CrowPanel Advanced

7inch/9inch/10.1inch ESP32-P4 HMI

www.elecrow.com

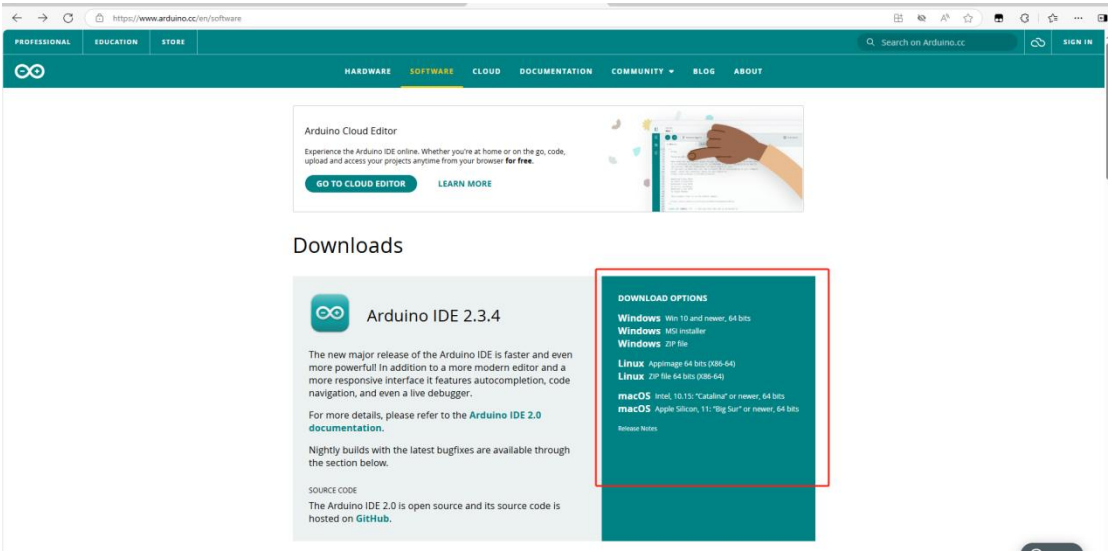
Preface

Installing Arduino IDE

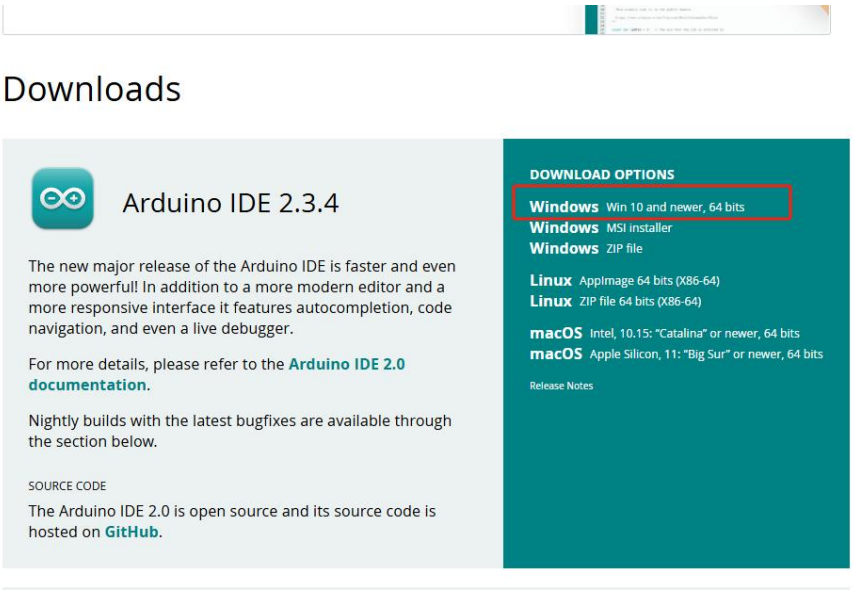
1. Open the official Arduino website

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

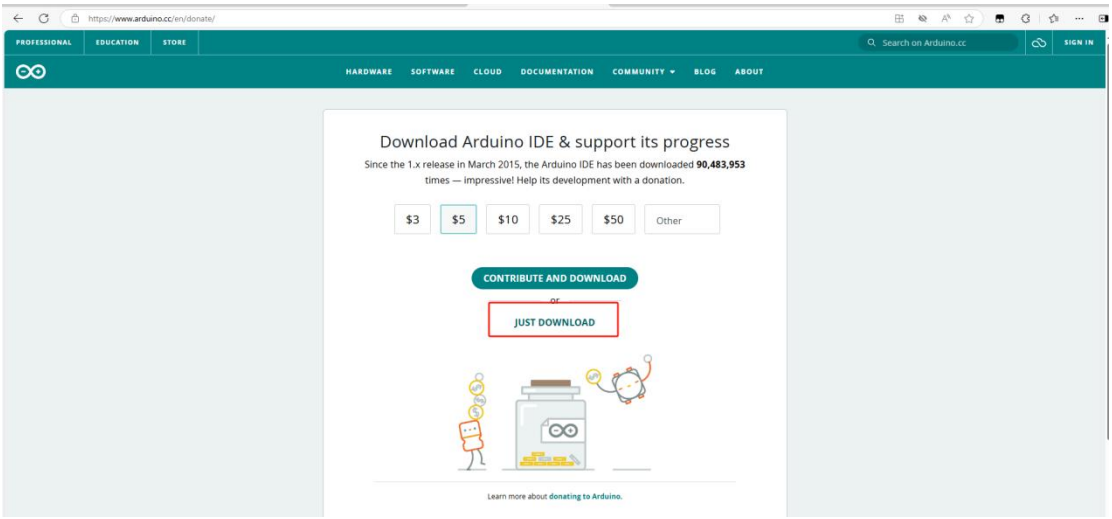
2. Select and install the appropriate version according to your operating system.



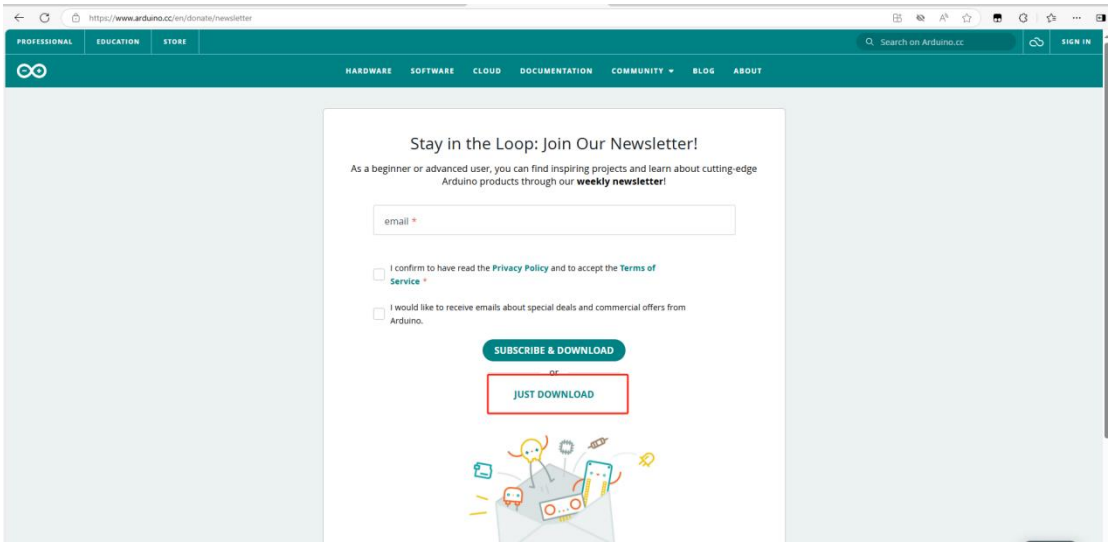
2.1 Here I choose the first one.



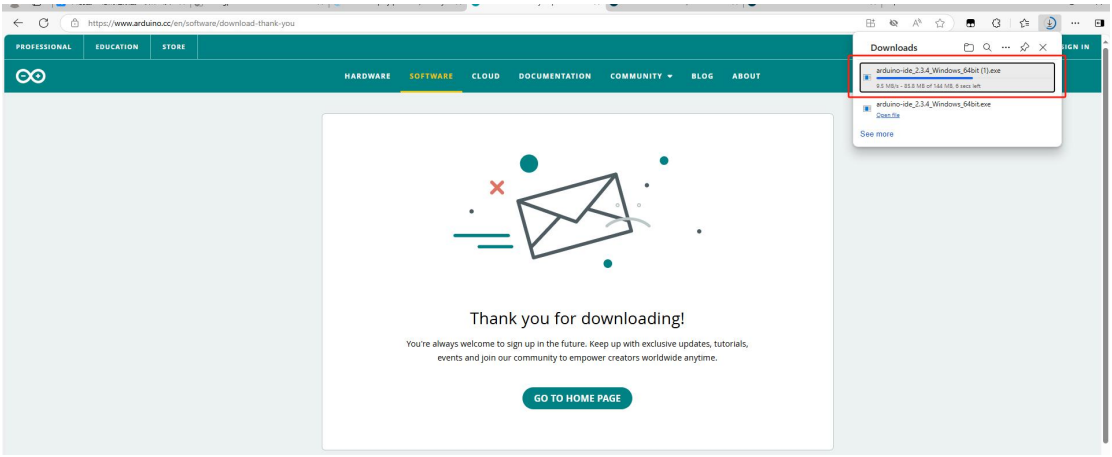
3. Select "Download Only" if you wish.



3.3 The same as above.

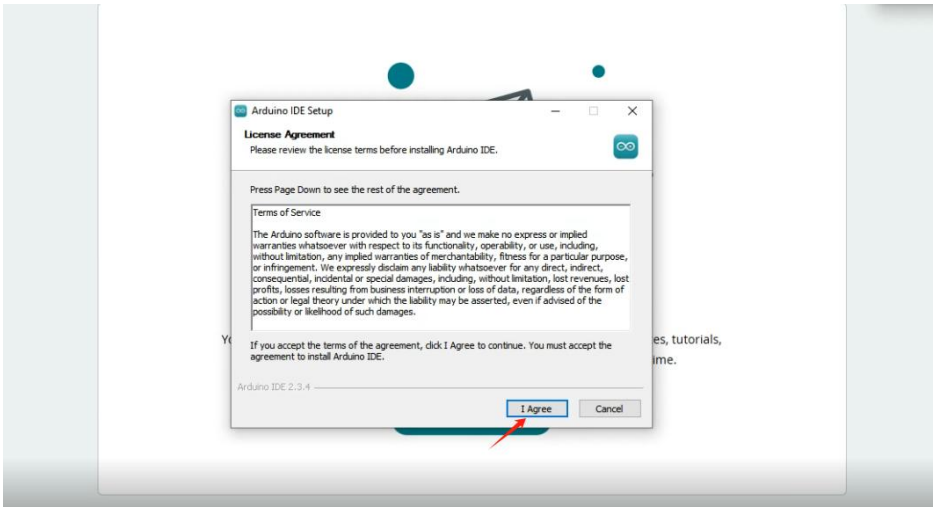


4. Wait for the download to complete

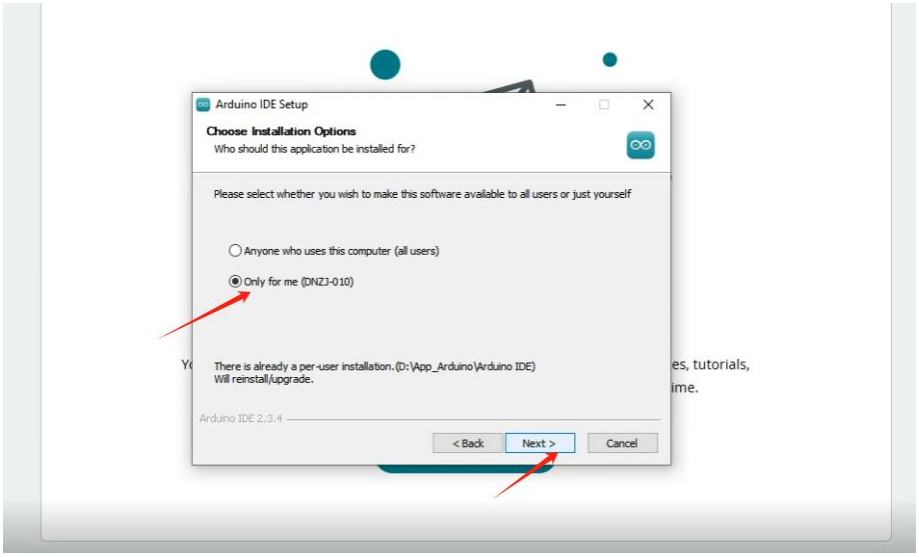


5. Confirm the download process

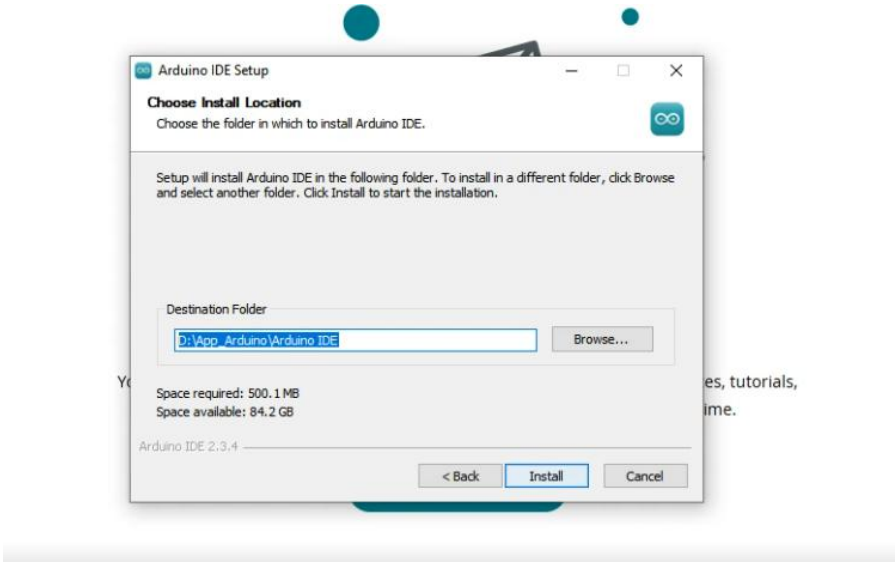
5.1 Accept the options



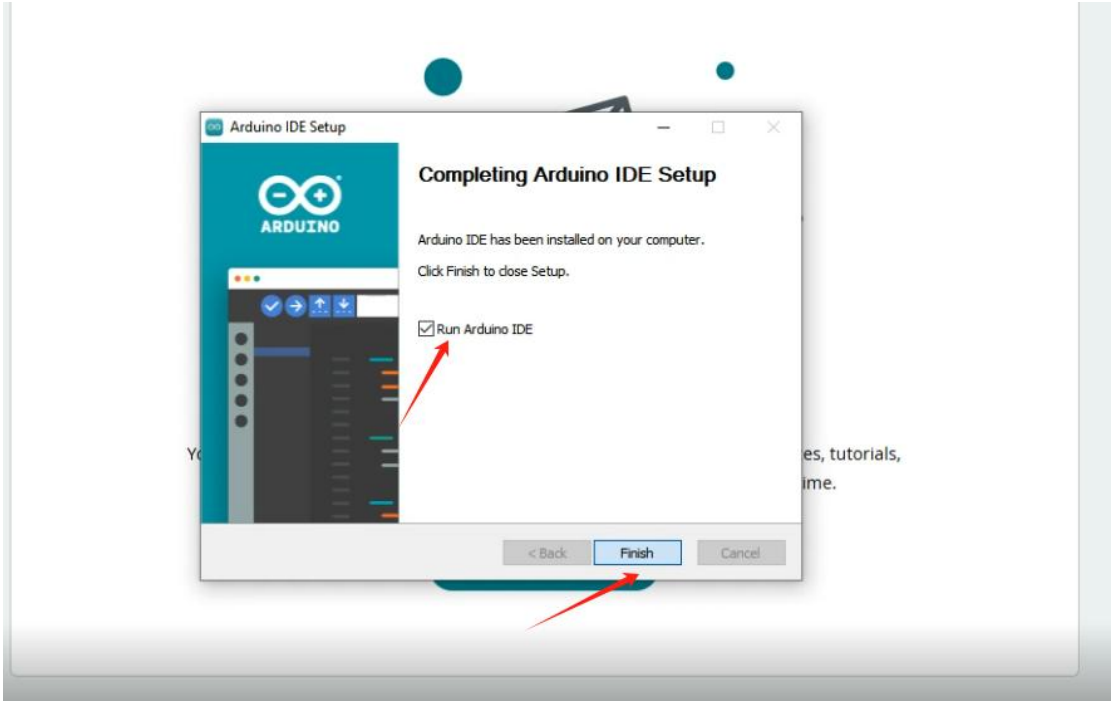
5.2 As shown in the figure, determine



5.3 Customize your path and click "Install"

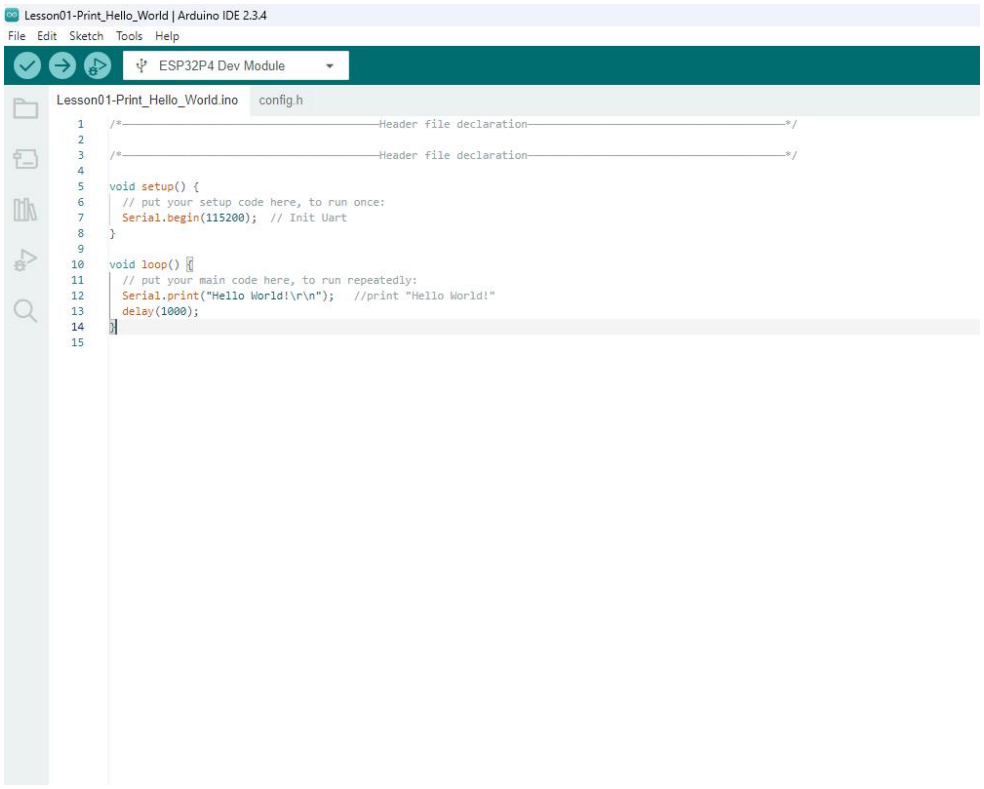


5.4 After the installation is completed, you can start using the Arduino IDE.

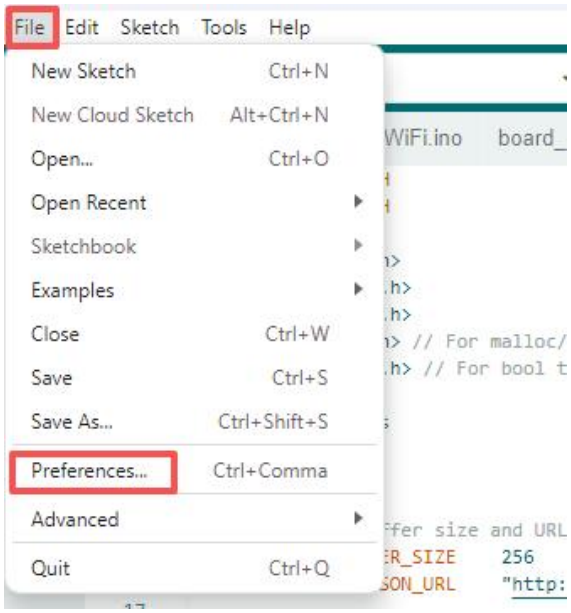


Installing the ESP32-P4 Development Board

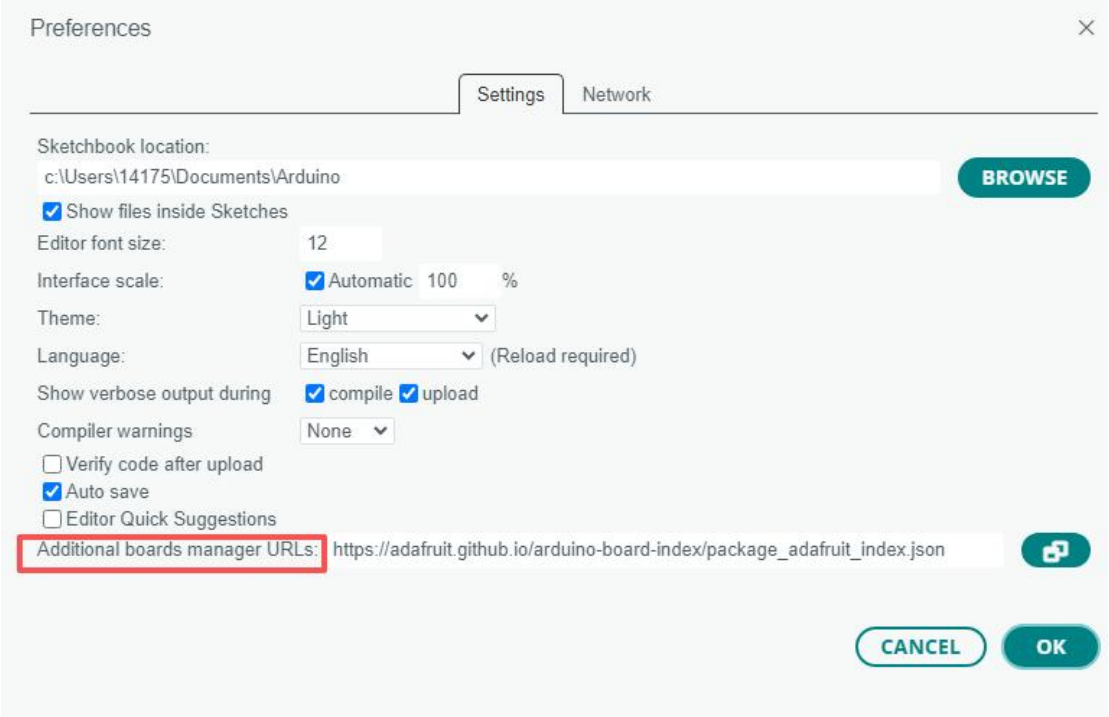
After the installation is completed, open the Arduino IDE.
First, randomly open a project.



Go to "Preferences".

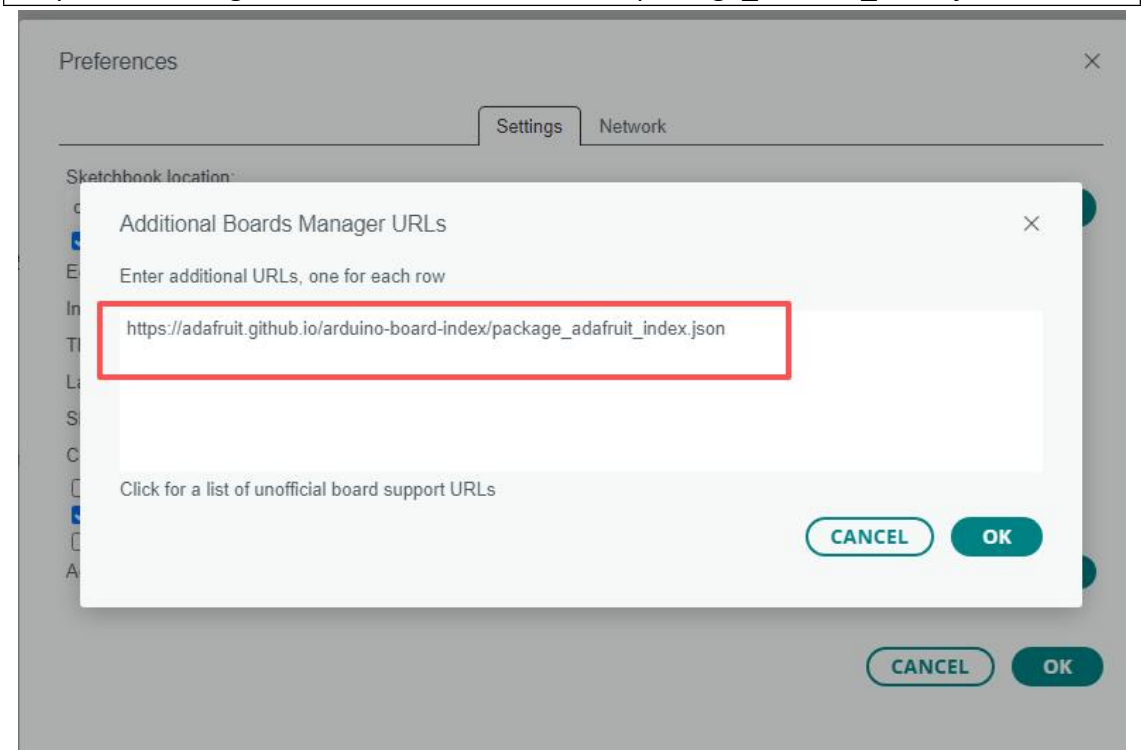


Configure "Additional Boards Manager URLs"

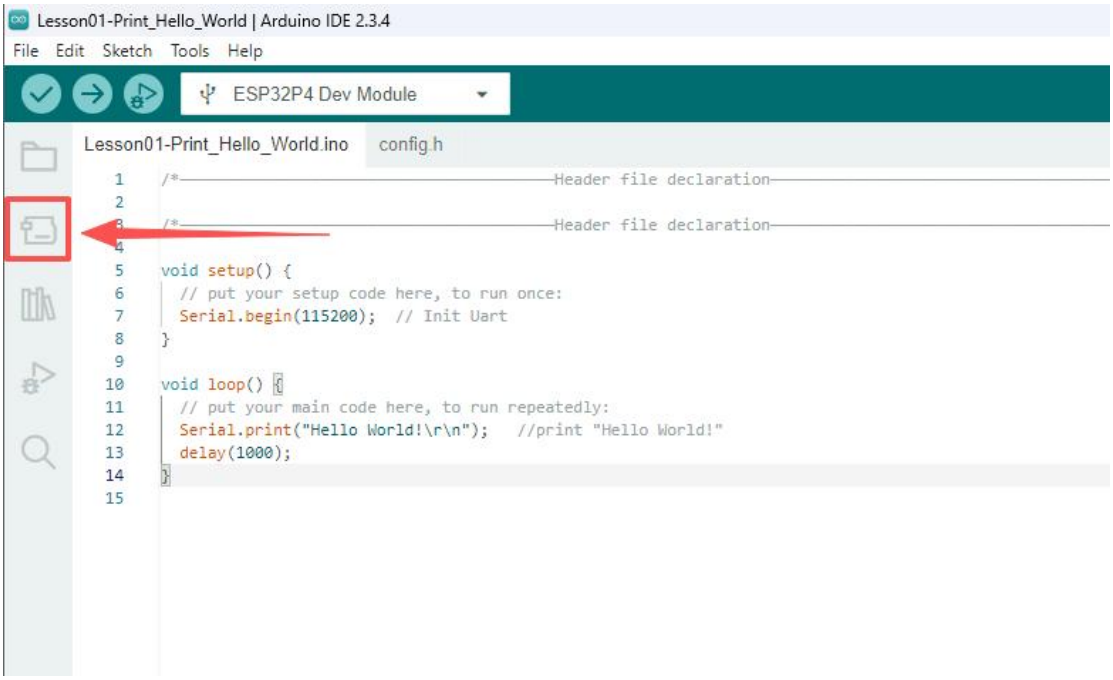


Add this in.

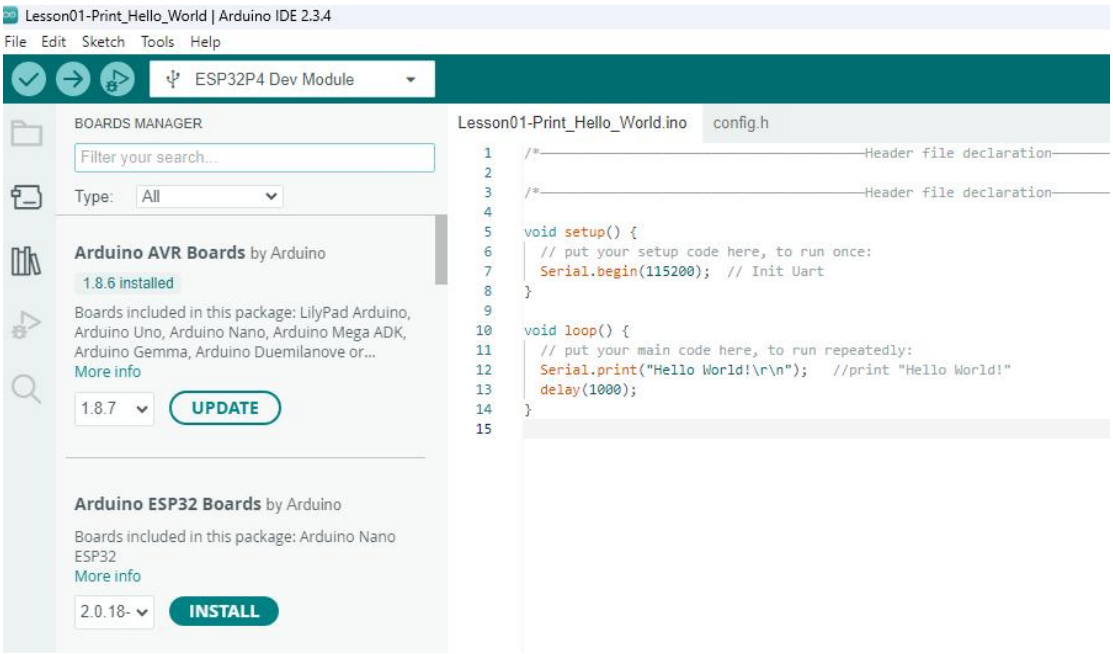
https://adafruit.github.io/arduino-board-index/package_adafruit_index.json



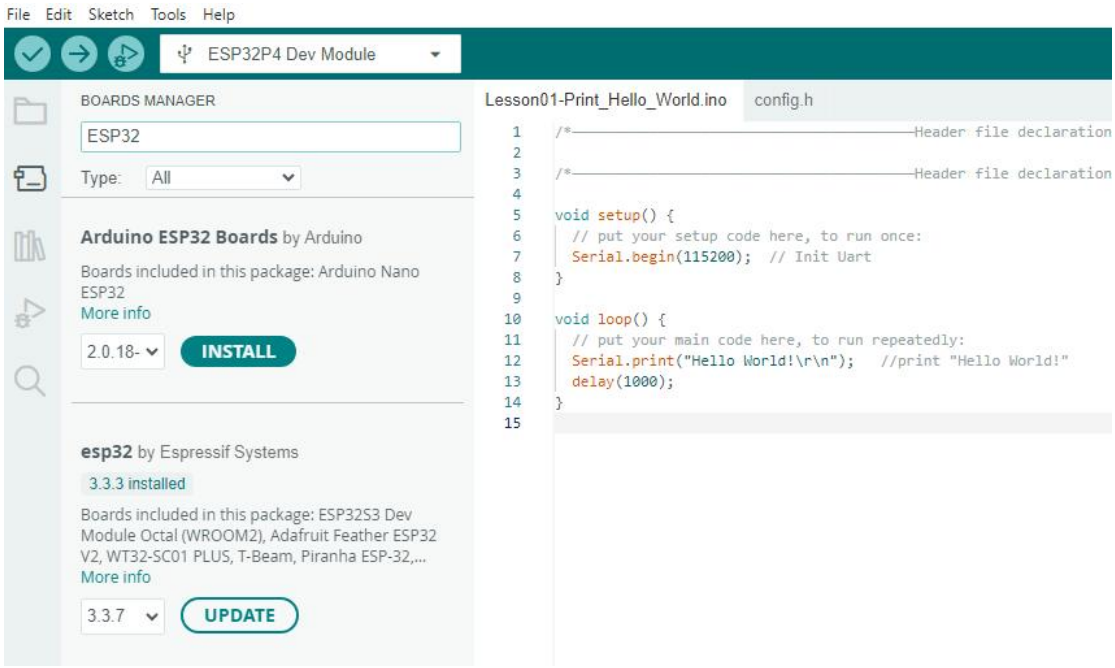
First, click on "board manage" on the left side.



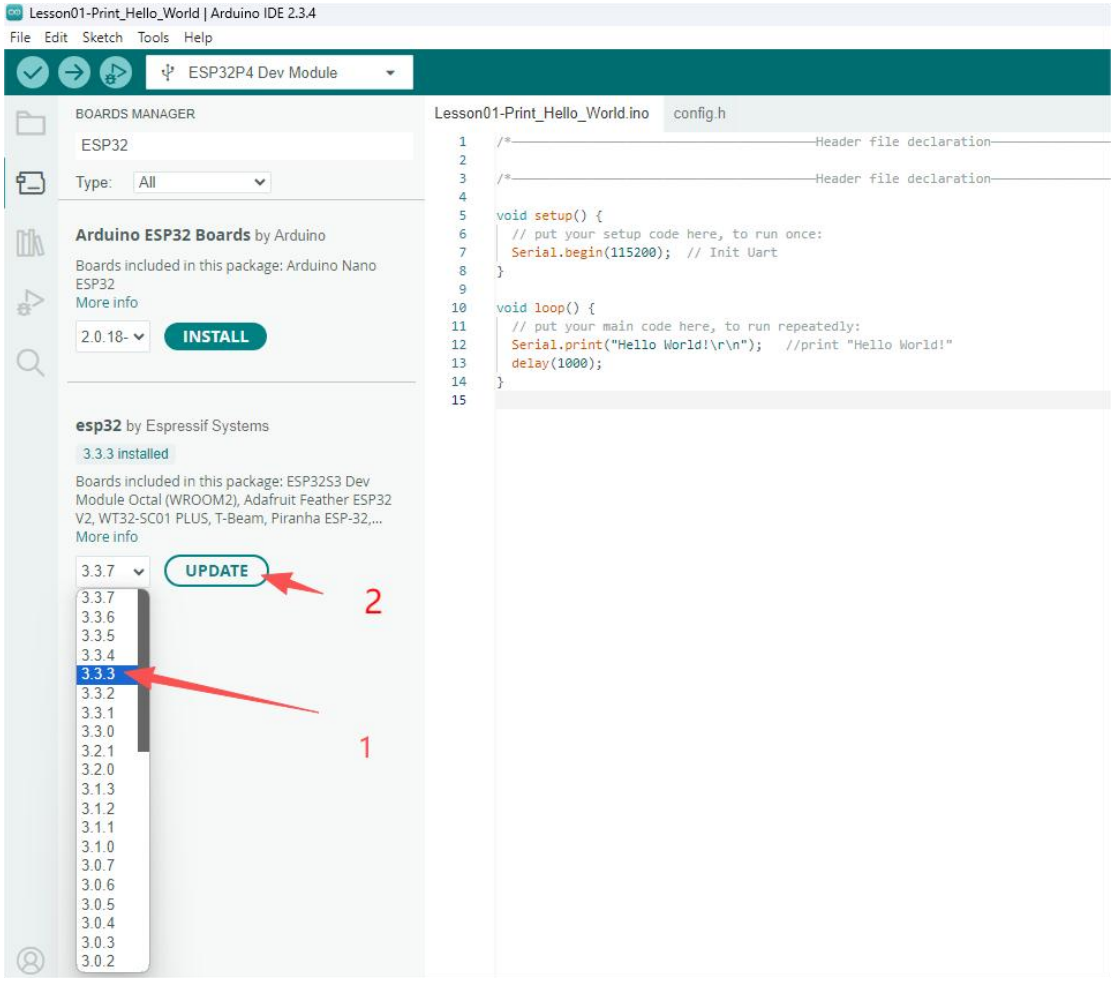
After opening



Search for ESP32 here



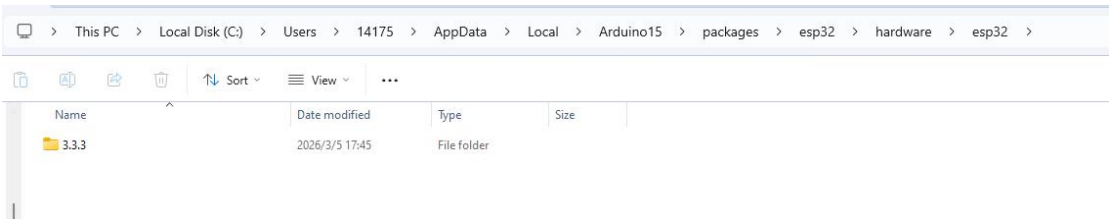
Select version 3.3.3 here and proceed with the installation.



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.



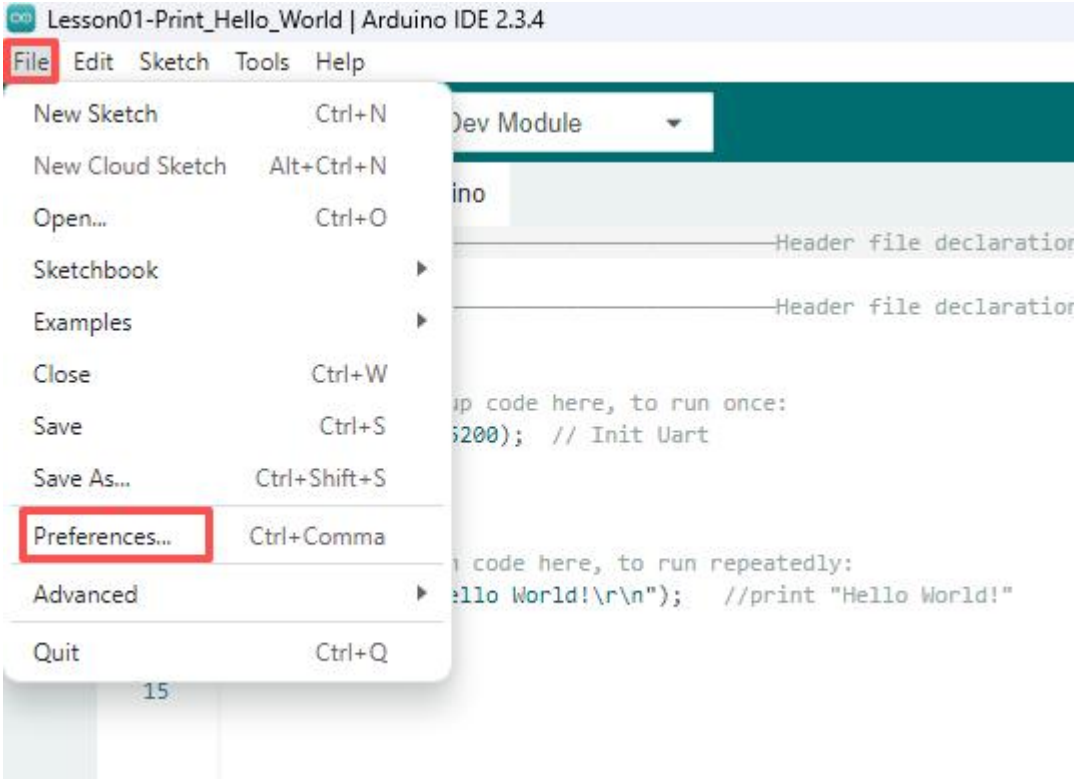
In this way, you have completed the necessary preparations for compiling the code in the Arduino IDE. Next, all you need to do is write your appropriate code and use the corresponding library files to run it.

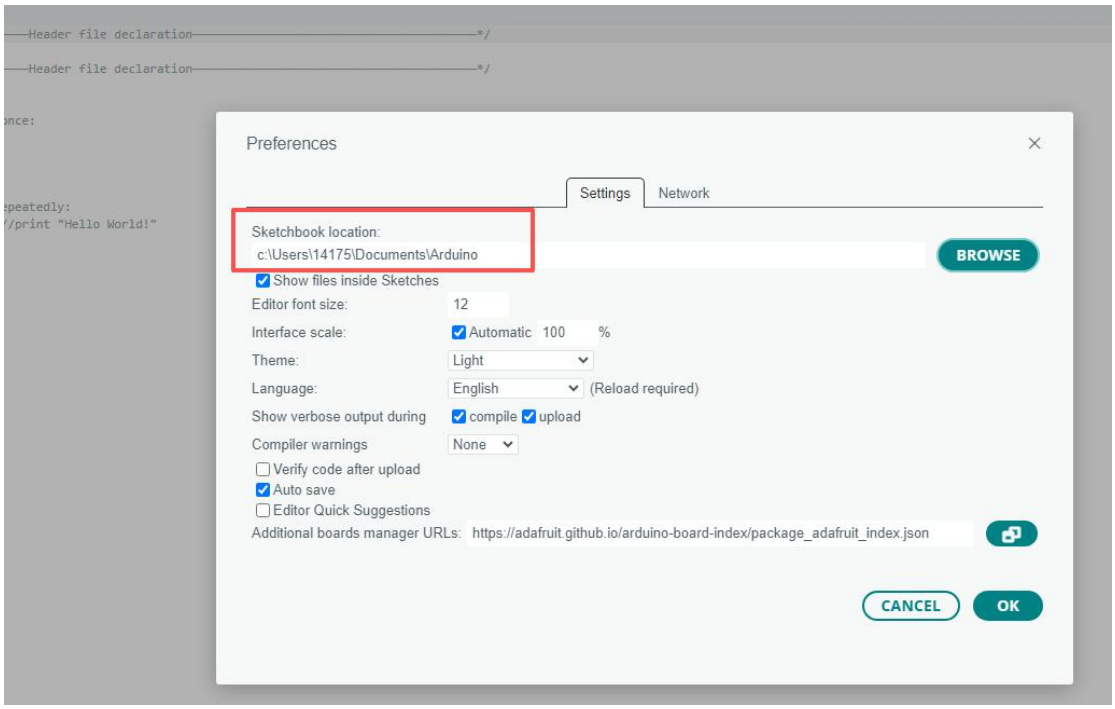
Importing Library Files (Important)

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-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/libraries

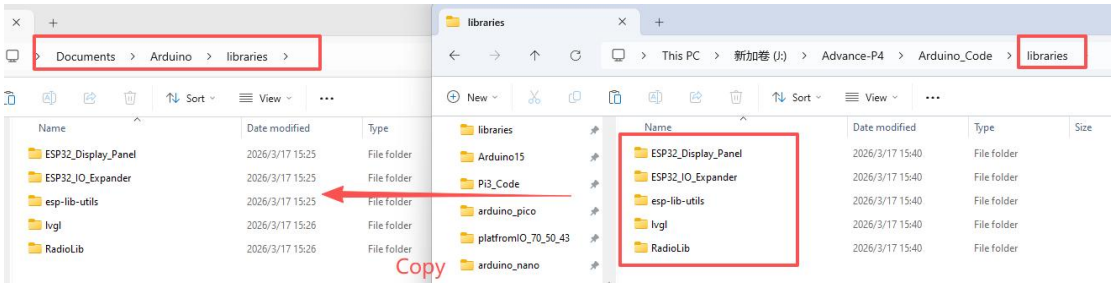
Enter this path and check the location where the library files are stored.





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.



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

Lesson01--- Print "Hello World"

Introduction

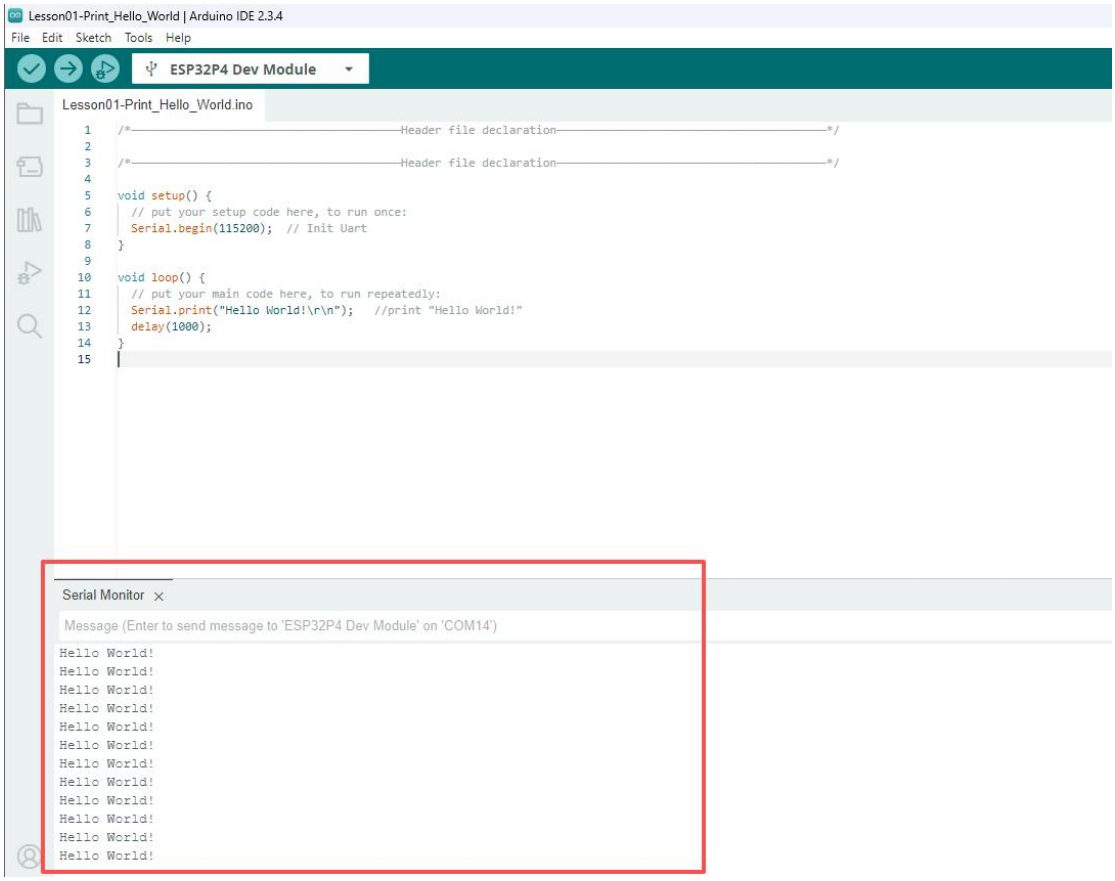
In this lesson, we will officially begin writing code in the Arduino-IDE environment to drive the Advance-P4 development board. Subsequent lessons will follow a gradual "from simple to complex" design, helping you progressively master the Arduino-IDE development framework and the usage logic of the ESP32-P4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



Hardware Used in This Lesson

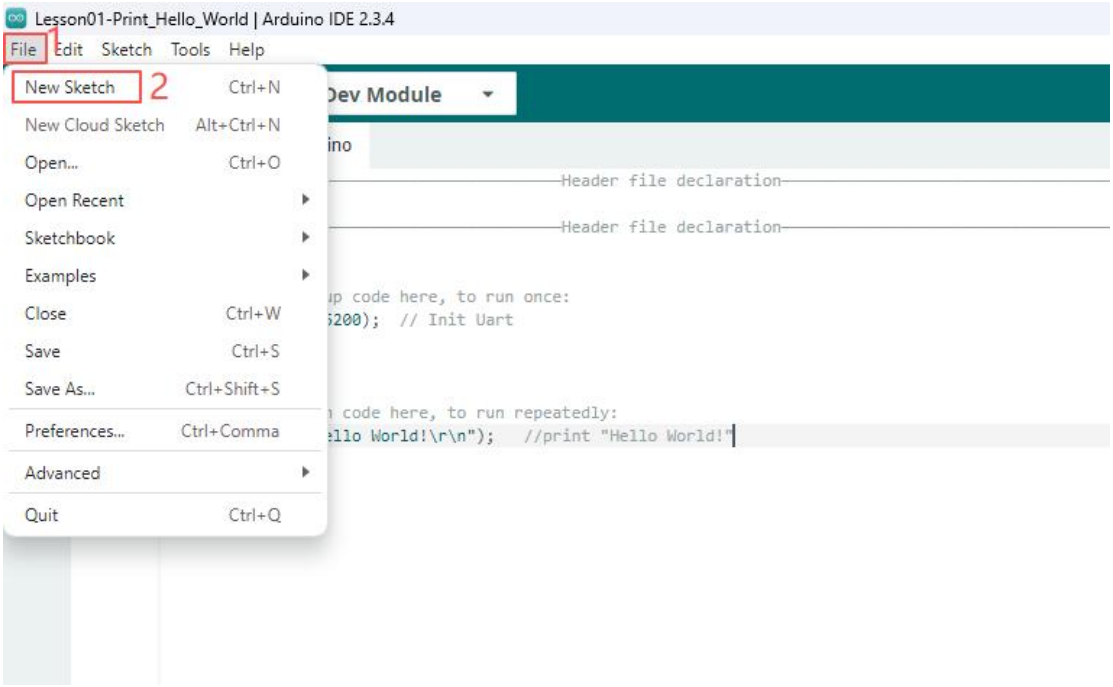
This lesson involves no hardware usage; it solely teaches you how to create a new project and how to upload code to the ESP32-P4 chip using the Arduino-IDE.

Complete Code

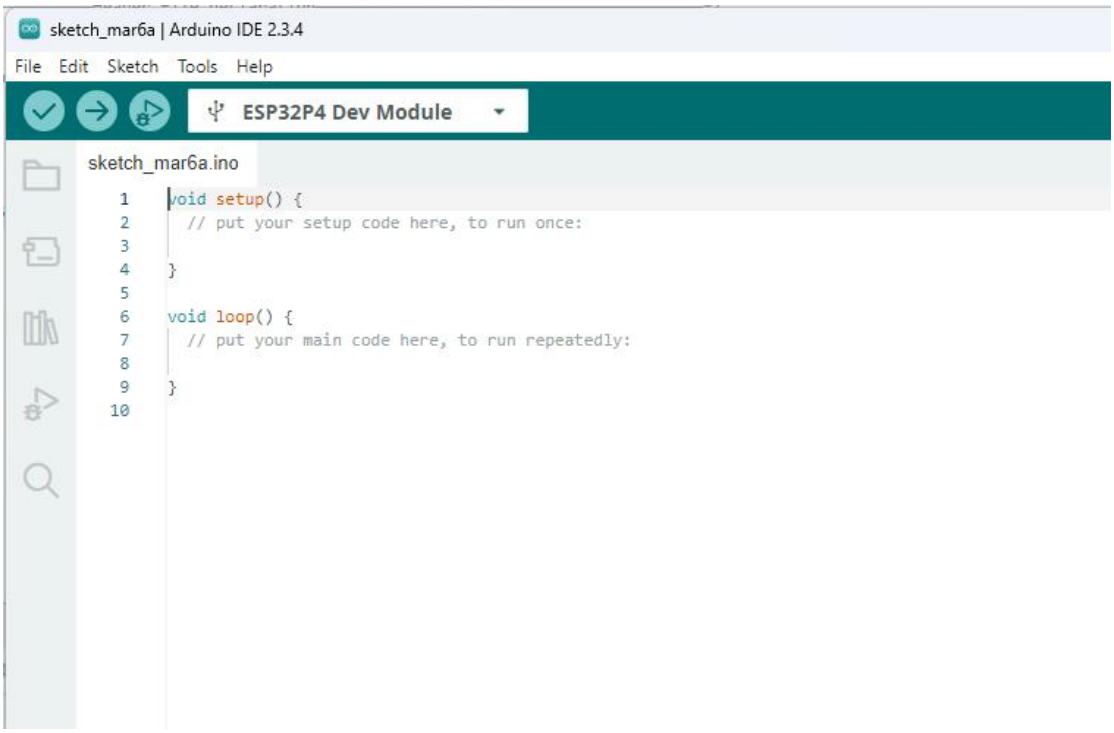
Kindly click the link below to view the full code implementation.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/Lesson01-Print_Hello_World

Key Explanations

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

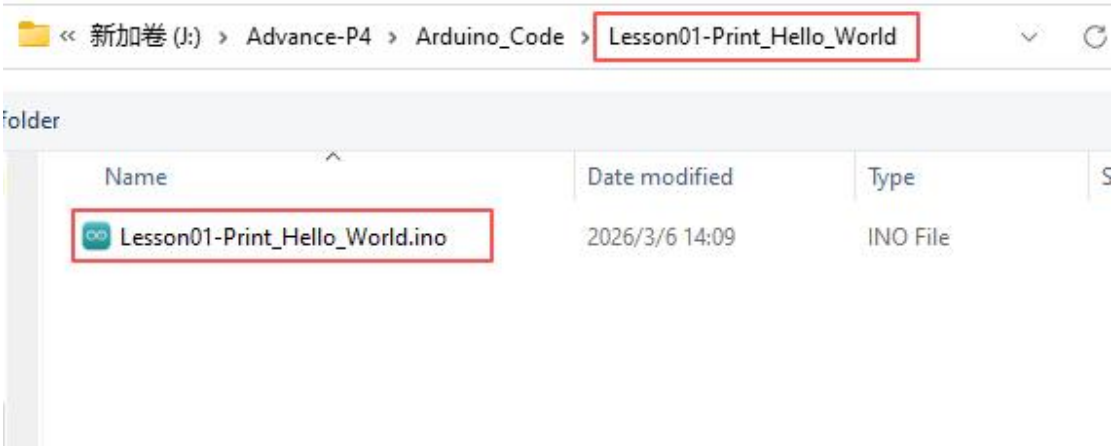


And just like that, a new project has been created.



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.



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

```
1  void setup() {
2    // put your setup code here, to run once:
3
4  }
5
6  void loop() {
7    // put your main code here, to run repeatedly:
8
9  }
10
```

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:

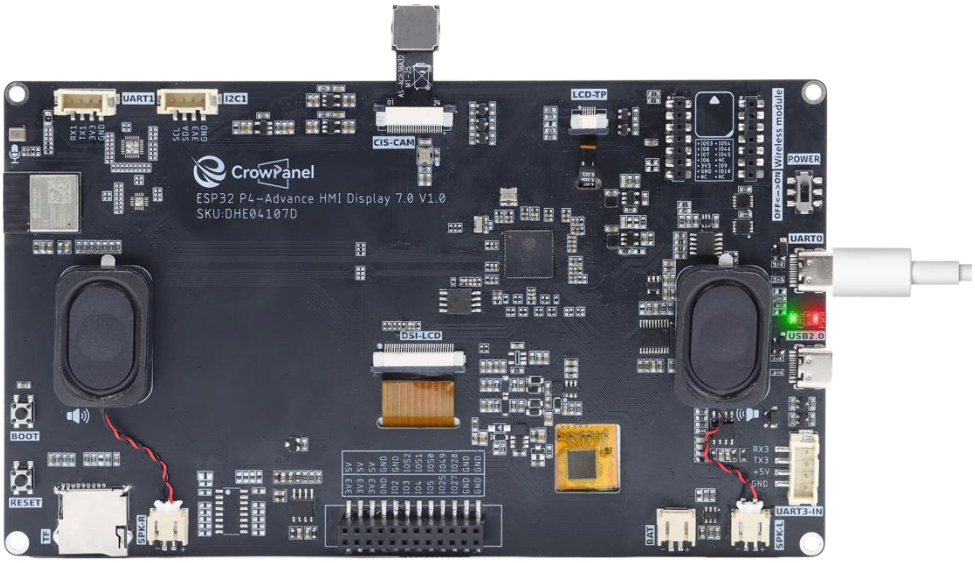
- setup() runs once—focus on initialization configurations (pins, serial, module setup, etc.). This is the "preparation phase" of your program.
- loop() runs infinitely—focus on business logic (reading data, controlling hardware, making judgments, etc.). This is the "runtime phase" of your program.

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

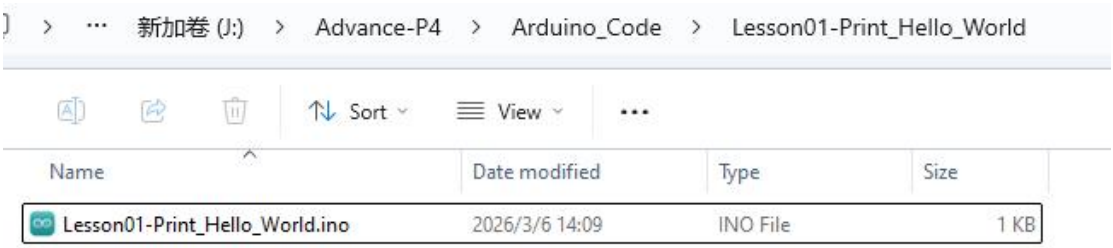
Programming Steps

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

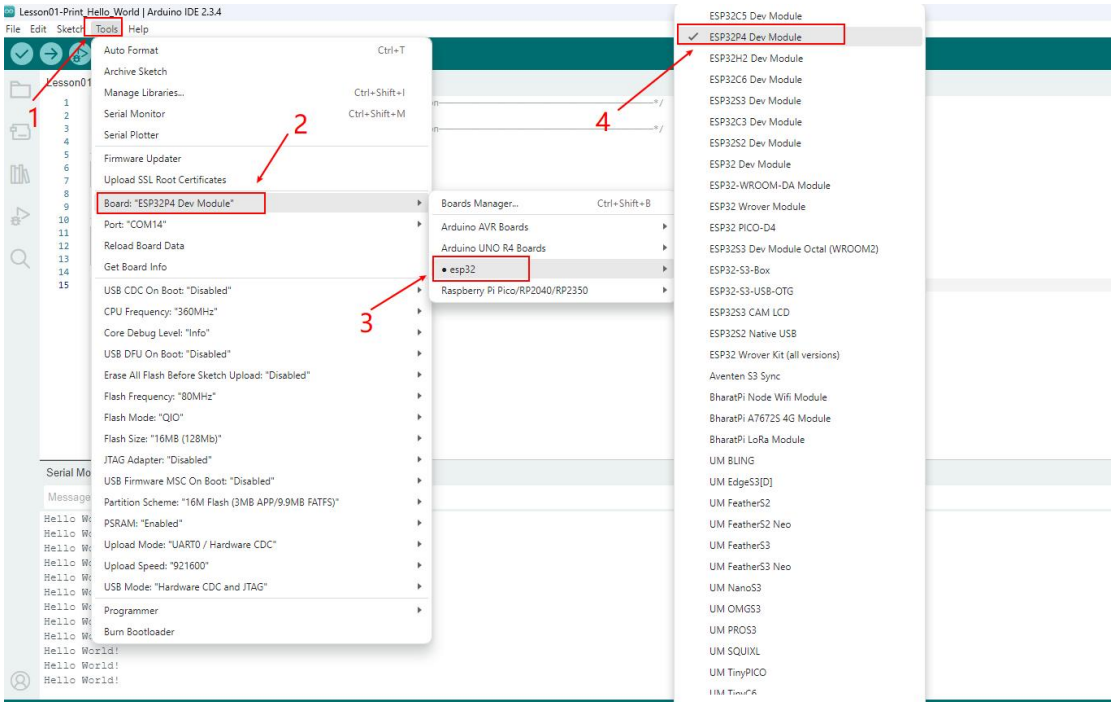
First, connect the Advance-P4 device to your computer host via a USB cable. (Connect to UART0)



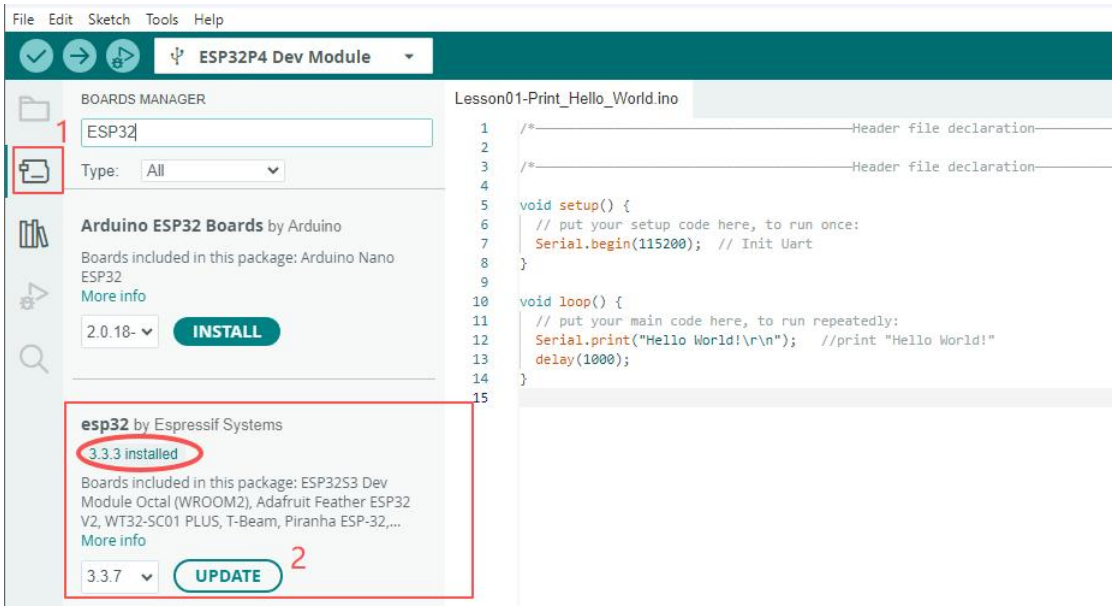
Open the code for this lesson from the GitHub repository link provided above.



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

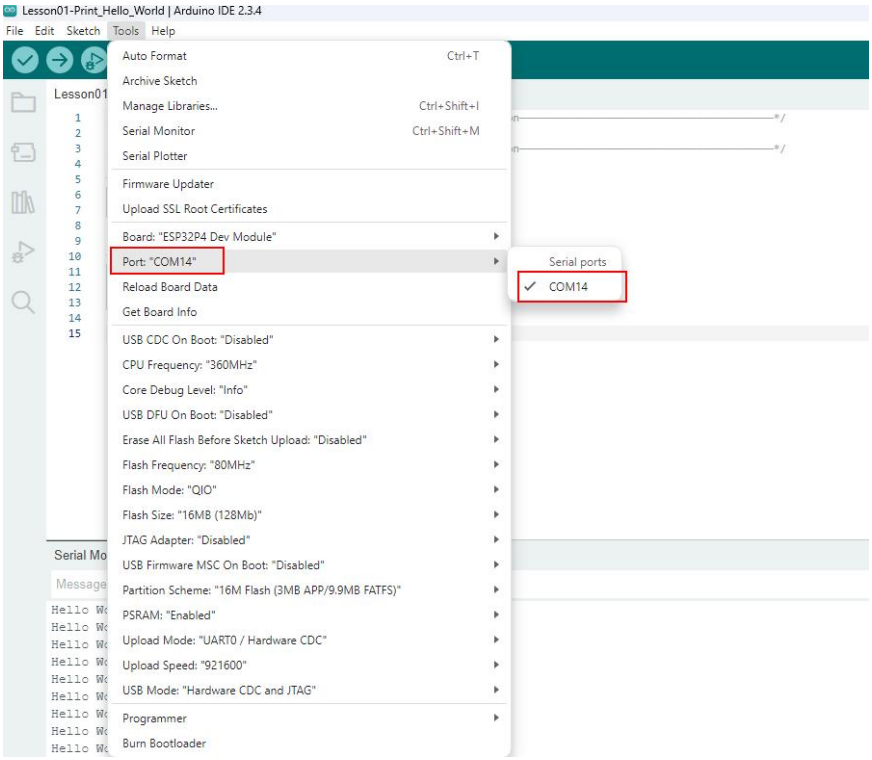


(If you don't see ESP32 in the list here, please follow the instructions from the preface to install ESP32 version **3.3.3.**)

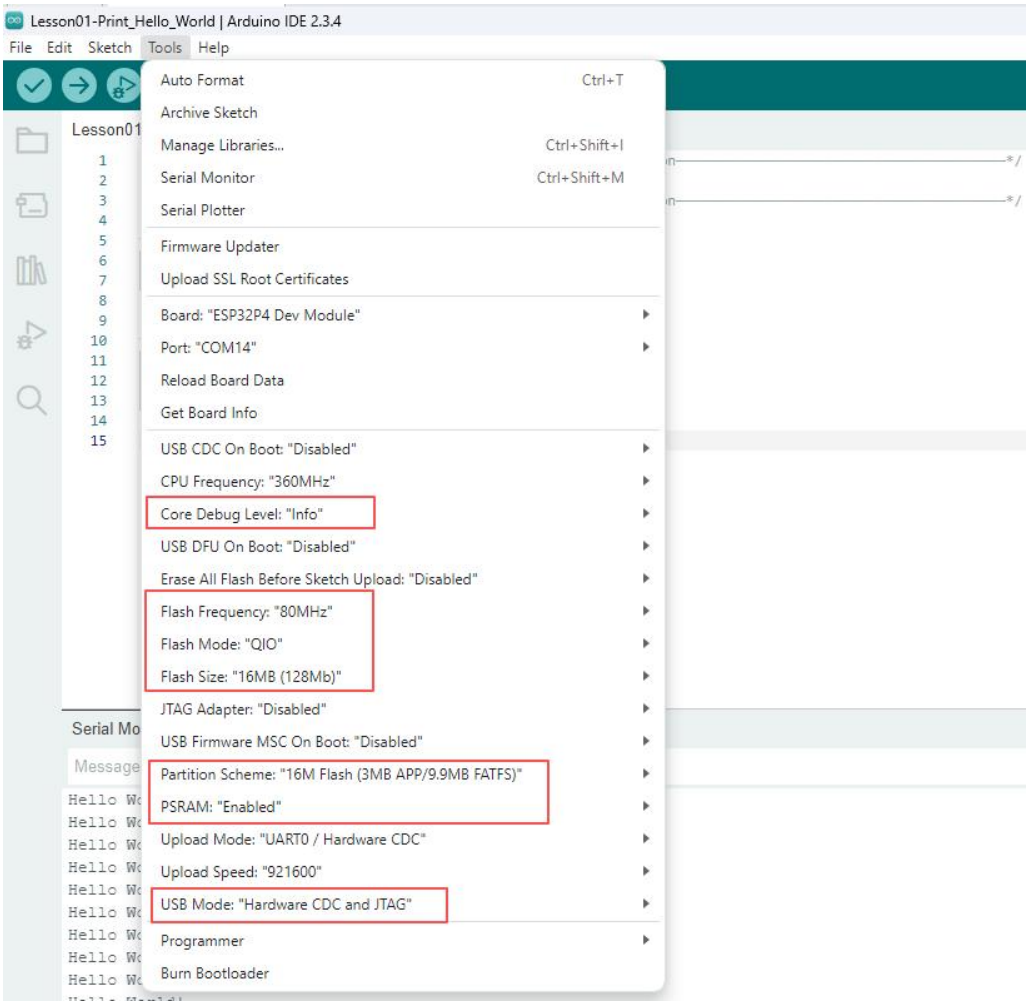


Then select Tools again and verify your COM port number.

(This assumes you have already connected your CrowPanel Advanced ESP32-P4 HMI AI Display [7-inch/9-inch/10.1-inch] to your computer.)



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



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

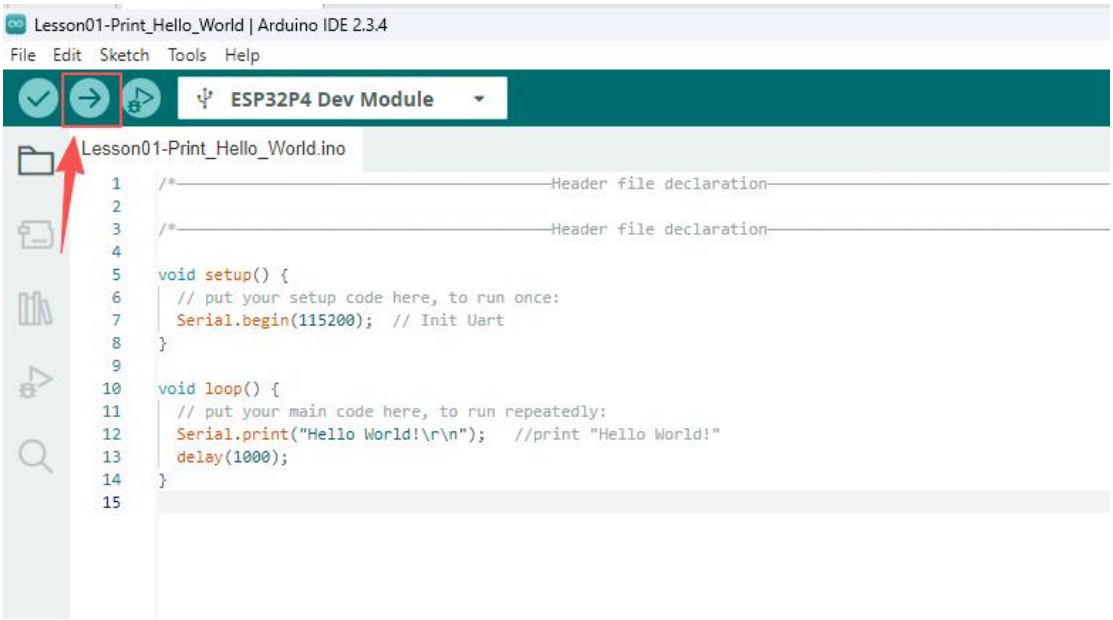
models, multimedia, large buffers) may run out of memory. Enabling PRAM allows the system to place heap memory, large variables, and caches in PSRAM, significantly enhancing multitasking and complex program execution capabilities. If your development board lacks PSRAM, you must disable this—otherwise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

5. USB Mode: "Hardware CDC and JTAG"

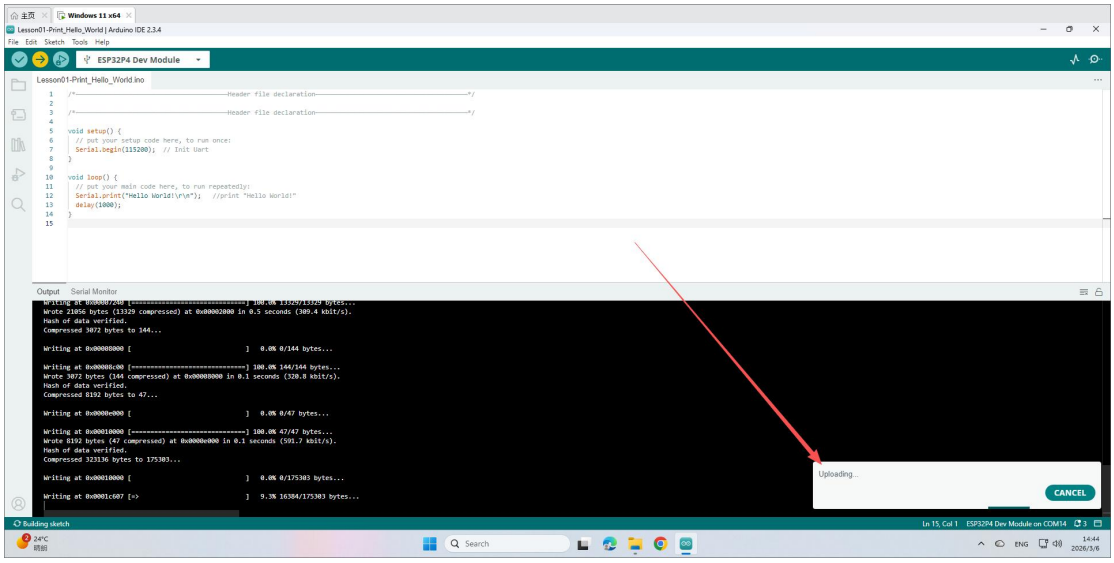
Purpose: Configures USB port functionality: Hardware CDC—hardware-level USB virtual serial port for serial communication (log printing, debugging), more stable and faster than software CDC; JTAG—hardware debugging interface for online debugging, breakpoints, and register inspection.
Why set this: The 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.

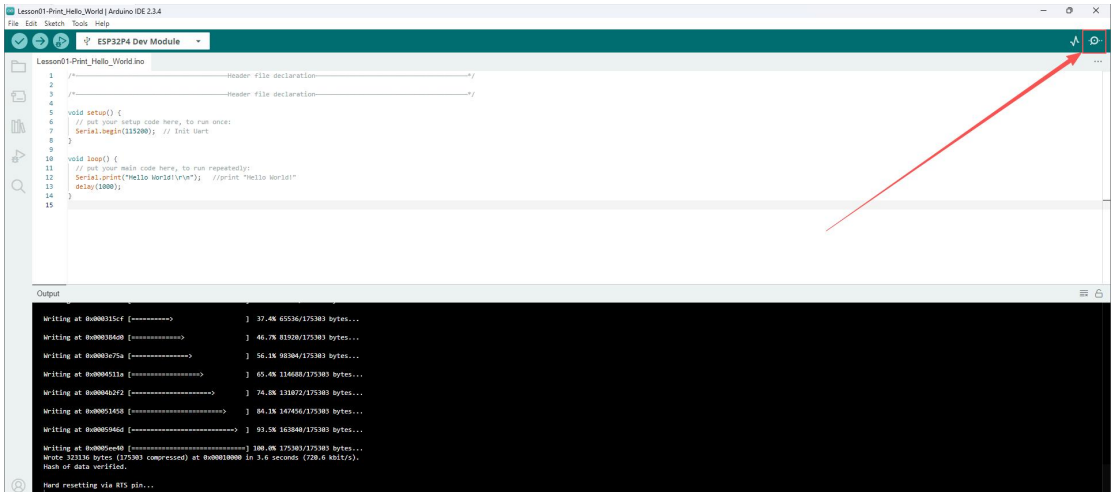
Once configured, upload the code.



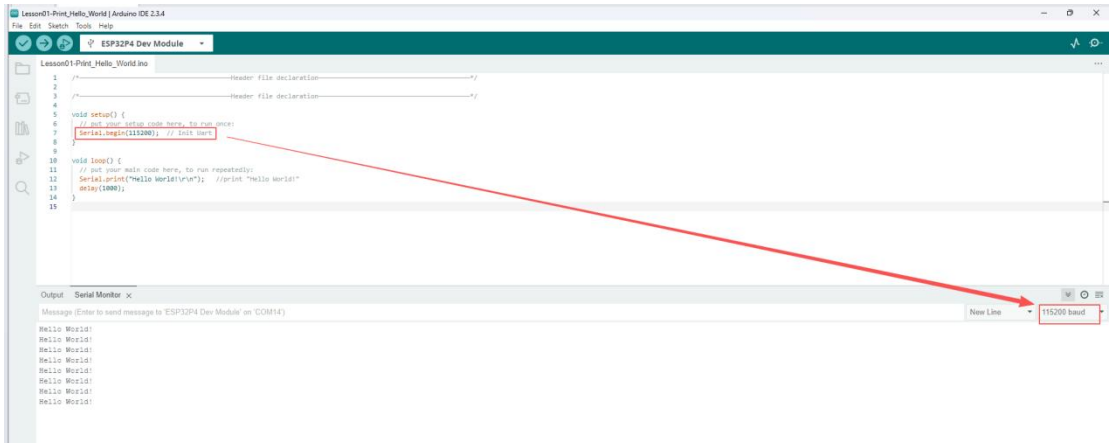
After waiting a moment, you will see the message indicating successful code upload in the output console.



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.



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.



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

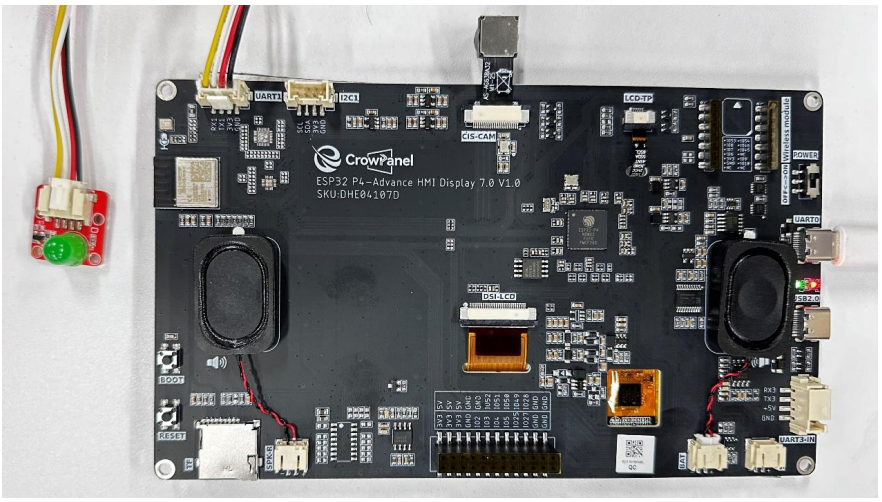
In this lesson, we will begin exploring the simplest control logic in the Arduino-IDE to perform level control on the UART1 interface of the Advance-P4. This will make the LED connected to the UART1 interface turn on for one second and off for one second in a continuous cycle.

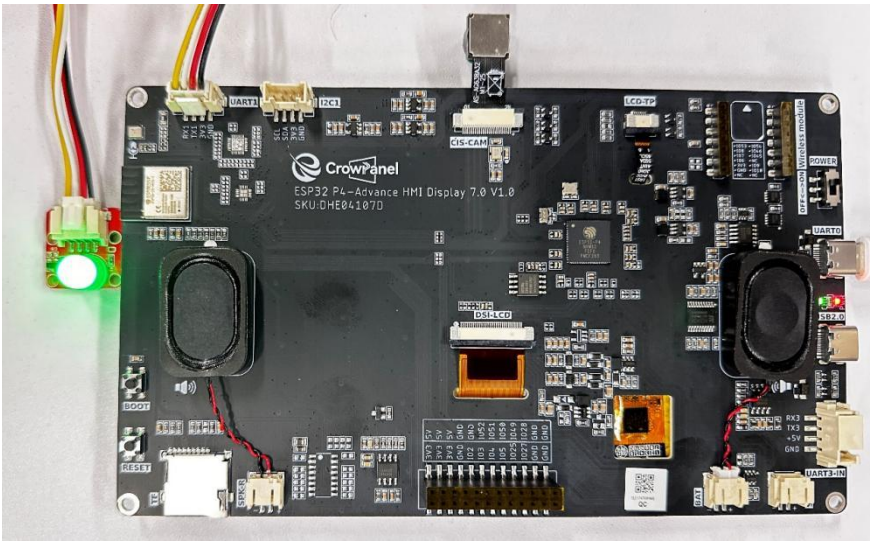
Learning Goals

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

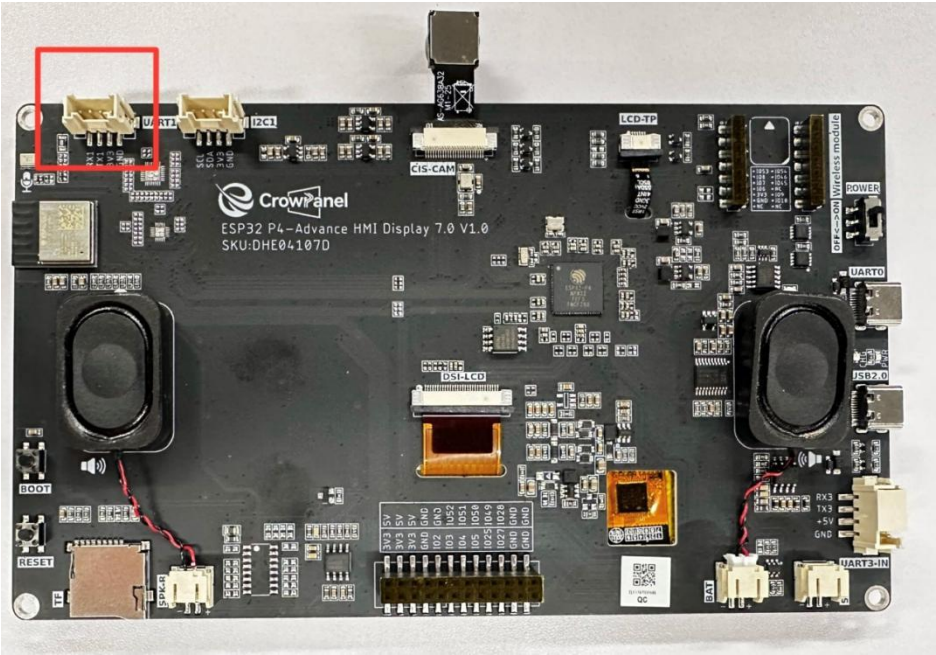
After running the code, you will be able to observe that the LED connected to the UART1 interface will light up for one second and then go off for one second.





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.

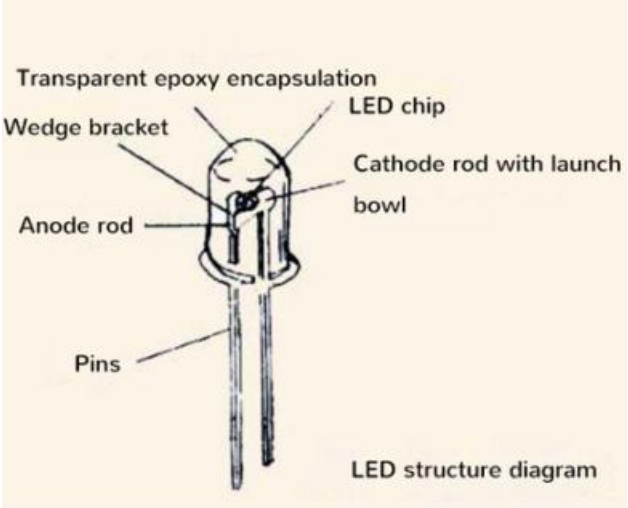
④ Interrupt support: GPIO pins support interrupt functionality, which can trigger interrupts when the signal changes. This is suitable for real-time response applications such as button detection and sensor triggering.
⑤ Status indication: GPIO pins can be used as LED indicators, achieving status visualization 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.

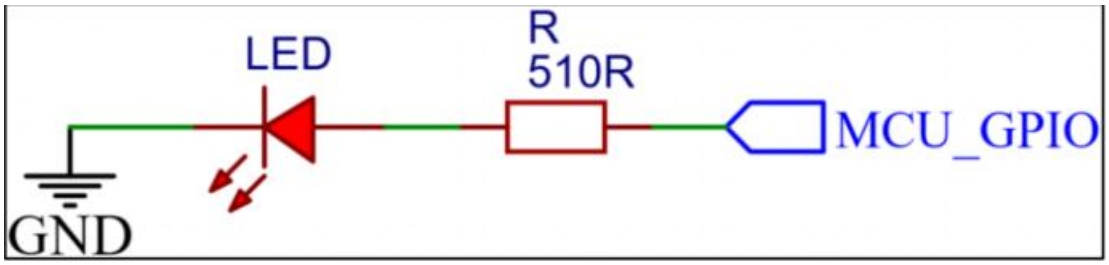
② **Principle of LED Lighting Driver**

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.

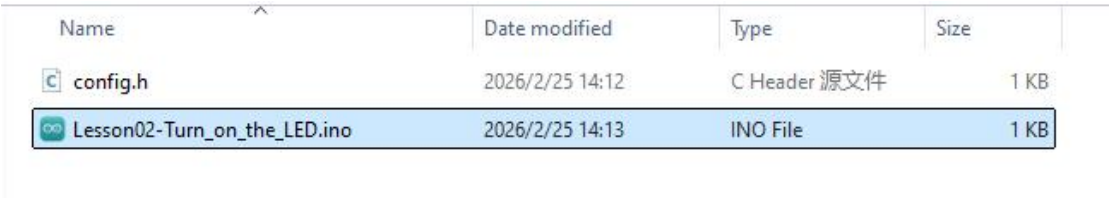
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/Lesson02-Turn_on_the_LED

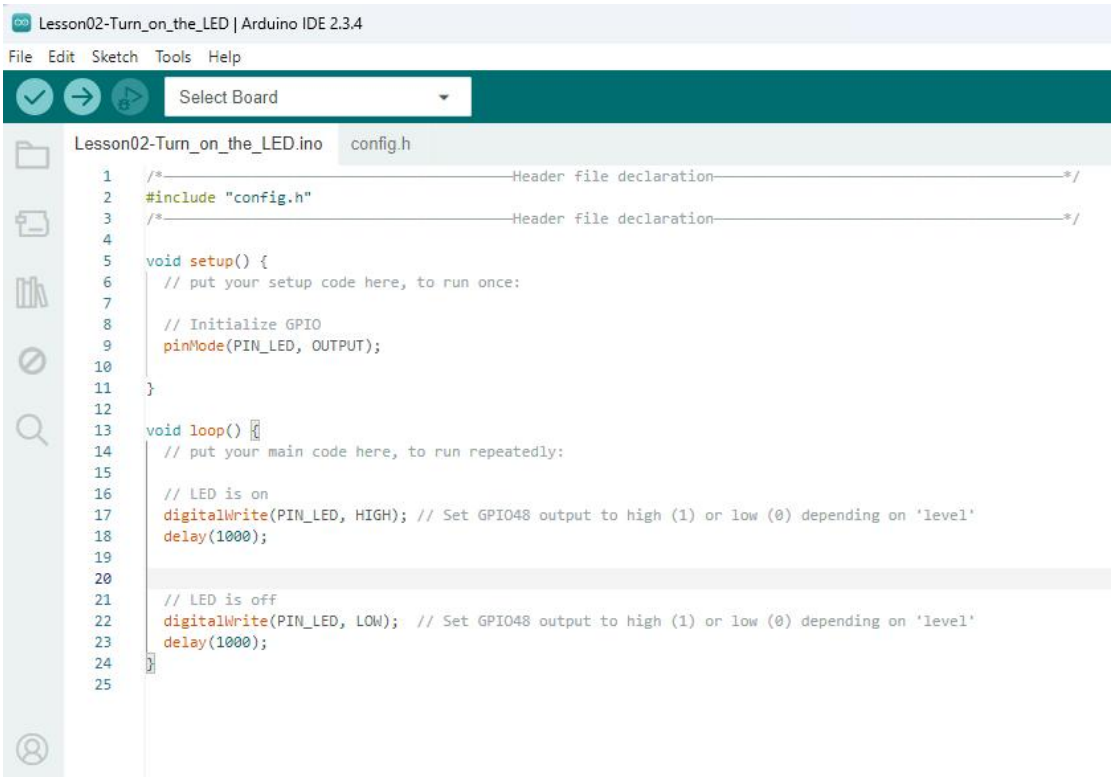
Key Explanations

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).



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

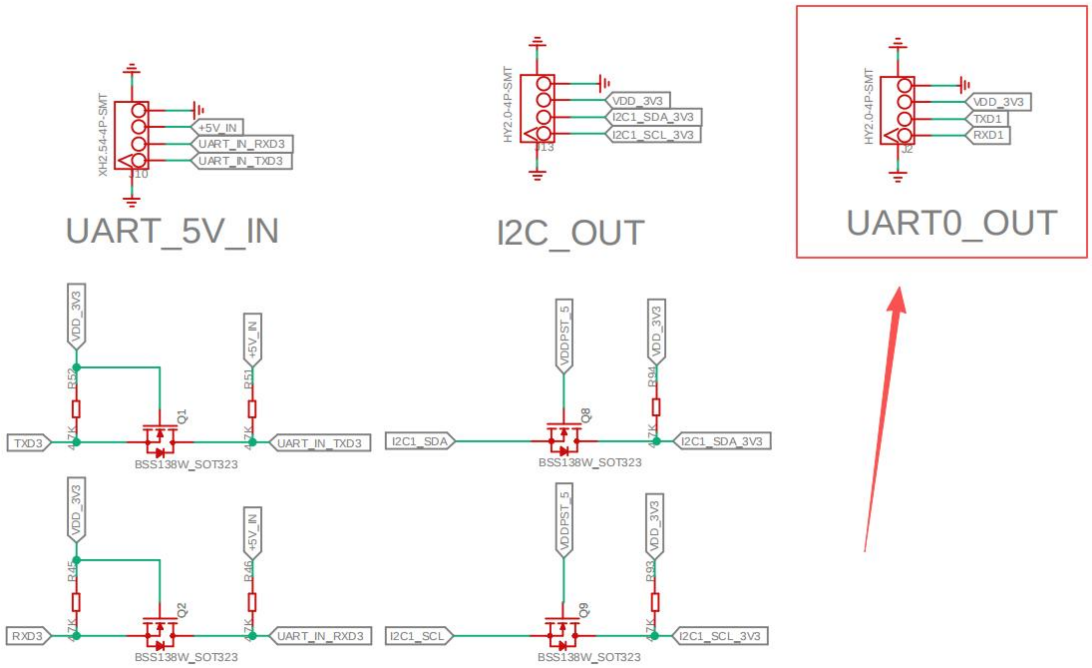


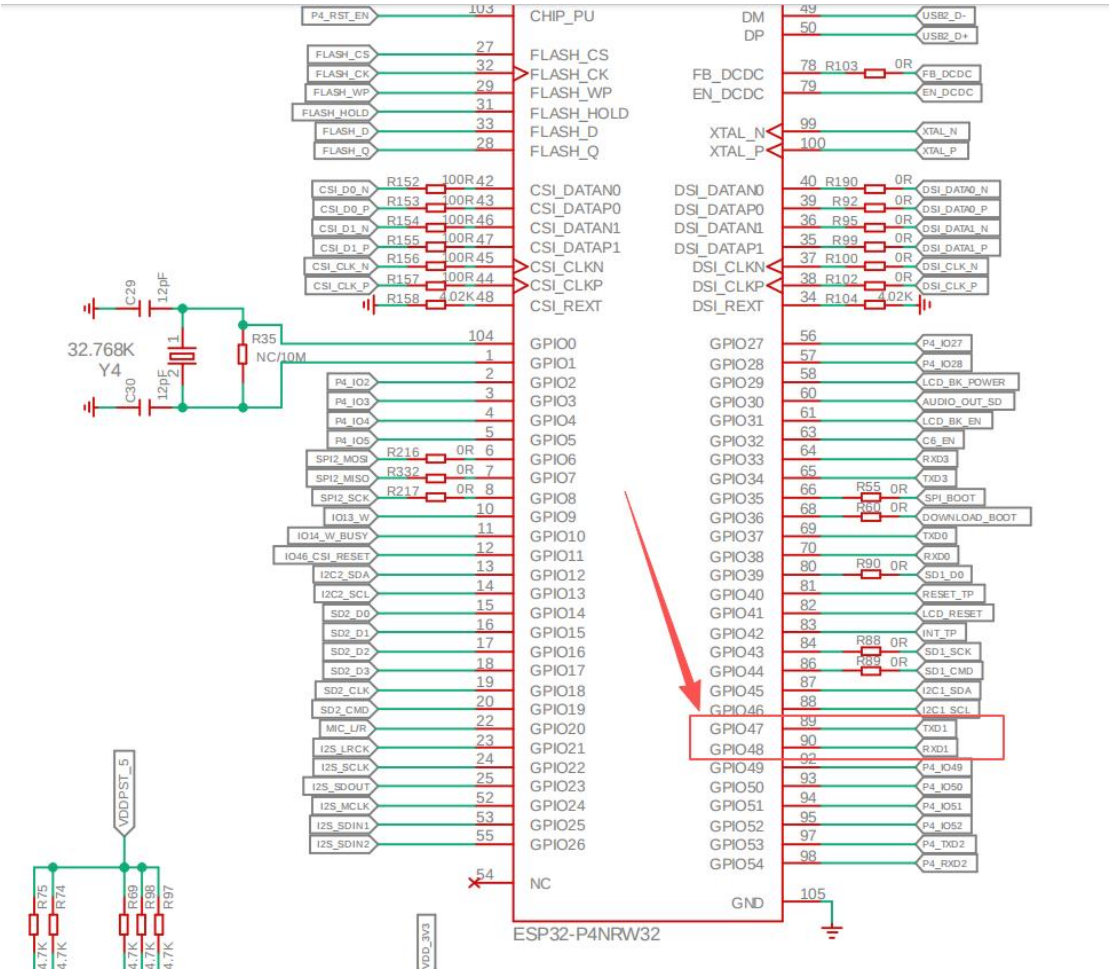
In subsequent code, the ".ino" file will primarily implement our functionality.

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."

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


UART_5V_IN
+5V_IN
UART_IN_RXD3
UART_IN_TXD3
I2C_OUT
VDD_3V3
I2C1_SDA_3V3
I2C1_SCL_3V3
UART0_OUT
VDD_3V3
TXD1
RXD1
TXD3
RXD3
I2C1_SDA
I2C1_SCL
BSS138W_SOT323


ESP32-P4NRW32
CHIP_PU
FLASH_CS
FLASH_CK
FLASH_WP
FLASH_HOLD
FLASH_D
FLASH_Q
CSI_DATAN0
CSI_DATAP0
CSI_DATAN1
CSI_DATAP1
CSI_CLKN
CSI_CLKP
CSI_REXT
DSI_DATAN0
DSI_DATAP0
DSI_DATAN1
DSI_DATAP1
DSI_CLKN
DSI_CLKP
DSI_REXT
XTAL_N
XTAL_P
FB_DCDC
EN_DCDC
DM
DP
GPIO0
GPIO1
GPIO2
GPIO3
GPIO4
GPIO5
GPIO6
GPIO7
GPIO8
GPIO9
GPIO10
GPIO11
GPIO12
GPIO13
GPIO14
GPIO15
GPIO16
GPIO17
GPIO18
GPIO19
GPIO20
GPIO21
GPIO22
GPIO23
GPIO24
GPIO25
GPIO26
GPIO27
GPIO28
GPIO29
GPIO30
GPIO31
GPIO32
GPIO33
GPIO34
GPIO35
GPIO36
GPIO37
GPIO38
GPIO39
GPIO40
GPIO41
GPIO42
GPIO43
GPIO44
GPIO45
GPIO46
GPIO47
GPIO48
GPIO49
GPIO50
GPIO51
GPIO52
GPIO53
GPIO54
NC
GND
32.768K
Y4
NC/10M

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).

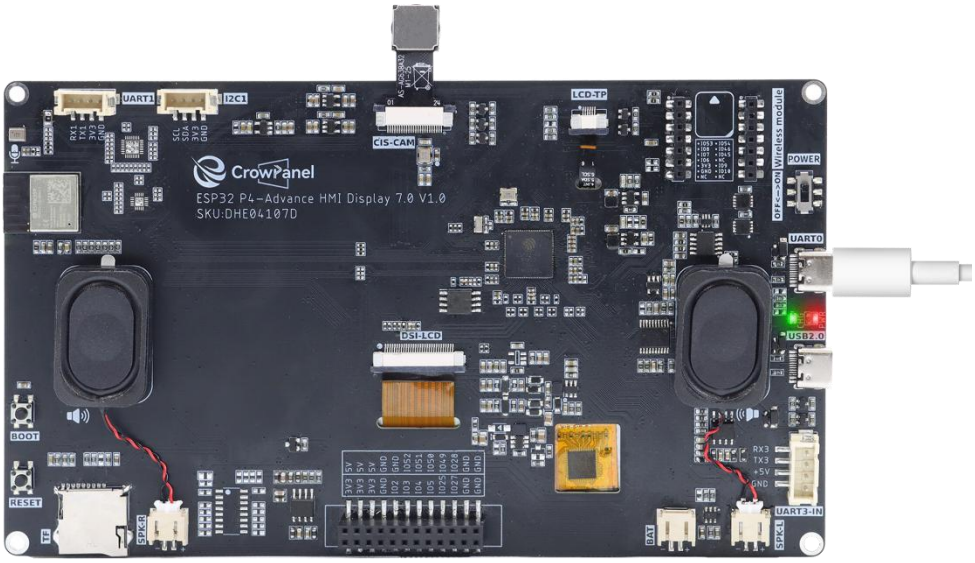
Note: delay() is "blocking delay"—during the delay period, the ESP32-P4 suspends other operations until the delay completes.

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).

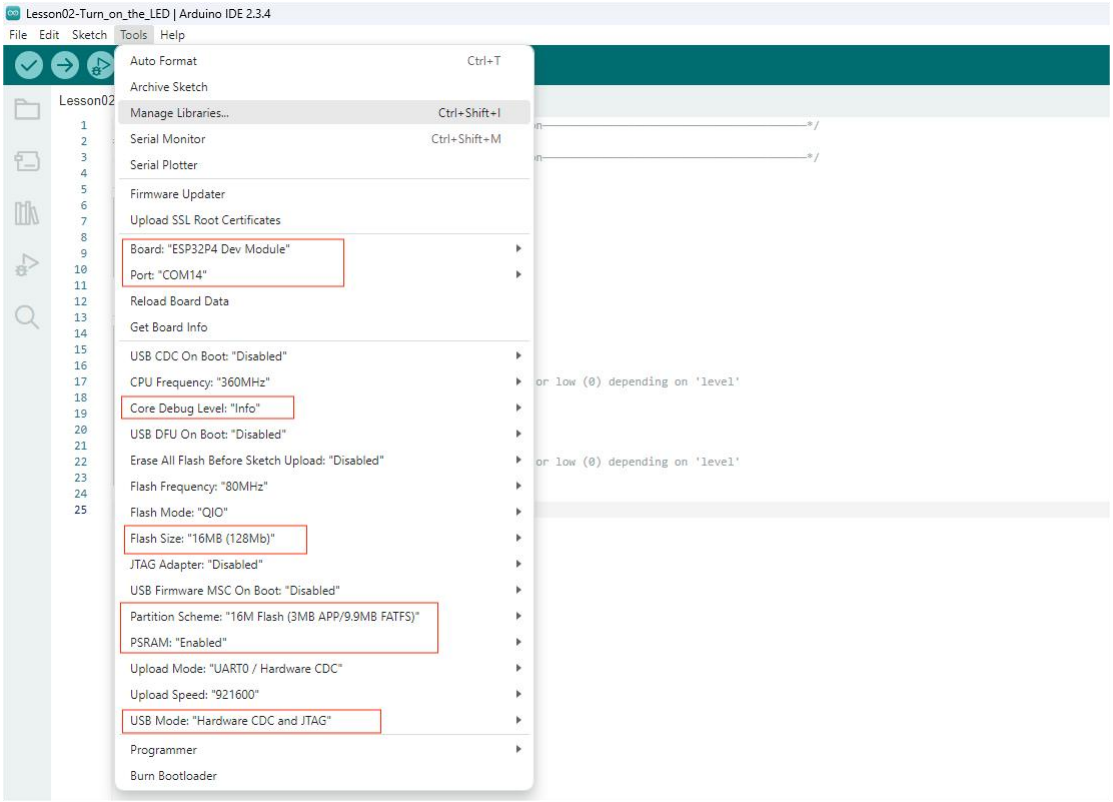
delay(1000);： Pauses for another 1 second, maintaining the LED off state for 1 second.

Programming Steps

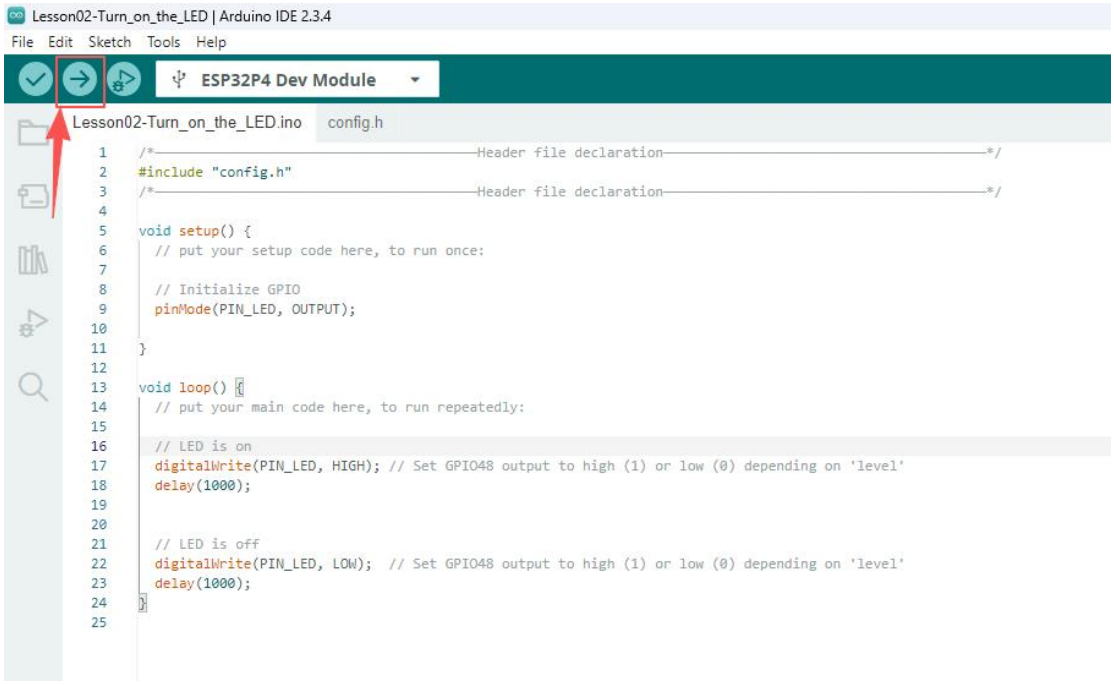
Now that the code is ready, we need to flash the ESP32-P4 to see the results in action. First, connect the Advance-P4 device to your computer host via a USB cable.



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



Then we compile and upload the code.

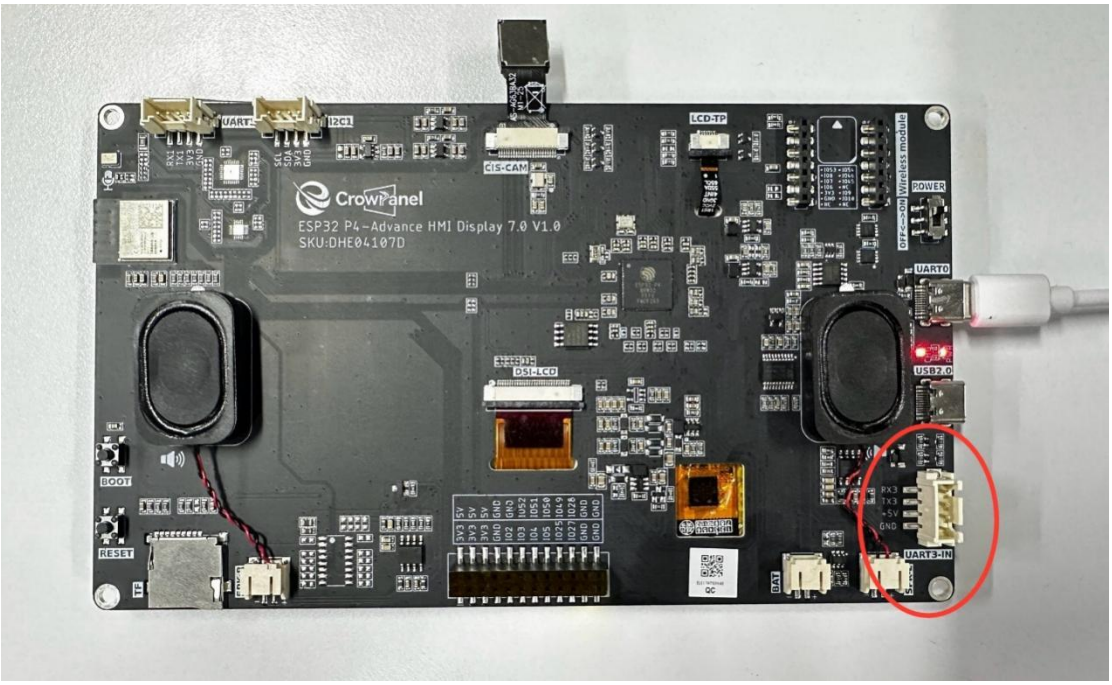


After waiting a moment, you will see the LED connected to UART1 on your Advance-P4 blinking on for one second and off for one second, repeating this cycle continuously.

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.

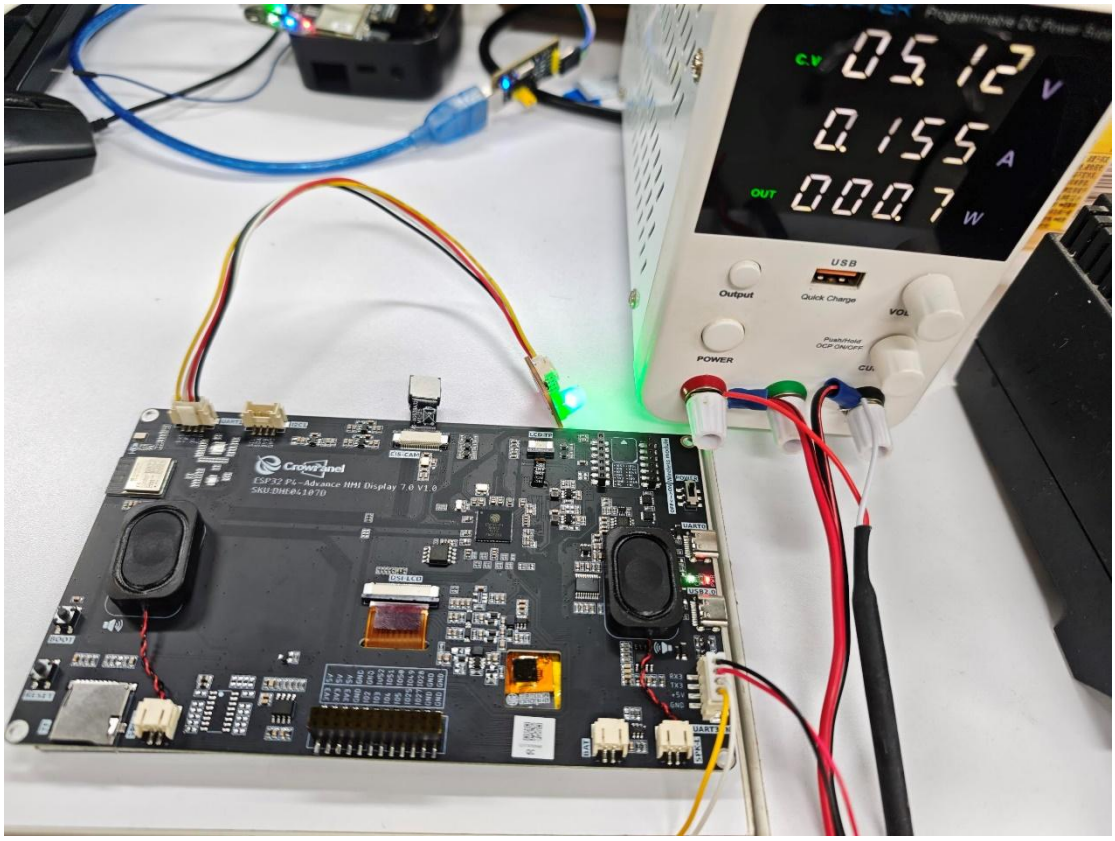


Learning Goals

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

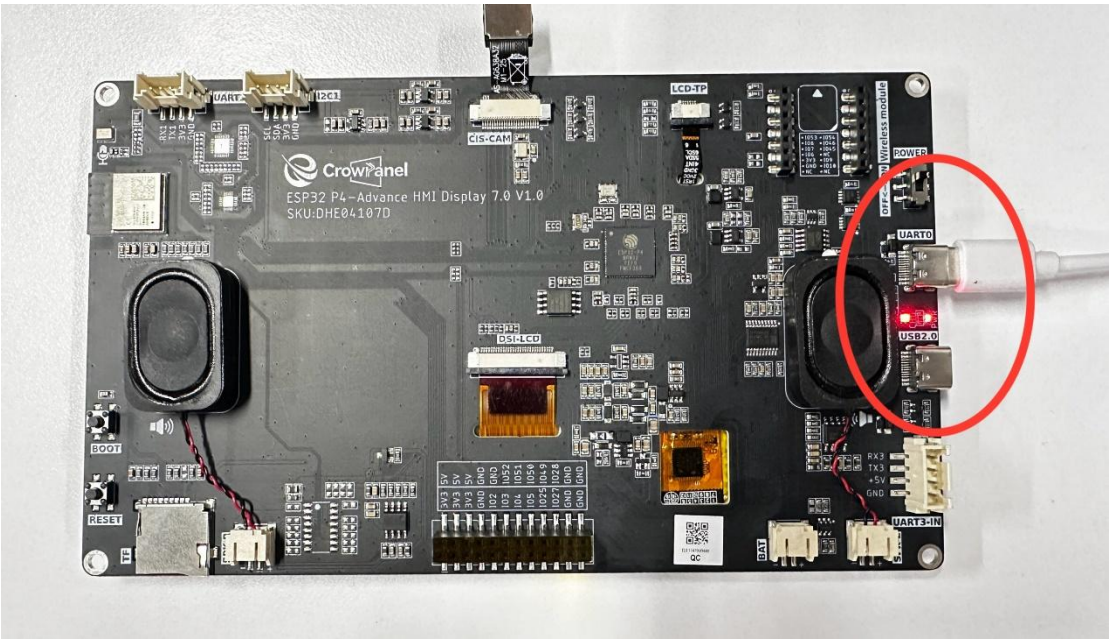
This lesson involves no hardware usage; it solely explains the function of the UART3-IN interface on the Advance-P4 development board.

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.



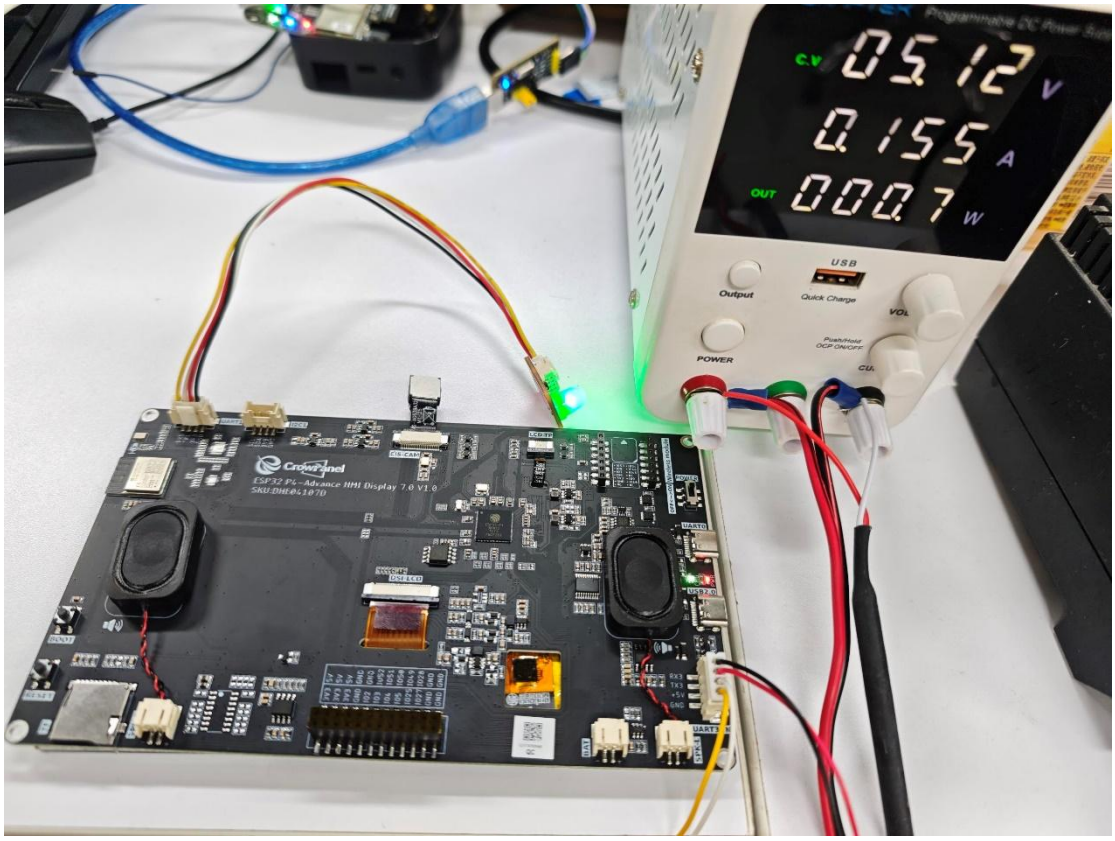
Then we come back to the UART3-IN interface. This interface is similar in function to the UART0 interface we just discussed. It can supply power, but it cannot upload code.

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

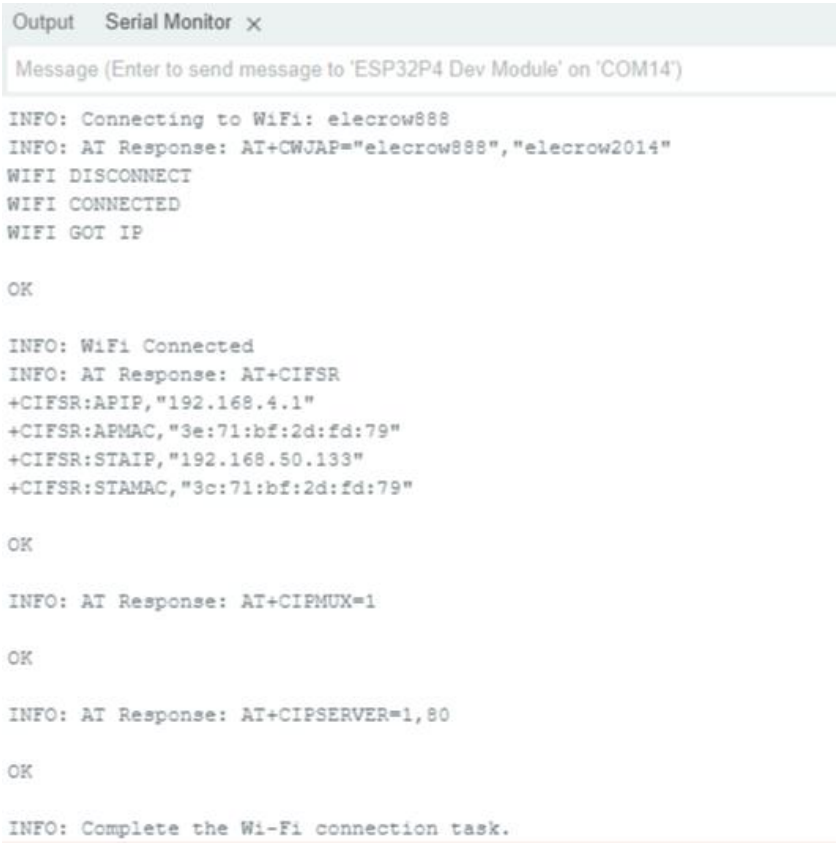
In this class, we will start teaching you how to use the serial port component. We will communicate with the Wi-Fi serial module through the UART1 interface on the Advance-P4.
The Advance-P4 connects to the Wi-Fi module via the serial port. After sending the AT command to the Wi-Fi module, it enables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

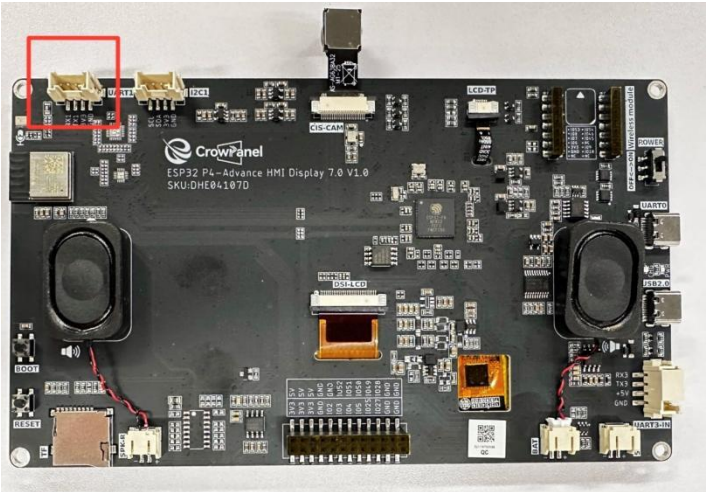
Preview of the Result

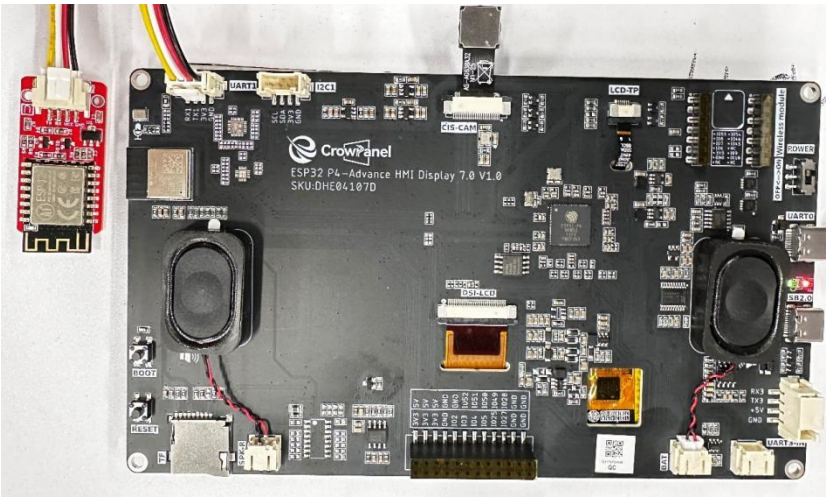
After running the code, you will be able to see the AT commands you sent on the monitor of Arduino-IDE, as well as the responses returned to you by the Wi-Fi module via the serial port.



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

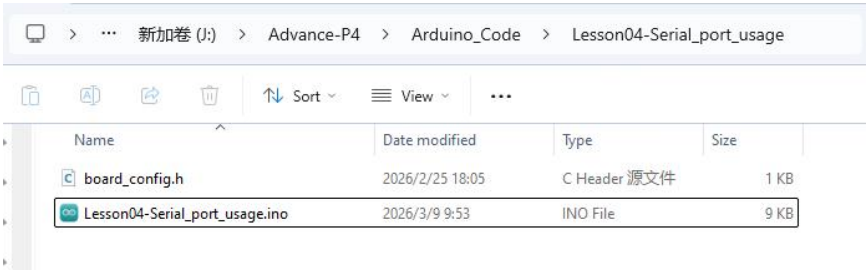
First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/Lesson04-Serial_port_usage

Key Explanations

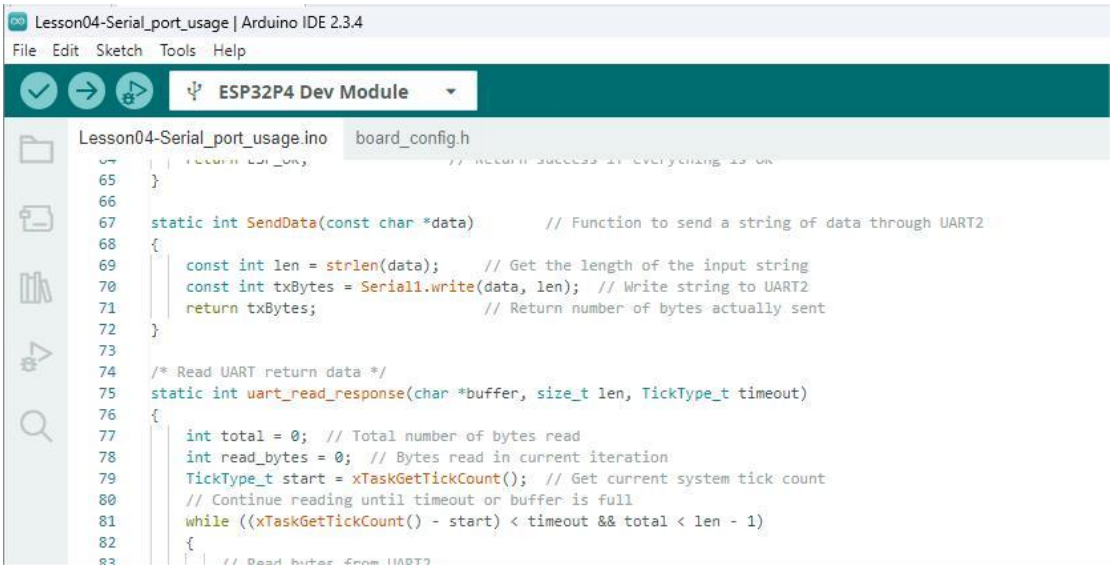
The focus of this lesson is on how to use the serial port. Send commands to the ESP32-P4 chip via the serial port, and after the ESP32-P4 receives the commands, it controls the WiFi module connected to UART1 to establish a WiFi connection. Once the connection is successful, it sends relevant status messages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



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

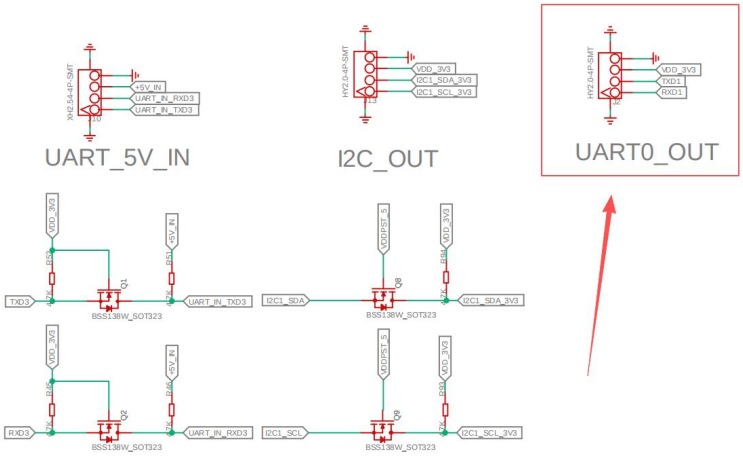


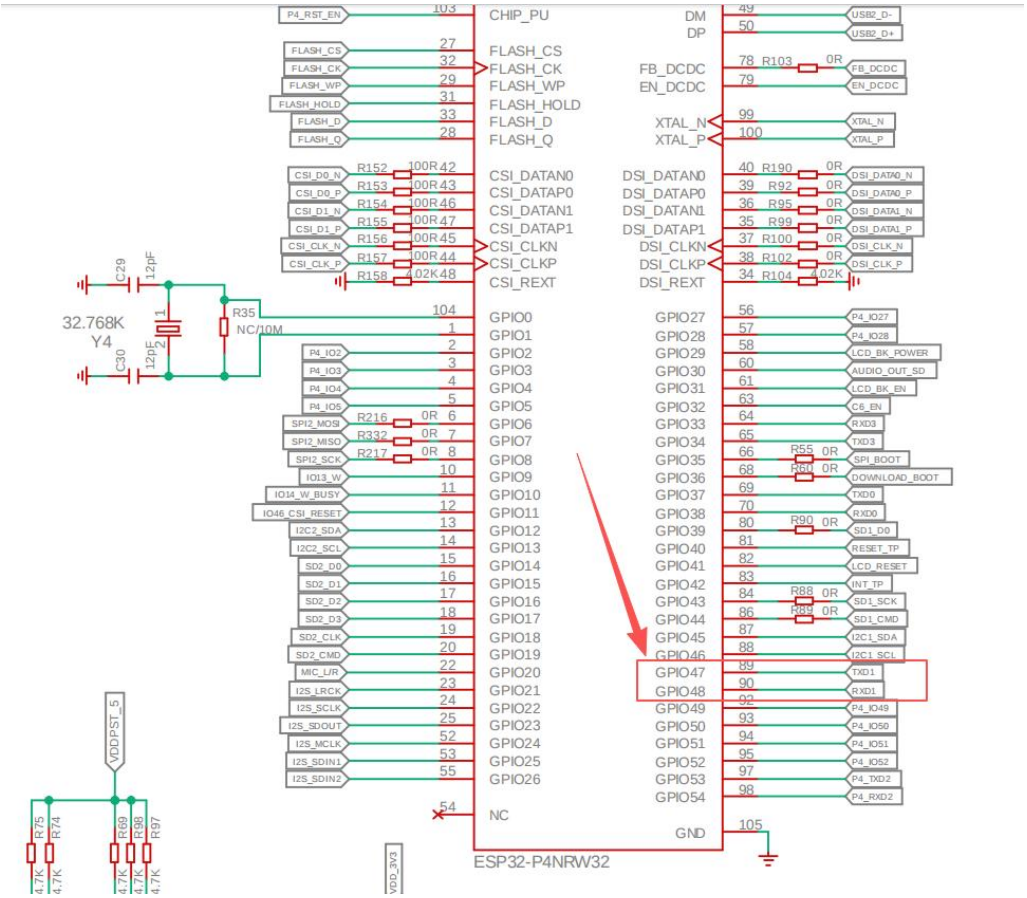
Let's first take a look at the board_config.h file.

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()

```
74  /* Read UART return data */
75  static int uart_read_response(char *buffer, size_t len, TickType_t timeout)
76  {
77      int total = 0;  // Total number of bytes read
78      int read_bytes = 0;  // Bytes read in current iteration
79      TickType_t start = xTaskGetTickCount();  // Get current system tick count
80      // Continue reading until timeout or buffer is full
81      while ((xTaskGetTickCount() - start) < timeout && total < len - 1)
82      {
83          // Read bytes from UART2
84          if (Serial1.available()) {
85              read_bytes = Serial1.read((uint8_t *)(buffer + total), len - total - 1);
86              if (read_bytes > 0)
87              {
88                  total += read_bytes;  // Accumulate total bytes read
89              }
90          } else {
91              vTaskDelay(pdMS_TO_TICKS(5));
92          }
93      }
94      buffer[total] = '\0';  // Null-terminate the response string
95      return total;  // Return total bytes read
96  }
```

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.

```
81      while ((xTaskGetTickCount() - start) < timeout && total < len - 1)
82      {
```

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.

```
84          if (Serial1.available()) {
85              read_bytes = Serial1.read((uint8_t *)(buffer + total), len - total - 1);
86              if (read_bytes > 0)
87              {
88                  total += read_bytes;  // Accumulate total bytes read
89              }
```

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.

```
90          } else {
91              vTaskDelay(pdMS_TO_TICKS(5));
92          }
93      }
94      buffer[total] = '\0';  // Null-terminate the response string
95      return total;  // Return total bytes read
96  }
```

2.4 AT Command Send Function send_at_command()

```
/* 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).

```
/* 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.

```
    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.

```
153    // Attempt to connect to WiFi, maximum 5 tries
154    bool connected = false;
155    for (int i = 0; i < 5; i++)
156    {
157        if (connect_wifi())
158        {
159            connected = true;  // Mark as connected if successful
160            break;
161        }
162        vTaskDelay(pdMS_TO_TICKS(2000));  // Delay between connection attempts
163    }
164
165    if (!connected)
166    {
167        PRINTF_ERROR(TAG, "Cannot connect to WiFi, stopping task");  // Log failure after all attempts
168        vTaskDelete(NULL);  // Delete task if connection failed
169    }
```

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).

```
171        // Get IP address of the module
172        send_at_command("AT+CIFSR", pdMS_TO_TICKS(1000));
173        // Enable multiple connections mode
174        send_at_command("AT+CIPMUX=1", pdMS_TO_TICKS(1000));
175        // Start TCP server on port 80
176        send_at_command("AT+CIPSERVER=1,80", pdMS_TO_TICKS(1000));
177
178        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.

```
179
180        while (1)
181        {
182            // TODO: Can read UART data here to process TCP requests
183            vTaskDelay(pdMS_TO_TICKS(1000));  // Delay to reduce CPU usage
184        }
185    }
```

2.7 Arduino Framework Entry Functions

```
187    void setup() {
188      // put your setup code here, to run once:
189
190      // Create WiFi task with 4096 bytes stack, priority 5
191        xTaskCreate(wifi_task, "wifi_task", 4096, NULL, 5, NULL);
192
193    }
194
195    void loop() {
196      // put your main code here, to run repeatedly:
197
198    }
```

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.

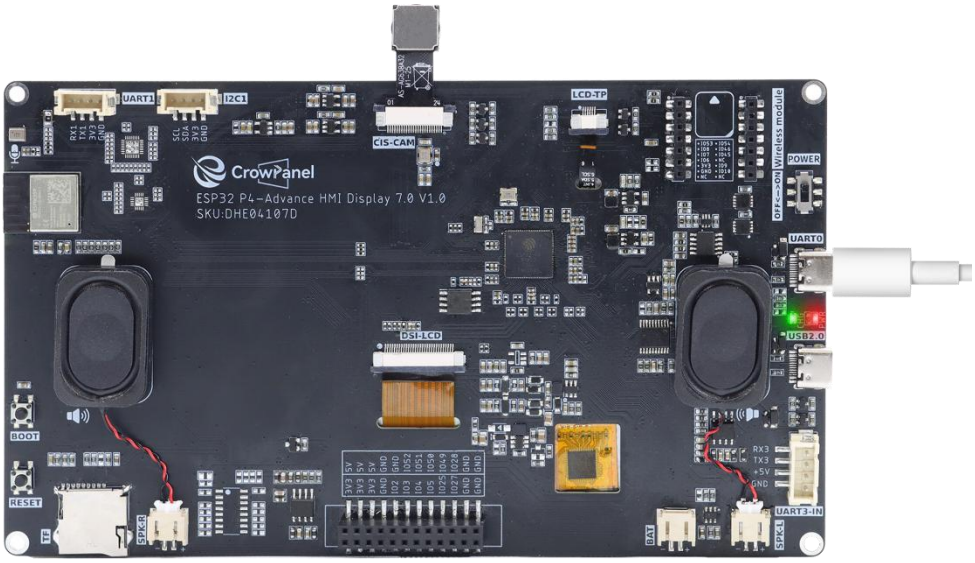
In the setup() function, the program calls xTaskCreate() to create a task named "wifi_task". The actual function this task runs is the wifi_task() explained earlier. It allocates 4096 bytes of stack space for this task, passes NULL as the parameter, sets task priority to 5, and the last parameter is 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.

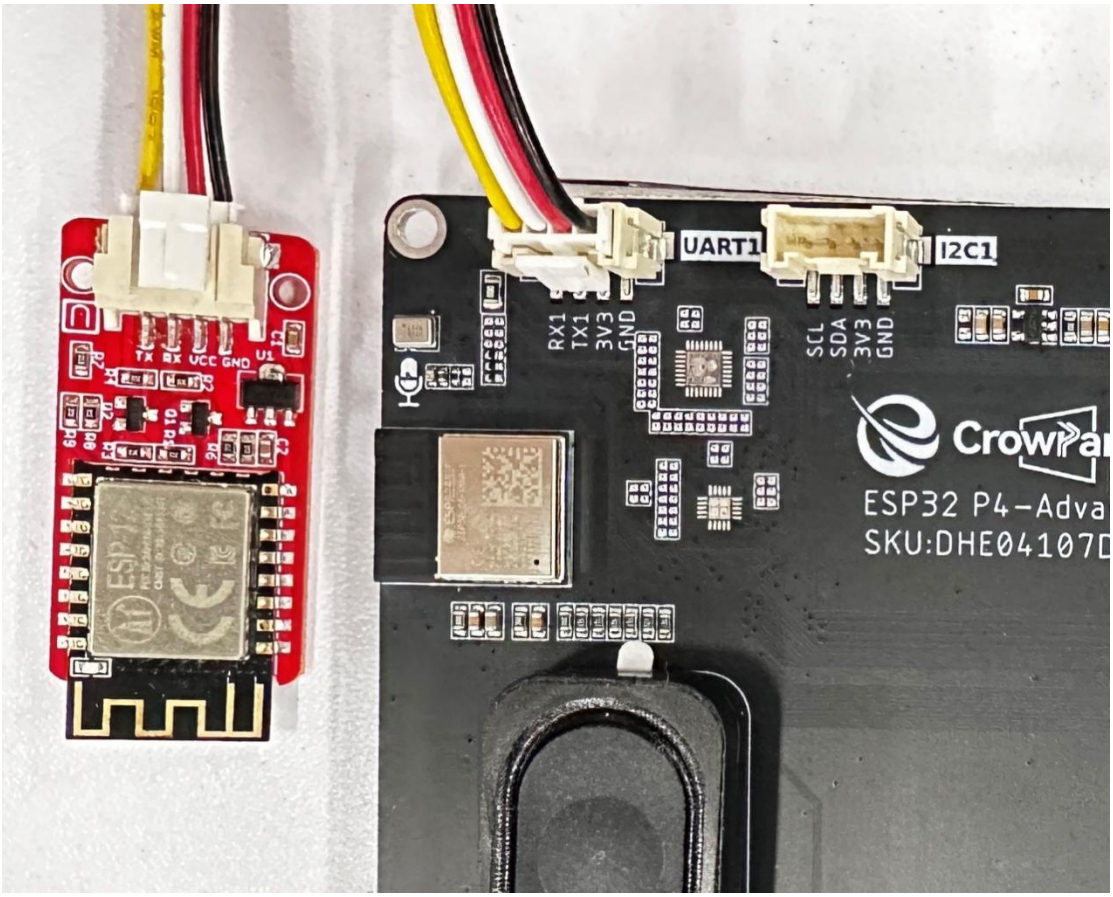
The loop() function remains empty because when using FreeRTOS, main logic is typically placed in individual tasks rather than inside Arduino's loop() cycle.

Programming Steps

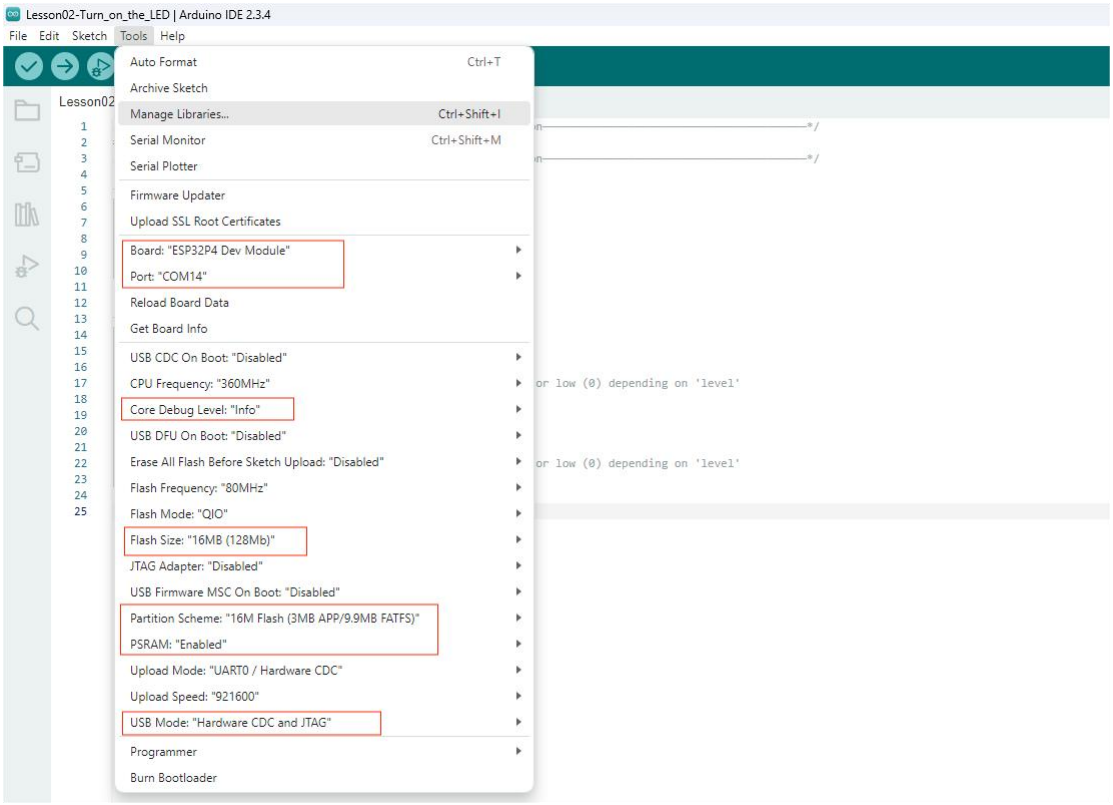
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



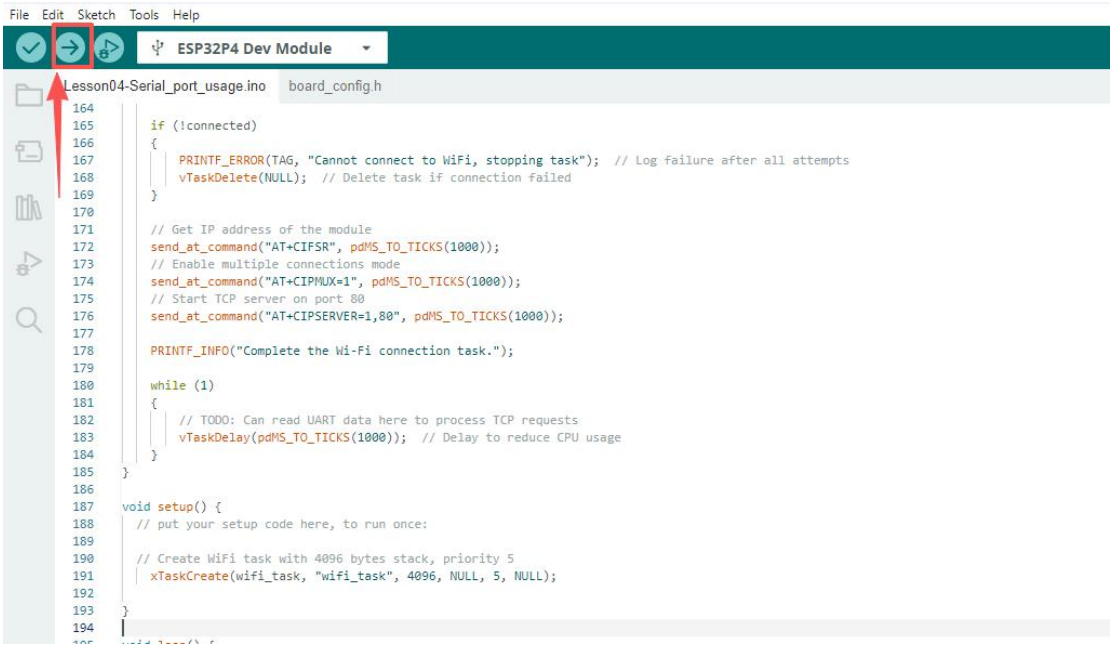
Then, connect an ESP8266 wifi module to the UART1 interface.
(Connect the VCC of UART1 interface to the VCC pin of the wifi module)
(Connect the GND of UART1 interface to the GND pin of the wifi module)
(Turn the TX of UART1 interface to the RX pin of the wifi module) (Cross connection)
(Turn the RX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



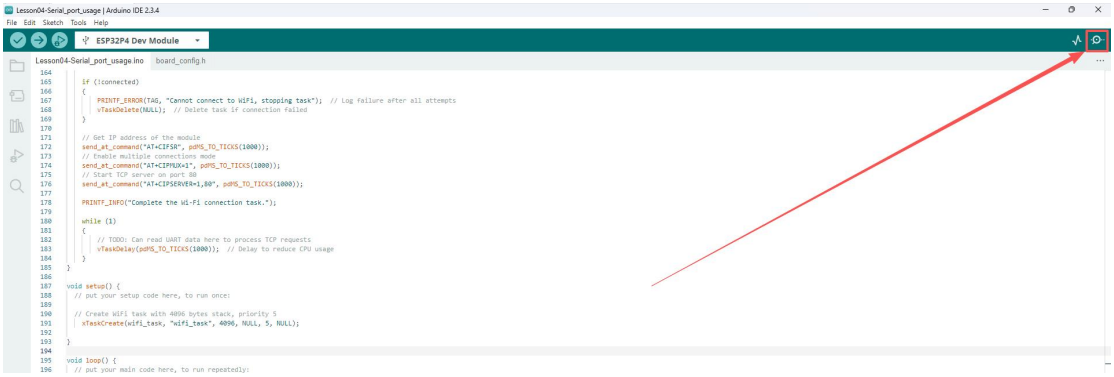
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



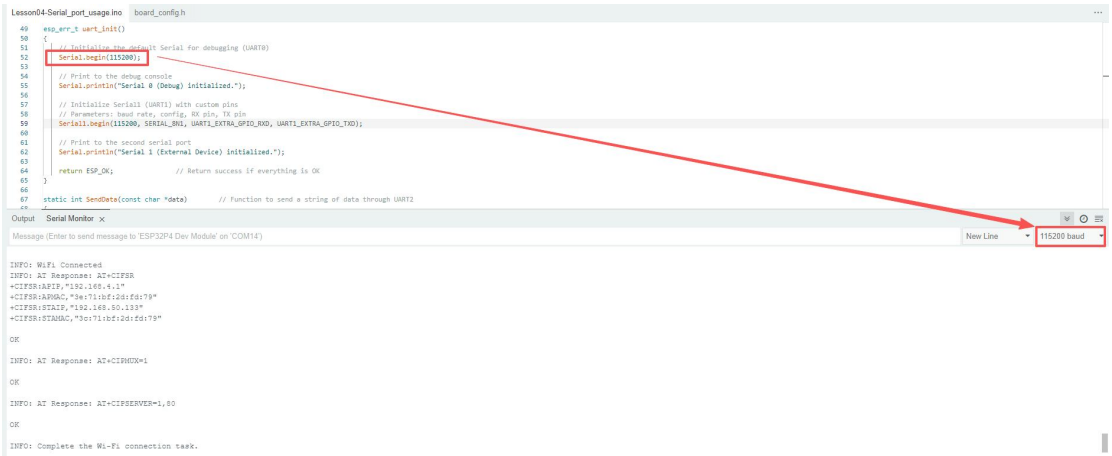
Then we compile and upload the code.



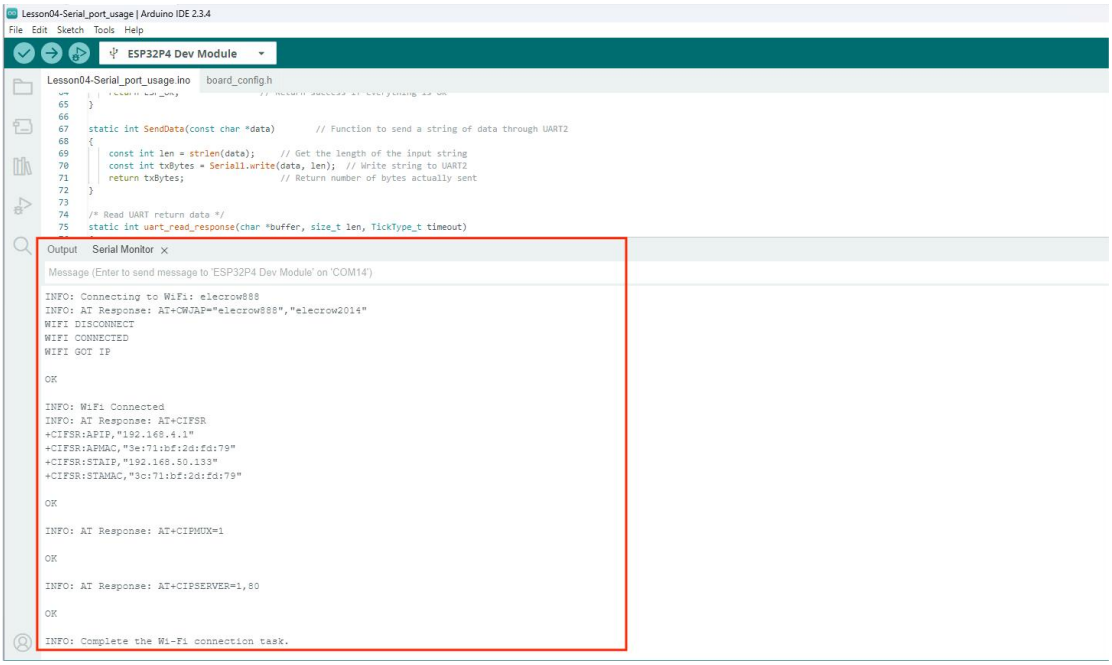
After the code upload completes, you can open the Serial Monitor built into the Arduino-IDE.



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



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
65 }
66
67 static int SendData(const char *data) // Function to send a string of data through UART2
68 {
69 const int len = strlen(data); // Get the length of the input string
70 const int txBytes = Serial1.write(data, len); // Write string to UART2
71 return txBytes; // Return number of bytes actually sent
72 }
73
74 /* Read UART return data */
75 static int uart_read_response(char *buffer, size_t len, TickType_t timeout)
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.

Lesson05--- Touchscreen

Introduction

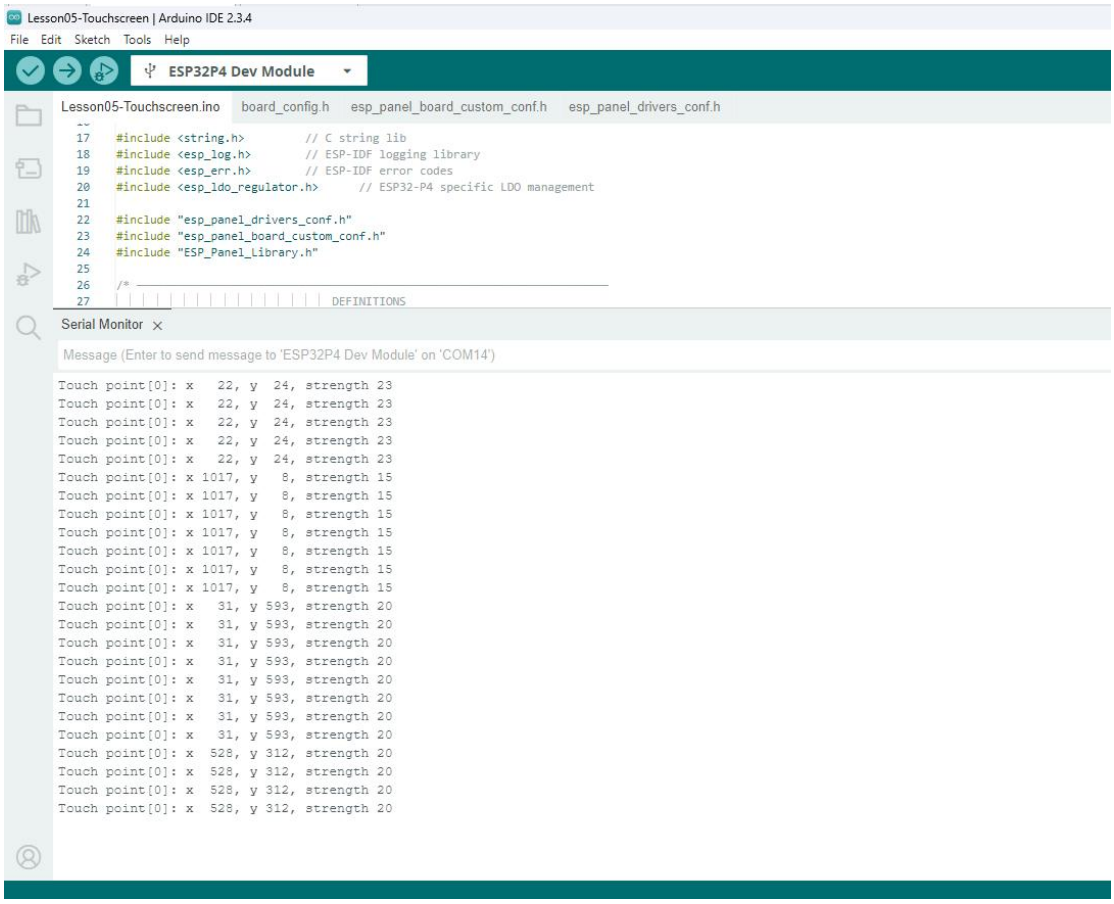
In this lesson, we will learn how to use the touch screen functionality of the CrowPanel Advanced ESP32-P4 HMI AI Display. When you tap the screen, you will see the touch coordinates displayed in the Serial Monitor.

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

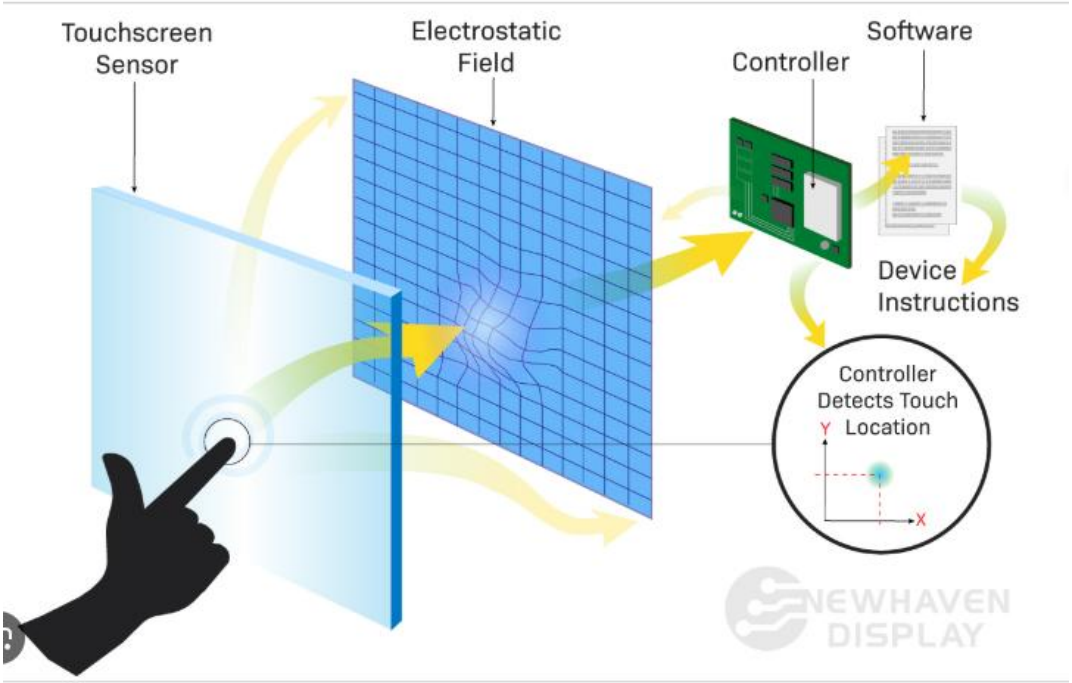
After running the code, you will be able to see the coordinates returned by the ESP32-P4 to you through the monitor on the Arduino-IDE at the moment when you touched the screen.



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.

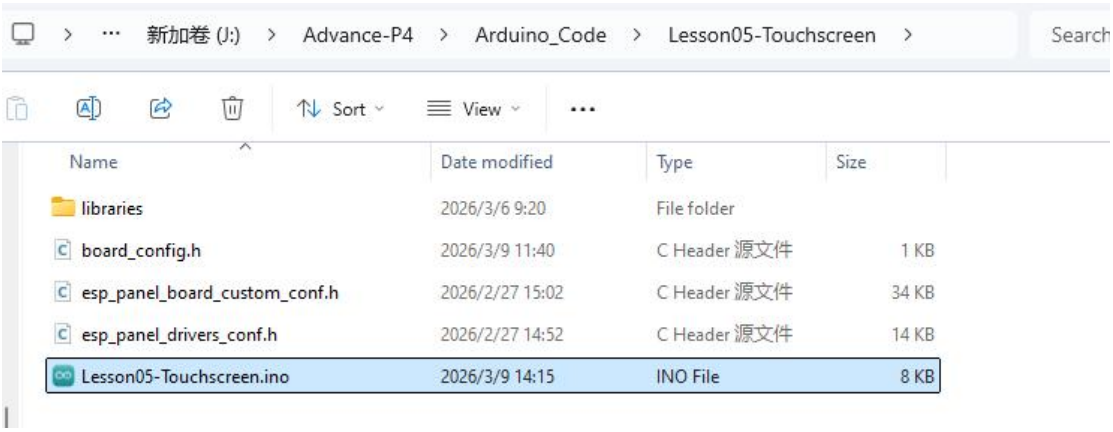
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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.

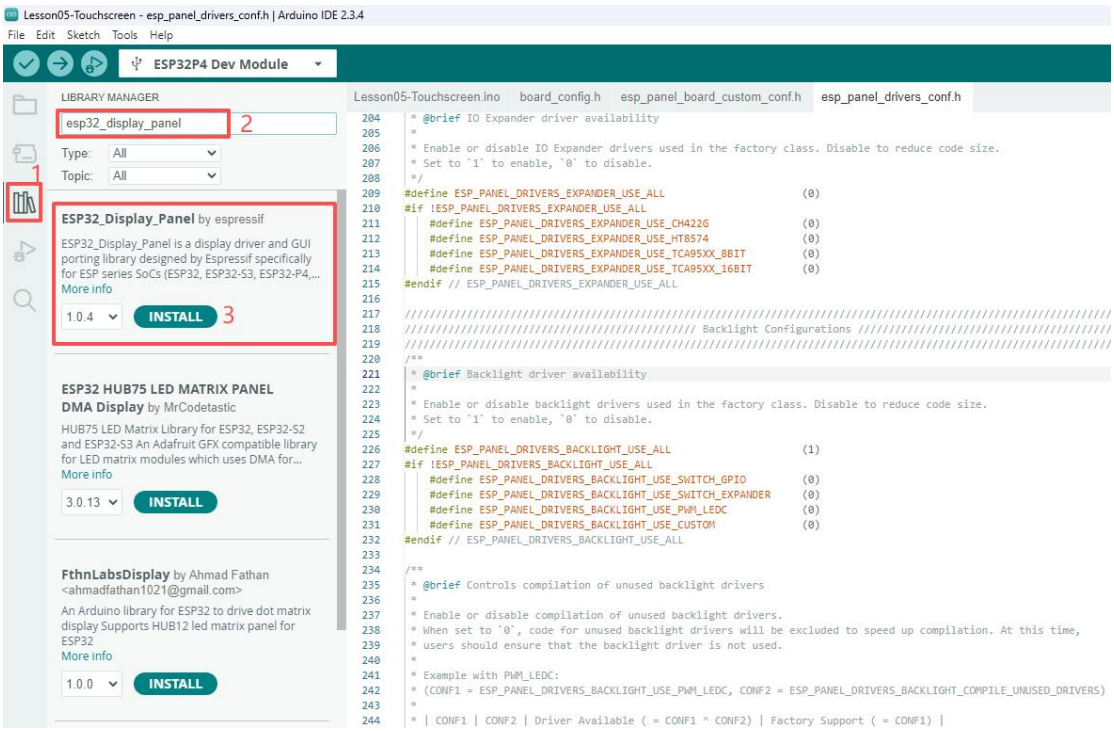
Double-click to open this lesson's code (the .ino file).



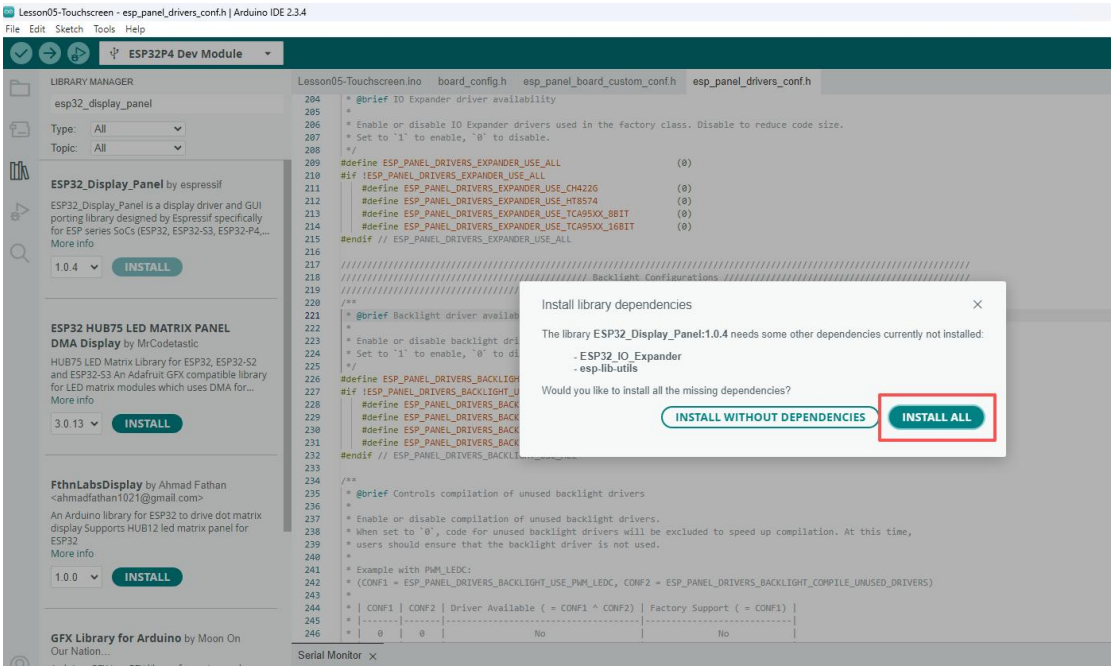
After opening, you can see the project's code, configuration file board_config.h, and other related screen driver files. Let's understand the project architecture for this lesson.

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?

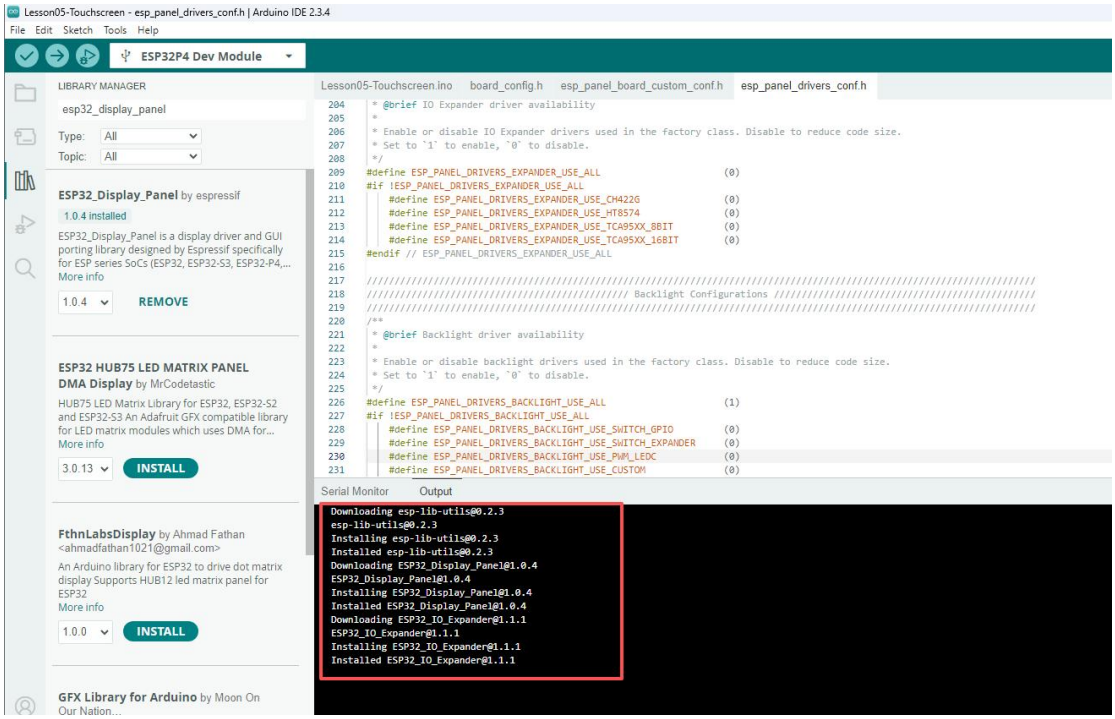
Search for esp32_display_panel in the Library Manager, select version 1.0.4 and install it.



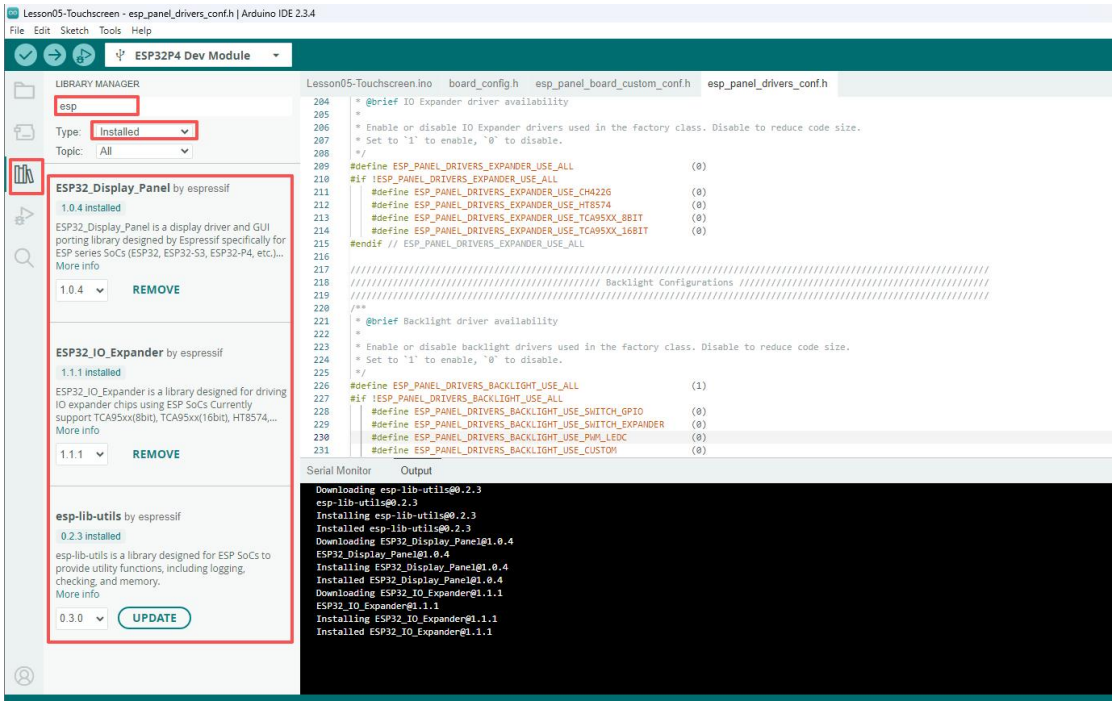
It will prompt you whether to install the related dependencies for ESP32_Display_Panel. We select Install All, as we need these dependencies to use it properly.



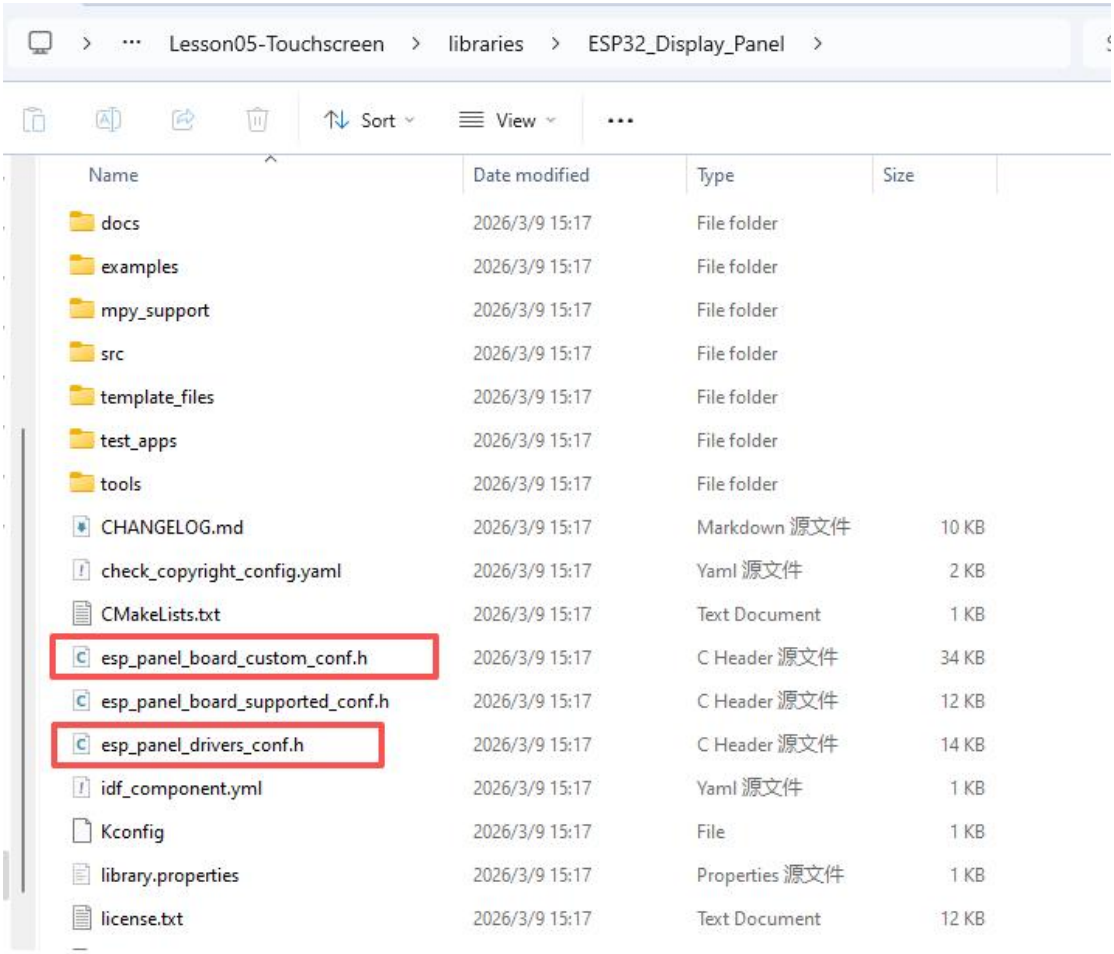
Installation completed successfully.



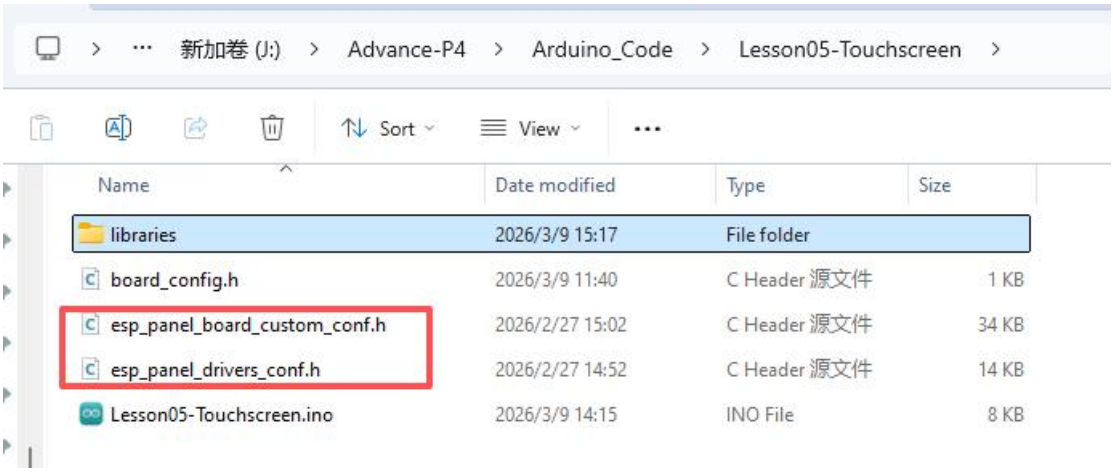
You can also see the installed libraries in the left sidebar.



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.



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



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

This is a custom board-level hardware configuration file, specifically used to hardcode all hardware parameters of our ESP32-P4 hardware (1024×600 EK79007 MIPI-DSI screen + GT911 touch screen) at once, including resolution, bus type, pins, I2C, reset/interrupt pins, timing, rotation/mirroring, etc. This allows the display library to precisely match your board and is the core configuration for software-hardware interfacing.

esp_panel_drivers_conf.h

This is a library driver trimming switch file, used to only enable the drivers you actually need (MIPI-DSI bus, EK79007 screen driver, GT911 touch driver) and disable all other unrelated driver code, thereby reducing firmware size, speeding up compilation, and avoiding driver conflicts. It is the master switch for controlling the library's compilation scope.

board_config.h

Configures the touch pins used by 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      (40)    // Reset pin
#define Touch_GPIO_INT      (42)    // Interrupt pin

#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

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

```

Alright, after understanding these configuration files, let's look at our main code .ino file.

This code is an Arduino program based on the ESP32-P4. Its core functionality is to initialize the MIPI-DSI display + GT911 touch screen, and read touch coordinates in real-time for printing.
Next, let's go through the code section by section to understand what each part does.

1.Header Files

```
/* ---------------------------------------------------------------
                        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
#include <esp_ldo_regulator.h>      // ESP32-P4 specific LDO management

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

```

This code serves to include various header files required for program operation, providing foundational support for subsequent screen display, touch reading, power management, and log debugging functionalities.

Among them, "#include "board_config.h"" is used to load the development board's pin configuration and hardware definitions; "#include <Arduino.h>" is the Arduino core library, providing Arduino APIs such as Serial, delay(), setup(), loop(), etc., so it is usually placed first to ensure the compiler correctly recognizes these functions.

Next, string.h provides C language string processing functions (such as strlen, strcpy, etc.), esp_log.h and esp_err.h are logging and error code libraries from the ESP-IDF framework, used for printing debug information and determining whether function execution was successful.

esp_ldo_regulator.h is a power management library specific to the ESP32-P4, used to control the chip's internal LDO voltage regulator, thereby providing appropriate voltage for the screen interface or touch circuit.

The last three files, esp_panel_drivers_conf.h, esp_panel_board_custom_conf.h, and ESP_Panel_Library.h, belong to the ESP32 Display Panel Library related configurations and driver interfaces. They are used to specify the display driver chip used (such as EK79007), touch chip (such as GT911), and the panel's hardware configuration, and provide unified APIs to initialize the screen, control display, and read touch data.

2. Macro Definitions

```
26  /* ──────────────────────────────────────────────────────────────────────
27  |||||||||||||||| DEFINITIONS
28  ────────────────────────────────────────────────────────────────────── */
29  #define PRINTF_ORIGINAL(fmt, ...) Serial.printf(fmt, ##__VA_ARGS__);
30  #define PRINTF_PRINT(fmt, ...)    Serial.print(fmt);
31  #define PRINTF_LN(fmt, ...)       Serial.println(fmt);
32
33  #define PRINTF_ERROR(fmt, ...)      do { \
34  ||||||||||||||||||||||||||||||||||||||  Serial.print("ERROR"); \
35  ||||||||||||||||||||||||||||||||||||||  Serial.printf(fmt, ##__VA_ARGS__); \
36  ||||||||||||||||||||||||||||||||||||||  Serial.print("\r\n"); \
37  ||||||||||||||||||||||||||||||||||||  } while(0)
38  #define PRINTF_WARN(fmt, ...)       do { \
39  ||||||||||||||||||||||||||||||||||||||  Serial.print("WARN: "); \
40  ||||||||||||||||||||||||||||||||||||||  Serial.printf(fmt, ##__VA_ARGS__); \
41  ||||||||||||||||||||||||||||||||||||||  Serial.print("\r\n"); \
42  ||||||||||||||||||||||||||||||||||||  } while(0)
43  #define PRINTF_INFO(fmt, ...)       do { \
44  ||||||||||||||||||||||||||||||||||||||  Serial.print("INFO: "); \
45  ||||||||||||||||||||||||||||||||||||||  Serial.printf(fmt, ##__VA_ARGS__); \
46  ||||||||||||||||||||||||||||||||||||||  Serial.print("\r\n"); \
47  ||||||||||||||||||||||||||||||||||||  } while(0)
48  #define PRINTF_DEBUG(fmt, ...)      do { \
49  ||||||||||||||||||||||||||||||||||||||  Serial.print("DEBUG: "); \
50  ||||||||||||||||||||||||||||||||||||||  Serial.printf(fmt, ##__VA_ARGS__); \
51  ||||||||||||||||||||||||||||||||||||||  Serial.print("\r\n"); \
52  ||||||||||||||||||||||||||||||||||||  } while(0)
53
```

This code mainly defines a set of macros for serial port log output, aiming to unify the program's debug information printing method, making the code more

standardized and clear during debugging.

The first three, PRINTF_ORIGINAL, PRINTF_PRINT, and PRINTF_LN, are simple wrappers for Arduino serial functions Serial.printf(), Serial.print(), and Serial.println(), used for directly outputting strings or formatted content.

The following PRINTF_ERROR, PRINTF_WARN, PRINTF_INFO, and PRINTF_DEBUG automatically add different log level prefixes (such as "ERROR", "WARN", "INFO", "DEBUG") when printing information, allowing quick distinction between error messages, warning messages, general information, or debug information in the serial monitor. For example, when calling PRINTF_INFO("WiFi Connected"), the serial port outputs INFO: WiFi Connected.

These macros internally use Serial.print and Serial.printf combined to output content, and manually add \r\n at the end to achieve line breaks. At the same time, the do { ... } while(0) syntax encapsulates multiple statements into a single entity, making the macro syntactically look like an ordinary function call and avoiding logical errors when used in if statements and similar constructs.

3. Variable Definitions

```
54  // Touch panel resolution
55  #define V_size 600      // Vertical resolution (Y-axis)
56  #define H_size 1024     // Horizontal resolution (X-axis)
57  /* ──────────────────────────────────────────────────────────
58  |||||||||||||||| GLOBAL VARIABLES
59  ────────────────────────────────────────────────────────── */
60  static const char *TAG = "TOUCH_APP";  // Tag for logging messages
61
62  // --- Declare the panel pointer globally ---
63  ESP_Panel *panel = nullptr;
64  /* ──────────────────────────────────────────────────────────
65  |||||||||||||||| FUNCTION PROTOTYPES
```

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

First, through macro definitions "#define V_size 600" and "#define H_size 1024", the touch screen resolution is specified, where V_size represents the vertical direction (Y-axis) resolution of 600 pixels, and H_size represents the horizontal direction (X-axis) resolution of 1024 pixels. This corresponds to the actual display resolution of 1024×600, and can serve as a reference size for subsequent touch coordinate processing or interface layout.

Next, "static const char *TAG = "TOUCH_APP";" is defined. This is a log tag string, typically used to identify the current module source when printing log information, for example indicating that the log comes from the touch application module, making it convenient to quickly locate issues during debugging.

The last line "ESP_Panel *panel = nullptr;" declares an ESP_Panel type pointer variable and initializes it to nullptr (indicating it currently points to no object). This pointer will point to an ESP_Panel object during program runtime, used to uniformly manage the display screen and touch device, such as initializing the screen, reading touch data, and other operations. Since it is defined as a global variable, this panel control object can be directly accessed in setup(), loop(), or other functions.

4. setup Section

The purpose of this setup() function code is to complete serial port debugging initialization, power configuration, and display and touch panel initialization when the system starts up, enabling the entire display and touch system to work properly.

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

```
71
72  void setup() {
73    // put your setup code here, to run once:
74
75    // Initialize the default Serial for debugging (UART0)
76    Serial.begin(115200);
77
78    // --- Power Configuration (LDO3 for MIPI D-PHY) ---
79    // ESP32-P4's MIPI D-PHY requires specific voltage to function.
80    // LDO3 is typically routed to the MIPI power rail on P4 hardware.
81    esp_err_t err = ESP_OK;
82    esp_ldo_channel_handle_t ldo3_handle = NULL;
83    esp_ldo_channel_config_t ldo3_cfg = {
84      .chan_id = 3,           // LDO Channel 3
85      .voltage_mv = 2500,     // Set to 2500mV (2.5V)
86    };
87
88    Serial.println("Initializing LDO3 to 2.5V...");
89    err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
90    if (err != ESP_OK) {
91      Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
92    } else {
93      Serial.println("LDO3 Power enabled successfully.");
94    }
95
96    // --- Power Configuration (LDO4 for I2C/touch pull up) ---
97    esp_ldo_channel_handle_t ldo4_handle = NULL;
98    esp_ldo_channel_config_t ldo4_cfg = {
99      .chan_id = 4,           // LDO Channel 4
100     .voltage_mv = 3300,     // Set to 3300mV (3.3V)
101   };
102
103   Serial.println("Initializing LDO4 to 3.3V...");
104   err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
105   if (err != ESP_OK) {
106     Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
107   } else {
108     Serial.println("LDO4 Power enabled successfully.");
109   }
110
111   // --- Initialize Display and Touch Panel ---
112   panel = new ESP_Panel();
113
```

First, serial communication is started via "Serial.begin(115200)" for outputting debug information. Then the program configures the ESP32-P4's internal LDO voltage regulator power supply: it first creates an "ldo3_cfg" structure and sets "chan_id = 3", "voltage_mv = 2500", then calls "esp_ldo_acquire_channel()" to enable LDO3 and output 2.5V voltage.

This is the operating voltage required for MIPI D-PHY (used for the MIPI-DSI display interface); if enabling fails, it prints error information via "esp_err_to_name()". Subsequently, LDO4 is configured to output 3.3V, which is typically used as the pull-up power supply for the I2C bus or touch chip (such as GT911).

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

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

  // --- Power Configuration (LDO3 for MIPI D-PHY) ---
  // ESP32-P4's MIPI D-PHY requires specific voltage to function.
  // LDO3 is typically routed to the MIPI power rail on P4 hardware.
  esp_err_t err = ESP_OK;
  esp_ldo_channel_handle_t ldo3_handle = NULL;
  esp_ldo_channel_config_t ldo3_cfg = {
    .chan_id = 3,           // LDO Channel 3
    .voltage_mv = 2500,     // Set to 2500mV (2.5V)
  };

  Serial.println("Initializing LDO3 to 2.5V...");
  err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
  if (err != ESP_OK) {
    Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
  } else {
    Serial.println("LDO3 Power enabled successfully.");
  }

  // --- Power Configuration (LDO4 for I2C/touch pull up) ---
  esp_ldo_channel_handle_t ldo4_handle = NULL;
  esp_ldo_channel_config_t ldo4_cfg = {
    .chan_id = 4,           // LDO Channel 4
    .voltage_mv = 3300,     // Set to 3300mV (3.3V)
  };

  Serial.println("Initializing LDO4 to 3.3V...");
  err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
  if (err != ESP_OK) {
    Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
  } else {
    Serial.println("LDO4 Power enabled successfully.");
  }
```

After power configuration is complete, the program creates a display panel control object via "panel = new ESP_Panel();", then calls "panel->init()" to initialize the MIPI-DSI bus as well as the screen driver EK79007 and touch chip GT911 (specific configurations are determined by the configuration files in the library); if initialization fails, the program enters an infinite loop and stops 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 (EK79007 + GT911)...");
  if (!panel->init()) {
    Serial.println("Panel Initialization Failed!");
    while (1) delay(100);
  }
```

Then "panel->begin()" is called to start the display panel, including executing the screen startup sequence and turning on the backlight; if this fails, the program also stops. Finally, it prints "Display and Touch system online," indicating that the display and touch system have been successfully initialized and are ready for normal use.

```
  // 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.");
}
```

5. loop Section

The purpose of this loop() function code is to continuously read touch data from the touch screen during program operation, and print the touch point coordinates and touch strength information 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 obtains the current panel's touch control object via "panel->getTouch()" and stores it in the touch variable.

If this object is not null (indicating the touch device has been successfully initialized), the program creates an "ESP_PanelTouchPoint point[10]" array to store touch data. Here it can store up to 10 touch points, because touch chips like GT911 support multi-touch.

```
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
```

Then "touch->readPoints(point, 10, 0)" is called to read data from the touch hardware. This function reads raw touch data and parses it into coordinate information, returning the number of touch points currently detected as "point_num".

The program then iterates through each touch point using a for loop, and uses "Serial.printf()" to print the X coordinate, Y coordinate, and touch strength of that touch point, allowing you to see the touch position in real-time in the Serial Monitor.

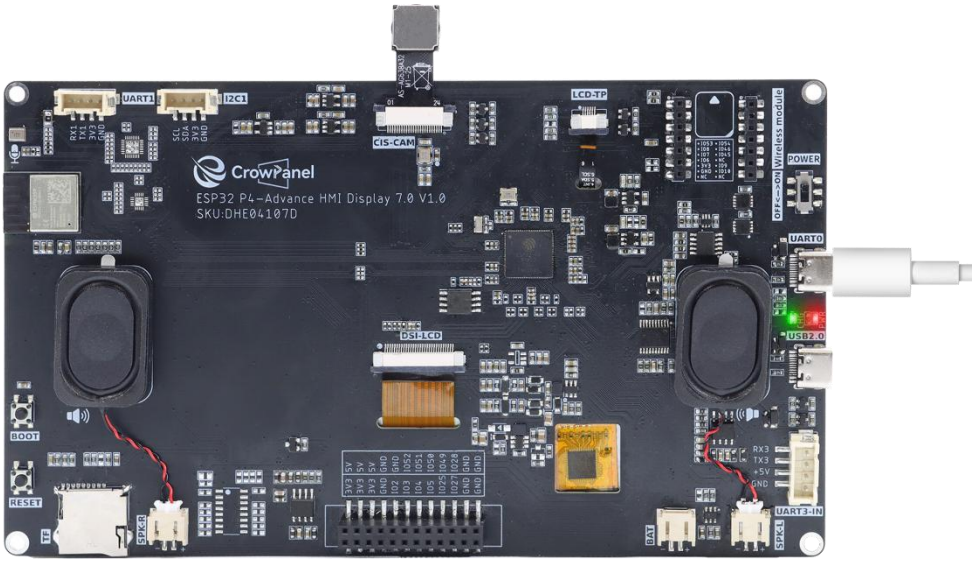
```
142     // --- Check the number of touch points ---
143     int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()
144
145     for (int i=0; i<point_num; i++) {
146         // Print the coordinates to Serial
147         Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\n", i, point[i].x, point[i].y, point[i].strength);
148     }
149
```

If the touch controller does not have interrupt mode enabled (isInterruptEnabled() returns false), the program executes delay(20) for a brief pause to reduce read frequency and maintain system stability. If the touch object is null (indicating the touch device was not properly initialized or does not exist), the program prints "IDLE loop" once per second to indicate that no touch device is currently working.

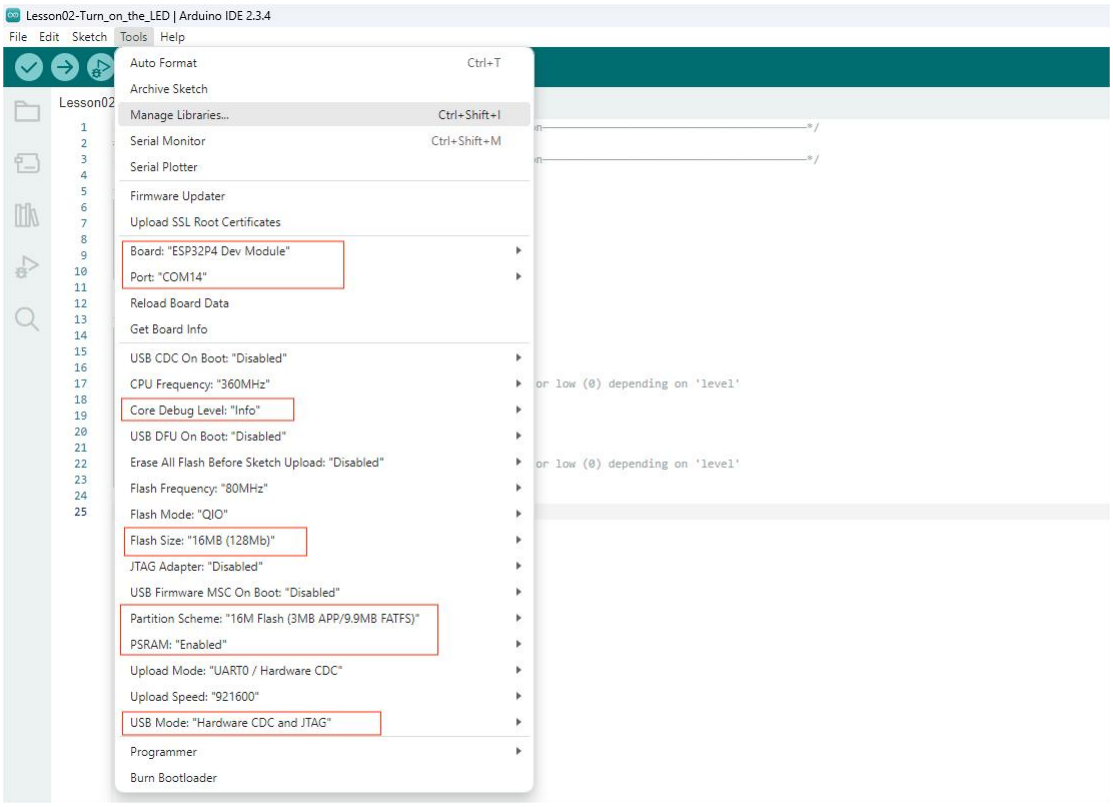
```
150         if (!touch->isInterruptEnabled()) {
151             delay(20);  // Small delay to maintain stability
152         }
153     } else {
154         Serial.println("IDLE loop");
155         delay(1000);
156     }
157 }
158
```

Programming Steps

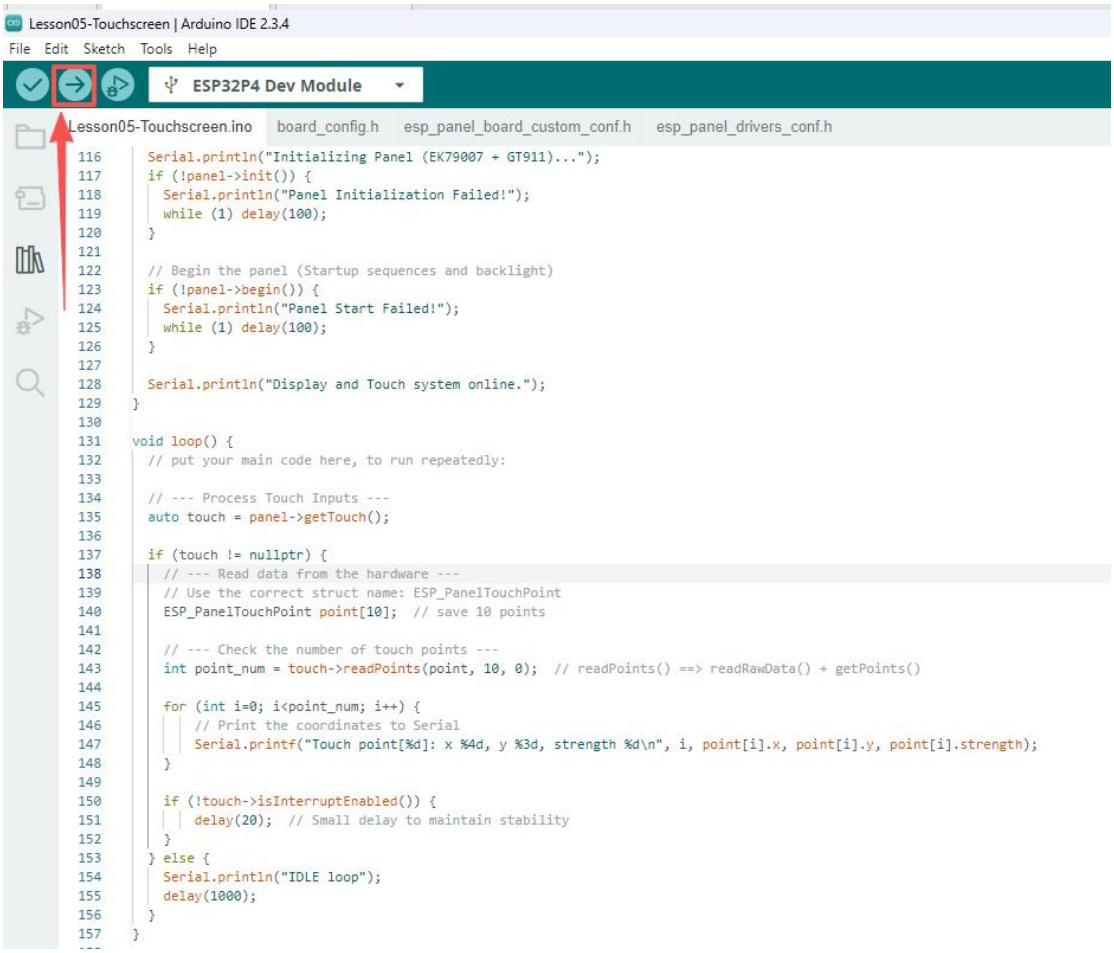
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



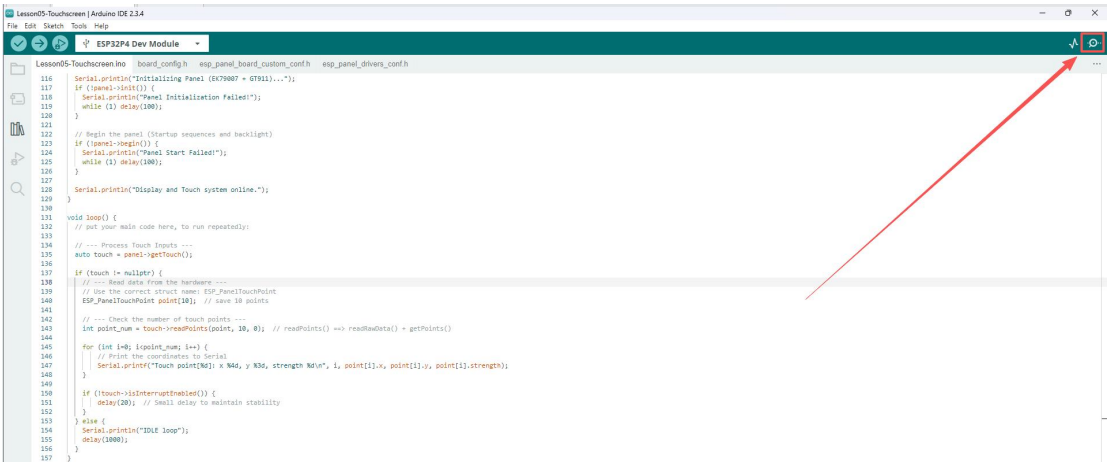
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



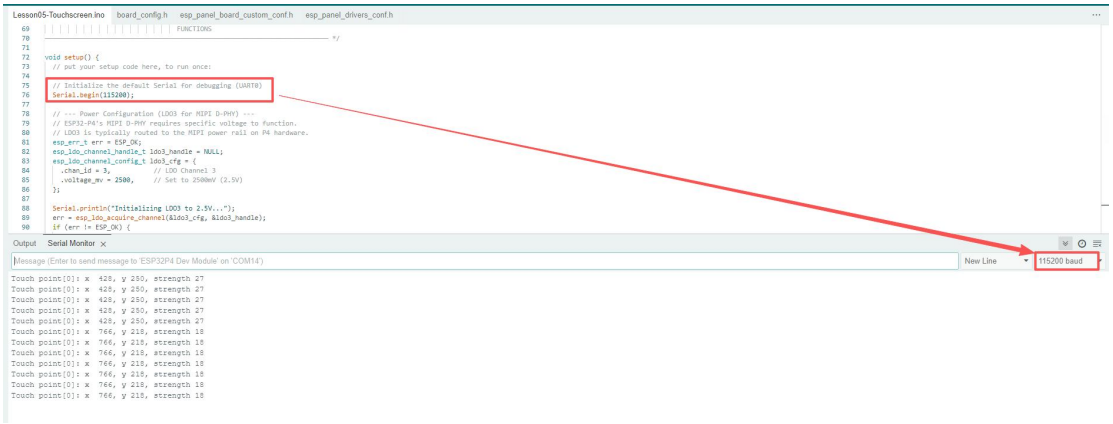
Then we compile and upload the code.



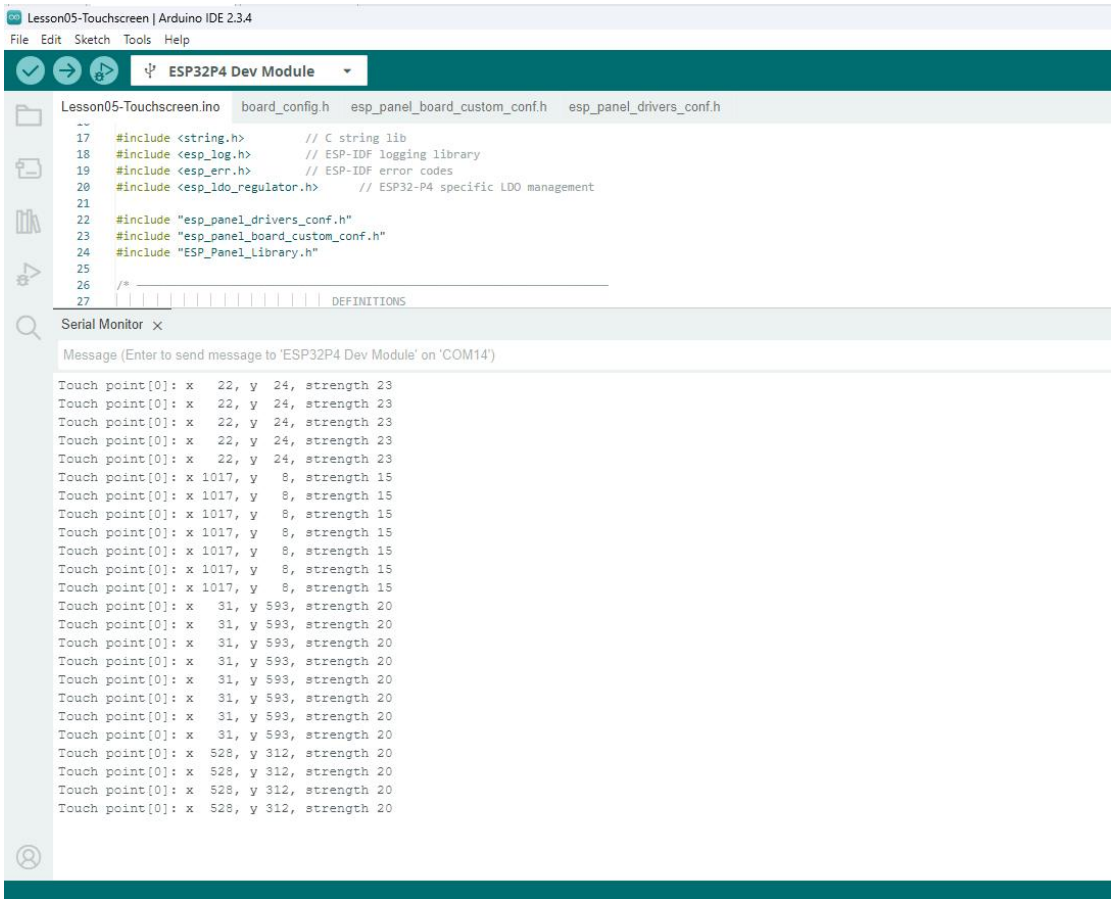
After the code upload completes, you can open the Serial Monitor built into the Arduino-IDE.



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



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


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
#include <string.h> // C string lib
#include <esp_log.h> // ESP-IDF logging library
#include <esp_err.h> // ESP-IDF error codes
#include <esp_ldo_regulator.h> // ESP32-P4 specific LDO management
#include "esp_panel_drivers_conf.h"
#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"
DEFINITIONS
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')
Touch point[0]: x 22, y 24, strength 23
Touch point[0]: x 1017, y 8, strength 15
Touch point[0]: x 31, y 593, strength 20
Touch point[0]: x 528, y 312, strength 20

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.

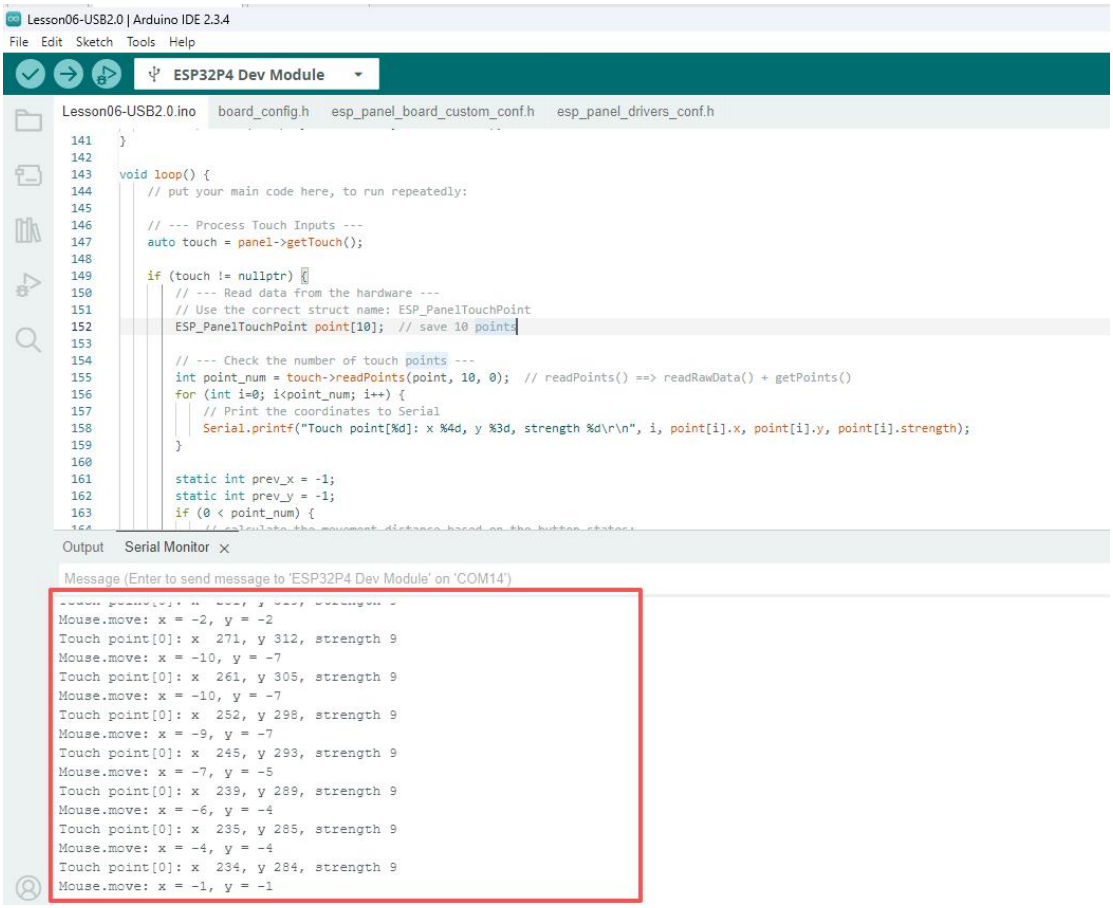
This will enable us to use the USB2.0 interface on the Advance-P4 as a mouse. When you swipe on the Advance-P4 screen, you will find that the mouse on your computer moves accordingly.

Learning Goals

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

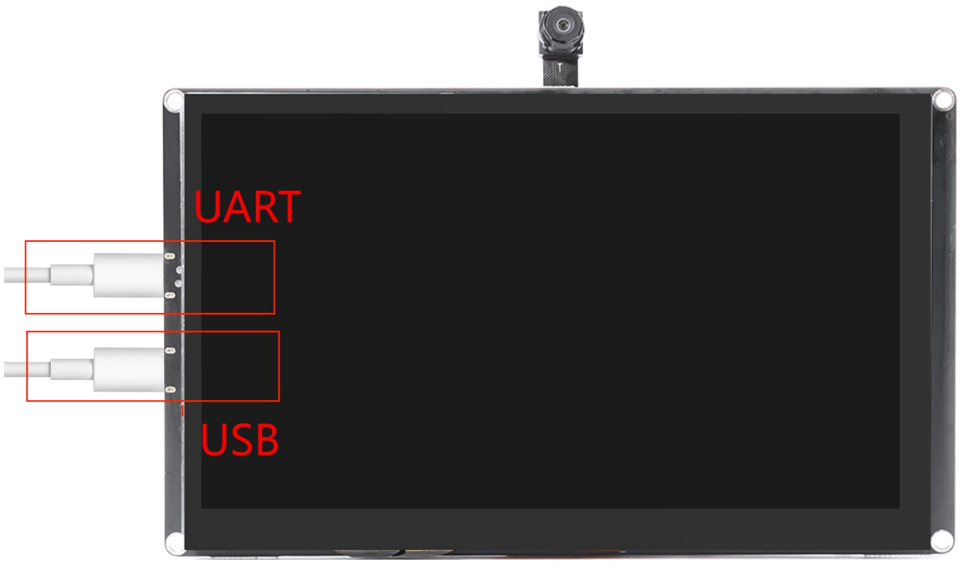
Preview of the Result

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.



Hardware Used in This Lesson

USB 2.0 on the Advance-P4



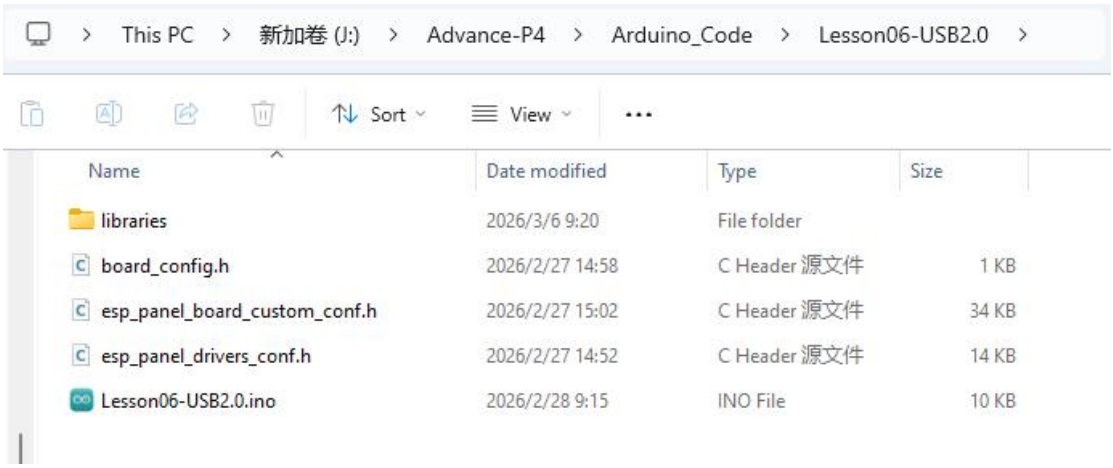
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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).



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

```
/* ──────────────────────────────────────────────────────────────────────
   |||||||||||||||| 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
#include <esp_ldo_regulator.h>      // ESP32-P4 specific LDO management

#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

```
// Touch panel resolution
#define V_size 600      // Vertical resolution (Y-axis)
#define H_size 1024     // Horizontal resolution (X-axis)
/* ──────────────────────────────────────────────────────────────────────
   |||||||||||||||| GLOBAL VARIABLES
   ──────────────────────────────────────────────────────────────────────*/
static const char *TAG = "TOUCH_APP";  // Tag for logging messages

// --- 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 definition of touch screen resolution and the

declaration of global objects, preparing for subsequent touch reading, log debugging, and USB mouse control.

First, through "#define V_size 600" and "#define H_size 1024", the touch screen resolution is defined, where V_size represents the vertical direction (Y-axis) resolution of 600 pixels, and H_size represents the horizontal direction (X-axis) resolution of 1024 pixels. These macro constants are typically used for coordinate calculation, boundary judgment, or coordinate mapping.

Next, "static const char *TAG = "TOUCH_APP";" defines a log tag string TAG, which can be used to identify the log source module when using the ESP-IDF logging system (such as ESP_LOGI, ESP_LOGE), making it convenient to distinguish output from different functional modules during debugging.

Then, "ESP_Panel *panel = nullptr;" declares an ESP_Panel class pointer variable and initializes it to 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 runtime. 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 a USB HID mouse object instance. This object encapsulates the operation interfaces of the mouse device. The program can send mouse movement, click, and other commands to the computer through it, such as Mouse.move() to control cursor movement, Mouse.press() and Mouse.release() to control mouse buttons, thereby achieving the function of operating the computer mouse using 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.

```
79  void setup() {
80      // put your setup code here, to run once:
81
82      // Initialize the default Serial for debugging (UART0)
83      Serial.begin(115200);
84
85      // initialize mouse control:
86      Mouse.begin();
87      USB.begin();
```

Then the program configures the ESP32-P4 chip's LDO power module: first defining the error status variable "esp_err_t err", then configuring the LDO3 channel ("chan_id = 3") to output 2.5V voltage. This voltage is used to power the MIPI D-PHY display interface. The power is enabled via "esp_ldo_acquire_channel()" and checked for success; if it fails, error information is output via the serial port. Next, the LDO4 channel ("chan_id = 4") is configured to output 3.3V voltage, used for the I2C bus and touch screen pull-up power. This is also initialized and status information is output.

```
esp_err_t err = ESP_OK;
esp_ldo_channel_handle_t ldo3_handle = NULL;
esp_ldo_channel_config_t ldo3_cfg = {
    .chan_id = 3,           // LDO Channel 3
    .voltage_mv = 2500,     // Set to 2500mV (2.5V)
};

Serial.println("Initializing LDO3 to 2.5V...");
err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
if (err != ESP_OK) {
    Serial.printf("LDO3 Power Error: %s\r\n", esp_err_to_name(err));
} else {
    Serial.println("LDO3 Power enabled successfully.");
}

// --- Power Configuration (LDO4 for I2C/touch pull up) ---
esp_ldo_channel_handle_t ldo4_handle = NULL;
esp_ldo_channel_config_t ldo4_cfg = {
    .chan_id = 4,           // LDO Channel 4
    .voltage_mv = 3300,     // Set to 3300mV (3.3V)
};

Serial.println("Initializing LDO4 to 3.3V...");
err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
if (err != ESP_OK) {
    Serial.printf("LDO4 Power Error: %s\r\n", esp_err_to_name(err));
} else {
    Serial.println("LDO4 Power enabled successfully.");
}
```

After power configuration is complete, the program creates an ESP_Panel object "panel = new ESP_Panel()", used to manage the display screen and touch screen hardware. Then "panel->init()" is called to initialize the MIPI-DSI bus, EK79007 display driver chip, and GT911 touch controller (related configurations come from ESP_Panel_Conf.h in the library files). If initialization fails, the program enters an infinite loop and stops 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 (EK79007 + GT911)...");
if (!panel->init()) {
    Serial.println("Panel Initialization Failed!");
    while (1) delay(100);
}
```

Then "panel->begin()" is called to start the display panel, including executing the display startup sequence and turning on the backlight. If this fails, the program also stops. Finally, "Display and Touch system online." is printed to the serial port, indicating that the screen display and touch system have successfully started and can

now enter the main loop to process touch input and control the mouse.

```
133     // Begin the panel (Startup sequences and backlight)
134     if (!panel->begin()) {
135         Serial.println("Panel Start Failed!");
136         while (1) delay(100);
137     }
138
139     Serial.println("Display and Touch system online.");
140 }
141
```

4、loop Section

This loop() function code implements the core logic of continuously reading touch screen data and converting touch operations into USB mouse control signals.

The program first obtains the touch controller object via "panel->getTouch()". If the object exists, it indicates that touch functionality is available.
It then creates an "ESP_PanelTouchPoint point[10]" array to store up to 10 touch point data, and calls "touch->readPoints(point, 10, 0)" to read the current number of touch points and their coordinate information from the touch chip (such as GT911).

The program uses a for loop to output each touch point's X coordinate, Y coordinate, and touch strength (pressure/strength) to the serial port for debugging purposes.

```
142 void loop() {
143     // put your main code here, to run repeatedly:
144
145     // --- Process Touch Inputs ---
146     auto touch = panel->getTouch();
147
148     if (touch != nullptr) {
149         // --- Read data from the hardware ---
150         // Use the correct struct name: ESP_PanelTouchPoint
151         ESP_PanelTouchPoint point[10];  // save 10 points
152
153         // --- Check the number of touch points ---
154         int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()
155         for (int i=0; i<point_num; i++) {
156             // Print the coordinates to Serial
157             Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\r\n", i, point[i].x, point[i].y, point[i].strength);
158         }
```

Then two static variables "prev_x" and "prev_y" are used to record the previous touch position. When at least one touch point is detected ("point_num > 0"), the program calculates the displacement difference "xDistance" and "yDistance" between the current touch point and the previous touch point.

If the displacement is not zero, it calls "Mouse.move(xDistance, yDistance, 0)" to send a mouse movement command to the computer, causing the mouse cursor to move in the direction of the finger swipe. Then "prev_x" and "prev_y" are updated to the current coordinates.

```
160         static int prev_x = -1;
161         static int prev_y = -1;
162         if (0 < point_num) {
163           // calculate the movement distance based on the button states:
164           if (-1 != prev_x || -1 != prev_y) {
165             int xDistance = (point[0].x - prev_x);
166             int yDistance = (point[0].y - prev_y);
167             // if X or Y is non-zero, move:
168             if ((xDistance!=0) || (yDistance!=0)) {
169               Mouse.move(xDistance, yDistance, 0);
170               Serial.printf("Mouse.move: x = %2d, y = %2d\r\n", xDistance, yDistance);
171             }
172           }
173           prev_x = point[0].x;
174           prev_y = point[0].y;
```

At the same time, the program simulates mouse button logic: when a touch is detected, if the left mouse button is not already pressed, it executes "Mouse.press(MOUSE_LEFT)", equivalent to holding down the left mouse button.

```
176             // if the mouse is not pressed, press it:
177             if (!Mouse.isPressed(MOUSE_LEFT)) {
178               Mouse.press(MOUSE_LEFT);
179             }
```

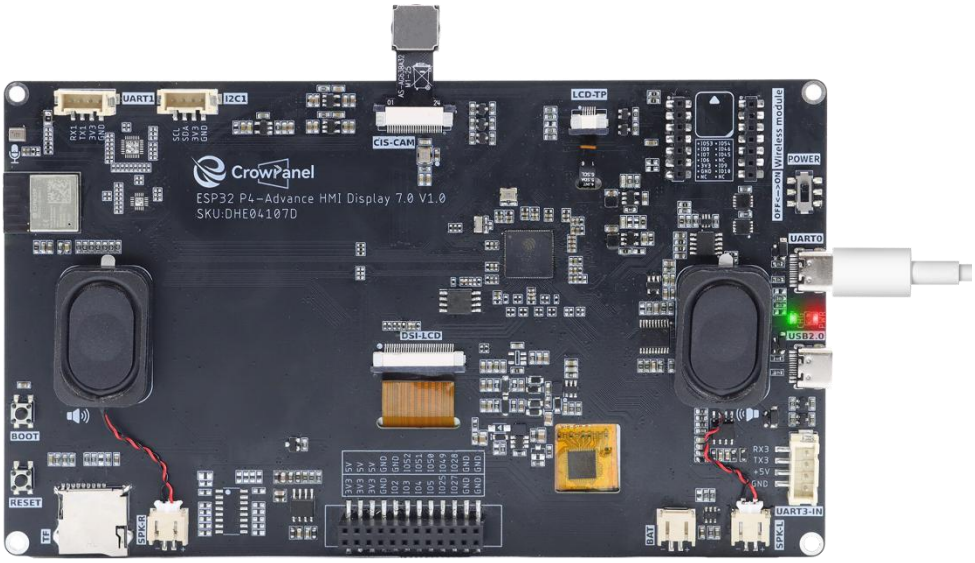
When there are no touch points (finger leaves the screen), the program resets "prev_x" and "prev_y" to -1, and calls "Mouse.release(MOUSE_LEFT)" to release the left mouse button, thereby achieving a mouse operation effect similar to "click and drag."

Finally, "delay(20)" limits the loop execution speed to avoid excessively fast mouse movement. If the touch object does not exist, the program outputs "IDLE loop" and checks the status once per second.

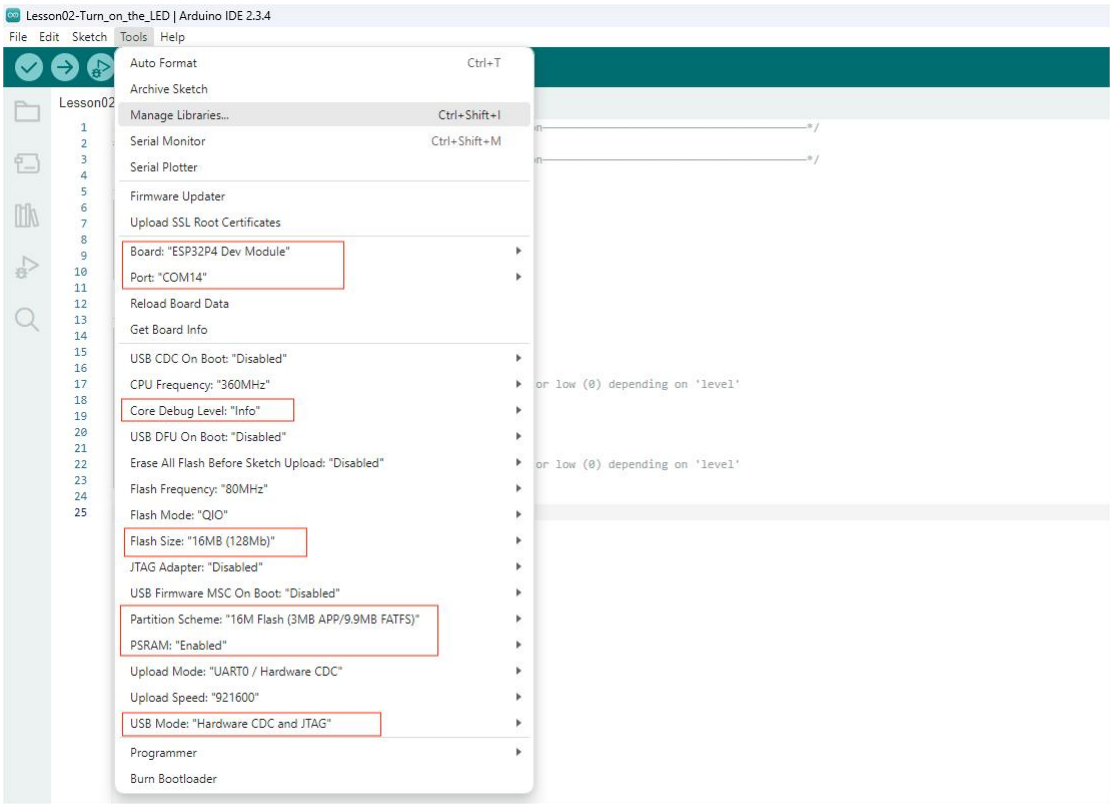
```
180         } else {
181           prev_x = -1;
182           prev_y = -1;
183
184           // if the mouse is pressed, release it:
185           if (Mouse.isPressed(MOUSE_LEFT)) {
186             Mouse.release(MOUSE_LEFT);
187           }
188         }
189
190         // a delay so the mouse doesn't move too fast:
191         delay(20);
192
193       } else {
194         Serial.println("IDLE loop");
195         delay(1000);
196       }
197     }
```

Programming Steps

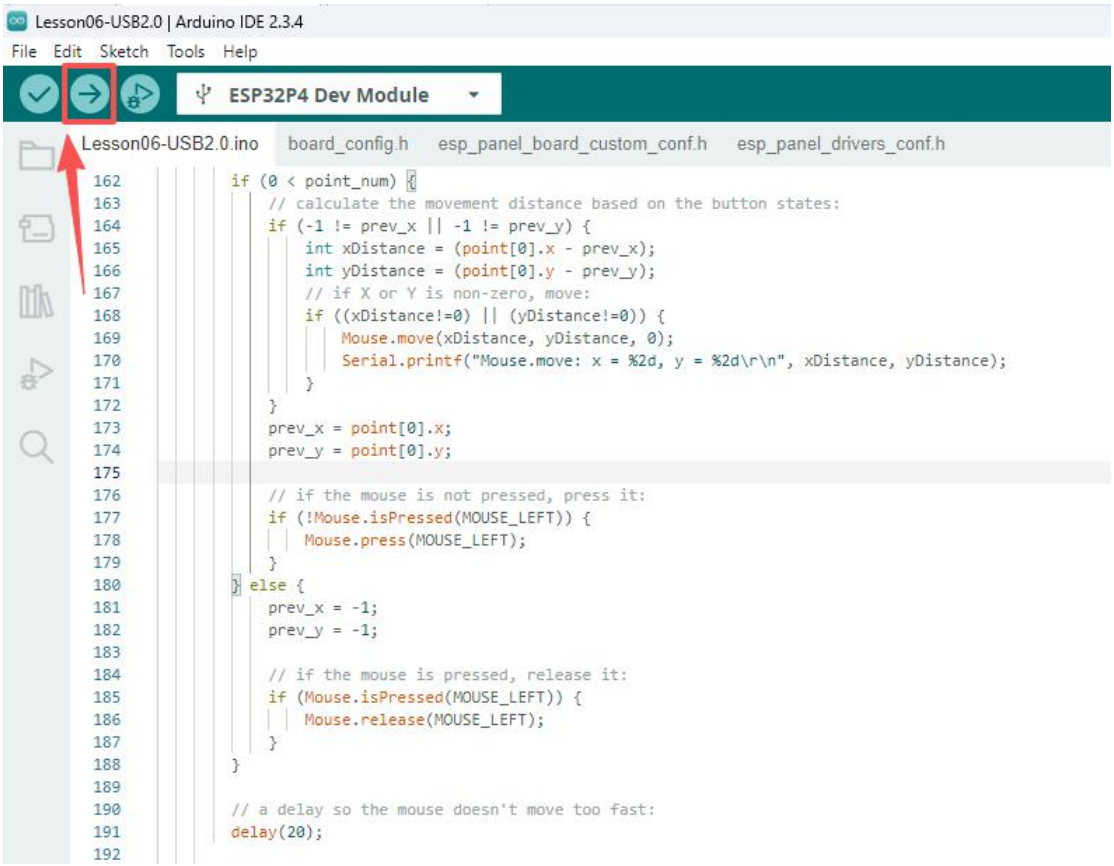
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



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

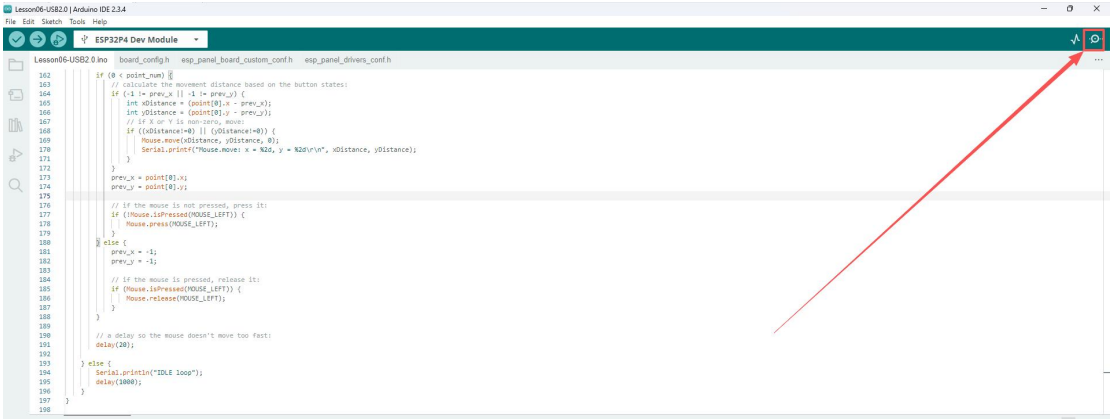


Then we compile and upload the code.

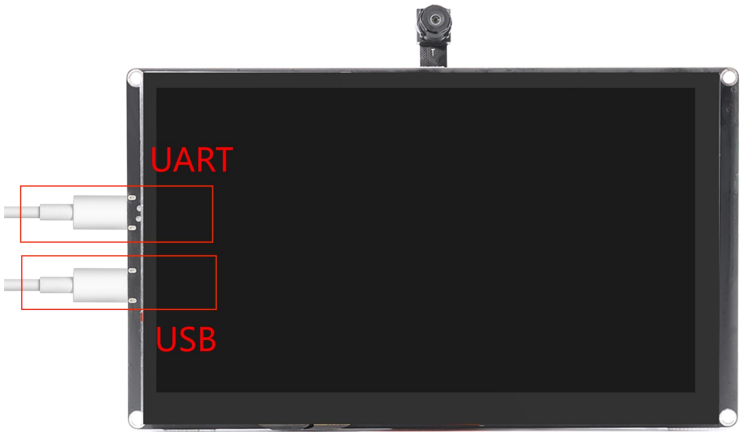



```
162         if (0 < point_num) {
163           // calculate the movement distance based on the button states:
164           if (-1 != prev_x || -1 != prev_y) {
165             int xDistance = (point[0].x - prev_x);
166             int yDistance = (point[0].y - prev_y);
167             // if X or Y is non-zero, move:
168             if ((xDistance!=0) || (yDistance!=0)) {
169               Mouse.move(xDistance, yDistance, 0);
170               Serial.printf("Mouse.move: x = %2d, y = %2d\r\n", xDistance, yDistance);
171             }
172           }
173           prev_x = point[0].x;
174           prev_y = point[0].y;
175
176           // if the mouse is not pressed, press it:
177           if (!Mouse.isPressed(MOUSE_LEFT)) {
178             Mouse.press(MOUSE_LEFT);
179           }
180         } else {
181           prev_x = -1;
182           prev_y = -1;
183
184           // if the mouse is pressed, release it:
185           if (Mouse.isPressed(MOUSE_LEFT)) {
186             Mouse.release(MOUSE_LEFT);
187           }
188         }
189
190         // a delay so the mouse doesn't move too fast:
191         delay(20);
192
```

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.



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.

```
153
154     // --- Check the number of touch points ---
155     int point_num = touch->readPoints(point, 10, 0);  // readPoints() ==> readRawData() + getPoints()
156     for (int i=0; i<point_num; i++) {
157         // Print the coordinates to Serial
158         Serial.printf("Touch point[%d]: x %4d, y %3d, strength %d\r\n", i, point[i].x, point[i].y, point[i].strength);
159     }
160
161     static int prev_x = -1;
162     static int prev_y = -1;
163     if (0 < point_num) {
```

Output Serial Monitor ×

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

```
Mouse.move: x = -2, y = -2
Touch point[0]: x  271, y 312, strength 9
Mouse.move: x = -10, y = -7
Touch point[0]: x  261, y 305, strength 9
Mouse.move: x = -10, y = -7
Touch point[0]: x  252, y 298, strength 9
Mouse.move: x = -9, y = -7
Touch point[0]: x  245, y 293, strength 9
Mouse.move: x = -7, y = -5
Touch point[0]: x  239, y 289, strength 9
Mouse.move: x = -6, y = -4
Touch point[0]: x  235, y 285, strength 9
Mouse.move: x = -4, y = -4
Touch point[0]: x  234, y 284, strength 9
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.

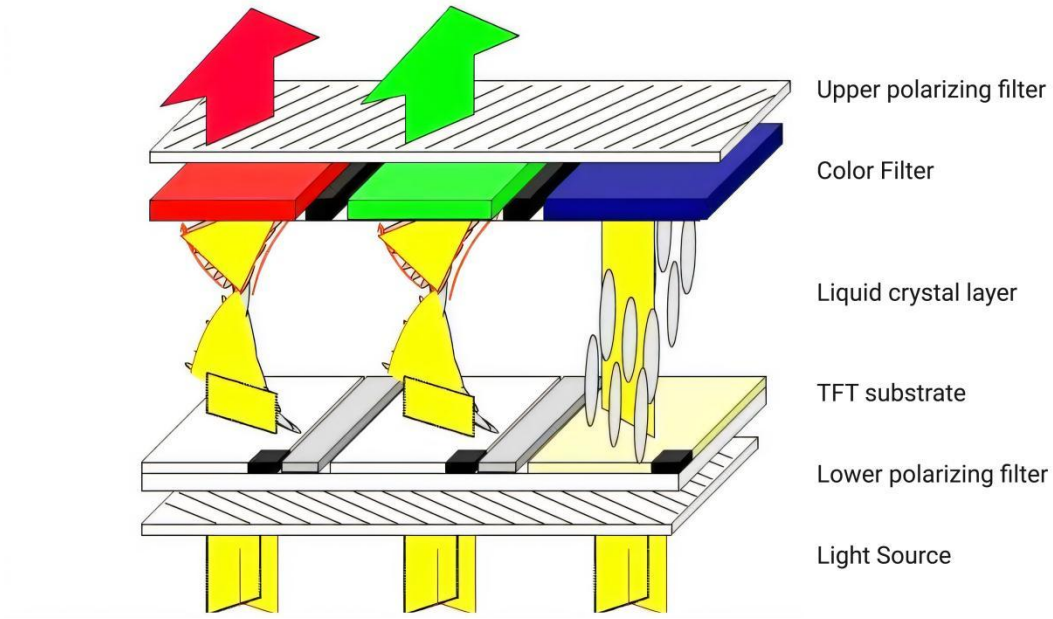
Preview of the Result

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.

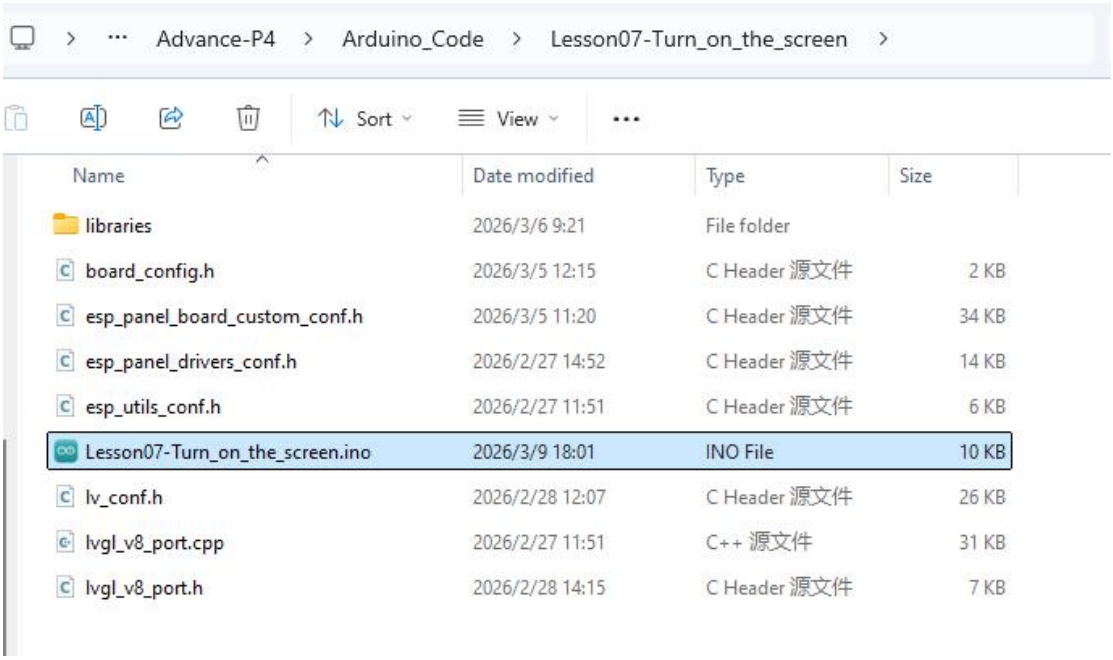
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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).



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.

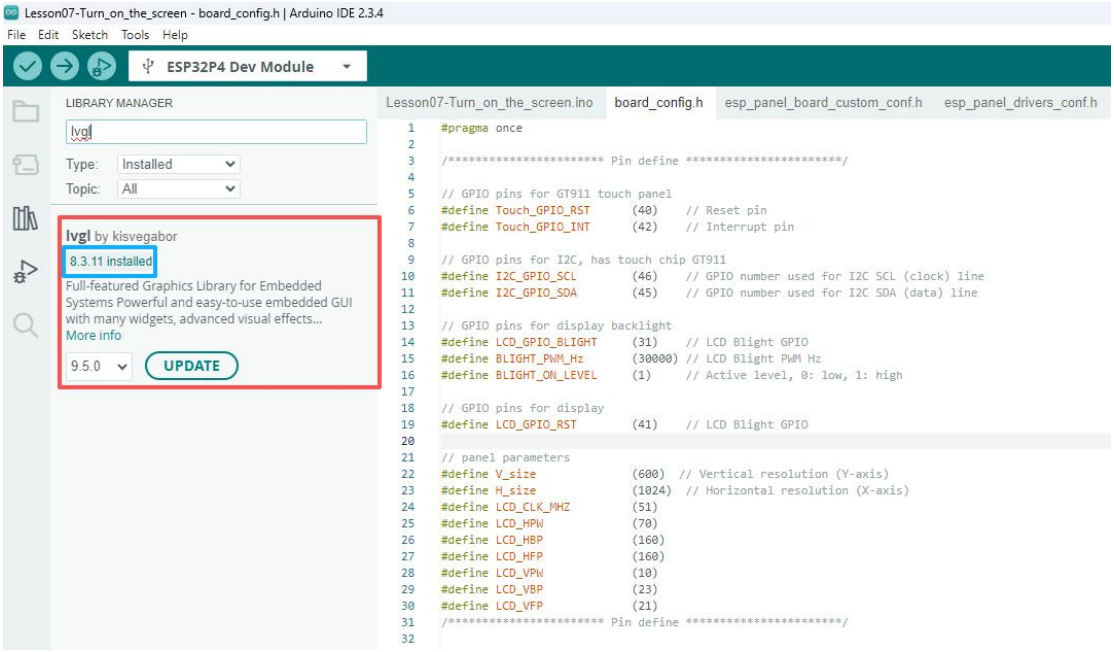
Among them:
1. The libraries folder is the project's dependency library storage directory, storing core third-party libraries such as ESP32_Display_Panel and LVGL v8 according to Arduino library management standards.

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.



2. board_config.h: Hardware Pin and Panel Parameter Configuration File

board_config.h is the project's underlying hardware configuration file, specifically defining all hardware-related pin mappings and panel parameters for the ESP32-P4 development board. Through macro definitions, it clarifies the GT911 touch chip's reset/interrupt pins, I2C communication pins, LCD backlight control pin, LCD reset pin, as well as core hardware parameters such as screen resolution (1024 × 600) and MIPI-DSI timing parameters (clock frequency, front/back porch/sync pulse width). This is the foundation that binds the entire project to the physical development board hardware. All upper-layer drivers and applications rely on these configurations to correctly address and control the hardware.

```
#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 **********/
```

3.esp_panel_board_custom_conf.h ： ESP Display Panel Custom Board-Level Configuration File

esp_panel_board_custom_conf.h is the custom board-level configuration file for the ESP32_Display_Panel library. It inherits hardware parameters from board_config.h and encapsulates them into macro definitions recognizable by the library. It uniformly configures the LCD controller (EK79007), bus type (MIPI-DSI), resolution, touch controller (GT911), I2C communication parameters, backlight control (PWM mode), etc. It also defines hook function interfaces before and after library initialization. This is the core configuration file for the ESP32_Display_Panel library to adapt to custom ESP32-P4 development boards, determining how the library initializes LCD, touch, backlight, and other peripherals.

4. esp_panel_drivers_conf.h：ESP Display Panel Driver Selection Configuration File

esp_panel_drivers_conf.h is the driver trimming and compilation configuration file for the ESP32_Display_Panel library. Through macro definitions, it precisely controls which driver modules are included during library compilation. It disables all unrelated drivers, only enabling the MIPI-DSI bus driver, EK79007 LCD driver, GT911 touch driver, and PWM backlight driver. It also configures maximum touch points, compilation optimization options, etc. This ensures the library functions fully adapt to project hardware while minimizing compiled code size to the greatest extent. It is the core of library-level functional trimming and performance optimization.

5. esp_utils_conf.h：ESP Utils Library General Configuration File

esp_utils_conf.h is the general configuration file for the esp_utils tool library that the project depends on. It manages the tool library's basic behaviors: including error handling methods for null pointers/memory allocation (print error logs), log output level (INFO level), memory allocation strategy (default using standard library malloc/free, switchable to ESP-IDF exclusive memory allocation), etc. It provides unified logging, memory management, and error handling specifications for the entire project, forming the foundation for ensuring project code robustness and maintainability.

6. lv_conf.h：LVGL Graphics Library Core Configuration File

lv_conf.h is the core compilation configuration file for the LVGL v8.3.11 graphics library. Through macro definitions, it comprehensively manages LVGL's functional scope and resource consumption: configuring color depth as RGB565 (16-bit), memory allocation strategy as system malloc/free, hardware abstraction layer parameters (refresh rate 30ms, input device read cycle 30ms), while enabling complex drawing engine (shadows, gradients, rounded corners, etc.), full range of common widgets (buttons, sliders, labels, etc.), Montserrat full font family, as well as widget demonstrations, performance benchmarking, and other Demo functions. It is the "master switch" of the LVGL graphics system, determining what UI capabilities LVGL can provide and how much memory it consumes. It is key to adapting to ESP32-P4 hardware resources.

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

8. lvgl_v8_port.cpp：LVGL Porting Layer Hardware Adaptation Implementation File
lvgl_v8_port.cpp is the core implementation file for porting LVGL to the ESP32-P4 platform, fully implementing the adaptation logic between LVGL and hardware: implementing anti-tearing optimization for MIPI-DSI screens (synchronizing frame buffer switching through VSYNC interrupts), optimizing rotation algorithms for 16-bit color depth screens (block copy improves performance by 10~20 times), completing LCD display buffer initialization, touch interrupt synchronization and coordinate mapping, while creating an independent LVGL task to handle timer events, and ensuring multi-threaded access safety to LVGL API through recursive mutexes. It is the core carrier transforming LVGL from a universal graphics library into a runnable graphics system on ESP32-P4.

(The above library files can be used directly. We have already modified them for you, and they can smoothly run screen display-related functions and LVGL functions on the CrowPanel Advanced ESP32-P4 HMI AI Display.)

9. Lesson07-Turn_on_the_screen.ino：Arduino Main Program Entry File
Lesson07-Turn_on_the_screen.ino is the project's main program entry file, following the Arduino programming paradigm, containing setup() and loop() functions. In setup(), it sequentially completes serial port debugging initialization, ESP32-P4 exclusive LDO power configuration (providing 2.5V/3.3V power for MIPI-DSI and I2C), ESP32_Display_Panel library initialization (LCD + touch), LVGL porting layer initialization, and calls LVGL interfaces to create basic UI (such as displaying "Hello Elecrow"). In loop(), it maintains LVGL task operation. It is the complete process carrier from hardware power-on to graphical interface display for the entire project, and also the core entry for users to write business logic.

Alright, next let's understand the code content in the main code ino file.

1. Header Files

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

```
14  #include "board_config.h"   // board pin define
15  #include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs
16
17  #include <string.h>         // C string lib
18  #include <esp_log.h>        // ESP-IDF logging library
19  #include <esp_err.h>        // ESP-IDF error codes
20  #include <esp_ldo_regulator.h>      // ESP32-P4 specific LDO management
21
22  #include "esp_panel_drivers_conf.h"
23  #include "esp_panel_board_custom_conf.h"
24  #include "ESP_Panel_Library.h"
25
26  #include <lvgl.h>
27  #include "lvgl_v8_port.h"
```

First, #include "board_config.h" is used to load the development board's hardware configuration file, which typically contains board-level information such as GPIO pin definitions and display interface configurations.

#include <Arduino.h> is the Arduino core library. It provides Arduino's basic functions such as setup(), loop(), Serial serial communication, and delay(), so it must be placed relatively early to ensure these APIs can be correctly recognized.

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

#include <esp_ldo_regulator.h> is an ESP32-P4 exclusive LDO power management library, used to configure the chip's internal low-dropout linear regulator, providing stable voltage for the MIPI display interface or other peripherals.

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, used to configure and drive display and touch screen hardware (such as EK79007 display driver and GT911 touch chip).

Finally, #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 widgets and graphics functions, while lvgl_v8_port.h is responsible for connecting LVGL with underlying display drivers and touch drivers, enabling developers to create graphical interfaces on the screen and respond to touch input.

2. Namespace

The purpose of these two lines of code is to introduce contents from specified namespaces, allowing the program to omit namespace prefixes when using classes, functions, or variables from these libraries, thereby simplifying code writing.

```
28
29  using namespace esp_panel::drivers;
30  using namespace esp_panel::board;
31  /*
```

using namespace esp_panel::drivers; indicates introducing all members from the drivers namespace in the esp_panel library (such as LCD drivers, touch drivers, and related classes and interfaces) directly into the current file scope. This way, when using these driver classes, you don't need to write the complete esp_panel::drivers::class name.

using namespace esp_panel::board; introduces contents from the board namespace. This namespace typically contains development board-related wrapper classes, such as the Board class, which is responsible for uniformly managing hardware resources like display screens, touch screens, and bus interfaces. Therefore, in the code you can directly write Board *board = new Board(); without needing to write esp_panel::board::Board.

3. Macro Definitions

This code creates a set of shortcut names for commonly used colors for the LVGL graphical interface library through #define macro definitions, 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 lv_color_make(R, G, B) function. This function is a color constructor provided by LVGL, used to generate an LVGL-recognizable color type lv_color_t based on RGB primary color values (red, green, blue).

For example, LV_COLOR_RED is defined as lv_color_make(0xFF, 0x00, 0x00), indicating red component of 255, green of 0, and blue of 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 define green, blue, white, black, gray, and yellow respectively.
The purpose of these macros is to improve code readability and simplify UI development. When developers set LVGL widget styles or background colors, they only need to write LV_COLOR_RED or LV_COLOR_WHITE, without having to repeatedly write the RGB parameters of lv_color_make(…) every time, making interface code clearer and more readable.

4. Global Variables

This code is mainly used to define log identifiers and declare display panel object pointers.

```
static const char *TAG = "TOUCH_APP";  // Tag for logging messages

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

First, static const char *TAG = "TOUCH_APP"; defines a string constant pointer TAG with value "TOUCH_APP". This is typically used as a tag in the logging system. When using ESP-IDF logging functions (such as ESP_LOGI, ESP_LOGE, etc.) to output debug information, this tag can be used to identify the log source module, making it easier to distinguish log information from different functional modules during debugging. Here static indicates the variable's scope is limited to the current file, and const char * indicates this is a pointer to a constant string.

Next, ESP_Panel *panel = nullptr; declares an ESP_Panel class pointer variable panel and initializes it to a null pointer nullptr. This object is mainly used to manage display and touch screen hardware (such as initializing display interfaces, reading touch data, etc.). Since it is defined as a global variable, it can be shared and accessed in the program's setup(), loop(), and other functions. Typically, an actual object is created for it during the program initialization phase to complete control of the display and touch devices.

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.

```
87  static void lvgl_show_hello_elecrow(void) {
88      // 1. Lock LVGL: ensure thread-safe operations
89      if (lvgl_port_lock(-1) != true) {  // 0 means non-blocking wait for the lock (timeout = 0)
90          MAIN_ERROR("LVGL lock failed");  // Print error if lock fails
91          return;  // Exit function
92      }
```

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.

```
94      // 2. Create screen background (optional: set background color for better text visibility)
95      lv_obj_t *screen = lv_scr_act();  // Get current active screen object
96      lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);  // Set background color to white
97      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.

```
99      // 3. Create label object (parent object = current screen)
100     lv_obj_t *hello_label = lv_label_create(screen);  // Create label
101     if (hello_label == NULL) {  // Check if creation failed
102         MAIN_ERROR("Create LVGL label failed");  // Log error
103         lvgl_port_unlock();  // Unlock LVGL before returning
104         return;  // Exit function
105     }
```

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

```
107     // 4. Set label text content
108     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().

```
110     // 5. Configure label style (font, color, background)
111     static lv_style_t label_style;  // Define a style object
112     lv_style_init(&label_style);    // Initialize style object
113     // Set font: Montserrat size 42 (must be enabled in LVGL config)
114     lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
115     // Set text color to black (contrast with white background)
116     lv_style_set_text_color(&label_style, LV_COLOR_BLACK);
117     // Set label background transparent (avoid blocking screen background)
118     lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
119     // Apply style to the label
120     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.

```
    // 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. setup

This setup() function code mainly completes system initialization, power configuration, display and touch hardware startup, and LVGL graphical interface initialization and UI creation.

The program first initializes serial communication via "Serial.begin(115200)" for outputting debug information. It then configures the ESP32-P4 chip's internal LDO power module: first defining the error status variable "esp_err_t err", then configuring LDO3 (channel 3) to output 2.5V voltage. This voltage is used to power the MIPI D-PHY display interface, enabling this power channel via "esp_ldo_acquire_channel()", while printing initialization success or failure information to the serial port. Next, it configures LDO4 (channel 4) to output 3.3V voltage, providing power for the I2C bus and touch screen pull-up resistors.

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

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

    // --- Power Configuration (LDO3 for MIPI D-PHY) ---
    // ESP32-P4's MIPI D-PHY requires specific voltage to function.
    // LDO3 is typically routed to the MIPI power rail on P4 hardware.
    esp_err_t err = ESP_OK;
    esp_ldo_channel_handle_t ldo3_handle = NULL;
    esp_ldo_channel_config_t ldo3_cfg = {
        .chan_id = 3,           // LDO Channel 3
        .voltage_mv = 2500,     // Set to 2500mV (2.5V)
    };

    Serial.println("Initializing LDO3 to 2.5V...");
    err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
    if (err != ESP_OK) {
        Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
    } else {
        Serial.println("LDO3 Power enabled successfully.");
    }

    // --- Power Configuration (LDO4 for I2C/touch pull up) ---
    esp_ldo_channel_handle_t ldo4_handle = NULL;
    esp_ldo_channel_config_t ldo4_cfg = {
        .chan_id = 4,           // LDO Channel 4
        .voltage_mv = 3300,     // Set to 3300mV (3.3V)
    };

    Serial.println("Initializing LDO4 to 3.3V...");
    err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
    if (err != ESP_OK) {
        Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
    } else {
        Serial.println("LDO4 Power enabled successfully.");
    }
```

After power initialization is complete, the program creates a Board class object "board". This object is used to uniformly manage the display and touch devices on the development board, and calls "board->init()" to initialize the MIPI-DSI display bus, EK79007 LCD driver, and GT911 touch chip.

```
168     // --- Initialize Display and Touch Panel ---
169     Board *board = new Board();
170     // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
171     Serial.println("Initializing Panel (EK79007 + GT911)...");
172     assert(board->init());
173
```

If LVGL_PORT_AVOID_TEARING_MODE (screen tearing prevention mode) is enabled, the program obtains the LCD object and sets the number of frame buffers via "configFrameBufferNumber()" to reduce screen tearing issues during refresh.

```
174 #if LVGL_PORT_AVOID_TEARING_MODE
175     auto lcd = board->getLCD();
176     // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
177     lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
178 #endif
```

Then "board->begin()" is called to start the display, including executing the display startup sequence and turning on the backlight, and outputting system online information to the serial port.

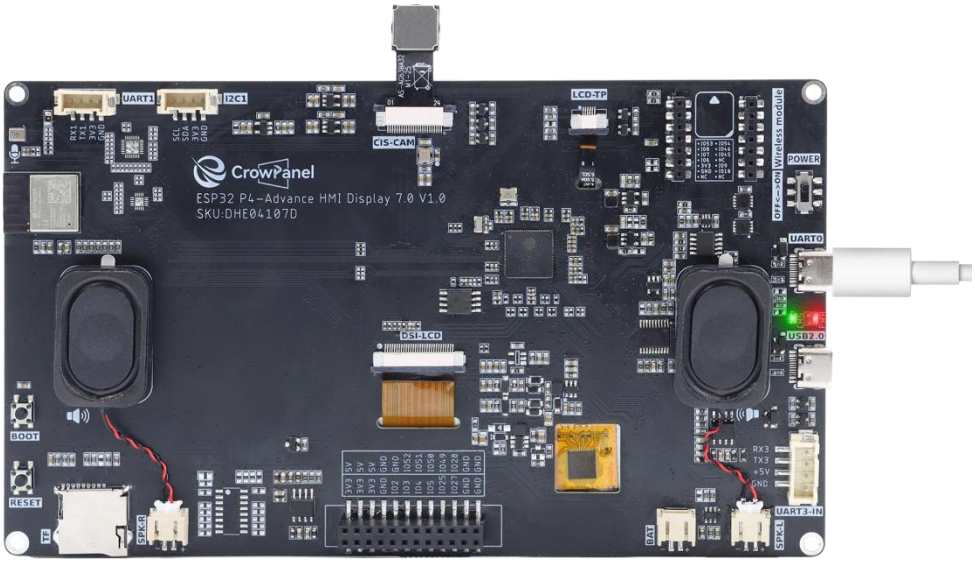
Next, the program initializes the LVGL graphics interface library. Through "lvgl_port_init(board->getLCD(), board->getTouch())", LVGL is bound to the underlying LCD display driver and touch driver, enabling LVGL to draw interfaces and receive touch input.

```
180     assert(board->begin());
181     Serial.println("Display and Touch system online.");
182
183     Serial.println("Initializing LVGL");
184     lvgl_port_init(board->getLCD(), board->getTouch());
185
186     Serial.println("Creating UI");
187
188     lvgl_show_hello_elecrow();
189 }
```

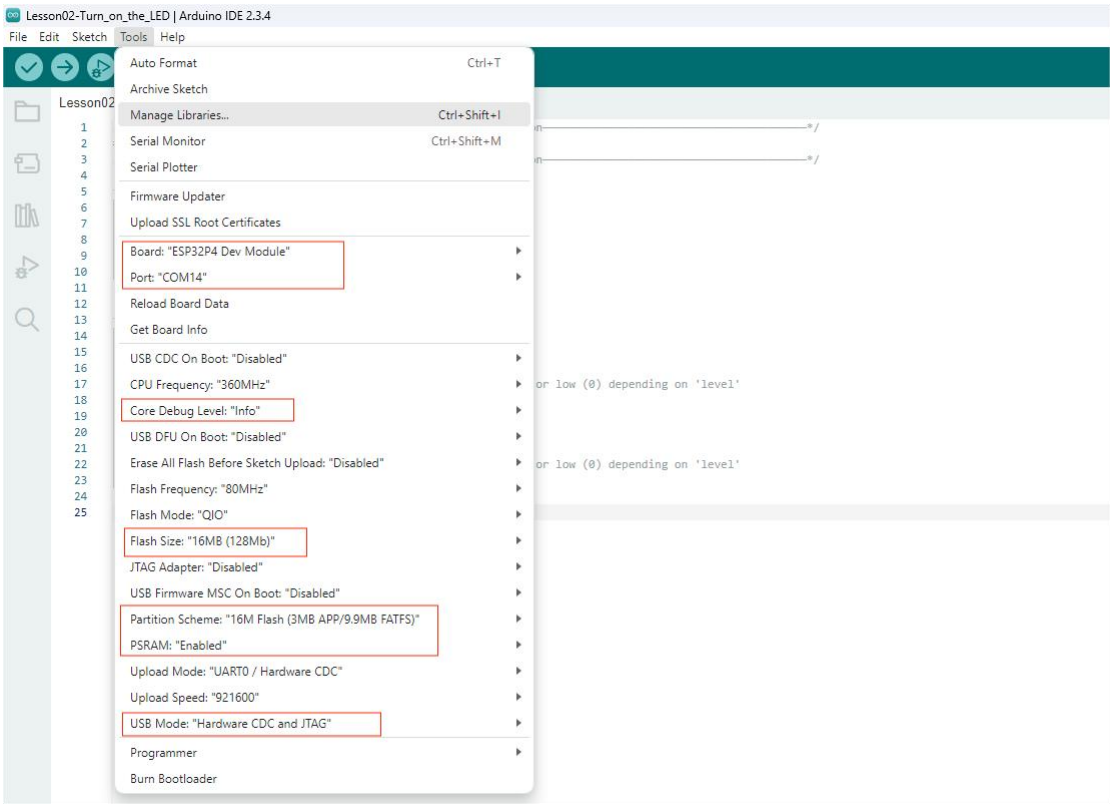
Finally, "lvgl_show_hello_elecrow()" is called to create a simple UI interface, displaying the "Hello Elecrow" text on the screen, thereby completing the initialization of the entire display system and graphical interface.

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.



Then we compile and upload the code.

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.

Lesson08---SD Card File Reading

Introduction

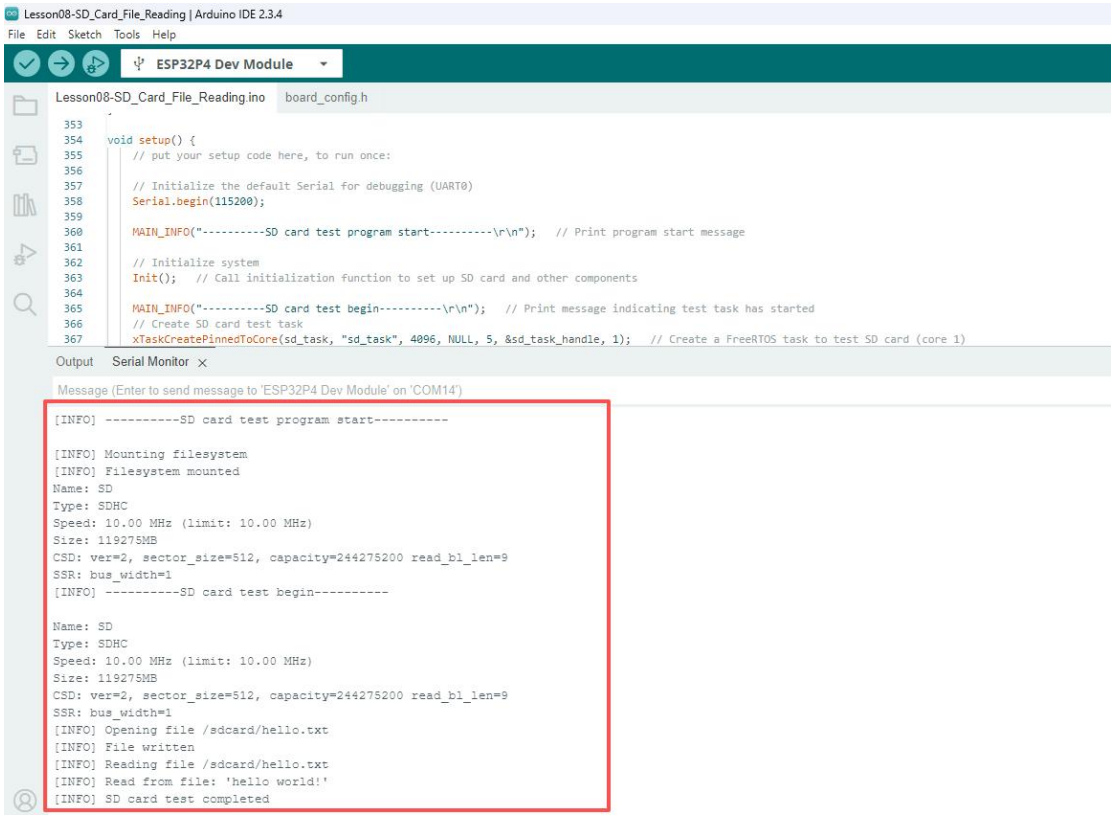
In this lesson, we will start teaching you how to use the SD card on the Advance-P4 development board to perform read and write operations on files stored in the SD card.

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.

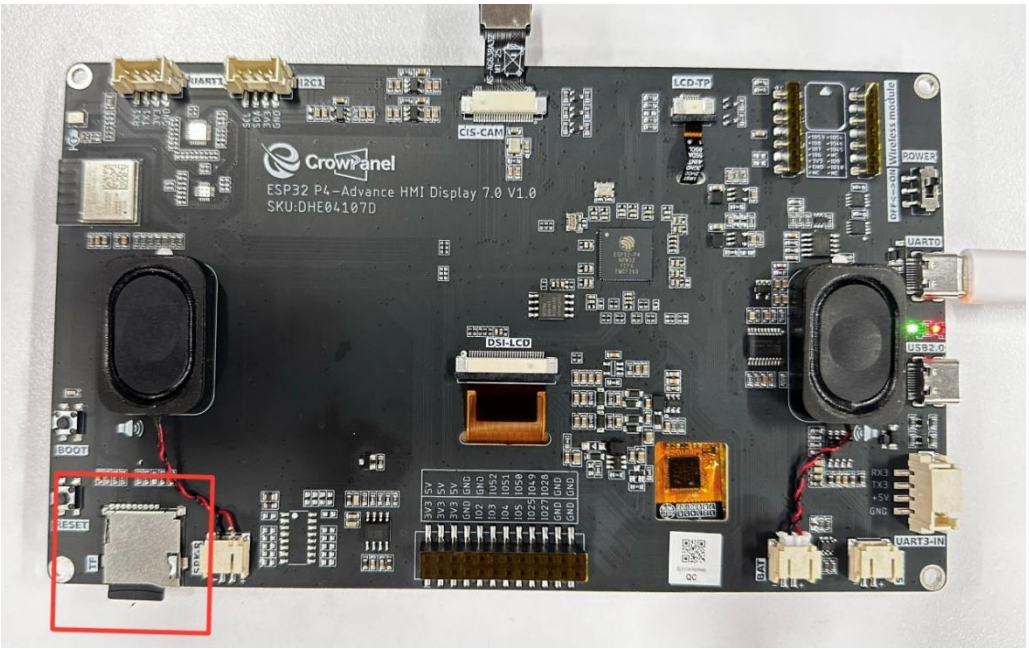
Preview of the Result

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.



Hardware Used in This Lesson

SD card on the Advance-P4



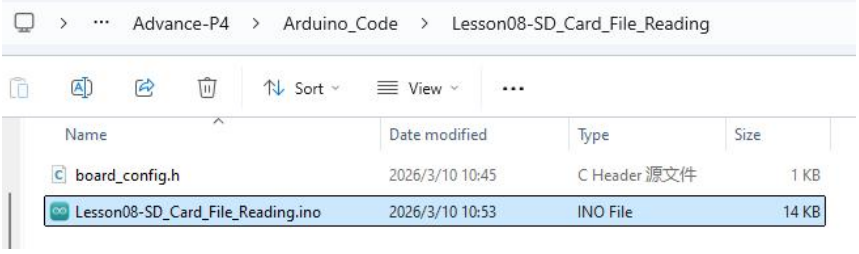
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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).



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.

Lesson08-SD_Card_File_Reading.ino | board_config.h

```
1   #pragma once
2
3   /********************** Pin define **********************/
4   // SD card GPIO with SD_MMC
5   #define SD_GPIO_MMC_CLK     (43)
6   #define SD_GPIO_MMC_CMD     (44)
7   #define SD_GPIO_MMC_D0      (39)
8
9   // SD card GPIO with SPI
10  #define SD_GPIO_SPI_CLK     SD_GPIO_MMC_CLK
11  #define SD_GPIO_SPI_MOSI    SD_GPIO_MMC_CMD
12  #define SD_GPIO_SPI_MISO    SD_GPIO_MMC_D0
13  /********************** Pin define **********************/
14
```

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.

```
4   #include "board_config.h"   // board pin define
5   #include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs
6
7   #include <string.h>         // Include standard string manipulation functions
8   #include <esp_log.h>        // ESP-IDF logging library
9   #include <esp_err.h>        // ESP-IDF error codes
10  #include <sys/unistd.h>         // Include system calls for file handling
11  #include <sys/stat.h>           // Include functions for file status and permissions
12  #include <esp_vfs_fat.h>        // Include ESP-IDF FAT filesystem support for SD card
13  #include <sdmmc_cmd.h>          // Include SDMMC card command definitions and helpers
14  #include <driver/sdmmc_host.h>  // Include SDMMC host driver for SD card communication
15  /*
```

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.

```
49  #define EXAMPLE_MAX_CHAR_SIZE 64   // Maximum character buffer size for file read/write operations
50  #define SD_MOUNT_POINT "/sdcard"   // Default SD card mount point path
51  /* ──────────────────────────────────────────────────────────────
52  │ │ │ │ │ │ │ │ │ │ │ │ │ │ │ │ GLOBAL VARIABLES
53  ──────────────────────────────────────────────────────────────── */
54  static sdmmc_card_t *card;
55  const char sd_mount_point[] = SD_MOUNT_POINT;
56
57  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.

```
66  esp_err_t create_file(const char *filename)
67  {
68      SD_INFO("Creating file %s", filename);
69      FILE *file = fopen(filename, "wb");
70      if (!file)
71      {
72          SD_ERROR("Failed to create file");
73          return ESP_FAIL;
74      }
75      fclose(file);
76      SD_INFO("File created");
77      return ESP_OK;
78  }
```

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.

```
80  esp_err_t write_string_file(const char *filename, char *data)
81  {
82      SD_INFO("Opening file %s", filename);
83      FILE *file = fopen(filename, "w");
84      if (!file)
85      {
86          SD_ERROR("Failed to open file for writing string");
87          return ESP_FAIL;
88      }
89      fprintf(file, "%s", data);
90      fclose(file);
91      SD_INFO("File written");
92      return ESP_OK;
93  }
```

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".

```
95  esp_err_t read_string_file(const char *filename)
96  {
97      SD_INFO("Reading file %s", filename);
98      FILE *file = fopen(filename, "r");
99      if (!file)
100     {
101         SD_ERROR("Failed to open file for reading string");
102
103         return ESP_FAIL;
104     }
105     char line[EXAMPLE_MAX_CHAR_SIZE];
106     fgets(line, sizeof(line), file);
107     fclose(file);
108
109     char *pos = strchr(line, '\n');
110     if (pos)
111     {
112         *pos = '\0';
113         SD_INFO("Read a line from file: '%s'", line);
114     }
115     else
116         SD_INFO("Read from file: '%s'", line);
117     return ESP_OK;
118 }
```

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

```
95  esp_err_t read_string_file(const char *filename)
96  {
97      SD_INFO("Reading file %s", filename);
98      FILE *file = fopen(filename, "r");
99      if (!file)
100     {
101         SD_ERROR("Failed to open file for reading string");
102
103         return ESP_FAIL;
104     }
105     char line[EXAMPLE_MAX_CHAR_SIZE];
106     fgets(line, sizeof(line), file);
107     fclose(file);
```

Lesson08-SD_Card_File_Reading.ino | board_config.h

```
46  #define SD_INFO(fmt, ...)           PRINTF_INFO(fmt, ##__VA_ARGS__)
47  #define SD_ERROR(fmt, ...)          PRINTF_ERROR(fmt, ##__VA_ARGS__)
48
49  #define EXAMPLE_MAX_CHAR_SIZE 64    // Maximum character buffer size for file read/write operations
50  #define SD_MOUNT_POINT "/sdcard"    // Default SD card mount point path
51  /* ---------------------------------------------------------------------
52                                  GLOBAL VARIABLES
53  --------------------------------------------------------------------- */
```

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.

```
120  esp_err_t write_file(const char *filename, char *data, size_t size)
121  {
122      size_t success_size = 0;
123      FILE *file = fopen(filename, "wb");
124      if (!file)
125      {
126          SD_ERROR("Failed to open file for writing");
127          return ESP_FAIL;
128      }
129      success_size = fwrite(data, 1, size, file);
130      if (success_size != size)
131      {
132          fclose(file);
133          SD_ERROR("Failed to write file");
134          return ESP_FAIL;
135      }
136      else
137      {
138          fclose(file);
139          SD_INFO("File written");
140      }
141      return ESP_OK;
142  }
```

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.

```
144  esp_err_t write_file_seek(const char *filename, void *data, size_t size, int32_t seek)
145  {
146      size_t success_size = 0;
147      FILE *file = fopen(filename, "wb");
148      if (!file)
149      {
150          SD_ERROR("Failed to open file for writing");
151          return ESP_FAIL;
152      }
153      if (fseek(file, seek, SEEK_SET) != 0)
154      {
155          SD_ERROR("Failed to seek file");
156          return ESP_FAIL;
157      }
158      success_size = fwrite(data, 1, size, file);
159      if (success_size != size)
160      {
161          fclose(file);
162          SD_ERROR("Failed to write file");
163          return ESP_FAIL;
164      }
165      else
166      {
167          fclose(file);
168          SD_INFO("File written");
169      }
170      return ESP_OK;
171  }
```

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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
305  void sd_task(void *param)   // SD card test task function
306  {
307      esp_err_t err = ESP_OK;   // Variable to store function return values (error codes)
308
309      const char *file_hello = SD_MOUNT_POINT "/hello.txt";   // File path for SD card test file
310      char *data = "hello world!";   // Data to be written into the file
311
312      // Get SD card information
313      get_sd_card_info();   // Print SD card info such as size, type, and speed
314
315      while (1)   // Infinite loop to perform read/write test
316      {
317          // Write data to file
318          err = write_string_file(file_hello, data);   // Write the "hello world!" string to the file
319          if (err != ESP_OK)   // Check if writing failed
320          {
321              MAIN_ERROR("Write file failed");   // Print error message if writing fails
322              continue;   // Continue to next iteration of loop
323          }
324
325          vTaskDelay(200 / portTICK_PERIOD_MS);   // Delay 200ms to allow SD card to complete internal operations
326
327          // Read data from file
328          err = read_string_file(file_hello);   // Read the content from the written file
329          if (err != ESP_OK)   // Check if reading failed
330          {
331              MAIN_ERROR("Read file failed");   // Print error message if reading fails
332          }
333
334          vTaskDelay(1000 / portTICK_PERIOD_MS);   // Delay 1 second before repeating the test
335          MAIN_INFO("SD card test completed");   // Log message indicating test finished successfully
336          vTaskDelete(NULL);   // Delete this task after finishing the test
337      }
338  }
```

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.

```
305  void sd_task(void *param)   // SD card test task function
306  {
307      esp_err_t err = ESP_OK;   // Variable to store function return values (error codes)
308
309      const char *file_hello = SD_MOUNT_POINT "/hello.txt";   // File path for SD card test file
310      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.

```
312      // Get SD card information
313      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;

```
315     while (1)   // Infinite loop to perform read/write test
316     {
317         // Write data to file
318         err = write_string_file(file_hello, data);   // Write the "hello world!" string to the file
319         if (err != ESP_OK)   // Check if writing failed
320         {
321             MAIN_ERROR("Write file failed");   // Print error message if writing fails
322             continue;   // Continue to next iteration of loop
323         }
```

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.

```
325         vTaskDelay(200 / portTICK_PERIOD_MS);   // Delay 200ms to allow SD card to complete internal operations
326
327         // Read data from file
328         err = read_string_file(file_hello);   // Read the content from the written file
329         if (err != ESP_OK)   // Check if reading failed
330         {
331             MAIN_ERROR("Read file failed");   // Print error message if reading fails
332         }
```

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.

```
333
334         vTaskDelay(1000 / portTICK_PERIOD_MS);   // Delay 1 second before repeating the test
335         MAIN_INFO("SD card test completed");   // Log message indicating test finished successfully
336         vTaskDelete(NULL);   // Delete this task after finishing the test
337     }
338 }
339
```

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.

```
340 void Init(void)   // System initialization function
341 {
342     esp_err_t err = ESP_OK;   // Variable to store error codes
343
344     // Initialize SD card
345     err = sd_init();   // Call SD card initialization function
346     // Check if initialization failed
347     while (ESP_OK != err) {
348         MAIN_ERROR("%s initialization failed [ %s ]", "SD card", esp_err_to_name(err));   // Print module name and error description
349         delay (1000);
350         err = sd_init();   // Call SD card initialization function
351     }
352 }
```

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.

```
354  void setup() {
355      // put your setup code here, to run once:
356
357      // Initialize the default Serial for debugging (UART0)
358      Serial.begin(115200);
359
360      MAIN_INFO("----------SD card test program start----------\r\n");   // Print program start message
361
362      // Initialize system
363      Init();   // Call initialization function to set up SD card and other components
364
365      MAIN_INFO("----------SD card test begin----------\r\n");   // Print message indicating test task has started
366      // Create SD card test task
367      xTaskCreatePinnedToCore(sd_task, "sd_task", 4096, NULL, 5, &sd_task_handle, 1);   // Create a FreeRTOS task to test SD card (core 1)
368  }
```

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.

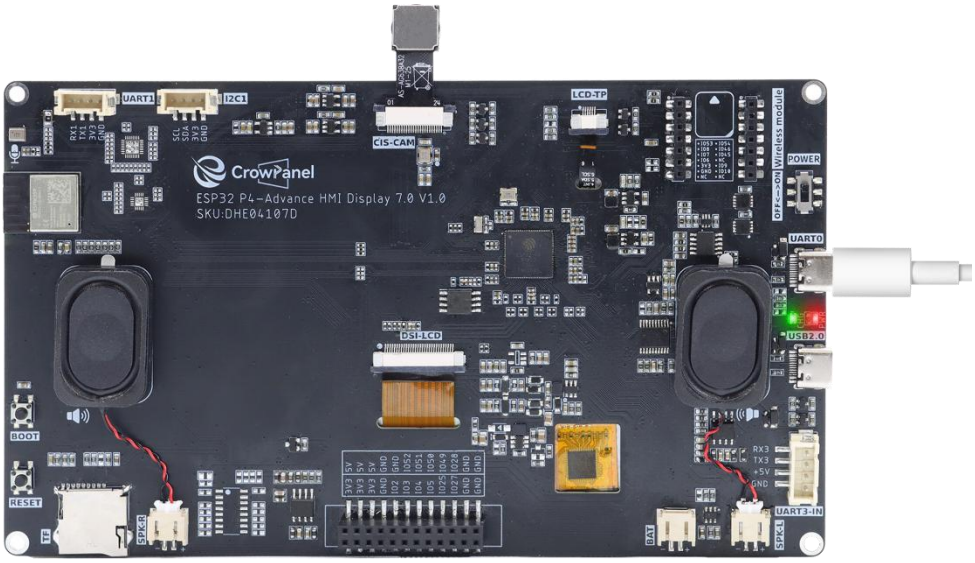
After the initialization is completed, the program prints the log "SD card test begin" again to prompt that the SD card test is about to start, and then calls "xTaskCreatePinnedToCore()" to create a FreeRTOS task "sd_task" for executing the SD card read and write tests.

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.

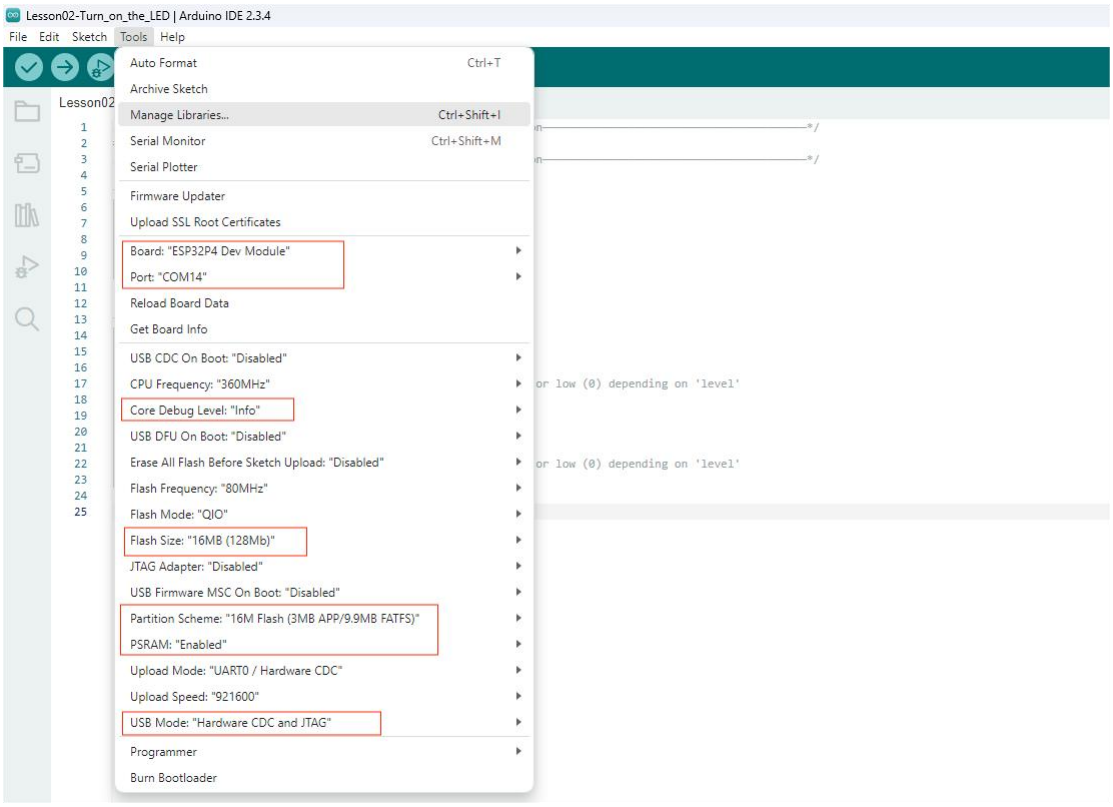
In this way, the read and write tests of the SD card will run in an independent FreeRTOS task without blocking the main program, thereby achieving a more stable and efficient system operation.

Programming Steps

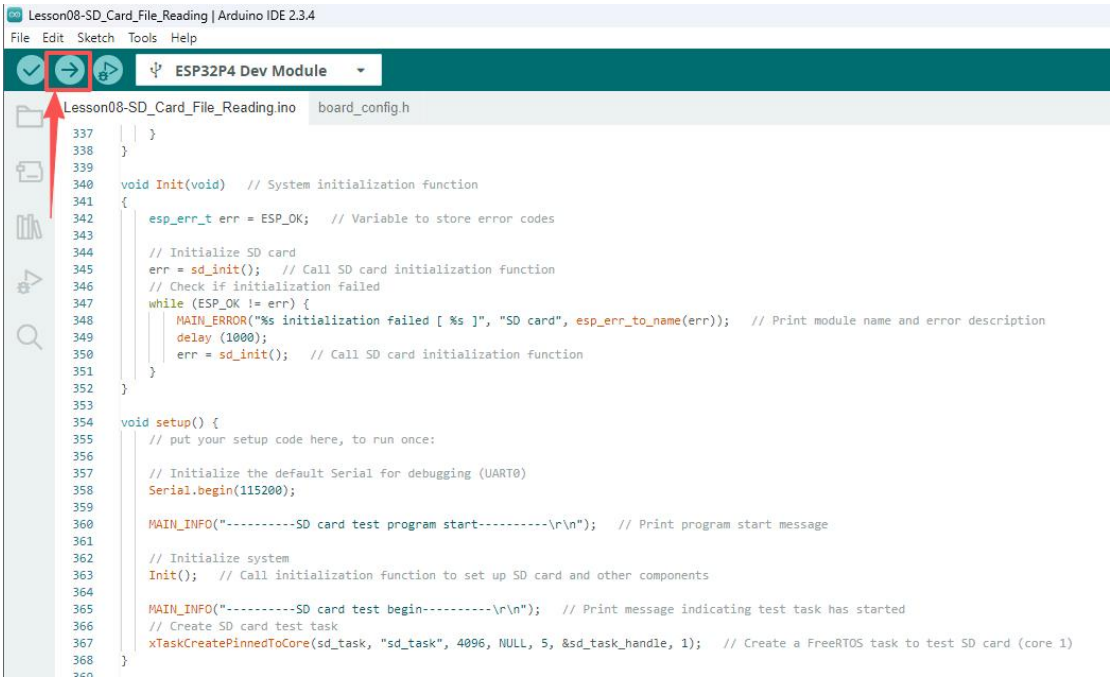
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



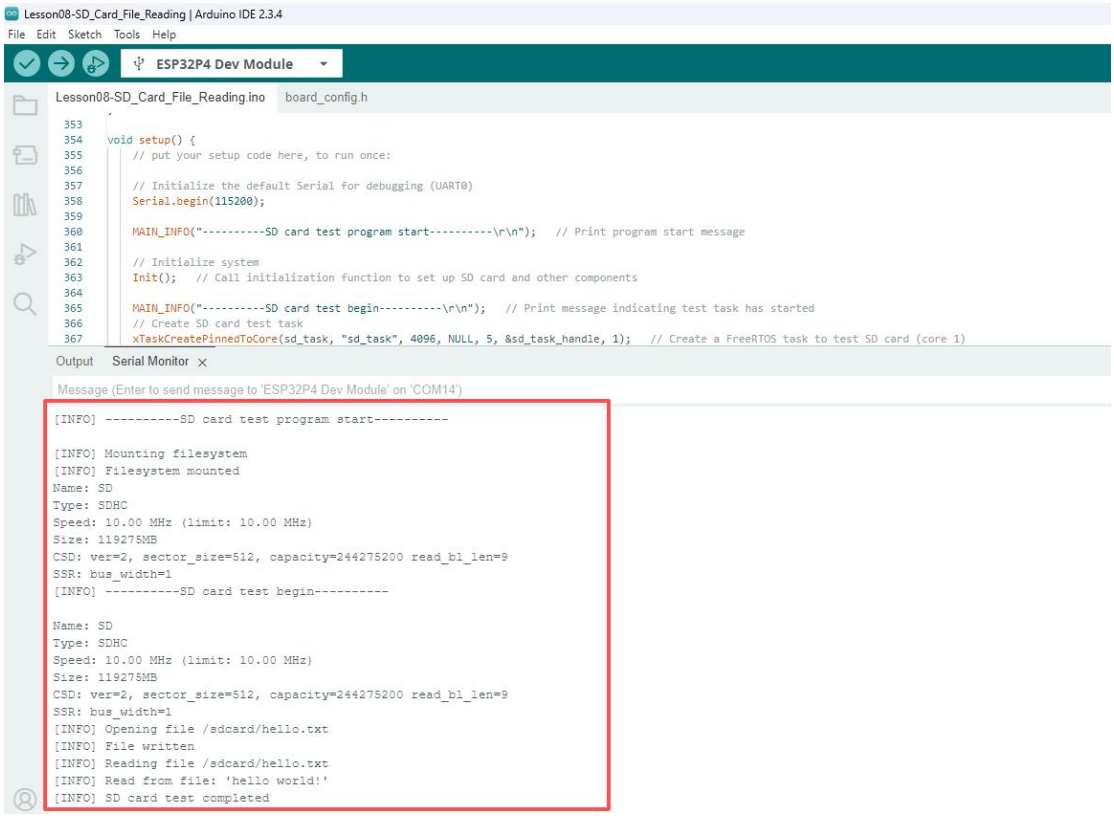
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



Then we compile and upload the code.



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
353
354 void setup() {
355 // put your setup code here, to run once:
356
357 // Initialize the default Serial for debugging (UART0)
358 Serial.begin(115200);
359
360 MAIN_INFO("----------SD card test program start----------\r\n"); // Print program start message
361
362 // Initialize system
363 Init(); // Call initialization function to set up SD card and other components
364
365 MAIN_INFO("----------SD card test begin----------\r\n"); // Print message indicating test task has started
366 // Create SD card test task
367 xTaskCreatePinnedToCore(sd_task, "sd_task", 4096, NULL, 5, &sd_task_handle, 1); // Create a FreeRTOS task to test SD card (core 1)
Output
Serial Monitor
Message (Enter to send message to 'ESP32P4 Dev Module' on 'COM14')
[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

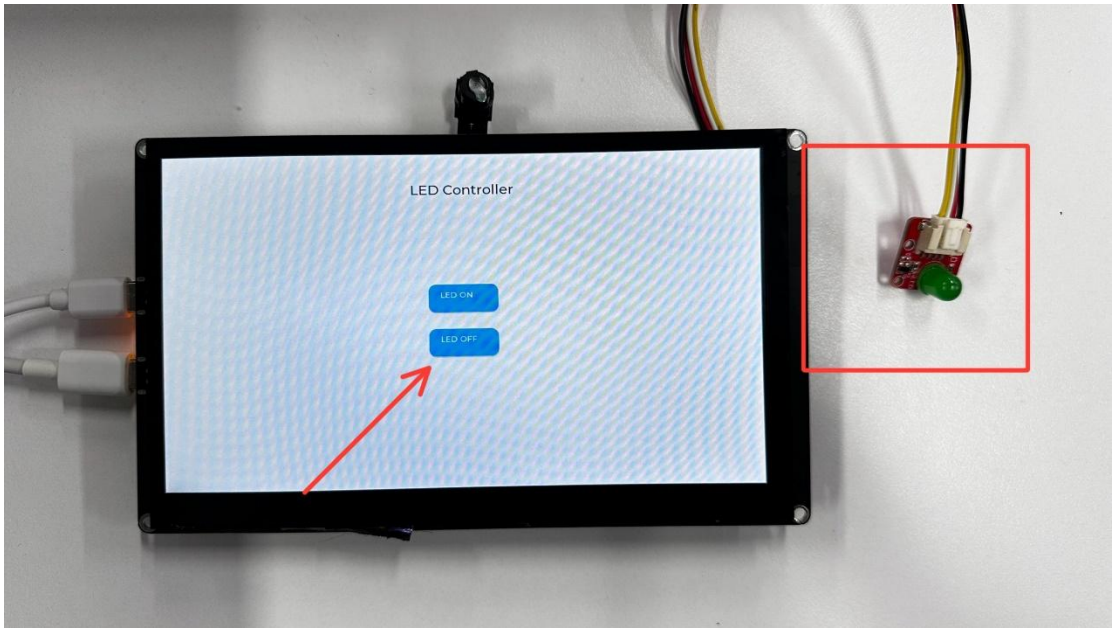
In previous courses, we separately lit an LED, implemented touch testing, and lit up the screen.In this lesson, we will use LVGL to create two buttons to control the LED connected to the UART1 interface for turning on and off operations.
Pressing the ON button can turn on the LED, and pressing the OFF 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.

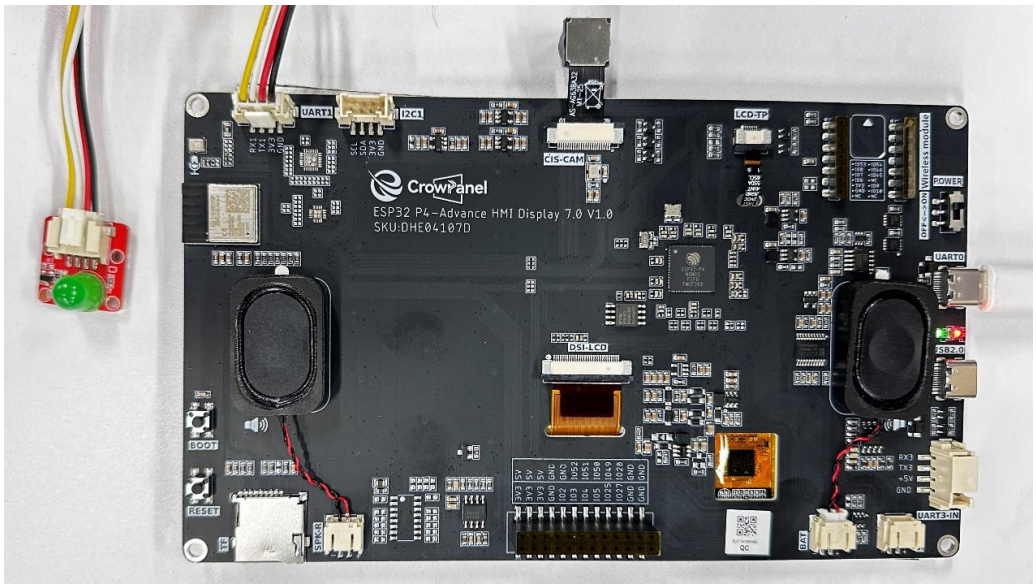
Preview of the Result

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.



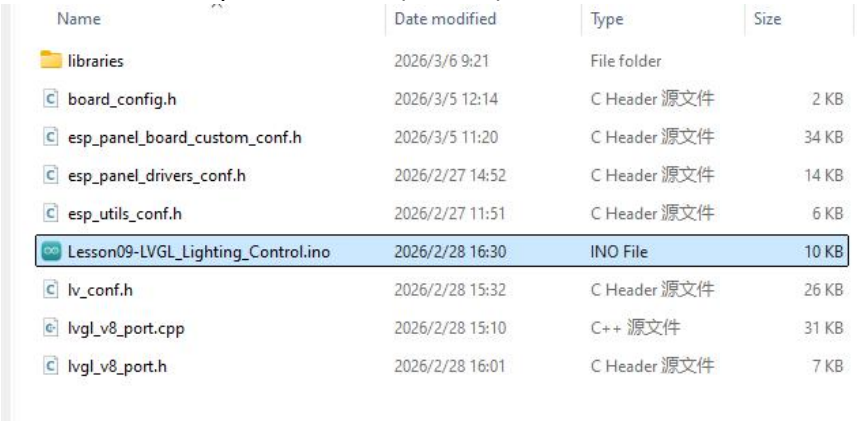
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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.



Name	Date modified	Type	Size
libraries	2026/3/6 9:21	File folder	
board_config.h	2026/3/5 12:14	C Header 源文件	2 KB
esp_panel_board_custom_conf.h	2026/3/5 11:20	C Header 源文件	34 KB
esp_panel_drivers_conf.h	2026/2/27 14:52	C Header 源文件	14 KB
esp_utils_conf.h	2026/2/27 11:51	C Header 源文件	6 KB
Lesson09-LVGL_Lighting_Control.ino	2026/2/28 16:30	INO File	10 KB
lv_conf.h	2026/2/28 15:32	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/28 15:10	C++ 源文件	31 KB
lvgl_v8_port.h	2026/2/28 16:01	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.

First, let's take a look at the content in board_config.h.

```
#pragma once

/********************** Pin define **********************/
// GPIO pins for LED moudule
#define PIN_LED             (48)    // 1:Turn on LED, 0:Turn off LED
#define LED_ON              ( 1)
#define LED_OFF             ( 0)

// 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 **********************/
```

This code is a hardware configuration header file, which is mainly used to centrally define the GPIO pins used by various peripherals on the development board and the display-related parameters, so as to facilitate unified calling and management throughout the program. At the beginning of the file, the "#pragma once" directive is used to prevent the header file from being included repeatedly, thereby avoiding the problem of duplicate definitions during compilation.

Subsequently, the code declares the pins and parameters of different hardware modules through a series of "#define" macro definitions: first, it defines the GPIO pin of the LED module, where "PIN_LED" is GPIO48, and specifies the on and off levels of the LED through the "LED_ON" and "LED_OFF" macros;

Then it defines the pins of the GT911 touch screen controller, including "Touch_GPIO_RST" (GPIO40, used for touch chip reset) and "Touch_GPIO_INT" (GPIO42, used for touch interrupt signal);

Next, it defines the I2C bus pins: "I2C_GPIO_SCL" is GPIO46 (clock line), and "I2C_GPIO_SDA" is GPIO45 (data line). This I2C bus is used to communicate with the GT911 touch control chip.

After that, the code defines the parameters related to LCD backlight control, among which "LCD_GPIO_BLIGHT" is the backlight control pin (GPIO31), "BLIGHT_PWM_Hz" sets the PWM dimming frequency to 30kHz, and "BLIGHT_ON_LEVEL" indicates that the effective level of the backlight is high level.

Then it defines the LCD display reset pin "LCD_GPIO_RST" (GPIO41), which is used to perform hardware reset on the display when the system starts up.

The last part is the timing and resolution parameters of the display panel, including the screen resolution "H_size=1024" (horizontal pixels) and "V_size=600" (vertical pixels), as well as the timing parameters required by the MIPI/LCD driver, such as pixel clock "LCD_CLK_MHZ", horizontal sync pulse width "LCD_HPW", horizontal front/back porch (HFP/HBP), and vertical sync pulse width and front/back porch (VPW/VBP/VFP). These parameters determine how the display controller correctly drives the screen refresh.

(The meaning and function of each file here have been explained in detail in Lesson 7, so we will not elaborate on them here. If you have any questions, you can review the content of the previous lessons.)

Next, we will focus on learning the main ".ino" code file to see how to realize LED lighting control through the LVGL interface.

(The specific explanations of header files, namespaces, macro definitions and global

variables have been detailed in Lesson 7, so we will not go into too much detail here.)

```
#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
#include <esp_ldo_regulator.h>      // ESP32-P4 specific LDO management

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

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

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

```
#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.

```
100  /* Create LED control UI */
101  static void create_led_control_ui(void)
102  {
103      // Lock LVGL: ensure thread-safe operations
104      if (lvgl_port_lock(-1) != true) {  // 0 means non-blocking wait for the lock (timeout = 0)
105          MAIN_ERROR("LVGL lock failed");  // Print error if lock fails
106          return;  // Exit function
107      }
```

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.

```
108     // Create main screen
109     lv_obj_t *scr = lv_scr_act();
110     lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFFFF), LV_PART_MAIN);  // Set white background
111
112     // Create title label
113     lv_obj_t *label = lv_label_create(scr);
114     lv_label_set_text(label, "LED Controller");
115     lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 50);
116     // Font size
117     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.

```
119     // Create LED ON button
120     lv_obj_t *btn_on = lv_btn_create(scr);
121     lv_obj_set_size(btn_on, 120, 50);
122     lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
123     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".

```
125     // ON button label
126     lv_obj_t *label_on = lv_label_create(btn_on);
127     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.

```
129     // Create LED OFF button
130     lv_obj_t *btn_off = lv_btn_create(scr);
131     lv_obj_set_size(btn_off, 120, 50);
132     lv_obj_align(btn_off, LV_ALIGN_CENTER, 0, 40);
133     lv_obj_add_event_cb(btn_off, btn_off_click_event, LV_EVENT_CLICKED, NULL);
134
135     // OFF button label
136     lv_obj_t *label_off = lv_label_create(btn_off);
137     lv_label_set_text(label_off, "LED OFF");
138
139     lvgl_port_unlock();
140 }
```

3. Setup Section

This setup() function serves as the system initialization entry point for the entire program, and is executed only once after the device is powered on or reset. Its main tasks are to sequentially complete serial port debugging initialization, power supply

(LDO) configuration, display and touch screen driver initialization, LVGL graphics library startup, LED control GPIO initialization, and finally create the graphical user interface (UI).

The program first starts the default debugging serial port (UART0) via Serial.begin(115200) and sets the baud rate to 115200, so as to output running information and debugging logs in the serial monitor.

Subsequently, it starts configuring the power supply of the ESP32-P4: first, it configures the LDO3 voltage regulation channel, setting chan_id = 3 and voltage_mv = 2500 through the esp_ldo_channel_config_t structure, which means setting the output voltage of LDO3 to 2.5V — this voltage is typically used to power the MIPI D-PHY display interface.

Then it calls esp_ldo_acquire_channel() to apply for the LDO channel and enable the power supply; if it fails, an error message is printed, otherwise a prompt indicating successful power supply startup is output.

Next, the program configures the LDO4 voltage regulation channel in the same way, setting the voltage to 3.3V, which is usually used to power the I2C bus and touch screen pull-up resistors to ensure the normal operation of the touch controller.

```
142  void setup() {
143      // put your setup code here, to run once:
144
145      // Initialize the default Serial for debugging (UART0)
146      Serial.begin(115200);
147
148      // --- Power Configuration (LDO3 for MIPI D-PHY) ---
149      // ESP32-P4's MIPI D-PHY requires specific voltage to function.
150      // LDO3 is typically routed to the MIPI power rail on P4 hardware.
151      esp_err_t err = ESP_OK;
152      esp_ldo_channel_handle_t ldo3_handle = NULL;
153      esp_ldo_channel_config_t ldo3_cfg = {
154          .chan_id = 3,           // LDO Channel 3
155          .voltage_mv = 2500,     // Set to 2500mV (2.5V)
156      };
157
158      Serial.println("Initializing LDO3 to 2.5V...");
159      err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
160      if (err != ESP_OK) {
161          Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
162      } else {
163          Serial.println("LDO3 Power enabled successfully.");
164      }
165
166      // --- Power Configuration (LDO4 for I2C/touch pull up) ---
167      esp_ldo_channel_handle_t ldo4_handle = NULL;
168      esp_ldo_channel_config_t ldo4_cfg = {
169          .chan_id = 4,           // LDO Channel 4
170          .voltage_mv = 3300,     // Set to 3300mV (3.3V)
171      };
172
173      Serial.println("Initializing LDO4 to 3.3V...");
174      err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
175      if (err != ESP_OK) {
176          Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
177      } else {
178          Serial.println("LDO4 Power enabled successfully.");
179      }
```

Upon power-on initialization completion, the program instantiates a "Board" object (designated as "board") to manage the hardware drivers for the display panel and touchscreen. It subsequently invokes "board->init()" to initialize the underlying MIPI-DSI communication bus, along with the device drivers for the EK79007 LCD controller and the GT911 touch controller.

```
181     // --- Initialize Display and Touch Panel ---
182     Board *board = new Board();
183     // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
184     Serial.println("Initializing Panel (EK79007 + GT911)...");
185     assert(board->init());
```

If the "LVGL_PORT_AVOID_TEARING_MODE" (tear-free mode) is enabled, the code additionally retrieves the LCD object and configures the frame buffer count via "configFrameBufferNumber()" to mitigate image tearing issues during screen refresh cycles.

```
186 #if LVGL_PORT_AVOID_TEARING_MODE
187     auto lcd = board->getLCD();
188     // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
189     lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
190 #endif
```

Subsequently, "board->begin()" is invoked to boot up the entire display and touch system, followed by a console output of "Display and Touch system online." to indicate that both the screen and touch functionality are fully operational.

```
191     assert(board->begin());
192     Serial.println("Display and Touch system online.");
193
```

The program then proceeds to initialize the LVGL graphics library, binding it to the LCD display driver and touch input device through "lvgl_port_init(board->getLCD(), board->getTouch())", thereby enabling LVGL to handle screen rendering and touch event processing.

```
194     Serial.println("Initializing LVGL");
195     lvgl_port_init(board->getLCD(), board->getTouch());
196
```

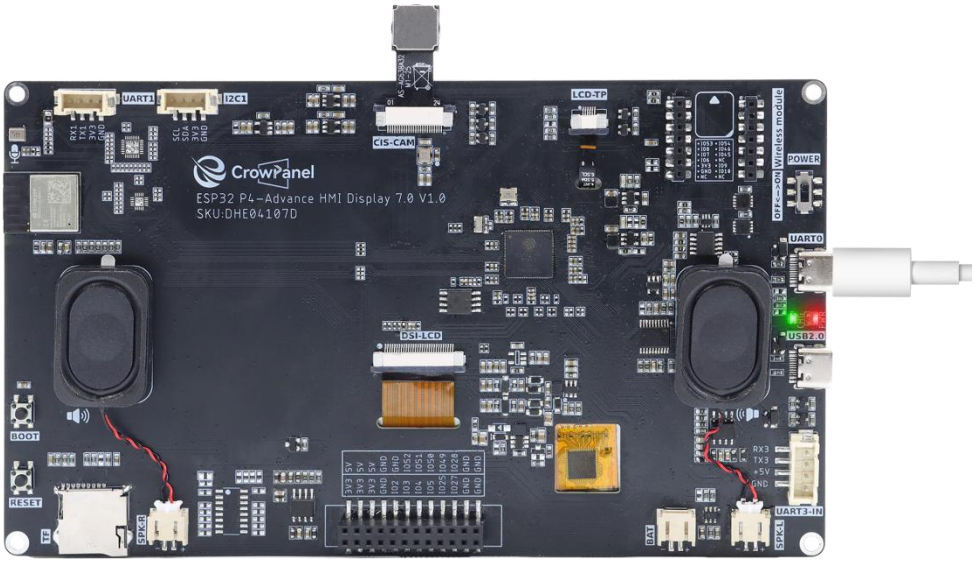
The program then initializes GPIO pin 48 for LED control: it configures the pin as output mode using "pinMode(PIN_LED, OUTPUT)", sets the LED default state to off via "digitalWrite(PIN_LED, LED_OFF)", and outputs a log message indicating the LED has been initialized.

```
197     Serial.println("Creating UI");
198
199     // Initialize LED control GPIO (GPIO48)
200     MAIN_INFO("Initializing GPIO48 for LED...");
201
202     pinMode(PIN_LED, OUTPUT);
203     digitalWrite(PIN_LED, LED_OFF);
204     MAIN_INFO("LED initialized to OFF state");
205
206     create_led_control_ui();
207 }
```

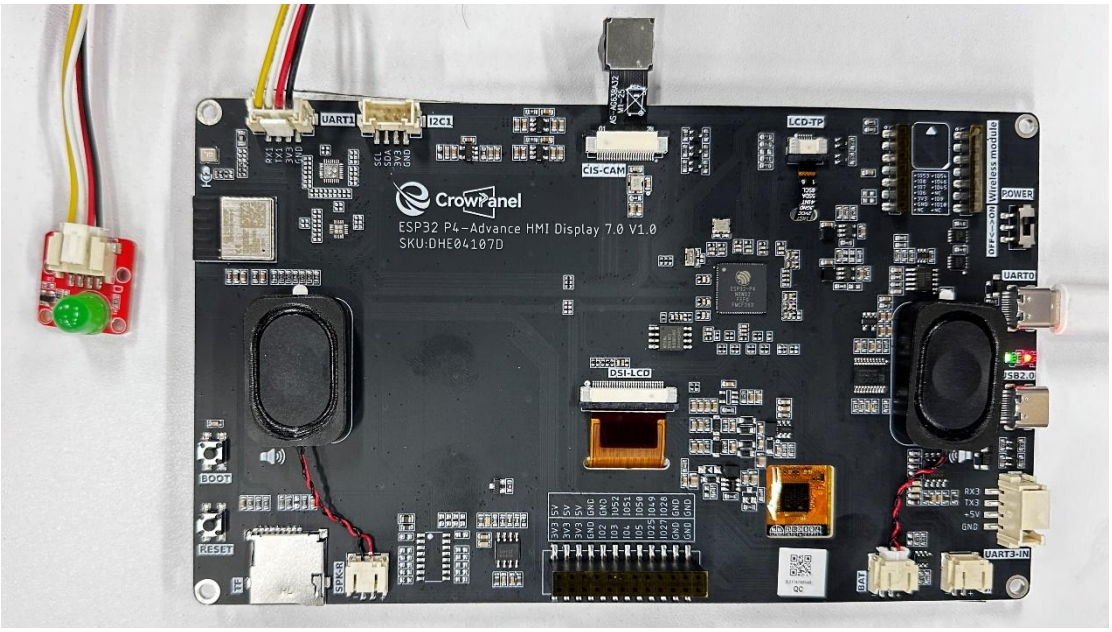
Finally, "create_led_control_ui()" is invoked to construct the graphical interface, generating an LED control panel on the screen (comprising a title and ON/OFF buttons), thereby allowing users to control the onboard LED through touchscreen interactions.

Programming Steps

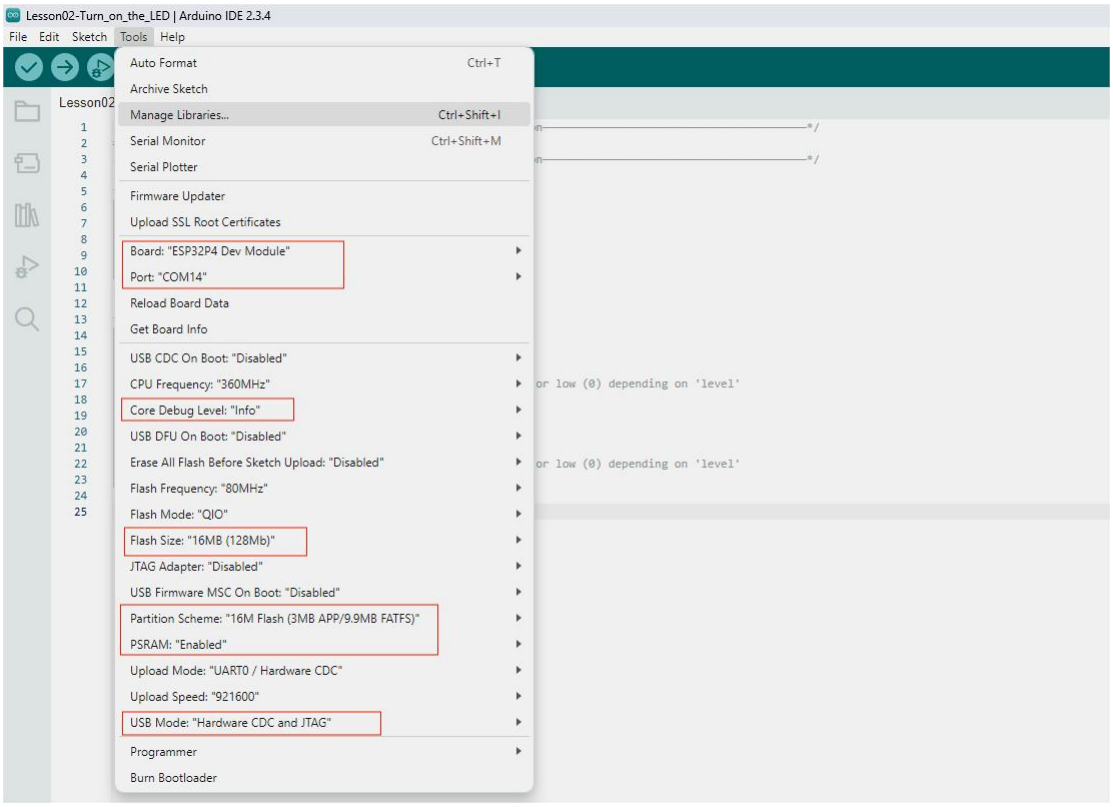
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



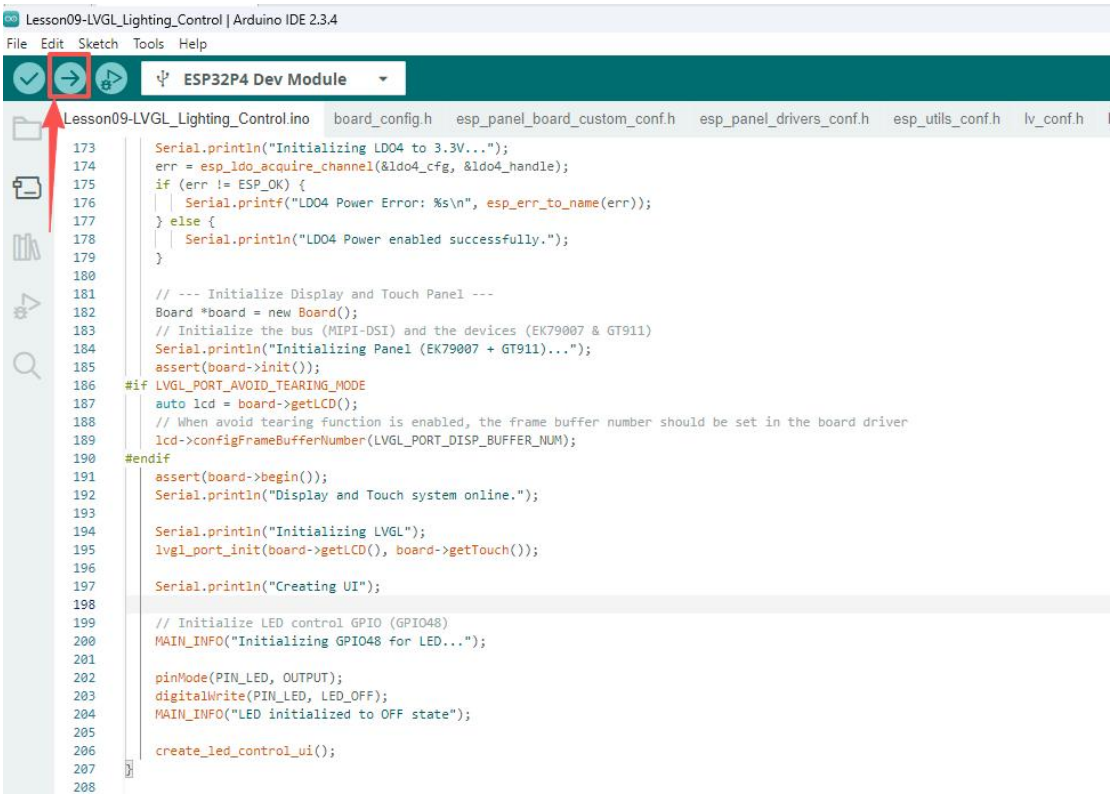
Also, remember to connect an LED to the UART1 interface.



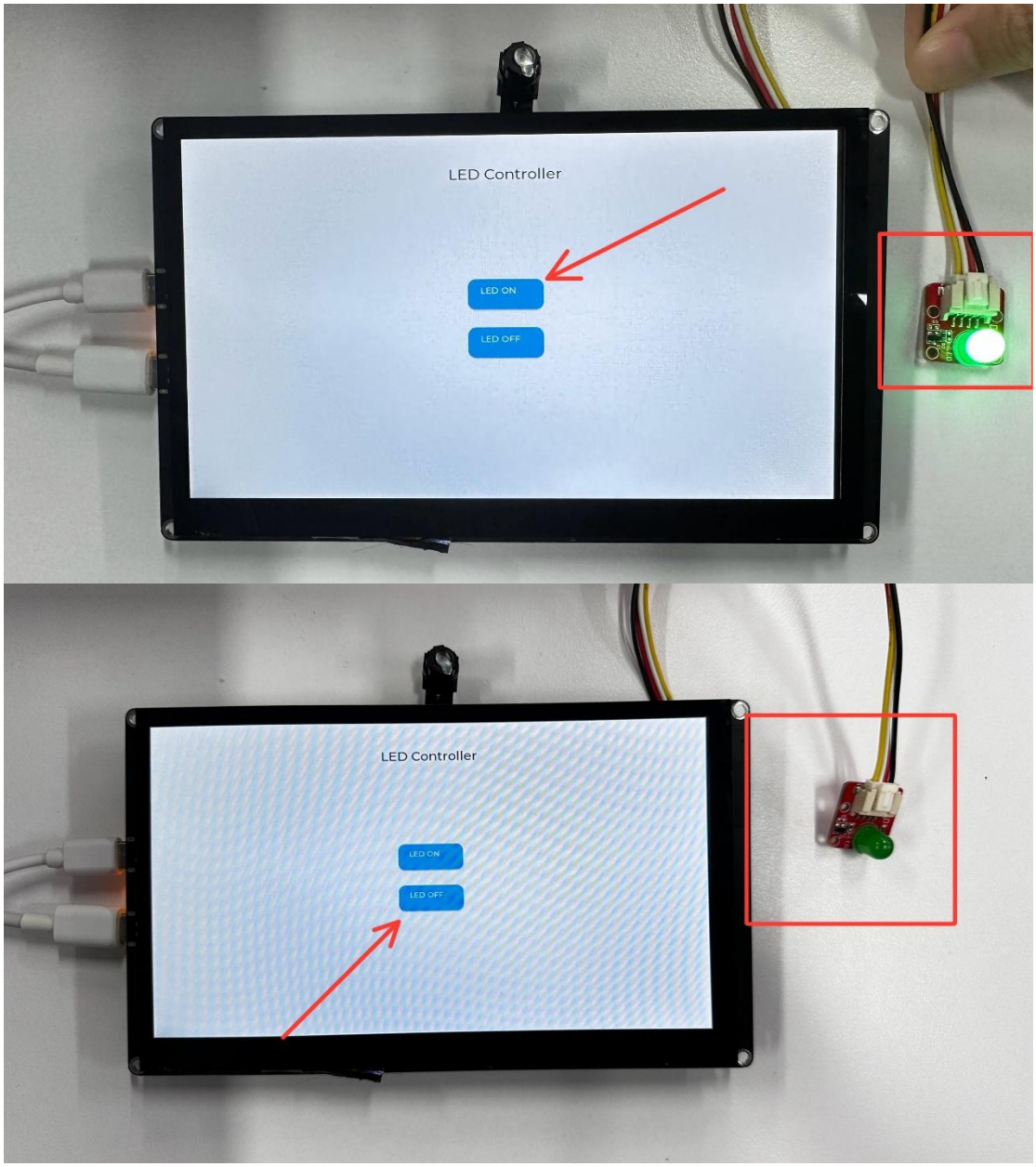
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



Then we compile and upload the code.



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.


LED Controller
LED ON
LED OFF
LED Controller
LED ON
LED OFF

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

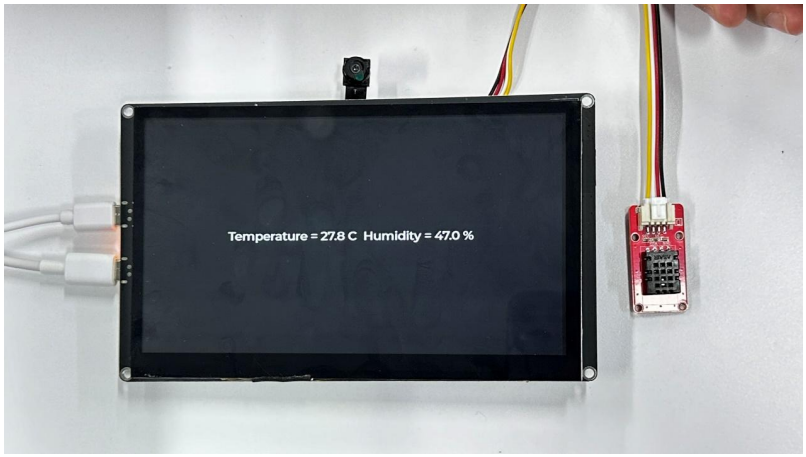
1. Understand the I2C bus communication principles of the ESP32-P4 development board and the operational mechanisms of the DHT20 temperature and humidity sensor (humidity/temperature detection principles).

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.

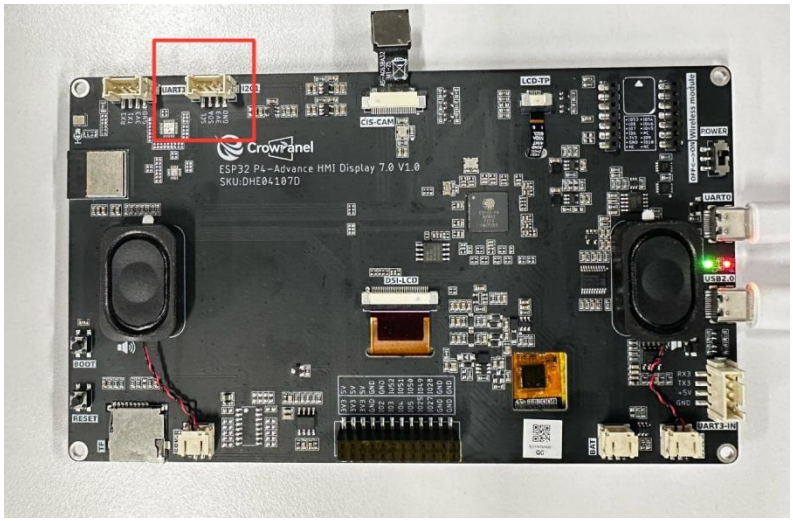
Preview of the Result

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.

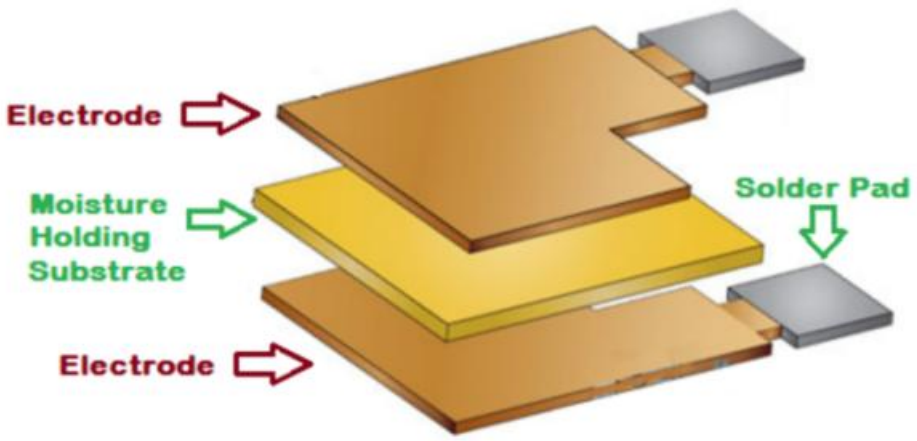


Hardware Used in This Lesson

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

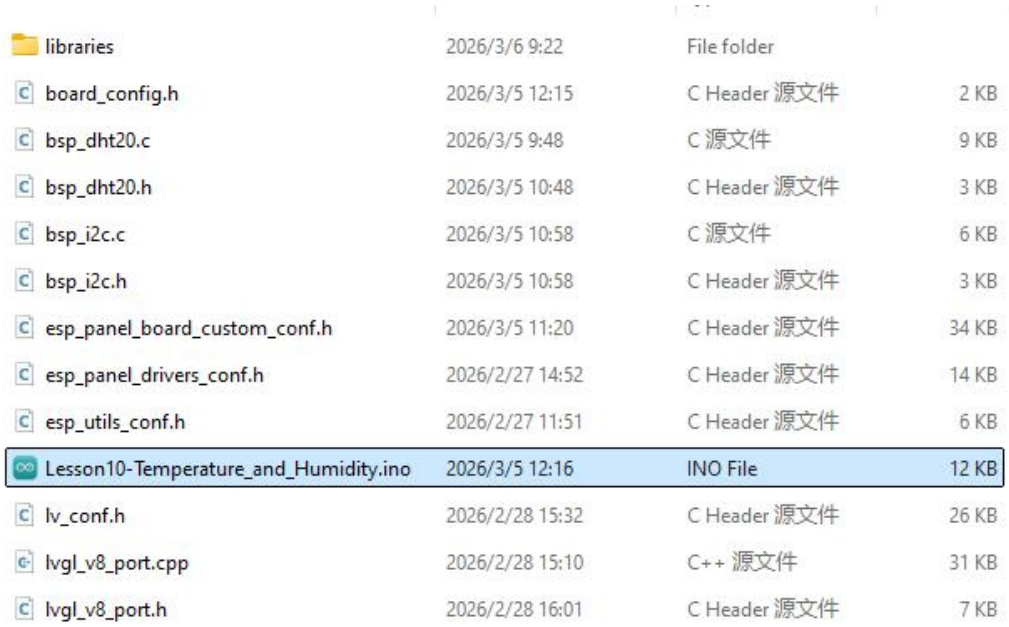
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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).



libraries	2026/3/6 9:22	File folder	
board_config.h	2026/3/5 12:15	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
esp_panel_board_custom_conf.h	2026/3/5 11:20	C Header 源文件	34 KB
esp_panel_drivers_conf.h	2026/2/27 14:52	C Header 源文件	14 KB
esp_utils_conf.h	2026/2/27 11:51	C Header 源文件	6 KB
Lesson10-Temperature_and_Humidity.ino	2026/3/5 12:16	INO File	12 KB
lv_conf.h	2026/2/28 15:32	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/28 15:10	C++ 源文件	31 KB
lvgl_v8_port.h	2026/2/28 16:01	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.

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 pure text display framework from Lesson 7, this project's framework includes additional components for "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".)

```
#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.

```
14  #include "board_config.h"   // board pin define
15  #include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs
16
17  #include <string.h>         // C string lib
18  #include <esp_log.h>        // ESP-IDF logging library
19  #include <esp_err.h>        // ESP-IDF error codes
20  #include <esp_ldo_regulator.h>      // ESP32-P4 specific LDO management
21
22  #include "esp_panel_drivers_conf.h"
23  #include "esp_panel_board_custom_conf.h"
24  #include "ESP_Panel_Library.h"
25
26  #include <lvgl.h>
27  #include "lvgl_v8_port.h"
28
29  #include "bsp_i2c.h"
30  #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.

```
108  void update_dht20_value(float temperature, float humidity)
109  {
110      if (dht20_data)
111      {
112          char buffer[60];
113          snprintf(buffer, sizeof(buffer), "Temperature = %.1f C  Humidity = %.1f %%", temperature, humidity); /*Format the data into a string*/
114          lv_label_set_text(dht20_data, buffer);                                                               /*Set a new text for a label*/
115      }
116  }
117
```

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.

```
118  void dht20_read_task(void *param)
119  {
120      static dht20_data_t measurements;
121      while (1)
122      {
123          /*The function for determining whether the DHT20 sensor is ready or not*/
124          if (dht20_is_calibrated() == ESP_OK) {
125              MAIN_INFO("is calibrated....");
126          } else {
127              MAIN_INFO("is NOT calibrated....");
128
129              /*Reinitialize the DHT20 sensor*/
130              if (dht20_begin() != ESP_OK) {
131                  MAIN_ERROR("dht20 init again false");
132                  vTaskDelay(100 / portTICK_PERIOD_MS);
133                  continue;
134              }
135          }
136
137          /*Read the temperature and humidity data from the DHT20 sensor*/
138          if (dht20_read_data(&measurements) != ESP_OK) {
139              if (lvgl_port_lock(-1)) {
140                  lv_label_set_text(dht20_data, "dht20 read data error"); /*Read failure message displayed*/
141                  lvgl_port_unlock();
142              }
143              MAIN_ERROR("dht20 read data error");
144          }
145          /*Read successfully*/
146          else {
147              if (lvgl_port_lock(-1)) {
148                  update_dht20_value(measurements.temperature, measurements.humidity); /*Update the DHT20 data displayed on the screen*/
149                  lvgl_port_unlock();
150              }
151              MAIN_INFO("Temperature:\t%.1fC", measurements.temperature);
152              MAIN_INFO("Humidity:   \t%.1f%%", measurements.humidity);
153          }
154          vTaskDelay(1000 / portTICK_PERIOD_MS);
155      }
156  }
```

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.

```
118  void dht20_read_task(void *param)
119  {
120      static dht20_data_t measurements;
121      while (1)
122      {
```

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.

```
123         /*The function for determining whether the DHT20 sensor is ready or not*/
124         if (dht20_is_calibrated() == ESP_OK) {
125             MAIN_INFO("is calibrated....");
126         } else {
127             MAIN_INFO("is NOT calibrated....");
128 
129             /*Reinitialize the DHT20 sensor*/
130             if (dht20_begin() != ESP_OK) {
131                 MAIN_ERROR("dht20 init again false");
132                 vTaskDelay(100 / portTICK_PERIOD_MS);
133                 continue;
134             }
135         }
```

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.

```
137         /*Read the temperature and humidity data from the DHT20 sensor*/
138         if (dht20_read_data(&measurements) != ESP_OK) {
139             if (lvgl_port_lock(-1)) {
140                 lv_label_set_text(dht20_data, "dht20 read data error"); /*Read failure message displayed*/
141                 lvgl_port_unlock();
142             }
143             MAIN_ERROR("dht20 read data error");
144         }
```

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.

```
145         /*Read successfully*/
146         else {
147             if (lvgl_port_lock(-1)) {
148                 update_dht20_value(measurements.temperature, measurements.humidity); /*Update the DHT20 data displayed on the screen*/
149                 lvgl_port_unlock();
150             }
```

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.

```
151                 MAIN_INFO("Temperature:\t%.1fC", measurements.temperature);
152                 MAIN_INFO("Humidity:   \t%.1f%%", measurements.humidity);
153             }
154             vTaskDelay(1000 / portTICK_PERIOD_MS);
155         }
156     }
```

6. ldo_init

This code defines a function "ldo_init()", whose primary purpose is to configure and enable two LDO (Low Dropout Regulator) power channels for critical hardware modules on the development board, thereby providing stable voltage to the display

interface and touch/I2C circuits.

```
158  void ldo_init()
159  {
160      // --- Power Configuration (LDO3 for MIPI D-PHY) ---
161      // ESP32-P4's MIPI D-PHY requires specific voltage to function.
162      // LDO3 is typically routed to the MIPI power rail on P4 hardware.
163      esp_err_t err = ESP_OK;
164      esp_ldo_channel_handle_t ldo3_handle = NULL;
165      esp_ldo_channel_config_t ldo3_cfg = {
166          .chan_id = 3,           // LDO Channel 3
167          .voltage_mv = 2500,     // Set to 2500mV (2.5V)
168      };
169
170      Serial.println("Initializing LDO3 to 2.5V...");
171      err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
172      if (err != ESP_OK) {
173          Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
174      } else {
175          Serial.println("LDO3 Power enabled successfully.");
176      }
177
178      // --- Power Configuration (LDO4 for I2C/touch pull up) ---
179      esp_ldo_channel_handle_t ldo4_handle = NULL;
180      esp_ldo_channel_config_t ldo4_cfg = {
181          .chan_id = 4,           // LDO Channel 4
182          .voltage_mv = 3300,     // Set to 3300mV (3.3V)
183      };
184
185      Serial.println("Initializing LDO4 to 3.3V...");
186      err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
187      if (err != ESP_OK) {
188          Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
189      } else {
190          Serial.println("LDO4 Power enabled successfully.");
191      }
192  }
```

The function begins by declaring an error status variable "esp_err_t err" and LDO handle variables. It then proceeds to configure LDO3: through the "esp_ldo_channel_config_t ldo3_cfg" structure, it specifies "chan_id = 3" and sets the output voltage to 2500mV (2.5V)—the required supply voltage for the ESP32-P4's MIPI D-PHY display interface.

```
158  void ldo_init()
159  {
160      // --- Power Configuration (LDO3 for MIPI D-PHY) ---
161      // ESP32-P4's MIPI D-PHY requires specific voltage to function.
162      // LDO3 is typically routed to the MIPI power rail on P4 hardware.
163      esp_err_t err = ESP_OK;
164      esp_ldo_channel_handle_t ldo3_handle = NULL;
165      esp_ldo_channel_config_t ldo3_cfg = {
166          .chan_id = 3,           // LDO Channel 3
167          .voltage_mv = 2500,     // Set to 2500mV (2.5V)
168      };
```

The program acquires and enables this LDO channel via "esp_ldo_acquire_channel()". If the return value is not "ESP_OK", indicating power initialization failure, the program outputs an error message using "Serial.printf()". Upon success, a confirmation message is printed to the serial port indicating that LDO3 has been successfully activated.

```
170      Serial.println("Initializing LDO3 to 2.5V...");
171      err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
172      if (err != ESP_OK) {
173          Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
174      } else {
175          Serial.println("LDO3 Power enabled successfully.");
176      }
```

Subsequently, the program configures LDO4 in the same manner: setting "chan_id" to 4 and the output voltage to 3300mV (3.3V)—a voltage level typically used for I2C bus pull-up resistors and touch chip power supply (such as the GT911 touch controller).
It again invokes "esp_ldo_acquire_channel()" to enable this power channel, printing success or failure messages based on the returned status.

```
178     // --- Power Configuration (LDO4 for I2C/touch pull up) ---
179     esp_ldo_channel_handle_t ldo4_handle = NULL;
180     esp_ldo_channel_config_t ldo4_cfg = {
181         .chan_id = 4,           // LDO Channel 4
182         .voltage_mv = 3300,     // Set to 3300mV (3.3V)
183     };
184
185     Serial.println("Initializing LDO4 to 3.3V...");
186     err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
187     if (err != ESP_OK) {
188         Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
189     } else {
190         Serial.println("LDO4 Power enabled successfully.");
191     }
192 }
```

7. display_touch_lvgl_init

This code defines a function "display_touch_lvgl_init()", whose primary purpose is to complete the comprehensive initialization of the display panel, touchscreen, and graphics library—enabling the system to properly render graphical interfaces and respond to touch operations.

```
194 void display_touch_lvgl_init()
195 {
196     // --- Initialize Display and Touch Panel ---
197     Board *board = new Board();
198     // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
199     Serial.println("Initializing Panel (EK79007 + GT911)...");
200     assert(board->init());
201 #if LVGL_PORT_AVOID_TEARING_MODE
202     auto lcd = board->getLCD();
203     // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
204     lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
205 #endif
206     assert(board->begin());
207     Serial.println("Display and Touch system online.");
208
209     Serial.println("Initializing LVGL");
210     lvgl_port_init(board->getLCD(), board->getTouch());
211 }
```

The function first instantiates a development board control object "board" through "new Board()", which typically encapsulates underlying hardware interfaces including the display bus, LCD driver, and touch driver.

The program then outputs debug information "Initializing Panel (EK79007 + GT911)..." and invokes "board->init()" to initialize the display hardware bus and devices—comprising the EK79007 LCD controller driven via the MIPI-DSI bus, and the capacitive touch control chip GT911 communicating over I2C.

"assert()" is employed to immediately terminate the program upon initialization failure, preventing the system from continuing operation when hardware has not been properly initialized.

```
194  void display_touch_lvgl_init()
195  {
196      // --- Initialize Display and Touch Panel ---
197      Board *board = new Board();
198      // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
199      Serial.println("Initializing Panel (EK79007 + GT911)...");
200      assert(board->init());
```

Subsequently, when the conditional compilation flag "LVGL_PORT_AVOID_TEARING_MODE" is enabled, the program retrieves the LCD object via "board->getLCD()" and invokes "configFrameBufferNumber()" to configure the frame buffer count—thereby reducing potential screen tearing artifacts during display refresh cycles.

```
201  #if LVGL_PORT_AVOID_TEARING_MODE
202      auto lcd = board->getLCD();
203      // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
204      lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
205  #endif
```

The system then calls "board->begin()" to boot up the entire display and touch subsystem, bringing the hardware into normal operating state, and prints the message "Display and Touch system online." to the serial port.

```
206      assert(board->begin());
207      Serial.println("Display and Touch system online.");
208
209      Serial.println("Initializing LVGL");
210      lvgl_port_init(board->getLCD(), board->getTouch());
211  }
212
```

Finally, the program outputs "Initializing LVGL" and invokes "lvgl_port_init(board->getLCD(), board->getTouch())" to initialize the LVGL graphics library, simultaneously registering the LCD display driver and touch input device with the LVGL system — enabling LVGL to handle interface rendering and receive touch events.

8. Setup Section

This code defines the "setup()" function in the Arduino program, whose role is to complete the initialization of the entire hardware system and graphical interface after system power-on or reset, preparing the groundwork for subsequent operation.

```
213  void setup() {
214      // put your setup code here, to run once:
215
216      // Initialize the default Serial for debugging (UART0)
217      Serial.begin(115200);
218
219      ldo_init();
220
221  #if (1 == ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST)
222      i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA);
223  #endif
224
225      //Since the display library automatically initializes the I2C bus for touch, there is no need to initialize the i2c bus again
226      //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
227      /**
228       * If set to 1, the bus will skip to initialize the corresponding host. Users need to initialize the host in advance.
229       *
230       * For drivers which created by this library, even if they use the same host, the host will be initialized only once.
231       * So it is not necessary to set the macro to `1`. For other drivers (like `Wire`), please set the macro to `1`
232       * ensure that the host is initialized only once.
233       */
234      // #define ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST        (0)     // 0/1. Typically set to 0
235      display_touch_lvgl_init();
236
237      dht20_begin();
238
239      Serial.println("Creating UI");
240
241      dht20_display();
242  }
```

The function first initializes serial communication (UART0) via "Serial.begin(115200)" for outputting debug information. It then invokes "ldo_init()" to initialize the LDO power modules on the development board, providing stable voltage to critical hardware—such as powering the ESP32-P4's display interface.

```
213  void setup() {
214      // put your setup code here, to run once:
215
216      // Initialize the default Serial for debugging (UART0)
217      Serial.begin(115200);
218
219      ldo_init();
```

The code then employs conditional compilation to check whether the macro "ESP_PANEL_BOARD_TOUCH_BUS_SKIP_INIT_HOST" is defined as 1. If so, it manually initializes the I2C bus by calling "i2c_init(I2C_GPIO_SCL, I2C_GPIO_SDA)".

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

The code comments explain that, by default, the display driver library automatically initializes the I2C bus used by the touchscreen—therefore manual re-initialization is typically unnecessary. Should users wish to manually control the I2C initialization flow, they must modify this macro definition in the configuration file "esp_panel_board_custom_conf.h".

The program subsequently invokes "display_touch_lvgl_init()" to initialize the display and touch system. This function boots the LCD display driver (such as EK79007), the GT911 touch controller, and initializes the LVGL graphics library — endowing the system with graphical display and touch interaction capabilities.

```
235     display_touch_lvgl_init();
236
237     dht20_begin();
238
239     Serial.println("Creating UI");
240
241     dht20_display();
242 }
```

The program then invokes "dht20_begin()" to initialize the DHT20 temperature and humidity sensor, preparing it for subsequent environmental data acquisition. Finally, it outputs the prompt message via "Serial.println("Creating UI")" and calls "dht20_display()" to create the temperature and humidity display label in the LVGL interface—thereby establishing a UI element on screen for presenting temperature and humidity data.

9. 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.

```
244 void loop() {
245     // put your main code here, to run repeatedly:
246
247     dht20_read_task(nullptr);
248 }
```

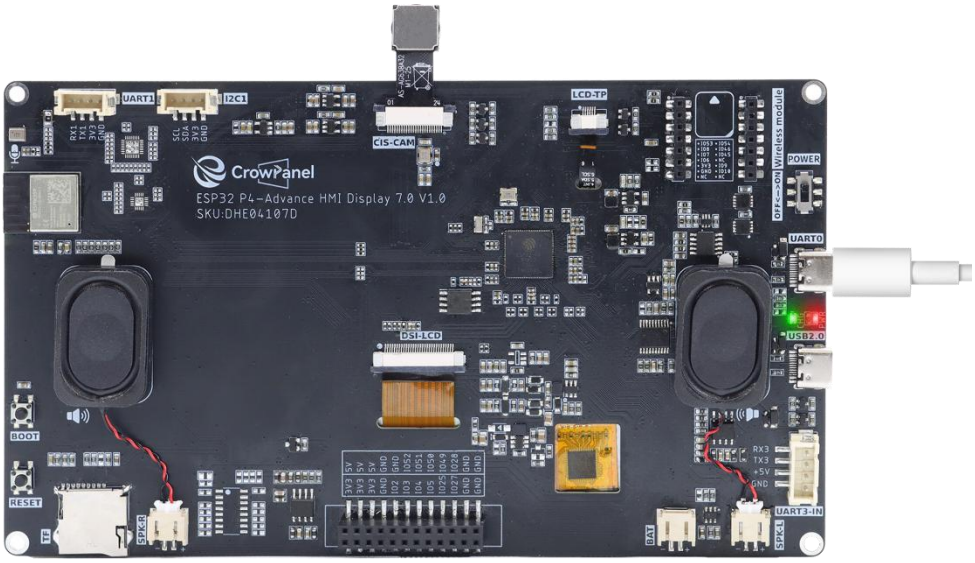
The "nullptr" is passed as an argument because this function was originally designed as a task function (typically used in operating system tasks or threads), requiring a "void *" type parameter. Since it is not actually utilized here, passing a null pointer suffices.

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.

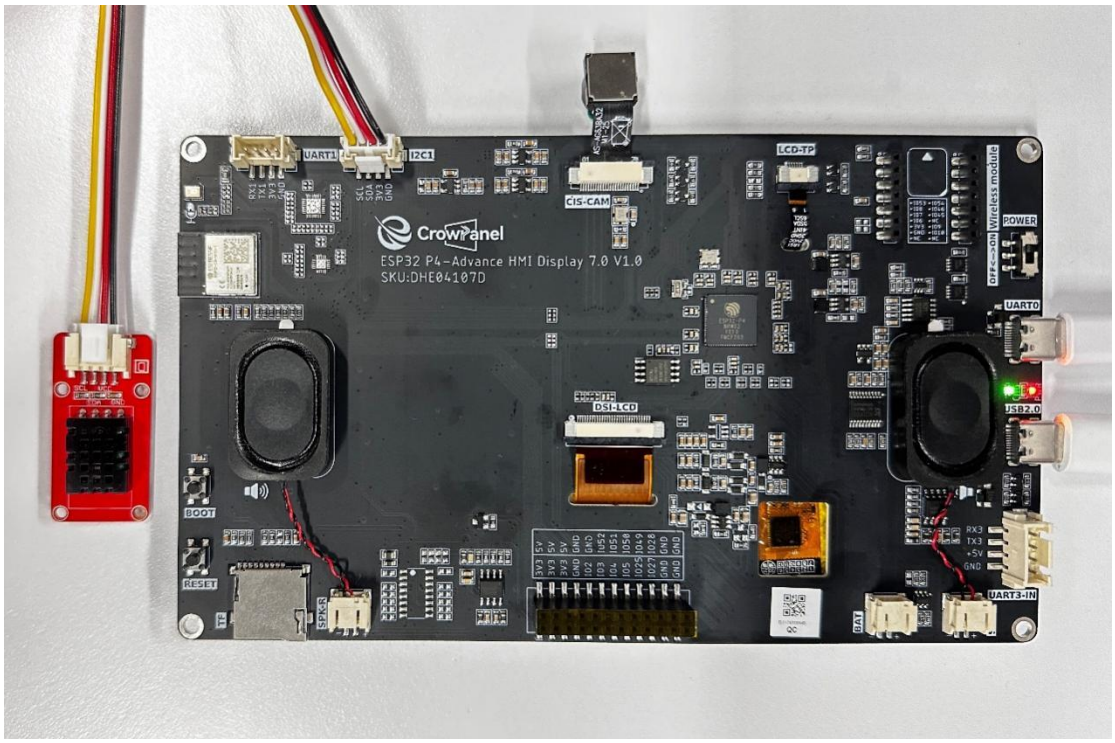
As "dht20_read_task()" internally contains a "while(1)" infinite loop with timed delay logic, once this function is invoked within "loop()", the program remains inside this task indefinitely, continuously executing sensor reading and interface update operations. This implements a continuous monitoring function that reads temperature and humidity at fixed intervals and refreshes the screen display accordingly.

Programming Steps

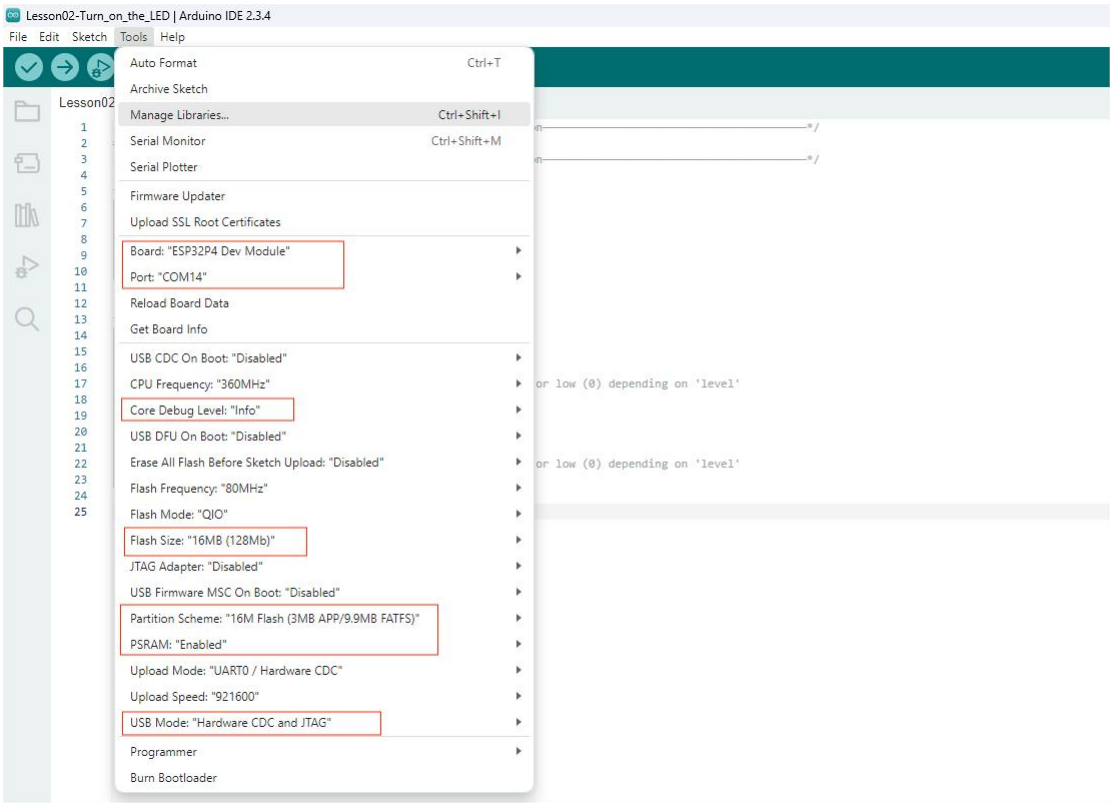
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



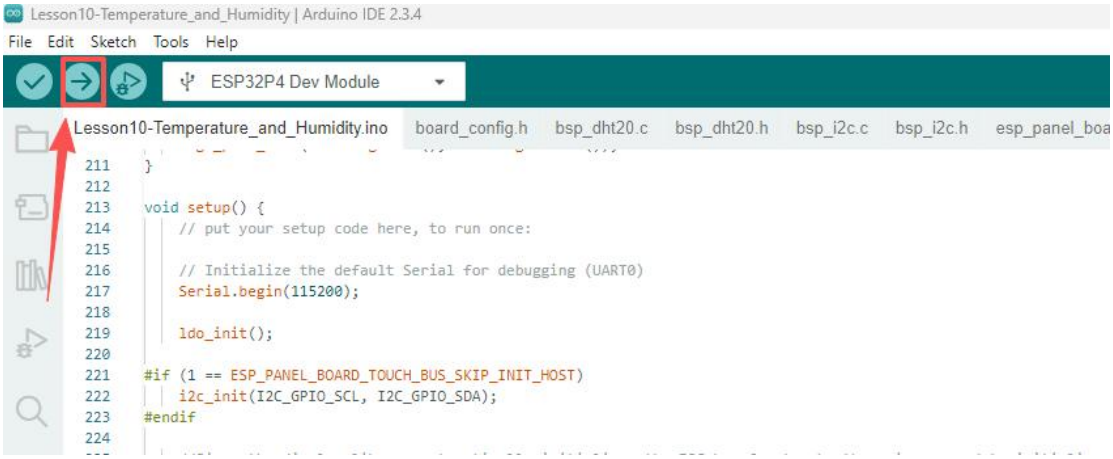
After connecting the Advance-P4 board, connect the temperature and humidity sensor to the I2C interface.



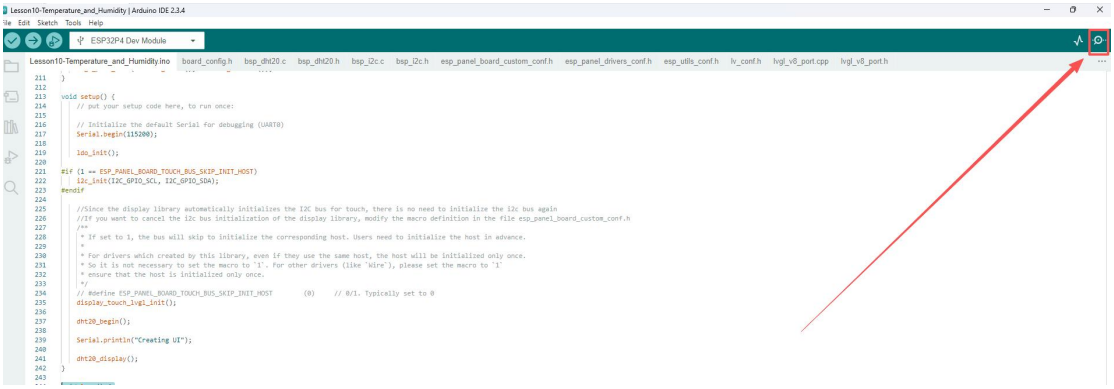
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



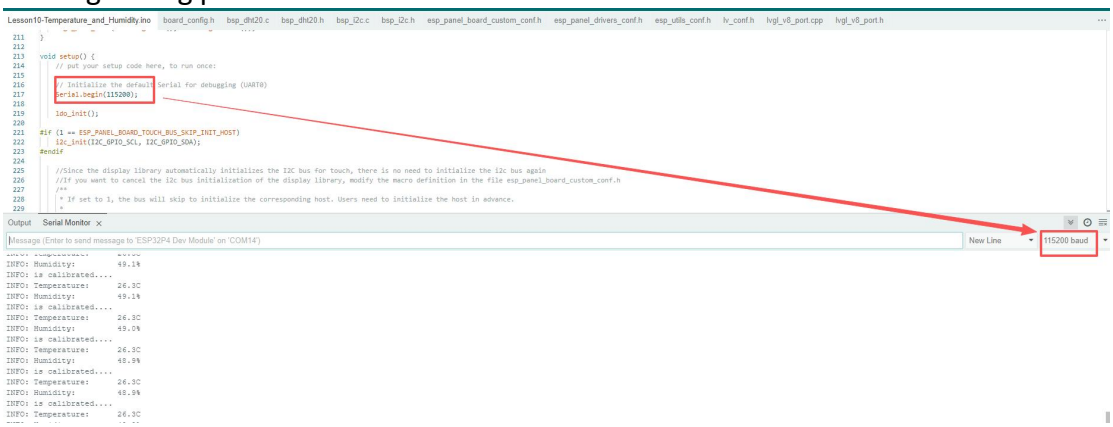
Then we compile and upload the code.



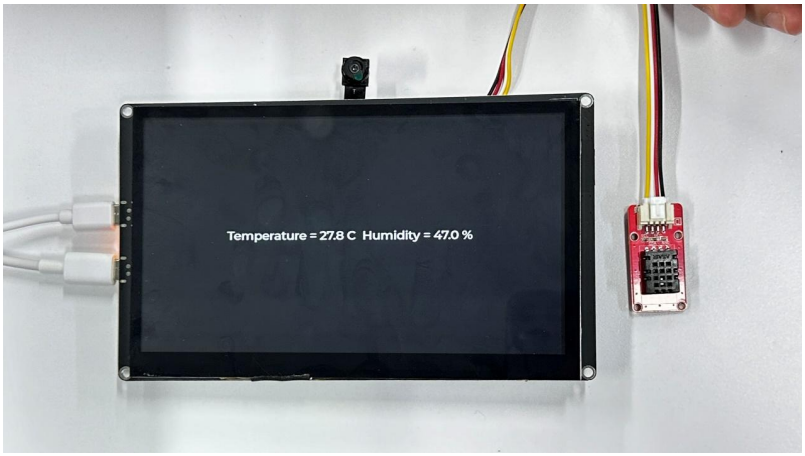
Wait for the compilation to complete, then open the Serial Monitor built into the Arduino IDE.



Configure the baud rate in your code, and you'll see the temperature and humidity readings being printed.



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.



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

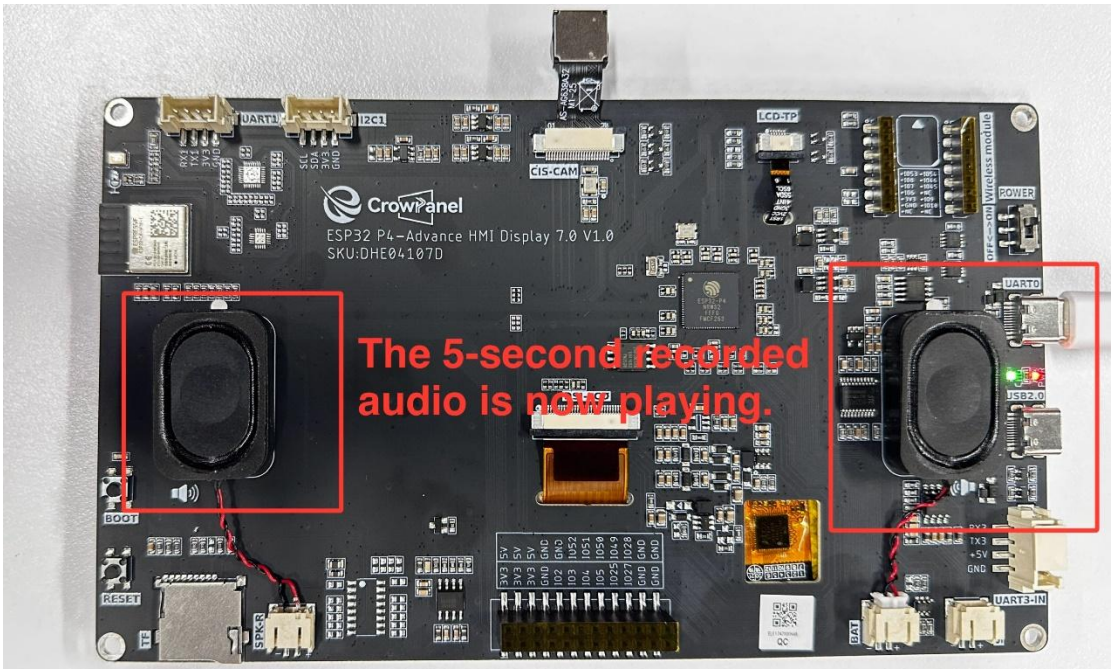
1. Understand the principles of the I2S/PDM audio interface on the ESP32-P4 development board, as well as the working mechanisms of microphones (acoustic-to-electrical conversion) and speakers (electrical-to-acoustic conversion).

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.

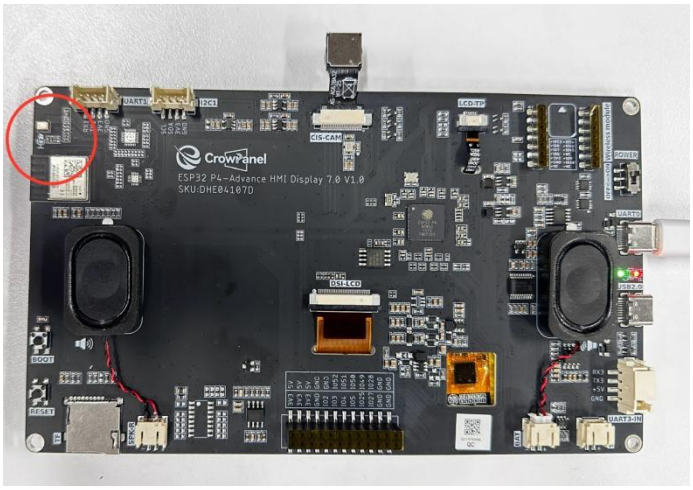
Preview of the Result

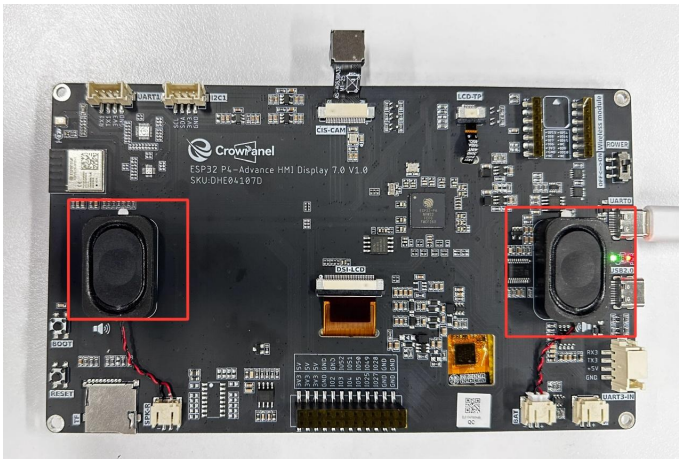
After running the code, you will be able to speak near the Advance-P4. The Advance-P4 will use its microphone to record the current sound within 5 seconds, then play it back automatically.



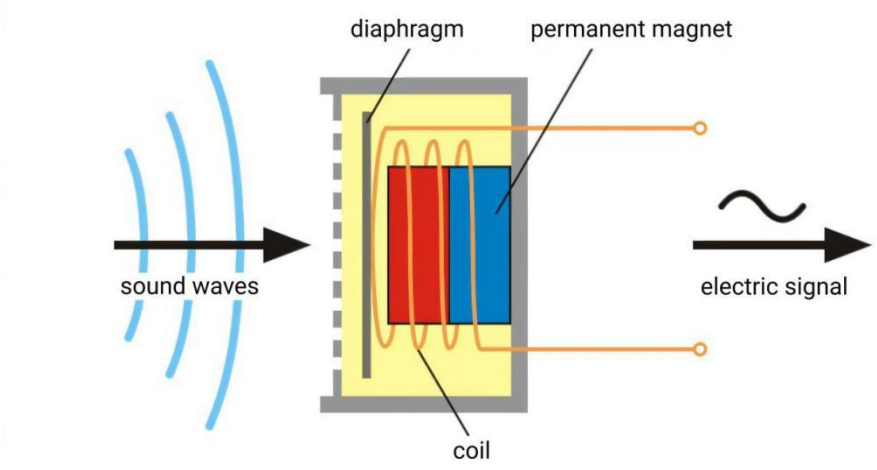
Hardware Used in This Lesson

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.)

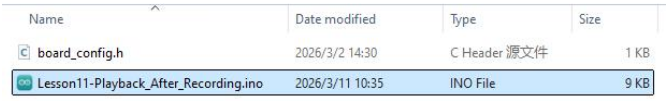
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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.

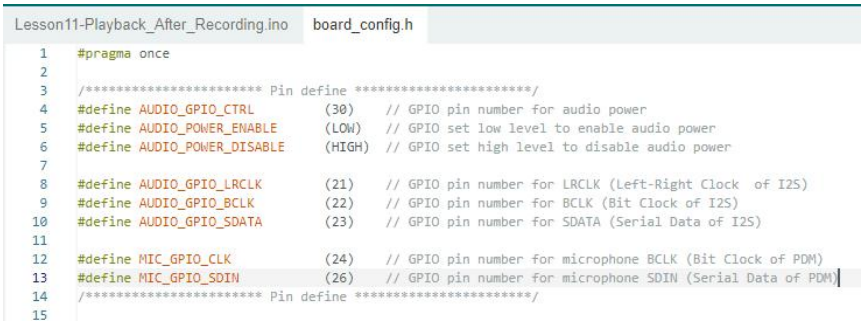
Double-click to open the code for this lesson (the .ino file).



Name	Date modified	Type	Size
board_config.h	2026/3/2 14:30	C Header 源文件	1 KB
Lesson11-Playback_After_Recording.ino	2026/3/11 10:35	INO File	9 KB

Once opened, you will see the project code along with the configuration file "board_config.h".

The "board_config.h" file serves to centrally define all GPIO pins and control parameters related to the audio system on the development board. This allows the main program to reference macro names when using hardware, rather than directly writing specific pin numbers—thereby enhancing code readability, maintainability, and hardware portability.



Lesson11-Playback_After_Recording.ino | board_config.h

```
1   #pragma once
2
3   /********************** Pin define **********************/
4   #define AUDIO_GPIO_CTRL         (30)    // GPIO pin number for audio power
5   #define AUDIO_POWER_ENABLE      (LOW)   // GPIO set low level to enable audio power
6   #define AUDIO_POWER_DISABLE     (HIGH)  // GPIO set high level to disable audio power
7
8   #define AUDIO_GPIO_LRCLK        (21)    // GPIO pin number for LRCLK (Left-Right Clock  of I2S)
9   #define AUDIO_GPIO_BCLK         (22)    // GPIO pin number for BCLK (Bit Clock of I2S)
10  #define AUDIO_GPIO_SDATA        (23)    // GPIO pin number for SDATA (Serial Data of I2S)
11
12  #define MIC_GPIO_CLK            (24)    // GPIO pin number for microphone BCLK (Bit Clock of PDM)
13  #define MIC_GPIO_SDIN           (26)    // GPIO pin number for microphone SDIN (Serial Data of PDM)
14  /********************** Pin define **********************/
15
```

The file begins with #pragma once to prevent duplicate header file inclusion.
The code is then organized into a Pin define (pin definition) section:
First, AUDIO_GPIO_CTRL (GPIO30) is defined as the audio power control pin, with two macros AUDIO_POWER_ENABLE and AUDIO_POWER_DISABLE specifying the power control logic—where setting LOW enables audio power and HIGH disables it. This allows the program to control the audio module's power supply state simply by invoking these macros.

Next, the I2S speaker interface pins are defined: AUDIO_GPIO_LRCLK (GPIO21) as the left-right channel clock LRCLK, AUDIO_GPIO_BCLK (GPIO22) as the bit clock BCLK, and AUDIO_GPIO_SDATA (GPIO23) as the serial audio data output line SDATA. These pins are used to transmit digital audio data to the speaker or amplifier module via I2S.

Finally, the microphone input pins are defined: MIC_GPIO_CLK (GPIO24) as the microphone clock signal line, and MIC_GPIO_SDIN (GPIO26) as the microphone data input line. These two pins connect to the PDM digital microphone, transferring captured audio data to the ESP32-P4's audio interface for processing.
Now let's focus on the implementation in the main .ino code file to understand how this lesson's functionality is achieved.

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.

```
4   #include "board_config.h"   // board pin define
5   #include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recognition of Arduino APIs
6
7   #include <string.h>         // Include standard string manipulation functions
8   #include <esp_log.h>        // ESP-IDF logging library
9   #include <esp_err.h>        // ESP-IDF error codes
10
11  #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.

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.

```
54  /*microphone*/
55  static I2SClass i2s_mic; // Create an I2SClass microphone instance
56  /*loudspeaker*/
57  static I2SClass i2s_spk; // Create an I2SClass speaker instance
58
59  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.

```
61  void recording()
62  {
63      uint8_t *origin_buffer;
64      int16_t *read_buffer;
65      size_t read_bytes;
66      int16_t *out_buffer;
67      out_buffer = (int16_t*)heap_caps_malloc((16000 * 2 * 5), MALLOC_CAP_SPIRAM | MALLOC_CAP_DMA | MALLOC_CAP_32BIT);
68      if (out_buffer==NULL) {
69          Serial.println("Memory allocation failure!");
70      }
```

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.

```
72      Serial.println("Recording 5 seconds of audio data...");
73      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".

```
74      read_bytes -= 44;
75      read_buffer = (int16_t*)(44 + origin_buffer);
76      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.

```
78      Serial.printf("read_bytes = %d\n", read_bytes);
79      Serial.println("Recording end");
80
81      // Find the maximum absolute value (peak) in the raw data
82      int32_t max_val = 0;
83      for (size_t i = 0; i < num_samples; i++) {
84          // Use abs() function to ensure absolute value is taken
85          int32_t current_val = abs(read_buffer[i]);
86          if (current_val > max_val) {
87              max_val = current_val;
88          }
89      }
```

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;
        }
    }
}
```

Upon processing completion, the program initiates audio playback: it first enables the audio power via "digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_ENABLE)", then utilizes the speaker object "i2s_spk.write()" to transmit the processed audio data to the speaker through I2S for playback.

```
Serial.println("Playing the recorded audio...");
digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_ENABLE);  // Enable audio power
/* Skip the 4000 bytes of audio data because the first few microphone audio data
i2s_spk.write((uint8_t*)out_buffer + 4000, read_bytes - 4000);
digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE); // Disable audio power

free(origin_buffer);

free(out_buffer);
}
```

The code deliberately skips the first 4000 bytes of data, as the ESP32-P4 may produce popping noises at the beginning when using the Arduino audio library for recording—necessitating the discarding of a small audio segment.

Upon playback completion, the audio power is disabled, and the memory occupied by origin_buffer and out_buffer is released.

4. mic_loudspeaker_init

This code defines the function mic_loudspeaker_init(), whose primary role is to initialize the microphone input and speaker output interfaces in the audio system, enabling the device to perform recording and playback operations.

```
133  void mic_loudspeaker_init()
134  {
135      pinMode(AUDIO_GPIO_CTRL, OUTPUT);
136      digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE);   // Disable audio power
137
138      i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN); // Configure pins for microphone input
139      // Start I2S at 16 kHz frequency, 16-bit depth, mono
140      if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
141          Serial.println("PDM input initialization failed!");
142          while (1) delay(1000);  // Halt execution
143      }
144
145      i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  // BCLK, LRCLK, DOUT
146      // Start I2S with the same parameters, but in output mode, using mono
147      if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_BOTH)) {
148          Serial.println("I2S output mode initialization failed!");
149          while (1) delay(1000);
150      }
151  }
```

The function first configures the audio power control pin as output mode via "pinMode(AUDIO_GPIO_CTRL, OUTPUT)", and ensures the audio module is in a safe state at system startup by disabling the audio power through "digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE)".

```
133  void mic_loudspeaker_init()
134  {
135      pinMode(AUDIO_GPIO_CTRL, OUTPUT);
136      digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE);
```

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.

```
138      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.

```
139      // Start I2S at 16 kHz frequency, 16-bit depth, mono
140      if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
141          Serial.println("PDM input initialization failed!");
142          while (1) delay(1000);  // Halt execution
143      }
```

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.

```
145      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.

```
146     // Start I2S with the same parameters, but in output mode, using mono
147     if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_BOTH)) {
148         Serial.println("I2S output mode initialization failed!");
149         while (1) delay(1000);
150     }
151 }
```

5. Setup Section

This code defines the "setup()" function in the Arduino program, which executes only once after system power-on or reset to complete initialization work before the program begins running.

```
154 void setup() {
155     // put your setup code here, to run once:
156
157     // Initialize the default Serial for debugging (UART0)
158     Serial.begin(115200);
159
160     mic_loudspeaker_init();
161 }
```

The function first initializes the serial communication interface (UART0) via Serial.begin(115200), setting the baud rate to 115200. This enables the program to output debug information to the computer through the serial port, such as device status, error prompts, or runtime logs.

The program then invokes the mic_loudspeaker_init() function to initialize the audio system. This function completes the configuration of microphone input and speaker output interfaces—including setting the audio power control pin, configuring the microphone's PDM input pins, initializing the digital audio interface, and launching the audio output channel—thereby endowing the development board with audio recording and playback capabilities.

The entire initialization process primarily prepares the groundwork for subsequent I2S-based audio acquisition and playback 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.

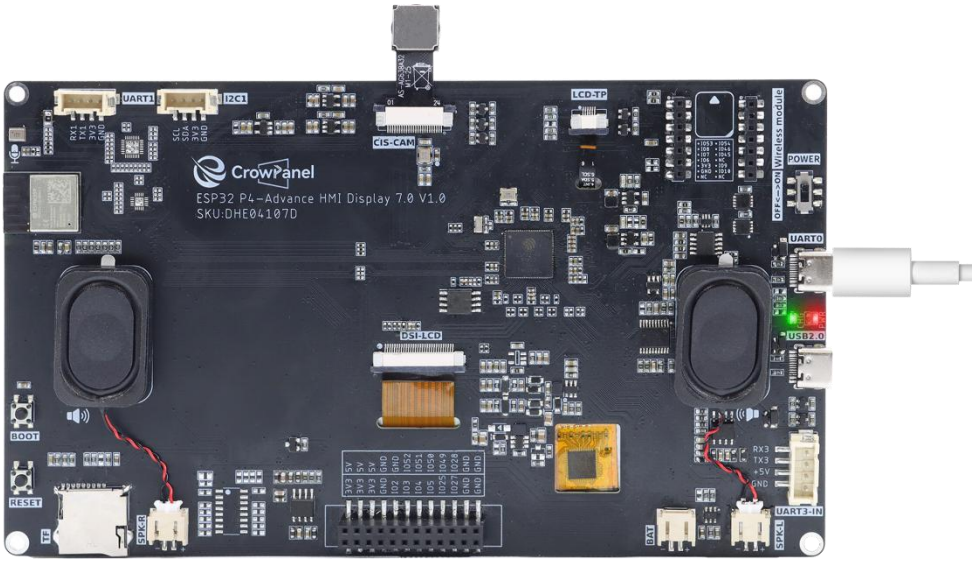
```
163  void loop() {
164      // put your main code here, to run repeatedly:
165
166      recording();
167  }
```

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.

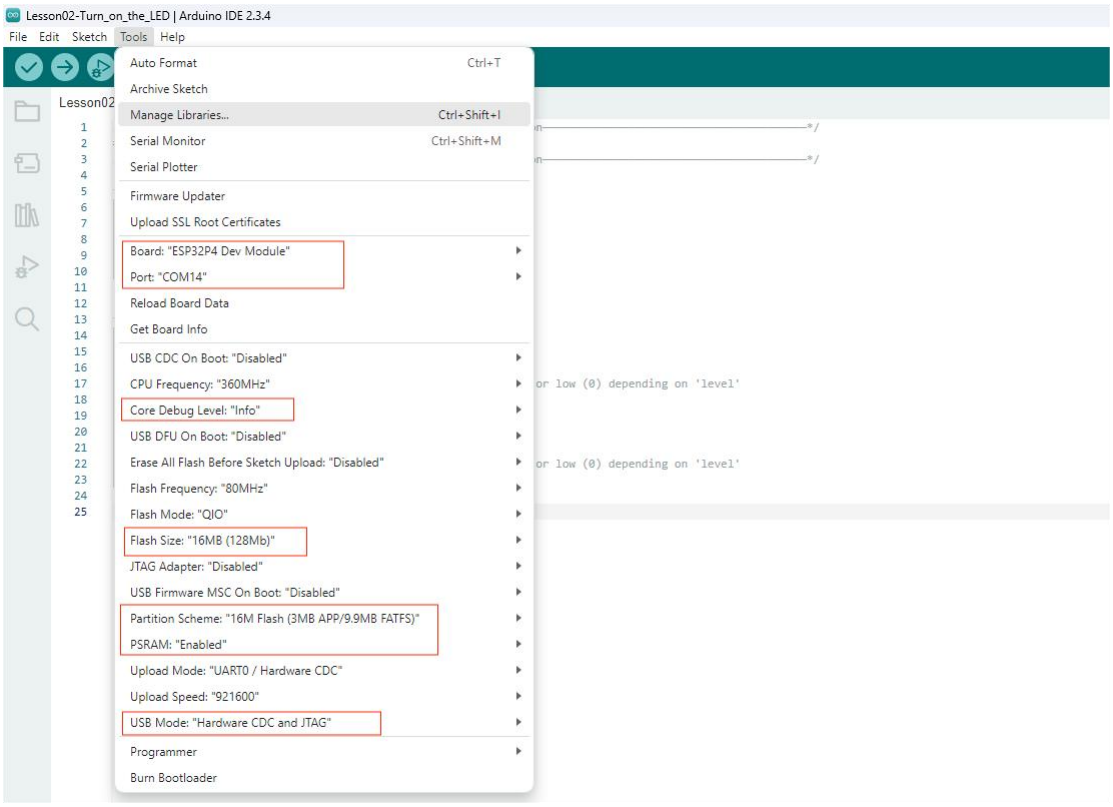
Since "loop()" executes continuously throughout program runtime, the system repeatedly performs the recording → audio processing → speaker playback sequence—forming a real-time audio monitoring system that enables the device to continuously perform audio acquisition and playback operations via I2S on 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.

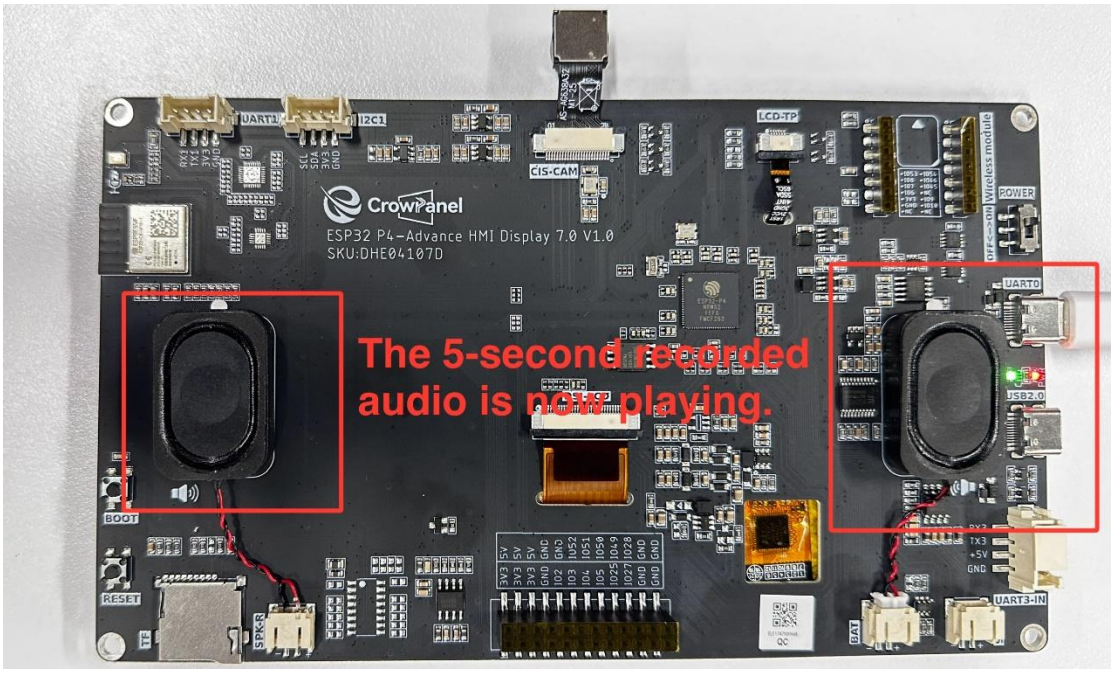


Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



Then we compile and upload the code.

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.


Crowpanel
ESP32 P4-Advance HMI Display 7.0 V1.0
SKU:DHE04107D
The 5-second recorded audio is now playing.
UART1
I2C1
CIS-CAM
LCD-TP
Wireless module
POWER
UART0
USB2.0
UART3-IN
BOOT
RESET
SPK-R
TF
BAT

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

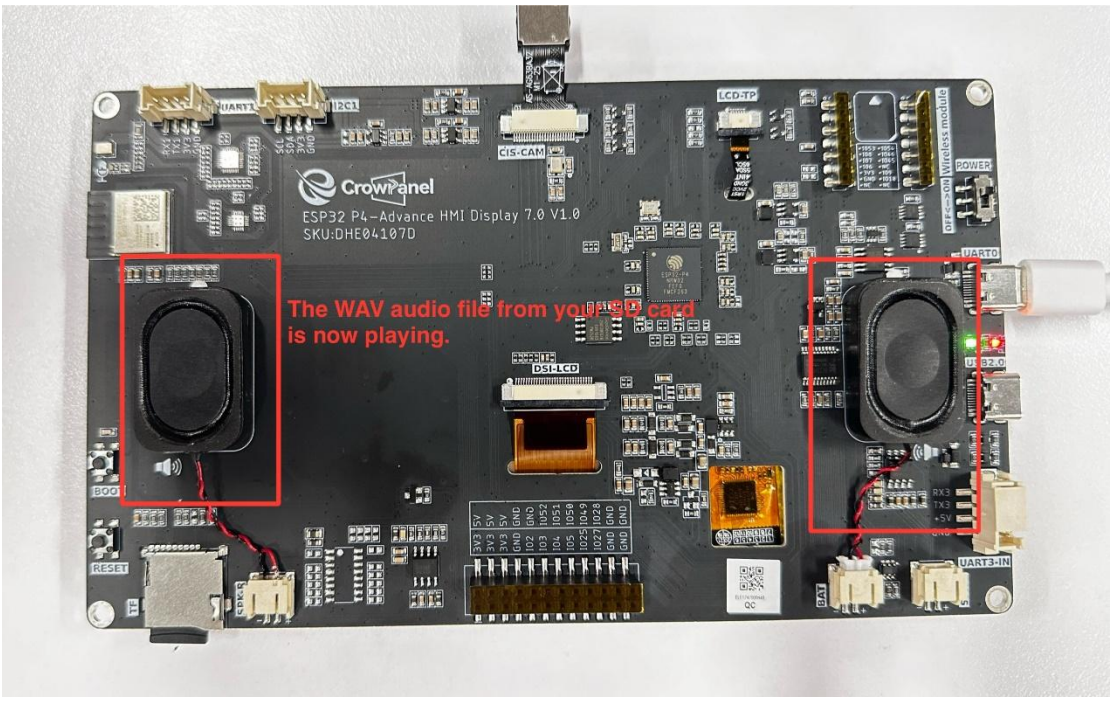
1. Understand the collaborative working principles of SD card file reading and I2S audio playback on the ESP32-P4 development board, and master the structure and validation rules of standard WAV file headers.

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.

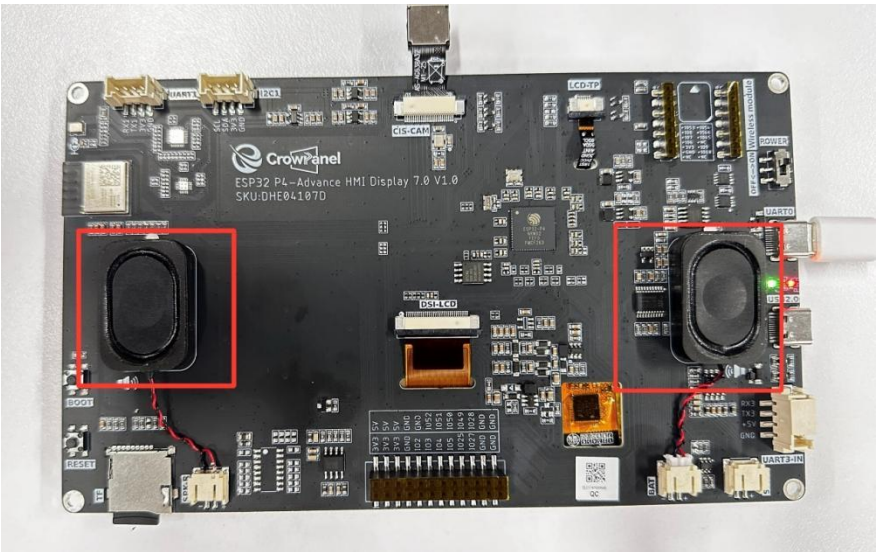
Preview of the Result

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.

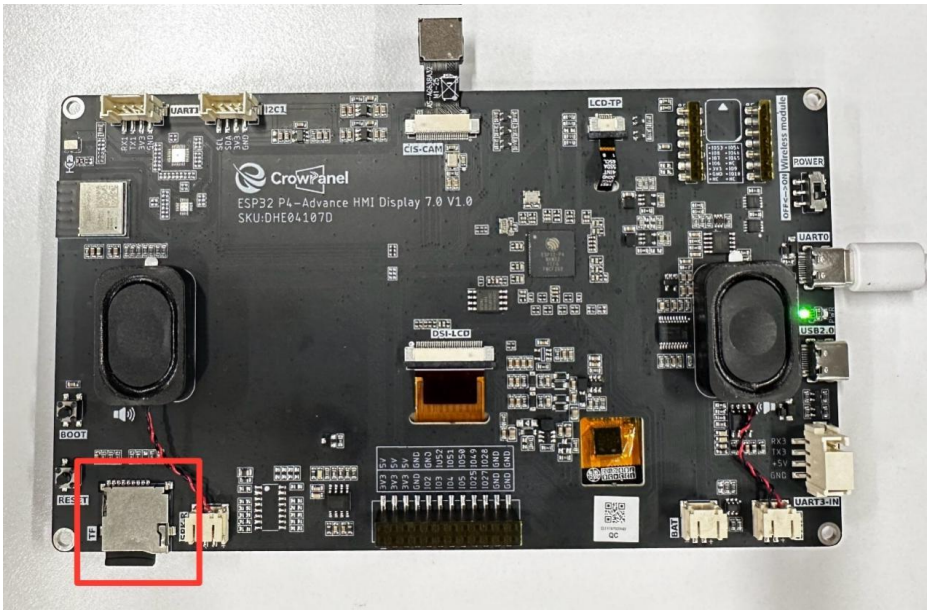


Hardware Used in This Lesson

Speaker on the Advance-P4



SD Card on the Advance-P4



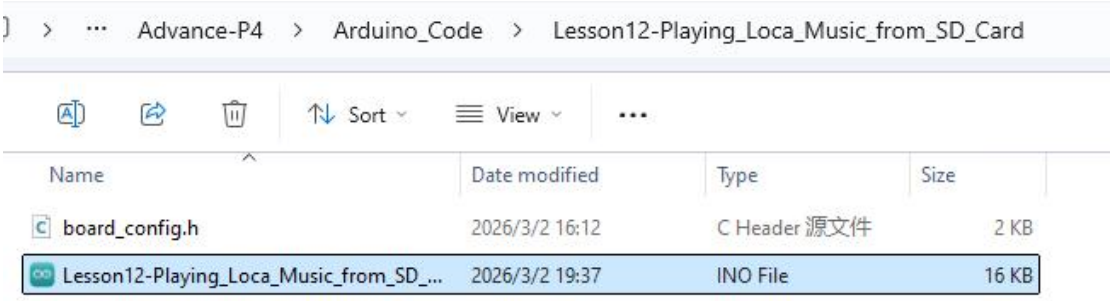
Complete Code

First, click the GitHub link below to download the code for this lesson.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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).



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.

```
126  bool validate_wav_header(FILE *file)
127  {
128      if (file == NULL)
129      {
130          AUDIO_ERROR("File pointer is NULL");
131          return false;
132      }
```

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.

```
133     long original_position = ftell(file); /*Store current file
134     if (original_position == -1)
135     {
136         AUDIO_ERROR("Cannot get current file position");
137         return false;
138     }
```

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.

```
139     if (fseek(file, 0, SEEK_SET) != 0) /*Rewind to beginning of file*/
140     {
141         AUDIO_ERROR("Cannot seek to file beginning");
142         return false;
143     }
```

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".

```
144     uint8_t header[44]; /*Read and validate WAV header*/
145     size_t bytes_read = fread(header, 1, 44, file);
146     if (bytes_read != 44)
147     {
148         AUDIO_ERROR("Cannot read complete WAV header (%d bytes)", bytes_read);
149         fseek(file, original_position, SEEK_SET);
150         return false;
151     }
```

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.

```
152     if (memcmp(header, "RIFF", 4) != 0) /*Validate RIFF chunk descriptor*/
153     {
154         AUDIO_ERROR("Invalid RIFF header");
155         fseek(file, original_position, SEEK_SET);
156         return false;
157     }
```

It then checks whether the bytes at offset 8 contain "WAVE" to confirm that the file type is indeed in WAV format.

```
158     if (memcmp(header + 8, "WAVE", 4) != 0) /*Validate WAVE format*/
159     {
160         AUDIO_ERROR("Invalid WAVE format");
161         fseek(file, original_position, SEEK_SET);
162         return false;
163     }
```

It then checks for the "fmt " sub-chunk identifier at offset 12 bytes, indicating the

presence of the format information block in the file.

```
164     if (memcmp(header + 12, "fmt ", 4) != 0) /*Validate fmt subchunk*/
165     {
166         AUDIO_ERROR("Invalid fmt subchunk");
167         fseek(file, original_position, SEEK_SET);
168         return false;
169     }
```

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.

```
170     uint16_t audio_format = *(uint16_t *)(header + 20); /*Check audio format (should be 1 for PCM)*/
171     if (audio_format != 1)
172     {
173         AUDIO_ERROR("Unsupported audio format: %d (only PCM supported)", audio_format);
174         fseek(file, original_position, SEEK_SET);
175         return false;
176     }
```

It then reads "(header + 22)" to obtain the number of channels "num_channels", verifying that it is either 1 (mono) or 2 (stereo).

```
177     uint16_t num_channels = *(uint16_t *)(header + 22); /*Check number of c
178     if (num_channels != 1 && num_channels != 2)
179     {
180         AUDIO_ERROR("Unsupported number of channels: %d", num_channels);
181         fseek(file, original_position, SEEK_SET);
182         return false;
183     }
```

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).

```
184     uint32_t sample_rate = *(uint32_t *)(header + 24); /*Check sample rate (support common rates)*/
185     if (sample_rate != 8000 && sample_rate != 16000 && sample_rate != 22050 && sample_rate != 44100 && sample_rate != 48000)
186     {
187         AUDIO_ERROR("Uncommon sample rate: %lu Hz", sample_rate);
188         fseek(file, original_position, SEEK_SET);
189         return false;
190     }
```

It then reads "(header + 34)" to obtain the bits per sample "bits_per_sample", permitting only 8, 16, 24, or 32 bits.

```
191     uint16_t bits_per_sample = *(uint16_t *)(header + 34); /*Check bits per sample (support 8, 16, 24, 32)*/
192     if (bits_per_sample != 8 && bits_per_sample != 16 && bits_per_sample != 24 && bits_per_sample != 32)
193     {
194         AUDIO_ERROR("Unsupported bits per sample: %d", bits_per_sample);
195         fseek(file, original_position, SEEK_SET);
196         return false;
197     }
```

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.

```
198     if (memcmp(header + 36, "data", 4) != 0) /*Validate data subchunk*/
199     {
200         AUDIO_ERROR("Invalid data subchunk");
201         fseek(file, original_position, SEEK_SET);
202         return false;
203     }
```

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.

```
204     /*Get file size from RIFF header for additional validation*/
205     uint32_t file_size = *(uint32_t *)(header + 4) + 8; // RIFF block size + 8-byte header
206     uint32_t data_size = *(uint32_t *)(header + 40);
207
208     AUDIO_INFO("WAV File Info: %d channels, %lu Hz, %d bits, %lu bytes data, %lu bytes total",
209                num_channels, sample_rate, bits_per_sample, data_size, file_size);
210     /*Restore original file position*/
211     fseek(file, original_position, SEEK_SET);
212     return true;
213 }
```

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.

```
215 esp_err_t Audio_play_wav_sd(const char *filename)
216 {
217     esp_err_t err = ESP_OK;
218     if (filename == NULL)
219         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.

```
221     FILE *fh = fopen(filename, "rb");
222     if (fh == NULL)
223     {
224         AUDIO_ERROR("Failed to open file");
225         return ESP_ERR_INVALID_ARG;
226     }
```

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.

```
227     if (!validate_wav_header(fh)) /*Validate WAV header*/
228     {
229         AUDIO_ERROR("Invalid WAV file format: %s", filename);
230         fclose(fh);
231         return ESP_ERR_INVALID_ARG;
232     }
```

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.

```
233     if (fseek(fh, 44, SEEK_SET) != 0) /*Skip 44-byte WAV header*/
234     {
235         AUDIO_ERROR("Failed to seek file");
236         fclose(fh);
237         return ESP_FAIL;
238     }
```

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.

```
239     /*Buffer configuration*/
240     const size_t SAMPLES_PER_BUFFER = 512;
241     const size_t INPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * sizeof(int16_t);
242     const size_t OUTPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * 2 * sizeof(int16_t);
243     /* Allocate buffers*/
244     int16_t *input_buf = (int16_t*)heap_caps_malloc(INPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM | MALLOC_CAP_DMA | MALLOC_CAP_32BIT);
245     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.

```
247     if (input_buf == NULL || output_buf == NULL)
248     {
249         AUDIO_ERROR("Failed to allocate audio buffers");
250         if (input_buf)
251             free(input_buf);
252         if (output_buf)
253             free(output_buf);
254         fclose(fh);
255         return ESP_ERR_NO_MEM;
256     }
```

It then enables the audio power via "digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_ENABLE)", bringing the amplifier or audio circuitry into operational state.

```
257     size_t samples_read = 0;
258     size_t bytes_to_write = 0;
259     size_t bytes_written = 0;
260     size_t total_samples = 0;
261     int32_t volume_data = 0;
262     digitalWrite(AUDIO_GPIO_CTRL, 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.

```
263     while (1)
264     {
265         samples_read = fread(input_buf, sizeof(int16_t), SAMPLES_PER_BUFFER, fh);
266         if (samples_read == 0)
267             break;
268         for (size_t i = 0; i < samples_read; i++) /*convert mono to stereo*/
269         {
270             volume_data = input_buf[i] * 1; /*Linear multiplication*/
271             if (volume_data > 32767)
272                 volume_data = 32767;
273             else if (volume_data < -32768)
274                 volume_data = -32768;
275             output_buf[i] = (int16_t)volume_data;     /*Left channel*/
276         }
```

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.

```
278             bytes_to_write = samples_read * sizeof(int16_t);
279             bytes_written = 0;
280
281             bytes_written = i2s_spk.write((uint8_t*)output_buf, bytes_to_write);  /*I2S write data*/
282             if (bytes_written != bytes_to_write)
283             {
284                 AUDIO_ERROR("I2S write failed: %s, written: %d/%d", esp_err_to_name(err), bytes_written, bytes_to_write);
285                 break;
286             }
```

After each successful data transmission, the program accumulates total_samples to track the number of audio samples played.

```
286             }
287             total_samples += samples_read;
288         }
```

Upon loop termination, the function enters the cleanup phase: it disables the audio power via "digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE)", releases the previously allocated memory for "input_buf" and "output_buf", and closes the file handle "fh".

```
289         /*Cleanup*/
290         digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE);   // Disable audio power
291         free(input_buf);
292         free(output_buf);
293         fclose(fh);
294         AUDIO_INFO("Audio playback completed: %d samples", total_samples);
295         return err;
296     }
```

Finally, it outputs a log message indicating audio playback completion and the total sample count played, then returns the function execution result err—signifying that the entire WAV audio file has been successfully read from the SD card and played back through the I2S interface.

3.mic_loudspeaker_init

This code implements an audio device initialization function "mic_loudspeaker_init()", whose primary role is to complete I2S audio initialization for both microphone input and speaker output interfaces at system startup, while controlling the audio power state—preparing the groundwork for subsequent audio acquisition or playback functionality.

```
312 void mic_loudspeaker_init()
313 {
314     pinMode(AUDIO_GPIO_CTRL, OUTPUT);
315     digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE);   // Disable audio power
316
317     i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN); // Configure pins for microphone input
318     // Start I2S at 16 kHz frequency, 16-bit depth, mono
319     if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
320         Serial.println("PDM input initialization failed!");
321         while (1) delay(1000);  // Halt execution
322     }
323
324     i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA);  // BCLK, LRCLK, DOUT
325     // Start I2S with the same parameters, but in output mode, using mono
326     if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_STEREO, I2S_STD_SLOT_BOTH)) {
327         Serial.println("I2S output mode initialization failed!");
328         while (1) delay(1000);
329     }
330 }
```

The function begins by configuring the AUDIO_GPIO_CTRL pin as output mode via

pinMode(AUDIO_GPIO_CTRL, OUTPUT). This pin is typically used to control the power switch of the audio amplifier or audio circuitry. It then sets this pin to the disabled state through digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE)—turning off the audio power during the initialization phase to prevent the speaker from producing noise or abnormal signals during startup.

```
312  void mic_loudspeaker_init()
313  {
314      pinMode(AUDIO_GPIO_CTRL, OUTPUT);
315      digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE);   // Disable audio power
```

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.

```
317      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.

```
319      if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
320          Serial.println("PDM input initialization failed!");
321          while (1) delay(1000);  // Halt execution
322      }
```

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.

```
324      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.

```
326      if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_STEREO, I2S_STD_SLOT_BOTH)) {
327          Serial.println("I2S output mode initialization failed!");
328          while (1) delay(1000);
329      }
330  }
```

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 Arduino program's "setup()" initialization function, whose primary role is to execute system initialization once upon device power-on or reset—completing initialization of the serial debugging interface, SD card storage system, and audio input/output devices, while simultaneously reading and playing a WAV audio file from the SD card.

```
333  void setup() {
334      // put your setup code here, to run once:
335
336      // Initialize the default Serial for debugging (UART0)
337      Serial.begin(115200);
338
339      // Initialize sd card
340      sd_card_init();   // Call initialization function to set up SD card and other components
341
342      mic_loudspeaker_init();
343
344      Audio_play_wav_sd(SD_MOUNT_POINT"/huahai.wav");
345  }
```

The function begins by invoking "Serial.begin(115200)" to initialize the default serial port (UART0) with a baud rate of 115200. This enables developers to view debug information, error prompts, and log outputs during program execution through the serial monitor—facilitating development and troubleshooting.

The program then calls sd_card_init() to initialize the SD card module. This function completes SD card interface configuration, SDMMC communication initialization, and FAT file system mounting—enabling the system to access files on the SD card. Should SD card initialization fail, the function typically continues attempting reinitialization internally to ensure proper recognition and mounting of the SD card storage device.

Subsequently, the program invokes mic_loudspeaker_init() to initialize the audio hardware devices. This function is primarily responsible for configuring the I2S interfaces used by the microphone and speaker—including setting corresponding GPIO pins, configuring audio sample rate (such as 16 kHz), data bit width (16 bits), and channel mode (mono or stereo), while also initializing the audio power control pin—thereby ensuring the system can properly perform audio acquisition and output.

Upon completing these foundational hardware initializations, the program finally calls Audio_play_wav_sd(SD_MOUNT_POINT"/huahai.wav") to read a WAV audio file named huahai.wav (with full path /sdcard/huahai.wav) from the mounted SD card file system, and transmits the audio data to the speaker via the I2S interface for playback.

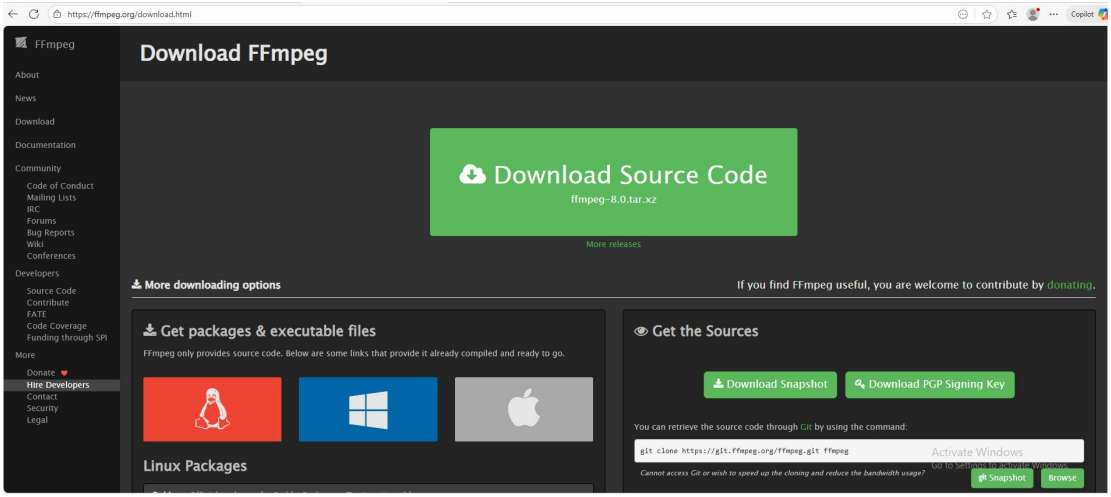
5. Converting MP3 to WAV

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).

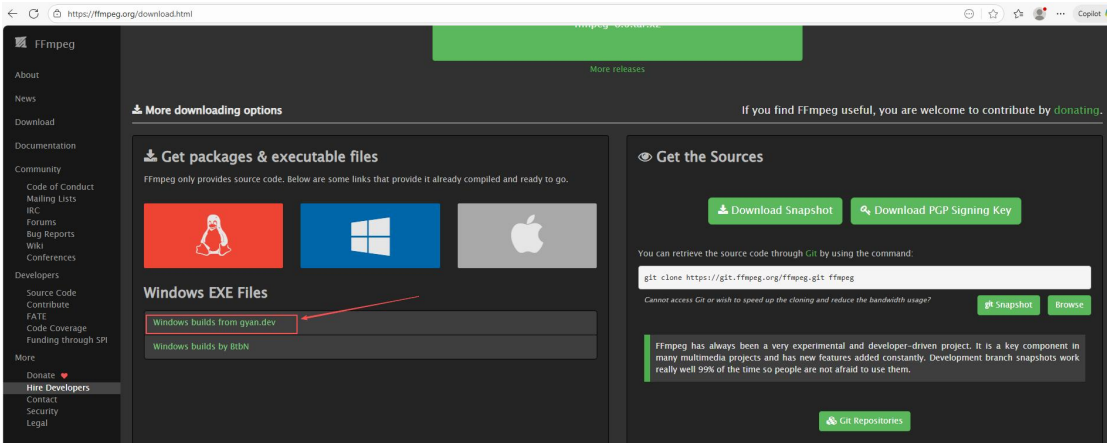
Next, I will show you how to convert an MP3 audio file to a WAV audio file that meets the specifications of 16kHz, 16-bit, and stereo (dual-channel).

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.

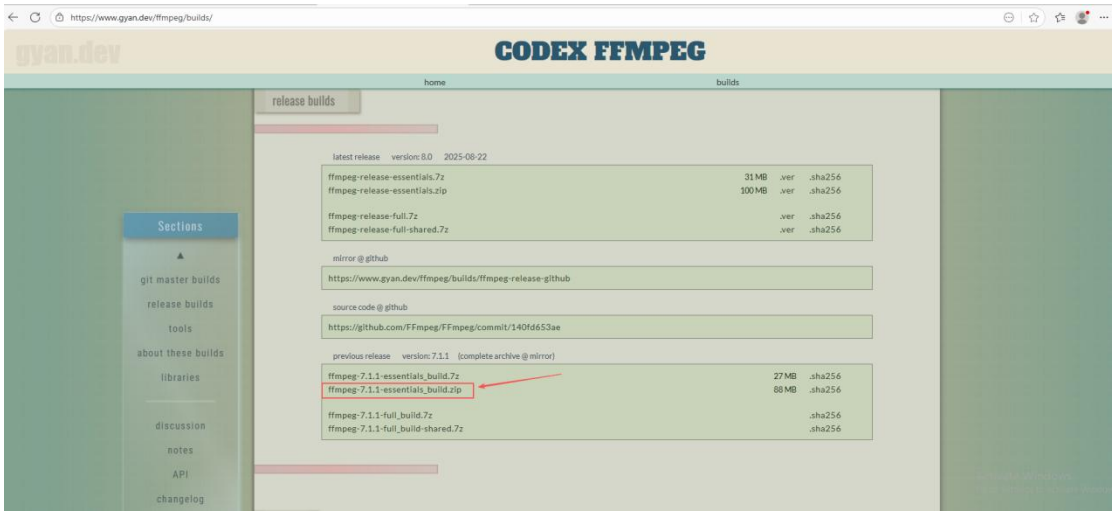
Open the following link to download FFmpeg:
https://ffmpeg.org/download.html



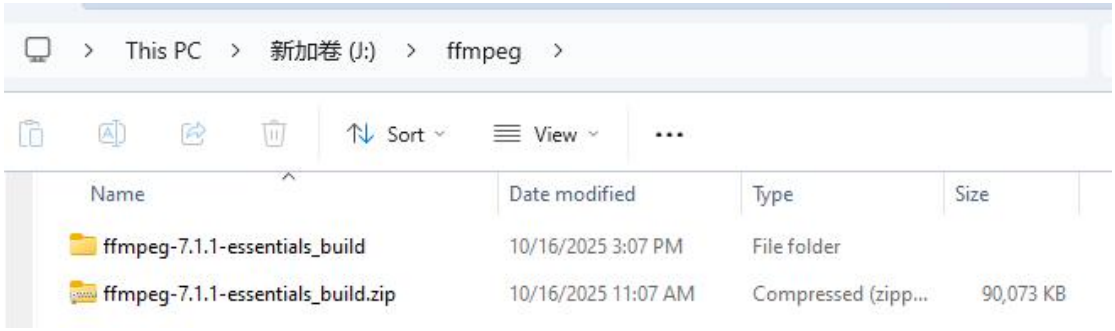
Taking Windows as an Example:
Select the installation package "Windows builds from gyan.dev".



Scroll down to find the "release builds" section, then select "ffmpeg-7.1.1-essentials_build.zip".

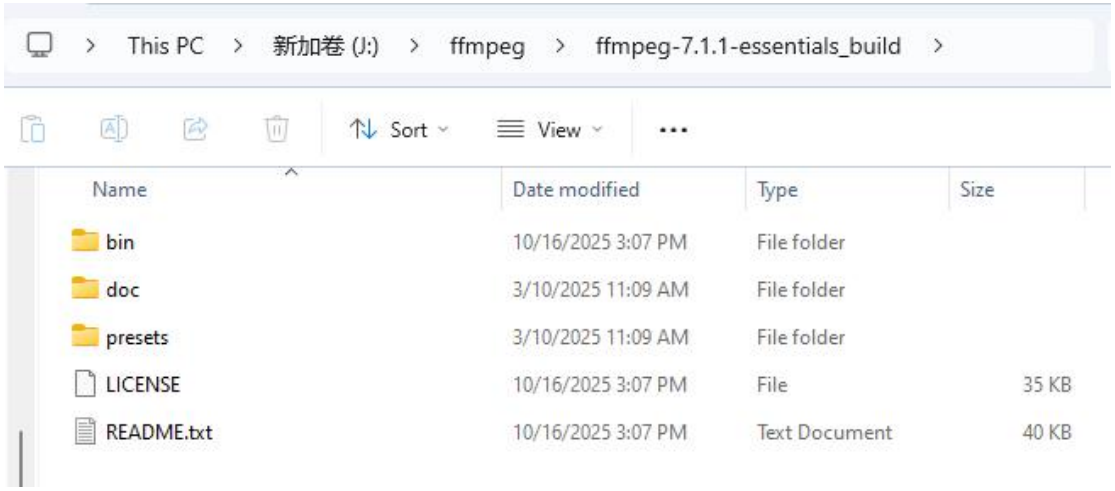


Once the download is complete, extract the file to get the "FFmpeg" folder.



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.



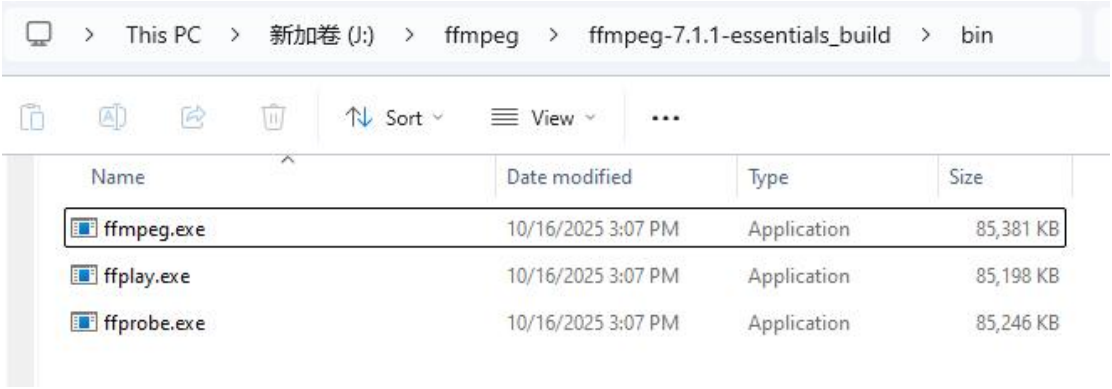
Directory Structure of the Extracted Folder

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.

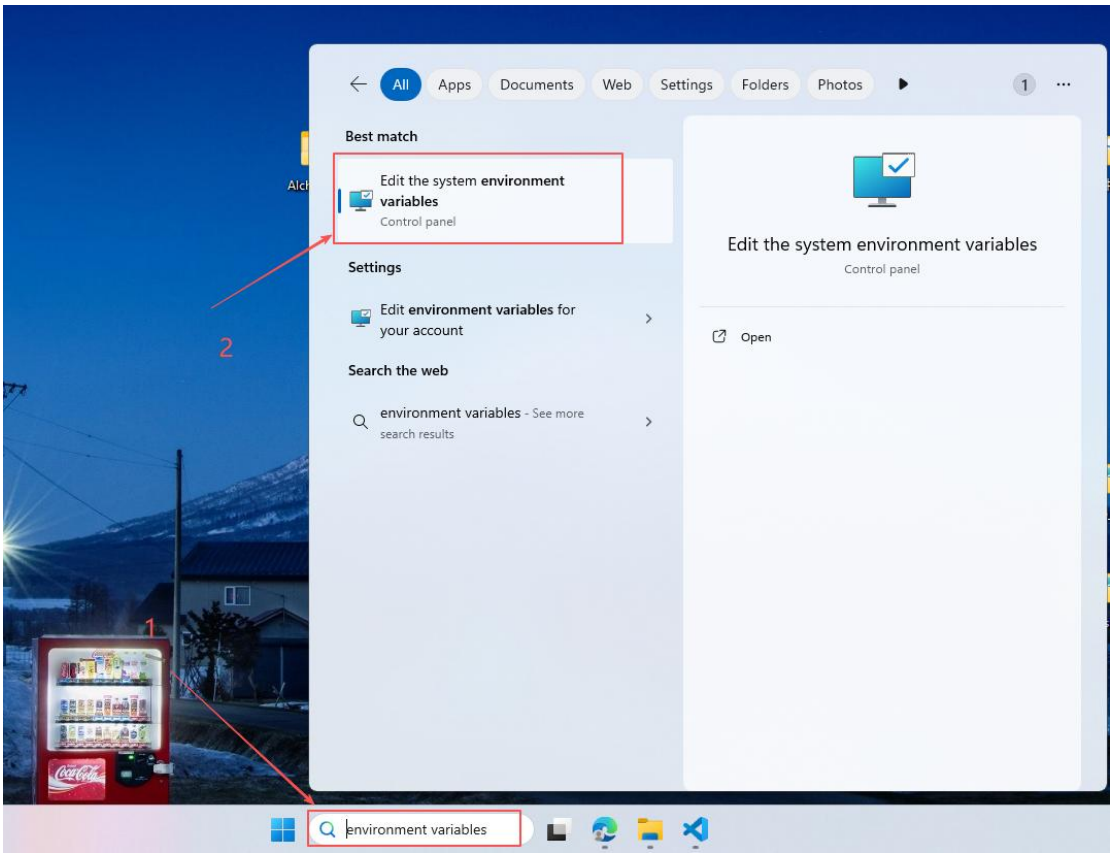
- **"presets"**: Preconfigured formats and encoding schemes.

Navigate to the **"bin"** directory, and you will see three core executable files of FFmpeg: **"ffmpeg.exe"**, **"ffplay.exe"**, and **"ffprobe.exe"**.

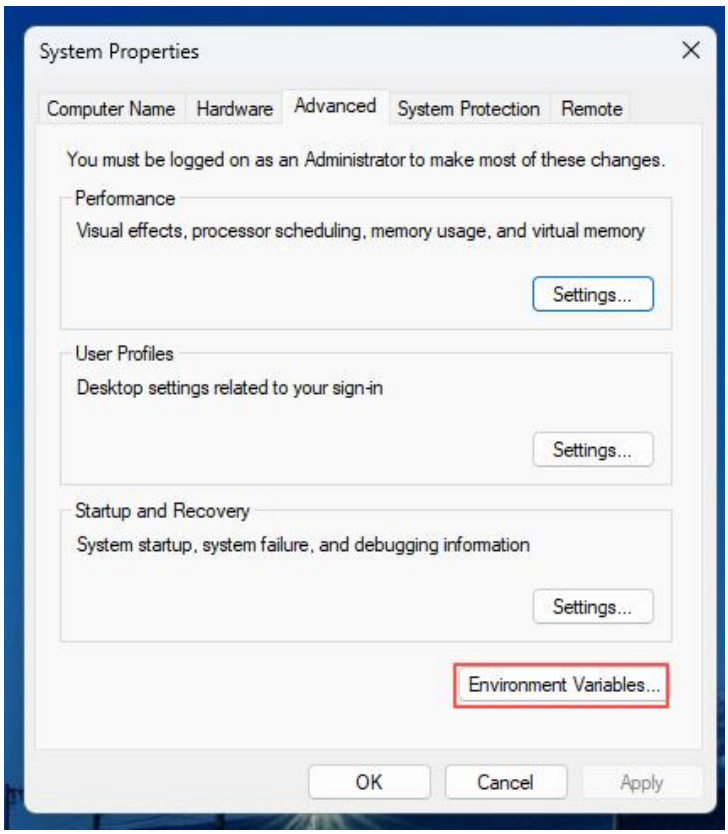


To conveniently call FFmpeg directly in the command line, you need to add it to the system's environment variables.

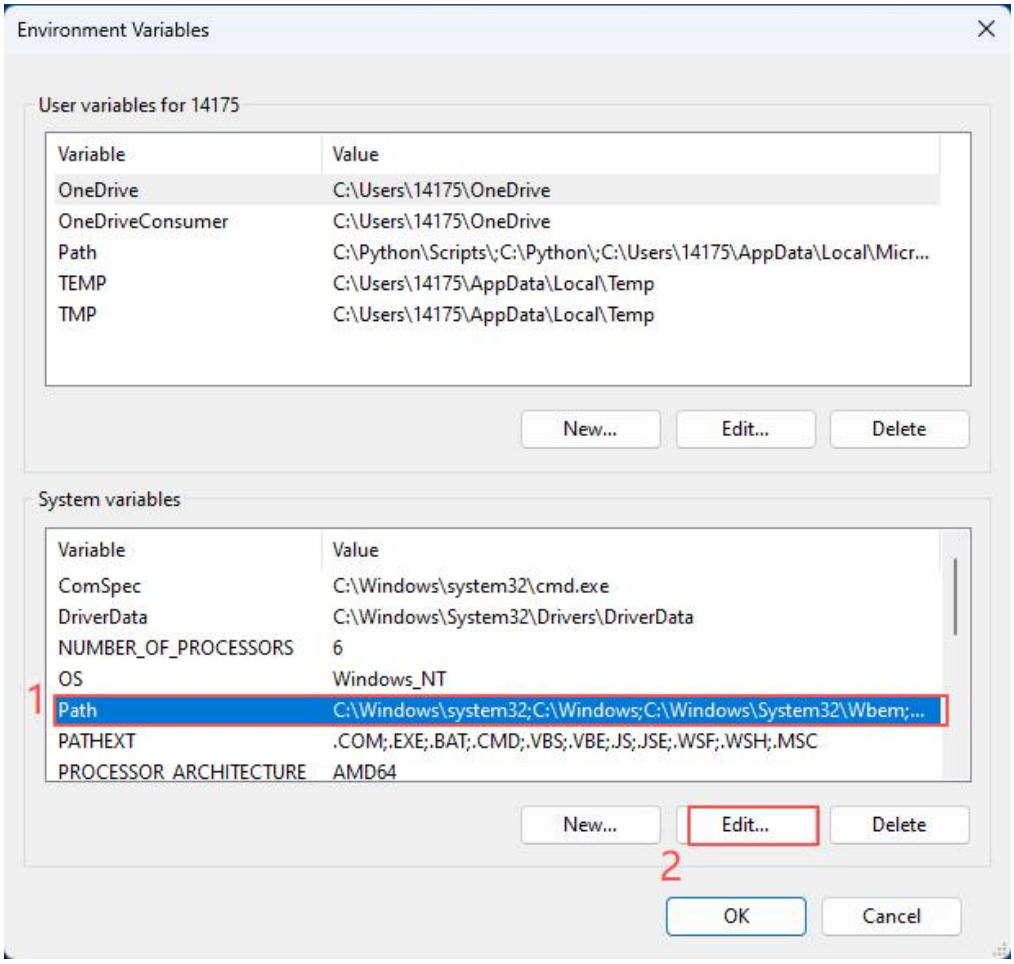
Search for "Environment Variables" in the Start Menu at the bottom left of the desktop, find "Edit the system environment variables", and click to open it.



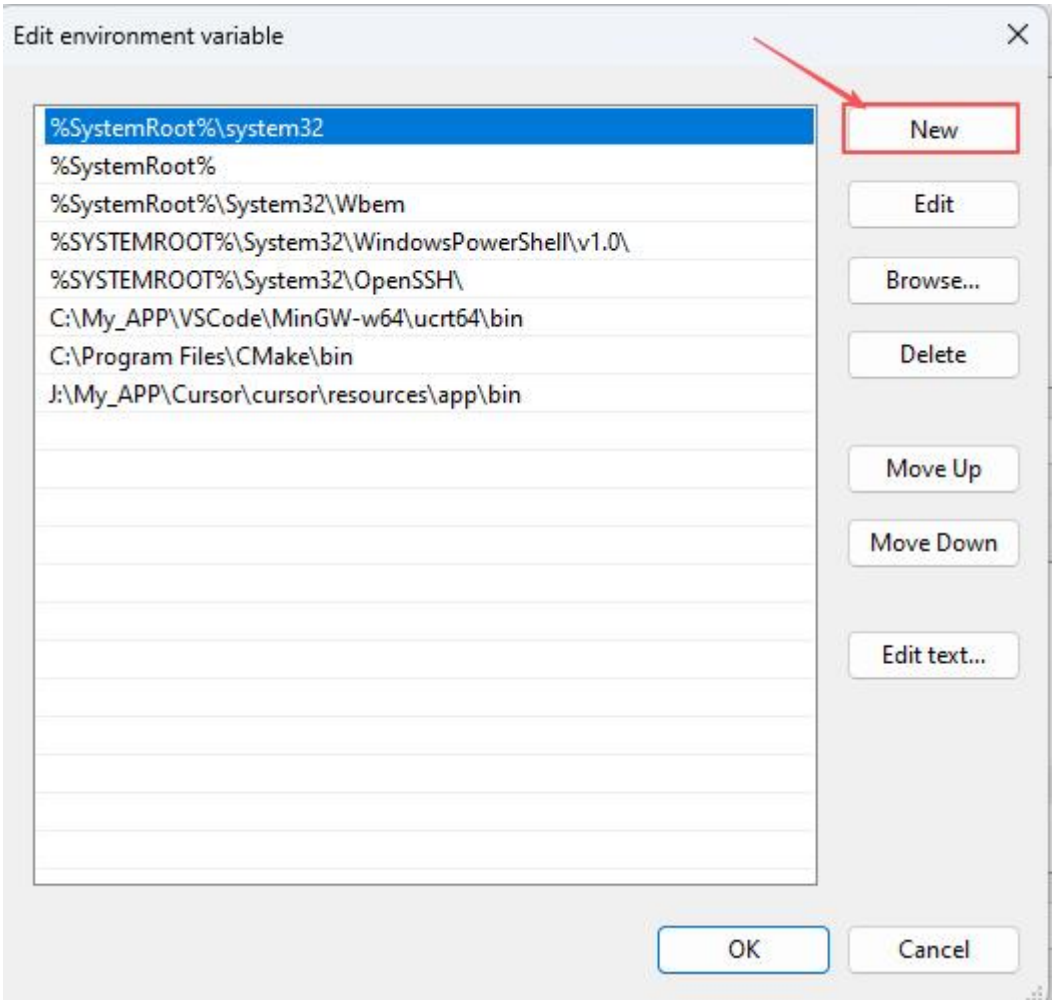
Click the "Environment Variables" button.



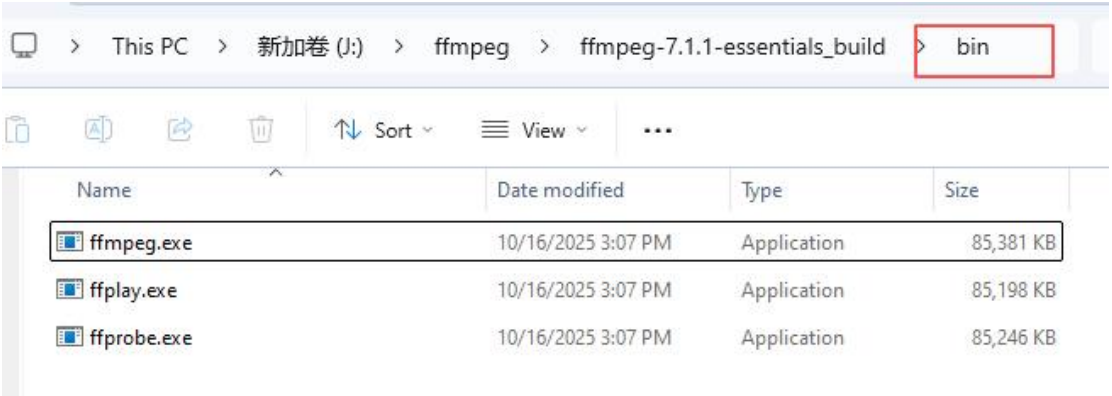
Locate the "Path" entry under "System Variables" and click "Edit".

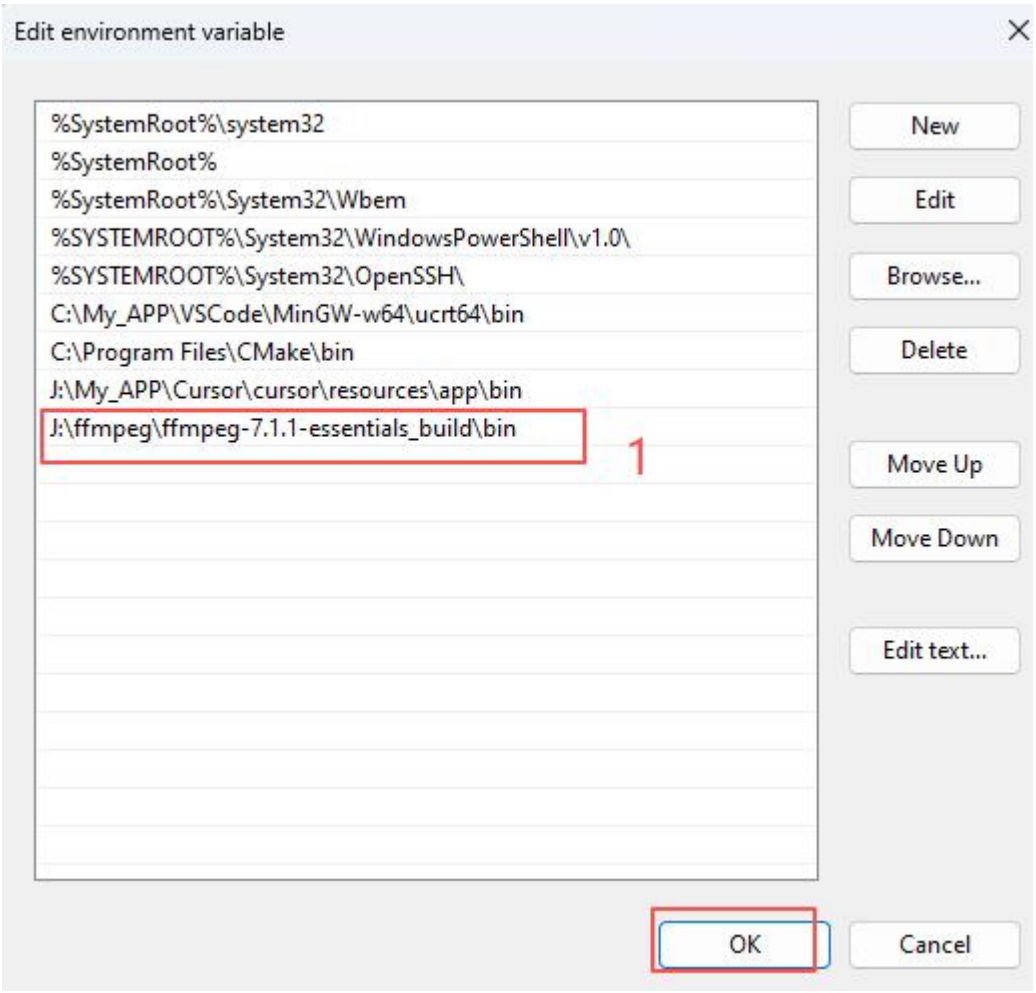


In the "Edit environment variable" window, click "New".



Enter the path to the "bin" folder of FFmpeg (use your own FFmpeg path)



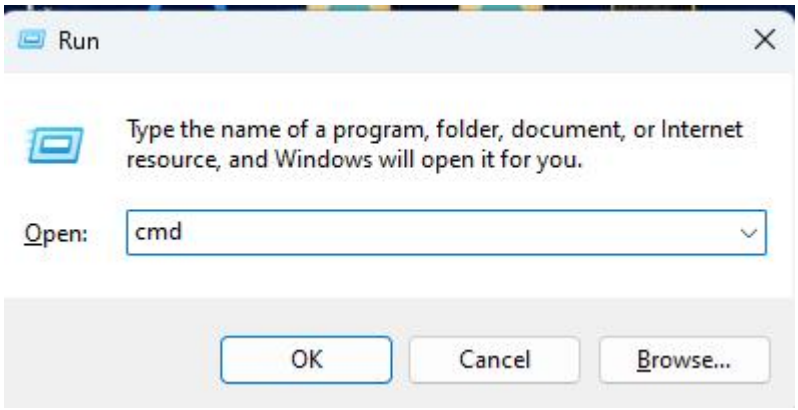


Remember to save the settings after entering the path.

Note: Ensure the path is accurate so the system can correctly locate the FFmpeg files.

Verifying Successful FFmpeg Installation

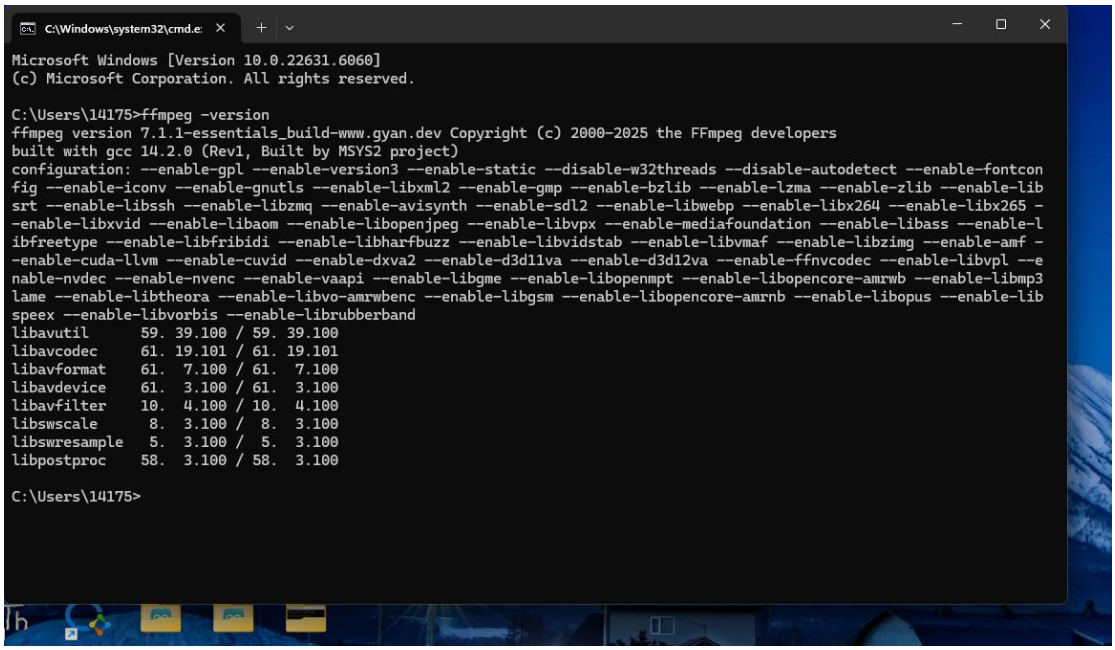
Press the **Win + R** keys, then type "cmd" to open the command line window.



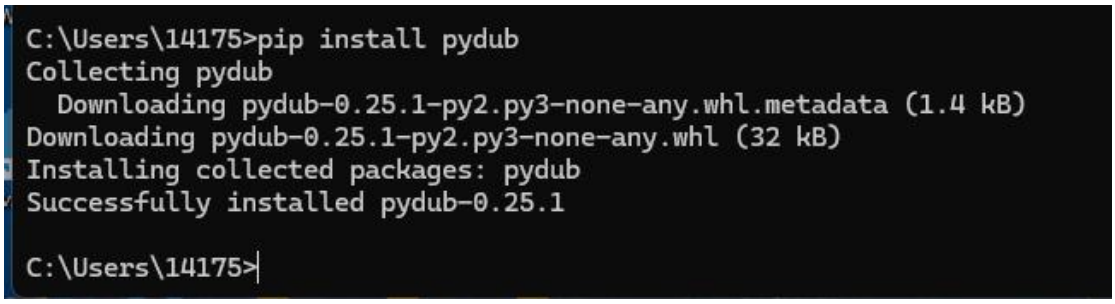
Type the following command in the command line to check the FFmpeg version:

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).



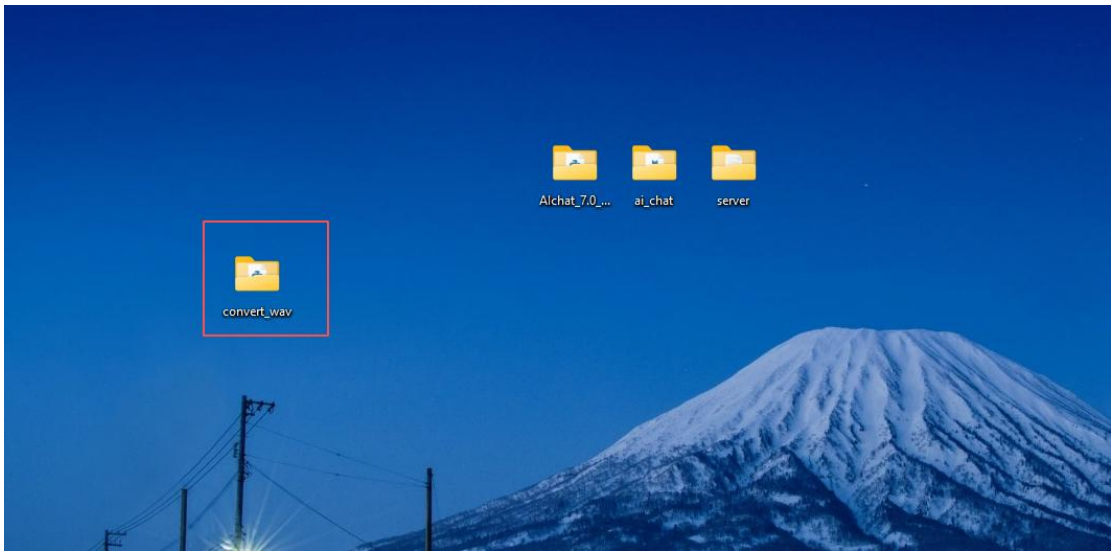
Then, still in the command window, install the dependency by running:
pip install pydub



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-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/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:\ \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;

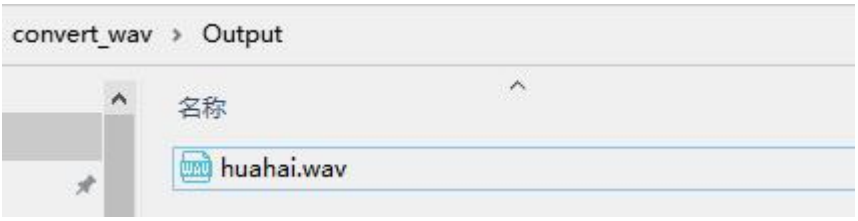
✅ Includes audioop;

✅ Perfectly compatible with most AI, audio, and data analysis libraries.

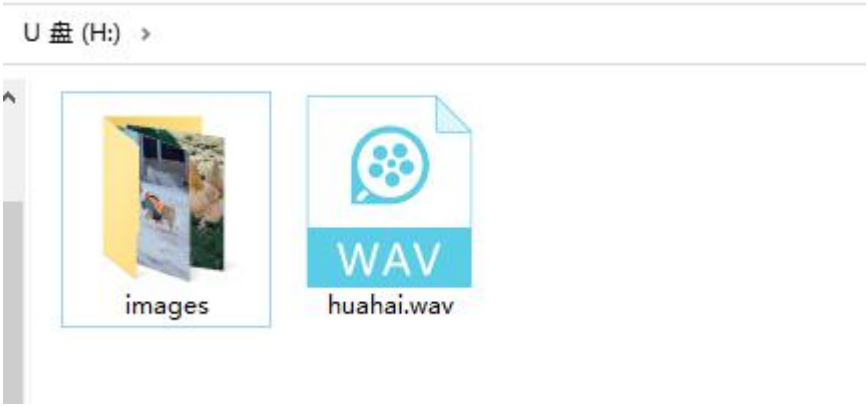
Run our script:

```
C:\                    \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.
```

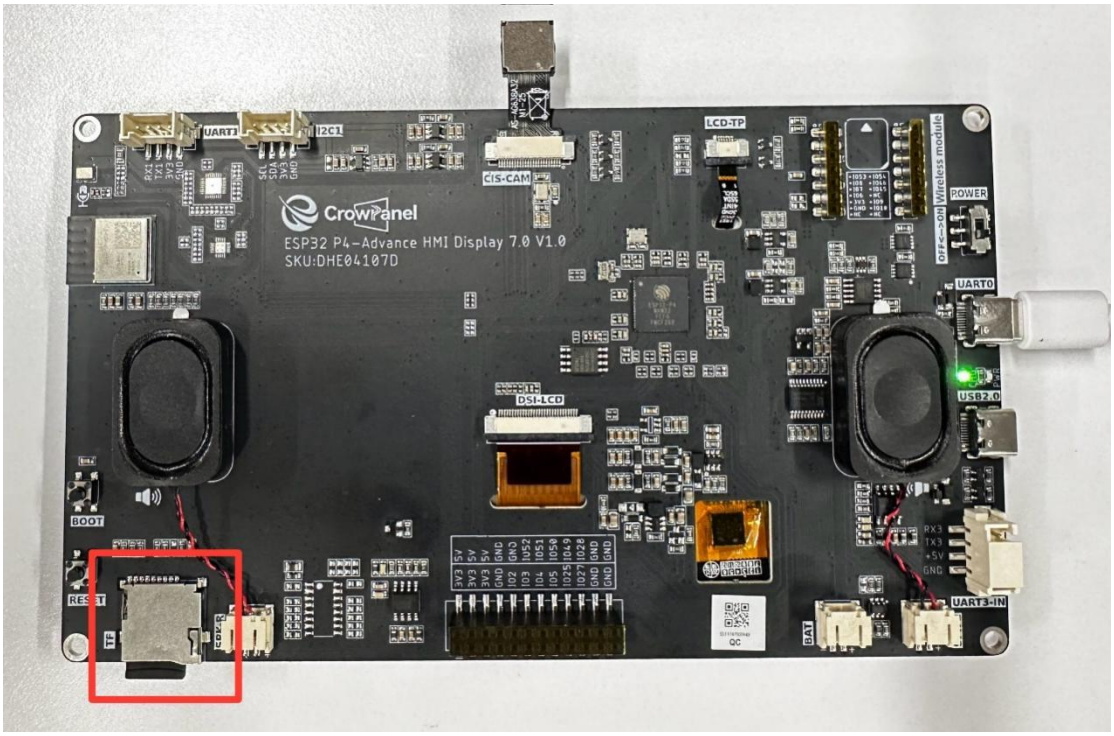
You will find the generated WAV files in the "Output" folder.



Then move this file to a USB flash drive.

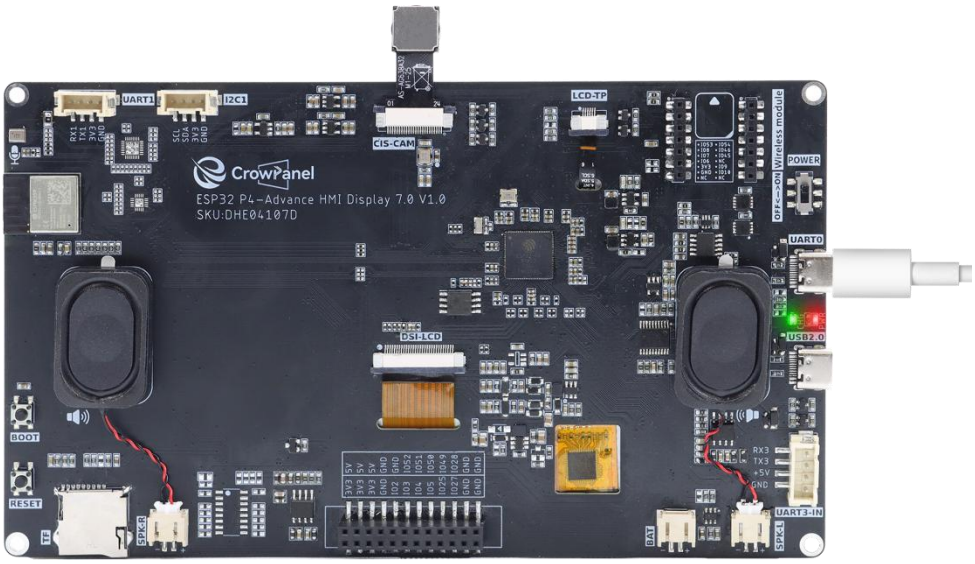


Finally, remove the SD card and insert it into the **Advance-P4** board.

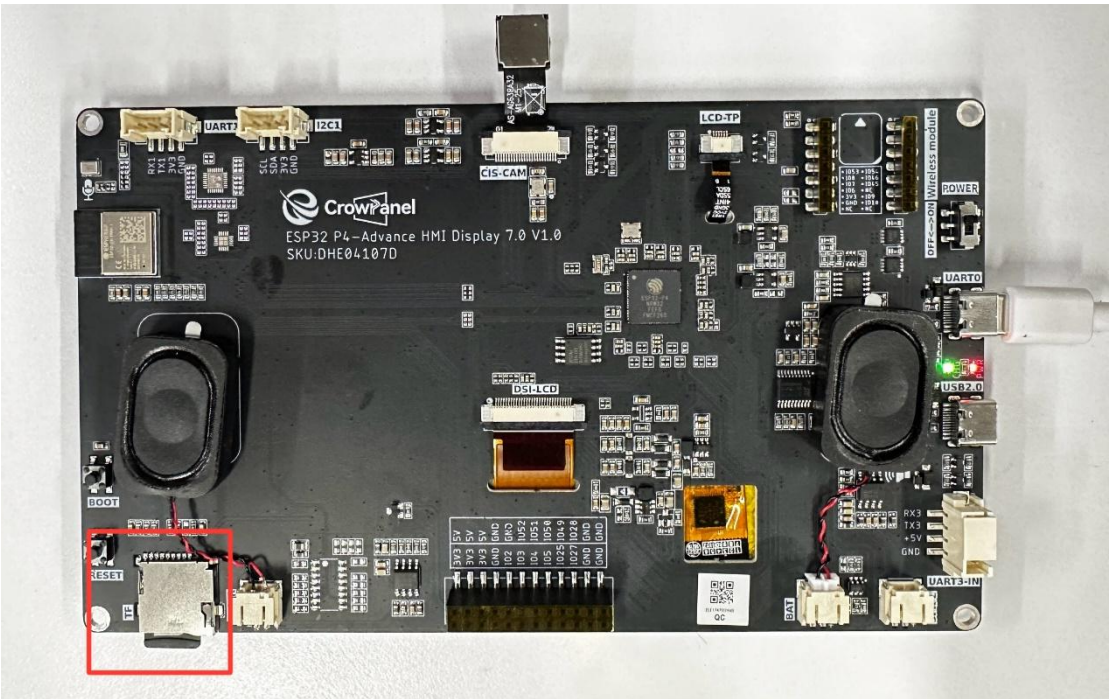


Programming Steps

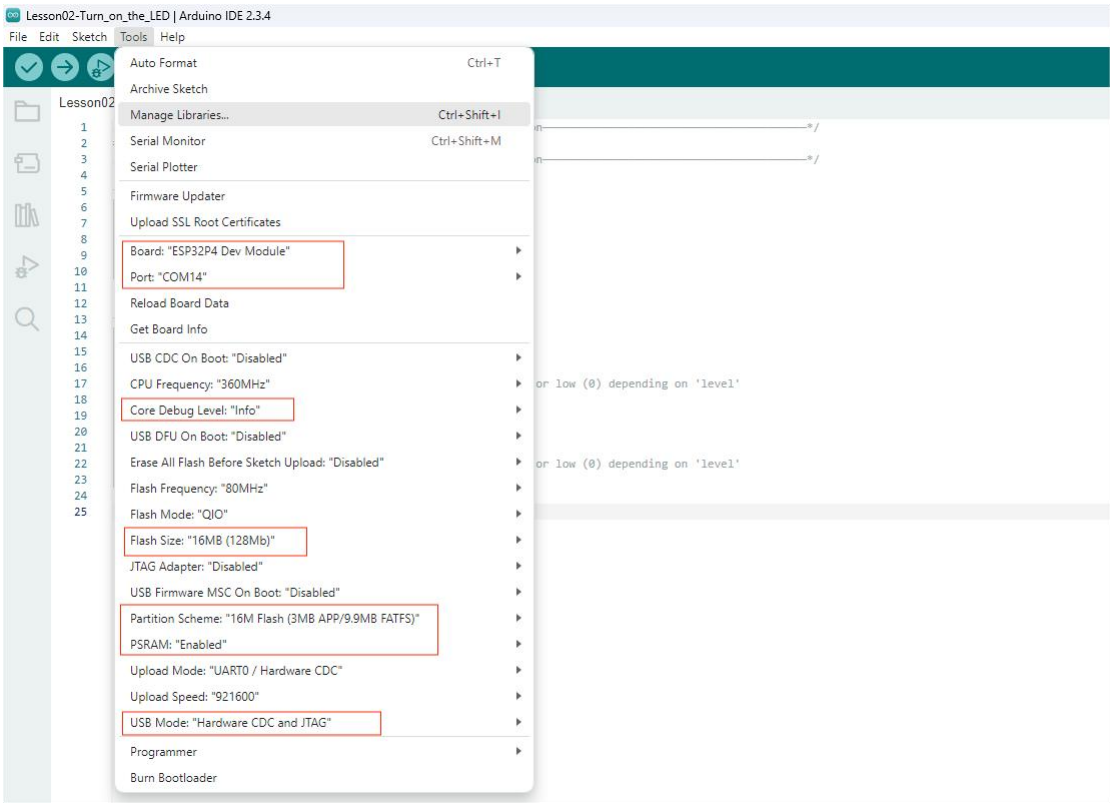
Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
First, we connect the Advance-P4 device to our computer host via the USB cable.



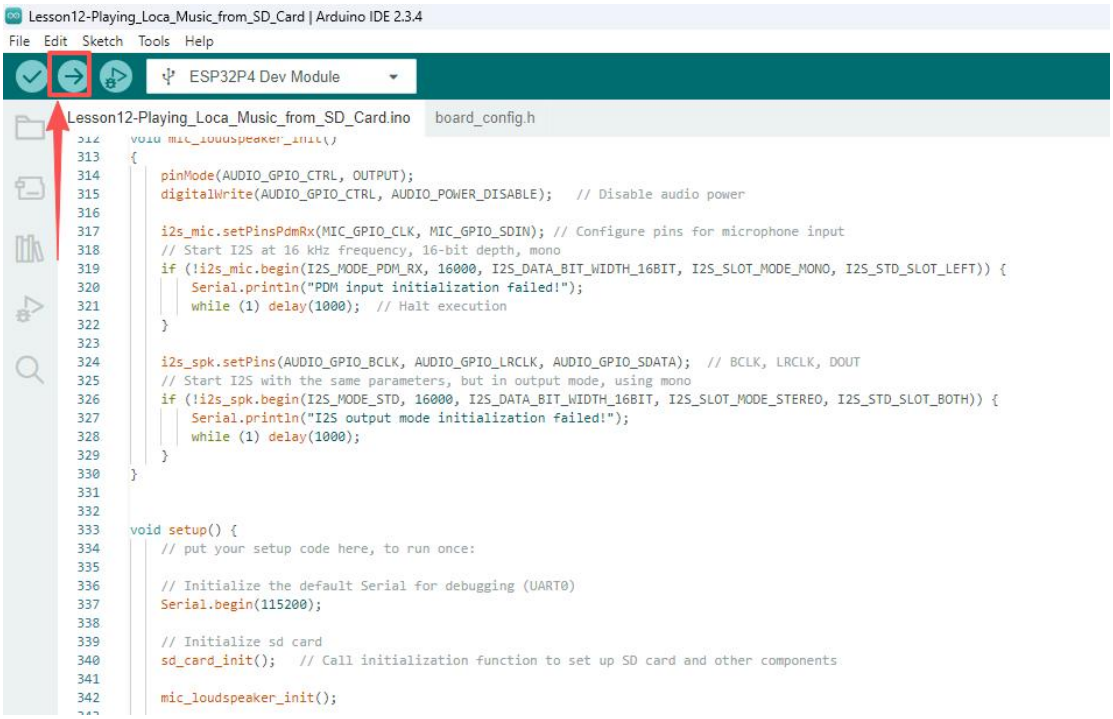
First, double-check two things: whether the converted WAV audio file has been placed in the SD card, and whether the SD card is inserted into the SD card slot of the **Advance-P4**.



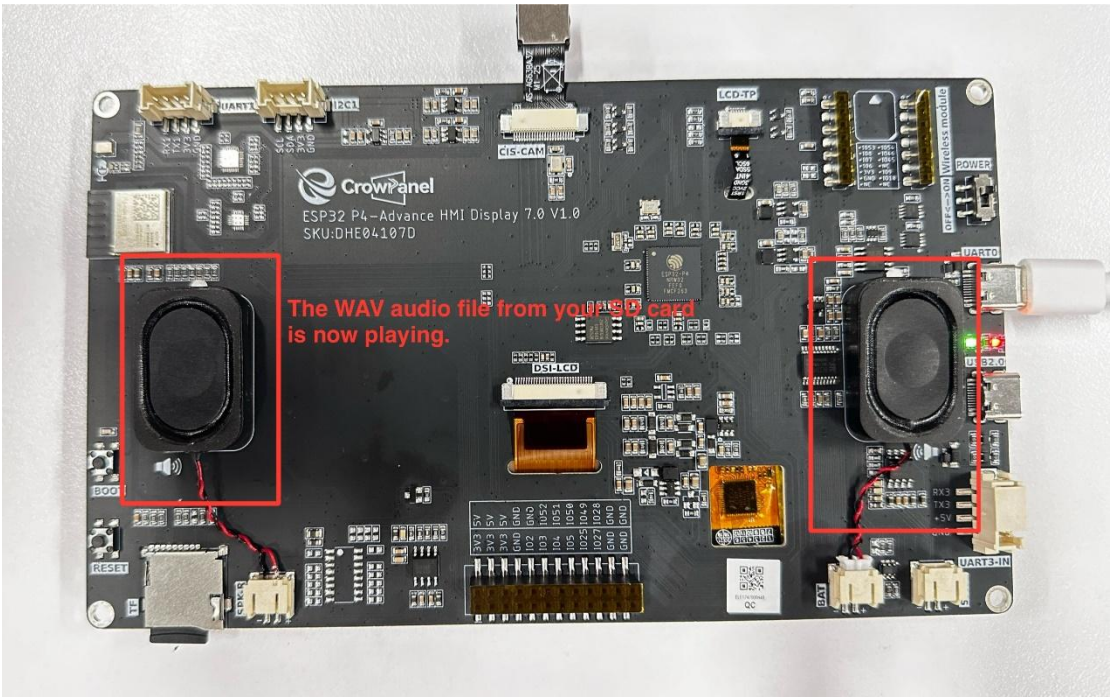
Here, follow the steps from Lesson 1 to first select the development board, COM port, Flash Size, Partition Scheme, PSRAM, and other options.



Then we compile and upload the code.



Once the code runs, you will hear the speaker on the **Advance-P4** playing the WAV audio stored in your SD card.


CrowPanel
ESP32 P4-Advance HMI Display 7.0 V1.0
SKU:DHE04107D
The WAV audio file from your SD card
is now playing.
DSI-LCD
CIS-CAM
LCD-TP
Wireless module
POWER
UART0
UART3-IN
RESET
BOOT

Lesson13--- 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

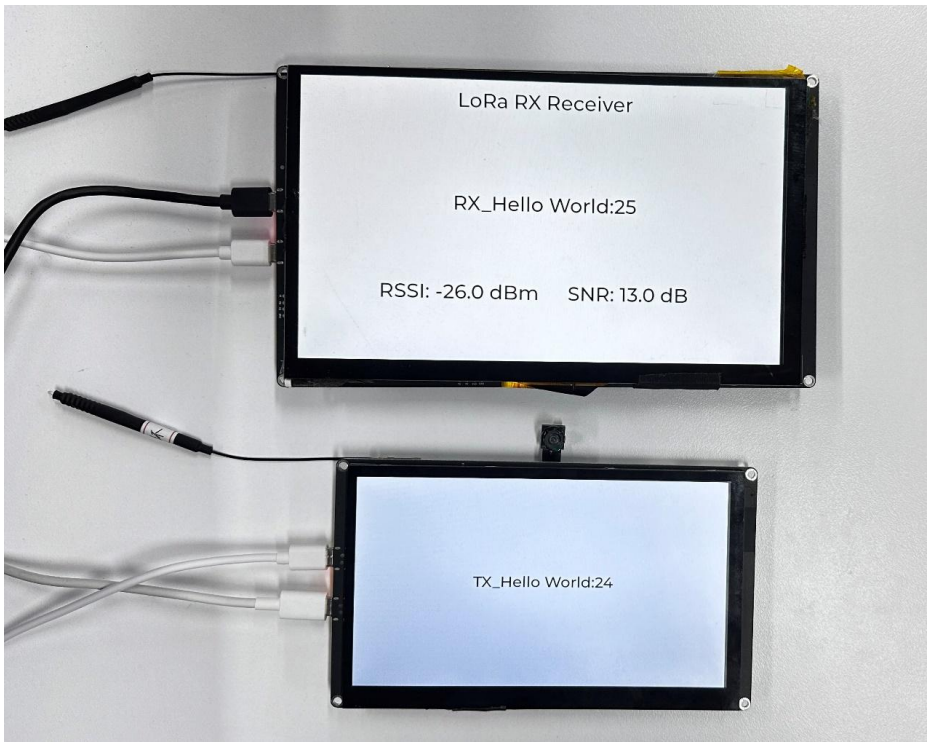
1. Understand the working principles of the SX1262 LoRa module (SPI communication, wireless transceiver mechanism), as well as the hardware pin adaptation rules between ESP32-P4 and the LoRa module (reset, DIO interrupt, SPI chip select, etc.).

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).

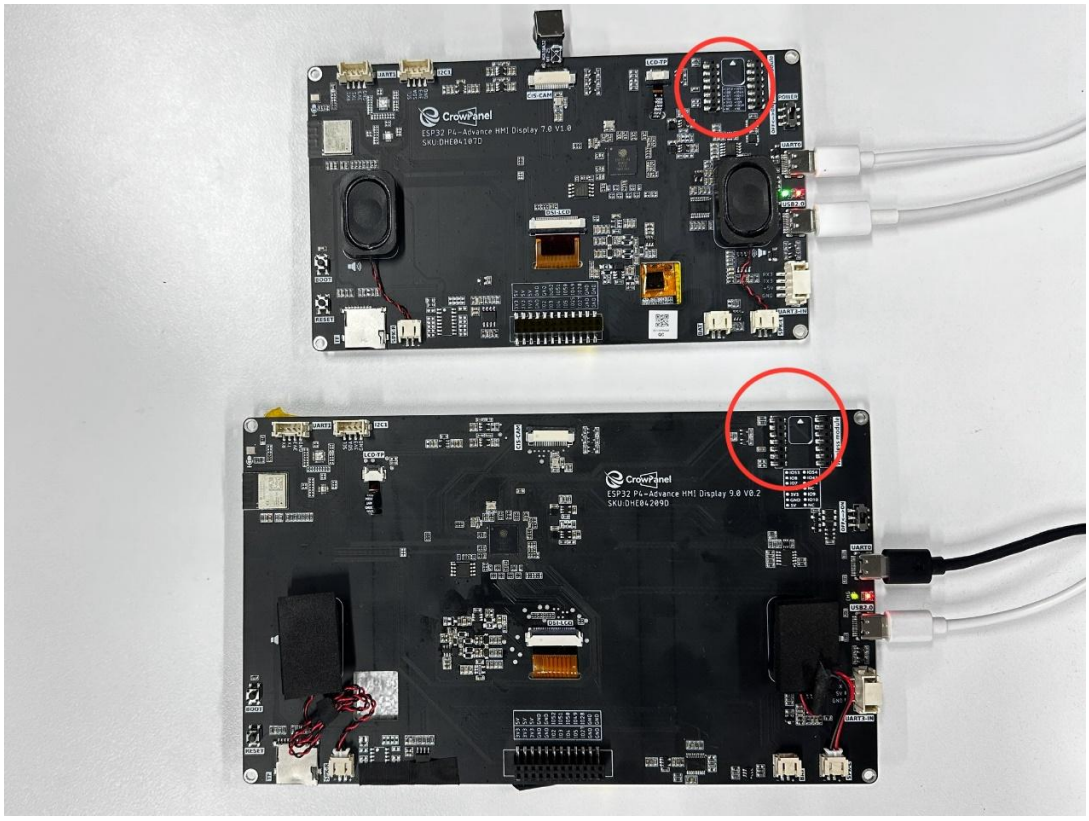
Preview of the Result

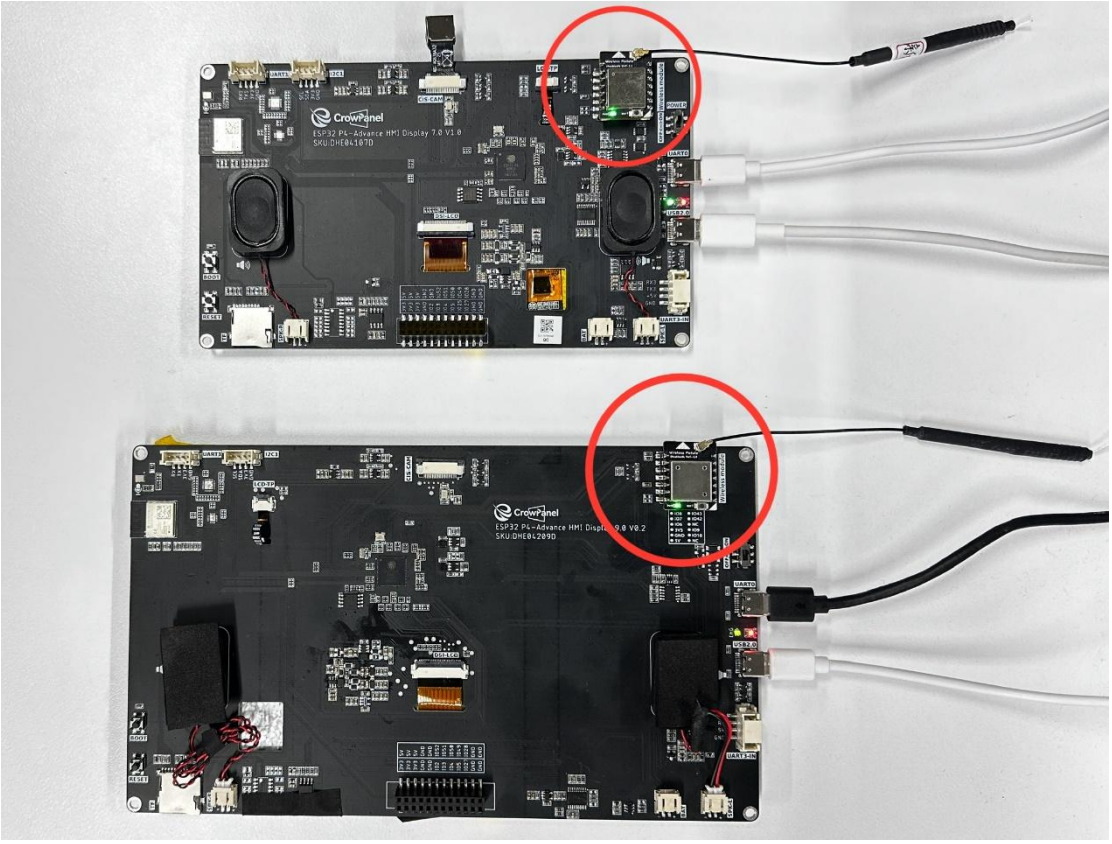
After inserting the SX1262 LoRa modules into both Advance-P4 development boards and running the respective codes, you will observe the following behavior:
On the transmitting Advance-P4 board, the screen will display the message TX_Hello World:i, with the value of i increasing by 1 every second.
Similarly, 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.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)

Kindly click the link below to view the full code implementation.

Transmitting end code：
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/Lesson13_TX_SX1262_Wireless_Module

Receiving end code：
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/Lesson13_RX_SX1262_Wireless_Module

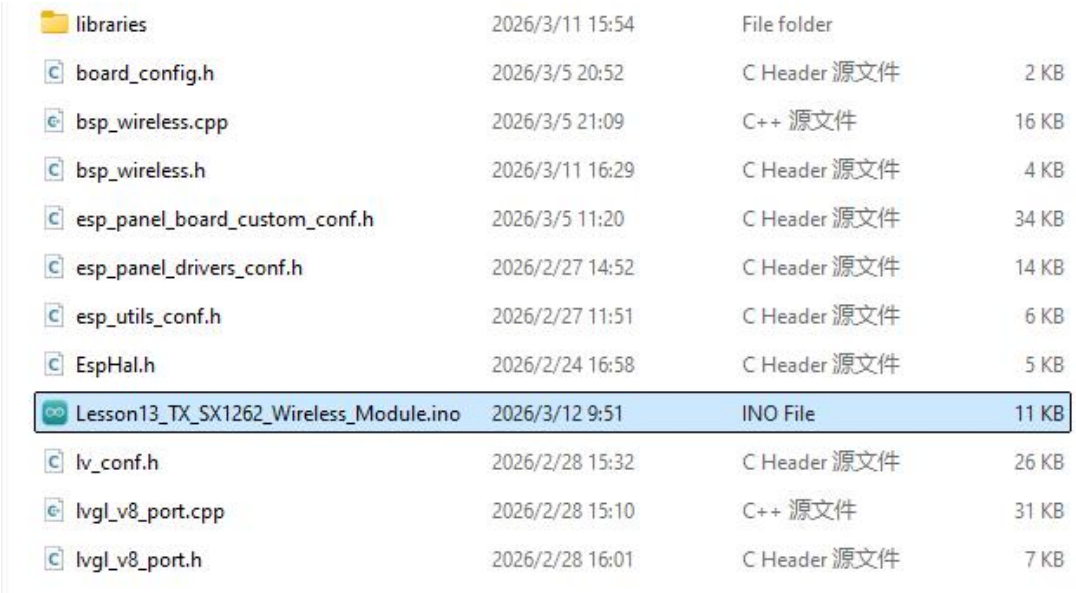
Key Explanations

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).

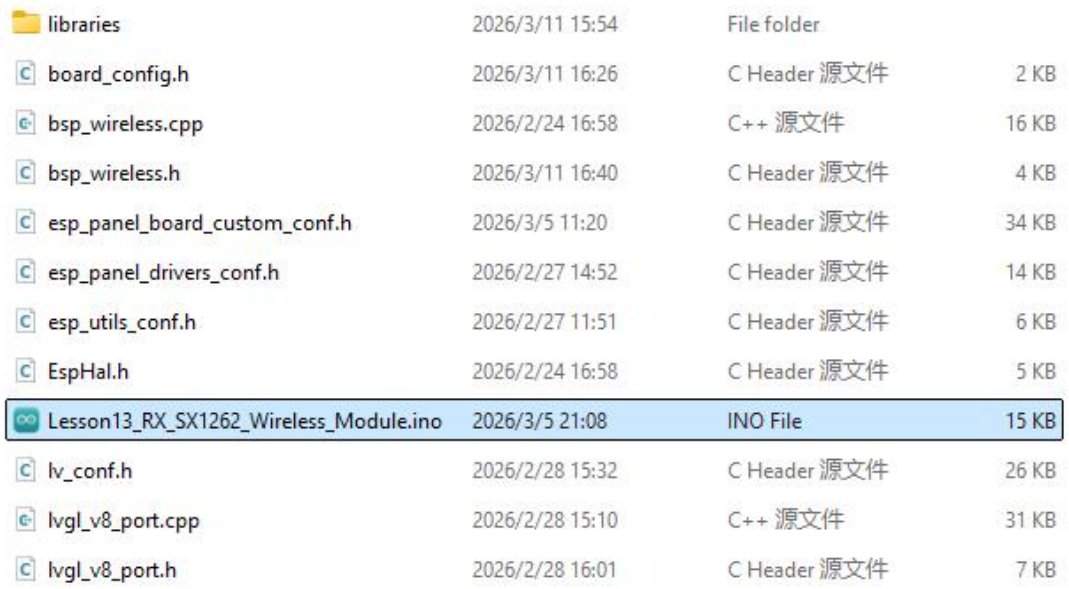
The code for this lesson includes both the transmitter and receiver sides.

Transmitter:



libraries	2026/3/11 15:54	File folder	
board_config.h	2026/3/5 20:52	C Header 源文件	2 KB
bsp_wireless.cpp	2026/3/5 21:09	C++ 源文件	16 KB
bsp_wireless.h	2026/3/11 16:29	C Header 源文件	4 KB
esp_panel_board_custom_conf.h	2026/3/5 11:20	C Header 源文件	34 KB
esp_panel_drivers_conf.h	2026/2/27 14:52	C Header 源文件	14 KB
esp_utils_conf.h	2026/2/27 11:51	C Header 源文件	6 KB
EspHal.h	2026/2/24 16:58	C Header 源文件	5 KB
Lesson13_TX_SX1262_Wireless_Module.ino	2026/3/12 9:51	INO File	11 KB
lv_conf.h	2026/2/28 15:32	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/28 15:10	C++ 源文件	31 KB
lvgl_v8_port.h	2026/2/28 16:01	C Header 源文件	7 KB

Receiver:



libraries	2026/3/11 15:54	File folder	
board_config.h	2026/3/11 16:26	C Header 源文件	2 KB
bsp_wireless.cpp	2026/2/24 16:58	C++ 源文件	16 KB
bsp_wireless.h	2026/3/11 16:40	C Header 源文件	4 KB
esp_panel_board_custom_conf.h	2026/3/5 11:20	C Header 源文件	34 KB
esp_panel_drivers_conf.h	2026/2/27 14:52	C Header 源文件	14 KB
esp_utils_conf.h	2026/2/27 11:51	C Header 源文件	6 KB
EspHal.h	2026/2/24 16:58	C Header 源文件	5 KB
Lesson13_RX_SX1262_Wireless_Module.ino	2026/3/5 21:08	INO File	15 KB
lv_conf.h	2026/2/28 15:32	C Header 源文件	26 KB
lvgl_v8_port.cpp	2026/2/28 15:10	C++ 源文件	31 KB
lvgl_v8_port.h	2026/2/28 16:01	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:

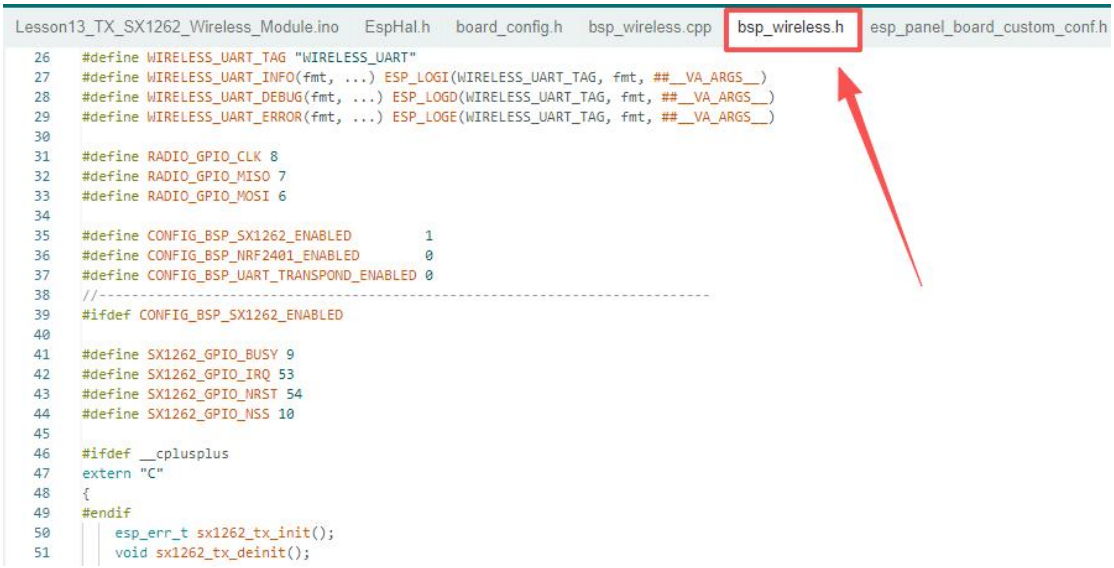
- It encodes and modulates the data (such as strings or sensor information) sent from the main controller and transmits it wirelessly.
- It also receives wireless data packets sent from other devices via LoRa.
- Through a callback mechanism, it passes the received data back to the upper-layer application.

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.



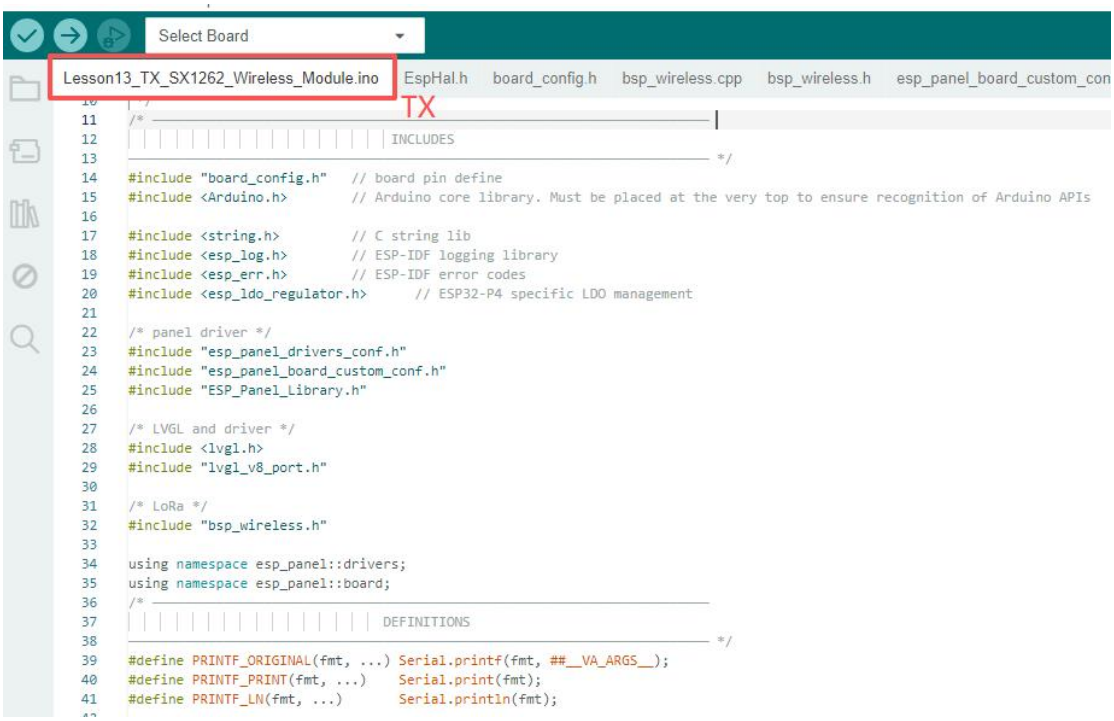
Here, enable the macro definition for the SX1262 wireless module by setting it to 1. The code you will subsequently use will be SX1262-related, while other wireless modules will remain commented out by default—that is, we set other wireless modules to 0.

(Enable the one that corresponds to the wireless module you are using.)

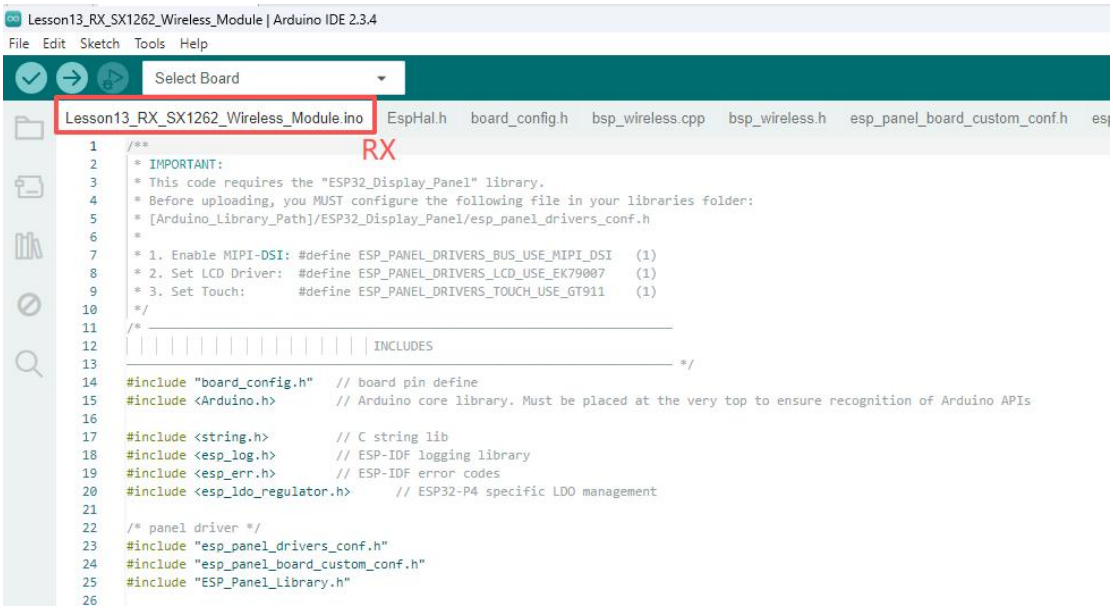
As shown in the figure, we have enabled the SX1262-related configuration, so the other wireless modules are currently disabled and not in use.
Within the bsp_wireless component, you only need to know when to call the provided interfaces that we have written.

Next, let's focus on understanding the bsp_wireless component itself.

The following section shows the transmitter (TX) side of the project:



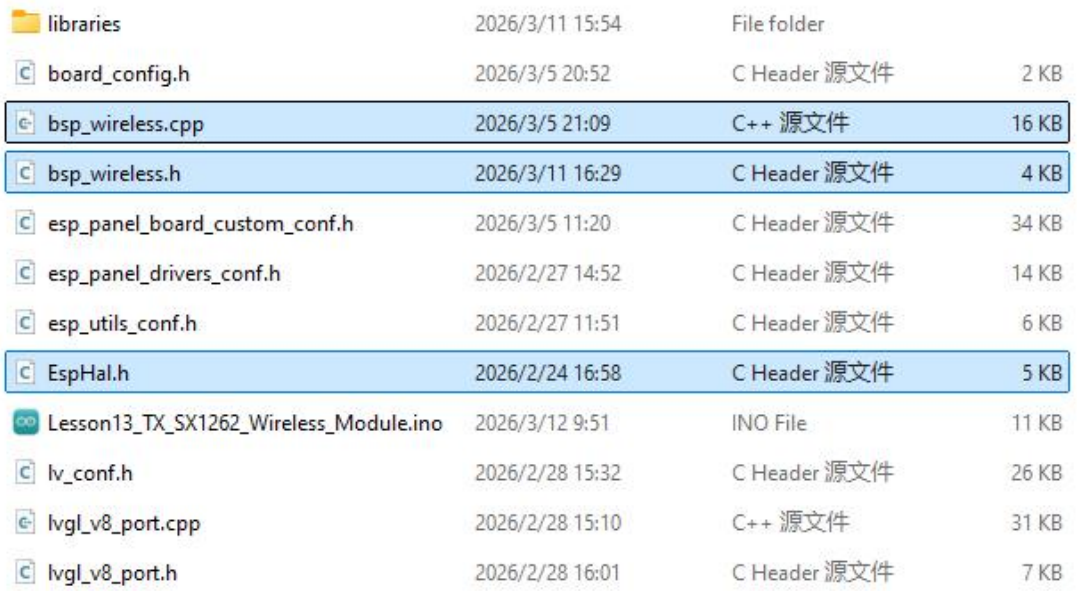
The following section shows the receiver (RX) side of the project:



In these two projects, the only difference lies in the main functions: Lesson13_TX_SX1262_Wireless_Module.ino for the transmitter and Lesson13_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.)

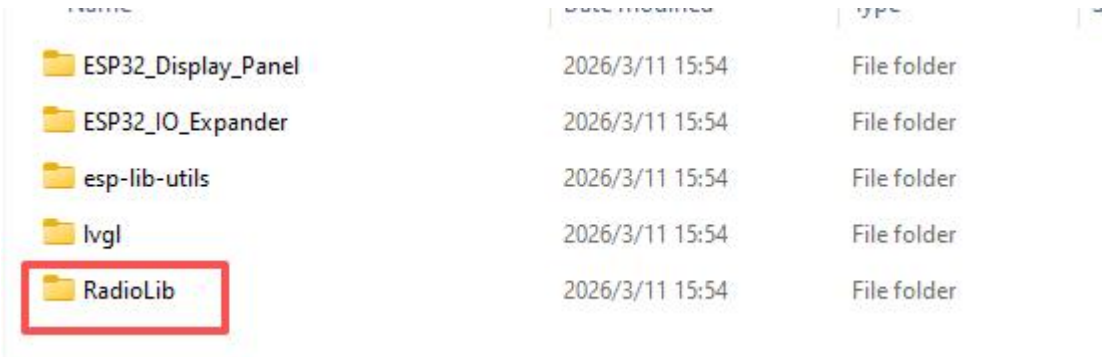
You can see in this folder there are "bsp_wireless.h", "bsp_wireless.cpp", and "EspHal.h".



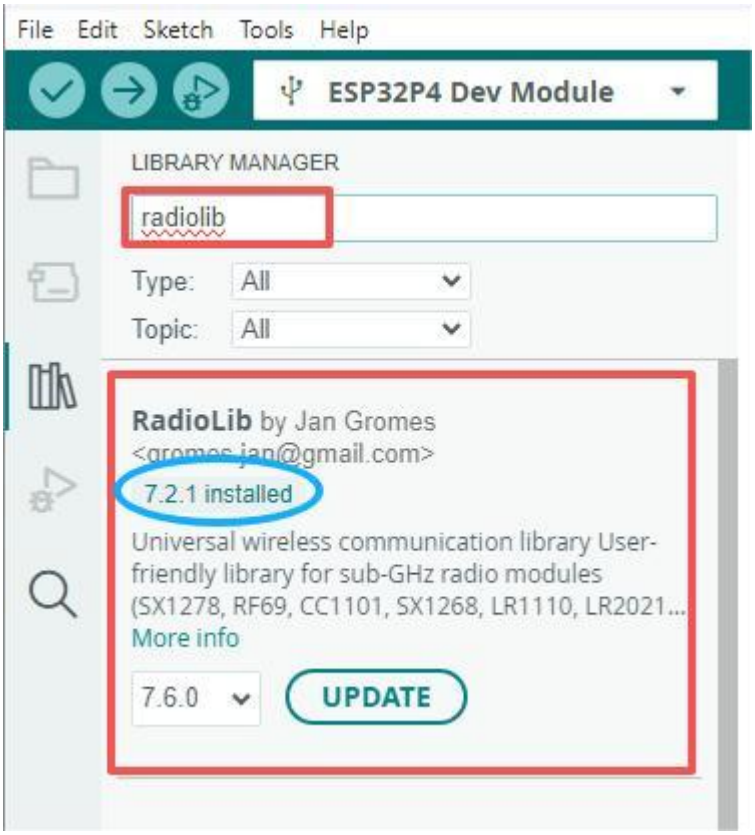
The purpose of EspHal.h is to convert C code from ESP-IDF into the Arduino-style C++ code required by the RadioLib component library.

And RadioLib is already prepared in the libraries folder.

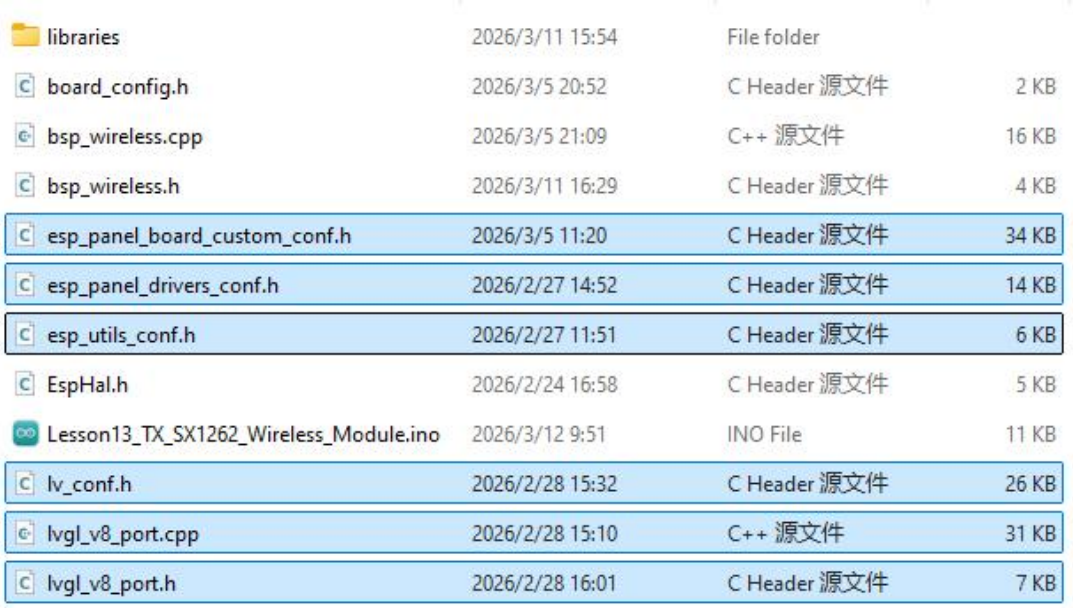
Since this is an official library, we need to rely on it to implement the SX1262 LoRa wireless transmission or reception functionality on our Advance-P4.



You can also download version 7.2.1 of the RadioLib library in the Arduino IDE.



The other code files are display-related driver code that was covered in detail in Lesson 7.



Looking at the board_config.h file:

This code serves as the board-level hardware configuration file board_config.h in the project. Its primary role is to centrally define the GPIO pin numbers used by various peripherals on the development board, along with certain hardware parameters—thereby providing unified hardware interface configuration throughout the entire project. This prevents repetition of pin numbers across different source files, enhancing code readability, maintainability, and portability.

Lesson13_RX_SX1262_Wireless_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h | esp_pa

```
#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)


/* SPI BUS */
#define PIN_SPI_SCK         ( 8)
#define PIN_SPI_MOSI        ( 6)
#define PIN_SPI_MISO        ( 7)
/* LoRa Interfaces */
#define LORA_RESET          (54)    // RST
#define LORA_DIO1           (53)    // IRQ
#define LORA_DIO2           ( 9)    // BUSY
#define LORA_CS             (10)
#define LORA_SCK            PIN_SPI_SCK
#define LORA_MOSI           PIN_SPI_MOSI
#define LORA_MISO           PIN_SPI_MISO
#define LORA_DIO3_TCXO_VOLTAGE  (3.3)
/********************** Pin define **********************/

```

The file begins with the "#pragma once" directive, indicating this header file will only be included once during compilation—preventing duplicate definition issues caused by multiple inclusions.

The code then categorizes pin definitions by functional module:

First, the GT911 touchscreen controller pin definitions: "Touch_GPIO_RST" is defined as 40 for touch chip reset control; "Touch_GPIO_INT" is defined as 42 for touch interrupt signal output, triggered when the screen is touched to notify the main controller.

Next, the I2C bus pin definitions: "I2C_GPIO_SCL" is 46 for the I2C clock line (SCL), and "I2C_GPIO_SDA" is 45 for the I2C data line (SDA). This I2C bus is primarily used for communication with the touch control chip GT911.

Then, LCD display backlight control configurations: including "LCD_GPIO_BLIGHT"

(GPIO31) for controlling screen backlight brightness, "BLIGHT_PWM_Hz" set to 30000 indicating a backlight PWM dimming frequency of 30 kHz, and "BLIGHT_ON_LEVEL" indicating the active level for backlight (1 meaning high level turns on backlight).

The LCD display reset pin "LCD_GPIO_RST" (GPIO41) is then defined, used for hardware reset of the display during system startup.

Following this, a set of LCD panel timing parameters and resolution parameters are defined: for example, "V_size" of 600 indicates vertical resolution of 600 pixels, "H_size" of 1024 indicates horizontal resolution of 1024 pixels. Additionally, LCD pixel clock "LCD_CLK_MHZ" of 51 MHz is defined, along with horizontal sync pulse width "LCD_HPW", horizontal back porch "LCD_HBP", horizontal front porch "LCD_HFP", and vertical sync pulse width "LCD_VPW", vertical back porch "LCD_VBP", vertical front porch "LCD_VFP" and other display timing parameters. These parameters are used to correctly generate display synchronization signals when driving RGB or parallel interface LCDs.

SPI bus pins are then defined, including "PIN_SPI_SCK" (clock), "PIN_SPI_MOSI" (Master Out Slave In), and "PIN_SPI_MISO" (Master In Slave Out). These SPI pins will be shared among other peripherals.

The final section defines the LoRa wireless communication module interface: for example, "LORA_RESET" for module reset, "LORA_DIO1" and "LORA_DIO2" for interrupt and status signals, "LORA_CS" for SPI chip select signal. Meanwhile, the LoRa module's SPI clock, MOSI, and MISO directly reuse the SPI bus pins defined earlier, with "LORA_DIO3_TCXO_VOLTAGE" set to 3.3V indicating the supply voltage for the module's internal TCXO clock.

Now that the entire project framework is clear, let's examine the "bsp_wireless" component in detail—that is, "bsp_wireless.h" and "bsp_wireless.cpp".

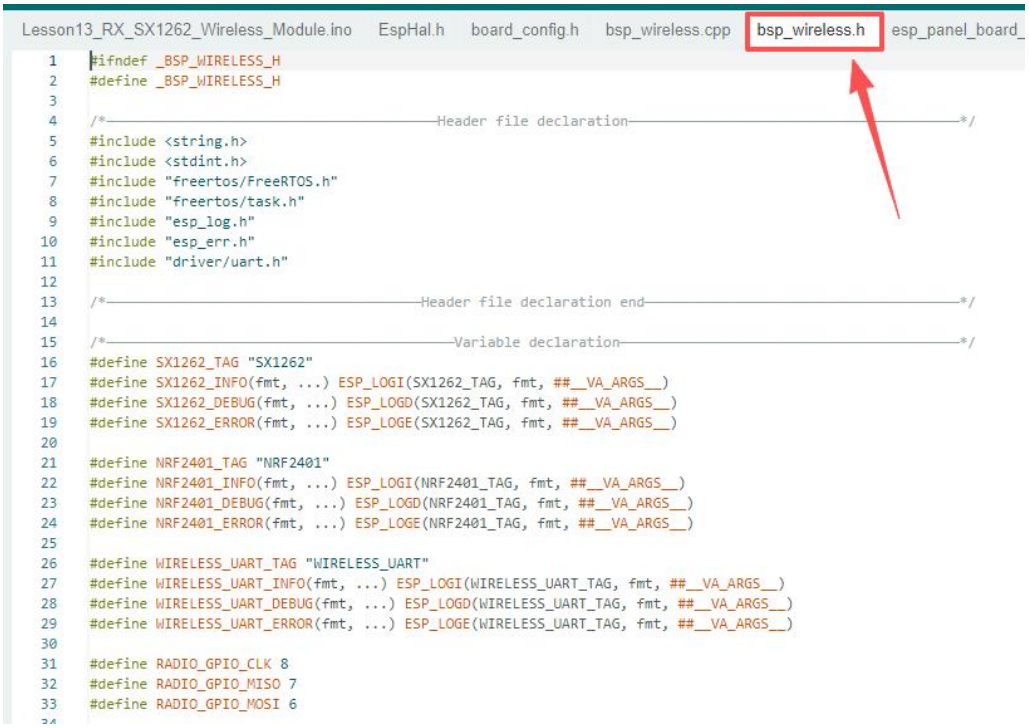
SX1262 LoRa Code

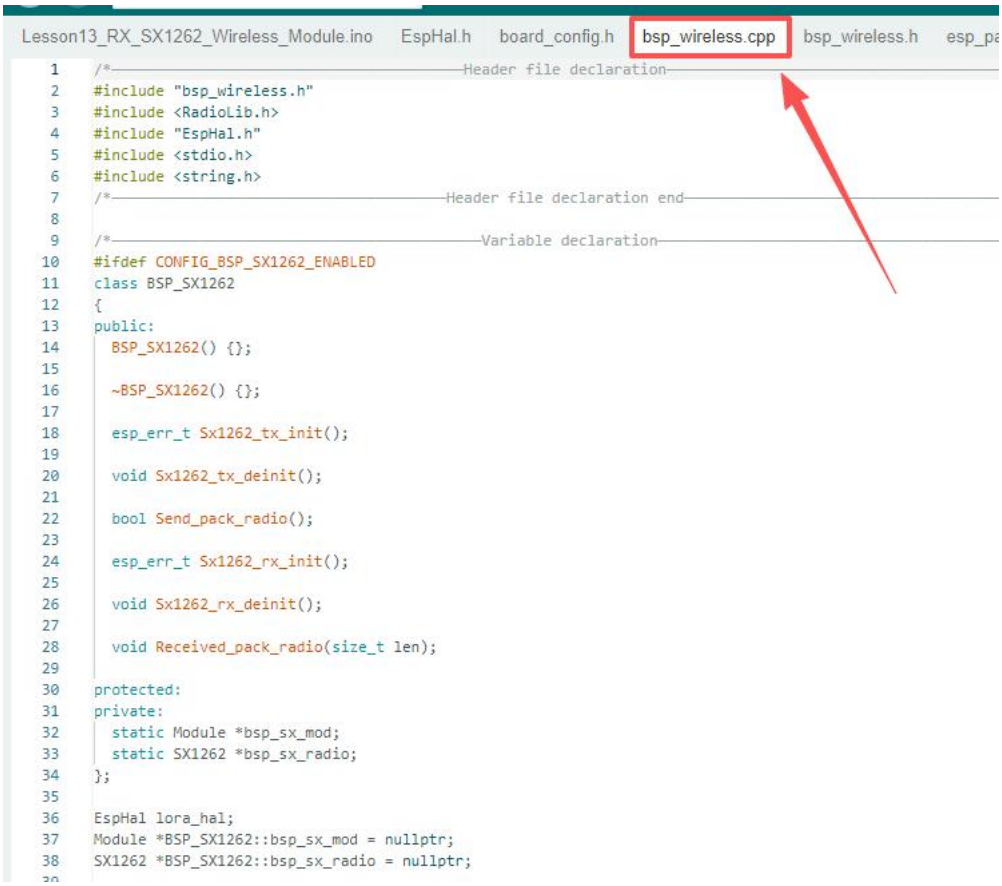
The SX1262 LoRa transmission and reception code consists of two files: bsp_wireless.cpp and bsp_wireless.h.

Next, we will first analyze the SX1262-related code in bsp_wireless.h.

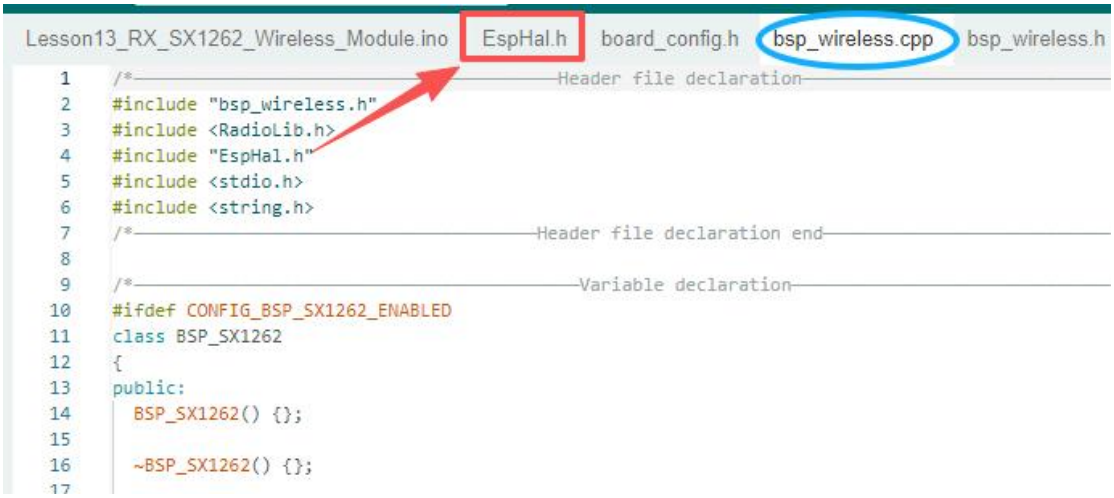
bsp_wireless.h is the header file for the SX1262 LoRa wireless module.
Its main purposes are:
To declare the functions, macros, and variables implemented in bsp_wireless.cpp for external use.

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.

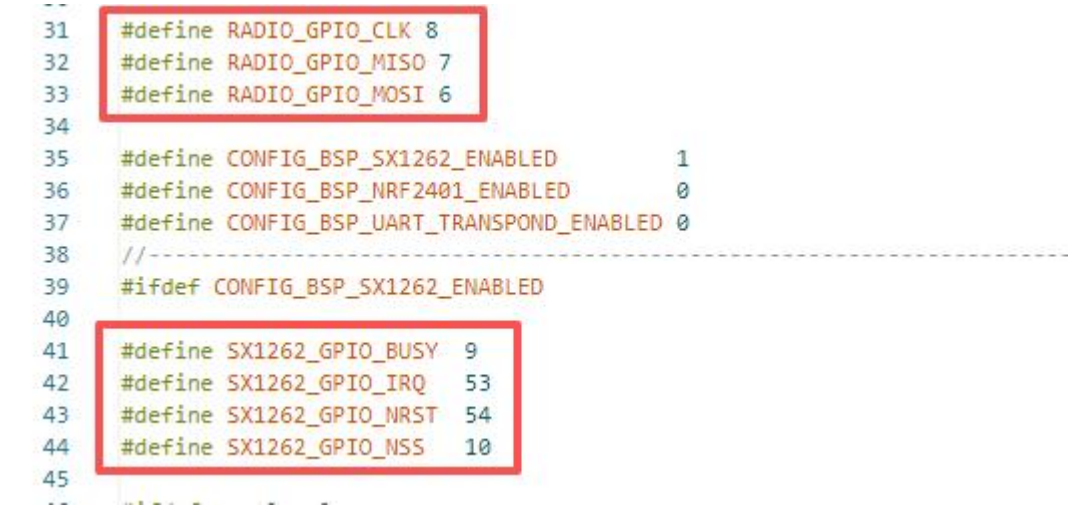




Since the function implementation in bsp_wireless.cpp uses the function encapsulation from EspHal.h, the reference to the header file needs to be placed in the .cpp file.
Take #include <RadioLib.h> as an example; this is a library under the network component.



Returning to bsp_wireless.h,here we declare the pins used by the wireless module.



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  9
#define SX1262_GPIO_IRQ   53
#define SX1262_GPIO_NRST  54
#define SX1262_GPIO_NSS   10

#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.

```
36  EspHal lora_hal;
37  Module *BSP_SX1262::bsp_sx_mod = nullptr;
38  SX1262 *BSP_SX1262::bsp_sx_radio = nullptr;
39
40  static int lora_transmissionState = RADIOLIB_ERR_NONE;
41  volatile bool lora_transmittedFlag = true;
42  volatile bool lora_receivedFlag = false;
43  static size_t lora_received_len = 0;
44
45  // Pointer to the data reception callback function
46  static void (*rx_data_callback)(const char* data, size_t len, float rssi, float snr) = NULL;
47  #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 1.6, 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.

```
96  esp_err_t BSP_SX1262::Sx1262_tx_init()
97  {
98    lora_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI);
99    lora_hal.setSpiFrequency(8000000);
100   lora_hal.spiBegin();
101
102   bsp_sx_mod = new Module(&lora_hal, SX1262_GPIO_NSS, SX1262_GPIO_IRQ, SX1262_GPIO_NRST, SX1262_GPIO_BUSY);
103   bsp_sx_radio = new SX1262(bsp_sx_mod);
104   int state = bsp_sx_radio->begin(915.0, 125.0, 7, 7, RADIOLIB_SX126X_SYNC_WORD_PRIVATE, 22, 8, 1.6);
105   if (state != RADIOLIB_ERR_NONE)
106   {
107     SX1262_ERROR("radio tx init failed, code :%d", state);
108     lora_hal.spiEnd();
109     return ESP_FAIL;
110   }
111   // bsp_sx_radio->setCurrentLimit(60);
112   bsp_sx_radio->setPacketSentAction(set_sx1262_tx_flag);
113   return ESP_OK;
114 }
```

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, 1.6);
  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.

Lesson13_RX_SX1262_Wireless_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h | esp_pa

```
 */
/* ─────────────────────────────────────────────────────────────
 | | | | | | | | | | | | | | | INCLUDES
 ──────────────────────────────────────────────────────────── */
#include "board_config.h"   // board pin define
#include <Arduino.h>        // Arduino core library. Must be placed at the very top to ensure recogniti

#include <string.h>         // C string lib
#include <esp_log.h>        // ESP-IDF logging library
#include <esp_err.h>        // ESP-IDF error codes
#include <esp_ldo_regulator.h>      // ESP32-P4 specific LDO management

/* panel driver */
#include "esp_panel_drivers_conf.h"
#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

/* LVGL and driver */
#include <lvgl.h>
#include "lvgl_v8_port.h"

/* LoRa */
#include "bsp_wireless.h"

using namespace esp_panel::drivers;
using namespace esp_panel::board;
/* ─────────────────────────────────────────────────────────────
 | | | | | | | | | | | | | | | DEFINITIONS
 ──────────────────────────────────────────────────────────── */
#define PRINTF_ORIGINAL(fmt, ...) Serial.printf(fmt, ##__VA_ARGS__);
#define PRINTF_PRINT(fmt, ...)    Serial.print(fmt);
#define PRINTF_LN(fmt, ...)       Serial.println(fmt);
```

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.

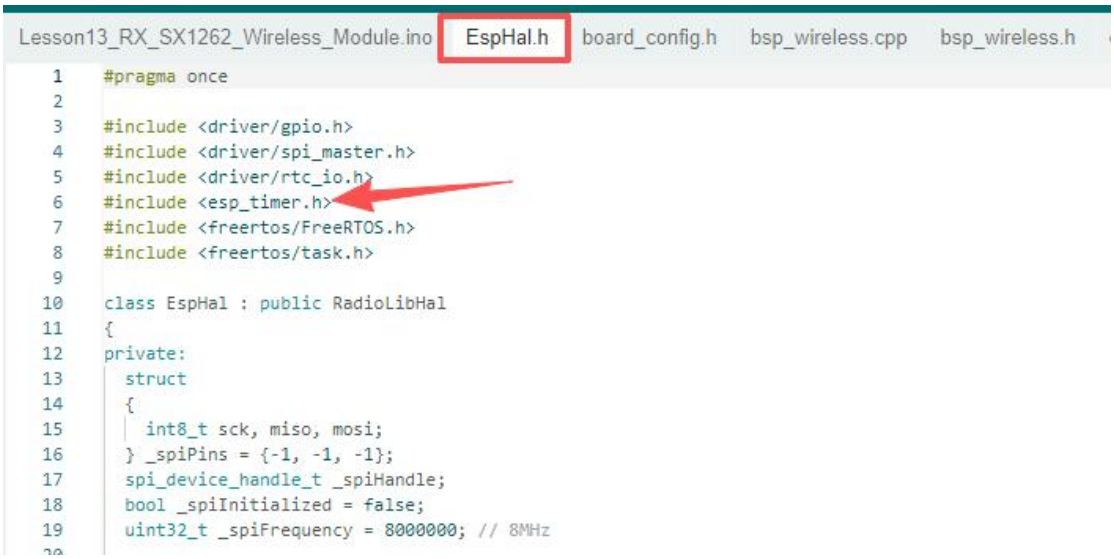
Lesson13_RX_SX1262_Wireless_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h | esp_panel_b

```
#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──────────────────────*/
```

Lesson13_RX_SX1262_Wireless_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h

```
/*──────────────────────────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.



Sender Main Function

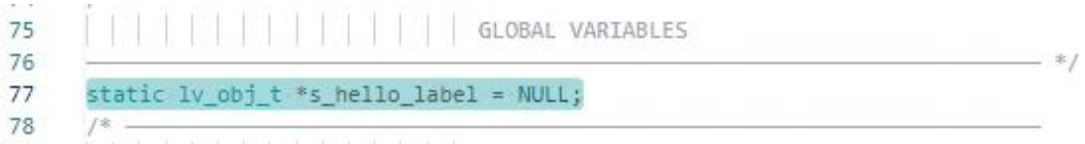
Next, let's examine how the main ".ino" code utilizes the interfaces from the above libraries to implement LoRa message transmission.

When the program runs, the general flow is as follows:

The complete workflow of this program can be summarized as follows: system startup → LDO power initialization → MIPI display and touch initialization → LVGL graphics interface startup → LoRa wireless module initialization → creation of two real-time tasks (one responsible for sending LoRa data packets every second, another responsible for updating the on-screen transmission counter every second) → real-time display of wireless transmission count on screen with serial port log output.

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.



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.

```
86  static void lvgl_show_counter_label_init(void)
87  {
88      if (lvgl_port_lock(0) != true) {
89          MAIN_ERROR("LVGL lock failed");
90          return;
91      }
92
93      lv_obj_t *screen = lv_scr_act();
94      lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);
95      lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);
96
97      s_hello_label = lv_label_create(screen);
98      if (s_hello_label == NULL) {
99          MAIN_ERROR("Create LVGL label failed");
100         lvgl_port_unlock();
101         return;
102     }
103     static lv_style_t label_style;
104     lv_style_init(&label_style);
105     lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
106     lv_style_set_text_color(&label_style, lv_color_black());
107     lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
108     lv_obj_add_style(s_hello_label, &label_style, LV_PART_MAIN);
109
110     lv_label_set_text(s_hello_label, "TX_Hello World:0");
111     lv_obj_center(s_hello_label);
112
113     lvgl_port_unlock();
114 }
```

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.

```
86  static void lvgl_show_counter_label_init(void)
87  {
88      if (lvgl_port_lock(0) != true) {
89          MAIN_ERROR("LVGL lock failed");
90          return;
91      }
```

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.

```
93      lv_obj_t *screen = lv_scr_act();
94      lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);
95      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.

```
97      s_hello_label = lv_label_create(screen);
98      if (s_hello_label == NULL) {
99          MAIN_ERROR("Create LVGL label failed");
100         lvgl_port_unlock();
101         return;
102     }
```

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.

```
103     static lv_style_t label_style;
104     lv_style_init(&label_style);
105     lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
106     lv_style_set_text_color(&label_style, lv_color_black());
107     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.

```
109         lv_obj_add_style(s_hello_label, &label_style, LV_PART_MAIN);
110
111         lv_label_set_text(s_hello_label, "TX_Hello World:0");
112         lv_obj_center(s_hello_label);
113
114         lvgl_port_unlock();
115     }
```

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.

```
117 static void ui_counter_task(void *param)
118 {
119     char text[48];
120     TickType_t last_wake_time = xTaskGetTickCount();
121     const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms
122
123     for (;;) {
124         uint32_t i = sx1262_get_tx_counter();
125         int n = snprintf(text, sizeof(text), "TX_Hello World:%lu", (unsigned long)i);
126         (void)n;
127
128         if (lvgl_port_lock(0) == true) {
129             if (s_hello_label != NULL) {
130                 lv_label_set_text(s_hello_label, text);
131             }
132             lvgl_port_unlock();
133         }
134
135         MAIN_INFO("TX msg: %s", text);
136
137         // Use absolute time to ensure an exact one-second interval
138         vTaskDelayUntil(&last_wake_time, frequency);
139     }
140 }
```

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.

```
117  static void ui_counter_task(void *param)
118  {
119      char text[48];
120      TickType_t last_wake_time = xTaskGetTickCount();
121      const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms
122
```

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.

```
123      for (;;) {
124          uint32_t i = sx1262_get_tx_counter();
125          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.

```
128              if (lvgl_port_lock(0) == true) {
129                  if (s_hello_label != NULL) {
130                      lv_label_set_text(s_hello_label, text);
131                  }
132                  lvgl_port_unlock();
133              }
```

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.

```
135              MAIN_INFO("TX msg: %s", text);
136
137              // Use absolute time to ensure an exact one-second interval
138              vTaskDelayUntil(&last_wake_time, frequency);
139          }
140  }
```

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.

```
142  static void lora_tx_task(void *param)
143  {
144      TickType_t last_wake_time = xTaskGetTickCount();
145      const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms
146
147      while (1) {
148          bool lora_tx_OK = false;
149          lora_tx_OK = send_lora_pack_radio();
150          if (lora_tx_OK != true) {
151              MAIN_ERROR("LoRa TX failed");
152          }
153
154          vTaskDelayUntil(&last_wake_time, frequency);
155      }
156  }
```

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.

```
142  static void lora_tx_task(void *param)
143  {
144      TickType_t last_wake_time = xTaskGetTickCount();
145      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.

```
147      while (1) {
148          bool lora_tx_OK = false;
149          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.

```
150             if (lora_tx_OK != true) {
151                 MAIN_ERROR("LoRa TX failed");
152             }
153
154             vTaskDelayUntil(&last_wake_time, frequency);
155         }
156     }
```

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 ldo_init

This code implements a power initialization function "ldo_init()", whose primary role is to configure and enable the internal LDO (Low Dropout Regulator) channels of the ESP32-P4 chip at system startup—providing stable operating voltage for the display interface and touch/I2C circuits.

```
158 void ldo_init()
159 {
160     // --- Power Configuration (LDO3 for MIPI D-PHY) ---
161     // ESP32-P4's MIPI D-PHY requires specific voltage to function.
162     // LDO3 is typically routed to the MIPI power rail on P4 hardware.
163     esp_err_t err = ESP_OK;
164     esp_ldo_channel_handle_t ldo3_handle = NULL;
165     esp_ldo_channel_config_t ldo3_cfg = {
166         .chan_id = 3,           // LDO Channel 3
167         .voltage_mv = 2500,     // Set to 2500mV (2.5V)
168     };
169
170     Serial.println("Initializing LDO3 to 2.5V...");
171     err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
172     if (err != ESP_OK) {
173         Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
174     } else {
175         Serial.println("LDO3 Power enabled successfully.");
176     }
177
178     // --- Power Configuration (LDO4 for I2C/touch pull up) ---
179     esp_ldo_channel_handle_t ldo4_handle = NULL;
180     esp_ldo_channel_config_t ldo4_cfg = {
181         .chan_id = 4,           // LDO Channel 4
182         .voltage_mv = 3300,     // Set to 3300mV (3.3V)
183     };
184
185     Serial.println("Initializing LDO4 to 3.3V...");
186     err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
187     if (err != ESP_OK) {
188         Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
189     } else {
190         Serial.println("LDO4 Power enabled successfully.");
191     }
192 }
```

The function begins by defining an esp_err_t type variable err to store the error status after function execution, along with an esp_ldo_channel_handle_t type handle variable to represent the successfully acquired LDO channel control object.

```
158  void ldo_init()
159  {
160      // --- Power Configuration (LDO3 for MIPI D-PHY) ---
161      // ESP32-P4's MIPI D-PHY requires specific voltage to function.
162      // LDO3 is typically routed to the MIPI power rail on P4 hardware.
163      esp_err_t err = ESP_OK;
164      esp_ldo_channel_handle_t ldo3_handle = NULL;
```

The first configuration is for the LDO3 power channel. The program creates a structure esp_ldo_channel_config_t ldo3_cfg, where chan_id = 3 selects the LDO3 channel, and voltage_mv = 2500 sets the output voltage to 2500mV (2.5V).

```
165      esp_ldo_channel_config_t ldo3_cfg = {
166          .chan_id = 3,           // LDO Channel 3
167          .voltage_mv = 2500,     // Set to 2500mV (2.5V)
168      };
169
```

This voltage is primarily used to power the MIPI D-PHY display interface circuit, as the MIPI display interface in ESP32-P4's hardware design typically requires a dedicated ~2.5V power rail for normal operation. The program then outputs debug information via Serial.println() indicating LDO3 initialization, and calls esp_ldo_acquire_channel() to request and enable this LDO channel. If the returned result err is not ESP_OK, indicating power configuration failure, the program uses Serial.printf() to output an error message and converts the error code to a readable string via esp_err_to_name(); if successful, it prints a confirmation message indicating LDO3 power has been successfully enabled.

```
170      Serial.println("Initializing LDO3 to 2.5V...");
171      err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
172      if (err != ESP_OK) {
173          Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
174      } else {
175          Serial.println("LDO3 Power enabled successfully.");
176      }
```

The program then continues to configure the LDO4 power channel, following essentially the same procedure as LDO3: first defining handle ldo4_handle and configuration structure ldo4_cfg, where chan_id = 4 selects the LDO4 channel, and voltage_mv = 3300 sets the output voltage to 3300mV (3.3V).

```
178      // --- Power Configuration (LDO4 for I2C/touch pull up) ---
179      esp_ldo_channel_handle_t ldo4_handle = NULL;
180      esp_ldo_channel_config_t ldo4_cfg = {
181          .chan_id = 4,           // LDO Channel 4
182          .voltage_mv = 3300,     // Set to 3300mV (3.3V)
183      };
184
```

This voltage is primarily used to power the I2C bus pull-up resistors and touch controller—for example, common capacitive touch chips like GT911 typically operate in 3.3V voltage environments, requiring stable power support.

```
185     Serial.println("Initializing LDO4 to 3.3V...");
186     err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
187     if (err != ESP_OK) {
188         Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
189     } else {
190         Serial.println("LDO4 Power enabled successfully.");
191     }
192 }
```

The program similarly prints initialization information via serial port and calls esp_ldo_acquire_channel() to enable LDO4. If configuration fails, it prints an error message; if successful, it indicates the power has been properly activated.

⑥ display_touch_lvgl_init

This code implements a display, touchscreen, and graphical interface system initialization function "display_touch_lvgl_init()", whose primary role is to complete the comprehensive initialization of display hardware, touch controller, and graphics library at system startup—enabling the screen to properly display graphical interfaces and support touch input.

```
194 void display_touch_lvgl_init()
195 {
196     // --- Initialize Display and Touch Panel ---
197     Board *board = new Board();
198     // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
199     Serial.println("Initializing Panel (EK79007 + GT911)...");
200     assert(board->init());
201 #if LVGL_PORT_AVOID_TEARING_MODE
202     auto lcd = board->getLCD();
203     // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
204     lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
205 #endif
206     assert(board->begin());
207     Serial.println("Display and Touch system online.");
208
209     Serial.println("Initializing LVGL");
210     lvgl_port_init(board->getLCD(), board->getTouch());
211 }
```

The function begins by creating a Board class object via Board *board = new Board();. This class from ESP_Panel_Library primarily encapsulates hardware resources on the development board including the display panel, touchscreen, and communication buses—allowing users to complete initialization and control through a unified interface.

```
194 void display_touch_lvgl_init()
195 {
196     // --- Initialize Display and Touch Panel ---
197     Board *board = new Board();
```

The program then outputs "Initializing Panel (EK79007 + GT911)..." via the serial port to indicate the system is initializing display and touch devices. It subsequently calls board->init() to complete low-level hardware initialization—typically configuring the MIPI-DSI display bus, GPIOs, I2C touch bus, and related drivers, while loading specific hardware drivers such as the display control chip EK79007 and capacitive touch control chip GT911.

```
198     // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
199     Serial.println("Initializing Panel (EK79007 + GT911)...");
200     assert(board->init());
```

The function uses assert() for verification; if initialization fails, the program halts immediately to prevent subsequent execution with improperly initialized hardware.

The code then includes a conditional compilation section #if LVGL_PORT_AVOID_TEARING_MODE. When tear-free (anti-tearing) mode is enabled, the program obtains the LCD display object via board->getLCD() and calls configFrameBufferNumber() to set the number of frame buffers used by LVGL. This multi-buffering mechanism reduces display tearing artifacts during refresh—thereby improving screen display stability and visual quality.

```
201 #if LVGL_PORT_AVOID_TEARING_MODE
202     auto lcd = board->getLCD();
203     // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
204     lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
205 #endif
```

The program then calls board->begin() to complete the final device startup process. This function typically actually starts the display driver, initializes the touch controller, and brings the hardware into operational state—again using assert() to ensure successful execution.
Upon completion, the program outputs "Display and Touch system online." via the serial port, indicating the display and touch system is functioning normally.

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

Finally, the program begins initializing the graphical interface system. It prints "Initializing LVGL" to indicate the graphics library is about to start, then calls lvgl_port_init(board->getLCD(), board->getTouch()) to initialize the LVGL porting layer—registering the LCD display device and touch input device with the LVGL system. This enables LVGL to refresh the screen through the LCD driver and receive user input events through the touch driver.

```
209     Serial.println("Initializing LVGL");
210     lvgl_port_init(board->getLCD(), board->getTouch());
211 }
```

⑦ setup

This code is the Arduino program's "setup()" initialization function, which executes only once after system power-on or reset. It is primarily responsible for completing the entire system's hardware initialization, interface initialization, and task creation—establishing a complete runtime environment for subsequent program operation.

The function first calls Serial.begin(115200) to initialize serial communication (UART0) with a baud rate of 115200, used for outputting debug logs and system operation information.

It then calls ldo_init() to configure the internal LDO regulator channels of the ESP32-P4, enabling the system to provide stable power for the display interface and touch/I2C circuits.

```
213  void setup() {
214      // put your setup code here, to run once:
215
216      // Initialize the default Serial for debugging (UART0)
217      Serial.begin(115200);
218
219      ldo_init();
```

Subsequently, it calls display_touch_lvgl_init() to complete the initialization of display and touchscreen hardware, and start the LVGL graphics library—thereby endowing the screen with the capability to display graphical interfaces and receive touch input.

```
221      display_touch_lvgl_init();
```

After completing the display system initialization, the program begins initializing the wireless communication module. It calls sx1262_tx_init() to initialize the LoRa wireless chip. If the return value is not ESP_OK, indicating initialization failure, it outputs an error message via the logging macro MAIN_ERROR() to prompt wireless module startup failure. If initialization succeeds, the system can proceed with LoRa data transmission.

```
223      // lora tx init
224      esp_err_t err = sx1262_tx_init();
225      if (err != ESP_OK) {  // Check error
226          MAIN_ERROR("Wireless Module init...");  // Handle failure
227      }
```

The program then calls lvgl_show_counter_label_init() to create an LVGL label control on screen for displaying the wireless transmission counter—for example, "TX_Hello World:0"—and prints a log via MAIN_INFO() indicating the interface has been successfully displayed.

```
229      lvgl_show_counter_label_init();
230      MAIN_INFO("-------- LVGL Show OK ----------");
231
```

The program continues outputting logs indicating successful wireless module initialization, then uses xTaskCreatePinnedToCore() to create two FreeRTOS tasks:

The first task ui_counter_task is allocated 4096 bytes of stack space and pinned to CPU Core 0. Its primary role is to read the LoRa transmission counter every second and update the text label on screen.

The second task lora_tx_task is allocated 8192 bytes of stack space and pinned to CPU Core 1. It is responsible for sending a LoRa data packet to the wireless module every second.

```
233     MAIN_INFO("The wireless module initialization was successful.");  // Print success log
234     // Create tasks and use the same priority to ensure synchronization
235     xTaskCreatePinnedToCore(ui_counter_task, "ui_counter", 4096, NULL,
236                             configMAX_PRIORITIES - 5, NULL, 0);
237
238     xTaskCreatePinnedToCore(lora_tx_task, "sx1262_tx", 8192, NULL,
239                             configMAX_PRIORITIES - 5, NULL, 1);
240
241     MAIN_INFO("Tasks created, starting synchronized transmission...");
242 }
```

Both tasks are set to priority configMAX_PRIORITIES - 5, ensuring they have identical scheduling priority in the system—thereby achieving relatively synchronized operation. Finally, the program outputs the log "Tasks created, starting synchronized transmission..." indicating system initialization is complete and the system is entering the multi-tasking runtime phase.

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 Variables

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.

```
74  /* ──────────────────────────────────────────────────────────────────────
75                                  GLOBAL VARIABLES
76  ──────────────────────────────────────────────────────────────────────── */
77  static lv_obj_t *s_rx_label = NULL;            // LVGL label object to display received data
78  static lv_obj_t *s_rssi_label = NULL;          // LVGL label object to display RSSI value
79  static lv_obj_t *s_snr_label = NULL;           // LVGL label object to display SNR value
80  static uint32_t rx_packet_count = 0;           // Counter for the number of received LoRa packets
81  /* ──────────────────────────────────────────────────────────────────────
```

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 code is the Arduino program's "setup()" initialization function, whose role is to complete the initialization of the entire LoRa receiving system, display interface, and multi-tasking environment after system power-on or reset—thereby building an embedded system capable of receiving wireless data and displaying it on screen.

The program begins by calling Serial.begin(115200) to initialize the serial port (UART0) with a baud rate of 115200, used for outputting system debug information and runtime logs. It then calls ldo_init() to configure the internal LDO power channels of the ESP32-P4 chip, providing stable operating voltage for the display interface, touch

controller, and I2C pull-up circuits.

```
264  void setup() {
265      // put your setup code here, to run once:
266
267      // Initialize the default Serial for debugging (UART0)
268      Serial.begin(115200);
269
270      ldo_init();
```

Next, the program calls display_touch_lvgl_init() to complete the initialization of display and touchscreen hardware, while simultaneously starting the LVGL graphics library porting layer—endowing the system with graphical display and touch input capabilities.

```
272      display_touch_lvgl_init();
```

After the graphics system is ready, the program calls lvgl_show_rx_interface_init() to create and initialize an LVGL receiving interface (RX Interface). This interface typically contains text labels or other controls used to display received wireless data or reception status on screen. It outputs the prompt message "-------- LVGL RX Interface OK ----------" via the logging macro MAIN_INFO() to indicate the interface has been successfully created.

```
274      lvgl_show_rx_interface_init();  // Initialize LVGL user interface
275      MAIN_INFO("-------- LVGL RX Interface OK ----------");  // Log successful UI init
```

The program then begins initializing the LoRa wireless receiving module. It calls sx1262_rx_init() to initialize the SX1262 LoRa chip in receive mode—including configuring wireless frequency, communication parameters, and receive state machine.

If the function return value is not ESP_OK, indicating initialization failure, the program outputs an error log via MAIN_ERROR() to prompt wireless module startup failure. If initialization succeeds, it prints "The wireless module RX initialization was successful." via MAIN_INFO() to indicate the LoRa receiving module is functioning normally.

```
277      // Wireless RX init (LoRa receiver initialization)
278      esp_err_t err = sx1262_rx_init();  // Initialize SX1262 LoRa receiver
279      if (err != ESP_OK) {
280          MAIN_ERROR("Wireless Module RX init...");  // Halt if failed
281      }
282      MAIN_INFO("The wireless module RX initialization was successful.");  // Log success
```

The program then calls sx1262_set_rx_callback(rx_data_callback) to register a receive callback function rx_data_callback.

When the wireless module receives a new LoRa data packet, the underlying driver automatically invokes this callback function—triggering data processing logic in the program such as parsing data, updating the interface, or recording reception information. It outputs the log "RX callback registered" to indicate successful callback function registration.

```
284     //  (Set callback function for received data)
285     sx1262_set_rx_callback(rx_data_callback);  // Register LoRa RX callback function
286     MAIN_INFO("RX callback registered");  // Log callback registration success
287
```

Finally, the program creates a FreeRTOS task lora_rx_task via xTaskCreatePinnedToCore(), allocating 4096 bytes of task stack space and pinning this task to CPU Core 1, with task priority set to configMAX_PRIORITIES - 5. This task's primary role is to continuously monitor LoRa reception status and process received data—thereby ensuring stable operation of the wireless receiving function.

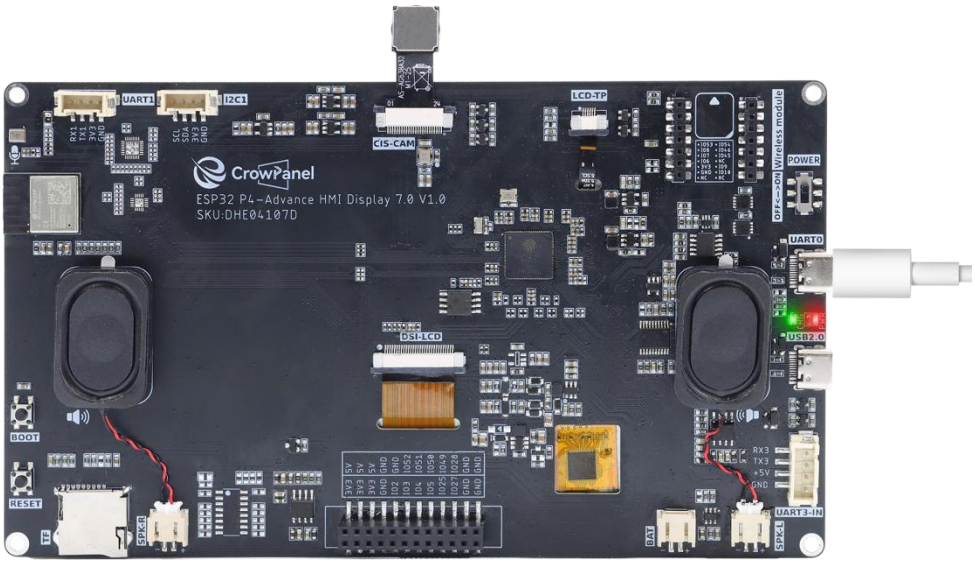
```
288     // (Create LoRa receiving task)
289     xTaskCreatePinnedToCore(lora_rx_task, "sx1262_rx", 4096, NULL,
290                             configMAX_PRIORITIES - 5, NULL, 1);  // Create FreeRTOS task pinned to core 1
291
292     MAIN_INFO("LoRa RX receiver started, waiting for data...");  // Log start message
293 }
```

After task creation, the program outputs the log "LoRa RX receiver started, waiting for data..." via the logging macro, indicating the system has entered LoRa receive mode and is beginning to wait for wireless data.

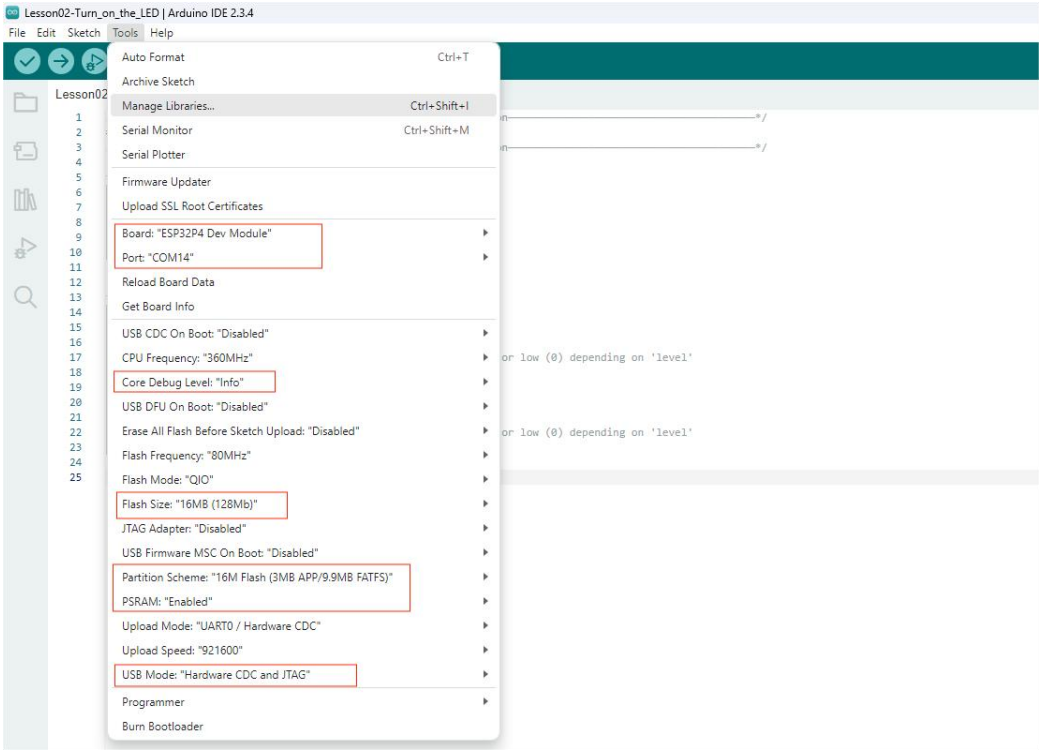
Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.

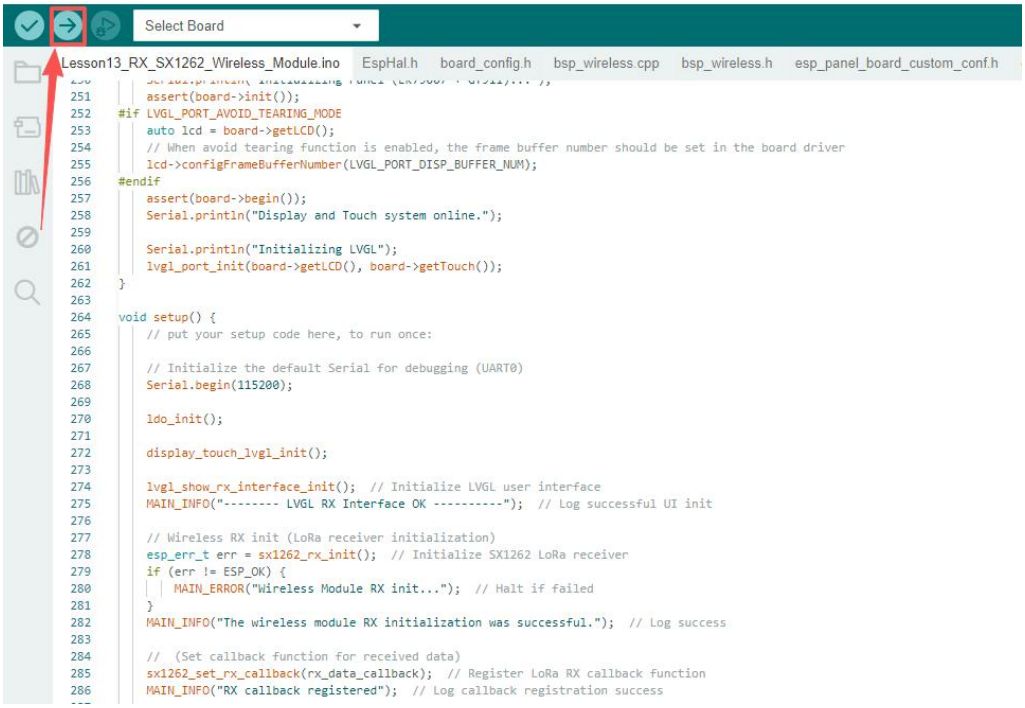
First, we connect the Advance-P4 device to our computer host via the USB cable.



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



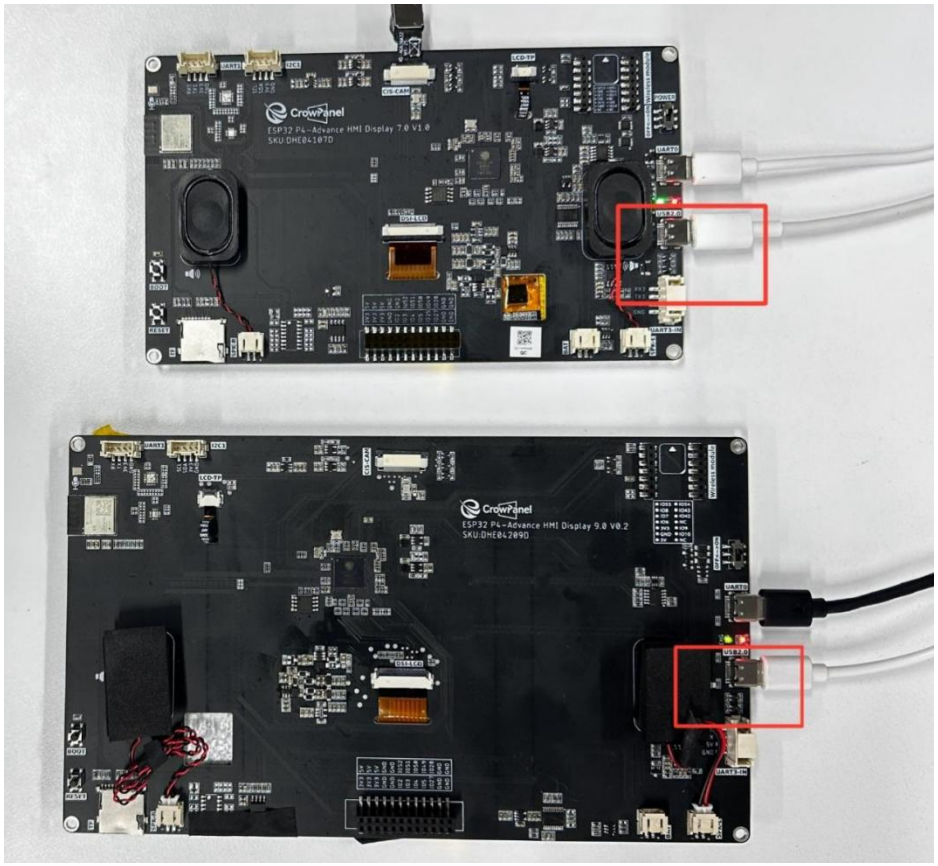
Then we compile and upload the code.



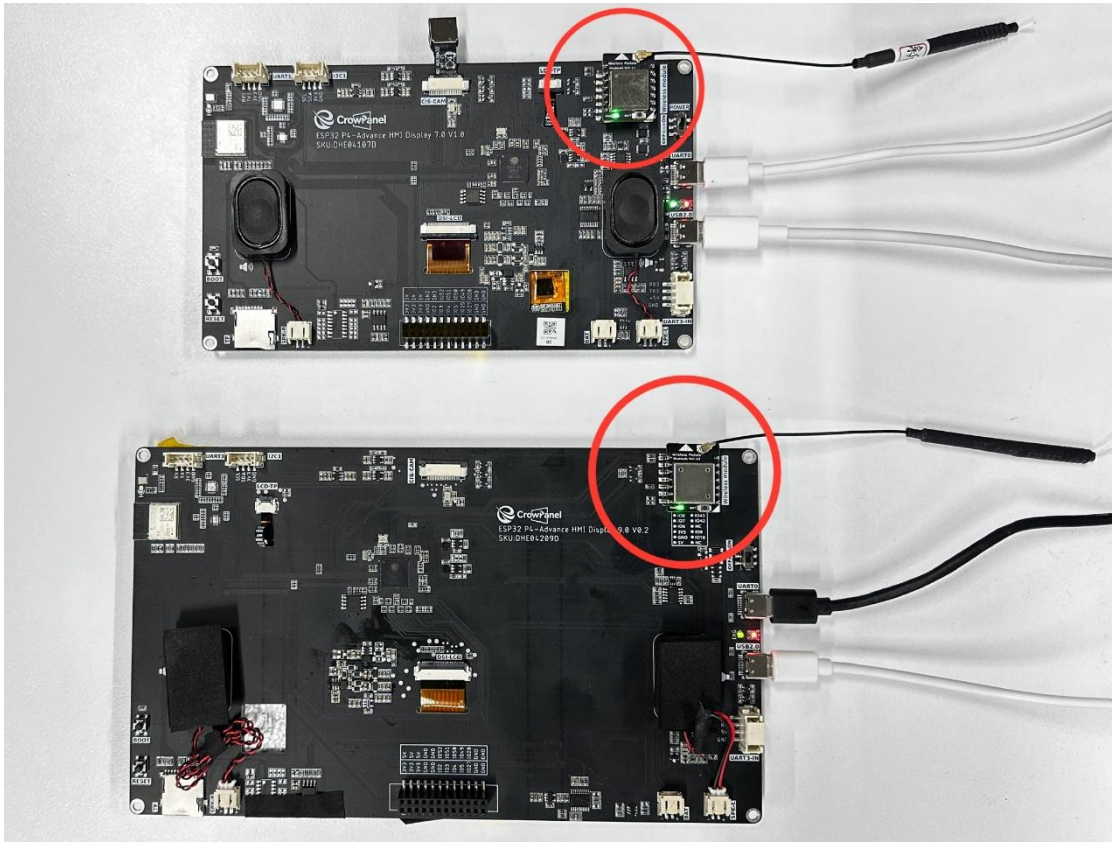
Wait for the code upload to complete.

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.)

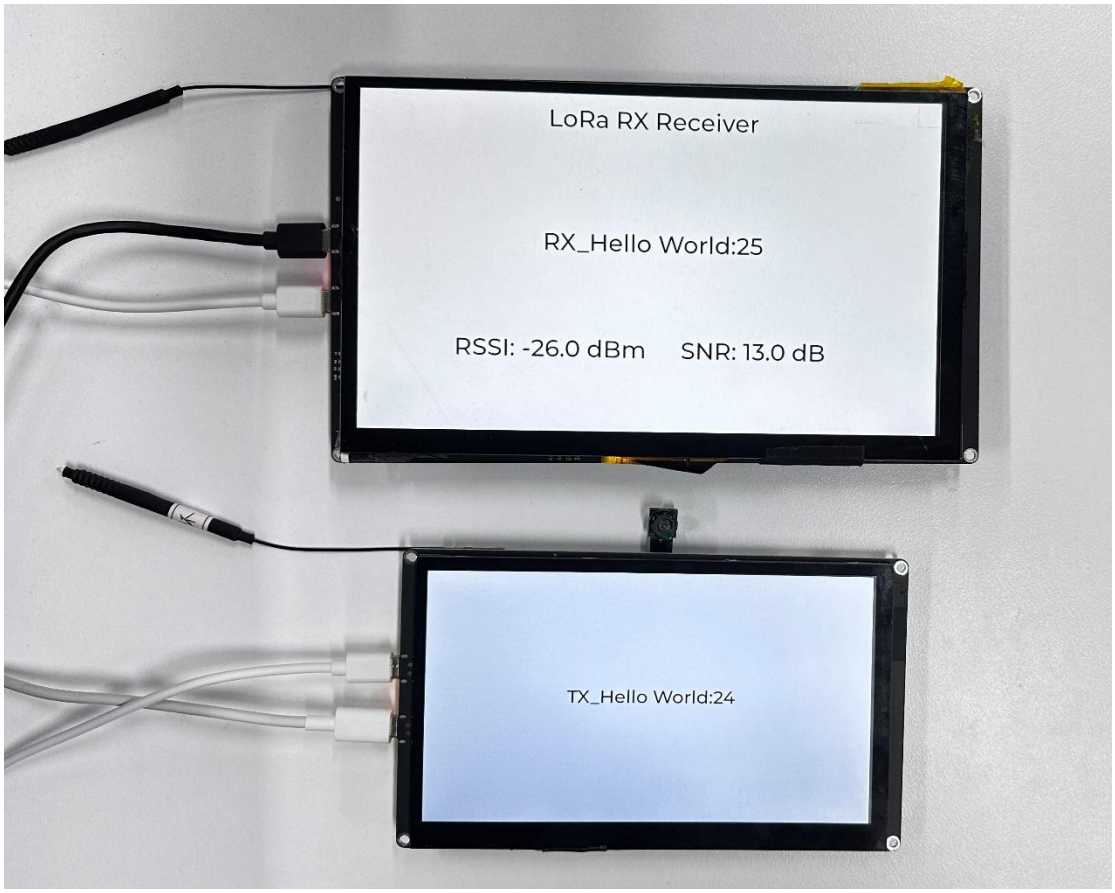


Insert the LoRa module SX1262 into the two Advance-P4 development boards respectively.



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.



Lesson14--- 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).

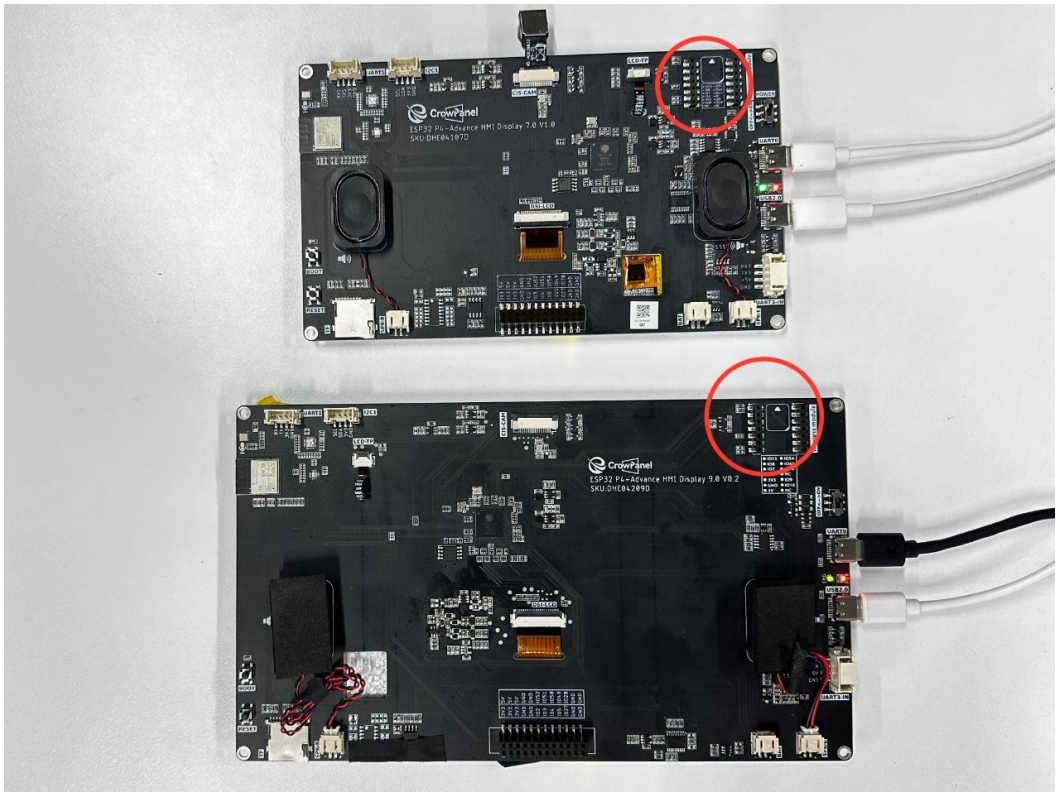
3. Master the design of nRF24L01 transceiver logic based on FreeRTOS multi-tasking (separation of transmission task, reception task, and UI update task), as well as thread-safe implementation of dynamic LVGL interface updates (transmission/reception counter display).

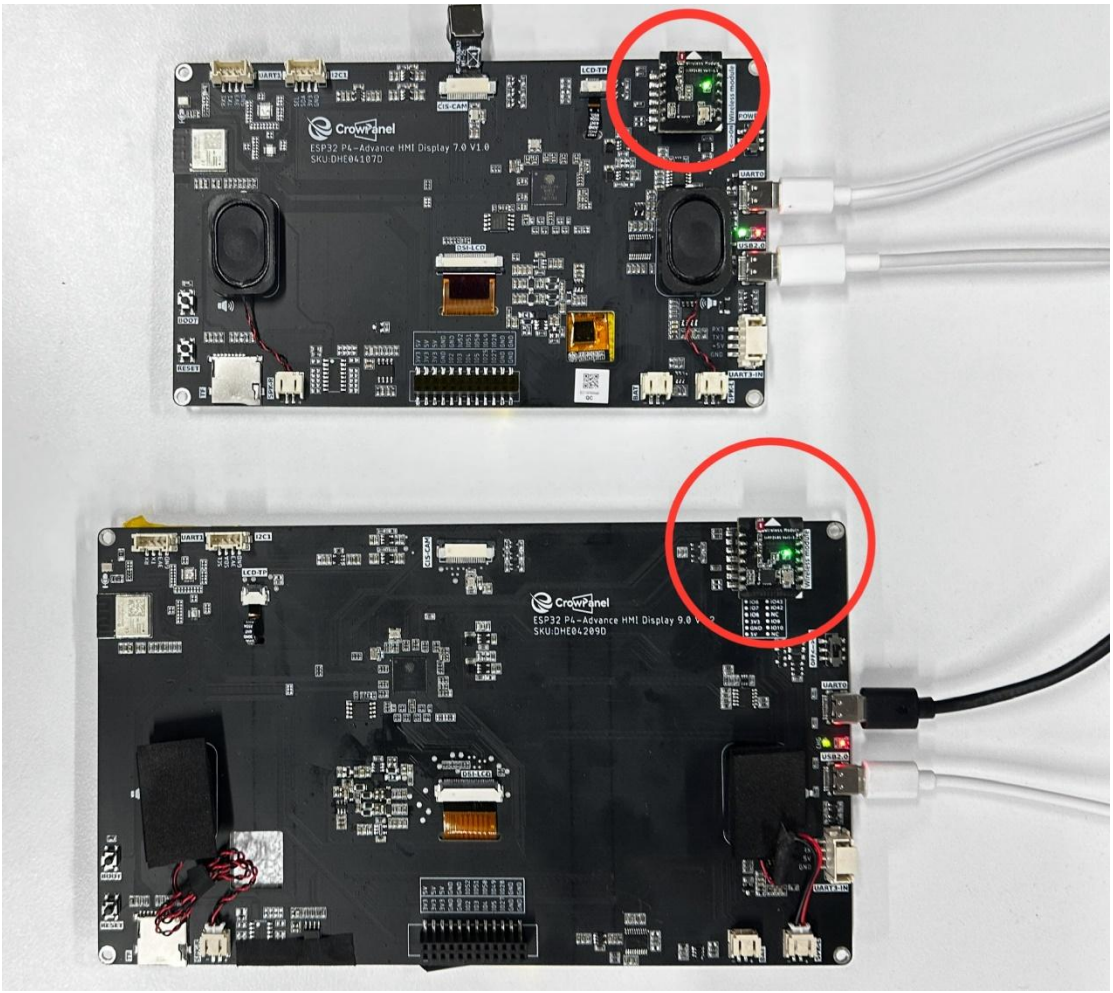
Preview of the Result

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.
(Friendly reminder: The 7-inch, 9-inch, and 10.1-inch displays share the same code from the link below, as these three sizes differ only in dimensions for your selection—the hardware schematics and wiring remain identical.)

Kindly click the link below to view the full code implementation.

Transmitting end code：
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/Lesson14_TX_nRF2401_Wireless_RF_Module

Receiving end code：
https://github.com/Elecrow-RD/CrowPanel-Advanced-7inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Arduino_Code/Lesson14_RX_nRF2401_Wireless_RF_Module

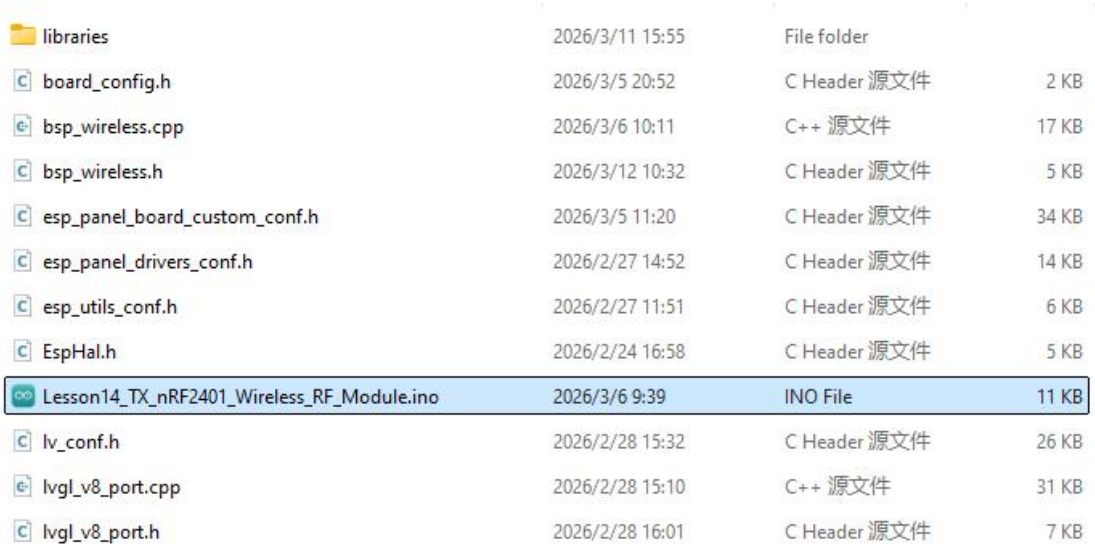
Key Explanations

The focus of this lesson is on how to use the wireless module, including initializing the nRF2401 module and sending or receiving information.

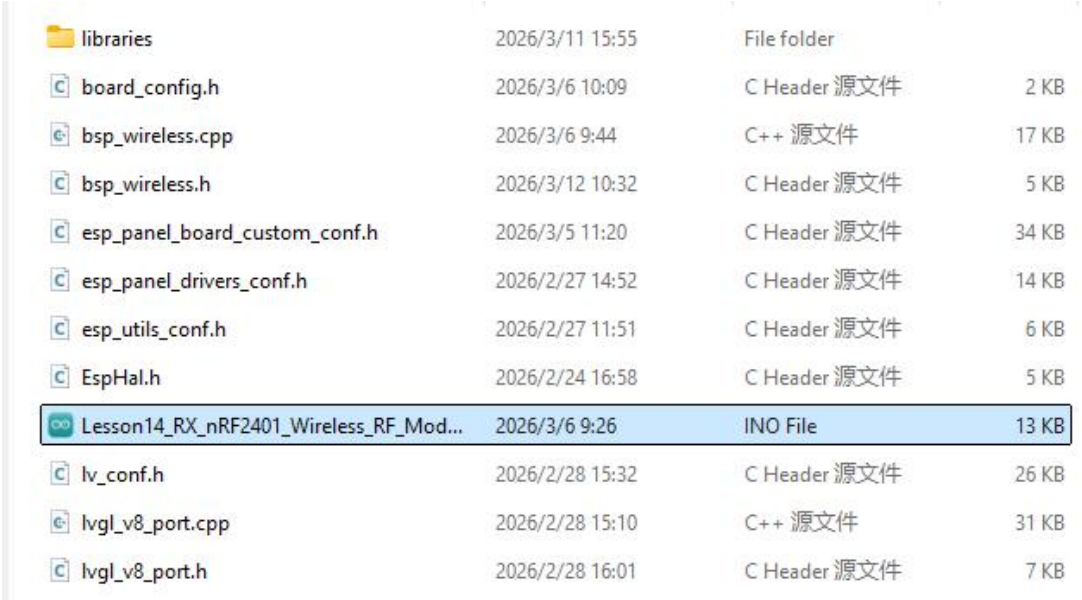
Double-click to open the code for this lesson (the .ino file).

The code for this lesson includes a transmitter side and a receiver side:

Transmitter side:



Receiver side:



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.

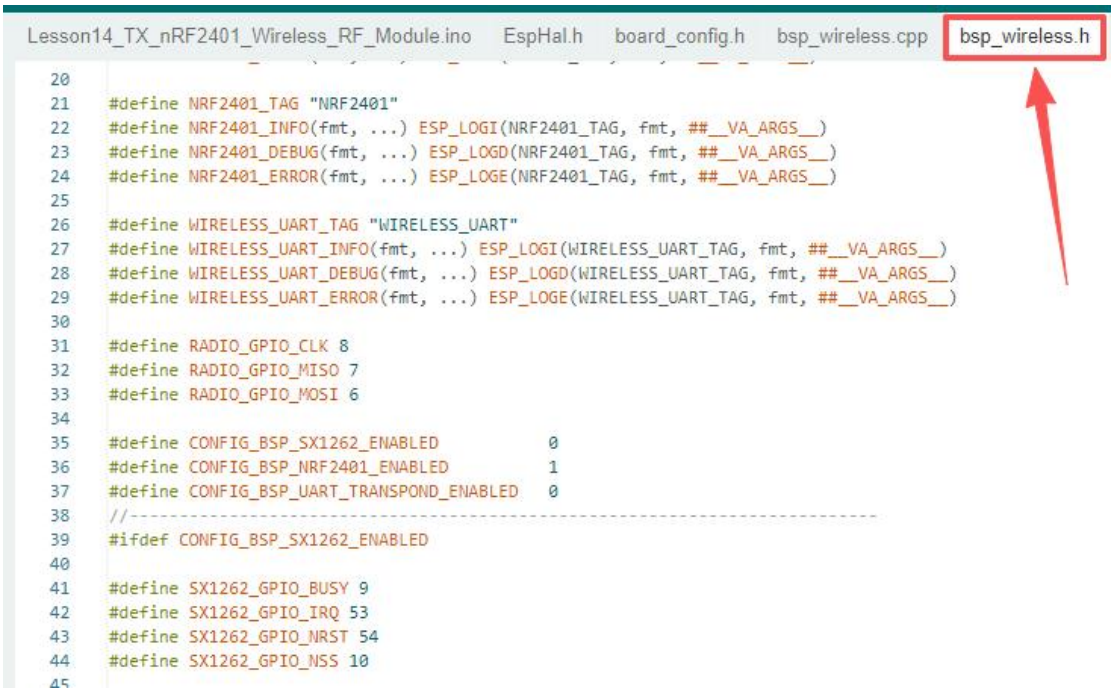
- It also handles the reception of wireless data packets sent by other devices via the nRF2401.
- It returns the received data to the upper-layer application through a callback mechanism.

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.



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.

Lesson14_TX_nRF2401_Wireless_RF_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h

```
20
21  #define NRF2401_TAG "NRF2401"
22  #define NRF2401_INFO(fmt, ...) ESP_LOGI(NRF2401_TAG, fmt, ##__VA_ARGS__)
23  #define NRF2401_DEBUG(fmt, ...) ESP_LOGD(NRF2401_TAG, fmt, ##__VA_ARGS__)
24  #define NRF2401_ERROR(fmt, ...) ESP_LOGE(NRF2401_TAG, fmt, ##__VA_ARGS__)
25
26  #define WIRELESS_UART_TAG "WIRELESS_UART"
27  #define WIRELESS_UART_INFO(fmt, ...) ESP_LOGI(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
28  #define WIRELESS_UART_DEBUG(fmt, ...) ESP_LOGD(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
29  #define WIRELESS_UART_ERROR(fmt, ...) ESP_LOGE(WIRELESS_UART_TAG, fmt, ##__VA_ARGS__)
30
31  #define RADIO_GPIO_CLK 8
32  #define RADIO_GPIO_MISO 7
33  #define RADIO_GPIO_MOSI 6
34
35  #define CONFIG_BSP_SX1262_ENABLED          0
36  #define CONFIG_BSP_NRF2401_ENABLED         1
37  #define CONFIG_BSP_UART_TRANSPOND_ENABLED  0
38  //------------------------------------------------------------------------------
39  #ifdef CONFIG_BSP_SX1262_ENABLED
40
41  #define SX1262_GPIO_BUSY 9
42  #define SX1262_GPIO_IRQ 53
43  #define SX1262_GPIO_NRST 54
44  #define SX1262_GPIO_NSS 10
45
```

(Enable the one that corresponds to the wireless module you are using.)

```
67  //------------------------------------------------------------------------------
68
69  #ifdef CONFIG_BSP_NRF2401_ENABLED                                   OPEN
70
71  #define NRF24_GPIO_IRQ 9
72  #define NRF24_GPIO_CE 53
73  #define NRF24_GPIO_CS 54
74
75  #ifdef __cplusplus
76  extern "C"
77  {
78  #endif
79      esp_err_t nrf24_tx_init();
80      void nrf24_tx_deinit();
81      bool send_nrf24_pack_radio();
82      uint32_t nrf24_get_tx_counter();
83      void nrf24_inc_tx_counter();
84
85      esp_err_t nrf24_rx_init();
86      void nrf24_rx_deinit();
87      void received_nrf24_pack_radio(size_t len);
88      void nrf24_set_rx_callback(void (*callback)(const char* data, size_t len));
89  #ifdef __cplusplus
90  }
91  #endif
92  #endif
93  //------------------------------------------------------------------------------
94
95  #ifdef CONFIG_BSP_UART_TRANSPOND_ENABLED
96
97  #define UART_GPIO_TXD 53                                            CLOSE
98  #define UART_GPIO_RXD 54
99  #ifdef __cplusplus
100 extern "C"
101 {
102 #endif
103     esp_err_t uart_transpond_init();
104     void uart_transpond_deinit();
105 #ifdef __cplusplus
106 }
107 #endif
108 #endif
```

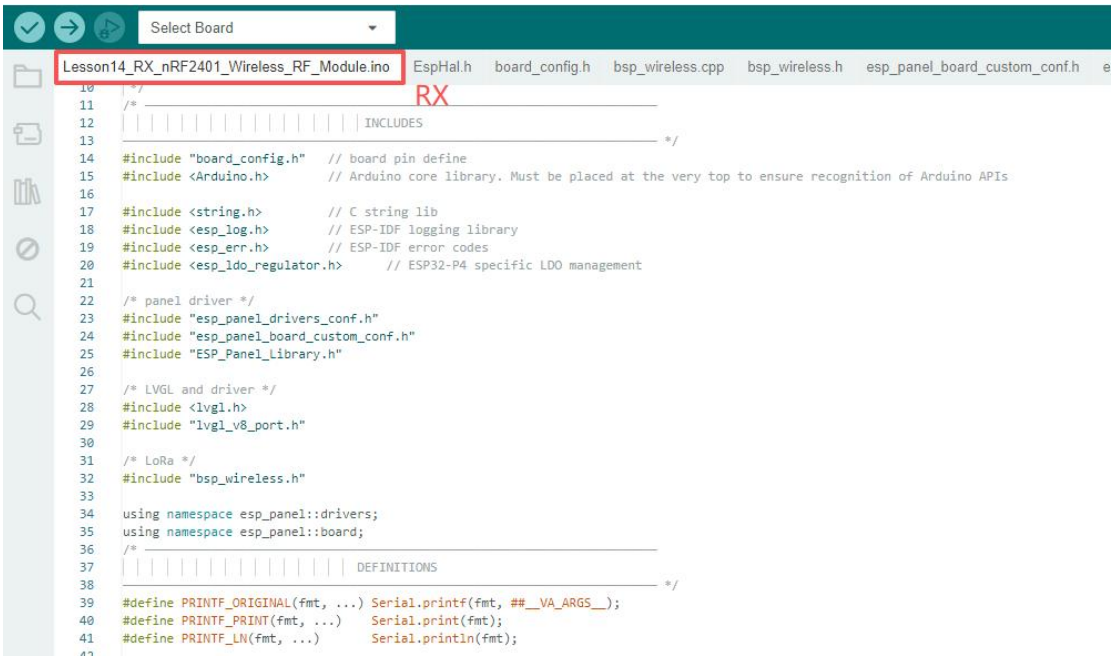
As shown in the figure, we have enabled the nRF2401 configuration, so the other modules are temporarily disabled and not applicable.
In the bsp_wireless component, you only need to call the prepared interfaces when needed.

Next, let’s focus on understanding the bsp_wireless component.

The following section shows the transmitter (TX) side of the project:


```
#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 8
#define RADIO_GPIO_MISO 7
#define RADIO_GPIO_MOSI 6

#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 9
#define SX1262_GPIO_IRQ 53
#define SX1262_GPIO_NRST 54
#define SX1262_GPIO_NSS 10
```

The following section shows the receiver (RX) side of the project:




```
/* ---------------------------------------------------------------------------
                                  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
#include <esp_ldo_regulator.h>      // ESP32-P4 specific LDO management

/* panel driver */
#include "esp_panel_drivers_conf.h"
#include "esp_panel_board_custom_conf.h"
#include "ESP_Panel_Library.h"

/* LVGL and driver */
#include <lvgl.h>
#include "lvgl_v8_port.h"

/* LoRa */
#include "bsp_wireless.h"

using namespace esp_panel::drivers;
using namespace esp_panel::board;
/* ---------------------------------------------------------------------------
                                 DEFINITIONS
--------------------------------------------------------------------------- */
#define PRINTF_ORIGINAL(fmt, ...) Serial.printf(fmt, ##__VA_ARGS__);
#define PRINTF_PRINT(fmt, ...)    Serial.print(fmt);
#define PRINTF_LN(fmt, ...)       Serial.println(fmt);
```

In these two projects, the only difference lies in the main functions: Lesson14_TX_nRF2401_Wireless_RF_Module.ino for the transmitter and Lesson14_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.)

Other library files under this project have been explained in detail in the previous lesson, so they will not be elaborated on 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.

Lesson14_RX_nRF2401_Wireless_RF_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h

```
65
66  #endif
67  //------------------------------------------------------------------------
68
69  #ifdef CONFIG_BSP_NRF2401_ENABLED
70
71  #define NRF24_GPIO_IRQ 9
72  #define NRF24_GPIO_CE 53
73  #define NRF24_GPIO_CS 54
74
75  #ifdef __cplusplus
76  extern "C"
77  {
78  #endif
79      esp_err_t nrf24_tx_init();
80      void nrf24_tx_deinit();
81      bool send_nrf24_pack_radio();
82      uint32_t nrf24_get_tx_counter();
83      esp_err_t nrf24_rx_init();
84      void nrf24_rx_deinit();
85      void received_nrf24_pack_radio(size_t len);
86      void nrf24_set_rx_callback(void (*callback)(const char* data, size_t len));
87  #ifdef __cplusplus
88  }
89  #endif
90  #endif
```

Lesson14_RX_nRF2401_Wireless_RF_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h

```
48
49  #ifdef CONFIG_BSP_NRF2401_ENABLED
50  class BSP_NRF2401
51  {
52  public:
53    BSP_NRF2401() {};
54
55    ~BSP_NRF2401() {};
56
57    esp_err_t NRF24_tx_init();
58
59    void NRF24_tx_deinit();
60
61    bool Send_pack_radio();
62
63    esp_err_t NRF24_rx_init();
64
65    void NRF24_rx_deinit();
66
67    void Received_pack_radio(size_t len);
68
69  protected:
```

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)

Lesson14_RX_nRF2401_Wireless_RF_Module.ino | EspHal.h | board_config.h | bsp_wireless.cpp | bsp_wireless.h

```
a/*-------------------------------------------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,
this is where we declare the pins used by the wireless module.

```
#define RADIO_GPIO_CLK 8
#define RADIO_GPIO_MISO 7
#define RADIO_GPIO_MOSI 6

#ifdef CONFIG_BSP_NRF2401_ENABLED

#define NRF24_GPIO_IRQ 9
#define NRF24_GPIO_CE 53
#define NRF24_GPIO_CS 54
```

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.)

```
67  //------------------------------------------------------------------------
68
69  #ifdef CONFIG_BSP_NRF2401_ENABLED
70
71  #define NRF24_GPIO_IRQ 9
72  #define NRF24_GPIO_CE 53
73  #define NRF24_GPIO_CS 54
74
75  #ifdef __cplusplus
76  extern "C"
77  {
78  #endif
79      esp_err_t nrf24_tx_init();
80      void nrf24_tx_deinit();
81      bool send_nrf24_pack_radio();
82      uint32_t nrf24_get_tx_counter();
83      void nrf24_inc_tx_counter();
84
85      esp_err_t nrf24_rx_init();
86      void nrf24_rx_deinit();
87      void received_nrf24_pack_radio(size_t len);
88      void nrf24_set_rx_callback(void (*callback)(const char* data, size_t len));
89  #ifdef __cplusplus
90  }
91  #endif
92  #endif
93  //------------------------------------------------------------------------
```

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.

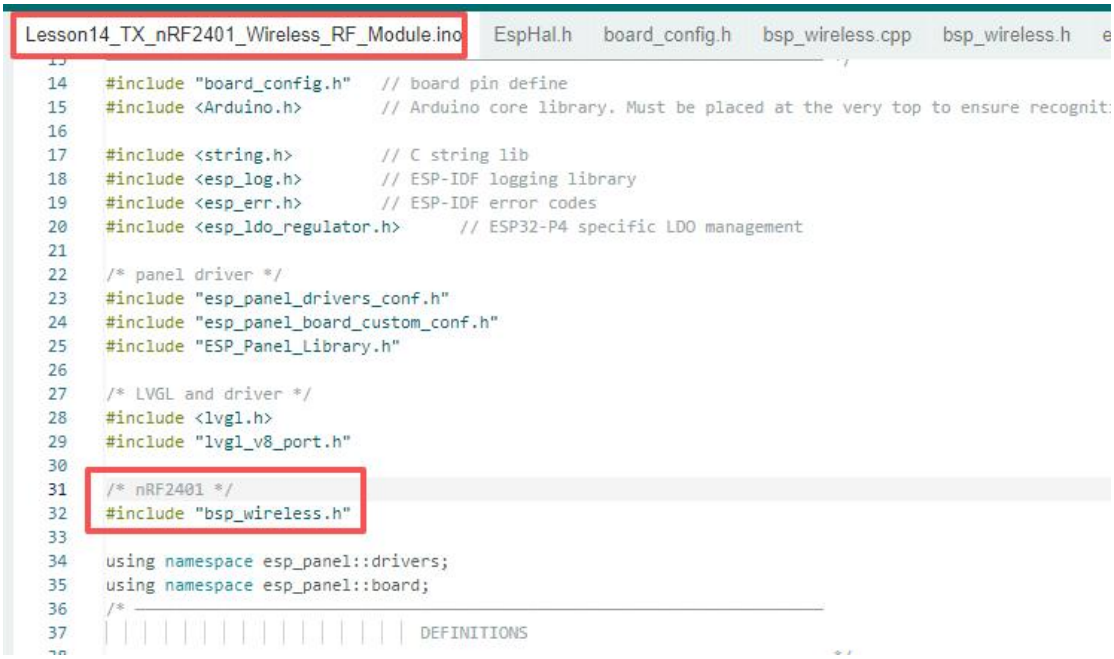
Internally, it creates a BSP_NRF2401 object and calls its member function Received_pack_radio(len) to read the data, log the results, and notify the upper-layer application via a callback.
Purpose: Provides an upper-layer interface to trigger the nRF2401 data reception processing flow.

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, you only need to include the library files in your functional code.



Sender Main Function

Let's first explain the transmitter's main function file, Lesson14_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: system power-on → initialize the serial port → configure the power supply LDO → initialize the display screen, touch screen and LVGL graphics system → initialize the nRF24L01 wireless module → create interface labels → create two parallel tasks (one responsible for wireless transmission and the other 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 with wireless transmission + real-time status display on a graphical interface.

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.

```
86  static void lvgl_show_counter_label_init(void)
87  {
88      if (lvgl_port_lock(0) != true) {
89          MAIN_ERROR("LVGL lock failed");
90          return;
91      }
92
93      lv_obj_t *screen = lv_scr_act();
94      lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);
95      lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);
96
97      s_hello_label = lv_label_create(screen);
98      if (s_hello_label == NULL) {
99          MAIN_ERROR("Create LVGL label failed");
100         lvgl_port_unlock();
101         return;
102     }
103     static lv_style_t label_style;
104     lv_style_init(&label_style);
105     lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
106     lv_style_set_text_color(&label_style, lv_color_black());
107     lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
108     lv_obj_add_style(s_hello_label, &label_style, LV_PART_MAIN);
109
110     lv_label_set_text(s_hello_label, "NRF24_TX_Hello World:0");
111     lv_obj_center(s_hello_label);
112
113     lvgl_port_unlock();
114 }
```

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.

```
116  static void ui_counter_task(void *param)
117  {
118      char text[48];
119      TickType_t last_wake_time = xTaskGetTickCount();
120      const TickType_t frequency = pdMS_TO_TICKS(1000); // 1 second = 1000ms
121
122      for (;;) {
123          uint32_t i = nrf24_get_tx_counter();
124          int n = snprintf(text, sizeof(text), "NRF24_TX_Hello World:%lu", (unsigned long)i);
125          (void)n;
126
127          if (lvgl_port_lock(0) == true) {
128              if (s_hello_label != NULL) {
129                  lv_label_set_text(s_hello_label, text);
130              }
131              lvgl_port_unlock();
132          }
133
134          // MAIN_INFO("NRF24 TX msg: %s", text);
135
136          // Use absolute time to ensure an exact one-second interval
137          vTaskDelayUntil(&last_wake_time, frequency);
138      }
139  }
```

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.

```
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);
    }
}
```

ldo_init

This code implements a power supply initialization function "ldo_init()", whose main role is to configure and enable the LDO (Low Dropout Regulator) channels inside the ESP32-P4 chip when the system starts up, providing stable operating voltages for the display interface and touch/I2C circuits.

```
void ldo_init()
{
    // --- Power Configuration (LDO3 for MIPI D-PHY) ---
    // ESP32-P4's MIPI D-PHY requires specific voltage to function.
    // LDO3 is typically routed to the MIPI power rail on P4 hardware.
    esp_err_t err = ESP_OK;
    esp_ldo_channel_handle_t ldo3_handle = NULL;
    esp_ldo_channel_config_t ldo3_cfg = {
        .chan_id = 3,           // LDO Channel 3
        .voltage_mv = 2500,     // Set to 2500mV (2.5V)
    };

    Serial.println("Initializing LDO3 to 2.5V...");
    err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
    if (err != ESP_OK) {
        Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
    } else {
        Serial.println("LDO3 Power enabled successfully.");
    }

    // --- Power Configuration (LDO4 for I2C/touch pull up) ---
    esp_ldo_channel_handle_t ldo4_handle = NULL;
    esp_ldo_channel_config_t ldo4_cfg = {
        .chan_id = 4,           // LDO Channel 4
        .voltage_mv = 3300,     // Set to 3300mV (3.3V)
    };

    Serial.println("Initializing LDO4 to 3.3V...");
    err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
    if (err != ESP_OK) {
        Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
    } else {
        Serial.println("LDO4 Power enabled successfully.");
    }
}
```

At the start of the function, a variable "err" of type "esp_err_t" is defined to store the error status after function execution, and a handle variable of type "esp_ldo_channel_handle_t" is also defined to represent the successfully acquired LDO channel control object.

```
158  void ldo_init()
159  {
160      // --- Power Configuration (LDO3 for MIPI D-PHY) ---
161      // ESP32-P4's MIPI D-PHY requires specific voltage to function.
162      // LDO3 is typically routed to the MIPI power rail on P4 hardware.
163      esp_err_t err = ESP_OK;
164      esp_ldo_channel_handle_t ldo3_handle = NULL;
```

First, the LDO3 power channel is configured. The program creates a structure "esp_ldo_channel_config_t ldo3_cfg", where "chan_id = 3" indicates selecting the LDO3 channel, and "voltage_mv = 2500" means setting the output voltage to 2500mV (2.5V).

```
165      esp_ldo_channel_config_t ldo3_cfg = {
166          .chan_id = 3,           // LDO Channel 3
167          .voltage_mv = 2500,     // Set to 2500mV (2.5V)
168      };
169
```

This voltage is mainly used to power the MIPI D-PHY display interface circuit, because in the hardware design of ESP32-P4, the MIPI display interface usually requires a dedicated power rail of approximately 2.5V to work properly. The program then outputs debugging information via "Serial.println()" to indicate that LDO3 is being initialized, and subsequently calls the "esp_ldo_acquire_channel()" function to apply for and enable the LDO channel. If the returned result "err" is not equal to "ESP_OK", it means the power supply configuration has failed. In this case, the program will output an error message using "Serial.printf()", and convert the error code into a human-readable string through "esp_err_to_name()". If successful, a prompt message is printed to indicate that the LDO3 power supply has been enabled successfully.

```
170      Serial.println("Initializing LDO3 to 2.5V...");
171      err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
172      if (err != ESP_OK) {
173          Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
174      } else {
175          Serial.println("LDO3 Power enabled successfully.");
176      }
```

Next, the program proceeds to configure the LDO4 power channel, following a process essentially identical to that of LDO3: first, it defines a handle "ldo4_handle" and a configuration structure "ldo4_cfg", where "chan_id = 4" indicates the selection of the LDO4 channel, and "voltage_mv = 3300" means setting the output voltage to 3300mV (3.3V).

```
178     // --- Power Configuration (LDO4 for I2C/touch pull up) ---
179     esp_ldo_channel_handle_t ldo4_handle = NULL;
180     esp_ldo_channel_config_t ldo4_cfg = {
181         .chan_id = 4,           // LDO Channel 4
182         .voltage_mv = 3300,     // Set to 3300mV (3.3V)
183     };
184
```

This voltage is mainly used to power the I2C bus and the pull-up resistors of the touch controller. For example, the common capacitive touch chip GT911 typically operates in a 3.3V voltage environment, so stable power supply support is required.

```
185     Serial.println("Initializing LDO4 to 3.3V...");
186     err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
187     if (err != ESP_OK) {
188         Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
189     } else {
190         Serial.println("LDO4 Power enabled successfully.");
191     }
192 }
```

The program also prints initialization information via the serial port and calls "esp_ldo_acquire_channel()" to enable LDO4. If the configuration fails, an error message is printed; if successful, a prompt is given to indicate that the power supply has been turned on normally.

display_touch_lvgl_init

This code implements an initialization function display_touch_lvgl_init() for the display screen, touch screen, and graphical interface system. Its main role is to complete the overall initialization of the display hardware, touch controller, and graphical interface library when the system starts up, enabling the screen to normally display a graphical interface and support touch input.

```
194 void display_touch_lvgl_init()
195 {
196     // --- Initialize Display and Touch Panel ---
197     Board *board = new Board();
198     // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
199     Serial.println("Initializing Panel (EK79007 + GT911)...");
200     assert(board->init());
201 #if LVGL_PORT_AVOID_TEARING_MODE
202     auto lcd = board->getLCD();
203     // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
204     lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
205 #endif
206     assert(board->begin());
207     Serial.println("Display and Touch system online.");
208
209     Serial.println("Initializing LVGL");
210     lvgl_port_init(board->getLCD(), board->getTouch());
211 }
```

At the start of the function, a Board class object is created via Board *board = new Board();. This class is derived from ESP_Panel_Library and is mainly used to encapsulate hardware resources such as the display screen, touch screen, and communication bus on the development board, allowing users to complete initialization and control through a unified interface.

```
194  void display_touch_lvgl_init()
195  {
196      // --- Initialize Display and Touch Panel ---
197      Board *board = new Board();
```

The program then outputs "Initializing Panel (EK79007 + GT911)..." via the serial port to indicate that the system is initializing the display and touch devices, and subsequently calls board->init() to complete the underlying hardware initialization. This step typically configures the MIPI-DSI display bus, GPIO, I2C touch bus, and related drivers, and loads specific hardware drivers such as the display control chip EK79007 and the capacitive touch control chip GT911.

```
198      // Initialize the bus (MIPI-DSI) and the devices (EK79007 & GT911)
199      Serial.println("Initializing Panel (EK79007 + GT911)...");
200      assert(board->init());
```

The function uses assert() for detection; if the initialization fails, the program will stop running directly to prevent subsequent program execution when the hardware has not been properly initialized.

Subsequently, the code includes a conditional compilation section #if LVGL_PORT_AVOID_TEARING_MODE. When the Tearing Prevention mode is enabled, the program obtains the LCD display object via board->getLCD() and calls configFrameBufferNumber() to set the number of frame buffers used by LVGL. This multi-buffering mechanism can reduce display tearing during refresh, thereby improving the stability and visual effect of screen display.

```
201  #if LVGL_PORT_AVOID_TEARING_MODE
202      auto lcd = board->getLCD();
203      // When avoid tearing function is enabled, the frame buffer number should be set in the board driver
204      lcd->configFrameBufferNumber(LVGL_PORT_DISP_BUFFER_NUM);
205  #endif
```

Next, the program calls board->begin() to complete the final startup process of the device. This function usually truly starts the display driver, initializes the touch controller, and puts the hardware into working state, with assert() also used to ensure successful execution.

After the initialization is completed, the program outputs "Display and Touch system online." via the serial port, indicating that the display and touch system is working normally.

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

Finally, the program starts initializing the graphical interface system, prints "Initializing LVGL" to indicate that the graphics library is about to be started, and then calls lvgl_port_init(board->getLCD(), board->getTouch()) to initialize the LVGL porting layer. This registers the LCD display device and touch input device with the LVGL system, enabling LVGL to refresh the screen through the LCD driver and receive user input events through the touch driver.

```
209      Serial.println("Initializing LVGL");
210      lvgl_port_init(board->getLCD(), board->getTouch());
211  }
```

setup

This code is the system initialization entry function setup() of the entire program, which is executed only once after the device is powered on or reset. Its main role is to sequentially complete serial port debugging, power supply configuration, display and touch system initialization, wireless module initialization, and multi-task creation, preparing the system to enter a normal operating state.

The program first calls Serial.begin(115200) to initialize serial communication (UART0) and sets the baud rate to 115200, which is used to output debugging information and system running logs, making it convenient for developers to observe the program execution status through the serial monitor. Next, it calls the ldo_init() function to configure the LDO voltage regulation module inside the ESP32-P4 chip, providing stable operating voltages for the display interface, touch circuits, and related peripherals.

```
214  void setup() {
215      // put your setup code here, to run once:
216
217      // Initialize the default Serial for debugging (UART0)
218      Serial.begin(115200);
219
220      ldo_init();
```

Subsequently, the program calls display_touch_lvgl_init() to initialize the display and touch system. This function starts the display driver and touch controller, and registers the display device with the graphical interface library LVGL, enabling the system to have graphical interface display capabilities.

```
222      display_touch_lvgl_init();
```

After completing the initialization of the display system, the program starts initializing the wireless communication module by calling nrf24_tx_init() to enable the transmission function of the wireless chip nRF24L01. If the return value of the function is not equal to ESP_OK, it indicates that the wireless module initialization has failed, and an error log is output through MAIN_ERROR() at this time; if the initialization is successful, a prompt message is printed through MAIN_INFO(), indicating that the wireless communication module is ready.

```
224      // nRF24L01 wireless init
225      esp_err_t err = nrf24_tx_init();
226      if (err != ESP_OK) {  // Check error
227          MAIN_ERROR("nRF24L01 Wireless Module init...");  // Handle failure
228      }
229      MAIN_INFO("The nRF24L01 wireless module initialization was successful.");
```

Then the program calls 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 outputs a prompt through the log indicating that the interface has been loaded successfully.

```
231     lvgl_show_counter_label_init();
232     MAIN_INFO("-------- LVGL Show OK ----------");
233
```

After completing all initializations, the program creates two FreeRTOS tasks through xTaskCreatePinnedToCore() and pins them to run on different CPU cores respectively:

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;

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. The priority of both tasks is set to configMAX_PRIORITIES - 5, ensuring they have the same priority in system scheduling, thus maintaining a relatively synchronized execution rhythm.

```
234     // Create tasks and use the same priority to ensure synchronization
235     xTaskCreatePinnedToCore(ui_counter_task, "ui_counter", 4096, NULL,
236                             configMAX_PRIORITIES - 5, NULL, 0);
237
238     xTaskCreatePinnedToCore(nrf24_tx_task, "nrf24_tx", 8192, NULL,
239                             configMAX_PRIORITIES - 5, NULL, 1);
240
241     MAIN_INFO("Tasks created, starting synchronized transmission...");
242 }
```

Finally, the program outputs the log information "Tasks created, starting synchronized transmission...", indicating that all system initializations are completed and the system starts to enter the multi-task operation phase.

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.

```
/**
 * @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.

```
157  static void nrf24_rx_task(void *param)
158  {
159      while (1) {  // Infinite loop for continuous checking
160          //  (Check if data has been received)
161          // Note: nRF24L01 doesn't have the same data received flag as SX1262
162          // We'll use a reasonable maximum packet size for nRF24L01 (32 bytes)
163          received_nrf24_pack_radio(32);  // Handle received packet data with maximum size
164          vTaskDelay(10 / portTICK_PERIOD_MS); // Check every 10ms to reduce CPU usage
165      }
166  }
```

setup:

This code is the system initialization function setup() of the program, which is executed only once after the device is powered on or reset. Its main role is to sequentially complete serial port debugging, power supply configuration, display and touch system initialization, wireless receiving module initialization, graphical interface creation, and reception task startup, thereby building an embedded system that can receive wireless data and display it on the screen interface.

```
223  void setup() {
224      // put your setup code here, to run once:
225
226      // Initialize the default Serial for debugging (UART0)
227      Serial.begin(115200);
228
229      ldo_init();
230
231      display_touch_lvgl_init();
232
233      // nRF24L01 Wireless RX init (nRF24L01 receiver initialization)
234      esp_err_t err = nrf24_rx_init();  // Initialize nRF24L01 receiver
235      if (err != ESP_OK) {
236          MAIN_ERROR("nRF24L01 Wireless Module RX init...");  // Halt if failed
237      }
238      MAIN_INFO("The nRF24L01 wireless module RX initialization was successful.");  // Log success
239
240      lvgl_show_rx_interface_init();  // Initialize LVGL user interface
241      MAIN_INFO("-------- LVGL RX Interface OK ----------");  // Log successful UI init
242
243      //  (Set callback function for received data)
244      nrf24_set_rx_callback(rx_data_callback);  // Register nRF24L01 RX callback function
245      MAIN_INFO("RX callback registered");  // Log callback registration success
246
247      // (Create nRF24L01 receiving task)
248      xTaskCreatePinnedToCore(nrf24_rx_task, "nrf24_rx", 4096, NULL,
249                              configMAX_PRIORITIES - 5, NULL, 1);  // Create FreeRTOS task pinned to core 1
250
251      MAIN_INFO("nRF24L01 RX receiver started, waiting for data...");  // Log start message
252  }
```

The program first calls Serial.begin(115200) to initialize serial communication (UART0) and sets the baud rate to 115200, which is used to output system logs and debugging

information, making it convenient for developers to observe the system operation status through the serial monitor. Subsequently, it calls the ldo_init() function to configure the LDO voltage regulation module inside the ESP32-P4 chip, providing stable voltages for the display interface, capacitive touch circuits, and I2C pull-up power supply, thus ensuring the normal operation of display and touch hardware.

```
223  void setup() {
224      // put your setup code here, to run once:
225
226      // Initialize the default Serial for debugging (UART0)
227      Serial.begin(115200);
228
229      ldo_init();
```

Next, the program calls display_touch_lvgl_init() to initialize the display screen and touch system. This function starts the display driver chip EK79007 and touch control chip GT911, and registers the display device and touch input with the graphical interface library LVGL, enabling the system to have complete graphical display and touch interaction capabilities.

```
231      display_touch_lvgl_init();
```

After completing the initialization of the display system, the program starts initializing the wireless communication module by calling nrf24_rx_init() to enable the reception mode of the wireless chip nRF24L01. This function usually configures parameters such as wireless frequency, communication channel, data rate, and receive buffer. If the return value is not equal to ESP_OK, it indicates that the wireless module initialization has failed, and the program will output an error log through MAIN_ERROR() at this time; if the initialization is successful, a prompt message is printed through MAIN_INFO(), indicating that the wireless receiving module is working normally.

```
233      // nRF24L01 Wireless RX init (nRF24L01 receiver initialization)
234      esp_err_t err = nrf24_rx_init();  // Initialize nRF24L01 receiver
235      if (err != ESP_OK) {
236          MAIN_ERROR("nRF24L01 Wireless Module RX init...");  // Halt if failed
237      }
238      MAIN_INFO("The nRF24L01 wireless module RX initialization was successful.");  // Log success
```

Next, the program calls lvgl_show_rx_interface_init() to create an LVGL graphical interface (RX reception interface). This interface usually includes text labels or information areas for displaying received wireless data or reception status, and outputs the log "-------- LVGL RX Interface OK ----------" to indicate that the interface initialization is
successful.

```
240      lvgl_show_rx_interface_init();  // Initialize LVGL user interface
241      MAIN_INFO("-------- LVGL RX Interface OK ----------");  // Log successful UI init
242
```

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, so that data processing logic is executed in the program, such as parsing data content, updating interface display, or recording reception information, and outputs the log "RX callback registered" to indicate successful callback registration.

```
243     //  (Set callback function for received data)
244     nrf24_set_rx_callback(rx_data_callback);  // Register nRF24L01 RX callback function
245     MAIN_INFO("RX callback registered");  // Log callback registration success
```

Finally, the program creates a FreeRTOS reception task nrf24_rx_task through xTaskCreatePinnedToCore(), allocates 4096 bytes of task stack space for it, and pins the task to run on CPU Core 1. The task priority is set to configMAX_PRIORITIES - 5, which ensures the task has a high scheduling priority, so it can continuously monitor the reception status of the wireless module and process the received data.

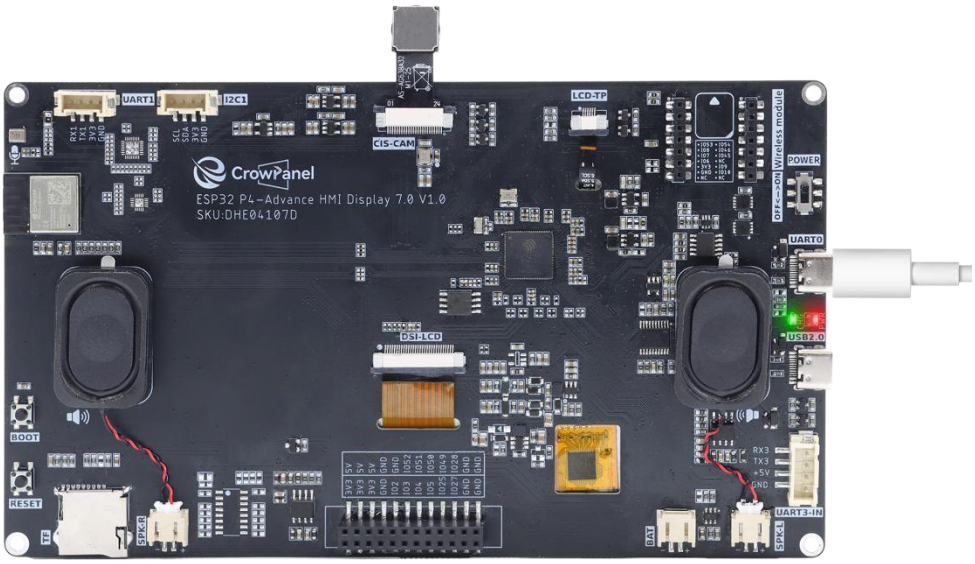
```
246
247     // (Create nRF24L01 receiving task)
248     xTaskCreatePinnedToCore(nrf24_rx_task, "nrf24_rx", 4096, NULL,
249                             configMAX_PRIORITIES - 5, NULL, 1);  // Create FreeRTOS task pinned to core 1
250
251     MAIN_INFO("nRF24L01 RX receiver started, waiting for data...");  // Log start message
252 }
```

After the task is created, the program outputs the log "nRF24L01 RX receiver started, waiting for data..." to indicate that the system has entered the wireless reception mode and started waiting for data to arrive.

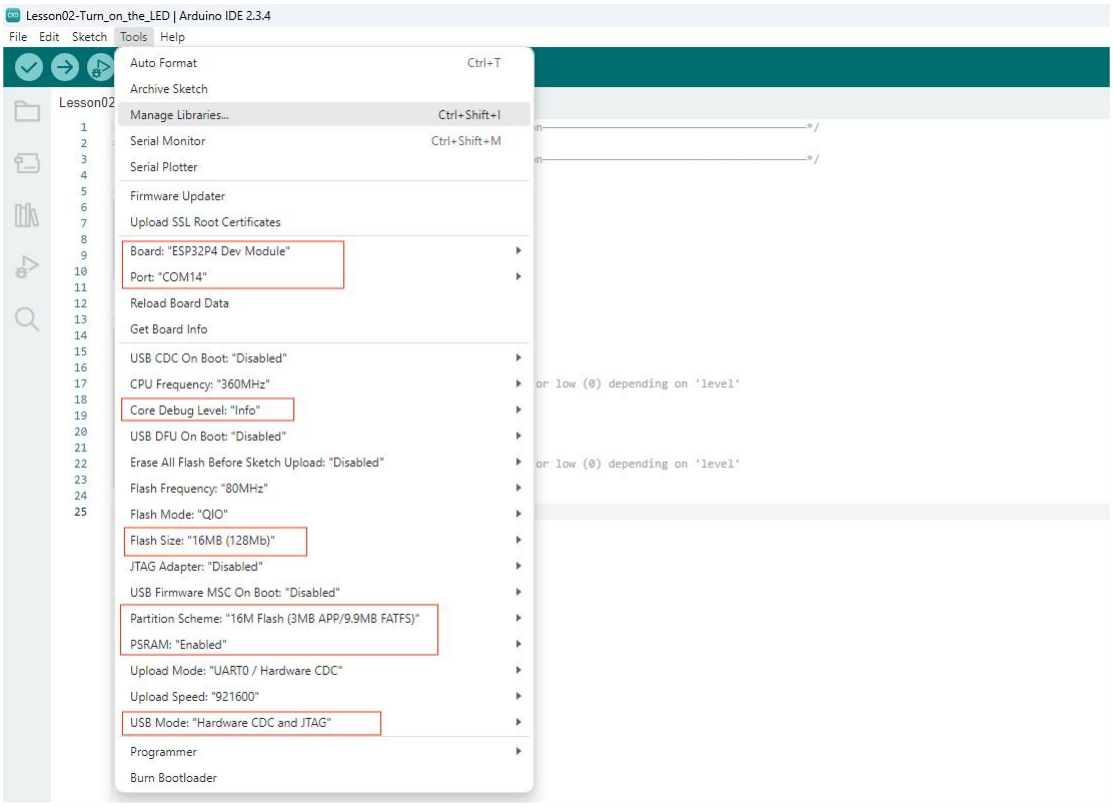
Programming Steps

Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.

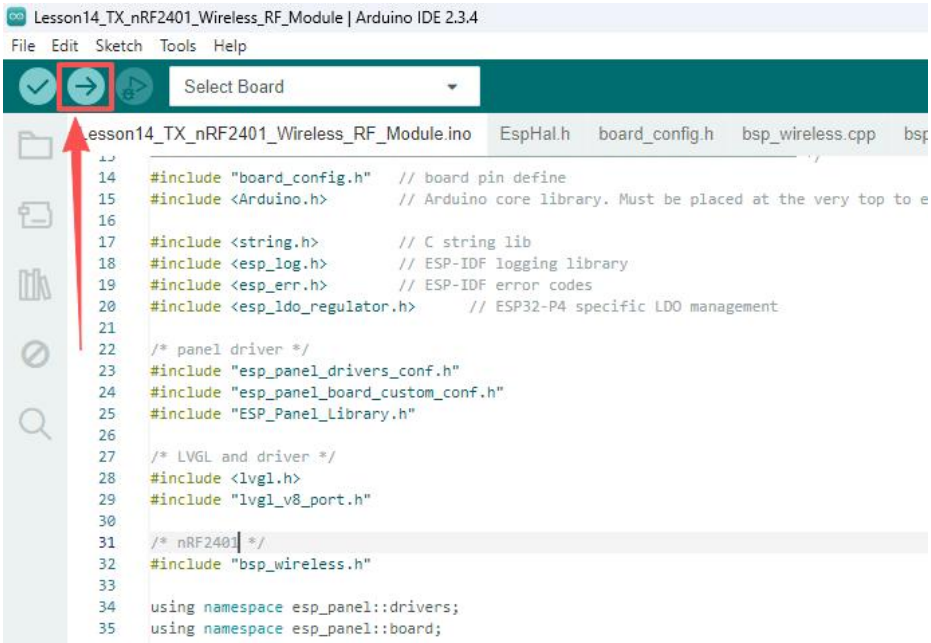
First, we connect the Advance-P4 device to our computer host via the USB cable.



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

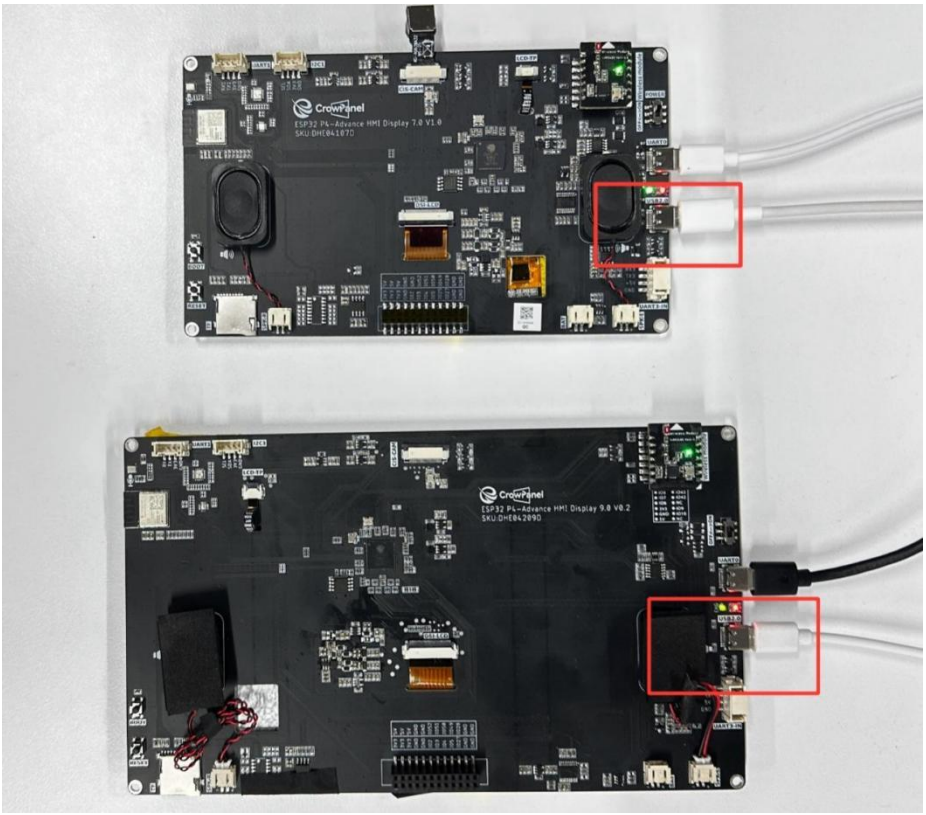


Then we compile and upload the code.

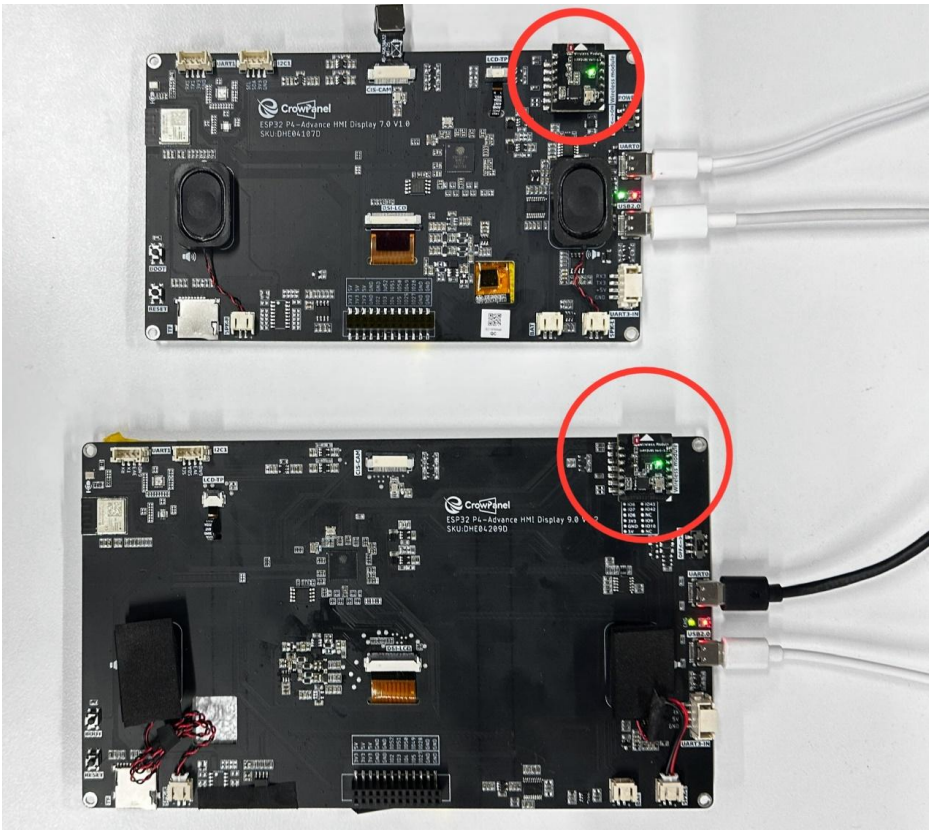


Wait for the code upload to complete.

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.


ELECROW
MAKE YOUR MAKING EASIER