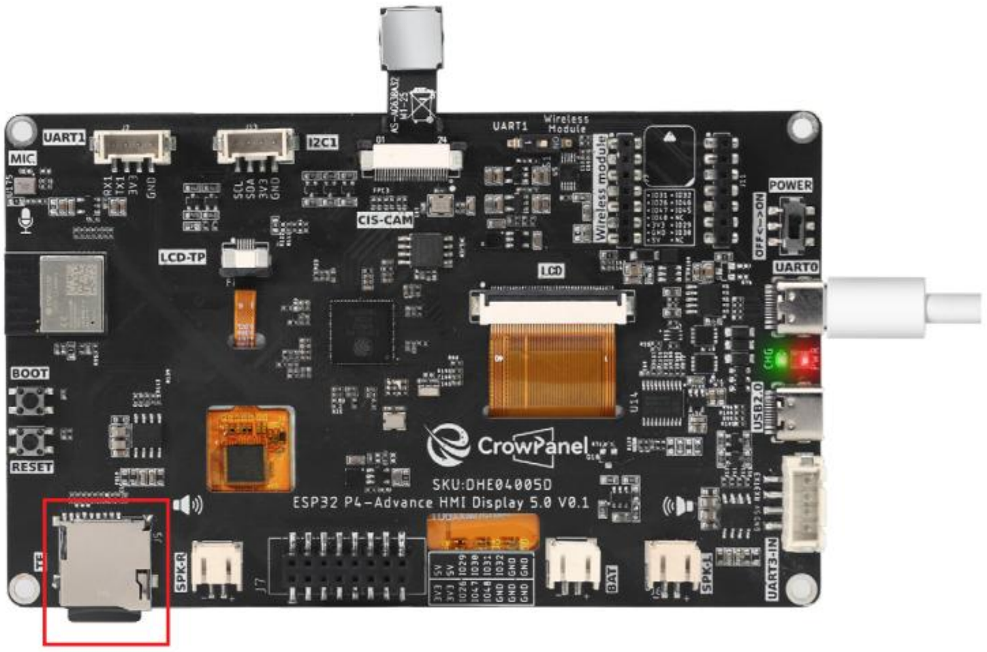

Lesson08-SD_Card_File_Reading | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson08-SD_Card_File_Reading.ino
board_config.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
void setup() {
  // put your setup code here, to run once:

  // Initialize the default Serial for debugging (UART0)
  Serial.begin(115200);

  MAIN_INFO("----------SD card test program start----------\r\n");   // Print program start message

  // Initialize system
  Init();   // Call initialization function to set up SD card and other components

  MAIN_INFO("----------SD card test begin----------\r\n");   // Print message indicating test task has started
  // Create SD card test task
  xTaskCreatePinnedToCore(sd_task, "sd_task", 4096, NULL, 5, &sd_task_handle, 1);   // Create a FreeRTOS task to test SD card (core 1)
```

```
[INFO] ----------SD card test program start----------

[INFO] Mounting filesystem
[INFO] Filesystem mounted
Name: SD
Type: SDHC
Speed: 10.00 MHz (limit: 10.00 MHz)
Size: 119275MB
CSD: ver=2, sector_size=512, capacity=244275200 read_bl_len=9
SSR: bus_width=1
[INFO] ----------SD card test begin----------

Name: SD
Type: SDHC
Speed: 10.00 MHz (limit: 10.00 MHz)
Size: 119275MB
CSD: ver=2, sector_size=512, capacity=244275200 read_bl_len=9
SSR: bus_width=1
[INFO] Opening file /sdcard/hello.txt
[INFO] File written
[INFO] Reading file /sdcard/hello.txt
[INFO] Read from file: 'hello world!'
[INFO] SD card test completed
```

Hardware Used in This Lesson

SD card on the Advance-P4



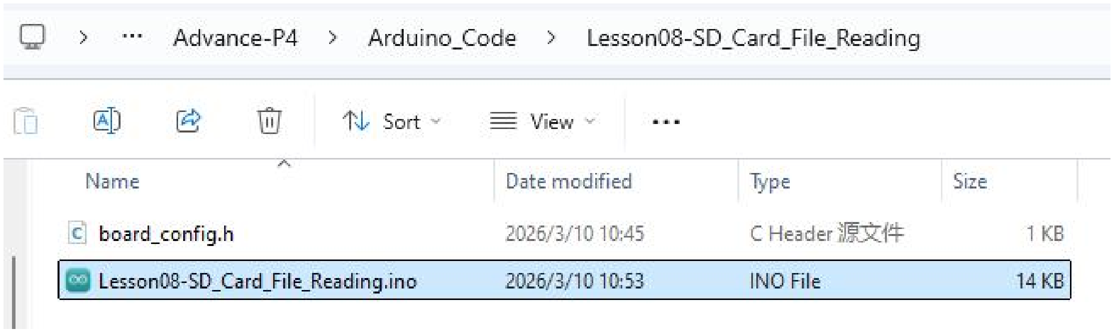
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson08-SD_Card_File_Reading

Key Explanations

The focus of this lesson is how to use the "SD card", how to initialize it, and how to read and write files.

Double-click to open this lesson's code (the .ino file).

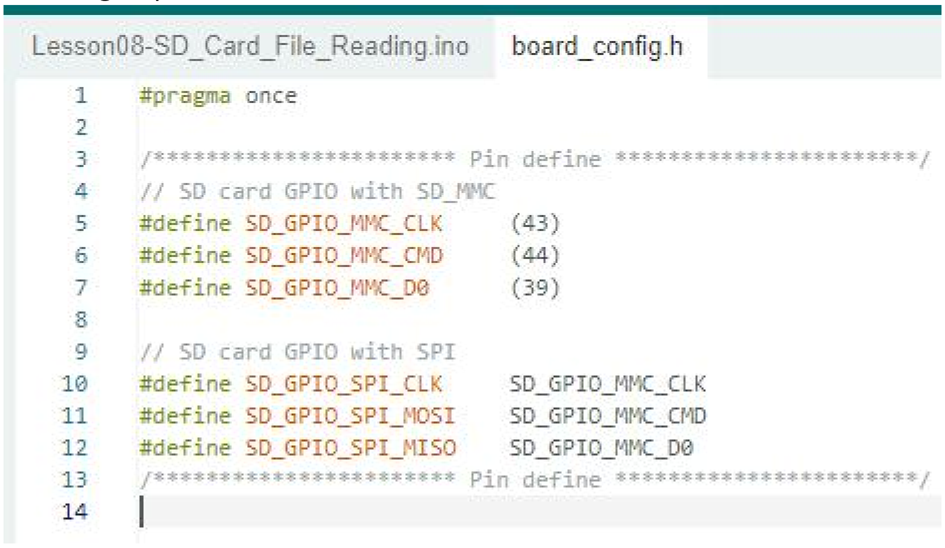


Name	Date modified	Type	Size
board_config.h	2026/3/10 10:45	C Header 源文件	1 KB
Lesson08-SD_Card_File_Reading.ino	2026/3/10 10:53	INO File	14 KB

After opening, you can see the project's code and configuration file board_config.h.

Let's understand the project architecture for this lesson.

This board_config.h header file is a hardware board-level configuration file. Its core purpose is to uniformly declare and manage the GPIO pin numbers corresponding to the SD card and two different communication interfaces (SD_MMC and SPI) through macro definitions. Among them, the SPI interface reuses the CLK, CMD, and D0 pins of the SD_MMC interface, achieving centralized management of hardware pins. This also facilitates subsequent code to directly reference pin numbers through macro names, improving code readability and maintainability. Meanwhile, the #pragma once directive ensures this header file is only included once during compilation, avoiding duplicate definition errors.



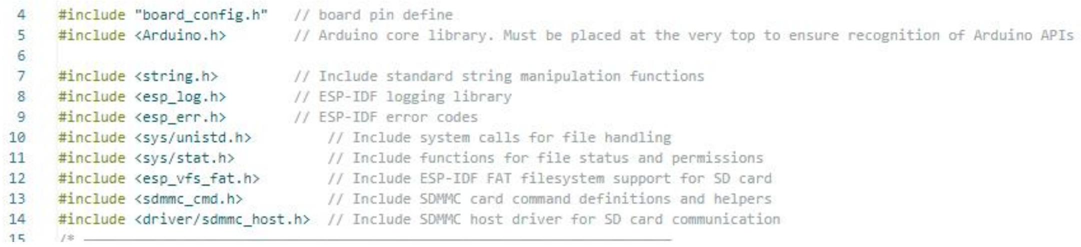
```
#pragma once

/*********************** Pin define ***********************/
// SD card GPIO with SD_MMC
#define SD_GPIO_MMC_CLK     (43)
#define SD_GPIO_MMC_CMD     (44)
#define SD_GPIO_MMC_D0      (39)

// SD card GPIO with SPI
#define SD_GPIO_SPI_CLK     SD_GPIO_MMC_CLK
#define SD_GPIO_SPI_MOSI    SD_GPIO_MMC_CMD
#define SD_GPIO_SPI_MISO    SD_GPIO_MMC_D0
/*********************** Pin define ***********************/

```

Next, let's take a look at the code in the main code .ino file together.

Code for SD Card File Reading and Writing

1. Header Files

This code is mainly used to include various header files required for program operation, providing underlying support for subsequent SD card file system read/write functionality.



```
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs

#include <string.h>         // Include standard string manipulation functions
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes
#include <sys/unistd.h>         // Include system calls for file handling
#include <sys/stat.h>           // Include functions for file status and permissions
#include <esp_vfs_fat.h>        // Include ESP-IDF FAT filesystem support for SD card
#include <sdmmc_cmd.h>          // Include SDMMC card command definitions and helpers
#include <driver/sdmmc_host.h>  // Include SDMMC host driver for SD card communication
/*
```

First, "#include "board_config.h"" introduces the development board hardware configuration file. This file typically defines SD card interfaces, GPIO pins, and hardware connection information for other peripherals.

Next, "<Arduino.h>" is included. This is the core library of the Arduino framework, providing basic APIs such as Serial, delay(), setup(), loop(), etc. Therefore, it must be placed in a relatively early position to ensure Arduino functions can be correctly recognized.

Then "<string.h>" provides standard C language string processing functions (such as strchr(), strlen(), etc.), used for processing string data after file reading.

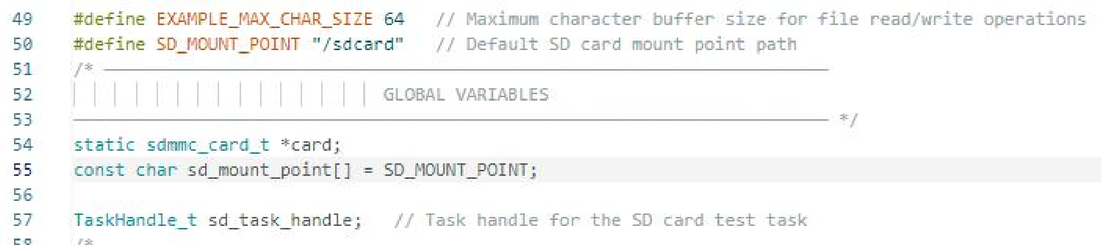
"<esp_log.h>" and "<esp_err.h>" come from the ESP-IDF underlying development framework, used for log output and error code definition respectively, facilitating the program to print debug information during operation and determine whether function execution was successful.

Next, "<sys/unistd.h>" and "<sys/stat.h>" belong to the POSIX system interface library, providing system calls related to file operations and file status management, enabling the program to use file read/write interfaces similar to Linux.

Finally, the three groups of header files "<esp_vfs_fat.h>", "<sdmmc_cmd.h>", and "<driver/sdmmc_host.h>" constitute the core of ESP32's SD card file system support: among them, esp_vfs_fat.h is responsible for mounting the FAT file system to the Virtual File System (VFS), enabling the program to access the SD card through standard functions such as fopen(), fread(), fwrite(); sdmmc_cmd.h provides SD card protocol commands and card information structures; while sdmmc_host.h is responsible for the SDMMC host controller driver, used to communicate with the SD card through hardware interfaces (such as CLK, CMD, D0 pins, etc.).

2. Macro Definitions and Global Variables

This code is mainly used to define constants and global variables related to the SD card file system, providing basic data structures for subsequent SD card initialization, file read/write, and task management.



```
#define EXAMPLE_MAX_CHAR_SIZE 64   // Maximum character buffer size for file read/write operations
#define SD_MOUNT_POINT "/sdcard"   // Default SD card mount path
/* ─────────────────────────────────────────────────────────
| | | | | | | | | | | | | | | GLOBAL VARIABLES
───────────────────────────────────────────────────────── */
static sdmmc_card_t *card;
const char sd_mount_point[] = SD_MOUNT_POINT;

TaskHandle_t sd_task_handle;   // Task handle for the SD card test task
/*
```

First, "#define EXAMPLE_MAX_CHAR_SIZE 64" defines a macro constant used to limit the maximum length of the character buffer for file read/write operations to 64 bytes. This avoids buffer overflow when reading text file content, improving program runtime safety.

Next, "#define SD_MOUNT_POINT "/sdcard"" defines the mount path for the SD card in the system. In ESP32's Virtual File System (VFS), after the SD card is mounted, it will be mapped to the /sdcard directory. Therefore, subsequent file operations in the program can directly access files in the SD card through paths like /sdcard/hello.txt.

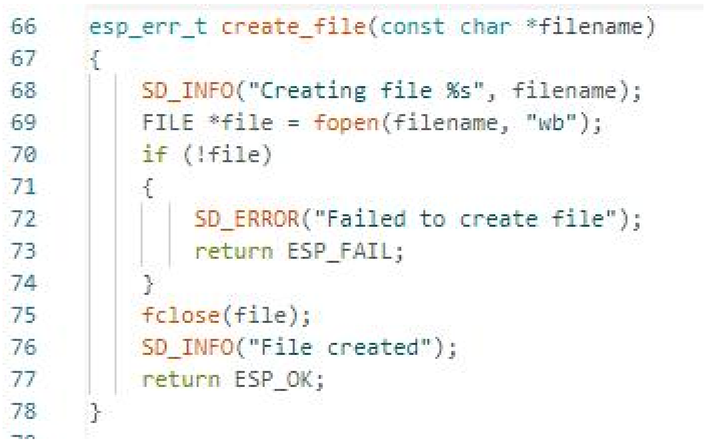
Then a global pointer variable "static sdmmc_card_t *card" is declared. It is used to save the SD card device information structure. After SD card initialization succeeds, this structure will record card type, capacity, operating frequency, and CID/CSD information. The program can obtain detailed hardware information of the SD card through it.

Next, "const char sd_mount_point[] = SD_MOUNT_POINT;" defines a constant string array used to save the SD card mount path. This allows the variable to be uniformly used as the file system path parameter in function calls, improving code maintainability and readability.

Finally, the variable "TaskHandle_t sd_task_handle" is defined. This is a FreeRTOS task handle used to save the control block address of the subsequently created SD card test task (sd_task). Through this handle, task management can be performed, such as querying task status, deleting tasks, or task control.

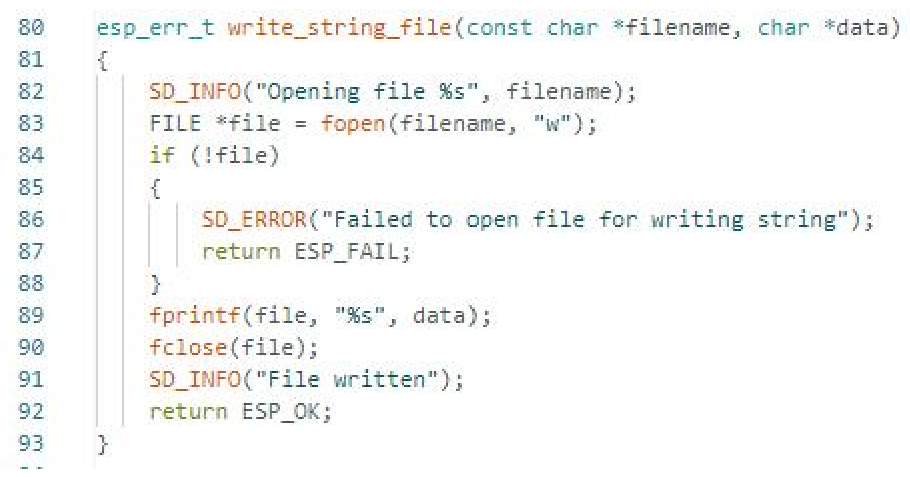
3. create_file：

Use fopen(filename, "wb") to create a file in binary write mode;Close the file immediately after successful creation;Return ESP_FAIL if opening fails.
Function: Ensure that an empty file exists on the SD card.


```
esp_err_t create_file(const char *filename)
{
    SD_INFO("Creating file %s", filename);
    FILE *file = fopen(filename, "wb");
    if (!file)
    {
        SD_ERROR("Failed to create file");
        return ESP_FAIL;
    }
    fclose(file);
    SD_INFO("File created");
    return ESP_OK;
}
```

4. write_string_file：

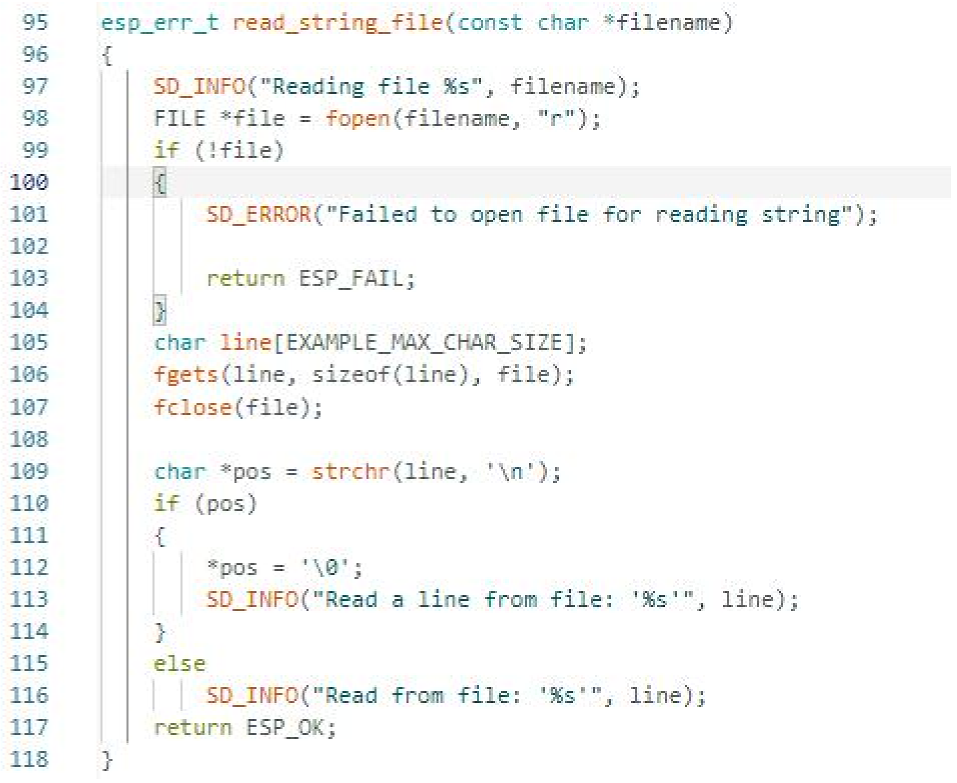
Open the file in text write mode using fopen(filename, "w");
Write the string using fprintf(file, "%s", data);
Close the file after writing.
Function: Save a section of text (string) into a file on the SD card.



```c
esp_err_t write_string_file(const char *filename, char *data)
{
    SD_INFO("Opening file %s", filename);
    FILE *file = fopen(filename, "w");
    if (!file)
    {
        SD_ERROR("Failed to open file for writing string");
        return ESP_FAIL;
    }
    fprintf(file, "%s", data);
    fclose(file);
    SD_INFO("File written");
    return ESP_OK;
}
```

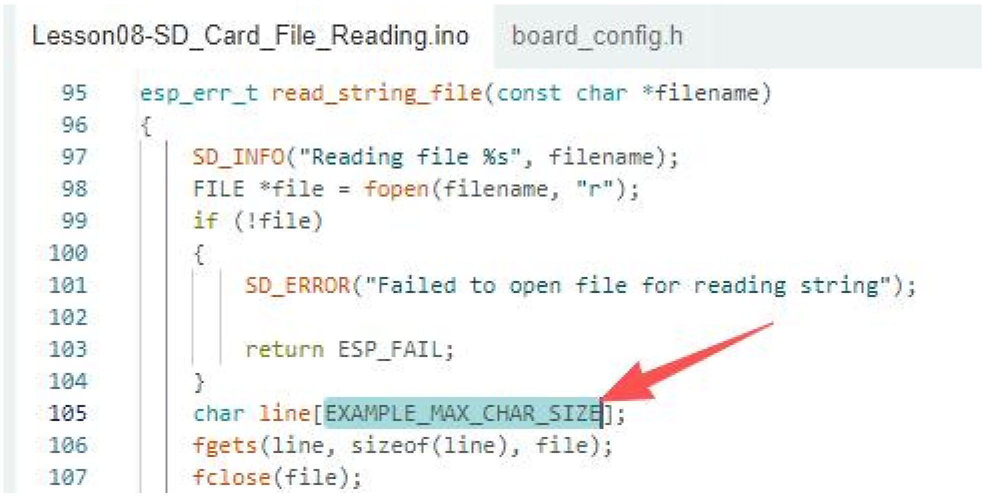
5. read_string_file:

Open the file for reading;
Use "fgets()" to read a line of text;
Check if there is a newline character "\n", and if so, replace it with a string terminator;
Print the read content.
Function: Read a line of text content from the file and output it to the "log".

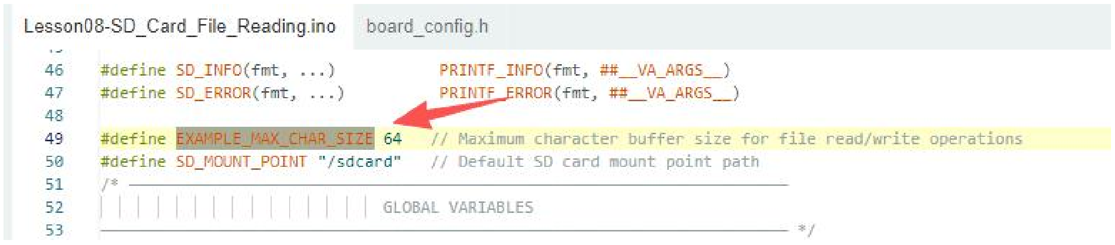


```c
esp_err_t read_string_file(const char *filename)
{
    SD_INFO("Reading file %s", filename);
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        SD_ERROR("Failed to open file for reading string");

        return ESP_FAIL;
    }
    char line[EXAMPLE_MAX_CHAR_SIZE];
    fgets(line, sizeof(line), file);
    fclose(file);

    char *pos = strchr(line, '\n');
    if (pos)
    {
        *pos = '\0';
        SD_INFO("Read a line from file: '%s'", line);
    }
    else
        SD_INFO("Read from file: '%s'", line);
    return ESP_OK;
}
```

Note:
The maximum number of characters that can be read here is 64. If you need to read more characters, you will need to adjust the size.


Lesson08-SD_Card_File_Reading.ino board_config.h

```
esp_err_t read_string_file(const char *filename)
{
    SD_INFO("Reading file %s", filename);
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        SD_ERROR("Failed to open file for reading string");

        return ESP_FAIL;
    }
    char line[EXAMPLE_MAX_CHAR_SIZE];
    fgets(line, sizeof(line), file);
    fclose(file);
```

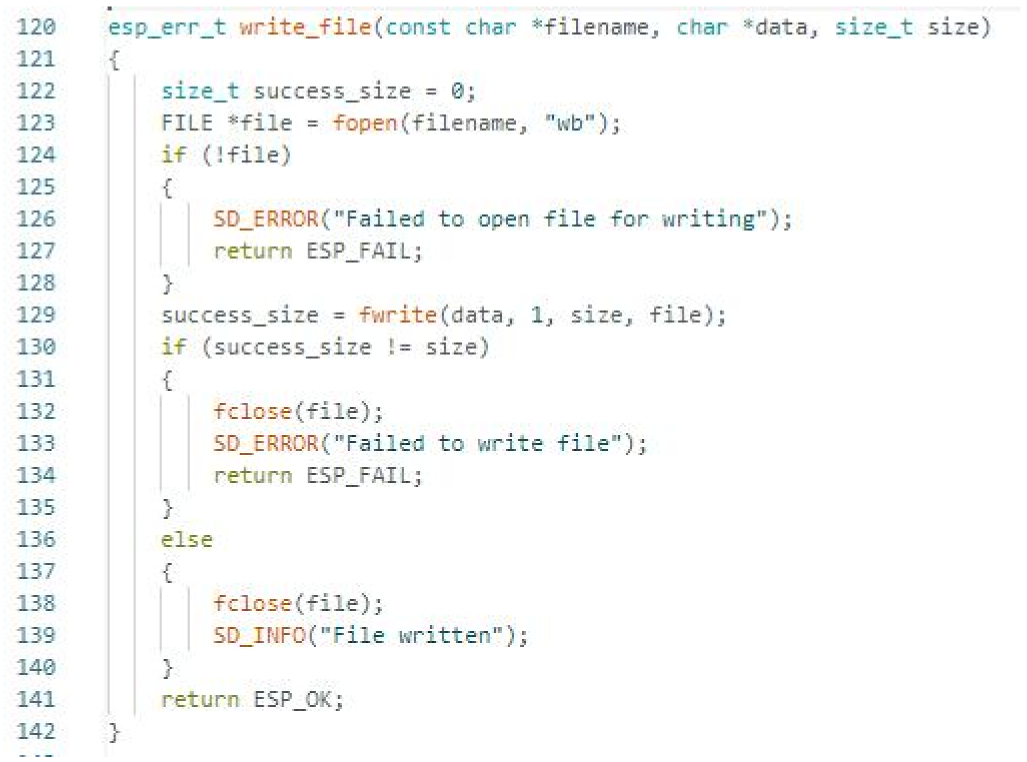

Lesson08-SD_Card_File_Reading.ino board_config.h

```
#define SD_INFO(fmt, ...)           PRINTF_INFO(fmt, ##__VA_ARGS__)
#define SD_ERROR(fmt, ...)          PRINTF_ERROR(fmt, ##__VA_ARGS__)

#define EXAMPLE_MAX_CHAR_SIZE 64    // Maximum character buffer size for file read/write operations
#define SD_MOUNT_POINT "/sdcard"    // Default SD card mount point path
/* ──────────────────────────────────────────────────────────────
│ │ │ │ │ │ │ │ │ │ │ │ │ │ GLOBAL VARIABLES
────────────────────────────────────────────────────────────── */
```

6. write_file:

Open the file in binary write mode ("wb");
Use "fwrite()" to write the "data" in memory to the file;
If the number of bytes written is not equal to "size", it indicates a write failure;

Finally, close the file.
Function: Suitable for writing binary data or image files.

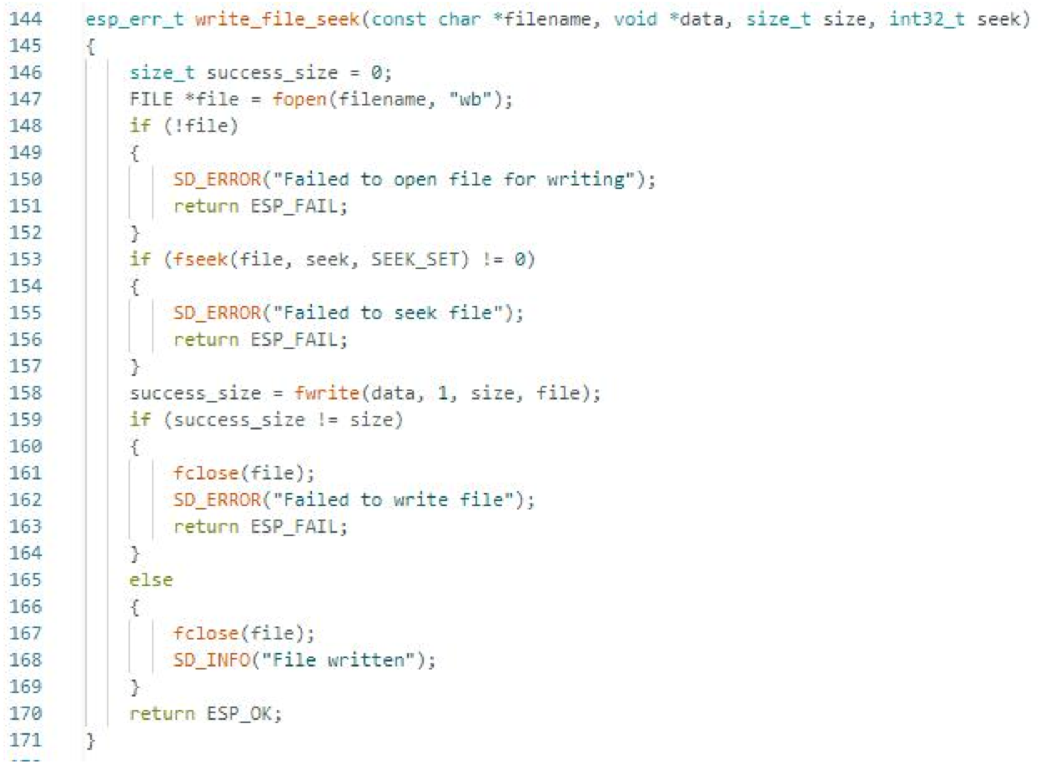


```
esp_err_t write_file(const char *filename, char *data, size_t size)
{
    size_t success_size = 0;
    FILE *file = fopen(filename, "wb");
    if (!file)
    {
        SD_ERROR("Failed to open file for writing");
        return ESP_FAIL;
    }
    success_size = fwrite(data, 1, size, file);
    if (success_size != size)
    {
        fclose(file);
        SD_ERROR("Failed to write file");
        return ESP_FAIL;
    }
    else
    {
        fclose(file);
        SD_INFO("File written");
    }
    return ESP_OK;
}
```

7. write_file_seek:

Open the file;
Call "fseek()" to move the file write pointer to the specified offset;
Then execute "fwrite()";

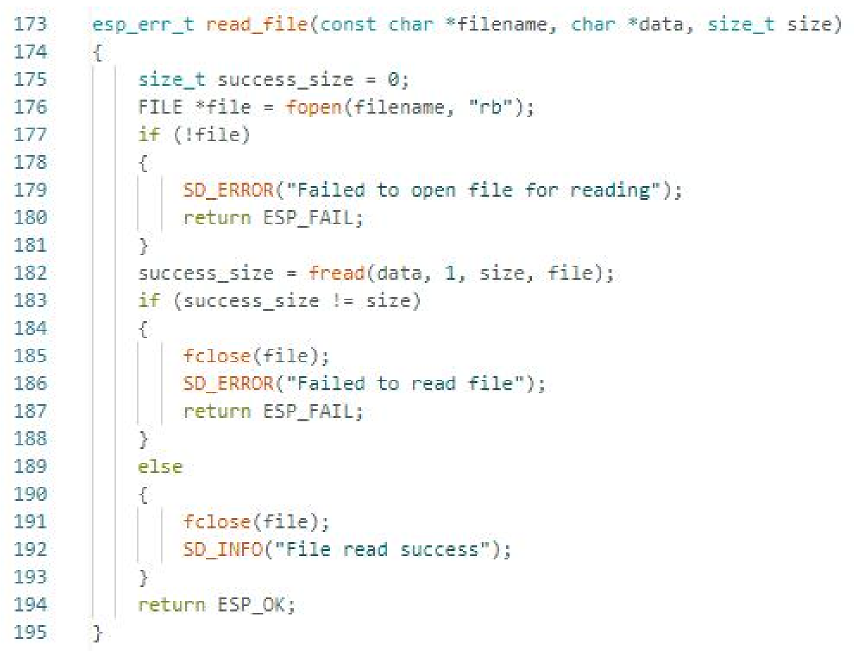
Return an error if the operation fails.
Function: Write data at a specific position in the file, commonly used for log appending or data block replacement.



```
esp_err_t write_file_seek(const char *filename, void *data, size_t size, int32_t seek)
{
    size_t success_size = 0;
    FILE *file = fopen(filename, "wb");
    if (!file)
    {
        SD_ERROR("Failed to open file for writing");
        return ESP_FAIL;
    }
    if (fseek(file, seek, SEEK_SET) != 0)
    {
        SD_ERROR("Failed to seek file");
        return ESP_FAIL;
    }
    success_size = fwrite(data, 1, size, file);
    if (success_size != size)
    {
        fclose(file);
        SD_ERROR("Failed to write file");
        return ESP_FAIL;
    }
    else
    {
        fclose(file);
        SD_INFO("File written");
    }
    return ESP_OK;
}
```

8. read_file:

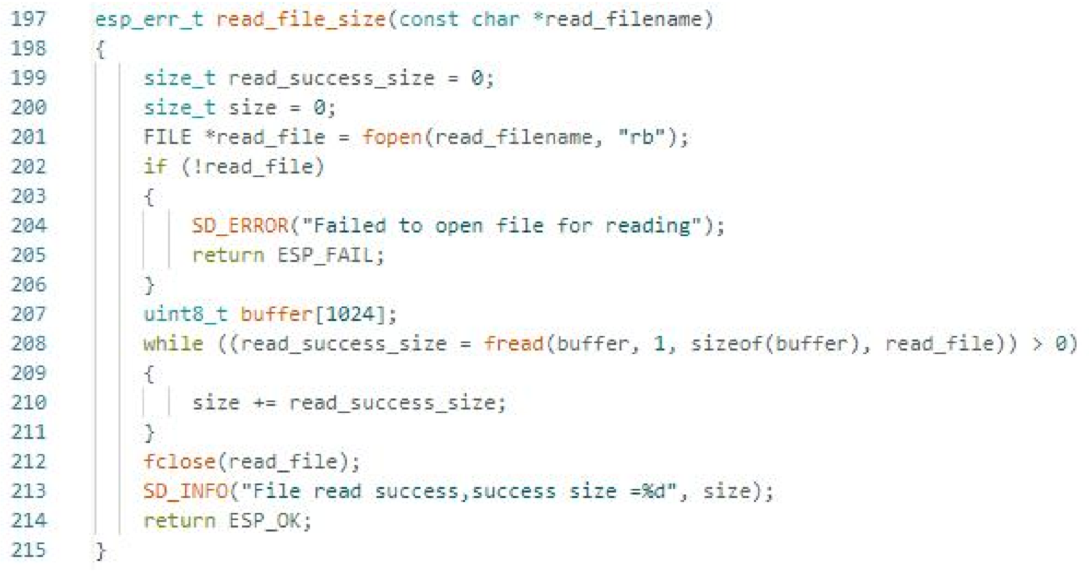
Open the file;
Use "fread()" to read a fixed-size data from the file;
If the number of bytes read does not match the expected value, an error is reported;
Otherwise, close the file and return success.
Function: Read binary files or fixed-length data blocks.



```c
esp_err_t read_file(const char *filename, char *data, size_t size)
{
    size_t success_size = 0;
    FILE *file = fopen(filename, "rb");
    if (!file)
    {
        SD_ERROR("Failed to open file for reading");
        return ESP_FAIL;
    }
    success_size = fread(data, 1, size, file);
    if (success_size != size)
    {
        fclose(file);
        SD_ERROR("Failed to read file");
        return ESP_FAIL;
    }
    else
    {
        fclose(file);
        SD_INFO("File read success");
    }
    return ESP_OK;
}
```

9. read_file_size:

Read all data blocks in the file in a loop;
Accumulate the "size" to get the total number of bytes of the file;
Output the total size of the file.
Function: Calculate the file size and verify the correctness of writing.



```c
esp_err_t read_file_size(const char *read_filename)
{
    size_t read_success_size = 0;
    size_t size = 0;
    FILE *read_file = fopen(read_filename, "rb");
    if (!read_file)
    {
        SD_ERROR("Failed to open file for reading");
        return ESP_FAIL;
    }
    uint8_t buffer[1024];
    while ((read_success_size = fread(buffer, 1, sizeof(buffer), read_file)) > 0)
    {
        size += read_success_size;
    }
    fclose(read_file);
    SD_INFO("File read success,success size =%d", size);
    return ESP_OK;
}
```

10. read_write_file:

Open the source file (for reading) and the target file (for writing);
Read 1024-byte content from the source file in blocks;
Write the content to the target file;
Check whether the number of written bytes is consistent with the number of read bytes;
Finally, close the files and output the message indicating successful copying.

Function: Implement file copying operation.

```
esp_err_t read_write_file(const char *read_filename, char *write_filename)
{
    size_t read_success_size = 0;
    size_t write_success_size = 0;
    size_t size = 0;
    FILE *read_file = fopen(read_filename, "rb");
    FILE *write_file = fopen(write_filename, "wb");
    if (!read_file)
    {
        SD_ERROR("Failed to open file for reading");
        return ESP_FAIL;
    }
    if (!write_file)
    {
        SD_ERROR("Failed to open file for writing");
        return ESP_FAIL;
    }
    uint8_t buffer[1024];
    while ((read_success_size = fread(buffer, 1, sizeof(buffer), read_file)) > 0)
    {
        write_success_size = fwrite(buffer, 1, read_success_size, write_file);
        if (write_success_size != read_success_size)
        {
            SD_ERROR("inconsistent reading and writing of data");
            return ESP_FAIL;
        }
        size += write_success_size;
    }
    fclose(read_file);
    fclose(write_file);
    SD_INFO("File read and write success,success size =%d", size);
    return ESP_OK;
}
```

11. sd_init:

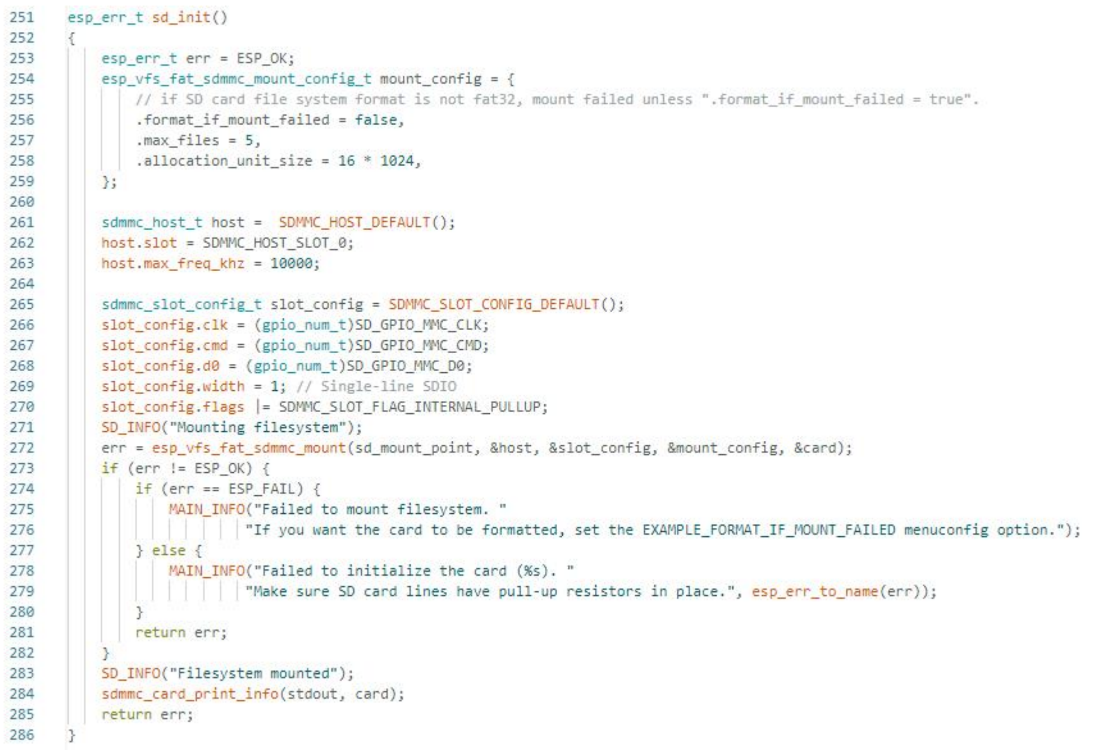
Create an "esp_vfs_fat_sdmmc_mount_config_t" configuration structure to set:

- "format_if_mount_failed = false" → Do not automatically format;
- "max_files = 5" → Maximum 5 files can be opened simultaneously;
- "allocation_unit_size = 16 * 1024" → Each cluster size is 16KB;

Initialize "sdmmc_host_t" and "sdmmc_slot_config_t":

- Set clock, command, and data line pins;
- Set bus width (1-line mode);
- Reduce the clock frequency to 10MHz to improve stability;

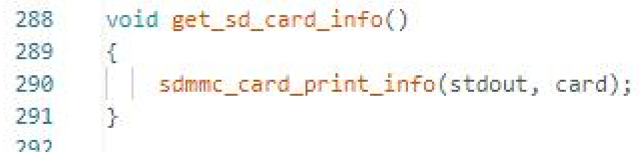
Call "esp_vfs_fat_sdmmc_mount()" to mount the SD card file system to "/sdcard";
If successful, print card information.
Function: Mount the SD card and establish the "FAT" file system.



```c
esp_err_t sd_init()
{
    esp_err_t err = ESP_OK;
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
        // if SD card file system format is not fat32, mount failed unless ".format_if_mount_failed = true".
        .format_if_mount_failed = false,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024,
    };

    sdmmc_host_t host =  SDMMC_HOST_DEFAULT();
    host.slot = SDMMC_HOST_SLOT_0;
    host.max_freq_khz = 10000;

    sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
    slot_config.clk = (gpio_num_t)SD_GPIO_MMC_CLK;
    slot_config.cmd = (gpio_num_t)SD_GPIO_MMC_CMD;
    slot_config.d0 = (gpio_num_t)SD_GPIO_MMC_D0;
    slot_config.width = 1; // Single-line SDIO
    slot_config.flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP;
    SD_INFO("Mounting filesystem");
    err = esp_vfs_fat_sdmmc_mount(sd_mount_point, &host, &slot_config, &mount_config, &card);
    if (err != ESP_OK) {
        if (err == ESP_FAIL) {
            MAIN_INFO("Failed to mount filesystem. "
                      "If you want the card to be formatted, set the EXAMPLE_FORMAT_IF_MOUNT_FAILED menuconfig option.");
        } else {
            MAIN_INFO("Failed to initialize the card (%s). "
                      "Make sure SD card lines have pull-up resistors in place.", esp_err_to_name(err));
        }
        return err;
    }
    SD_INFO("Filesystem mounted");
    sdmmc_card_print_info(stdout, card);
    return err;
}
```

12. get_sd_card_info:

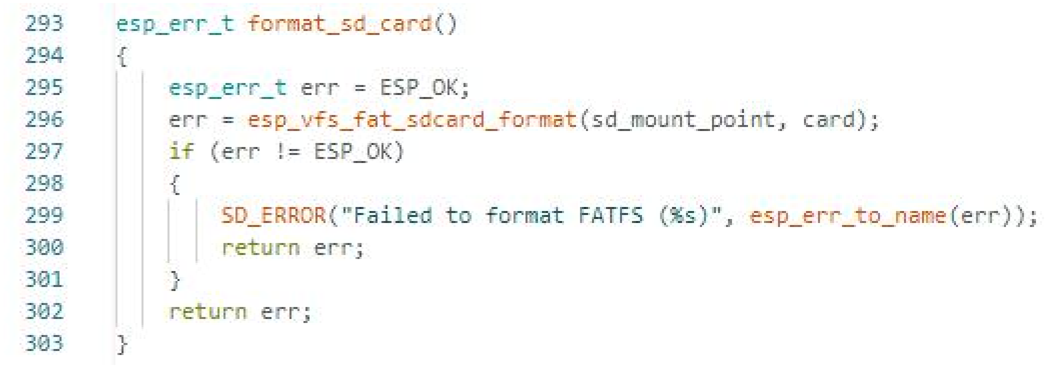
Print detailed information such as the type, capacity, and speed of the SD card to the console.



```c
void get_sd_card_info()
{
    sdmmc_card_print_info(stdout, card);
}
```

13. format_sd_card:

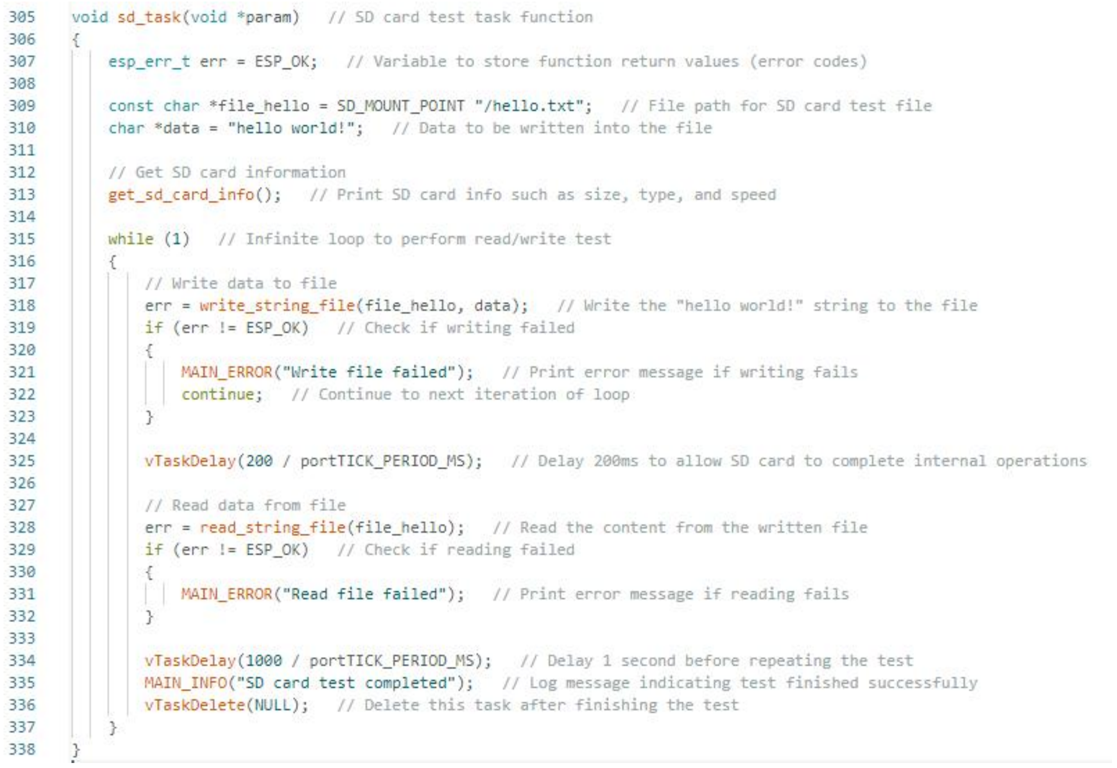
Call "esp_vfs_fat_sdcard_format()" to format the "FAT" file system;
Output an error message if formatting fails.
Function: Clear the SD card file system and reformat it.



```c
esp_err_t format_sd_card()
{
    esp_err_t err = ESP_OK;
    err = esp_vfs_fat_sdcard_format(sd_mount_point, card);
    if (err != ESP_OK)
    {
        SD_ERROR("Failed to format FATFS (%s)", esp_err_to_name(err));
        return err;
    }
    return err;
}
```

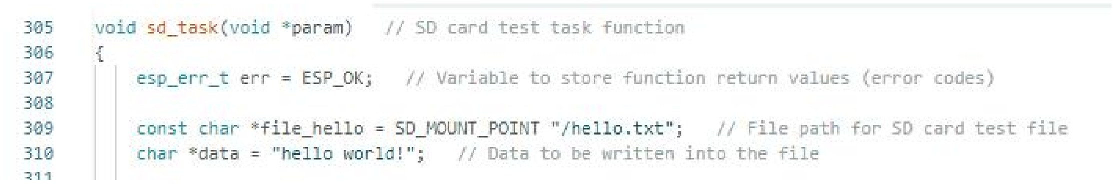
14. sd_task

This code implements an SD card test task sd_task based on FreeRTOS, which is used to verify whether the SD card file system can perform read and write operations normally.



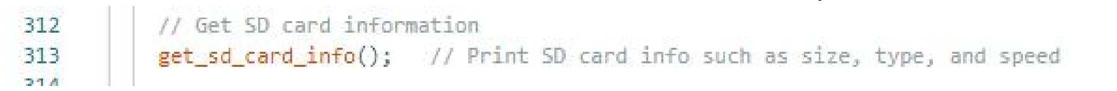
```c
void sd_task(void *param)   // SD card test task function
{
    esp_err_t err = ESP_OK;   // Variable to store function return values (error codes)

    const char *file_hello = SD_MOUNT_POINT "/hello.txt";   // File path for SD card test file
    char *data = "hello world!";   // Data to be written into the file

    // Get SD card information
    get_sd_card_info();   // Print SD card info such as size, type, and speed

    while (1)   // Infinite loop to perform read/write test
    {
        // Write data to file
        err = write_string_file(file_hello, data);   // Write the "hello world!" string to the file
        if (err != ESP_OK)   // Check if writing failed
        {
            MAIN_ERROR("Write file failed");   // Print error message if writing fails
            continue;   // Continue to next iteration of loop
        }

        vTaskDelay(200 / portTICK_PERIOD_MS);   // Delay 200ms to allow SD card to complete internal operations

        // Read data from file
        err = read_string_file(file_hello);   // Read the content from the written file
        if (err != ESP_OK)   // Check if reading failed
        {
            MAIN_ERROR("Read file failed");   // Print error message if reading fails
        }

        vTaskDelay(1000 / portTICK_PERIOD_MS);   // Delay 1 second before repeating the test
        MAIN_INFO("SD card test completed");   // Log message indicating test finished successfully
        vTaskDelete(NULL);   // Delete this task after finishing the test
    }
}
```

The function first defines an "esp_err_t err" variable to save the return results after each function execution (in ESP-IDF, ESP_OK typically indicates success, other values indicate errors). Next, it defines the file path "const char *file_hello = SD_MOUNT_POINT "/hello.txt"". This path indicates creating or accessing a test file named hello.txt in the SD card already mounted to the /sdcard directory. It also defines a string pointer "char *data = "hello world!"" as the test data to be written to the file.



```c
void sd_task(void *param)   // SD card test task function
{
    esp_err_t err = ESP_OK;   // Variable to store function return values (error codes)

    const char *file_hello = SD_MOUNT_POINT "/hello.txt";   // File path for SD card test file
    char *data = "hello world!";   // Data to be written into the file
```

At the start of the task, get_sd_card_info() is called to obtain and print the basic information of the SD card, such as capacity, type, and communication speed, which is used to confirm that the SD card has been initialized correctly.



```c
    // Get SD card information
    get_sd_card_info();   // Print SD card info such as size, type, and speed
```

Subsequently, the program enters a "while(1)" loop to execute read and write tests: First, "write_string_file(file_hello, data)" is called to write the string "hello world!" into the hello.txt file. If the writing fails, an error log is output through "MAIN_ERROR()" and the program re-enters the next loop;

```c
        while (1)   // Infinite loop to perform read/write test
        {
            // Write data to file
            err = write_string_file(file_hello, data);   // Write the "hello world!" string to the file
            if (err != ESP_OK)   // Check if writing failed
            {
                MAIN_ERROR("Write file failed");   // Print error message if writing fails
                continue;   // Continue to next iteration of loop
            }
```

If the writing is successful, the program delays for approximately 200 milliseconds via "vTaskDelay(200 / portTICK_PERIOD_MS)" to allow the SD card a certain amount of time to complete internal write operations. Then, it calls "read_string_file(file_hello)" to read the content from the file just written and print it out; if the reading fails, an error log is also output.

```c
            vTaskDelay(200 / portTICK_PERIOD_MS);   // Delay 200ms to allow SD card to complete internal operations

            // Read data from file
            err = read_string_file(file_hello);   // Read the content from the written file
            if (err != ESP_OK)   // Check if reading failed
            {
                MAIN_ERROR("Read file failed");   // Print error message if reading fails
            }
```

After completing the read and write operations, the task delays for another 1 second and prints "SD card test completed" to indicate the completion of one SD card read and write test.

```c

            vTaskDelay(1000 / portTICK_PERIOD_MS);   // Delay 1 second before repeating the test
            MAIN_INFO("SD card test completed");   // Log message indicating test finished successfully
            vTaskDelete(NULL);   // Delete this task after finishing the test
        }
    }

```

Finally, "vTaskDelete(NULL)" is called to delete the current task, making the FreeRTOS task end automatically after completing one test. Overall, the function of this function is to complete a complete test process of SD card information reading, file writing and file reading in an independent task to confirm that the SD card and its FAT file system can work normally.

15. Init

This code implements a system initialization function "Init()", whose main function is to complete the initialization of the SD card and ensure its successful mounting. If the initialization fails, it will keep retrying until it succeeds.

```c
void Init(void)   // System initialization function
{
    esp_err_t err = ESP_OK;   // Variable to store error codes

    // Initialize SD card
    err = sd_init();   // Call SD card initialization function
    // Check if initialization failed
    while (ESP_OK != err) {
        MAIN_ERROR("%s initialization failed [ %s ]", "SD card", esp_err_to_name(err));   // Print module name and error description
        delay (1000);
        err = sd_init();   // Call SD card initialization function
    }
}
```

At the start of the function, a variable "esp_err_t err = ESP_OK" is defined to store the return status code after the function is executed. Among them, "esp_err_t" is a standard error type used in ESP-IDF to indicate the execution result of a function, and

"ESP_OK" means the execution is successful.

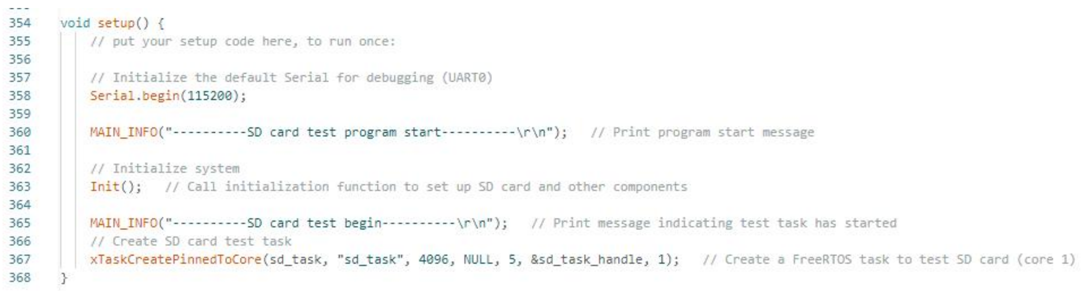
Subsequently, the program calls the "sd_init()" function to attempt to initialize the SD card. This function usually completes operations such as SDMMC interface configuration, SD card communication initialization, and FAT file system mounting, and returns the corresponding execution result.

Then the program judges whether the initialization is successful through "while (ESP_OK != err)". If the return value is not "ESP_OK", it indicates that the SD card initialization has failed, such as the SD card not being inserted, wrong pin connection, or file system mounting failure. At this time, the program will call "MAIN_ERROR()" to print an error log, where "esp_err_to_name(err)" will convert the error code into a readable error name to facilitate debugging and locating the problem.

Then the program delays for 1 second through "delay(1000)" to avoid wasting system resources due to frequent repeated attempts when the initialization fails. After the delay ends, it calls "sd_init()" again to re-initialize the SD card. This process will keep looping until the SD card is successfully initialized and returns "ESP_OK".

16. Setup Section

This code is the "setup()" initialization function in an Arduino program, which is executed only once when the device is powered on or the system starts up. It is mainly responsible for starting the debugging serial port, completing system initialization, and creating a FreeRTOS task for testing the read and write functions of the SD card.



```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    MAIN_INFO("----------SD card test program start----------\r\n");   // Print program start message

    // Initialize system
    Init();   // Call initialization function to set up SD card and other components

    MAIN_INFO("----------SD card test begin----------\r\n");   // Print message indicating test task has started
    // Create SD card test task
    xTaskCreatePinnedToCore(sd_task, "sd_task", 4096, NULL, 5, &sd_task_handle, 1);   // Create a FreeRTOS task to test SD card (core 1)
}
```

First, the program initializes the default serial port (UART0) via "Serial.begin(115200)" and sets the communication baud rate to 115200, so as to output debugging information in the serial monitor.

Subsequently, it calls "MAIN_INFO()" to print the startup prompt "SD card test program start", which is used to indicate that the SD card test program has started running.

Then, it invokes the "Init()" function to perform system initialization. Inside this function, an attempt is made to initialize the SD card and mount the FAT file system. If the initialization fails, it will automatically retry in a loop until successful, thus ensuring that the SD card is ready before the system proceeds to run.

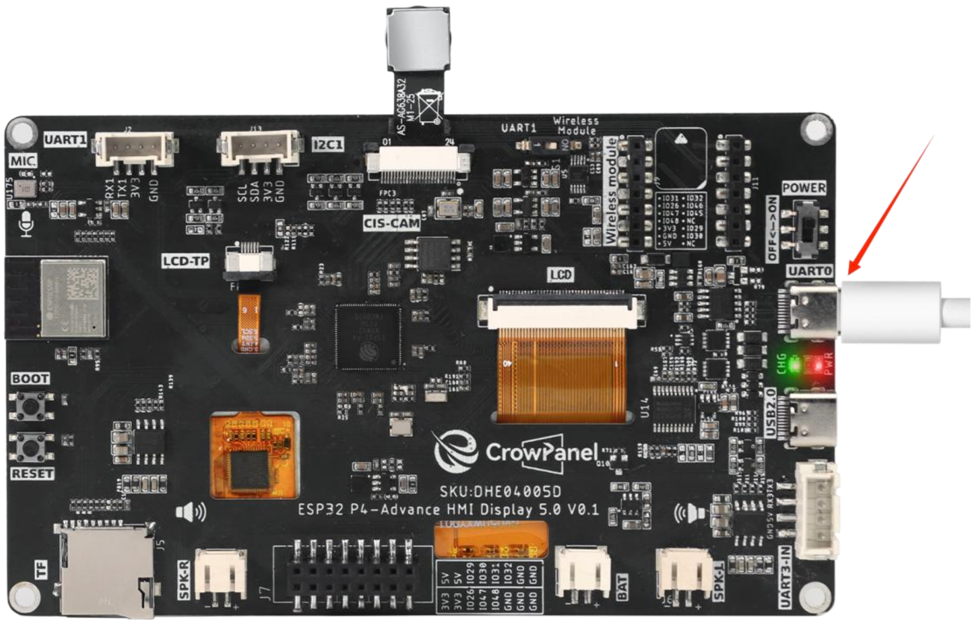
After the initialization is completed, the program prints the log "SD card test begin" again to prompt that the SD card test is about to start, and then calls "xTaskCreatePinnedToCore()" to create a FreeRTOS task "sd_task" for executing the SD card read and write tests.

The parameters of this function represent in sequence: the task function is "sd_task", the task name is "sd_task", the task stack size is 4096 bytes, the task parameter is NULL, the task priority is 5, the task handle is saved to the "sd_task_handle" variable, and the task is pinned to run on CPU Core 1.

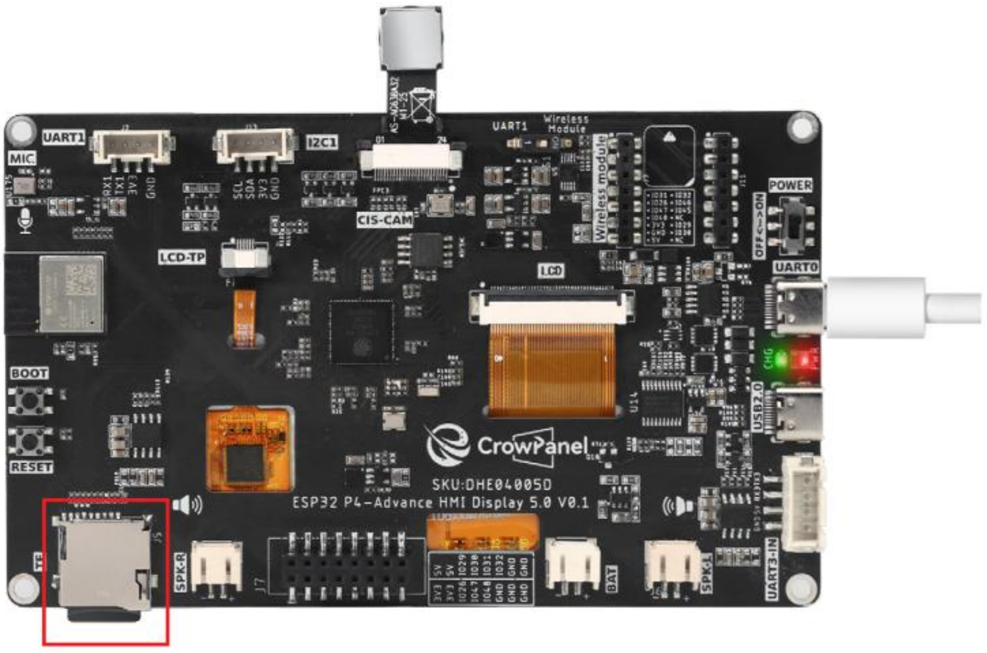
In this way, the read and write tests of the SD card will run in an independent FreeRTOS task without blocking the main program, thereby achieving a more stable and efficient system operation.

Programming Steps

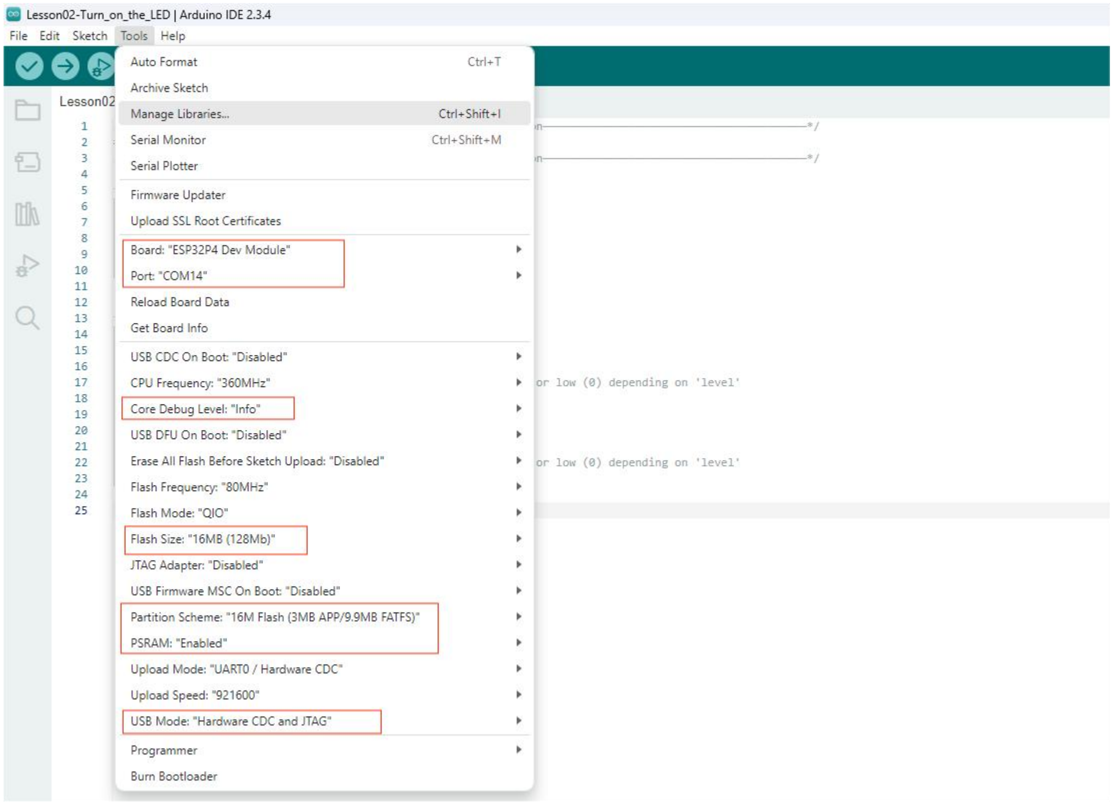
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



Then insert the SD card you will use into the SD card slot of the Advance-P4.



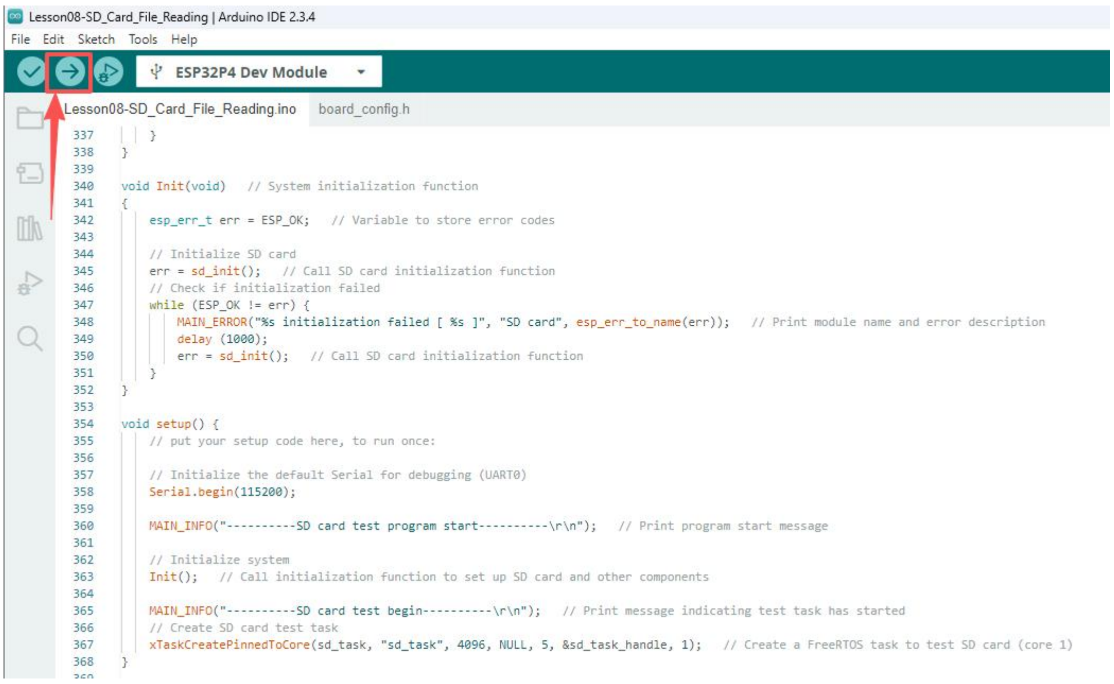
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

File Edit Sketch Tools Help

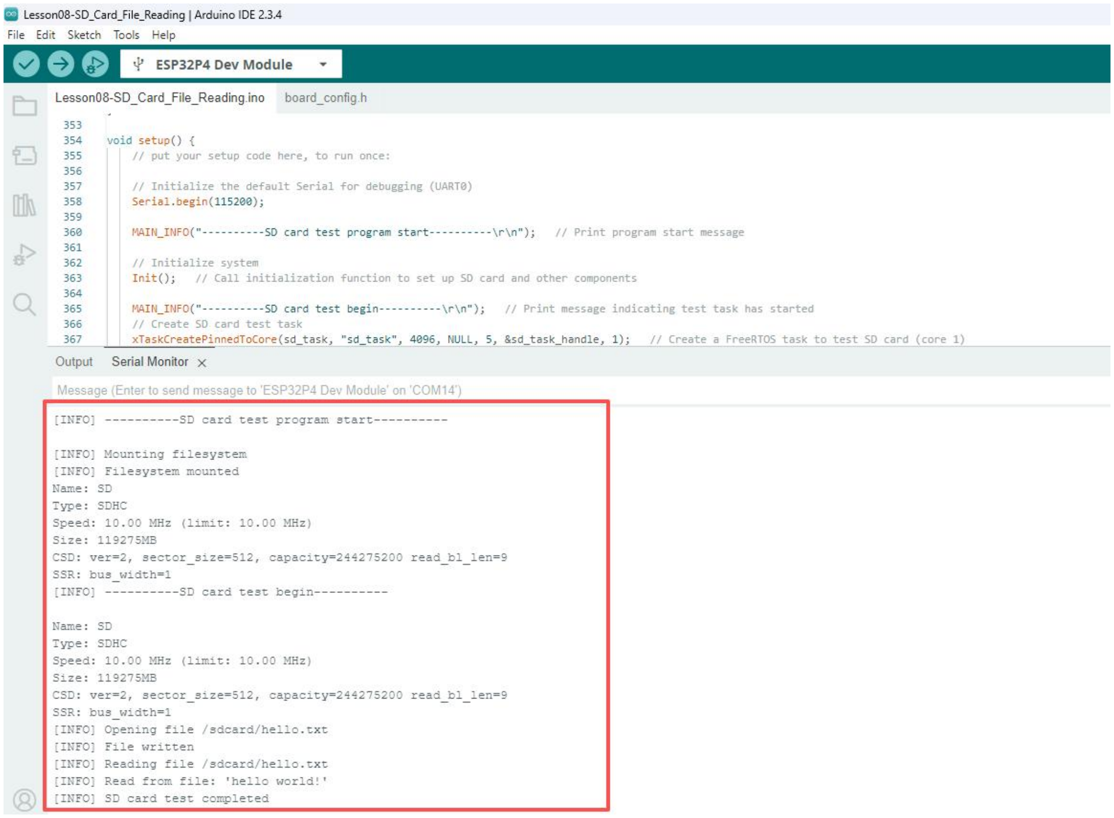
Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader

Then we compile and upload the code.


Lesson08-SD_Card_File_Reading | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson08-SD_Card_File_Reading.ino
board_config.h

```
    }
}

void Init(void)   // System initialization function
{
    esp_err_t err = ESP_OK;   // Variable to store error codes

    // Initialize SD card
    err = sd_init();   // Call SD card initialization function
    // Check if initialization failed
    while (ESP_OK != err) {
        MAIN_ERROR("%s initialization failed [ %s ]", "SD card", esp_err_to_name(err));   // Print module name and error description
        delay (1000);
        err = sd_init();   // Call SD card initialization function
    }
}

void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    MAIN_INFO("----------SD card test program start----------\r\n");   // Print program start message

    // Initialize system
    Init();   // Call initialization function to set up SD card and other components

    MAIN_INFO("----------SD card test begin----------\r\n");   // Print message indicating test task has started
    // Create SD card test task
    xTaskCreatePinnedToCore(sd_task, "sd_task", 4096, NULL, 5, &sd_task_handle, 1);   // Create a FreeRTOS task to test SD card (core 1)
}
```

After running the code, you will be able to visually see that a file named "hello.txt" appears in the SD card, with the content "hello world!" already written in it.


Lesson08-SD_Card_File_Reading | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson08-SD_Card_File_Reading.ino
board_config.h
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    MAIN_INFO("----------SD card test program start----------\r\n");   // Print program start message

    // Initialize system
    Init();   // Call initialization function to set up SD card and other components

    MAIN_INFO("----------SD card test begin----------\r\n");   // Print message indicating test task has started
    // Create SD card test task
    xTaskCreatePinnedToCore(sd_task, "sd_task", 4096, NULL, 5, &sd_task_handle, 1);   // Create a FreeRTOS task to test SD card (core 1)
```

```
[INFO] ----------SD card test program start----------

[INFO] Mounting filesystem
[INFO] Filesystem mounted
Name: SD
Type: SDHC
Speed: 10.00 MHz (limit: 10.00 MHz)
Size: 119275MB
CSD: ver=2, sector_size=512, capacity=244275200 read_bl_len=9
SSR: bus_width=1
[INFO] ----------SD card test begin----------

Name: SD
Type: SDHC
Speed: 10.00 MHz (limit: 10.00 MHz)
Size: 119275MB
CSD: ver=2, sector_size=512, capacity=244275200 read_bl_len=9
SSR: bus_width=1
[INFO] Opening file /sdcard/hello.txt
[INFO] File written
[INFO] Reading file /sdcard/hello.txt
[INFO] Read from file: 'hello world!'
[INFO] SD card test completed
```

Lesson09--- LVGL Lighting Control

Introduction

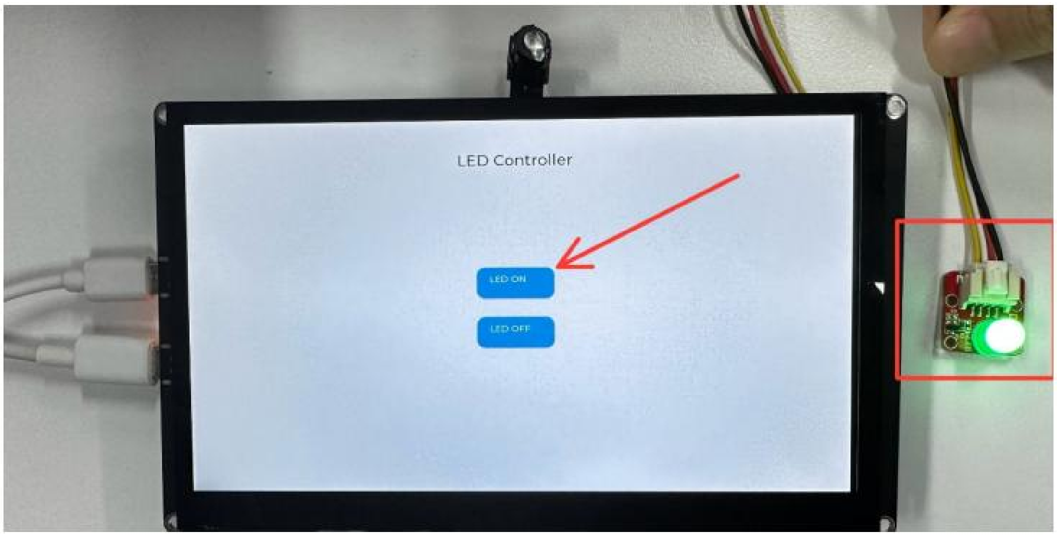
In previous courses, we separately lit an LED, implemented touch testing, and lit up the screen.In this lesson, we will use LVGL to create two buttons to control the LED connected to the UART1 interface for turning on and off operations.
Pressing the ON button can turn on the LED, and pressing the OFF button can turn off the LED.

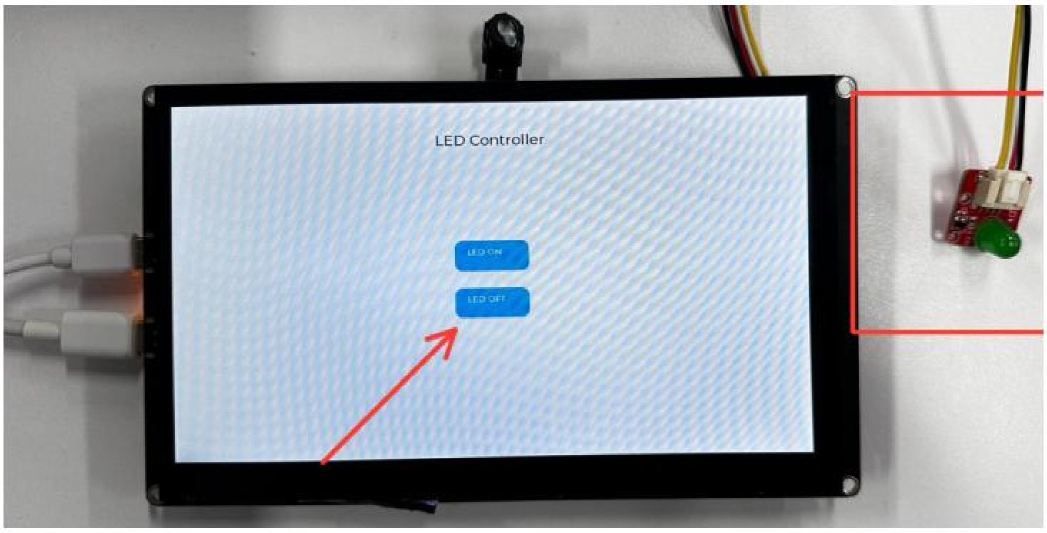
Learning Goals

1. Understand the GPIO pin configuration of the ESP32-P4 development board and the hardware control principle of the LED (the UART1 interface is connected to the LED).
2. Master the UI creation methods of the LVGL graphics library, including the drawing and layout of buttons and labels, as well as the binding of callback functions for button click events.
3. Master the linkage method between LVGL and the touch screen/display driver, and realize the complete process from touch interface operation to hardware (LED) control.

Preview of the Result

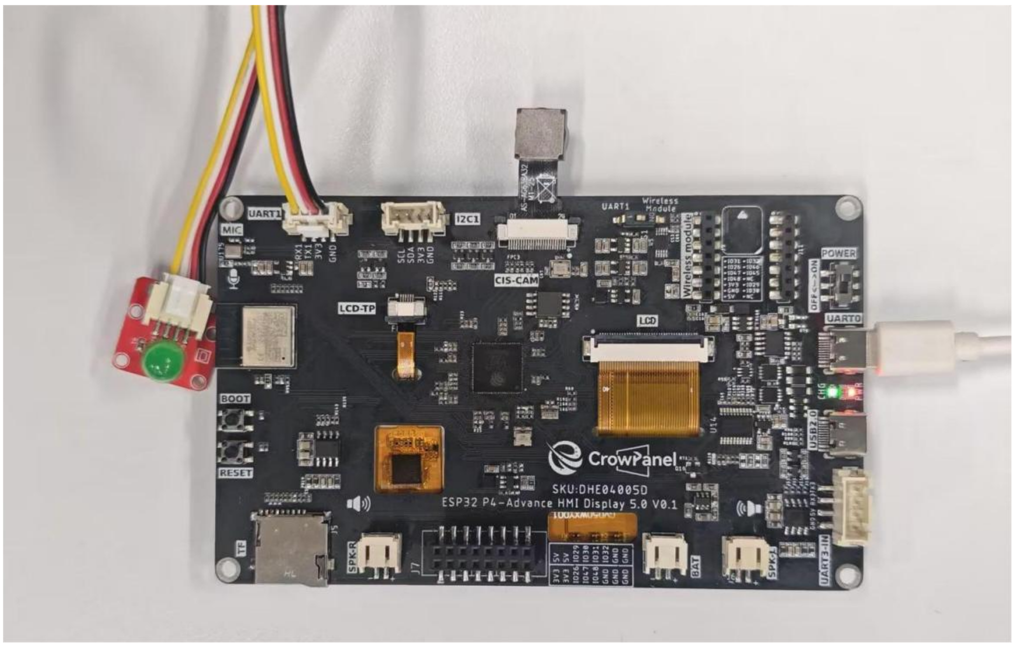
After running the code, when you press the "LED ON" button on the Advance-P4, you will be able to turn on the LED; when you press the "LED OFF" button, you will be able to turn off the LED.





Hardware Used in This Lesson

The UART1 interface on the Advance-P4 is connected to an LED.



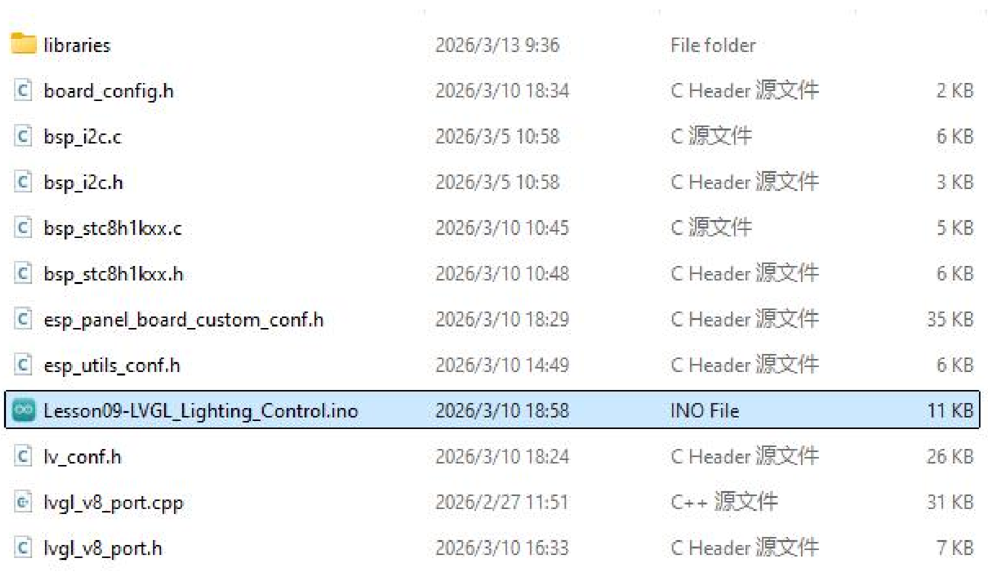
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson09-LVGL_Lighting_Control

Key Explanations

How to bind the screen touch with the LVGL interface? And how to light up the LED light by touching the LVGL interface? Next, let's explore these questions together.

Double-click to open the code (ino file) for this lesson.



libraries	2026/3/13 9:36	File folder	
board_config.h	2026/3/10 18:34	C Header 源文件	2 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/10 10:48	C Header 源文件	6 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
Lesson09-LVGL_Lighting_Control.ino	2026/3/10 18:58	INO File	11 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

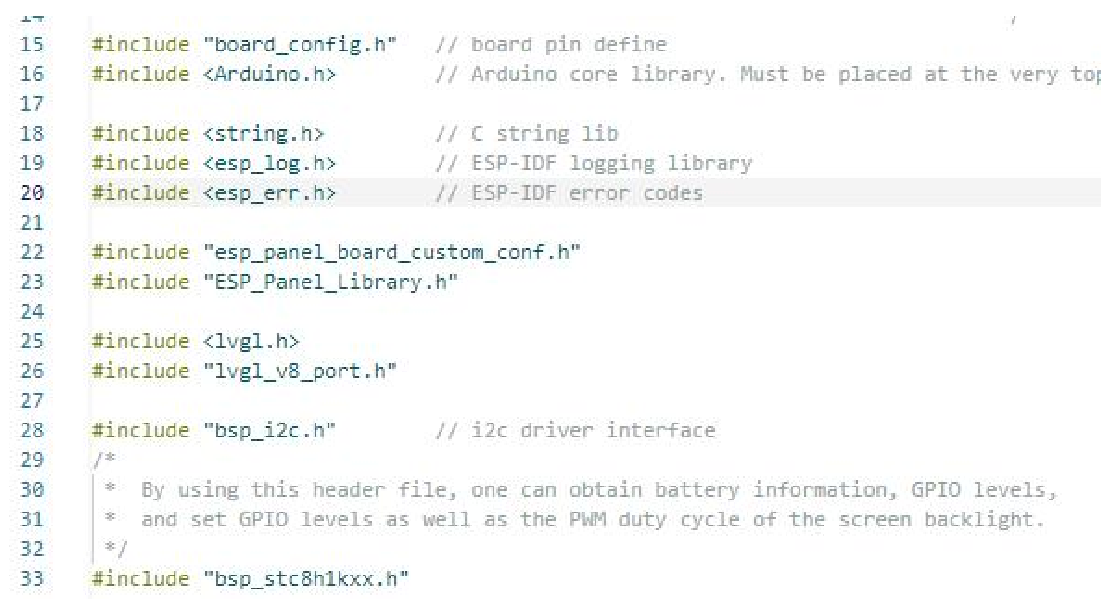
After opening it, you can see the code of this project, the configuration file "board_config.h", as well as other related screen driver files.

In the preface, we have already imported all the library files into the Arduino's "libraries" folder. If you haven't done so, please follow the instructions in the preface to import all the library files into the "libraries" folder of Arduino.

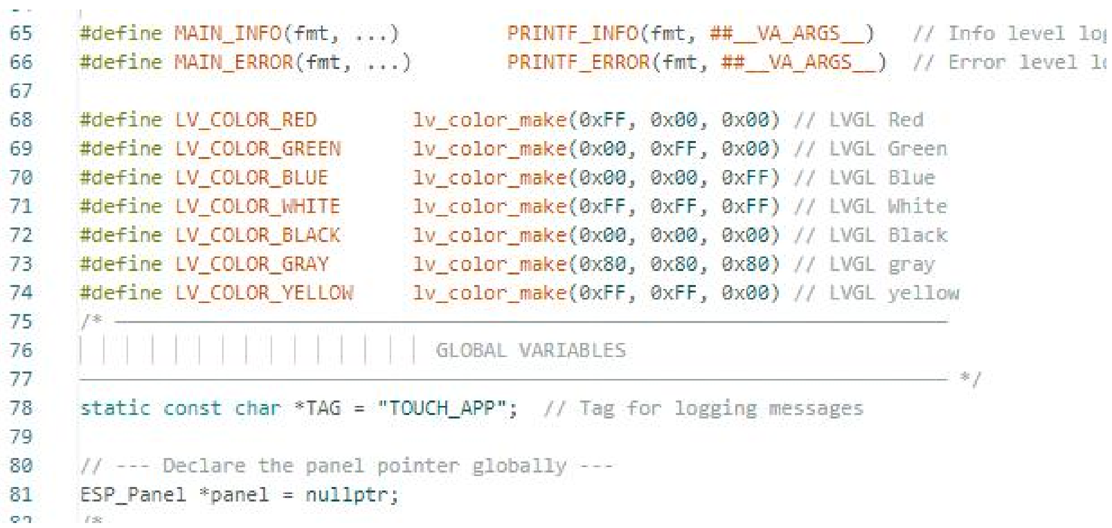
(The meanings and functions of each of these files have been elaborated in detail in Lesson 7. We will not go into further details here. If you have any questions, you can review the previous course content.)

Next, we will focus on studying the .ino main code file to see how to implement the control points of the LVGL interface for lighting.

(The detailed explanations of header files, namespaces, macro definitions, and global variables have been covered in Lesson 7. We will not go into excessive detail here.)



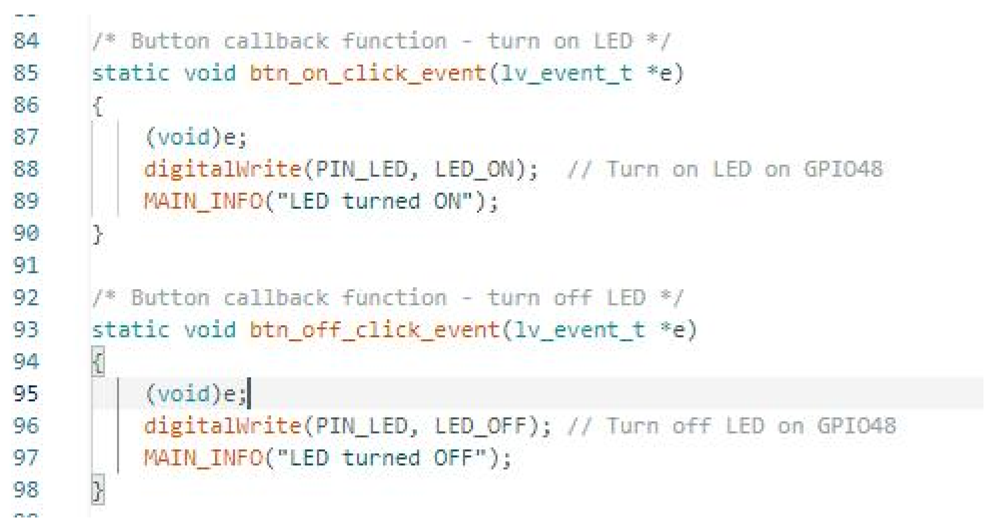
```
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very to

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

#include <lvgl.h>
#include "lvgl_v8_port.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"
```



```
#define MAIN_INFO(fmt, ...)         PRINTF_INFO(fmt, ##__VA_ARGS__)   // Info level lo
#define MAIN_ERROR(fmt, ...)        PRINTF_ERROR(fmt, ##__VA_ARGS__)  // Error level l

#define LV_COLOR_RED        lv_color_make(0xFF, 0x00, 0x00) // LVGL Red
#define LV_COLOR_GREEN      lv_color_make(0x00, 0xFF, 0x00) // LVGL Green
#define LV_COLOR_BLUE       lv_color_make(0x00, 0x00, 0xFF) // LVGL Blue
#define LV_COLOR_WHITE      lv_color_make(0xFF, 0xFF, 0xFF) // LVGL White
#define LV_COLOR_BLACK      lv_color_make(0x00, 0x00, 0x00) // LVGL Black
#define LV_COLOR_GRAY       lv_color_make(0x80, 0x80, 0x80) // LVGL gray
#define LV_COLOR_YELLOW     lv_color_make(0xFF, 0xFF, 0x00) // LVGL yellow
/* ---------------------------------------------------------------------------
 | | | | | | | | | | | | | | | GLOBAL VARIABLES
 ----------------------------------------------------------------------------- */
static const char *TAG = "TOUCH_APP";  // Tag for logging messages

// --- Declare the panel pointer globally ---
ESP_Panel *panel = nullptr;
```

1. Callback Functions

This code defines two LVGL button event callback functions, which are used to control the on and off of the LED on the development board when the user clicks the screen buttons, and output log information through the serial port.



```
/* Button callback function - turn on LED */
static void btn_on_click_event(lv_event_t *e)
{
    (void)e;
    digitalWrite(PIN_LED, LED_ON);  // Turn on LED on GPIO48
    MAIN_INFO("LED turned ON");
}

/* Button callback function - turn off LED */
static void btn_off_click_event(lv_event_t *e)
{
    (void)e;
    digitalWrite(PIN_LED, LED_OFF); // Turn off LED on GPIO48
    MAIN_INFO("LED turned OFF");
}
```

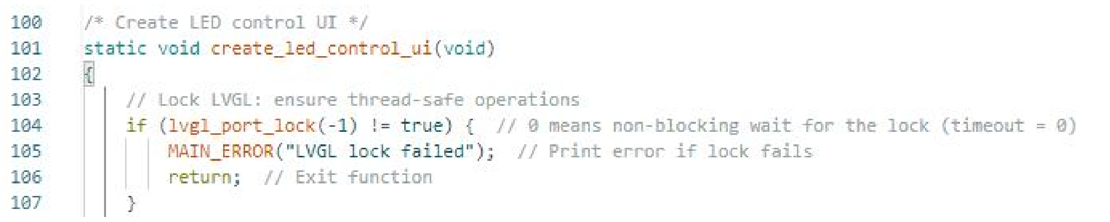
The first function btn_on_click_event(lv_event_t *e) serves as the event handler for the LED turn-on button. When the user clicks the "LED ON" button on the touch screen interface and triggers the LV_EVENT_CLICKED event, LVGL invokes this function. The function parameter lv_event_t *e is an LVGL event structure that contains information such as the object that triggered the event and the event type. However, it is not utilized in this example, so (void)e; is used to explicitly mark it as an unused parameter to prevent compiler warnings. Subsequently, the function calls digitalWrite(PIN_LED, LED_ON) to set the GPIO pin defined as PIN_LED (GPIO48 in the code) to the LED_ON level, thereby lighting up the LED, and outputs an info-level log via MAIN_INFO("LED turned ON") to indicate that the LED has been turned on.

The second function btn_off_click_event(lv_event_t *e) features essentially the same structure and logic. It is used to handle the click event of the LED turn-off button: when the user clicks the "LED OFF" button, the function also ignores the event parameter e, then calls digitalWrite(PIN_LED, LED_OFF) to set GPIO48 to the level state that turns off the LED, thus turning off the LED, and prints log information via MAIN_INFO("LED turned OFF").

2. UI Interface Creation

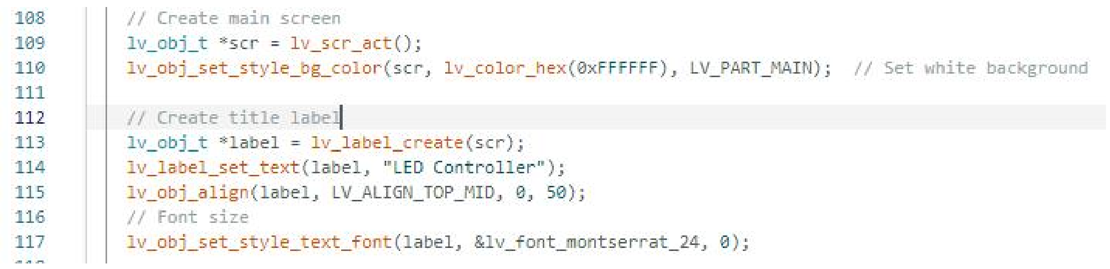
This code defines a function create_led_control_ui() for creating an LED control interface (UI) based on the LVGL graphics library. Its main function is to draw a title and two buttons on the screen, and bind the button click events to the LED control functions, thereby realizing the control of the on and off of the development board's LED through the touch screen.

At the start of the function, lvgl_port_lock(-1) is called to lock LVGL. This is because LVGL may be accessed by different tasks in a multi-tasking environment such as ESP32's FreeRTOS, and locking ensures that the UI creation process is thread-safe. The parameter -1 means to wait indefinitely until the lock is acquired; if the locking fails, an error message is printed via MAIN_ERROR() and the function returns directly.



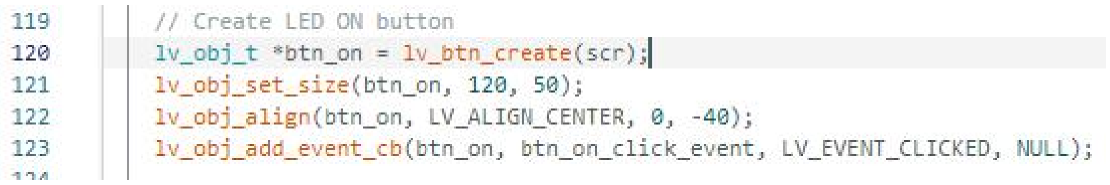
```
/* Create LED control UI */
static void create_led_control_ui(void)
{
    // Lock LVGL: ensure thread-safe operations
    if (lvgl_port_lock(-1) != true) {  // 0 means non-blocking wait for the lock (timeout = 0)
        MAIN_ERROR("LVGL lock failed");  // Print error if lock fails
        return;  // Exit function
    }
```

Subsequently, "lv_scr_act()" is called to get the current active screen object "scr", and "lv_obj_set_style_bg_color()" is used to set the screen background color to white. Then a label object "label" is created as the interface title, which is added to the screen via "lv_label_create(scr)", and "lv_label_set_text()" is used to set the text content to "LED Controller". In addition, "lv_obj_align()" is used to align it to the center of the top of the screen, and "lv_obj_set_style_text_font()" is used to set the font to "lv_font_montserrat_24" for a larger display effect.



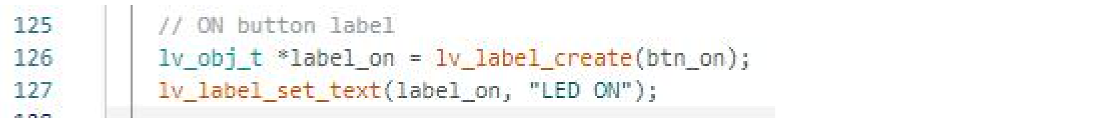
```
    // Create main screen
    lv_obj_t *scr = lv_scr_act();
    lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFFFF), LV_PART_MAIN);  // Set white background

    // Create title label
    lv_obj_t *label = lv_label_create(scr);
    lv_label_set_text(label, "LED Controller");
    lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 50);
    // Font size
    lv_obj_set_style_text_font(label, &lv_font_montserrat_24, 0);
```

Afterwards, the program creates the first button "btn_on", sets its size to 120×50, and places it at a position slightly above the center of the screen. Then "lv_obj_add_event_cb()" is used to bind the button click event "LV_EVENT_CLICKED" to the callback function "btn_on_click_event", so that the LED lighting operation will be executed when the user clicks this button.



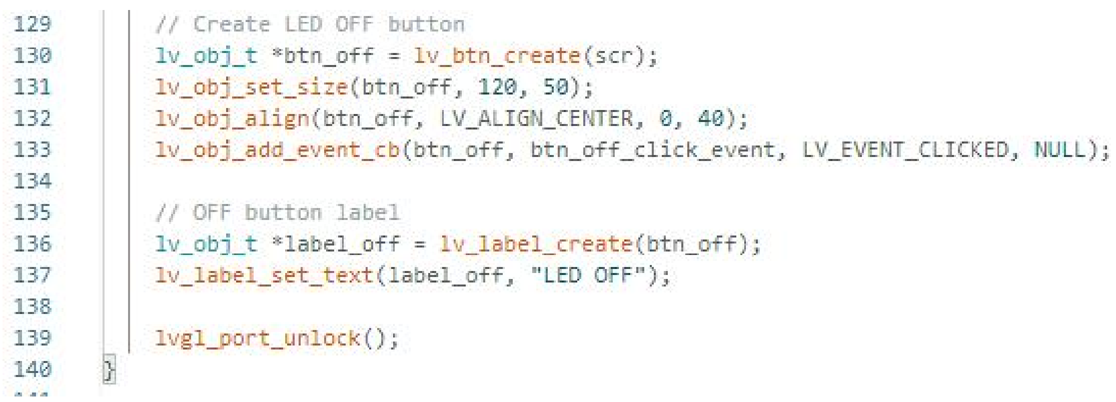
```
    // Create LED ON button
    lv_obj_t *btn_on = lv_btn_create(scr);
    lv_obj_set_size(btn_on, 120, 50);
    lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
    lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);
```

A label object "label_on" is then created inside the button to display the button text "LED ON".



```
    // ON button label
    lv_obj_t *label_on = lv_label_create(btn_on);
    lv_label_set_text(label_on, "LED ON");
```

Next, the same method is used to create the second button "btn_off", which is placed slightly below the center of the screen. Its click event is bound to "btn_off_click_event" to turn off the LED, and a text label "LED OFF" is added inside the button. Finally, "lvgl_port_unlock()" is called to release the previously acquired LVGL lock, allowing other tasks to access LVGL continuously.

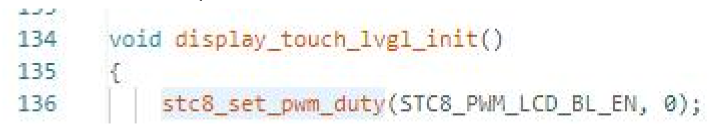


```
    // Create LED OFF button
    lv_obj_t *btn_off = lv_btn_create(scr);
    lv_obj_set_size(btn_off, 120, 50);
    lv_obj_align(btn_off, LV_ALIGN_CENTER, 0, 40);
    lv_obj_add_event_cb(btn_off, btn_off_click_event, LV_EVENT_CLICKED, NULL);

    // OFF button label
    lv_obj_t *label_off = lv_label_create(btn_off);
    lv_label_set_text(label_off, "LED OFF");

    lvgl_port_unlock();
}
```

3. display_touch_lvgl_init

The main function of this display_touch_lvgl_init() function is to initialize the screen backlight, RGB display, touchscreen, and LVGL graphical interface system, thereby enabling the device to have complete graphic display and touch input capabilities.

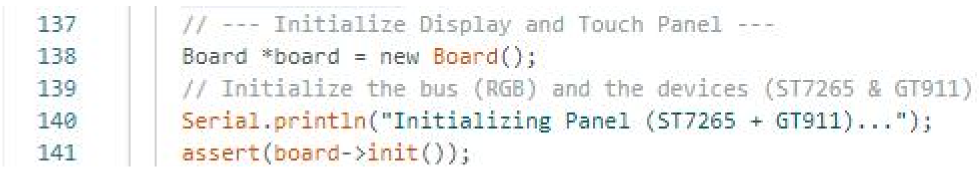
The function begins by calling stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0) to set the LCD backlight PWM duty cycle to 0%, which means turning off the screen backlight first. This step is usually used to prevent screen flickering or abnormal brightness during the display initialization process. The PWM control is handled by the external coprocessor STC8H1KXX.



```
void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);
```

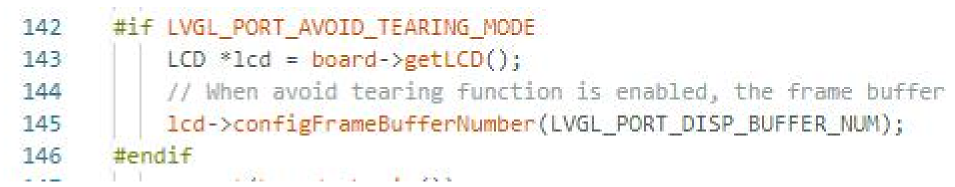
Then the program creates a Board object named "board" using the "ESP32_Display_Panel" library, which is used to uniformly manage the display bus, LCD controller, and touch device. The statement "Initializing Panel (ST7265 + GT911)..." is printed, indicating that the system is initializing the display and touch hardware. The LCD driver chip is ST7265 and the touch controller is GT911.

The program initializes the display bus (RGB interface) and related devices through "board->init()". If the initialization fails, the assert() function will trigger the program to stop.



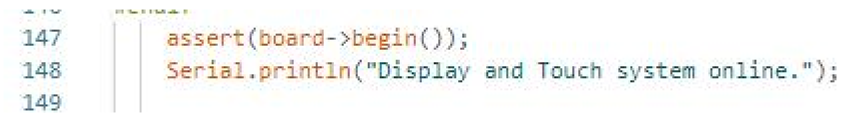
```
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
```

Next, under the conditional compilation #if LVGL_PORT_AVOID_TEARING_MODE, if the tearing avoidance mode is enabled, the program will obtain the LCD object board->getLCD() and call configFrameBufferNumber() to set the number of frame buffers, in order to reduce the problem of image tearing that occurs during screen refresh.



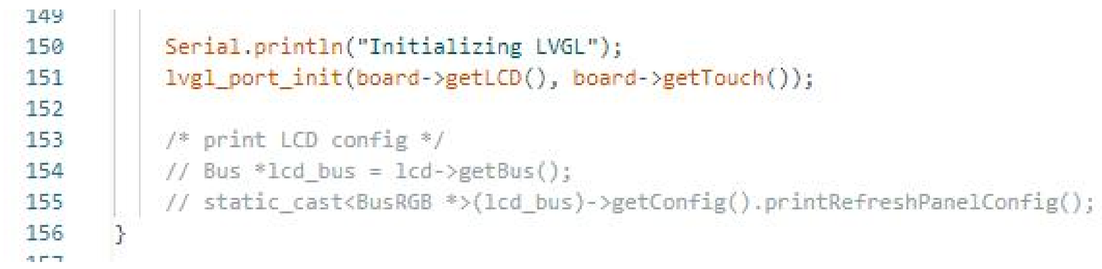
```
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
```

Then, call board->begin() to start the display panel. This process usually involves the LCD startup sequence, driver configuration, and display interface startup. Once it is successful, print "Display and Touch system online." to indicate that the display and touch system is running normally.



```
    assert(board->begin());
    Serial.println("Display and Touch system online.");
```

Finally, the program initializes the LVGL graphics library. By calling lvgl_port_init(board->getLCD(), board->getTouch()), the LCD display device and touch input device are registered in LVGL, enabling LVGL to handle screen drawing and touch event processing. The function also retains a section of commented-out debugging code at the end, which is used to obtain the RGB bus configuration and print the LCD refresh parameters.



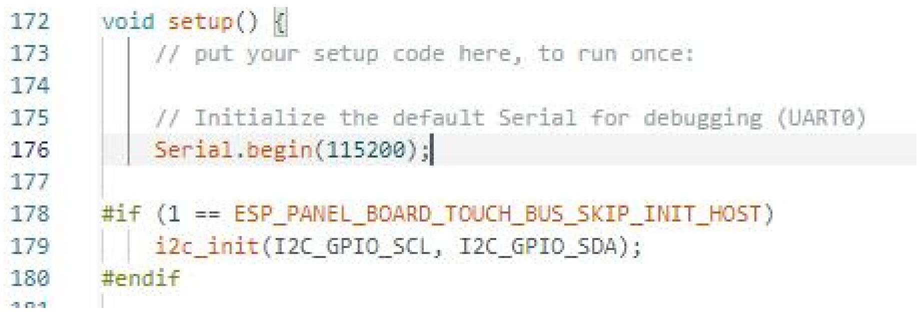
```
    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());

    /* print LCD config */
    // Bus *lcd_bus = lcd->getBus();
    // static_cast<BusRGB *>(lcd_bus)->getConfig().printRefreshPanelConfig();
}
```

4. setup

This setup() function serves as the entry point for the system initialization of the entire program. It is executed only once after the device is powered on or reset. Its main tasks include sequentially completing serial port debugging initialization, display and touchscreen driver initialization, LVGL graphics library startup, and LED control GPIO initialization, and ultimately creating the graphical interface UI.

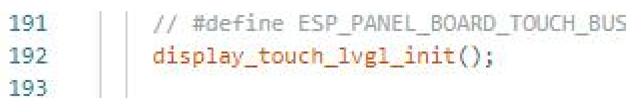
The function first initiates serial communication (UART0) through Serial.begin(115200), setting the baud rate to 115200, which is used to output debugging information;

Then, through conditional compilation #if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST), it determines whether manual initialization of the I²C bus is required. If the macro ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST is set to 1, it calls i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) to initialize the I²C interface using the specified SCL and SDA pins. In this way, the main control ESP32-P4 can communicate with external devices.



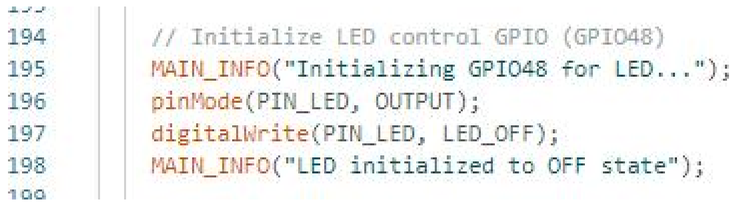
```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif
```

Then, the program calls the function display_touch_lvgl_init() to complete the initialization of the display screen, touch screen, and LVGL graphics system, including the configuration of the RGB screen driver, touch controller, and LVGL display interface;



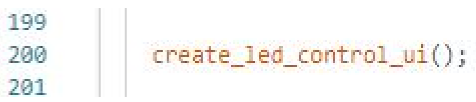
```
    // #define ESP_PANEL_BOARD_TOUCH_BUS
    display_touch_lvgl_init();
```

Then the program initializes the GPIO48 pin used to control the LED: The pin is set to output mode using pinMode(PIN_LED, OUTPUT), and the default state of the LED is set to off by calling digitalWrite(PIN_LED, LED_OFF). At the same time, a log message is output to indicate that the LED has been initialized.



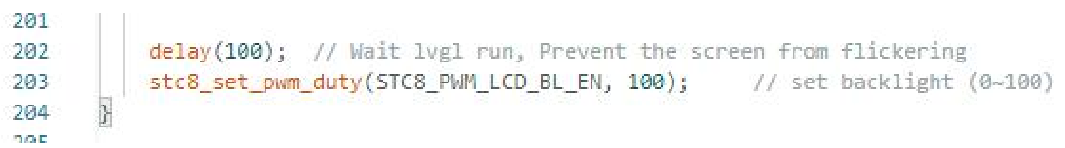
```
    // Initialize LED control GPIO (GPIO48)
    MAIN_INFO("Initializing GPIO48 for LED...");
    pinMode(PIN_LED, OUTPUT);
    digitalWrite(PIN_LED, LED_OFF);
    MAIN_INFO("LED initialized to OFF state");
```

Finally, the function create_led_control_ui() is called to create the graphical interface, which generates the LED control interface (title + ON/OFF buttons) on the screen. This enables users to control the LEDs on the development board through the touchscreen buttons.



```
    create_led_control_ui();
```

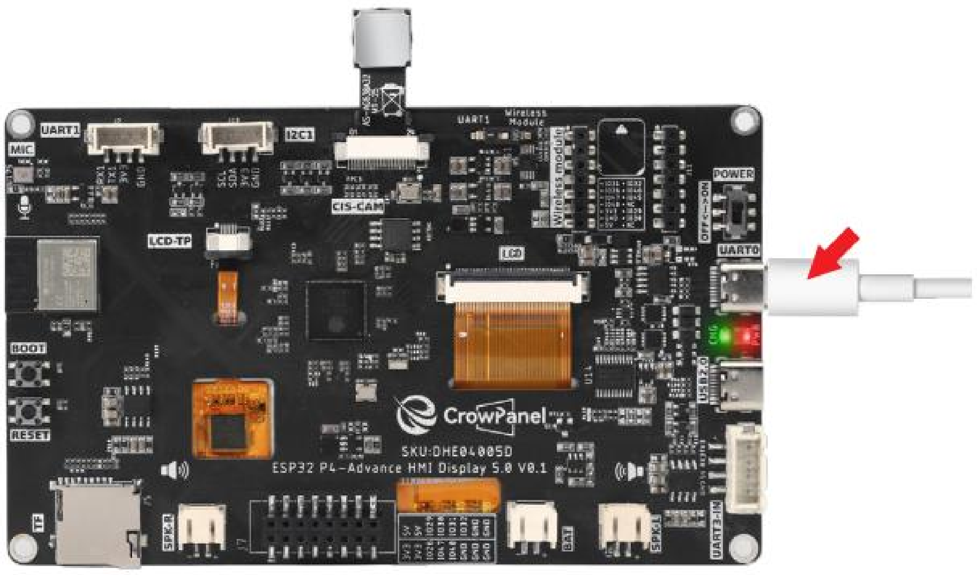
Then, the program executes delay(100) to wait for approximately 100ms, allowing LVGL sufficient time to complete the interface rendering. This is to prevent screen flickering caused by immediately turning on the backlight after the screen initialization is completed. Finally, it calls stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100) to set the LCD backlight PWM duty cycle to 100%. By controlling the backlight brightness through STC8H1KXX, the screen is lit up and the already drawn interface is displayed completely.



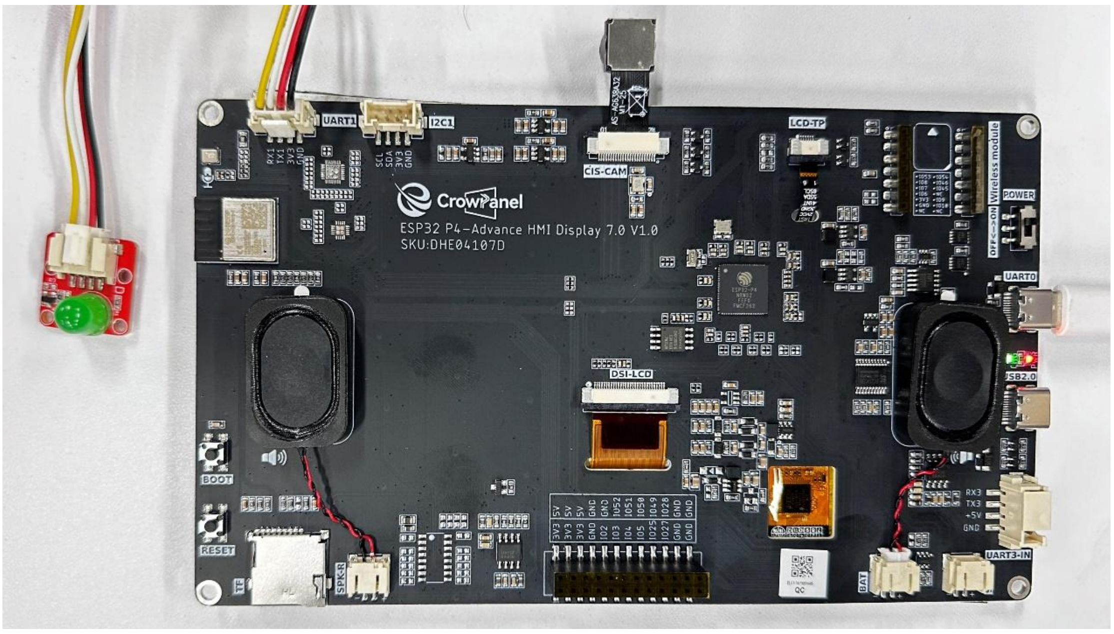
```
    delay(100);  // Wait lvgl run, Prevent the screen from flickering
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);      // set backlight (0~100)
}
```

Programming Steps

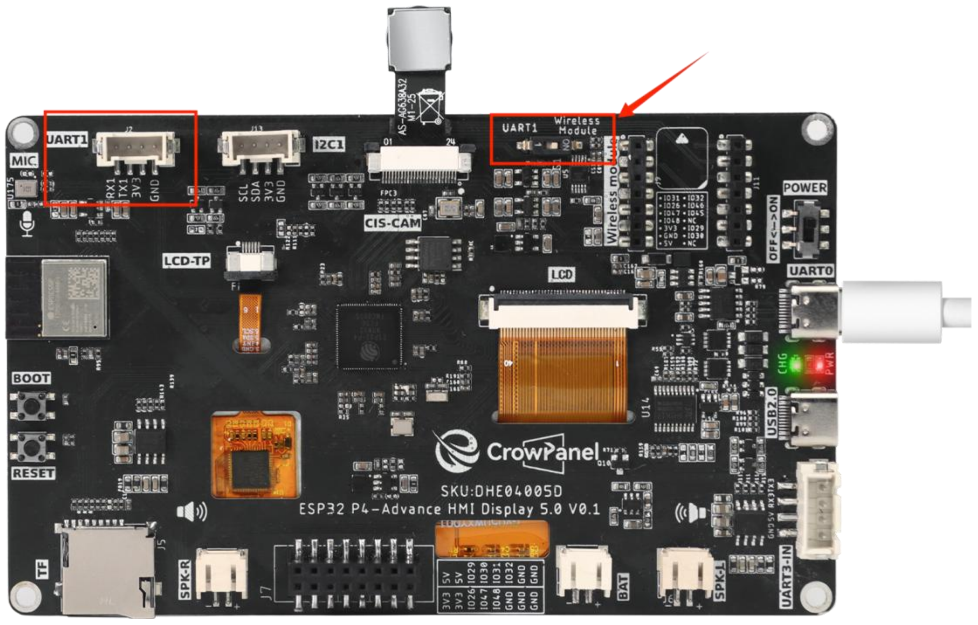
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



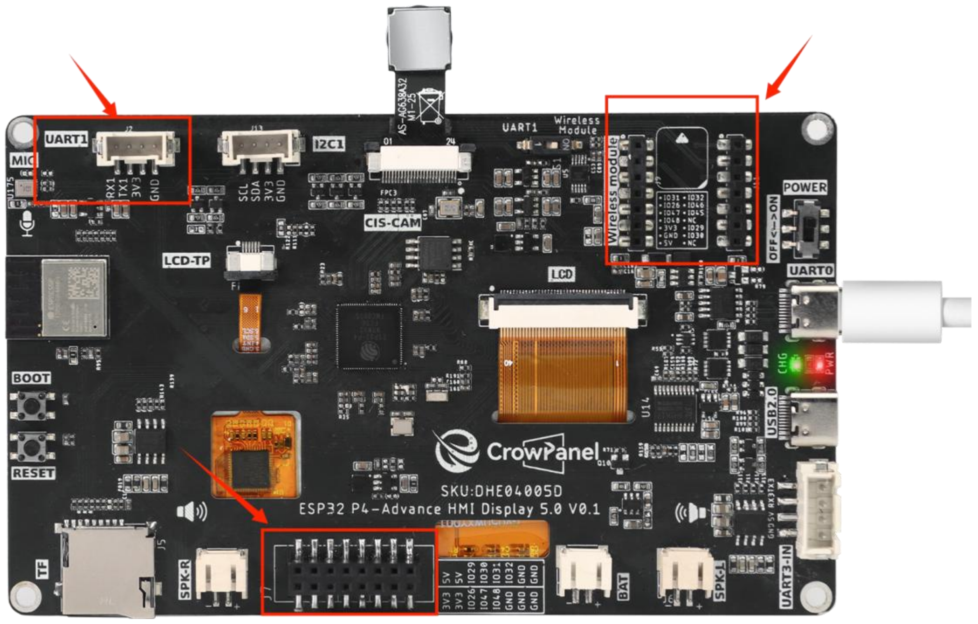
Also, remember to connect an LED to the UART1 interface.



Then, switch the toggle switch on the 5-inch Advance-P4 to the UART1 position. Only in this way can the UART1 interface be used.



This is the design on the hardware side.

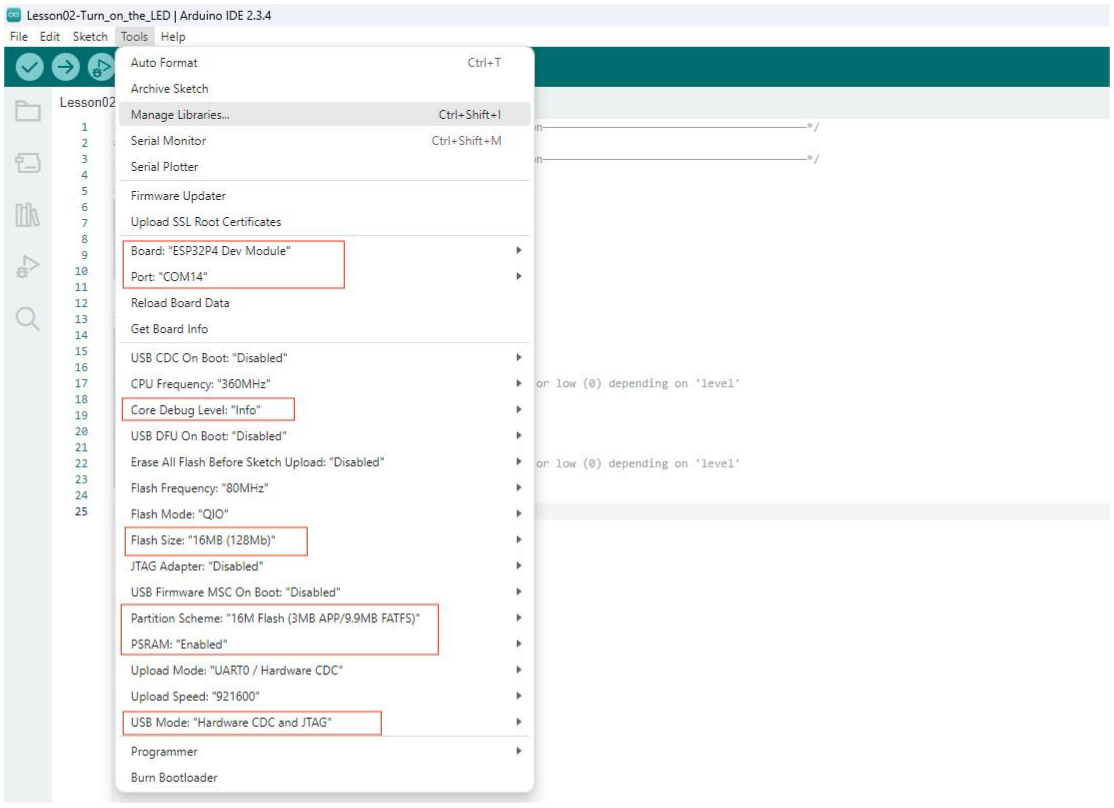


Switch to UART1 port:
Among the three interfaces shown in the figure, only the UART1 interface can be used at this time.
Alternatively, the expansion header at the bottom can also be used.
That is, either the UART1 interface or the expansion header can be used, but not both.
Switch to Wireless Module port:
Among the three interfaces shown in the figure, only the wireless module can be used at this time.
Alternatively, the expansion header at the bottom can also be used.
That is, either the wireless module or the expansion header can be used, but not both.

Summary:
The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.
The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

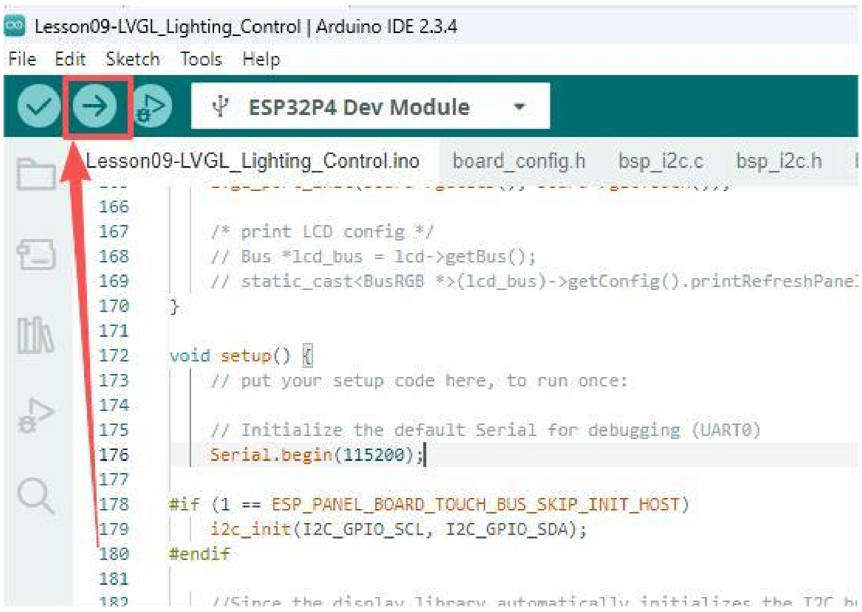
Note: The H2 and C6 wireless modules can be used simultaneously with UART1.
The Lora, 2.4GHz, and WiFi-Halow wireless modules can be used with UART1, but not simultaneously.

Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

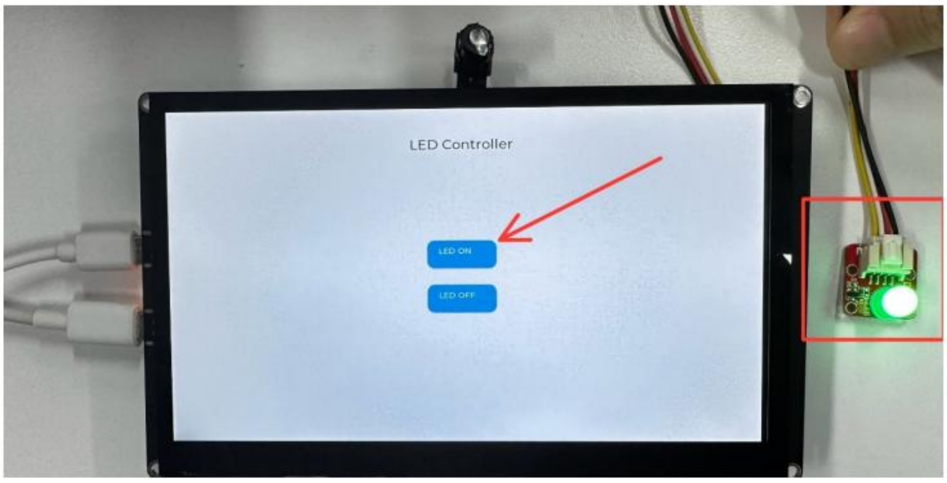
Board: "ESP32P4 Dev Module"
Port: "COM14"
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"

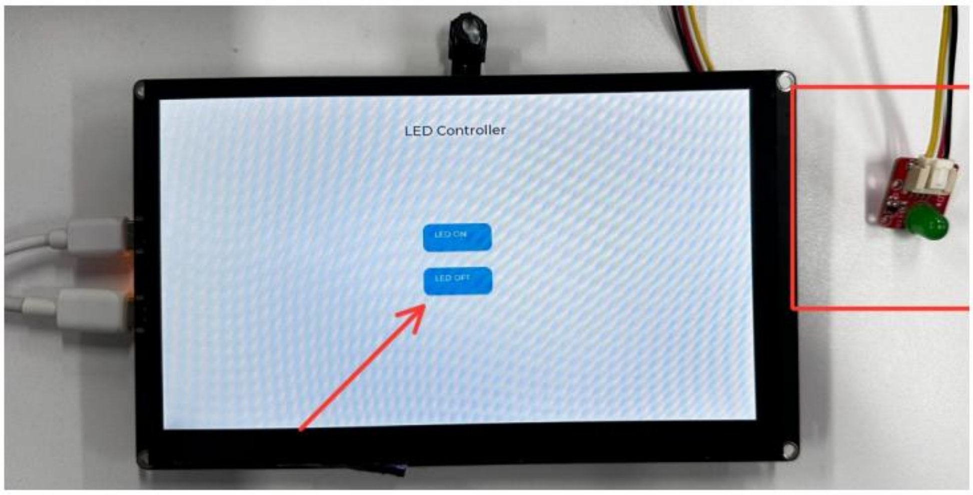
Then we compile and upload the code.


Lesson09-LVGL_Lighting_Control | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson09-LVGL_Lighting_Control.ino
board_config.h
bsp_i2c.c
bsp_i2c.h

```
    /* print LCD config */
    // Bus *lcd_bus = lcd->getBus();
    // static_cast<BusRGB *>(lcd_bus)->getConfig().printRefreshPanel
}

void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif

    //Since the display library automatically initializes the I2C bu
```

After running the code, when you tap the "LED ON" button on the Advance-P4's touchscreen, you will be able to turn on the LED; tapping the "LED OFF" button will allow you to turn off the LED.





Lesson10---Temperature and Humidity

Introduction

In this lesson, we will guide you through using the I2C interface on the Advance-P4 development board. We will connect a temperature and humidity sensor to the I2C interface, then display the acquired sensor values on the screen.

The key learning objective of this lesson is mastering the usage of the I2C interface.

Learning Goals

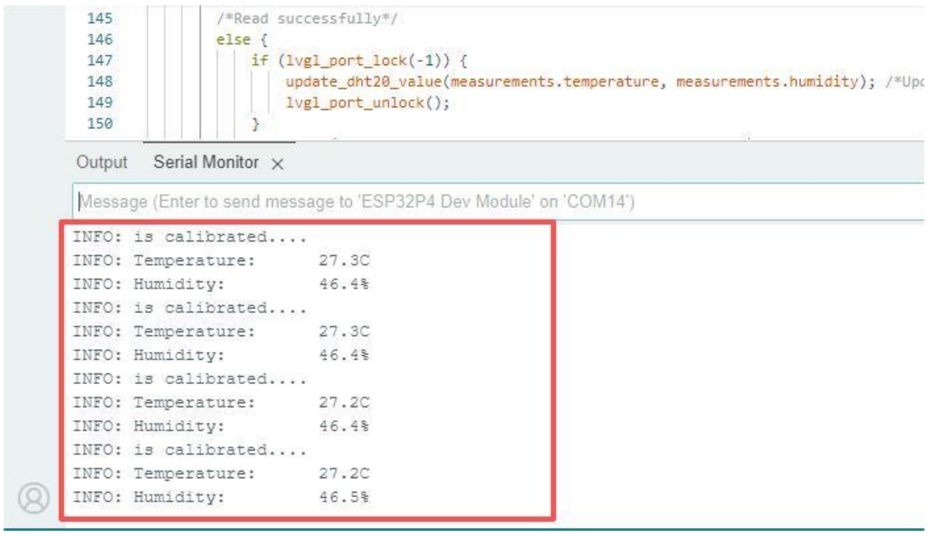
1. Understand the I2C bus communication principles of the ESP32-P4 development board and the operational mechanisms of the DHT20 temperature and humidity sensor (humidity/temperature detection principles).

2. Master the initialization and configuration of the I2C bus, as well as the methods for invoking the DHT20 sensor driver (initialization, calibration detection, data reading, and CRC verification).

3. Master the implementation of dynamic updates for LVGL text labels, completing the end-to-end workflow from sensor data acquisition to real-time on-screen display of temperature and humidity readings.

Preview of the Result

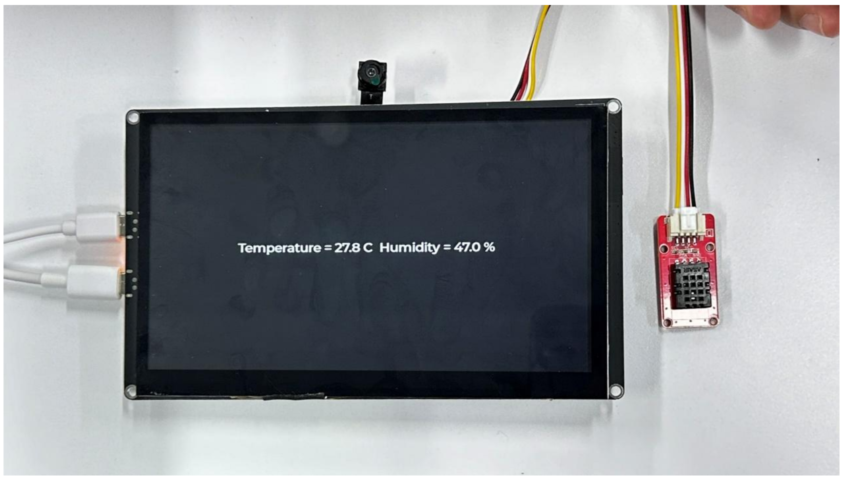
After running the code, you will be able to visually see the real-time temperature and humidity collected by the temperature and humidity sensor displayed on the screen of the Advance-P4.



```
/*Read successfully*/
else {
    if (lvgl_port_lock(-1)) {
        update_dht20_value(measurements.temperature, measurements.humidity); /*Upd
        lvgl_port_unlock();
    }
```

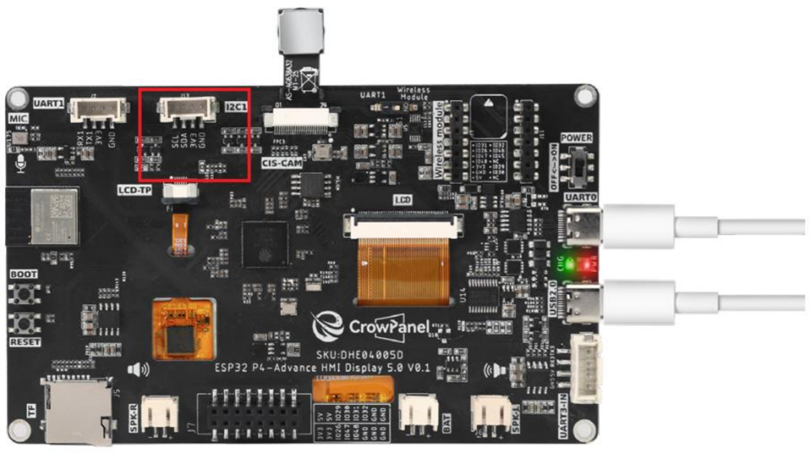
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
INFO: is calibrated....
INFO: Temperature:       27.3C
INFO: Humidity:          46.4%
INFO: is calibrated....
INFO: Temperature:       27.3C
INFO: Humidity:          46.4%
INFO: is calibrated....
INFO: Temperature:       27.2C
INFO: Humidity:          46.4%
INFO: is calibrated....
INFO: Temperature:       27.2C
INFO: Humidity:          46.5%
```

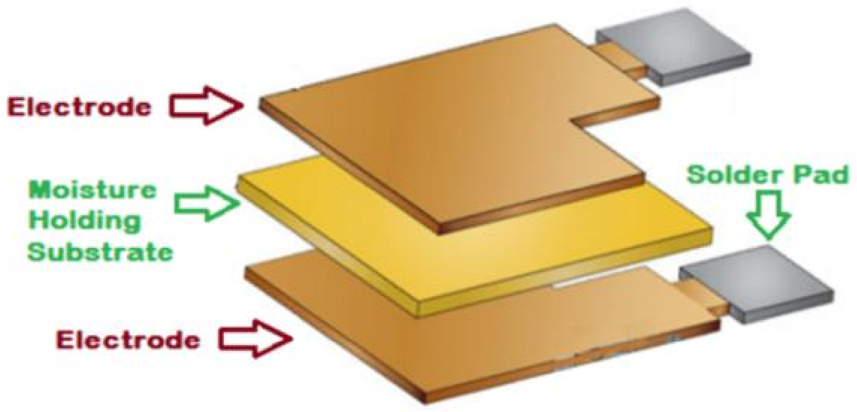


Hardware Used in This Lesson

I2C Interface on the Advance-P4



temperature and humidity sensor Schematic Diagram



In the temperature and humidity sensor, humidity detection relies on hygroscopic materials. These materials absorb or release water in response to changes in environmental humidity, thereby altering their own electrical properties (such as resistance, capacitance, etc.). The sensor obtains humidity information by detecting the changes in the electrical signal between the material and the electrodes. Temperature detection typically uses thermal-sensitive elements (such as thermistors). When the temperature changes, the resistance value of the thermal-sensitive element changes. The sensor measures this resistance change and converts it to obtain the temperature value. Finally, it combines the data from both to determine the temperature and humidity conditions.

If you want to purchase, you can click the link below to learn more about this module.

Purchase link: https://www.elecrow.com/crowtail-dht20.html

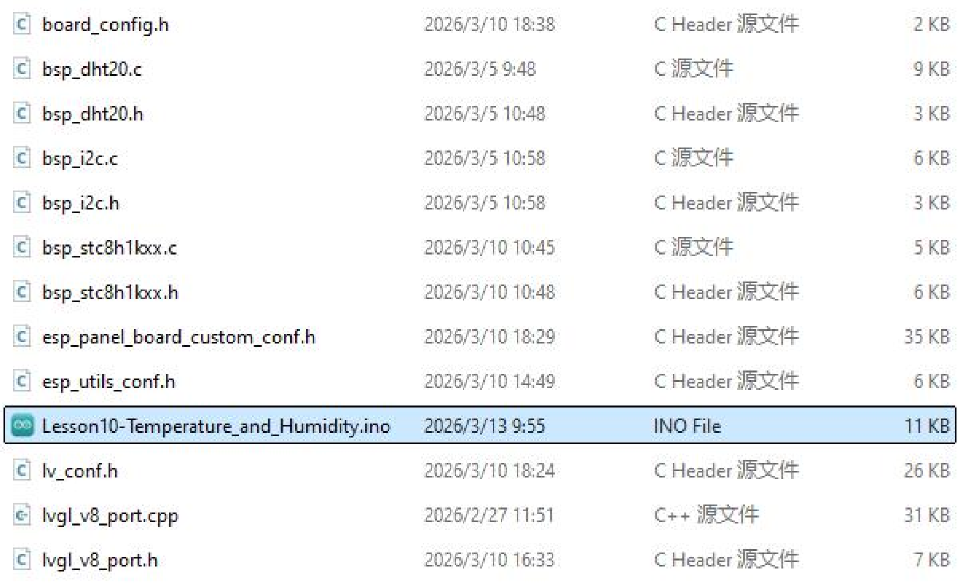
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson10-Temperature_and_Humidity

Key Explanations

How do we use the I2C protocol? And how do we read data from a temperature and humidity sensor via the I2C interface? Let's dive into the code implementation together.

Double-click to open the code for this lesson (the .ino file).



board_config.h	2026/3/10 18:38	C Header 源文件	2 KB
bsp_dht20.c	2026/3/5 9:48	C 源文件	9 KB
bsp_dht20.h	2026/3/5 10:48	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/10 10:48	C Header 源文件	6 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
Lesson10-Temperature_and_Humidity.ino	2026/3/13 9:55	INO File	11 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

Once opened, you will see the project code along with the configuration file "board_config.h", as well as other relevant display driver files.

(The meanings and functions of each of these files have been elaborated in detail in Lesson 7. We will not go into further details here. If you have any questions, you can review the previous course content.)

In the preface, we have already imported all the library files into the Arduino's "libraries" folder. If you haven't done so, please follow the instructions in the preface to import all the library files into the "libraries" folder of Arduino.

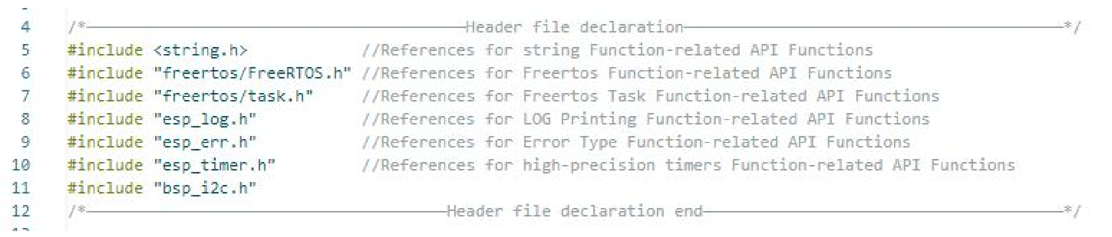
Compared to the framework of the pure display text function in Lesson 7, this project's framework includes additional contents related to the files "bsp_dht20" and "bsp_i2c".

Temperature and Humidity Acquisition Code

The driver code for the temperature and humidity sensor consists of two files: "bsp_dht20.c" and "bsp_dht20.h".
Next, we will first analyze the "bsp_dht20.h" program.
"bsp_dht20.h" is the header file for the temperature and humidity acquisition module, and its main purposes are:

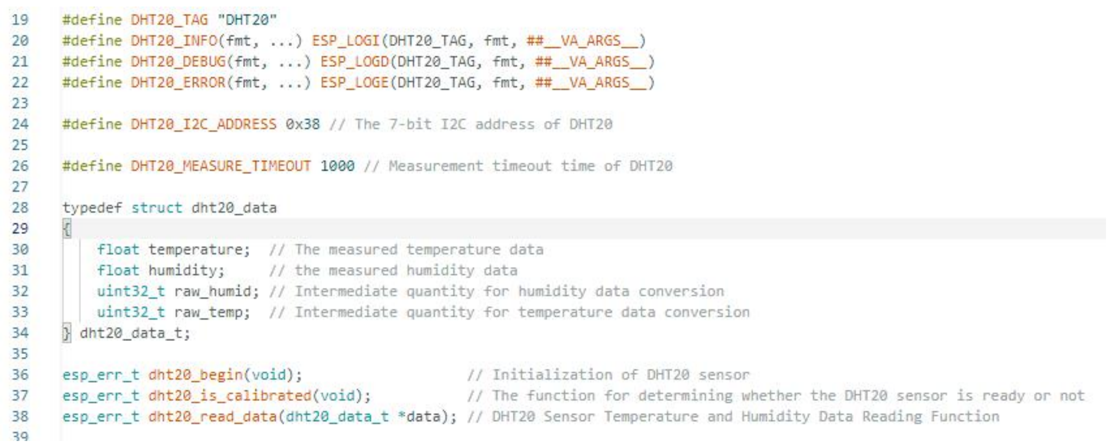
- To declare the functions, macros, and variables implemented in "bsp_dht20.c" for use by external programs. This allows other .c files to call functions from this module simply by adding #include "bsp_dht20.h".
- In other words, it acts as an interface layer—it exposes which functions and constants are available for external use while hiding the internal implementation details of the module.

In this component, all the libraries we need to use are included in the "bsp_dht20.h" file, enabling unified management.



```
/*————————————————————————————————————————Header file declaration————————————————————————————————————————*/
#include <string.h>             //References for string Function-related API Functions
#include "freertos/FreeRTOS.h" //References for Freertos Function-related API Functions
#include "freertos/task.h"      //References for Freertos Task Function-related API Functions
#include "esp_log.h"            //References for LOG Printing Function-related API Functions
#include "esp_err.h"            //References for Error Type Function-related API Functions
#include "esp_timer.h"          //References for high-precision timers Function-related API Functions
#include "bsp_i2c.h"
/*————————————————————————————————————————Header file declaration end————————————————————————————————————*/
```

Then, we declare the variables we need to use, as well as the functions—whose specific implementations are in "bsp_dht20.c".
Centralizing these declarations in "bsp_dht20.h" is for the convenience of calling and management. (We will understand their roles when they are used in "bsp_dht20.c".)



```c
#define DHT20_TAG "DHT20"
#define DHT20_INFO(fmt, ...) ESP_LOGI(DHT20_TAG, fmt, ##__VA_ARGS__)
#define DHT20_DEBUG(fmt, ...) ESP_LOGD(DHT20_TAG, fmt, ##__VA_ARGS__)
#define DHT20_ERROR(fmt, ...) ESP_LOGE(DHT20_TAG, fmt, ##__VA_ARGS__)

#define DHT20_I2C_ADDRESS 0x38 // The 7-bit I2C address of DHT20

#define DHT20_MEASURE_TIMEOUT 1000 // Measurement timeout time of DHT20

typedef struct dht20_data
{
    float temperature;  // The measured temperature data
    float humidity;     // the measured humidity data
    uint32_t raw_humid; // Intermediate quantity for humidity data conversion
    uint32_t raw_temp;  // Intermediate quantity for temperature data conversion
} dht20_data_t;

esp_err_t dht20_begin(void);                 // Initialization of DHT20 sensor
esp_err_t dht20_is_calibrated(void);         // The function for determining whether the DHT20 sensor is ready or not
esp_err_t dht20_read_data(dht20_data_t *data); // DHT20 Sensor Temperature and Humidity Data Reading Function
```

Let's now examine the specific functions of each function in "bsp_dht20.c".

The bsp_dht20 component is primarily used to communicate with the DHT20 temperature and humidity sensor via the I2C bus. It implements functions such as sensor initialization, status detection, data reading, and verification to obtain environmental temperature and humidity data.

Then the following functions are the interfaces we call to initialize the temperature and humidity sensor and obtain its readings.

- The 'print_binary' function: Its role is to convert a 16-bit integer 'value' into a corresponding binary string. It can be used in scenarios where data needs to be visually displayed in binary form, such as checking register values or the binary composition of sensor data.
- The 'print_byte' function: This function splits an 8-bit byte 'byte' into high 4 bits and low 4 bits, then converts them into a binary string prefixed with '0b' to make the data more readable. It is useful when debugging I2C communication data that requires formatted printing of single-byte data, such as status bytes or data bytes returned by the sensor.
- The 'dht20_reset_register' function: Its main function is to reset a specified register. The specific operation is to first read the current value of the register, then rewrite it according to the requirements of the DHT20 protocol. It can be used when sensor initialization fails or the status is abnormal, requiring resetting of key registers (such as calibration or configuration registers like '0x1B', '0x1C', '0x1E') to restore the sensor to normal working condition.
- The dht20_status function: Sends the 0x71 command via I2C and reads the value of DHT20's status register to obtain the sensor's current working status, such as whether calibration is completed or a measurement is in progress. It is used to check if the sensor status is normal before initialization, confirm if the sensor is ready before measurement, or troubleshoot to identify the cause of abnormal sensor status.
- The dht20_reset_sensor function: Continuously detects the sensor's status. If the status does not meet expectations (status value does not match 0x18, where

0x18 typically indicates calibration completion and readiness), it repeatedly resets key registers until the status is normal or the retry limit of 255 times is reached. It is used during sensor initialization (e.g., called in dht20_begin) to ensure the sensor enters a working state, or to attempt recovery after sensor communication anomalies.

- The dht20_begin function: Initializes the DHT20 sensor through a process that registers the sensor's device address via I2C to obtain a handle, then calls dht20_reset_sensor to check and reset the sensor. It returns an error code if initialization fails. This function must be called during system startup or before the first use of the sensor; otherwise, subsequent data reading may fail.
- The dht20_is_calibrated function: Checks if the sensor has completed calibration by determining whether a specific bit in the status register is 0x18—calibration completion is a prerequisite for the sensor's normal operation. It is used to confirm sensor readiness after initialization, verify normal sensor status before measurement, and avoid reading invalid data.
- The dht20_crc8 function: Calculates the checksum of data using the CRC8 algorithm specified in the DHT20 protocol (polynomial 0x31) to verify the integrity of received data. It is used after reading sensor data (e.g., in dht20_read_data) to compare the calculated CRC value with the CRC byte returned by the sensor, determining if errors occurred during data transmission.
- The dht20_read_data function: Fully implements the temperature and humidity data reading process, including sending measurement commands (0xAC, 0x33, 0x00), waiting for the sensor to complete measurement (with timeout detection), reading 7 bytes of data (including status, humidity, temperature, and CRC), and parsing raw data into actual temperature and humidity values (humidity in percentage, temperature in Celsius) after CRC verification. This core function of the component is called when environmental temperature and humidity need to be obtained, but it requires the sensor to be initialized and calibrated beforehand (confirmed via dht20_begin and dht20_is_calibrated).

That concludes our introduction to the bsp_dht20 component—you only need to understand how to call these interfaces.

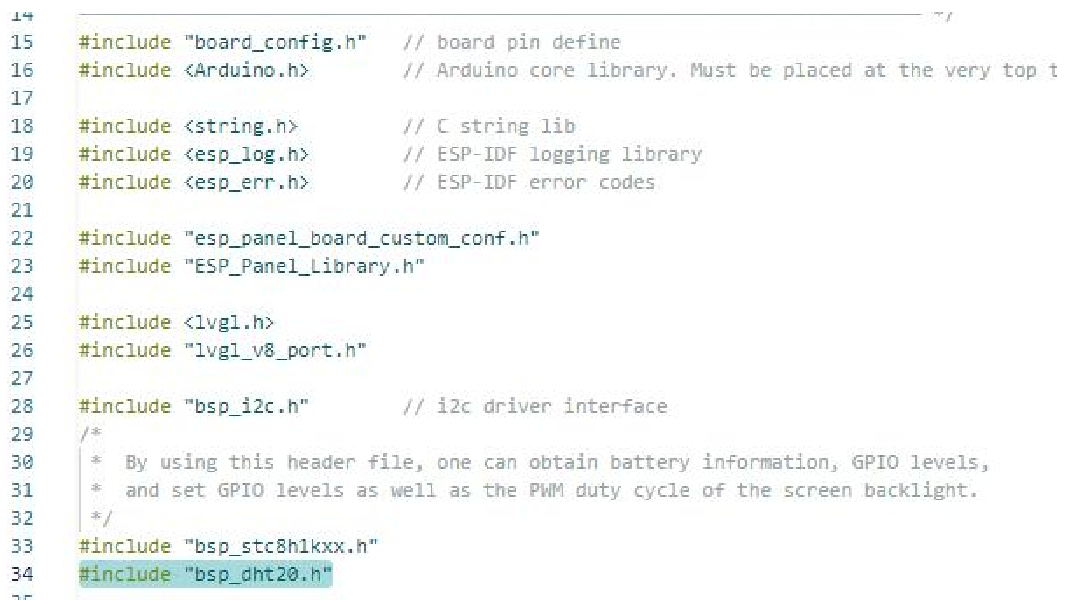
(The purpose and function of each file here have been covered in detail in Lesson 7, so we won't elaborate further. Should you have any questions, please refer back to the previous course content.)

Next, let's focus our attention on the ".ino" main code file to explore how real-time temperature and humidity values are displayed on the screen.

1. Header Files

The other library files were thoroughly explained in Lesson 7.

These two lines utilize the "#include" preprocessor directive to incorporate two hardware driver-related header files: "bsp_i2c.h" and "bsp_dht20.h". Their purpose is to provide the program with the I2C bus communication interface, as well as the driver functions and data structure definitions for the DHT20 temperature and humidity sensor.



```
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top t

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

#include <lvgl.h>
#include "lvgl_v8_port.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"
#include "bsp_dht20.h"
```

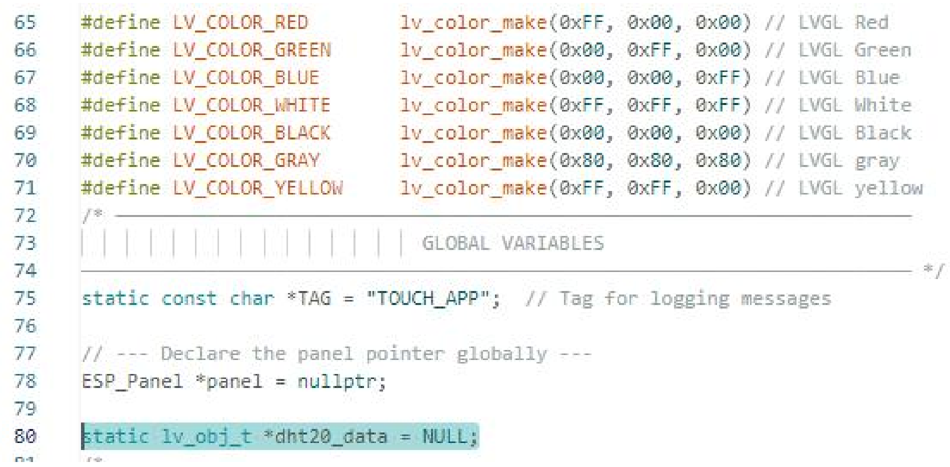
Specifically, bsp_i2c.h serves as a low-level I2C bus driver module that encapsulates I2C initialization functions (such as i2c_init()), read/write functions, and ESP32 I2C controller-related configurations. This enables developers to rapidly establish I2C communication channels by specifying SDA and SCL pins.

Meanwhile, bsp_dht20.h is a DHT20 sensor driver interface file implemented atop the I2C bus. It internally invokes the I2C driver to communicate with the DHT20 chip and provides a suite of operational functions: dht20_begin() for sensor initialization, dht20_is_calibrated() for checking calibration completion status, dht20_read_data() for retrieving temperature and humidity data, along with data structure definitions (such as dht20_data_t, comprising temperature and humidity members) for data storage.

2. Global Variables

The other macro definitions and global variables were thoroughly covered in Lesson 7.

The line static lv_obj_t *dht20_data = NULL; declares a static LVGL object pointer variable used to hold the text label object that displays DHT20 temperature and humidity data on the screen.



```
#define LV_COLOR_RED        lv_color_make(0xFF, 0x00, 0x00) // LVGL Red
#define LV_COLOR_GREEN      lv_color_make(0x00, 0xFF, 0x00) // LVGL Green
#define LV_COLOR_BLUE       lv_color_make(0x00, 0x00, 0xFF) // LVGL Blue
#define LV_COLOR_WHITE      lv_color_make(0xFF, 0xFF, 0xFF) // LVGL White
#define LV_COLOR_BLACK      lv_color_make(0x00, 0x00, 0x00) // LVGL Black
#define LV_COLOR_GRAY       lv_color_make(0x80, 0x80, 0x80) // LVGL gray
#define LV_COLOR_YELLOW     lv_color_make(0xFF, 0xFF, 0x00) // LVGL yellow
/* -------------------------------------------------------------------------
 |  |  |  |  |  |  |  |  |  |  |  |  |  |  GLOBAL VARIABLES
 ---------------------------------------------------------------------------- */
static const char *TAG = "TOUCH_APP";  // Tag for logging messages

// --- Declare the panel pointer globally ---
ESP_Panel *panel = nullptr;

static lv_obj_t *dht20_data = NULL;
```

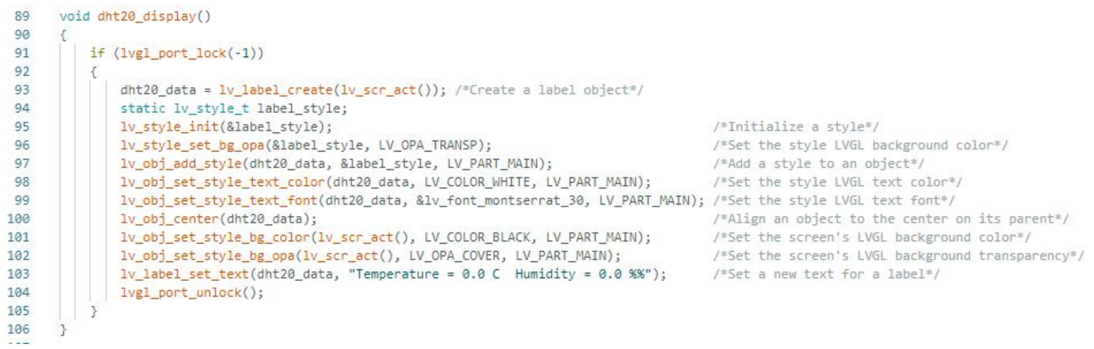
Here, lv_obj_t represents the base object type for all UI components in the LVGL graphics library (such as buttons, labels, images, etc.), making lv_obj_t * a pointer to an LVGL interface object. The variable name dht20_data indicates that this object is designated for displaying temperature and humidity information from the DHT20 sensor.

The static keyword restricts the variable's scope to within the current source file (preventing direct access from other .cpp files), while ensuring it is instantiated only once during program execution and persists in memory throughout runtime—making it well-suited for storing UI control objects.

Finally, = NULL signifies that the pointer initially references no object at program startup, meaning the corresponding LVGL label component has yet to be created. When lv_label_create() is subsequently executed to instantiate the text label, the returned object address is assigned to dht20_data. The program can then invoke functions such as lv_label_set_text() through this pointer to dynamically update the temperature and humidity data displayed on screen.

3. dht20_display():

This code defines a function dht20_display() whose purpose is to create a text interface on screen for displaying temperature and humidity data, while configuring text styling and screen background settings.



```
void dht20_display()
{
    if (lvgl_port_lock(-1))
    {
        dht20_data = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t label_style;
        lv_style_init(&label_style);                                            /*Initialize a style*/
        lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);                       /*Set the style LVGL background color*/
        lv_obj_add_style(dht20_data, &label_style, LV_PART_MAIN);               /*Add a style to an object*/
        lv_obj_set_style_text_color(dht20_data, LV_COLOR_WHITE, LV_PART_MAIN);  /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(dht20_data, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_center(dht20_data);                                              /*Align an object to the center on its parent*/
        lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN);  /*Set the screen's LVGL background color*/
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN);      /*Set the screen's LVGL background transparency*/
        lv_label_set_text(dht20_data, "Temperature = 0.0 C  Humidity = 0.0 %%");  /*Set a new text for a label*/
        lvgl_port_unlock();
    }
}
```

The function first invokes "lvgl_port_lock(-1)" to acquire the LVGL mutex lock, ensuring thread-safe operations on LVGL graphic objects in a multi-tasking environment. Upon successfully obtaining the lock, it proceeds to create interface elements.

The program creates a label object on the currently active screen via "lv_label_create(lv_scr_act())", storing the returned object pointer in the global variable "dht20_data". This label will subsequently be used to display temperature and humidity data from the DHT20 sensor.

Next, the code declares and initializes an "lv_style_t" style object named "label_style", sets its background opacity to transparent using "lv_style_set_bg_opa()", and applies this style to the label object.

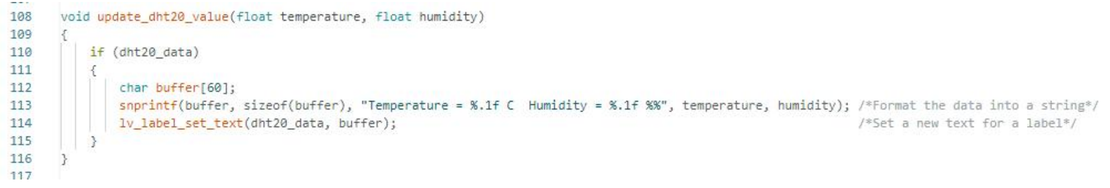
It then configures the text color to white through "lv_obj_set_style_text_color()", and sets the font to "lv_font_montserrat_30" via "lv_obj_set_style_text_font()" for clear on-screen visibility.

Afterwards, "lv_obj_center()" is employed to position the label at the screen's center, while "lv_obj_set_style_bg_color()" and "lv_obj_set_style_bg_opa()" are used to set the entire screen background to solid black — achieving a high-contrast black-background-with-white-text display effect.

Finally, "lv_label_set_text()" sets the initial display content to "Temperature = 0.0 C Humidity = 0.0 %" as placeholder text for the temperature and humidity data, followed by "lvgl_port_unlock()" to release the previously acquired LVGL lock.

4. void update_dht20_value(float temperature, float humidity):

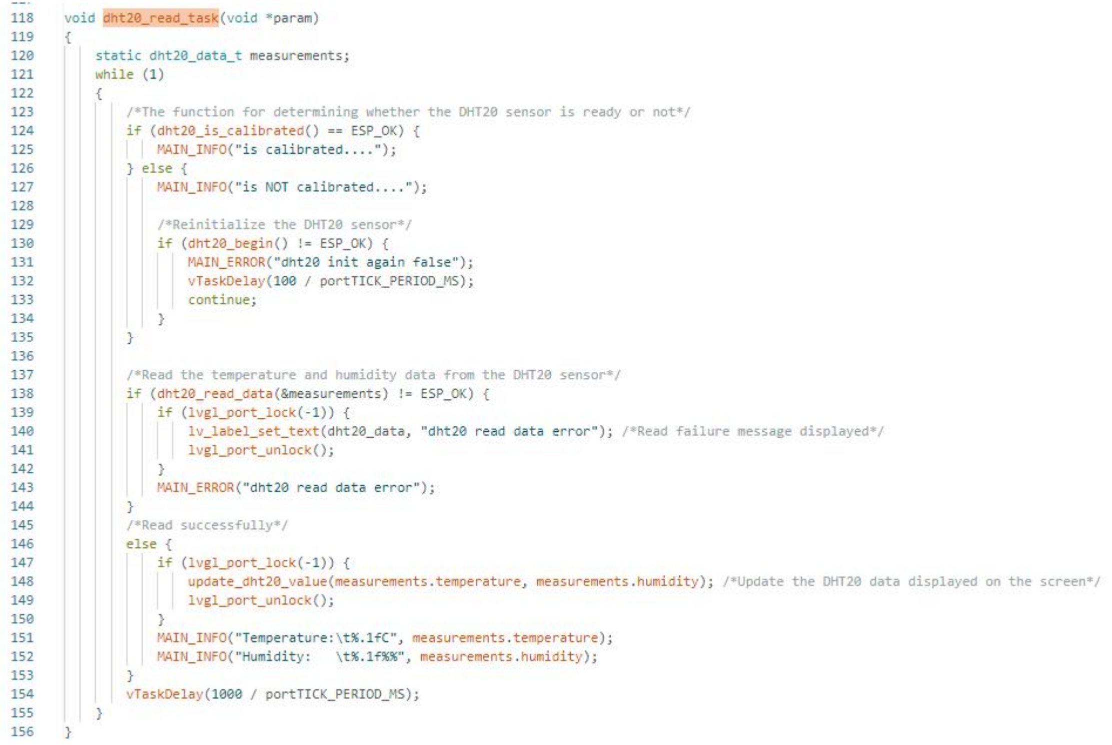
This function is used to update the display content of temperature and humidity data on the LVGL interface:
First, it checks whether the temperature and humidity display label dht20_data is valid. If valid, it uses snprintf to format the incoming temperature (temperature) and humidity (humidity) values into a string in the format of "Temperature = X.X C Humidity = X.X %". Then, it calls the LVGL interface lv_label_set_text to update the formatted string to the label, realizing real-time refresh of data on the screen.



```
void update_dht20_value(float temperature, float humidity)
{
    if (dht20_data)
    {
        char buffer[60];
        snprintf(buffer, sizeof(buffer), "Temperature = %.1f C  Humidity = %.1f %%", temperature, humidity); /*Format the data into a string*/
        lv_label_set_text(dht20_data, buffer);                                                             /*Set a new text for a label*/
    }
}
```

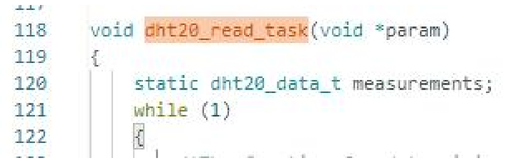
5. void dht20_read_task(void *param):

This code defines a function "dht20_read_task(void *param)", whose purpose is to periodically read data from the temperature and humidity sensor and update the results to both the screen display and serial port logs.



```
void dht20_read_task(void *param)
{
    static dht20_data_t measurements;
    while (1)
    {
        /*The function for determining whether the DHT20 sensor is ready or not*/
        if (dht20_is_calibrated() == ESP_OK) {
            MAIN_INFO("is calibrated....");
        } else {
            MAIN_INFO("is NOT calibrated....");

            /*Reinitialize the DHT20 sensor*/
            if (dht20_begin() != ESP_OK) {
                MAIN_ERROR("dht20 init again false");
                vTaskDelay(100 / portTICK_PERIOD_MS);
                continue;
            }
        }

        /*Read the temperature and humidity data from the DHT20 sensor*/
        if (dht20_read_data(&measurements) != ESP_OK) {
            if (lvgl_port_lock(-1)) {
                lv_label_set_text(dht20_data, "dht20 read data error"); /*Read failure message displayed*/
                lvgl_port_unlock();
            }
            MAIN_ERROR("dht20 read data error");
        }
        /*Read successfully*/
        else {
            if (lvgl_port_lock(-1)) {
                update_dht20_value(measurements.temperature, measurements.humidity); /*Update the DHT20 data displayed on the screen*/
                lvgl_port_unlock();
            }
            MAIN_INFO("Temperature:\t%.1fC", measurements.temperature);
            MAIN_INFO("Humidity:   \t%.1f%%", measurements.humidity);
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

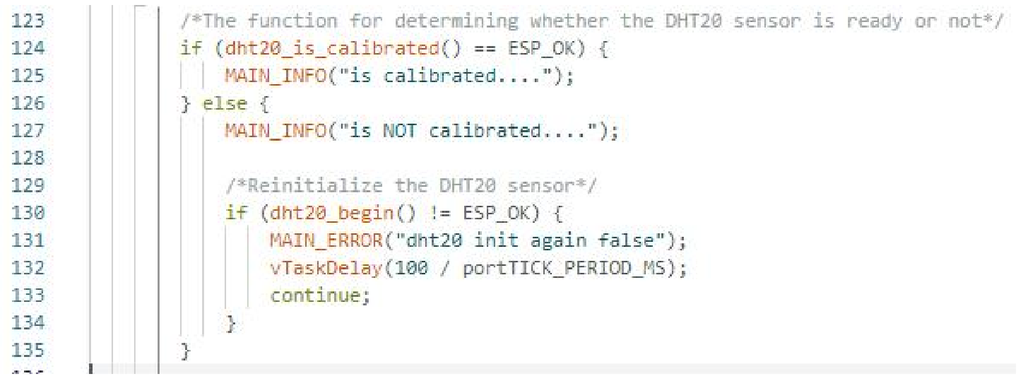
Within the function, a static structure variable "measurements" is first defined to store the temperature and humidity data read from the DHT20 sensor. The program then enters a "while(1)" infinite loop to continuously execute data acquisition tasks.



```
void dht20_read_task(void *param)
{
    static dht20_data_t measurements;
    while (1)
    {
```

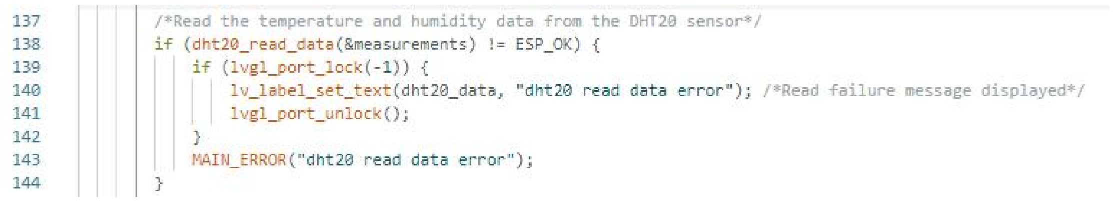
At the beginning of each loop iteration, the program first invokes "dht20_is_calibrated()" to verify whether the sensor has completed calibration. If it returns "ESP_OK", indicating normal sensor status, a log message "Calibrated" is printed.

If the sensor is not calibrated, a prompt message is output and "dht20_begin()" is called to reinitialize the sensor. Should the initialization fail, an error log is printed, followed by a 100ms delay before proceeding to the next loop iteration.



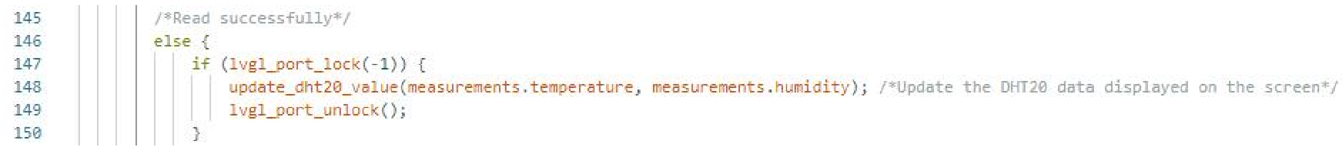
```
        /*The function for determining whether the DHT20 sensor is ready or not*/
        if (dht20_is_calibrated() == ESP_OK) {
            MAIN_INFO("is calibrated....");
        } else {
            MAIN_INFO("is NOT calibrated....");

            /*Reinitialize the DHT20 sensor*/
            if (dht20_begin() != ESP_OK) {
                MAIN_ERROR("dht20 init again false");
                vTaskDelay(100 / portTICK_PERIOD_MS);
                continue;
            }
        }
```

The program then retrieves temperature and humidity data from the sensor via "dht20_read_data(&measurements)". If the read operation fails, it first acquires the LVGL thread lock through "lvgl_port_lock(-1)", then invokes "lv_label_set_text()" to update the on-screen label content to "dht20 read data error" as a failure notification. Finally, it releases the LVGL lock and outputs an error log.



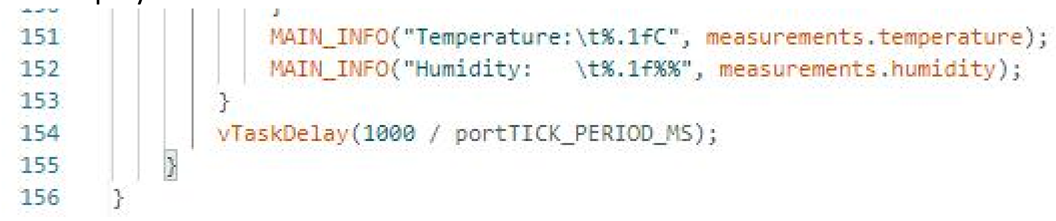
```c
            /*Read the temperature and humidity data from the DHT20 sensor*/
            if (dht20_read_data(&measurements) != ESP_OK) {
                if (lvgl_port_lock(-1)) {
                    lv_label_set_text(dht20_data, "dht20 read data error"); /*Read failure message displayed*/
                    lvgl_port_unlock();
                }
                MAIN_ERROR("dht20 read data error");
            }
```

If the read operation succeeds, the program similarly acquires the LVGL lock first, then invokes the "update_dht20_value()" function to update the screen's text label with the retrieved "measurements.temperature" and "measurements.humidity" values, before releasing the lock.



```c
            /*Read successfully*/
            else {
                if (lvgl_port_lock(-1)) {
                    update_dht20_value(measurements.temperature, measurements.humidity); /*Update the DHT20 data displayed on the screen*/
                    lvgl_port_unlock();
                }
```

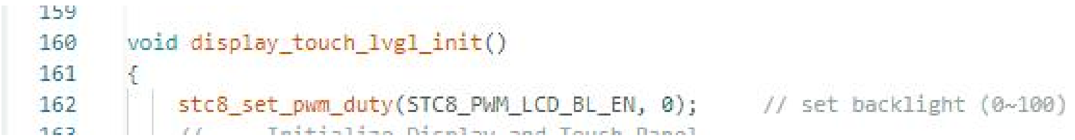
Simultaneously, the program outputs the current temperature and humidity as formatted strings to the serial port via "MAIN_INFO()"—for instance, "Temperature: 25.3C" and "Humidity: 60.5%"—facilitating debugging and monitoring. Finally, "vTaskDelay(1000 / portTICK_PERIOD_MS)" is invoked to delay the task for 1 second, thereby achieving a 1-second interval for temperature and humidity data acquisition and display refresh.



```c
                    MAIN_INFO("Temperature:\t%.1fC", measurements.temperature);
                    MAIN_INFO("Humidity:   \t%.1f%%", measurements.humidity);
                }
                vTaskDelay(1000 / portTICK_PERIOD_MS);
            }
    }
```

6. display_touch_lvgl_init

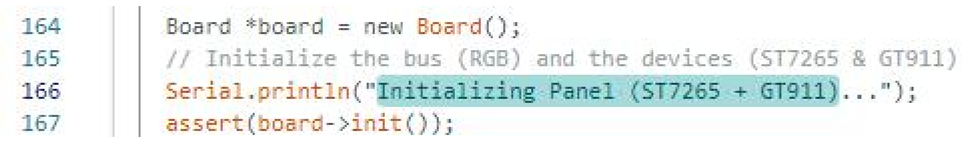
This code defines a function named "display_touch_lvgl_init()", whose main function is to complete the overall initialization of the display, touchscreen and graphics interface library, enabling the system to display the graphical interface normally and respond to touch operations.
The function begins by calling stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0), which sets the LCD backlight PWM duty cycle to 0%, that is, it first turns off the backlight. This step is usually used to prevent screen flickering or abnormal brightness during the display initialization process. The PWM control is handled by the external coprocessor STC8H1KXX.



```c
void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);     // set backlight (0~100)
```

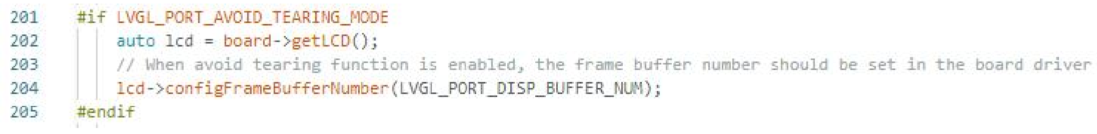
Display the underlying hardware interfaces such as the bus, LCD driver, and touch driver.
Subsequently, the program outputs debugging information "Initializing Panel (ST7265 + GT911)." and calls board->init() to initialize the display hardware bus and devices.

The assert() function is used to immediately terminate the program in case of initialization failure, thereby preventing the system from continuing to run when the hardware has not been properly initialized.



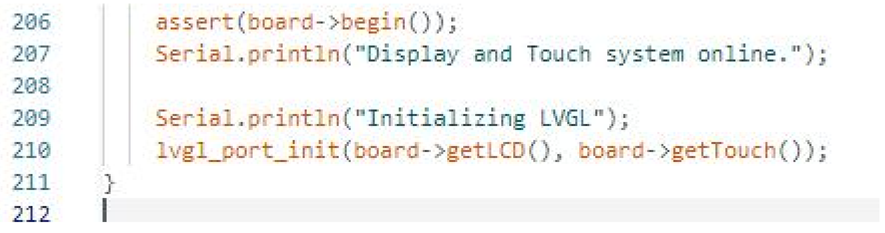
```
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
```

Then, when the conditional compilation LVGL_PORT_AVOID_TEARING_MODE is enabled, the program will obtain the LCD object board->getLCD(), and call configFrameBufferNumber() to set the number of frame buffers, in order to reduce the possibility of screen tearing that may occur during screen refresh.



```
#if LVGL_PORT_AVOID_TEARING_MODE
    auto lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
```

Then, call board->begin() to start the entire display and touch system, enabling the hardware to enter a normal working state, and print the prompt "Display and Touch system online." via the serial port.

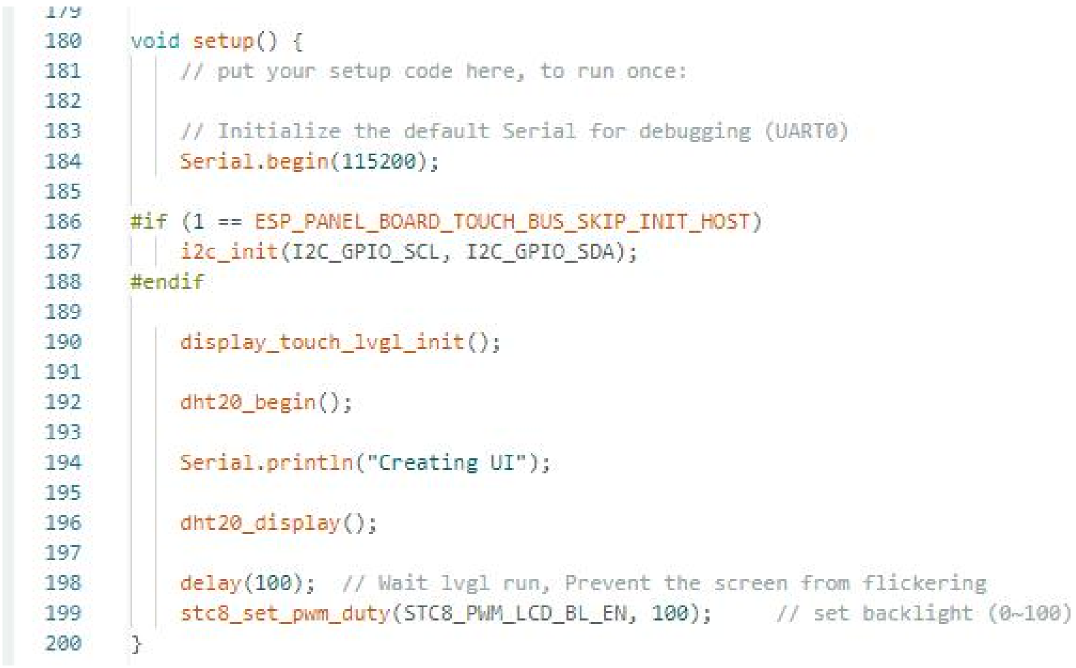


```
    assert(board->begin());
    Serial.println("Display and Touch system online.");

    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());
}

```

Finally, the program outputs "Initializing LVGL", and calls lvgl_port_init(board->getLCD(), board->getTouch()) to initialize the graphics library LVGL. At the same time, it registers the LCD display driver and touch input device into the LVGL system, enabling LVGL to handle interface drawing and receive touch events.

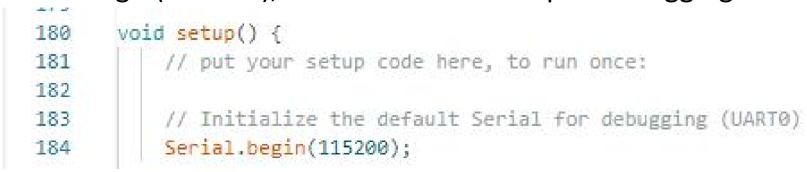
7. setup

This code defines the setup() function in the Arduino program. Its function is to complete the initialization of the entire hardware system and the graphical interface after the system is powered on or reset, preparing for the subsequent operation.



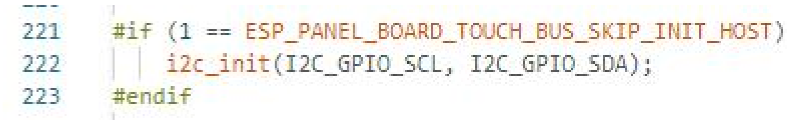
```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif

    display_touch_lvgl_init();

    dht20_begin();

    Serial.println("Creating UI");

    dht20_display();

    delay(100);  // Wait lvgl run, Prevent the screen from flickering
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);     // set backlight (0~100)
}
```

The function first initializes the serial communication (UART0) through Serial.begin(115200), which is used to output debugging information.



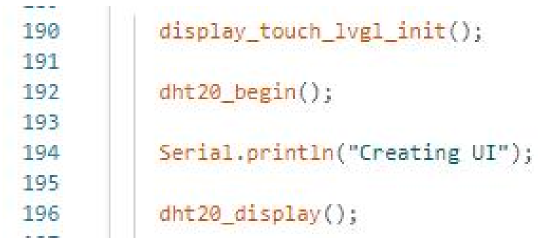
```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);
```

Subsequently, the code uses conditional compilation to check whether the macro ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST is equal to 1. If it is 1, it calls i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) to manually initialize the I2C bus.



```
#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif
```

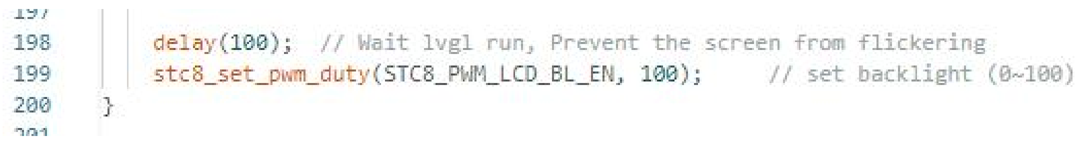
Subsequently, the program calls the function display_touch_lvgl_init() to initialize the display and touch system. This function will activate the LCD display driver, the touch controller GT911, and initialize the graphics interface library LVGL, enabling the system to have the capabilities of graphical display and touch interaction.



```
    display_touch_lvgl_init();

    dht20_begin();

    Serial.println("Creating UI");

    dht20_display();
```

Next, the function dht20_begin() is called to initialize the temperature and humidity sensor DHT20, so as to read the environmental data later. Finally, the program outputs the prompt message "Creating UI" through Serial.println() and calls dht20_display() to create the temperature and humidity display labels on the LVGL interface, thereby establishing a UI element on the screen for displaying temperature and humidity data.

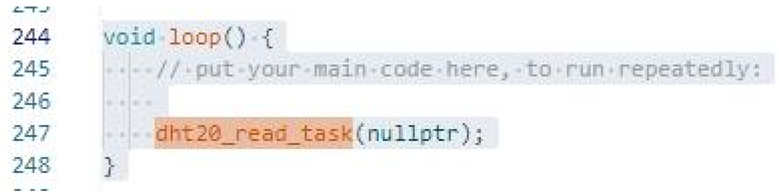
Then, the program executes delay(100) to wait for approximately 100ms, allowing LVGL sufficient time to complete the interface rendering, to avoid screen flickering caused by immediately turning on the backlight after screen initialization; finally, it calls stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100) to set the LCD backlight PWM duty cycle to 100%, controlling the backlight brightness through STC8H1KXX, thereby lighting up the screen and fully displaying the interface that has been drawn.



```
    delay(100);  // Wait lvgl run, Prevent the screen from flickering
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);    // set backlight (0~100)
}
```

8. Loop Section

This code defines the "loop()" function in the Arduino program, which serves as the main execution loop that runs continuously after system startup. Within this function, the program invokes "dht20_read_task(nullptr)" to continuously read data from the temperature and humidity sensor and update the display interface.



```
void loop() {
    // put your main code here, to run repeatedly:
    
    dht20_read_task(nullptr);
}
```

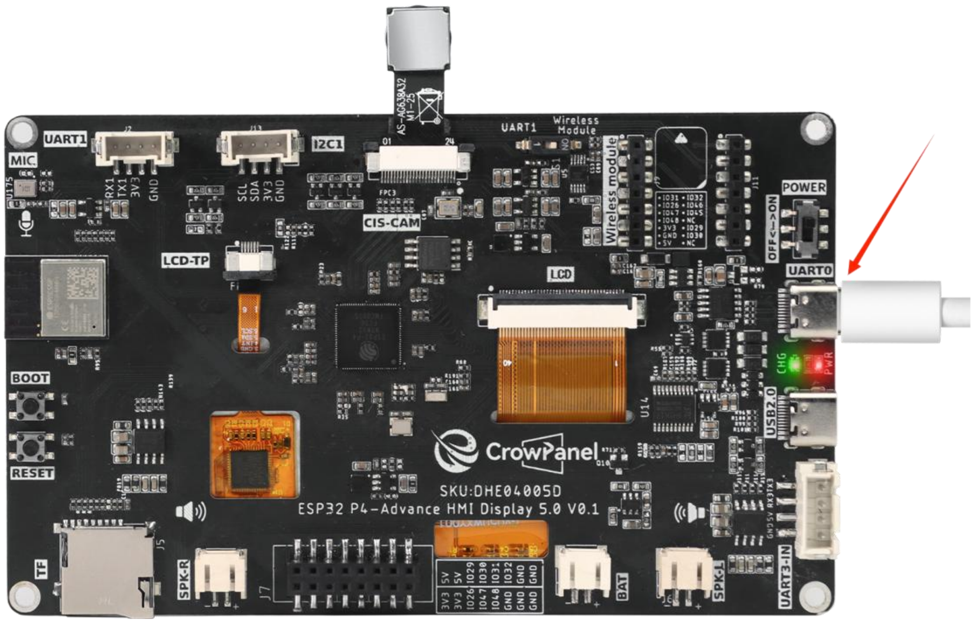
The "nullptr" is passed as an argument because this function was originally designed as a task function (typically used in operating system tasks or threads), requiring a "void *" type parameter. Since it is not actually utilized here, passing a null pointer suffices.

The function "dht20_read_task()" continuously detects and reads temperature and humidity data from the DHT20, outputs the results to the serial port log, and updates the on-screen text label through LVGL—achieving real-time display of temperature and humidity information.

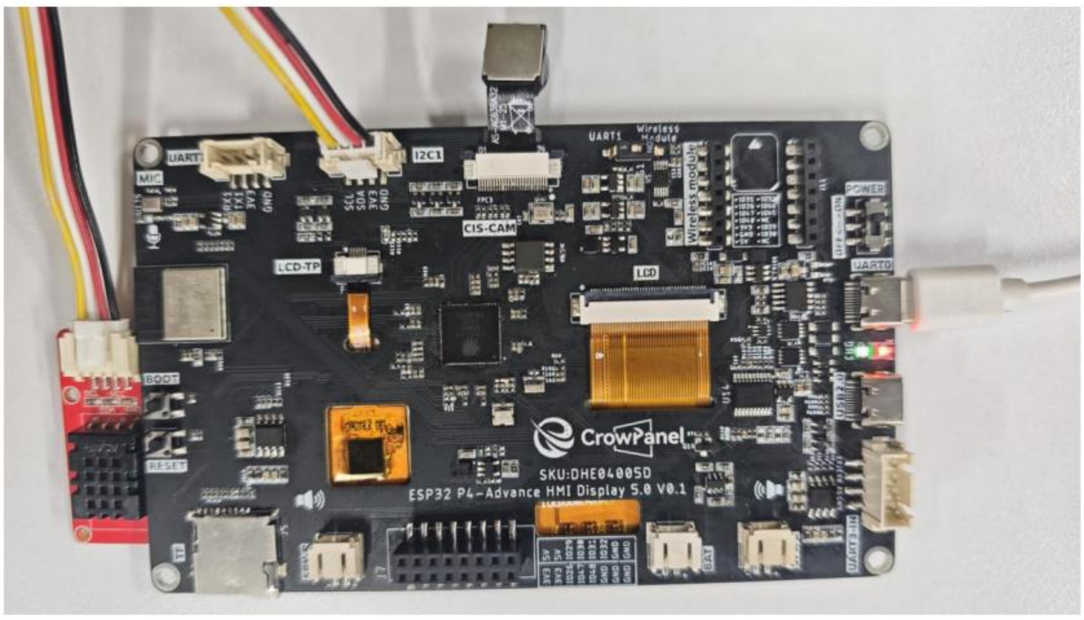
As "dht20_read_task()" internally contains a "while(1)" infinite loop with timed delay logic, once this function is invoked within "loop()", the program remains inside this task indefinitely, continuously executing sensor reading and interface update operations. This implements a continuous monitoring function that reads temperature and humidity at fixed intervals and refreshes the screen display accordingly.

Programming Steps

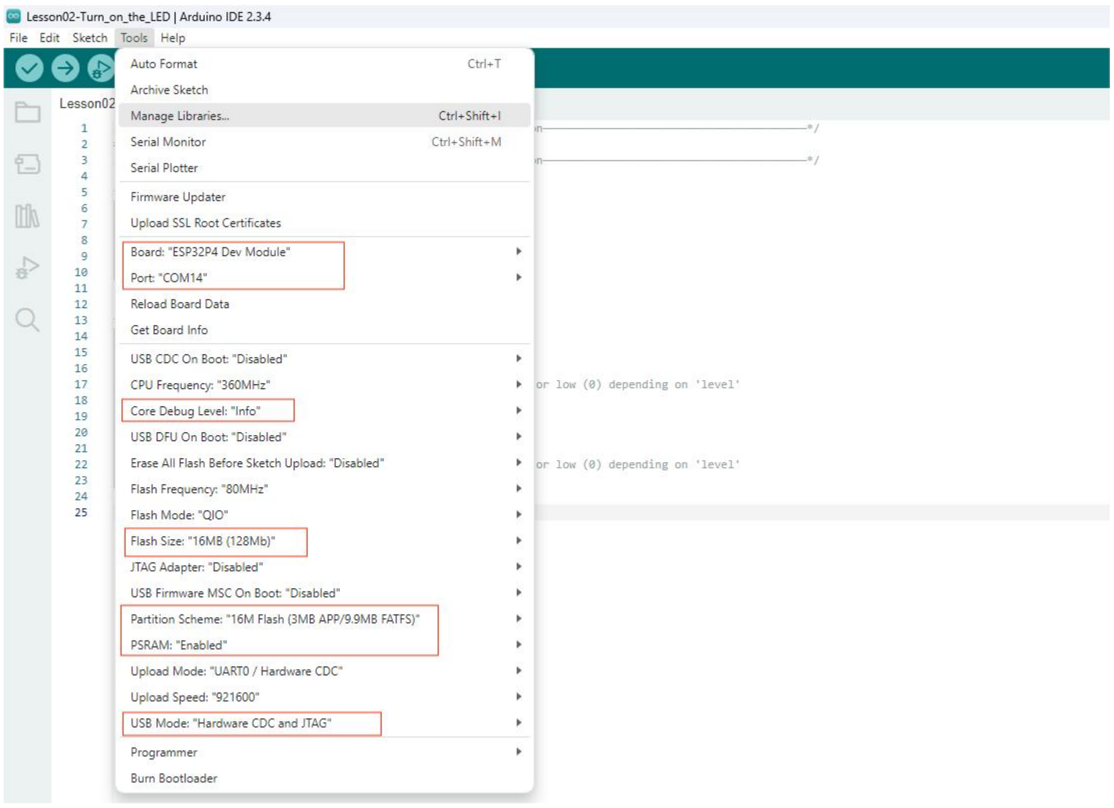
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



After connecting the Advance-P4 board, connect the temperature and humidity sensor to the I2C interface.

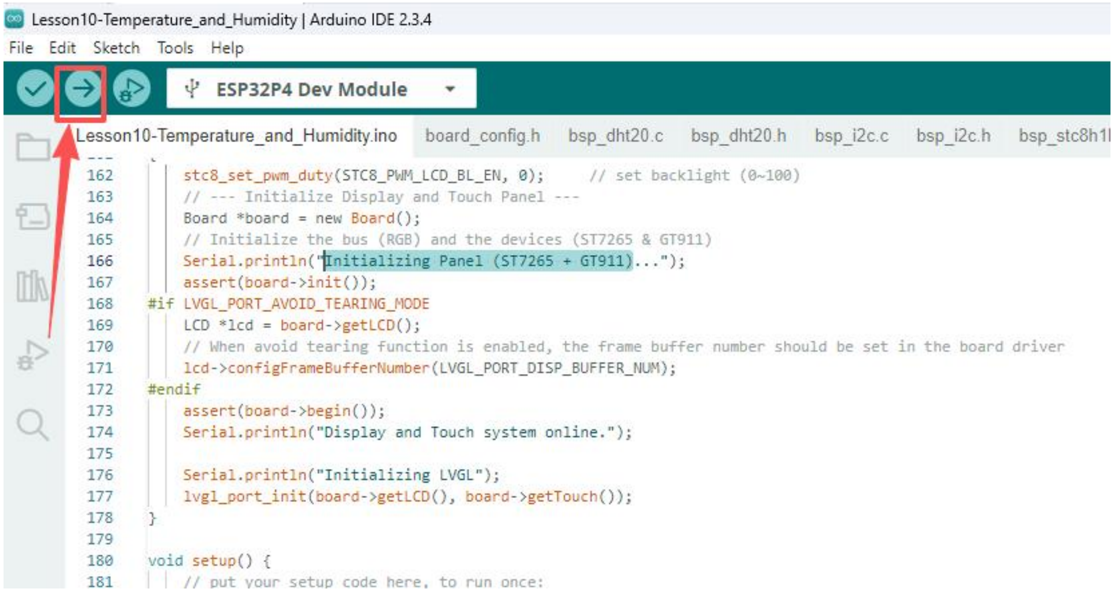


Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Reload Board Data
Get Board Info
Programmer
Burn Bootloader

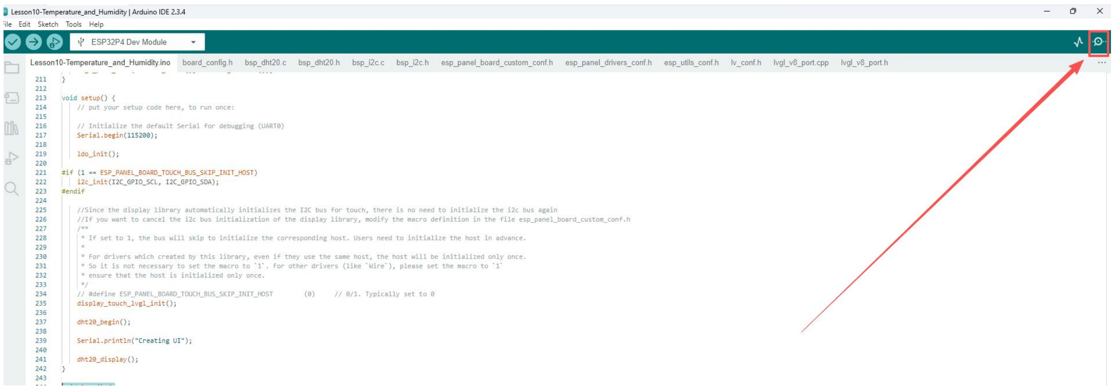
```
Board: "ESP32P4 Dev Module"
Port: "COM14"
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
```

Then we compile and upload the code.


Lesson10-Temperature_and_Humidity | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson10-Temperature_and_Humidity.ino
board_config.h
bsp_dht20.c
bsp_dht20.h
bsp_i2c.c
bsp_i2c.h

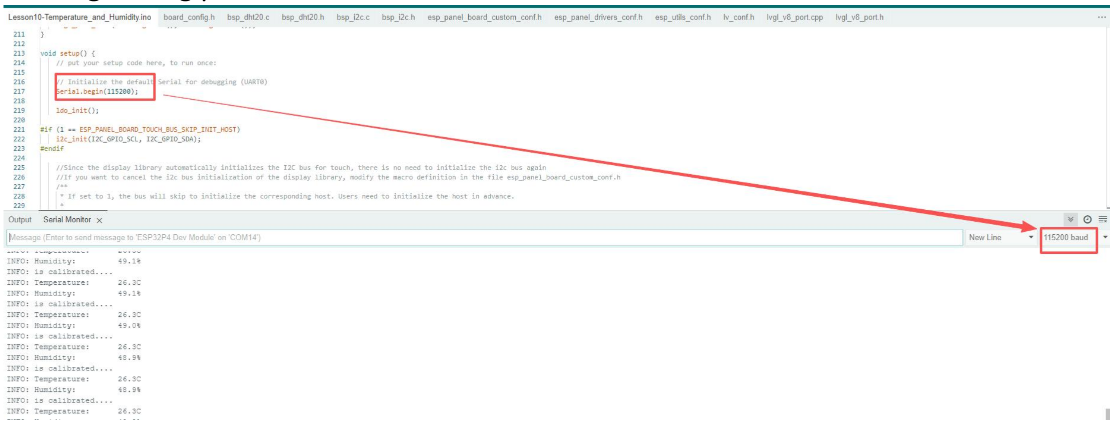
```
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);    // set backlight (0~100)
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
    assert(board->begin());
    Serial.println("Display and Touch system online.");

    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());
}

void setup() {
    // put your setup code here, to run once:
```

Wait for the compilation to complete, then open the Serial Monitor built into the Arduino IDE.



```
}

void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    ldo_init();

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif

    //Since the display library automatically initializes the I2C bus for touch, there is no need to initialize the i2c bus again
    //If you want to cancel the i2c bus initialization of the display library, modify the macro definition in the file esp_panel_board_custom_conf.h
    /**
     * If set to 1, the bus will skip to initialize the corresponding host. Users need to initialize the host in advance.
     *
     * For drivers which created by this library, even if they use the same host, the host will be initialized only once.
     * So it is not necessary to set the macro to `1`. For other drivers (like `Wire`), please set the macro to `1`
     * ensure that the host is initialized only once.
     */
    // #define ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST        (0)     // 0/1. Typically set to 0
    display_touch_lvgl_init();

    dht20_begin();

    Serial.println("Creating UI");

    dht20_display();
}
```

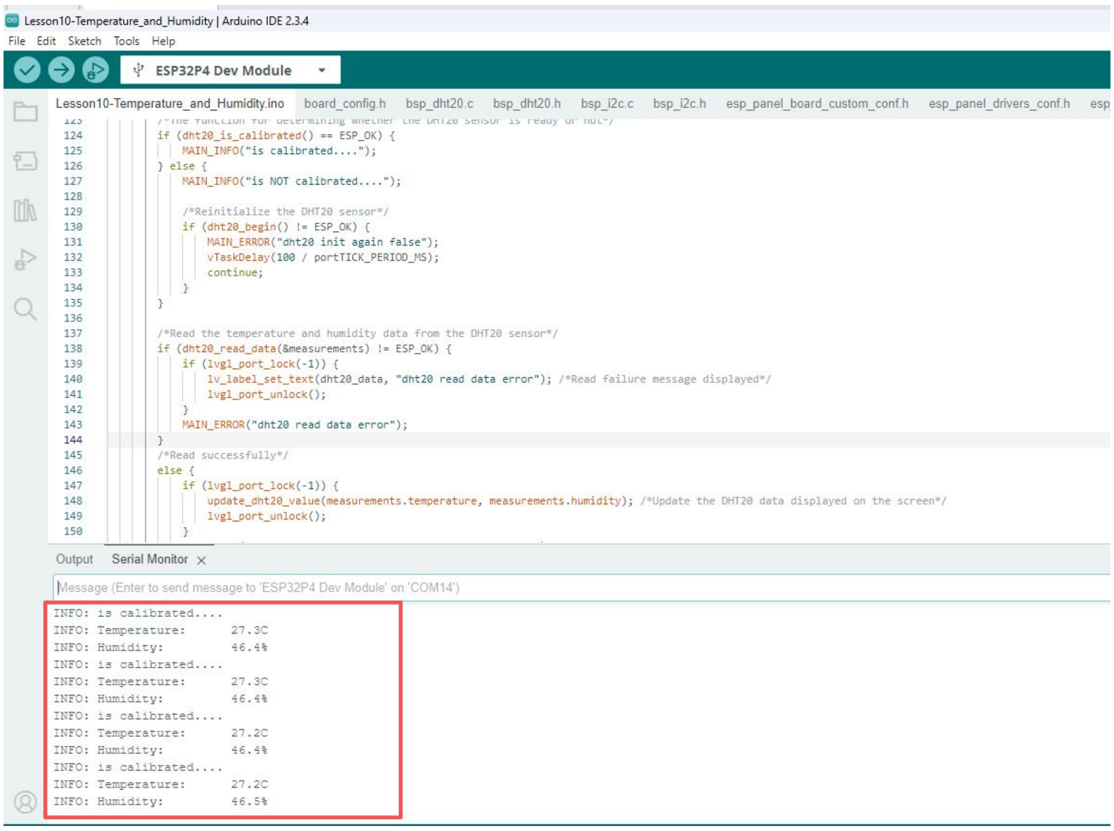
Configure the baud rate in your code, and you'll see the temperature and humidity readings being printed.



```
}

void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    ldo_init();

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif

    //Since the display library automatically initializes the I2C bus for touch, there is no need to initialize the i2c bus again
    //If you want to cancel the i2c bus initialization of the display library, modify the macro definition in the file esp_panel_board_custom_conf.h
    /**
     * If set to 1, the bus will skip to initialize the corresponding host. Users need to initialize the host in advance.
     *
```

```
INFO: Humidity:         49.1%
INFO: is calibrated....
INFO: Temperature:      26.3C
INFO: Humidity:         49.1%
INFO: is calibrated....
INFO: Temperature:      26.3C
INFO: Humidity:         49.0%
INFO: is calibrated....
INFO: Temperature:      26.3C
INFO: Humidity:         48.9%
INFO: is calibrated....
INFO: Temperature:      26.3C
INFO: Humidity:         48.9%
INFO: is calibrated....
INFO: Temperature:      26.3C
```

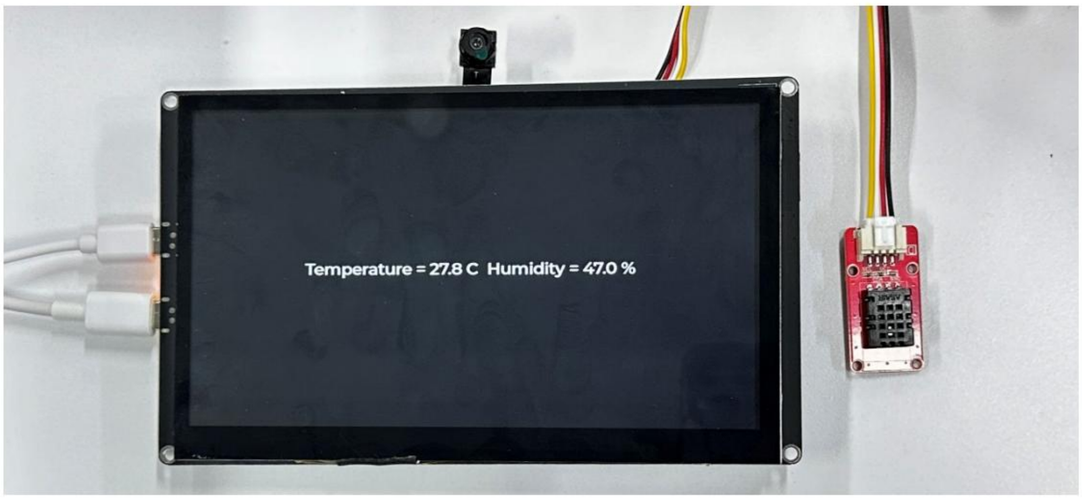
After successful flashing, you will see that the screen of your Advance-P4 lights up, and the data collected by the temperature and humidity sensor is displayed on the screen in real time.


Lesson10-Temperature_and_Humidity | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson10-Temperature_and_Humidity.ino
board_config.h
bsp_dht20.c
bsp_dht20.h
bsp_i2c.c
bsp_i2c.h
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h
Output
Serial Monitor

```
        /*The function for determining whether the DHT20 sensor is ready or not*/
        if (dht20_is_calibrated() == ESP_OK) {
          MAIN_INFO("is calibrated....");
        } else {
          MAIN_INFO("is NOT calibrated....");

          /*Reinitialize the DHT20 sensor*/
          if (dht20_begin() != ESP_OK) {
            MAIN_ERROR("dht20 init again false");
            vTaskDelay(100 / portTICK_PERIOD_MS);
            continue;
          }
        }

        /*Read the temperature and humidity data from the DHT20 sensor*/
        if (dht20_read_data(&measurements) != ESP_OK) {
          if (lvgl_port_lock(-1)) {
            lv_label_set_text(dht20_data, "dht20 read data error"); /*Read failure message displayed*/
            lvgl_port_unlock();
          }
          MAIN_ERROR("dht20 read data error");
        }
        /*Read successfully*/
        else {
          if (lvgl_port_lock(-1)) {
            update_dht20_value(measurements.temperature, measurements.humidity); /*Update the DHT20 data displayed on the screen*/
            lvgl_port_unlock();
          }
```

Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
INFO: is calibrated....
INFO: Temperature:      27.3C
INFO: Humidity:         46.4%
INFO: is calibrated....
INFO: Temperature:      27.3C
INFO: Humidity:         46.4%
INFO: is calibrated....
INFO: Temperature:      27.2C
INFO: Humidity:         46.4%
INFO: is calibrated....
INFO: Temperature:      27.2C
INFO: Humidity:         46.5%
```



Lesson11---Playback After Recording

Introduction

In this lesson, we will teach you how to use the microphone and speaker on the Advance-P4 board. We will complete a project: record audio for 5 seconds, then automatically play back the 5-second audio clip.

Learning Goals

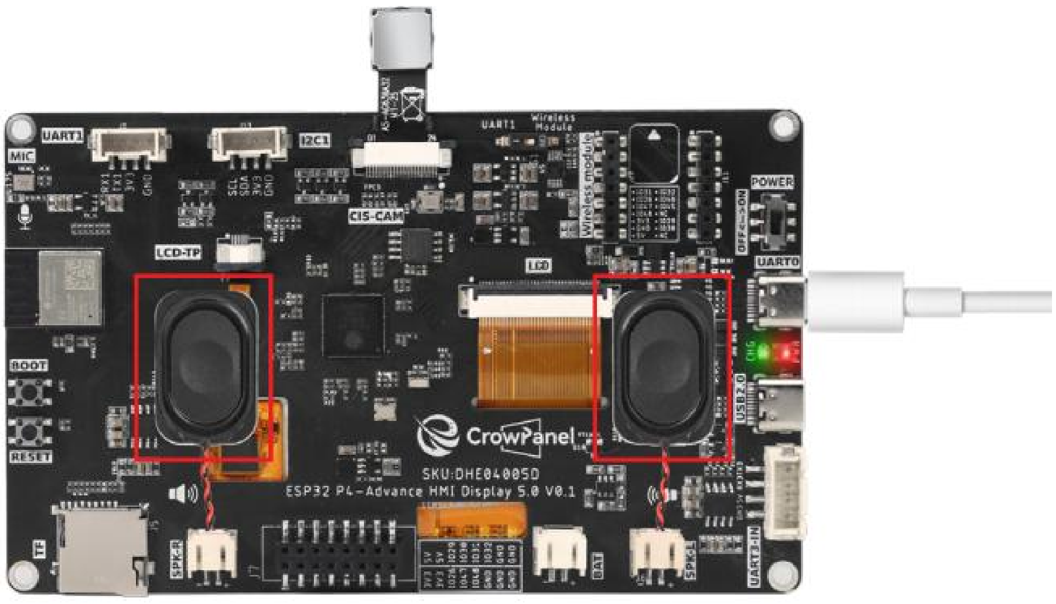
1. Understand the principles of the I2S/PDM audio interface on the ESP32-P4 development board, as well as the working mechanisms of microphones (acoustic-to-electrical conversion) and speakers (electrical-to-acoustic conversion).

2. Master the initialization and configuration of the I2S audio interface (PDM input for microphone, I2S output for speaker), along with the implementation methods for 5-second audio recording (WAV format) and playback.

3. Grasp the core logic of audio data processing (WAV header removal, volume gain control, anti-clipping distortion), completing the full closed-loop workflow of recording, processing, and playback.

Preview of the Result

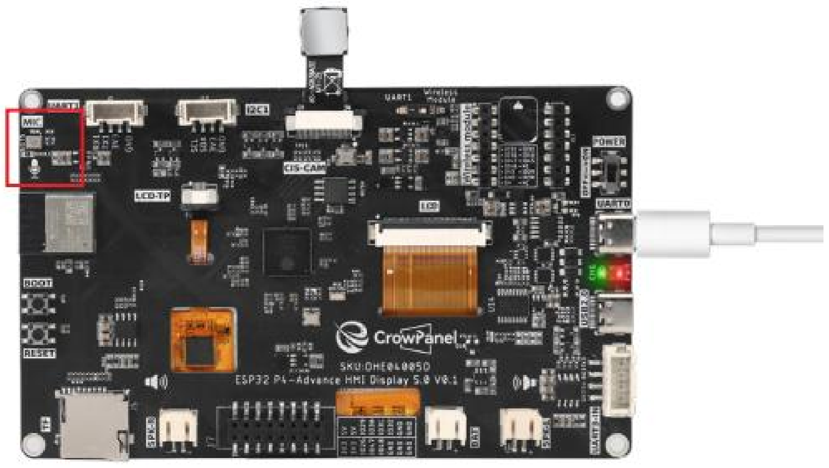
After running the code, you will be able to speak near the Advance-P4. The Advance-P4 will use its microphone to record the current sound within 5 seconds, then play it back automatically.

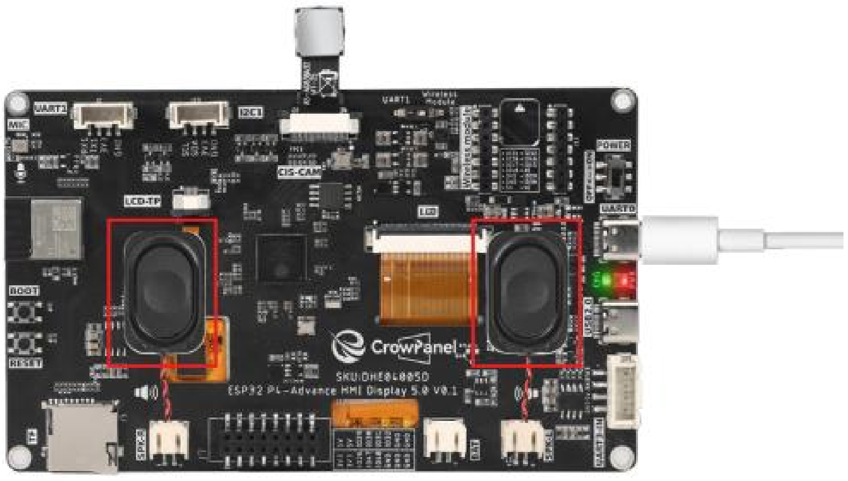


The 5-second recorded audio is now playing.

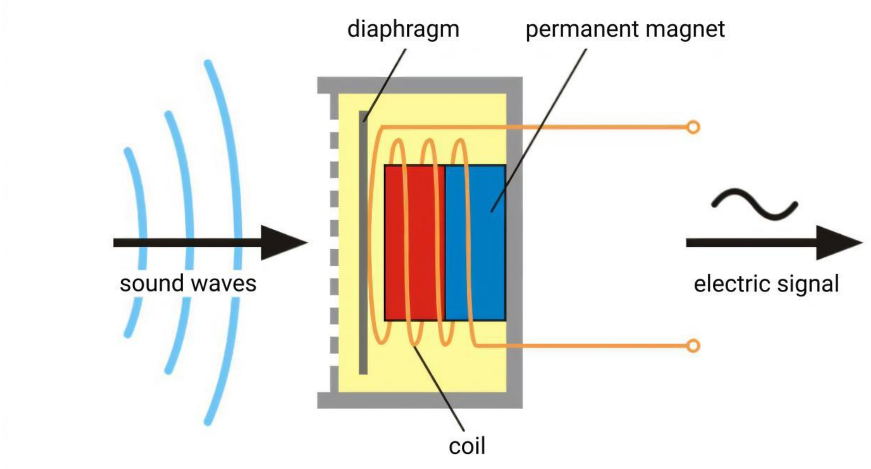
Hardware Used in This Lesson

Microphone and Speaker on the Advance-P4





Microphone and Speaker Schematic Diagrams



When an audio signal enters in the form of sound waves, it causes the diaphragm to vibrate. The diaphragm is connected to a coil, which is sleeved around a magnetic core (located in a magnetic field). The vibration makes the coil move in the magnetic field, cutting through the magnetic field lines. According to the law of electromagnetic induction, an electrical signal corresponding to the variation pattern of the audio signal is generated in the coil, thereby realizing the conversion of sound signals to electrical signals.(For a speaker, this is the reverse process of converting electrical signals to sound signals: an energized coil is forced to vibrate in a magnetic field, which drives the diaphragm to vibrate and produce sound.)

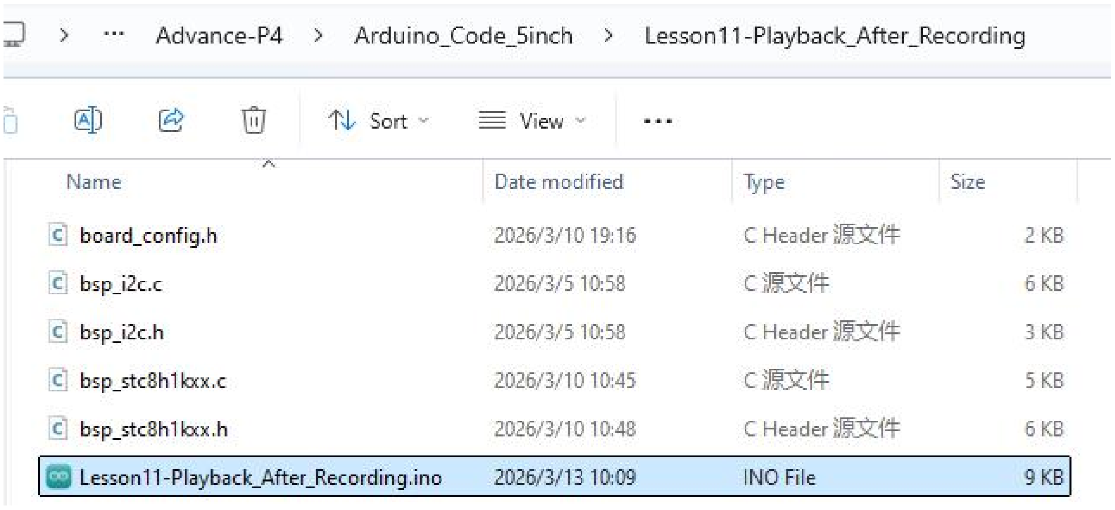
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson11-Playback_After_Recording

Key Explanations

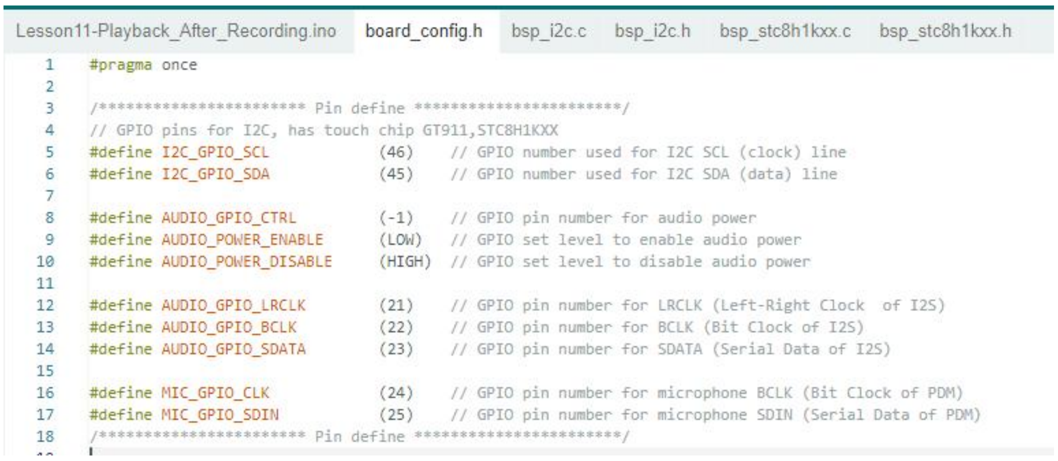
How exactly is the 5-second audio recording implemented, and how is the recorded audio subsequently played back? Let's examine the code implementation together.

Double-click to open the code for this lesson (the .ino file).


> ··· Advance-P4 > Arduino_Code_5inch > Lesson11-Playback_After_Recording
Sort
View
···

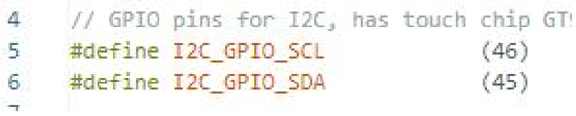
Name	Date modified	Type	Size
board_config.h	2026/3/10 19:16	C Header 源文件	2 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/10 10:48	C Header 源文件	6 KB
Lesson11-Playback_After_Recording.ino	2026/3/13 10:09	INO File	9 KB

After opening, you can see the code of this project and the configuration file board_config.h.
This code is a hardware pin configuration header file, which is used to centrally define the GPIO pins used by each peripheral on the development board, making it convenient for the program to uniformly call and manage them in different modules.
The #pragma once at the beginning of the file is used to prevent the header file from being included multiple times, ensuring that it is loaded only once during compilation.



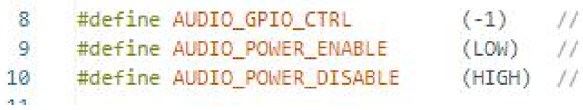
```
#pragma once

/*********************** Pin define ***********************/
// GPIO pins for I2C, has touch chip GT911,STC8H1KXX
#define I2C_GPIO_SCL            (46)    // GPIO number used for I2C SCL (clock) line
#define I2C_GPIO_SDA            (45)    // GPIO number used for I2C SDA (data) line

#define AUDIO_GPIO_CTRL         (-1)    // GPIO pin number for audio power
#define AUDIO_POWER_ENABLE      (LOW)   // GPIO set level to enable audio power
#define AUDIO_POWER_DISABLE     (HIGH)  // GPIO set level to disable audio power

#define AUDIO_GPIO_LRCLK        (21)    // GPIO pin number for LRCLK (Left-Right Clock  of I2S)
#define AUDIO_GPIO_BCLK         (22)    // GPIO pin number for BCLK (Bit Clock of I2S)
#define AUDIO_GPIO_SDATA        (23)    // GPIO pin number for SDATA (Serial Data of I2S)

#define MIC_GPIO_CLK            (24)    // GPIO pin number for microphone BCLK (Bit Clock of PDM)
#define MIC_GPIO_SDIN           (25)    // GPIO pin number for microphone SDIN (Serial Data of PDM)
/*********************** Pin define ***********************/
```

Next, a set of pins related to the I^2 C bus is defined. Among them, I2C_GPIO_SCL is set to GPIO46, serving as the clock line (SCL) of the I^2 C, and I2C_GPIO_SDA is set to GPIO45, serving as the data line (SDA). This I^2 C bus is mainly used for communication with the GT911 touch controller and the STC8H1KXX coprocessor, so as to obtain touch data, read battery information or control the backlight, etc.



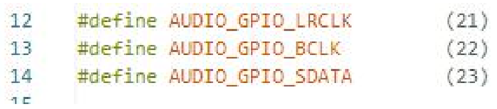
```
// GPIO pins for I2C, has touch chip GT
#define I2C_GPIO_SCL            (46)
#define I2C_GPIO_SDA            (45)
```

Subsequently, the code defines a set of audio-related pins. Among them, AUDIO_GPIO_CTRL represents the audio power control pin. If it is -1, it indicates that no dedicated GPIO is used to control the audio power. AUDIO_POWER_ENABLE and AUDIO_POWER_DISABLE respectively define the levels to be output when turning on or off the audio power.



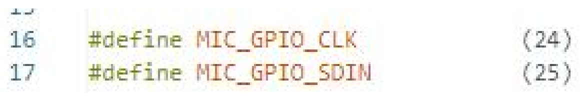
```
#define AUDIO_GPIO_CTRL         (-1)    //
#define AUDIO_POWER_ENABLE      (LOW)   //
#define AUDIO_POWER_DISABLE     (HIGH)  //
```

Then, three I^2 S audio interface pins were defined: AUDIO_GPIO_LRCLK (GPIO21) as the left and right channel clock LRCLK, AUDIO_GPIO_BCLK (GPIO22) as the bit clock BCLK, and AUDIO_GPIO_SDATA (GPIO23) as the audio data line SDATA. These pins are usually used to connect audio codecs or power amplifier chips to achieve audio playback functionality.



```
#define AUDIO_GPIO_LRCLK        (21)
#define AUDIO_GPIO_BCLK         (22)
#define AUDIO_GPIO_SDATA        (23)
```

In addition, the code also defines two PDM microphone interface pins: MIC_GPIO_CLK (GPIO24) as the microphone clock signal, and MIC_GPIO_SDIN (GPIO25) as the microphone data input, which is used to collect the audio data from the digital microphone.



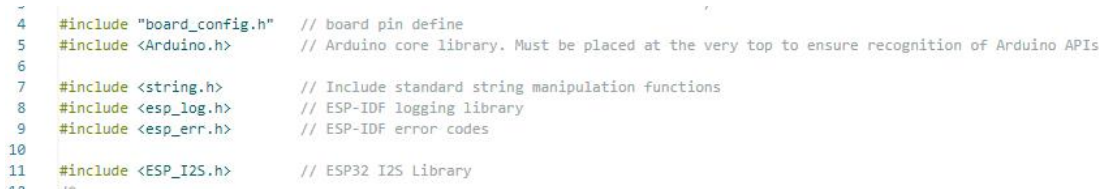
```
#define MIC_GPIO_CLK            (24)
#define MIC_GPIO_SDIN           (25)
```

Then we focused on the code implementation of the main code file "ino", to understand how the functions of this lesson were achieved.

(The meanings and functions of each of the remaining files have been elaborated in detail in Lesson 7. We will not go into further details here. If you have any questions, you can review the previous course content.)

1. Header Files

This code serves to include the necessary header files at the program's outset, providing essential functional support for subsequent audio processing, log output, and hardware control.



```
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs

#include <string.h>         // Include standard string manipulation functions
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

#include <ESP_I2S.h>        // ESP32 I2S Library
```

First, #include "board_config.h" introduces the custom development board configuration file, which defines GPIO pins related to the audio system (such as speaker, microphone, and audio power control pins). This allows the main program to use macro names directly without writing specific pin numbers.

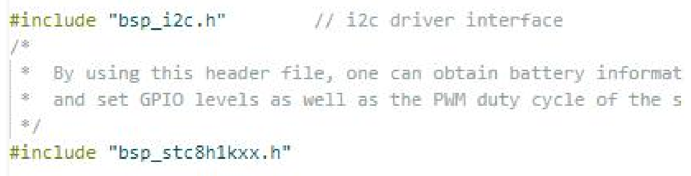
Next, <Arduino.h> is included—this is the Arduino core library providing foundational functions and interfaces such as pinMode(), digitalWrite(), Serial.begin(), etc. It must be placed early to ensure proper recognition of Arduino APIs.

The code then introduces <string.h>, a standard C library providing string manipulation functions such as memcpy(), strlen(), etc., facilitating character data processing in the program.

Subsequently, <esp_log.h> and <esp_err.h> are included—both libraries from ESP-IDF, serving the logging output system and error code definitions respectively. These enable the program to output debug information and determine function execution status.

Finally, <ESP_I2S.h> is included—this is the Arduino audio library for controlling I2S, providing the I2S class and related functions. It enables the program to implement digital audio functionalities such as microphone recording and speaker playback on the ESP32-P4.

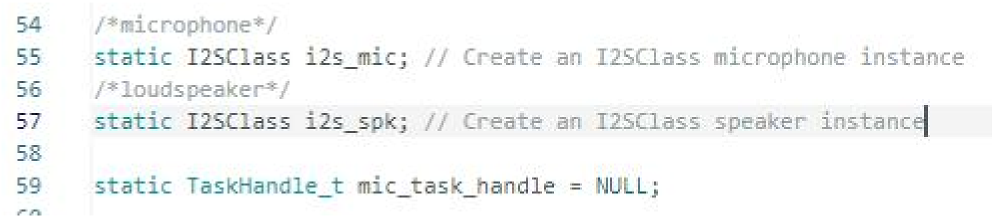
bsp_i2c.h provides the I²C communication interface; bsp_stc8h1kxx.h is used to obtain battery information, control GPIO, and adjust the screen backlight PWM through the STC8H1KXX coprocessor.



```
#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery informat
 *  and set GPIO levels as well as the PWM duty cycle of the s
 */
#include "bsp_stc8h1kxx.h"
```

2. Global Variables

This code section primarily defines global objects and task handles related to audio input and output, preparing the groundwork for subsequent recording and playback functionality.


```
/*microphone*/
static I2SClass i2s_mic; // Create an I2SClass microphone instance
/*loudspeaker*/
static I2SClass i2s_spk; // Create an I2SClass speaker instance

static TaskHandle_t mic_task_handle = NULL;
```

First, static I2SClass i2s_mic; creates an object of type I2SClass named i2s_mic for controlling microphone data input. This object receives audio data from the digital microphone via the I2S interface. The preceding static keyword restricts the variable's visibility to within the current source file, preventing access from other files and thereby enhancing code encapsulation.

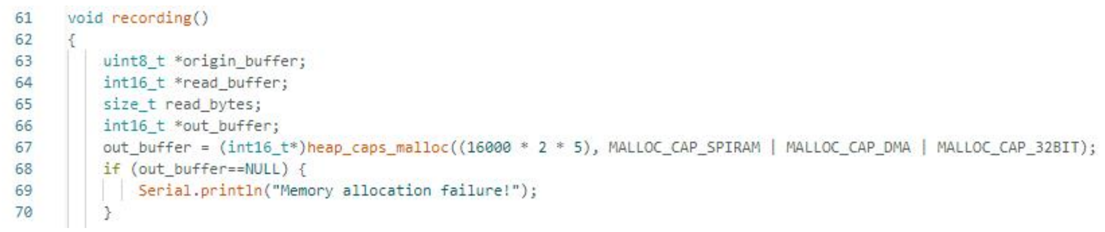
Next, static I2SClass i2s_spk; creates another I2SClass object named i2s_spk for controlling speaker audio output—sending processed digital audio data to the speaker or amplifier module for playback. These two objects respectively handle audio input (recording) and audio output (playback) functionalities.

The final line static TaskHandle_t mic_task_handle = NULL; defines a task handle variable for storing the audio acquisition task handle. This type is typically used for FreeRTOS task management, such as creating, suspending, or deleting tasks. Here it is initialized to NULL, indicating that no related task has been created yet.

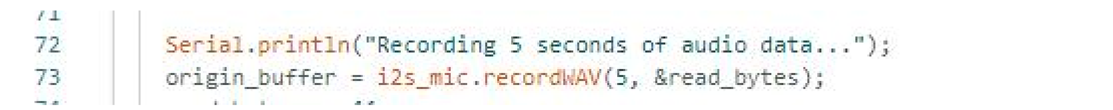
3. recording

This code defines a function recording(), whose primary role is to complete a full workflow of audio acquisition, volume processing, and speaker playback.

The function first declares multiple buffer pointer variables, then allocates a buffer out_buffer in external memory via heap_caps_malloc() for storing processed audio data. This function allows specifying memory attributes (such as SPIRAM, DMA accessibility, etc.) to ensure audio data can be efficiently read by hardware interfaces.

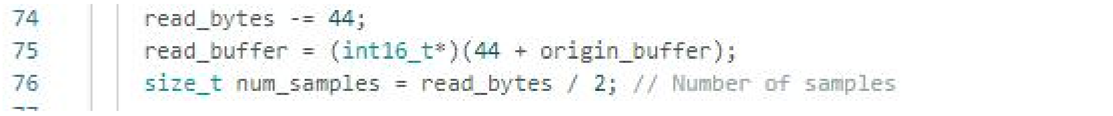

```
void recording()
{
    uint8_t *origin_buffer;
    int16_t *read_buffer;
    size_t read_bytes;
    int16_t *out_buffer;
    out_buffer = (int16_t*)heap_caps_malloc((16000 * 2 * 5), MALLOC_CAP_SPIRAM | MALLOC_CAP_DMA | MALLOC_CAP_32BIT);
    if (out_buffer==NULL) {
        Serial.println("Memory allocation failure!");
    }
```

The program then invokes "i2s_mic.recordWAV(5, &read_bytes)" on the microphone object to begin recording—capturing 5 seconds of audio data and returning the raw data buffer "origin_buffer" in WAV format, while simultaneously obtaining the total number of bytes read.



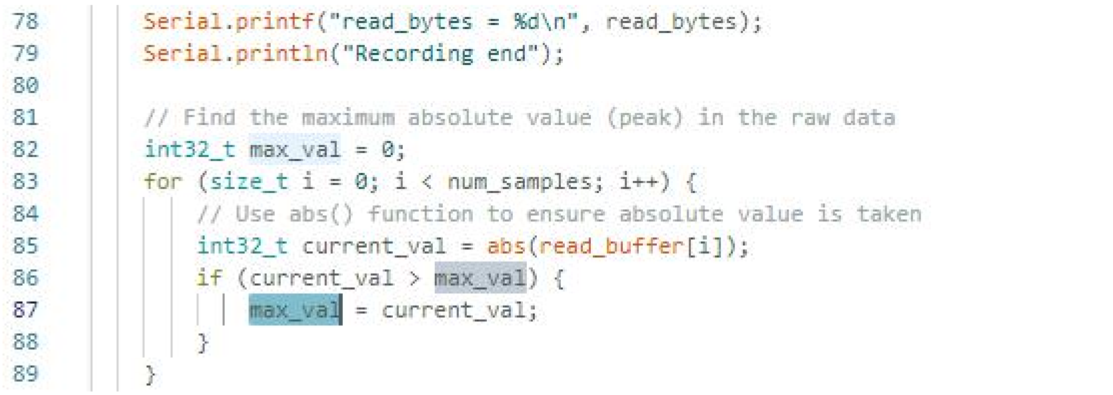
```
Serial.println("Recording 5 seconds of audio data...");
origin_buffer = i2s_mic.recordWAV(5, &read_bytes);
```

Since the first 44 bytes of a WAV file constitute the header information, the code removes this header length via "read_bytes -= 44", repositions "read_buffer" to point to the actual 16-bit audio sample data, and calculates the sample count "num_samples".



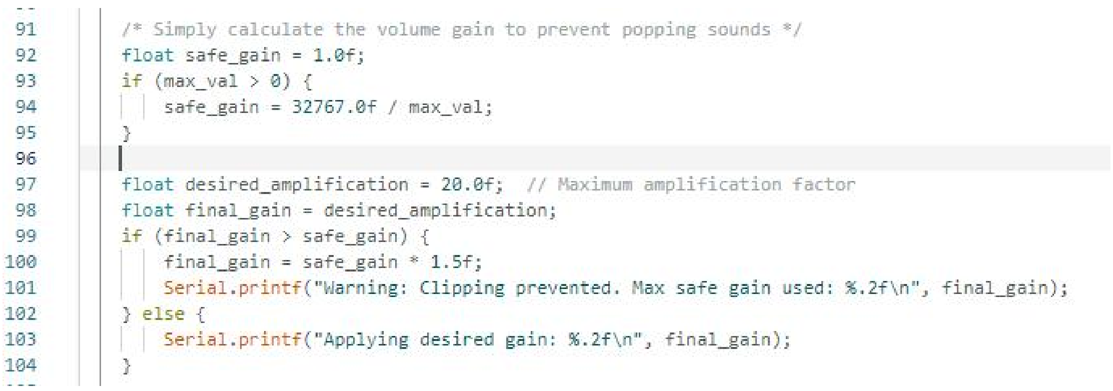
```
read_bytes -= 44;
read_buffer = (int16_t*)(44 + origin_buffer);
size_t num_samples = read_bytes / 2; // Number of samples
```

The program then iterates through all sample values, calculating the absolute value of each sample point via "abs()" to determine the maximum amplitude "max_val" across the entire audio segment.



```
Serial.printf("read_bytes = %d\n", read_bytes);
Serial.println("Recording end");

// Find the maximum absolute value (peak) in the raw data
int32_t max_val = 0;
for (size_t i = 0; i < num_samples; i++) {
    // Use abs() function to ensure absolute value is taken
    int32_t current_val = abs(read_buffer[i]);
    if (current_val > max_val) {
        max_val = current_val;
    }
}
```

The program then calculates a safe gain "safe_gain" based on this maximum value, ensuring that audio amplification does not exceed the 16-bit audio range (-32768 to 32767). This is compared against the desired amplification factor "desired_amplification" (20x), yielding the "final_gain"—thereby achieving automatic volume gain control while preventing clipping distortion.



```
/* Simply calculate the volume gain to prevent popping sounds */
float safe_gain = 1.0f;
if (max_val > 0) {
    safe_gain = 32767.0f / max_val;
}

float desired_amplification = 20.0f;  // Maximum amplification factor
float final_gain = desired_amplification;
if (final_gain > safe_gain) {
    final_gain = safe_gain * 1.5f;
    Serial.printf("Warning: Clipping prevented. Max safe gain used: %.2f\n", final_gain);
} else {
    Serial.printf("Applying desired gain: %.2f\n", final_gain);
}
```

The program then iterates through all sample points once more, multiplying the raw audio data by "final_gain" and writing the results to "out_buffer", while employing conditional checks to constrain the value range—ensuring no overflow occurs.

```
    // Multiply the original data by the gain coefficient
    for (size_t i=0; i<num_samples; i+=1) {
        if (read_buffer[i] < 0) {
            if (-32768 / final_gain < read_buffer[i]) {
                out_buffer[i] = (read_buffer[i]) * final_gain;
            } else {
                out_buffer[i] = -32768;
            }
        } else {
            if (read_buffer[i]< 32767 / final_gain) {
                out_buffer[i] = (read_buffer[i]) * final_gain;
            } else {
                out_buffer[i] = 32767;
            }
        }
    }
```

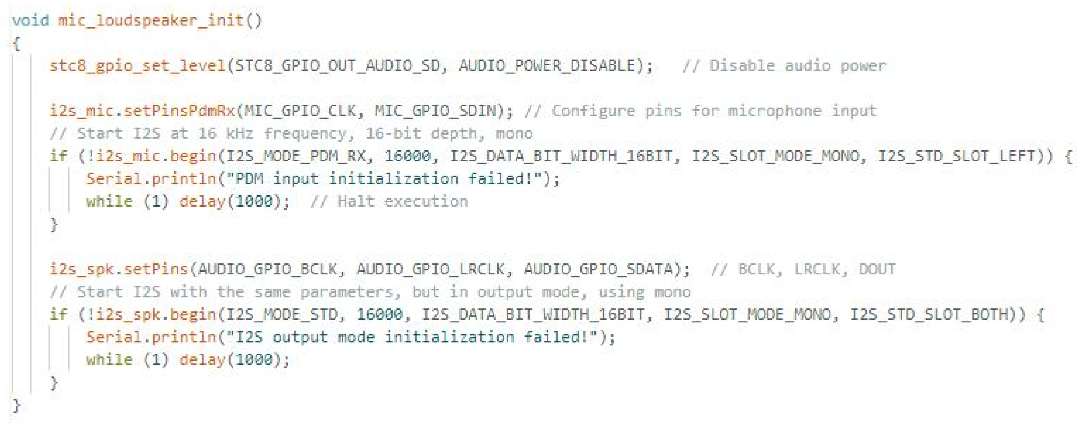
After the audio processing is completed, the program uses stc8_gpio_set_level() to control the external STC8H1KXX to turn on the audio power supply, and then calls i2s_spk.write() to send the processed PCM data to the speaker for playback. Since there is often a sudden noise at the beginning of the recording, the code deliberately skips the first 4000 bytes of data before playing to avoid "bursting noise". After the playback is completed, the audio power supply is turned off, and the memory occupied by origin_buffer and out_buffer is released. After the audio processing is completed, the program uses stc8_gpio_set_level() to control the external STC8H1KXX to turn on the audio power supply, and then calls i2s_spk.write() to send the processed PCM data to the speaker for playback.

```
    Serial.println("Playing the recorded audio...");
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_ENABLE);  // Enable audio power
    /* Skip the 4000 bytes of audio data because the first few microphone audio data collected t
    i2s_spk.write((uint8_t*)out_buffer + 4000, read_bytes - 4000);
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE); // Disable audio power

    free(origin_buffer);

    free(out_buffer);
}
```

The code deliberately skips the first 4000 bytes of data because when using the Arduino audio library to record audio on the ESP32-P4, there might be a burst of noise at the beginning. Therefore, a small section of audio data needs to be discarded. After the playback is completed, turn off the audio power supply and release the memory occupied by origin_buffer and out_buffer.

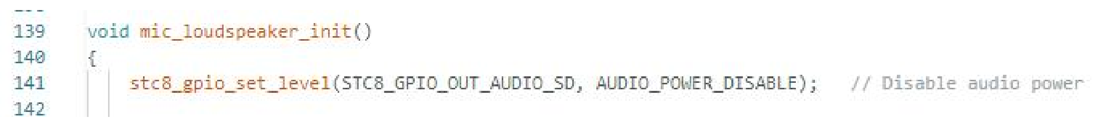
4. mic_loudspeaker_init

This code defines the function mic_loudspeaker_init(), whose main function is to initialize the microphone input and speaker output interfaces in the audio system, enabling the device to perform recording and playback operations.



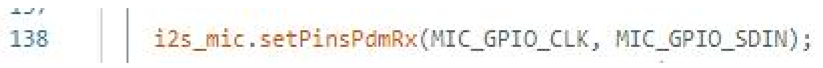
```
void mic_loudspeaker_init()
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);   // Disable audio power

    i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN); // Configure pins for microphone input
    // Start I2S at 16 kHz frequency, 16-bit depth, mono
    if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
        Serial.println("PDM input initialization failed!");
        while (1) delay(1000);  // Halt execution
    }

    i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  // BCLK, LRCLK, DOUT
    // Start I2S with the same parameters, but in output mode, using mono
    if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_BOTH)) {
        Serial.println("I2S output mode initialization failed!");
        while (1) delay(1000);
    }
}
```

The function begins by calling stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE) to turn off the audio power. This step is controlled by the external coprocessor STC8H1KXX to regulate the power supply of the audio module. This ensures that the speaker will not produce noise or malfunction before the audio device is initialized.



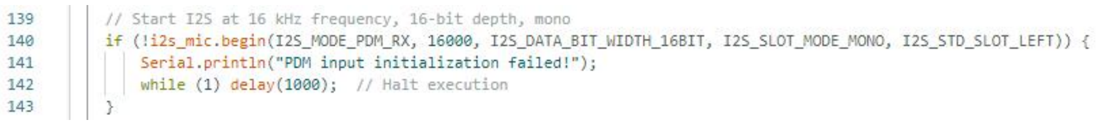
```
void mic_loudspeaker_init()
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);   // Disable audio power
```

The program then invokes "i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN)" to configure the microphone input pins—where "MIC_GPIO_CLK" serves as the microphone clock signal line and "MIC_GPIO_SDIN" as the data input line. These two pins are used to receive audio data from the PDM digital microphone.



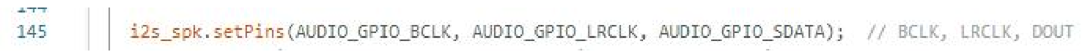
```
    i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN);
```

The program then launches the microphone's audio acquisition interface via "i2s_mic.begin()", configuring the operating mode as "I2S_MODE_PDM_RX" (PDM receive mode), sample rate at 16kHz, data bit width of 16 bits, channel mode as mono, and specifying the use of left channel data. Should initialization fail, the program prints an error message and enters a "while(1)" infinite loop to halt execution.



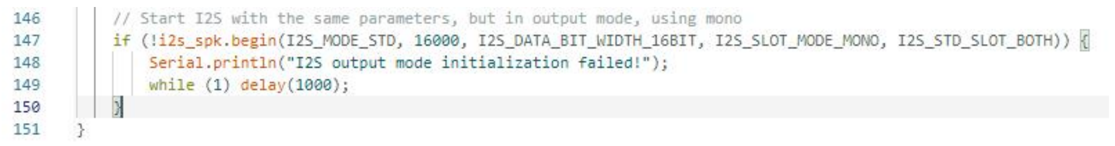
```
    // Start I2S at 16 kHz frequency, 16-bit depth, mono
    if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
        Serial.println("PDM input initialization failed!");
        while (1) delay(1000);  // Halt execution
    }
```

The program then proceeds to configure the speaker output interface. It sets the I2S output pins via "i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA)"—where BCLK serves as the bit clock, LRCLK as the left-right channel clock, and SDATA as the serial audio data line.



```
    i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  // BCLK, LRCLK, DOUT
```

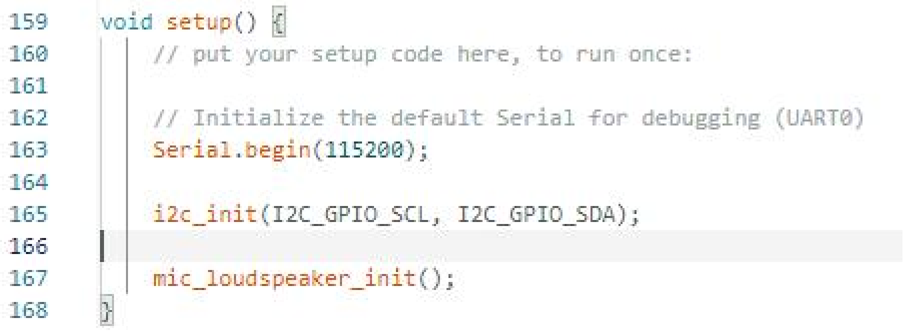
The program then invokes "i2s_spk.begin()" to launch the speaker audio output interface, operating in standard I2S output mode with a sample rate of 16kHz, 16-bit width, mono output, and simultaneous duplication to both left and right channels. Should initialization fail, it similarly outputs an error message and halts program execution.



```
    // Start I2S with the same parameters, but in output mode, using mono
    if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_BOTH)) {
      Serial.println("I2S output mode initialization failed!");
      while (1) delay(1000);
    }
}
```

5. setup

This code defines the setup() function in the Arduino program. It is executed only once when the system is powered on or reset, and is used to perform the initialization tasks required before the program starts running.



```
void setup() {
  // put your setup code here, to run once:

  // Initialize the default Serial for debugging (UART0)
  Serial.begin(115200);

  i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);

  mic_loudspeaker_init();
}
```

The function first initializes the serial communication interface (UART0) through Serial.begin(115200), setting the baud rate to 115200. This enables the program to output debugging information to the computer via the serial port, such as device status, error messages or operation logs.

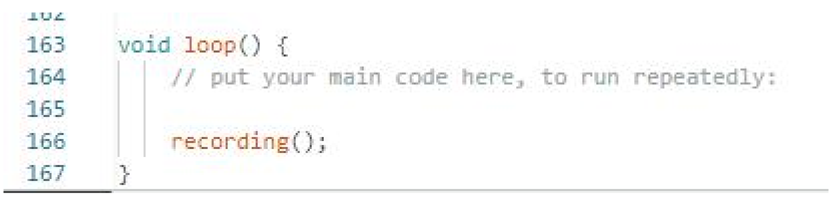
Subsequently, the function i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) is called to initialize the I²C communication bus. Here, I2C_GPIO_SCL and I2C_GPIO_SDA respectively represent the clock line and data line pins of the I²C. This bus is mainly used for communication with external devices, such as reading battery information through the STC8H1KXX coprocessor, controlling the GPIO state or managing the audio power supply, etc.

After completing the I²C initialization, the program calls the mic_loudspeaker_init() function to initialize the audio system. This function configures the microphone input and speaker output interfaces, including setting the audio power control pin, configuring the PDM input pin of the microphone, initializing the digital audio interface, and starting the audio output channel. Thus, the development board gains the ability to record and play sounds.

The entire initialization process is mainly to prepare for the subsequent audio acquisition and playback via I2S on the ESP32-P4.

6. Loop Section

This code defines the loop() function in the Arduino program, which serves as the main execution loop that runs continuously after initialization is complete.



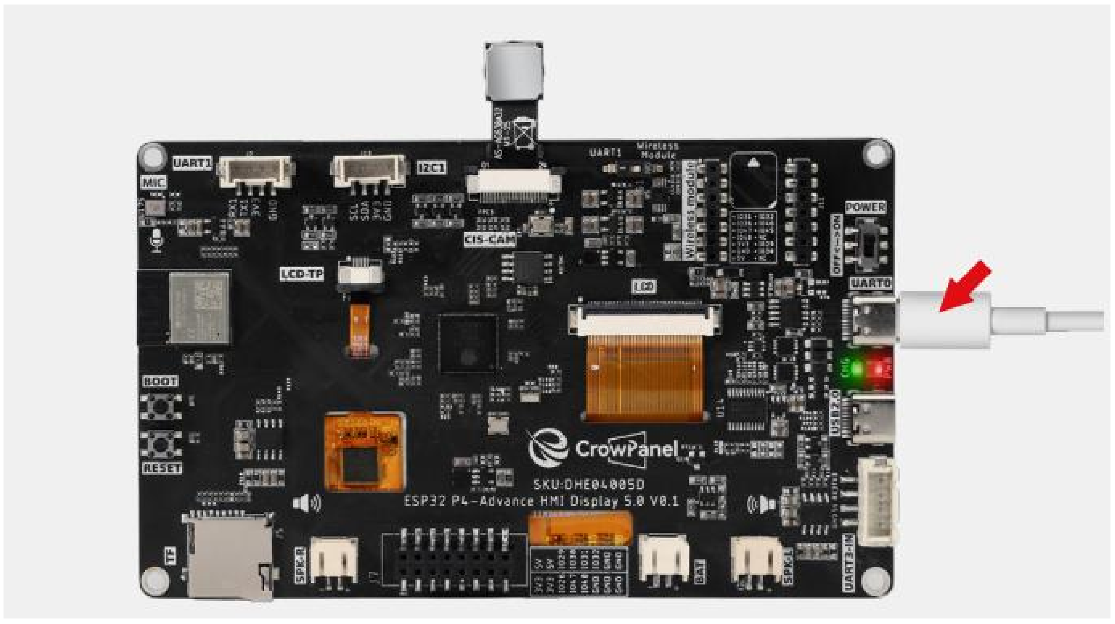
```
void loop() {
  // put your main code here, to run repeatedly:

  recording();
}
```

Within this function, the program invokes the "recording()" function once per loop iteration, thereby triggering the complete audio processing workflow: first capturing a segment of audio data through the microphone, then performing simple volume gain calculation and amplification on the acquired audio, and finally playing back the processed audio through the speaker.

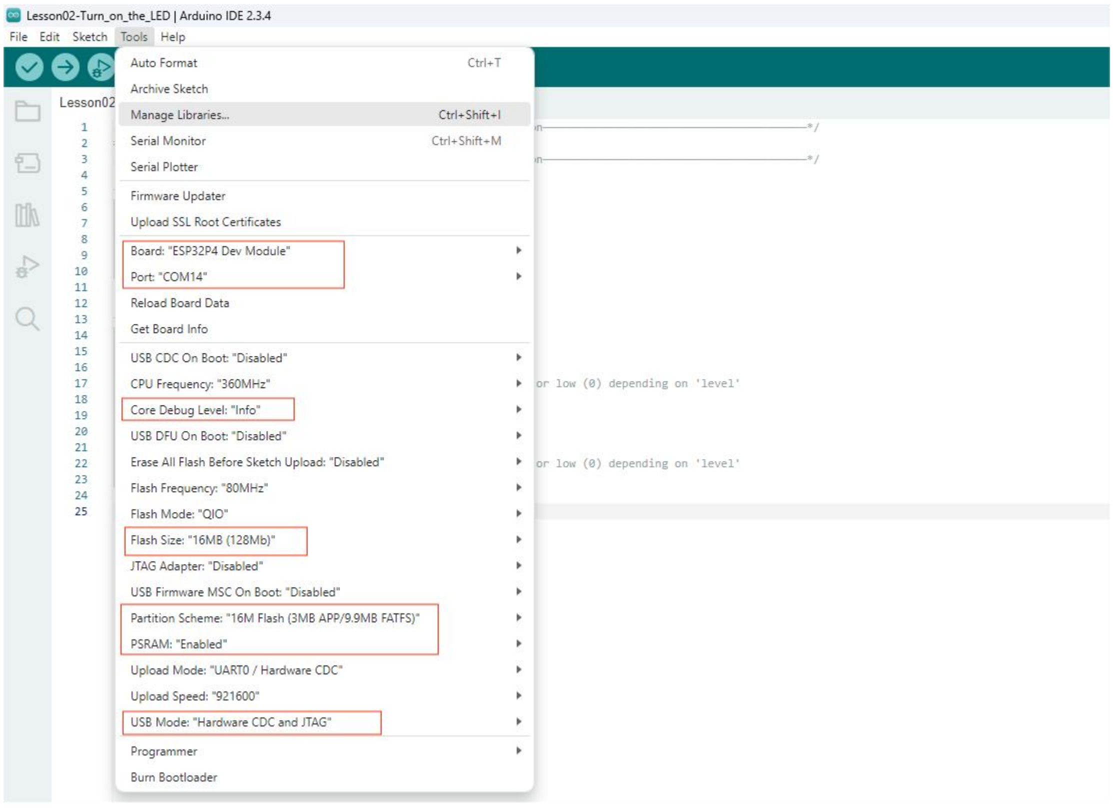
Since "loop()" executes continuously throughout program runtime, the system repeatedly performs the recording → audio processing → speaker playback sequence—forming a real-time audio monitoring system that enables the device to continuously perform audio acquisition and playback operations via I2S on the ESP32-P4.

Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.

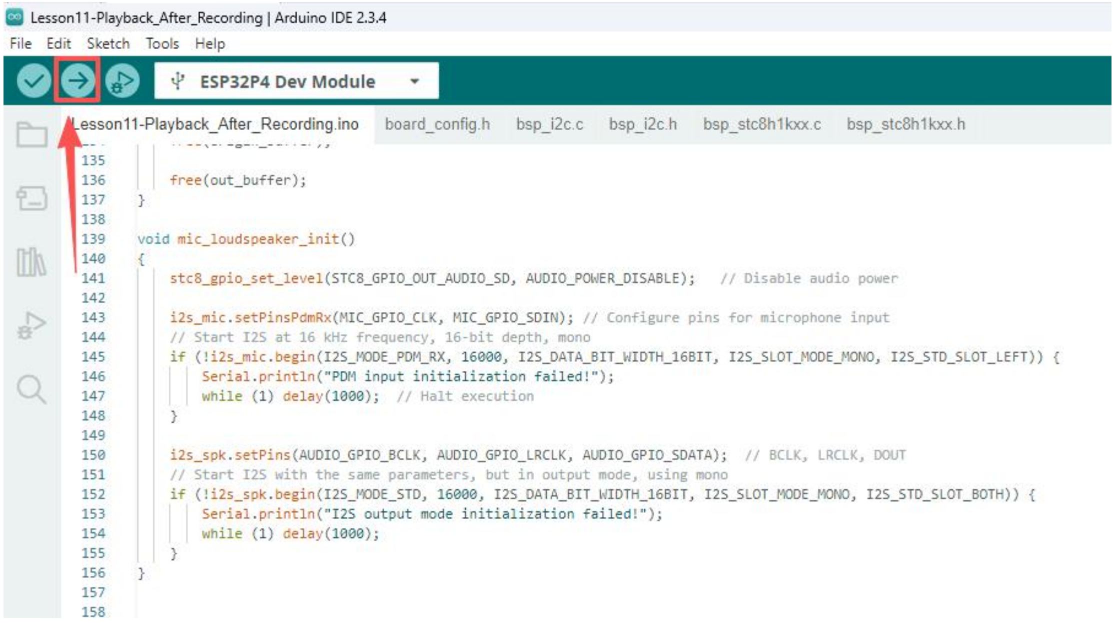


Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4
File Edit Sketch Tools Help
Lesson02

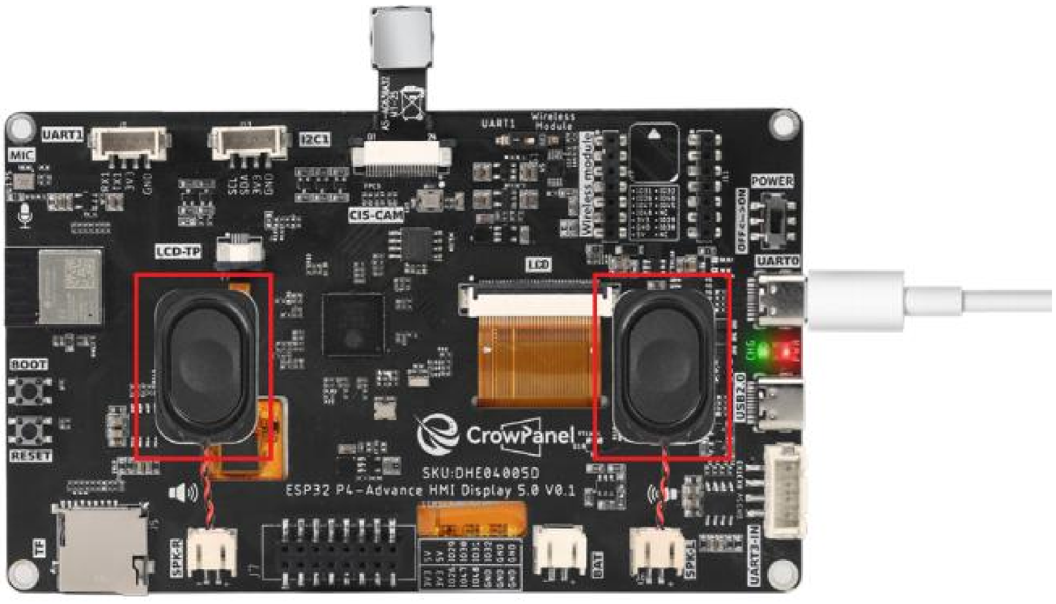
```
Auto Format                                   Ctrl+T
Archive Sketch
Manage Libraries...                           Ctrl+Shift+I
Serial Monitor                                Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader
```

Then we compile and upload the code.


Lesson11-Playback_After_Recording | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson11-Playback_After_Recording.ino
board_config.h
bsp_i2c.c
bsp_i2c.h
bsp_stc8h1kxx.c
bsp_stc8h1kxx.h

```
        free(out_buffer);
    }

void mic_loudspeaker_init()
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);   // Disable audio power

    i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN); // Configure pins for microphone input
    // Start I2S at 16 kHz frequency, 16-bit depth, mono
    if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
        Serial.println("PDM input initialization failed!");
        while (1) delay(1000);  // Halt execution
    }

    i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  // BCLK, LRCLK, DOUT
    // Start I2S with the same parameters, but in output mode, using mono
    if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_BOTH)) {
        Serial.println("I2S output mode initialization failed!");
        while (1) delay(1000);
    }
}
```

Once the flashing is successful, you can speak near the Advance-P4 device. The Advance-P4 will use its microphone to record the current sound within 5 seconds, and then play it back automatically.



The 5-second recorded audio is now playing.

Lesson12--- Playing Local Music from SD Card

Introduction

This lesson will integrate previous coursework by combining SD card reading with speaker output to implement playback of WAV music files stored on the SD card.

Learning Goals

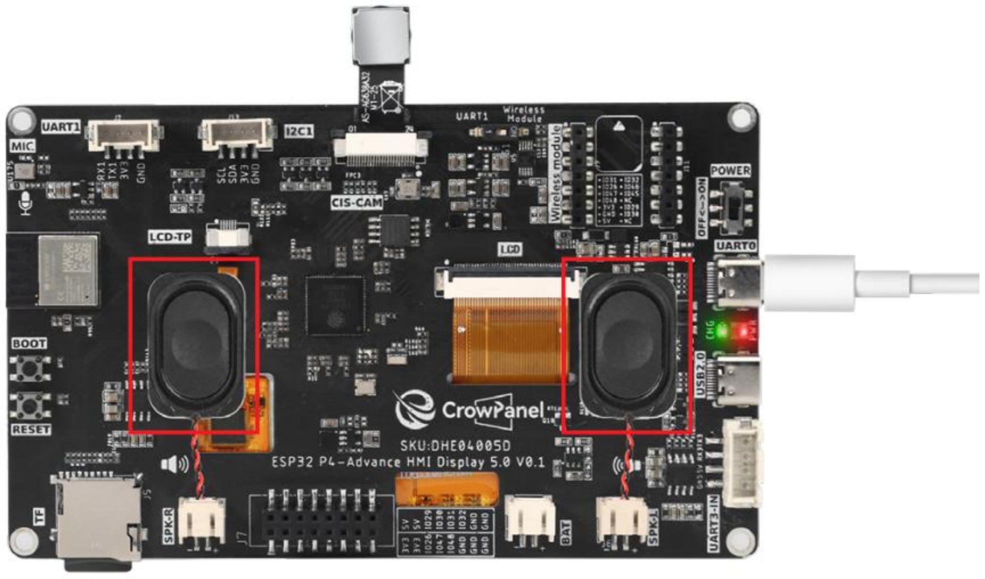
1. Understand the collaborative working principles of SD card file reading and I2S audio playback on the ESP32-P4 development board, and master the structure and validation rules of standard WAV file headers.

2. Master the implementation methods for reading WAV audio files from the SD card (skipping file headers, chunked data reading) and outputting to the speaker via the I2S interface.

3. Master the operation of FFmpeg tool to convert MP3 to WAV files with specified parameters (16kHz, 16-bit, stereo), adapting to the development board's audio playback requirements.

Preview of the Result

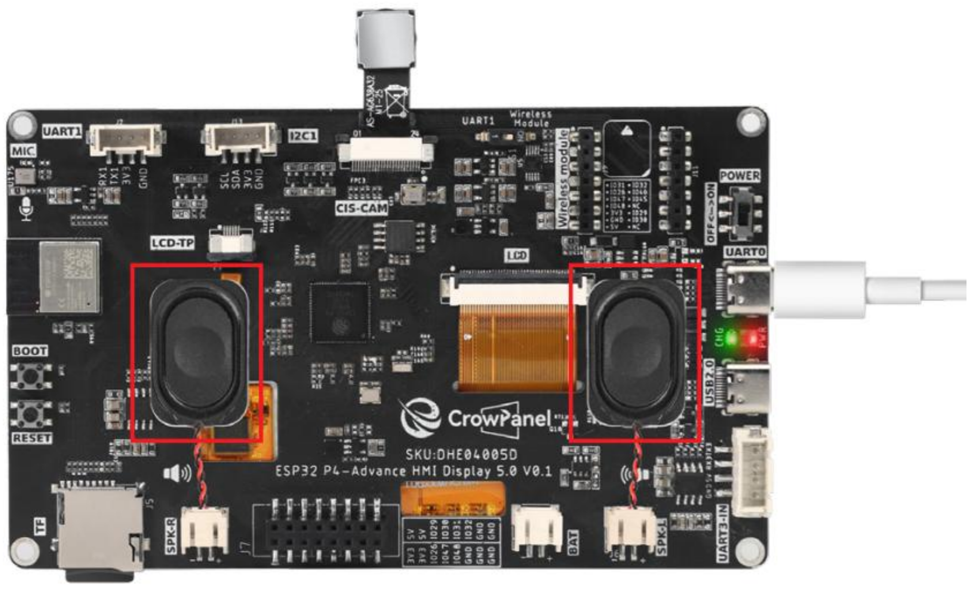
After running the code, you will be able to hear the WAV audio saved in your SD card playing through the speaker on the Advance-P4.



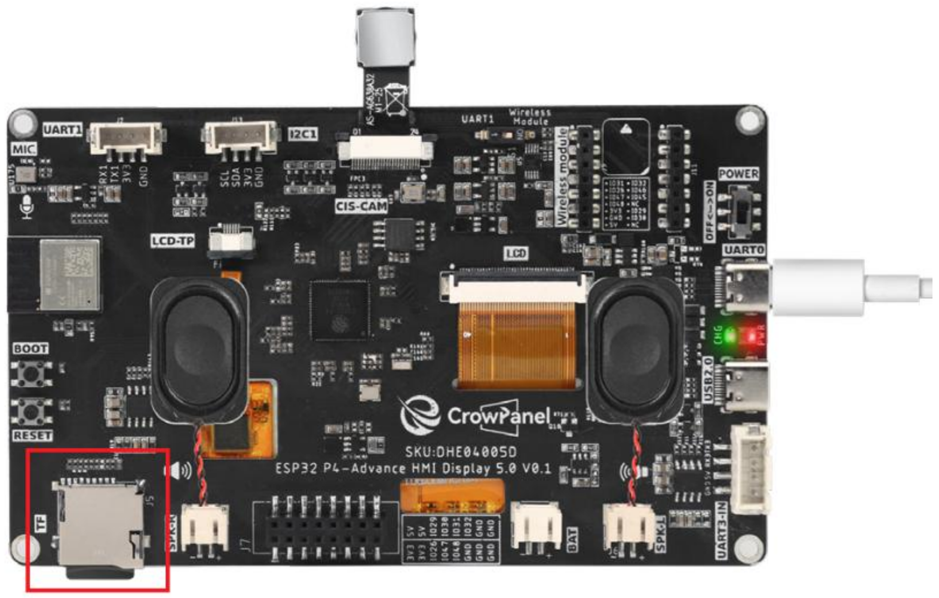
The WAV audio file from your SD card is now playing.

Hardware Used in This Lesson

Speaker on the Advance-P4



SD Card on the Advance-P4



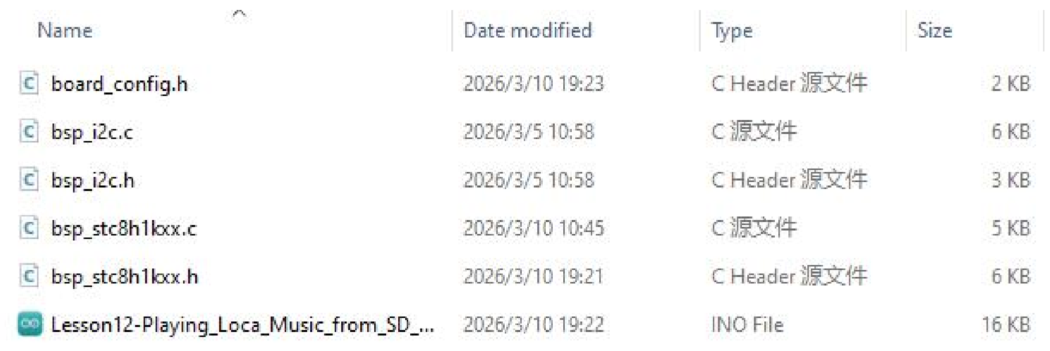
Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson12-Playing_Loca_Music_from_SD_Card

Key Explanations

Now let's examine how to integrate SD card reading functionality with speaker playback capabilities.

Double-click to open the code for this lesson (the .ino file).



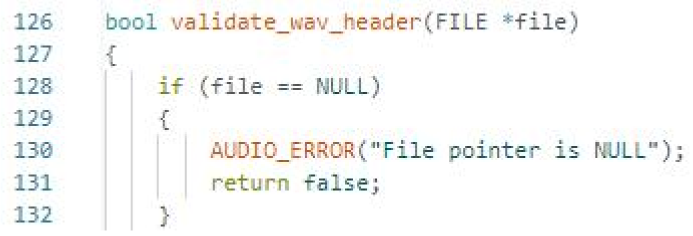
Name	Date modified	Type	Size
board_config.h	2026/3/10 19:23	C Header 源文件	2 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/10 19:21	C Header 源文件	6 KB
Lesson12-Playing_Loca_Music_from_SD_...	2026/3/10 19:22	INO File	16 KB

Once opened, you will see the project code along with the configuration file board_config.h.
In this project, board_config.h contains the pin definitions and configurations for both the SD card and speaker.
We will primarily examine how the functionality is implemented in the main .ino code file.
Much of the code here has been covered in previous lessons—for example, SD card reading-related code was explained in detail in Lesson 8, and speaker playback initialization code was thoroughly covered in Lesson 11.
Therefore, moving forward, we will only explain code that hasn't been previously covered, helping you learn with greater precision.

1. validate_wav_header

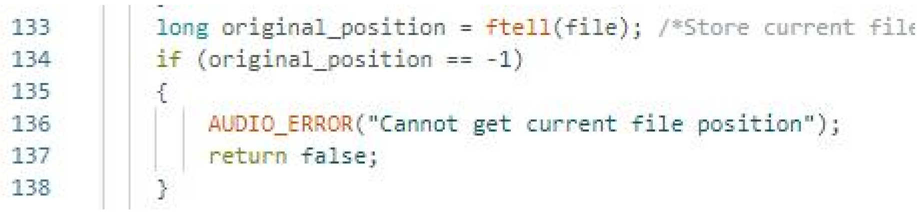
This code implements a function validate_wav_header(FILE *file) for detecting whether a WAV audio file header conforms to standard format. Its primary role is to perform comprehensive format validation on WAV files from the SD card before playback begins—ensuring the file is a standard PCM WAV audio file that can be correctly parsed and played by the system, thereby avoiding playback failures or program exceptions due to file corruption or format incompatibility.

At the start of the function, it first checks whether the passed file pointer file is null. If null, this indicates the file was not properly opened; the function then outputs an error message via the logging macro and returns false to terminate the check.



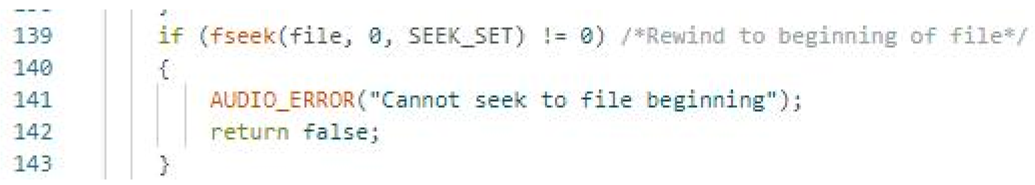
```
bool validate_wav_header(FILE *file)
{
    if (file == NULL)
    {
        AUDIO_ERROR("File pointer is NULL");
        return false;
    }
```

If the file pointer is valid, the function retrieves the current file pointer position via "ftell(file)" and saves it to the variable "original_position". This allows the file pointer to be restored to its original position after the check completes, preventing interference with subsequent file reading operations.



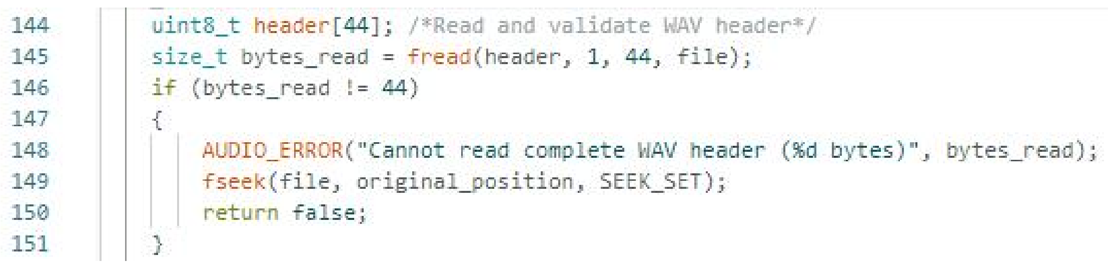
```
    long original_position = ftell(file); /*Store current fil
    if (original_position == -1)
    {
        AUDIO_ERROR("Cannot get current file position");
        return false;
    }
```

The program then uses "fseek(file, 0, SEEK_SET)" to move the file pointer to the beginning of the file, since the WAV file header is located at the very start. It then defines a buffer "uint8_t header[44]" and reads 44 bytes of header data from the file via "fread()"—the standard length of a PCM WAV file header.



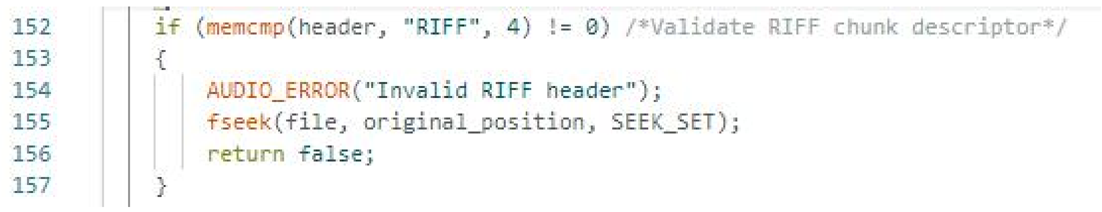
```
    if (fseek(file, 0, SEEK_SET) != 0) /*Rewind to beginning of file*/
    {
        AUDIO_ERROR("Cannot seek to file beginning");
        return false;
    }
```

If the number of bytes read is less than 44, this indicates the file is incomplete or has an abnormal format. The program will output an error message, restore the file pointer position, and return "false".



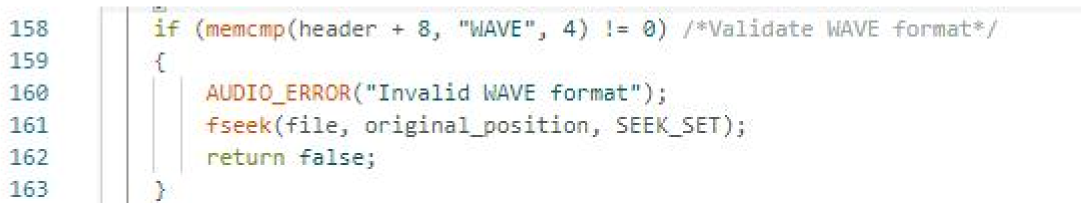
```
    uint8_t header[44]; /*Read and validate WAV header*/
    size_t bytes_read = fread(header, 1, 44, file);
    if (bytes_read != 44)
    {
        AUDIO_ERROR("Cannot read complete WAV header (%d bytes)", bytes_read);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
```

After successfully reading the header data, the function proceeds to check the WAV file structure item by item: it first uses "memcmp(header, "RIFF", 4)" to verify whether the file begins with the "RIFF" identifier—the RIFF chunk signature of WAV files.



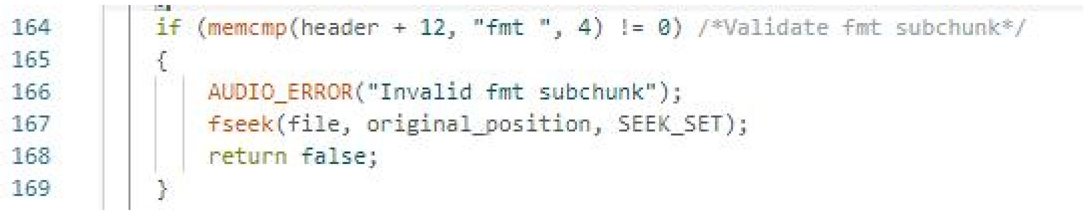
```
    if (memcmp(header, "RIFF", 4) != 0) /*Validate RIFF chunk descriptor*/
    {
        AUDIO_ERROR("Invalid RIFF header");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
```

It then checks whether the bytes at offset 8 contain "WAVE" to confirm that the file type is indeed in WAV format.



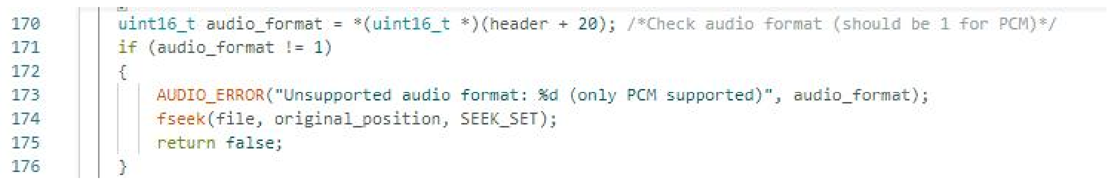
```c
if (memcmp(header + 8, "WAVE", 4) != 0) /*Validate WAVE format*/
{
    AUDIO_ERROR("Invalid WAVE format");
    fseek(file, original_position, SEEK_SET);
    return false;
}
```

It then checks for the "fmt " sub-chunk identifier at offset 12 bytes, indicating the presence of the format information block in the file.



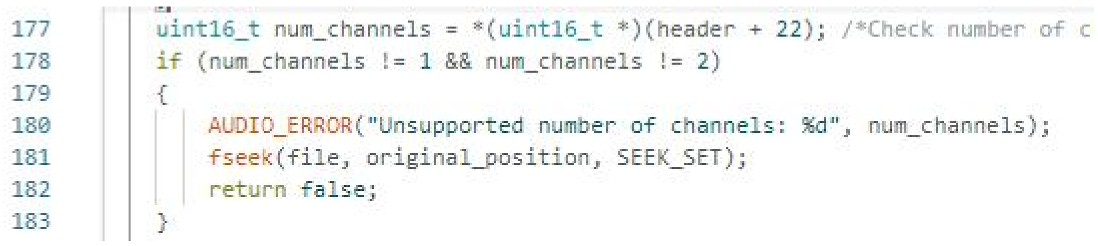
```c
if (memcmp(header + 12, "fmt ", 4) != 0) /*Validate fmt subchunk*/
{
    AUDIO_ERROR("Invalid fmt subchunk");
    fseek(file, original_position, SEEK_SET);
    return false;
}
```

The program then parses key audio parameters from the header data—for instance, reading the audio format "audio_format" via "(uint16_t *)(header + 20)" and verifying that it equals 1, since 1 represents PCM (Pulse Code Modulation) uncompressed audio format—the most common and readily playable format.



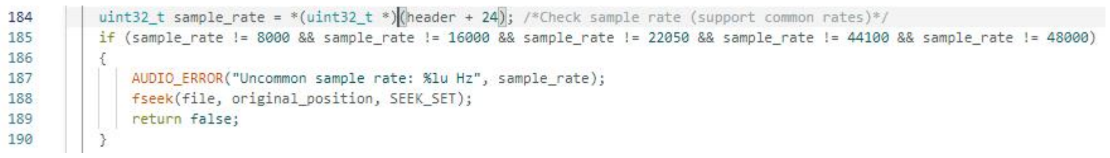
```c
uint16_t audio_format = *(uint16_t *)(header + 20); /*Check audio format (should be 1 for PCM)*/
if (audio_format != 1)
{
    AUDIO_ERROR("Unsupported audio format: %d (only PCM supported)", audio_format);
    fseek(file, original_position, SEEK_SET);
    return false;
}
```

It then reads "(header + 22)" to obtain the number of channels "num_channels", verifying that it is either 1 (mono) or 2 (stereo).



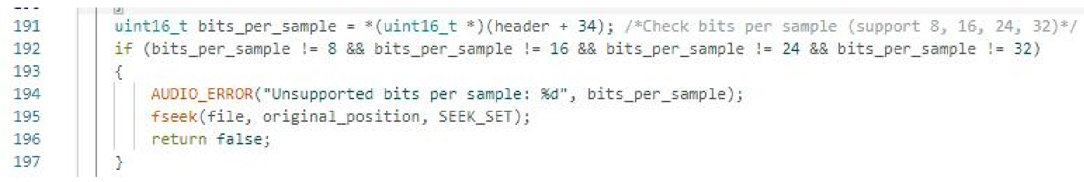
```c
uint16_t num_channels = *(uint16_t *)(header + 22); /*Check number of c
if (num_channels != 1 && num_channels != 2)
{
    AUDIO_ERROR("Unsupported number of channels: %d", num_channels);
    fseek(file, original_position, SEEK_SET);
    return false;
}
```

It then reads "(header + 24)" to obtain the sample rate "sample_rate", verifying that it belongs to common sampling rates (8000, 16000, 22050, 44100, 48000 Hz).



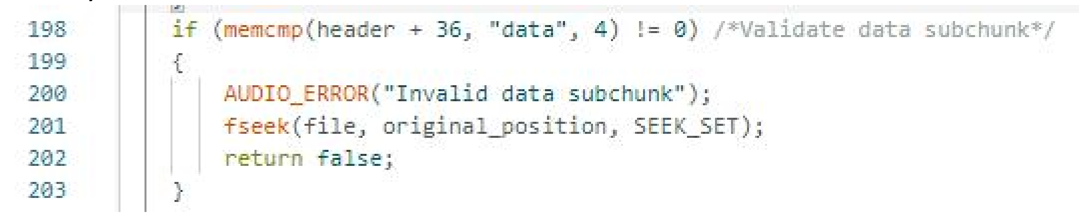
```c
uint32_t sample_rate = *(uint32_t *)(header + 24); /*Check sample rate (support common rates)*/
if (sample_rate != 8000 && sample_rate != 16000 && sample_rate != 22050 && sample_rate != 44100 && sample_rate != 48000)
{
    AUDIO_ERROR("Uncommon sample rate: %lu Hz", sample_rate);
    fseek(file, original_position, SEEK_SET);
    return false;
}
```

It then reads "(header + 34)" to obtain the bits per sample "bits_per_sample", permitting only 8, 16, 24, or 32 bits.



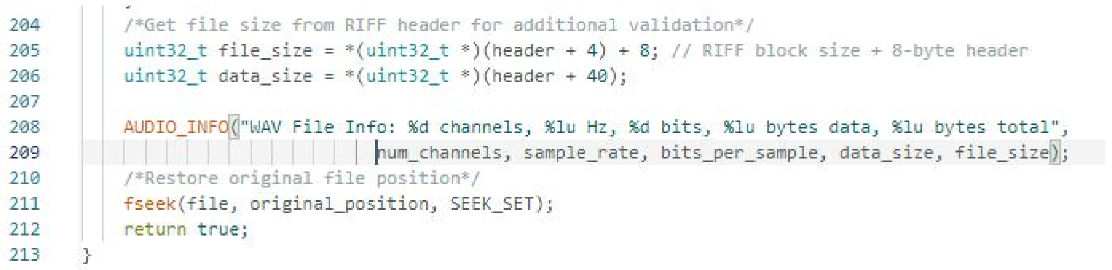
```c
uint16_t bits_per_sample = *(uint16_t *)(header + 34); /*Check bits per sample (support 8, 16, 24, 32)*/
if (bits_per_sample != 8 && bits_per_sample != 16 && bits_per_sample != 24 && bits_per_sample != 32)
{
    AUDIO_ERROR("Unsupported bits per sample: %d", bits_per_sample);
    fseek(file, original_position, SEEK_SET);
    return false;
}
```

Upon completing these parameter checks, the program further verifies the presence of the "data" identifier at offset 36 bytes to confirm that the audio data chunk actually exists.



```
    if (memcmp(header + 36, "data", 4) != 0) /*Validate data subchunk*/
    {
        AUDIO_ERROR("Invalid data subchunk");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
```

Should any of these verification steps fail, the function prints the corresponding error message, restores the file pointer to its original position, and returns "false" to indicate invalid file format. If all checks pass, the function proceeds to read the RIFF chunk size "(header+4)" and data chunk size "(header+40)" from the header to calculate the total file size and audio data size. It then outputs detailed WAV file information via logs—including number of channels, sample rate, bit depth, data size, and total file size—facilitating debugging and audio parameter verification for developers.



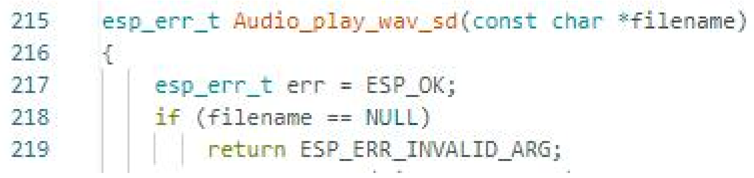
```
    /*Get file size from RIFF header for additional validation*/
    uint32_t file_size = *(uint32_t *)(header + 4) + 8; // RIFF block size + 8-byte header
    uint32_t data_size = *(uint32_t *)(header + 40);

    AUDIO_INFO("WAV File Info: %d channels, %lu Hz, %d bits, %lu bytes data, %lu bytes total",
               num_channels, sample_rate, bits_per_sample, data_size, file_size);
    /*Restore original file position*/
    fseek(file, original_position, SEEK_SET);
    return true;
}
```

Finally, the function restores the file pointer to its position prior to the function call via fseek(file, original_position, SEEK_SET) to ensure subsequent file reading operations are not affected, and returns true—indicating the WAV file header has passed validation and can be safely processed or played.

2.Audio_play_wav_sd

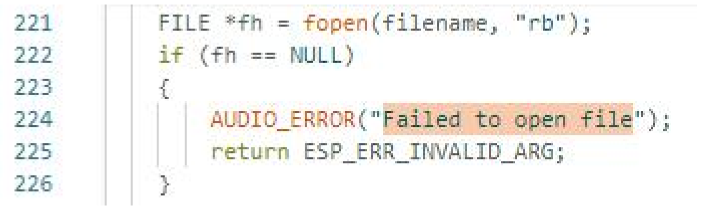
This code implements a function Audio_play_wav_sd(const char *filename) that reads a WAV audio file from the SD card and plays it through the I2S interface to the speaker. The core function of this entire routine is to complete the full workflow of audio file opening, format validation, audio data reading, data processing, and I2S output playback.

At the start of the function, it first declares an esp_err_t err variable initialized to ESP_OK for tracking execution status. It then checks whether the passed file path parameter filename is null; if so, it immediately returns ESP_ERR_INVALID_ARG indicating invalid parameters.



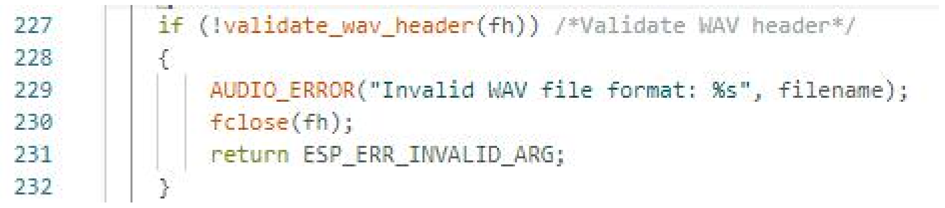
```
esp_err_t Audio_play_wav_sd(const char *filename)
{
    esp_err_t err = ESP_OK;
    if (filename == NULL)
        return ESP_ERR_INVALID_ARG;
```

It then uses "fopen(filename, "rb")" to open the audio file from the SD card in binary read mode. If the file fails to open, it outputs an error log and returns the corresponding error code.



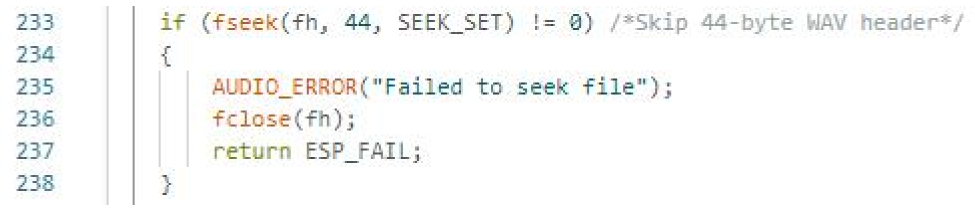
```
FILE *fh = fopen(filename, "rb");
if (fh == NULL)
{
    AUDIO_ERROR("Failed to open file");
    return ESP_ERR_INVALID_ARG;
}
```

Once the file is successfully opened, the function invokes "validate_wav_header(fh)" to validate the WAV file header—confirming whether the file conforms to the required PCM WAV audio format. If validation fails, indicating incorrect file format, the program closes the file and returns an error.



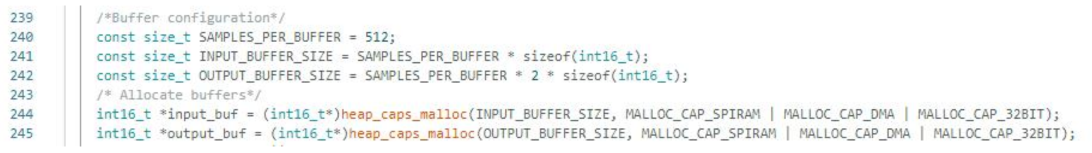
```
if (!validate_wav_header(fh)) /*Validate WAV header*/
{
    AUDIO_ERROR("Invalid WAV file format: %s", filename);
    fclose(fh);
    return ESP_ERR_INVALID_ARG;
}
```

Upon successful validation, the function uses "fseek(fh, 44, SEEK_SET)" to move the file pointer to byte position 44, since the first 44 bytes of a standard WAV file constitute the header information—the actual audio sample data begins after byte 44.



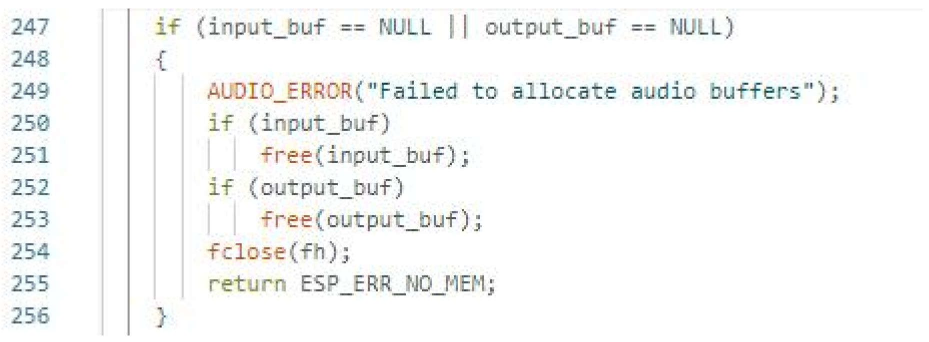
```
if (fseek(fh, 44, SEEK_SET) != 0) /*Skip 44-byte WAV header*/
{
    AUDIO_ERROR("Failed to seek file");
    fclose(fh);
    return ESP_FAIL;
}
```

The program then configures the audio buffer size, defining 512 samples per read ("SAMPLES_PER_BUFFER"), and calculates the sizes for the input buffer "INPUT_BUFFER_SIZE" and output buffer "OUTPUT_BUFFER_SIZE". It then dynamically allocates two buffers "input_buf" and "output_buf" via "heap_caps_malloc()" in memory regions supporting DMA and external PSRAM—ensuring efficient data access during I2S transmission. Should memory allocation fail, it releases already allocated resources and returns an out-of-memory error.



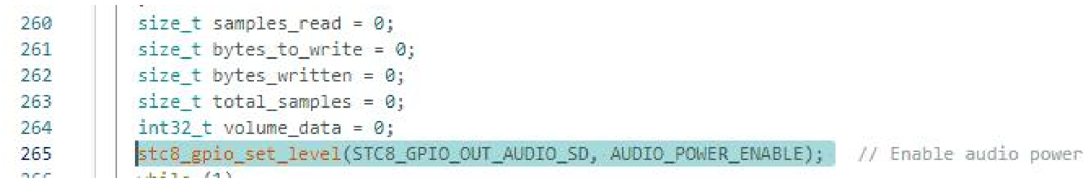
```
/*Buffer configuration*/
const size_t SAMPLES_PER_BUFFER = 512;
const size_t INPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * sizeof(int16_t);
const size_t OUTPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * 2 * sizeof(int16_t);
/* Allocate buffers*/
int16_t *input_buf = (int16_t*)heap_caps_malloc(INPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM | MALLOC_CAP_DMA | MALLOC_CAP_32BIT);
int16_t *output_buf = (int16_t*)heap_caps_malloc(OUTPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM | MALLOC_CAP_DMA | MALLOC_CAP_32BIT);
```

Upon successful buffer creation, the program initializes multiple variables for tracking read sample count, written byte count, and total playback sample count.



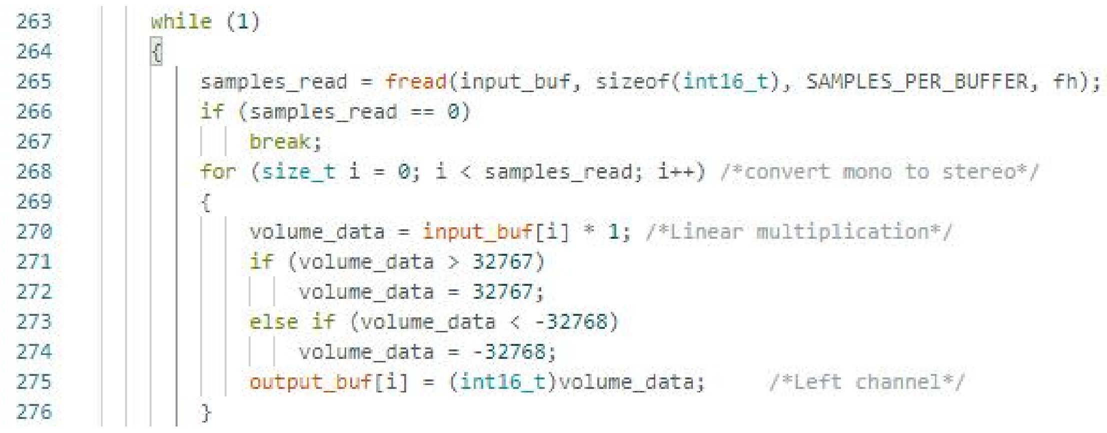
```
if (input_buf == NULL || output_buf == NULL)
{
    AUDIO_ERROR("Failed to allocate audio buffers");
    if (input_buf)
        free(input_buf);
    if (output_buf)
        free(output_buf);
    fclose(fh);
    return ESP_ERR_NO_MEM;
}
```

And the audio power is turned on through "stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_ENABLE);", allowing the power amplifier or audio circuit to enter the working state.



```
size_t samples_read = 0;
size_t bytes_to_write = 0;
size_t bytes_written = 0;
size_t total_samples = 0;
int32_t volume_data = 0;
stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_ENABLE);   // Enable audio power
```

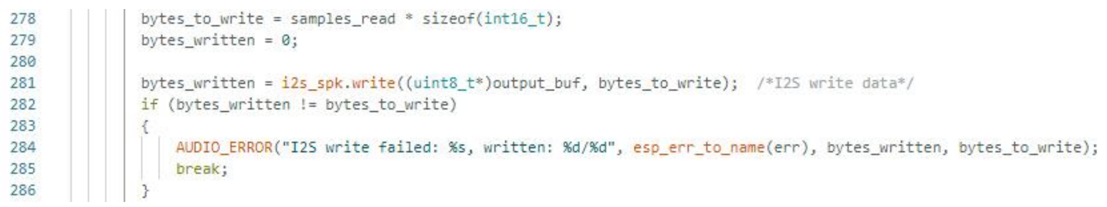
The program then enters a "while(1)" loop, continuously reading 512 "int16_t" audio samples from the file into "input_buf" using "fread()". If the number of samples read is 0, indicating end-of-file has been reached, the loop exits.



```
while (1)
{
    samples_read = fread(input_buf, sizeof(int16_t), SAMPLES_PER_BUFFER, fh);
    if (samples_read == 0)
        break;
    for (size_t i = 0; i < samples_read; i++) /*convert mono to stereo*/
    {
        volume_data = input_buf[i] * 1; /*Linear multiplication*/
        if (volume_data > 32767)
            volume_data = 32767;
        else if (volume_data < -32768)
            volume_data = -32768;
        output_buf[i] = (int16_t)volume_data;     /*Left channel*/
    }
```

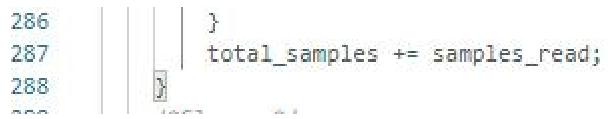
For each sample value read, the program performs simple volume processing within a "for" loop—applying linear gain amplification via "volume_data = input_buf[i] * 1" (current gain factor is 1, indicating no volume change). The result is then constrained within upper and lower bounds to ensure the data remains within the 16-bit audio permitted range (-32768 to 32767), preventing overflow. The processed data is then written to "output_buf" (the current code only writes left channel data).

Upon completing data processing, the program calculates the number of bytes to be written to I2S, and invokes "i2s_spk.write((uint8_t*)output_buf, bytes_to_write)" to transmit the audio data to the speaker via the I2S interface. If the actual number of bytes written does not match the expected value, this indicates an I2S write issue—the program prints an error log and exits the loop.



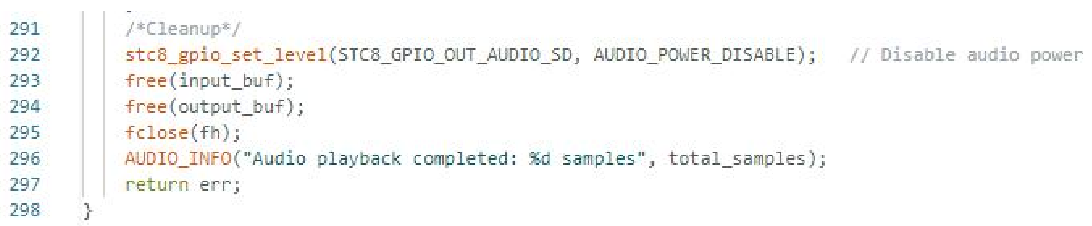
```
bytes_to_write = samples_read * sizeof(int16_t);
bytes_written = 0;

bytes_written = i2s_spk.write((uint8_t*)output_buf, bytes_to_write);  /*I2S write data*/
if (bytes_written != bytes_to_write)
{
    AUDIO_ERROR("I2S write failed: %s, written: %d/%d", esp_err_to_name(err), bytes_written, bytes_to_write);
    break;
}
```

After each successful data transmission, the program accumulates total_samples to track the number of audio samples played.



```
    }
    total_samples += samples_read;
}
```

After the loop ends, the function enters the cleanup stage. It disables the audio power by calling "stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);", then releases the memory previously allocated for "input_buf" and "output_buf", and finally closes the file handle "fh".

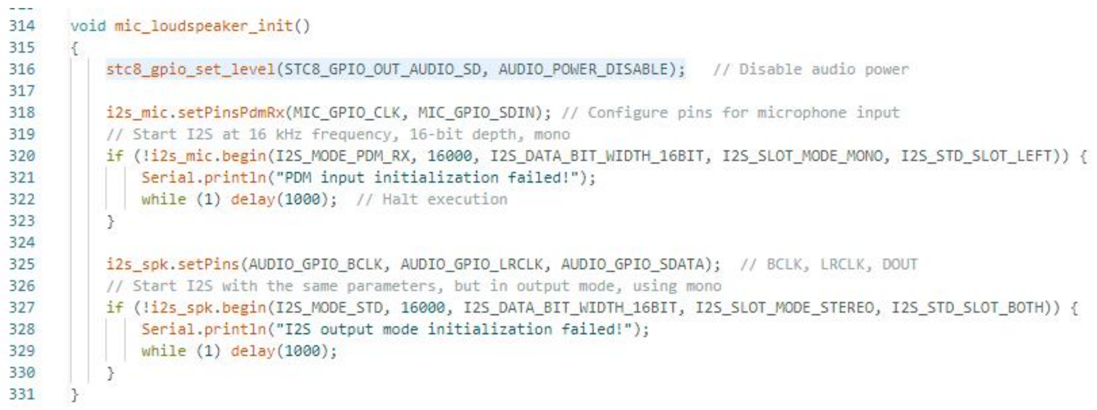


```
    /*Cleanup*/
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);   // Disable audio power
    free(input_buf);
    free(output_buf);
    fclose(fh);
    AUDIO_INFO("Audio playback completed: %d samples", total_samples);
    return err;
}
```

Finally, output a log message indicating that the audio playback is complete and the total number of played samples, and return the function execution result "err", indicating that the entire WAV audio file has been successfully read from the SD card and played through the I2S interface.

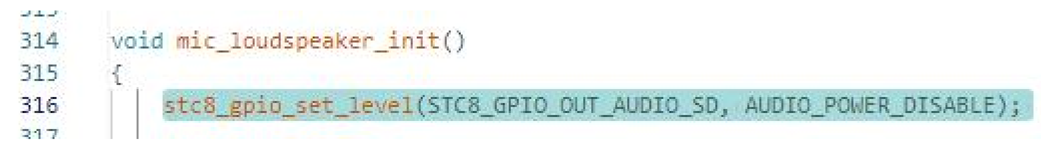
3.mic_loudspeaker_init

This code implements an audio device initialization function named mic_loudspeaker_init(), whose main function is to complete the I2S audio initialization of the microphone input interface and the speaker output interface during system startup. At the same time, it controls the audio power status to prepare for subsequent audio acquisition or playback functions.



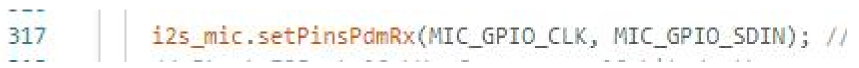
```
void mic_loudspeaker_init()
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);   // Disable audio power

    i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN); // Configure pins for microphone input
    // Start I2S at 16 kHz frequency, 16-bit depth, mono
    if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
        Serial.println("PDM input initialization failed!");
        while (1) delay(1000);  // Halt execution
    }

    i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  // BCLK, LRCLK, DOUT
    // Start I2S with the same parameters, but in output mode, using mono
    if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_STEREO, I2S_STD_SLOT_BOTH)) {
        Serial.println("I2S output mode initialization failed!");
        while (1) delay(1000);
    }
}
```

At the beginning of the function, this pin is usually used to control the power switch of the audio amplifier or the audio circuit. Then, by using the statement stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);, this pin is set to the off state, that is, the audio power is turned off at the system initialization stage to avoid noise or abnormal signals from the speaker during the initialization process.



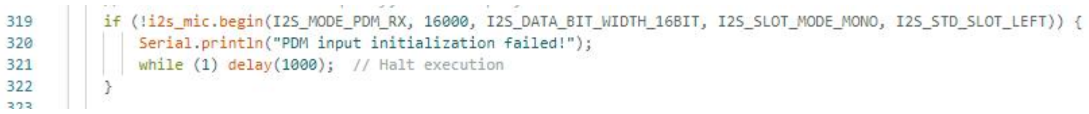
```
void mic_loudspeaker_init()
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, AUDIO_POWER_DISABLE);
```

The program then proceeds to initialize the microphone interface. It first invokes "i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN)" to configure the I2S receive pins used by the PDM microphone—where "MIC_GPIO_CLK" outputs the PDM clock signal and "MIC_GPIO_SDIN" receives the microphone's data input signal.



```
    i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN); //
```

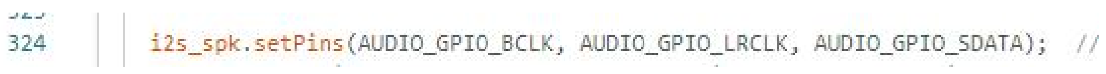
It subsequently calls i2s_mic.begin() to launch the I2S receive module and configure audio parameters: I2S_MODE_PDM_RX indicates PDM (Pulse Density Modulation) mode for audio acquisition, sample rate set to 16000 Hz (16 kHz), data bit width of 16 bits, channel mode as mono (I2S_SLOT_MODE_MONO), with I2S_STD_SLOT_LEFT specifying left channel data usage.



```
    if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
        Serial.println("PDM input initialization failed!");
        while (1) delay(1000);  // Halt execution
    }
```

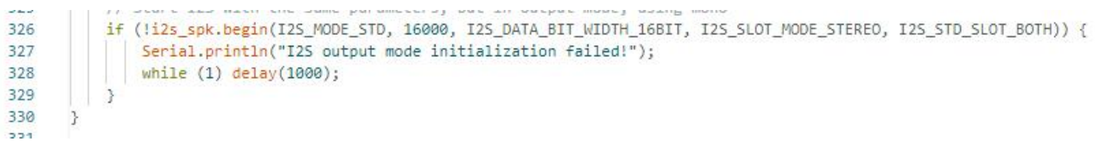
Should the microphone I2S initialization fail, the program prints the error message "PDM input initialization failed!" via the serial port and enters a "while(1)" infinite loop with "delay(1000)"—halting program execution for one-second intervals to facilitate troubleshooting by the developer.

Upon completing microphone initialization, the function continues with speaker output interface initialization. It first invokes "i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA)" to configure the three critical pins for I2S audio output—where "AUDIO_GPIO_BCLK" is the Bit Clock, "AUDIO_GPIO_LRCLK" is the Left/Right Clock (also known as WS), and "AUDIO_GPIO_SDATA" is the audio data output pin.



```
    i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  //
```

It then calls "i2s_spk.begin()" to launch the I2S output module with audio parameters similar to the input: sample rate likewise at 16000 Hz, bit width of 16 bits, but channel mode set to stereo ("I2S_SLOT_MODE_STEREO"), with "I2S_STD_SLOT_BOTH" indicating simultaneous use of both left and right channels for audio output.

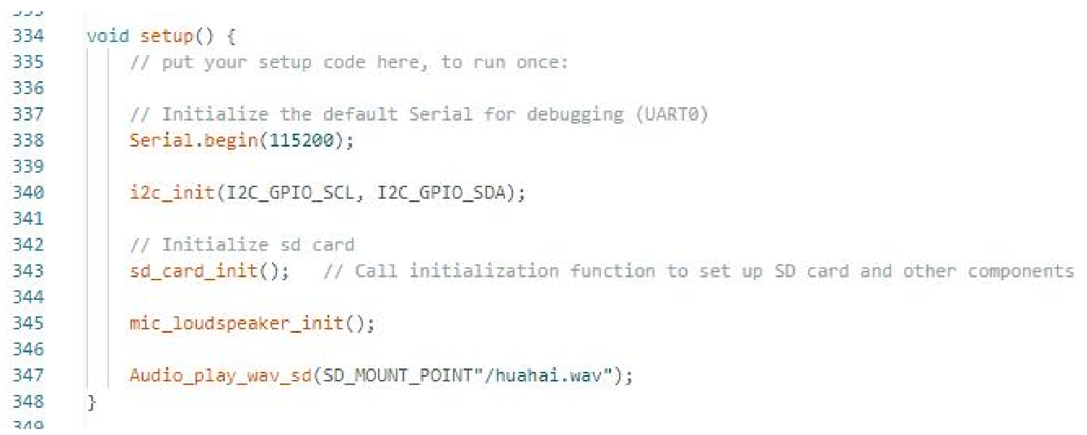


```
    if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_STEREO, I2S_STD_SLOT_BOTH)) {
        Serial.println("I2S output mode initialization failed!");
        while (1) delay(1000);
    }
}
```

Should the I2S speaker initialization fail, the program similarly prints the error message "I2S output mode initialization failed!" and enters an infinite loop to halt program execution.

4.setup

This code implements the setup() initialization function of the Arduino program. Its main function is to perform a system initialization, I2C initialization, and complete the serial debugging interface, SD card storage system, and audio input/output device initialization after the device is powered on or reset. At the same time, it reads and plays a WAV audio file from the SD card.



```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);

    // Initialize sd card
    sd_card_init();   // Call initialization function to set up SD card and other components

    mic_loudspeaker_init();

    Audio_play_wav_sd(SD_MOUNT_POINT"/huahai.wav");
}
```

At the beginning of the function, Serial.begin(115200) is called to initialize the default serial port (UART0) and set the communication baud rate to 115200. This enables developers to view debugging information, error messages and log outputs during the program's execution through the serial monitor, facilitating development and problem resolution.

Subsequently, the function i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) is called to initialize the I^2 C communication bus. Here, I2C_GPIO_SCL and I2C_GPIO_SDA respectively represent the clock line and data line pins of the I^2 C. This bus is mainly used for communication with external devices, such as reading battery information through the STC8H1KXX coprocessor, controlling the GPIO state, or managing the audio power supply, etc.

After completing the I^2 C initialization, the program then calls the sd_card_init() function to initialize the SD card module. This function will complete the configuration of the SD card interface, the initialization of SDMMC communication, and the mounting of the FAT file system, enabling the system to access the files on the SD card normally. If the SD card initialization fails, this function usually continuously attempts to reinitialize, ensuring that the system can correctly identify and mount the SD card storage device.

Subsequently, the program calls the function mic_loudspeaker_init() to initialize the audio hardware device. This function is mainly responsible for configuring the I2S interface used by the microphone and the speaker, including setting the corresponding GPIO pins. It also configures the audio sampling rate (such as 16 kHz), data bit width (16 bits), and channel mode (mono or stereo), and initializes the audio power control pins to ensure that the system can correctly perform audio acquisition and audio output.

After completing the initialization of these basic hardware, the program finally calls the function Audio_play_wav_sd(SD_MOUNT_POINT"/huahai.wav"), which reads the WAV audio file named huahai.wav (with the full path of /sdcard/huahai.wav) from the already-mounted SD card file system and sends the audio data through the I2S interface to the speaker for playback.

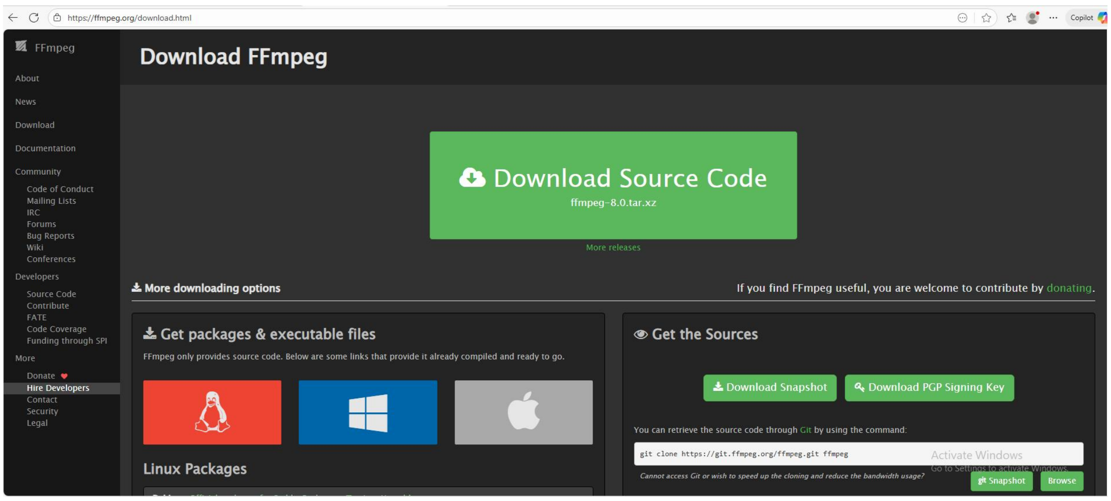
5.Converting MP3 to WAV

As mentioned above, if you want to play audio based on the code of this lesson, the audio must meet the requirement of being a WAV file with 16kHz sampling rate, 16-bit bit depth, and stereo format (i.e., dual-channel).

Next, I will show you how to convert an MP3 audio file to a WAV audio file that meets the specifications of 16kHz, 16-bit, and stereo (dual-channel).

FFmpeg is an open-source toolkit for processing multimedia files such as video and audio. It supports conversion, cutting, and editing of almost all multimedia formats, making it an essential tool for developers and multimedia professionals.

Open the following link to download FFmpeg:
https://ffmpeg.org/download.html



Download FFmpeg

More downloading options

If you find FFmpeg useful, you are welcome to contribute by donating.

Get packages & executable files

FFmpeg only provides source code. Below are some links that provide it already compiled and ready to go.

Linux Packages

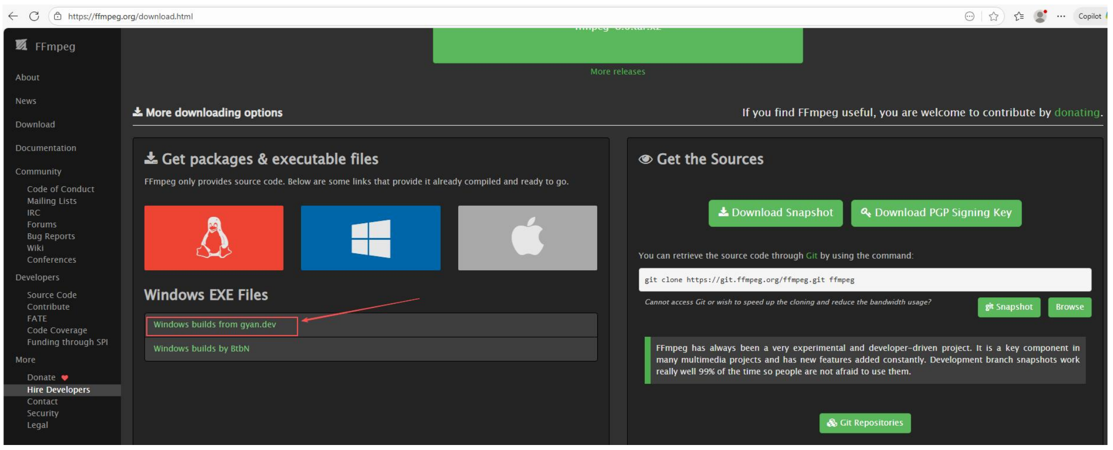
Get the Sources

You can retrieve the source code through Git by using the command:

```
git clone https://git.ffmpeg.org/ffmpeg.git ffmpeg
```

Cannot access Git or wish to speed up the cloning and reduce the bandwidth usage?

Taking Windows as an Example:
Select the installation package "Windows builds from gyan.dev".



If you find FFmpeg useful, you are welcome to contribute by donating.

FFmpeg only provides source code. Below are some links that provide it already compiled and ready to go.

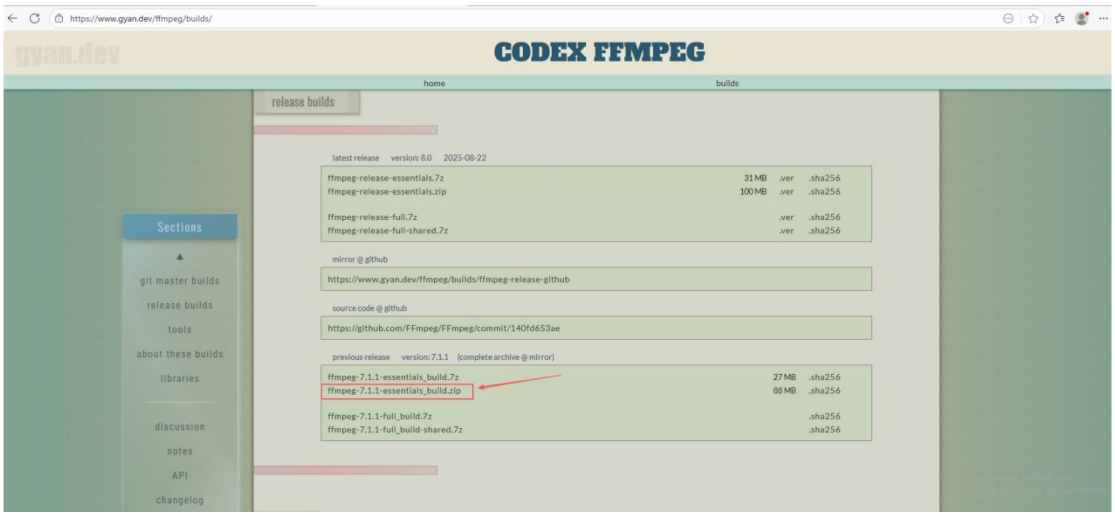
You can retrieve the source code through Git by using the command:

```
git clone https://git.ffmpeg.org/ffmpeg.git ffmpeg
```

Cannot access Git or wish to speed up the cloning and reduce the bandwidth usage?

FFmpeg has always been a very experimental and developer-driven project. It is a key component in many multimedia projects and has new features added constantly. Development branch snapshots work really well 99% of the time so people are not afraid to use them.

Scroll down to find the "release builds" section, then select "ffmpeg-7.1.1-essentials_build.zip".



latest release version: 8.0 2025-08-22

ffmpeg-release-essentials.7z	31 MB	.ver	.sha256
ffmpeg-release-essentials.zip	100 MB	.ver	.sha256
ffmpeg-release-full.7z		.ver	.sha256
ffmpeg-release-full-shared.7z		.ver	.sha256

mirror @ github

https://www.gyan.dev/ffmpeg/builds/ffmpeg-release-github

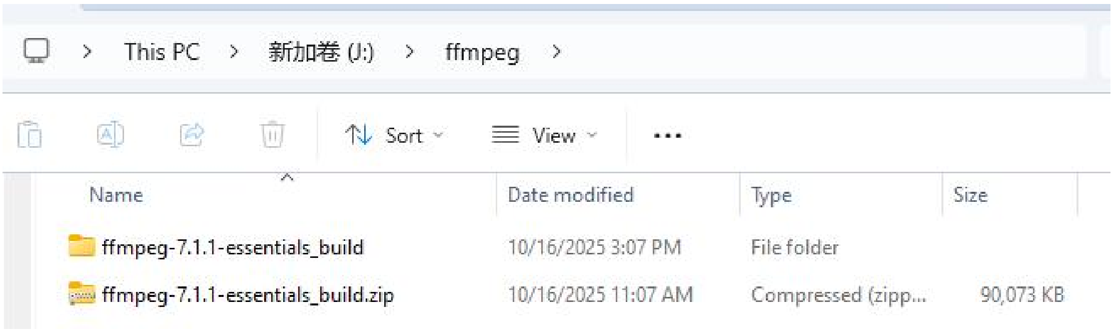
source code @ github

https://github.com/FFmpeg/FFmpeg/commit/140fd653ae

previous release version: 7.1.1 (complete archive @ mirror)

ffmpeg-7.1.1-essentials_build.7z	27 MB	.sha256
ffmpeg-7.1.1-essentials_build.zip	88 MB	.sha256
ffmpeg-7.1.1-full_build.7z		.sha256
ffmpeg-7.1.1-full_build-shared.7z		.sha256

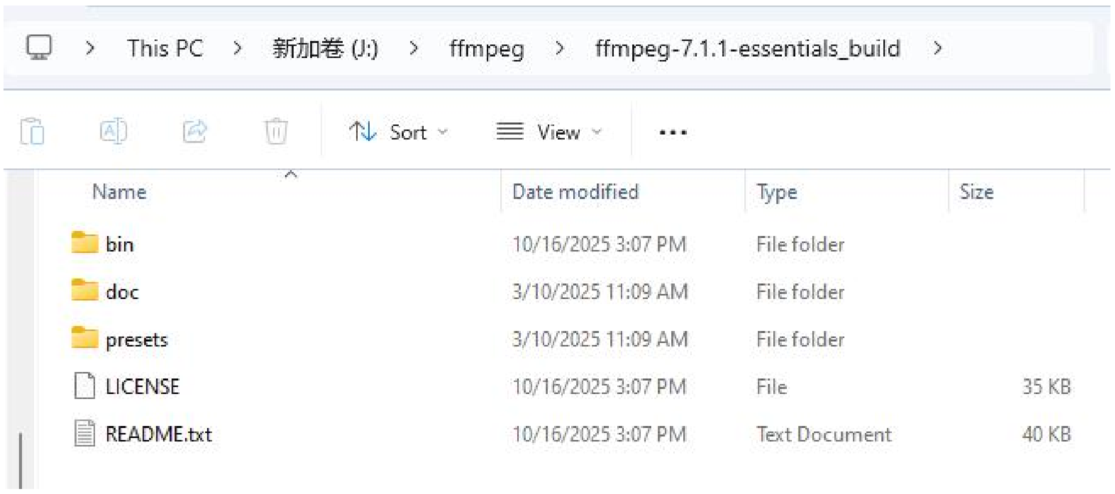
Once the download is complete, extract the file to get the "FFmpeg" folder.



Name	Date modified	Type	Size
ffmpeg-7.1.1-essentials_build	10/16/2025 3:07 PM	File folder	
ffmpeg-7.1.1-essentials_build.zip	10/16/2025 11:07 AM	Compressed (zipp...	90,073 KB

Recommended Saving Path

It is recommended to extract and save the folder to a non-system drive (not the C drive). This avoids occupying space on the C drive (system drive), ensuring the stability and performance of the system.


This PC > 新加卷 (J:) > ffmpeg > ffmpeg-7.1.1-essentials_build >

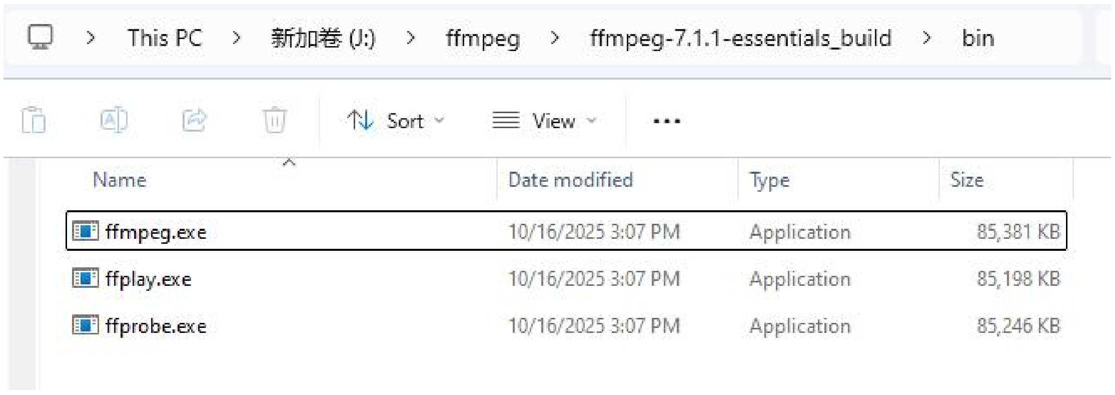
Name	Date modified	Type	Size
bin	10/16/2025 3:07 PM	File folder	
doc	3/10/2025 11:09 AM	File folder	
presets	3/10/2025 11:09 AM	File folder	
LICENSE	10/16/2025 3:07 PM	File	35 KB
README.txt	10/16/2025 3:07 PM	Text Document	40 KB

Directory Structure of the Extracted Folder

The extracted folder should contain the following directories:

- **"bin"**: The folder containing FFmpeg executable files. All commands to run FFmpeg must be executed via the files in this directory.
- **"doc"**: Documentation and reference materials.
- **"presets"**: Preconfigured formats and encoding schemes.

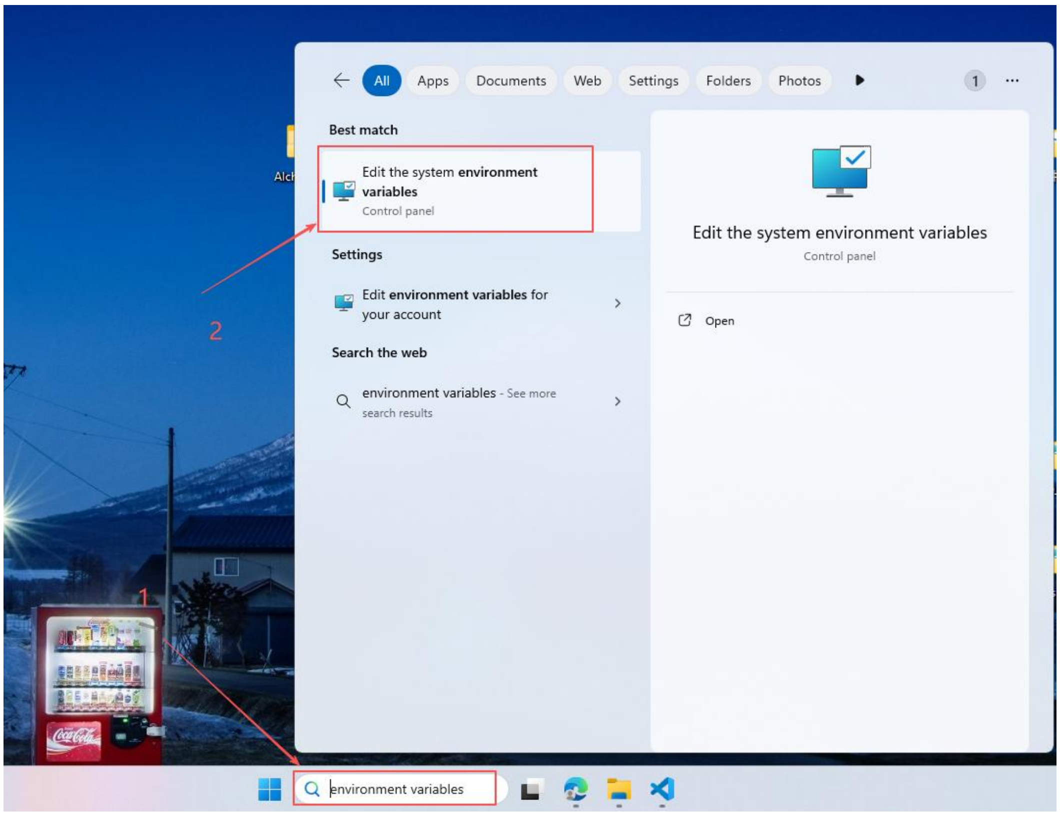
Navigate to the **"bin"** directory, and you will see three core executable files of FFmpeg: **"ffmpeg.exe"**, **"ffplay.exe"**, and **"ffprobe.exe"**.


This PC > 新加卷 (J:) > ffmpeg > ffmpeg-7.1.1-essentials_build > bin

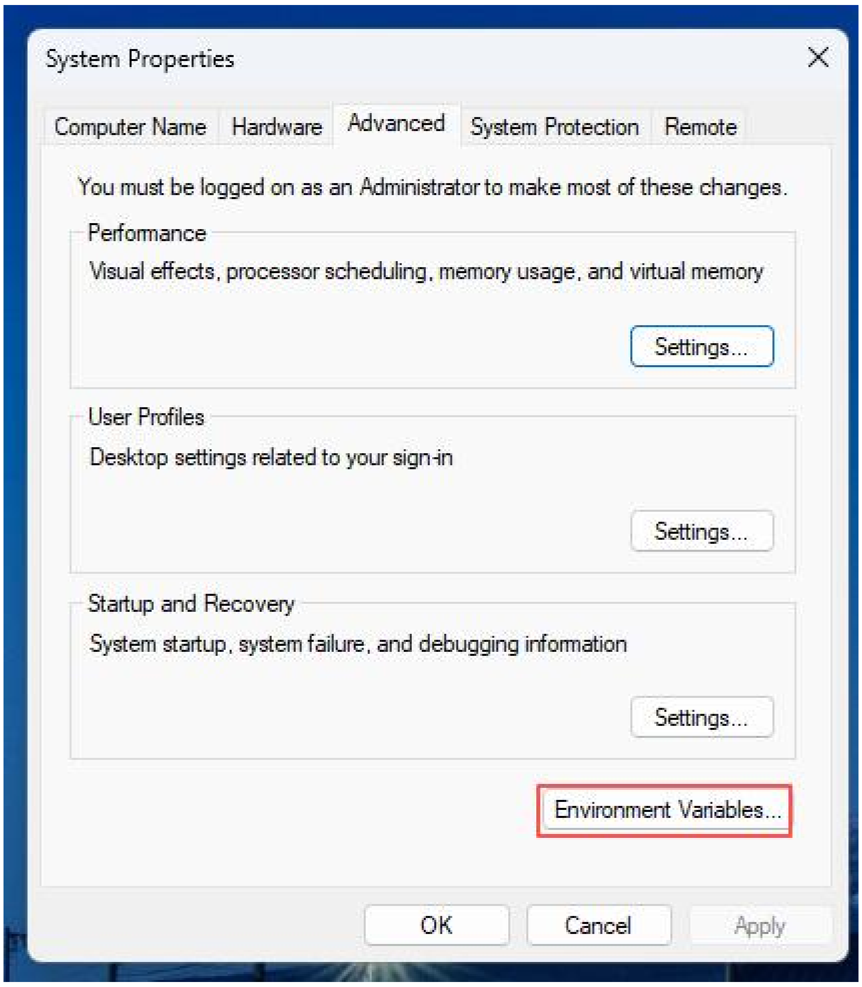
Name	Date modified	Type	Size
ffmpeg.exe	10/16/2025 3:07 PM	Application	85,381 KB
ffplay.exe	10/16/2025 3:07 PM	Application	85,198 KB
ffprobe.exe	10/16/2025 3:07 PM	Application	85,246 KB

To conveniently call FFmpeg directly in the command line, you need to add it to the system's environment variables.

Search for "Environment Variables" in the Start Menu at the bottom left of the desktop, find "Edit the system environment variables", and click to open it.


All
Apps
Documents
Web
Settings
Folders
Photos
Best match
Edit the system environment variables
Control panel
Settings
Edit environment variables for your account
Search the web
environment variables - See more search results
Edit the system environment variables
Control panel
Open
2
1
environment variables

Click the "Environment Variables" button.


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...

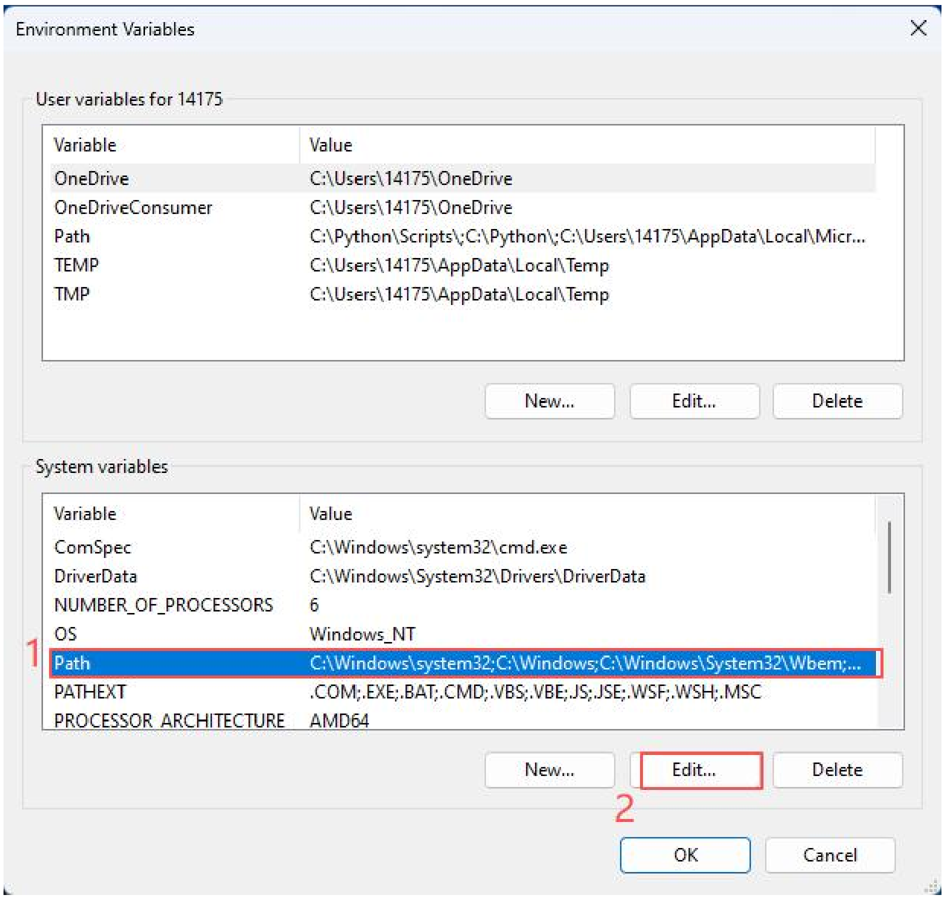
User Profiles
Desktop settings related to your sign-in
Settings...

Startup and Recovery
System startup, system failure, and debugging information
Settings...

Environment Variables...

OK | Cancel | Apply

Locate the "Path" entry under "System Variables" and click "Edit".



Environment Variables

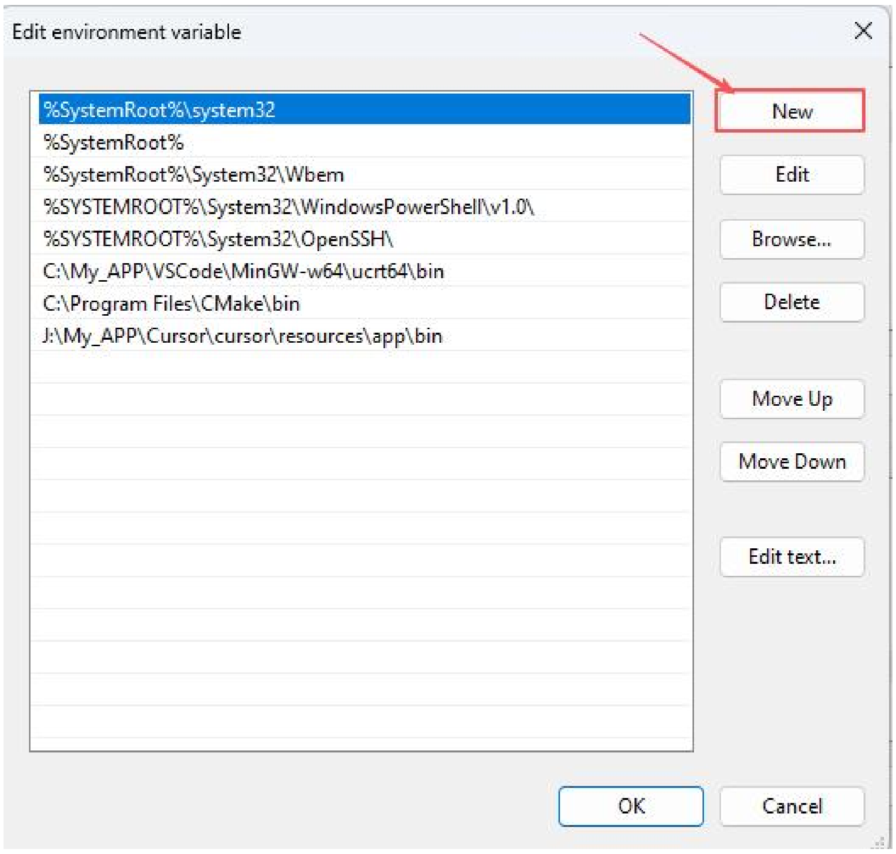
User variables for 14175

Variable	Value
OneDrive	C:\Users\14175\OneDrive
OneDriveConsumer	C:\Users\14175\OneDrive
Path	C:\Python\Scripts\;C:\Python\;C:\Users\14175\AppData\Local\Micr...
TEMP	C:\Users\14175\AppData\Local\Temp
TMP	C:\Users\14175\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	6
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

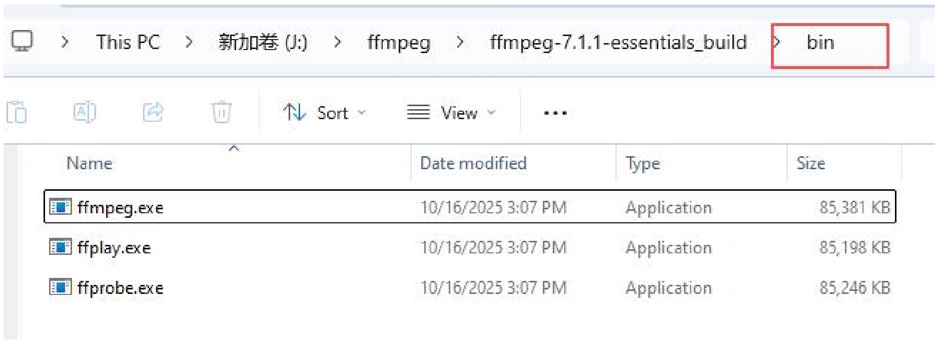
In the "Edit environment variable" window, click "New".



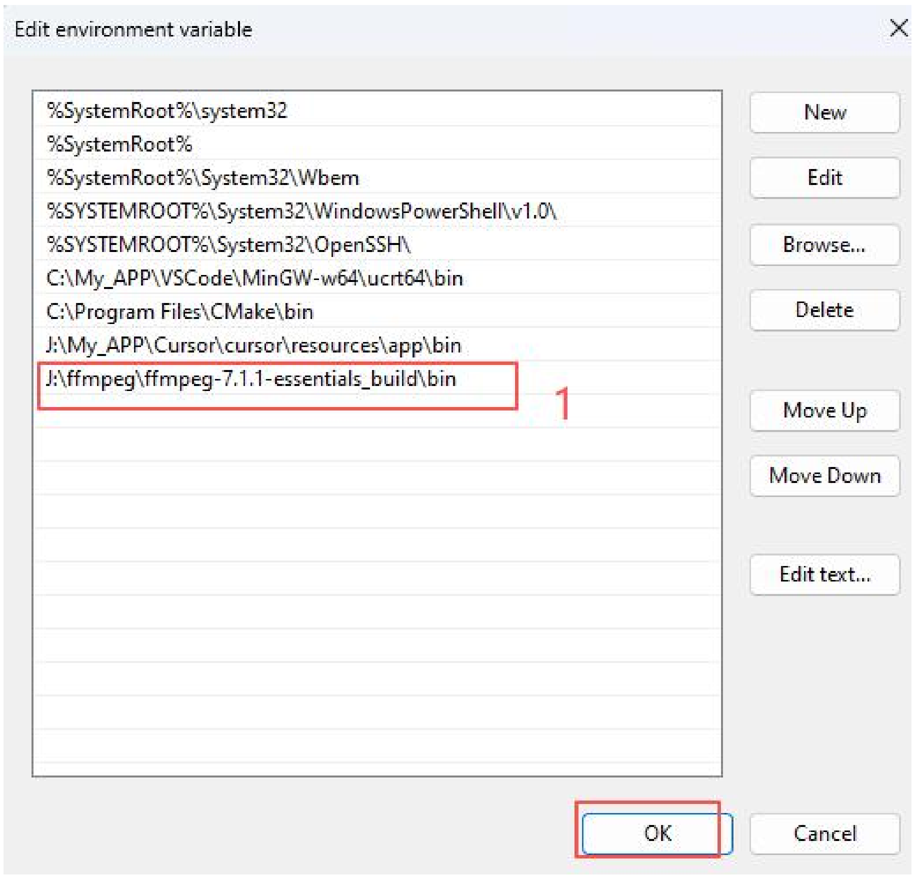
Edit environment variable

```
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\My_APP\VSCode\MinGW-w64\ucrt64\bin
C:\Program Files\CMake\bin
J:\My_APP\Cursor\cursor\resources\app\bin
```

Enter the path to the "bin" folder of FFmpeg (use your own FFmpeg path)


This PC > 新加卷 (J:) > ffmpeg > ffmpeg-7.1.1-essentials_build > bin
Sort
View

Name	Date modified	Type	Size
ffmpeg.exe	10/16/2025 3:07 PM	Application	85,381 KB
ffplay.exe	10/16/2025 3:07 PM	Application	85,198 KB
ffprobe.exe	10/16/2025 3:07 PM	Application	85,246 KB



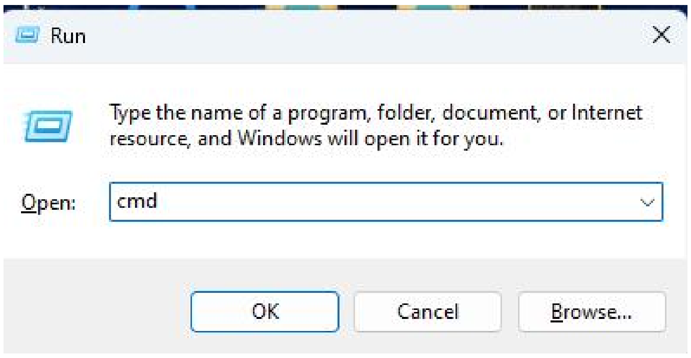
```
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\My_APP\VSCode\MinGW-w64\ucrt64\bin
C:\Program Files\CMake\bin
J:\My_APP\Cursor\cursor\resources\app\bin
J:\ffmpeg\ffmpeg-7.1.1-essentials_build\bin
```

Remember to save the settings after entering the path.

Note: Ensure the path is accurate so the system can correctly locate the FFmpeg files.

Verifying Successful FFmpeg Installation

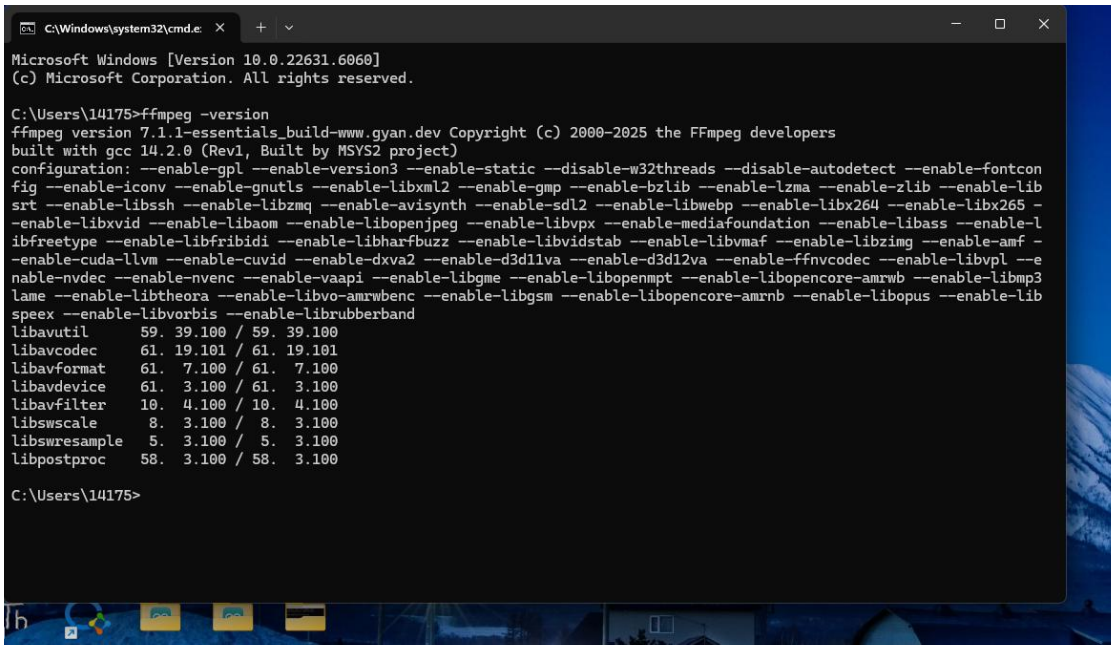
Press the **Win + R** keys, then type "cmd" to open the command line window.



Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

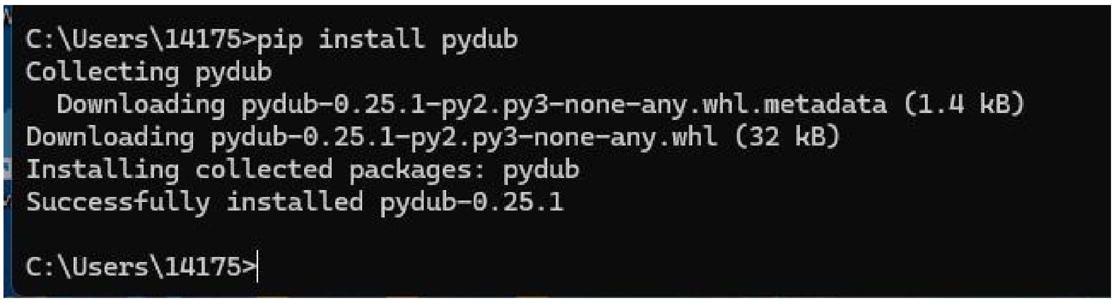
Open: cmd

Type the following command in the command line to check the FFmpeg version:
ffmpeg -version
If the FFmpeg version number and related information are displayed correctly, it indicates that the installation is successful (as shown in the figure below).



```
Microsoft Windows [Version 10.0.22631.6060]
(c) Microsoft Corporation. All rights reserved.

C:\Users\14175>ffmpeg -version
ffmpeg version 7.1.1-essentials_build-www.gyan.dev Copyright (c) 2000-2025 the FFmpeg developers
built with gcc 14.2.0 (Rev1, Built by MSYS2 project)
configuration: --enable-gpl --enable-version3 --enable-static --disable-w32threads --disable-autodetect --enable-fontcon
fig --enable-iconv --enable-gnutls --enable-libxml2 --enable-gmp --enable-bzlib --enable-lzma --enable-zlib --enable-lib
srt --enable-libssh --enable-libzmq --enable-avisynth --enable-sdl2 --enable-libwebp --enable-libx264 --enable-libx265 -
-enable-libxvid --enable-libaom --enable-libopenjpeg --enable-libvpx --enable-mediafoundation --enable-libass --enable-l
ibfreetype --enable-libfribidi --enable-libharfbuzz --enable-libvidstab --enable-libvmaf --enable-libzimg --enable-amf -
-enable-cuda-llvm --enable-cuvid --enable-dxva2 --enable-d3d11va --enable-d3d12va --enable-ffnvcodec --enable-libvpl --e
nable-nvdec --enable-nvenc --enable-vaapi --enable-libgme --enable-libopenmpt --enable-libopencore-amrwb --enable-libmp3
lame --enable-libtheora --enable-libvo-amrwbenc --enable-libgsm --enable-libopencore-amrnb --enable-libopus --enable-lib
speex --enable-libvorbis --enable-librubberband
libavutil      59. 39.100 / 59. 39.100
libavcodec     61. 19.101 / 61. 19.101
libavformat    61.  7.100 / 61.  7.100
libavdevice    61.  3.100 / 61.  3.100
libavfilter    10.  4.100 / 10.  4.100
libswscale      8.  3.100 /  8.  3.100
libswresample   5.  3.100 /  5.  3.100
libpostproc    58.  3.100 / 58.  3.100

C:\Users\14175>
```

Then, still in the command window, install the dependency by running:
pip install pydub

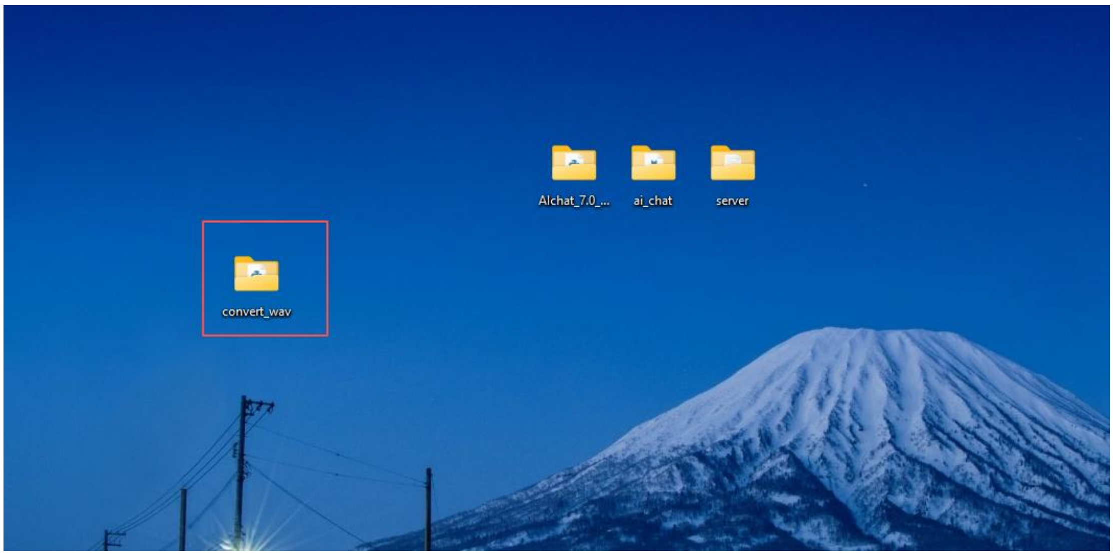


```
C:\Users\14175>pip install pydub
Collecting pydub
  Downloading pydub-0.25.1-py2.py3-none-any.whl.metadata (1.4 kB)
Downloading pydub-0.25.1-py2.py3-none-any.whl (32 kB)
Installing collected packages: pydub
Successfully installed pydub-0.25.1

C:\Users\14175>
```

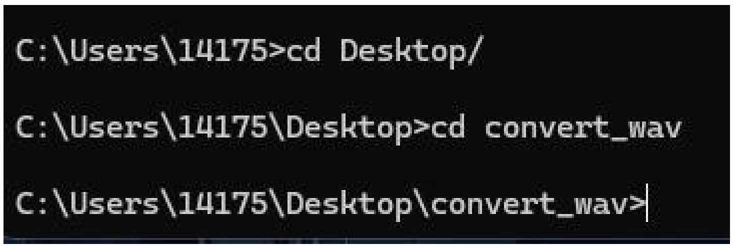
After installation, open the script code we prepared for converting MP3 to WAV format (meeting the specifications of 16kHz, 16-bit, and stereo/dual-channel) in the provided code package.

Click the link below to open the script code:
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/convert_wav

Now I have placed this script on my desktop.

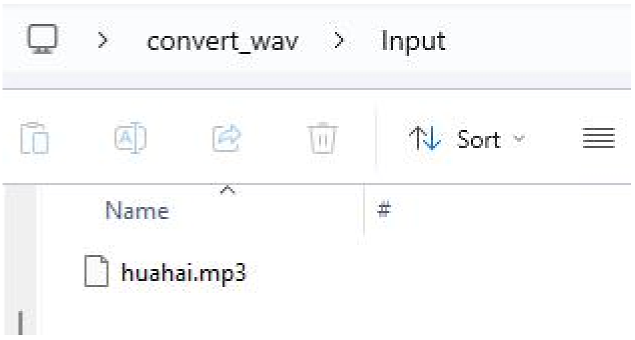


In the command window, I navigate to this path.

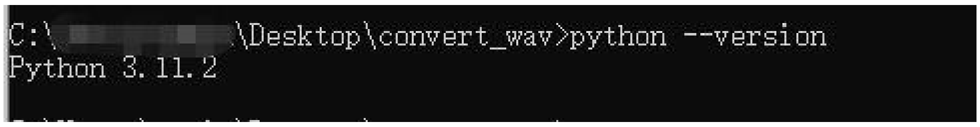


```
C:\Users\14175>cd Desktop/

C:\Users\14175\Desktop>cd convert_wav

C:\Users\14175\Desktop\convert_wav>
```

Then put your MP3 files in the "Input" folder.



Run this script code. (Ensure your Python environment is Python 3.11.2.)

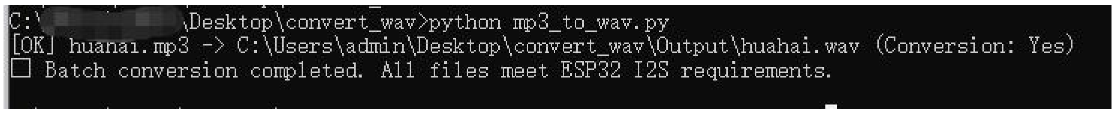


```
C:\[redacted]\Desktop\convert_wav>python --version
Python 3.11.2
```

Starting from Python 3.13:The official team removed the audioop module (which pydub depends on).Some third-party libraries (such as pyaudio, pygame, pydub) are not yet fully compatible.
For Python 3.11.x:

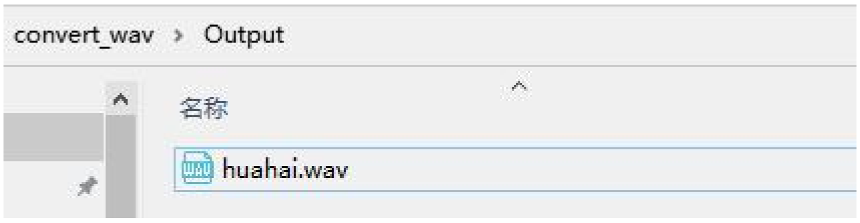
- ✓ Stable, mature, and highly compatible;
- ✓ Includes audioop;
- ✓ Perfectly compatible with most AI, audio, and data analysis libraries.

Run our script:

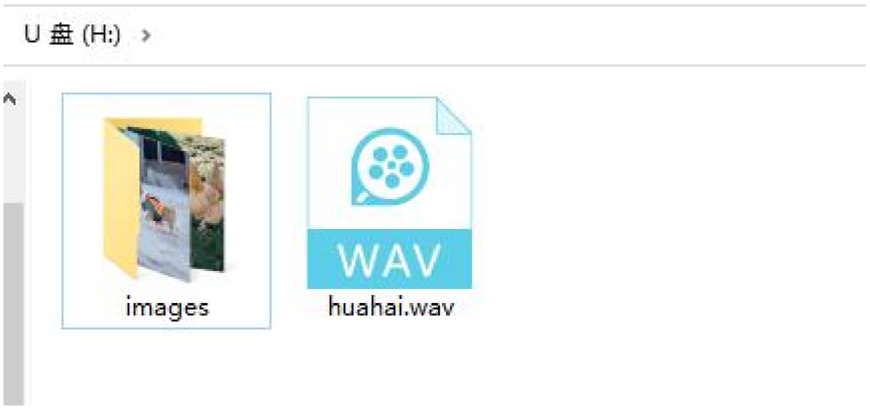


```
C:\[redacted]\Desktop\convert_wav>python mp3_to_wav.py
[OK] huahai.mp3 -> C:\Users\admin\Desktop\convert_wav\Output\huahai.wav (Conversion: Yes)
☐ Batch conversion completed. All files meet ESP32 I2S requirements.
```

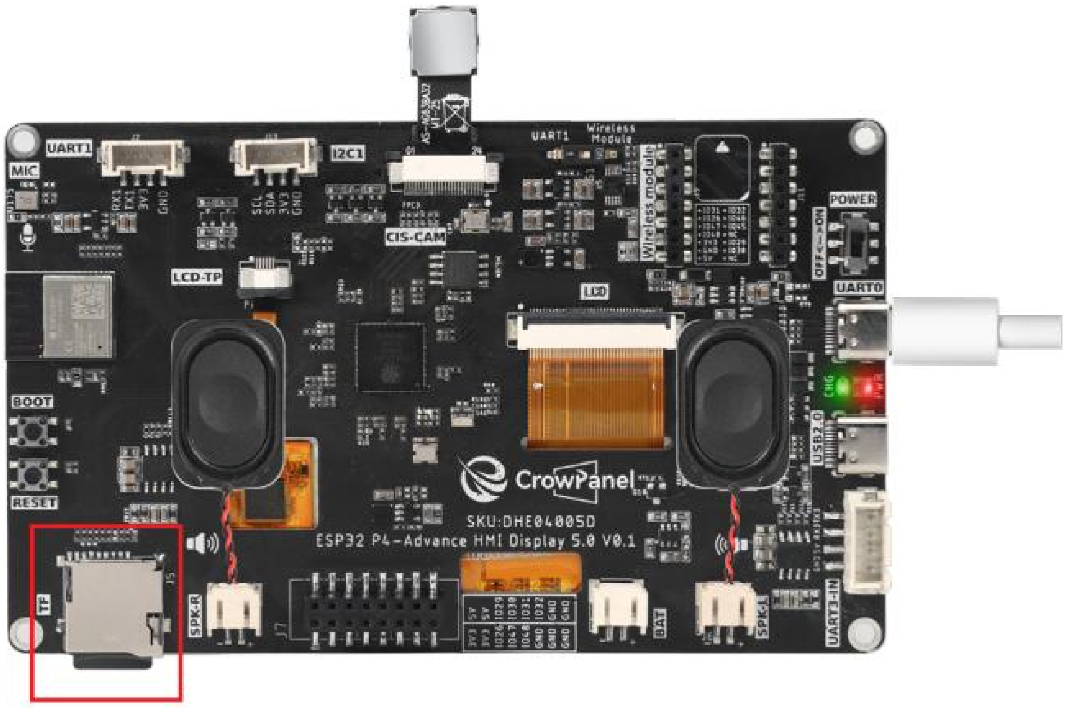
You will find the generated WAV files in the "Output" folder.



Then move this file to a USB flash drive.

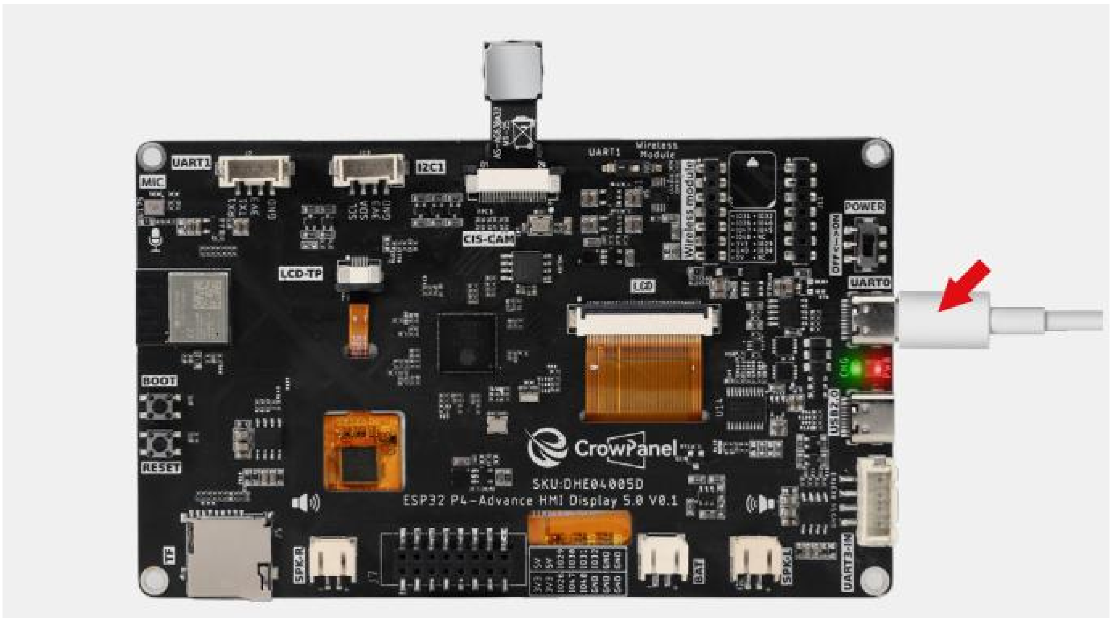


Finally, remove the SD card and insert it into the **Advance-P4** board.

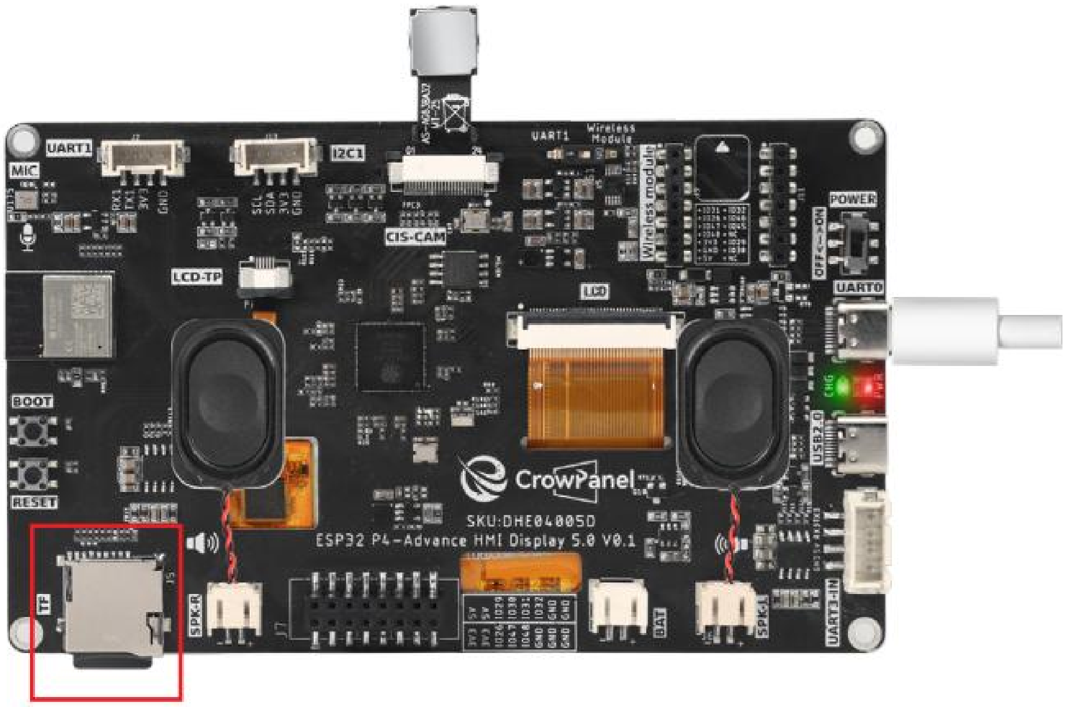


Programming Steps

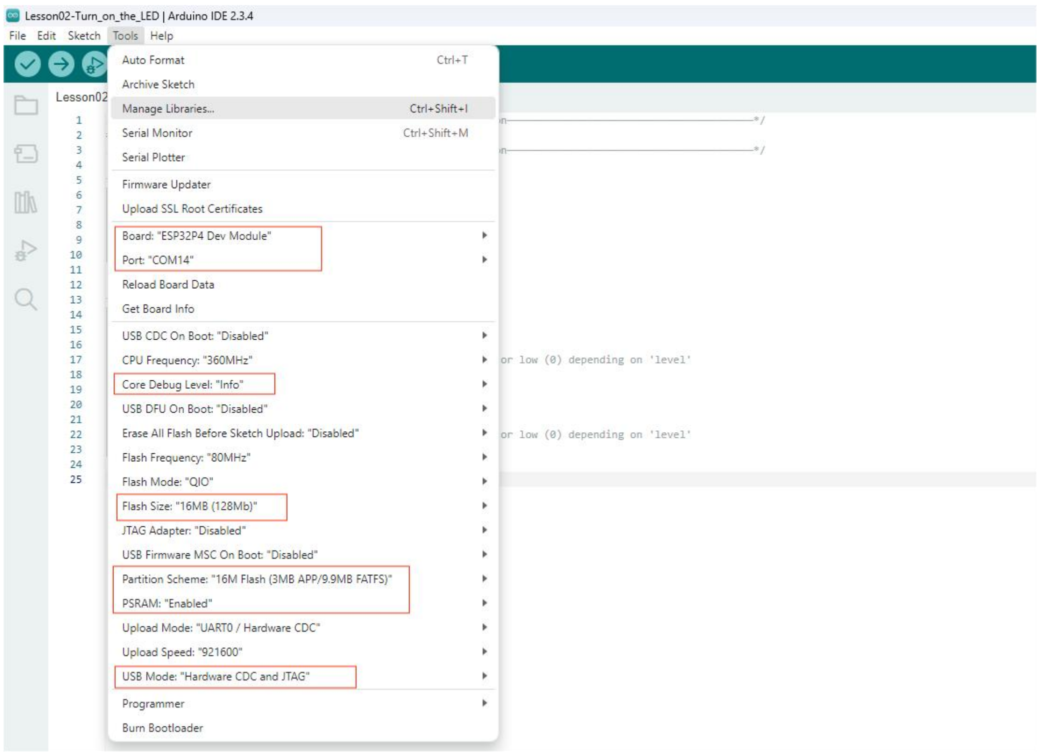
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



First, double-check two things: whether the converted WAV audio file has been placed in the SD card, and whether the SD card is inserted into the SD card slot of the **Advance-P4**.

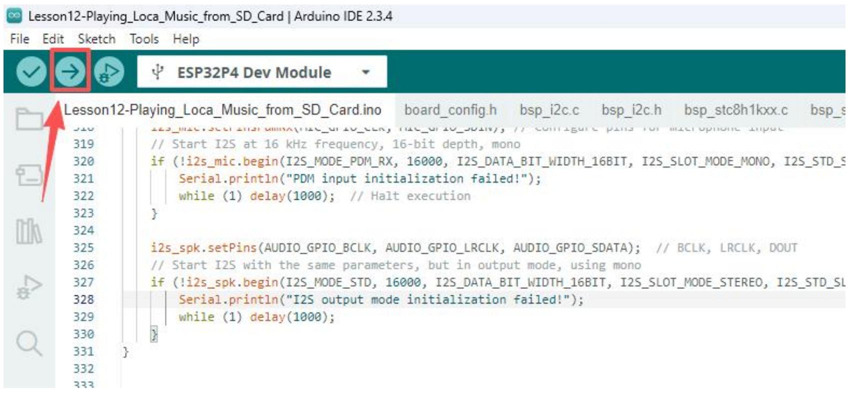


Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

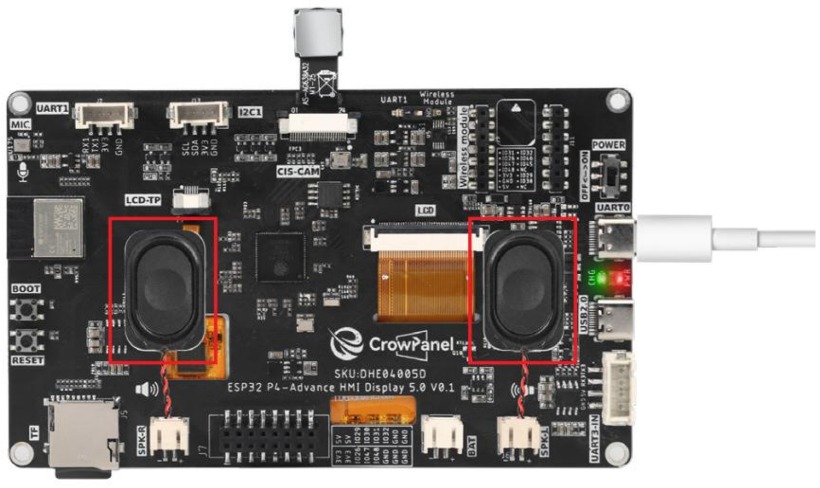
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"

Then we compile and upload the code.


Lesson12-Playing_Loca_Music_from_SD_Card | Arduino IDE 2.3.4

```
    // Start I2S at 16 kHz frequency, 16-bit depth, mono
    if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_S
        Serial.println("PDM input initialization failed!");
        while (1) delay(1000);  // Halt execution
    }

    i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  // BCLK, LRCLK, DOUT
    // Start I2S with the same parameters, but in output mode, using mono
    if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_STEREO, I2S_STD_SL
        Serial.println("I2S output mode initialization failed!");
        while (1) delay(1000);
    }
}
```

Once the code runs, you will hear the speaker on the **Advance-P4** playing the WAV audio stored in your SD card.



The WAV audio file from your SD card is now playing.

Lesson13---Get weather temperature via WiFi

Introduction

In this class, we will learn a new component - the Wi-Fi component. We will also use the display-related components we learned before to achieve Wi-Fi connection together, and then obtain the weather conditions and temperature of your local area, and display the relevant information on the Advance-P4.

Learning Goals

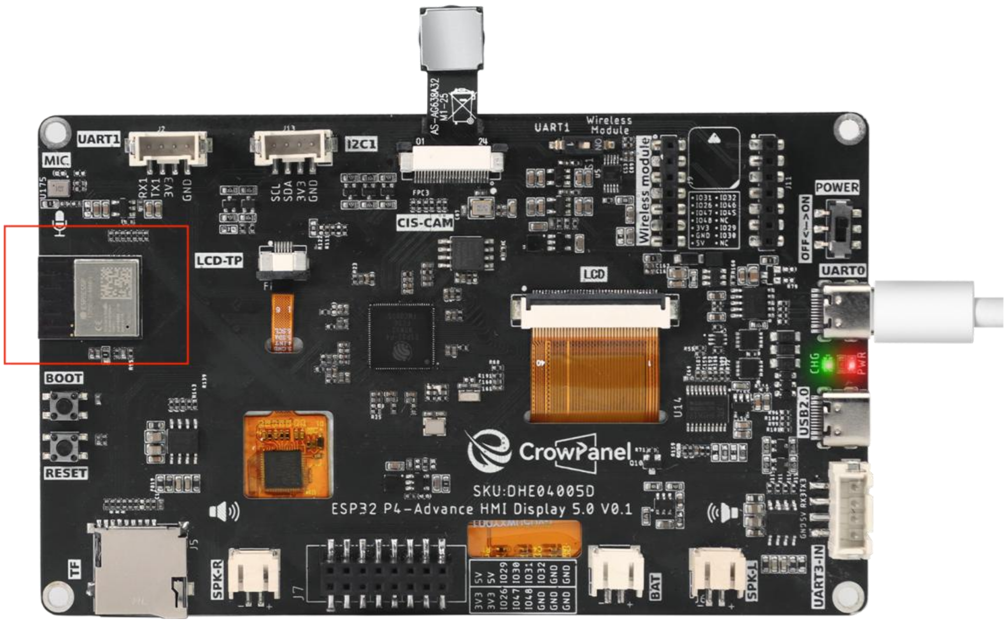
1. Understand the master-slave architecture of ESP32-P4 and ESP32-C6: ESP32-P4 acts as the application processor responsible for graphic display, while ESP32-C6 serves as the Wi-Fi co-processor, providing network capabilities through SDIO. Master the communication mechanism between the two.
2. Master the Wi-Fi connection process: Configure the STA mode, connect to a specified router, and the core methods for the HTTP client to initiate requests and parse JSON format weather data.
3. Master LVGL interface development: Including image resource conversion (using LVGL tools to convert to C arrays), multi-text label layout, thread-safe operations of the interface (locking and unlocking), as well as the engineering implementation of screen backlight control and time synchronization.
4. Master engineering adaptation techniques: For the problem of compilation failure of large resources (images), expand the storage by modifying the partition table to ensure the normal compilation and burning of the code.

Preview of the Result

After running the code, you will be able to see the relevant information about the local weather on the screen of the Advance-P4.

Hardware Used in This Lesson

The ESP32-C6 Wi-Fi module on the Advance-P4



The ESP32-P4 does not integrate Wi-Fi wireless radio and MAC/PHY modules by itself. Its positioning is as a high-performance application processor, mainly responsible for graphic display, audio and video, multimedia, human-computer interaction, and complex business logic, rather than wireless communication. Therefore, when ESP32-P4 needs to connect to the network, the Wi-Fi function must be realized through an external network coprocessor.

The ESP32-C6 is a complete Wi-Fi 6 + Bluetooth + 802.15.4 wireless SoC that has a mature and stable network protocol stack and radio capabilities. In the actual solution, ESP32-P4 communicates with ESP32-C6 through interfaces such as SPI, UART, and SDIO. The C6 is responsible for the underlying network work such as Wi-Fi scanning, authentication, encryption, and TCP/IP protocol processing, while P4 only needs to send network requests and receive data through the SDIO communication protocol.

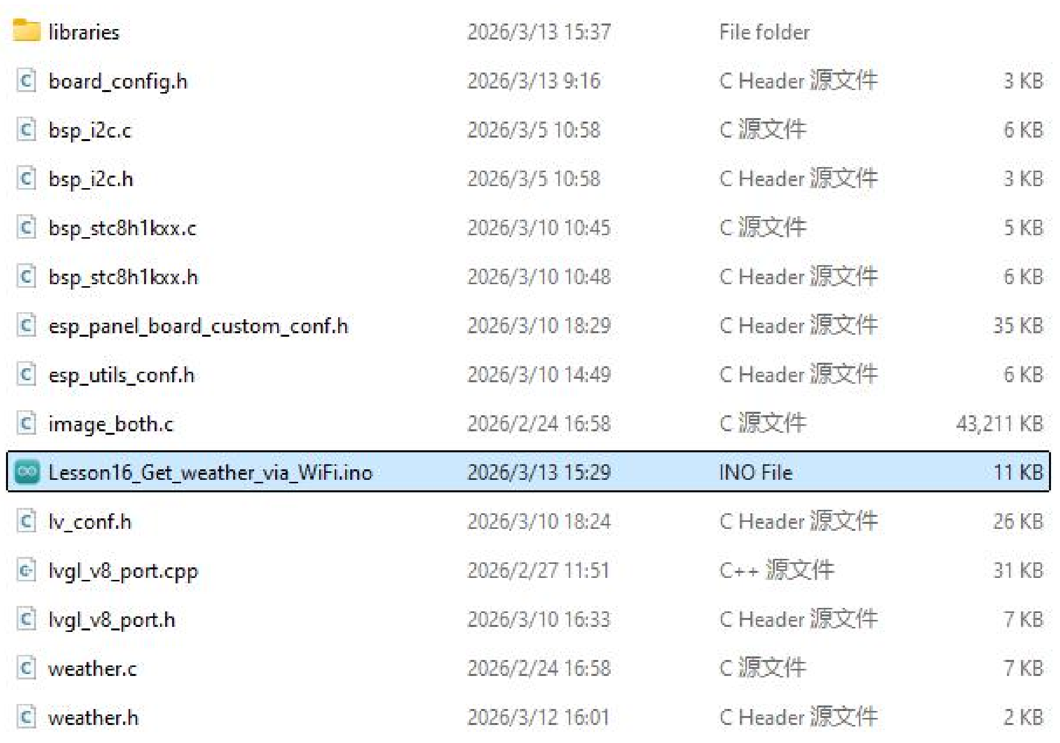
This "master control + wireless coprocessor" architecture not only reduces the power consumption and design complexity of P4, but also improves the reliability and scalability of the system. It is a typical design concept adopted by Espressif in high-end human-computer interaction and edge computing products.

Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson13_Get_weather_via_WiFi

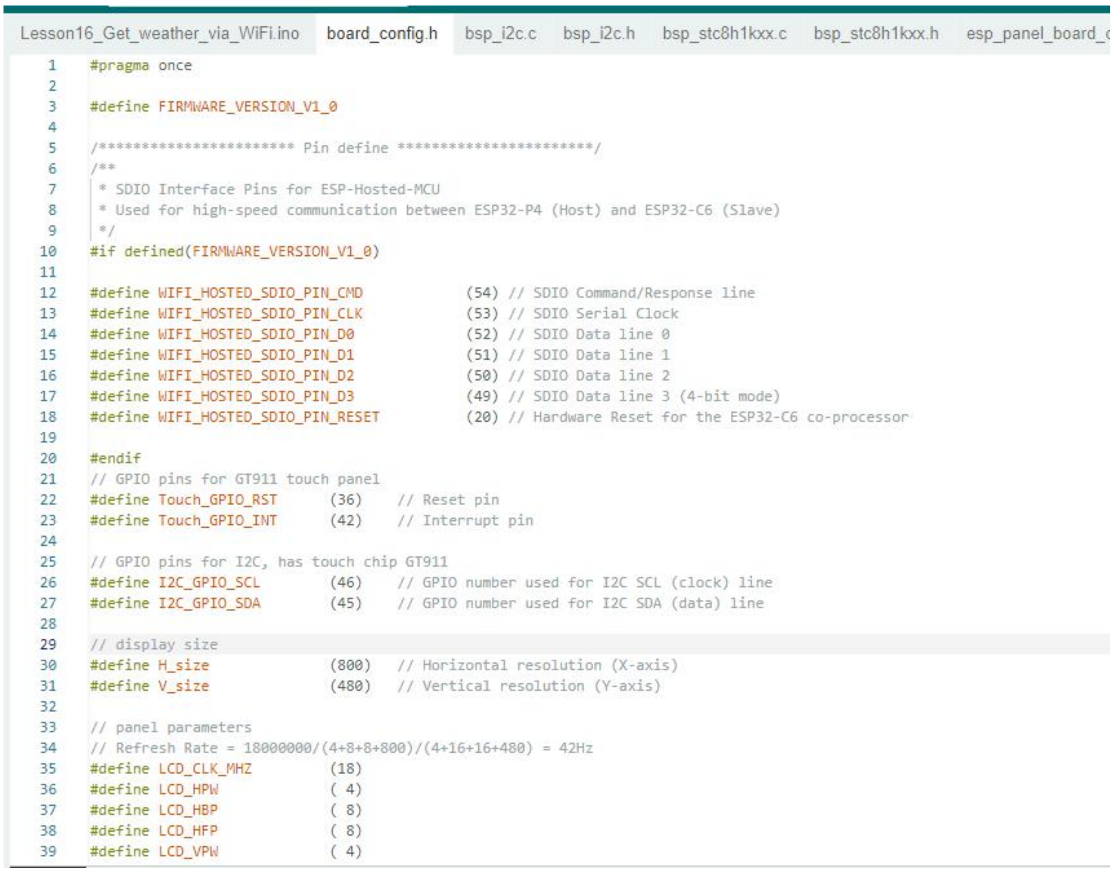
Key Explanations

The main focus of this class is on how to connect to Wi-Fi and how to use Wi-Fi to connect to the network to obtain weather information.
Next, let's take a look at how it is implemented in the code.
Double-click to open the code (ino file) for this lesson.



libraries	2026/3/13 15:37	File folder	
board_config.h	2026/3/13 9:16	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/10 10:48	C Header 源文件	6 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
image_both.c	2026/2/24 16:58	C 源文件	43,211 KB
Lesson16_Get_weather_via_WiFi.ino	2026/3/13 15:29	INO File	11 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB
weather.c	2026/2/24 16:58	C 源文件	7 KB
weather.h	2026/3/12 16:01	C Header 源文件	2 KB

The code segment "board_config.h" is a header file for the development board's hardware configuration. It is used to uniformly define the GPIO pins, communication interfaces, and display parameters of each hardware module in the system, making it convenient for the underlying drivers and application programs to access.



```
#pragma once

#define FIRMWARE_VERSION_V1_0

/*********************** Pin define ***********************/
/**
 * SDIO Interface Pins for ESP-Hosted-MCU
 * Used for high-speed communication between ESP32-P4 (Host) and ESP32-C6 (Slave)
 */
#if defined(FIRMWARE_VERSION_V1_0)

#define WIFI_HOSTED_SDIO_PIN_CMD          (54) // SDIO Command/Response line
#define WIFI_HOSTED_SDIO_PIN_CLK          (53) // SDIO Serial Clock
#define WIFI_HOSTED_SDIO_PIN_D0           (52) // SDIO Data line 0
#define WIFI_HOSTED_SDIO_PIN_D1           (51) // SDIO Data line 1
#define WIFI_HOSTED_SDIO_PIN_D2           (50) // SDIO Data line 2
#define WIFI_HOSTED_SDIO_PIN_D3           (49) // SDIO Data line 3 (4-bit mode)
#define WIFI_HOSTED_SDIO_PIN_RESET        (20) // Hardware Reset for the ESP32-C6 co-processor

#endif
// GPIO pins for GT911 touch panel
#define Touch_GPIO_RST      (36)    // Reset pin
#define Touch_GPIO_INT      (42)    // Interrupt pin

// GPIO pins for I2C, has touch chip GT911
#define I2C_GPIO_SCL        (46)    // GPIO number used for I2C SCL (clock) line
#define I2C_GPIO_SDA        (45)    // GPIO number used for I2C SDA (data) line

// display size
#define H_size              (800)   // Horizontal resolution (X-axis)
#define V_size              (480)   // Vertical resolution (Y-axis)

// panel parameters
// Refresh Rate = 18000000/(4+8+8+800)/(4+16+16+480) = 42Hz
#define LCD_CLK_MHZ         (18)
#define LCD_HPW             ( 4)
#define LCD_HBP             ( 8)
#define LCD_HFP             ( 8)
#define LCD_VPW             ( 4)
```

The beginning of the file uses #pragma once to prevent the header file from being included multiple times, and through #define FIRMWARE_VERSION_V1_0, the current firmware version is specified, allowing conditional compilation based on different hardware configurations of different versions.

First, the code defines a set of SDIO communication interface pins, which are used for high-speed communication between the main control ESP32-P4 and the coprocessor ESP32-C6. This architecture belongs to the ESP-Hosted-MCU mode, where P4 acts as the Host and C6 acts as the Slave to provide Wi-Fi or wireless network functions. The SDIO interface includes the CMD command line, the CLK clock line, and four data lines D0 to D3 (4-bit mode), and also defines WIFI_HOSTED_SDIO_PIN_RESET for resetting the C6 coprocessor.

The following code defines two control pins of the touchscreen controller GT911: Touch_GPIO_RST is used for resetting the touch chip, and Touch_GPIO_INT is used for the touch interrupt signal; at the same time, the I^2 C bus pins used by the touch chip, I2C_GPIO_SCL and I2C_GPIO_SDA, are also defined.

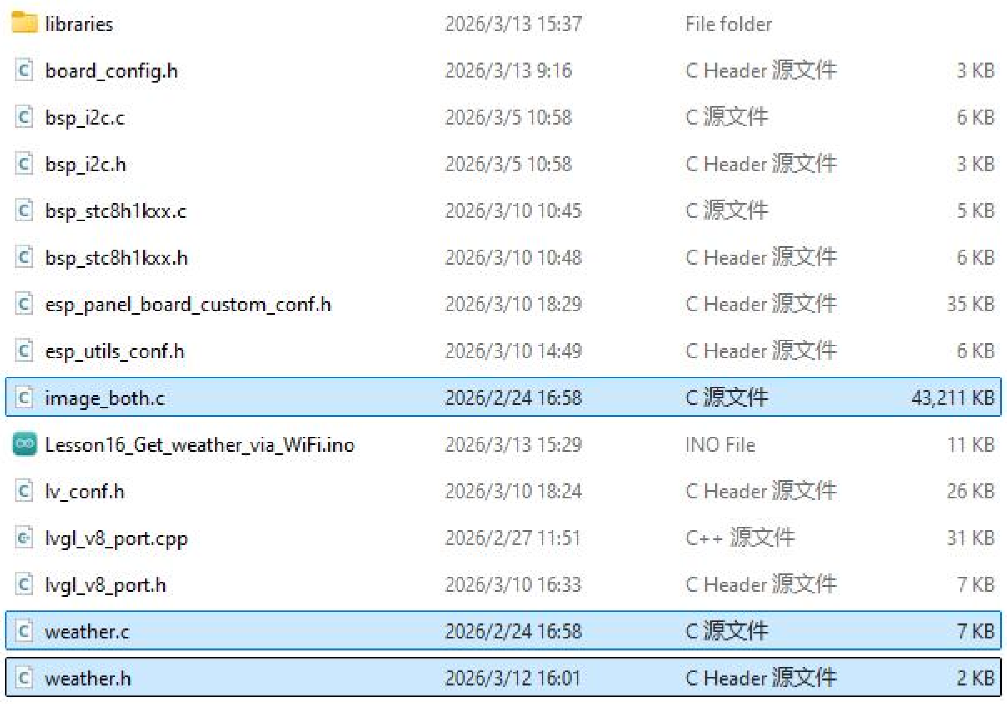
Subsequently, the code provided the basic parameters of the LCD display screen, including H_size (horizontal resolution 800) and V_size (vertical resolution 480), indicating that the screen is an 800×480 pixel RGB LCD screen.

Then a set of LCD timing parameters were defined, such as the pixel clock

LCD_CLK_MHZ which is 18 MHz, as well as the pulse widths and the pre- and post-halting times related to horizontal and vertical synchronization (HPW, HBP, HFP, VPW, VBP, VFP). These parameters jointly determine the screen refresh timing. Based on the annotations, the screen refresh rate is approximately 42 Hz.

The final code defines the pin mappings for all the pins of the RGB parallel display interface, including the synchronization signals HSYNC, VSYNC, data enable DE, pixel clock PCLK, and the 16 data lines DATA0 to DATA15. These data lines are responsible for transmitting pixel color data to the LCD during each pixel clock cycle.

It can be seen that compared with the previous project architecture, there are now three additional files here.



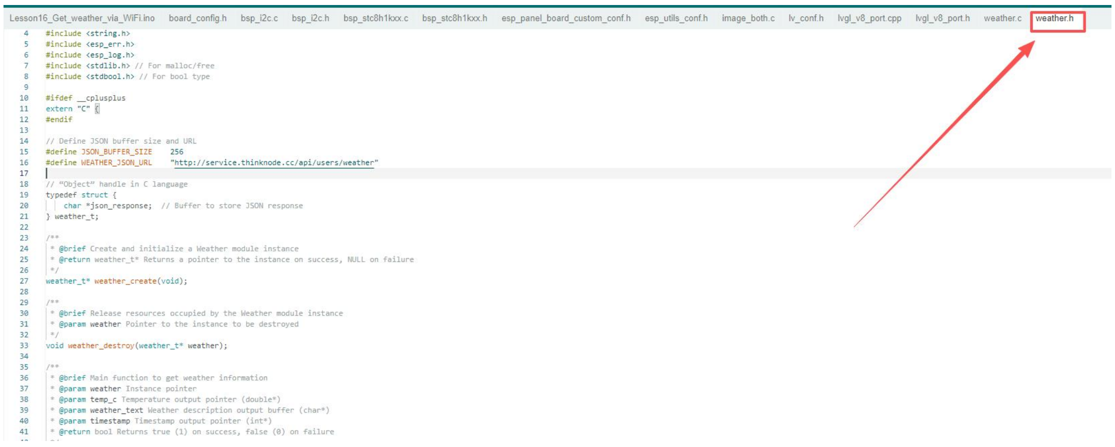
libraries	2026/3/13 15:37	File folder	
board_config.h	2026/3/13 9:16	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/10 10:48	C Header 源文件	6 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
image_both.c	2026/2/24 16:58	C 源文件	43,211 KB
Lesson16_Get_weather_via_WiFi.ino	2026/3/13 15:29	INO File	11 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB
weather.c	2026/2/24 16:58	C 源文件	7 KB
weather.h	2026/3/12 16:01	C Header 源文件	2 KB

Among them, image_both.c is the file where we used a tool to convert the image into a C array. That means this file is actually an image, which is the background image in the effect diagram. In the following lessons, we will teach you how to generate such an image.

And weather.c and weather.h are the code files that we use to obtain weather information. Next, let's take a closer look at them.

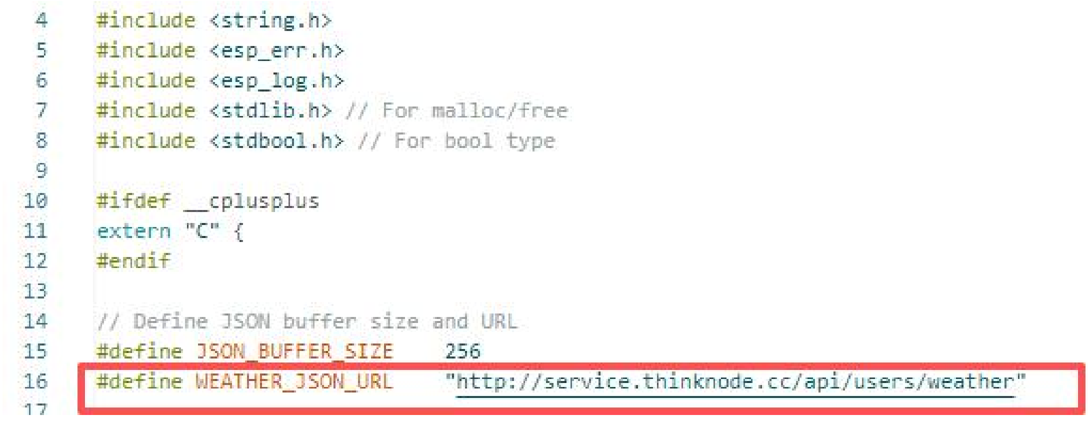
So how can we obtain the local weather conditions? Here we offer an application called "weather" to get the weather information.

It's the same. Let's first take a look at the content of the weather.h file.



```c
#include <string.h>
#include <esp_err.h>
#include <esp_log.h>
#include <stdlib.h> // For malloc/free
#include <stdbool.h> // For bool type

#ifdef __cplusplus
extern "C" {
#endif

// Define JSON buffer size and URL
#define JSON_BUFFER_SIZE    256
#define WEATHER_JSON_URL    "http://service.thinknode.cc/api/users/weather"

// "Object" handle in C language
typedef struct {
    char *json_response;  // Buffer to store JSON response
} weather_t;

/**
 * @brief Create and initialize a Weather module instance
 * @return weather_t* Returns a pointer to the instance on success, NULL on failure
 */
weather_t* weather_create(void);

/**
 * @brief Release resources occupied by the Weather module instance
 * @param weather Pointer to the instance to be destroyed
 */
void weather_destroy(weather_t* weather);

/**
 * @brief Main function to get weather information
 * @param weather Instance pointer
 * @param temp_c Temperature output pointer (double*)
 * @param weather_text Weather description output buffer (char*)
 * @param timestamp Timestamp output pointer (int*)
 * @return bool Returns true (1) on success, false (0) on failure
```

Let me focus on this particular URL link.(weather.h)

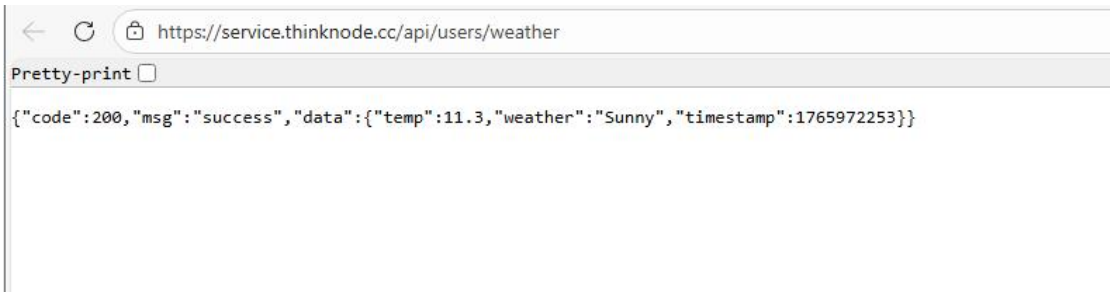


```c
#include <string.h>
#include <esp_err.h>
#include <esp_log.h>
#include <stdlib.h> // For malloc/free
#include <stdbool.h> // For bool type

#ifdef __cplusplus
extern "C" {
#endif

// Define JSON buffer size and URL
#define JSON_BUFFER_SIZE    256
#define WEATHER_JSON_URL    "http://service.thinknode.cc/api/users/weather"
```

This link is the server address of our Elecrow company. You can set this link as the fixed URL for accessing the weather information.

Because no matter where you are, when you connect to a Wi-Fi network and access this URL link to obtain the local weather conditions, our URL will, based on the IP of the Wi-Fi network you are connected to, obtain the weather conditions and local time stamp of the location corresponding to your IP.

This is very convenient for you to use. Just directly access this URL.
You can open this URL link to have a look:



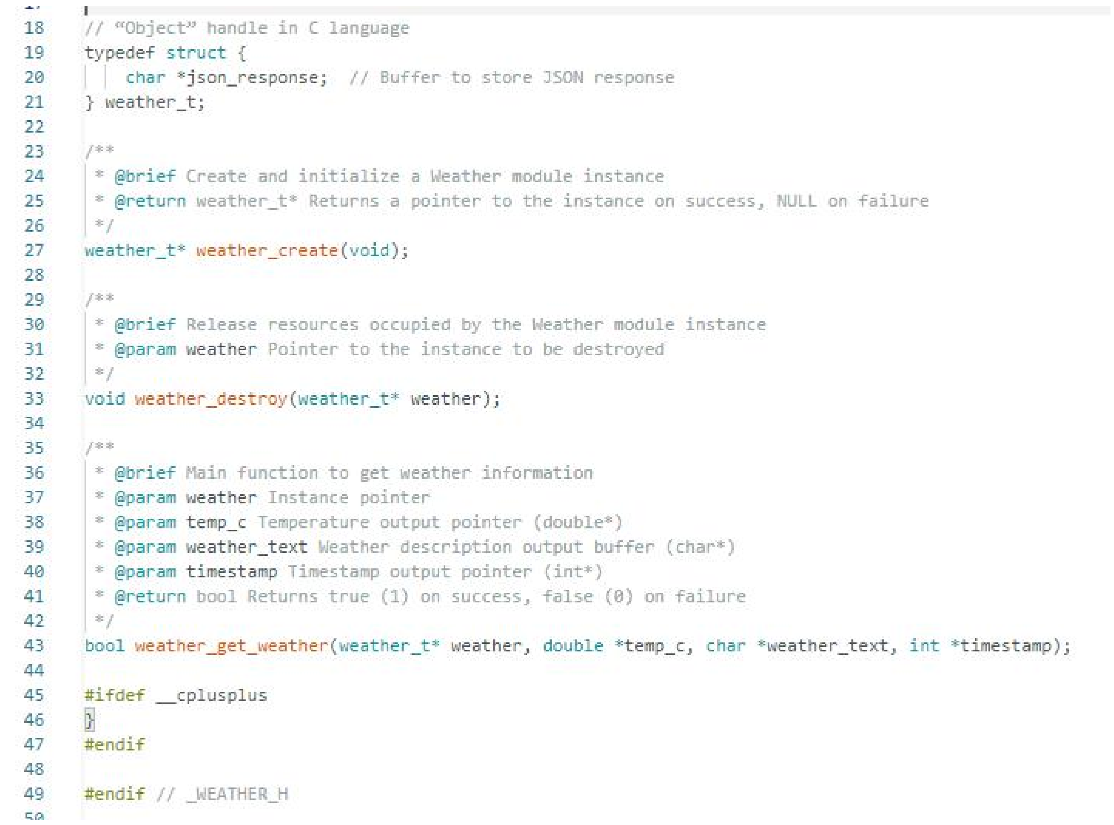
https://service.thinknode.cc/api/users/weather

Pretty-print

```
{"code":200,"msg":"success","data":{"temp":11.3,"weather":"Sunny","timestamp":1765972253}}
```

(This URL link does not support a large number of accesses)
What we have is a JSON code. You can extract the information you need from it. We will also explain it next.

Then, like the function names listed below, we will proceed to explain in detail in the weather.c file. The declaration in weather.h is to provide an interface that is convenient for use in the .c file.



```
// "Object" handle in C language
typedef struct {
    char *json_response;  // Buffer to store JSON response
} weather_t;

/**
 * @brief Create and initialize a Weather module instance
 * @return weather_t* Returns a pointer to the instance on success, NULL on failure
 */
weather_t* weather_create(void);

/**
 * @brief Release resources occupied by the Weather module instance
 * @param weather Pointer to the instance to be destroyed
 */
void weather_destroy(weather_t* weather);

/**
 * @brief Main function to get weather information
 * @param weather Instance pointer
 * @param temp_c Temperature output pointer (double*)
 * @param weather_text Weather description output buffer (char*)
 * @param timestamp Timestamp output pointer (int*)
 * @return bool Returns true (1) on success, false (0) on failure
 */
bool weather_get_weather(weather_t* weather, double *temp_c, char *weather_text, int *timestamp);

#ifdef __cplusplus
}
#endif

#endif // _WEATHER_H
```

weather.c

Entering the code for the main implementation, let's take a look at what each function does and how to call them.

1. weather_create

The function weather_create is responsible for creating and initializing an instance of the weather module, which is an alternative implementation of a C++ constructor in the C language. It allocates memory for the weather_t structure itself, allocates space for the internal json_response buffer (used to store JSON data returned by HTTP), and clears this buffer to ensure the safety and reliability of subsequent string concatenation.

It is invoked before the program needs to use the "weather" functions. Typically, it is called once after the system initialization is completed and the Wi-Fi connection is successful to obtain a valid pointer to the weather object.

2.weather_destroy

The role of weather_destroy is to release all resources allocated by weather_create, including the JSON buffer and the weather_t structure itself, to prevent memory leaks. This is the equivalent implementation of a "destructor" in the C language.

It should be invoked when the weather functionality is no longer required by the program—for example, before system shutdown, before task exit, or when the module is dynamically unloaded. It must be used in pairs with weather_create to form complete resource lifecycle management.

3.http_event_handle

http_event_handler is the event callback function of the ESP-IDF HTTP client, used to receive data returned by the server during the HTTP request process. Whenever the underlying TCP/HTTP receives a segment of data, this function will be automatically called, and by using evt->user_data, it retrieves the current weather_t instance and safely concatenates the received data into the json_response buffer.

The invocation timing is not directly called by user code, but is triggered internally by esp_http_client_perform(), which is an indispensable part of the HTTP communication mechanism.

4.weather_http_get_json

The function of weather_http_get_json is to initiate an HTTP request and fully obtain the weather JSON data. It encapsulates the entire process of HTTP client configuration, initialization, execution, and resource cleanup. Inside the function, the old JSON buffer will be cleared, the request URL and callback function will be configured, and then the HTTP request will be blocked and waited for completion. Finally, the original JSON string returned by the server will be saved in weather->json_response.

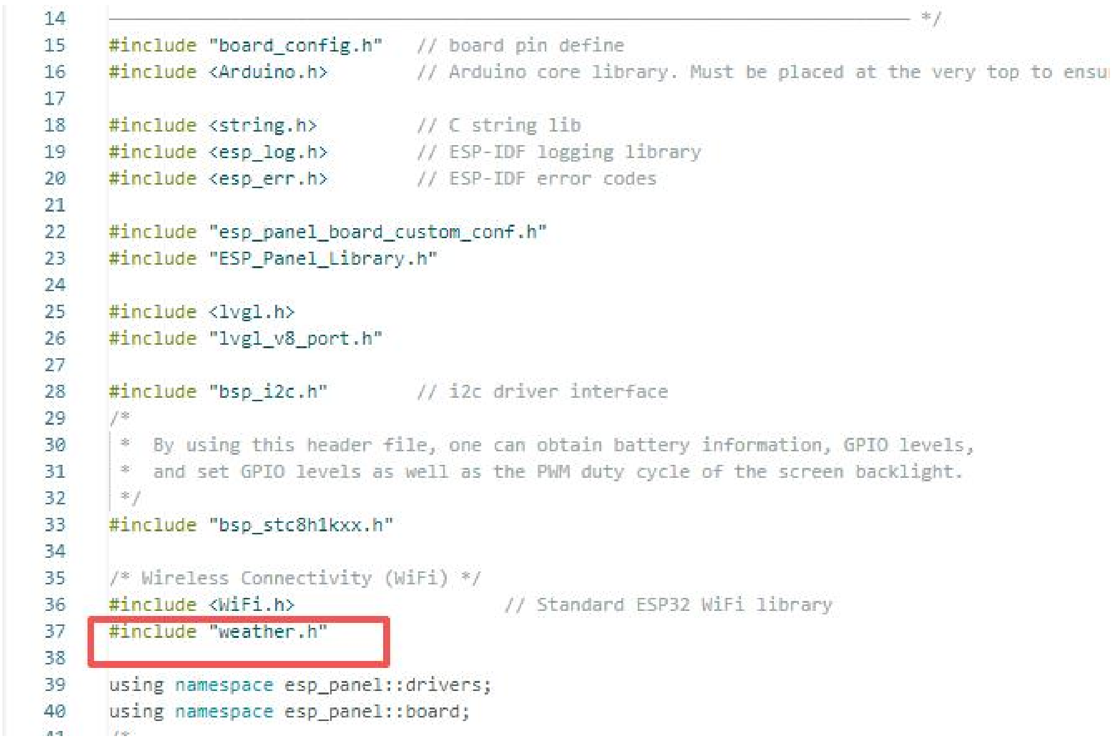
This function is called when it is necessary to "obtain the latest raw weather data". However, it is an internal implementation function and is only used by this module. It will not be directly called by the outside.

5.weather_analyse_weather_json
The role of weather_analyse_weather_json is to parse the JSON string returned by the server and extract the data that the business layer actually cares about, including temperature, weather description, and timestamp. It uses the cJSON library to parse the JSON structure and performs validity checks on each field, ensuring that the data format is correct before outputting the data to the variables provided by the caller.
It is invoked after successfully obtaining HTTP JSON data, and is used to convert "string data" into "directly usable business data". This function also belongs to the internal functions of the module.

6.weather_get_weather
weather_get_weather is the only core external interface function of the entire weather module, responsible for providing an "end-to-end weather retrieval service" to external callers. Internally, it first invokes weather_http_get_json to fetch the latest weather data from the network, then calls weather_analyse_weather_json to parse the JSON and output the results. Callers do not need to care about the details of HTTP, JSON, or memory management.
It is invoked when the Wi-Fi connection is successfully established and the application layer requires real-time weather information—for example, scenarios such as UI refresh, periodic weather data updates, etc.

This concludes our explanation of the weather.c file.

From now on, all you need to do is to call the "weather.h" file within the "ino" file.

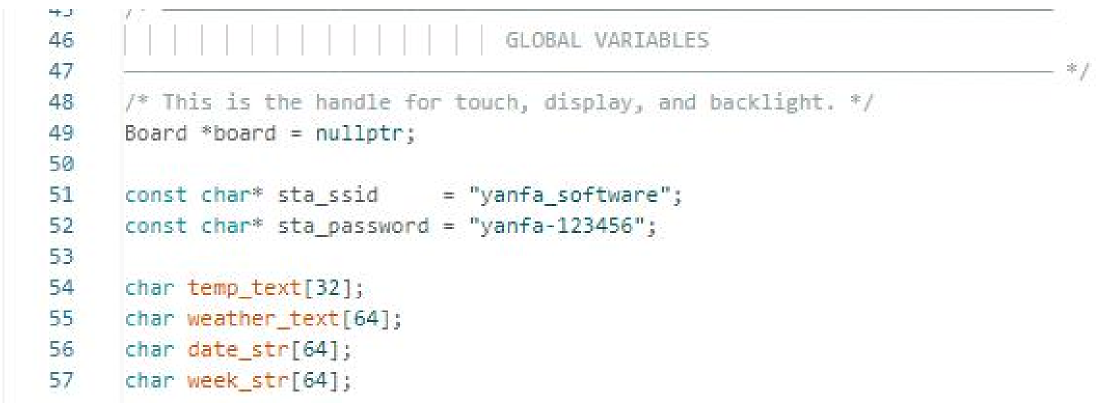


```
──────────────────────────────────────────────────────────── */
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensu

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

#include <lvgl.h>
#include "lvgl_v8_port.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"

/* Wireless Connectivity (WiFi) */
#include <WiFi.h>                   // Standard ESP32 WiFi library
#include "weather.h"

using namespace esp_panel::drivers;
using namespace esp_panel::board;
```

Then, next, let's take a closer look at the content of the Lesson13_Get_weather_via_WiFi.ino code file.

1. Global Variables

This code is mainly used to define the global variables and string buffers that are required for the program's operation. It provides the basic data structure for subsequent WiFi connection, weather data storage, and interface display.



```
/*
|||||||||||||||| GLOBAL VARIABLES
                                                                          */
/* This is the handle for touch, display, and backlight. */
Board *board = nullptr;

const char* sta_ssid     = "yanfa_software";
const char* sta_password = "yanfa-123456";

char temp_text[32];
char weather_text[64];
char date_str[64];
char week_str[64];
```

First, "Board *board = nullptr;" defines a pointer to an object of the "Board" class and initializes it to "nullptr" (a null pointer). The "Board" class is derived from "ESP32_Display_Panel", which is responsible for uniformly managing the display-related hardware on the development board, such as the LCD screen, RGB bus, and touch controller. During the program initialization stage, this object is typically created by "board = new Board();", and then the hardware initialization of the display and touch screen is completed by calling "board->init()" and "board->begin()". Therefore, defining "board" as a global pointer allows multiple functions in the program (such as the display initialization function or interface function) to access the same hardware control object.

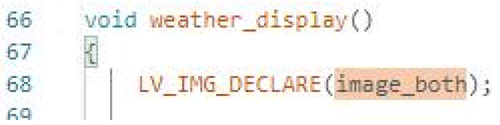
The following two lines of code: const char* sta_ssid = "yanfa_software"; and const char* sta_password = "yanfa-123456"; are used to define the WiFi name (SSID) and password required for the device to connect to the wireless network. Here, const char* is used to represent a pointer to a string constant, indicating that these strings will not be modified during the program's execution. Later, when calling WiFi.begin(sta_ssid, sta_password) (from WiFi.h), these two variables will be used to connect to the specified router.

The last four lines of code define four character arrays: temp_text[32], weather_text[64], date_str[64], and week_str[64]. These are buffers used to store string data. After the program obtains the weather information, it will format the temperature, weather description, date, and week data using functions such as snprintf() or strftime() and save them to these arrays. For example, temp_text may store "25.4°C", weather_text may store "Partly Cloudy", date_str may store "2026/03/13", and week_str may store "Friday". These strings will then be passed to the lv_label text control created by LVGL and displayed on the LCD screen as the weather interface information.

2.weather_display

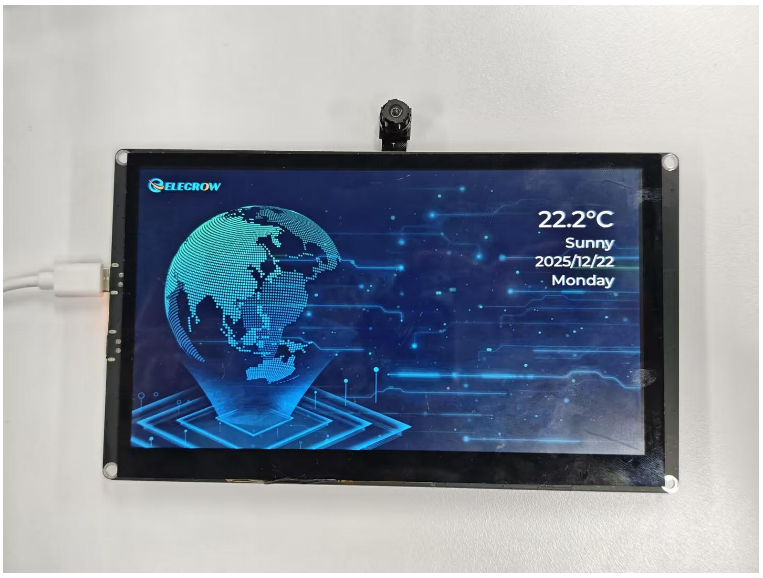
The main function of this weather_display() function is to use LVGL to create and display a complete weather interface on the LCD screen, including background images, temperature, weather conditions, date and week information.

At the beginning of the function, LV_IMG_DECLARE(image_both) is used to declare a pre-compiled image resource that has been incorporated into the program, which is used as the background image for the interface.

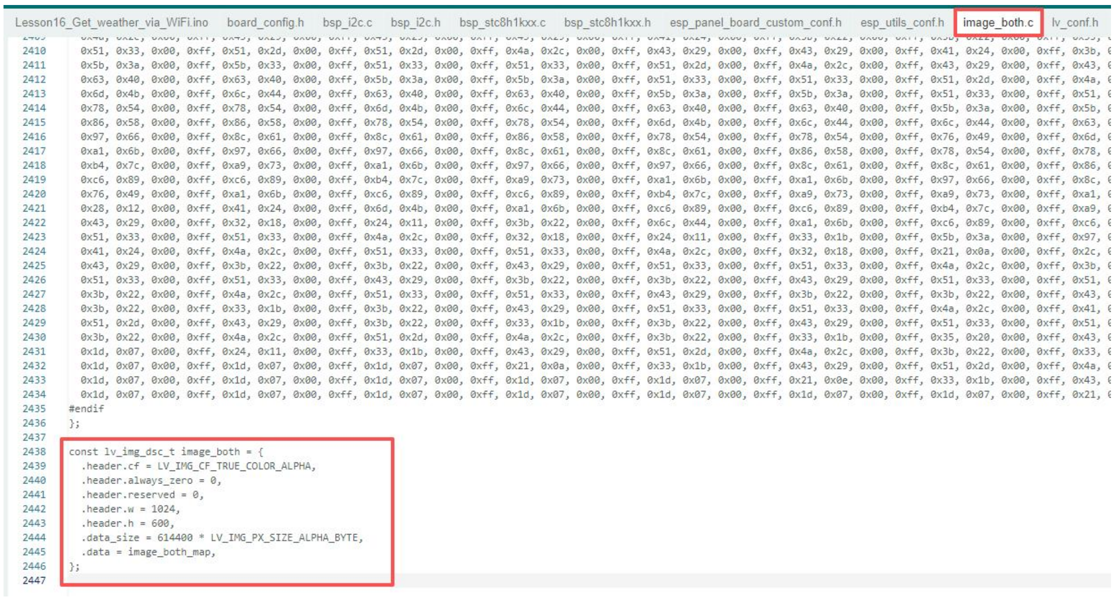


```
void weather_display()
{
    LV_IMG_DECLARE(image_both);
```

This "image_both.c" is our image with the Earth as the background, converted into a.c file.



As can be seen in image_both.c, the data for image conversion is presented.



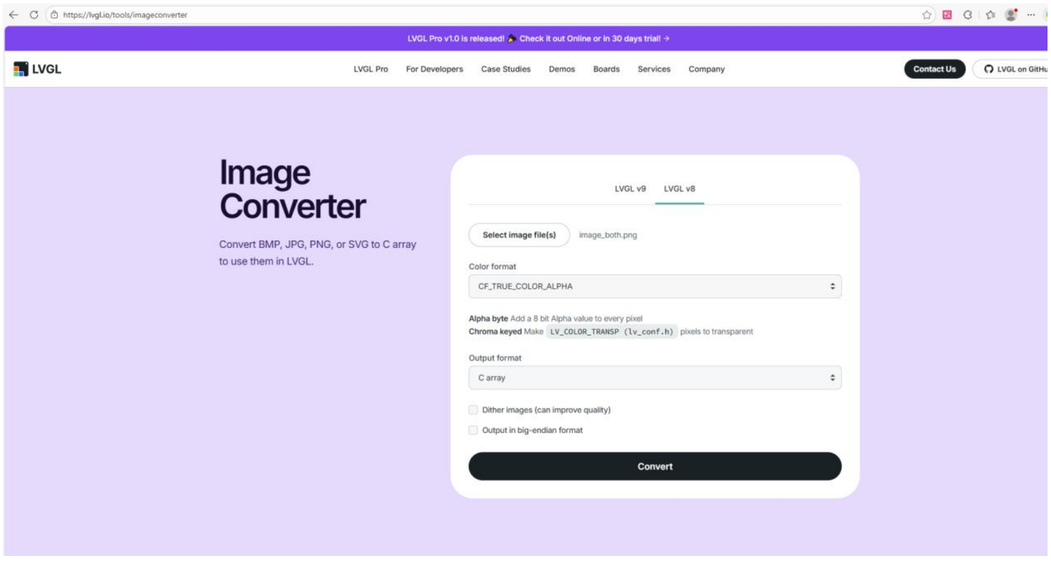
```
  0x51, 0x33, 0x00, 0xff, 0x51, 0x2d, 0x00, 0xff, 0x51, 0x2d, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x41, 0x24, 0x00, 0xff, 0x3b,
  0x5b, 0x3a, 0x00, 0xff, 0x5b, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x2d, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x43,
  0x63, 0x40, 0x00, 0xff, 0x63, 0x40, 0x00, 0xff, 0x5b, 0x3a, 0x00, 0xff, 0x5b, 0x3a, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x2d, 0x00, 0xff, 0x4a,
  0x6d, 0x4b, 0x00, 0xff, 0x6c, 0x44, 0x00, 0xff, 0x63, 0x40, 0x00, 0xff, 0x63, 0x40, 0x00, 0xff, 0x5b, 0x3a, 0x00, 0xff, 0x5b, 0x3a, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51,
  0x78, 0x54, 0x00, 0xff, 0x78, 0x54, 0x00, 0xff, 0x6d, 0x4b, 0x00, 0xff, 0x6c, 0x44, 0x00, 0xff, 0x63, 0x40, 0x00, 0xff, 0x63, 0x40, 0x00, 0xff, 0x5b, 0x3a, 0x00, 0xff, 0x5b,
  0x86, 0x58, 0x00, 0xff, 0x86, 0x58, 0x00, 0xff, 0x78, 0x54, 0x00, 0xff, 0x78, 0x54, 0x00, 0xff, 0x6d, 0x4b, 0x00, 0xff, 0x6c, 0x44, 0x00, 0xff, 0x6c, 0x44, 0x00, 0xff, 0x63,
  0x97, 0x66, 0x00, 0xff, 0x8c, 0x61, 0x00, 0xff, 0x8c, 0x61, 0x00, 0xff, 0x86, 0x58, 0x00, 0xff, 0x78, 0x54, 0x00, 0xff, 0x78, 0x54, 0x00, 0xff, 0x76, 0x49, 0x00, 0xff, 0x6d,
  0xa1, 0x6b, 0x00, 0xff, 0x97, 0x66, 0x00, 0xff, 0x97, 0x66, 0x00, 0xff, 0x8c, 0x61, 0x00, 0xff, 0x8c, 0x61, 0x00, 0xff, 0x86, 0x58, 0x00, 0xff, 0x78, 0x54, 0x00, 0xff, 0x78,
  0xb4, 0x7c, 0x00, 0xff, 0xa9, 0x73, 0x00, 0xff, 0xa1, 0x6b, 0x00, 0xff, 0x97, 0x66, 0x00, 0xff, 0x97, 0x66, 0x00, 0xff, 0x8c, 0x61, 0x00, 0xff, 0x8c, 0x61, 0x00, 0xff, 0x86,
  0xc6, 0x89, 0x00, 0xff, 0xc6, 0x89, 0x00, 0xff, 0xb4, 0x7c, 0x00, 0xff, 0xa9, 0x73, 0x00, 0xff, 0xa1, 0x6b, 0x00, 0xff, 0xa1, 0x6b, 0x00, 0xff, 0x97, 0x66, 0x00, 0xff, 0x8c,
  0x76, 0x49, 0x00, 0xff, 0xa1, 0x6b, 0x00, 0xff, 0xc6, 0x89, 0x00, 0xff, 0xc6, 0x89, 0x00, 0xff, 0xb4, 0x7c, 0x00, 0xff, 0xa9, 0x73, 0x00, 0xff, 0xa9, 0x73, 0x00, 0xff, 0xa1,
  0x28, 0x12, 0x00, 0xff, 0x41, 0x24, 0x00, 0xff, 0x6d, 0x4b, 0x00, 0xff, 0xa1, 0x6b, 0x00, 0xff, 0xc6, 0x89, 0x00, 0xff, 0xc6, 0x89, 0x00, 0xff, 0xb4, 0x7c, 0x00, 0xff, 0xa9,
  0x43, 0x29, 0x00, 0xff, 0x32, 0x18, 0x00, 0xff, 0x24, 0x11, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x6c, 0x44, 0x00, 0xff, 0xa1, 0x6b, 0x00, 0xff, 0xc6, 0x89, 0x00, 0xff, 0xc6,
  0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x32, 0x18, 0x00, 0xff, 0x24, 0x11, 0x00, 0xff, 0x33, 0x1b, 0x00, 0xff, 0x5b, 0x3a, 0x00, 0xff, 0x97,
  0x41, 0x24, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x32, 0x18, 0x00, 0xff, 0x21, 0x0a, 0x00, 0xff, 0x2c,
  0x43, 0x29, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x3b,
  0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51,
  0x3b, 0x22, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x43,
  0x3b, 0x22, 0x00, 0xff, 0x33, 0x1b, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x41,
  0x51, 0x2d, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x33, 0x1b, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x51, 0x33, 0x00, 0xff, 0x51,
  0x3b, 0x22, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x51, 0x2d, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x33, 0x1b, 0x00, 0xff, 0x35, 0x20, 0x00, 0xff, 0x43,
  0x1d, 0x07, 0x00, 0xff, 0x24, 0x11, 0x00, 0xff, 0x33, 0x1b, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x51, 0x2d, 0x00, 0xff, 0x4a, 0x2c, 0x00, 0xff, 0x3b, 0x22, 0x00, 0xff, 0x33,
  0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x21, 0x0a, 0x00, 0xff, 0x33, 0x1b, 0x00, 0xff, 0x43, 0x29, 0x00, 0xff, 0x51, 0x2d, 0x00, 0xff, 0x4a,
  0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x21, 0x0e, 0x00, 0xff, 0x33, 0x1b, 0x00, 0xff, 0x43,
  0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x1d, 0x07, 0x00, 0xff, 0x21,
#endif
};

const lv_img_dsc_t image_both = {
  .header.cf = LV_IMG_CF_TRUE_COLOR_ALPHA,
  .header.always_zero = 0,
  .header.reserved = 0,
  .header.w = 1024,
  .header.h = 600,
  .data_size = 614400 * LV_IMG_PX_SIZE_ALPHA_BYTE,
  .data = image_both_map,
};
```

How to convert the image you want to display on Advance-P4 into a .c file that can be displayed?

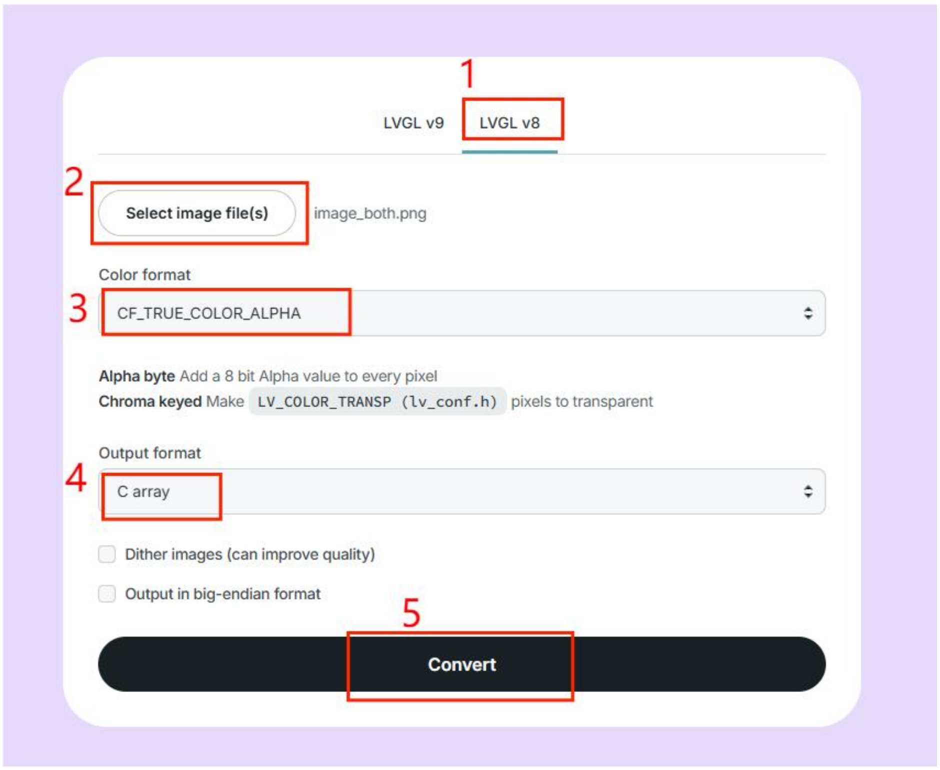
First, open this link provided by the LVGL official website:

https://lvgl.io/tools/imageconverter

This is a tool provided by LVGL itself for converting images into C arrays.


LVGL Pro v1.0 is released! Check it out Online or in 30 days trial!
LVGL
LVGL Pro
For Developers
Case Studies
Demos
Boards
Services
Company
Contact Us
LVGL on GitHub
Image Converter
Convert BMP, JPG, PNG, or SVG to C array to use them in LVGL.
LVGL v9
LVGL v8
Select image file(s)
image_both.png
Color format
CF_TRUE_COLOR_ALPHA
Alpha byte Add a 8 bit Alpha value to every pixel
Chroma keyed Make LV_COLOR_TRANSP (lv_conf.h) pixels to transparent
Output format
C array
Dither images (can improve quality)
Output in big-endian format
Convert

Then, make the selection as indicated in the figure below.


1
LVGL v9
LVGL v8
2
Select image file(s)
image_both.png
Color format
3
CF_TRUE_COLOR_ALPHA
Alpha byte Add a 8 bit Alpha value to every pixel
Chroma keyed Make LV_COLOR_TRANSP (lv_conf.h) pixels to transparent
Output format
4
C array
Dither images (can improve quality)
Output in big-endian format
5
Convert

Step 1: The LVGL-related components in our project all use the v8 version. Therefore, we choose LVGL v8.

Step 2: Select the image you want to convert. This image must maintain a resolution of 1024*600, as our screen resolution is 1024*600.

Step 3: Make sure to select CF_TRUE_COLOR_ALPHA. Since our screen is 16-bit, LVGL will automatically convert the 32-bit pixels into the 16-bit pixels of the screen configuration, and the transparency effect will still exist.

Step 4: Choose the method of converting to a C array.

Step 5: Perform the conversion and obtain the corresponding .c file.

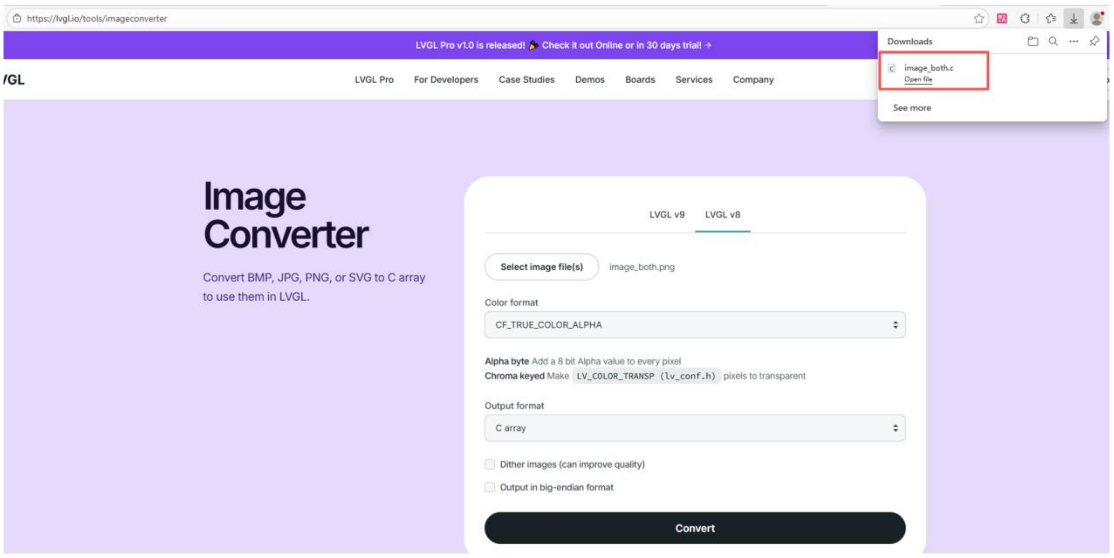

https://lvgl.io/tools/imageconverter
LVGL Pro v1.0 is released! Check it out Online or in 30 days trial! →
LVGL Pro
For Developers
Case Studies
Demos
Boards
Services
Company
Downloads
image_both.c
Open file
See more
LVGL v9
LVGL v8
Select image file(s)
Convert
Dither images (can improve quality)
Output in big-endian format

Image Converter

Convert BMP, JPG, PNG, or SVG to C array to use them in LVGL.

image_both.png

Color format: CF_TRUE_COLOR_ALPHA

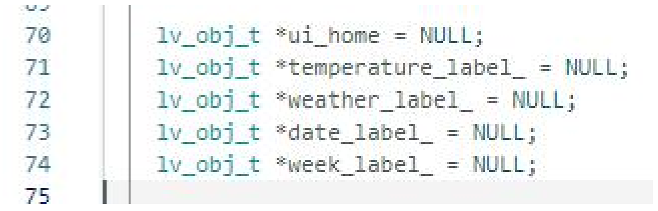
Alpha byte Add a 8 bit Alpha value to every pixel
Chroma keyed Make `LV_COLOR_TRANSP (lv_conf.h)` pixels to transparent

Output format: C array

Then just place this file in the same directory level as the main ino code.
Look back at the main code of ino.

Then multiple pointer variables of type lv_obj_t* (ui_home.temperature_label_.weather_label_.date_label_.week_label_) were defined.

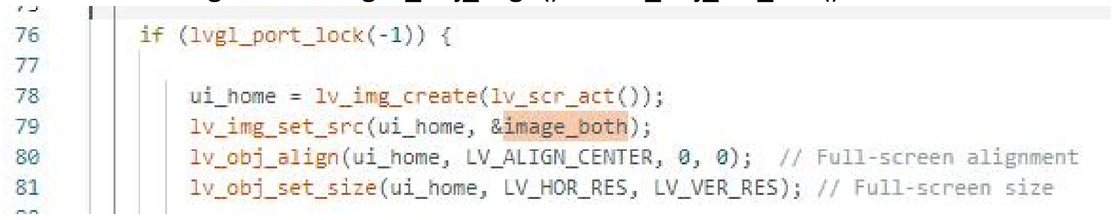
These variables respectively represent the interface objects created by LVGL, such as background objects and different text labels.



```
    lv_obj_t *ui_home = NULL;
    lv_obj_t *temperature_label_ = NULL;
    lv_obj_t *weather_label_ = NULL;
    lv_obj_t *date_label_ = NULL;
    lv_obj_t *week_label_ = NULL;
```

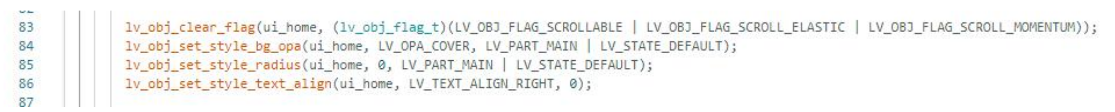
Then, the program acquires the LVGL mutex lock using lvgl_port_lock(-1), which is a common practice when working with LVGL graphics objects in a multi-task or multi-thread environment to ensure that the interface creation process is not interrupted by other tasks.

After entering the critical section, it first creates a picture object on the current active screen using lv_img_create(lv_scr_act()), sets its image resource to image_both using lv_img_set_src(), and then aligns and sizes the object to be a full-screen background using lv_obj_align() and lv_obj_set_size().



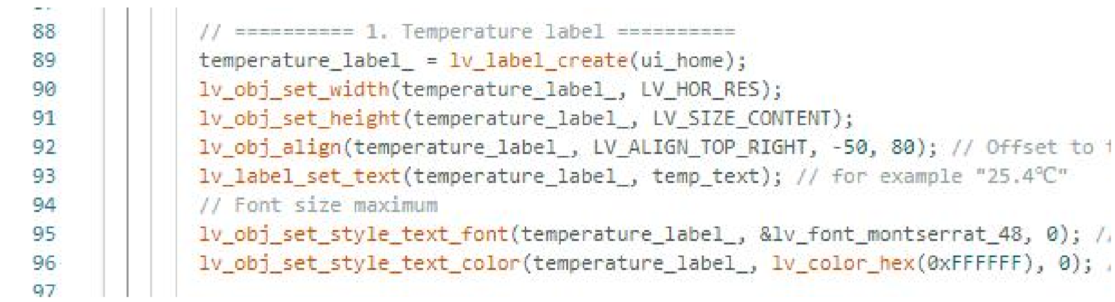
```
if (lvgl_port_lock(-1)) {

    ui_home = lv_img_create(lv_scr_act());
    lv_img_set_src(ui_home, &image_both);
    lv_obj_align(ui_home, LV_ALIGN_CENTER, 0, 0);  // Full-screen alignment
    lv_obj_set_size(ui_home, LV_HOR_RES, LV_VER_RES); // Full-screen size
```

At the same time, call lv_obj_clear_flag() to disable the scrolling-related properties, so that the background interface cannot be dragged, and set the background to be non-transparent. The corner radius is 0 and the text is right-aligned, etc. styles.



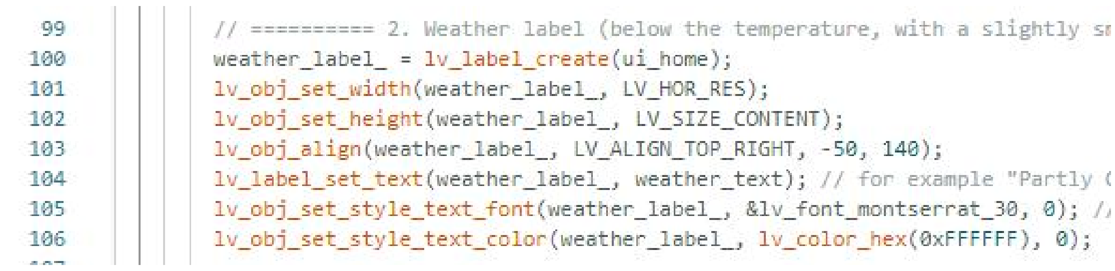
```
lv_obj_clear_flag(ui_home, (lv_obj_flag_t)(LV_OBJ_FLAG_SCROLLABLE | LV_OBJ_FLAG_SCROLL_ELASTIC | LV_OBJ_FLAG_SCROLL_MOMENTUM));
lv_obj_set_style_bg_opa(ui_home, LV_OPA_COVER, LV_PART_MAIN | LV_STATE_DEFAULT);
lv_obj_set_style_radius(ui_home, 0, LV_PART_MAIN | LV_STATE_DEFAULT);
lv_obj_set_style_text_align(ui_home, LV_TEXT_ALIGN_RIGHT, 0);
```

Subsequently, the program creates four text labels in sequence: The first one is the temperature label named "temperature_label_", which is created using lv_label_create() and placed on the background object. Its width is set to the screen width. The position is at the upper right corner with an offset of (-50, 80). The displayed content comes from the global variable temp_text (for example, "25.4° C"), and it uses the large font lv_font_montserrat_48 and white text to highlight the temperature information.



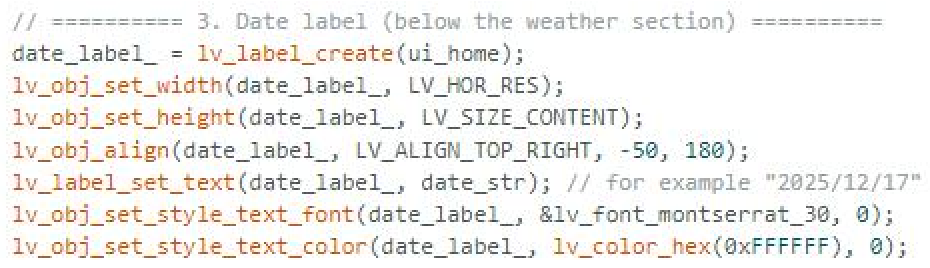
```
// ========== 1. Temperature label ==========
temperature_label_ = lv_label_create(ui_home);
lv_obj_set_width(temperature_label_, LV_HOR_RES);
lv_obj_set_height(temperature_label_, LV_SIZE_CONTENT);
lv_obj_align(temperature_label_, LV_ALIGN_TOP_RIGHT, -50, 80); // Offset to t
lv_label_set_text(temperature_label_, temp_text); // for example "25.4℃"
// Font size maximum
lv_obj_set_style_text_font(temperature_label_, &lv_font_montserrat_48, 0); /
lv_obj_set_style_text_color(temperature_label_, lv_color_hex(0xFFFFFF), 0);
```

The second label, named "weather_label_", is used to display the weather description (such as "Partly Cloudy"), located below the temperature, and has a slightly smaller font size (lv_font_montserrat_30);

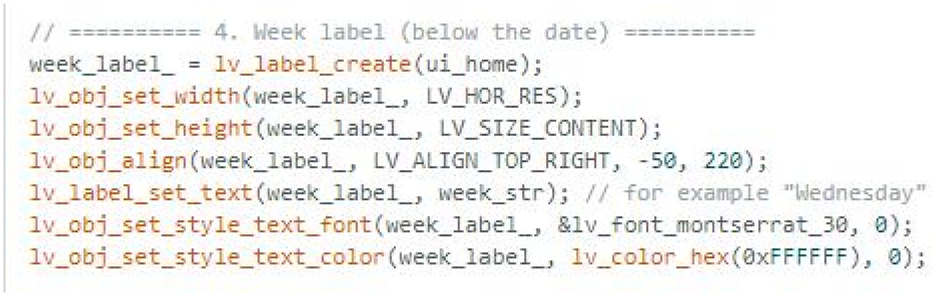


```
// ========== 2. Weather label (below the temperature, with a slightly s
weather_label_ = lv_label_create(ui_home);
lv_obj_set_width(weather_label_, LV_HOR_RES);
lv_obj_set_height(weather_label_, LV_SIZE_CONTENT);
lv_obj_align(weather_label_, LV_ALIGN_TOP_RIGHT, -50, 140);
lv_label_set_text(weather_label_, weather_text); // for example "Partly (
lv_obj_set_style_text_font(weather_label_, &lv_font_montserrat_30, 0); /
lv_obj_set_style_text_color(weather_label_, lv_color_hex(0xFFFFFF), 0);
```

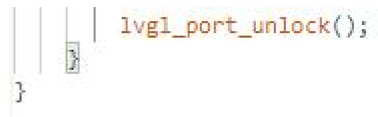
The third label, named "date_label_", is used to display the date string "date_str" (for example, "2025/12/17"), and its position continues to be arranged downward.


```
// ========== 3. Date label (below the weather section) ==========
date_label_ = lv_label_create(ui_home);
lv_obj_set_width(date_label_, LV_HOR_RES);
lv_obj_set_height(date_label_, LV_SIZE_CONTENT);
lv_obj_align(date_label_, LV_ALIGN_TOP_RIGHT, -50, 180);
lv_label_set_text(date_label_, date_str); // for example "2025/12/17"
lv_obj_set_style_text_font(date_label_, &lv_font_montserrat_30, 0);
lv_obj_set_style_text_color(date_label_, lv_color_hex(0xFFFFFF), 0);
```

The fourth label, named "week_label_", is used to display the week information, "week_str" (such as "Wednesday"), and it is also presented in a smaller font and arranged below the date.


```
// ========== 4. Week label (below the date) ==========
week_label_ = lv_label_create(ui_home);
lv_obj_set_width(week_label_, LV_HOR_RES);
lv_obj_set_height(week_label_, LV_SIZE_CONTENT);
lv_obj_align(week_label_, LV_ALIGN_TOP_RIGHT, -50, 220);
lv_label_set_text(week_label_, week_str); // for example "Wednesday"
lv_obj_set_style_text_font(week_label_, &lv_font_montserrat_30, 0);
lv_obj_set_style_text_color(week_label_, lv_color_hex(0xFFFFFF), 0);
```

The entire interface layout adopts right alignment, ensuring that all the text is neatly arranged in the right area of the background image.

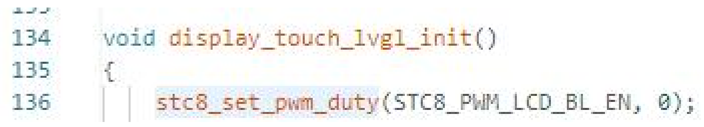

```
            lvgl_port_unlock();
        }
}
```

Finally, the LVGL lock is released using lvgl_port_unlock(), allowing other tasks to continue accessing the graphics system.

3.display_touch_lvgl_init

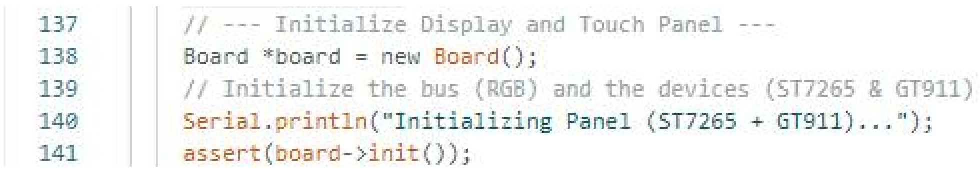
The main function of this display_touch_lvgl_init() function is to initialize the screen backlight, RGB display screen, touch screen, and LVGL graphical interface system, thereby enabling the device to have complete graphic display and touch input capabilities.

The function begins by calling stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0) to set the LCD backlight PWM duty cycle to 0%, which means turning off the screen backlight first. This step is usually used to prevent screen flickering or abnormal brightness during the display initialization process. The PWM control is handled by the external coprocessor STC8H1KXX.


```
void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);
```

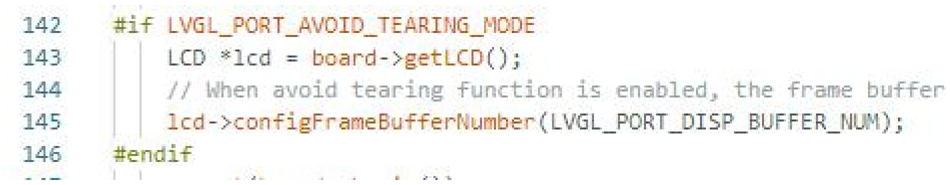
Then the program creates a Board object named "board" using the "ESP32_Display_Panel" library, which is used to uniformly manage the display bus, the LCD controller, and the touch device. Subsequently, it prints "Initializing Panel (ST7265 + GT911)..." indicating that the system is initializing the display and touch

hardware. The LCD driver chip is ST7265 and the touch controller is GT911.
The program initializes the display bus (RGB interface) and related devices through "board->init()". If the initialization fails, the assert() function will trigger the program to stop.



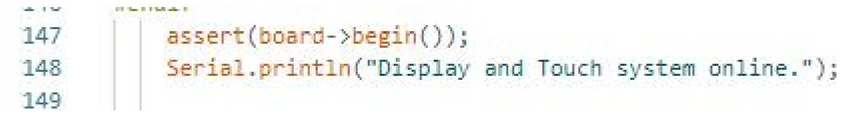
```
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
```

Next, under the conditional compilation #if LVGL_PORT_AVOID_TEARING_MODE, if the tearing avoidance mode is enabled, the program will obtain the LCD object board->getLCD() and call configFrameBufferNumber() to set the number of frame buffers, in order to reduce the problem of image tearing that occurs during screen refresh.



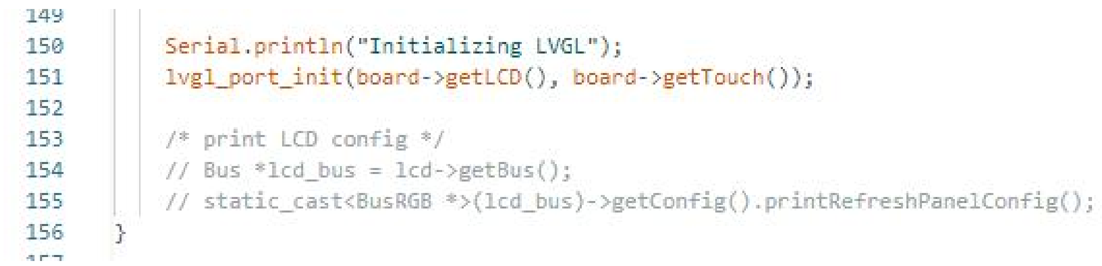
```
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
```

Then, call board->begin() to start the display panel. This process usually involves the LCD startup sequence, driver configuration, and display interface startup. Once it is successful, print "Display and Touch system online." to indicate that the display and touch system is running normally.



```
    assert(board->begin());
    Serial.println("Display and Touch system online.");
```

Finally, the program initializes the LVGL graphics library. By calling lvgl_port_init(board->getLCD(), board->getTouch()), the LCD display device and touch input device are registered in LVGL, enabling LVGL to handle screen drawing and touch event processing. The function also retains a section of commented-out debugging code at the end, which is used to obtain the RGB bus configuration and print the LCD refresh parameters.

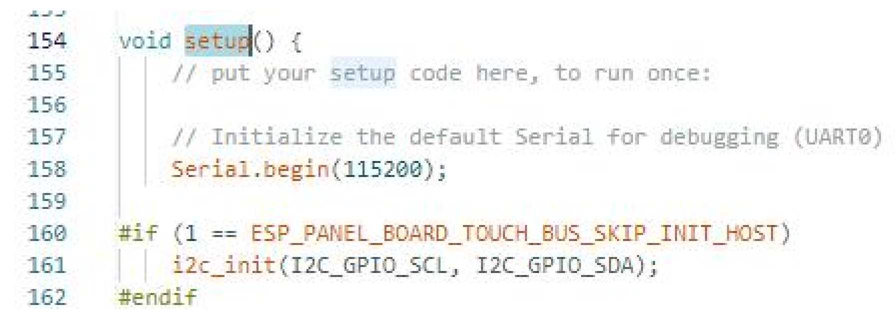


```
    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());

    /* print LCD config */
    // Bus *lcd_bus = lcd->getBus();
    // static_cast<BusRGB *>(lcd_bus)->getConfig().printRefreshPanelConfig();
}
```

4.setup

This setup() function is the system initialization entry point for the entire program. In Arduino-based programs, it is executed only once when the device is powered on or reset. Its main function is to complete serial debugging initialization, I2C bus initialization, display startup with LVGL graphics system, WiFi network connection, weather data acquisition, time synchronization, interface display, backlight activation, and a series of other initialization operations.

The function begins by initializing the default serial port (UART0) through Serial.begin(115200) for subsequent output of debugging information in the serial monitor; then, in the conditional compilation statement #if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST), it calls i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) to initialize the I2C bus, which is mainly used for communication with the touch chip.

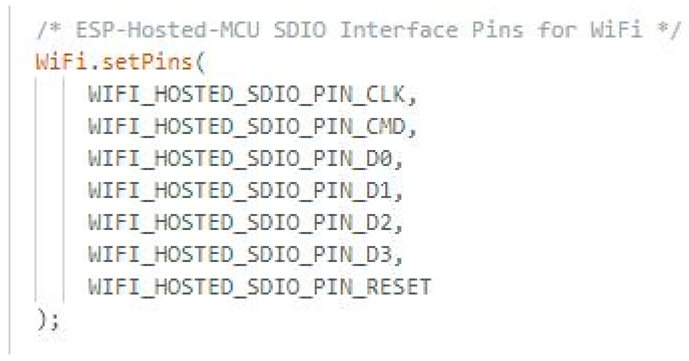


```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif
```

Then the program calls the function display_touch_lvgl_init(), which completes the initialization of the LCD display screen, the touch controller, and the LVGL graphics library, enabling the system to have the ability to display a graphical interface.

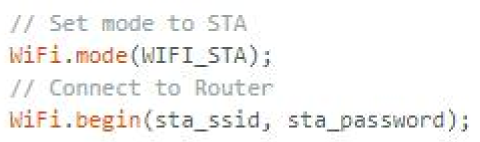
```
    display_touch_lvgl_init();
```

Then the program configures the WiFi hardware interface and sets the SDIO communication pins through WiFi.setPins(). This is because this system uses the ESP32-P4 main controller to communicate with the WiFi coprocessor via SDIO.



```
/* ESP-Hosted-MCU SDIO Interface Pins for WiFi */
WiFi.setPins(
    WIFI_HOSTED_SDIO_PIN_CLK,
    WIFI_HOSTED_SDIO_PIN_CMD,
    WIFI_HOSTED_SDIO_PIN_D0,
    WIFI_HOSTED_SDIO_PIN_D1,
    WIFI_HOSTED_SDIO_PIN_D2,
    WIFI_HOSTED_SDIO_PIN_D3,
    WIFI_HOSTED_SDIO_PIN_RESET
);
```

Then, by using WiFi.mode(WIFI_STA), the device is set to STA (Station) mode, and the WiFi.begin(sta_ssid, sta_password) function is called to connect to the specified router.



```
// Set mode to STA
WiFi.mode(WIFI_STA);
// Connect to Router
WiFi.begin(sta_ssid, sta_password);
```

The program enters a while loop (while (WiFi.status() != WL_CONNECTED)) to wait for a successful network connection. It prints a dot every 500 milliseconds. Once the connection is successful, it prints the IP address obtained by the device.

```
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
Serial.print("\n");
Serial.print("[STA] Connected. IP: ");
Serial.println(WiFi.localIP());
```

After the network connection is completed, the program creates a weather service object through the function weather_create() and calls the function weather_get_weather() to obtain weather data from the network interface, including the temperature temp_c, the weather description string weather_text, and the timestamp returned by the server;

```
weather_t* weather_handle = weather_create();
double temp_c = 0.0;
int timestamp = 0;
if (weather_get_weather(weather_handle, &temp_c, weather_text, &timestamp)) {
```

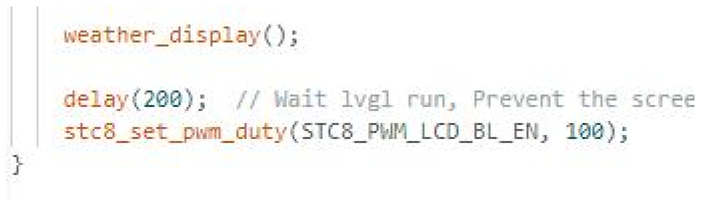
If the acquisition is successful, use snprintf() to format the temperature into a string with the unit and store it in temp_text, such as "25.4° C". At the same time, construct a timeval structure using the timestamp and call settimeofday() to synchronize the system time.

```
weather_t* weather_handle = weather_create();
double temp_c = 0.0;
int timestamp = 0;
if (weather_get_weather(weather_handle, &temp_c, weather_text, &timestamp))
    snprintf(temp_text, sizeof(temp_text), "%.1lf°C", temp_c);
    struct timeval tv = {
        .tv_sec = 0,  // second
        .tv_usec = 0,   // Microsecond (0-999999)
    };
    tv.tv_sec = timestamp;
    settimeofday(&tv, NULL);
```

Then, using time().localtime() and strftime(), the current time is converted into a readable format and separate date strings (such as "2026/03/13") and week strings (such as "Friday") are generated. Finally, log debugging information is output using ESP_LOGI().

```
            time_t now = time(NULL);
            struct tm *local_time;
            local_time = localtime(&now);  // Convert to local time
            strftime(date_str, sizeof(date_str), "%Y/%m/%d", local_time);
            strftime(week_str, sizeof(week_str), "%A", local_time);
            ESP_LOGI(TAG, "time(NULL): %d", (int)time(NULL));
        }
```

Next, the program calls the function weather_display() to create and display the weather interface, and then draws the temperature, weather, date and day of the week information onto the screen.

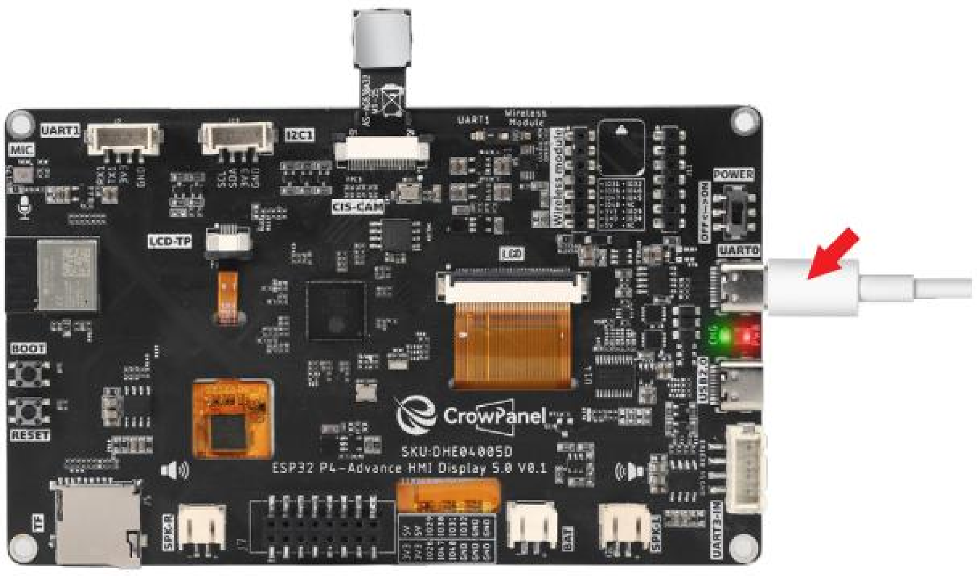


```
    weather_display();

    delay(200);  // Wait lvgl run, Prevent the scree
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);
}
```

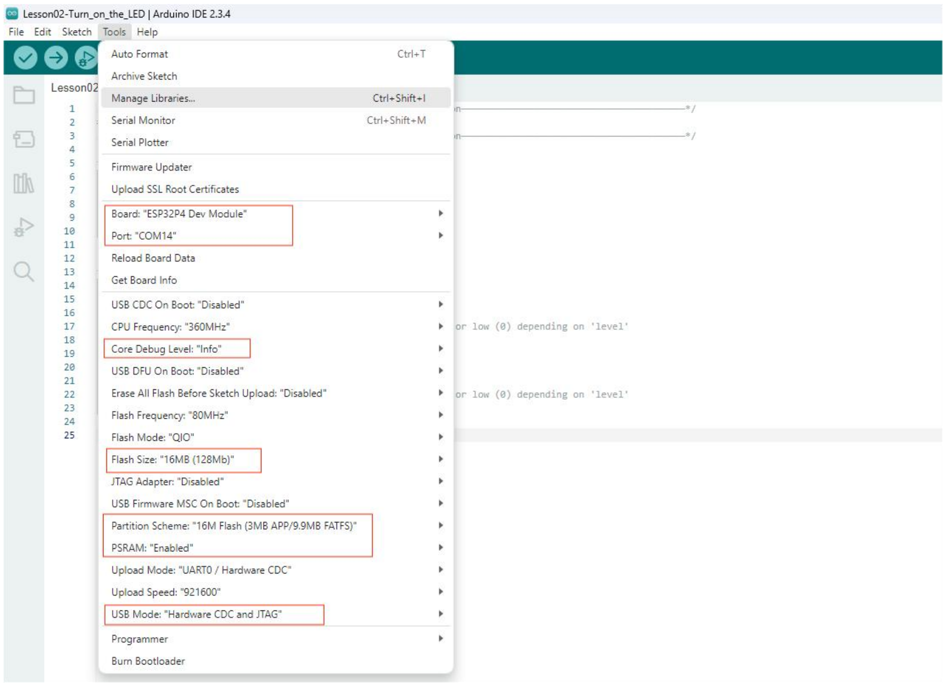
Finally, by using delay(200), a little time is given to the LVGL rendering system to complete the interface refresh, to prevent flickering when the screen is initialized. Then, the function stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100) is called to turn on the LCD backlight and set the brightness to 100%, so that the user can see the final weather display interface.

Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



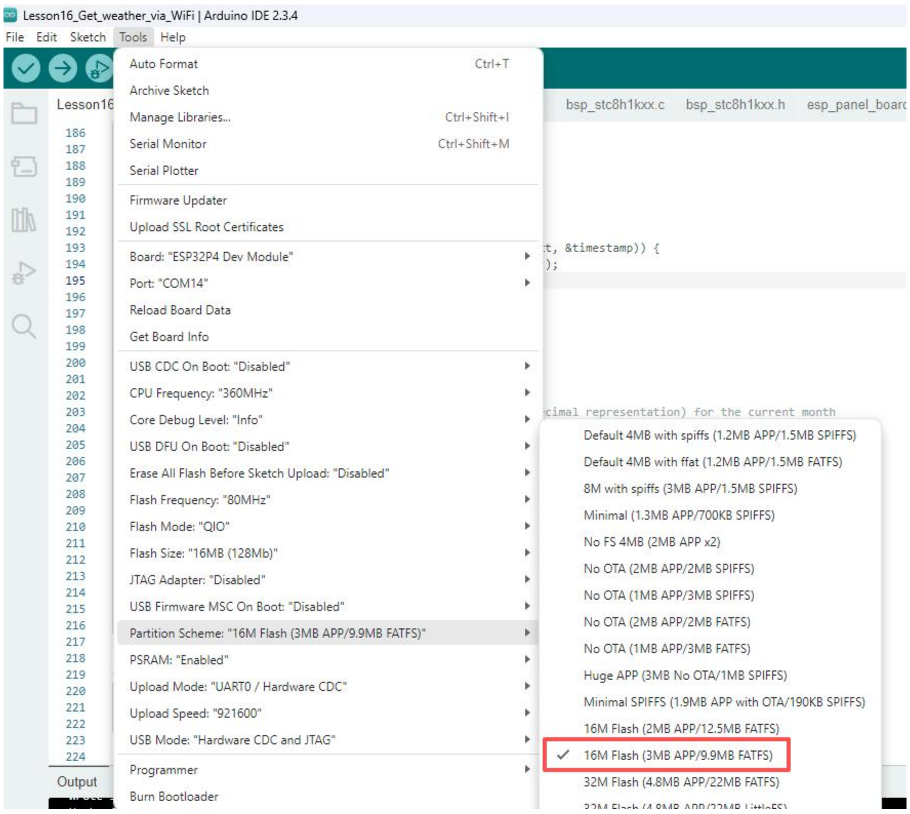
Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

File Edit Sketch Tools Help

```
Auto Format                                      Ctrl+T
Archive Sketch
Manage Libraries...                              Ctrl+Shift+I
Serial Monitor                                   Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader
```

Attention in this class!!

Due to our previous operations, we added images, which will cause the memory used for compiling this project to be too large, resulting in compilation failure. Therefore, we need to expand the partition table used to make the code compile successfully.



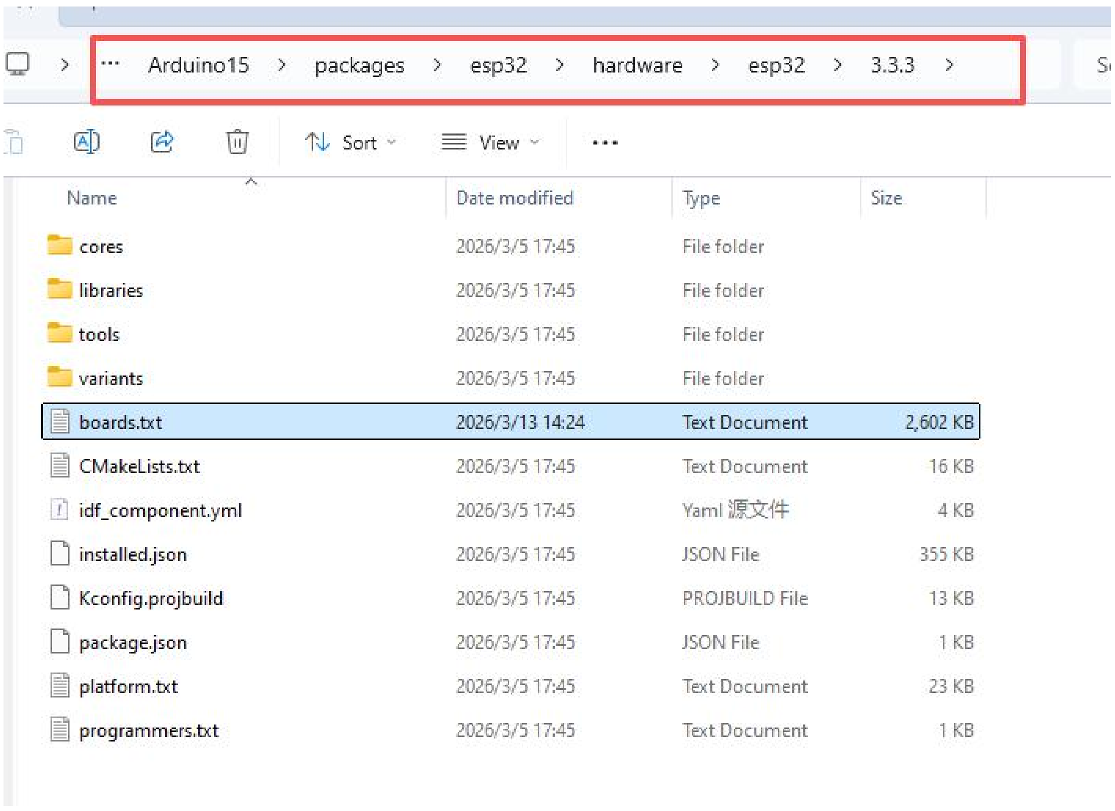
Lesson16_Get_weather_via_WiFi | Arduino IDE 2.3.4

File Edit Sketch Tools Help

```
Auto Format                                      Ctrl+T
Archive Sketch
Manage Libraries...                              Ctrl+Shift+I
Serial Monitor                                   Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader
```

```
Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS)
Default 4MB with ffat (1.2MB APP/1.5MB FATFS)
8M with spiffs (3MB APP/1.5MB SPIFFS)
Minimal (1.3MB APP/700KB SPIFFS)
No FS 4MB (2MB APP x2)
No OTA (2MB APP/2MB SPIFFS)
No OTA (1MB APP/3MB SPIFFS)
No OTA (2MB APP/2MB FATFS)
No OTA (1MB APP/3MB FATFS)
Huge APP (3MB No OTA/1MB SPIFFS)
Minimal SPIFFS (1.9MB APP with OTA/190KB SPIFFS)
16M Flash (2MB APP/12.5MB FATFS)
✓ 16M Flash (3MB APP/9.9MB FATFS)
32M Flash (4.8MB APP/22MB FATFS)
```

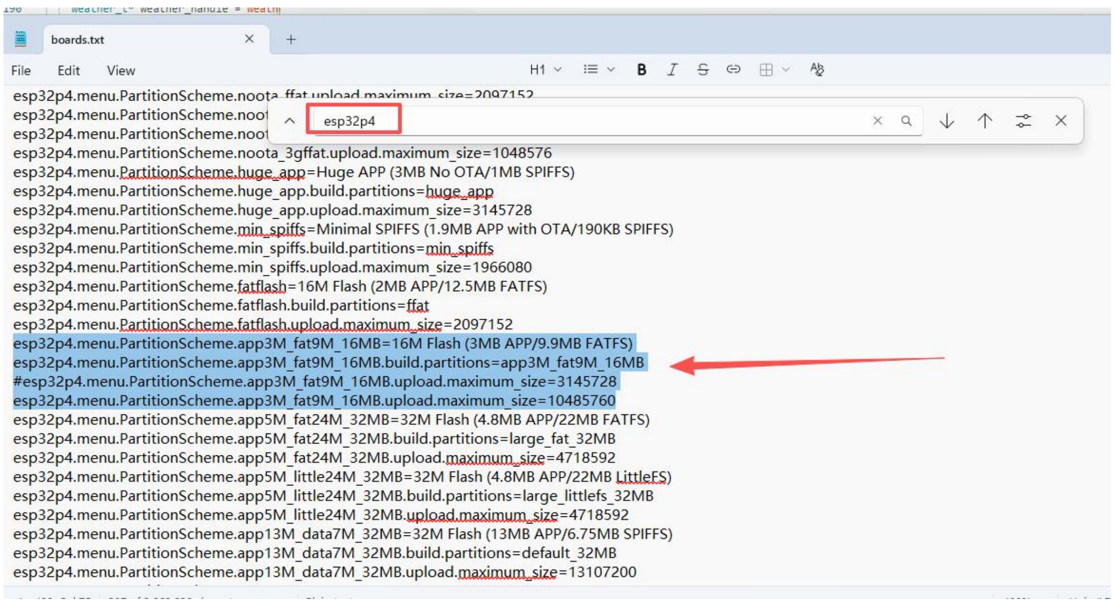
Come to this path of the file system



```
... > Arduino15 > packages > esp32 > hardware > esp32 > 3.3.3 >
```

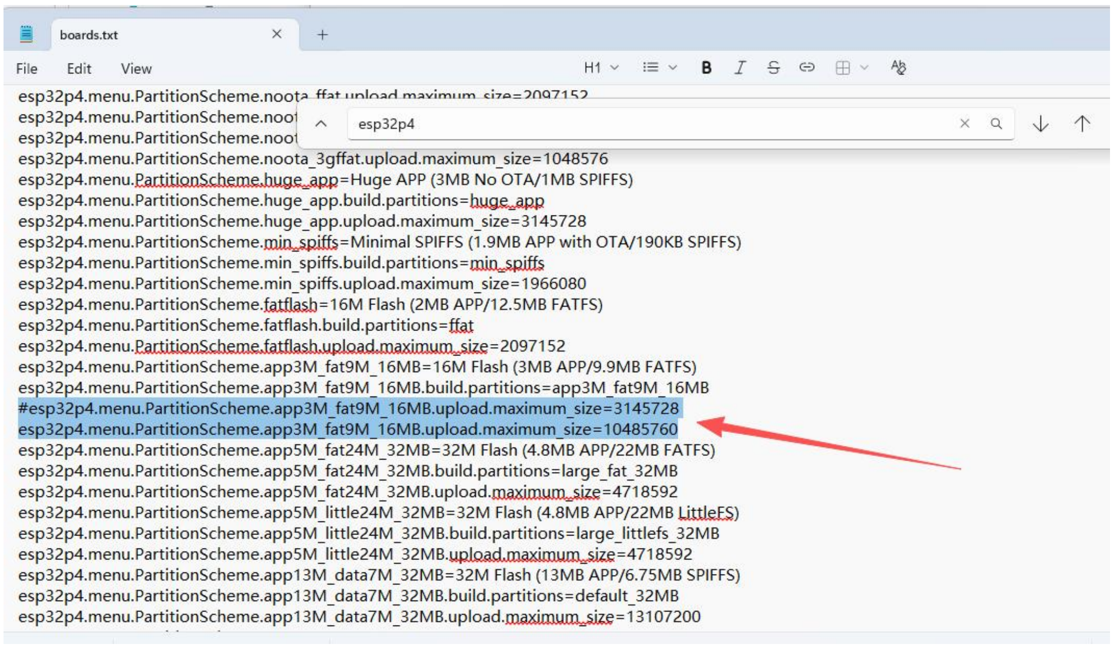
Name	Date modified	Type	Size
cores	2026/3/5 17:45	File folder	
libraries	2026/3/5 17:45	File folder	
tools	2026/3/5 17:45	File folder	
variants	2026/3/5 17:45	File folder	
boards.txt	2026/3/13 14:24	Text Document	2,602 KB
CMakeLists.txt	2026/3/5 17:45	Text Document	16 KB
idf_component.yml	2026/3/5 17:45	Yaml 源文件	4 KB
installed.json	2026/3/5 17:45	JSON File	355 KB
Kconfig.projbuild	2026/3/5 17:45	PROJBUILD File	13 KB
package.json	2026/3/5 17:45	JSON File	1 KB
platform.txt	2026/3/5 17:45	Text Document	23 KB
programmers.txt	2026/3/5 17:45	Text Document	1 KB

Open the "boards.txt" file and search for "esp32p4". Locate this section of the area.



```
esp32p4.menu.PartitionScheme.noota_ffat.upload.maximum_size=2097152
esp32p4.menu.PartitionScheme.noot
esp32p4.menu.PartitionScheme.noot
esp32p4.menu.PartitionScheme.noota_3gffat.upload.maximum_size=1048576
esp32p4.menu.PartitionScheme.huge_app=Huge APP (3MB No OTA/1MB SPIFFS)
esp32p4.menu.PartitionScheme.huge_app.build.partitions=huge_app
esp32p4.menu.PartitionScheme.huge_app.upload.maximum_size=3145728
esp32p4.menu.PartitionScheme.min_spiffs=Minimal SPIFFS (1.9MB APP with OTA/190KB SPIFFS)
esp32p4.menu.PartitionScheme.min_spiffs.build.partitions=min_spiffs
esp32p4.menu.PartitionScheme.min_spiffs.upload.maximum_size=1966080
esp32p4.menu.PartitionScheme.fatflash=16M Flash (2MB APP/12.5MB FATFS)
esp32p4.menu.PartitionScheme.fatflash.build.partitions=ffat
esp32p4.menu.PartitionScheme.fatflash.upload.maximum_size=2097152
esp32p4.menu.PartitionScheme.app3M_fat9M_16MB=16M Flash (3MB APP/9.9MB FATFS)
esp32p4.menu.PartitionScheme.app3M_fat9M_16MB.build.partitions=app3M_fat9M_16MB
#esp32p4.menu.PartitionScheme.app3M_fat9M_16MB.upload.maximum_size=3145728
esp32p4.menu.PartitionScheme.app3M_fat9M_16MB.upload.maximum_size=10485760
esp32p4.menu.PartitionScheme.app5M_fat24M_32MB=32M Flash (4.8MB APP/22MB FATFS)
esp32p4.menu.PartitionScheme.app5M_fat24M_32MB.build.partitions=large_fat_32MB
esp32p4.menu.PartitionScheme.app5M_fat24M_32MB.upload.maximum_size=4718592
esp32p4.menu.PartitionScheme.app5M_little24M_32MB=32M Flash (4.8MB APP/22MB LittleFS)
esp32p4.menu.PartitionScheme.app5M_little24M_32MB.build.partitions=large_littlefs_32MB
esp32p4.menu.PartitionScheme.app5M_little24M_32MB.upload.maximum_size=4718592
esp32p4.menu.PartitionScheme.app13M_data7M_32MB=32M Flash (13MB APP/6.75MB SPIFFS)
esp32p4.menu.PartitionScheme.app13M_data7M_32MB.build.partitions=default_32MB
esp32p4.menu.PartitionScheme.app13M_data7M_32MB.upload.maximum_size=13107200
```

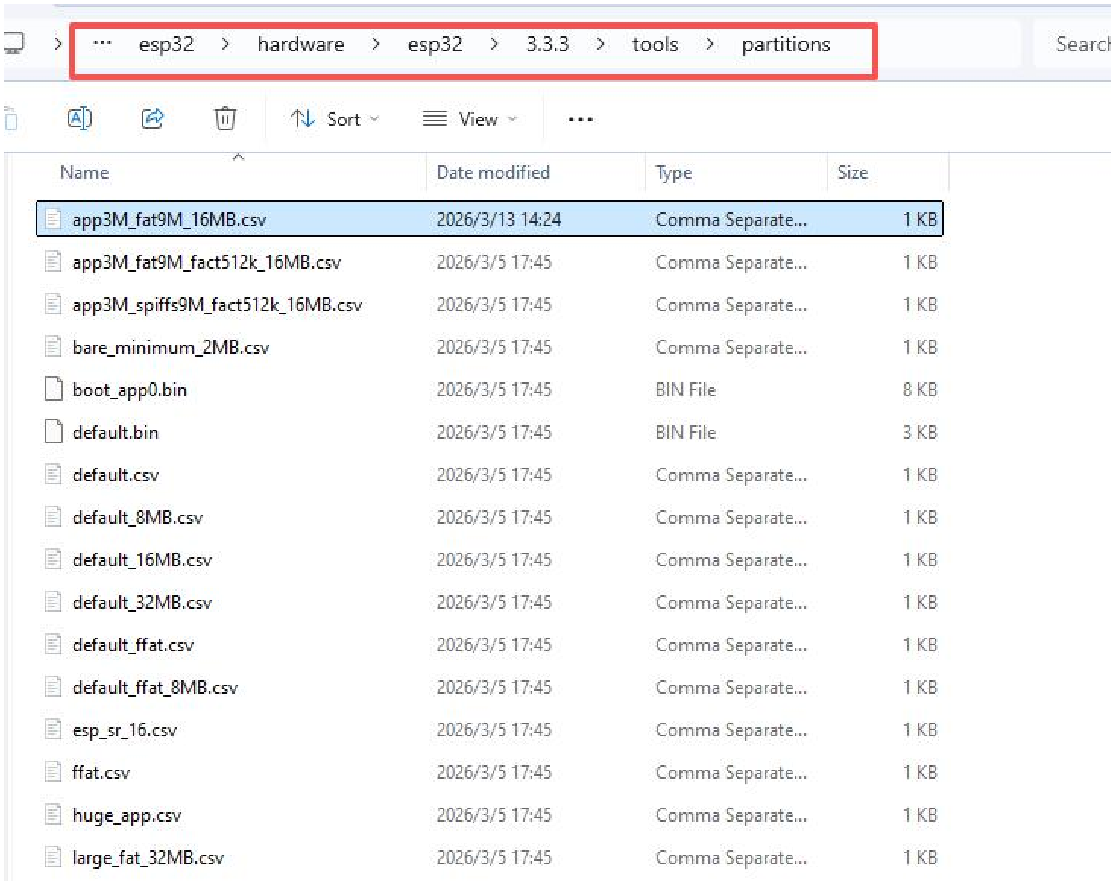
First, add a "#" symbol in front of this size for annotation, and then add a new size.



```
esp32p4.menu.PartitionScheme.noota_ffat.upload.maximum_size=2097152
esp32p4.menu.PartitionScheme.noot
esp32p4.menu.PartitionScheme.noot
esp32p4.menu.PartitionScheme.noota_3gffat.upload.maximum_size=1048576
esp32p4.menu.PartitionScheme.huge_app=Huge APP (3MB No OTA/1MB SPIFFS)
esp32p4.menu.PartitionScheme.huge_app.build.partitions=huge_app
esp32p4.menu.PartitionScheme.huge_app.upload.maximum_size=3145728
esp32p4.menu.PartitionScheme.min_spiffs=Minimal SPIFFS (1.9MB APP with OTA/190KB SPIFFS)
esp32p4.menu.PartitionScheme.min_spiffs.build.partitions=min_spiffs
esp32p4.menu.PartitionScheme.min_spiffs.upload.maximum_size=1966080
esp32p4.menu.PartitionScheme.fatflash=16M Flash (2MB APP/12.5MB FATFS)
esp32p4.menu.PartitionScheme.fatflash.build.partitions=ffat
esp32p4.menu.PartitionScheme.fatflash.upload.maximum_size=2097152
esp32p4.menu.PartitionScheme.app3M_fat9M_16MB=16M Flash (3MB APP/9.9MB FATFS)
esp32p4.menu.PartitionScheme.app3M_fat9M_16MB.build.partitions=app3M_fat9M_16MB
#esp32p4.menu.PartitionScheme.app3M_fat9M_16MB.upload.maximum_size=3145728
esp32p4.menu.PartitionScheme.app3M_fat9M_16MB.upload.maximum_size=10485760
esp32p4.menu.PartitionScheme.app5M_fat24M_32MB=32M Flash (4.8MB APP/22MB FATFS)
esp32p4.menu.PartitionScheme.app5M_fat24M_32MB.build.partitions=large_fat_32MB
esp32p4.menu.PartitionScheme.app5M_fat24M_32MB.upload.maximum_size=4718592
esp32p4.menu.PartitionScheme.app5M_little24M_32MB=32M Flash (4.8MB APP/22MB LittleFS)
esp32p4.menu.PartitionScheme.app5M_little24M_32MB.build.partitions=large_littlefs_32MB
esp32p4.menu.PartitionScheme.app5M_little24M_32MB.upload.maximum_size=4718592
esp32p4.menu.PartitionScheme.app13M_data7M_32MB=32M Flash (13MB APP/6.75MB SPIFFS)
esp32p4.menu.PartitionScheme.app13M_data7M_32MB.build.partitions=default_32MB
esp32p4.menu.PartitionScheme.app13M_data7M_32MB.upload.maximum_size=13107200
```

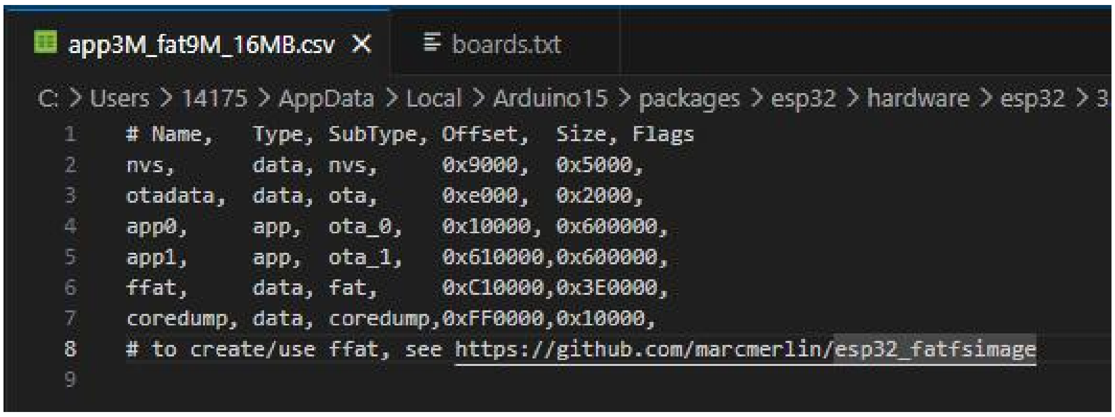
The size of this partition table occupies 10,485,760 bytes. This is because we modified this partition table.

Go to the following path and find app3M_fat9M_16MB.csv



Name	Date modified	Type	Size
app3M_fat9M_16MB.csv	2026/3/13 14:24	Comma Separate...	1 KB
app3M_fat9M_fact512k_16MB.csv	2026/3/5 17:45	Comma Separate...	1 KB
app3M_spiffs9M_fact512k_16MB.csv	2026/3/5 17:45	Comma Separate...	1 KB
bare_minimum_2MB.csv	2026/3/5 17:45	Comma Separate...	1 KB
boot_app0.bin	2026/3/5 17:45	BIN File	8 KB
default.bin	2026/3/5 17:45	BIN File	3 KB
default.csv	2026/3/5 17:45	Comma Separate...	1 KB
default_8MB.csv	2026/3/5 17:45	Comma Separate...	1 KB
default_16MB.csv	2026/3/5 17:45	Comma Separate...	1 KB
default_32MB.csv	2026/3/5 17:45	Comma Separate...	1 KB
default_ffat.csv	2026/3/5 17:45	Comma Separate...	1 KB
default_ffat_8MB.csv	2026/3/5 17:45	Comma Separate...	1 KB
esp_sr_16.csv	2026/3/5 17:45	Comma Separate...	1 KB
ffat.csv	2026/3/5 17:45	Comma Separate...	1 KB
huge_app.csv	2026/3/5 17:45	Comma Separate...	1 KB
large_fat_32MB.csv	2026/3/5 17:45	Comma Separate...	1 KB

Use the contents of our partition table



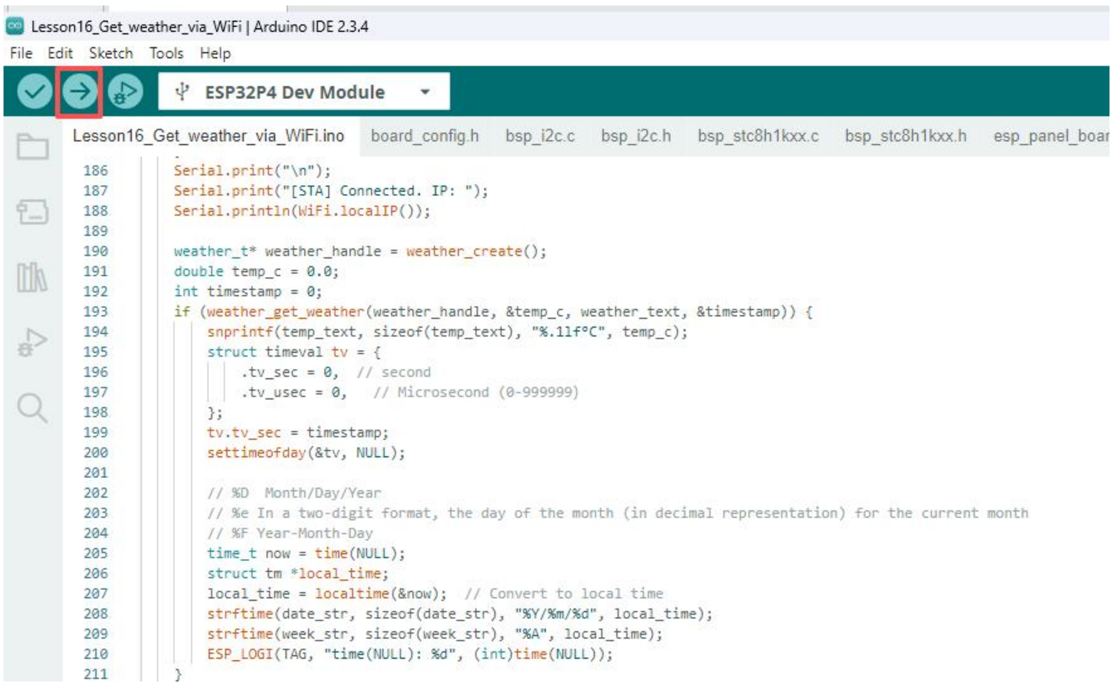
app3M_fat9M_16MB.csv
boards.txt
C: > Users > 14175 > AppData > Local > Arduino15 > packages > esp32 > hardware > esp32 > 3

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x5000,
otadata,  data, ota,     0xe000,  0x2000,
app0,     app,  ota_0,   0x10000, 0x600000,
app1,     app,  ota_1,   0x610000,0x600000,
ffat,     data, fat,     0xC10000,0x3E0000,
coredump, data, coredump,0xFF0000,0x10000,
# to create/use ffat, see https://github.com/marcmerlin/esp32_fatfsimage
```

```
# Name,     Type, SubType, Offset,   Size, Flags
nvs,          data, nvs,        0x9000,   0x5000,
otadata,    data, ota,          0xe000,   0x2000,
app0,         app,   ota_0,      0x10000, 0x600000,
app1,         app,   ota_1,      0x610000,0x600000,
ffat,          data, fat,          0xC10000,0x3E0000,
coredump, data, coredump,0xFF0000,0x10000,
# to create/use ffat, see https://github.com/marcmerlin/esp32_fatfsimage
```

After making the revisions, be sure to save the document.

After that, we will compile and burn the code.

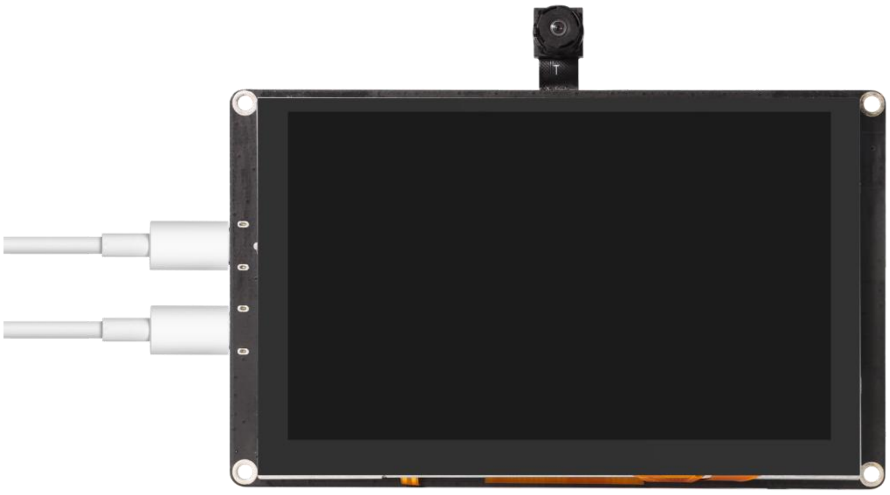


Lesson16_Get_weather_via_WiFi | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson16_Get_weather_via_WiFi.ino
board_config.h
bsp_i2c.c
bsp_i2c.h
bsp_stc8h1kxx.c
bsp_stc8h1kxx.h
esp_panel_boar

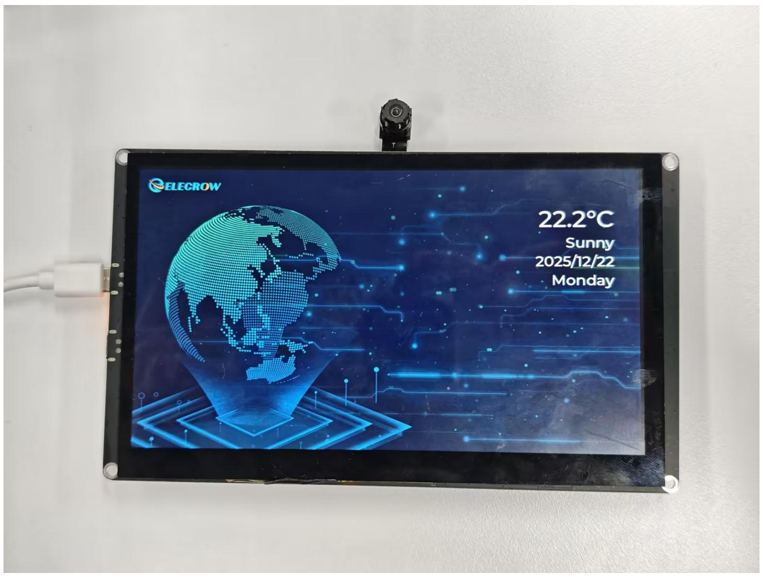
```
    Serial.print("\n");
    Serial.print("[STA] Connected. IP: ");
    Serial.println(WiFi.localIP());

    weather_t* weather_handle = weather_create();
    double temp_c = 0.0;
    int timestamp = 0;
    if (weather_get_weather(weather_handle, &temp_c, weather_text, &timestamp)) {
      snprintf(temp_text, sizeof(temp_text), "%.1lf°C", temp_c);
      struct timeval tv = {
        .tv_sec = 0,  // second
        .tv_usec = 0,   // Microsecond (0-999999)
      };
      tv.tv_sec = timestamp;
      settimeofday(&tv, NULL);

      // %D  Month/Day/Year
      // %e In a two-digit format, the day of the month (in decimal representation) for the current month
      // %F Year-Month-Day
      time_t now = time(NULL);
      struct tm *local_time;
      local_time = localtime(&now);  // Convert to local time
      strftime(date_str, sizeof(date_str), "%Y/%m/%d", local_time);
      strftime(week_str, sizeof(week_str), "%A", local_time);
      ESP_LOGI(TAG, "time(NULL): %d", (int)time(NULL));
    }
```

After waiting for a while, the code compilation and upload were completed.

At this point, please remember to use another Type-C cable to connect your Advance-P4 through the USB2.0 interface. This interface provides a maximum current of about 500mA for the computer's USB-A interface. When the Advance-P4 is using more external devices, especially the screen, it needs a sufficient current source. (It is recommended to use a charger for connection.)



After running the code, you will be able to see the local weather conditions you have obtained on the screen of the Advance-P4.



Lesson14--- SX1262 Wireless Module

Introduction

In this lesson, we will begin exploring the use of wireless modules. Since the SX1262 LoRa module supports both transmission and reception, two Advance-P4 development boards and two SX1262 LoRa communication modules are required.

The objective of this lesson is to implement a case study where, when an SX1262 LoRa module is connected to the wireless module slot of the Advance-P4 board, the transmitting board displays "**TX_Hello World:i**" on its screen, while the receiving board displays "**RX_Hello World:i**" along with related LoRa signal information.

Learning Goals

1. Understand the working principles of the SX1262 LoRa module (SPI communication, wireless transceiver mechanism), as well as the hardware pin adaptation rules between ESP32-P4 and the LoRa module (reset, DIO interrupt, SPI chip select, etc.).

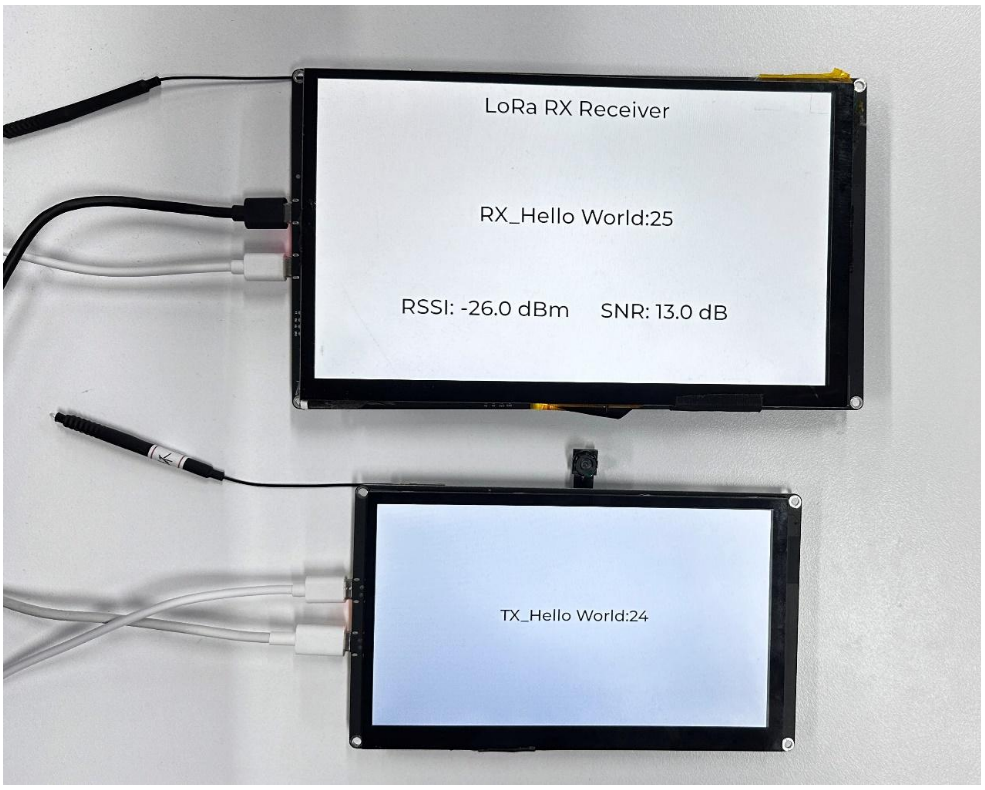
2. Master the core implementation methods for SX1262 module initialization configuration (frequency, bandwidth, spreading factor, etc.), data transmission (timed packet sending, transmission status detection), and reception (interrupt callbacks, RSSI/SNR parsing).

3. Master the design of LoRa transceiver logic based on FreeRTOS multi-tasking (separation of transmission task, reception task, and UI update task), as well as the implementation of dynamic LVGL interface updates (transmission/reception counters, signal parameter display).

Preview of the Result

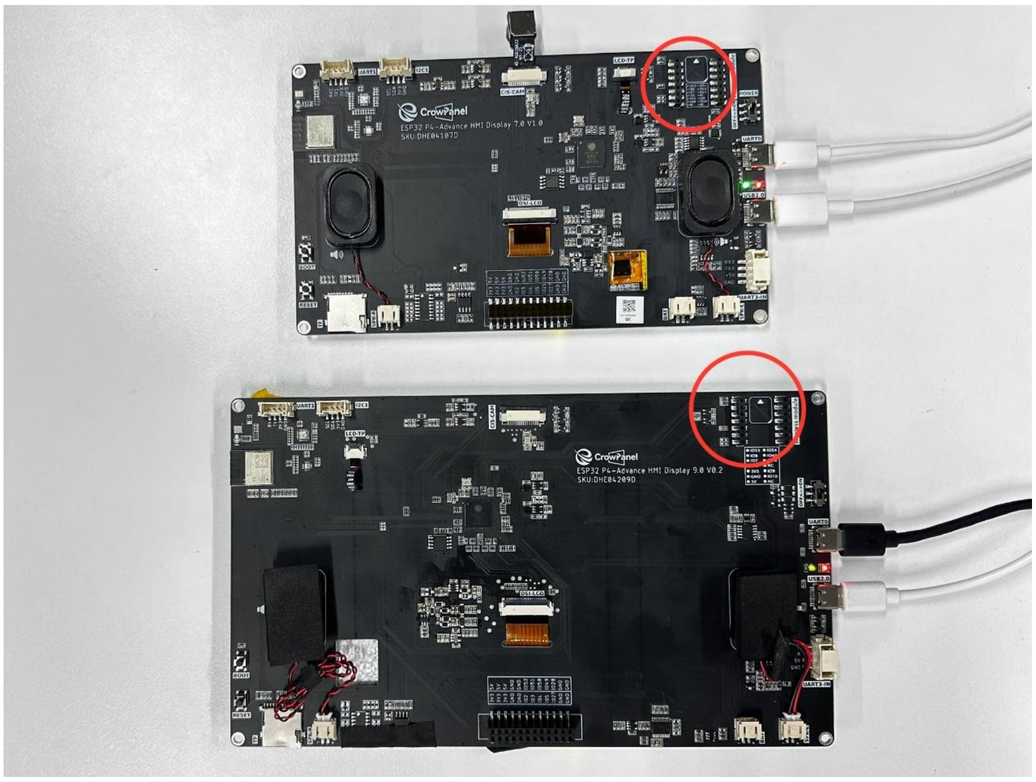
After inserting the SX1262 LoRa modules into both Advance-P4 development boards and running the respective codes, you will observe the following behavior:
On the transmitting Advance-P4 board, the screen will display the message TX_Hello World:i, with the value of i increasing by 1 every second.

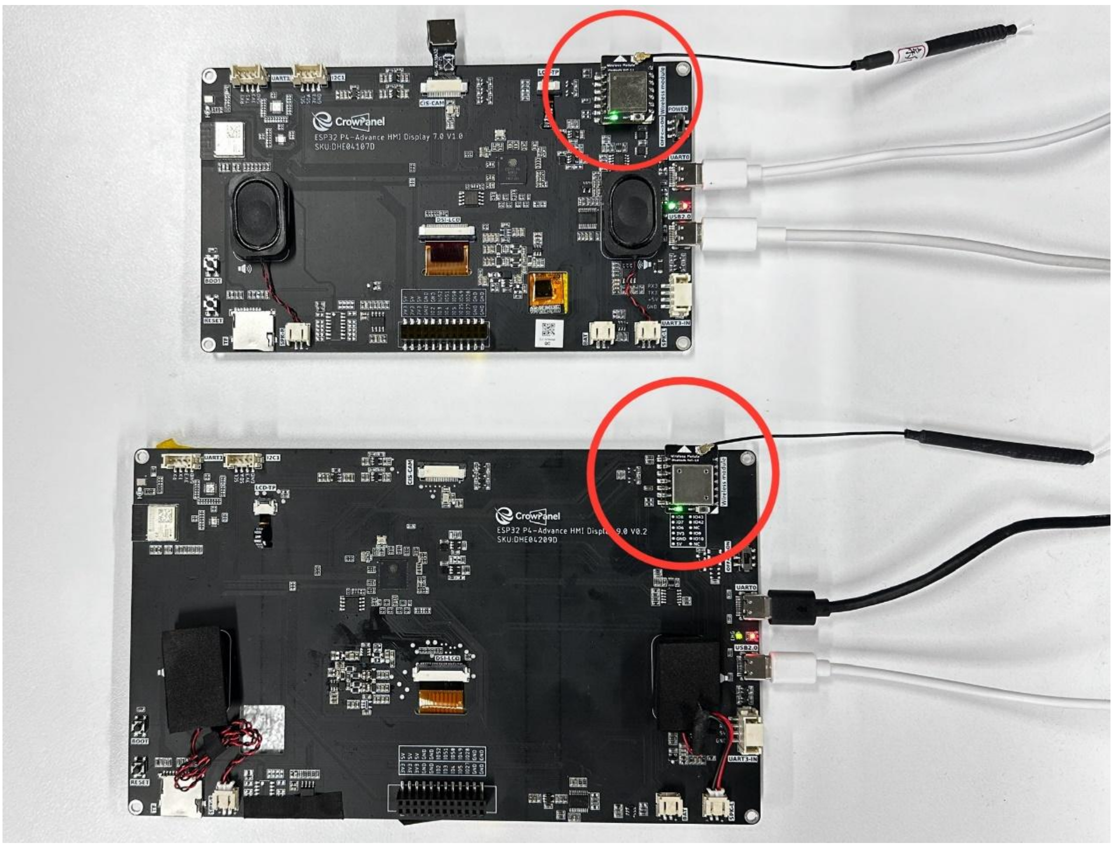
Similarly, on the receiving Advance-P4 board, the screen will display RX_Hello World:i whenever a message is received, with i also incrementing by 1 each second. In addition, the screen will show relevant reception signal information such as RSSI and SNR.



Hardware Used in This Lesson

SX1262 Wireless Module on the Advance-P4





Complete Code

First, click the GitHub link below to download the code for this lesson.

Kindly click the link below to view the full code implementation.

Transmitting end code:
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson14_TX_SX1262_Wireless_Module

Receiving end code:
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson14_RX_SX1262_Wireless_Module

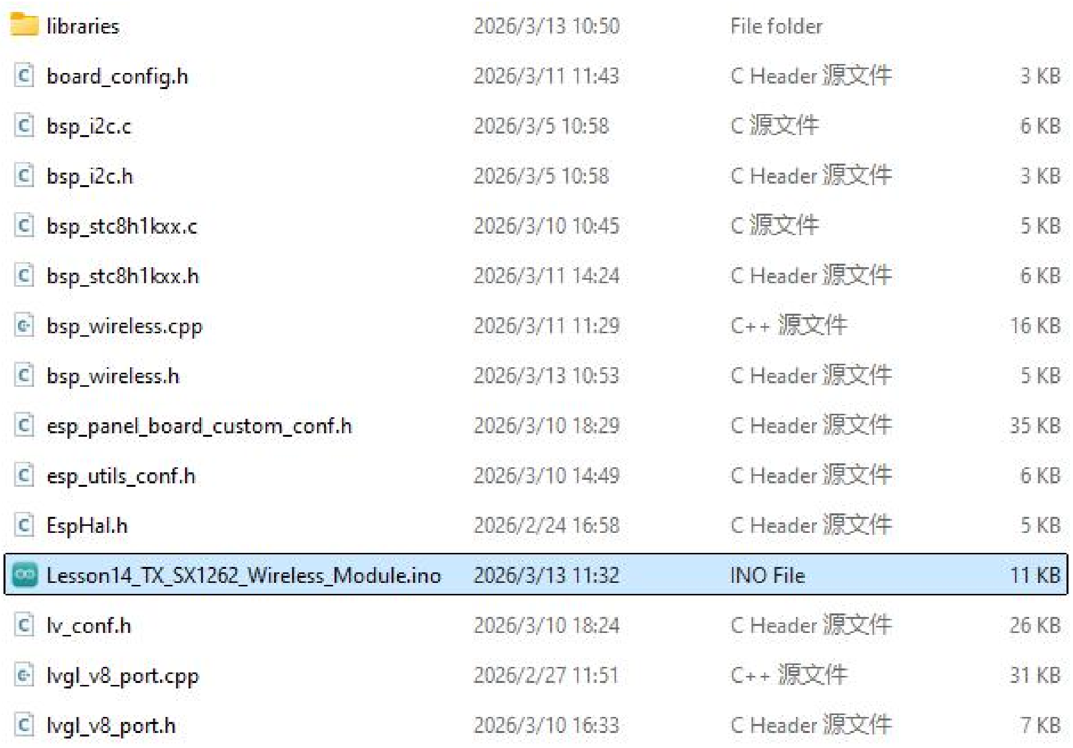
Key Explanations

The main focus of this lesson is to learn how to use the wireless module, including how to initialize the SX1262 LoRa module and send or receive data.

Double-click to open the code for this lesson (the .ino file).

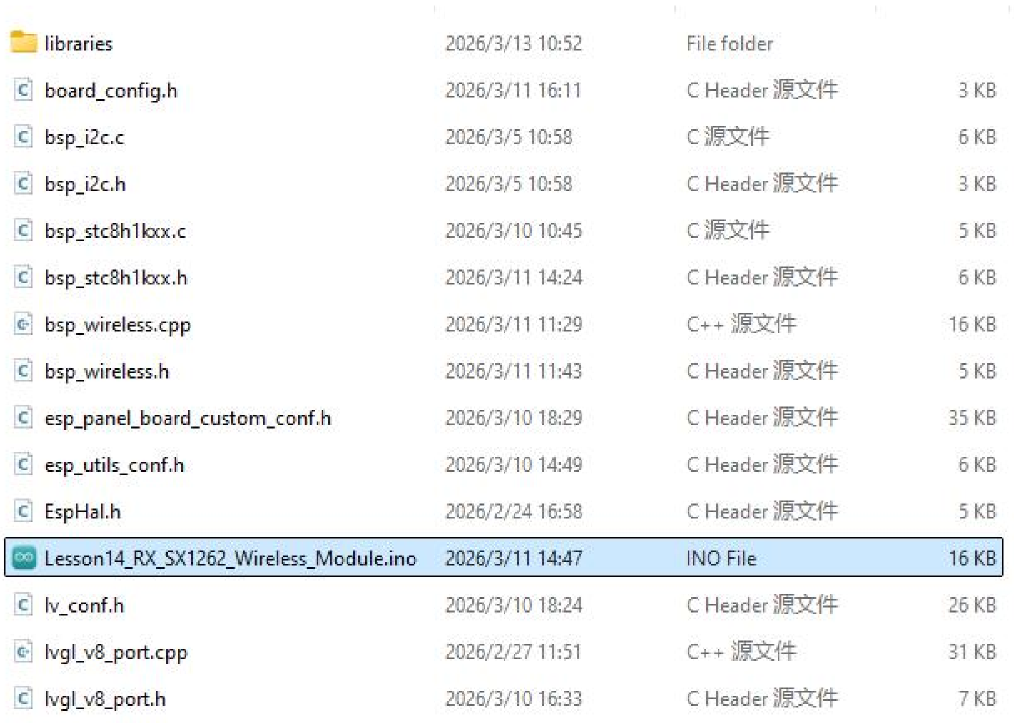
The code for this lesson includes both the transmitter and receiver sides.

Transmitter:



libraries	2026/3/13 10:50	File folder	
board_config.h	2026/3/11 11:43	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/11 14:24	C Header 源文件	6 KB
bsp_wireless.cpp	2026/3/11 11:29	C++ 源文件	16 KB
bsp_wireless.h	2026/3/13 10:53	C Header 源文件	5 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
EspHal.h	2026/2/24 16:58	C Header 源文件	5 KB
Lesson14_TX_SX1262_Wireless_Module.ino	2026/3/13 11:32	INO File	11 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

Receiver:



libraries	2026/3/13 10:52	File folder	
board_config.h	2026/3/11 16:11	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/11 14:24	C Header 源文件	6 KB
bsp_wireless.cpp	2026/3/11 11:29	C++ 源文件	16 KB
bsp_wireless.h	2026/3/11 11:43	C Header 源文件	5 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
EspHal.h	2026/2/24 16:58	C Header 源文件	5 KB
Lesson14_RX_SX1262_Wireless_Module.ino	2026/3/11 14:47	INO File	16 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

In this section, we will introduce a new component called bsp_wireless.

The main functions of this component are as follows:

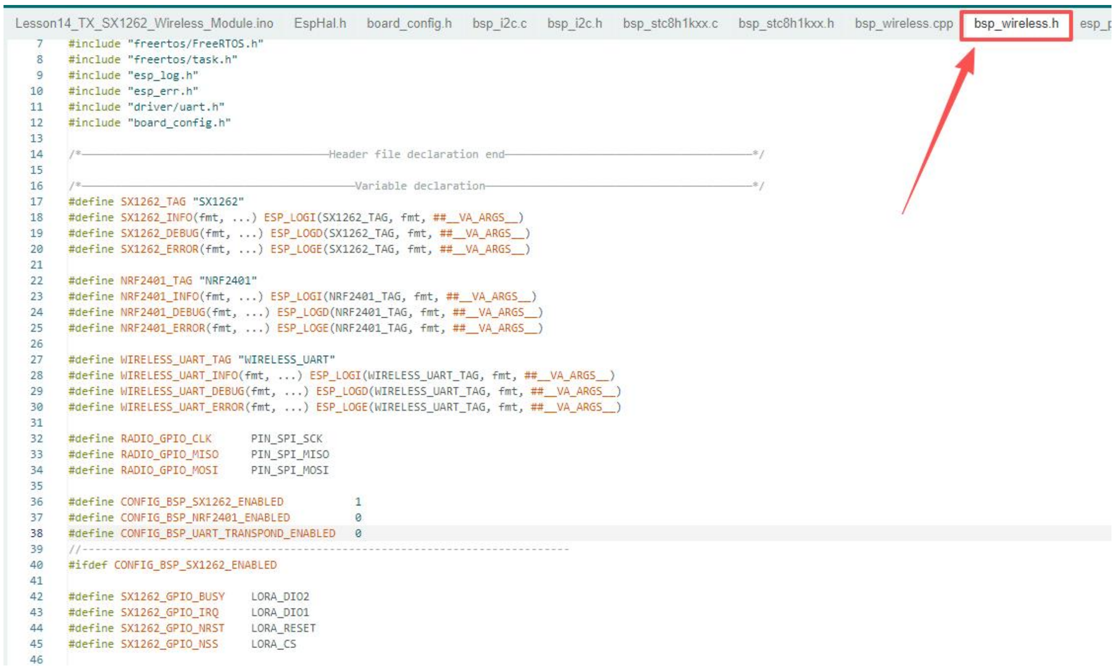
- It encodes and modulates the data (such as strings or sensor information) sent from the main controller and transmits it wirelessly.
- It also receives wireless data packets sent from other devices via LoRa.
- Through a callback mechanism, it passes the received data back to the upper-layer application.

In addition to the above functions, this component also integrates the experimental functionalities for the remaining three wireless modules: nRF2401, ESP32-C6, and ESP32-H2.

Since the functions of each wireless module in the code are encapsulated within #ifdef and #endif directives, and in this lesson we are using the SX1262 module, we only need to enable the SX1262-related configurations.

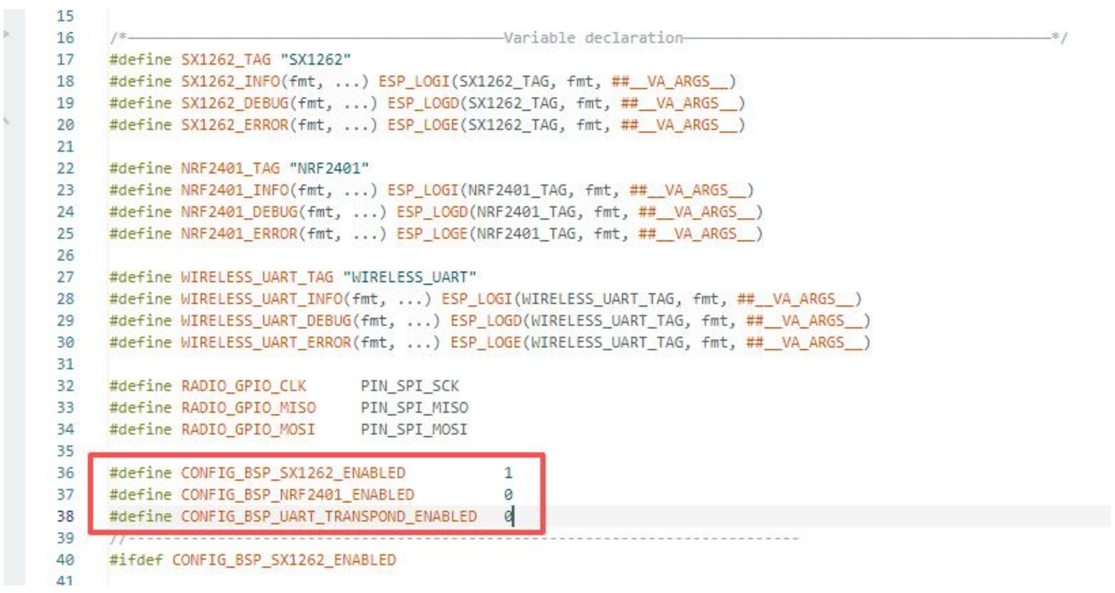
How to enable it:

Open the bsp_wireless.h file.



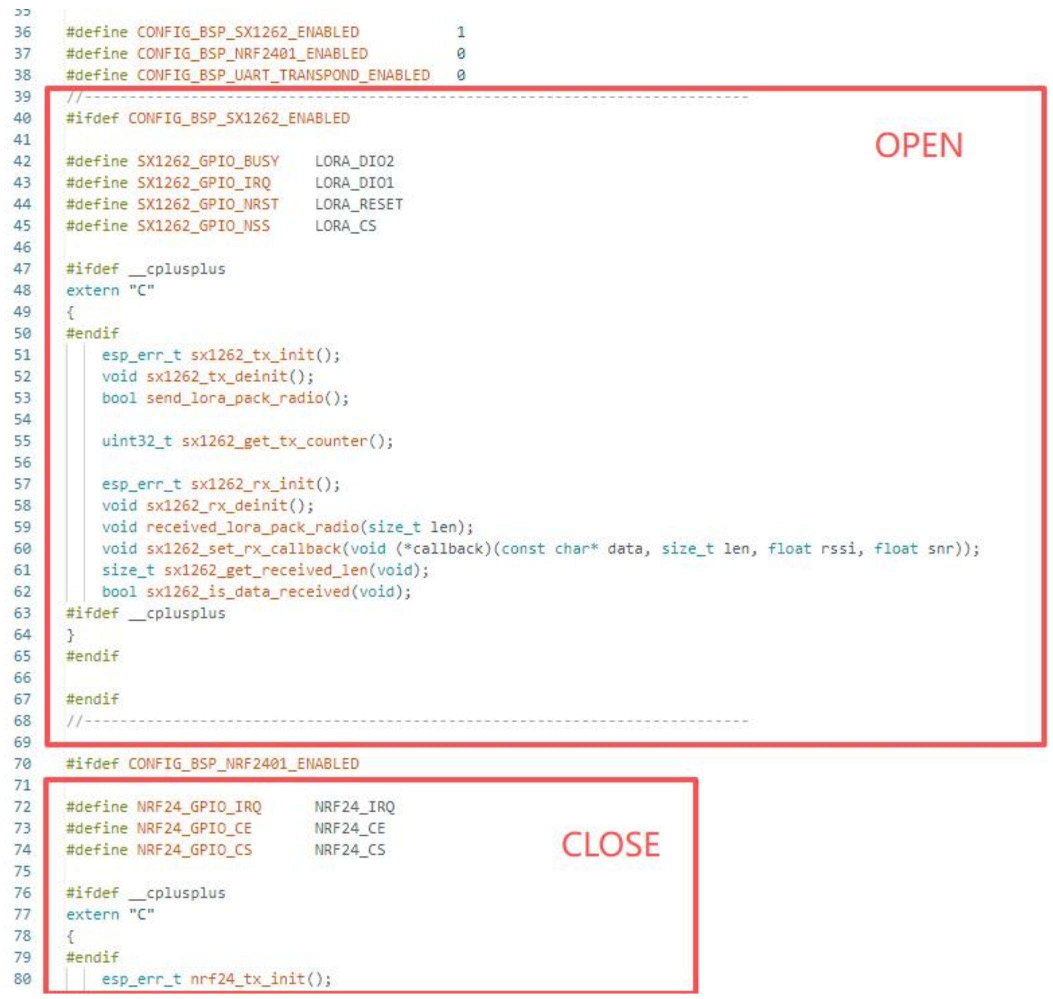
```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/uart.h"
#include "board_config.h"

/*----------------------------------------Header file declaration end----------------------------------------*/

/*----------------------------------------Variable declaration----------------------------------------*/
#define SX1262_TAG "SX1262"
#define SX1262_INFO(fmt, ...) ESP_LOGI(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_DEBUG(fmt, ...) ESP_LOGD(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_ERROR(fmt, ...) ESP_LOGE(SX1262_TAG, fmt, ##__VA_ARGS__)

#define NRF2401_TAG "NRF2401"
#define NRF2401_INFO(fmt, ...) ESP_LOGI(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_DEBUG(fmt, ...) ESP_LOGD(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_ERROR(fmt, ...) ESP_LOGE(NRF2401_TAG, fmt, ##__VA_ARGS__)

#define WIRELESS_UART_TAG "WIRELESS_UART"
#define WIRELESS_UART_INFO(fmt, ...) ESP_LOGI(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_DEBUG(fmt, ...) ESP_LOGD(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_ERROR(fmt, ...) ESP_LOGE(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)

#define RADIO_GPIO_CLK      PIN_SPI_SCK
#define RADIO_GPIO_MISO     PIN_SPI_MISO
#define RADIO_GPIO_MOSI     PIN_SPI_MOSI

#define CONFIG_BSP_SX1262_ENABLED          1
#define CONFIG_BSP_NRF2401_ENABLED         0
#define CONFIG_BSP_UART_TRANSPOND_ENABLED  0
//---------------------------------------------------------------------------
#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY    LORA_DIO2
#define SX1262_GPIO_IRQ     LORA_DIO1
#define SX1262_GPIO_NRST    LORA_RESET
#define SX1262_GPIO_NSS     LORA_CS

```

Here, enable the macro definition for the SX1262 wireless module by setting it to 1. The code you will subsequently use will be SX1262-related, while other wireless modules will remain commented out by default—that is, we set other wireless modules to 0.



```c

/*-------------------------------------Variable declaration-------------------------------------*/
#define SX1262_TAG "SX1262"
#define SX1262_INFO(fmt, ...) ESP_LOGI(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_DEBUG(fmt, ...) ESP_LOGD(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_ERROR(fmt, ...) ESP_LOGE(SX1262_TAG, fmt, ##__VA_ARGS__)

#define NRF2401_TAG "NRF2401"
#define NRF2401_INFO(fmt, ...) ESP_LOGI(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_DEBUG(fmt, ...) ESP_LOGD(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_ERROR(fmt, ...) ESP_LOGE(NRF2401_TAG, fmt, ##__VA_ARGS__)

#define WIRELESS_UART_TAG "WIRELESS_UART"
#define WIRELESS_UART_INFO(fmt, ...) ESP_LOGI(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_DEBUG(fmt, ...) ESP_LOGD(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_ERROR(fmt, ...) ESP_LOGE(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)

#define RADIO_GPIO_CLK       PIN_SPI_SCK
#define RADIO_GPIO_MISO      PIN_SPI_MISO
#define RADIO_GPIO_MOSI      PIN_SPI_MOSI

#define CONFIG_BSP_SX1262_ENABLED           1
#define CONFIG_BSP_NRF2401_ENABLED          0
#define CONFIG_BSP_UART_TRANSPOND_ENABLED   0
//---------------------------------------------------------------------------
#ifdef CONFIG_BSP_SX1262_ENABLED

```

(Enable the one that corresponds to the wireless module you are using.)



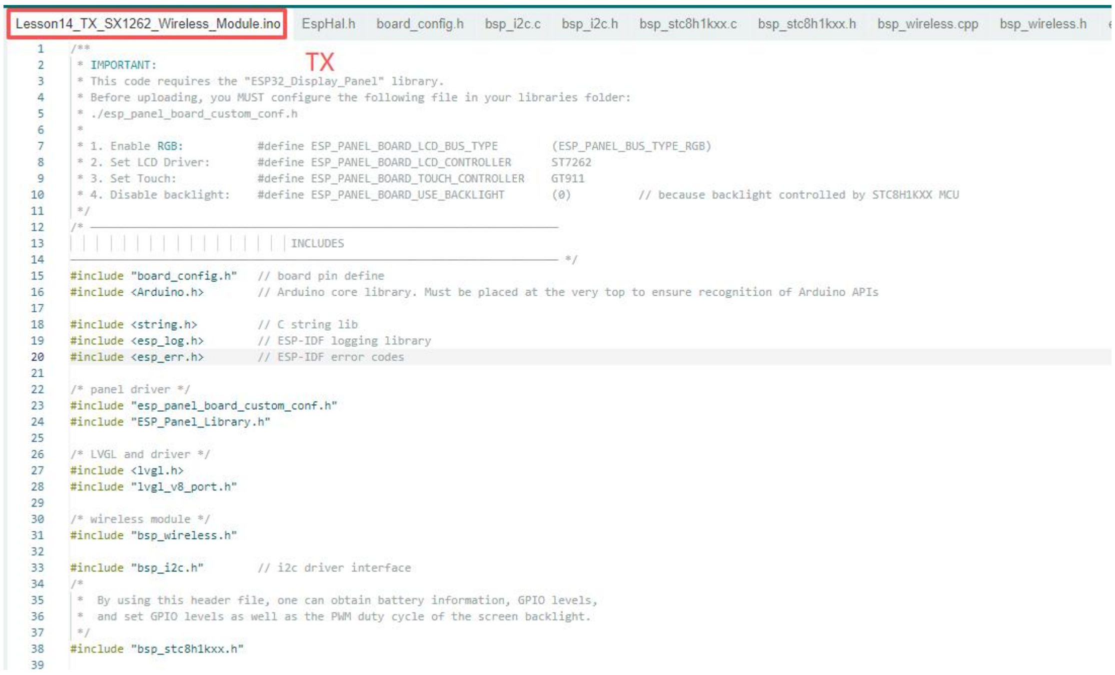
```c
#define CONFIG_BSP_SX1262_ENABLED           1
#define CONFIG_BSP_NRF2401_ENABLED          0
#define CONFIG_BSP_UART_TRANSPOND_ENABLED   0
//---------------------------------------------------------------------------
#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY     LORA_DIO2
#define SX1262_GPIO_IRQ      LORA_DIO1
#define SX1262_GPIO_NRST     LORA_RESET
#define SX1262_GPIO_NSS      LORA_CS

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t sx1262_tx_init();
    void sx1262_tx_deinit();
    bool send_lora_pack_radio();

    uint32_t sx1262_get_tx_counter();

    esp_err_t sx1262_rx_init();
    void sx1262_rx_deinit();
    void received_lora_pack_radio(size_t len);
    void sx1262_set_rx_callback(void (*callback)(const char* data, size_t len, float rssi, float snr));
    size_t sx1262_get_received_len(void);
    bool sx1262_is_data_received(void);
#ifdef __cplusplus
}
#endif

#endif
//---------------------------------------------------------------------------

#ifdef CONFIG_BSP_NRF2401_ENABLED

#define NRF24_GPIO_IRQ       NRF24_IRQ
#define NRF24_GPIO_CE        NRF24_CE
#define NRF24_GPIO_CS        NRF24_CS

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t nrf24_tx_init();
```

As shown in the figure, we have enabled the SX1262-related configuration, so the other wireless modules are currently disabled and not in use.

Within the bsp_wireless component, you only need to know when to call the provided interfaces that we have written.

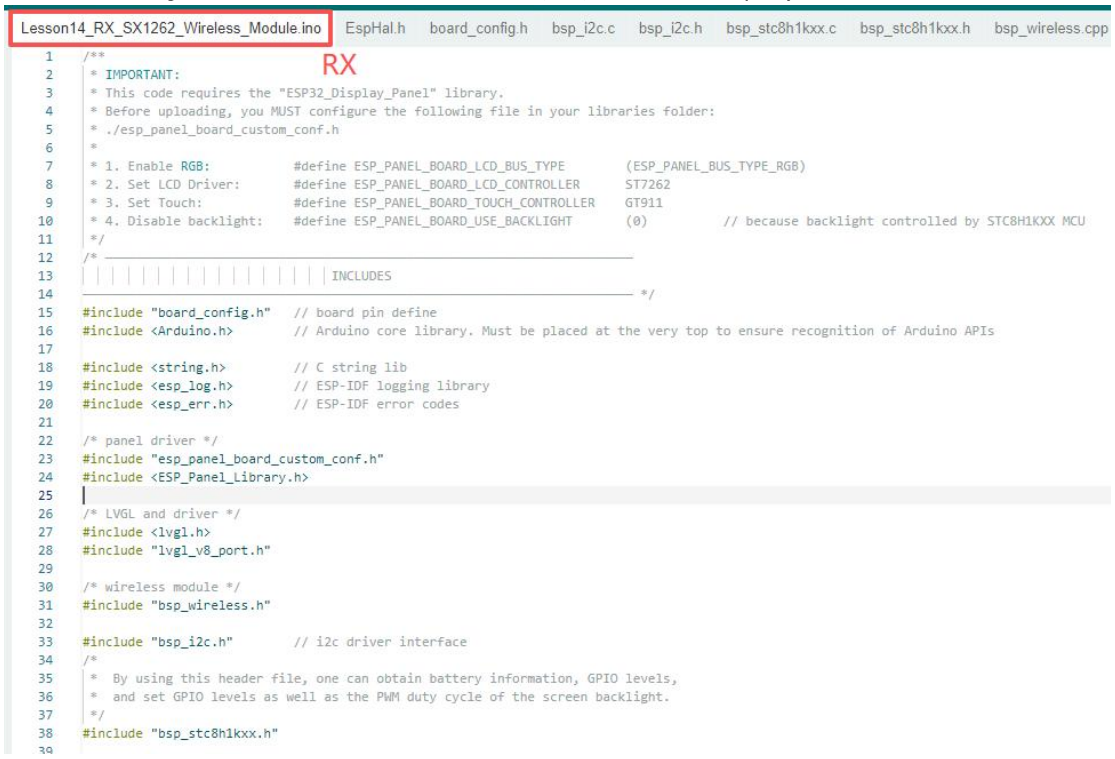
Next, let's focus on understanding the bsp_wireless component itself.

The following section shows the transmitter (TX) side of the project:



```
/**
 * IMPORTANT:
 * This code requires the "ESP32_Display_Panel" library.
 * Before uploading, you MUST configure the following file in your libraries folder:
 * ./esp_panel_board_custom_conf.h
 *
 * 1. Enable RGB:          #define ESP_PANEL_BOARD_LCD_BUS_TYPE       (ESP_PANEL_BUS_TYPE_RGB)
 * 2. Set LCD Driver:      #define ESP_PANEL_BOARD_LCD_CONTROLLER     ST7262
 * 3. Set Touch:           #define ESP_PANEL_BOARD_TOUCH_CONTROLLER   GT911
 * 4. Disable backlight:   #define ESP_PANEL_BOARD_USE_BACKLIGHT      (0)        // because backlight controlled by STC8H1KXX MCU
 */
/* ---------------------------------------------------------------------------
 | | | | | | | | | | | | | | | | | INCLUDES
 --------------------------------------------------------------------------- */
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

/* panel driver */
#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

/* LVGL and driver */
#include <lvgl.h>
#include "lvgl_v8_port.h"

/* wireless module */
#include "bsp_wireless.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"
```

The following section shows the receiver (RX) side of the project:

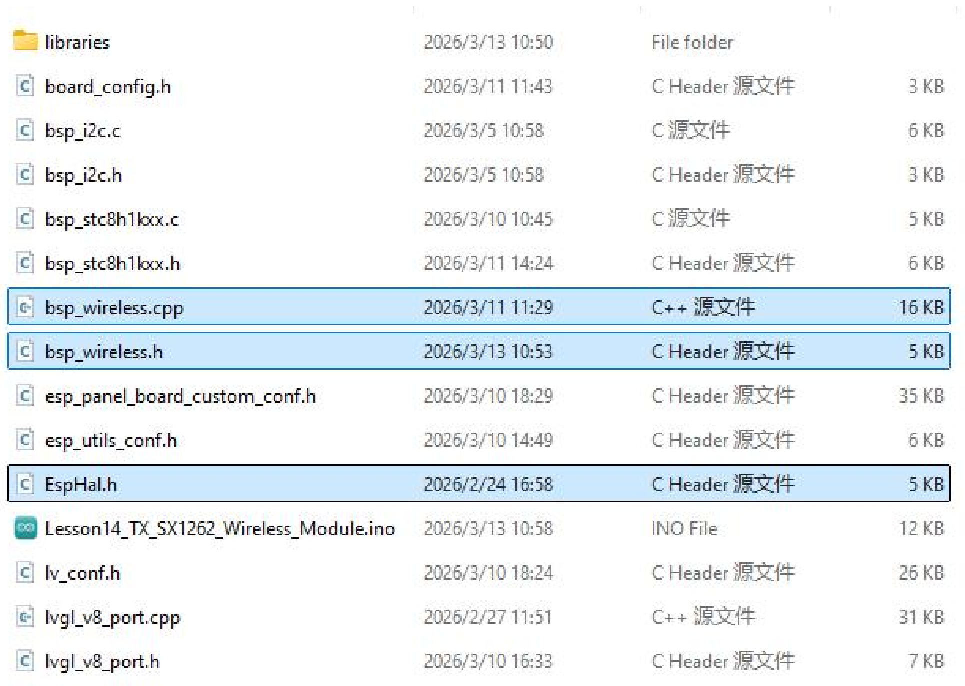


```
/**
 * IMPORTANT:
 * This code requires the "ESP32_Display_Panel" library.
 * Before uploading, you MUST configure the following file in your libraries folder:
 * ./esp_panel_board_custom_conf.h
 *
 * 1. Enable RGB:          #define ESP_PANEL_BOARD_LCD_BUS_TYPE       (ESP_PANEL_BUS_TYPE_RGB)
 * 2. Set LCD Driver:      #define ESP_PANEL_BOARD_LCD_CONTROLLER     ST7262
 * 3. Set Touch:           #define ESP_PANEL_BOARD_TOUCH_CONTROLLER   GT911
 * 4. Disable backlight:   #define ESP_PANEL_BOARD_USE_BACKLIGHT      (0)        // because backlight controlled by STC8H1KXX MCU
 */
/* ---------------------------------------------------------------------------
 | | | | | | | | | | | | | | | | | INCLUDES
 --------------------------------------------------------------------------- */
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

/* panel driver */
#include "esp_panel_board_custom_conf.h"
#include <ESP_Panel_Library.h>

/* LVGL and driver */
#include <lvgl.h>
#include "lvgl_v8_port.h"

/* wireless module */
#include "bsp_wireless.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"
```

In these two projects, the only difference lies in the main functions: Lesson14_TX_SX1262_Wireless_Module.ino for the transmitter and Lesson14_RX_SX1262_Wireless_Module.ino for the receiver.

All other code files are identical. (For convenience, we have prepared both main functions for you to use separately.)

You can see in this folder there are "bsp_wireless.h", "bsp_wireless.cpp", and "EspHal.h".

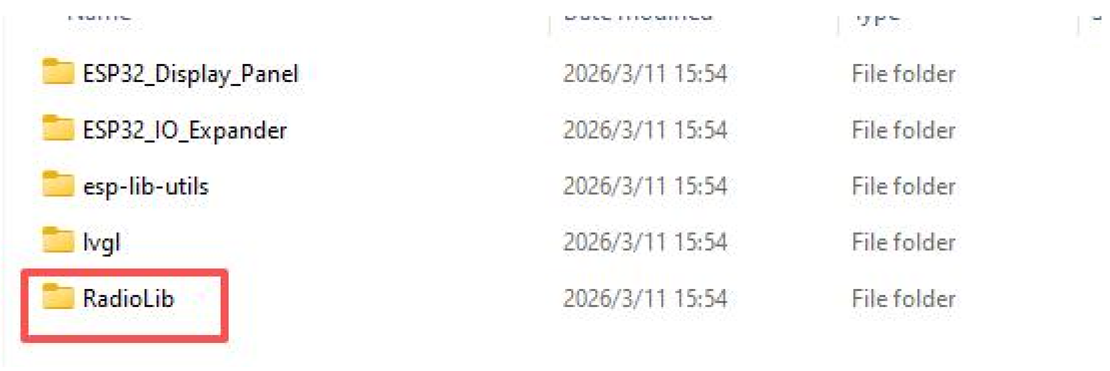


libraries	2026/3/13 10:50	File folder	
board_config.h	2026/3/11 11:43	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/11 14:24	C Header 源文件	6 KB
bsp_wireless.cpp	2026/3/11 11:29	C++ 源文件	16 KB
bsp_wireless.h	2026/3/13 10:53	C Header 源文件	5 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
EspHal.h	2026/2/24 16:58	C Header 源文件	5 KB
Lesson14_TX_SX1262_Wireless_Module.ino	2026/3/13 10:58	INO File	12 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

The purpose of EspHal.h is to convert C code from ESP-IDF into the Arduino-style C++ code required by the RadioLib component library.

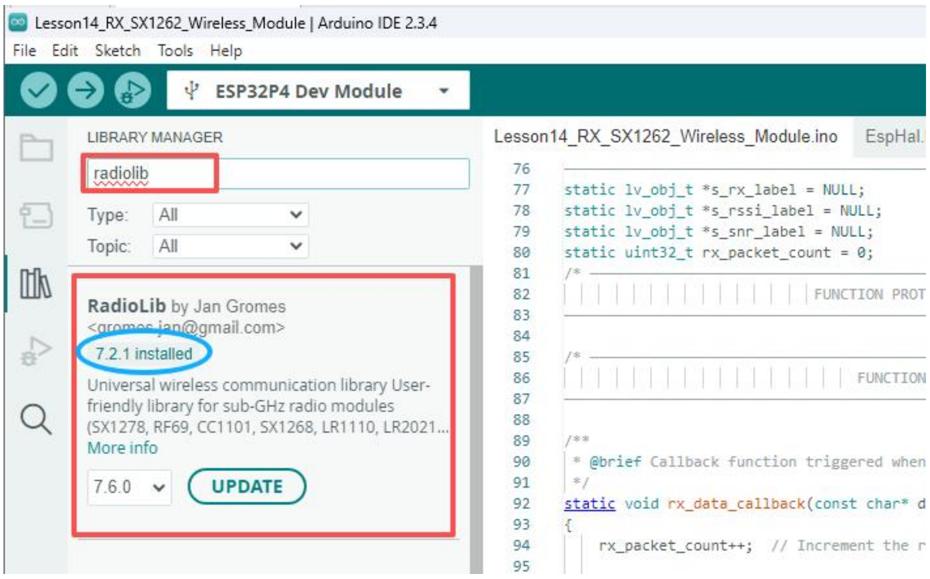
And RadioLib is already prepared in the libraries folder.

Since this is an official library, we need to rely on it to implement the SX1262 LoRa wireless transmission or reception functionality on our Advance-P4.



Name	Date modified	Type
ESP32_Display_Panel	2026/3/11 15:54	File folder
ESP32_IO_Expander	2026/3/11 15:54	File folder
esp-lib-utils	2026/3/11 15:54	File folder
lvgl	2026/3/11 15:54	File folder
RadioLib	2026/3/11 15:54	File folder

You can also download version 7.2.1 of the RadioLib library in the Arduino IDE.



Lesson14_RX_SX1262_Wireless_Module | Arduino IDE 2.3.4

LIBRARY MANAGER: radiolib — Type: All — Topic: All

RadioLib by Jan Gromes <gromes.jan@gmail.com>
7.2.1 installed
Universal wireless communication library User-friendly library for sub-GHz radio modules (SX1278, RF69, CC1101, SX1268, LR1110, LR2021...
More info
7.6.0 UPDATE

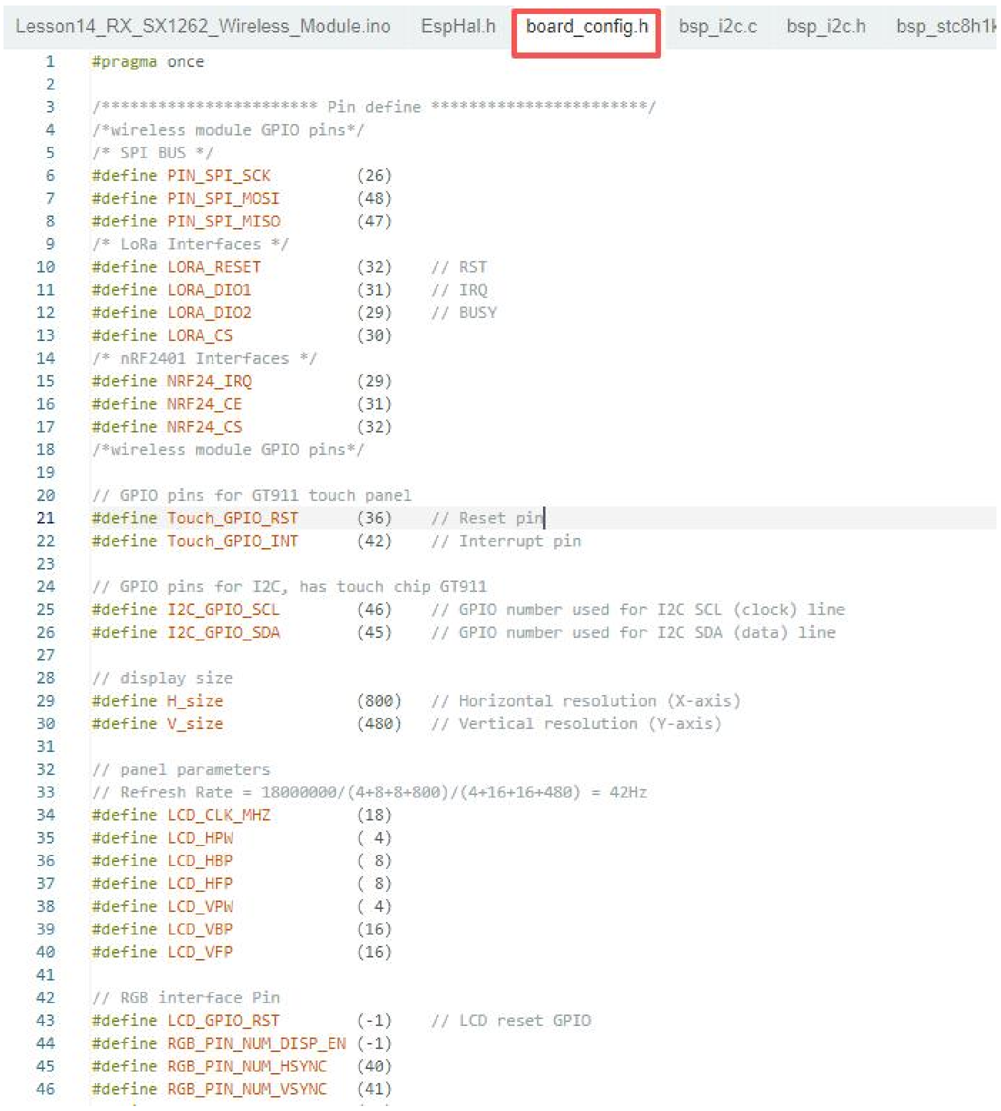
```
static lv_obj_t *s_rx_label = NULL;
static lv_obj_t *s_rssi_label = NULL;
static lv_obj_t *s_snr_label = NULL;
static uint32_t rx_packet_count = 0;
/*
                                          FUNCTION PROT

/*
                                          FUNCTION

/**
 * @brief Callback function triggered when
 */
static void rx_data_callback(const char* d
{
    rx_packet_count++;  // Increment the r
```

Then the other code files are the screen display-related driver codes that were elaborately explained in Lesson 7.

Look at the board_config.h file again.

This code is a header file for the development board hardware configuration. Its main function is to uniformly define all the GPIO pins and display parameters for the wireless module, touch screen, I²C bus, and RGB display screen.

This enables the program to directly call these macro definitions when initializing each peripheral, eliminating the need to repeatedly write pin numbers in multiple files, thereby improving the readability and maintainability of the code.



```
#pragma once

/*********************** Pin define **********************/
/*wireless module GPIO pins*/
/* SPI BUS */
#define PIN_SPI_SCK         (26)
#define PIN_SPI_MOSI        (48)
#define PIN_SPI_MISO        (47)
/* LoRa Interfaces */
#define LORA_RESET          (32)    // RST
#define LORA_DIO1           (31)    // IRQ
#define LORA_DIO2           (29)    // BUSY
#define LORA_CS             (30)
/* nRF2401 Interfaces */
#define NRF24_IRQ           (29)
#define NRF24_CE            (31)
#define NRF24_CS            (32)
/*wireless module GPIO pins*/

// GPIO pins for GT911 touch panel
#define Touch_GPIO_RST      (36)    // Reset pin
#define Touch_GPIO_INT      (42)    // Interrupt pin

// GPIO pins for I2C, has touch chip GT911
#define I2C_GPIO_SCL        (46)    // GPIO number used for I2C SCL (clock) line
#define I2C_GPIO_SDA        (45)    // GPIO number used for I2C SDA (data) line

// display size
#define H_size              (800)   // Horizontal resolution (X-axis)
#define V_size              (480)   // Vertical resolution (Y-axis)

// panel parameters
// Refresh Rate = 18000000/(4+8+8+800)/(4+16+16+480) = 42Hz
#define LCD_CLK_MHZ         (18)
#define LCD_HPW             ( 4)
#define LCD_HBP             ( 8)
#define LCD_HFP             ( 8)
#define LCD_VPW             ( 4)
#define LCD_VBP             (16)
#define LCD_VFP             (16)

// RGB interface Pin
#define LCD_GPIO_RST        (-1)    // LCD reset GPIO
#define RGB_PIN_NUM_DISP_EN (-1)
#define RGB_PIN_NUM_HSYNC   (40)
#define RGB_PIN_NUM_VSYNC   (41)
```

The #pragma once at the beginning of the file is used to prevent the header file from being included multiple times. Firstly, the code defines a set of SPI bus pins: PIN_SPI_SCK, PIN_SPI_MOSI, PIN_SPI_MISO, which correspond to the SPI clock line, the host output and slave input line, and the host input and slave output line respectively. These pins are used to connect the wireless communication module.

Subsequently, interfaces were reserved for the two wireless modules. One group was for the control pins of the SX1262 LoRa module, including the reset pin LORA_RESET, the interrupt pin LORA_DIO1, the busy status pin LORA_DIO2, and the chip select pin LORA_CS.

The other group consists of the control pins of the nRF24L01 module, such as the interrupt NRF24_IRQ, the enable NRF24_CE, and the chip select NRF24_CS. These modules communicate with the main controller via SPI.
The following code defines two control pins of the capacitive touch screen controller GT911: Touch_GPIO_RST is used to reset the touch chip, and Touch_GPIO_INT is used for the touch interrupt signal;

At the same time, the I²C bus pins used by the touch chip, namely I2C_GPIO_SCL and I2C_GPIO_SDA, were also defined.

Subsequently, the code provided the resolution parameters for the LCD display screen: H_size is 800 pixels for horizontal resolution, and V_size is 480 pixels for vertical resolution.

Subsequently, a set of LCD timing parameters was defined, including the pixel clock LCD_CLK_MHZ (18 MHz), the horizontal synchronization pulse width LCD_HPW, the horizontal front and back blanking LCD_HFP/HBP, the vertical synchronization pulse width LCD_VPW, and the vertical front and back blanking LCD_VFP/VBP. These parameters determine the refresh timing of the RGB screen, and in the comments, it is given that the calculated refresh rate based on these parameters is approximately 42 Hz.

The final code defines the pin mapping of the RGB parallel interface, including the synchronization signals HSYNC, VSYNC, data enable DE, pixel clock PCLK, and 16 RGB data lines DATA0 to DATA15. These data lines are responsible for transmitting color data to the LCD during each pixel clock cycle.

OK, now that the overall framework of the project has been clearly understood, let's take a closer look at this bsp_wireless component, namely bsp_wireless.h and bsp_wireless.cpp.

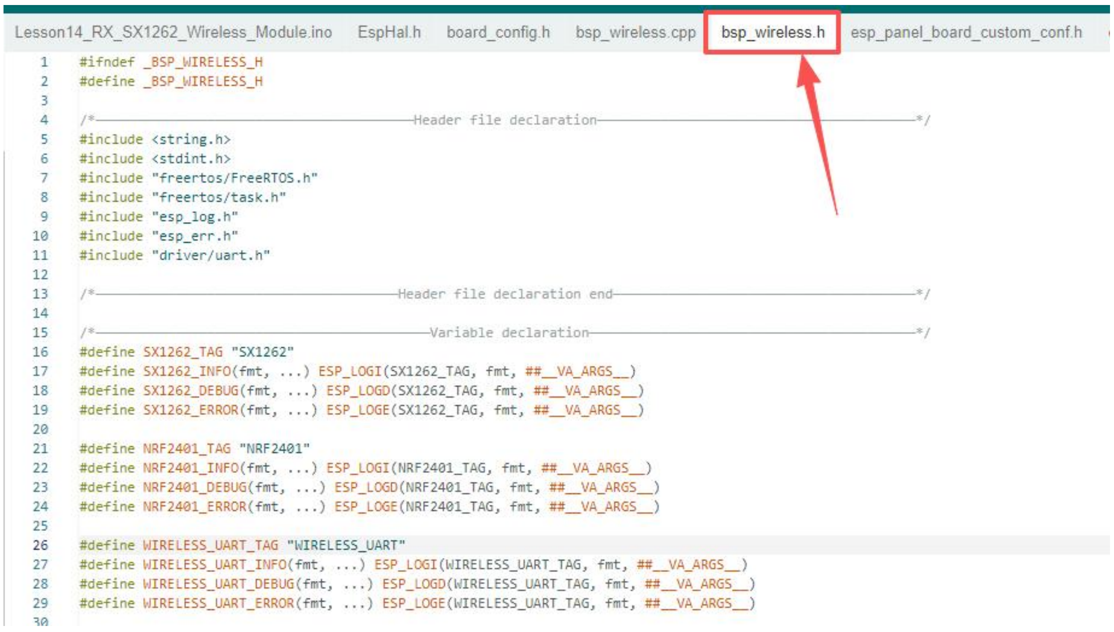
SX1262 LoRa Code

The SX1262 LoRa transmission and reception code consists of two files: bsp_wireless.cpp and bsp_wireless.h.

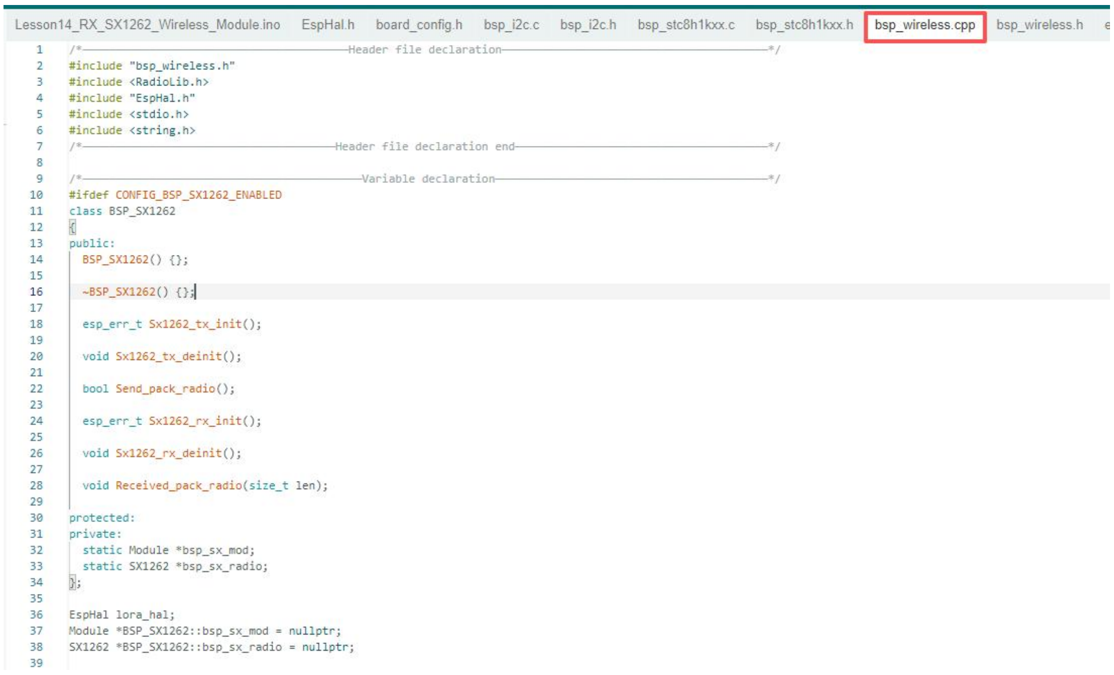
Next, we will first analyze the SX1262-related code in bsp_wireless.h.

bsp_wireless.h is the header file for the SX1262 LoRa wireless module.
Its main purposes are:
To declare the functions, macros, and variables implemented in bsp_wireless.cpp for external use.

To allow other .c files to simply #include "bsp_wireless.h" in order to call this module. In other words, it serves as the interface layer, exposing which functions and constants can be used externally while hiding the internal details of the module. Any libraries required for this component are included in both bsp_wireless.h and bsp_wireless.cpp.


Lesson14_RX_SX1262_Wireless_Module.ino
EspHal.h
board_config.h
bsp_wireless.cpp
bsp_wireless.h
esp_panel_board_custom_conf.h

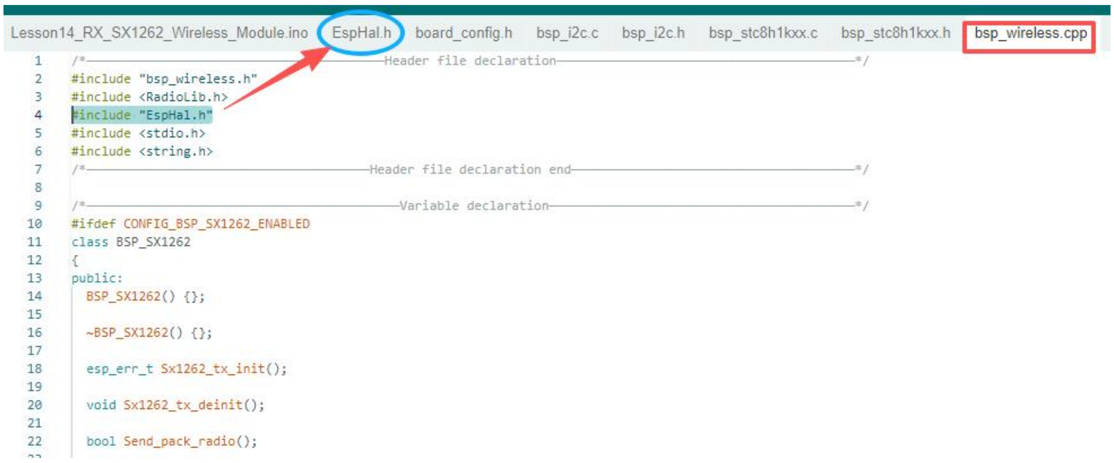
```
#ifndef _BSP_WIRELESS_H
#define _BSP_WIRELESS_H

/*——————————————————————————Header file declaration——————————————————————————*/
#include <string.h>
#include <stdint.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/uart.h"

/*——————————————————————————Header file declaration end——————————————————————————*/

/*——————————————————————————Variable declaration——————————————————————————*/
#define SX1262_TAG "SX1262"
#define SX1262_INFO(fmt, ...) ESP_LOGI(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_DEBUG(fmt, ...) ESP_LOGD(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_ERROR(fmt, ...) ESP_LOGE(SX1262_TAG, fmt, ##__VA_ARGS__)

#define NRF2401_TAG "NRF2401"
#define NRF2401_INFO(fmt, ...) ESP_LOGI(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_DEBUG(fmt, ...) ESP_LOGD(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_ERROR(fmt, ...) ESP_LOGE(NRF2401_TAG, fmt, ##__VA_ARGS__)

#define WIRELESS_UART_TAG "WIRELESS_UART"
#define WIRELESS_UART_INFO(fmt, ...) ESP_LOGI(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_DEBUG(fmt, ...) ESP_LOGD(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_ERROR(fmt, ...) ESP_LOGE(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)

```



```
/*——————————————————————————Header file declaration——————————————————————————*/
#include "bsp_wireless.h"
#include <RadioLib.h>
#include "EspHal.h"
#include <stdio.h>
#include <string.h>
/*——————————————————————————Header file declaration end——————————————————————————*/

/*——————————————————————————Variable declaration——————————————————————————*/
#ifdef CONFIG_BSP_SX1262_ENABLED
class BSP_SX1262
{
public:
  BSP_SX1262() {};

  ~BSP_SX1262() {};

  esp_err_t Sx1262_tx_init();

  void Sx1262_tx_deinit();

  bool Send_pack_radio();

  esp_err_t Sx1262_rx_init();

  void Sx1262_rx_deinit();

  void Received_pack_radio(size_t len);

protected:
private:
  static Module *bsp_sx_mod;
  static SX1262 *bsp_sx_radio;
};

EspHal lora_hal;
Module *BSP_SX1262::bsp_sx_mod = nullptr;
SX1262 *BSP_SX1262::bsp_sx_radio = nullptr;

```

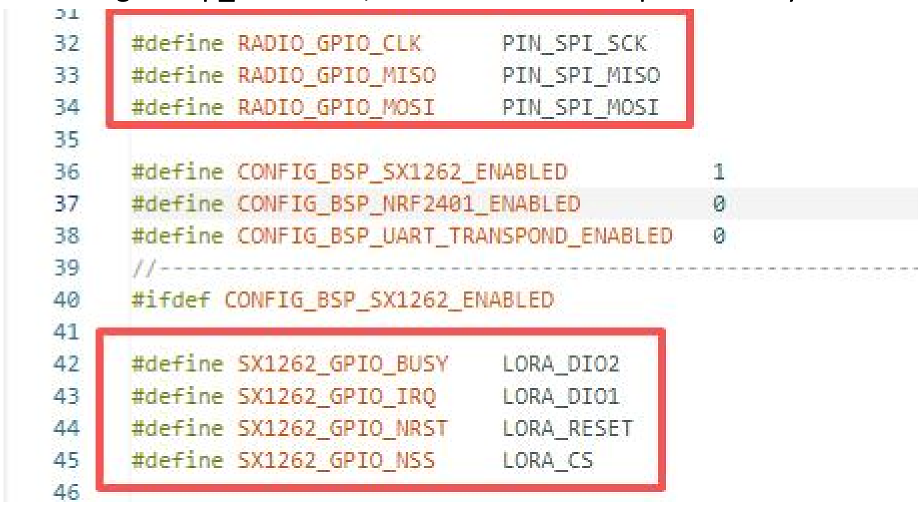
Since the function implementation in bsp_wireless.cpp uses the function encapsulation from EspHal.h, the reference to the header file needs to be placed in the .cpp file.

Take #include <RadioLib.h> as an example; this is a library under the network component.



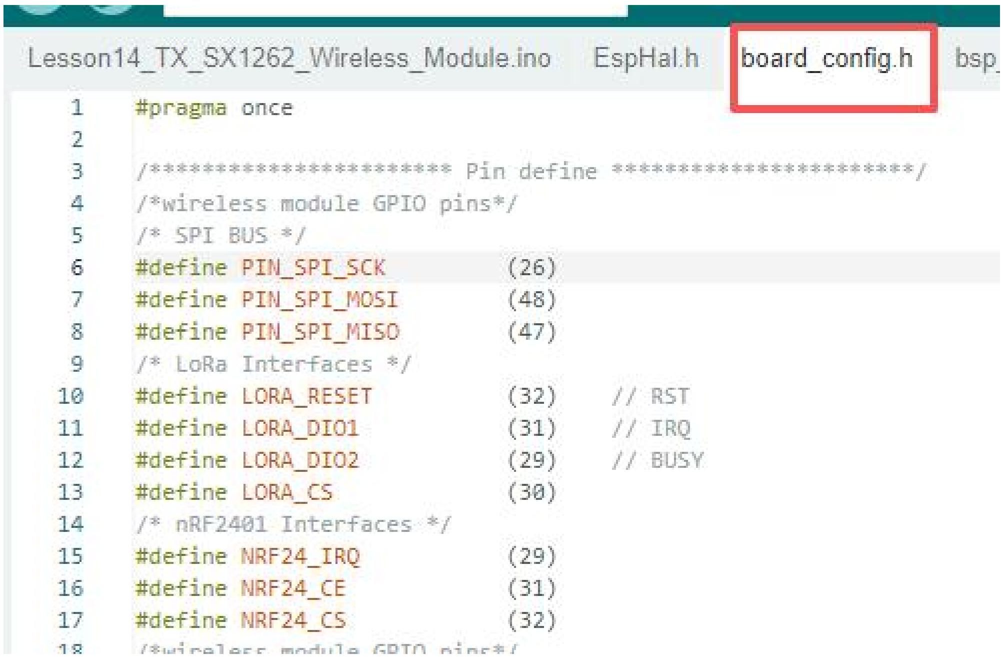
```
/*———————————————————————Header file declaration———————————————————————*/
#include "bsp_wireless.h"
#include <RadioLib.h>
#include "EspHal.h"
#include <stdio.h>
#include <string.h>
/*———————————————————————Header file declaration end———————————————————————*/

/*———————————————————————Variable declaration———————————————————————*/
#ifdef CONFIG_BSP_SX1262_ENABLED
class BSP_SX1262
{
public:
  BSP_SX1262() {};

  ~BSP_SX1262() {};

  esp_err_t Sx1262_tx_init();

  void Sx1262_tx_deinit();

  bool Send_pack_radio();
```

Returning to bsp_wireless.h,here we declare the pins used by the wireless module.



```
#define RADIO_GPIO_CLK        PIN_SPI_SCK
#define RADIO_GPIO_MISO       PIN_SPI_MISO
#define RADIO_GPIO_MOSI       PIN_SPI_MOSI

#define CONFIG_BSP_SX1262_ENABLED              1
#define CONFIG_BSP_NRF2401_ENABLED             0
#define CONFIG_BSP_UART_TRANSPOND_ENABLED      0
//-------------------------------------------------------------
#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY      LORA_DIO2
#define SX1262_GPIO_IRQ       LORA_DIO1
#define SX1262_GPIO_NRST      LORA_RESET
#define SX1262_GPIO_NSS       LORA_CS

```

The specific pin numbers are defined in the file "board_config.h".

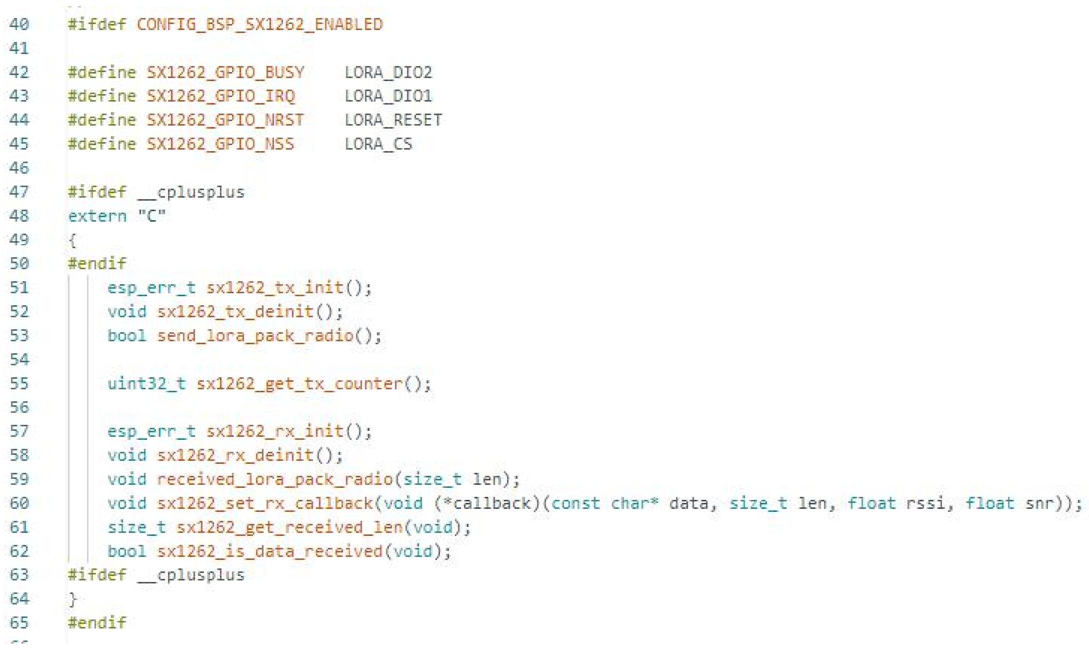


```
#pragma once

/*********************** Pin define ***********************/
/*wireless module GPIO pins*/
/* SPI BUS */
#define PIN_SPI_SCK         (26)
#define PIN_SPI_MOSI        (48)
#define PIN_SPI_MISO        (47)
/* LoRa Interfaces */
#define LORA_RESET          (32)    // RST
#define LORA_DIO1           (31)    // IRQ
#define LORA_DIO2           (29)    // BUSY
#define LORA_CS             (30)
/* nRF2401 Interfaces */
#define NRF24_IRQ           (29)
#define NRF24_CE            (31)
#define NRF24_CS            (32)
/*wireless module GPIO pins*/
```

The pin assignments should not be modified, otherwise the wireless module will not work due to incorrect connections.

Next, we declare the variables and functions that we will use. The actual implementation of these functions is in bsp_wireless.cpp.

By placing them all in bsp_wireless.h, it becomes easier to call and manage them. (We will explore their specific functionality when we look at bsp_wireless.cpp.)



```
#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY    LORA_DIO2
#define SX1262_GPIO_IRQ     LORA_DIO1
#define SX1262_GPIO_NRST    LORA_RESET
#define SX1262_GPIO_NSS     LORA_CS

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t sx1262_tx_init();
    void sx1262_tx_deinit();
    bool send_lora_pack_radio();

    uint32_t sx1262_get_tx_counter();

    esp_err_t sx1262_rx_init();
    void sx1262_rx_deinit();
    void received_lora_pack_radio(size_t len);
    void sx1262_set_rx_callback(void (*callback)(const char* data, size_t len, float rssi, float snr));
    size_t sx1262_get_received_len(void);
    bool sx1262_is_data_received(void);
#ifdef __cplusplus
}
#endif
```

Next, let's take a look at bsp_wireless.cpp to understand the specific function of each function.

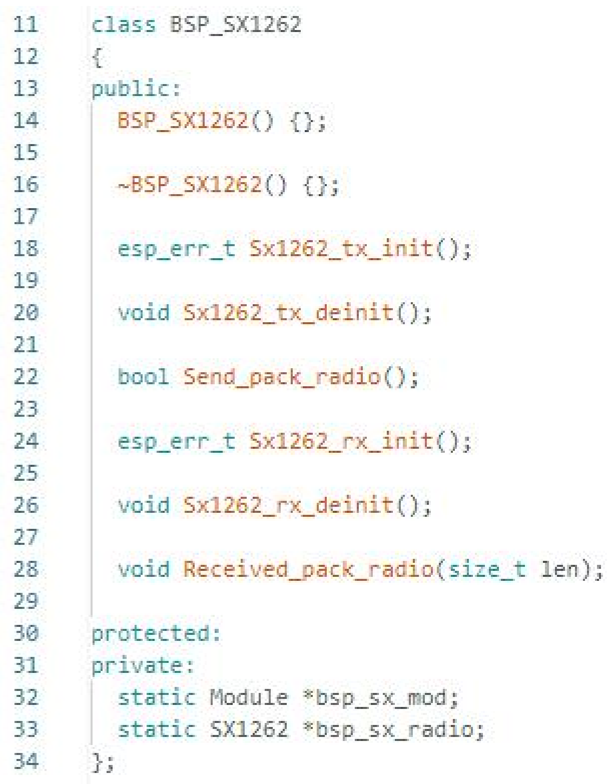
The bsp_wireless component implements LoRa data transmission and reception, communicates with the main controller via the SPI interface, and handles the sending and receiving at the wireless data link layer.

Here, we won't go into the detailed code. Instead, we will explain the purpose of each function and when to call them.

BSP_SX1262 Class：

This indicates that:

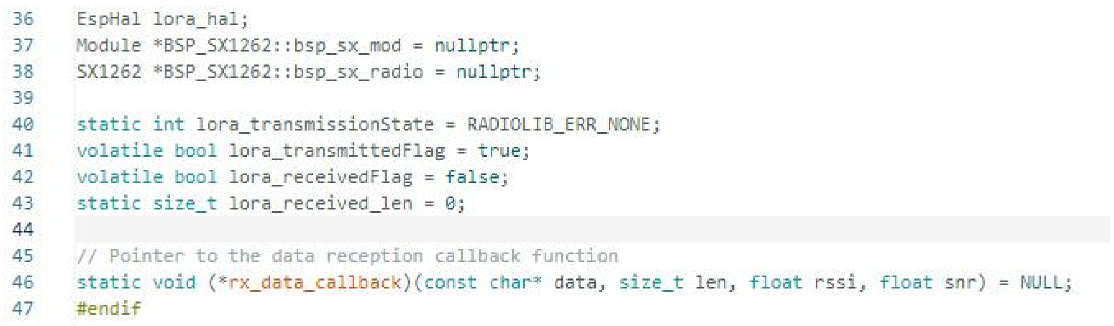
- It is a C++ wrapper class for operating the SX1262 module.
- It mainly provides functions for initialization, de-initialization, and data transmission/reception.
- All hardware operations are performed based on the RadioLib library.
- bsp_sx_mod and bsp_sx_radio are object pointers in memory for the SX1262 module (statically shared).



```
class BSP_SX1262
{
public:
  BSP_SX1262() {};

  ~BSP_SX1262() {};

  esp_err_t Sx1262_tx_init();

  void Sx1262_tx_deinit();

  bool Send_pack_radio();

  esp_err_t Sx1262_rx_init();

  void Sx1262_rx_deinit();

  void Received_pack_radio(size_t len);

protected:
private:
  static Module *bsp_sx_mod;
  static SX1262 *bsp_sx_radio;
};
```

Defines the **core global variables required by the SX1262 LoRa module driver**, used to manage the module instance, status, and data callbacks:

- lora_hal is the low-level SPI hardware abstraction layer object, responsible for SPI communication.
- bsp_sx_mod and bsp_sx_radio point to the generic RadioLib module object and the SX1262 module object, respectively. They encapsulate the specific hardware pins and transmission/reception interfaces. These objects are created during module initialization (e.g., Sx1262_tx_init() or Sx1262_rx_init()) and released or set to standby during de-initialization.
- lora_transmissionState records the status code of the last transmission operation for debugging and error handling.
- lora_transmittedFlag is the transmission completion flag, set by the transmission interrupt callback set_sx1262_tx_flag(), indicating that the module is ready to send a new data packet.
- lora_receivedFlag is the reception completion flag, set by the reception interrupt callback set_sx1262_rx_flag(), indicating that new data is available to read.
- lora_received_len stores the length of the most recently received data.
- rx_data_callback is a function pointer that allows the upper layer to register a callback. When the SX1262 receives data, this callback is automatically triggered, passing the received data, its length, RSSI, and SNR information to the upper-level processing.



```
EspHal lora_hal;
Module *BSP_SX1262::bsp_sx_mod = nullptr;
SX1262 *BSP_SX1262::bsp_sx_radio = nullptr;

static int lora_transmissionState = RADIOLIB_ERR_NONE;
volatile bool lora_transmittedFlag = true;
volatile bool lora_receivedFlag = false;
static size_t lora_received_len = 0;

// Pointer to the data reception callback function
static void (*rx_data_callback)(const char* data, size_t len, float rssi, float snr) = NULL;
#endif
```

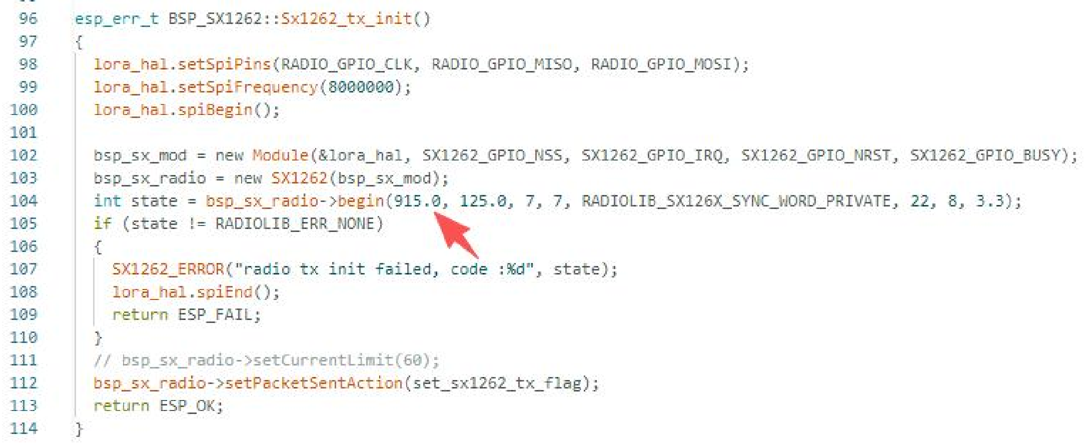
Sx1262_tx_init():

The function Sx1262_tx_init() in the BSP_SX1262 class is used to initialize the SX1262 module for data transmission.

- The function first uses lora_hal to configure the SPI pins (RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI) and the SPI clock frequency (8 MHz), then calls spiBegin() to start SPI communication, providing the module with a low-level communication interface.
- Next, it creates a Module object bsp_sx_mod to encapsulate the SX1262 hardware pins (NSS, IRQ, NRST, BUSY) and uses this module object to create the SX1262 instance bsp_sx_radio. By calling begin(), it configures the LoRa parameters (915 MHz frequency, 125 kHz bandwidth, spreading factor 7, coding rate 4/7, sync word, 22 dBm power, pre-gain 8, LNA 3.3, etc.), completing the module initialization.
- Finally, it calls setPacketSentAction(set_sx1262_tx_flag) to register the transmission completion callback, which sets lora_transmittedFlag whenever a data packet is sent, indicating that the module is ready to send the next packet.

This function is usually called at system startup or before starting LoRa data transmission. It only needs to be initialized once to ensure the module is in a transmittable state, after which data packets can be sent periodically using Send_pack_radio().

If two LoRa modules are used for transmission and reception, they must operate on the same frequency band.



```
esp_err_t BSP_SX1262::Sx1262_tx_init()
{
    lora_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI);
    lora_hal.setSpiFrequency(8000000);
    lora_hal.spiBegin();

    bsp_sx_mod = new Module(&lora_hal, SX1262_GPIO_NSS, SX1262_GPIO_IRQ, SX1262_GPIO_NRST, SX1262_GPIO_BUSY);
    bsp_sx_radio = new SX1262(bsp_sx_mod);
    int state = bsp_sx_radio->begin(915.0, 125.0, 7, 7, RADIOLIB_SX126X_SYNC_WORD_PRIVATE, 22, 8, 3.3);
    if (state != RADIOLIB_ERR_NONE)
    {
        SX1262_ERROR("radio tx init failed, code :%d", state);
        lora_hal.spiEnd();
        return ESP_FAIL;
    }
    // bsp_sx_radio->setCurrentLimit(60);
    bsp_sx_radio->setPacketSentAction(set_sx1262_tx_flag);
    return ESP_OK;
}
```

In bsp_sx_radio->begin(), the 915.0 MHz represents the operating center frequency of the SX1262. This can be changed according to the LoRa frequency regulations of different regions:

- China commonly uses 433 MHz or 470–510 MHz
- Europe uses 868 MHz
- The United States and Australia use 915 MHz
- Japan uses 923 MHz

When changing the frequency, the transmitter and receiver must match, otherwise communication will fail. Additionally, ensure that the selected frequency falls within the legally allowed ISM band for that region.
Parameters such as bandwidth and spreading factor can generally remain unchanged, although some frequency bands may have officially recommended values.

Send_pack_radio:

The function Send_pack_radio() in the BSP_SX1262 class is the core function for sending LoRa data packets.

- It first checks the transmission completion flag lora_transmittedFlag. If it is true, it indicates that the previous packet has been sent and the module is ready to send new data.
- If so, the flag is reset to false to prevent duplicate transmissions. The function then checks lora_transmissionState to determine whether the previous transmission was successful and prints the corresponding log.
- Next, it calls bsp_sx_radio->finishTransmit() to complete any remaining operations from the previous transmission, ensuring the module is ready for use. The transmission counter sx1262_tx_counter is incremented, and a text message with the counter is formatted and stored in the static buffer text.
- The function then calculates the message length and calls bsp_sx_radio->startTransmit() to initiate the transmission of the new data packet. It also updates lora_transmissionState to record the status of this transmission. If the transmission fails to start, an error message is printed.
- Finally, the function returns true if the transmission event has been handled, or false if the module is not yet ready to send.

This function is usually called periodically in the main loop or task scheduler to poll and send LoRa data packets, and it must ensure that the previous transmission is complete before sending a new packet.

sx1262_get_tx_counter()

This is a C-style interface used to obtain the value of the SX1262 module's transmitted packet counter sx1262_tx_counter. The function simply returns the global static variable sx1262_tx_counter and does not modify any state. It is typically used in applications to query the number of packets sent, for example, for debugging, statistics, or displaying the transmission count. It can be called at any time and does not depend on the transmission or reception status.

sx1262_tx_init()

This is a C-style wrapper interface for initializing the SX1262 transmission functionality. Inside the function, a BSP_SX1262 object is created, and its method Sx1262_tx_init() is called to complete the LoRa module SPI configuration, module object creation, parameter initialization, and registration of the transmission completion callback. The function returns ESP_OK if initialization is successful, or ESP_FAIL if it fails. This function is typically called once at system startup or before starting data transmission to ensure that the module is in a ready-to-transmit state.

sx1262_tx_deinit()

This is a C-style de-initialization interface for the SX1262 transmission function. Inside the function, a BSP_SX1262 object is created, and its method Sx1262_tx_deinit() is called to shut down the transmission functionality. During de-initialization, it calls finishTransmit() to complete any ongoing transmission, clears the transmission callback, switches the module to standby mode, and closes the SPI interface. This function is generally called when the system is shutting down, the module no longer needs to send data, or it enters low-power mode, releasing resources and ensuring the module safely stops.

send_lora_pack_radio()

This is a C-style interface used to trigger the SX1262 to send a data packet. Inside the function, a BSP_SX1262 object is created, and its method Send_pack_radio() is called. It polls the transmission completion flag lora_transmittedFlag and, when ready, generates a data packet and starts transmission. The function returns true if the transmission event has been handled, or false if the module is not yet ready. It is usually called periodically in the main loop or task scheduler to achieve continuous or scheduled data transmission.

set_sx1262_rx_flag()

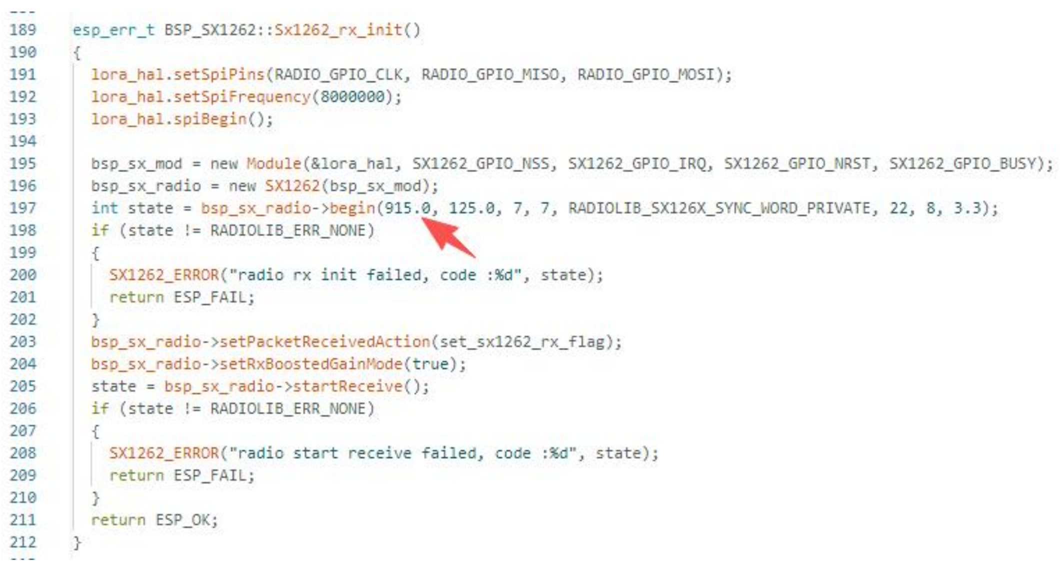
This is a static internal function used as the callback for SX1262 reception completion. Inside the function, it sets the global reception flag lora_receivedFlag to true, notifying the system that a new data packet has been received.
It is not called directly. Instead, it is registered by calling bsp_sx_radio->setPacketReceivedAction(set_sx1262_rx_flag), and the SX1262 hardware automatically triggers it each time a reception is completed, driving the data reception processing logic.

Sx1262_rx_init()

The function **Sx1262_rx_init()** in the **BSP_SX1262** class is used to initialize the SX1262 module for reception.

- The function first uses lora_hal to configure the SPI pins (RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI) and the SPI clock frequency (8 MHz), then calls spiBegin() to start SPI communication, providing a low-level interface for the SX1262.
- Next, it creates a Module object bsp_sx_mod and an SX1262 object bsp_sx_radio to encapsulate the hardware pins and transmission/reception interfaces. It then calls begin() to configure the LoRa parameters (915 MHz frequency, 125 kHz bandwidth, spreading factor 7, coding rate 4/7, sync word, 22 dBm power, etc.), completing module initialization. If initialization fails, an error is printed and the function returns a failure status.
- It then registers the reception completion callback via setPacketReceivedAction(set_sx1262_rx_flag), so that the module automatically sets lora_receivedFlag whenever a packet is received.
- The function calls setRxBoostedGainMode(true) to enable boosted gain mode for improved reception sensitivity, then calls startReceive() to start reception mode. If starting reception fails, it prints an error and returns failure.

This function is usually called once at system startup or before starting LoRa data reception to ensure the module is in a receivable state, after which received data can be processed via polling or callback.



```
esp_err_t BSP_SX1262::Sx1262_rx_init()
{
  lora_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI);
  lora_hal.setSpiFrequency(8000000);
  lora_hal.spiBegin();

  bsp_sx_mod = new Module(&lora_hal, SX1262_GPIO_NSS, SX1262_GPIO_IRQ, SX1262_GPIO_NRST, SX1262_GPIO_BUSY);
  bsp_sx_radio = new SX1262(bsp_sx_mod);
  int state = bsp_sx_radio->begin(915.0, 125.0, 7, 7, RADIOLIB_SX126X_SYNC_WORD_PRIVATE, 22, 8, 3.3);
  if (state != RADIOLIB_ERR_NONE)
  {
    SX1262_ERROR("radio rx init failed, code :%d", state);
    return ESP_FAIL;
  }
  bsp_sx_radio->setPacketReceivedAction(set_sx1262_rx_flag);
  bsp_sx_radio->setRxBoostedGainMode(true);
  state = bsp_sx_radio->startReceive();
  if (state != RADIOLIB_ERR_NONE)
  {
    SX1262_ERROR("radio start receive failed, code :%d", state);
    return ESP_FAIL;
  }
  return ESP_OK;
}
```

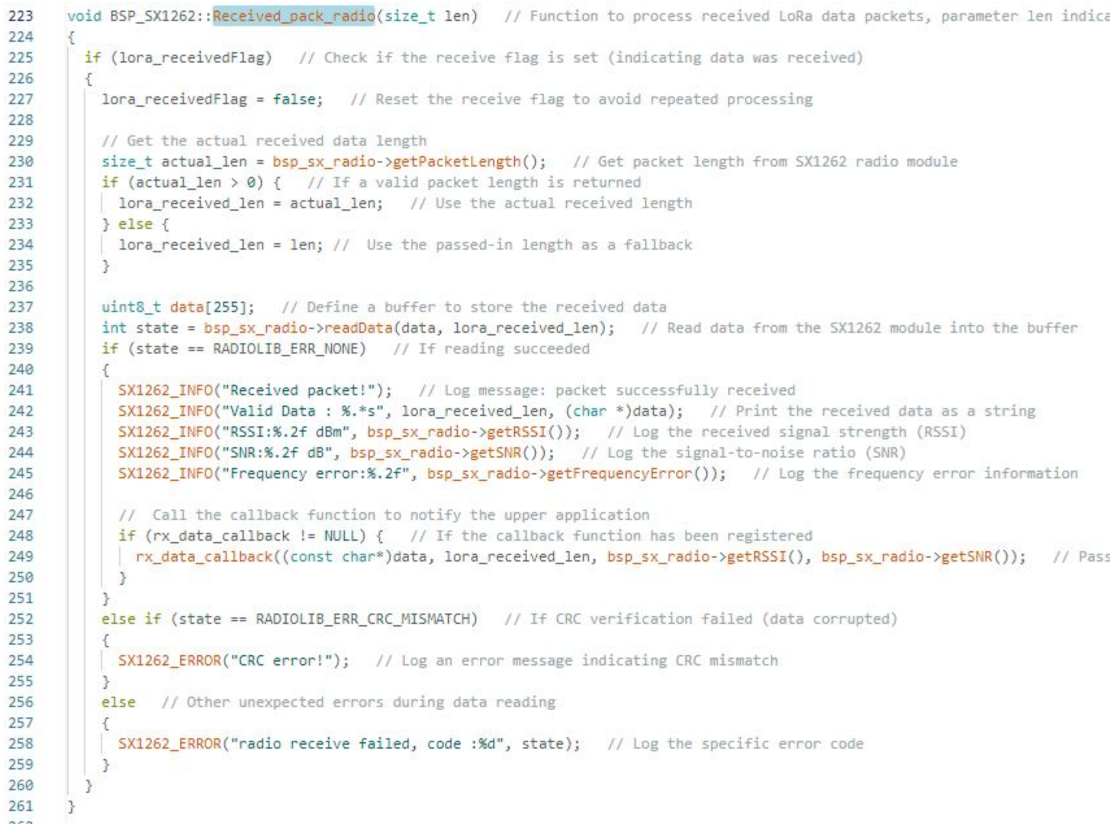
Here, we are initializing the **receiver module**. Similarly, by keeping the frequency band at **915 MHz**, the module can successfully receive the data sent from the transmitter.

Received_pack_radio:

The function Received_pack_radio(size_t len) in the BSP_SX1262 class is the core function for handling received LoRa data packets.

- The function first checks the reception flag lora_receivedFlag. If it is true, it indicates that a new data packet has arrived. The flag is then reset to false to prevent duplicate processing.
- It then obtains the actual length of the received data via bsp_sx_radio->getPacketLength(). If a valid length is returned, it is used; otherwise, the externally provided len serves as a fallback.
- Next, a buffer data[255] is defined, and bsp_sx_radio->readData() is called to read the received data into the buffer. If reading succeeds, the function prints the received data, RSSI (Received Signal Strength), SNR (Signal-to-Noise Ratio), and frequency offset. If a callback function rx_data_callback has been registered, it passes the data, length, and signal parameters to the upper-level application for processing.

This function is usually called periodically in the main loop or tasks. It executes after the SX1262 reception interrupt sets lora_receivedFlag, allowing the upper-level application to retrieve and process received packets promptly and reliably.



```
void BSP_SX1262::Received_pack_radio(size_t len)   // Function to process received LoRa data packets, parameter len indica
{
  if (lora_receivedFlag)   // Check if the receive flag is set (indicating data was received)
  {
    lora_receivedFlag = false;   // Reset the receive flag to avoid repeated processing

    // Get the actual received data length
    size_t actual_len = bsp_sx_radio->getPacketLength();   // Get packet length from SX1262 radio module
    if (actual_len > 0) {   // If a valid packet length is returned
      lora_received_len = actual_len;   // Use the actual received length
    } else {
      lora_received_len = len; //  Use the passed-in length as a fallback
    }

    uint8_t data[255];   // Define a buffer to store the received data
    int state = bsp_sx_radio->readData(data, lora_received_len);   // Read data from the SX1262 module into the buffer
    if (state == RADIOLIB_ERR_NONE)   // If reading succeeded
    {
      SX1262_INFO("Received packet!");   // Log message: packet successfully received
      SX1262_INFO("Valid Data : %.*s", lora_received_len, (char *)data);   // Print the received data as a string
      SX1262_INFO("RSSI:%.2f dBm", bsp_sx_radio->getRSSI());   // Log the received signal strength (RSSI)
      SX1262_INFO("SNR:%.2f dB", bsp_sx_radio->getSNR());   // Log the signal-to-noise ratio (SNR)
      SX1262_INFO("Frequency error:%.2f", bsp_sx_radio->getFrequencyError());   // Log the frequency error information

      //  Call the callback function to notify the upper application
      if (rx_data_callback != NULL) {   // If the callback function has been registered
        rx_data_callback((const char*)data, lora_received_len, bsp_sx_radio->getRSSI(), bsp_sx_radio->getSNR());   // Pass
      }
    }
    else if (state == RADIOLIB_ERR_CRC_MISMATCH)   // If CRC verification failed (data corrupted)
    {
      SX1262_ERROR("CRC error!");   // Log an error message indicating CRC mismatch
    }
    else   // Other unexpected errors during data reading
    {
      SX1262_ERROR("radio receive failed, code :%d", state);   // Log the specific error code
    }
  }
}
```

sx1262_rx_init()

This is a C-style interface used to initialize the SX1262 module's reception function. Inside the function, a BSP_SX1262 object is created, and its member function Sx1262_rx_init() is called to complete SPI configuration, module initialization, parameter setup, registration of the reception callback, and starting reception mode. The function returns ESP_OK if initialization succeeds, or ESP_FAIL if it fails. This function is typically called once at system startup or before starting LoRa data reception to ensure the module is in a ready-to-receive state.

sx1262_rx_deinit()

This is a C-style de-initialization interface for the SX1262 reception function. Inside the function, a BSP_SX1262 object is created, and its method Sx1262_rx_deinit() is called to shut down the reception functionality. The de-initialization process includes clearing the reception callback, switching the module to standby mode, delaying to ensure safe shutdown, and closing the SPI interface. This function is generally called when the system is shutting down, the module no longer needs to receive data, or it enters low-power mode.

received_lora_pack_radio(size_t len)

This is a C-style interface used to handle received LoRa data packets. Inside the function, a BSP_SX1262 object is created, and its method Received_pack_radio(len) is called. The function processes the data by checking the reception flag, reading the data, printing logs, and invoking the upper-layer callback function.
This function is generally called periodically in the main loop or tasks and executes after lora_receivedFlag is set, ensuring that the upper-level application can timely retrieve and handle received data packets.

sx1262_set_rx_callback(void (*callback)(const char* data, size_t len, float rssi, float snr))

This function is used to register the upper-layer callback rx_data_callback. When the SX1262 module receives a data packet, this callback is automatically triggered, passing the data, length, RSSI, and SNR information to the upper-layer application. This function is typically called once after initializing the reception functionality to bind the data processing logic.

sx1262_get_received_len()

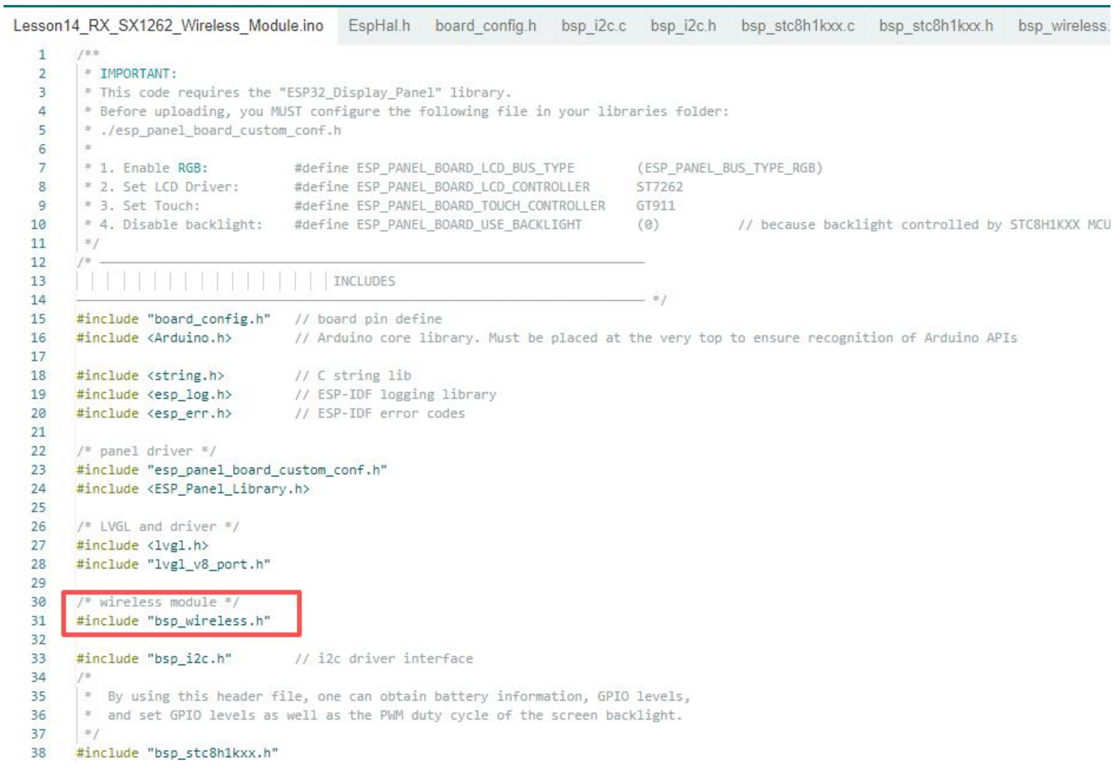
This is a query interface that returns the length of the most recently received data lora_received_len. Internally, the function simply returns the static variable without modifying any state. It is usually called when processing received data or performing debug/statistics, to obtain the actual length of the received packet.

sx1262_is_data_received()

This is a status query interface that returns the reception flag lora_receivedFlag, used to determine whether a new data packet has arrived. The function simply returns the status of the global variable without modifying it. It is typically polled in the main loop or tasks to decide whether to call received_lora_pack_radio() to process new data.

That concludes the introduction to the bsp_wireless component. You only need to know how to call these interfaces.

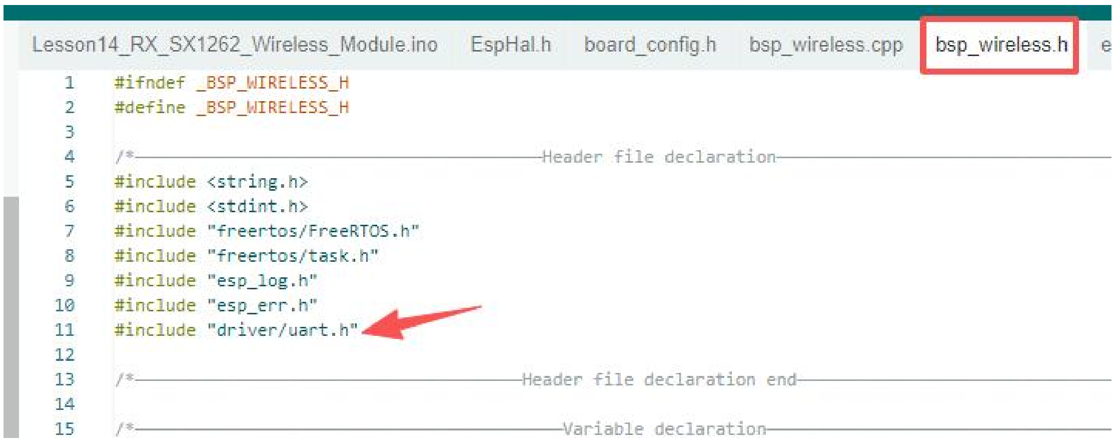
If you wish to use these functions, you simply need to include the library files in your functional code.



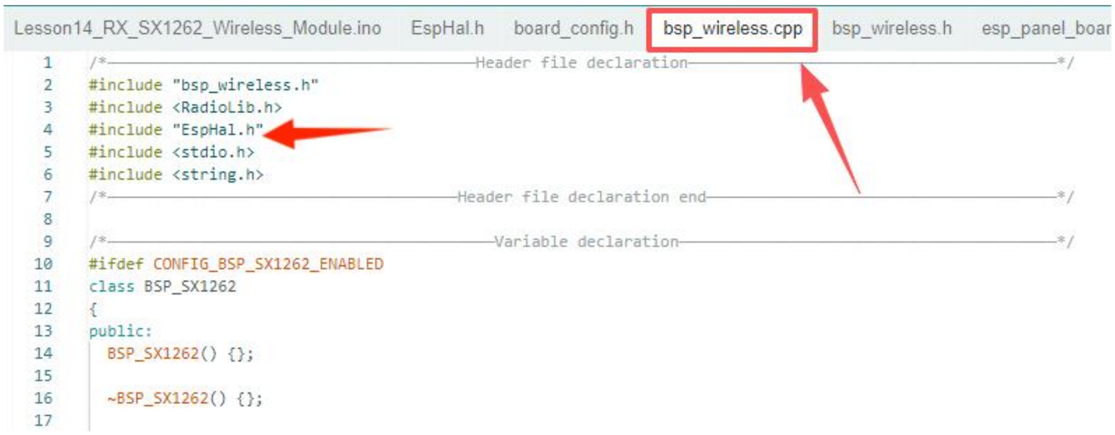
Lesson14_RX_SX1262_Wireless_Module.ino | EspHal.h | board_config.h | bsp_i2c.c | bsp_i2c.h | bsp_stc8h1kxx.c | bsp_stc8h1kxx.h | bsp_wireless.

```
/**
 * IMPORTANT:
 * This code requires the "ESP32_Display_Panel" library.
 * Before uploading, you MUST configure the following file in your libraries folder:
 * ./esp_panel_board_custom_conf.h
 *
 * 1. Enable RGB:         #define ESP_PANEL_BOARD_LCD_BUS_TYPE        (ESP_PANEL_BUS_TYPE_RGB)
 * 2. Set LCD Driver:     #define ESP_PANEL_BOARD_LCD_CONTROLLER      ST7262
 * 3. Set Touch:          #define ESP_PANEL_BOARD_TOUCH_CONTROLLER    GT911
 * 4. Disable backlight:  #define ESP_PANEL_BOARD_USE_BACKLIGHT       (0)         // because backlight controlled by STC8H1KXX MCU
 */
/* ───────────────────────────────────────────────────
| | | | | | | | | | | | | | | | INCLUDES
──────────────────────────────────────────────────── */
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

/* panel driver */
#include "esp_panel_board_custom_conf.h"
#include <ESP_Panel_Library.h>

/* LVGL and driver */
#include <lvgl.h>
#include "lvgl_v8_port.h"

/* wireless module */
#include "bsp_wireless.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
 *  By using this header file, one can obtain battery information, GPIO levels,
 *  and set GPIO levels as well as the PWM duty cycle of the screen backlight.
 */
#include "bsp_stc8h1kxx.h"
```

The reason driver, esp_timer, and RadioLib are included here is that we call them in bsp_wireless.h and bsp_wireless.cpp. Other libraries are system libraries and do not need to be explicitly added.

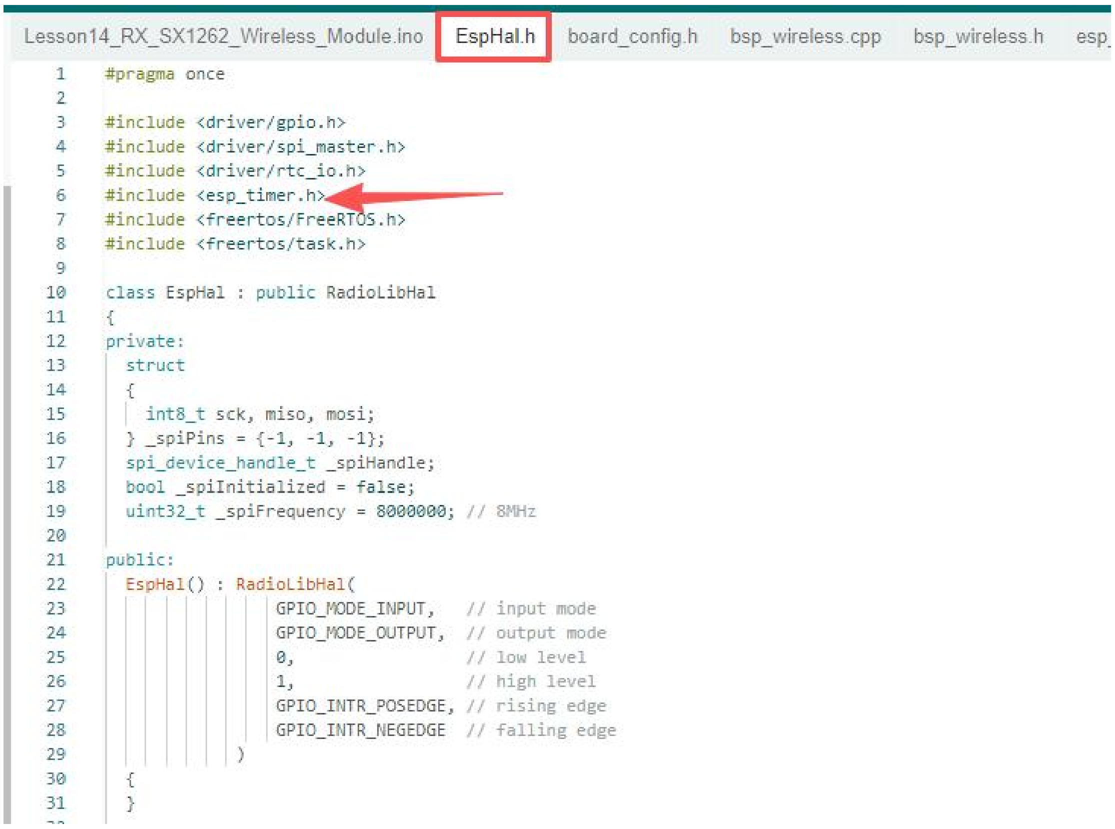


```
#ifndef _BSP_WIRELESS_H
#define _BSP_WIRELESS_H

/*————————————————————————Header file declaration————————————————————
#include <string.h>
#include <stdint.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/uart.h"

/*————————————————————————Header file declaration end————————————————
 
/*————————————————————————Variable declaration———————————————————————
```



```
/*————————————————————————Header file declaration————————————————————*/
#include "bsp_wireless.h"
#include <RadioLib.h>
#include "EspHal.h"
#include <stdio.h>
#include <string.h>
/*————————————————————————Header file declaration end————————————————*/

/*————————————————————————Variable declaration———————————————————————*/
#ifdef CONFIG_BSP_SX1262_ENABLED
class BSP_SX1262
{
public:
  BSP_SX1262() {};

  ~BSP_SX1262() {};

```

As well as the esp_timer used in the EspHal.h file.



```
#pragma once

#include <driver/gpio.h>
#include <driver/spi_master.h>
#include <driver/rtc_io.h>
#include <esp_timer.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>

class EspHal : public RadioLibHal
{
private:
  struct
  {
    int8_t sck, miso, mosi;
  } _spiPins = {-1, -1, -1};
  spi_device_handle_t _spiHandle;
  bool _spiInitialized = false;
  uint32_t _spiFrequency = 8000000; // 8MHz

public:
  EspHal() : RadioLibHal(
                  GPIO_MODE_INPUT,    // input mode
                  GPIO_MODE_OUTPUT,   // output mode
                  0,                  // low level
                  1,                  // high level
                  GPIO_INTR_POSEDGE,  // rising edge
                  GPIO_INTR_NEGEDGE   // falling edge
                )
  {
  }
```

Sender Main Function

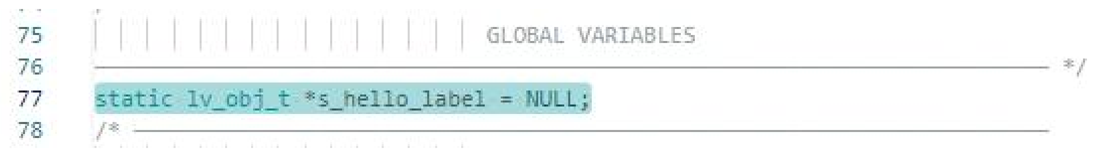
Next, let's examine how the main ".ino" code utilizes the interfaces from the above libraries to implement LoRa message transmission.

When the program runs, the general flow is as follows:

The complete workflow of this program can be summarized as follows: System startup → Initialize display and touch screen → Start LVGL graphical interface → Initialize LoRa wireless module → Create two real-time tasks (one responsible for sending LoRa data packets every second, and the other responsible for updating the screen display and sending count every second) → Real-time display of wireless transmission count on the screen and output logs through the serial port.

① **Global Variables**

This line of code defines a global pointer variable s_hello_label pointing to an LVGL graphic object, used to store the address of a text label (Label) object in the program—enabling access and updates to this label across different functions or tasks.



```
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |   GLOBAL VARIABLES
 ─────────────────────────────────────────────────────── */
static lv_obj_t *s_hello_label = NULL;
/* ──────────────────────────────────────────────────────
```

Specifically, lv_obj_t is the foundational data structure type in LVGL used to represent all graphic interface objects. Whether buttons, labels, images, or containers, all exist internally within LVGL as lv_obj_t objects; while lv_obj_t * represents a pointer to such an object. Therefore, s_hello_label is essentially a pointer variable that stores the memory address of a label object.

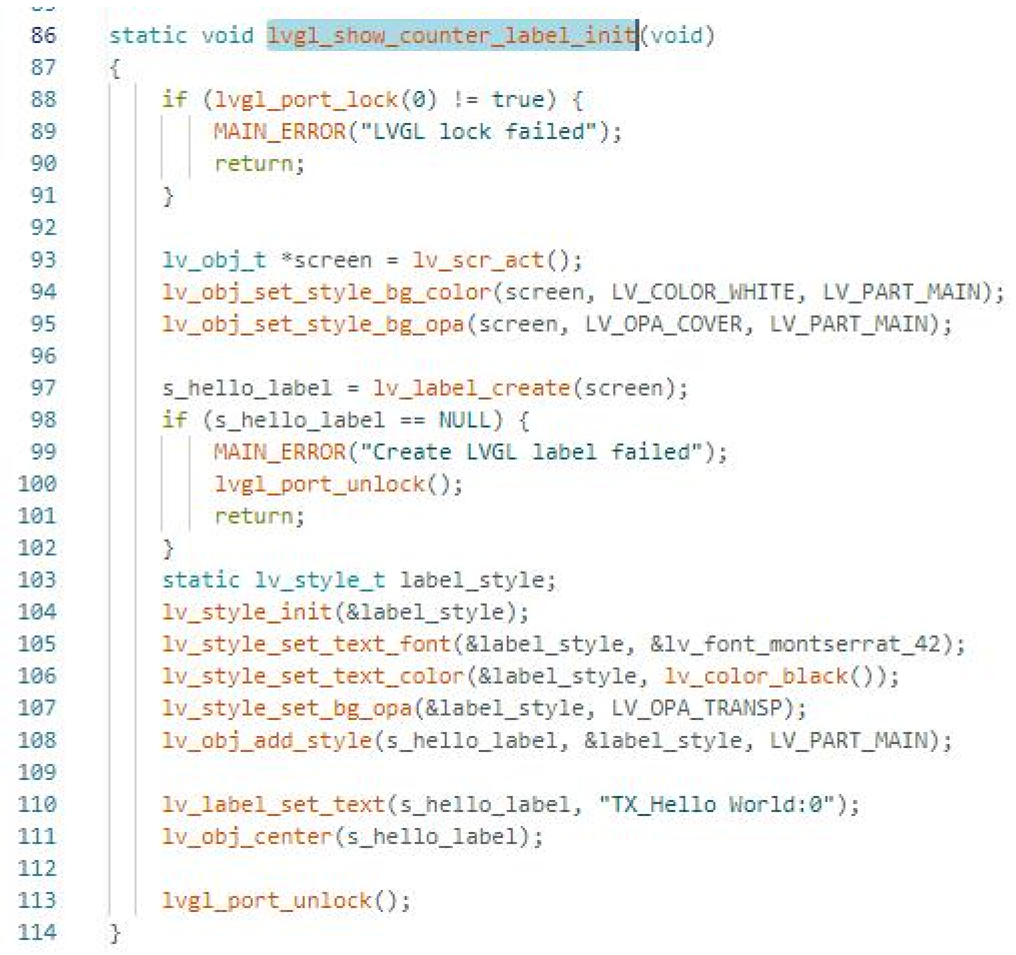
The static keyword in the code indicates that this variable has file scope and static storage duration—meaning it can only be accessed within the current source file, but persists throughout the entire program runtime without being released when functions terminate. This ensures the object pointer can be shared among multiple functions.

The s_ prefix in the variable name is typically a coding convention used to denote static global variables, facilitating quick identification of variable scope for developers reading the code.
Finally, initializing this pointer to NULL indicates that no corresponding LVGL label object has been created at program startup. Only after subsequently calling lv_label_create() to create the interface label will the returned object address be assigned to s_hello_label.

The advantage of this design is that in other tasks (such as interface update tasks), this label object can be accessed through s_hello_label, and functions like lv_label_set_text() can be called to dynamically modify label content—for example, real-time updating of counter information on screen—thereby achieving sharing and interaction between tasks and the graphical interface.

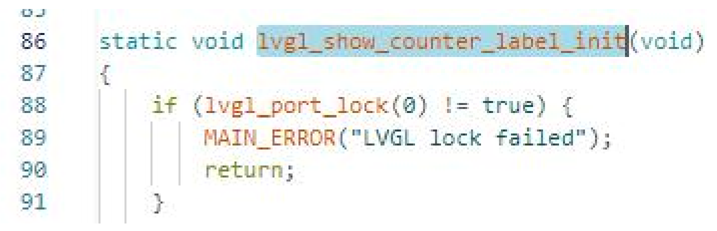
② lvgl_show_counter_label_init

This code implements a function "lvgl_show_counter_label_init()" used to initialize the LVGL graphical interface and create a counter display label on screen. Its primary role is to create a text label and display initial information after system startup, preparing for subsequent real-time updates of LoRa transmission counters.



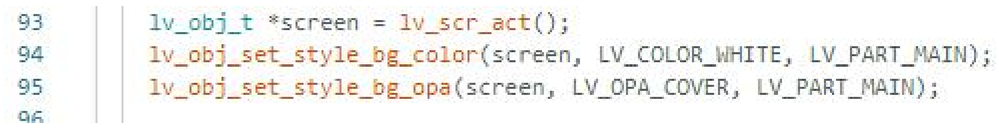
```c
static void lvgl_show_counter_label_init(void)
{
    if (lvgl_port_lock(0) != true) {
        MAIN_ERROR("LVGL lock failed");
        return;
    }

    lv_obj_t *screen = lv_scr_act();
    lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);
    lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);

    s_hello_label = lv_label_create(screen);
    if (s_hello_label == NULL) {
        MAIN_ERROR("Create LVGL label failed");
        lvgl_port_unlock();
        return;
    }
    static lv_style_t label_style;
    lv_style_init(&label_style);
    lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
    lv_style_set_text_color(&label_style, lv_color_black());
    lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
    lv_obj_add_style(s_hello_label, &label_style, LV_PART_MAIN);

    lv_label_set_text(s_hello_label, "TX_Hello World:0");
    lv_obj_center(s_hello_label);

    lvgl_port_unlock();
}
```

At the start of the function, "lvgl_port_lock(0)" is first called to lock the graphics library. This is necessary because the program runs in a FreeRTOS multi-tasking environment, and LVGL is not thread-safe. Therefore, the LVGL access lock must be acquired before modifying interface objects. If the lock fails, an error message is output via the logging macro and the function returns immediately—preventing conflicts caused by multiple tasks simultaneously operating on the graphical interface.



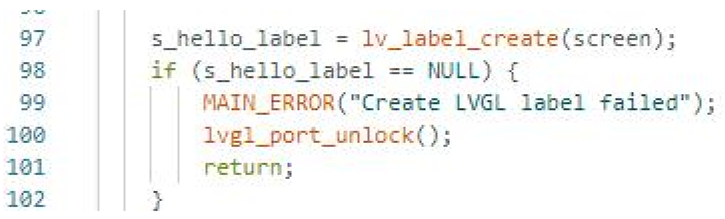
```c
static void lvgl_show_counter_label_init(void)
{
    if (lvgl_port_lock(0) != true) {
        MAIN_ERROR("LVGL lock failed");
        return;
    }
```

After acquiring the lock, the program obtains the currently active screen object (i.e., the current display screen container) via lv_scr_act(), and uses lv_obj_set_style_bg_color() and lv_obj_set_style_bg_opa() to set the screen background color to white with full opacity—ensuring the entire screen displays as a pure white background.



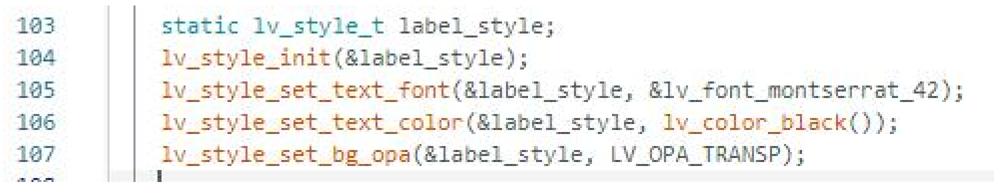
```
    lv_obj_t *screen = lv_scr_act();
    lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);
    lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);
```

The program then calls lv_label_create(screen) to create a text label object on the current screen, saving its address to the global pointer variable s_hello_label. This allows subsequent tasks to access and update this label's content through this pointer. If label creation fails, an error message is output, the LVGL lock is released, and the function exits.



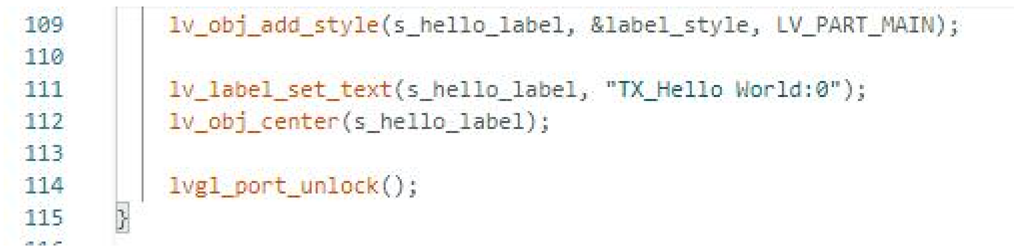
```
    s_hello_label = lv_label_create(screen);
    if (s_hello_label == NULL) {
        MAIN_ERROR("Create LVGL label failed");
        lvgl_port_unlock();
        return;
    }
```

Upon successful label creation, the program defines a static style object label_style, initializes it via lv_style_init(), and then sequentially sets the style's text font to lv_font_montserrat_42 (42-point Montserrat font), text color to black, and background opacity to transparent—ensuring the label text displays clearly against the white background.



```
    static lv_style_t label_style;
    lv_style_init(&label_style);
    lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
    lv_style_set_text_color(&label_style, lv_color_black());
    lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
```

After completing style configuration, the program applies this style to the s_hello_label label object via lv_obj_add_style(). It then calls lv_label_set_text() to set the label's initial display content to "TX_Hello World:0", indicating the current wireless transmission count is 0, and uses lv_obj_center() to automatically center the label on screen for a cleaner interface layout.

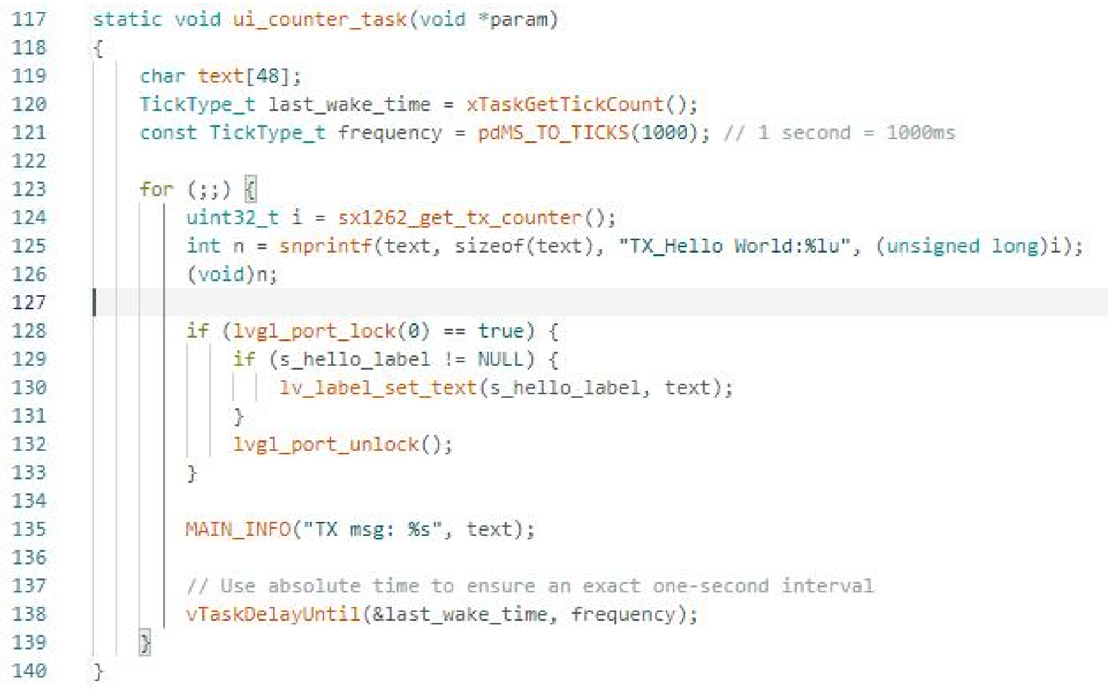


```
    lv_obj_add_style(s_hello_label, &label_style, LV_PART_MAIN);

    lv_label_set_text(s_hello_label, "TX_Hello World:0");
    lv_obj_center(s_hello_label);

    lvgl_port_unlock();
}
```

Finally, after all interface objects are created and configured, the program calls lvgl_port_unlock() to release the LVGL access lock—allowing other tasks to safely access the graphical interface.

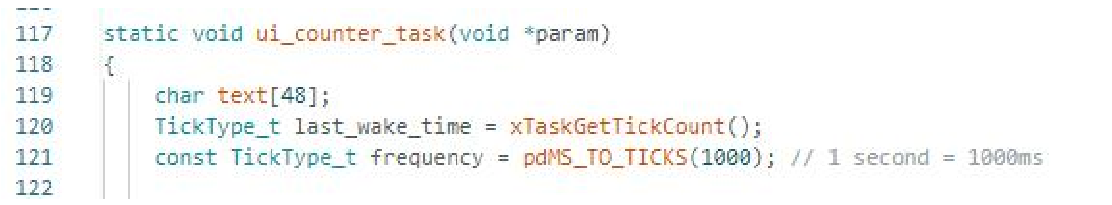
③ ui_counter_task

This code implements a FreeRTOS task function "ui_counter_task()", whose primary role is to periodically retrieve the LoRa module's transmission counter, update the on-screen text label in real-time, and output log information via the serial port—thereby achieving visualized display of wireless transmission status.



```
static void ui_counter_task(void *param)
{
    char text[48];
    TickType_t last_wake_time = xTaskGetTickCount();
    const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms

    for (;;) {
        uint32_t i = sx1262_get_tx_counter();
        int n = snprintf(text, sizeof(text), "TX_Hello World:%lu", (unsigned long)i);
        (void)n;

        if (lvgl_port_lock(0) == true) {
            if (s_hello_label != NULL) {
                lv_label_set_text(s_hello_label, text);
            }
            lvgl_port_unlock();
        }

        MAIN_INFO("TX msg: %s", text);

        // Use absolute time to ensure an exact one-second interval
        vTaskDelayUntil(&last_wake_time, frequency);
    }
}
```

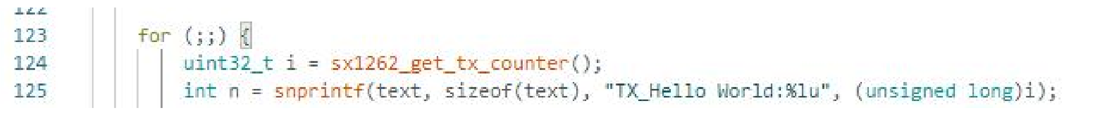
The function begins by defining a character array text[48] to store the string content to be displayed. It then calls xTaskGetTickCount() to obtain the current RTOS clock tick count, saving it to the last_wake_time variable—used subsequently for precise periodic task scheduling. Next, it converts 1000 milliseconds to system tick values via pdMS_TO_TICKS(1000) and saves it to frequency, indicating this task's execution period is 1 second.



```
static void ui_counter_task(void *param)
{
    char text[48];
    TickType_t last_wake_time = xTaskGetTickCount();
    const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms

```

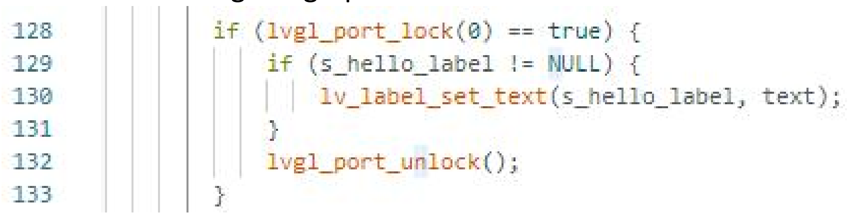
The program then enters an infinite loop for(;;)—a common execution structure for FreeRTOS tasks, where the task runs continuously without exiting.

During each loop iteration, it first calls sx1262_get_tx_counter() to retrieve the current LoRa module's transmission counter value, saving the result to variable i. It then uses snprintf() to format this counter value into the string "TX_Hello World:%lu" and writes it to the text buffer—for example, "TX_Hello World:15"—making the string ready for direct screen display.



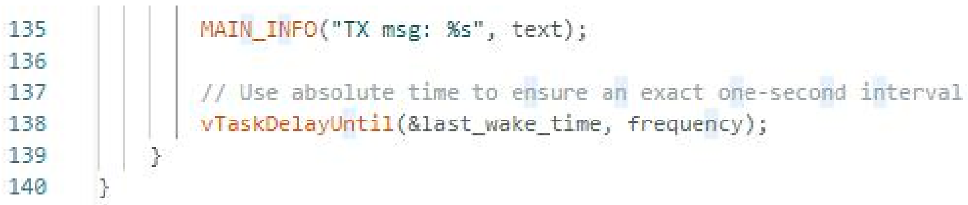
```
    for (;;) {
        uint32_t i = sx1262_get_tx_counter();
        int n = snprintf(text, sizeof(text), "TX_Hello World:%lu", (unsigned long)i);
```

The program then attempts to acquire the LVGL access lock via lvgl_port_lock(0), as LVGL is not thread-safe in multi-tasking environments. Therefore, any task must acquire the lock before modifying interface objects. If the lock is successfully obtained, it further checks whether the global label object pointer s_hello_label is non-null. If non-null, it calls lv_label_set_text() to write the new string text to the label—thereby updating the transmission counter displayed on screen. After updating, it calls lvgl_port_unlock() to release the LVGL lock, allowing other tasks to continue accessing the graphical interface.



```
if (lvgl_port_lock(0) == true) {
    if (s_hello_label != NULL) {
        lv_label_set_text(s_hello_label, text);
    }
    lvgl_port_unlock();
}
```

After completing the interface update, the program outputs the current transmission information via the logging macro MAIN_INFO() to the serial port—for example, "TX msg: TX_Hello World:15"—facilitating observation of system operation status through the serial monitor during debugging.

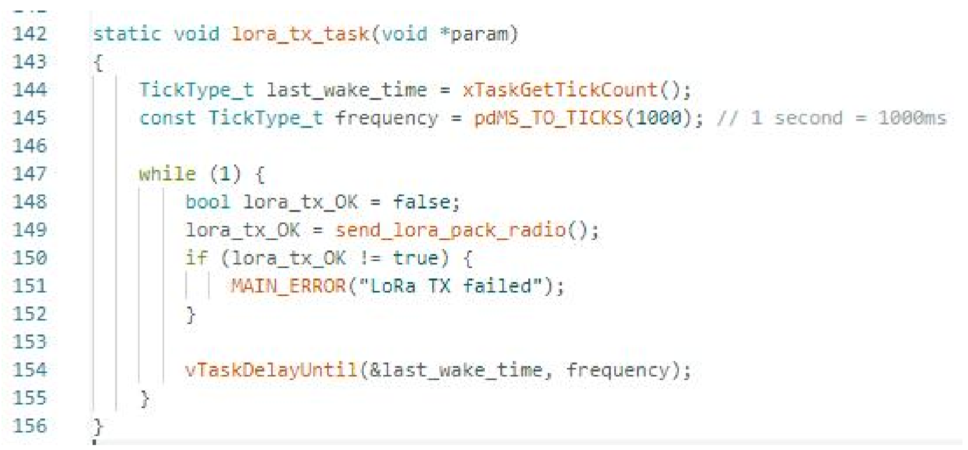


```
            MAIN_INFO("TX msg: %s", text);

            // Use absolute time to ensure an exact one-second interval
            vTaskDelayUntil(&last_wake_time, frequency);
        }
    }
```

Finally, the task calls vTaskDelayUntil(&last_wake_time, frequency) to enter delay wait. This delay method differs from ordinary vTaskDelay() in that it uses absolute time scheduling to control task periodicity—thereby ensuring the task always executes at precise 1-second intervals, without accumulating errors even if internal execution times vary slightly.

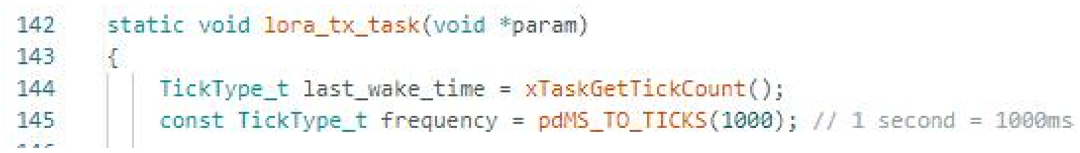
④ **lora_tx_task**

This code implements a FreeRTOS task function "lora_tx_task()", whose primary role is to transmit data packets to the LoRa wireless module at fixed time intervals and detect whether transmission succeeds—thereby achieving continuous wireless data transmission functionality.



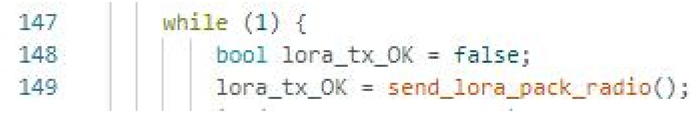
```
static void lora_tx_task(void *param)
{
    TickType_t last_wake_time = xTaskGetTickCount();
    const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms

    while (1) {
        bool lora_tx_OK = false;
        lora_tx_OK = send_lora_pack_radio();
        if (lora_tx_OK != true) {
            MAIN_ERROR("LoRa TX failed");
        }

        vTaskDelayUntil(&last_wake_time, frequency);
    }
}
```

The function begins by calling xTaskGetTickCount() to obtain the current RTOS clock tick count, saving it to the variable last_wake_time—this variable records the time point when the task last woke up. It then converts 1000 milliseconds to system tick values via pdMS_TO_TICKS(1000) and stores it in the constant frequency, indicating this task's execution period is 1 second.



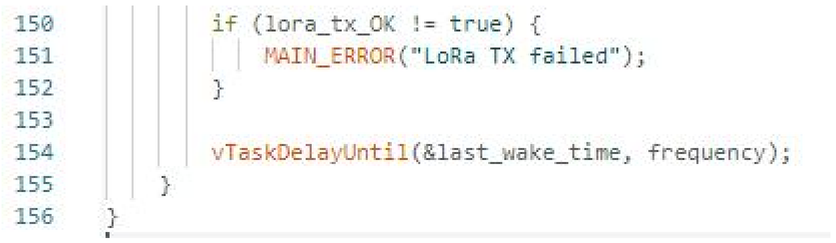
```c
static void lora_tx_task(void *param)
{
    TickType_t last_wake_time = xTaskGetTickCount();
    const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms
```

The program then enters an infinite loop while(1)—a common task structure in FreeRTOS that ensures continuous task execution without exiting during system operation. During each loop iteration, it first defines a boolean variable lora_tx_OK initialized to false, then calls the send_lora_pack_radio() function to transmit a data packet to the LoRa wireless module. This function completes underlying operations including data encapsulation, SPI communication, and triggering wireless transmission, returning whether transmission succeeded.



```c
    while (1) {
        bool lora_tx_OK = false;
        lora_tx_OK = send_lora_pack_radio();
```

The program assigns this return value to the lora_tx_OK variable. If transmission fails (i.e., return value is not true), it outputs the error message "LoRa TX failed" via the logging macro MAIN_ERROR() to the serial port—alerting to wireless transmission problems and facilitating timely detection of communication anomalies during debugging. If transmission succeeds, no error message is output and the task continues to the next step.



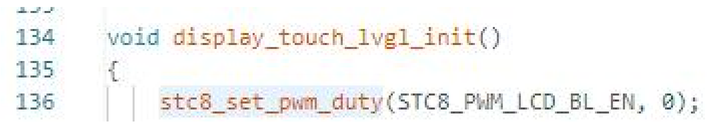
```c
        if (lora_tx_OK != true) {
            MAIN_ERROR("LoRa TX failed");
        }

        vTaskDelayUntil(&last_wake_time, frequency);
    }
}
```

Finally, the task calls vTaskDelayUntil(&last_wake_time, frequency) to enter periodic delay. This delay method differs from ordinary vTaskDelay() in that it uses an absolute-time periodic scheduling mechanism—ensuring the task always runs at fixed time intervals, i.e., executing transmission operations every 1 second. Even if internal execution times vary slightly, this prevents period drift—thereby achieving stable and precise timed transmission.

⑤ **display_touch_lvgl_init**

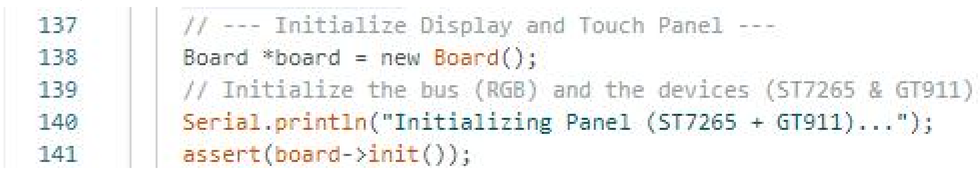
The main function of this display_touch_lvgl_init() function is to initialize the screen backlight, RGB display screen, touch screen, and LVGL graphical interface system, thereby enabling the device to have complete graphic display and touch input capabilities.

The function begins by calling stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0) to set the LCD backlight PWM duty cycle to 0%, which means turning off the screen backlight first. This step is usually used to prevent screen flickering or abnormal brightness during the display initialization process. The PWM control is handled by the external coprocessor STC8H1KXX.



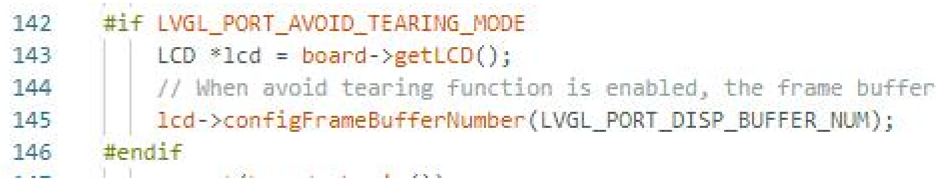
```
void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);
```

Then the program creates a Board object named "board" using the "ESP32_Display_Panel" library, which is used to uniformly manage the display bus, LCD controller, and touch device. Subsequently, it prints "Initializing Panel (ST7265 + GT911)..." indicating that the system is initializing the display and touch hardware. The LCD driver chip is ST7265 and the touch controller is GT911.
The program initializes the display bus (RGB interface) and related devices through "board->init()". If the initialization fails, the assert() function will trigger the program to stop.



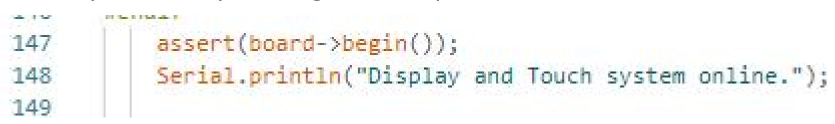
```
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
```

Next, under the conditional compilation #if LVGL_PORT_AVOID_TEARING_MODE, if the tearing avoidance mode is enabled, the program will obtain the LCD object board->getLCD() and call configFrameBufferNumber() to set the number of frame buffers, in order to reduce the problem of image tearing that occurs during screen refresh.



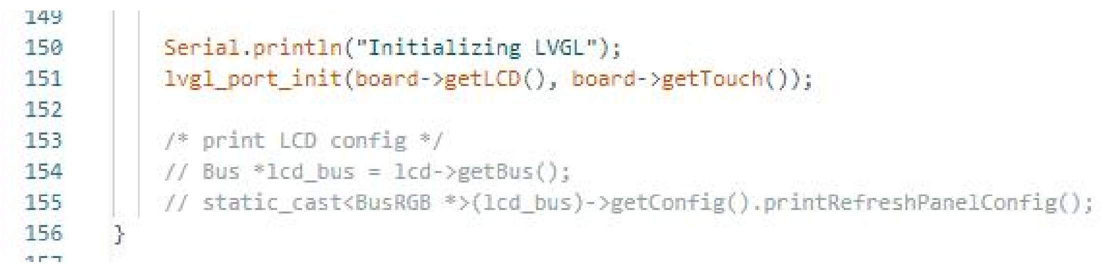
```
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
```

Then, call board->begin() to start the display panel. This process usually involves LCD startup timing, driver configuration, and display interface startup. Once it is successful, print "Display and Touch system online." to indicate that the display and touch system is operating normally.



```
    assert(board->begin());
    Serial.println("Display and Touch system online.");

```

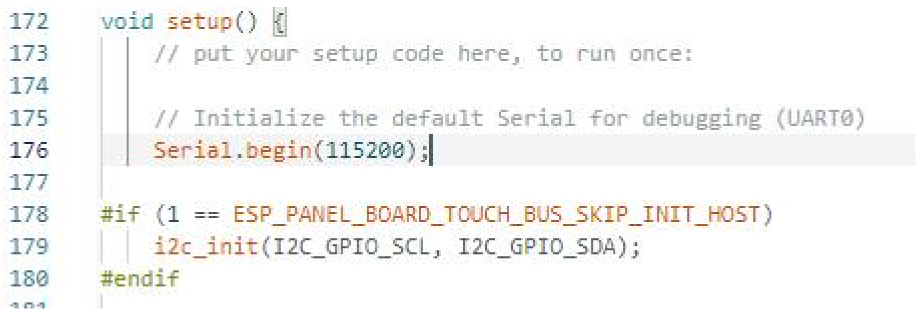
Finally, the program initializes the LVGL graphics library. By calling lvgl_port_init(board->getLCD(), board->getTouch()), the LCD display device and touch input device are registered in LVGL, enabling LVGL to handle screen drawing and touch event processing. The function also retains a section of commented-out debugging code at the end, which is used to obtain the RGB bus configuration and print the LCD refresh parameters.



```
    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());

    /* print LCD config */
    // Bus *lcd_bus = lcd->getBus();
    // static_cast<BusRGB *>(lcd_bus)->getConfig().printRefreshPanelConfig();
}
```

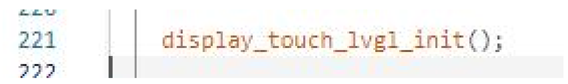
⑥**setup**
This code is the setup() initialization function of the Arduino program. It is executed only once when the system is powered on or reset. Its main responsibility is to complete the hardware initialization of the entire system, the initialization of the interface, and the creation of tasks, thus establishing a complete operating environment for the subsequent program execution.

The function first initiates serial communication (UART0) through Serial.begin(115200), setting the baud rate to 115200, which is used to output debugging information;
Then, through conditional compilation #if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST), it determines whether manual initialization of the I²C bus is required. If the macro ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST is set to 1, it calls i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) to initialize the I²C interface using the specified SCL and SDA pins. In this way, the main control ESP32-P4 can communicate with external devices.



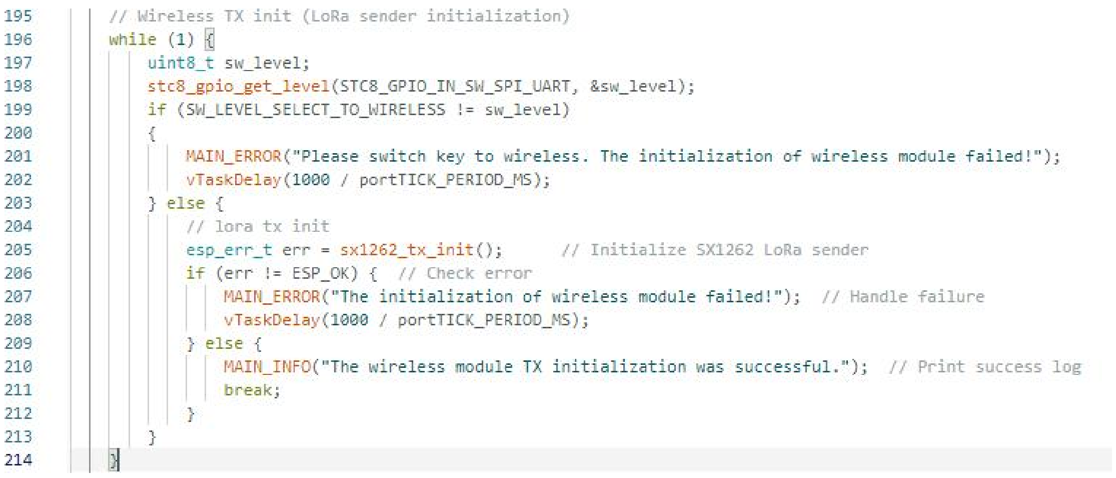
```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif
```

Then, the function display_touch_lvgl_init() is called to complete the initialization of the display and touch screen hardware, and to start the graphics interface library LVGL, thereby enabling the screen to have the ability to display graphical interfaces and receive touch inputs.



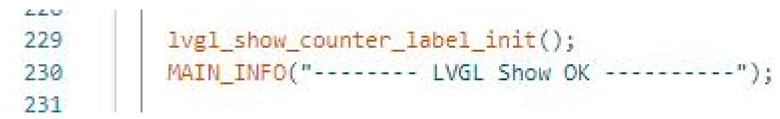
```
        display_touch_lvgl_init();
```

Next, the program enters a while(1) loop for initializing the wireless communication module: Firstly, it reads the status of a hardware switch STC8_GPIO_IN_SW_SPI_UART from the external coprocessor STC8H1KXX using stc8_gpio_get_level(). If the switch has not been switched to the wireless mode (SW_LEVEL_SELECT_TO_WIRELESS), the program will continuously prompt the user to switch the switch; when the switch is correct, the program calls sx1262_tx_init() to initialize the SX1262 wireless transmission module. If the initialization fails, it will continue to retry in the loop; if successful, it will print the log and exit the loop.



```
// Wireless TX init (LoRa sender initialization)
while (1) {
    uint8_t sw_level;
    stc8_gpio_get_level(STC8_GPIO_IN_SW_SPI_UART, &sw_level);
    if (SW_LEVEL_SELECT_TO_WIRELESS != sw_level)
    {
        MAIN_ERROR("Please switch key to wireless. The initialization of wireless module failed!");
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    } else {
        // lora tx init
        esp_err_t err = sx1262_tx_init();      // Initialize SX1262 LoRa sender
        if (err != ESP_OK) {  // Check error
            MAIN_ERROR("The initialization of wireless module failed!");  // Handle failure
            vTaskDelay(1000 / portTICK_PERIOD_MS);
        } else {
            MAIN_INFO("The wireless module TX initialization was successful.");  // Print success log
            break;
        }
    }
}
```

Next, the program calls the function lvgl_show_counter_label_init() to create an LVGL label control on the screen, which is used to display the wireless transmission count, such as "TX_Hello World:0", and prints a log using MAIN_INFO() to indicate that the interface has been successfully displayed.

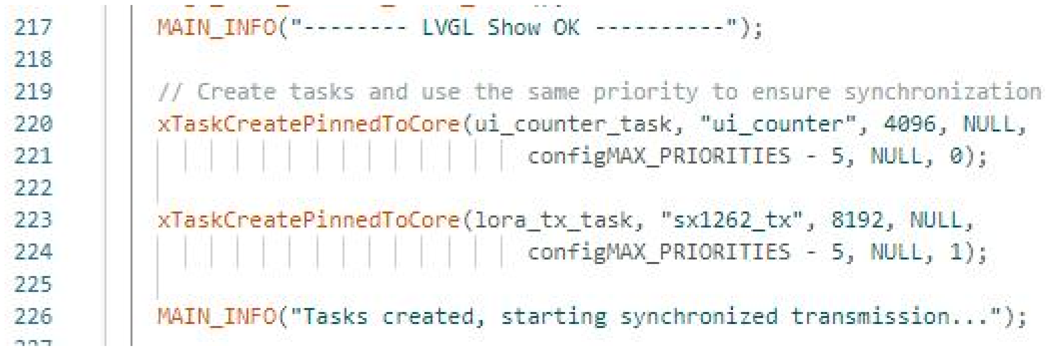


```
    lvgl_show_counter_label_init();
    MAIN_INFO("-------- LVGL Show OK ----------");
```

Then the program continued to output log messages indicating that the wireless module initialization was successful. Then, using xTaskCreatePinnedToCore(), two FreeRTOS tasks were created:

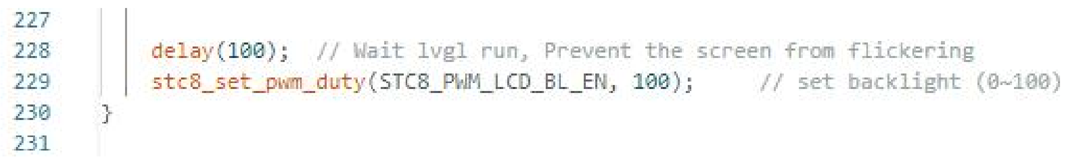
The first task, ui_counter_task, allocated 4096 bytes of stack space and was pinned to run on CPU Core 0. Its main function was to read the LoRa transmission count every second and update the text label on the screen;

The second task, lora_tx_task, allocated 8192 bytes of stack space and was pinned to run on CPU Core 1. It was responsible for sending a LoRa data packet to the wireless module every second.



```
    MAIN_INFO("-------- LVGL Show OK ----------");

    // Create tasks and use the same priority to ensure synchronization
    xTaskCreatePinnedToCore(ui_counter_task, "ui_counter", 4096, NULL,
                            configMAX_PRIORITIES - 5, NULL, 0);

    xTaskCreatePinnedToCore(lora_tx_task, "sx1262_tx", 8192, NULL,
                            configMAX_PRIORITIES - 5, NULL, 1);

    MAIN_INFO("Tasks created, starting synchronized transmission...");
```

The priorities of both tasks were set to configMAX_PRIORITIES - 5, ensuring that they have the same scheduling priority in the system and achieving a relatively synchronized operation effect. Finally, the program outputs "Tasks created, starting synchronized transmission..." in the log, indicating that the system initialization has been completed and it has entered the multi-task operation stage.



```
        delay(100);  // Wait lvgl run, Prevent the screen from flickering
        stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);     // set backlight (0~100)
    }
```

Finally, the program waits for a short period using delay(100) to allow LVGL to complete the first interface rendering, in order to avoid screen flickering. Then, it calls stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100) to set the LCD backlight PWM duty cycle to 100% through the STC8 coprocessor, thus officially turning on the screen.

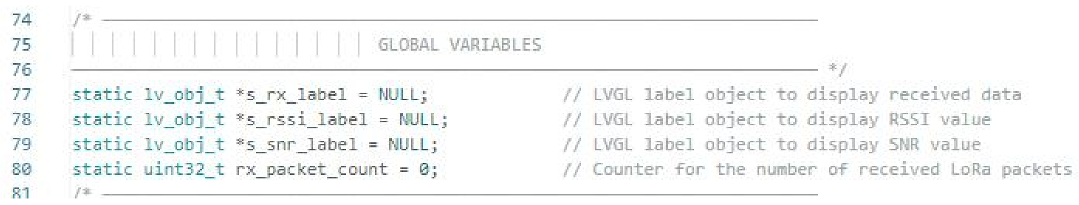
Receiver Main Function

The above is the main function code for the transmitter. Next, let's take a look at the main function code for the receiver.

① **global variable**

This section of code defines several static global variables that are crucial in the LoRa reception program:

- Then, **static lv_obj_t *s_rx_label = NULL;** defines a pointer to an LVGL label object, which is used to display the received LoRa data content on the screen.
- **static lv_obj_t *s_rssi_label = NULL;** is an interface label used to display the RSSI (signal strength) value, allowing users to know the strength of the received signal.
- **static lv_obj_t *s_snr_label = NULL;** defines another LVGL label, which is used to display the SNR (signal-to-noise ratio) value to help determine the quality of the received signal.
- Finally, **static uint32_t rx_packet_count = 0;** is a counting variable used to record the number of received LoRa data packets. It increments by 1 each time data is received, enabling real-time display of the reception count and system working status on the interface.



```
/* --------------------------------------------------------------------------
                              GLOBAL VARIABLES
-------------------------------------------------------------------------- */
static lv_obj_t *s_rx_label = NULL;          // LVGL label object to display received data
static lv_obj_t *s_rssi_label = NULL;        // LVGL label object to display RSSI value
static lv_obj_t *s_snr_label = NULL;         // LVGL label object to display SNR value
static uint32_t rx_packet_count = 0;         // Counter for the number of received LoRa packets
/* --------------------------------------------------------------------------
```

② **rx_data_callback:**

The function **rx_data_callback()** is the core callback function of the entire LoRa receiving program. It is automatically triggered and executed when the wireless module successfully receives a frame of LoRa data, and is used to process the reception event and update the interface display in real time.

- First, the function increments the reception count by rx_packet_count++ to record the arrival of a new data packet.

- Then, it calls lvgl_port_lock(0) to acquire a lock, ensuring safe operation of the LVGL graphical interface in a multi-tasking environment.
- If the lock is successfully acquired, it updates three interface elements in sequence: first, it checks whether s_rx_label exists; if it does, it uses snprintf() to format the string "RX_Hello World:<Number>", and updates the reception count displayed on the screen via lv_label_set_text().
- Next, it updates the signal strength label s_rssi_label to display the current RSSI value (Received Signal Strength Indicator, in dBm) on the interface.
- Then, it updates the signal-to-noise ratio label s_snr_label to display the SNR value (Signal-to-Noise Ratio, in dB) of the current received signal, reflecting the signal quality.
- After the interface update is completed, the function calls lvgl_port_unlock() to release the lock.
- Finally, it prints a log via MAIN_INFO(), outputting the serial number of the data received this time, the RSSI, and the SNR value to the console, facilitating debugging and system status monitoring.

Overall, the function's role is to synchronously update the screen and logs each time a LoRa data packet arrives, intuitively reflecting the system's real-time reception status and signal quality. It is a key link for data visualization and operation monitoring in the application.

③ lvgl_show_rx_interface_init:

The function **lvgl_show_rx_interface_init()** is the initialization function for the LoRa receiver interface. It is responsible for creating and beautifying the graphical interface used to display LoRa reception status before system startup or the beginning of the reception task.

The function first acquires the LVGL graphics lock via **lvgl_port_lock(0)**, ensuring safe operation of interface objects in a multi-threaded environment.

Then it calls **lv_scr_act()** to obtain the currently active screen object and sets the screen background to white with full opacity, providing a clear display background.

Next, it defines and initializes a general style **info_style**, uniformly setting the font size, text color (black), and transparent background, which is shared by the RSSI and SNR labels.

Subsequently, it creates four main interface elements in sequence:

1 Title label title_label — displays the title "LoRa RX Receiver", using a large font style and centered at the top of the screen to identify the interface function.

2 Received content label s_rx_label — shows the currently received LoRa message content, initially set to "RX_Hello World:0", positioned slightly above the center of the screen.

3 Signal strength label s_rssi_label — displays the RSSI (Received Signal Strength), initially "RSSI: -- dBm", placed at the lower left of the interface.

4 Signal-to-noise ratio label s_snr_label — displays the SNR (Signal-to-Noise Ratio), initially "SNR: -- dB", positioned at the lower right, symmetrical to the RSSI label.

All labels use predefined styles to ensure consistent fonts and colors. After creating the interface, the function calls lvgl_port_unlock() to release the lock, allowing other tasks to access the LVGL system.

Overall, the function initializes the visual interface for the LoRa receiver, providing a clear UI layout for real-time display of received data (such as message content, signal strength, and SNR). It serves as the core initialization function for the graphical display in the program.

④ **lora_rx_task:**

The function **lora_rx_task()** is the **LoRa reception task**, responsible for continuously detecting and processing data packets received from the **SX1262 module** during system operation.

- The function runs in a dedicated FreeRTOS task, using an infinite loop to continuously listen for LoRa signals.
- Inside the loop, it first calls sx1262_is_data_received() to check whether a new data packet has arrived.
- If a reception event is detected, it calls sx1262_get_received_len() to obtain the length of the received data, then passes this length as a parameter to received_lora_pack_radio(len), which handles data parsing and display logic (e.g., updating the received content, RSSI, and SNR on the interface).
- If no data is currently received, the program delays 10 ms using vTaskDelay(10 / portTICK_PERIOD_MS), reducing CPU usage and maintaining balanced task execution.

Overall, this function maintains the real-time listening mechanism for the LoRa receiver, ensuring that any incoming wireless data is captured and processed promptly. It is the core background task responsible for data reception and event handling in the LoRa communication system.

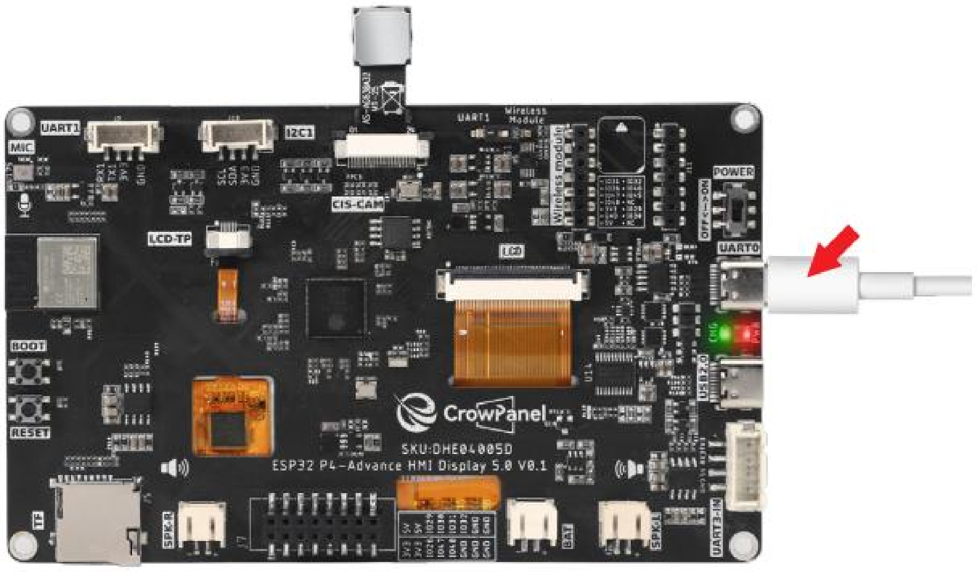
⑤ **setup**

This part of the code is consistent in logic and code with the setup section of the sending end mentioned above.

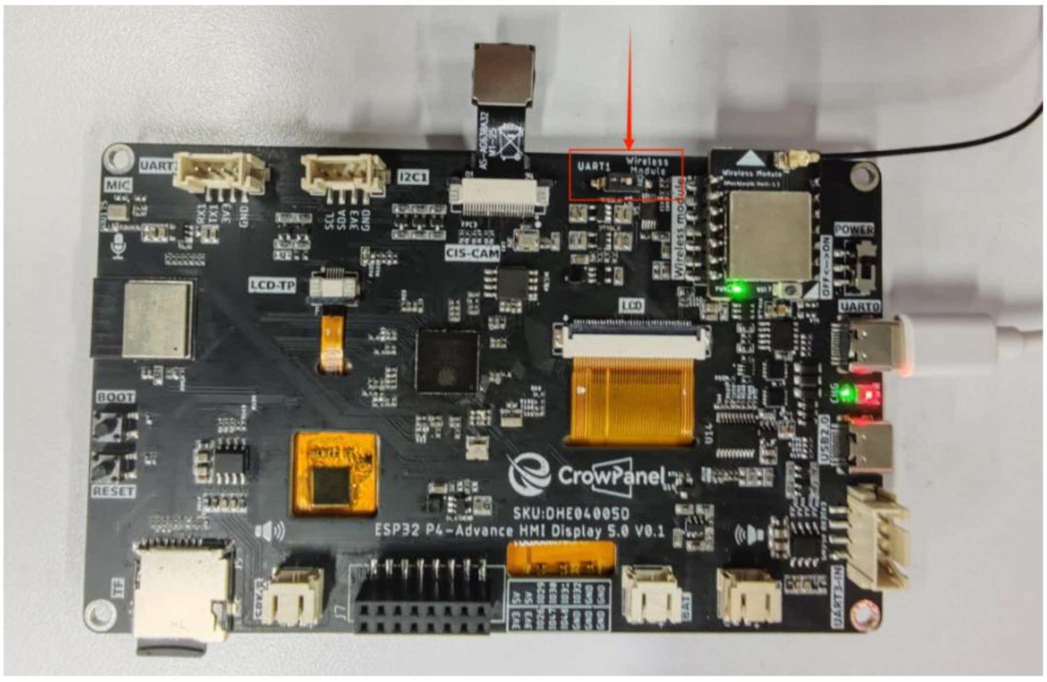
Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.

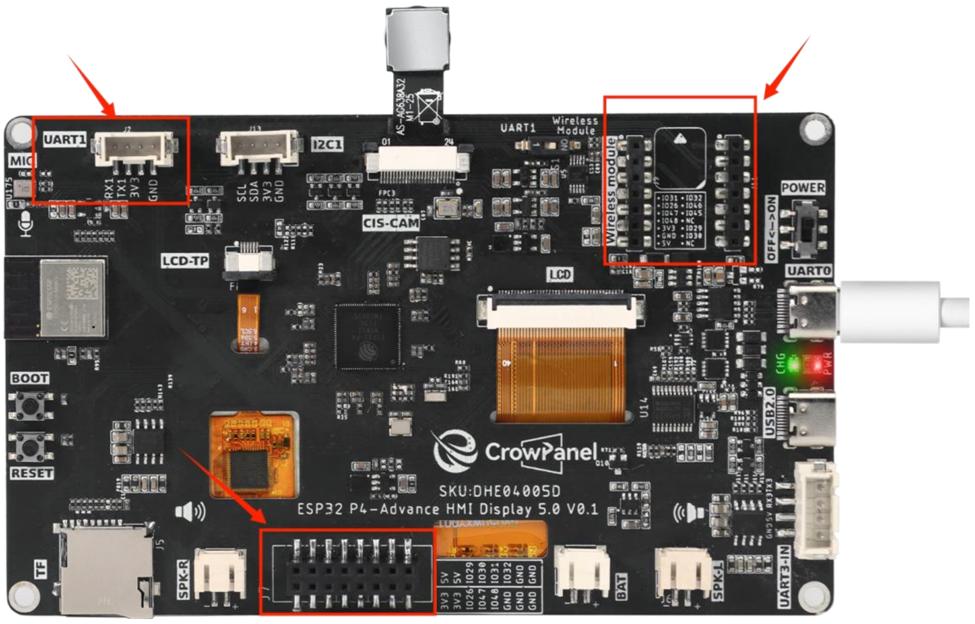
First, we connect the Advance-P4 device to our computer host via the USB cable.



For the 5-inch Advance-P4 product, the jumpers need to be switched on the hardware in
order to use the wireless module. (Switch to the side with the wireless module)



This is the design on the hardware side.



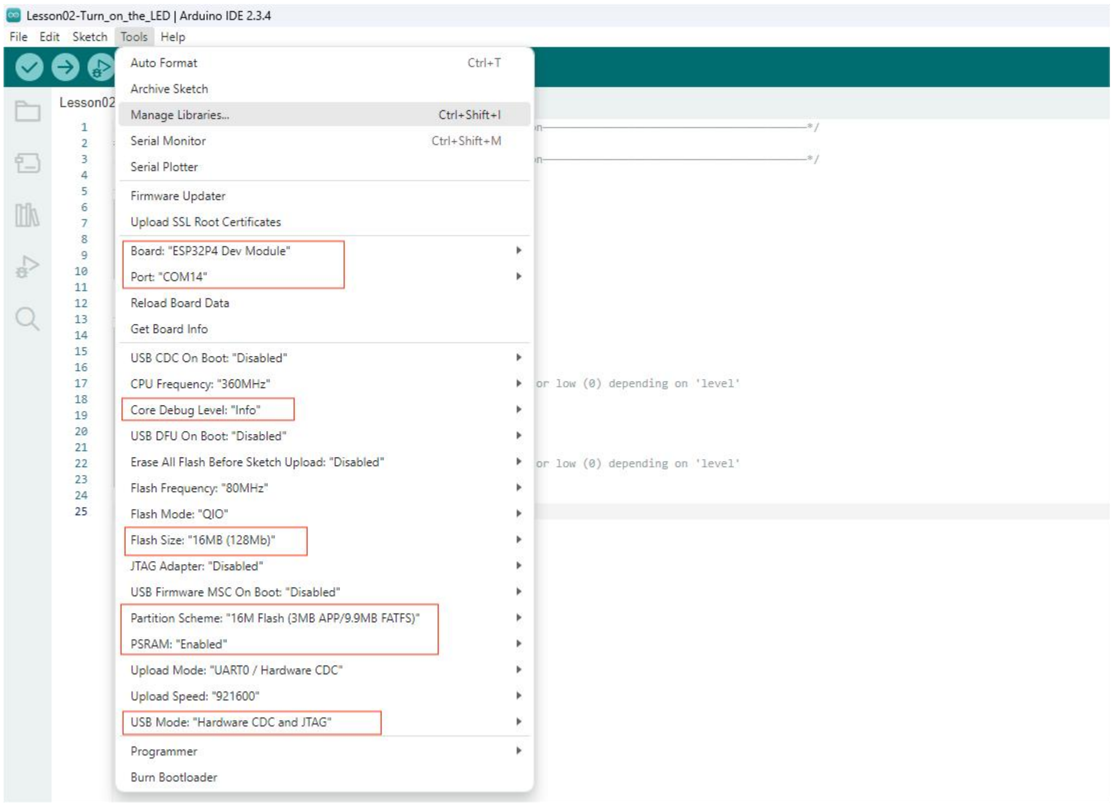
Switch to UART1 port:
Among the three interfaces shown in the figure, only the UART1 interface can be used at this time.
Alternatively, the expansion header at the bottom can also be used.
That is, either the UART1 interface or the expansion header can be used, but not both.

Switch to Wireless Module port:
Among the three interfaces shown in the figure, only the wireless module can be used at this time.
Alternatively, the expansion header at the bottom can also be used.
That is, either the wireless module or the expansion header can be used, but not both.

Summary:
The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.
The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

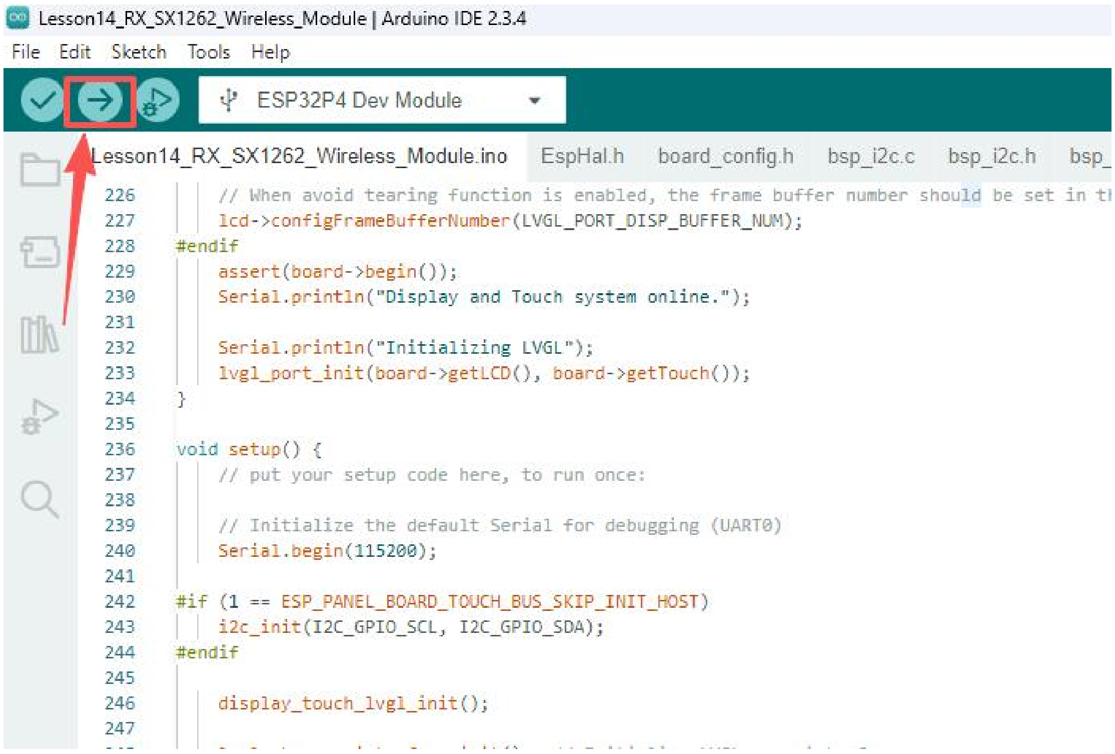
Note: The H2 and C6 wireless modules can be used simultaneously with UART1.
The Lora, 2.4GHz, and WiFi-Halow wireless modules can be used with UART1, but not simultaneously.

Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4
File Edit Sketch Tools Help
Lesson02

```
Auto Format                                        Ctrl+T
Archive Sketch
Manage Libraries...                                Ctrl+Shift+I
Serial Monitor                                     Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader
```

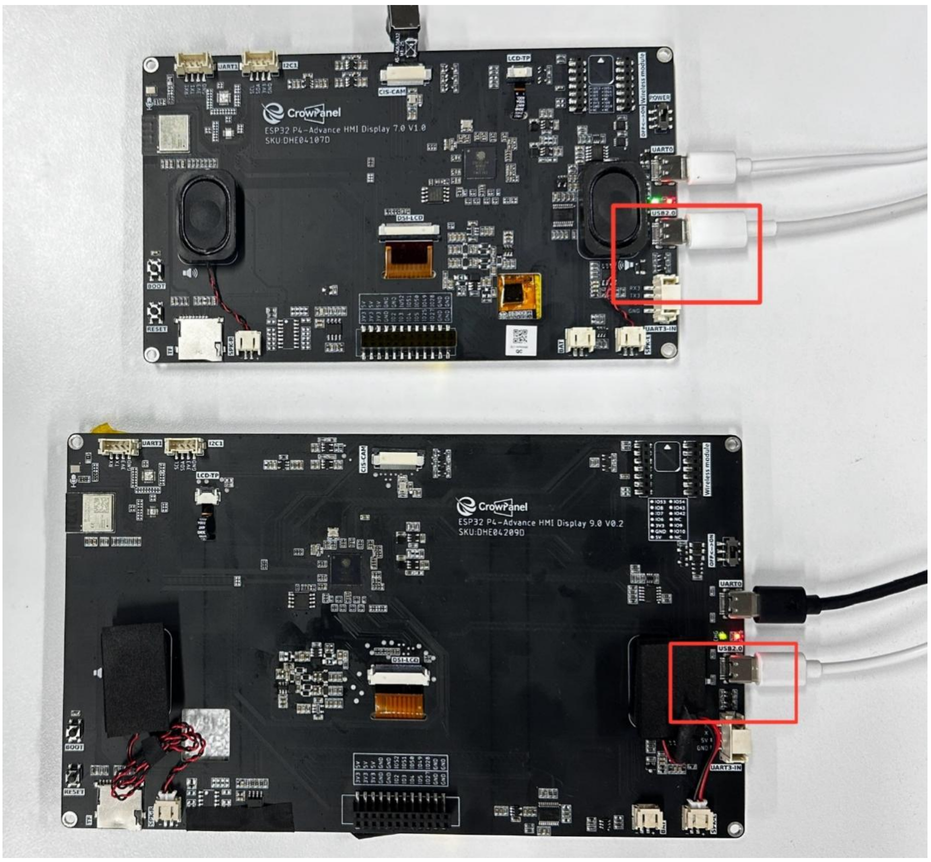
Then we compile and upload the code.


Lesson14_RX_SX1262_Wireless_Module | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson14_RX_SX1262_Wireless_Module.ino
EspHal.h
board_config.h
bsp_i2c.c
bsp_i2c.h
bsp_

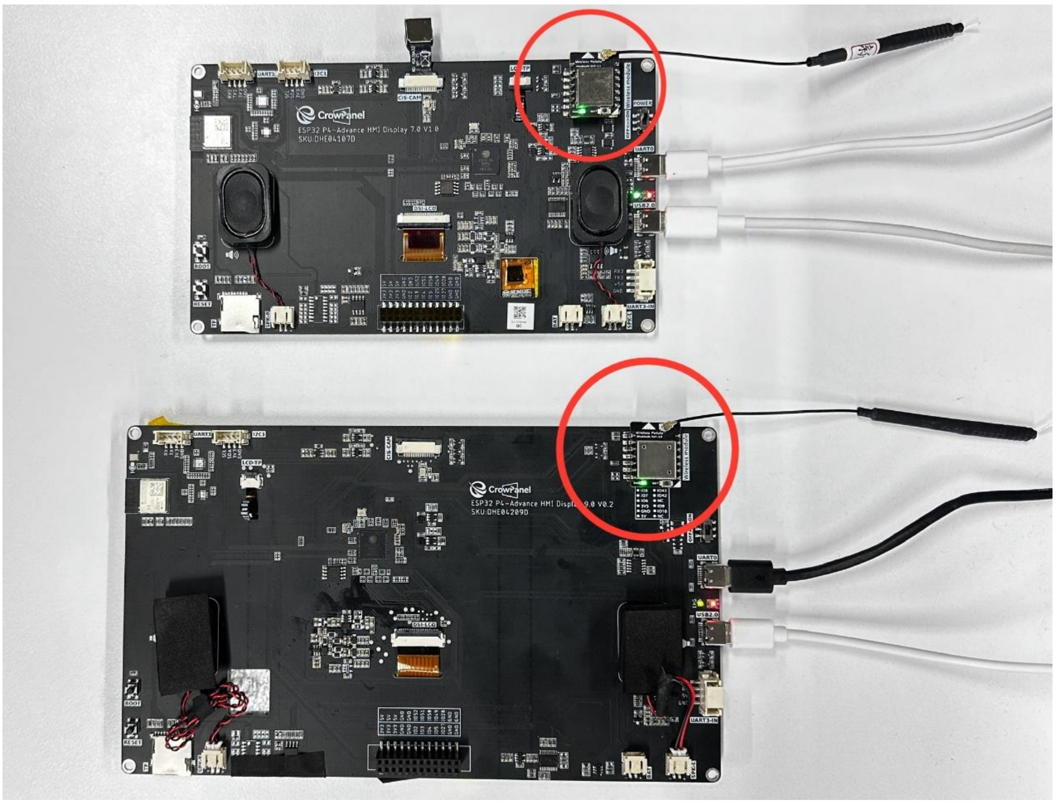
```
    // When avoid tearing function is enabled, the frame buffer number should be set in th
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
    assert(board->begin());
    Serial.println("Display and Touch system online.");

    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());
}

void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif

    display_touch_lvgl_init();

```

Wait for the code upload to complete.

At this point, remember to connect your Advance-P4 using an additional Type-C cable via the **USB 2.0 interface**. This is because the maximum current provided by a computer's USB-A port is generally 500mA, while the Advance-P4 requires a sufficient power supply when using multiple peripherals—especially a display. (Using a dedicated charger is recommended.)

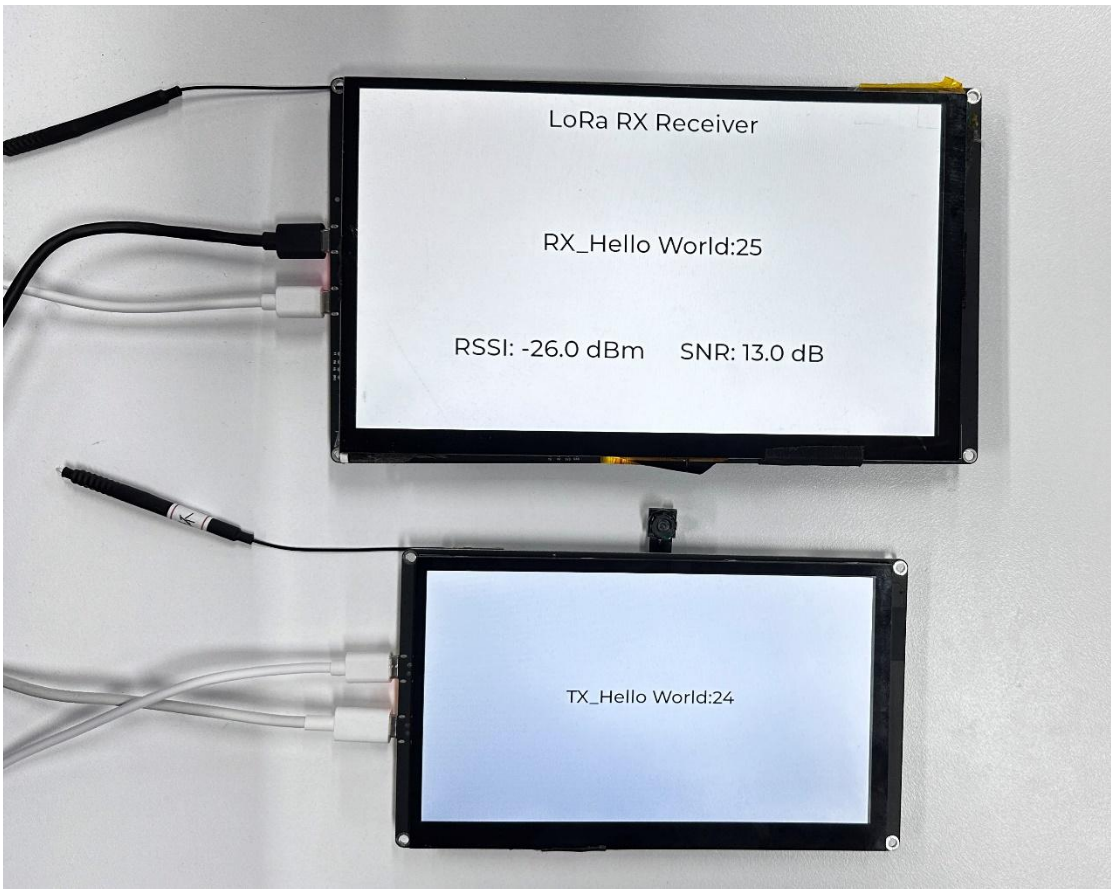


Insert the LoRa module SX1262 into the two Advance-P4 development boards respectively.



After inserting the modules and running the code on each board respectively, you will be able to see the LoRa module transmitting "TX_Hello World:i" on the screen of the transmitter-side Advance-P4, with the value of "i" increasing by 1 every second. Similarly, on the screen of the receiver-side Advance-P4, you can see the LoRa module receiving "RX_Hello World:i". When a message is received, "i" also increases by 1 every second. At the same time, you can also view the relevant received signal status: RSSI and SNR.

- **RSSI (Received Signal Strength Indicator)** indicates the strength of the received signal, with the unit of **dBm (decibel-milliwatts)**. A larger value (closer to 0) means a stronger signal; a smaller value (e.g., -120 dBm) means a weaker signal. It can reflect the distance between the receiver and the transmitter, as well as the stability of the communication link.
- **SNR (Signal-to-Noise Ratio)** represents the ratio of the signal to noise, also with the unit of **dB (decibels)**. A higher SNR indicates better signal quality and lower noise; an excessively low SNR (even negative values) means the signal is severely interfered with by noise.



Lesson15--- nRF2401 Wireless RF Module

Introduction

In this lesson, we will start using another wireless module. Since we will implement the transmission and reception functions of the nRF2401 module, two Advance-P4 development boards and two nRF2401 wireless RF (Radio Frequency) communication modules are required.

The project to be completed in this lesson is as follows: When the nRF2401 module is connected to the wireless module slot of the Advance-P4, the transmitter-side Advance-P4 screen will display **"NRF24_TX_Hello World:i"**, and the corresponding receiver-side Advance-P4 screen will display **"NRF24_RX_Hello World:i"**. The value of "i" on the receiver will only increment by 1 when it receives the signal from the transmitter.

Learning Goals

1. Understand the working principles of the nRF24L01 2.4GHz wireless module (SPI communication, transceiver pipe address matching, 2.4G band parameter configuration), as well as the hardware pin adaptation rules between ESP32-P4 and nRF24L01 (CS chip select, IRQ interrupt, CE enable, etc.).

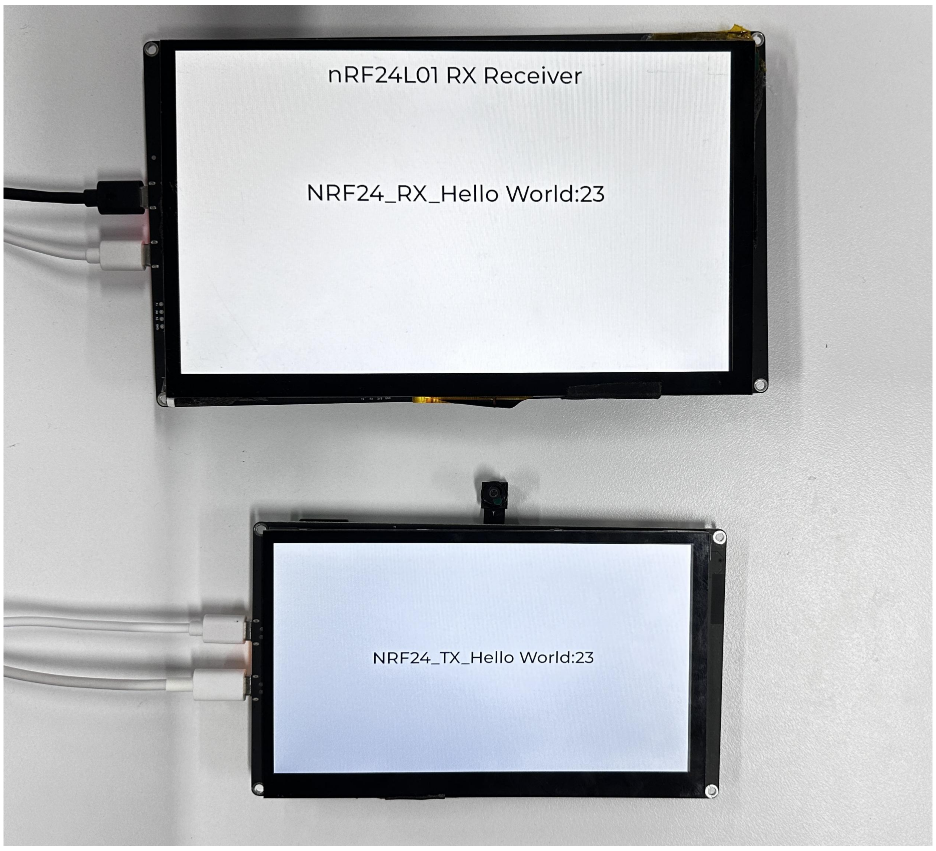
2. Master the core implementation methods for nRF24L01 module initialization configuration (2.4GHz frequency, 250kbps data rate, communication channel, transceiver pipe addresses), data transmission (timed packet sending, transmission counter updates), and reception (interrupt callbacks, data reading).

3. Master the design of nRF24L01 transceiver logic based on FreeRTOS multi-tasking (separation of transmission task, reception task, and UI update task), as well as thread-safe implementation of dynamic LVGL interface updates (transmission/reception counter display).

Preview of the Result

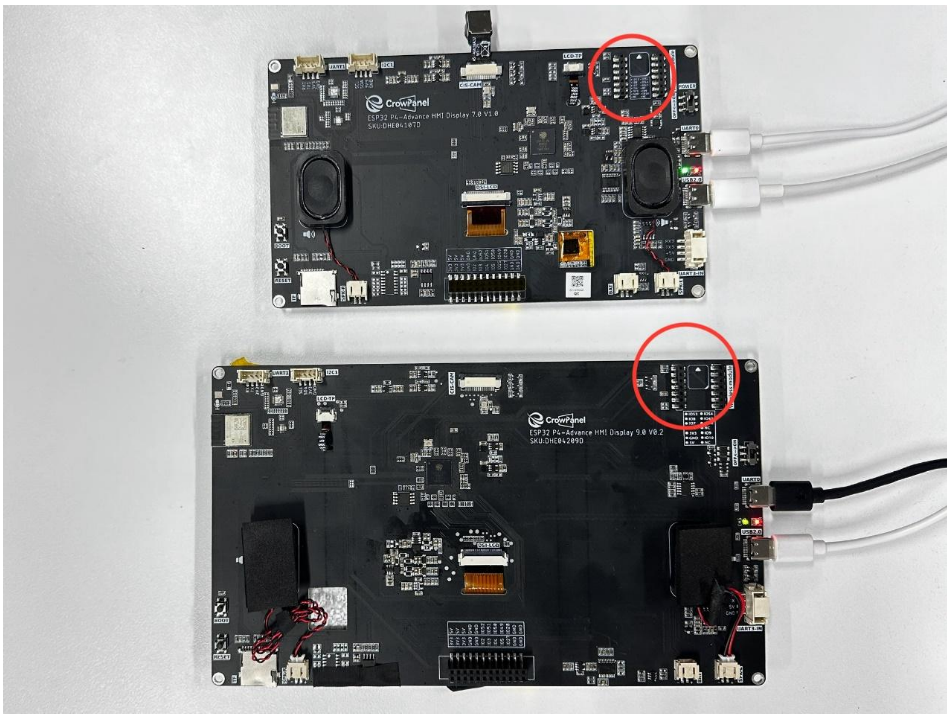
After inserting the nRF2401 wireless RF modules into the two Advance-P4 development boards and running the code on each respectively, you will be able to see the nRF2401 module transmitting "NRF24_TX_Hello World:i" on the screen of the transmitter-side Advance-P4, with the value of "i" increasing by 1 every second.

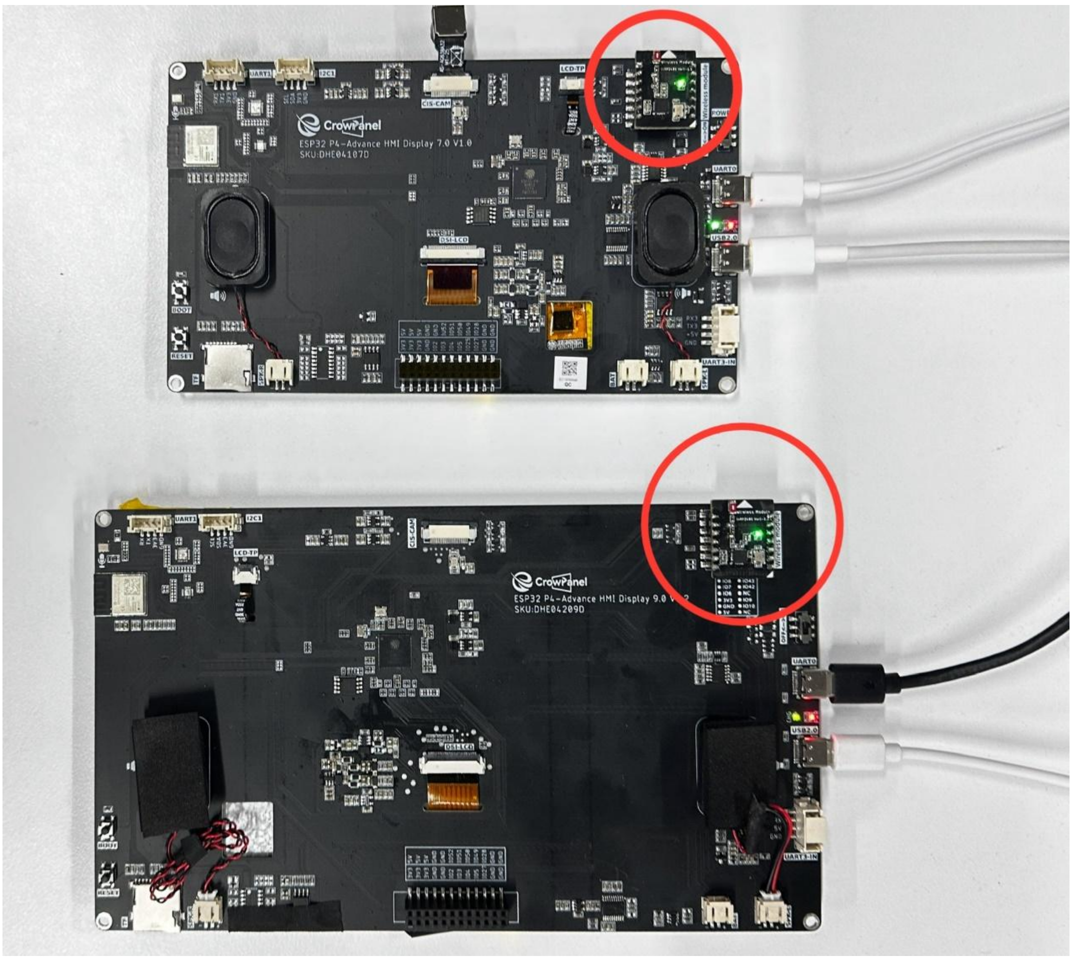
Similarly, on the screen of the receiver-side Advance-P4, you can see the nRF2401 module receiving "NRF24_RX_Hello World:i". When a message is received, "i" also increases by 1 every second.



Hardware Used in This Lesson

nRF2401 Wireless Module on Advance-P4





Complete Code

First, click the GitHub link below to download the code for this lesson.

Kindly click the link below to view the full code implementation.
Transmitting end code：
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson15_TX_nRF2401_Wireless_RF_Module

Receiving end code：
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson15_RX_nRF2401_Wireless_RF_Module

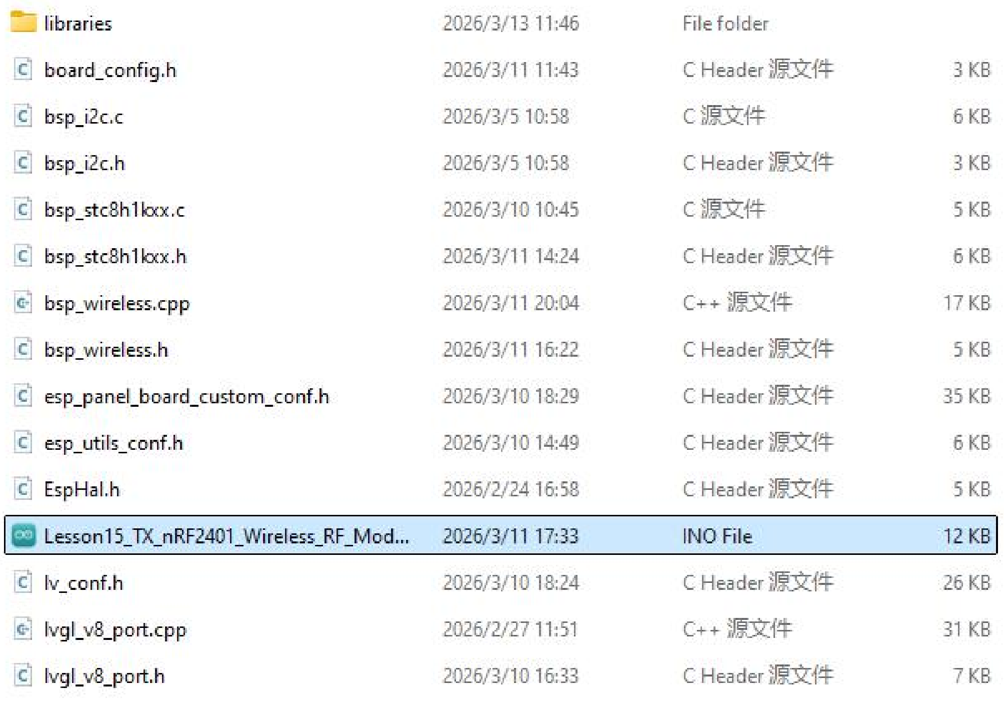
Key Explanations

The focus of this lesson is on how to use the wireless module, including initializing the nRF2401 module and sending or receiving information.

Double-click to open the code for this lesson (the .ino file).

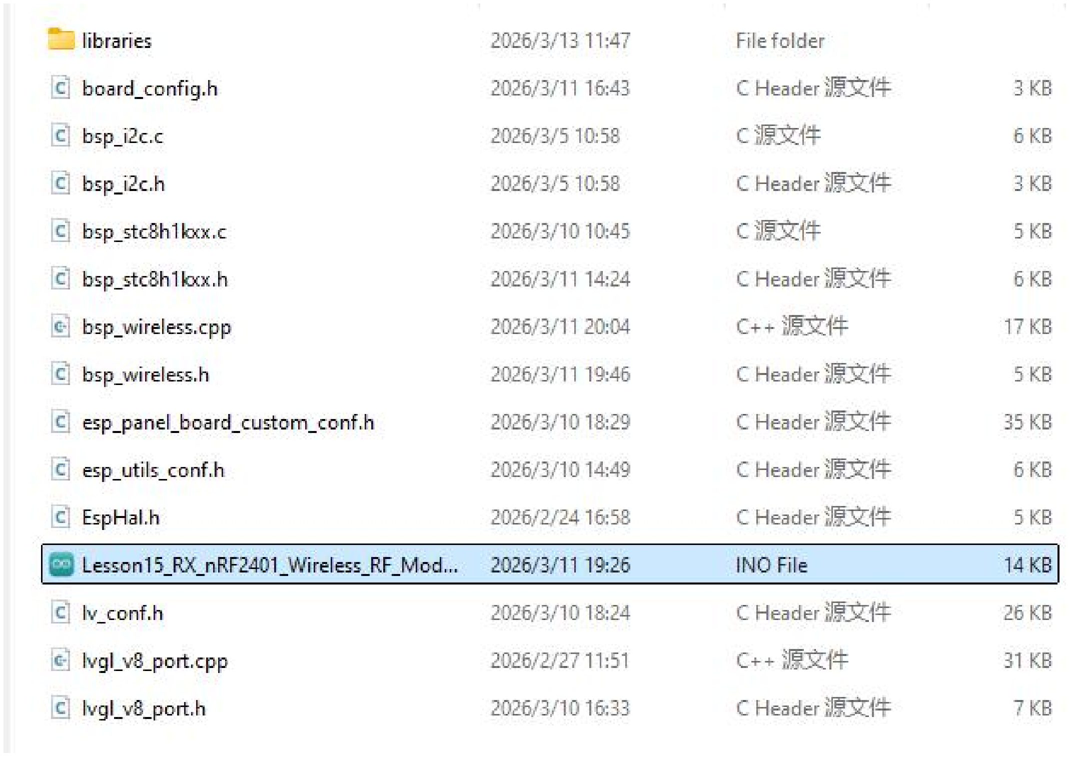
The code for this lesson includes a transmitter side and a receiver side:

Transmitter side:



libraries	2026/3/13 11:46	File folder	
board_config.h	2026/3/11 11:43	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/11 14:24	C Header 源文件	6 KB
bsp_wireless.cpp	2026/3/11 20:04	C++ 源文件	17 KB
bsp_wireless.h	2026/3/11 16:22	C Header 源文件	5 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
EspHal.h	2026/2/24 16:58	C Header 源文件	5 KB
Lesson15_TX_nRF2401_Wireless_RF_Mod...	2026/3/11 17:33	INO File	12 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

Receiver side:



libraries	2026/3/13 11:47	File folder	
board_config.h	2026/3/11 16:43	C Header 源文件	3 KB
bsp_i2c.c	2026/3/5 10:58	C 源文件	6 KB
bsp_i2c.h	2026/3/5 10:58	C Header 源文件	3 KB
bsp_stc8h1kxx.c	2026/3/10 10:45	C 源文件	5 KB
bsp_stc8h1kxx.h	2026/3/11 14:24	C Header 源文件	6 KB
bsp_wireless.cpp	2026/3/11 20:04	C++ 源文件	17 KB
bsp_wireless.h	2026/3/11 19:46	C Header 源文件	5 KB
esp_panel_board_custom_conf.h	2026/3/10 18:29	C Header 源文件	35 KB
esp_utils_conf.h	2026/3/10 14:49	C Header 源文件	6 KB
EspHal.h	2026/2/24 16:58	C Header 源文件	5 KB
Lesson15_RX_nRF2401_Wireless_RF_Mod...	2026/3/11 19:26	INO File	14 KB
lv_conf.h	2026/3/10 18:24	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/27 11:51	C++ 源文件	31 KB
lvgl_v8_port.h	2026/3/10 16:33	C Header 源文件	7 KB

Here, we will still use the bsp_wireless component from the previous lesson.

The main functions of this component are as follows:

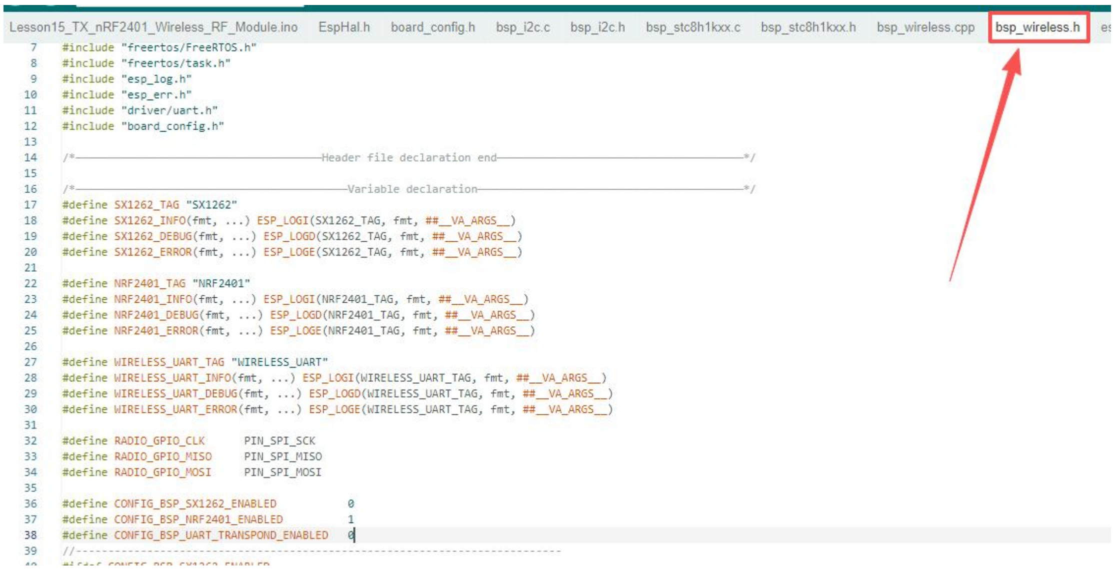
- It is responsible for encoding and modulating data sent by the main controller (such as strings, sensor information, etc.) before transmitting it.
- It also handles the reception of wireless data packets sent by other devices via the nRF2401.
- It returns the received data to the upper-layer application through a callback mechanism.

In addition to the aforementioned functions, we have also encapsulated the relevant experimental functions of the remaining three wireless modules - nRF2401, LoRa module, ESP32-C6, and ESP32-H2 - into this component.

Since in the code, the function usage of each wireless module is wrapped with ifdef and endif, and we are using the nRF2401 wireless module in this lesson, we only need to enable the configurations related to nRF2401.

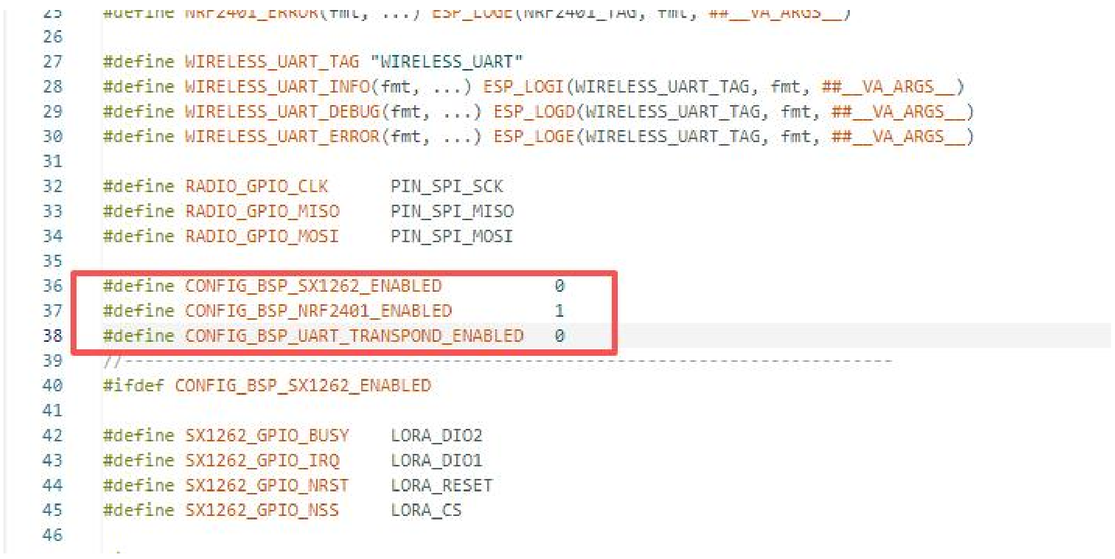
How to enable it:

Open the bsp_wireless.h file.



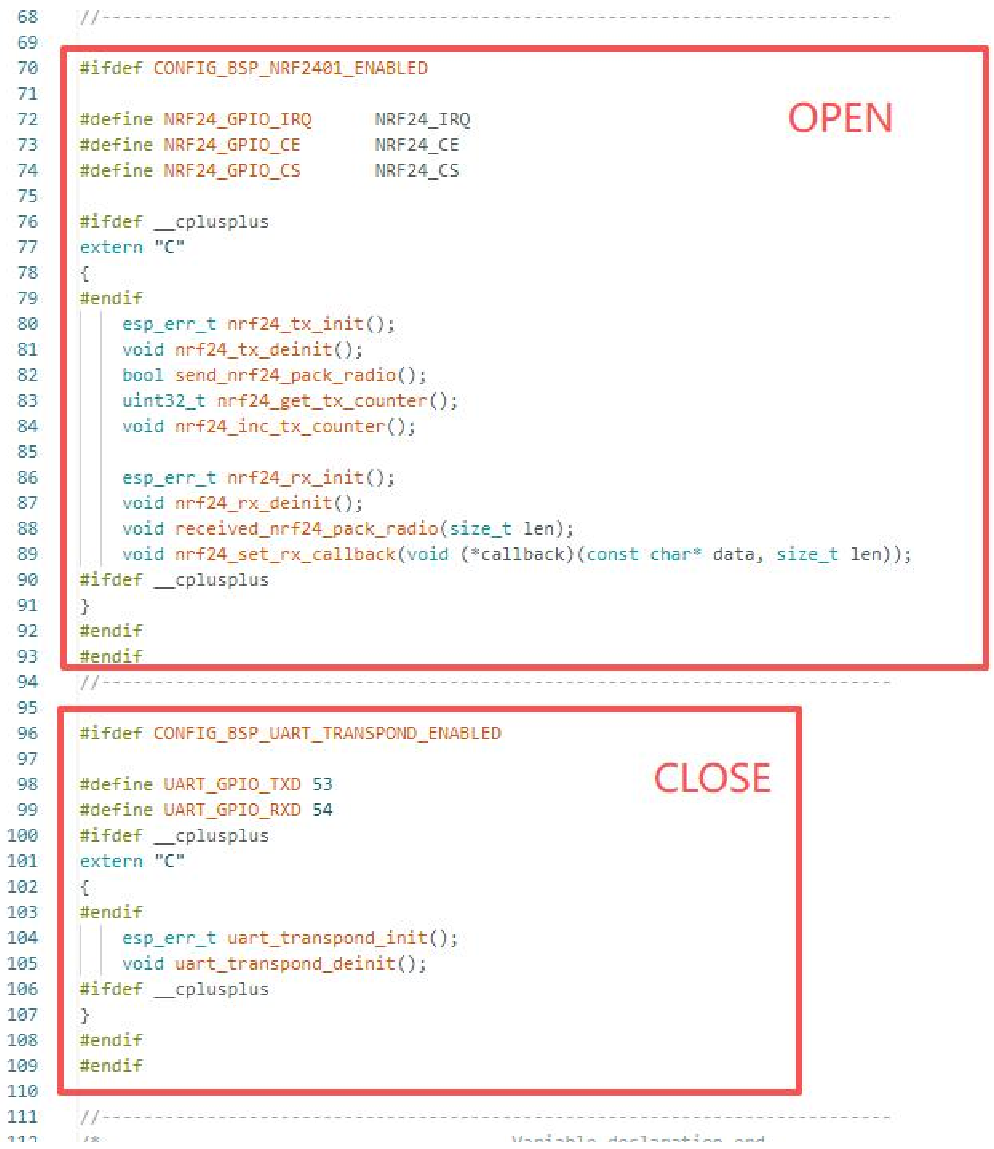
```c
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/uart.h"
#include "board_config.h"

/*————————————————————————————————Header file declaration end————————————————————————————————*/

/*————————————————————————————————————Variable declaration————————————————————————————————————*/
#define SX1262_TAG "SX1262"
#define SX1262_INFO(fmt, ...) ESP_LOGI(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_DEBUG(fmt, ...) ESP_LOGD(SX1262_TAG, fmt, ##__VA_ARGS__)
#define SX1262_ERROR(fmt, ...) ESP_LOGE(SX1262_TAG, fmt, ##__VA_ARGS__)

#define NRF2401_TAG "NRF2401"
#define NRF2401_INFO(fmt, ...) ESP_LOGI(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_DEBUG(fmt, ...) ESP_LOGD(NRF2401_TAG, fmt, ##__VA_ARGS__)
#define NRF2401_ERROR(fmt, ...) ESP_LOGE(NRF2401_TAG, fmt, ##__VA_ARGS__)

#define WIRELESS_UART_TAG "WIRELESS_UART"
#define WIRELESS_UART_INFO(fmt, ...) ESP_LOGI(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_DEBUG(fmt, ...) ESP_LOGD(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_ERROR(fmt, ...) ESP_LOGE(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)

#define RADIO_GPIO_CLK      PIN_SPI_SCK
#define RADIO_GPIO_MISO     PIN_SPI_MISO
#define RADIO_GPIO_MOSI     PIN_SPI_MOSI

#define CONFIG_BSP_SX1262_ENABLED          0
#define CONFIG_BSP_NRF2401_ENABLED         1
#define CONFIG_BSP_UART_TRANSPOND_ENABLED  0
//-------------------------------------------------------------------------
```

Here, enable the macro definition of the nRF2401 wireless module by setting the macro definition to 1. From now on, you will use the code related to nRF2401, while the macro definitions for other wireless modules will be commented out by default—that is, we set the macro definitions for other wireless modules to 0.



```
#define NRF2401_ERROR(fmt, ...) ESP_LOGE(NRF2401_TAG, fmt, ##__VA_ARGS__)

#define WIRELESS_UART_TAG "WIRELESS_UART"
#define WIRELESS_UART_INFO(fmt, ...) ESP_LOGI(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_DEBUG(fmt, ...) ESP_LOGD(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_ERROR(fmt, ...) ESP_LOGE(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)

#define RADIO_GPIO_CLK      PIN_SPI_SCK
#define RADIO_GPIO_MISO     PIN_SPI_MISO
#define RADIO_GPIO_MOSI     PIN_SPI_MOSI

#define CONFIG_BSP_SX1262_ENABLED           0
#define CONFIG_BSP_NRF2401_ENABLED          1
#define CONFIG_BSP_UART_TRANSPOND_ENABLED   0
//-----------------------------------------------------------------------------
#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY    LORA_DIO2
#define SX1262_GPIO_IRQ     LORA_DIO1
#define SX1262_GPIO_NRST    LORA_RESET
#define SX1262_GPIO_NSS     LORA_CS

```

(Enable the one that corresponds to the wireless module you are using.)



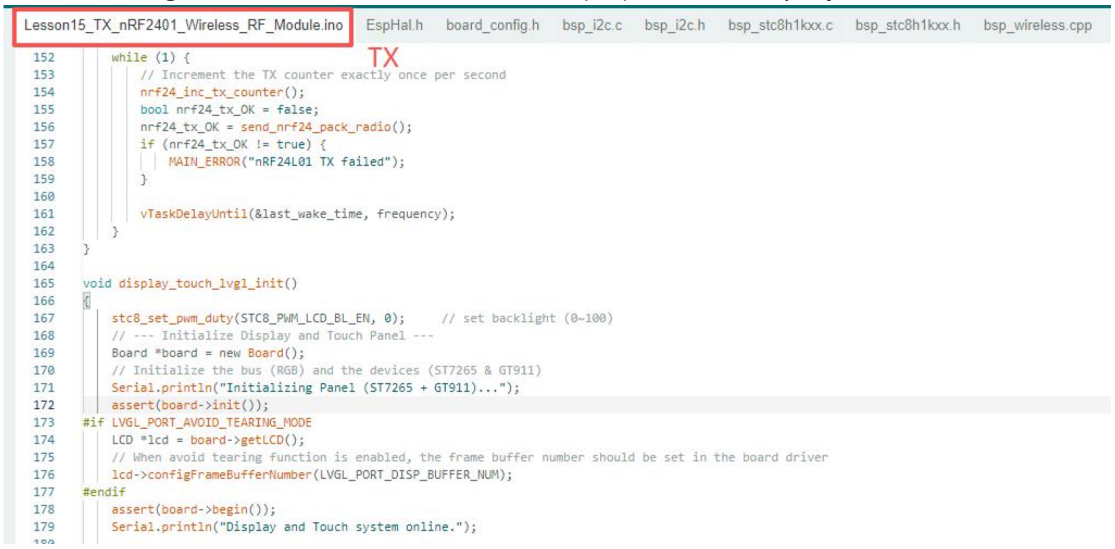
```
//-----------------------------------------------------------------------------

#ifdef CONFIG_BSP_NRF2401_ENABLED

#define NRF24_GPIO_IRQ      NRF24_IRQ
#define NRF24_GPIO_CE       NRF24_CE
#define NRF24_GPIO_CS       NRF24_CS

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t nrf24_tx_init();
    void nrf24_tx_deinit();
    bool send_nrf24_pack_radio();
    uint32_t nrf24_get_tx_counter();
    void nrf24_inc_tx_counter();

    esp_err_t nrf24_rx_init();
    void nrf24_rx_deinit();
    void received_nrf24_pack_radio(size_t len);
    void nrf24_set_rx_callback(void (*callback)(const char* data, size_t len));
#ifdef __cplusplus
}
#endif
#endif
//-----------------------------------------------------------------------------

#ifdef CONFIG_BSP_UART_TRANSPOND_ENABLED

#define UART_GPIO_TXD 53
#define UART_GPIO_RXD 54
#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t uart_transpond_init();
    void uart_transpond_deinit();
#ifdef __cplusplus
}
#endif
#endif

//-----------------------------------------------------------------------------
/*                              Variable declaration and
```

As shown in the figure, we have enabled the nRF2401 configuration, so the other modules are temporarily disabled and not applicable.

In the bsp_wireless component, you only need to call the prepared interfaces when needed.

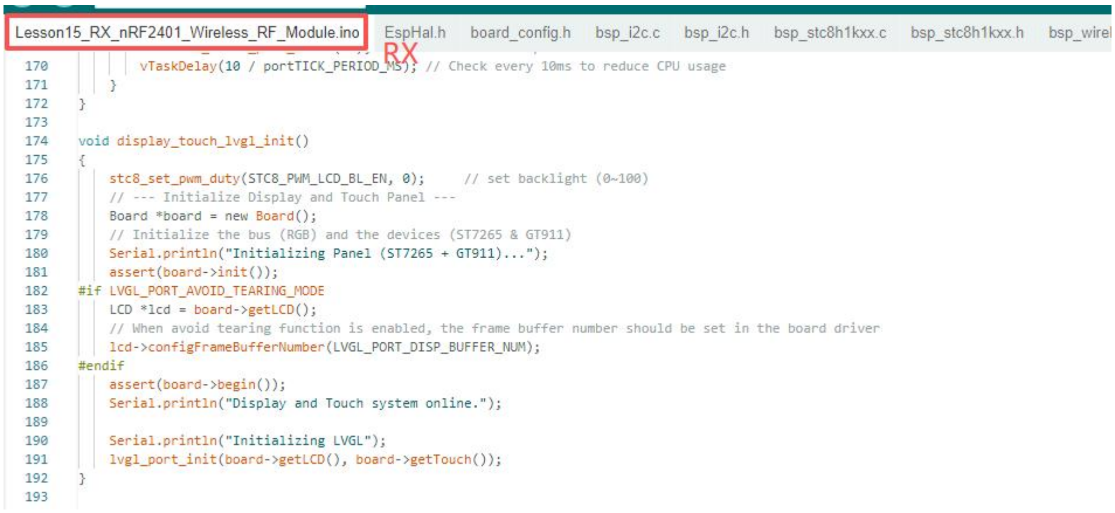
Next, let's focus on understanding the bsp_wireless component.

The following section shows the transmitter (TX) side of the project:



```
        while (1) {
            // Increment the TX counter exactly once per second
            nrf24_inc_tx_counter();
            bool nrf24_tx_OK = false;
            nrf24_tx_OK = send_nrf24_pack_radio();
            if (nrf24_tx_OK != true) {
                MAIN_ERROR("nRF24L01 TX failed");
            }

            vTaskDelayUntil(&last_wake_time, frequency);
        }
}

void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);     // set backlight (0~100)
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
    assert(board->begin());
    Serial.println("Display and Touch system online.");
```

The following section shows the receiver (RX) side of the project:



```
            vTaskDelay(10 / portTICK_PERIOD_MS); // Check every 10ms to reduce CPU usage
        }
}

void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);     // set backlight (0~100)
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
    assert(board->begin());
    Serial.println("Display and Touch system online.");

    Serial.println("Initializing LVGL");
    lvgl_port_init(board->getLCD(), board->getTouch());
}
```

In these two projects, the only difference lies in the main functions: Lesson15_TX_nRF2401_Wireless_RF_Module.ino for the transmitter and Lesson15_RX_nRF2401_Wireless_RF_Module.ino for the receiver.

All other code files are identical. (For convenience, we have prepared both main functions for you to use separately.)

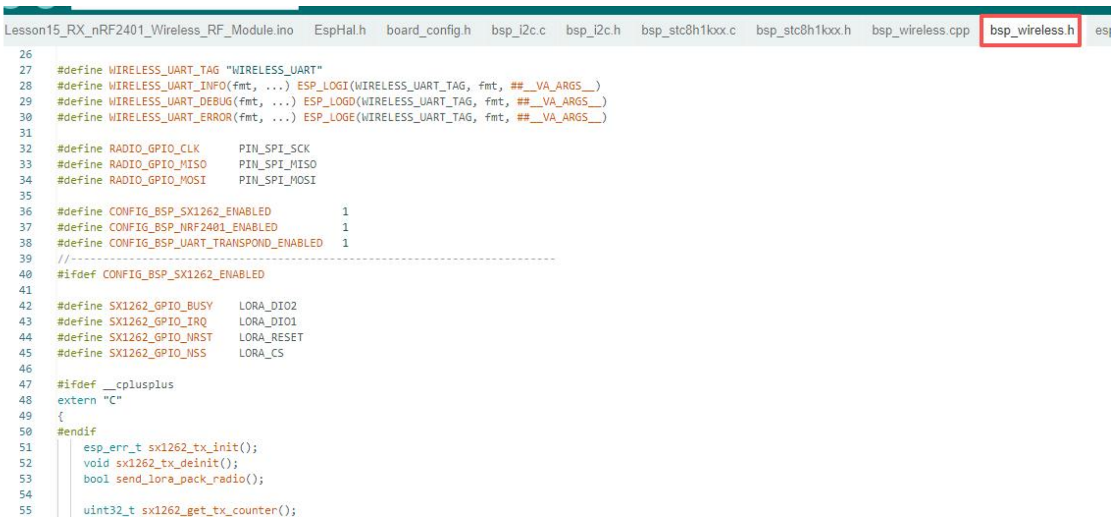
The other library files under this project have been elaborately explained in the previous class. Therefore, we will not go into too much detail about them here.

nRF2401 Communication Code

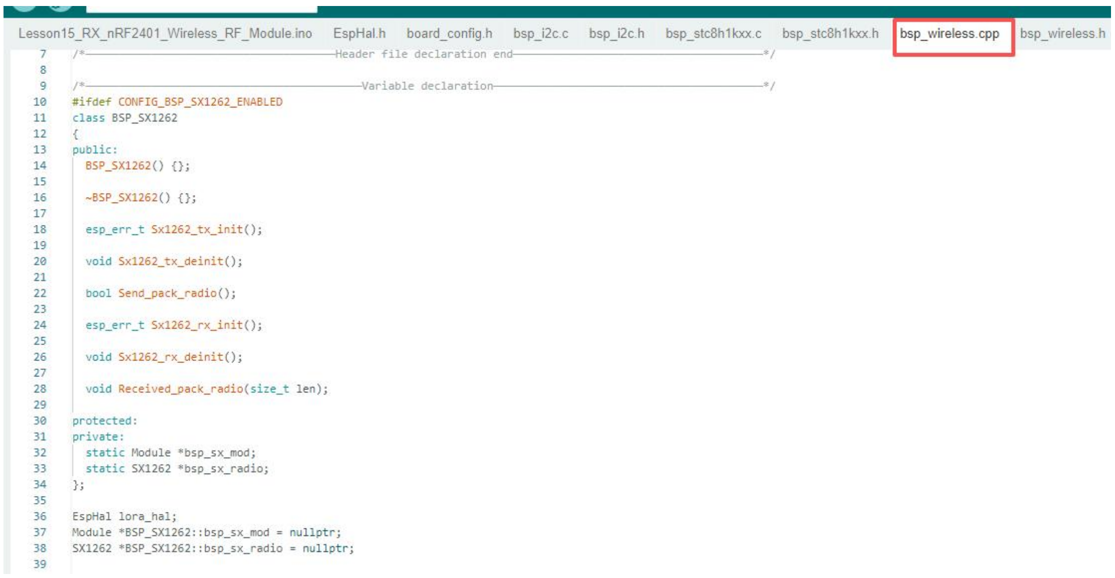
The code for nRF2401 transmission and reception consists of two files: "bsp_wireless.cpp" and "bsp_wireless.h".
Next, we will first analyze the nRF2401-related code in the "bsp_wireless.h" program. "bsp_wireless.h" is the header file for the nRF2401 wireless module, primarily used to:

- Declare functions, macros, and variables implemented in "bsp_wireless.cpp" for use by external programs.
- Allow other .c files to call this module simply by adding #include "bsp_wireless.h".

In other words, it acts as an interface layer that exposes which functions and constants are available to the outside, while hiding the internal details of the module.
In this component, the libraries we need to use are placed in the "bsp_wireless.h" and "bsp_wireless.cpp" files.



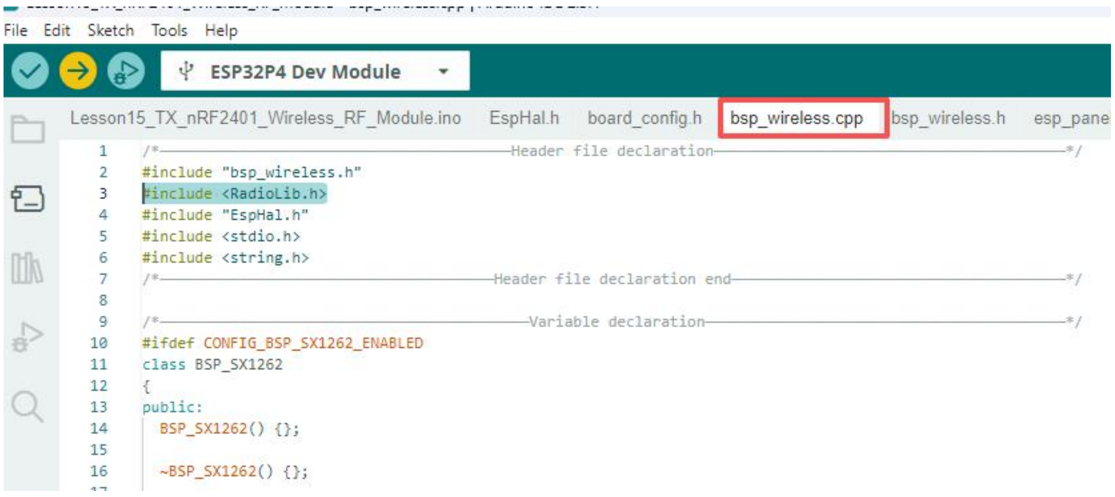
```
#define WIRELESS_UART_TAG "WIRELESS_UART"
#define WIRELESS_UART_INFO(fmt, ...) ESP_LOGI(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_DEBUG(fmt, ...) ESP_LOGD(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
#define WIRELESS_UART_ERROR(fmt, ...) ESP_LOGE(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)

#define RADIO_GPIO_CLK      PIN_SPI_SCK
#define RADIO_GPIO_MISO     PIN_SPI_MISO
#define RADIO_GPIO_MOSI     PIN_SPI_MOSI

#define CONFIG_BSP_SX1262_ENABLED          1
#define CONFIG_BSP_NRF2401_ENABLED         1
#define CONFIG_BSP_UART_TRANSPOND_ENABLED  1
//---------------------------------------------------------------------------
#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY    LORA_DIO2
#define SX1262_GPIO_IRQ     LORA_DIO1
#define SX1262_GPIO_NRST    LORA_RESET
#define SX1262_GPIO_NSS     LORA_CS

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t sx1262_tx_init();
    void sx1262_tx_deinit();
    bool send_lora_pack_radio();

    uint32_t sx1262_get_tx_counter();
```



```
/*------------------------------Header file declaration end------------------------------*/

/*--------------------------------Variable declaration--------------------------------*/
#ifdef CONFIG_BSP_SX1262_ENABLED
class BSP_SX1262
{
public:
  BSP_SX1262() {};

  ~BSP_SX1262() {};

  esp_err_t Sx1262_tx_init();

  void Sx1262_tx_deinit();

  bool Send_pack_radio();

  esp_err_t Sx1262_rx_init();

  void Sx1262_rx_deinit();

  void Received_pack_radio(size_t len);

protected:
private:
  static Module *bsp_sx_mod;
  static SX1262 *bsp_sx_radio;
};

EspHal lora_hal;
Module *BSP_SX1262::bsp_sx_mod = nullptr;
SX1262 *BSP_SX1262::bsp_sx_radio = nullptr;
```

Since the function implementations in bsp_wireless.cpp use the function wrappers

provided in EspHal.h, the header file needs to be included in the .cpp file.
For example, #include <RadioLib.h> (this is a library under the networking components)



```
/*————————————————————Header file declaration————————————————————*/
#include "bsp_wireless.h"
#include <RadioLib.h>
#include "EspHal.h"
#include <stdio.h>
#include <string.h>
/*————————————————————Header file declaration end————————————————————*/

/*————————————————————Variable declaration————————————————————*/
#ifdef CONFIG_BSP_SX1262_ENABLED
class BSP_SX1262
{
public:
  BSP_SX1262() {};

  ~BSP_SX1262() {};
```

Returning to bsp_wireless.h,
this is where we declare the pins used by the wireless module.

```
#define RADIO_GPIO_CLK      PIN_SPI_SCK
#define RADIO_GPIO_MISO     PIN_SPI_MISO
#define RADIO_GPIO_MOSI     PIN_SPI_MOSI

```

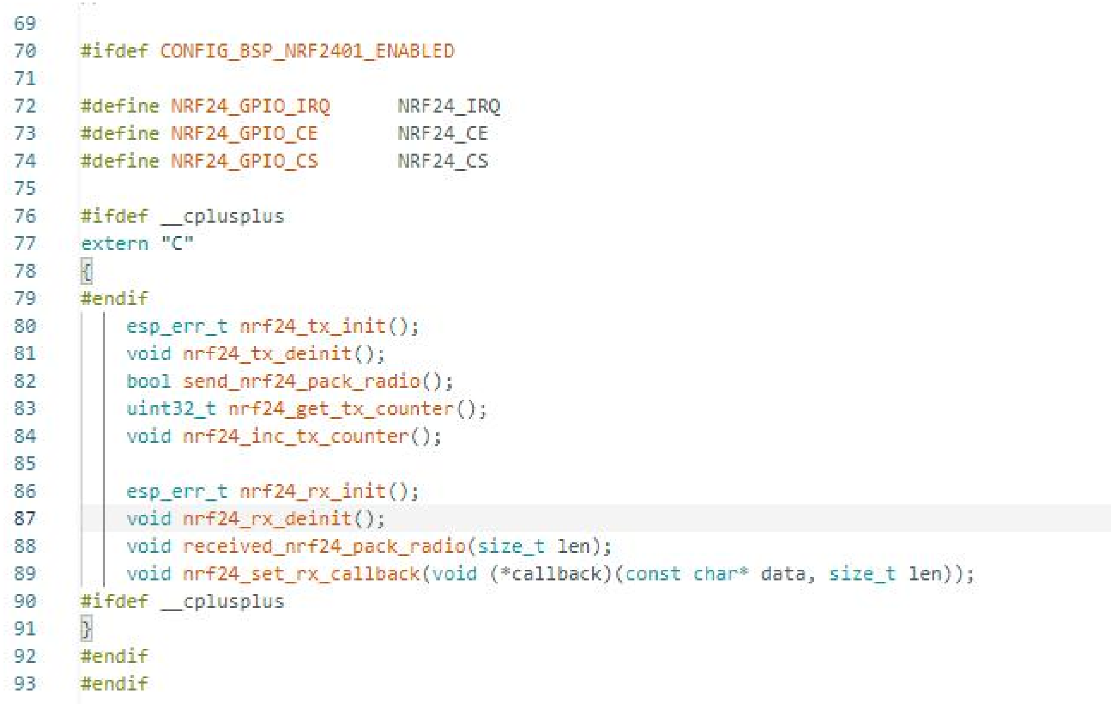
```

#define NRF24_GPIO_IRQ      NRF24_IRQ
#define NRF24_GPIO_CE       NRF24_CE
#define NRF24_GPIO_CS       NRF24_CS

```

The specific pin numbers are defined in the file "board_config.h".

```
/*wireless module GPIO pins*/
/* SPI BUS */
#define PIN_SPI_SCK         (26)
#define PIN_SPI_MOSI        (48)
#define PIN_SPI_MISO        (47)
/* LoRa Interfaces */
#define LORA_RESET          (32)    // RST
#define LORA_DIO1           (31)    // IRQ
#define LORA_DIO2           (29)    // BUSY
#define LORA_CS             (30)
/* nRF2401 Interfaces */
#define NRF24_IRQ           (29)
#define NRF24_CE            (31)
#define NRF24_CS            (32)
/*wireless module GPIO pins*/
```

The pin definitions should not be modified; otherwise, the wireless module will not function correctly due to incorrect wiring.

Next, we declare the variables we need to use, as well as the functions. The actual implementations of these functions are in bsp_wireless.cpp. Placing all declarations in bsp_wireless.h is intended to make them easier to call and manage. (We will examine their specific functionality when they are used in bsp_wireless.cpp.)



```
#ifdef CONFIG_BSP_NRF2401_ENABLED

#define NRF24_GPIO_IRQ      NRF24_IRQ
#define NRF24_GPIO_CE       NRF24_CE
#define NRF24_GPIO_CS       NRF24_CS

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t nrf24_tx_init();
    void nrf24_tx_deinit();
    bool send_nrf24_pack_radio();
    uint32_t nrf24_get_tx_counter();
    void nrf24_inc_tx_counter();

    esp_err_t nrf24_rx_init();
    void nrf24_rx_deinit();
    void received_nrf24_pack_radio(size_t len);
    void nrf24_set_rx_callback(void (*callback)(const char* data, size_t len));
#ifdef __cplusplus
}
#endif
#endif
```

Next, let's take a look at the specific functionality of each function in bsp_wireless.cpp.
In the bsp_wireless component, BSP_NRF2401 is a BSP driver wrapper class for the nRF24L01 wireless transceiver module. It provides initialization, execution, de-initialization, and callback mechanisms for sending and receiving.

This allows the application layer to complete wireless communication simply by calling straightforward C interface functions (such as nrf24_tx_init() or send_nrf24_pack_radio()), without needing to directly manipulate the underlying SPI registers or the RadioLib interface.
Here, we won't go into a detailed code walkthrough; we will only explain the purpose of each function and the situations in which it should be called.

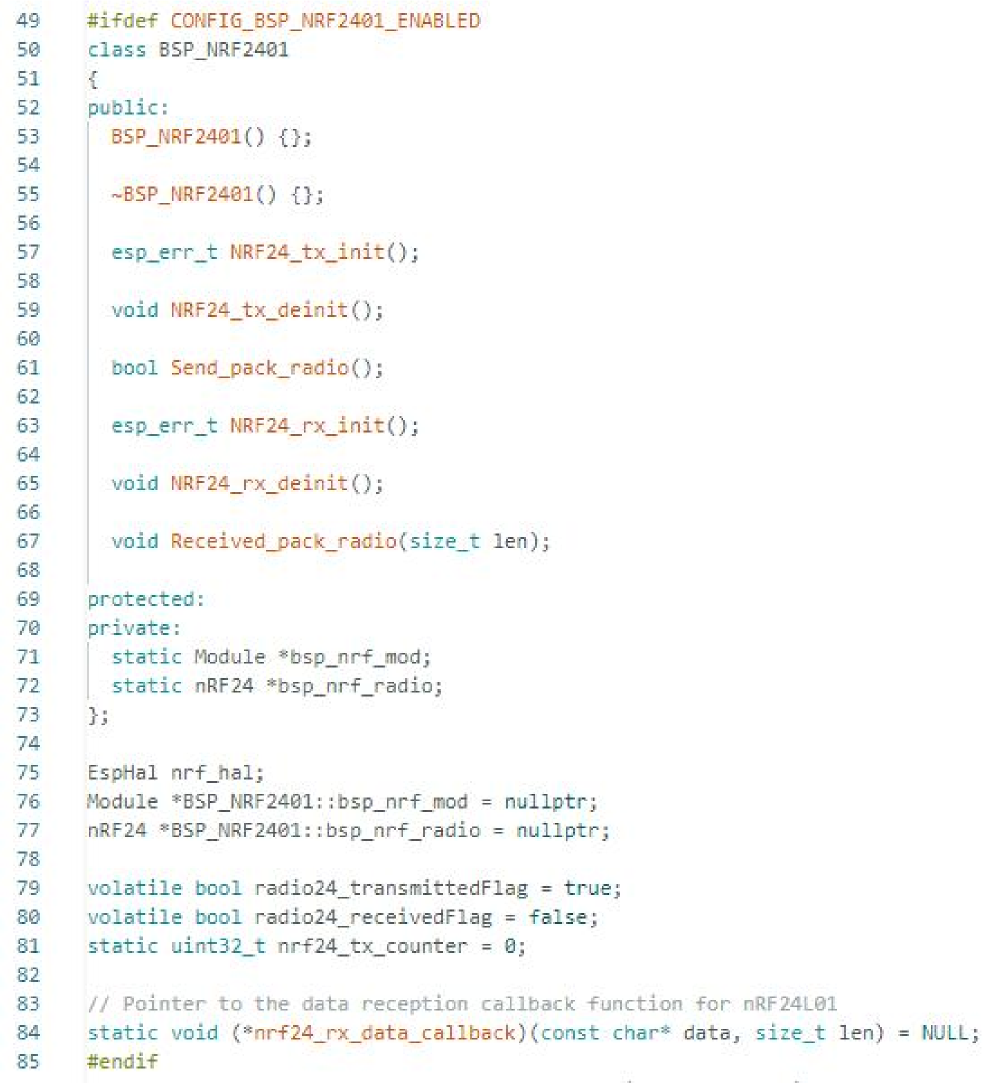
BSP_NRF2401 Class:

This means:
This code defines a class named BSP_NRF2401 to encapsulate the driver logic for the nRF2401 wireless transceiver module, implementing initialization, sending, and receiving functionalities for wireless communication.

- The class declares initialization and de-initialization functions for both the transmitter and receiver (such as NRF24_tx_init, NRF24_rx_init), as well as data sending and receiving handling functions (Send_pack_radio, Received_pack_radio).
- Two static pointers, bsp_nrf_mod and bsp_nrf_radio, are defined to point to the underlying hardware module object and the radio object, respectively, allowing global sharing.

- nrf_hal is the hardware abstraction layer object, used to manage hardware communication with the chip.
- Two volatile variables are defined: radio24_transmittedFlag indicates whether transmission is complete, and radio24_receivedFlag indicates whether reception is complete.
- nrf24_tx_counter is used to record the number of transmissions.
- Finally, a function pointer nrf24_rx_data_callback is defined to trigger an upper-layer callback when data is received.

Overall, this code establishes the basic control framework for the nRF2401 module, providing a unified interface and state management mechanism for subsequent wireless data transmission and reception.



```
#ifdef CONFIG_BSP_NRF2401_ENABLED
class BSP_NRF2401
{
public:
  BSP_NRF2401() {};

  ~BSP_NRF2401() {};

  esp_err_t NRF24_tx_init();

  void NRF24_tx_deinit();

  bool Send_pack_radio();

  esp_err_t NRF24_rx_init();

  void NRF24_rx_deinit();

  void Received_pack_radio(size_t len);

protected:
private:
  static Module *bsp_nrf_mod;
  static nRF24 *bsp_nrf_radio;
};

EspHal nrf_hal;
Module *BSP_NRF2401::bsp_nrf_mod = nullptr;
nRF24 *BSP_NRF2401::bsp_nrf_radio = nullptr;

volatile bool radio24_transmittedFlag = true;
volatile bool radio24_receivedFlag = false;
static uint32_t nrf24_tx_counter = 0;

// Pointer to the data reception callback function for nRF24L01
static void (*nrf24_rx_data_callback)(const char* data, size_t len) = NULL;
#endif
```

NRF24_tx_init:

Initializes the transmitter of the nRF2401 module by configuring the SPI interface, creating the communication object, setting the wireless parameters, and specifying the transmission channel, enabling the module to send data.

- At the beginning of the function, nrf_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI) sets the SPI communication pins between the nRF2401 and the main controller (Clock, Master In Slave Out, Master Out Slave In).

- setSpiFrequency(8000000) sets the SPI clock frequency to 8 MHz to improve communication speed.
- spiBegin() formally initializes the SPI bus.
- A module object bsp_nrf_mod is then created via new Module(...), binding the SPI interface along with control pins such as Chip Select (CS), Interrupt (IRQ), and Chip Enable (CE), providing a hardware interface for the nRF24 module.
- Next, bsp_nrf_radio = new nRF24(bsp_nrf_mod) creates the specific nRF24 radio object and begins the driver logic.
- Calling begin(2400, 250, 0, 5) completes the core initialization of the wireless module. The parameters represent, in order: operating frequency 2400 MHz (i.e., 2.4 GHz band), data rate 250 kbps, output power level 0 (typically 0 dBm), and communication channel number 5. If initialization fails (return value is not RADIOLIB_ERR_NONE), the error is logged and the function exits.
- Then, a transmit address is defined as uint8_t addr[] = {0x01, 0x02, 0x11, 0x12, 0xFF}, which is a 5-byte transmit pipe address (similar to a "device address" or "channel identifier" in wireless communication), ensuring that the transmitter and receiver communicate over the same address.
- setTransmitPipe(addr) sets this address as the current transmit pipe, allowing the module to send data through this channel. If configured successfully, the function returns ESP_OK, indicating that initialization is complete.

Send_pack_radio:

This function sends a wireless data packet through the nRF2401 module and records and prints the transmission status.

- Specifically, the function first defines a static character array text[32] to store the message to be sent. It then uses snprintf to format the message as "NRF24_TX_Hello World:<transmit_count>", where <transmit_count> comes from nrf24_tx_counter and represents the current number of transmissions.
- The function calculates the message length using strlen and stores it in tx_len.
- Next, it calls bsp_nrf_radio->transmit((uint8_t *)text, tx_len, 0) to send the message through the nRF2401 module. If the return value is RADIOLIB_ERR_NONE, the transmission is successful, and NRF2401_INFO prints the completion message along with the content sent. Otherwise, it prints a transmission failure message and the error code.
- The function finally returns true, indicating that the send operation has been executed.

nrf24_tx_init():

This is a C-language interface function used to initialize the nRF2401 transmitter module. Inside the function, a BSP_NRF2401 object obj is instantiated, and its member function NRF24_tx_init() is called to complete SPI configuration, wireless

parameter setup, and transmit pipe address configuration, returning the initialization result.
Purpose: Provides a unified interface for upper-layer or C code to prepare the nRF2401 module for data transmission.

nrf24_tx_deinit():

This is a C-language interface function used to release or shut down the nRF2401 transmitter resources. It creates a BSP_NRF2401 object internally and calls its member function NRF24_tx_deinit(), putting the wireless module into an idle state and closing the SPI bus.
Purpose: Called when the transmission task is finished or the module is no longer in use, safely releasing transmitter resources.

send_nrf24_pack_radio():

This is a C-language interface function used to send a data packet via the nRF2401. Internally, it creates a BSP_NRF2401 object and calls its member function Send_pack_radio() to send the formatted message and print the transmission result.
Purpose: Provides a simple interface for the upper layer to send wireless data without needing to handle the underlying driver details.

nrf24_get_tx_counter():

This is a C-language interface function used to get the current value of the nRF2401 transmit counter nrf24_tx_counter.
Purpose: Allows upper-layer programs to obtain the number of packets sent, useful for statistics or debugging.

nrf24_inc_tx_counter():

This is a C-language interface function used to increment the transmit counter nrf24_tx_counter by 1.
Purpose: Updates the counter after each successful packet transmission, used to record the number of sends or to mark a sequence number in the message.

set_rx_flag():

This is a static internal function called within the receive interrupt or callback, used to set radio24_receivedFlag to true, indicating that the nRF2401 module has received new data.
Purpose: Serves as a receive event flag to notify the upper-layer program that new data is available for processing.

NRF24_rx_init:

This function, **BSP_NRF2401::NRF24_rx_init()**, initializes the receiver side of the nRF2401 module, enabling it to receive wireless data.

- Specifically, the function first sets the SPI communication pins using nrf_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI), sets the SPI clock frequency to 8 MHz with setSpiFrequency(8000000), and initializes the SPI bus using spiBegin().
- A module object bsp_nrf_mod is then created via new Module(...), binding the SPI interface and control pins. Next, bsp_nrf_radio = new nRF24(bsp_nrf_mod) creates the nRF24 radio object.
- Calling bsp_nrf_radio->begin(2400, 250, 0, 5) initializes the wireless parameters, where 2400 represents the 2.4 GHz operating frequency, 250 is the data rate in kbps, 0 is the output power level, and 5 is the communication channel. If an error occurs, it logs the failure and returns.
- A receive pipe address is defined as addr[] = {0x01, 0x02, 0x11, 0x12, 0xFF}. The function then calls setReceivePipe(0, addr) to set pipe 0 as the receive address, ensuring the module only receives data sent to this address.
- setPacketReceivedAction(set_rx_flag) registers a receive callback, setting radio24_receivedFlag to notify the upper layer. Finally, startReceive() puts the module into receive mode. If successful, the function returns ESP_OK.

Received_pack_radio:

This function, **BSP_NRF2401::Received_pack_radio(size_t len)**, handles data packets received by the nRF2401 module.

- Specifically, the function first checks the receive flag radio24_receivedFlag. If it is true, it indicates that new data has arrived. The flag is then reset to false to avoid repeated processing.
- A buffer data[len] is defined to store the received data, and bsp_nrf_radio->readData(data, len) is called to read len bytes from the module.
- If the return value is RADIOLIB_ERR_NONE, the data is successfully read. The function uses NRF2401_INFO to print a success message along with the received data, and checks whether the callback function pointer nrf24_rx_data_callback has been registered. If it is registered, the callback is called to notify the upper-layer application.
- If reading fails, NRF2401_ERROR prints the error code. Finally, bsp_nrf_radio->startReceive() is called to re-enter receive mode, waiting for the next data packet.

nrf24_rx_init()

This is a C-language interface function used to initialize the receiver side of the nRF2401 module. Internally, a BSP_NRF2401 object obj is instantiated, and its member function NRF24_rx_init() is called to complete SPI configuration, wireless parameter initialization, receive pipe address setup, and callback registration, returning the initialization result.
Purpose: Provides a unified interface for upper-layer or C-language programs to prepare the nRF2401 module for data reception.

nrf24_rx_deinit()

This is a C-language interface function used to release the nRF2401 receiver resources. Internally, a BSP_NRF2401 object is created, and its member function NRF24_rx_deinit() is called to put the module into an idle state, clear callbacks, and close the SPI bus.
Purpose: Called when the reception task is finished or the module is no longer in use, safely releasing receiver resources.

received_nrf24_pack_radio(size_t len)

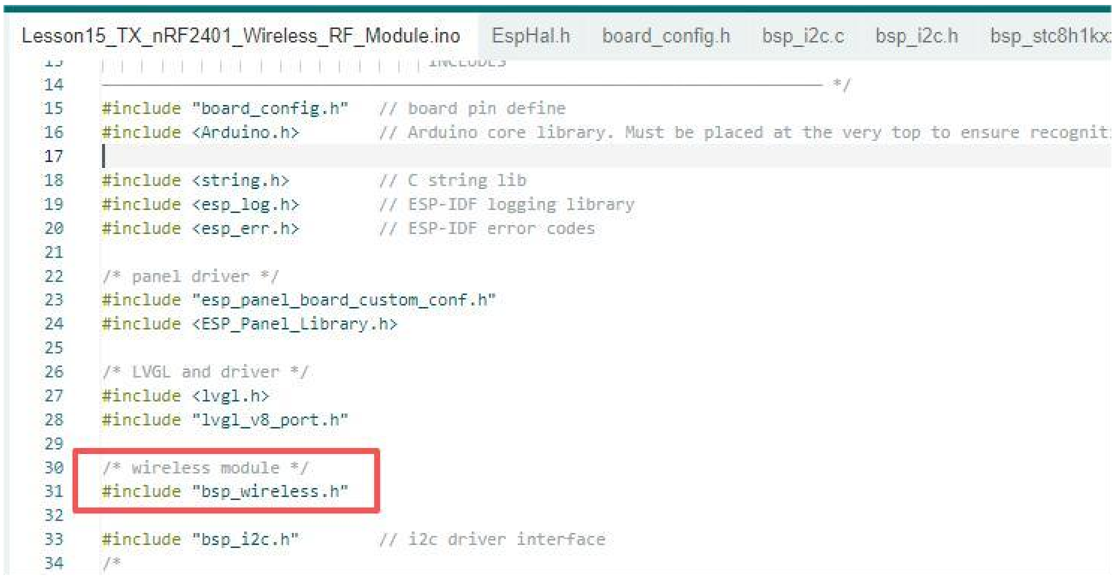
This is a C-language interface function used to handle received data packets. Internally, it creates a BSP_NRF2401 object and calls its member function Received_pack_radio(len) to read the data, log the results, and notify the upper-layer application via a callback.
Purpose: Provides an upper-layer interface to trigger the nRF2401 data reception processing flow.

nrf24_set_rx_callback(void (*callback)(const char* data, size_t len))

This is a C-language interface function used to register a callback for received data, notifying the upper-layer application when data arrives. Internally, the passed function pointer is saved to nrf24_rx_data_callback.
Purpose: Allows the upper-layer program to set a custom callback for immediate processing or response upon receiving nRF2401 data.

We will conclude the introduction of the bsp_wireless component here. It is enough for everyone to understand how to call these interfaces.

If you want to use these functions, all you need to do is to fill in the library file in the function code.



```
/*───────────────────────────────────────────────── INCLUDES
──────────────────────────────────────────────────────── */
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognit

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes

/* panel driver */
#include "esp_panel_board_custom_conf.h"
#include <ESP_Panel_Library.h>

/* LVGL and driver */
#include <lvgl.h>
#include "lvgl_v8_port.h"

/* wireless module */
#include "bsp_wireless.h"

#include "bsp_i2c.h"        // i2c driver interface
/*
```

Sender Main Function

Let’s first explain the transmitter’s main function file, Lesson15_TX_nRF2401_Wireless_RF_Module.ino, to see how it calls the interfaces to send information via the nRF2401.

When the program runs, the general workflow is as follows:

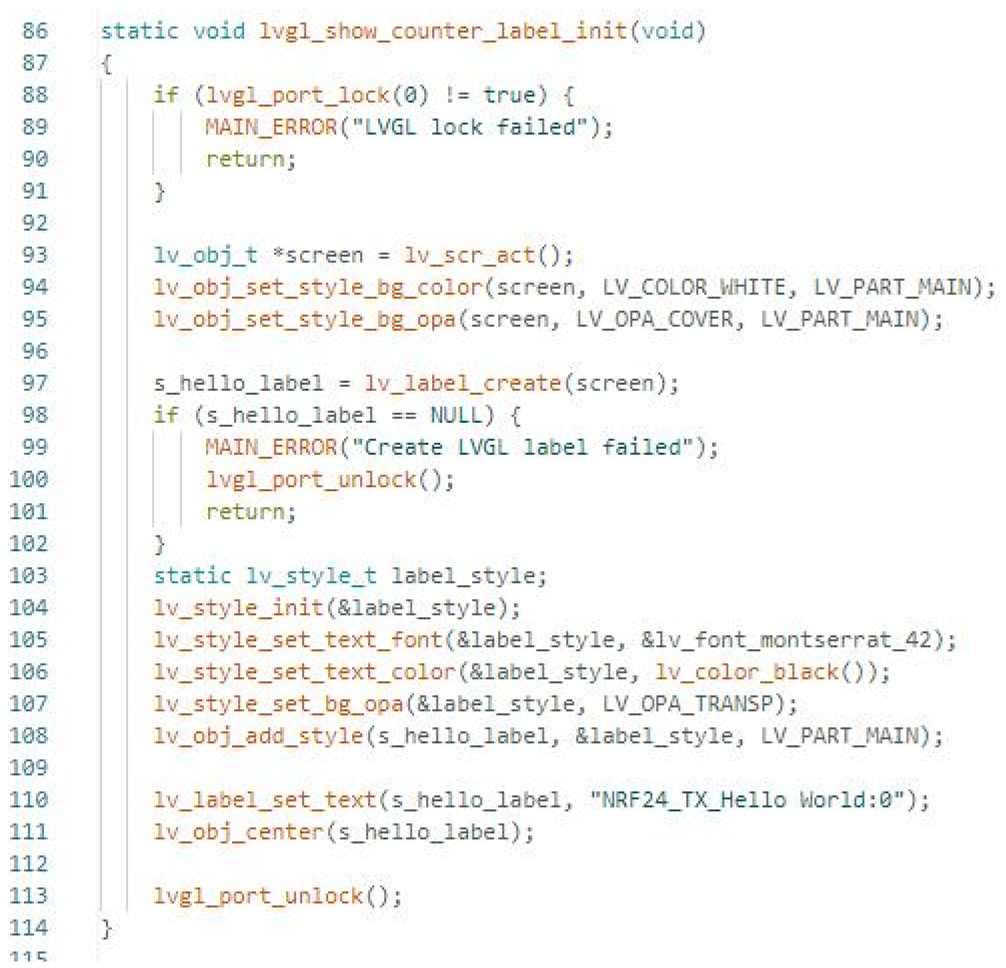
The overall workflow can be summarized as follows: power on the system → initialize the serial port → initialize the display, touch screen and LVGL graphics system → initialize the nRF24L01 wireless module → create interface labels → create two parallel tasks (one for wireless transmission and one for interface update) → send wireless data once per second and display the number of transmissions in real time on the screen. This constitutes an embedded demonstration system that combines wireless transmission and real-time display of graphical interface status.

lvgl_show_counter_label_init:

The function lvgl_show_counter_label_init() initializes the counter label on the LVGL display, used to show the nRF24L01 transmit count. Its workflow and purpose of each step can be summarized as follows:

- First, lvgl_port_lock(0) is called to lock LVGL resources, preventing concurrent access.
- The current active screen is obtained via lv_scr_act(), and the background is set to white and fully covering.
- A label is created using lv_label_create(screen) and checked for successful creation; if creation fails, the lock is released and the function returns.
- The label style is initialized with lv_style_init, setting the font size, text color to black, and background to transparent, and the style is applied to the label.
- lv_label_set_text sets the initial text to "NRF24_TX_Hello World:0", and lv_obj_center centers the label on the screen.
- Finally, lvgl_port_unlock() releases the LVGL resource lock.

Overall, this function creates and initializes a styled, dynamically updatable label to display the transmit count.



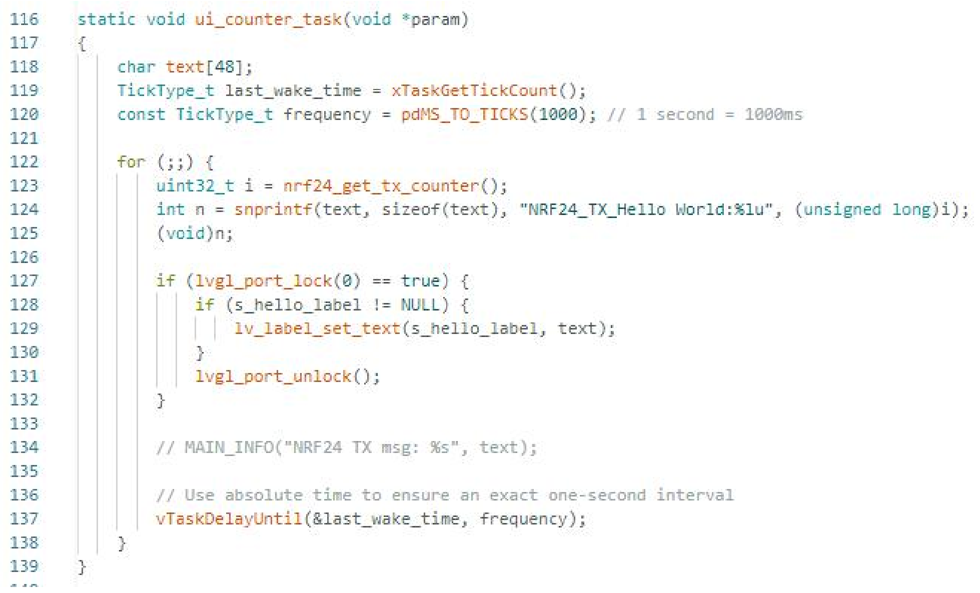
```
static void lvgl_show_counter_label_init(void)
{
    if (lvgl_port_lock(0) != true) {
        MAIN_ERROR("LVGL lock failed");
        return;
    }

    lv_obj_t *screen = lv_scr_act();
    lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);
    lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);

    s_hello_label = lv_label_create(screen);
    if (s_hello_label == NULL) {
        MAIN_ERROR("Create LVGL label failed");
        lvgl_port_unlock();
        return;
    }
    static lv_style_t label_style;
    lv_style_init(&label_style);
    lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
    lv_style_set_text_color(&label_style, lv_color_black());
    lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
    lv_obj_add_style(s_hello_label, &label_style, LV_PART_MAIN);

    lv_label_set_text(s_hello_label, "NRF24_TX_Hello World:0");
    lv_obj_center(s_hello_label);

    lvgl_port_unlock();
}
```

ui_counter_task:

The function ui_counter_task() is responsible for refreshing the nRF24L01 transmission count information displayed on the LCD every second.
Its workflow and the role of each part can be summarized as follows:

- First, define a character array text[48] to store the display text.
- Record the system tick count last_wake_time when the task starts, and set the loop interval to 1000ms (1 second).
- Enter an infinite loop. In each loop, first read the current transmission count using nrf24_get_tx_counter(), and format it into the string "NRF24_TX_Hello World:<count value>" using snprintf.
- Attempt to lock the LVGL resource with lvgl_port_lock(0). If successful and the label exists, call lv_label_set_text to update the display text and release the lock.
- Finally, use vTaskDelayUntil to delay according to absolute time to ensure an accurate one-second cycle, realizing the function of updating the display every second.

Overall, its role is to continuously refresh the transmission count on the interface to achieve real-time display.



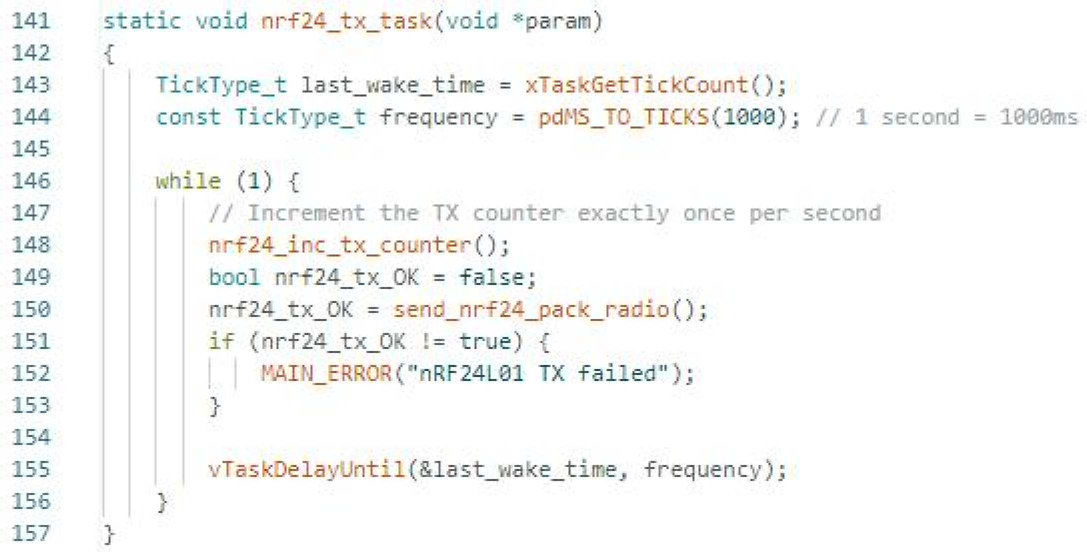
```c
static void ui_counter_task(void *param)
{
    char text[48];
    TickType_t last_wake_time = xTaskGetTickCount();
    const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms

    for (;;) {
        uint32_t i = nrf24_get_tx_counter();
        int n = snprintf(text, sizeof(text), "NRF24_TX_Hello World:%lu", (unsigned long)i);
        (void)n;

        if (lvgl_port_lock(0) == true) {
            if (s_hello_label != NULL) {
                lv_label_set_text(s_hello_label, text);
            }
            lvgl_port_unlock();
        }

        // MAIN_INFO("NRF24 TX msg: %s", text);

        // Use absolute time to ensure an exact one-second interval
        vTaskDelayUntil(&last_wake_time, frequency);
    }
}
```

nrf24_tx_task:

The function nrf24_tx_task() is responsible for transmitting nRF24L01 wireless data packets once per second and maintaining the transmission counter.
Its workflow and the role of each part can be summarized as follows:

- First, it records the system tick count last_wake_time when the task starts and sets the loop interval to 1000ms (1 second).
- It enters an infinite loop. In each iteration, it first calls nrf24_inc_tx_counter() to increment the transmission counter.
- Then, it calls send_nrf24_pack_radio() to transmit a data packet containing the current count. It uses nrf24_tx_OK to check if the transmission is successful; if failed, it prints an error log.
- Finally, it uses vTaskDelayUntil(&last_wake_time, frequency) to delay by 1 second based on absolute time, ensuring precise transmission intervals.

Overall, its role is to automatically send count data every second, update the counter, and implement the timed wireless transmission function of the nRF24L01.

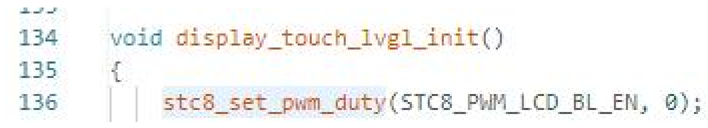


```c
static void nrf24_tx_task(void *param)
{
    TickType_t last_wake_time = xTaskGetTickCount();
    const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms

    while (1) {
        // Increment the TX counter exactly once per second
        nrf24_inc_tx_counter();
        bool nrf24_tx_OK = false;
        nrf24_tx_OK = send_nrf24_pack_radio();
        if (nrf24_tx_OK != true) {
            MAIN_ERROR("nRF24L01 TX failed");
        }

        vTaskDelayUntil(&last_wake_time, frequency);
    }
}
```

display_touch_lvgl_init

The main function of this display_touch_lvgl_init() function is to initialize the screen backlight, RGB display screen, touch screen, and LVGL graphical interface system, thereby enabling the device to have complete graphic display and touch input capabilities.

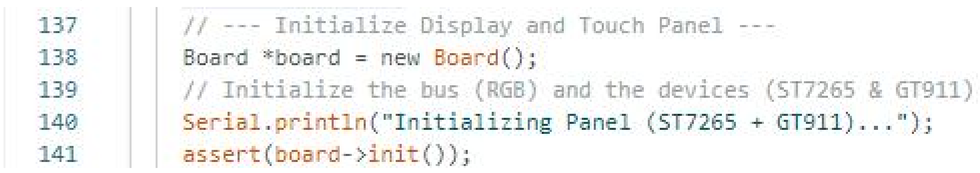
The function begins by calling stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0) to set the LCD backlight PWM duty cycle to 0%, which means turning off the screen backlight first. This step is usually used to prevent screen flickering or abnormal brightness during the display initialization process. The PWM control is handled by the external coprocessor STC8H1KXX.



```
void display_touch_lvgl_init()
{
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 0);
```

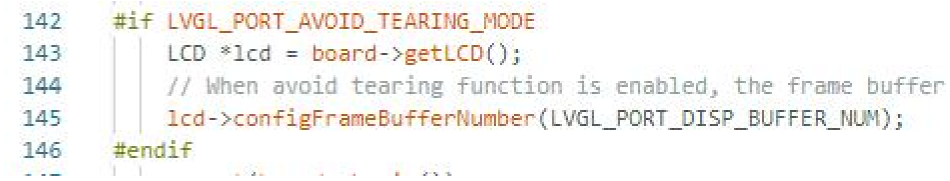
Then the program creates a Board object named "board" using the "ESP32_Display_Panel" library, which is used to uniformly manage the display bus, LCD controller, and touch device. Subsequently, it prints "Initializing Panel (ST7265 + GT911)..." indicating that the system is initializing the display and touch hardware. The LCD driver chip is ST7265 and the touch controller is GT911.

The program initializes the display bus (RGB interface) and related devices through "board->init()". If the initialization fails, the assert() function will trigger the program to stop.



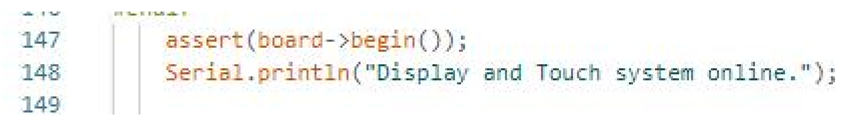
```
    // --- Initialize Display and Touch Panel ---
    Board *board = new Board();
    // Initialize the bus (RGB) and the devices (ST7265 & GT911)
    Serial.println("Initializing Panel (ST7265 + GT911)...");
    assert(board->init());
```

Next, under the conditional compilation #if LVGL_PORT_AVOID_TEARING_MODE, if the tearing avoidance mode is enabled, the program will obtain the LCD object board->getLCD() and call configFrameBufferNumber() to set the number of frame buffers, in order to reduce the problem of image tearing that occurs during screen refresh.



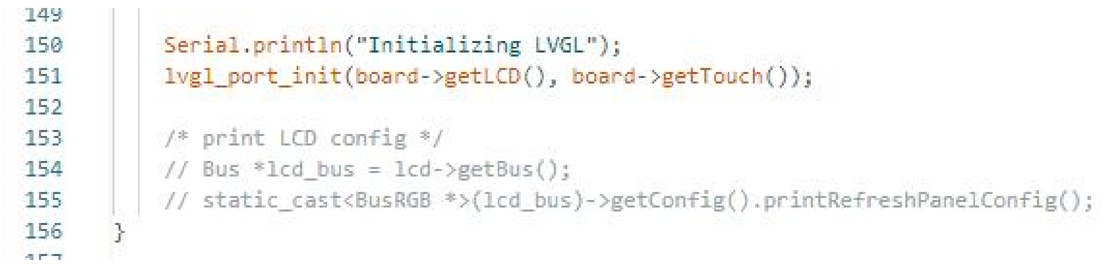
```
#if LVGL_PORT_AVOID_TEARING_MODE
    LCD *lcd = board->getLCD();
    // When avoid tearing function is enabled, the frame buffer
    lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
#endif
```

Then, call board->begin() to start the display panel. This process usually involves LCD startup timing, driver configuration, and display interface startup. Once it is successful, print "Display and Touch system online." to indicate that the display and touch system is operating normally.



```
        assert(board->begin());
        Serial.println("Display and Touch system online.");

```

Finally, the program initializes the LVGL graphics library. By calling lvgl_port_init(board->getLCD(), board->getTouch()), the LCD display device and the touch input device are registered in LVGL, enabling LVGL to handle screen drawing and touch event processing.

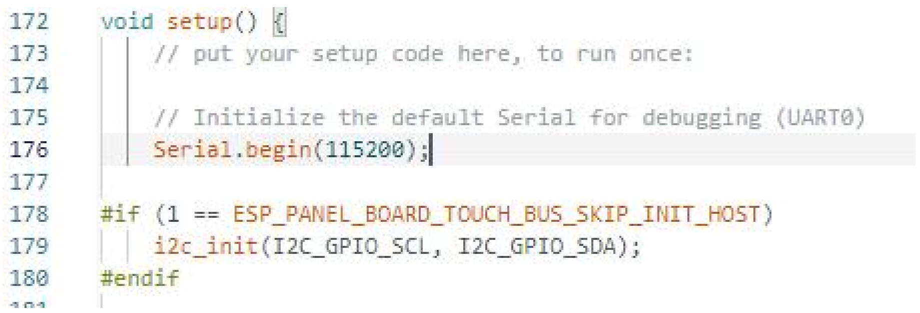


```

        Serial.println("Initializing LVGL");
        lvgl_port_init(board->getLCD(), board->getTouch());

        /* print LCD config */
        // Bus *lcd_bus = lcd->getBus();
        // static_cast<BusRGB *>(lcd_bus)->getConfig().printRefreshPanelConfig();
    }
```

setup

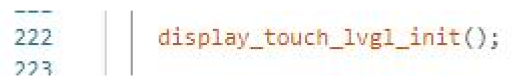
This code is the system initialization entry function named "setup()" of the entire program. It is executed only once after the device is powered on or reset. Its main function is to sequentially complete serial port debugging, power configuration, display and touch system initialization, wireless module initialization, and multi-task creation, preparing the system for normal operation.

The function first initiates serial communication (UART0) through Serial.begin(115200), setting the baud rate to 115200, which is used to output debugging information;
Then, through conditional compilation #if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST), it determines whether manual initialization of the I²C bus is required. If the macro ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST is set to 1, it calls i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA) to initialize the I²C interface using the specified SCL and SDA pins. In this way, the main control ESP32-P4 can communicate with external devices.



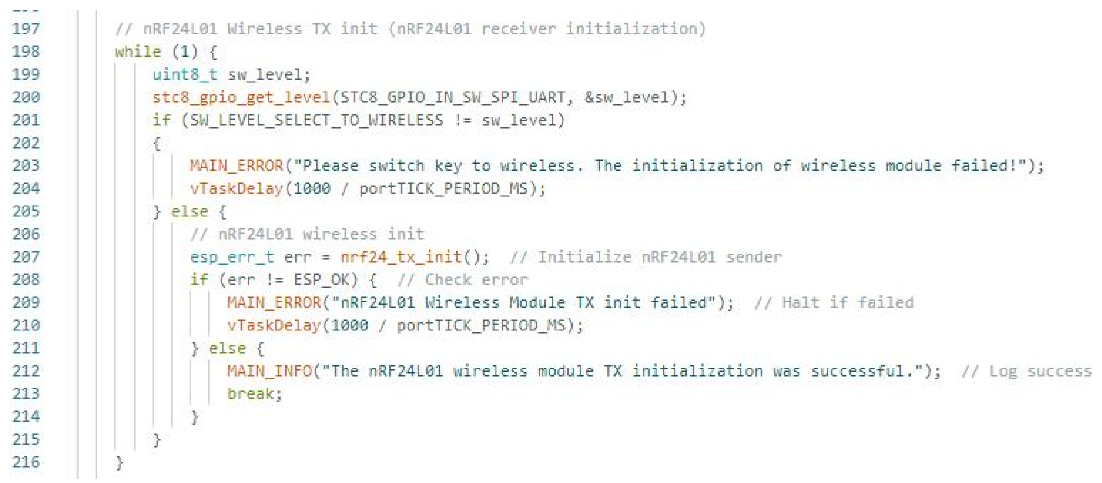
```
void setup() {
    // put your setup code here, to run once:

    // Initialize the default Serial for debugging (UART0)
    Serial.begin(115200);

#if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
    i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
#endif
```

Subsequently, the program calls the function display_touch_lvgl_init() to initialize the display and touch system. This function starts the display driver, the touch controller, and registers the display device in the graphics interface library LVGL, enabling the system to have the capability of displaying on a graphical interface.



```
display_touch_lvgl_init();
```

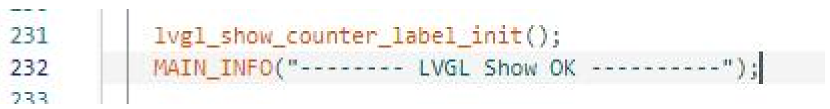
Next, the program enters a while(1) loop for initializing the wireless communication module: Firstly, it reads the status of a hardware switch STC8_GPIO_IN_SW_SPI_UART from the external processor STC8H1KXX using stc8_gpio_get_level(). If the switch has not been switched to the wireless mode (SW_LEVEL_SELECT_TO_WIRELESS), the program will continuously prompt the user to switch the switch; when the switch is correct, the program calls nrf24_tx_init() to initialize the nRF2401 wireless transmission module. If the initialization fails, it will continue to retry in the loop; if successful, it will print the log and exit the loop.



```
// nRF24L01 Wireless TX init (nRF24L01 receiver initialization)
while (1) {
    uint8_t sw_level;
    stc8_gpio_get_level(STC8_GPIO_IN_SW_SPI_UART, &sw_level);
    if (SW_LEVEL_SELECT_TO_WIRELESS != sw_level)
    {
        MAIN_ERROR("Please switch key to wireless. The initialization of wireless module failed!");
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    } else {
        // nRF24L01 wireless init
        esp_err_t err = nrf24_tx_init();  // Initialize nRF24L01 sender
        if (err != ESP_OK) {  // Check error
            MAIN_ERROR("nRF24L01 Wireless Module TX init failed");  // Halt if failed
            vTaskDelay(1000 / portTICK_PERIOD_MS);
        } else {
            MAIN_INFO("The nRF24L01 wireless module TX initialization was successful.");  // Log success
            break;
        }
    }
}
```

After completing the initialization of the display system, the program begins to initialize the wireless communication module. By calling the function nrf24_tx_init(), the transmission function of the wireless chip nRF24L01 is activated. If the return value of the function is not equal to ESP_OK, it indicates that the wireless module initialization has failed. At this time, the error log is output through MAIN_ERROR(). If the initialization is successful, a prompt message is printed through MAIN_INFO(), indicating that the wireless communication module is ready.

Then, the program calls the function lvgl_show_counter_label_init() to create an LVGL label control and display it in the center of the screen, which is used to show the current number of wireless transmissions, such as "NRF24_TX_Hello World:0", and through the log output, it indicates that the interface has been successfully loaded.

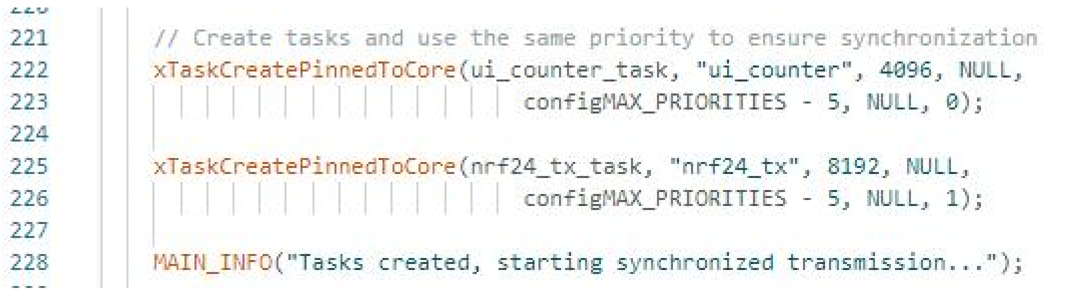


```
lvgl_show_counter_label_init();
MAIN_INFO("-------- LVGL Show OK ----------");
```

After completing all initializations, the program creates two FreeRTOS tasks using xTaskCreatePinnedToCore() and pins them to different CPU cores for execution:
The ui_counter_task is allocated 4096 bytes of stack space and runs on Core 0, responsible for reading the transmission count every second and updating the text

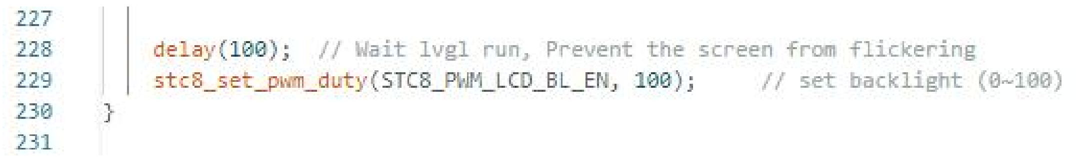
label on the screen;

The nrf24_tx_task is allocated 8192 bytes of stack space and runs on Core 1, responsible for sending a data packet to the wireless module every second and updating the transmission count. Both tasks have their priorities set to configMAX_PRIORITIES - 5, ensuring they have the same priority in the system scheduling and thus can maintain a relatively synchronized execution rhythm.



```
    // Create tasks and use the same priority to ensure synchronization
    xTaskCreatePinnedToCore(ui_counter_task, "ui_counter", 4096, NULL,
                            configMAX_PRIORITIES - 5, NULL, 0);

    xTaskCreatePinnedToCore(nrf24_tx_task, "nrf24_tx", 8192, NULL,
                            configMAX_PRIORITIES - 5, NULL, 1);

    MAIN_INFO("Tasks created, starting synchronized transmission...");
```

Finally, the program outputs the log message "Tasks created, starting synchronized transmission..." indicating that the system initialization has been fully completed and it has begun to enter the multi-task operation stage.



```
    delay(100);  // Wait lvgl run, Prevent the screen from flickering
    stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100);     // set backlight (0~100)
}
```

Finally, the program waits for a short period using delay(100) to allow LVGL to complete the first interface rendering to avoid screen flickering. Then, it calls stc8_set_pwm_duty(STC8_PWM_LCD_BL_EN, 100) to set the LCD backlight PWM duty cycle to 100% through the STC8 coprocessor, thus officially turning on the screen.

Receiver Main Function

The above is the main function code for the transmitter. Next, let’s take a look at the main function code for the receiver.
Open your receiver code in the same way as you did for the transmitter.

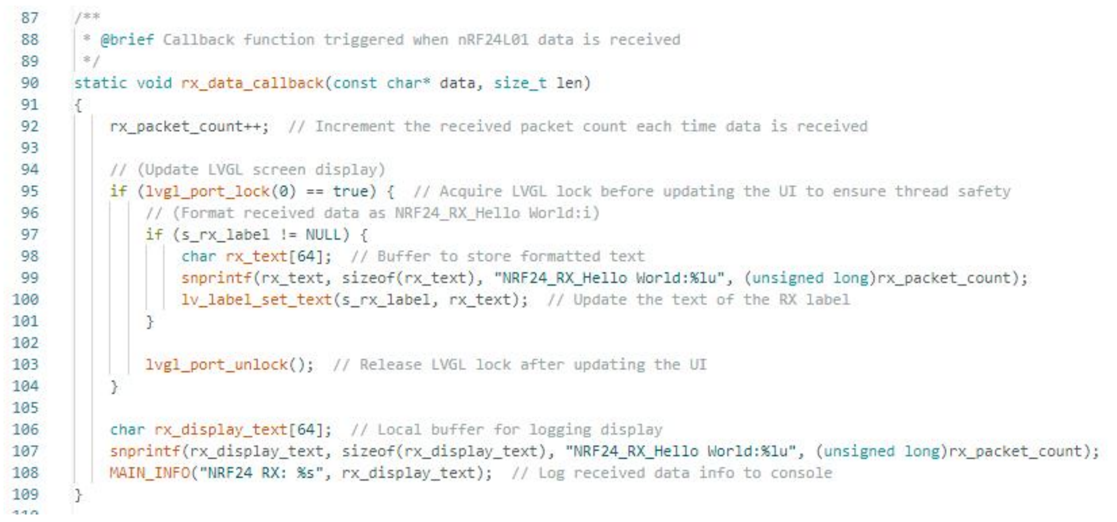
rx_data_callback:

rx_data_callback() is the callback function triggered when the nRF24L01 receives data. Its role is to count received data packets, update the interface display, and print logs.
The specific workflow is as follows:

- First, rx_packet_count++ increments the receive counter by 1.
- Then, it attempts to acquire the LVGL lock with lvgl_port_lock(0) to ensure thread safety. If successful and s_rx_label has been created, it formats the current receive count into the string "NRF24_RX_Hello World:i" using snprintf and calls lv_label_set_text to update the display label.

- After updating the interface, it releases the lock with lvgl_port_unlock().
- Finally, it formats the receive count using the local buffer rx_display_text and prints a log via MAIN_INFO, facilitating debugging and monitoring of reception status.

Overall, its role is to promptly update the interface and logs whenever the nRF24L01 receives data, enabling real-time feedback.



```c
/**
 * @brief Callback function triggered when nRF24L01 data is received
 */
static void rx_data_callback(const char* data, size_t len)
{
    rx_packet_count++;  // Increment the received packet count each time data is received

    // (Update LVGL screen display)
    if (lvgl_port_lock(0) == true) {  // Acquire LVGL lock before updating the UI to ensure thread safety
        // (Format received data as NRF24_RX_Hello World:i)
        if (s_rx_label != NULL) {
            char rx_text[64];  // Buffer to store formatted text
            snprintf(rx_text, sizeof(rx_text), "NRF24_RX_Hello World:%lu", (unsigned long)rx_packet_count);
            lv_label_set_text(s_rx_label, rx_text);  // Update the text of the RX label
        }

        lvgl_port_unlock();  // Release LVGL lock after updating the UI
    }

    char rx_display_text[64];  // Local buffer for logging display
    snprintf(rx_display_text, sizeof(rx_display_text), "NRF24_RX_Hello World:%lu", (unsigned long)rx_packet_count);
    MAIN_INFO("NRF24 RX: %s", rx_display_text);  // Log received data info to console
}
```

lvgl_show_rx_interface_init:

lvgl_show_rx_interface_init() is a function used to initialize the LVGL display interface for the nRF24L01 receiver. Its role is to create and layout interface elements for displaying received data.

The specific workflow is as follows:

- First, it attempts to acquire the LVGL lock with lvgl_port_lock(0) to ensure thread safety. If it fails, it prints an error and returns.
- It retrieves the screen object with lv_scr_act() and sets the background color to white with full opacity.
- It creates a title label title_label and sets its text to "nRF24L01 RX Receiver". It initializes the style title_style (large font, black text, transparent background), applies this style, and positions the title at the top center of the screen.
- Next, it creates a receive information label s_rx_label with initial text "NRF24_RX_Hello World:0". It defines the style rx_style (large font, black text, transparent background), applies this style, and positions the label slightly above the center of the screen.
- Finally, it releases the LVGL lock with lvgl_port_unlock().

Overall, its role is to provide an LVGL interface for the receiver to display received data in real time.

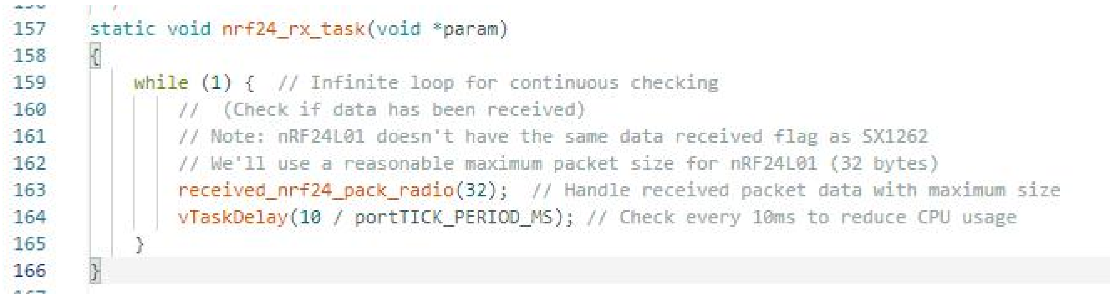
nrf24_rx_task:

nrf24_rx_task() is a FreeRTOS task function for the nRF2401 receiver, responsible for continuously polling and receiving wireless data.

- The function enters an infinite loop while(1) to ensure continuous operation.

- In each loop iteration, it calls received_nrf24_pack_radio(32) to check for and process received data packets. The parameter 32 represents the maximum packet length supported by the nRF24L01.
- It then delays for 10 milliseconds using vTaskDelay(10 / portTICK_PERIOD_MS) to reduce CPU usage.

Overall, its role is to periodically poll the nRF2401 receive buffer and trigger processing/callbacks when data is available, enabling real-time data reception.

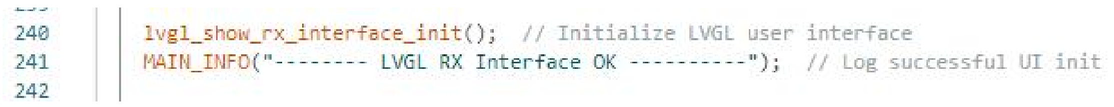


```
static void nrf24_rx_task(void *param)
{
    while (1) {  // Infinite loop for continuous checking
        //  (Check if data has been received)
        // Note: nRF24L01 doesn't have the same data received flag as SX1262
        // We'll use a reasonable maximum packet size for nRF24L01 (32 bytes)
        received_nrf24_pack_radio(32);  // Handle received packet data with maximum size
        vTaskDelay(10 / portTICK_PERIOD_MS); // Check every 10ms to reduce CPU usage
    }
}
```

setup:

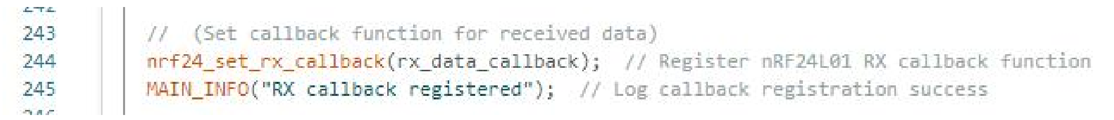
This code is the system initialization function "setup()" of the program. It is executed only once after the device is powered on or reset. Its main function is to sequentially complete serial port debugging, display and touch system initialization, wireless receiving module initialization, graphical interface creation, and receiving task startup, thereby building an embedded system capable of receiving wireless data and displaying it on the screen interface.
This part of the code is consistent in logic and code with the setup section of the sending end mentioned above. Here are the differences.

Next, the program calls the function lvgl_show_rx_interface_init() to create an LVGL graphical interface (the RX reception interface). This interface usually contains text labels or information areas, which are used to display the received wireless data or the reception status. A log message "-------- LVGL RX Interface OK ----------" is also provided to indicate that the interface initialization was successful.



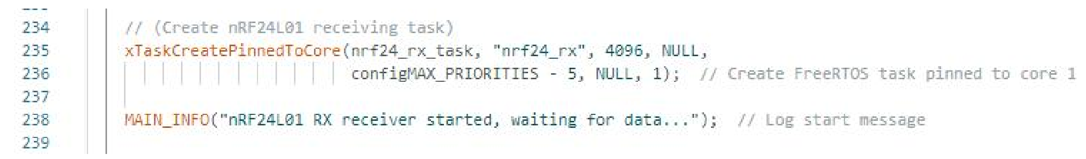
```
lvgl_show_rx_interface_init();  // Initialize LVGL user interface
MAIN_INFO("-------- LVGL RX Interface OK ----------");  // Log successful UI init
```

Subsequently, the program calls nrf24_set_rx_callback(rx_data_callback) to register a wireless reception callback function rx_data_callback. When the wireless module receives a new data packet, the underlying driver will automatically trigger this callback function, thereby executing the data processing logic in the program, such as parsing the data content, updating the interface display, or recording the received information, and outputting the log "RX callback registered" to indicate that the callback registration was successful.



```
//  (Set callback function for received data)
nrf24_set_rx_callback(rx_data_callback);  // Register nRF24L01 RX callback function
MAIN_INFO("RX callback registered");  // Log callback registration success
```

Finally, the program creates a FreeRTOS receiving task named nrf24_rx_task using xTaskCreatePinnedToCore(). It allocates 4096 bytes of task stack space for this task and pins the task to run on CPU Core 1. The task priority is set to configMAX_PRIORITIES - 5, which ensures that the task has a high scheduling priority, allowing it to continuously monitor the receiving status of the wireless module and process the received data.

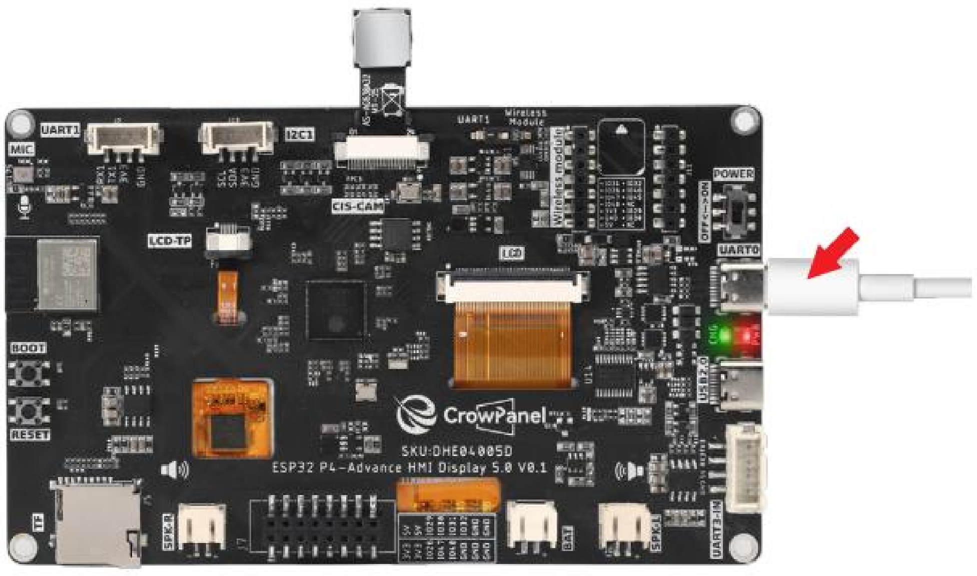


```
// (Create nRF24L01 receiving task)
xTaskCreatePinnedToCore(nrf24_rx_task, "nrf24_rx", 4096, NULL,
                        configMAX_PRIORITIES - 5, NULL, 1);  // Create FreeRTOS task pinned to core 1

MAIN_INFO("nRF24L01 RX receiver started, waiting for data...");  // Log start message

```

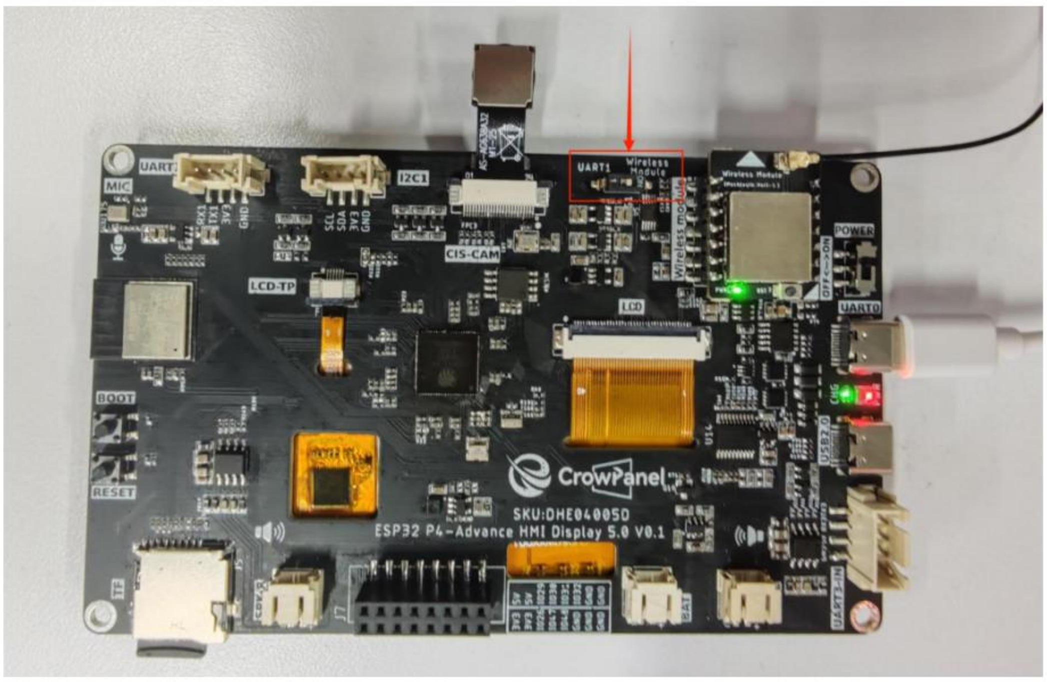
After the task is created, the program informs the system through the log "nRF24L01 RX receiver started, waiting for data..." that the system has entered the wireless receiving mode and has begun to wait for the data to arrive.

Programming Steps

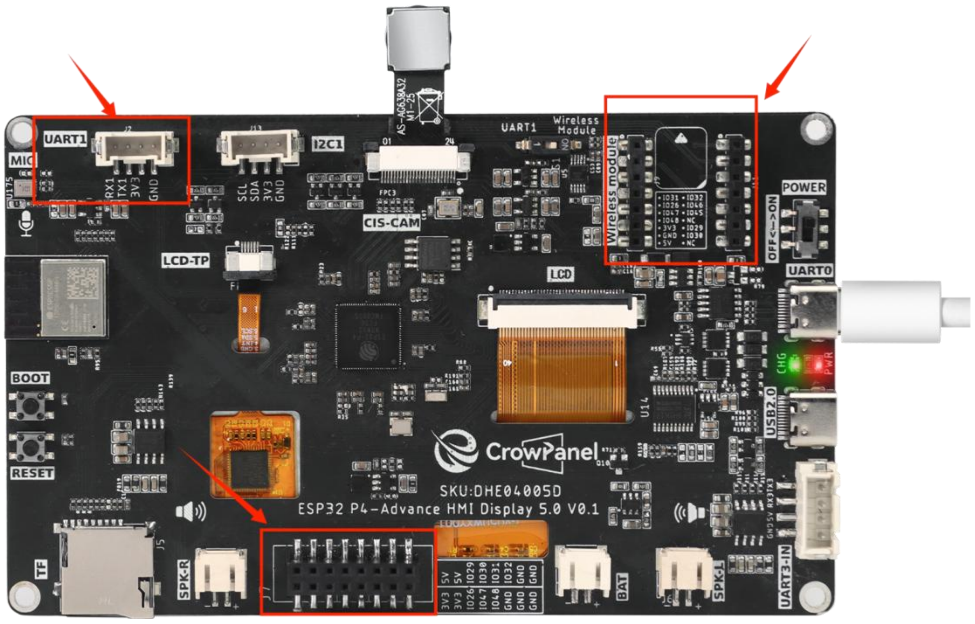
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



For the 5-inch Advance-P4 product, the jumpers need to be switched on the hardware in
order to use the wireless module. (Switch to the side with the wireless module)



This is the design on the hardware side.



Switch to UART1 port:

Among the three interfaces shown in the figure, only the UART1 interface can be used at this time.

Alternatively, the expansion header at the bottom can also be used.

That is, either the UART1 interface or the expansion header can be used, but not both.

Switch to Wireless Module port:

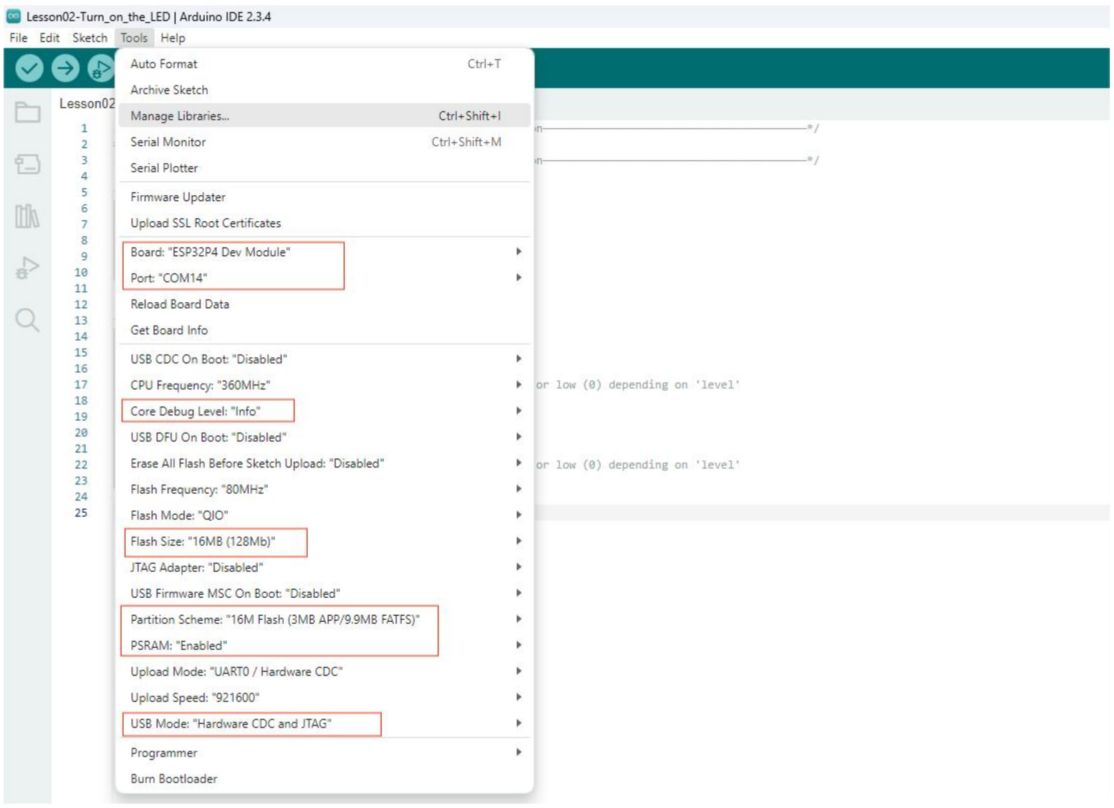
Among the three interfaces shown in the figure, only the wireless module can be used at this time.

Alternatively, the expansion header at the bottom can also be used.
That is, either the wireless module or the expansion header can be used, but not both.

Summary:
The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.
The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

Note: The H2 and C6 wireless modules can be used simultaneously with UART1.
The Lora, 2.4GHz, and WiFi-Halow wireless modules can be used with UART1, but not simultaneously.

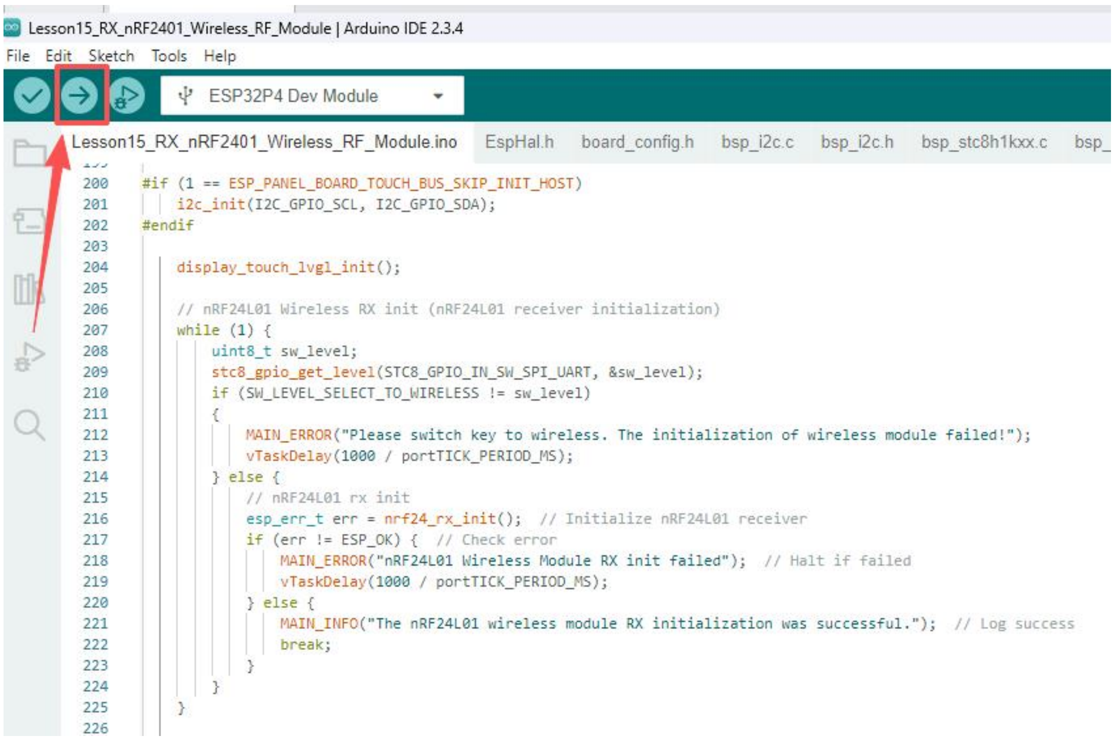
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.


Lesson02-Turn_on_the_LED | Arduino IDE 2.3.4

File Edit Sketch Tools Help

Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter
Firmware Updater
Upload SSL Root Certificates
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "Info"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (3MB APP/9.9MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
Programmer
Burn Bootloader

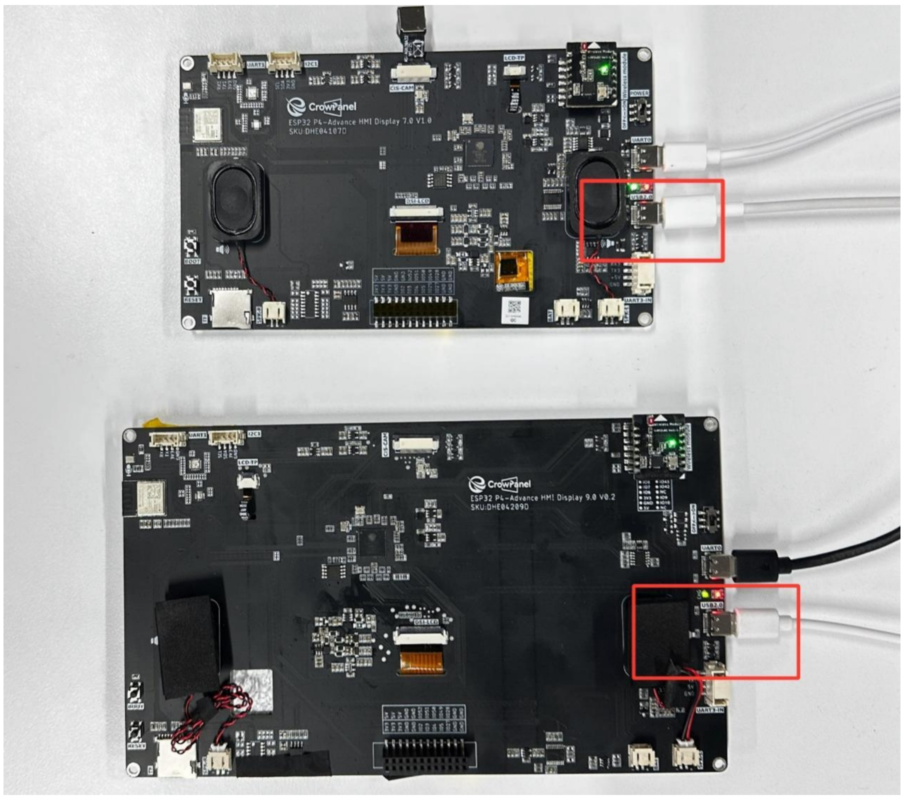
Then we compile and upload the code.


Lesson15_RX_nRF2401_Wireless_RF_Module | Arduino IDE 2.3.4
File Edit Sketch Tools Help
ESP32P4 Dev Module
Lesson15_RX_nRF2401_Wireless_RF_Module.ino
EspHal.h
board_config.h
bsp_i2c.c
bsp_i2c.h
bsp_stc8h1kxx.c

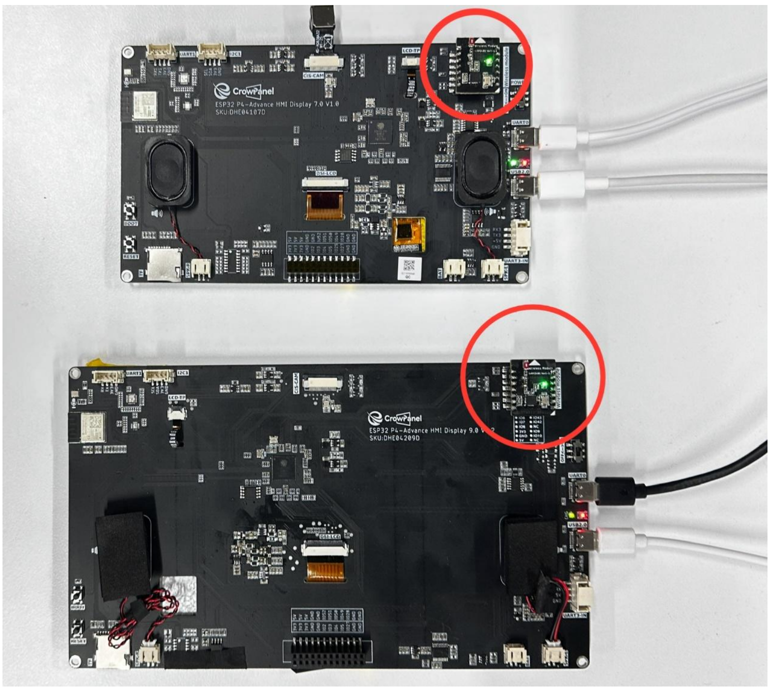
```
    #if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
        i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
    #endif

        display_touch_lvgl_init();

        // nRF24L01 Wireless RX init (nRF24L01 receiver initialization)
        while (1) {
            uint8_t sw_level;
            stc8_gpio_get_level(STC8_GPIO_IN_SW_SPI_UART, &sw_level);
            if (SW_LEVEL_SELECT_TO_WIRELESS != sw_level)
            {
                MAIN_ERROR("Please switch key to wireless. The initialization of wireless module failed!");
                vTaskDelay(1000 / portTICK_PERIOD_MS);
            } else {
                // nRF24L01 rx init
                esp_err_t err = nrf24_rx_init();  // Initialize nRF24L01 receiver
                if (err != ESP_OK) {  // Check error
                    MAIN_ERROR("nRF24L01 Wireless Module RX init failed");  // Halt if failed
                    vTaskDelay(1000 / portTICK_PERIOD_MS);
                } else {
                    MAIN_INFO("The nRF24L01 wireless module RX initialization was successful.");  // Log success
                    break;
                }
            }
        }

```

Wait for the code upload to complete.

At this point, remember to connect your Advance-P4 using an additional Type-C cable via the USB 2.0 interface. This is because the maximum current provided by a computer’s USB-A port is generally 500mA, and the Advance-P4 requires a sufficient power supply when using multiple peripherals—especially the screen. (It is recommended to connect it to a charger.)

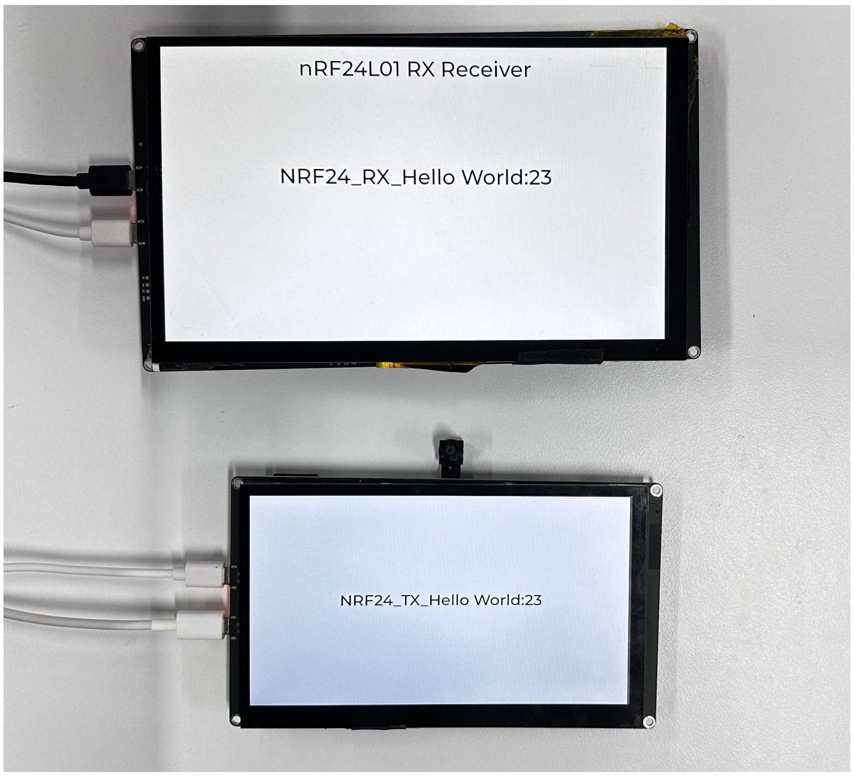


Insert the nRF2401 wireless RF module into each of the two Advance-P4 development boards.



After running the code on both boards respectively, you will be able to see on the transmitter's Advance-P4 screen that the nRF2401 module is sending data labeled "NRF24_TX_Hello World:i", where "i" increases by 1 every second.

Similarly, on the receiver's Advance-P4 screen, you will see that the nRF2401 module is receiving data labeled "NRF24_RX_Hello World:i"; after receiving the message, "i" will also increase by 1 every second.



```
nRF24L01 RX Receiver

NRF24_RX_Hello World:23
```

```
NRF24_TX_Hello World:23
```

Lesson16---WiFi Mode

Introduction

In this section, we will conduct a systematic study on the Wi-Fi core functions of the ESP32 series chips (including models such as ESP32-C6): Firstly, we will deeply analyze the underlying fundamental principles of Wi-Fi communication, including the technical logic and application scenarios of the three core working modes: Station (STA), SoftAP (AP), and STA+AP coexistence.

Learning Goals

1. Understand the working principles, underlying logic and typical application scenarios of the three core Wi-Fi modes (STA, SoftAP, STA+AP coexistence) of the ESP32 series chips (such as ESP32-C6).

2. Master the core programming process of Wi-Fi: protocol stack / driver initialization, mode and parameter configuration, connection startup and implementation methods of event callbacks (disconnection/reconnection, device access, etc.).

3. Master the code implementation of the three Wi-Fi modes (STA connecting to the router, AP creating a hotspot, STA+AP coexistence + NAT shared external network) under the ESP32-P4 + ESP32-C6 architecture.

ESP32 Wi-Fi Function Overview

The ESP32 series of chips, with their highly integrated wireless communication capabilities, have become the core preferred solution in fields such as the Internet of Things (IoT), smart home, and industrial automation. Most models of this series are equipped with high-performance wireless modules, among which 2.4 GHz Wi-Fi is the basic standard, which can meet the needs of most low-power, short-range data transmission scenarios.

The new models represented by ESP32-C6 have further expanded the boundaries of wireless capabilities, not only supporting 5 GHz dual-band Wi-Fi, but also being compatible with the new Wi-Fi 6 (802.11ax) protocol, while also being compatible with traditional standards such as 802.11b/g/n. They can provide higher transmission rates, lower network latency, and more stable concurrent connection experiences for multiple devices.

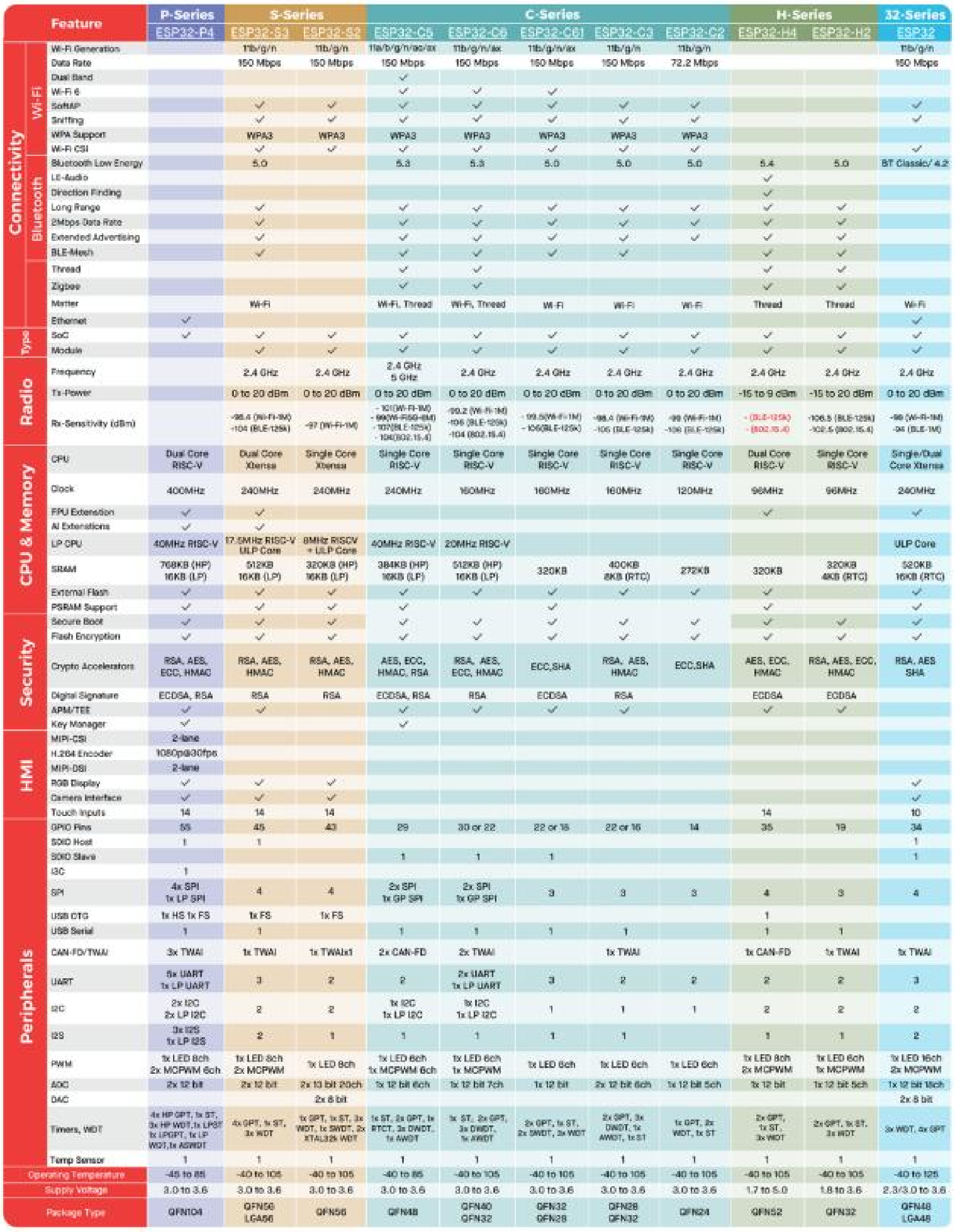
It is worth noting that some special models in the ESP32 family (such as the P series and H series) have simplified the Wi-Fi function modules to focus on exclusive scenarios such as high-performance computing and industrial-level control, in exchange for more abundant computing resources and more stable industrial-level performance. The details of the wireless functions supported by each model can be accurately selected according to the official document "ESP32 Product Overview" released by Espressif.

The Wi-Fi function of the ESP32 series chips offers a wide range of working modes, which can flexibly adapt to various development requirements:

- Station (STA) mode: The chip acts as a Wi-Fi client, actively connecting to existing wireless networks such as routers and hotspots, obtaining a dynamic IP address and joining the local network, enabling data communication with devices in the external network or within the local network;
- SoftAP (soft AP) mode: The chip creates an independent Wi-Fi hotspot by itself, allowing devices such as mobile phones, computers, and other single-chip microcontrollers to connect. This builds a point-to-point or small local area network without the need for an external router, suitable for local device debugging, offline data interaction, etc.;
- STA + AP coexistence mode: The chip runs both of these modes simultaneously. It can connect to the external network to achieve data upload and remote control, and also act as a local hotspot for debugging devices to connect, balancing the dual needs of remote operation and local development.

In terms of security and performance, the Wi-Fi function of ESP32 also performs exceptionally well: In terms of security, it supports mainstream encryption protocols such as WPA2, WPA3, and enterprise-level authentication, which can effectively ensure the security of data transmission and prevent network attacks and data leakage;

In terms of performance, its maximum transmission rate can reach 150 Mbps (with some high-end models having even higher rates), and it is equipped with multiple refined power-saving modes, which can dynamically adjust power consumption according to actual business needs, extend the battery-powered device's battery life, and some models also support multi-antenna diversity technology, further enhancing the stability and anti-interference ability of wireless signals in complex environments.
https://products.espressif.com/static/Espressif%20SoC%20Product%20Portfolio.pdf

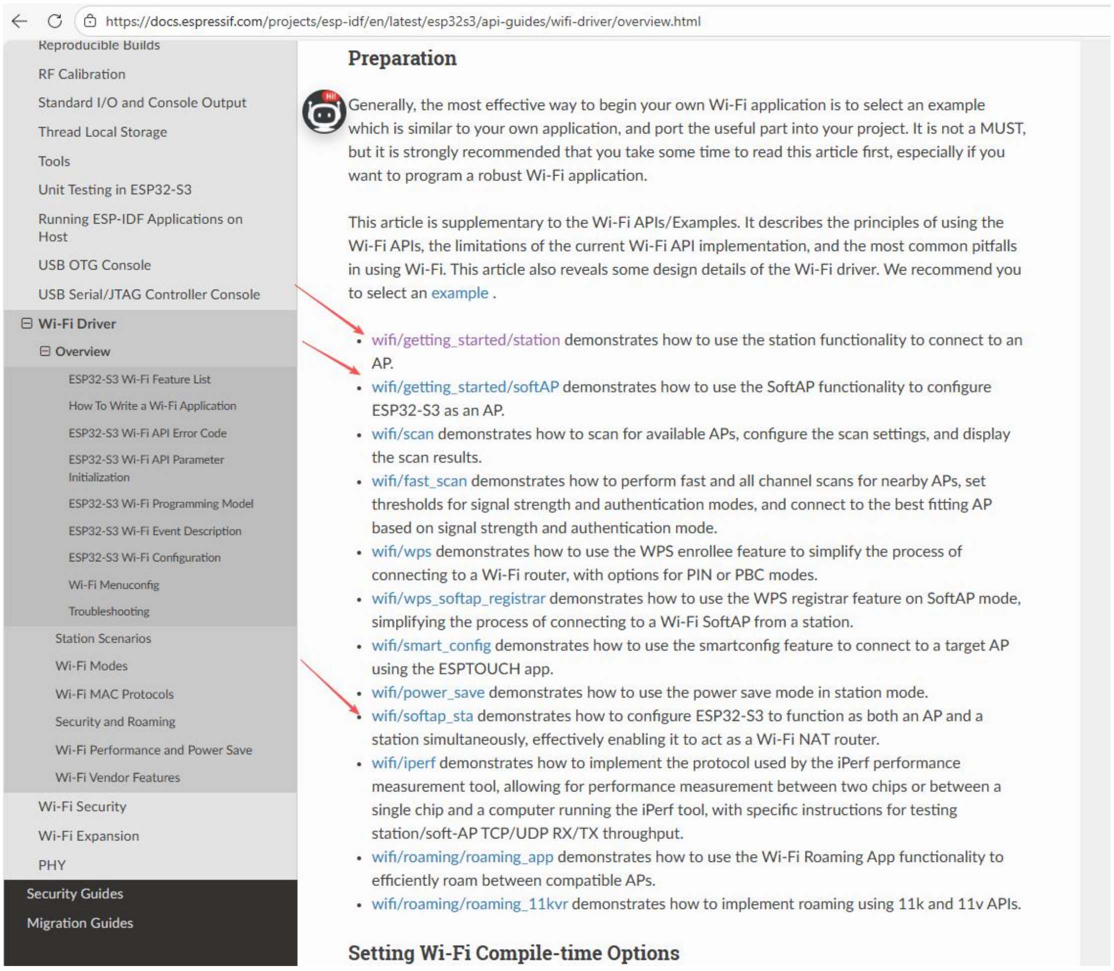


	Feature	P-Series	S-Series		C-Series					H-Series		32-Series
		ESP32-P4	ESP32-S3	ESP32-S2	ESP32-C5	ESP32-C6	ESP32-C61	ESP32-C3	ESP32-C2	ESP32-H4	ESP32-H2	ESP32
Connectivity / Wi-Fi	Wi-Fi Generation		11b/g/n	11b/g/n	11a/b/g/n/ac/ax	11b/g/n/ax	11b/g/n/ax	11b/g/n	11b/g/n			11b/g/n
	Data Rate		150 Mbps	150 Mbps	150 Mbps	150 Mbps	150 Mbps	150 Mbps	72.2 Mbps			150 Mbps
	Dual Band				✓							
	Wi-Fi 6				✓	✓	✓					
	SoftAP		✓	✓	✓	✓	✓	✓	✓			✓
	Sniffing		✓	✓	✓	✓	✓	✓	✓			✓
	WPA Support		WPA3	WPA3	WPA3	WPA3	WPA3	WPA3	WPA3			
	Wi-Fi CSI		✓	✓	✓	✓	✓	✓	✓			✓
Bluetooth	Bluetooth Low Energy		5.0		5.3	5.3	5.0	5.0	5.0	5.4	5.0	BT Classic/ 4.2
	LE-Audio									✓		
	Direction Finding									✓		
	Long Range		✓		✓	✓	✓	✓	✓	✓	✓	
	2Mbps Data Rate		✓		✓	✓	✓	✓	✓	✓	✓	
	Extended Advertising		✓		✓	✓	✓	✓	✓	✓	✓	
	BLE-Mesh		✓		✓	✓	✓	✓		✓	✓	
	Thread				✓	✓				✓	✓	
	Zigbee				✓	✓				✓	✓	
	Matter		Wi-Fi		Wi-Fi, Thread	Wi-Fi, Thread	Wi-Fi	Wi-Fi	Wi-Fi	Thread	Thread	Wi-Fi
	Ethernet	✓										✓
Type	SoC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Module		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Radio	Frequency		2.4 GHz	2.4 GHz	2.4 GHz 5 GHz	2.4 GHz	2.4 GHz	2.4 GHz	2.4 GHz	2.4 GHz	2.4 GHz	2.4 GHz
	Tx-Power		0 to 20 dBm	0 to 20 dBm	0 to 20 dBm	0 to 20 dBm	0 to 20 dBm	0 to 20 dBm	0 to 20 dBm	-15 to 9 dBm	-15 to 20 dBm	0 to 20 dBm
	Rx-Sensitivity (dBm)		-98.4 (Wi-Fi-1M) -104 (BLE-125k)	-97 (Wi-Fi-1M)	-101(Wi-Fi-1M) -99(Wi-Fi5G-6M) -107(BLE-125k) -104(802.15.4)	-99.2 (Wi-Fi-1M) -106 (BLE-125k) -104 (802.15.4)	-99.5(Wi-Fi-1M) -106(BLE-125K)	-98.4 (Wi-Fi-1M) -106 (BLE-125k)	-99 (Wi-Fi-1M) -106 (BLE-125k)	- (BLE-125k) - (802.15.4)	-106.5 (BLE-125k) -102.5 (802.15.4)	-98 (Wi-Fi-1M) -94 (BLE-1M)
CPU & Memory	CPU	Dual Core RISC-V	Dual Core Xtensa	Single Core Xtensa	Single Core RISC-V	Single Core RISC-V	Single Core RISC-V	Single Core RISC-V	Single Core RISC-V	Dual Core RISC-V	Single Core RISC-V	Single/Dual Core Xtensa
	Clock	400MHz	240MHz	240MHz	240MHz	160MHz	160MHz	160MHz	120MHz	96MHz	96MHz	240MHz
	FPU Extension	✓	✓							✓		✓
	AI Extensions	✓	✓									
	LP CPU	40MHz RISC-V	17.5MHz RISC-V ULP Core	8MHz RISCV + ULP Core	40MHz RISC-V	20MHz RISC-V						ULP Core
	SRAM	768KB (HP) 16KB (LP)	512KB 16KB (LP)	320KB (HP) 16KB (LP)	384KB (HP) 16KB (LP)	512KB (HP) 16KB (LP)	320KB	400KB 8KB (RTC)	272KB	320KB	320KB 4KB (RTC)	520KB 16KB (RTC)
	External Flash	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
	PSRAM Support	✓	✓	✓	✓		✓			✓		✓
Security	Secure Boot	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Flash Encryption	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Crypto Accelerators	RSA, AES, ECC, HMAC	RSA, AES, HMAC	RSA, AES, HMAC	AES, ECC, HMAC, RSA	RSA, AES, ECC, HMAC	ECC,SHA	RSA, AES, HMAC	ECC,SHA	AES, ECC, HMAC	RSA, AES, ECC, HMAC	RSA, AES SHA
	Digital Signature	ECDSA, RSA	RSA	RSA	ECDSA, RSA	RSA	ECDSA	RSA		ECDSA	ECDSA	
	APM/TEE	✓	✓		✓	✓	✓	✓		✓	✓	
	Key Manager	✓			✓							
HMI	MIPI-CSI	2-lane										
	H.264 Encoder	1080p@30fps										
	MIPI-DSI	2-lane										
	RGB Display	✓	✓	✓								✓
	Camera Interface	✓	✓	✓								✓
	Touch Inputs	14	14	14						14		10
Peripherals	GPIO Pins	55	45	43	29	30 or 22	22 or 16	22 or 16	14	35	19	34
	SDIO Host	1	1									1
	SDIO Slave				1	1	1					1
	I3C	1										
	SPI	4x SPI 1x LP SPI	4	4	2x SPI 1x GP SPI	2x SPI 1x GP SPI	3	3	3	4	3	4
	USB OTG	1x HS 1x FS	1x FS	1x FS						1		
	USB Serial	1	1		1	1	1	1		1	1	
	CAN-FD/TWAI	3x TWAI	1x TWAI	1x TWAIx1	2x CAN-FD	2x TWAI		1x TWAI		1x CAN-FD	1x TWAI	1x TWAI
	UART	5x UART 1x LP UART	3	2	2	2x UART 1x LP UART	3	2	2	2	2	3
	I2C	2x I2C 2x LP I2C	2	2	1x I2C 1x LP I2C	1x I2C 1x LP I2C	1	1	1	2	2	2
	I2S	3x I2S 1x LP I2S	2	1	1	1	1	1		1	1	2
	PWM	1x LED 8ch 2x MCPWM 6ch	1x LED 8ch 2x MCPWM	1x LED 8ch	1x LED 6ch 1x MCPWM 6ch	1x LED 6ch 1x MCPWM	1x LED 6ch	1x LED 6ch	1x LED 6ch	1x LED 8ch 2x MCPWM	1x LED 6ch 1x MCPWM	1x LED 16ch 2x MCPWM
	ADC	2x 12 bit	2x 12 bit	2x 13 bit 20ch	1x 12 bit 6ch	1x 12 bit 7ch	1x 12 bit	2x 12 bit 6ch	1x 12 bit 5ch	1x 12 bit	1x 12 bit 5ch	1x 12 bit 18ch
	DAC			2x 8 bit								2x 8 bit
	Timers, WDT	4x HP GPT, 1x ST, 3x HP WDT,1x LPST 1x LPGPT, 1x LP WDT,1x ASWDT	4x GPT, 1x ST, 3x WDT	1x GPT, 1x ST, 3x WDT, 1x SWDT, 2x XTAL32k WDT	1x ST, 2x GPT, 1x RTCT, 3x DWDT, 1x AWDT	1x ST, 2x GPT, 3x DWDT, 1x AWDT	2x GPT, 1x ST, 2x SWDT, 3x WDT	2x GPT, 3x DWDT, 1x AWDT, 1x ST	1x GPT, 2x WDT, 1x ST	2x GPT, 1x ST, 3x WDT	2x GPT, 1x ST, 3x WDT	3x WDT, 4x GPT
	Temp Sensor	1	1	1	1	1	1	1	1	1	1	1
Operating Temperature		-45 to 85	-40 to 105	-40 to 105	-40 to 85	-40 to 105	-40 to 105	-40 to 105	-40 to 105	-40 to 105	-40 to 105	-40 to 125
Supply Voltage		3.0 to 3.6	3.0 to 3.6	3.0 to 3.6	3.0 to 3.6	3.0 to 3.6	3.0 to 3.6	3.0 to 3.6	3.0 to 3.6	1.7 to 5.0	1.8 to 3.6	2.3/3.0 to 3.6
Package Type		QFN104	QFN56 LGA56	QFN56	QFN48	QFN40 QFN32	QFN32 QFN28	QFN28 QFN32	QFN24	QFN52	QFN32	QFN48 LGA48

General steps for Wi-Fi programming

When developing Wi-Fi applications using the ESP32 series chips based on the ESP-IDF framework, regardless of whether the goal is to configure the Station (STA) client mode or the SoftAP (AP) access point mode, the core programming process follows three main stages: initialization, configuration, connection, and event handling. The operation logic of each stage is highly universal and standardized. If you want to deeply understand the complete process of specific scenarios (such as disconnection and reconnection in STA mode, and device access management in AP

mode), you can refer to the detailed guidelines in the official ESP-IDF documentation for "Wi-Fi Station Development Process" and "Wi-Fi AP Development Process".
Official link::
https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-guides/wifi-driver/overview.html


https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-guides/wifi-driver/overview.html

Reproducible Builds
RF Calibration
Standard I/O and Console Output
Thread Local Storage
Tools
Unit Testing in ESP32-S3
Running ESP-IDF Applications on Host
USB OTG Console
USB Serial/JTAG Controller Console
Wi-Fi Driver
Overview
ESP32-S3 Wi-Fi Feature List
How To Write a Wi-Fi Application
ESP32-S3 Wi-Fi API Error Code
ESP32-S3 Wi-Fi API Parameter Initialization
ESP32-S3 Wi-Fi Programming Model
ESP32-S3 Wi-Fi Event Description
ESP32-S3 Wi-Fi Configuration
Wi-Fi Menuconfig
Troubleshooting
Station Scenarios
Wi-Fi Modes
Wi-Fi MAC Protocols
Security and Roaming
Wi-Fi Performance and Power Save
Wi-Fi Vendor Features
Wi-Fi Security
Wi-Fi Expansion
PHY
Security Guides
Migration Guides

Preparation

Generally, the most effective way to begin your own Wi-Fi application is to select an example which is similar to your own application, and port the useful part into your project. It is not a MUST, but it is strongly recommended that you take some time to read this article first, especially if you want to program a robust Wi-Fi application.

This article is supplementary to the Wi-Fi APIs/Examples. It describes the principles of using the Wi-Fi APIs, the limitations of the current Wi-Fi API implementation, and the most common pitfalls in using Wi-Fi. This article also reveals some design details of the Wi-Fi driver. We recommend you to select an example .

- wifi/getting_started/station demonstrates how to use the station functionality to connect to an AP.
- wifi/getting_started/softAP demonstrates how to use the SoftAP functionality to configure ESP32-S3 as an AP.
- wifi/scan demonstrates how to scan for available APs, configure the scan settings, and display the scan results.
- wifi/fast_scan demonstrates how to perform fast and all channel scans for nearby APs, set thresholds for signal strength and authentication modes, and connect to the best fitting AP based on signal strength and authentication mode.
- wifi/wps demonstrates how to use the WPS enrollee feature to simplify the process of connecting to a Wi-Fi router, with options for PIN or PBC modes.
- wifi/wps_softap_registrar demonstrates how to use the WPS registrar feature on SoftAP mode, simplifying the process of connecting to a Wi-Fi SoftAP from a station.
- wifi/smart_config demonstrates how to use the smartconfig feature to connect to a target AP using the ESPTOUCH app.
- wifi/power_save demonstrates how to use the power save mode in station mode.
- wifi/softap_sta demonstrates how to configure ESP32-S3 to function as both an AP and a station simultaneously, effectively enabling it to act as a Wi-Fi NAT router.
- wifi/iperf demonstrates how to implement the protocol used by the iPerf performance measurement tool, allowing for performance measurement between two chips or between a single chip and a computer running the iPerf tool, with specific instructions for testing station/soft-AP TCP/UDP RX/TX throughput.
- wifi/roaming/roaming_app demonstrates how to use the Wi-Fi Roaming App functionality to efficiently roam between compatible APs.
- wifi/roaming/roaming_11kvr demonstrates how to implement roaming using 11k and 11v APIs.

Setting Wi-Fi Compile-time Options

1. Initialization phase: Establish the basic environment for Wi-Fi operation

The core objective of this phase is to complete the initialization of the underlying protocol stack, hardware drivers, and system event framework, laying a solid foundation for the operation of Wi-Fi functions. The specific operations are as follows:

Initialization of protocol stack and network interface: Firstly, initialize the lightweight TCP/IP protocol stack (LwIP), which is the core underlying component for implementing network data transmission; simultaneously, initialize the esp-netif network interface management module, which is responsible for uniformly managing various network interfaces of ESP32 (such as STA, AP, Ethernet, etc.), serving as the key bridge between upper-layer applications and the underlying hardware.

Building event-driven framework: Create the system's default event loop (esp_event) and register Wi-Fi-related event handling callbacks. The Wi-Fi status changes (such as successful connection, obtaining IP, disconnection, etc.) are triggered in the form of asynchronous events, and the event loop is the core mechanism for implementing status monitoring and automated processing.

Creation of default network interface: Create corresponding default network interface instances (such as ESP_NETIF_DEFAULT_WIFI_STA or ESP_NETIF_DEFAULT_WIFI_AP) based on the target mode (STA/AP), bind the network interface with the Wi-Fi hardware, and ensure that data can be transmitted and received through the specified interface.

Wi-Fi driver initialization: Call the esp_wifi_init() interface to complete the initialization of the Wi-Fi hardware driver. This operation will start the Wi-Fi task thread inside ESP32, complete the adaptation of the radio module, protocol stack, and hardware, and ensure that the Wi-Fi hardware is in a configurable and operational state.

2. Configuration Phase: Customizing Wi-Fi Operating Parameters

After initialization is completed, the core operating parameters of Wi-Fi need to be configured according to actual business requirements to match the target network environment or functional requirements:

Connection Parameter Configuration: Define the wifi_config_t structure variable and fill in the core parameters corresponding to the mode - in STA mode, the SSID (network name), password, authentication method (such as WPA2-PSK) of the target Wi-Fi need to be configured; in AP mode, the hotspot name, password, maximum number of connected devices, authentication method, and channel need to be configured.

Mode Setting: Call the esp_wifi_set_mode() interface to clearly set the Wi-Fi operating mode (such as WIFI_MODE_STA, WIFI_MODE_AP, or WIFI_MODE_APSTA), and this operation will lock the core working logic of Wi-Fi.

Configuration Parameters Take Effect: Call the esp_wifi_set_config() interface to apply the parameters configured in the wifi_config_t to the corresponding network interface (STA/AP), so that the parameters are bound with the hardware driver and protocol stack.

3. Connection and Event Handling Phase: Initiating Wi-Fi and Handling State Changes

After configuration is completed, start the Wi-Fi function and handle various asynchronous states through event callbacks to ensure the stable operation of the Wi-Fi function:

Starting Wi-Fi function: Call the esp_wifi_start() interface to start the Wi-Fi module. At this point, the ESP32 will initialize the radio module according to the configured mode and enter the "waiting to connect (STA)" or "hotspot broadcasting (AP)" state.

Initiating network connection (only in STA mode): Call the esp_wifi_connect() interface to send a connection request to the configured target Wi-Fi network; in AP mode, this step is not required and the hotspot will be automatically broadcasted and waiting for device access.

Asynchronous event handling: By registering the WIFI_EVENT and IP_EVENT event callback functions, handle various critical states - such as "connection success/failure" "obtaining IP address" "connection disconnection" in STA mode, and "device access/disconnection" in AP mode. Event handling is the core to ensure the robustness of the Wi-Fi function, for example, adding reconnection logic in the "connection disconnection" event and triggering data transmission tasks in the "obtaining IP" event.

After understanding the above professional knowledge, next we will guide you to have a better understanding of these three modes.

Station (STA) mode

① Professional Definition

The Station (STA) mode refers to the device operating in the IEEE 802.11 Station role, actively scanning (Active Scan) or passively scanning (Passive Scan) to search for and associate with an Access Point (AP), then obtaining network parameters (IP / Gateway / DNS) through the DHCP protocol, and finally joining an Infrastructure Network.

You don't need to memorize all the key words at once, but being familiar with them is very important.

② What exactly did the STA mode do in the engineering aspect?

Under the STA mode, the ESP32-C6 actually completed a complete set of processes:

- Channel scanning (Channel Scan)
  2.4GHz (Channels 1–13)
- Authentication WPA2 / WPA3
- Association
- 4-Way Handshake for Key Agreement
- DHCP Request
- Completion of IP Layer Network Establishment

In your code, you only wrote one sentence:

```
bsp_wifi_connect(ssid, password);
```

But the underlying layer has already executed the entire Wi-Fi protocol stack.

③ Understanding STA

You can think of STA as:

- "I'm just an ordinary user, and I want to access the internet"

Its characteristics are very clear:

❌ No Wi-Fi

❌ Regardless of others

✅ Only concerned with:

- Whether I have connected
- Whether I have an IP address
- Whether I can access the server

④ Our role in the project

In the ESP32-P4 + ESP32-C6 architecture:

C6 (STA)

- Connect to the router
- Access the internet
- Request weather / time / cloud services

P4

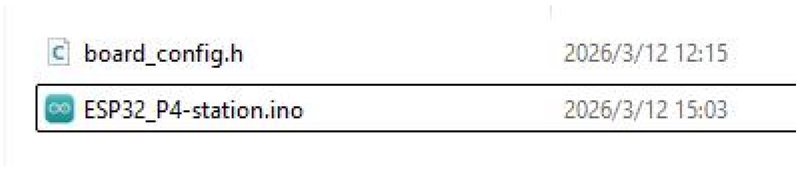
- Focus on UI / LVGL / Business Logic

This is a typical "network coprocessor" design concept.

Now that we have understood the Station mode, let's use it to enhance our understanding.

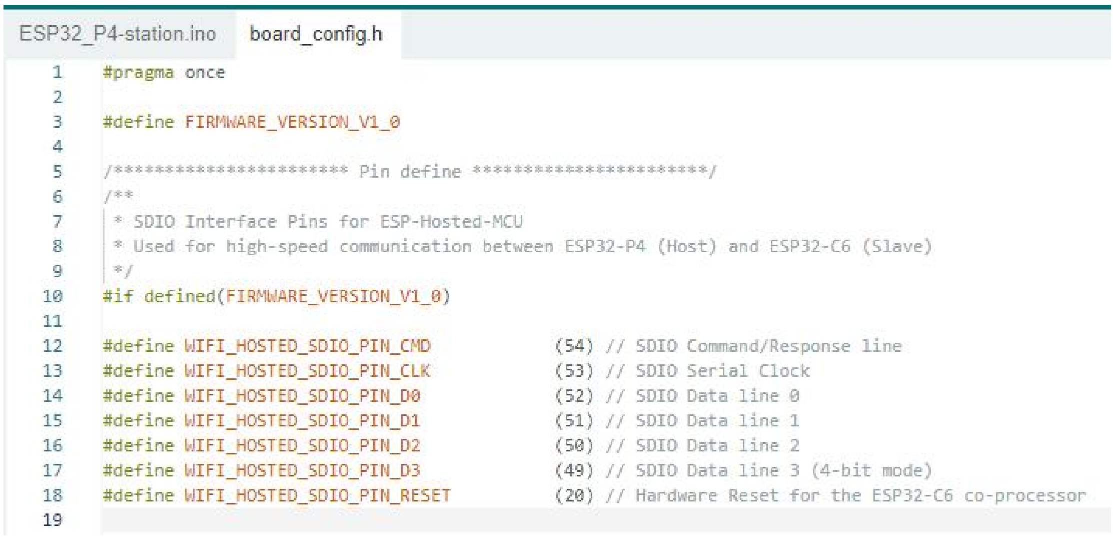
Click the link below to download the code for the Station mode.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson16-Wi-Fi_function/ESP32_P4-station

Double-click to open the code for this lesson (the ESP32_P4-station.ino file)



```
board_config.h          2026/3/12 12:15
ESP32_P4-station.ino    2026/3/12 15:03
```

The board_config.h file is mainly used to define the SDIO interface pin configurations required for high-speed communication between the ESP32 main controller and the WiFi coprocessor. It also uses conditional compilation to ensure that these pin definitions only take effect in the specified firmware version.



```
#pragma once

#define FIRMWARE_VERSION_V1_0

/*********************** Pin define **********************/
/**
 * SDIO Interface Pins for ESP-Hosted-MCU
 * Used for high-speed communication between ESP32-P4 (Host) and ESP32-C6 (Slave)
 */
#if defined(FIRMWARE_VERSION_V1_0)

#define WIFI_HOSTED_SDIO_PIN_CMD            (54) // SDIO Command/Response line
#define WIFI_HOSTED_SDIO_PIN_CLK            (53) // SDIO Serial Clock
#define WIFI_HOSTED_SDIO_PIN_D0             (52) // SDIO Data line 0
#define WIFI_HOSTED_SDIO_PIN_D1             (51) // SDIO Data line 1
#define WIFI_HOSTED_SDIO_PIN_D2             (50) // SDIO Data line 2
#define WIFI_HOSTED_SDIO_PIN_D3             (49) // SDIO Data line 3 (4-bit mode)
#define WIFI_HOSTED_SDIO_PIN_RESET          (20) // Hardware Reset for the ESP32-C6 co-processor
```

First of all, #pragma once is a header file protection directive. Its function is to ensure that this header file is included only once throughout the entire project compilation process, thereby avoiding compilation errors caused by duplicate definitions.

Then, the code defines the current firmware version macro using the #define directive, namely #define FIRMWARE_VERSION_V1_0. This macro is used to distinguish different hardware versions or firmware configurations within the project, facilitating the switching of different pin configurations or functionality implementations based on different versions within the same code project.

Then, in the "Pin define" section, through the conditional compilation statement #if defined(FIRMWARE_VERSION_V1_0), it is determined whether the current firmware version is V1.0. If it is, the following group of pin definitions related to WiFi communication will be enabled.

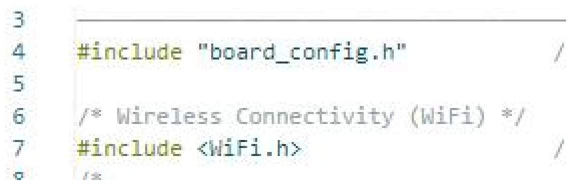
These macros describe the various signal lines of the SDIO (Secure Digital Input Output) communication interface. Among them, WIFI_HOSTED_SDIO_PIN_CMD represents the command/response line of SDIO, used for sending control instructions; WIFI_HOSTED_SDIO_PIN_CLK is the SDIO clock signal, used for synchronizing data transmission; WIFI_HOSTED_SDIO_PIN_D0 to WIFI_HOSTED_SDIO_PIN_D3 are four data lines, forming a 4-bit SDIO data bus, capable of achieving a higher data transmission rate; and WIFI_HOSTED_SDIO_PIN_RESET is used to control the hardware reset of the WiFi coprocessor. When it is necessary to restart the WiFi module, the reset operation can be performed through this pin.

The entire interface is mainly used for the ESP32-P4 main controller to act as the Host, communicating with the external ESP32-C6 module as the Slave. This architecture is commonly referred to as the ESP-Hosted-MCU solution. The main controller is responsible for running the application, while the WiFi chip is specifically dedicated to wireless communication, enabling high-speed and stable data exchange through SDIO.

Return to the main code file of ino

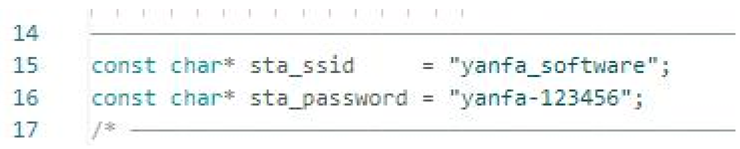
This program is an example code based on Arduino, whose main function is to initialize the WiFi hardware interface upon system startup, connect to the specified wireless network, and continuously monitor the network connection status during operation. If the connection is lost, it will automatically reconnect.

The beginning of the code imports the hardware configuration file of the development board through "#include "board_config.h"", which contains the definition of the SDIO pins used for WiFi; subsequently, it imports the standard WiFi.h library through "#include <WiFi.h>", which provides functions for connecting to wireless networks, obtaining IP addresses, and detecting network status.



```
#include "board_config.h"          /

/* Wireless Connectivity (WiFi) */
#include <WiFi.h>                  /
```

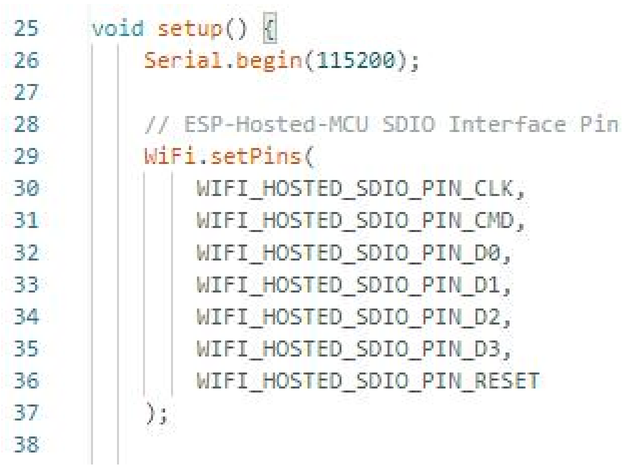
Next, the program defines two global string variables, sta_ssid and sta_password, which represent the name (SSID) and password of the WiFi network to be connected to, respectively. These parameters will be used when calling WiFi.begin() to establish the wireless connection.



```
const char* sta_ssid     = "yanfa_software";
const char* sta_password = "yanfa-123456";
/* ----------------------------------------------------------
```

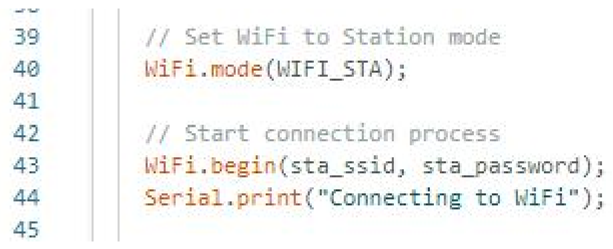
In the "setup()" function, the program first initializes the serial communication through "Serial.begin(115200)" to output debugging information in the serial monitor;

Then, it calls "WiFi.setPins()" to specify the SDIO pins used for WiFi communication. These pins are typically used for high-speed communication between the ESP32-P4 main controller and the ESP32-C6 coprocessor, including the clock line, command line, four data lines, and the reset control pin.



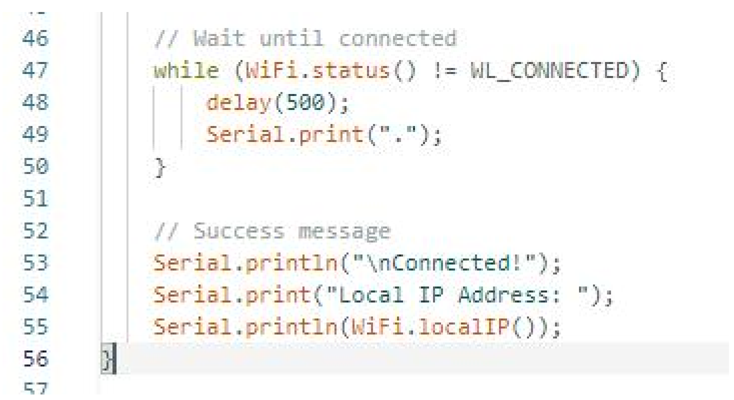
```
void setup() {
  Serial.begin(115200);

  // ESP-Hosted-MCU SDIO Interface Pin
  WiFi.setPins(
    WIFI_HOSTED_SDIO_PIN_CLK,
    WIFI_HOSTED_SDIO_PIN_CMD,
    WIFI_HOSTED_SDIO_PIN_D0,
    WIFI_HOSTED_SDIO_PIN_D1,
    WIFI_HOSTED_SDIO_PIN_D2,
    WIFI_HOSTED_SDIO_PIN_D3,
    WIFI_HOSTED_SDIO_PIN_RESET
  );

```

Next, the program sets the device to Station (client) mode by calling WiFi.mode(WIFI_STA), allowing it to connect to the router as a regular terminal. Then, it calls WiFi.begin(sta_ssid, sta_password) to start the connection to the specified wireless network.



```
  // Set WiFi to Station mode
  WiFi.mode(WIFI_STA);

  // Start connection process
  WiFi.begin(sta_ssid, sta_password);
  Serial.print("Connecting to WiFi");

```

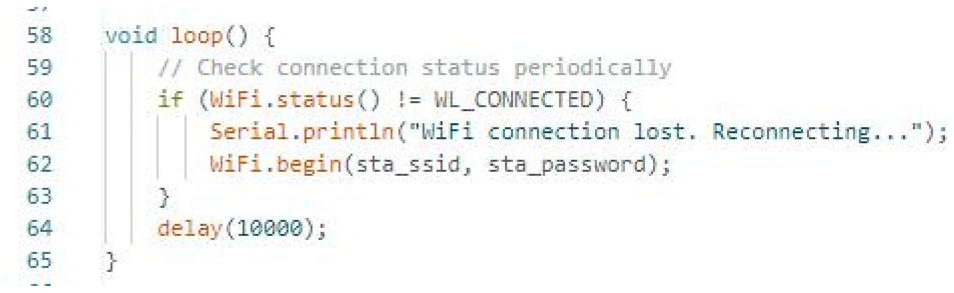
During the connection process, the program continuously checks the connection status through the while (WiFi.status() != WL_CONNECTED) loop. Every 500 milliseconds, it prints a dot on the serial port as a progress indicator for the connection. Once the connection is successful, the loop is exited and "Connected!" is printed, along with the local IP address obtained through WiFi.localIP().



```
    // Wait until connected
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }

    // Success message
    Serial.println("\nConnected!");
    Serial.print("Local IP Address: ");
    Serial.println(WiFi.localIP());
}
```

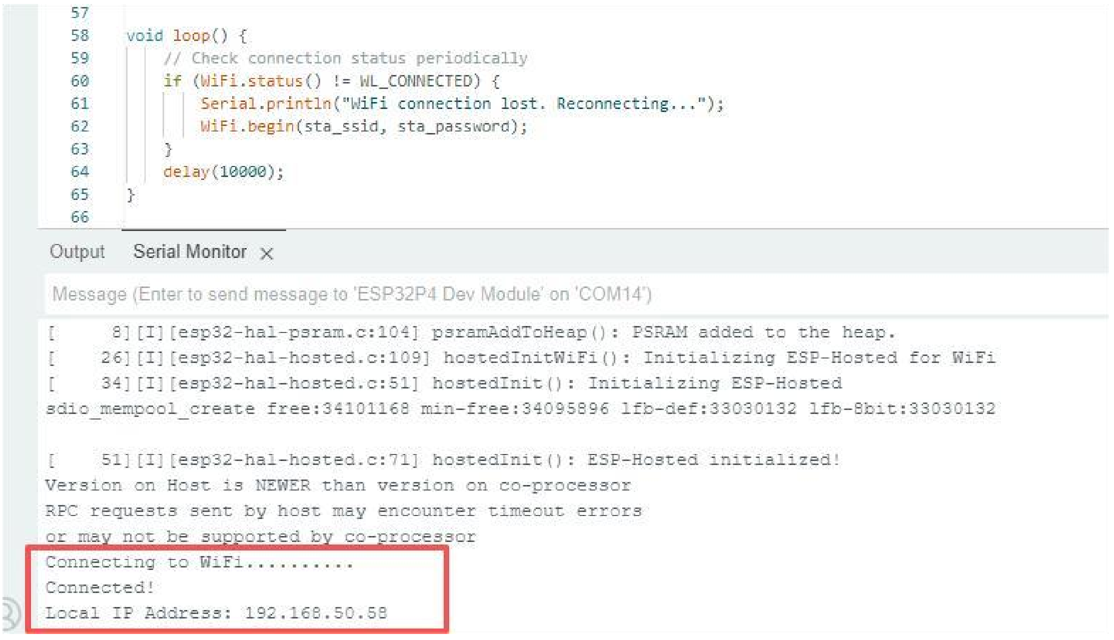
After the program enters the loop() main loop, it will check the WiFi status every 10 seconds. If it finds that WiFi.status() is not equal to WL_CONNECTED, it means that the network connection has been disconnected.

At this point, the program will print a prompt message on the serial port and call WiFi.begin() again to re-establish the connection, thereby implementing a simple WiFi automatic reconnection mechanism to ensure that the device can still reconnect after network anomalies or router restarts.



```
void loop() {
    // Check connection status periodically
    if (WiFi.status() != WL_CONNECTED) {
        Serial.println("WiFi connection lost. Reconnecting...");
        WiFi.begin(sta_ssid, sta_password);
    }
    delay(10000);
}
```

After uploading the code, you can observe the final result in the serial port monitor.


Output
Serial Monitor ×
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
void loop() {
    // Check connection status periodically
    if (WiFi.status() != WL_CONNECTED) {
        Serial.println("WiFi connection lost. Reconnecting...");
        WiFi.begin(sta_ssid, sta_password);
    }
    delay(10000);
}
```

```
[     8][I][esp32-hal-psram.c:104] psramAddToHeap(): PSRAM added to the heap.
[    26][I][esp32-hal-hosted.c:109] hostedInitWiFi(): Initializing ESP-Hosted for WiFi
[    34][I][esp32-hal-hosted.c:51] hostedInit(): Initializing ESP-Hosted
sdio_mempool_create free:34101168 min-free:34095896 lfb-def:33030132 lfb-8bit:33030132

[    51][I][esp32-hal-hosted.c:71] hostedInit(): ESP-Hosted initialized!
Version on Host is NEWER than version on co-processor
RPC requests sent by host may encounter timeout errors
or may not be supported by co-processor
Connecting to WiFi..........
Connected!
Local IP Address: 192.168.50.58
```

SoftAP (AP) mode

① Professional Definition

The SoftAP (Software Access Point) mode refers to the device simulating a Wi-Fi access point (Access Point) through software, periodically sending Beacon frames to broadcast the SSID, and acting as a DHCP Server and local gateway to construct an independent wireless local area network (WLAN).

② What does SoftAP do in engineering

In AP mode, the ESP32-C6 will:

Regular broadcast:

- SSID
- Encryption method
- Channel information

Accept client association requests

For each connected device:

- Assign IP
- Maintain connection table

That is to say:

- At this point, the ESP32 "behaves like a router"

③ Understanding SoftAP

You can think of SoftAP as:

- A "router without broadband"

It enables mobile phones to connect;

It allows for communication between devices;

But it does not access the internet by default.

④ Why are embedded devices particularly fond of using APs?

Because it solves a real engineering problem:

- When the device is powered on for the first time, it has no idea how to handle the Wi-Fi password?

Answer:

- Enable AP for the device
- Connect the mobile phone to the device
- Enter the actual Wi-Fi information
- Save to NVS
- Restart → STA mode

⑤ The actual applications in the architecture

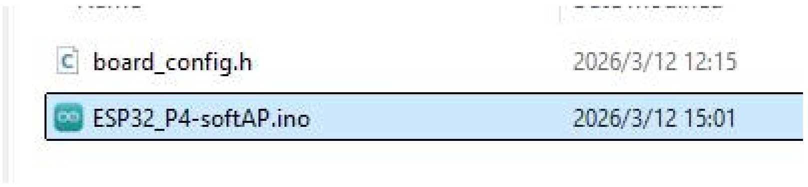
In the P4 + C6 project, the AP mode is typically used for:

- Distribution network
- Local web control
- Offline debugging

Now that we have understood the SoftAP mode, let's use it to enhance our understanding.

Click the link below to download the code for the SoftAP mode.
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson16-Wi-Fi_function/ESP32_P4-softAP

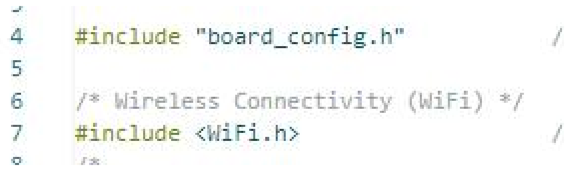
Double-click to open the code for this lesson (the ESP32_P4-softAP.ino file)



```
board_config.h          2026/3/12 12:15
ESP32_P4-softAP.ino     2026/3/12 15:01
```

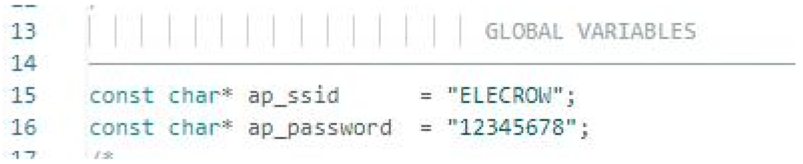
This program is an example code based on Arduino for creating a WiFi hotspot. Its main function is to enable the device to start up and create a wireless hotspot (Access Point) for other devices to connect to, and to monitor the number of clients connected to this hotspot in real time.

The initial part of the code imports the pin configuration file of the development board through "#include \"board_config.h\"", which contains the definition of the SDIO interface pins required by the WiFi module; subsequently, it imports the standard WiFi.h library through "#include <WiFi.h>". This library provides functions such as setting the WiFi mode, creating a hotspot, obtaining the IP address, and querying the number of connected devices.



```
#include "board_config.h"

/* Wireless Connectivity (WiFi) */
#include <WiFi.h>
```

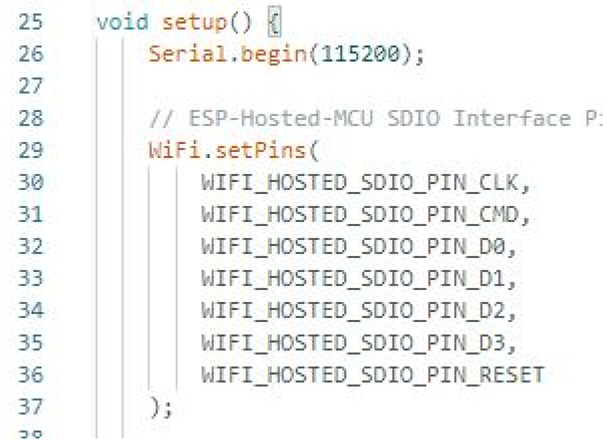
The program then defines two global string variables, ap_ssid and ap_password, which represent the name (SSID) of the WiFi hotspot created by the device and the connection password. In this example, the hotspot name is "ELECROW" and the password is "12345678". When other mobile phones or computers search for WiFi, they can see this hotspot and enter the password to connect.



```
| | | | | | | | | | | | | | GLOBAL VARIABLES
________________________________________________

const char* ap_ssid      = "ELECROW";
const char* ap_password  = "12345678";
```

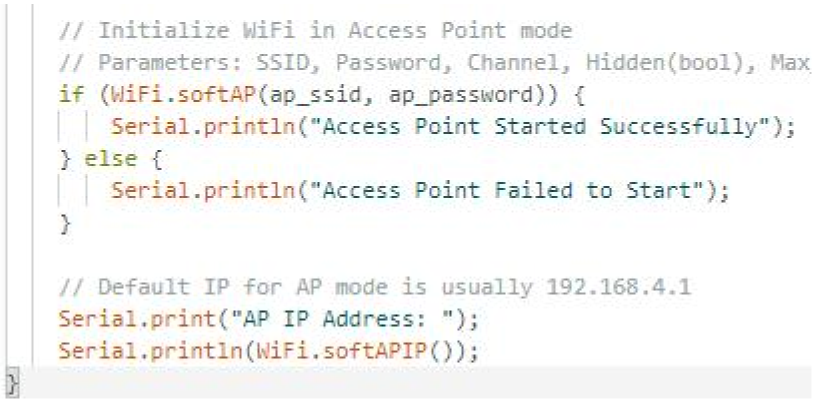
In the "setup()" function, the program first initializes the serial communication through "Serial.begin(115200)" to output the running status information in the serial monitor;

Then, it calls "WiFi.setPins()" to specify the SDIO pins used for WiFi communication. These pins are used for high-speed data communication between the main control chip ESP32-P4 and the WiFi coprocessor ESP32-C6, including the SDIO clock line, command line, four data lines, and the reset control pin.



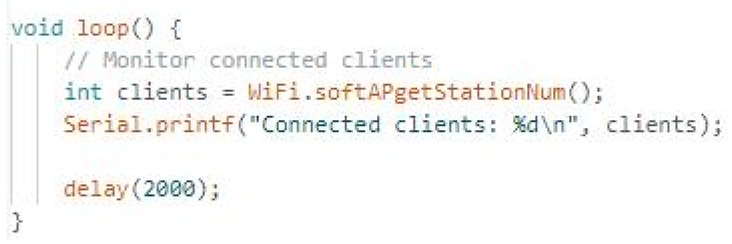
```
void setup() {
    Serial.begin(115200);

    // ESP-Hosted-MCU SDIO Interface P:
    WiFi.setPins(
        WIFI_HOSTED_SDIO_PIN_CLK,
        WIFI_HOSTED_SDIO_PIN_CMD,
        WIFI_HOSTED_SDIO_PIN_D0,
        WIFI_HOSTED_SDIO_PIN_D1,
        WIFI_HOSTED_SDIO_PIN_D2,
        WIFI_HOSTED_SDIO_PIN_D3,
        WIFI_HOSTED_SDIO_PIN_RESET
    );
```

Then, the program calls WiFi.softAP(ap_ssid, ap_password) to set the device to AP (Access Point) mode and create a wireless hotspot. If the hotspot is created successfully, it prints "Access Point Started Successfully" on the serial port; otherwise, it prints the failure information for the startup.



```
    // Initialize WiFi in Access Point mode
    // Parameters: SSID, Password, Channel, Hidden(bool), Max
    if (WiFi.softAP(ap_ssid, ap_password)) {
        Serial.println("Access Point Started Successfully");
    } else {
        Serial.println("Access Point Failed to Start");
    }

    // Default IP for AP mode is usually 192.168.4.1
    Serial.print("AP IP Address: ");
    Serial.println(WiFi.softAPIP());
}
```

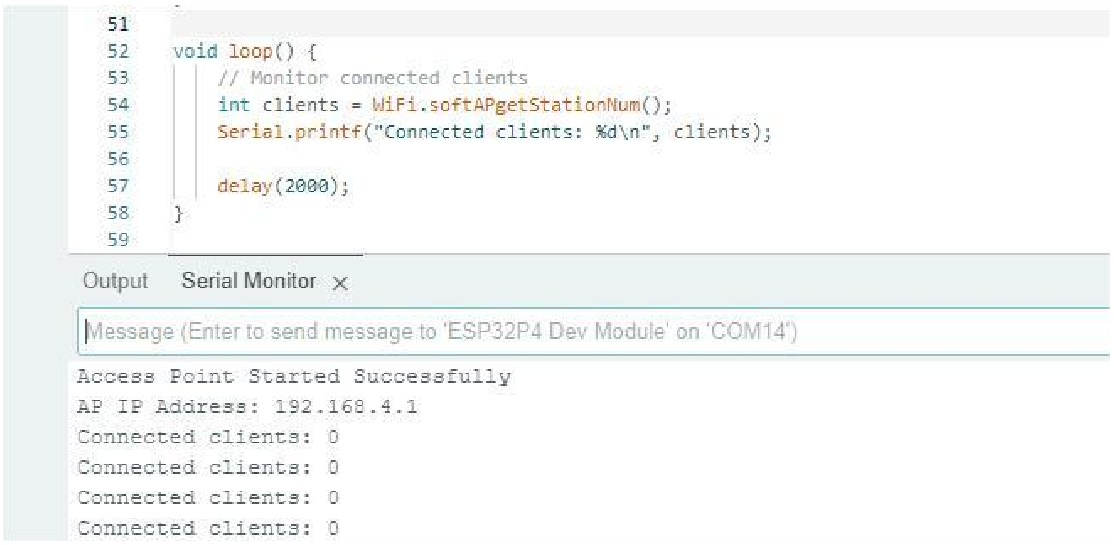
Once the hotspot is created successfully, the program uses WiFi.softAPIP() to obtain the IP address of the device in the hotspot mode and outputs it to the serial port. The default address is generally 192.168.4.1. Devices connected to this hotspot can communicate with the device via this address over the network.



```
void loop() {
    // Monitor connected clients
    int clients = WiFi.softAPgetStationNum();
    Serial.printf("Connected clients: %d\n", clients);

    delay(2000);
}
```

After entering the loop() main loop, the program will periodically call WiFi.softAPgetStationNum() to obtain the number of clients currently connected to this hotspot, and print it on the serial port every 2 seconds using Serial.printf(), such as "Connected clients: 2", thereby achieving real-time monitoring of the number of connected devices.

After uploading the code, you can connect to this Wi-Fi using your mobile phone.

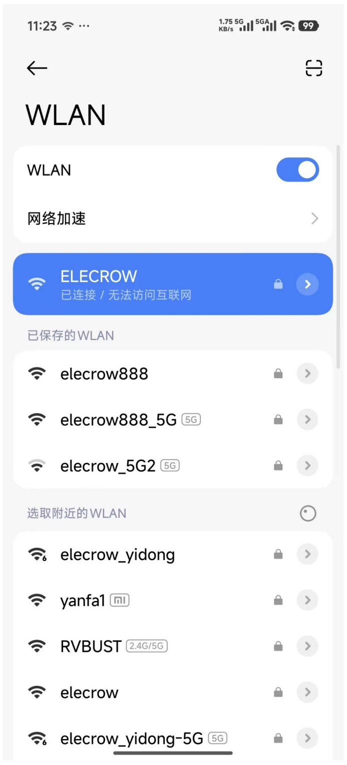


```
void loop() {
    // Monitor connected clients
    int clients = WiFi.softAPgetStationNum();
    Serial.printf("Connected clients: %d\n", clients);

    delay(2000);
}
```

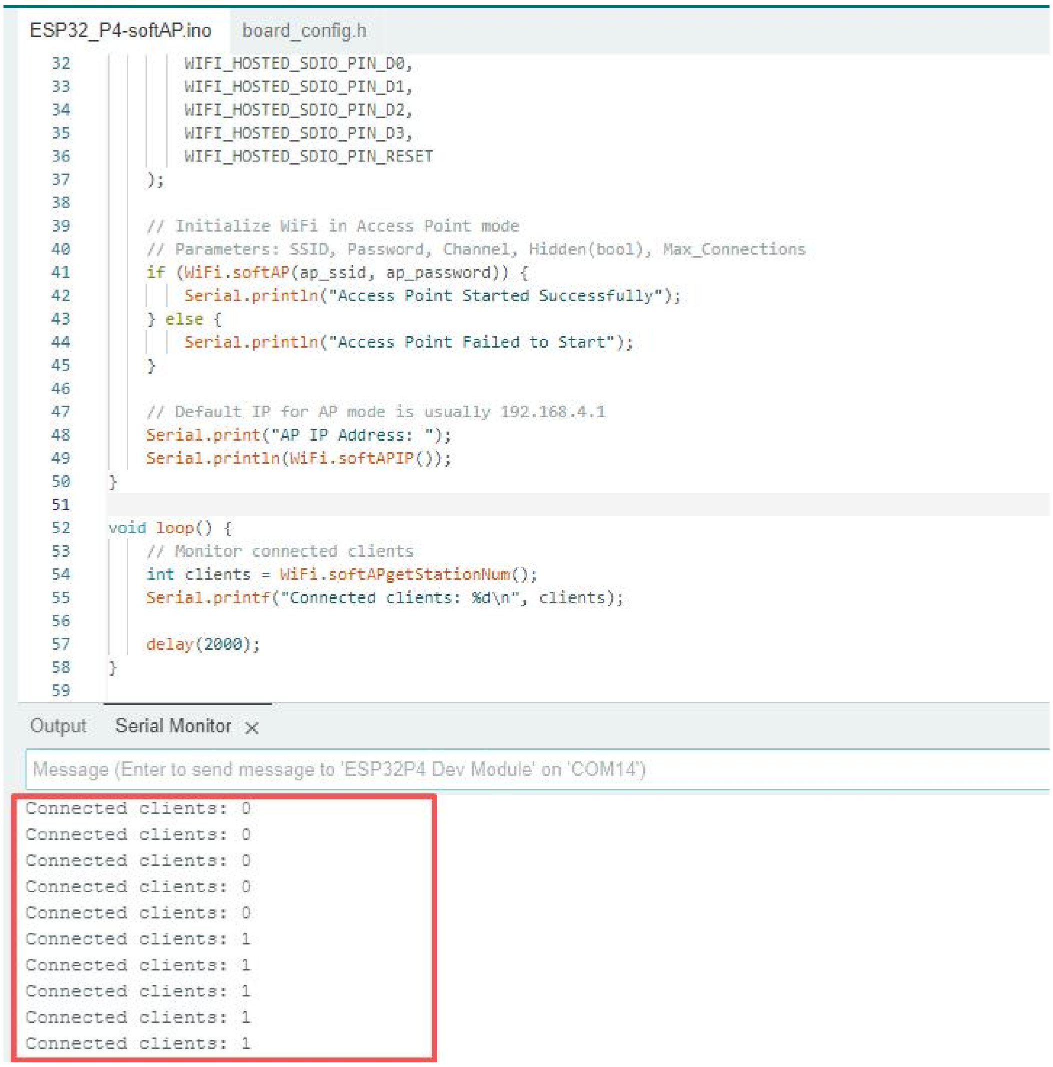
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
Access Point Started Successfully
AP IP Address: 192.168.4.1
Connected clients: 0
Connected clients: 0
Connected clients: 0
Connected clients: 0
```

At this moment, you should use your mobile phone or laptop to connect to the Wi-Fi of this router.



After waiting for the connection to be successful, you can see that it is connected.



ESP32_P4-softAP.ino   board_config.h

```
            WIFI_HOSTED_SDIO_PIN_D0,
            WIFI_HOSTED_SDIO_PIN_D1,
            WIFI_HOSTED_SDIO_PIN_D2,
            WIFI_HOSTED_SDIO_PIN_D3,
            WIFI_HOSTED_SDIO_PIN_RESET
        );

        // Initialize WiFi in Access Point mode
        // Parameters: SSID, Password, Channel, Hidden(bool), Max_Connections
        if (WiFi.softAP(ap_ssid, ap_password)) {
            Serial.println("Access Point Started Successfully");
        } else {
            Serial.println("Access Point Failed to Start");
        }

        // Default IP for AP mode is usually 192.168.4.1
        Serial.print("AP IP Address: ");
        Serial.println(WiFi.softAPIP());
    }

    void loop() {
        // Monitor connected clients
        int clients = WiFi.softAPgetStationNum();
        Serial.printf("Connected clients: %d\n", clients);

        delay(2000);
    }
```

Output   Serial Monitor

Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
Connected clients: 0
Connected clients: 0
Connected clients: 0
Connected clients: 0
Connected clients: 0
Connected clients: 1
Connected clients: 1
Connected clients: 1
Connected clients: 1
Connected clients: 1
```

However, this Wi-Fi has no network available for access. You can think of it as a router without an Ethernet cable. You can try to connect it, but you won't be able to access the internet.

STA + AP coexistence mode

① Professional Definition

The STA + AP coexistence mode (Station + SoftAP Concurrency) refers to the scenario on a single-radio-frequency (Single RF) Wi-Fi SoC where:

- A Station Interface (STA)
- A SoftAP Interface (AP)

are simultaneously running through a virtual network interface (Virtual Wi-Fi Interface, VIF), and the MAC layer scheduler (MAC Scheduler) switches the transmission and reception between the two roles through time-division multiplexing (TDM) in a time-sharing manner.

② Underlying Facts

TA + AP ≠ Two Wi-Fi modules

Rather:

- One set of RF
- One set of PHY
- One set of MAC
- Two logical identities

The ESP32-C6 "pretends" to be two devices at the protocol level.

③ How is the protocol and system layer implemented?

Virtual Interface (Virtual Interface, VIF)

In ESP-IDF, STA + AP is actually: Wi-Fi Driver

- netif_sta (STA interface)
- netif_ap (AP interface)

Each interface has:

- Independent MAC state machine
- Independent IP stack (lwIP)
- Independent DHCP behavior

However, they share the same radio frequency hardware.

④ What is Time Division Multiplexing (TDM) doing?

Since there is only one antenna, the ESP32-C6 will:

- Sometimes "listen to the router"
- Sometimes "send Beacons to the mobile phone"
- Sometimes receive data from the cloud
- Sometimes respond to the phone's requests

You can think of it as: The Wi-Fi chip rapidly switches between two identities.

This switching is transparent to the upper-layer applications.

⑤ Channel Binding

In the STA + AP mode, the channel of the AP must be the same as the channel of the router that the STA is connected to. Reason:

- Single RF cannot operate simultaneously on two channels.
- The STA has a higher priority than the AP.

Therefore:

- The router is on Channel 6.
- The SoftAP can only be on Channel 6.

⑥ Understanding STA + AP

Think of the device as a person:

➢ STA:

Using a mobile phone to call the company headquarters (the external world)

➢ AP:

At the same time, using a walkie-talkie, and everyone nearby can call you

One action, two communication methods.

⑦ The fundamental difference from the standalone STA/AP mode

The standalone STA mode only allows the ESP32 to act as a Wi-Fi client and connect to an existing wireless network outside, with the core requirement being "accessing the network to enable external communication";

The standalone AP mode only enables the ESP32 to become a Wi-Fi hotspot itself, with the core requirement being "building a local local area network for other devices to access";

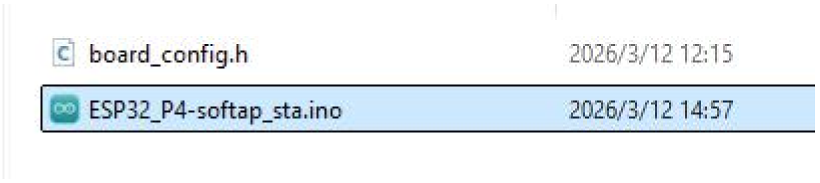
While the STA+AP coexistence mode simultaneously enables these two working modes on a single chip, its essence is to balance the "accessing an external network to achieve remote/external network communication" and "building a local hotspot to enable local device interaction" capabilities. Although it experiences a slight decrease in throughput due to radio frequency time division multiplexing, it can simultaneously meet the dual scenarios of remote operation, local debugging/ networking configuration, etc., expanding the functional boundaries of a single mode.

Now that we have understood the Sta+AP mode, let's use it to reinforce our understanding.

Click the link below to download the code for the Sta+AP mode.

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code_5inch/Lesson16-Wi-Fi_function/ESP32_P4-softap_sta

Double-click to open the code (ino file) for this lesson.



board_config.h	2026/3/12 12:15
ESP32_P4-softap_sta.ino	2026/3/12 14:57

This program is a network example code based on Arduino and WiFi.h. Its core function is to enable the device to simultaneously connect to external WiFi as a client (STA) and act as a hotspot (AP) for other devices to connect to, while also achieving network sharing through NAT technology.

The program begins by including "board_config.h" to import the pin configuration file of the development board, which defines the hardware pins required for WiFi SDIO communication;

Then, it includes "WiFi.h" to import the WiFi control library for ESP32, and includes "lwip/lwip_napt.h" to import the library required for NAT (Network Address Translation) network address translation functionality. This library comes from the lwIP network protocol stack and is used to implement network forwarding within the device.

```
#include "board_config.h"          //

/* Wireless Connectivity (WiFi) */
#include <WiFi.h>                  //
#include <lwip/lwip_napt.h>        //
```

Subsequently, the program defined four global variables: ap_ssid and ap_password are used to set the name and password of the hotspot created by the device (in this example, the hotspot is named "ELECROW"), while sta_ssid and sta_password are used to set the external router WiFi that the device needs to connect to (in this case, it is "yanfa_software").

```
const char* ap_ssid      = "ELECROW";
const char* ap_password  = "12345678";

const char* sta_ssid     = "yanfa_software";
const char* sta_password = "yanfa-123456";
/*
```

In the setup() initialization function, the program first starts the serial debugging interface through Serial.begin(115200) to output the device operation information in the serial monitor;

Then, it calls WiFi.setPins() to specify the SDIO interface pins used for communication by the WiFi module. These pins are used for high-speed data communication between the main control ESP32-P4 and the WiFi coprocessor ESP32-C6, including the clock line, command line, four data lines, and the reset control pin.

```
void setup() {
    Serial.begin(115200);

    // ESP-Hosted-MCU SDIO Interface Pins
    WiFi.setPins(
        WIFI_HOSTED_SDIO_PIN_CLK,
        WIFI_HOSTED_SDIO_PIN_CMD,
        WIFI_HOSTED_SDIO_PIN_D0,
        WIFI_HOSTED_SDIO_PIN_D1,
        WIFI_HOSTED_SDIO_PIN_D2,
        WIFI_HOSTED_SDIO_PIN_D3,
        WIFI_HOSTED_SDIO_PIN_RESET
    );
```

Next, the program calls WiFi.mode(WIFI_AP_STA), which sets the WiFi operation mode to the AP + STA dual mode. This means that the device can either connect to an existing router to access the network or create its own hotspot for other devices to connect to.

```
// 1. Set mode to AP + STA
WiFi.mode(WIFI_AP_STA);
```

Then, call WiFi.softAP(ap_ssid, ap_password) to start the hotspot, allowing mobile phones or computers to search for and connect to the "ELECROW" WiFi network.

```
// 2. Start AP
WiFi.softAP(ap_ssid, ap_password);
```

Then, using WiFi.begin(sta_ssid, sta_password), connect to the external router. And through the while (WiFi.status() != WL_CONNECTED) loop, continuously check the connection status. Before the connection is successful, print a "." to the serial port every 500ms as a progress indication of the connection.

```
// 3. Connect to Router
WiFi.begin(sta_ssid, sta_password);

while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
}
```

After the device successfully connects to the router, the program calls ip_napt_enable(WiFi.softAPIP(), 1) to enable the NAT (Network Address Translation) function. This step is the key to the entire program. It forwards the data traffic from the hotspot network (AP) to the router network (STA), thereby enabling the mobile phones or computers connected to the device's hotspot to access the Internet through this device, which is equivalent to turning the device into a simple WiFi repeater or a mini router.

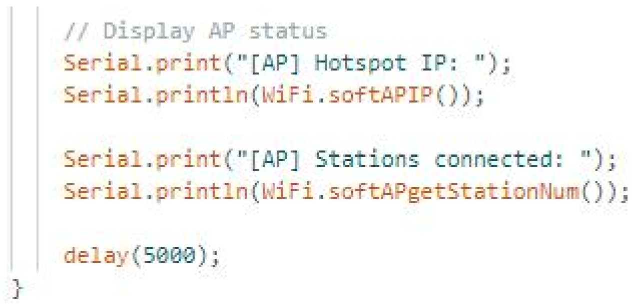
```
    // 4. ENABLE NAT (The magic happens here)
    // This allows traffic to flow from AP to STA
    ip_napt_enable(WiFi.softAPIP(), 1);

    Serial.println("\nNAT Enabled. Mobile device should have internet now.");
}
```

After entering the loop() main loop, the program prints the network status information every 5 seconds: First, it checks whether the STA mode has successfully connected to the router. If the connection is successful, it prints the IP address of the device in the router's network WiFi.localIP(), otherwise it prompts that it is connecting.

```
void loop() {
    // Check Station connection status
    if (WiFi.status() == WL_CONNECTED) {
        Serial.print("[STA] Connected. IP: ");
        Serial.println(WiFi.localIP());
    } else {
        Serial.println("[STA] Connecting to router...");
    }
```

Then print the IP address of the hotspot mode using WiFi.softAPIP() (usually

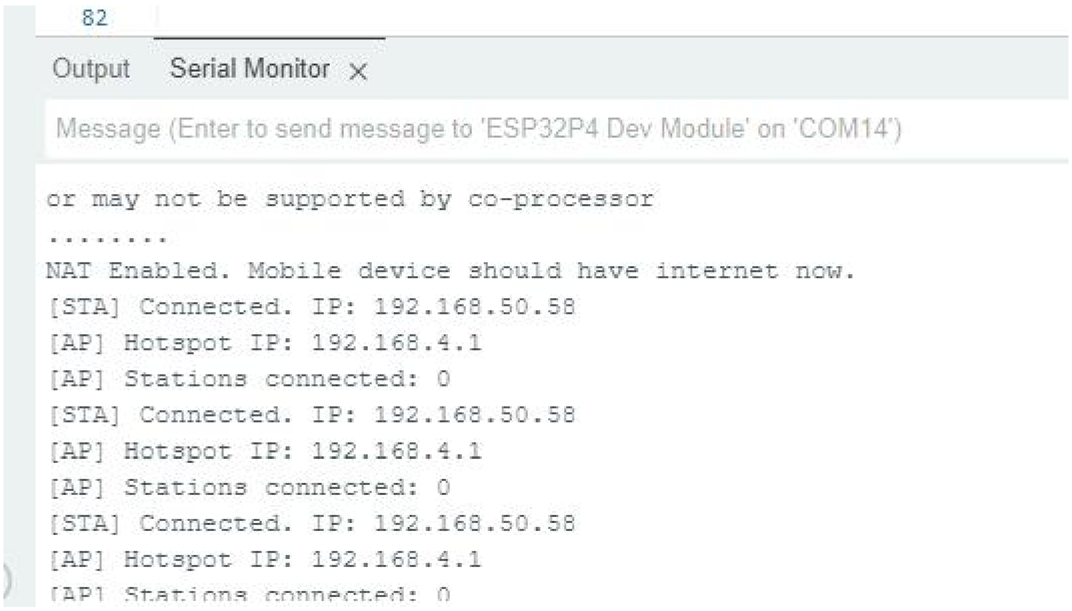
192.168.4.1), and display through WiFi.softAPgetStationNum() how many devices are currently connected to this hotspot.



```
    // Display AP status
    Serial.print("[AP] Hotspot IP: ");
    Serial.println(WiFi.softAPIP());

    Serial.print("[AP] Stations connected: ");
    Serial.println(WiFi.softAPgetStationNum());

    delay(5000);
}
```

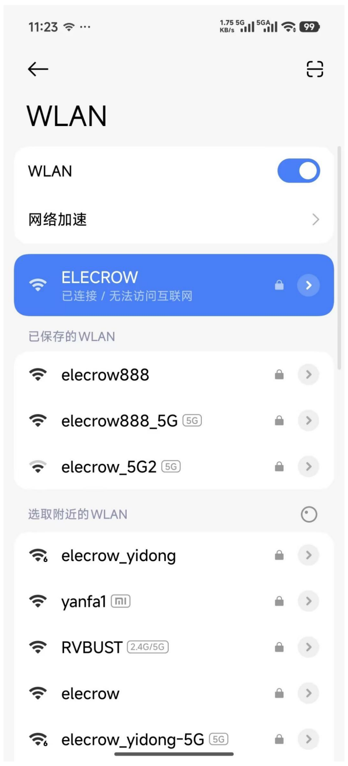
Overall, this code implements functions such as WiFi hotspot creation, router connection, NAT network sharing, and device monitoring, enabling the device to act as a small wireless gateway and providing internet access capabilities for devices connected to its hotspot.

After uploading the code, you can first open the serial port monitor to observe the process of connecting to Wi-Fi.

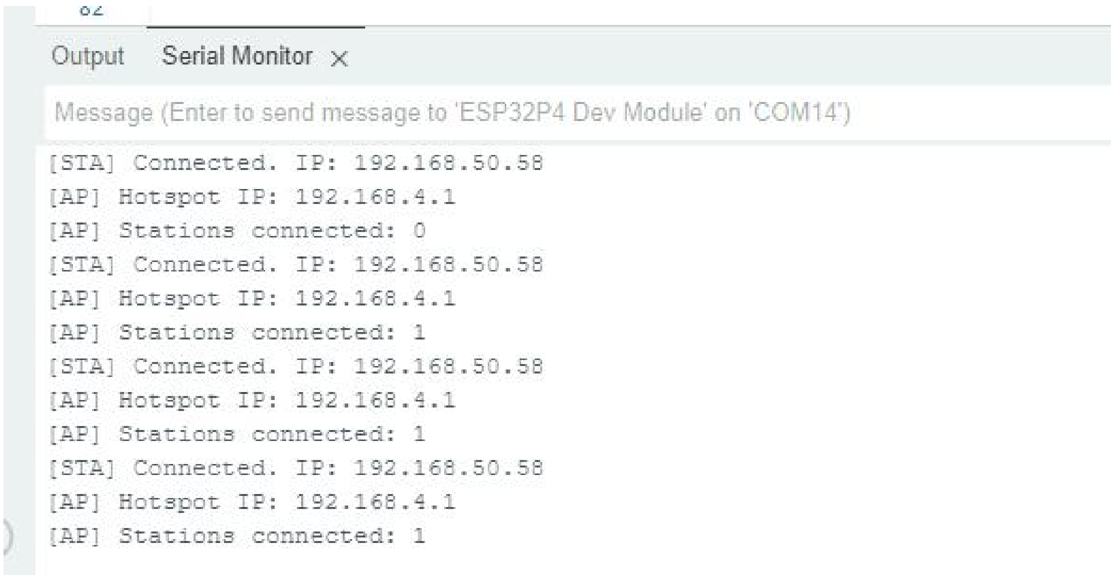

Output
Serial Monitor ×
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
or may not be supported by co-processor
........
NAT Enabled. Mobile device should have internet now.
[STA] Connected. IP: 192.168.50.58
[AP] Hotspot IP: 192.168.4.1
[AP] Stations connected: 0
[STA] Connected. IP: 192.168.50.58
[AP] Hotspot IP: 192.168.4.1
[AP] Stations connected: 0
[STA] Connected. IP: 192.168.50.58
[AP] Hotspot IP: 192.168.4.1
[AP] Stations connected: 0
```

Then you can also use your mobile phone or laptop to connect to the Wi-Fi of this router.



After the connection is successful, the current number of connected devices will also be displayed.


Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')

```
[STA] Connected. IP: 192.168.50.58
[AP] Hotspot IP: 192.168.4.1
[AP] Stations connected: 0
[STA] Connected. IP: 192.168.50.58
[AP] Hotspot IP: 192.168.4.1
[AP] Stations connected: 1
[STA] Connected. IP: 192.168.50.58
[AP] Hotspot IP: 192.168.4.1
[AP] Stations connected: 1
[STA] Connected. IP: 192.168.50.58
[AP] Hotspot IP: 192.168.4.1
[AP] Stations connected: 1
```

ELECROW

MAKE YOUR MAKING EASIER