

# CrowPanel Advanced 10.1inch ESP32- P4 HMI



## Table of contents

• Lesson 1 - Print "Hello World"	2
• Lesson 2 - Turn on the LED	11
• Lesson 3 - UART3-IN interface (external power supply)	26
• Lesson 4 - Serial port usage	29
• Lesson 5 - Touchscreen	50
• Lesson 6 - USB2.0	71
• Lesson 7 - Turn on the screen	87
• Lesson 8 - SD Card File Reading	105
• Lesson 9 - LVGL Lighting Control	126
• Lesson 10 - Temperature and Humidity	144
• Lesson 11 - Playback After Recording	160
• Lesson 12 - Playing Local Music from SD Card	177
• Lesson 13 - Camera Real-Time	203
• Lesson 14 - SX1262 Wireless Module	224
• Lesson 15 - nRF2401 Wireless RF Module	257
• Lesson 16 - Get weather temperature via WiFi	287
• Lesson 17 - WiFi Mode	317

# Lesson 01

## Print "Hello World"

### Introduction

---

In this class, we will officially learn to write code in the ESP-IDF environment to drive the Advance-P4 development board. The subsequent courses will follow a gradient design from simple to complex, helping you gradually master the ESP-IDF development framework and the usage logic of the ESP32-P4 chip, and establish a clear technical understanding. Specifically for this class, there are two core goals: First, to teach you how to create and burn a basic program in ESP-IDF, achieving the first "communication" between your computer and the ESP32-P4 chip on the Advance-P4 development board; second, to enable you to clearly see the "Hello World" information printed in real-time by the chip in the terminal window of the ESP-IDF tool, completing the crucial step from "configuring the environment" to "verifying the function".

### Hardware Used in This Lesson

---

This class does not involve the use of hardware. It is only to teach you how to create a new project and how to flash code to the ESP32-P4 chip on ESP-IDF.

### Operation Effect Diagram

---

- When running on the ESP32-P4, the serial terminal will output "Hello world" with an increasing counter every 1 second.



## Key Explanations

- First, let's talk about how to create a new project in the already installed ESP-IDF.
- Click on the ESP-IDF icon, then click "New project"



- Then a version of the ESP-IDF environment that you configured in the previous class will pop up.
- Select the 5.4.2 version that you previously set up.



- Then, enter this configuration interface. Here, fill in and set the name, path, target chip, serial port, and the folder name for the subsequent used component files of your newly created project.
- Finally, select the template.



- Choose ESP-IDF



- After selecting "Hello World", click "Confirm Creation" (you can also take a detailed look at the official introduction of this interface).



- Thus, we have successfully created the new project.



- Subsequently, we will modify the code based on this project and add the necessary components we need to use in the subsequent courses.
- Now, we can modify the `hello_world_main.c` function.
- Since in this class, I want to achieve the loop printing of "hello world:i" and continuously increment `i`, I deleted this sample code and replaced it with the code I wrote myself.



- Next, we will provide a detailed explanation of this code to help everyone have a clear understanding.
- When this code runs on the ESP32-P4, it outputs "Hello world" with an increasing counter every 1 second through the serial port. It utilizes the delay mechanism of FreeRTOS to achieve a non-blocking loop.

- The program first imports `stdio.h` to use `printf()` for outputting debugging information. Then, it includes `FreeRTOS.h` and `task.h`, allowing the use of task management and delay functions provided by FreeRTOS. Based on this, the main function uses `printf()` to print the content and controls the loop rhythm using `vTaskDelay()` to achieve outputting information every 1 second without blocking the operation of other system tasks.

```
main() {
1  #include <stdio.h>
2  #include "FreeRTOS.h"
3  #include "task.h"
4  int main() {
```

- In ESP-IDF, the entry function of the program is not `main()`, but `app_main()`.
- This function will be automatically called by the IDF framework.

Note: `app_main` is actually a FreeRTOS task (the main task), so you can write an infinite loop in it.

```
void app_main(void) {
1
2     int i = 0;
3     while (1) {
4         printf("Hello world: %d\n", i);
5         vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay 1 second
6     }
7 }
```

- `"i"` is a counter, with an initial value of 0.
- It increments after each loop.

```
void app_main(void) {
1
2     int i = 0;
3     while (1) {
4         printf("Hello world: %d\n", i);
5         vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay 1 second
6     }
7 }
```

- `printf("Hello world: %d\n", i++);`
- Output "Hello world: i" to the serial port.
- `i++`: First use the value of `i`, then increment `i` by 1.

```
void app_main(void) {
1
2     int i = 0;
3     while (1) {
4         printf("Hello world: %d\n", i++);
5         vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay 1 second
6     }
7 }
```

- `vTaskDelay(1000 / portTICK_PERIOD_MS)`: This function delays the current task for a certain period of time.
- Parameter explanation:
  - 1000: The duration of the delay (in milliseconds).
  - `portTICK_PERIOD_MS`: The number of milliseconds corresponding to one tick in the system.
- For example, if FreeRTOS is configured such that 1 tick = 1 ms, then  $1000 / 1 = 1000$  ticks = 1 second.
- Therefore, `vTaskDelay(1000 / portTICK_PERIOD_MS)`; is equivalent to delaying for 1 second.

```

1 #include <stdio.h>
2 #include "FreeRTOSConfig.h"
3 #include "FreeRTOS.h"
4
5 void app_main(void)
6 {
7     int i = 0;
8     while (1)
9     {
10        printf("Hello World\n");
11        vTaskDelay(1000 / portTICK_PERIOD_MS);
12    }
13 }

```

## Complete Code

---

Kindly click the link below to view the full code.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson01-Print\\_Hello\\_World](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson01-Print_Hello_World)

## Programming Steps

---

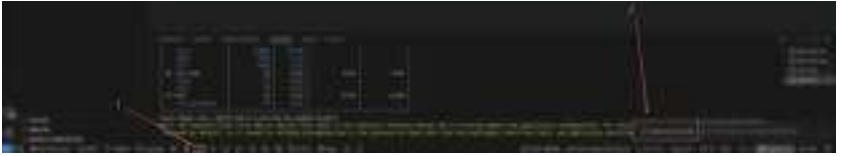
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable. (Connect UART0)



- In order of priority, select the ESP-IDF version 5.4.2 that you are currently using.
- We are using serial flash programming, so select UART.
- Since the serial port number displayed may vary depending on your device, after clicking 3, select the serial port that belongs to your own device.
- Make sure that you are using the esp32-p4 chip.



- After configuring 1, 2, 3, and 4 as mentioned above, we will proceed to compile the project to check if there are any issues with the code.
- First, click on 1 in the picture below, which represents the function of compiling the code.
- Wait for a while, after the code is compiled, you will be able to see the following information in the terminal, indicating that your code has been compiled successfully.



- Then, click the "Burn" button.



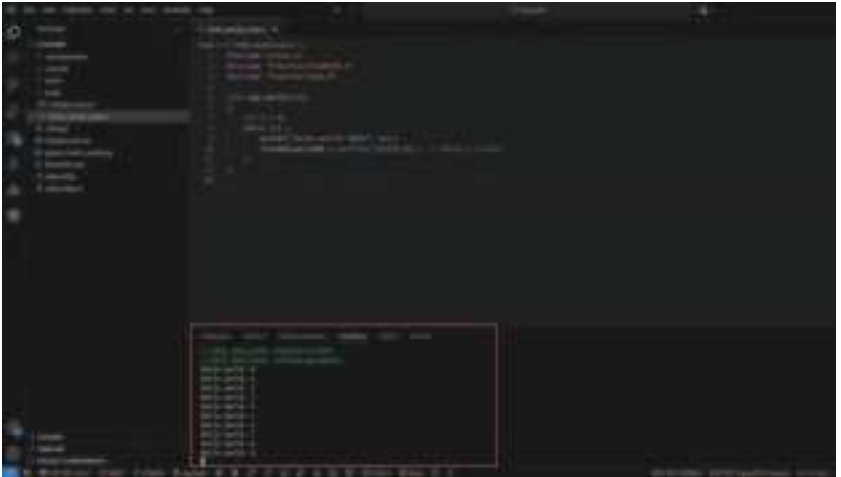
- After waiting for a while, you will be able to see from the displayed information on the output that the code has been uploaded successfully.



- Of course, you can also see from the upload process displayed on the terminal that your code has been uploaded successfully.



- Next, all you need to do is to open the serial port monitor, and then you will be able to see that "hello world" is being printed.



- So, that's all for this lesson. In the next class, we will gradually increase the difficulty level. We will teach you how to use components, how components are related to the main function, and how to have the main function utilize the interfaces within the components.

# Lesson 02

## Turn on the LED

### Introduction

---

In this class, we will start to explore the most important component in ESP-IDF.

In this class, we will use the `bsp_extra` component we have written ourselves to control the level of the UART1 interface on the Advance-P4, so 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 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.

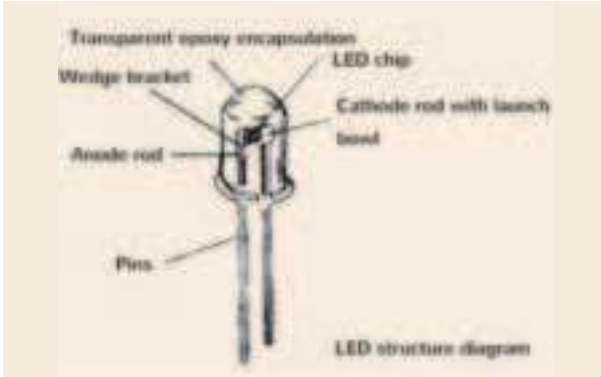
The GPIO functions of ESP32-P4 provide powerful hardware support for developers. In this chapter, we will delve into the application and configuration of GPIO through an example of lighting an LED.

## 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

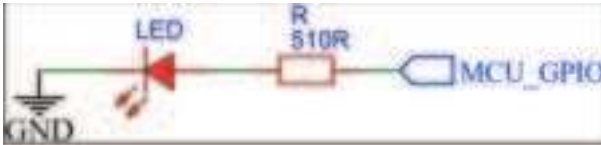
LED driving refers to providing appropriate current and voltage to LEDs through a stable power supply to ensure their normal lighting. The main driving methods for LEDs are constant current driving and constant voltage driving, among which constant current driving is more favored as it can limit the 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.

## Operation Effect Diagram

---

- 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.



## Key Explanations

---

Now let's talk about how the overall code framework is structured and connected after adding the `bsp_extra` component?

With this question in mind, let's explore it together.

First, click on the Github link below and download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson02-Turn\\_on\\_the%20LED](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson02-Turn_on_the%20LED)

Then, drag the code of this lesson into VS Code and open the project file.



The code in the subsequent courses will also be opened in this way.

From now on, there will be no further explanation on how to open the code.

- After opening it, you can see the framework of this project.



In the example of this class, a new folder named "bsp\_extra" was created under "LESSON02/peripheral". Inside the "bsp\_extra" folder, a new "include" folder, a "CMakeLists.txt" file, and a "kconfig" file were created.

The "bsp\_extra" folder contains the "bsp\_extra.c" driver file, and the "include" folder contains the "bsp\_extra.h" header file.

The "CMakeLists.txt" file integrates the driver into the build system, enabling the project to utilize the GPIO driver functionality.

The "kconfig" file loads the entire driver and GPIO pin definitions into the `sdkconfig` file within the IDF platform (which can be configured through the graphical interface).

## Initialization of GPIO code

- The GPIO source code consists of two files: "bsp\_extra.c" and "bsp\_extra.h".
- Next, we will first analyze the "bsp\_extra.h" program: it contains the relevant definitions and function declarations for GPIO pins.
- In this component, all the libraries we will use are placed in the "bsp\_extra.h" file, so they can be managed uniformly.

```
26 // ===== File Declaration ===== 27
// Include <string.h> library for string handling functions
#include <string.h>
// Include <stdio.h> library for basic printf, fprintf, etc.
#include <stdio.h>
// Include logging library for debug/info/error logs
#include "bsp_log.h"
// Include <stdint.h> library for basic integer types
#include <stdint.h>
// Include <stdbool.h> library for boolean types
#include <stdbool.h>
// ===== File Declaration ===== 28
```

- Below is the interface definition in the header file, which provides unified macros and function interfaces for the implementation file (.c).

```
void gpio_init_gpio48(); // Initialize GPIO48 pin
void gpio_init_gpio49(); // Initialize GPIO49 pin
void gpio_init_gpio50(); // Initialize GPIO50 pin
void gpio_init_gpio51(); // Initialize GPIO51 pin

// ===== GPIO48 Pin =====
// GPIO48 pin configuration and initialization
void gpio48_init();
// GPIO48 pin configuration and initialization
void gpio48_write(uint8_t data);
// GPIO48 pin configuration and initialization
void gpio48_read();
// ===== GPIO49 Pin =====
// GPIO49 pin configuration and initialization
void gpio49_init();
// GPIO49 pin configuration and initialization
void gpio49_write(uint8_t data);
// GPIO49 pin configuration and initialization
void gpio49_read();
// ===== GPIO50 Pin =====
// GPIO50 pin configuration and initialization
void gpio50_init();
// GPIO50 pin configuration and initialization
void gpio50_write(uint8_t data);
// GPIO50 pin configuration and initialization
void gpio50_read();
// ===== GPIO51 Pin =====
// GPIO51 pin configuration and initialization
void gpio51_init();
// GPIO51 pin configuration and initialization
void gpio51_write(uint8_t data);
// GPIO51 pin configuration and initialization
void gpio51_read();
// ===== File Declaration =====
```

- This is the content of "bsp\_extra.h" (which is also what needs to be done in every .h file).
- Next, we will analyze the code in the "bsp\_extra.c" file: including the initialization configuration and functional code for the LED pins.
- First, include the "bsp\_extra.h" that we just explained, so that we can use the macros and header files declared in "bsp\_extra.h".

```
1 // ===== File Declaration ===== 2
#include "bsp_extra.h" 3
// ===== File Declaration ===== 4
```

- The gpio\_extra\_init() function is used to configure GPIO48 of the ESP32-P4 as an output pin.

```
void gpio_extra_init()
{
    // ===== GPIO48 Pin =====
    // GPIO48 pin configuration and initialization
    gpio48_init();
    // ===== GPIO49 Pin =====
    // GPIO49 pin configuration and initialization
    gpio49_init();
    // ===== GPIO50 Pin =====
    // GPIO50 pin configuration and initialization
    gpio50_init();
    // ===== GPIO51 Pin =====
    // GPIO51 pin configuration and initialization
    gpio51_init();
}
// ===== File Declaration =====
```

- Define the return type: `esp_err_t`, which is the standard error code type of ESP-IDF.
- Variable `err`: Stores the return value of the function call, initially set to `ESP_OK` (success).
- `gpio_config_t gpio_config`: Prepare a configuration structure, which contains various settings for the pin.

- `.pin_bit_mask = (1ULL << 48) → Select GPIO48.`
- `.mode = GPIO_MODE_OUTPUT → Configure as output mode.`
- `.pull_up_en / .pull_down_en = false → Do not enable the internal pull-up/pull-down resistors.`
- `.intr_type = GPIO_INTR_DISABLE → Disable interrupts.`

- Call `gpio_config()` → Actually apply the configuration to the hardware.
- The `gpio_extra_set_level()` function is used to set the level (high or low) of this pin, thereby controlling external devices such as LEDs.

```

1 // gpio_extra_set_level()
2
3 #include "driver/gpio.h"
4
5 void gpio_extra_set_level(gpio_num_t pin, bool level) {
6     gpio_config_t config = {
7         .pin_bit_mask = (1ULL << pin),
8         .mode = GPIO_MODE_OUTPUT,
9         .pull_up_en = false,
10        .pull_down_en = false,
11        .intr_type = GPIO_INTR_DISABLE
12    };
13     gpio_config(&config);
14     gpio_set_level(pin, level);
15 }

```

- Function parameter `level`: Boolean value. True indicates a high level (1), and false indicates a low level (0).
- Call `gpio_set_level(48, level)`: Set GPIO48 to the corresponding level.

## CMkaLists.txt file

- The function of this example routine mainly relies on the `bsp_extra` driver. To successfully call the contents within the `bsp_extra` folder in the main function, a `CMakeLists.txt` file must be created and configured within the `bsp_extra` folder.
- The configuration details are as follows:

```

1 cmake_minimum_required(VERSION 3.5)
2
3 FILE(GLOB_RECURSE COMPONENT_SOURCES "*.c")
4
5 IDF_COMPONENT_REGISTER(SRC5 ${COMPONENT_SOURCES})
6
7 INCLUDE_DIRS "${INCLUDE_DIRS}"
8
9 REGISTERS_DRIVER

```

- In ESP-IDF, each component directory (such as peripheral) must have a `CMakeLists.txt` file, which mainly performs two tasks:

- Declaration of Source File

```
CMakeLists.txt
...
FILE(GLOB_RECURSE component_sources "${CMAKE_CURRENT_SOURCE_DIR}/src")
idf_component_register(SRCS ${component_sources}
                     INCLUDE_DIRS "include"
                     REQUIRES driver)
```

- SRCS specifies the .c files to be compiled within this component.
- INCLUDE\_DIRS specifies the paths of header files, allowing other components to #include.
- Define dependencies
- If your peripheral module needs to use the IDF library (such as a driver), write it in the REQUIRES section, for example:

```
idf_component_register(SRCS ${component_sources}
                     INCLUDE_DIRS "include"
                     REQUIRES driver)
```

- The "driver" library here is because we used the "gpio" library in the "bsp\_extra.h" file.

```
CMakeLists.txt
...
include("bsp_extra.h")
...
include("bsp_extra.h")
```

- "peripheral/CMakeLists.txt" is what tells ESP-IDF: which source files and header files are included in the peripheral component, as well as which libraries it depends on.
- If this file is missing, the code in the peripheral directory will not be compiled into your project.

Note: In the subsequent lessons, we will not start from scratch to create a new "CMakeLists.txt" file. Instead, we will make a few modifications to this existing file to incorporate more drivers into the build system.

## main

- The main folder is the core directory for program execution, and it contains the executable file main.c of the main function. Add the main folder to the CMakeLists.txt file of the build system.



```
void main()
{
    // Initialize GPIO
    gpio_extra_init();

    // Create task
    xTaskCreate(task_function, "Performance",
                1024, NULL, 1, NULL);

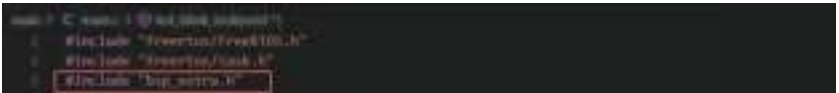
    while (1)
    {
        // Do nothing
    }
}
```

- This is the entry file of the entire application. In ESP-IDF, there is no int main(), but the program starts running from void app\_main(void).
- Let's first explain main.c.
- Introduce the types of FreeRTOS and the task APIs (such as xTaskCreate, vTaskDelay, etc.).



```
#include <freertos/freertos.h>
#include <freertos/task.h>
#include <bsp_extra.h>
```

- Our peripheral header files (placed in the "peripheral" component).
- "bsp\_extra.h" should declare interfaces such as gpio\_extra\_init() and gpio\_extra\_set\_level().



```
gpio_extra_init();
```

- Initialize GPIO (implemented in our peripheral/bsp\_extra)
- When explaining the "bsp\_extra.c" file, it was explained that here we can directly call it for use.



```
gpio_extra_set_level();
```

- Then it enters the while loop, causing the LED light to repeatedly turn on for one second and off for one second.
- Next, it calls the function for turning on or off the LED in the "bsp\_extra.c" file.
- Just by modifying parameter 1 or 0, it will take effect.
- 1: High level (on) 0: Low level (off)

```

100 void led_blink_task(void *pvParameters)
101 {
102     while(1)
103     {
104         gpio_write_pin_level(1);
105         vTaskDelay(pdMS_TO_TICKS(1000));
106
107         gpio_write_pin_level(0);
108         vTaskDelay(pdMS_TO_TICKS(1000));
109     }
110 }
  
```

- Then, the delay function from the FreeRTOS library is called to delay for one second.

```

104         gpio_write_pin_level(1);
105         vTaskDelay(pdMS_TO_TICKS(1000));
106
107         gpio_write_pin_level(0);
108         vTaskDelay(pdMS_TO_TICKS(1000));
  
```

- app\_main is the program entry point of ESP-IDF (called after system startup).
- In FreeRTOS, create a task named "led\_blink\_task", which will execute the led\_blink\_task function with a priority of 5 and using a 2048-byte stack to implement the LED blinking logic.

```

100 void led_blink_task(void *pvParameters)
101 {
102     while(1)
103     {
104         gpio_write_pin_level(1);
105         vTaskDelay(pdMS_TO_TICKS(1000));
106
107         gpio_write_pin_level(0);
108         vTaskDelay(pdMS_TO_TICKS(1000));
109     }
110 }
111
112 void app_main(void)
113 {
114     xTaskCreate(led_blink_task, "led_blink_task", 2048, NULL, 5, NULL);
  
```

- `xTaskCreate(led_blink_task, "led_blink_task", 2048, NULL, 5, NULL)`; Parameter meanings:
- `led_blink_task`: Entry function of the task
- `"led_blink_task"`: Task name (string)
- `2048`: Stack size of the task (on ESP-IDF, it is usually measured in bytes, and 2048 is a common value)
- `NULL`: Parameters passed to the task
- `5`: Task priority (5)
- `NULL`: Pointer to task handle (NULL should be passed if not needed)
- Now let's take a look at the `CMakeLists.txt` file in the main directory.
- The function of this CMake configuration is as follows:
- Collect all the `.c` source files in the `main/` directory as the source files for the component;
- Register the main component with the ESP-IDF build system and declare that it depends on the driver (an internal driver of ESP-IDF) and the custom component `bsp_extra`;
- This way, during the build process, ESP-IDF knows to build `bsp_extra` first, and then build `main`.



Note: In the subsequent courses, we will not start from scratch to create a new `CMakeLists.txt` file. Instead, we will make some minor modifications to this existing file to integrate other drivers into the main function.

## Complete Code

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson02-Turn\\_on\\_the%20LED](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson02-Turn_on_the%20LED)

## 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.



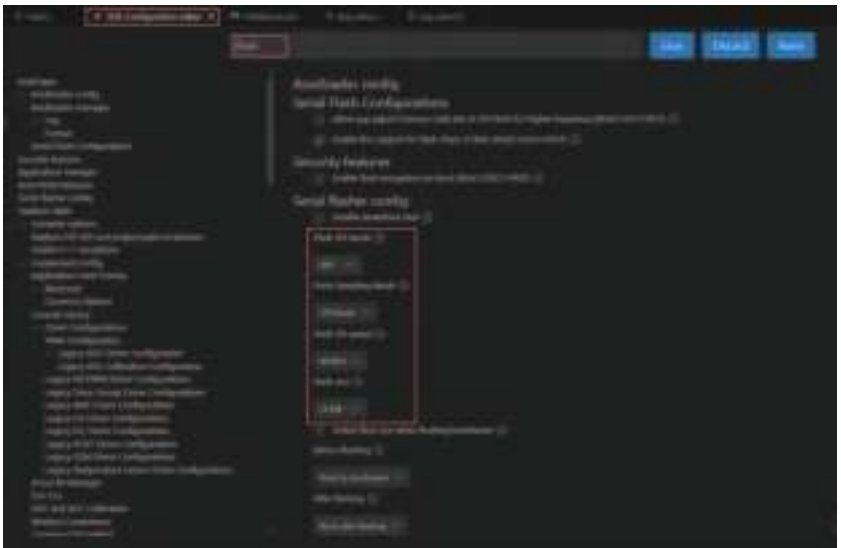
- Here, following the steps in the first section, we first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



- Wait for a moment for the loading process to complete, and then you can proceed with the relevant SDK configuration.



- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



- After the configuration is completed, remember to save your settings.



- After that, we will compile and burn the code (which was explained in detail in the first class)



- After waiting for a while, you will be able to see the LED connected to UART1 on your Advance-P4 turning on and off, remaining off for one second, and repeating this process over and over again.

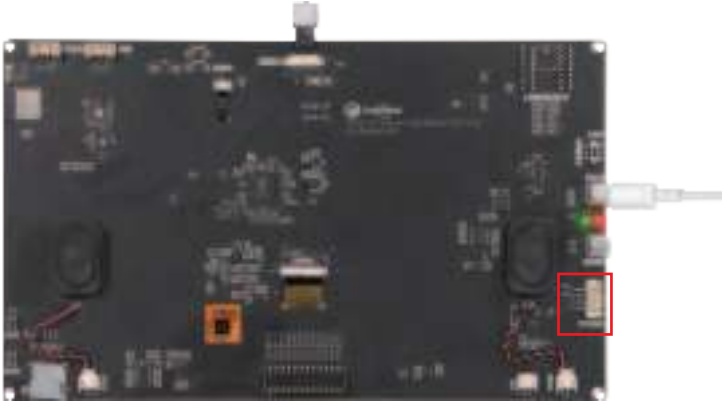
# Lesson 03

## 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.



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.

# Lesson 04

## 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.

### Hardware Used in This Lesson

---

#### The UART1 interface on the Advance-P4

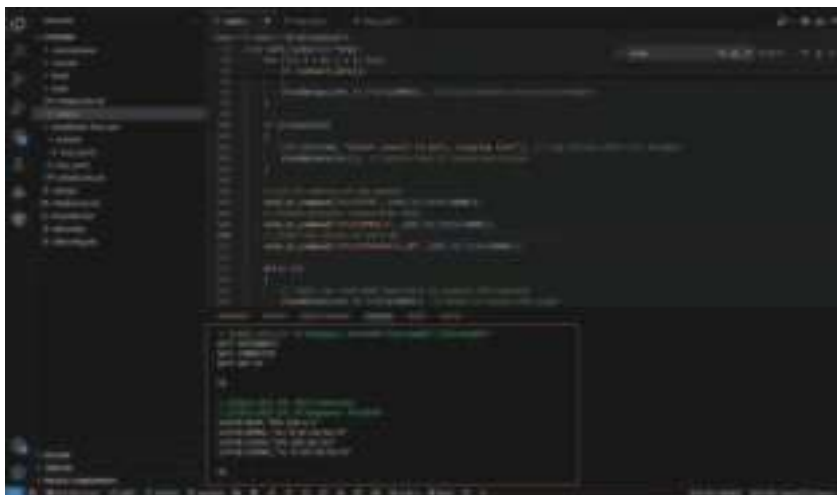




## Operation Effect Diagram

---

After running the code, you will be able to see the AT commands you sent on the monitor of ESP-IDF, as well as the responses returned to you by the Wi-Fi module via the serial port. (Green represents the Advance-P4's sending, and white represents the responses from the Wi-Fi module)



## Key Explanations

---

The main focus of this class is on how to use the serial port. Here, we will provide everyone with a new component called `bsp_uart`. This component is mainly used for initializing the serial port, configuring the serial port, and providing related interface usage. As you know, you can call the interfaces we have written at the appropriate time.

Next, we will focus on understanding the `bsp_uart` component.

First, click on the Github link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson04-Serial\\_port\\_usage](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson04-Serial_port_usage)

- Then, drag the code of this lesson into VS Code and open the project file.
- After opening it, you can see the framework of this project.



In the example of this class, a new folder named "bsp\_uart" was created under the "peripheral" directory. Inside the "bsp\_uart" folder, a new "include" folder and a "CMakeLists.txt" file were created.

The "bsp\_uart" folder contains the "bsp\_uart.c" driver file, and the "include" folder contains the "bsp\_uart.h" header file.

The "CMakeLists.txt" file will integrate the driver into the build system, enabling the project to utilize the serial communication functionality written in "bsp\_uart.c".

### Serial port communication code

- The driver code for serial port communication consists of two files: "bsp\_uart.c" and "bsp\_uart.h".
- Next, we will first analyze the "bsp\_uart.h" program.
- "bsp\_uart.h" is a header file for serial port communication, mainly used to:
  - Declare the functions, macros, and variables implemented in "bsp\_uart.c" for external programs to use
  - Enable other .c files to call this module simply by including "#include "bsp\_uart.h"
- In other words, it is the interface layer, exposing which functions and constants can be used externally, while hiding the internal details of the module.

- In this component, all the libraries we need to use are placed in the "bsp\_uart.h" file for centralized management.

```

1
2 // Standard C library for string manipulation
3 #include <string.h>
4 // Standard C input/output library (e.g., printf, scanf, etc.)
5 #include <stdio.h>
6 // FreeRTOS task management API
7 #include "FreeRTOS.h"
8 // FreeRTOS task API
9 #include "task.h"
10 // FreeRTOS queue API
11 #include "queue.h"
12 #include "bsp_uart.h"
13 // ...
14

```

- Then comes the declaration of the variables we need to use, as well as the declaration of the functions. The specific implementations of these functions are in "bsp\_uart.c".
- They are all uniformly placed in "bsp\_uart.h" for ease of calling and management. (When they are used in "bsp\_uart.c", we will understand their functions.)

```

15 // ...
16 // ...
17 // ...
18 // ...
19 // ...
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...
44 // ...
45 // ...
46 // ...
47 // ...
48 // ...
49 // ...
50 // ...
51 // ...
52 // ...
53 // ...
54 // ...
55 // ...
56 // ...
57 // ...
58 // ...
59 // ...
60 // ...
61 // ...
62 // ...
63 // ...
64 // ...
65 // ...
66 // ...
67 // ...
68 // ...
69 // ...
70 // ...
71 // ...
72 // ...
73 // ...
74 // ...
75 // ...
76 // ...
77 // ...
78 // ...
79 // ...
80 // ...
81 // ...
82 // ...
83 // ...
84 // ...
85 // ...
86 // ...
87 // ...
88 // ...
89 // ...
90 // ...
91 // ...
92 // ...
93 // ...
94 // ...
95 // ...
96 // ...
97 // ...
98 // ...
99 // ...
100 // ...

```

- We can see that there are two sets of serial port pins here. The first set is UART\_IN, which are the TX and RX pins of the UART3-IN interface, as shown in the figure. (This was not used in this lesson. We provided these pins to facilitate your future use. However, it should be noted that this interface cannot supply external power.)

```

101 // ...
102 // ...
103 // ...
104 // ...
105 // ...
106 // ...
107 // ...
108 // ...
109 // ...
110 // ...
111 // ...
112 // ...
113 // ...
114 // ...
115 // ...
116 // ...
117 // ...
118 // ...
119 // ...
120 // ...
121 // ...
122 // ...
123 // ...
124 // ...
125 // ...
126 // ...
127 // ...
128 // ...
129 // ...
130 // ...
131 // ...
132 // ...
133 // ...
134 // ...
135 // ...
136 // ...
137 // ...
138 // ...
139 // ...
140 // ...
141 // ...
142 // ...
143 // ...
144 // ...
145 // ...
146 // ...
147 // ...
148 // ...
149 // ...
150 // ...

```



- The other group is the UART1 interface used in this class. As we mentioned before, this interface can not only be used as a regular GPIO port, but also as a serial port. This class will be using this interface.

```
21 #define UART1_EXTRA_GPIO_IDX 47 // Define GPIO number 47 as UART1 TX pin
24 #define UART1_EXTRA_GPIO_RXD 48 // Define GPIO number 48 as UART1 RXD pin
```

- Let's take a look at "bsp\_uart.c" again, and see what each function specifically does.
- bsp\_uart: The bsp\_uart component encapsulates the ESP32 UART hardware and provides unified interfaces for initialization, data transmission, reception, and status management, shielding the details of the underlying driver, enabling upper-layer tasks (such as WiFi AT control tasks) to communicate with external devices through UART stably and reliably.

**Then the following functions are the interfaces we call to implement screen display.**

**uart\_init():**

- This function is responsible for initializing UART2 of ESP32-P4 and configuring its communication parameters, including baud rate, data bits, stop bits, parity bits, and flow control mode. It also installs the UART driver and specifies the TX/RX pins.
- By encapsulating the underlying uart\_driver\_install(), uart\_param\_config(), and uart\_set\_pin(), it shields the hardware details, allowing the upper-layer tasks to not need to worry about the cumbersome operations of UART initialization.

- After calling this function, the UART hardware is ready and can perform data transmission and reception. It is usually called during system startup or before communication is needed.
- There are a total of 3 serial port interfaces on our Advance-P4, namely UART0, UART1, and UART3-IN.
- UART0 is our default interface for power supply and uploading code. By default, it is UART\_NUM\_0.
- Then there are UART\_NUM\_1 and UART\_NUM\_2 left.
- Here, we can choose either of these two ports as we like, because we only use one serial port interface here. So I choose UART\_NUM\_2.
- If you also use the UART3-IN interface, make sure that the port number and pin you bind correspond and do not conflict.

```

#include <uart.h>
#include <pin.h>

#define UART_NUM_2
#define UART_PIN_TX
#define UART_PIN_RX

void setup() {
  uart_init(UART_NUM_2);
  pin_mode(UART_PIN_TX, OUTPUT);
  pin_mode(UART_PIN_RX, INPUT);
}

void loop() {
  // ...
}

```

### SendData(const char \*data):

- This function is used to send string data to UART2. It first calculates the length of the string, and then calls `uart_write_bytes()` to send the data to the UART hardware. The function returns the actual number of bytes sent, which is convenient for the upper layer to determine whether the transmission was successful. It encapsulates the underlying driver interface, allowing upper-level tasks or modules to safely send commands or data by simply calling `SendData()`, without having to handle the buffer and byte length every time.

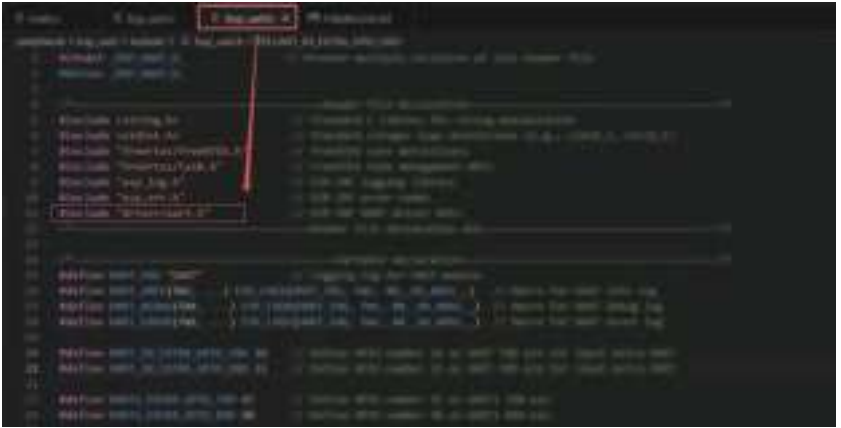
```
13 int main() { // Function to send a string of data through UART
14
15     // Create the string to be sent
16     char str[100] = "ESP8266"; // Set the length of the string
17     int len = strlen(str); // Get the length of the string
18     // Send the string to the UART
19     uart_write_bytes(UART_0, str, len); // Send string to UART
20     return 0; // Return status of the program
21 }
```

```
1 // UART peripheral
2 #include <driver/uart.h>
3 #include <string.h>
4
5 // UART pin configuration
6 #define UART_0_PIN_NUM 1 // TX pin
7 #define UART_0_PIN_NUM 2 // RX pin
8
9 // UART peripheral configuration
10 #define UART_0_BAUD_RATE 115200 // Baud rate
11 #define UART_0_DATA_BITS 8 // Data bits
12 #define UART_0_STOP_BITS 1 // Stop bits
13 #define UART_0_PARITY 0 // Parity
14 #define UART_0_TX_PIN 1 // TX pin
15 #define UART_0_RX_PIN 2 // RX pin
16
17 // UART peripheral initialization
18 void uart_init() {
19     // Initialize the UART peripheral
20     uart_config_t uart_config = {
21         .baud_rate = UART_0_BAUD_RATE,
22         .data_bits = UART_0_DATA_BITS,
23         .stop_bits = UART_0_STOP_BITS,
24         .parity = UART_0_PARITY,
25         .tx_pin = UART_0_TX_PIN,
26         .rx_pin = UART_0_RX_PIN,
27     };
28     uart_driver_install(UART_0, 1024, 0, 0, &uart_config, 0);
29 }
30
31 // UART peripheral write function
32 void uart_write(const char* str) {
33     int len = strlen(str);
34     uart_write_bytes(UART_0, str, len);
35 }
36
37 int main() {
38     // Initialize the UART peripheral
39     uart_init();
40     // Send a string of data through UART
41     uart_write("ESP8266");
42     return 0;
43 }
```

- That's all about the components of bsp\_uart. Just make sure you know how to call these interfaces.
- Then, if we need to make a call, we must also configure the "CMakeLists.txt" file located in the "bsp\_uart" folder.
- This file is placed in the "bsp\_uart" folder and its main function is to inform the build system (CMake) of ESP-IDF: how to compile and register the "bsp\_uart" component.

```
1 cmake_minimum_required(VERSION 3.10)
2 project("bsp_uart")
3
4 # The following line indicates the location of the component source files
5 set(COMPONENT_SRCS "bsp_uart.c")
6
7 # Register the component for linking
8 register_component(COMPONENT_SRCS)
9
10 # Add the component to the project
11 add_subdirectory("${CMAKE_CURRENT_SOURCE_DIR}")
```

- The reason why this is called "driver" is that we have called it in the "bsp\_uart.h" file (for other libraries that are system libraries, there is no need to add anything).



The screenshot shows a code editor with a dark theme. A red box highlights a function call in a header file, and a red arrow points from this call to the function's definition in another file. The code includes various system headers and defines macros for hardware registers and functions.

## Main function

- The main folder is the core directory for program execution, and it contains the executable file main.c for the main function.
- Add the main folder to the "CMakeLists.txt" file of the build system.



The screenshot shows a code editor with a dark theme. A blue box highlights the definition of the `app_main` function in a file named `main.c`. The function is declared as `void app_main(void)` and contains several lines of code, including comments and function calls.

- This is the entry file for the entire application. In ESP-IDF, there is no `int main()`, but the program starts running from `void app_main(void)`.

- In the ESP-IDF framework, `app_main()` is the main entry point of the entire program, equivalent to the `main()` function in standard C.
- When the ESP32-P4 powers on or restarts, the system will execute `app_main()` to start the user tasks and application logic.
- Let's explain `main.c`
- Function: Calls the interfaces in the `bsp_uart` component to allow the FreeRTOS scheduler to run the `wifi_task`, and send AT commands to control the wifi module to connect to the wifi.

### "`bsp_uart.h`":

This file imports the custom UART encapsulation component "`bsp_uart`", providing interfaces such as UART initialization, data transmission, data reception, and status management, enabling upper-layer tasks to conveniently communicate with external devices via UART.

### `#include "freertos/FreeRTOS.h"`:

This file imports the basic header file of the FreeRTOS kernel, providing basic operation functions and type definitions such as task scheduling, time management, semaphores, and queues, which are necessary for using FreeRTOS.

### `#include "freertos/task.h"`:

This file imports the interfaces related to task management in FreeRTOS, including functions such as `xTaskCreate()` for creating tasks, `vTaskDelay()` for task delay, and `vTaskDelete()` for task deletion, used for multi-task scheduling and management.

### `#include "string.h"`:

This file imports the string processing functions of the C standard library, such as `strlen()`, `strstr()`, and `snprintf()`, for string length calculation, substring search, and string formatting operations.

### `#include "esp_log.h"`:

This file imports the logging system interface provided by ESP-IDF, used for printing debug information, error information, and system status. It provides functions such as `ESP_LOGI()`, `ESP_LOGE()`, and `ESP_LOGD()`.

```
main > C main: > ..
1  #include "bsp_uart.h"
2  #include "freertos/FreeRTOS.h"
3  #include "freertos/task.h"
4  #include "string.h"
5  #include "esp_log.h"
```

- The name (SSID) of the WiFi was defined, which is used in the program to construct AT commands to enable the module to connect to the specified WiFi network.
- The password (Password) of the WiFi was also defined, which, along with the SSID, is used in the AT commands to connect to the WiFi network.

```

7 #define WIFI_SSID "elecrow888" // WiFi network name
8 #define WIFI_PASS "elecrow888" // WiFi network password

```

- Define a constant to represent the maximum length of the buffer for receiving AT command responses, which is 512 bytes. This ensures that the received data will not exceed the boundary.
- Define a static string as the log tag (Tag), which is used by log functions such as ESP\_LOGI() and ESP\_LOGE() to distinguish the outputs of different modules, facilitating debugging and problem location.

```

10 #define AT_BUFFER_SIZE 512 // Max size length for AT command responses
11 static const char *TAG = "WIFI AT"; // Tag for logging messages

```

### uart\_read\_response(char \*buffer, size\_t len, TickType\_t timeout):

- The "uart\_read\_response()" function is the core function in the bsp\_uart component for receiving data from the UART. It repeatedly calls the ESP32's "uart\_read\_bytes()" interface to store the data received by UART2 into the buffer provided by the user. It also supports timeout control.
- The function accumulates the actual received bytes each time it reads and adds '\0' at the end of the buffer to ensure that the returned data is a valid C string. It not only prevents buffer overflow but also continuously waits for data within the specified time, making it suitable for reading AT command responses or other data returned by external devices. This enables upper-level tasks to safely and reliably obtain the received data without directly operating the underlying UART driver.

```

14 #include <string.h>
15 #include <stdio.h>
16 #include <esp_log.h>
17 #include <esp_err.h>
18 #include <esp_system.h>
19 #include <driver/uart.h>
20 #include <freertos/FreeRTOS.h>
21 #include <freertos/task.h>
22 #include <freertos/timers.h>
23 #include <freertos/queue.h>
24 #include <freertos/semphr.h>
25 #include <freertos/event_groups.h>
26 #include <freertos/queue.h>
27 #include <freertos/semphr.h>
28 #include <freertos/event_groups.h>
29 #include <freertos/queue.h>
30 #include <freertos/semphr.h>
31 #include <freertos/event_groups.h>
32 #include <freertos/queue.h>
33 #include <freertos/semphr.h>
34 #include <freertos/event_groups.h>
35 #include <freertos/queue.h>
36 #include <freertos/semphr.h>
37 #include <freertos/event_groups.h>
38 #include <freertos/queue.h>
39 #include <freertos/semphr.h>
40 #include <freertos/event_groups.h>
41 #include <freertos/queue.h>
42 #include <freertos/semphr.h>
43 #include <freertos/event_groups.h>
44 #include <freertos/queue.h>
45 #include <freertos/semphr.h>
46 #include <freertos/event_groups.h>
47 #include <freertos/queue.h>
48 #include <freertos/semphr.h>
49 #include <freertos/event_groups.h>
50 #include <freertos/queue.h>
51 #include <freertos/semphr.h>
52 #include <freertos/event_groups.h>
53 #include <freertos/queue.h>
54 #include <freertos/semphr.h>
55 #include <freertos/event_groups.h>
56 #include <freertos/queue.h>
57 #include <freertos/semphr.h>
58 #include <freertos/event_groups.h>
59 #include <freertos/queue.h>
60 #include <freertos/semphr.h>
61 #include <freertos/event_groups.h>
62 #include <freertos/queue.h>
63 #include <freertos/semphr.h>
64 #include <freertos/event_groups.h>
65 #include <freertos/queue.h>
66 #include <freertos/semphr.h>
67 #include <freertos/event_groups.h>
68 #include <freertos/queue.h>
69 #include <freertos/semphr.h>
70 #include <freertos/event_groups.h>
71 #include <freertos/queue.h>
72 #include <freertos/semphr.h>
73 #include <freertos/event_groups.h>
74 #include <freertos/queue.h>
75 #include <freertos/semphr.h>
76 #include <freertos/event_groups.h>
77 #include <freertos/queue.h>
78 #include <freertos/semphr.h>
79 #include <freertos/event_groups.h>
80 #include <freertos/queue.h>
81 #include <freertos/semphr.h>
82 #include <freertos/event_groups.h>
83 #include <freertos/queue.h>
84 #include <freertos/semphr.h>
85 #include <freertos/event_groups.h>
86 #include <freertos/queue.h>
87 #include <freertos/semphr.h>
88 #include <freertos/event_groups.h>
89 #include <freertos/queue.h>
90 #include <freertos/semphr.h>
91 #include <freertos/event_groups.h>
92 #include <freertos/queue.h>
93 #include <freertos/semphr.h>
94 #include <freertos/event_groups.h>
95 #include <freertos/queue.h>
96 #include <freertos/semphr.h>
97 #include <freertos/event_groups.h>
98 #include <freertos/queue.h>
99 #include <freertos/semphr.h>
100 #include <freertos/event_groups.h>

```

## send\_at\_command(const char \*cmd, TickType\_t timeout):

- The "send\_at\_command()" function is a high-level wrapper function in the "bsp\_uart" component, used to send commands to the AT module and wait for a response.
- It first sends the AT instruction passed by the user to the UART using the "SendData()" function, and then sends a carriage return and line feed character as the command terminator; then it calls "uart\_read\_response()" to read the data returned by the module and save it in the buffer, while also printing the log for debugging purposes.
- The function checks if the returned string contains "OK". If it does, it means the command execution was successful and returns "true"; otherwise, it returns "false" indicating a command failure.

```
1 (UART) 0x10000000: 0x00000000
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

- This function encapsulates the complete process of sending, receiving and result judgment, enabling the upper-level tasks to safely and simply operate the AT module through a single interface, without having to deal with the details of the underlying UART reading and writing as well as response parsing.

```
35 // Send AT command and wait for its response.
36 static bool send_at_command(const char *cmd, TickType_t timeout)
37 {
38     char response[4096]; // Buffer to store response
39     SendData(cmd); // Send the AT command
40     SendData("\r\n"); // Send command terminator
41
42     uart_read_response(response, 4096, timeout); // Read response
43     log_printf(TAG, "AT Response: %s", response); // Log the response
44
45     // Check if response contains "OK"
46     if (strstr(response, "OK") != NULL)
47         return true; // Command successful
48     else
49         return false; // Command failed
50 }
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

## connect\_wifi():

- The "connect\_wifi()" function is a high-level encapsulation function used to enable the ESP32 to connect to a specified WiFi network through the AT module.
- First, it builds the AT command for connecting to WiFi, "AT+CWMODE=1, 'SSID', 'PASSWORD'", in a 128-byte buffer and prints a log message indicating that the WiFi name is being attempted to connect.
- Then, it calls the "send\_at\_command()" function to send the command and waits for the module's response, setting the timeout to 5 seconds.
- The function determines whether the connection was successful based on the response result: if the response is "OK", it prints the "WiFi Connected" log and returns true; if the connection was not successful, it prints an error log and returns false.
- This function encapsulates the complete process from building the AT command, sending the command to judging the connection result, allowing the upper-level tasks to directly call it to achieve WiFi connection without handling the underlying UART and command parsing details.

```
40: // WiFi connection function
41: static bool connect_wifi()
42: {
43:     char cmd[128]; // buffer to build AT command
44:
45:     // construct AT command to join WiFi network
46:     sprintf(cmd, "AT+CWMODE=1,\"%s\",\"%s\"", WiFi_SSID, WiFi_PASS);
47:     LOG_I(100, "Connecting to WiFi: %s", WiFi_SSID); // log connection attempt
48:
49:     // send command with 5-second timeout and return result
50:     if (send_at_command(cmd, 5000, 10000))
51:     {
52:         LOG_I(100, "WiFi Connected"); // log successful connection
53:         return true;
54:     }
55:     else
56:     {
57:         LOG_I(100, "Failed to connect WiFi"); // log connection failure
58:         return false;
59:     }
60: }
```

## wifi\_task(void \*arg):

- This function calls all the interfaces we discussed earlier.
- The function `wifi_task()` is a FreeRTOS task that communicates with the AT WiFi module via UART to achieve WiFi connection and initialization of the TCP server.
- The task first initializes the UART; if it fails, it deletes itself to ensure system stability;

```
18 void wifi_task(void *arg)
19 {
20     // Initialize UART communication
21     if (uart_init() != ESP_OK)
22     {
23         LOG_ERROR("UART init failed"); // Log UART initialization failure
24         vTaskDelete(NULL); // Delete current task if initialization fails
25         return;
26     }
27
28     // Configure module to AT mode using AT+MODEM=1 command
29     send_at_command("AT+MODEM=1", uart_0, 1000, 1000);
30
31     // Reset the module to AT mode
32     send_at_command("AT+RESTORE", uart_0, 1000, 1000);
33     vTaskDelay(1000); // Delay to allow module to restart
34
35     // Attempt to connect to WiFi, timeout = 100ms
36     send_command("AT+WiFi", uart_0);
37     for (int i = 0; i < 10; i++)
38     {
39         if (connect_wifi())
40         {
41             LOG_SUCCESS("WiFi connected successfully");
42             break;
43         }
44         vTaskDelay(100); // Delay between connection attempts
45     }
46
47     if (!connected)
48     {
49         LOG_ERROR("Failed connect to WiFi, stopping task"); // Log failure after all attempts
50         vTaskDelete(NULL); // Delete task if connection fails
51     }
52
53     // Set IP address on the module
54     send_at_command("AT+CWMODE=1", uart_0, 1000, 1000);
55     // Enable DHCP mode
56     send_at_command("AT+CWDHCP=1", uart_0, 1000, 1000);
57     // Wait 100 ms before the next step
58     send_at_command("AT+CWMODE=0", uart_0, 1000, 1000);
59
60     wifi = 1;
61
62     // Delay 100 ms after WiFi mode has been set properly
63     vTaskDelay(100); // Delay to avoid CPU usage
64 }
```

- Then set the module to the AP + STA mode and reset it to make the configuration take effect.

```

11 void WiFiManager::WiFiMode()
12 {
13     // WiFi module initialization
14     WiFi.mode(WIFI_AP_STA);
15
16     // Log module initialization progress
17     vTaskDelay(500); // Delay before task start
18     return;
19 }
20
21 // Connect to WiFi with timeout (seconds) = timeout
22 void WiFiManager::WiFiConnect(WiFiMode mode, uint8_t timeout)
23 {
24     // Start the WiFi module
25     WiFi.mode(WIFI_AP_STA);
26     vTaskDelay(500); // Delay to allow module to connect
27
28     // Attempt to connect to WiFi, timeout 5 times
29     bool connected = false;
30     for (int i = 0; i < 5; i++)
31     {
32         // connect_wifi()
33
34         connected = true; // Mark as connected if successful
35         break;
36     }
37     vTaskDelay(2000); // Delay between connection attempts
38 }

```

- Then, the process will repeatedly attempt to connect to the specified WiFi, up to 5 times. Each failure will cause a 2-second delay. If the connection is still unsuccessful in the end, an error message will be printed and the task will be deleted.

```

11 void WiFiManager::WiFiMode()
12 {
13     // WiFi module initialization
14     WiFi.mode(WIFI_AP_STA);
15
16     // Log module initialization progress
17     vTaskDelay(500); // Delay before task start
18     return;
19 }
20
21 // Connect to WiFi with timeout (seconds) = timeout
22 void WiFiManager::WiFiConnect(WiFiMode mode, uint8_t timeout)
23 {
24     // Start the WiFi module
25     WiFi.mode(WIFI_AP_STA);
26     vTaskDelay(500); // Delay to allow module to connect
27
28     // Attempt to connect to WiFi, timeout 5 times
29     bool connected = false;
30     for (int i = 0; i < 5; i++)
31     {
32         // connect_wifi()
33
34         connected = true; // Mark as connected if successful
35         break;
36     }
37     vTaskDelay(2000); // Delay between connection attempts
38 }
39
40 // Connected
41 bool WiFiManager::isConnected()
42 {
43     return WiFi.status() == WL_CONNECTED;
44 }
45
46 // Log module initialization progress
47 vTaskDelay(500); // Delay before task start
48 return;

```

- After the connection is successful, it obtains the module's IP address, enables the multi-connection mode, and starts the TCP server to listen on port 80.

```

100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

- Finally, it enters an infinite loop, retaining the interface for subsequent processing of TCP requests, and reducing CPU usage through delay, thereby completing the entire process of WiFi network management and services.

```

999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500

```

- Then comes the main function `app_main`.
- `app_main()` is the entry function of the ESP-IDF program, similar to the `main()` function in a standard C program. In this code, its role is very clear: it calls `xTaskCreate()` to create a FreeRTOS task named "wifi\_task", with the task function being `wifi_task`, allocating 4096 bytes of stack space, having a priority of 5, not passing any task parameters, and setting the task handle to NULL (not saving the task handle).
- The core meaning of this line of code is to encapsulate the WiFi initialization and TCP server logic into an independent task that runs under the management of the FreeRTOS scheduler. This keeps the main program entry point simple while ensuring that the WiFi connection task can be executed in parallel without blocking other tasks.

```

119 void app_main(void)
120 {
121     // Create WiFi task with 4096 bytes stack, priority 5
122     xTaskCreate(wifi_task, "wifi_task", 4096, NULL, 5, NULL);
123 }

```

- Now let's take a look at the "CMakeLists.txt" file in the "main" directory.
- The function of this CMake configuration is as follows:
  - Collect all the .c source files in the "main/" directory as the source files for the component;
  - Register the "main" component with the ESP-IDF build system and declare that it depends on the custom component "bsp\_uart".
  - This way, during the build process, ESP-IDF knows to build "bsp\_uart" first, and then build "main".

```

cmake_minimum_required(VERSION 3.10)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)

project("main")

file(GLOB SRC_FILES "${CMAKE_CURRENT_SOURCE_DIR}/*.c")

idf_component_register(SRCS ${SRC_FILES}
                      INCLUDE_DIRS "."
                      REQUIRES bsp_uart)

```

Note: In the subsequent courses, we will not start from scratch to create a new "CMakeLists.txt" file. Instead, we will make some minor modifications to this existing file to integrate other drivers into the main function.

## Complete Code

---

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson04-Serial\\_port\\_usage](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson04-Serial_port_usage)

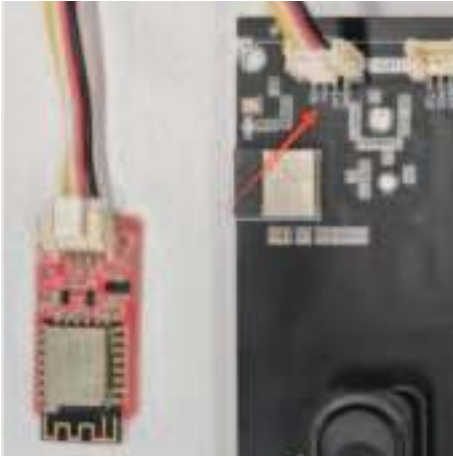
## Programming Steps

---

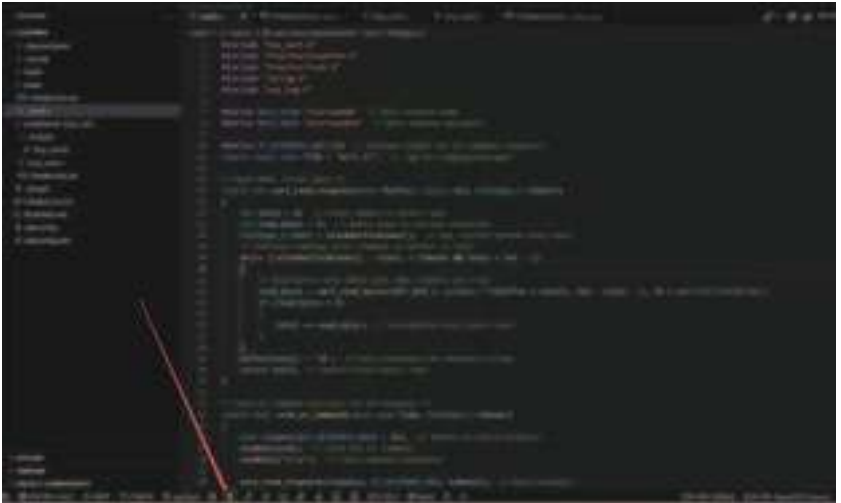
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable.



- 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)



- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



- Here, following the steps in the first section, we first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



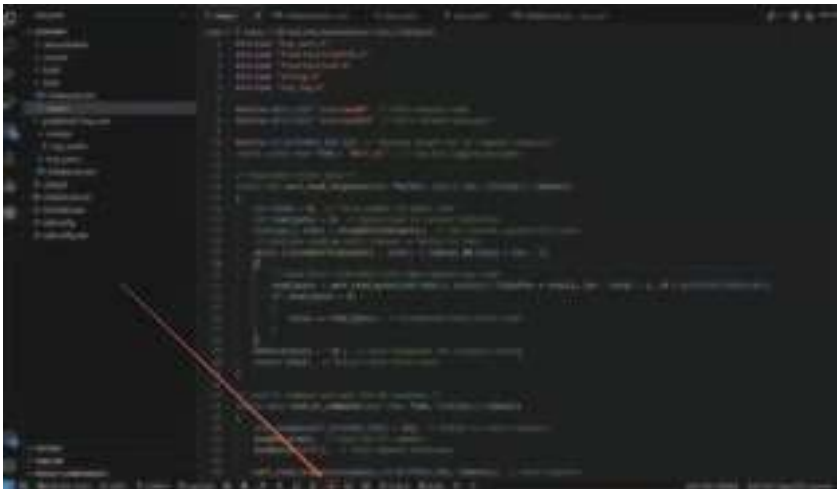
- Wait for a moment for the loading process to complete, and then you can proceed with the relevant SDK configuration.



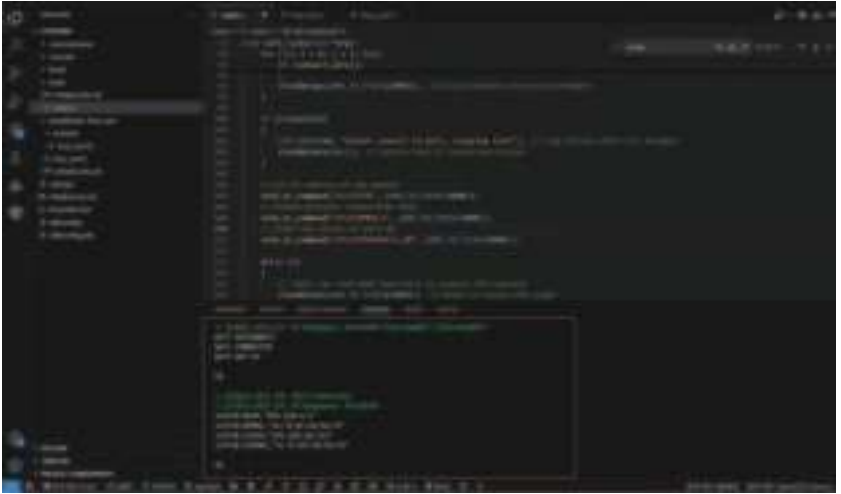
- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



- After the configuration is completed, remember to save your settings.
- After that, we will compile and burn the code (which was explained in detail in the first class)
- Here, we would like to introduce to you another very convenient feature. With just one button press, you can perform the tasks of compiling, uploading, and opening the monitor all at once. (The prerequisite is that the entire code is error-free.)



- After waiting for a while, the code compilation and upload were completed, and the monitor also opened.
- After burning the code, you will be able to see the AT commands you sent through the monitor on ESP-IDF, as well as the responses returned to you by the wifi module via the serial port. (Green is sent by Advance-P4, and white is the response from the wifi module)



# Lesson 05

## Touchscreen

### Introduction

---

In this class, we will gradually start to use multiple components together. We hope this will help everyone gain a deeper understanding of ESP-IDF and ESP32-P4.

In this class, we will use two components from the Advance-P4 category, namely `bsp_display` and `bsp_i2c`, to enable the screen to be touchable, and you will also be able to see the coordinates of your touch through the monitor.

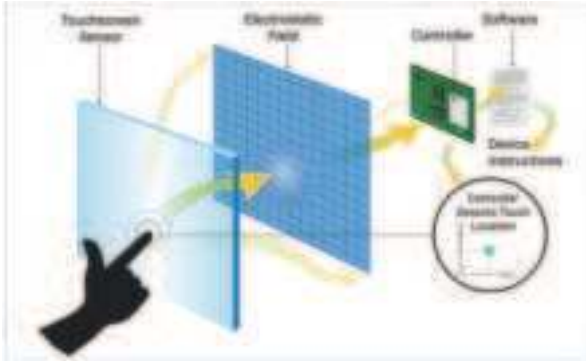
### Hardware Used in This Lesson

---

#### 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.).

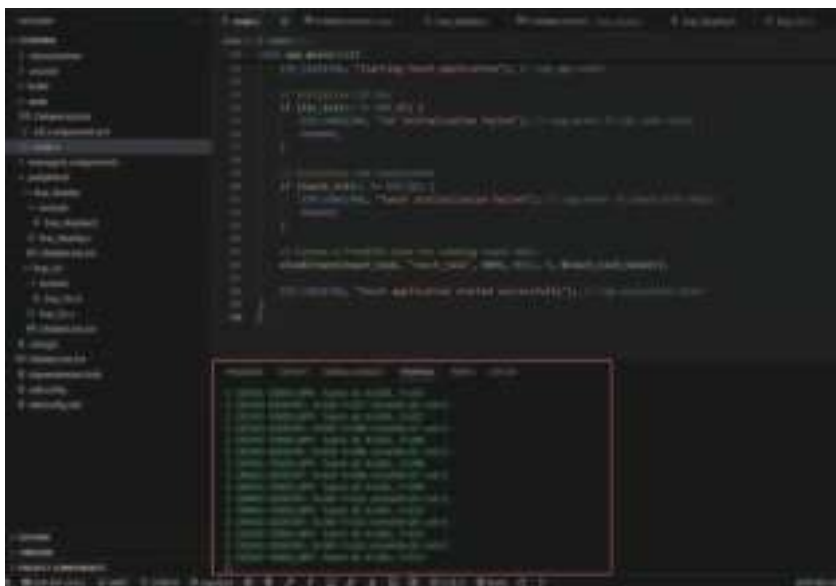
Finally, the main processor receives the coordinate data and further processes and parses these data using software.

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.

## Operation Effect Diagram

---

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

A screenshot of an IDE (likely ESP-IDF) with a dark theme. The main editor shows C++ code with comments in Chinese. A terminal window at the bottom is highlighted with a red box and displays green text output, which appears to be coordinate data. The terminal text is: 

```
11:58:05.540 [I] (32) touch: touch event: x=100, y=100
```

## Key Explanations

Now there are two components in this class (`bsp_display` and `bsp_i2c`). How should we handle the overall framework?

It's actually not difficult. Once you understand how one component is used, the two components are similar. First, click on the Github link below to download the code for this lesson.

<https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson05-Touchscreen>

- Then, drag the code of this lesson into VS Code and open the project file.



- After opening it, you can see the framework of this project.



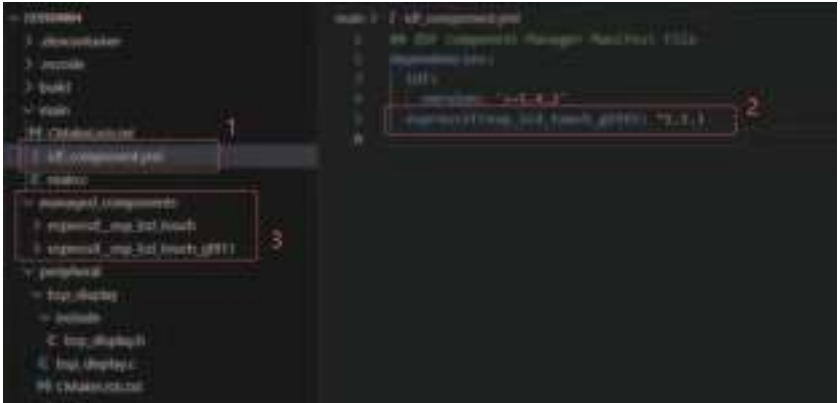
In the example of this class, a new folder named "bsp\_display" was created under the "peripheral" directory. Inside the "bsp\_display" folder, a new "include" folder and a "CMakeLists.txt" file were created.

The "bsp\_display" folder contains the "bsp\_display.c" driver file, and the "include" folder contains the "bsp\_display.h" header file.

The "CMakeLists.txt" file integrates the driver into the build system, enabling the project to utilize the touchscreen functionality written in "bsp\_display.c".



- In this case, we need to fill in the version of esp\_lcd\_touch\_gt911 in the idf\_component.yml file located in the main folder. Since this is an official library, we need to use the official library to achieve the touch function of the GT911 screen on our Advance-P4.



- When the project is compiled in the future, it will download the esp\_lcd\_touch\_gt911 library version 1.1.3. After the download, these network components will be saved in the "managed\_components" folder. (This is automatically generated after filling in the version number.)
- Then we will return to the "bsp\_display.h" file.
- We can see that the "bsp\_i2c.h" file is also included in it.



- This is another component that we are using in this class.

A screenshot of a code editor with a dark theme. On the left side, a function is declared within a header file structure, enclosed in a red rectangular box. A red arrow originates from this box and points to the right, where the corresponding implementation of that function is shown in a separate file. The code is written in C and includes various headers and function definitions.

- Because our GT911 screen touch driver uses I2C for communication control.
- Then, we declare the variables we need to use, as well as the functions. The specific implementation of these functions is in "bsp\_display.c".
- They are all unified in "bsp\_display.h" for ease of calling and management.

A screenshot of a code editor showing the implementation of a function in a C source file. The code is written in C and includes various headers and function definitions. The function implementation is shown in a separate file, as indicated by the red arrow in the previous image. The code is written in C and includes various headers and function definitions.

- Let's take a look at "bsp\_display.c" again, and see what each function does specifically.

### set\_coor:

This is an internal utility function used to update the global variables touch\_x, touch\_y, and is\_pressed, recording the latest touch point coordinates and press status. It is not called externally and is only used within this file to store touch data.

### get\_coor:

This is an external interface function used to return the current touch point coordinates and press status to the caller. By calling this function, upper-level applications can know the latest coordinates of the touch screen and whether it is pressed.

### touch\_init:

If you need to use the screen touch functionality, you must call this function in the main function.

This is the touch screen initialization function. Its main function is to configure the I2C bus and the parameters of the GT911 touch chip, and then create the handle of the touch screen driver. If the main I2C address initialization fails, it will try the backup address to ensure that the GT911 can be correctly recognized and driven. If successful, it returns ESP\_OK; if failed, it returns the corresponding error code.

### touch\_read:

This is the touch data reading function. Its main function is to read the raw data of the current touch point from the GT911, and then extract the touch point coordinates, intensity, and number of touch points.

If a touch is detected, it updates the global coordinates and prints debugging information; if no touch is detected, it sets the status to "invalid coordinates (0xffff, 0xffff) and not pressed". Finally, it returns ESP\_OK or the error code.

This is the component of the screen touch function. Just know how to call these interfaces.

Then, if you need to call it, we must configure the "CMakeLists.txt" file in the bsp\_display folder.

This file is placed in the bsp\_display folder and its main function is to tell the build system (CMake) of ESP-IDF how to compile and register this component.

(Here, we will explain in detail the construction of this "CMakeLists.txt". In the future, we will only tell you how to add and delete those libraries and components.)

- The following line of code will recursively search all the .c files in the current directory (and its subdirectories), and then place the results in the variable component\_sources.

```
esp32 > bsp_display > cd CMakeLists.txt
1 FILE(GLOB RECURSE component_sources "*.c");
2
3 idf_component_register(SRCS ${component_sources}
4                       INCLUDE_DIRS "include"
5                       REQUIRES esp_lcd_touch_gt911 bsp_lcd)
```

- This is a macro provided by ESP-IDF, used to register a component.
- SRCS specifies the source files that the component needs to be compiled. Here, it refers to all the .c files that were just found.

```

project > bsp_display > #include
1 FILE(GLOB_RECURSE component_sources "*.c")
2
3 #idf_component_register(SRCS ${component_sources}
4                       INCLUDE_DIRS "include"
5                       REQUIRES esp_lcd_touch_gt911 bsp_i2c)
6

```

- Specify the search path for header files.
- It indicates that the header files in the "bsp\_display/include" folder (such as "bsp\_display.h") will be made available for use by other components.
- This way, other components only need to #include "bsp\_display.h" to find the header files.

```

project > bsp_display > #include
1 FILE(GLOB_RECURSE component_sources "*.c")
2
3 #idf_component_register(SRCS ${component_sources}
4                       INCLUDE_DIRS "include"
5                       REQUIRES esp_lcd_touch_gt911 bsp_i2c)
6

```

- Specify the other components that the bsp\_display component depends on.
- This means: Before compiling bsp\_display, esp\_lcd\_touch\_gt911 (the GT911 touch driver) and bsp\_i2c (our own I2C wrapper) must be compiled first.
- At the same time, the dependencies will be automatically added during linking.

(In the future, when we modify other projects, simply add or remove the relevant components.)

```

project > bsp_display > #include
1 FILE(GLOB_RECURSE component_sources "*.c")
2
3 #idf_component_register(SRCS ${component_sources}
4                       INCLUDE_DIRS "include"
5                       REQUIRES esp_lcd_touch_gt911 bsp_i2c)
6

```

- The reason why `esp_lcd_touch_gt911` and `bsp_i2c` are used here is that we called them in the "`bsp_display.h`" file (if the other libraries are system libraries, then there is no need to add them)

```

#include <esp_err.h>
#include <esp_lcd_panel_io.h>
#include <esp_lcd_panel_vendor.h>
#include <esp_lcd_touch.h>
#include <esp_lcd_touch_gt911.h>
#include <bsp_i2c.h>

#define TOUCH_PANEL_IO_ADDR 0x3F000000
#define TOUCH_PANEL_IO_WIDTH 16
#define TOUCH_PANEL_IO_HEIGHT 16
#define TOUCH_PANEL_IO_NUM_LINES 16
#define TOUCH_PANEL_IO_NUM_COLUMNS 16

extern esp_lcd_touch_t touch;

void bsp_display_init(void);
void bsp_display_deinit(void);
void bsp_display_clear(void);
void bsp_display_draw(void);
void bsp_display_touch(void);

```

## I2C driver code

- Now that the relevant content of the screen touch driver has been explained, let's take a look at the content related to the I2C component.
- In "`bsp_i2c.h`", the same process is followed to declare and define the used libraries, variables, and functions, making it convenient to call them when using them.

```

#include <esp_err.h>
#include <esp_i2c.h>
#include <esp_i2c_driver.h>

#define I2C_DEVICE_ADDR 0x3F000000
#define I2C_DEVICE_WIDTH 16
#define I2C_DEVICE_HEIGHT 16
#define I2C_DEVICE_NUM_LINES 16
#define I2C_DEVICE_NUM_COLUMNS 16

extern esp_i2c_device_t i2c_device;

void bsp_i2c_init(void);
void bsp_i2c_deinit(void);
void bsp_i2c_clear(void);
void bsp_i2c_draw(void);
void bsp_i2c_touch(void);

```

- In "`bsp_i2c.c`", the library, variables and functions in "`bsp_i2c.h`" are fully utilized to implement the related functions.
- For the functions in "`bsp_i2c.c`", all you need to know is how to use them.

**print\_binary:**

Converts a 16-bit integer to a binary string (16 bits, with leading 0s padded), mainly used for printing values in binary form during debugging.

**print\_byte:**

Converts a byte (8 bits) to a string format like 0bXXXX YYYY (high 4 bits + low 4 bits), facilitating intuitive viewing of the binary content of the byte during debugging.

**i2c\_init:**

Initializes the I2C bus: configures the I2C port, SDA/SCL pins, clock source, filtering parameters and pull-up resistors, then creates an I2C master bus handle (saved in the global variable `i2c_bus_handle`), preparing for subsequent device communication.

**i2c\_dev\_register:**

Registers a slave device on the I2C bus (based on the 7-bit device address), and returns the handle of the device. When reading from or writing to this device in the future, this handle needs to be passed in.

**i2c\_read:**

Reads a certain number of data from the specified I2C device, and stores the result in the `read_buffer`. The underlying call is `i2c_master_receive`.

**i2c\_write:**

Writes a certain number of data to the specified I2C device, the underlying call is `i2c_master_transmit`.

**i2c\_write\_read:**

First writes a register address to the I2C device (`read_reg`), then reads the data from the corresponding register (`read_buffer`). This is a common process for reading registers, used to "select" the register before reading the value.

**i2c\_read\_reg:**

Performs the operation of "writing register address + reading data" at once (implemented by calling `i2c_master_transmit_receive`), which is more concise than `i2c_write_read`.

**i2c\_write\_reg:**

Writes a byte data to a certain register of the I2C device (register address + data), often used for configuring peripheral register.

- Let's talk about the role of the "CMakeLists.txt" file in the "bsp\_i2c" folder.
- This "CMakeLists.txt" is a build configuration file in the ESP-IDF framework used to manage the I2C driver components. Its main function is to tell the build system how to compile and integrate this I2C driver component.
- As mentioned earlier, here we only need to modify the components and libraries we are using at this point.

```

cmake_minimum_required(VERSION 3.10)
project("ESP32 I2C Master")

set(COMPONENT_ADD_INCLUDEDIRS "${COMPONENT_PATH}/include"
    "${COMPONENT_PATH}/../include")

add_compile_options(-std=c++11)

```

- Here, in the "bsp\_i2c.h" file, we have utilized "driver/i2c\_master.h" and "esp\_timer.h".

```

#ifndef BSP_I2C_H
#define BSP_I2C_H

#include "driver/i2c_master.h"
#include "esp_timer.h"

#endif

```

## Main function

- The main folder is the core directory for program execution, and it contains the executable file main.c for the main function.
- Add the main folder to the "CMakeLists.txt" file of the build system.

```

cmake_minimum_required(VERSION 3.10)
project("ESP32 I2C Master")

set(COMPONENT_ADD_INCLUDEDIRS "${COMPONENT_PATH}/include"
    "${COMPONENT_PATH}/../include")

add_compile_options(-std=c++11)

add_subdirectory("main")

target_link_libraries("target" PRIVATE "main")

```

- This is the entry file of the entire application. In ESP-IDF, there is no "int main()", but the program starts running from "void app\_main(void)".
- Let's first explain main.c.
- esp\_log.h: Provides the logging printing interface of ESP-IDF (such as ESP\_LOGI, ESP\_LOGE, etc.).
- freertos/FreeRTOS.h and freertos/task.h: Functions and task management interfaces related to FreeRTOS.
- "bsp\_i2c.h": Custom I2C driver, initializes the I2C bus and reads/writes devices.
- "bsp\_display.h": Custom touchscreen driver interface, provides functions such as touch\_init, touch\_read, get\_coor, etc.

```

main.c: C main() [touch_readmain.c]
1 #include "esp_log.h" // ESP-IDF logging functions
2 #include "freertos/FreeRTOS.h" // FreeRTOS task header
3 #include "freertos/task.h" // FreeRTOS task API
4 #include "bsp_i2c.h" // Custom I2C driver
5 #include "bsp_display.h" // Custom display/touch driver
6
7 #define TAG "TOUCH_APP" // logging tag for this application
8
9 #define TOUCH_TASK_HANDLE = 0 // handle for the touch reading task
10
11 // Task function: continuously read touch data and log coordinates
12 void touch_task(void *pvParameters)
13 {
14     while (1)
15     {
16         if (touch_read() == 0) // Read touch point
17             continue;
18         int x, y;
19         int pressed;
20         get_coord(&x, &y, &pressed); // Get current touch coordinates and status
21         if (pressed)
22             ESP_LOGI(TAG, "Touch at (%d, %d), %d", x, y); // Log touch coordinates
23     }
24     vTaskDelete(NULL); // Delete this current task
25 }
  
```

- TAG: Log identifier, used to distinguish the source of the log.
- touch\_task\_handle: FreeRTOS task handle, used to manage the touch reading task.

```

main.c: C main() [touch_readmain.c]
1 #include "esp_log.h" // ESP-IDF logging functions
2 #include "freertos/FreeRTOS.h" // FreeRTOS task header
3 #include "freertos/task.h" // FreeRTOS task API
4 #include "bsp_i2c.h" // Custom I2C driver
5 #include "bsp_display.h" // Custom display/touch driver
6
7 #define TAG "TOUCH_APP" // logging tag for this application
8
9 #define TOUCH_TASK_HANDLE = 0 // handle for the touch reading task
  
```



- Call the initialization code in "bsp\_i2c.c" to initialize the I2C bus, which is used for communication with the touch screen chip.

```

28 // Main application entry point
29 void app_main(void)
30 {
31     esp_log_printf(ESP_LOG_INFO, "Starting touch application"); // log app start
32
33     // initialize I2C bus
34     if (i2c_init() != ESP_OK) {
35         esp_log_printf(ESP_LOG_ERROR, "I2C initialization failed"); // log error if I2C init fails
36         return;
37     }
38
39     // initialize the touchscreen
40     if (touch_init() != ESP_OK) {
41         esp_log_printf(ESP_LOG_ERROR, "Touch initialization failed"); // log error if touch init fails
42         return;
43     }
44
45     // Create a FreeRTOS task for reading touch data
46     stackCreate(touch_task, "touch_task", 4096, 0, 1, &touch_task_handle);
47
48     esp_log_printf(ESP_LOG_INFO, "Touch application started successfully"); // log successful start
49 }

```

- Call the initialization screen touch code in "bsp\_display.c" to initialize the GT911 touch screen.
- If it fails, print the error log and return.

```

28 // Main application entry point
29 void app_main(void)
30 {
31     esp_log_printf(ESP_LOG_INFO, "Starting touch application"); // log app start
32
33     // initialize I2C bus
34     if (i2c_init() != ESP_OK) {
35         esp_log_printf(ESP_LOG_ERROR, "I2C initialization failed"); // log error if I2C init fails
36         return;
37     }
38
39     // initialize the touchscreen
40     if (touch_init() != ESP_OK) {
41         esp_log_printf(ESP_LOG_ERROR, "Touch initialization failed"); // log error if touch init fails
42         return;
43     }
44
45     // Create a FreeRTOS task for reading touch data
46     stackCreate(touch_task, "touch_task", 4096, 0, 1, &touch_task_handle);
47
48     esp_log_printf(ESP_LOG_INFO, "Touch application started successfully"); // log successful start
49 }

```



## Complete Code

---

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

<https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson05-Touchscreen>

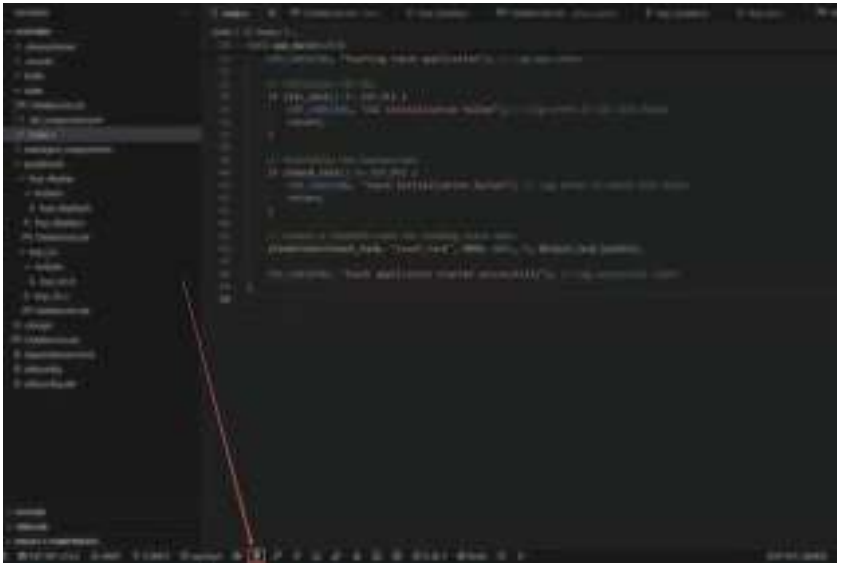
## Programming Steps

---

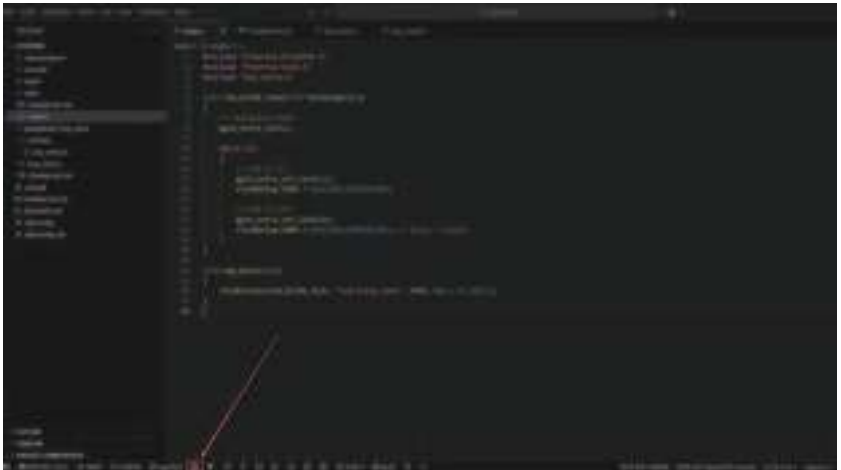
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable.



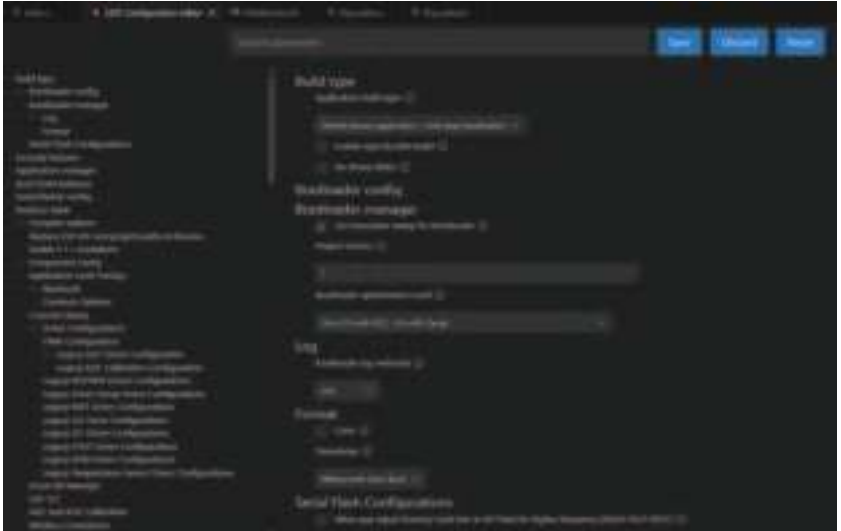
- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



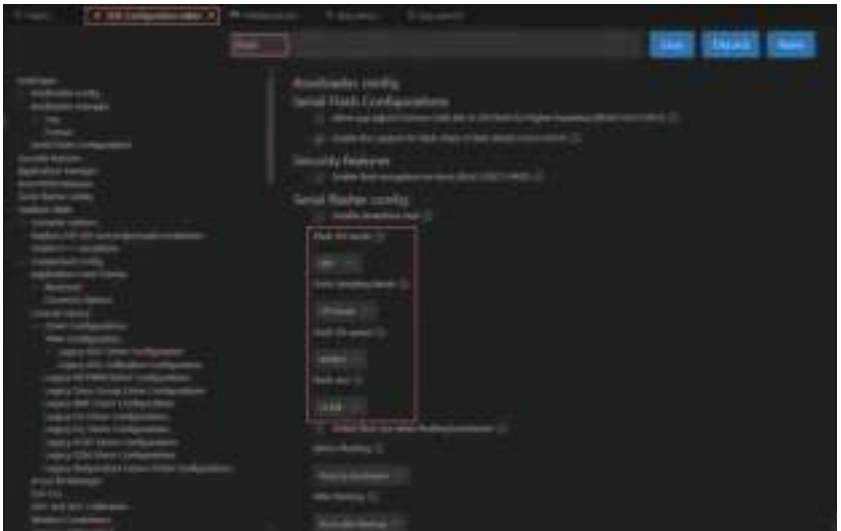
- Then, following the steps in the first section, select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



- Wait for a moment for the loading process to complete, and then you can proceed with the related SDK configuration.



- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



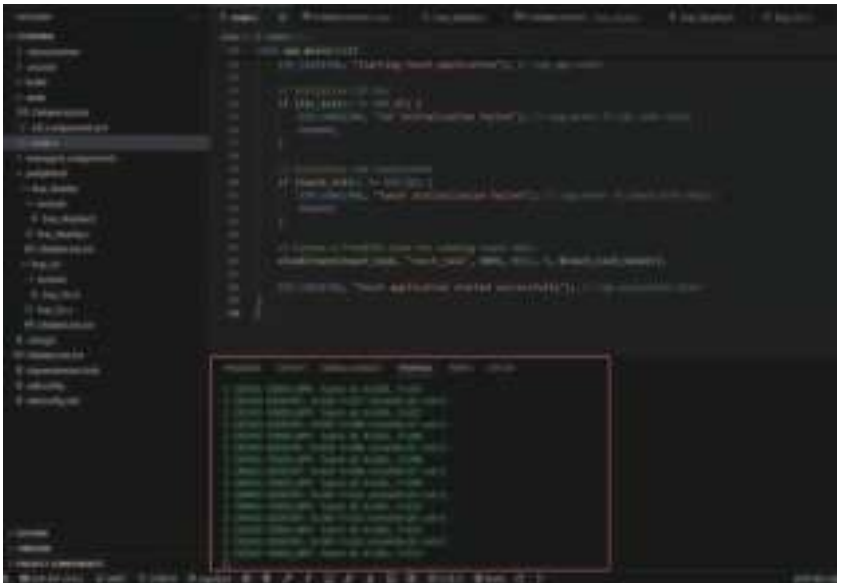
- After the configuration is completed, be sure to save your settings.



- Then we will compile and burn the code (as detailed in the first class).
- Here, we would like to introduce to you another very convenient feature. With just one button press, you can perform the compilation, upload, and open the monitor at once. (This is provided that the entire code is error-free.)



- After waiting for a while, the code compilation and upload process was completed, and the monitor also opened. By touching the Advance-P4 screen, you will be able to see the coordinates of the screen you touched displayed.



# Lesson 06

## USB2.0

### Introduction

---

In this class, we are expanding on what we learned in the previous class.

Before studying this class, please make sure you understand the implementation of the touch function in the previous class. This will be of great help to your learning of this class.

As you know, in the previous class, we already learned the two components, `bsp_display` and `bsp_i2c`. It was because we fully utilized these two components that our Advance-P4 screen could be made touchable.

In this class, we will add a new component, `bsp_usb`, on top of these two components. This will enable us to use the USB2.0 interface on our Advance-P4 to act as a mouse. When you slide on the screen of the Advance-P4, you will be able to see that the mouse on your computer also moves accordingly

### Hardware Used in This Lesson

---

#### USB 2.0 on the Advance-P4



## Operation Effect Diagram

---

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.



## Key Explanations

---

- Now, this class is about adding the bsp\_usb component based on the project from the previous class, so that we can slide and touch the Advance-P4 screen and control the computer mouse.
- The previous touch function has already been realized using the bsp\_usb and bsp\_i2c components from the previous class.
- Next, we will focus on understanding the bsp\_usb component.
- First, click on the Github link below to download the code for this lesson.

<https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson06-USB2.0>

- Then, drag the code of this lesson into VS Code and open the project file.

- After opening it, you can see the framework of this project.



In the example of this class, a new folder named "bsp\_usb" was created under the "peripheral" directory. Inside the "bsp\_usb" folder, a new "include" folder and a "CMakeLists.txt" file were created.

The "bsp\_usb" folder contains the "bsp\_usb.c" driver file, and the "include" folder contains the "bsp\_usb.h" header file.

The "CMakeLists.txt" file integrates the driver into the build system, enabling the project to utilize the USB2.0 transmission functionality written in "bsp\_usb.c".

## USB 2.0 driver code

- The USB2.0 driver consists of two files: "bsp\_usb.c" and "bsp\_usb.h".
- Next, we will first analyze the "bsp\_usb.h" program.
- "bsp\_usb.h" is the header file of the USB2.0 driver module, mainly used for:
  - Making the functions, macros and variable declarations implemented in "bsp\_usb.c" available for external programs to use
  - Allowing other .c files to simply include "#include "bsp\_usb.h" " to call this module
  - In other words, it is the interface layer, exposing which functions and constants can be used externally while hiding the internal details of the module.
  - In this component, all the libraries we need to use are placed in the "bsp\_usb.h" file for unified management.

```

4 #include <stdint.h> // Standard integer types
5 #include "esp_err.h" // ESP-IDF error handling definitions
6 #include "esp_log.h" // ESP-IDF logging functions
7 #include "freertos/FreeRTOS.h" // FreeRTOS base header
8 #include "freertos/task.h" // FreeRTOS task management APIs
9
10
11 #include "tinyusb.h" // TinyUSB core library
12 #include "class/hid/hid_device.h" // TinyUSB HID device class definitions

```

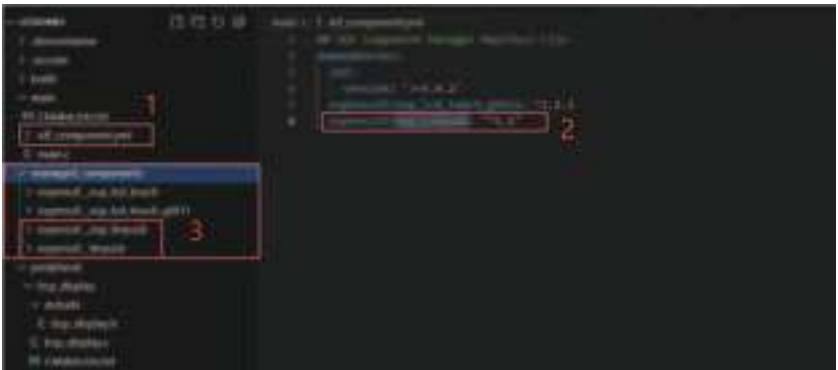
- Like the tinyusb.h (this is a library under the network component esp\_tinyusb)

```

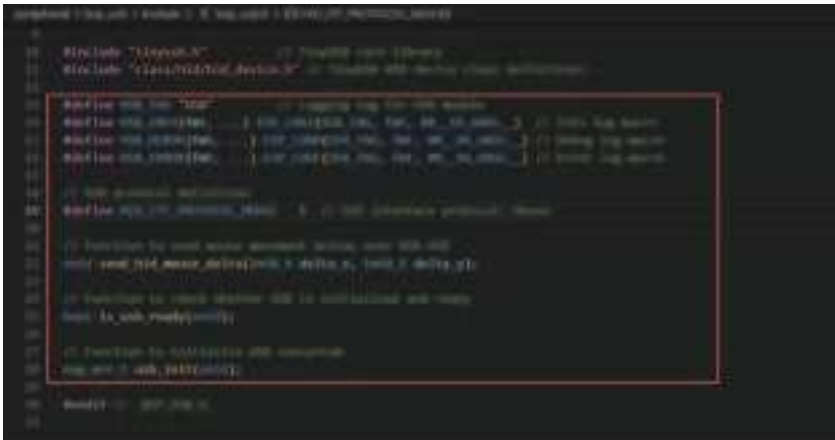
4 #include <stdint.h> // Standard integer types
5 #include "esp_err.h" // ESP-IDF error handling definitions
6 #include "esp_log.h" // ESP-IDF logging functions
7 #include "freertos/FreeRTOS.h" // FreeRTOS base header
8 #include "freertos/task.h" // FreeRTOS task management APIs
9
10
11 #include "tinyusb.h" // TinyUSB core library
12 #include "class/hid/hid_device.h" // TinyUSB HID device class definitions

```

- In this case, we need to fill in the version of esp\_tinyusb in the idf\_component.yml file located in the main folder.
- Since this is an official library, we need to use the official library to achieve the USB 2.0 transmission function on our Advance-P4.



- When the project is compiled in the future, it will download the 1.1 version of the esp\_tinyusb library. After the download, these network components will be saved in the "managed\_components" folder. (This is automatically generated after filling in the version number.)
- Then comes the declaration of the variables we need to use, as well as the declaration of the functions. The specific implementations of these functions are in "bsp\_usb.c".
- They are all unified in "bsp\_usb.h" for the convenience of calling and management.



- Let's take a closer look at "bsp\_usb.c", examining the specific functions of each one.
- bsp\_usb: This is a simple USB HID (mouse) module based on TinyUSB, including HID descriptors, TinyUSB callbacks, and external initialization/sending interfaces.
- Although these three functions have empty implementations, they must exist.
- They are callback interfaces for USB HID devices to communicate with the host –
  - tud\_hid\_descriptor\_report\_cb is used to return the HID report descriptor,
  - tud\_hid\_get\_report\_cb handles the GET\_REPORT request from the host,
  - tud\_hid\_set\_report\_cb handles the SET\_REPORT request or OUT data from the host.

#### **tud\_hid\_descriptor\_report\_cb:**

This callback is called by TinyUSB when the host requests the HID report descriptor through the control transfer. The function should return a pointer to a static or global descriptor array; in your implementation, it directly returns hid\_report\_descriptor, suitable for scenarios with only one HID interface.

#### **tud\_hid\_get\_report\_cb:**

This is the callback for handling the host's GET\_REPORT request: when the host wants to read the "input/characteristic" report from the device side, TinyUSB will call it. The function should fill the buffer with the report data and return the actual length; currently, you return 0 (indicating no provision), and TinyUSB will handle this request as a STALL.

### **tud\_hid\_set\_report\_cb:**

This callback is called when the host initiates a SET\_REPORT (or sends data through the OUT endpoint). The application should parse the contents of the buffer based on report\_id / report\_type and perform the corresponding actions.

Then the following function is the interface we call to implement the USB 2.0 transfer function.

usb\_init() → Initialize USB HID mouse device

send\_hid\_mouse\_delta() → Send mouse movement data

is\_usb\_ready() → Determine if USB is available

### **send\_hid\_mouse\_delta:**

This is an external sending interface used to send the mouse movement increment through HID to the host: The function first checks tud\_hid\_ready() (whether the device has been enumerated and the HID is available), and if ready, it calls tud\_hid\_mouse\_report(...) to send a mouse report containing the X/Y increment.

### **is\_usb\_ready:**

This is a simple query function that returns the result of tud\_hid\_ready() to determine if the TinyUSB HID interface is ready to send reports to the host (that is, whether the device has successfully enumerated and the HID interface is available).

### **usb\_init:**

This function constructs tinyusb\_config\_t (containing string descriptors, configuration descriptors, etc.) and calls tinyusb\_driver\_install(&usb\_cfg) to install the TinyUSB driver; it is responsible for starting the USB subsystem and exposing the HID device to the operating system (the host).

The above bsp\_usb component has realized the HID mouse function in the USB 2.0 device mode, enabling the ESP32P4 to simulate mouse operations.

That's all about the bsp\_usb component. Just know how to call these interfaces and you're good to go.

Then, if we need to make a call, we must also configure the "CMakeLists.txt" file located in the "bsp\_usb" folder.

This file is placed in the "bsp\_usb" folder and its main function is to inform the build system (CMake) of ESP-IDF: how to compile and register the "bsp\_usb" component.



- The reason why it is called esp\_tinyusb here is that we called it in the "bsp\_usb.h" file (for other libraries that are system libraries, there is no need to add anything).



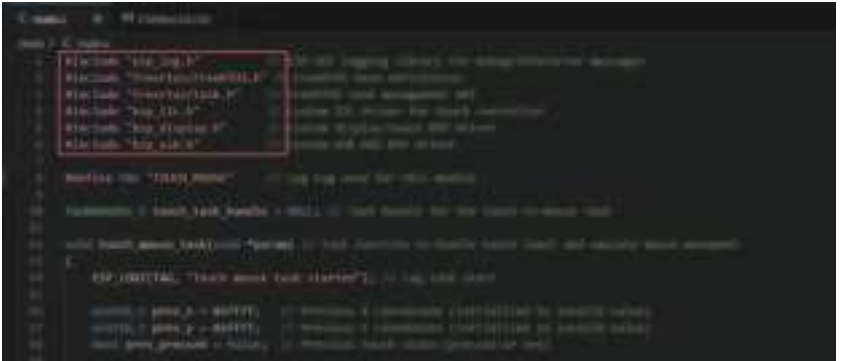
## Main function

The main folder is the core directory for program execution, and it contains the executable file main.c for the main function.

Add the main folder to the "CMakeLists.txt" file of the build system.



- This is the entry file of the entire application. In ESP-IDF, there is no "int main()", but the program starts running from "void app\_main(void)".
- Let's first explain main.c.
- esp\_log.h: Log printing in ESP-IDF (such as ESP\_LOGI/ESP\_LOGE, etc.).
- freertos/Freertos.h and freertos/task.h: Task management in FreeRTOS.
- bsp\_i2c.h: Initialize I2C for communication with the touch screen.
- bsp\_display.h: Obtain the touch screen coordinates.
- "bsp\_usb.h": USB HID mouse driver interface



- TAG: Log tag.
- touch\_task\_handle: FreeRTOS task handle, used to manage the touch mouse task.

```

#include "tag_log.h" // Tag for logging within the FreeRTOS environment
#include "FreeRTOS/FreeRTOS.h" // FreeRTOS basic definitions
#include "FreeRTOS/task.h" // FreeRTOS task definitions
#include "tag_log.h" // Tag for logging within the touch application
#include "tag_hid_mouse.h" // Mouse movement delta struct
#include "tag_hid.h" // HID mouse report structure

void* touch_mouse_task(void*) {
    TAG_LOG(TAG_INFO, "tag_log: touch mouse task started");
    // ... (rest of the function code)
}

```

### The touch\_mouse\_task function:

This function, named touch\_mouse\_task, serves to convert the finger movements on the touch screen into USB HID mouse movements. It continuously reads the touch screen coordinates and press status within an infinite loop. When the touch screen is pressed and the USB HID device is ready, it calculates the incremental movement (delta) of the finger and sends the mouse movement report to the computer via send\_hid\_mouse\_delta; when the finger is released, it resets the previous coordinates. The entire process cycles at a 10ms interval, achieving a mouse sampling rate of approximately 100Hz.

```

void touch_mouse_task(void* param) {
    // ... (initialization code)

    ESP_LOGE(TAG, "touch mouse task started"); // log task start

    int16_t prev_x = 0xFFFF; // Previous X coordinate (initialised to invalid value)
    int16_t prev_y = 0xFFFF; // Previous Y coordinate (initialised to invalid value)
    bool prev_pressed = 0; // Previous touch state (pressed or not)

    while (1) { // Infinite loop for continuous task execution
        if (touch_ready == ESP_OK) { // Touch screen ready
            int16_t x = 0; // Current X coordinate
            bool pressed; // Current touch state
            get_touch(&x, &y, &pressed); // Retrieve touch coordinates and state

            // Send mouse movement only when both X and Y are ready to being sent
            if (Device_M_Is_Ready()) {
                if (prev_pressed != pressed || prev_x != 0xFFFF || prev_y != 0xFFFF) {
                    // Calculate movement delta
                    int16_t delta_x = (x > 0xFFFF ? 0 : x) - prev_x;
                    int16_t delta_y = (y > 0xFFFF ? 0 : y) - prev_y;

                    // Send mouse movement with mouse
                    send_hid_mouse_delta(delta_x, delta_y);
                    ESP_LOGE(TAG, "touch mouse: dx=%d, dy=%d", delta_x, delta_y); // log movement
                }

                prev_x = x; // Update previous X
                prev_y = y; // Update previous Y
            } else if (!pressed) {
                // Reset previous coordinates when touch is released
                prev_x = 0xFFFF;
                prev_y = 0xFFFF;
            }

            prev_pressed = pressed; // Save current touch state
        }
    }
}

```

### The workflow of the touch\_mouse\_task code:

Call touch\_read() to obtain the touch screen status.

Use get\_coor() to get the current coordinates (x, y) and the pressed state pressed.

If the screen is pressed and the USB is ready:

Calculate  $\text{delta\_x} = x - \text{prev\_x}$ ,  $\text{delta\_y} = y - \text{prev\_y}$ .

Call send\_hid\_mouse\_delta(delta\_x, delta\_y) to send mouse movement.

Update prev\_x/prev\_y.

Reset prev\_x/prev\_y when releasing the touch.

Delay 10ms to achieve a 100Hz sampling rate.

Then comes the main function **app\_main**.

app\_main is the main entry function of the program. Its function is to initialize the system peripherals and start the touch mouse task. It sequentially completes the initialization of the I2C bus, the initialization of the touch screen, and the initialization of the USB HID subsystem. If any initialization fails, it records the error and exits.

After successful initialization, it creates a FreeRTOS task named touch\_mouse\_task to continuously read the touch screen input and convert it into mouse movement signals, and finally starts the entire touch mouse application.

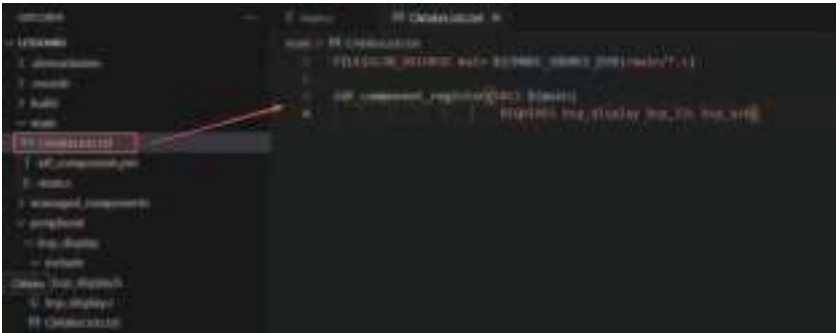
```
void app_main() {
    // I2C initialization
    if (!i2c_init()) {
        ESP_LOG_ERROR("I2C initialization failed");
        return;
    }

    // Touch screen initialization
    if (!touch_init()) {
        ESP_LOG_ERROR("Touch screen initialization failed");
        return;
    }

    // USB HID initialization
    if (!usb_hid_init()) {
        ESP_LOG_ERROR("USB HID initialization failed");
        return;
    }

    // Create a task to read the touch screen and send mouse movement
    xTaskCreate(touch_mouse_task, "touch_mouse_task", 4096, NULL, 1, &touch_mouse_task_hdl);
    if (touch_mouse_task_hdl == NULL) {
        ESP_LOG_ERROR("Failed to create touch mouse task");
        return;
    }
}
```

- Now let's take a look at the "CMakeLists.txt" file in the "main" directory.
- The function of this CMake configuration is as follows:
  - Collect all the .c source files in the "main/" directory as the source files for the component;
  - Register the main component with the ESP-IDF build system and declare that it depends on the custom component "bsp\_display", the custom component "bsp\_i2c", and the custom component "bsp\_usb".
  - This way, during the build process, ESP-IDF knows to build "bsp\_display", "bsp\_i2c", and "bsp\_usb" first, and then build "main".



Note: In the subsequent courses, we will not start from scratch to create a new "CMakeLists.txt" file. Instead, we will make some minor modifications to this existing file to integrate other drivers into the main function.

## Complete Code

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

<https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson06-USB2.0>

## 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.

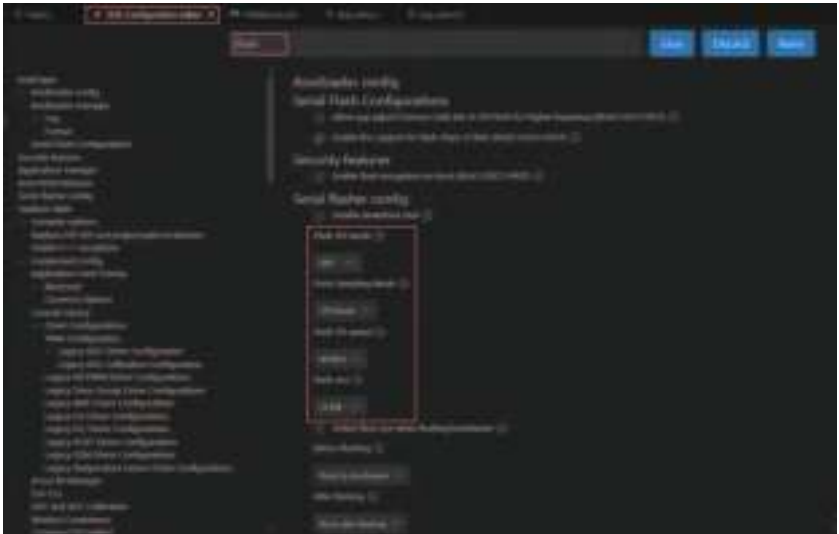


- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)





- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



- Then, search for "hid" in the search box.



- After the configuration is completed, be sure to save your settings.
- Then we will compile and burn the code (as detailed in the first class).
- Here, we would like to introduce to you another very convenient feature. With just one button press, you can perform the compilation, upload, and open the monitor at once. (This is provided that the entire code is error-free.)



- After waiting for a while, the code compilation and upload were completed, and the monitor also opened.
- At this point, please remember to use another Type-C cable to connect your Advance-P4 through the USB2.0 interface. Only in this way can you use the USB2.0 protocol for communication.



- 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.



# Lesson 07

## 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.

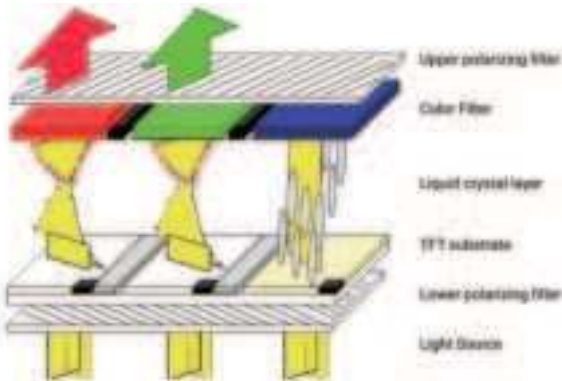
### Hardware Used in This Lesson

---

#### 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.

## Operation Effect Diagram

---

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



## Key Explanations

---

- The main focus of this class is to turn on the screen for display. Here, we will provide everyone with a new component called `bsp_illuminate`. This component is mainly responsible for driving the screen, turning on the backlight, and performing related displays. As you know, you can call the interface we have written at the appropriate time.
- Next, we will focus on understanding the `bsp_illuminate` component.
- First, click on the Github link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/v1.0/idf-code/Lesson07-Turn\\_on\\_the\\_screen](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/v1.0/idf-code/Lesson07-Turn_on_the_screen)

- Then, drag the code of this lesson into VS Code and open the project file.
- After opening it, you can see the framework of this project.



- Such as `esp_lcd_ek79007.h`, `esp_lvgl_port.h`, and `lvgl.h` (these are libraries under the network component)

```

1 // -----Header file declaration-----
2 #include "bsp_gpio.h" //GPIO pins used for LED control, as well as other I/O pins
3 #include "bsp_led.h" //GPIO pins used for LED control, as well as other I/O pins
4 #include "bsp_lcd_ek79007.h" //reference for LCD function-related variables
5 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
6 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
7 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
8 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
9 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
10 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
11 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
12 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
13 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
14 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
15 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
16 #include "bsp_lcd_panel_ops.h" //reference for LCD panel ops function-related variables
17 // -----Header file declaration end-----

```

- In this case, we need to fill in the versions of `esp_lcd_ek79007`, `esp_lvgl_port` and `lvgl` in the `idf_component.yml` file located in the main folder.
- Since these are official libraries, we need to use the official libraries to achieve the screen display function on our Advance-P4.



- When the project is compiled in the future, it will automatically download the `esp_lcd_ek79007` library version 1.0.2, the `esp_lvgl_port` version 2.6.0, and the `lvgl` version 8.3.11. After the download is completed, these network components will be saved in the `managed_components` folder. (This is automatically generated after filling in the version numbers.)
- Then comes the declaration of the variables we need to use, as well as the declaration of the functions. The specific implementations of these functions are in `"bsp_illuminate.c"`.

- They are all uniformly placed in "bsp\_illuminate.h" for ease of calling and management. (When used in "bsp\_illuminate.c", we will understand their functions later.)

- Let's take a look at "bsp\_illuminate.c" again. We'll examine the specific functions of each one.

### **bsp\_illuminate:**

This component provides underlying driver support for the subsequent application layer (such as in app\_main where it displays "Hello Elecrow"). It enables you to draw and display using the LVGL API without having to worry about the details of the hardware driver.

**Then the following functions are the interfaces we call to implement the screen display.**

**blight\_init / set\_lcd\_blight** → Control the backlight.

**display\_port\_init / display\_port\_deinit** → Manage the display interface resources.

**lvgl\_init** → Start the LVGL framework.

**display\_init** → Provide the encapsulation of the overall display initialization process.

### **blight\_init:**

Initialize the LCD backlight. It will configure the specified backlight GPIO as output mode, and then configure the PWM signal through the LEDC timer + channel to lay the foundation for subsequent adjustment of the backlight brightness.

### **set\_lcd\_blight(uint32\_t brightness) :**

Set the LCD backlight brightness. Based on the incoming brightness value (0–100), calculate the corresponding PWM duty cycle, call the LEDC API to update the duty cycle, and achieve the brightness adjustment of the backlight; if it is 0, then completely turn off the backlight.

### **display\_port\_init(void) :**

Initialize the display interface. It first configures and creates the MIPI DSI bus and DBI IO, then selects the color format according to the pixel depth, configures the DPI parameters (resolution, timing, etc.), and finally initializes the EK79007 controller panel through the vendor driver and completes the reset and startup.

### **display\_port\_deinit(void) :**

Reinitialize the display interface. It releases the panel, IO, and DSI bus resources, clears the related handles, and closes the backlight to ensure that the resources will not be leaked.

### **lvgl\_init() :**

Initialize the LVGL graphics library. It creates the LVGL task, timer, and memory configuration, then registers the previously created LCD panel in LVGL as a display device, sets the buffer, resolution, color format, refresh mode, etc., and prepares for the subsequent drawing of the graphical interface.

### **display\_init() :**

The complete display initialization entry function.

It calls the backlight initialization → display interface initialization → LVGL initialization in sequence. If any step fails, it immediately returns an error. Finally, it defaults to setting the backlight brightness to 0 (turn off the backlight). This is the overall entry point when called externally.

That's all about the components of `bsp_illuminate`. Just remember how to call these interfaces and you'll be fine.

Then, if we need to make a call, we must also configure the "CMakeLists.txt" file located in the "bsp\_illuminate" folder.

This file is placed in the "bsp\_illuminate" folder and its main function is to inform the build system (CMake) of ESP-IDF: how to compile and register the "bsp\_illuminate" component.



- The reason why it is driver, esp\_lcd\_ek79007, lvgl, and esp\_lvgl\_port is that we called them in "bsp\_illuminate.h" (for other libraries that are system libraries, there is no need to add them)



## Main function

The main folder is the core directory for program execution, and it contains the executable file main.c for the main function.

Add the main folder to the "CMakeLists.txt" file of the build system.



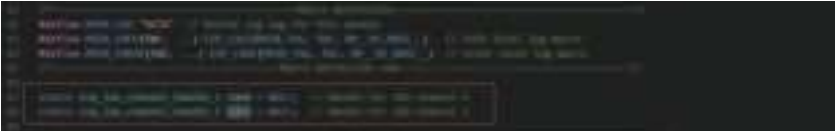
- This is the entry file of the entire application. In ESP-IDF, there is no "int main()". Instead, the program starts running from the "void app\_main(void)" function.
- Let's first explain main.c.
- On the ESP32-P4, it completes the acquisition of the power LDO → initialization of the screen driver → turning on the backlight → displaying the text "Hello Elecrow" in the center of the screen using LVGL.
- "bsp\_illuminate.h": This is a header file of the board support package (BSP), which encapsulates the initialization of LCD display screens and backlight control interfaces related to hardware, allowing the main program to directly call these functions without needing to concern about the underlying register operations.
- "lvgl.h": This is the main header file of the LVGL graphics library, providing functions for creating and managing GUI objects, setting styles, layouts, and event handling, enabling you to display text, graphics, and animations on the screen.
- "freertos/FreeRTOS.h": This is the core header file of FreeRTOS, defining the basic types, macros, and data structures of the operating system, providing underlying support for task scheduling, time management, and memory management.
- "freertos/task.h": This is the header file of FreeRTOS task management, providing API for creating, deleting, suspending, and delaying tasks, enabling the program to achieve concurrent execution of multiple tasks.
- "esp\_ldo\_regulator.h": This is the header file of the LDO (Low Dropout Linear Regulator) control interface provided by ESP-IDF, allowing the program to apply for, configure, and control LDO channels, providing stable voltages for peripherals such as LCD.
- "esp\_log.h": This is the header file of the log printing interface of ESP-IDF, providing log output of different levels (INFO, ERROR, etc.), enabling developers to debug and track the running status of the program.

```

// ...
// #include "esp_ldo_regulator.h"
// #include "esp_log.h"
// ...
// #define LDO_CHANNEL_3 3
// #define LDO_CHANNEL_4 4
// ...

```

- The following two lines of code define the control handles for LDO channels 3 and 4, which are used to bind to the actual LDO power channels during subsequent initialization, so that the program can control the output of different voltage power supplies.



### `lvgl_show_hello_elecrow():`

Function: Create a centered label on the current screen of LVGL and display the text "Hello Elecrow". Also, set the font size/color and other styles for the text. (If modifying the content, replace "Hello Elecrow") Key points:

First, call `lvgl_port_lock(0)` to attempt to acquire the LVGL mutex lock (0 indicates non-blocking immediate return), to prevent concurrent modification of LVGL objects. If the lock acquisition fails, the function simply returns and prints an error - this might not display the text because other tasks may hold the lock.

Use `lv_scr_act()` to obtain the current screen object and set the background to white (`LV_PART_MAIN`).

Create a label, set the text, initialize the static `lv_style_t` `label_style` and set the font (`lv_font_montserrat_42`), color to black, background transparent, and then add the style to the label.

Finally, call `lv_obj_center()` to center the label, release the LVGL lock `lvgl_port_unlock()` to allow the LVGL rendering task to continue working.

(The font `lv_font_montserrat_42` must be enabled and linked to the project during LVGL build, otherwise there will be compilation/linking or runtime issues.)



### `init_fail_handler(const char *module_name, esp_err_t err):`

Function: When the initialization of a certain module fails, this function will enter an infinite loop and print the error message (including the module name and error code string) once per second.



### `system_init(void):`

Function: System-level initialization. First, it acquires two LDO channels (ldo3/ldo4), then calls `display_init()` to initialize the display system, and finally turns on the backlight to the maximum brightness (`set_lcd_blight(100)`). Key points:

First, construct `esp_ldo_channel_config_t` (setting `chan_id = 3` and `4` with voltage `2500/3300 mV`), and use `esp_ldo_acquire_channel()` to obtain the channel handle. If it fails, call `init_fail_handler()` to shut down and print the error message.



- Key points: The function `system_init()` is blocking and critical: if it fails, it will enter an infinite loop in the `init_fail_handler()` and the `app_main` will not proceed.
- The function `lvgl_show_hello_elecrow()` simply returns after creating the LVGL object; the actual image is refreshed to the screen by LVGL's own rendering task or tick (depending on the implementation of `lvgl_port`).

```

int main(void) {
    /* Start LVGL (Electron Display) */
    lvgl_init();

    /* Register the LVGL component with the ESP-IDF build system */
    lvgl_register_component();

    /* Show the LVGL component */
    lvgl_show_hello_elecrow();

    /* End of main */
}

```

- Now let's take a look at the "CMakeLists.txt" file in the "main" directory.
- The function of this CMake configuration is as follows:
  - Collect all the .c source files in the "main/" directory as the source files for the component;
  - Register the "main" component with the ESP-IDF build system and declare that it depends on the custom component "bsp\_illuminate".
  - This way, during the build process, ESP-IDF knows to build "bsp\_illuminate" first, and then build "main".

```

cmake_minimum_required(VERSION 3.10)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)

project("main")

# Register the component with the ESP-IDF build system
lvgl_register_component()

# Add the source files in the main directory
add_compile_options("-std=c99")
add_compile_options("-std=gnu99")
add_compile_options("-std=gnu11")
add_compile_options("-std=c11")
add_compile_options("-std=c++11")
add_compile_options("-std=c++14")
add_compile_options("-std=c++17")
add_compile_options("-std=c++20")
add_compile_options("-std=c++23")
add_compile_options("-std=c++26")
add_compile_options("-std=c++29")
add_compile_options("-std=c++32")
add_compile_options("-std=c++35")
add_compile_options("-std=c++38")
add_compile_options("-std=c++41")
add_compile_options("-std=c++44")
add_compile_options("-std=c++47")
add_compile_options("-std=c++50")
add_compile_options("-std=c++53")
add_compile_options("-std=c++56")
add_compile_options("-std=c++59")
add_compile_options("-std=c++62")
add_compile_options("-std=c++65")
add_compile_options("-std=c++68")
add_compile_options("-std=c++71")
add_compile_options("-std=c++74")
add_compile_options("-std=c++77")
add_compile_options("-std=c++80")
add_compile_options("-std=c++83")
add_compile_options("-std=c++86")
add_compile_options("-std=c++89")
add_compile_options("-std=c++92")
add_compile_options("-std=c++95")
add_compile_options("-std=c++98")
add_compile_options("-std=c++01")
add_compile_options("-std=c++03")
add_compile_options("-std=c++05")
add_compile_options("-std=c++07")
add_compile_options("-std=c++09")
add_compile_options("-std=c++11")
add_compile_options("-std=c++13")
add_compile_options("-std=c++14")
add_compile_options("-std=c++16")
add_compile_options("-std=c++17")
add_compile_options("-std=c++19")
add_compile_options("-std=c++20")
add_compile_options("-std=c++21")
add_compile_options("-std=c++22")
add_compile_options("-std=c++23")
add_compile_options("-std=c++24")
add_compile_options("-std=c++25")
add_compile_options("-std=c++26")
add_compile_options("-std=c++27")
add_compile_options("-std=c++28")
add_compile_options("-std=c++29")
add_compile_options("-std=c++30")
add_compile_options("-std=c++31")
add_compile_options("-std=c++32")
add_compile_options("-std=c++33")
add_compile_options("-std=c++34")
add_compile_options("-std=c++35")
add_compile_options("-std=c++36")
add_compile_options("-std=c++37")
add_compile_options("-std=c++38")
add_compile_options("-std=c++39")
add_compile_options("-std=c++40")
add_compile_options("-std=c++41")
add_compile_options("-std=c++42")
add_compile_options("-std=c++43")
add_compile_options("-std=c++44")
add_compile_options("-std=c++45")
add_compile_options("-std=c++46")
add_compile_options("-std=c++47")
add_compile_options("-std=c++48")
add_compile_options("-std=c++49")
add_compile_options("-std=c++50")
add_compile_options("-std=c++51")
add_compile_options("-std=c++52")
add_compile_options("-std=c++53")
add_compile_options("-std=c++54")
add_compile_options("-std=c++55")
add_compile_options("-std=c++56")
add_compile_options("-std=c++57")
add_compile_options("-std=c++58")
add_compile_options("-std=c++59")
add_compile_options("-std=c++60")
add_compile_options("-std=c++61")
add_compile_options("-std=c++62")
add_compile_options("-std=c++63")
add_compile_options("-std=c++64")
add_compile_options("-std=c++65")
add_compile_options("-std=c++66")
add_compile_options("-std=c++67")
add_compile_options("-std=c++68")
add_compile_options("-std=c++69")
add_compile_options("-std=c++70")
add_compile_options("-std=c++71")
add_compile_options("-std=c++72")
add_compile_options("-std=c++73")
add_compile_options("-std=c++74")
add_compile_options("-std=c++75")
add_compile_options("-std=c++76")
add_compile_options("-std=c++77")
add_compile_options("-std=c++78")
add_compile_options("-std=c++79")
add_compile_options("-std=c++80")
add_compile_options("-std=c++81")
add_compile_options("-std=c++82")
add_compile_options("-std=c++83")
add_compile_options("-std=c++84")
add_compile_options("-std=c++85")
add_compile_options("-std=c++86")
add_compile_options("-std=c++87")
add_compile_options("-std=c++88")
add_compile_options("-std=c++89")
add_compile_options("-std=c++90")
add_compile_options("-std=c++91")
add_compile_options("-std=c++92")
add_compile_options("-std=c++93")
add_compile_options("-std=c++94")
add_compile_options("-std=c++95")
add_compile_options("-std=c++96")
add_compile_options("-std=c++97")
add_compile_options("-std=c++98")
add_compile_options("-std=c++99")
add_compile_options("-std=c++00")
add_compile_options("-std=c++01")
add_compile_options("-std=c++02")
add_compile_options("-std=c++03")
add_compile_options("-std=c++04")
add_compile_options("-std=c++05")
add_compile_options("-std=c++06")
add_compile_options("-std=c++07")
add_compile_options("-std=c++08")
add_compile_options("-std=c++09")
add_compile_options("-std=c++10")
add_compile_options("-std=c++11")
add_compile_options("-std=c++12")
add_compile_options("-std=c++13")
add_compile_options("-std=c++14")
add_compile_options("-std=c++15")
add_compile_options("-std=c++16")
add_compile_options("-std=c++17")
add_compile_options("-std=c++18")
add_compile_options("-std=c++19")
add_compile_options("-std=c++20")
add_compile_options("-std=c++21")
add_compile_options("-std=c++22")
add_compile_options("-std=c++23")
add_compile_options("-std=c++24")
add_compile_options("-std=c++25")
add_compile_options("-std=c++26")
add_compile_options("-std=c++27")
add_compile_options("-std=c++28")
add_compile_options("-std=c++29")
add_compile_options("-std=c++30")
add_compile_options("-std=c++31")
add_compile_options("-std=c++32")
add_compile_options("-std=c++33")
add_compile_options("-std=c++34")
add_compile_options("-std=c++35")
add_compile_options("-std=c++36")
add_compile_options("-std=c++37")
add_compile_options("-std=c++38")
add_compile_options("-std=c++39")
add_compile_options("-std=c++40")
add_compile_options("-std=c++41")
add_compile_options("-std=c++42")
add_compile_options("-std=c++43")
add_compile_options("-std=c++44")
add_compile_options("-std=c++45")
add_compile_options("-std=c++46")
add_compile_options("-std=c++47")
add_compile_options("-std=c++48")
add_compile_options("-std=c++49")
add_compile_options("-std=c++50")
add_compile_options("-std=c++51")
add_compile_options("-std=c++52")
add_compile_options("-std=c++53")
add_compile_options("-std=c++54")
add_compile_options("-std=c++55")
add_compile_options("-std=c++56")
add_compile_options("-std=c++57")
add_compile_options("-std=c++58")
add_compile_options("-std=c++59")
add_compile_options("-std=c++60")
add_compile_options("-std=c++61")
add_compile_options("-std=c++62")
add_compile_options("-std=c++63")
add_compile_options("-std=c++64")
add_compile_options("-std=c++65")
add_compile_options("-std=c++66")
add_compile_options("-std=c++67")
add_compile_options("-std=c++68")
add_compile_options("-std=c++69")
add_compile_options("-std=c++70")
add_compile_options("-std=c++71")
add_compile_options("-std=c++72")
add_compile_options("-std=c++73")
add_compile_options("-std=c++74")
add_compile_options("-std=c++75")
add_compile_options("-std=c++76")
add_compile_options("-std=c++77")
add_compile_options("-std=c++78")
add_compile_options("-std=c++79")
add_compile_options("-std=c++80")
add_compile_options("-std=c++81")
add_compile_options("-std=c++82")
add_compile_options("-std=c++83")
add_compile_options("-std=c++84")
add_compile_options("-std=c++85")
add_compile_options("-std=c++86")
add_compile_options("-std=c++87")
add_compile_options("-std=c++88")
add_compile_options("-std=c++89")
add_compile_options("-std=c++90")
add_compile_options("-std=c++91")
add_compile_options("-std=c++92")
add_compile_options("-std=c++93")
add_compile_options("-std=c++94")
add_compile_options("-std=c++95")
add_compile_options("-std=c++96")
add_compile_options("-std=c++97")
add_compile_options("-std=c++98")
add_compile_options("-std=c++99")
add_compile_options("-std=c++00")

```

Note: In the subsequent courses, we will not start from scratch to create a new "CMakeLists.txt" file. Instead, we will make some minor modifications to this existing file to integrate other drivers into the main function.

## Complete Code

---

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/v1.0/idf-code/Lesson07-Turn\\_on\\_the\\_screen](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/v1.0/idf-code/Lesson07-Turn_on_the_screen)

## Programming Steps

---

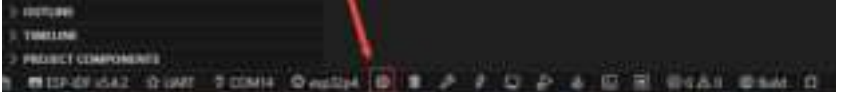
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable.



- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



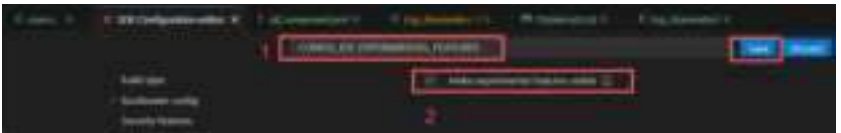
- Here, following the steps in the first section, first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



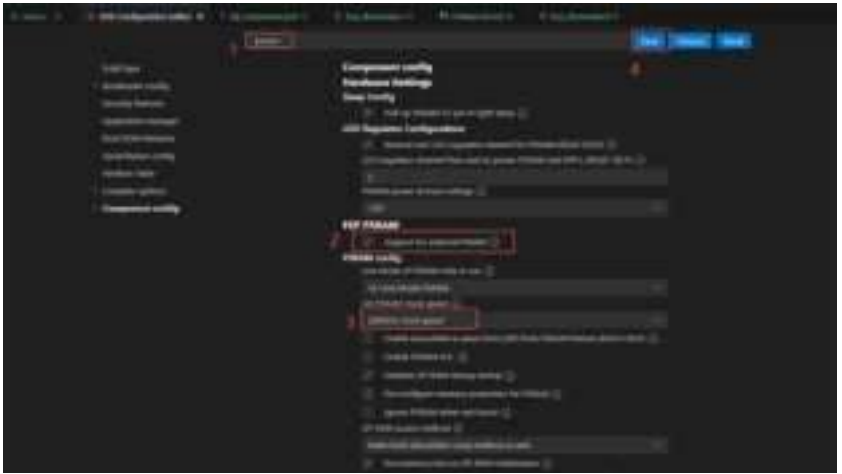
- Wait for a moment for the loading process to complete, and then you can proceed with the related SDK configuration.



After the SDK configuration is enabled, search for "CONFIG\_IDF\_EXPERIMENTAL\_FEATURES" in the search box, check the box, and then save the configuration.



- After setting "CONFIG\_IDF\_EXPERIMENTAL\_FEATURES", then search for PSRAM. Set it in sequence here to enable the 200M PSRAM, so that the screen can display a picture.
- Finally, remember to save the successfully configured configuration.
- Only by enabling the PSRAM option can there be sufficient RAM allocated to the screen. Enabling the CONFIG\_IDF\_EXPERIMENTAL\_FEATURES option allows you to select 200M PSRAM and use a higher RAM speed



- In order to meet the font size requirements for LVGL as specified in the previous code, here we need to search for "font", open the font, so that we can use the font set by LVGL.

```

48  lv_obj_t * label;
49  lv_obj_t * label2;
50
51  // 1. New label, font example
52  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
53
54  // 2. Configure label style (font, color, background)
55  lv_obj_set_style(label, LV_STYLE_FONT, LV_PART_MAIN);
56  lv_obj_set_style(label, LV_STYLE_COLOR, LV_PART_MAIN);
57  lv_obj_set_style(label, LV_STYLE_BG_COLOR, LV_PART_MAIN);
58
59  // 3. Set font, color, background
60  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
61  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
62  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
63  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
64  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
65  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
66  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
67  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
68  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
69  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
70  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
71  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
72  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
73  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
74  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
75  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
76  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
77  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
78  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
79  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
80  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
81  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
82  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
83  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
84  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
85  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
86  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
87  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
88  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
89  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
90  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
91  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
92  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
93  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
94  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
95  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
96  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
97  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
98  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
99  [lv_label_set_text(label, "Hello LVGL")]; // Set label text
100 [lv_label_set_text(label, "Hello LVGL")]; // Set label text

```



- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



After the configuration is completed, remember to save your settings.

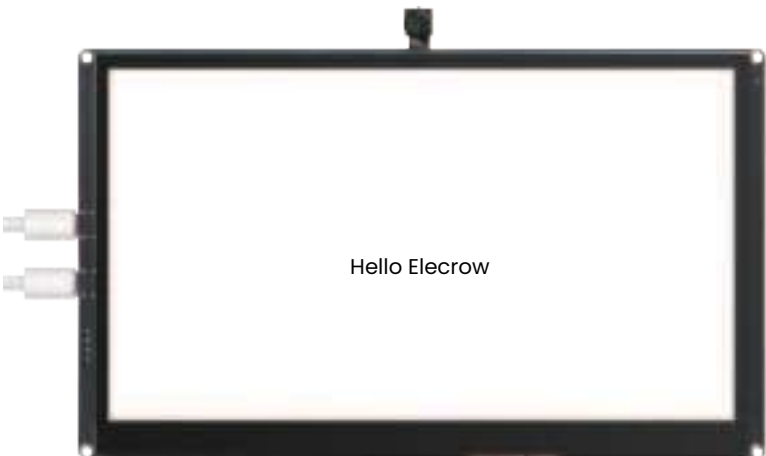
- Then we will compile and burn the code (as detailed in the first class).
- Here, we would like to introduce to you a very convenient feature. With just one button press, you can perform the compilation, upload, and open the monitor at once. (This is provided that the entire code is error-free.)



- After waiting for a while, the code compilation and upload were completed, and the monitor also opened.
- At this point, please remember to use another Type-C cable to connect your Advance-P4 through the USB2.0 interface. This interface provides a maximum current of about 500mA from the computer's USB-A port. When the Advance-P4 is using more external devices, especially the screen, it requires a sufficient current source. (It is recommended to use a charger for connection.)



- 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.



# Lesson 08

## 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.

### Hardware Used in This Lesson

---

#### SD card on the Advance-P4



## Operation Effect Diagram

---

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.



## 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.
- Here, we will prepare another new component "bsp\_sd" for everyone. The main function of this component is to implement the aforementioned file read and write operations.
- You only need to know when to call the interfaces we have written in it.
- Next, let's focus on understanding the "bsp\_sd" component.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson08-SD\\_Card\\_File\\_Reading](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson08-SD_Card_File_Reading)

- Then drag the code of this lesson into VS Code and open the project files.
- After opening, you can see the framework of this project.



In the example of this course, a new folder named `bsp_sd` is created under the peripheral directory. Within the `bsp_sd` folder, a new include folder and a "CMakeLists.txt" file are created.

The `bsp_sd` folder contains the driver file "bsp\_sd.c", and the include folder contains the header file "bsp\_sd.h".

The "CMakeLists.txt" file integrates the driver into the build system, enabling the project to utilize the SD card read/write functionality implemented in "bsp\_sd.c".

## Code for SD Card File Reading and Writing

- The code for SD card file reading and writing consists of two files: "bsp\_sd.c" and "bsp\_sd.h".
- Next, we will first analyze the "bsp\_sd.h" program.
- "bsp\_sd.h" is the header file of the file read-write module, and its main functions are as follows:
  - Declare the functions, macros, and variables implemented in "bsp\_sd.c" for use by external programs.
  - Allow other .c files to call this module simply by adding the directive `#include "bsp_sd.h"`.
  - In other words, it serves 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, all the libraries we need to use are included in the "bsp\_sd.h" file, enabling unified management.

```
1 //----- Header File Declaration -----#
2 #include <stdio.h> // include standard C input/output functions
3 #include <stdint.h> // include C99 standard integer types
4 #include <string.h> // include C standard string functions
5 #include <sys/types.h> // include C standard system types
6 #include <unistd.h> // include C standard UNIX system calls
7 #include <fcntl.h> // include C standard file control functions
8 #include <sys/stat.h> // include C standard file status functions
9 #include <sys/time.h> // include C standard time functions
10 #include <sys/mman.h> // include C standard memory management functions
11 //----- main file declaration and -----#
```

- Next, we declare the variables and functions we need to use. The specific implementation of these functions resides in "bsp\_sd.c".
- Concentrating these declarations in "bsp\_sd.h" is for the convenience of calling and management. (We will learn about their specific roles when they are used in "bsp\_sd.c".)

```

#ifndef BSP_SD_H
#define BSP_SD_H

#include "bsp_sd.h"

// SD card status
#define SD_CARD_READY 0x00000000
#define SD_CARD_BUSY 0x00000001
#define SD_CARD_ERROR 0x00000002

// SD card operations
void bsp_sd_init(void);
void bsp_sd_write(uint8_t *data, uint32_t len);
void bsp_sd_read(uint8_t *data, uint32_t len);
void bsp_sd_erase(uint32_t start, uint32_t end);
void bsp_sd_format(void);
void bsp_sd_mount(void);
void bsp_sd_unmount(void);
void bsp_sd_test(void);

#endif

```

- Now let's look at the specific functions of each function in "bsp\_sd.c".
- The "bsp\_sd" component provides significant support for everyone to use file read-write interfaces in the future. By understanding the functions of these functions clearly, you can flexibly read from and write to the SD card file system.
- It includes the custom header file "bsp\_sd.h", which defines function declarations, log macros, constants, and paths.
- "card" stores information such as the status, capacity, and speed of the SD card device.
- "sd\_mount\_point" is the file system mounting directory of the SD card.

```

// SD card initialization
void bsp_sd_init(void)
{
    // Initialize SD card
    bsp_sd_test();
    bsp_sd_format();
    bsp_sd_mount();
}

// SD card test
void bsp_sd_test(void)
{
    // Test SD card
    bsp_sd_read(0, 1024);
    bsp_sd_write(0, 1024);
}

// SD card format
void bsp_sd_format(void)
{
    // Format SD card
    bsp_sd_erase(0, 0xFFFFFFFF);
}

// SD card mount
void bsp_sd_mount(void)
{
    // Mount SD card
    bsp_sd_mount_point = "/mnt/sdcard";
}

// SD card unmount
void bsp_sd_unmount(void)
{
    // Unmount SD card
    bsp_sd_mount_point = "/";
}

```

### 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.

```
10 int main() {
11     // Create a file in binary write mode
12     FILE *file = fopen("test.txt", "wb");
13     if (file == NULL) {
14         // Failed to create file
15         return ESP_FAIL;
16     }
17     fclose(file);
18     // File created successfully
19     return ESP_OK;
20 }
```

### 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.

```
10 int main() {
11     // Open a file in text write mode
12     FILE *file = fopen("test.txt", "w");
13     if (file == NULL) {
14         // Failed to open file for writing string
15         return ESP_FAIL;
16     }
17     fprintf(file, "Hello, world!");
18     fclose(file);
19     // String written successfully
20     return ESP_OK;
21 }
```

### 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".

```
100 int main() {
101     char str[64];
102
103     // printf("Reading file %s", filename);
104     FILE *file = fopen(filename, "r");
105     if (!file)
106     {
107         // perror("Failed to open file for reading string");
108
109         return -1;
110     }
111
112     char data[64];
113     fgets(data, sizeof(data), file);
114     fclose(file);
115
116     char *ptr = strtok(data, "\n");
117     if (!ptr)
118     {
119         // printf("Read a line from file %s", filename);
120     }
121     else
122     {
123         // printf("Read from file %s", filename);
124         return 0;
125     }
126 }
```

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.

```
100 int main() {
101     char str[64];
102
103     // printf("Reading file %s", filename);
104     FILE *file = fopen(filename, "r");
105     if (!file)
106     {
107         // perror("Failed to open file for reading string");
108
109         return -1;
110     }
111
112     char data[64];
113     fgets(data, sizeof(data), file);
114     fclose(file);
115
116     char *ptr = strtok(data, "\n");
117     if (!ptr)
118     {
119         // printf("Read a line from file %s", filename);
120     }
121     else
122     {
123         // printf("Read from file %s", filename);
124         return 0;
125     }
126 }
```

```
100 int main() {
101     char str[64];
102
103     // printf("Reading file %s", filename);
104     FILE *file = fopen(filename, "r");
105     if (!file)
106     {
107         // perror("Failed to open file for reading string");
108
109         return -1;
110     }
111
112     char data[64];
113     fgets(data, sizeof(data), file);
114     fclose(file);
115
116     char *ptr = strtok(data, "\n");
117     if (!ptr)
118     {
119         // printf("Read a line from file %s", filename);
120     }
121     else
122     {
123         // printf("Read from file %s", filename);
124         return 0;
125     }
126 }
```

### **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.

```
47 int write_file(const char *filename, const char *data, size_t size)
48 {
49     int success_size = 0;
50     FILE *file = fopen(filename, "wb");
51     if (!file)
52     {
53         perror("failed to open file for writing");
54         return -1;
55     }
56     success_size = fwrite(data, 1, size, file);
57     if (success_size != size)
58     {
59         fclose(file);
60         perror("failed to write file");
61         return -1;
62     }
63     else
64     {
65         fclose(file);
66         printf("File written");
67     }
68     return success_size;
69 }
```

### **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.

```

200 int main() {
201     int i;
202     struct timeval tv;
203     FILE *file = fopen("data", "w");
204     if (!file)
205     {
206         perror("failed to open file for writing");
207         return 0;
208     }
209     if (fwrite("data", sizeof(char), 8, file) != 8)
210     {
211         perror("failed to write file");
212         return 0;
213     }
214     fclose(file);
215     if (system("cat > data"))
216     {
217         perror("failed to read file");
218         return 0;
219     }
220     else
221     {
222         printf("File read success");
223     }
224     return 0;
225 }

```

### 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.

```

228 int main() {
229     int i;
230     struct timeval tv;
231     FILE *file = fopen("data", "r");
232     if (!file)
233     {
234         perror("failed to open file for reading");
235         return 0;
236     }
237     int size = fread(data, 1, 8, file);
238     if (size != 8)
239     {
240         perror("failed to read file");
241         return 0;
242     }
243     else
244     {
245         printf("File read success");
246     }
247     return 0;
248 }

```

### 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.

```
10 int main() { read_file_size("src", "read_filename");
11 }
12
13 #include <stdio.h>
14 #include <string.h>
15
16 int read_file_size(const char *read_filename)
17 {
18     FILE *read_file = fopen(read_filename, "r");
19     if (!read_file)
20     {
21         perror("Failed to open file for reading");
22         return -1;
23     }
24     char *buffer = (char *) malloc(1024);
25     while (!feof(read_file) && fread(buffer, 1, 1024, read_file) > 0)
26     {
27         size += read_success_size;
28     }
29     fclose(read_file);
30     printf("File read success, success size = %d", size);
31     return 0;
32 }
```

### 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.

```

101 int sd_init(struct esp_vfs_fat_config_t *config, const char *read_filename, const char *write_filename)
102 {
103     char *read_success_size = 0;
104     char *write_success_size = 0;
105     size_t size = 0;
106     FILE *read_file = fopen(read_filename, "r");
107     FILE *write_file = fopen(write_filename, "w");
108     if (!read_file)
109     {
110         LOG_ERROR("failed to open file for reading");
111         return ESP_FAIL;
112     }
113     if (!write_file)
114     {
115         LOG_ERROR("failed to open file for writing");
116         return ESP_FAIL;
117     }
118     char *buffer[1024];
119     while ((read_success_size = fread(buffer, 1, sizeof(buffer), read_file)) > 0)
120     {
121         write_success_size = fwrite(buffer, 1, read_success_size, write_file);
122         if (write_success_size != read_success_size)
123         {
124             LOG_ERROR("inconsistent reading and writing of data");
125             return ESP_FAIL;
126         }
127         size += write_success_size;
128     }
129     fclose(read_file);
130     fclose(write_file);
131     LOG_INFO("file read and write success, success size %d", size);
132     return ESP_OK;
133 }

```

## 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.

```

104 int main() {
105     // Initialize SPI
106     spi_init(&spi, &spi_dev);
107     spi_set_speed(&spi, SPI_SPEED);
108     spi_set_mode(&spi, SPI_MODE0);
109     spi_set_cs(&spi, CS_PIN, CS_ACTIVE);
110     spi_set_pins(&spi, {
111         .miso = MISO_PIN,
112         .mosi = MOSI_PIN,
113         .sclk = SCLK_PIN,
114         .cs   = CS_PIN,
115     });
116     // Initialize SD card
117     sd_card = sd_card_init(&spi);
118     if (!sd_card) {
119         printf("Failed to initialize SD card\n");
120         return -1;
121     }
122     // Get SD card info
123     sd_card_info_t info;
124     if (!sd_card_get_info(&sd_card, &info)) {
125         printf("Failed to get SD card info\n");
126         return -1;
127     }
128     // Print SD card info
129     printf("SD card info: type=%d, capacity=%d, speed=%d\n",
130           info.type, info.capacity, info.speed);
131     // Format SD card
132     if (!sd_card_format(&sd_card, "fat32", 1)) {
133         printf("Failed to format SD card\n");
134         return -1;
135     }
136     // Mount SD card
137     if (!sd_card_mount(&sd_card, "/mnt/sdcard")) {
138         printf("Failed to mount SD card\n");
139         return -1;
140     }
141     // Write data to SD card
142     if (!sd_card_write(&sd_card, "data.txt", "Hello, World!")) {
143         printf("Failed to write data to SD card\n");
144         return -1;
145     }
146     // Read data from SD card
147     if (!sd_card_read(&sd_card, "data.txt", &data)) {
148         printf("Failed to read data from SD card\n");
149         return -1;
150     }
151     // Print data from SD card
152     printf("Data from SD card: %s\n", data);
153     return 0;
154 }

```

### get\_sd\_card\_info:

Print detailed information such as the type, capacity, and speed of the SD card to the console.

```

100 | printf("SD card info: type=%d, capacity=%d, speed=%d\n",
101 |       info.type, info.capacity, info.speed);
102 | }

```

### 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.

```

100 // Format SD card
101 if (!sd_card_format(&sd_card, "fat32", 1)) {
102     printf("Failed to format SD card\n");
103     return -1;
104 }
105 // Mount SD card
106 if (!sd_card_mount(&sd_card, "/mnt/sdcard")) {
107     printf("Failed to mount SD card\n");
108     return -1;
109 }
110 // Write data to SD card
111 if (!sd_card_write(&sd_card, "data.txt", "Hello, World!")) {
112     printf("Failed to write data to SD card\n");
113     return -1;
114 }
115 // Read data from SD card
116 if (!sd_card_read(&sd_card, "data.txt", &data)) {
117     printf("Failed to read data from SD card\n");
118     return -1;
119 }
120 // Print data from SD card
121 printf("Data from SD card: %s\n", data);
122 return 0;

```

- That concludes our introduction to the "bsp\_sd" component. It's sufficient for everyone to understand how to call these interfaces.
- If you need to call them, you must also configure the "CMakeLists.txt" file under the "bsp\_sd" folder.
- This file, placed in the "bsp\_sd" folder, mainly functions to tell the build system (CMake) of "ESP-IDF" how to compile and register the "bsp\_sd" component.

```

cmake_minimum_required(VERSION 3.10)
project("bsp_sd")

include_directories("${CMAKE_CURRENT_SOURCE_DIR}/include")

add_library(bsp_sd STATIC
    src/bsp_sd.c
)

target_link_libraries(bsp_sd PRIVATE
    "fatfs"
)
  
```

- The reason why "fatfs" is involved here is that we have called it in "bsp\_sd.h" (other libraries that are system libraries do not need to be added).

```

#ifndef BSP_SD_H
#define BSP_SD_H

#include "esp_err.h"

void bsp_sd_init(void);
void bsp_sd_write(uint32_t address, uint32_t data);
void bsp_sd_read(uint32_t address, uint32_t data);
void bsp_sd_erase(uint32_t address, uint32_t data);

#endif
  
```

## Main function

- The main folder is the core directory for program execution, and it contains the executable file main.c for the main function.
- Add the main folder to the "CMakeLists.txt" file of the build system.

```

cmake_minimum_required(VERSION 3.10)
project("main")

include_directories("${CMAKE_CURRENT_SOURCE_DIR}/include")

add_subdirectory("bsp_sd")

add_executable(main
    src/main.c
)

target_link_libraries(main PRIVATE
    bsp_sd
)
  
```

- This is the entry file of the entire application. In ESP-IDF, there is no `int main()`, and the program starts running from `void app_main(void)`.
- Let's first explain "main.c".
- When the program runs, the general process is as follows:

### Initialization Phase

`sd_init()` → Detects and mounts the SD card.

### File Operation Phase

Users call encapsulated functions such as:

`write_string_file()` to write data;

`read_string_file()` for reading and verification;

### Debug Log Output

All operations have "SD\_INFO()" log output for debugging purposes.

### Exception Handling

If file opening, reading, or writing fails, it will immediately return "ESP\_FAIL" and print an error log.

- Next, let's explain the main code file "main.c".

```

1 // This file is the main entry point of the application.
2 #include "main.h" // include the main header file containing variables, definitions and declarations
3 // ...
4 // ...

```

- First, it includes the custom main header file "main.h". This header file usually contains log macros, peripheral initialization declarations, SD card-related function declarations, and more.
- In essence, including this file enables the current "main.c" to call system initialization functions and SD card functional functions.
- Below is the content included in "main.h":



- The following defines a FreeRTOS task handle.
- It is used to record the created SD card test task "sd\_task", facilitating system management.

```

5 //
6 //
7 //
8 //
9 //
10 //
11 //
12 //
13 //
14 //
15 //
16 //
17 //
18 //
19 //
20 //
21 //
22 //
23 //
24 //
25 //
26 //
27 //
28 //
29 //
30 //
31 //
32 //
33 //
34 //
35 //
36 //
37 //
38 //
39 //
40 //
41 //
42 //
43 //
44 //
45 //
46 //
47 //
48 //
49 //
50 //
51 //
52 //
53 //
54 //
55 //
56 //
57 //
58 //
59 //
60 //
61 //
62 //
63 //
64 //
65 //
66 //
67 //
68 //
69 //
70 //
71 //
72 //
73 //
74 //
75 //
76 //
77 //
78 //
79 //
80 //
81 //
82 //
83 //
84 //
85 //
86 //
87 //
88 //
89 //
90 //
91 //
92 //
93 //
94 //
95 //
96 //
97 //
98 //
99 //
100 //

```

- The following is a FreeRTOS task function, whose main function is to repeatedly test the read and write functions of the SD card.

```

24 void sd_test(void *pvParameters) // SD card test task function
25 {
26     //log_printf(0, "sd_test: %d", 0); // variables to store function return values (error codes)
27
28     //open file "file_hello" in OS mode "w+" "hello.txt" // file path for the card test file
29     char *file = "hello.txt"; // data to be written into the file
30
31     //get file size information
32     get_fd_size(file); // Open the file with os_open, read, and close
33
34     while (1) // infinite loop to perform read/write test
35     {
36         //WRITE DATA TO FILE
37         int i = write_string_file(file, file, 0); // write the "hello.txt" string to the file
38         if (i == 0) // check if writing failed
39         {
40             //log_printf(0, "file %s failed", file); // Print error message if writing fails
41             continue; // continue to next iteration of loop
42         }
43
44         //Read data from file
45         int i = read_string_file(file, file); // read the content from the written file
46         if (i == 0) // check if reading failed
47         {
48             //log_printf(0, "file %s failed", file); // Print error message if reading fails
49         }
50
51         //taskDelay(1000); // delay 1 second before repeating the test
52         //log_printf(0, "sd_test: %d", 1); // log message following test finished successfully
53         //taskDelay(1000); // delay 1 second after finishing the test
54     }
55 }

```

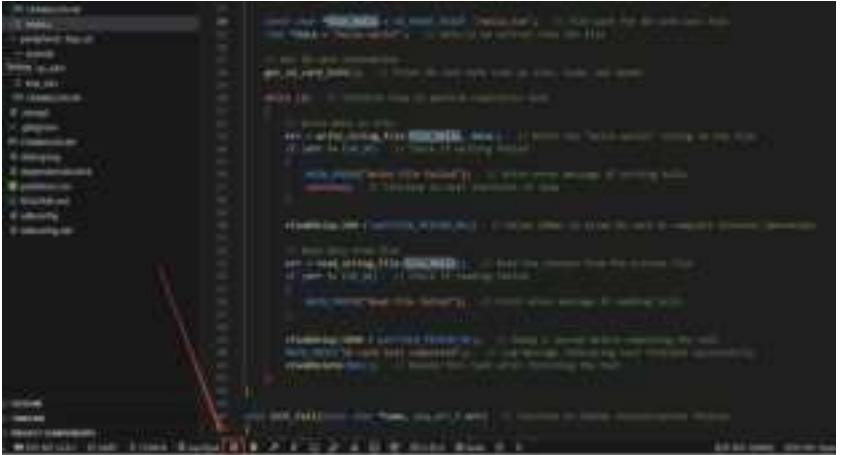
## Among them:

"file\_hello" is the file path (usually "/sdcard/hello.txt").

"data" is the string content to be written to the file.

Note: If your file name is too long, the read and write operations will eventually fail. You can do the following:

Click "SDK Configuration Editor".



- This way, you can adapt to longer file names.
- Then, the subsequent operations in the "sd\_task" function are as follows: first, obtain the SD card information, then write the data you want to write into the file with the specified path and name, and delay for 200ms. This delay is to wait for the write operation to stabilize and succeed, so that you can smoothly read out the content you wrote.

```

// ...
sd_card_info_t sd_card_info;
// ...
int sd_init() {
    // ...
    return 0;
}

```

- When the module initialization fails (such as the SD card not being inserted, wrong wiring, etc.), it will cyclically print error logs and block the program.
- The function is to prevent the execution of tasks in an error state from continuing.
- The code here calls "sd\_init" from the "bsp\_sd" component to initialize our SD card, which is a prerequisite for performing operations on the SD card.

```

// ...
int main() {
    // ...
    sd_init();
    // ...
}

```

- Then there is the main function app\_main.
- ESP-IDF projects start executing from app\_main():
  - Print startup information;
  - Call Init() to complete SD card initialization;
  - Create a task with: xTaskCreatePinnedToCore(sd\_task, "sd\_task", 4096, NULL, 5, &sd\_task\_handle, 1);
    - Name: sd\_task
    - Stack size: 4096 bytes
    - Priority: 5
    - Runs on CPU core 1
- Print "SD card test begin" to indicate that the test task has started.



- Finally, let's understand the "CMakeLists.txt" file in the "main" directory.
- The role of this CMake configuration is:
  - Collect all ".c" source files in the "main/" directory as the component's source files;
  - Register the "main" component with the ESP-IDF build system and declare that it depends on the custom component "bsp\_sd".
- This way, during the build process, ESP-IDF knows to first build "bsp\_sd" and then build "main".



Note: In subsequent courses, we will not create a new "CMakeLists.txt" file from scratch. Instead, we will make some minor modifications to this existing file to integrate other drivers into the main function.

## Complete Code

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson08-SD\\_Card\\_File\\_Reading](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson08-SD_Card_File_Reading)

## Programming Steps

---

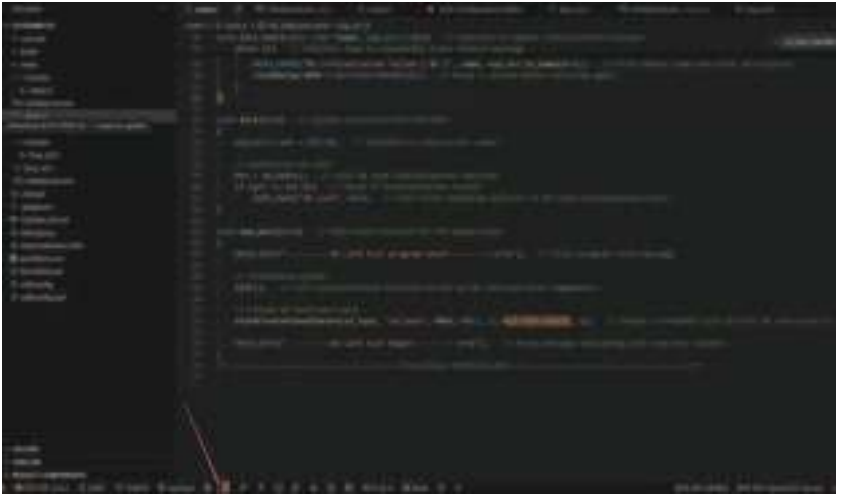
- Now the code is ready. Next, we need to flash it to the ESP32-P4 to observe the actual behavior.
- First, connect the Advance-P4 device to your computer via a USB cable.



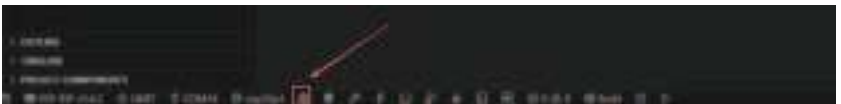
- Then insert the SD card you will use into the SD card slot of the Advance-P4.



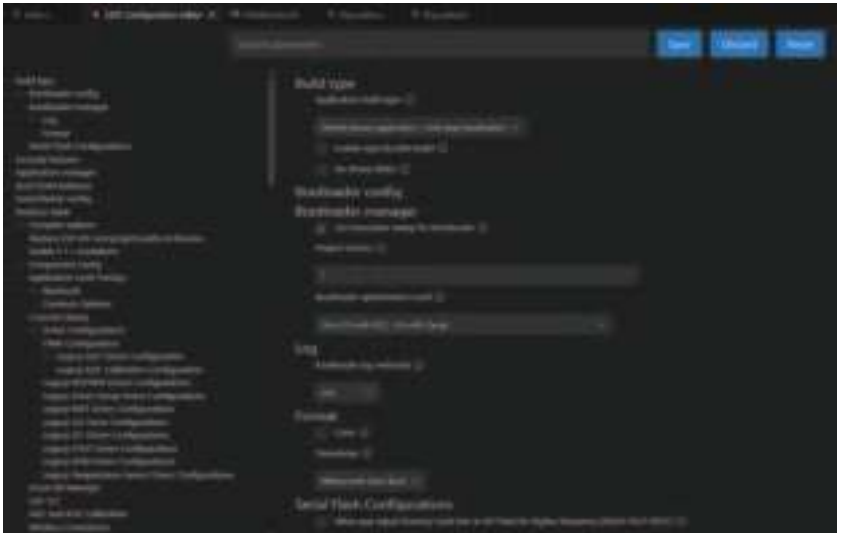
- Before starting the flashing process, first delete all files generated during compilation to restore the project to an "unbuilt" initial state. (This ensures that subsequent compilations are not affected by your previous build residues.)



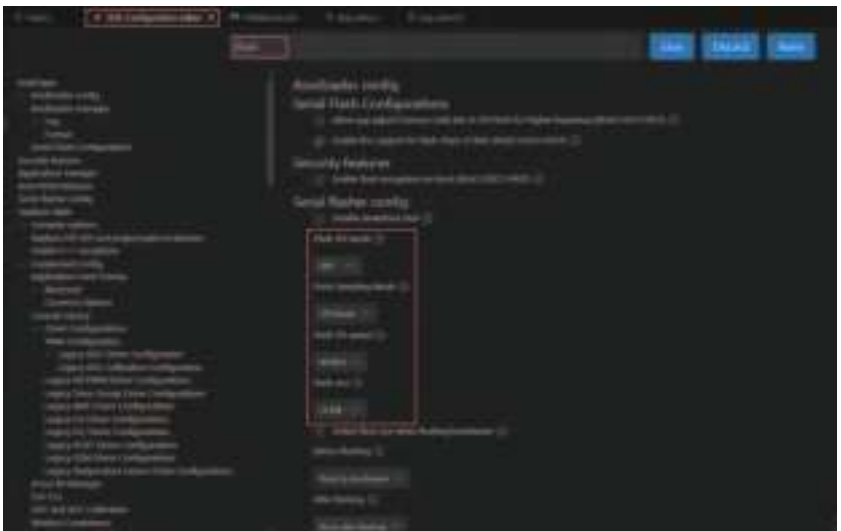
- First, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon shown in the figure.



- Wait for a moment while it loads, and then you can proceed with the relevant SDK configurations.



- Subsequently, search for "flash" in the search box(Ensure your flash configuration matches mine).



- After completing the configuration, remember to save your settings.
- Then we can compile and flash the code (as detailed in the first lesson).
- Here we'd like to introduce a very convenient feature: there's a single button that can execute compilation, uploading, and opening the monitor all at once. (This works on the premise that the entire code is error-free.)



- After waiting for a while, the code compilation and upload will be completed, and the monitor will open automatically.
- Once the code runs, 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 inside.



# Lesson 09

## 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.

### Hardware Used in This Lesson

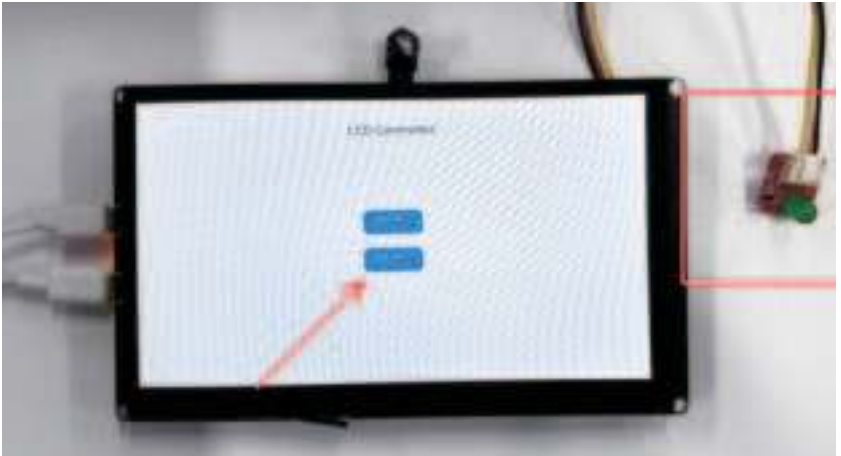
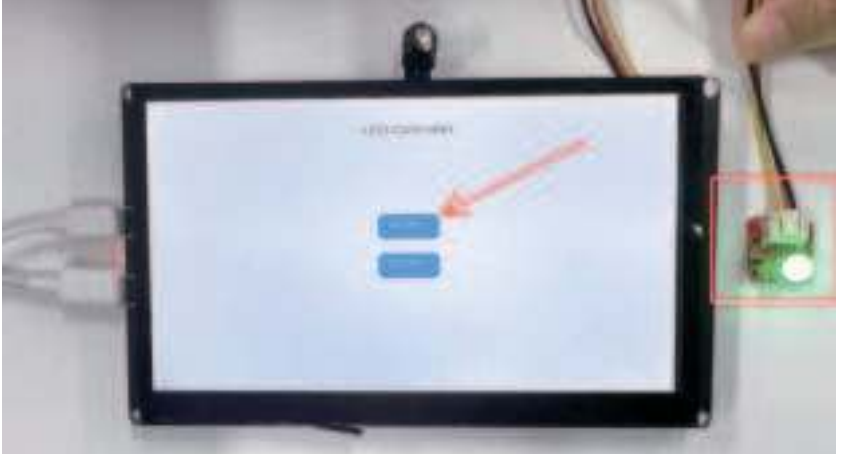
---

The UART1 interface on the Advance-P4 is connected to an LED.



## Operation Effect Diagram

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.



## Key Explanations

- Now, the focus of this lesson is on how to use LVGL to create button objects and display the LVGL interface on the screen to achieve interactive effects.
- First, click the GitHub link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson09-LVGL\\_Lighting\\_Control](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson09-LVGL_Lighting_Control)

- Then drag the code for this lesson into VS Code and open the project file.
- Once opened, you can see the framework of this project.



It can be seen that the components we use in this lesson are all those explained in previous sessions:

- bsp\_display: Touch-related driver.
- bsp\_i2c: Provides I2C driver support required for touch functionality.
- bsp\_extra: Used to control the LED connected to the UART1 interface.
- bsp\_illuminate: Responsible for screen initialization, screen lighting, and LVGL initialization.

## LVGL Initialization Code

- The components used in this lesson have been explained in detail in previous courses.
- Here, we will only describe the LVGL initialization in detail.
- `lvgl_init()` is the core initialization function of the entire graphic display system.
- It mainly completes the following tasks:
  - Initializes the LVGL operating task environment (task/timer)
  - Registers and binds the display driver (Display) with LVGL's rendering layer
  - Registers and binds the touch input driver (Touch) to the LVGL input system
- The purpose of doing this is to ensure that LVGL's graphic rendering, screen refreshing, and touch event handling are correctly linked with the underlying hardware.

A screenshot of a code editor with a dark background and light-colored text. The code is in C and shows the initialization of LVGL. It includes comments in Chinese and function calls like `lvgl_init()`. The code is partially obscured by a semi-transparent window in the foreground, which also contains code. The overall appearance is that of a development environment on a computer screen.

- This part starts the LVGL task and timer through `lvgl_port_init()`, completing the following:
  - Allocating stack space for the LVGL main task (LVGL task);
  - Setting the task priority;
  - Configuring LVGL's periodic refresh timer;
  - Defining the maximum sleep time (i.e., the time the LVGL main loop sleeps when idle);
- The LVGL task continuously calls `lv_timer_handler()` to refresh the UI, process animations, and respond to events.



- rotation: Screen rotation/mirror configuration
- flags:
  - buff\_dma, buff\_spiram: Whether the buffer is placed in internal memory or external "PSRAM"
  - full\_refresh: Whether to enable full-frame refresh mode
  - direct\_mode: Whether to directly output LVGL rendering results to the screen (reducing intermediate layers)
- Significance: All LVGL drawing operations will ultimately be updated to your screen through this display interface.



- Register the touch input device with LVGL so that it can receive finger touch events.
- The initialization content includes:
  - **disp**: The bound display object (the touch area corresponds to the screen)
  - **handle**: Touch driver handle (such as "FT5x06", "GT911", "CST816", etc.)
- Significance: Only in this way can LVGL's internal event system (such as button clicks, swipes) obtain touch coordinate data. After this part of the initialization, clicking buttons on the screen will produce visible effects.
- This concludes our explanation of the components.

## Main function

- The main folder is the core directory for program execution, which contains the main function executable file main.c.
- Add the main folder to the "CMakeLists.txt" file of the build system.



- This is the entry file of the entire application. In ESP-IDF, there is no `int main()`, and execution starts from `void app_main(void)`.
- Let's first explain main.c to see how the interfaces in these four components are called to achieve the LVGL lighting effect. It creates a simple interface on the touch screen, containing two buttons labeled "LED ON" and "LED OFF" to control the LED on GPIO48.

```
4 /* LDO channel handle */
5 static esp_ldo_channel_handle_t ldo01 = NULL;
6 static esp_ldo_channel_handle_t ldo04 = NULL;
```

```
78 static void system_init(void) {
79     esp_err_t err = ESP_OK;
80
81     /* 1. Initialization for (powered) for (unpowered)
82      * esp_ldo_channel_config_t ldo01 = {
83      *     .chan_id = 0,
84      *     .voltage_mv = 3000,
85      * };
86      * err = esp_ldo_acquire_channel(&ldo01, &ldo01);
87      * if (err != ESP_OK) task_delay_microseconds(100);
88      *
89      * esp_ldo_channel_config_t ldo04 = {
90      *     .chan_id = 4,
91      *     .voltage_mv = 3300,
92      * };
93      * err = esp_ldo_acquire_channel(&ldo04, &ldo04);
94      * if (err != ESP_OK) task_delay_microseconds(100);
95      *
96      * MAIN_GPIO_PIN and GPIO48 init success!
97     */
98 }
```

- Function: "LDO" (Low Dropout Regulator) is a low dropout regulator used to supply power to devices such as screens and touch chips.
- Two channels are enabled here:
  - "LDO3" outputs 2.5V (to power the screen)
  - "LDO4" outputs 3.3V (to power logic circuits or other peripherals)
- After successful initialization: Provides stable power for subsequent LCD and touch modules.

```

// 4. Initialize I2C (required for touch chip)
void initI2C() {
  Serial.println("Initializing I2C...");
  Wire.begin();
  if (Wire != I2C_OK) initFailHandler("I2C", Wire);
  Serial.println("I2C init success");
}

// 5. Initialize touch panel (low-level driver)
void initTouchPanel() {
  Serial.println("Initializing touch panel...");
  Wire.begin();
  if (Wire != I2C_OK) initFailHandler("Touch", Wire);
  Serial.println("Touch panel init success");
}

// 6. Initialize I2D hardware and I2D (must initialize before turning on backlight)
void initI2D() {
  if (Wire != I2C_OK) initFailHandler("I2D", Wire);
  Serial.println("I2D init success");
}

```

- Initialize the "I2C" bus for communication with the touch chip.
- The touch input part of LVGL usually needs to read coordinates via I2C.
- After successful initialization: The system can obtain touch event coordinate data through I2C.

```

// 4. Initialize I2C (required for touch chip)
void initI2C() {
  Serial.println("Initializing I2C...");
  Wire.begin();
  if (Wire != I2C_OK) initFailHandler("I2C", Wire);
  Serial.println("I2C init success");
}

// 5. Initialize touch panel (low-level driver)
void initTouchPanel() {
  Serial.println("Initializing touch panel...");
  Wire.begin();
  if (Wire != I2C_OK) initFailHandler("Touch", Wire);
  Serial.println("Touch panel init success");
}

```

- Function: Initialize the touch driver and register touch interrupts or polling read mechanisms.
- Enable LVGL to receive touch events (clicks, swipes, etc.).
- After successful initialization: User clicks on the screen can trigger LVGL events.

```

199 // 5. Turn on LCD backlight (brightness can be 0-255 = max)
200 lcd = display_init();
201 // (err != ESP_OK) init_fail_handler("LCD", err);
202 while (lcd != LCD_INIT_SUCCESS);
203
204 // 6. Turn on LED backlight (brightness can be 0-255 = max)
205 led = set_led_brightness(100);
206 // (err != ESP_OK) init_fail_handler("LED backlight", err);
207 while (led != LED_BACKLIGHT_OPENED (brightness: 100));
208

```

- Function: "display\_init()": Initialize the LCD hardware interface and initialize the LVGL library;
- "set\_lcd\_blight(100)": Turn on the screen backlight brightness (100 indicates maximum brightness).
- After successful initialization: The LVGL graphics system starts running, and the screen can display UI elements.

```

// 6. Initialize LED control GPIO (GPIO48)
PINN_INIT("initializing GPIO48 for LED...");
err = gpio_extra_init();
if (err != ESP_OK) init_fail_handler("GPIO48", err);
gpio_extra_set_level(false); // Initially turn off LED
PINN_INIT("LED initialized to OFF state");

```

- Function: Configure "GPIO48" as an output pin;
- Control the LED switch through "gpio\_extra\_set\_level(true/false)".
- After successful initialization: The system can turn the LED on or off through button clicks.

```

// 7. Create UI
create_led_control_ui();
while (true) {
    if (btn_on_click_event()) {
        // Turn on LED
    }
    if (btn_off_click_event()) {
        // Turn off LED
    }
}

```

Function: Create a concise interface using LVGL:

- Background: white;
- Title: "LED Controller";
- Two buttons:
  - "LED ON": Triggers btn\_on\_click\_event() to turn on the LED;
  - "LED OFF": Triggers btn\_off\_click\_event() to turn off the LED.

Now let's take a look inside this function.

```

24 // Create the control id
25 static void create_id_control_id(void)
26 {
27     // Create main screen
28     lv_obj_t *scr = lv_scr_act();
29     lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFF), LV_PART_MAIN); // Set white background
30
31     // Create title label
32     lv_obj_t *label = lv_label_create(scr);
33     lv_label_set_text(label, "LV Control");
34     lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 0);
35
36     // Main div
37     lv_obj_set_style_text_font(label, &lv_font_monospace_24, 0);
38
39     // Create on button
40     lv_obj_t *btn_on = lv_btn_create(scr);
41     lv_obj_set_size(btn_on, 100, 50);
42     lv_obj_align(btn_on, LV_ALIGN_BOTTOM_M, 0, -50);
43     lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);
44
45     // On button label
46     lv_obj_t *label_on = lv_label_create(btn_on);
47     lv_label_set_text(label_on, "ON");
48
49     // Create off button
50     lv_obj_t *btn_off = lv_btn_create(scr);
51     lv_obj_set_size(btn_off, 100, 50);
52     lv_obj_align(btn_off, LV_ALIGN_BOTTOM_M, 0, 50);
53     lv_obj_add_event_cb(btn_off, btn_off_click_event, LV_EVENT_CLICKED, NULL);
54
55     // Off button label
56     lv_obj_t *label_off = lv_label_create(btn_off);
57     lv_label_set_text(label_off, "OFF");
58 }

```

### lv\_scr\_act():

Obtains the currently active screen object (LVGL has only one main screen by default).

You can understand it as "I want to place things on the current screen".

### lv\_obj\_set\_style\_bg\_color()

Sets the background color of this screen to white (0xFFFF).

```

24 // Create the control id
25 static void create_id_control_id(void)
26 {
27     // Create main screen
28     lv_obj_t *scr = lv_scr_act();
29     lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFF), LV_PART_MAIN); // Set white background
30
31     // Create title label
32     lv_obj_t *label = lv_label_create(scr);
33     lv_label_set_text(label, "LV Control");
34     lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 0);
35
36     // Main div
37     lv_obj_set_style_text_font(label, &lv_font_monospace_24, 0);

```

- This section creates and configures a title text:
- 'lv\_label\_create(scr)': Creates a text label object on the main screen.
- 'lv\_label\_set\_text()': Sets the text content to "LED Controller".
- 'lv\_obj\_align()': Sets the alignment to top-center, with a downward offset of 50 pixels.
- 'lv\_obj\_set\_style\_text\_font()': Sets the font size to 24pt.
- Result: A large-sized title "LED Controller" is displayed centered at the top of the screen.

```

100 lv_obj_t *label;
101
102 lv_label_create(&label, scr);
103 lv_label_set_text(label, "LED Controller");
104 lv_obj_align(label, LV_ALIGN_TOP_CENTER, 0, -50);
105 lv_obj_set_style_text_font(label, &lv_font_montserrat_24, 0);
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

- 'lv\_btn\_create(scr)': Creates a button object and places it on the main screen.
- 'lv\_obj\_set\_size()': Sets the button size to 120×50 pixels.
- 'lv\_obj\_align()': Aligns the button to the center, with an upward offset of 40 pixels.
- 'lv\_obj\_add\_event\_cb()': Binds a button event—when the button is "clicked", it calls the 'btn\_on\_click\_event()' function.
- Within this function, 'gpio\_extra\_set\_level(true);' is executed → turning on the LED.
- Result: A button is created slightly above the center of the screen, used for "turning on the light".

```

100 lv_obj_t *label;
101
102 lv_label_create(&label, scr);
103 lv_label_set_text(label, "LED Controller");
104 lv_obj_align(label, LV_ALIGN_TOP_CENTER, 0, -50);
105 lv_obj_set_style_text_font(label, &lv_font_montserrat_24, 0);
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

- This label is a child object of the button (created within the button).
- Its text will be automatically displayed in the center of the button.
- Result: The text "LED ON" is displayed on the button.

```

// Create the LED control button
lv_obj_t * btn_led = lv_btn_create(app_win);
lv_obj_t * label_led = lv_label_create(btn_led);
lv_label_set_text(label_led, "LED ON");
lv_obj_set_pos(btn_led, 100, 100);
lv_obj_set_size(btn_led, 100, 50);
lv_obj_add_event_cb(btn_led, btn_led_click_event_cb, LV_EVENT_CLICKED, NULL);

// Create the LED control button
lv_obj_t * btn_led = lv_btn_create(app_win);
lv_obj_t * label_led = lv_label_create(btn_led);
lv_label_set_text(label_led, "LED OFF");
lv_obj_set_pos(btn_led, 100, 150);
lv_obj_set_size(btn_led, 100, 50);
lv_obj_add_event_cb(btn_led, btn_led_click_event_cb, LV_EVENT_CLICKED, NULL);

```

- The "OFF" button is created using the same logic.
- Now let's look at the events bound to these two buttons after they are clicked.

```

// LED ON event handler
void btn_led_click_event_cb(lv_event_t * e)
{
    lv_obj_t * btn = lv_event_get_target(e);
    lv_obj_t * label = lv_label_get(btn);
    lv_label_set_text(label, "LED OFF");
}

```

```

// LED OFF event handler
void btn_led_click_event_cb(lv_event_t * e)
{
    lv_obj_t * btn = lv_event_get_target(e);
    lv_obj_t * label = lv_label_get(btn);
    lv_label_set_text(label, "LED ON");
}

```

- Here are the event handlers triggered when the buttons are clicked, which turn the LED on or off with immediate response.
- Next is the main function `app_main`:
- Role: Serves as the program entry point, prints startup logs;
- Calls `system_init()` to complete all initializations;
- Enters a loop to keep the program running (LVGL's own tasks execute in the background).

```
111     printf("Hello World!\n");
112 }
113
114 int main()
115 {
116     // Register peripheral drivers (connecting I2C, SPI, UART, I2C and SPI hardware)
117     register_drivers();
118
119     // Register the main component with the ESP-IDF build system
120     ESP_REGISTER_COMPONENT(main, "main", "v1.0.0", "https://github.com/espressystems/esp-idf");
121
122     // Run the main component
123     ESP_RUN_COMPONENT(main);
124 }
125 }
```

- Finally, let's understand the "CMakeLists.txt" file in the main directory.
- The role of this CMake configuration is:
  - Collects all .c source files in the main/ directory as the component's source files;
  - Registers the "main" component with the ESP-IDF build system and declares that it depends on "bsp\_extra", "bsp\_display", "bsp\_illuminate", "bsp\_i2c", and "esp\_timer".
  - This way, during the build process, ESP-IDF knows to build these five components first, then build the "main" component.)

```
1 # Example CMakeLists.txt for a component
2
3 # The name of the component
4 component_register(NAME main)
5
6 # The component's source files
7 add_library(main SRCS main.c)
8
9 # The component's dependencies
10 component_dependencies(main bsp_extra bsp_display bsp_illuminate bsp_i2c esp_timer)
```

- Meanwhile, the header file references in main.h are also kept in sync.

```
1 #ifndef MAIN_H
2 #define MAIN_H
3
4 #include <stdio.h>
5 #include <string.h>
6 #include <esp_err.h>
7 #include <esp_log.h>
8 #include <esp_system.h>
9 #include <driver/i2c.h>
10 #include <driver/spi.h>
11 #include <driver/uart.h>
12 #include <esp_timer.h>
13 #include <bsp_extra.h>
14 #include <bsp_display.h>
15 #include <bsp_illuminate.h>
16 #include <bsp_i2c.h>
17
18 #endif
```

Note: In subsequent courses, we will not create a new "CMakeLists.txt" file from scratch. Instead, we will make minor modifications to this existing file to integrate other drivers into the main function.

## Complete Code

---

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson09-LVGL\\_Lighting\\_Control](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson09-LVGL_Lighting_Control)

## Programming Steps

---

- Now the code is ready. Next, we need to flash it to the ESP32-P4 to see the actual effect.
- First, connect the Advance-P4 device to our computer via a USB cable.



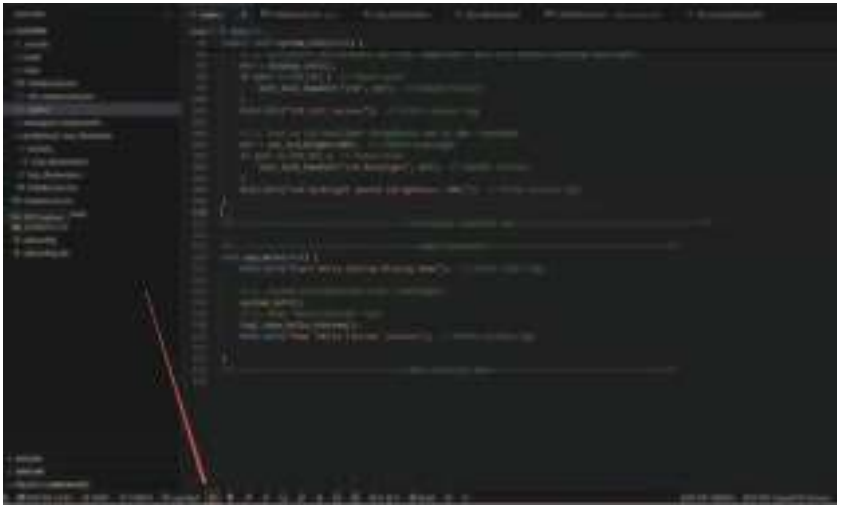
- Also, remember to connect an LED to the UART1 interface.



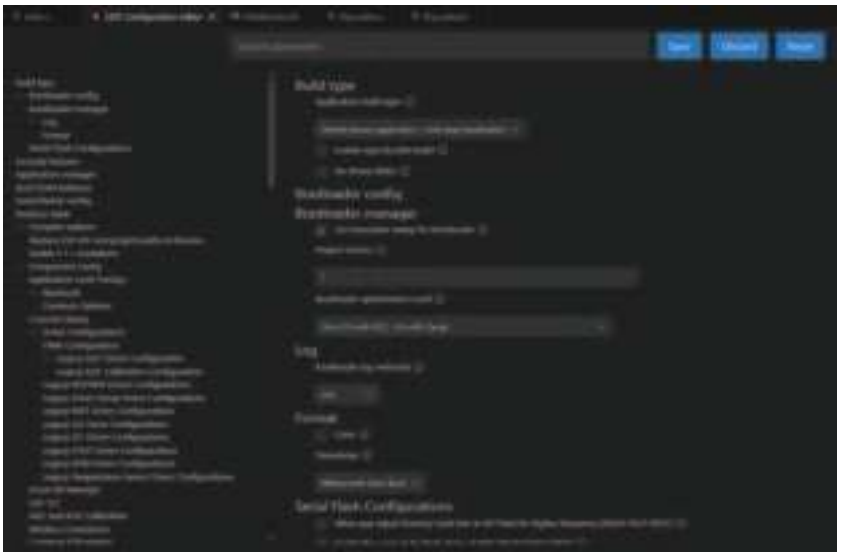
- Before starting the flashing preparation, delete all compiled files to restore the project to its initial "unbuilt" state. (This ensures that subsequent compilations are not affected by your previous operations.)



- Here, follow the steps from the first lesson to first select the ESP-IDF version, code upload method, serial port number, and the target chip (ESP32-P4).
- Next, we need to configure the SDK.
- Click on the icon shown in the figure below.



- Wait for a moment while the configuration loads, and then you can proceed with the relevant SDK configuration.



- Then, search for "flash" in the search box. (Make sure your flash configuration matches mine.)



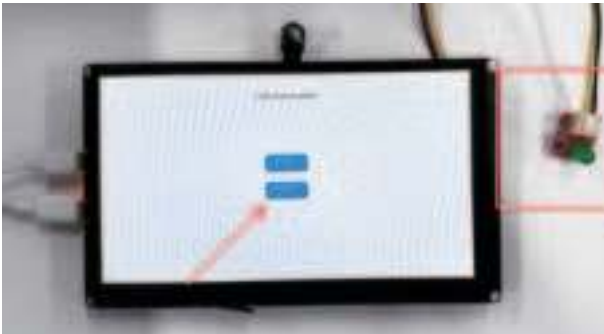
- After the configuration is completed, remember to save your settings.
- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we also want to share a very convenient feature: there is a single button that can execute compilation, uploading, and opening the monitor in one go. (This works on the premise that the entire code is error-free.)



- After waiting for a while, the code compilation and upload will be completed, and the monitor will open automatically.
- At this point, please remember to use an additional Type-C cable to connect your Advance-P4 via the USB 2.0 interface. This is because the maximum current provided by a computer's USB-A interface 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.)



- 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.



# Lesson 10

## Temperature and Humidity

### Introduction

---

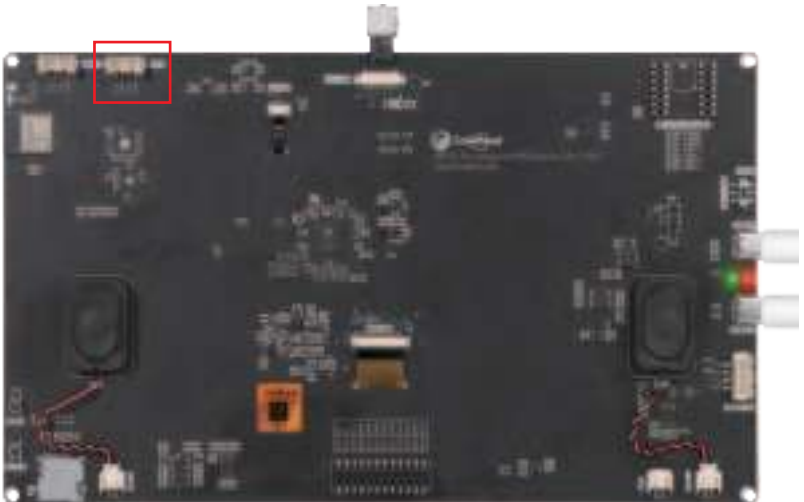
In this lesson, we will teach you how to use the I2C interface on the Advance-P4 board. We will connect a temperature and humidity sensor to the I2C interface, then display the values obtained from the sensor on the screen.

The key learning focus of this lesson is the use of the I2C interface. We will reuse the I2C component and screen display component covered in previous lessons, and additionally introduce a new temperature and humidity component: `bsp_dht20`.

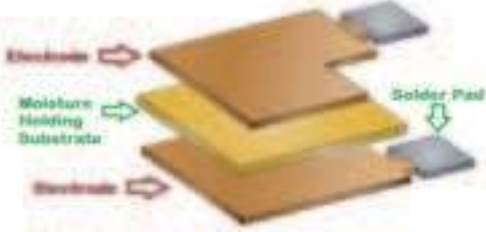
### Hardware Used in This Lesson

---

#### I2C Interface on the Advance-P4



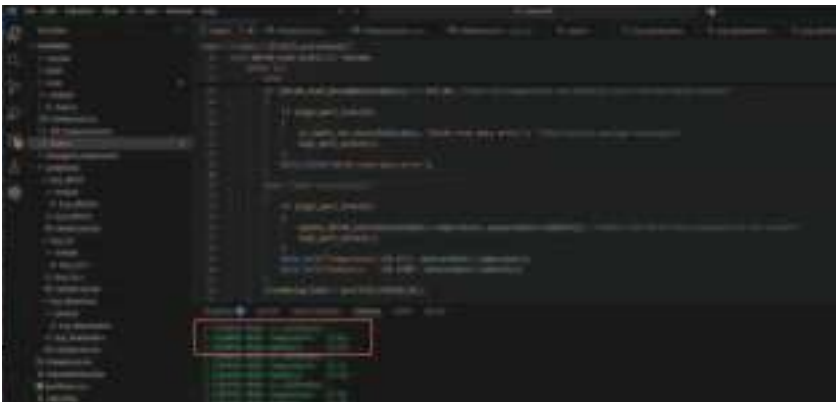
## Temperature and humidity sensor Schematic Diagram



- 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.

## Operation Effect Diagram

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.





## Key Explanations

---

- The focus of this lesson is on using the temperature and humidity sensor connected via the I2C interface. Here, we will prepare another new component for you: `bsp_dht20`. The main function of this component is to communicate with the DHT20 temperature and humidity sensor through the I2C bus, implementing functions such as sensor initialization, status detection, data reading, and verification to obtain environmental temperature and humidity data. You just need to know when to call the interfaces we have written in it.
- Next, let's focus on understanding the `bsp_dht20` component. (The `bsp_i2c` component and `bsp_dht20` component were explained in detail in previous courses.)
- First, click on the GitHub link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson10-Temperature\\_and\\_Humidity](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson10-Temperature_and_Humidity)

- Then drag the code for this lesson into VS Code and open the project file.
- Once opened, you can see the framework of this project.



In the example for this lesson, a new folder named `bsp_dht20` is created under the `peripheral\` directory. Within the `bsp_dht20\` folder, a new include folder and a "CMakeLists.txt" file are created.

The `bsp_dht20` folder contains the "bsp\_dht20.c" driver file, and the include folder contains the "bsp\_dht20.h" header file.

The "CMakeLists.txt" file integrates the driver into the build system, enabling the project to utilize the temperature and humidity acquisition functions predefined in "bsp\_dht20.c".

## 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.

```
1 //
2 #include <string.h> //String handling functions
3 #include <unistd.h> //Definitions for system-level I/O functions
4 #include <stdio.h> //Standard C I/O functions
5 #include <math.h> //Math functions
6 #include <sys/types.h> //System-level I/O functions
7 #include <sys/stat.h> //System-level I/O functions
8 #include <sys/time.h> //System-level I/O functions
9 #include <sys/timeb.h> //System-level I/O functions
10 #include <sys/types.h> //System-level I/O functions
11 #include <sys/stat.h> //System-level I/O functions
12 #include <sys/time.h> //System-level I/O functions
13 #include <sys/timeb.h> //System-level I/O functions
14 //-----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".)

```
1 //-----header file declaration start-----
2 #ifndef BSP_DHT20_H
3 #define BSP_DHT20_H
4
5 #include <stdint.h>
6
7 //-----function declarations-----
8 void bsp_dht20_init(void);
9 int bsp_dht20_read(void);
10 void bsp_dht20_write(uint8_t *data, uint8_t len);
11
12 #endif
```

- 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.
- If you need to call these interfaces, you must also configure the "CMakeLists.txt" file located in the `bsp_dht20` folder.
- This file, placed under the `bsp_dht20` folder, mainly functions to tell the ESP-IDF build system (CMake): how to compile and register the `bsp_dht20` component.



- The reason we include `bsp_i2c` and `esp_timer` here is that they are explicitly used in "bsp\_dht20.h". (Other system libraries do not need to be added because they are already integrated into the ESP-IDF framework by default.)



## Main function

- The main folder is the core directory for program execution, containing the main function executable file main.c.
- Add the main folder to the "CMakeLists.txt" file of the build system.



- This is the entry file of the entire application. In ESP-IDF, there is no `int main()`; instead, the program starts running from `void app_main(void)`.
- Let's first explain main.c:
  - First, the `Init` function is called to initialize the following components in sequence: LDO power supply (to provide power for peripherals), I2C bus (the foundation for sensor communication), DHT20 sensor (to complete registration and status calibration), and display module. If initialization fails, an error is reported in a loop through `init_fail`.
  - After successful initialization, set the screen backlight to 100%, call `dht20_display` to create an LVGL white text label (with a black background, initially displaying default temperature and humidity), then create the `read_dht20` task. This task cyclically checks the DHT20 calibration status every second (re-initializes if not calibrated), reads sensor data. If it fails, an error message is displayed on the screen; if successful, `update_dht20_value` is used to format and update the LVGL label to display real-time temperature and humidity.
- First is the reference to `main.h`, where we store the header files used and macro definitions.





```
TaskHandle_t read_dht20;
lv_obj_t *dht20_data = NULL;
lv_obj_t *temp_label;
lv_obj_t *humid_label;
lv_obj_t *temp_obj;
lv_obj_t *humid_obj;
lv_obj_t *temp_obj_label;
lv_obj_t *humid_obj_label;
lv_obj_t *temp_obj_label;
lv_obj_t *humid_obj_label;
lv_obj_t *temp_obj_label;
lv_obj_t *humid_obj_label;
```

- TaskHandle\_t read\_dht20;: Declares a FreeRTOS task handle, which is used to manage the lifecycle operations (such as creation and suspension) of the DHT20 data reading task.
- static lv\_obj\_t \*dht20\_data = NULL;: Declares a LVGL text label pointer (visible only within the current file), initially set to NULL. It is used to point to and manipulate the on-screen label that displays temperature and humidity data.
- The following two lines of code define the control handles for LDO channels 3 and 4. They are used to bind to actual LDO power channels during subsequent initialization, enabling the program to control power output at different voltages.

```
7  *lv_obj_t *lv_obj_create(lv_obj_t *parent, const lv_area_t *area, lv_obj_t *main_obj);  
8  *lv_obj_t *lv_obj_create(lv_obj_t *parent, const lv_area_t *area, lv_obj_t *main_obj);  
9  *lv_obj_t *lv_obj_create(lv_obj_t *parent, const lv_area_t *area, lv_obj_t *main_obj);  
10 *lv_obj_t *lv_obj_create(lv_obj_t *parent, const lv_area_t *area, lv_obj_t *main_obj);
```

### dht20\_display():

This function is used to initialize the text label for displaying temperature and humidity data in the LVGL graphical interface:

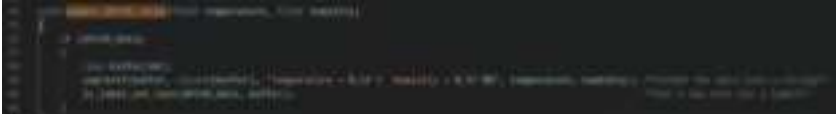
First, it acquires the LVGL operation lock via lvgl\_port\_lock(0) (to avoid multi-task conflicts). Then, it creates a text label object dht20\_data at the center of the screen, configures the label style (transparent background, white 30-point font), sets the screen to a black opaque background, and assigns the initial text "Temperature = 0.0 C Humidity = 0.0 %" to the label. Finally, it releases the LVGL lock, establishing a visual carrier for displaying real-time temperature and humidity data later.



### **void update\_dht20\_value(float temperature, float humidity):**

This function is used to update the display content of temperature and humidity data on the LVGL interface:

First, it checks whether the temperature and humidity display label `dht20_data` is valid. If valid, it uses `snprintf` to format the incoming temperature (`temperature`) and humidity (`humidity`) values into a string in the format of "Temperature = X.X C Humidity = X.X %". Then, it calls the LVGL interface `lv_label_set_text` to update the formatted string to the label, realizing real-time refresh of data on the screen.



### **void dht20\_read\_task(void \*param):**

This function is a FreeRTOS task function that executes periodically (every 1 second) in an infinite loop: It first checks if the DHT20 sensor is calibrated, and re-initializes it if not. If data reading fails, it displays an error message on the screen and prints a log. If reading succeeds, it updates the temperature and humidity data displayed on the screen and prints detailed logs, enabling continuous acquisition and visual display of sensor data.

### **void init\_fail(const char \*name, esp\_err\_t err):**

Function: When initialization of a module fails, this function is entered to run in an infinite loop, printing an error message (including the module name and error code string) once per second.



### **void Init(void):**

This function is used for system initialization, which configures the LDO3 (2.5V) and LDO4 (3.3V) power channels, initializes the I2C bus (with a 200ms delay for stabilization), initializes the DHT20 sensor and display module in sequence. If any step of initialization fails, it calls the 'init\_fail' function to output error messages in a loop, ensuring that subsequent operations are performed only after all hardware devices are ready.





Note: In the subsequent courses, we will not create a new "CMakeLists.txt" file from scratch. Instead, we will make minor modifications to this existing file to integrate other driver programs into the main function.

## Complete Code

---

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/lesson10-Temperature\\_and\\_Humidity](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/lesson10-Temperature_and_Humidity)

## Programming Steps

---

- Now the code is ready. Next, we need to flash it to the ESP32-P4 to see the actual effect.
- First, connect the Advance-P4 device to our computer via a USB cable.



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



- Before starting the preparation for flashing, first delete all files generated during compilation to restore the project to its initial "unbuilt" state. (This ensures that subsequent compilations are not affected by your previous build artifacts.)



- First, follow the steps in the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon shown in the figure below.



- Wait for a short loading period, and then you can proceed with the relevant SDK configuration.



- Then, search for flash in the search box. (Make sure your flash configuration is consistent with mine.)



- After the configuration is completed, remember to save your settings.

- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we also want to share a very convenient feature: there is a single button that can execute compilation, uploading, and opening the monitor in one go. (This works on the premise that the entire code is error-free.)



- Wait for a moment until the code compilation and upload are completed, and the monitor will open automatically.
- 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.



# Lesson 11

## Playback After Recording

### Introduction

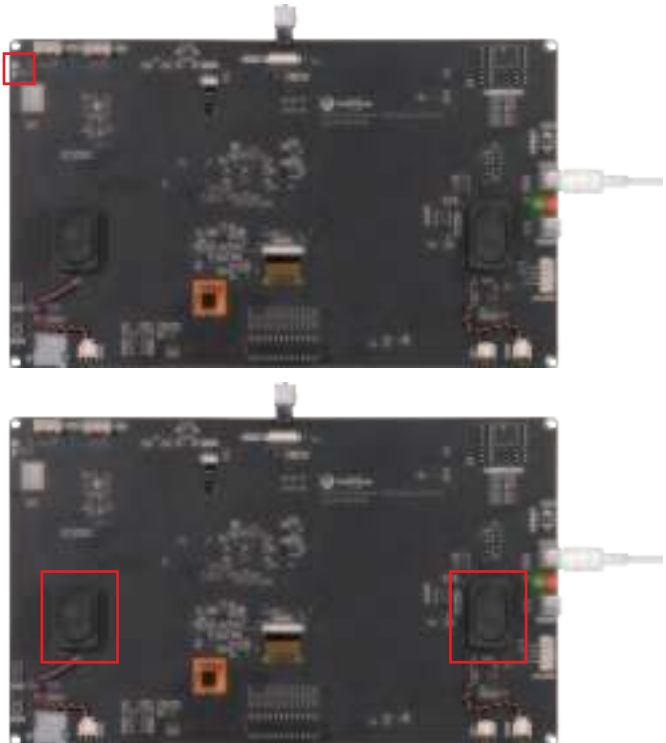
---

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.

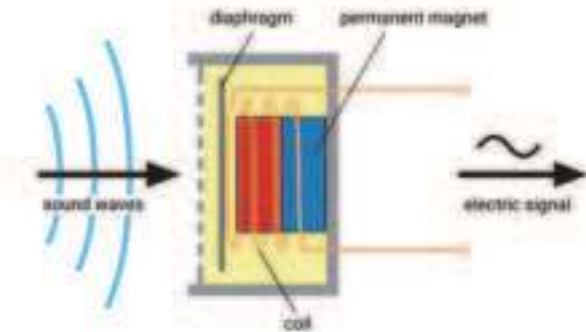
### Hardware Used in This Lesson

---

#### Microphone and Speaker on the Advance-P4



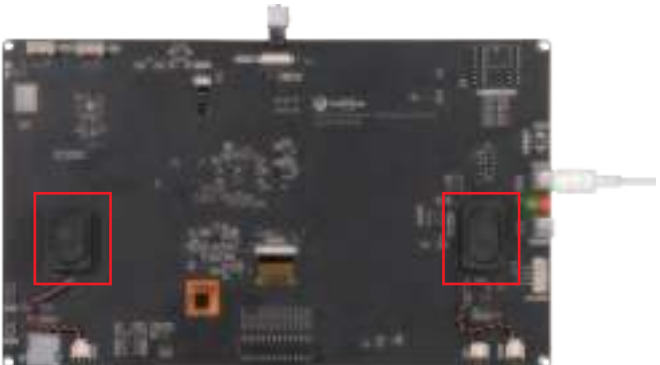
## Microphone and Speaker Schematic Diagrams



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.)

## Operation Effect Diagram

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



**The 5-second recorded audio is now playing.**

## Key Explanations

- The key focus of this lesson is the use of two components: `bsp_mic` and `bsp_audio`. Next, we will explain the functions of the definitions and functions in these components respectively. What you need to know is when to call the pre-written interfaces in them.
- Subsequently, we will focus on understanding these two components: `bsp_mic` and `bsp_audio`.
- First, click the GitHub link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson11-Playback\\_After\\_Recording](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson11-Playback_After_Recording)

- Then drag the code for this lesson into VS Code and open the project file.
- Once opened, you can see the framework of this project.



In the example of this lesson, new folders named "bsp\_mic" and "bsp\_audio" are created under "peripheral".

In the "bsp\_audio\" folder, a new "include" folder and a "CMakeLists.txt" file are created. (The same applies to "bsp\_mic".)

The "bsp\_audio" folder contains the "bsp\_audio.c" driver file, and the "include" folder contains the "bsp\_audio.h" header file. (The same applies to "bsp\_mic".)

The "CMakeLists.txt" file integrates the drivers into the build system, enabling the project to utilize the audio playback functions written in "bsp\_audio.c" and the audio recording functions written in "bsp\_mic.c".

### Code for "bsp\_audio"

Let's first look at the audio playback component, which includes two files: "bsp\_audio.c" and "bsp\_audio.h".

Next, we will first analyze the "bsp\_audio.h" program.

"bsp\_audio.h" is the header file for the audio playback module, mainly used to:



- It defines a global variable `tx_chan` with the type `i2s_chan_handle_t`, which is an I2S channel handle.
- This handle represents the audio output channel (TX), and all subsequent audio playback operations will be performed through this channel.

```

1 // tx_chan = I2S channel
2 tx_chan_handle_t tx_chan;
3 // tx_chan = I2S channel

```

- `audio_init`: This function is used to initialize and enable the I2S audio output channel. It configures parameters such as sample rate, bit width, clock, and pin settings, enabling the device to normally play audio data through the I2S interface.

```

11 // tx_chan = I2S channel
12 {
13     // tx_chan = I2S channel
14     i2s_chan_config_t chan_cfg = {
15         .clk = I2S_CLK_1,
16         .rate = I2S_RATE_48000,
17         .data_bits_per_sample = 16,
18         .dma_frame_size = 128,
19         .dma_channel = 0,
20         .dma_priority = 0,
21     };
22     // tx_chan = I2S channel
23     i2s_new_channel(&chan_cfg, &tx_chan, 0);
24     if (err != ESP_OK)
25         return err;
26     // tx_chan = I2S channel
27     i2s_port_config_t port_cfg = {
28         .clk_cfg = {
29             .sample_rate_hz = 48000,
30             .clk_src = I2S_CLK_SRC_DEFAULT,
31             .clock_source_div = I2S_CLK_SOURCE_DIV,
32         },
33         .slot_cfg = {
34             .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT,
35             .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO,
36             .slot_mode = I2S_SLOT_MODE_STEREO,
37             .slot_mask = I2S_SLOT_MASK_DEFAULT,
38             .slot_width = I2S_SLOT_WIDTH_DEFAULT,
39             .slot_pos = 0,
40             .slot_shift = 0,
41             .slot_align = 0,
42             .slot_offset = 0,
43             .slot_number_div = 1,
44         },
45         .gpio_cfg = {
46             .mclk = I2S_GPIO_MCLK,
47             .bclk = I2S_GPIO_BCLK,
48             .ws = I2S_GPIO_WS,
49             .data = I2S_GPIO_DATA,
50             .dss = I2S_GPIO_DSS,
51         },
52     };
53     // tx_chan = I2S channel
54     i2s_port_enable(&port_cfg, &tx_chan);
55     // tx_chan = I2S channel
56     i2s_channel_enable(&tx_chan);
57     // tx_chan = I2S channel
58     return ESP_OK;
59 }

```

- `audio_ctrl_init`: This function is used to initialize the audio power amplifier control pin, configuring it as an output mode to facilitate subsequent control of the power amplifier's on/off state.

```

00 esp_err_t audio_ctrl_init()
01 {
02     esp_err_t err = ESP_OK;
03     gpio_config_t audio_gpio_config = {
04         pin_bit_mask = (1ULL << GPIO_PIN_0),
05         mode = GPIO_MODE_OUTPUT,
06         pull_up_en = false,
07         pull_down_en = false,
08         intr_type = GPIO_INTR_DISABLE,
09     };
10     err = gpio_config(&audio_gpio_config);
11     if (err != ESP_OK)
12         return err;
13     return err;
14 }

```

- `set_audio_ctrl`: This function is used to control the on/off state of the audio power amplifier. It enables or disables the power amplifier by setting the level of the power amplifier control pin (active low).

```

00 esp_err_t set_audio_ctrl(bool state)
01 {
02     esp_err_t err = ESP_OK;
03     bool status = !state;
04     err = gpio_set_level(GPIO_PIN_0, status);
05     return err;
06 }

```

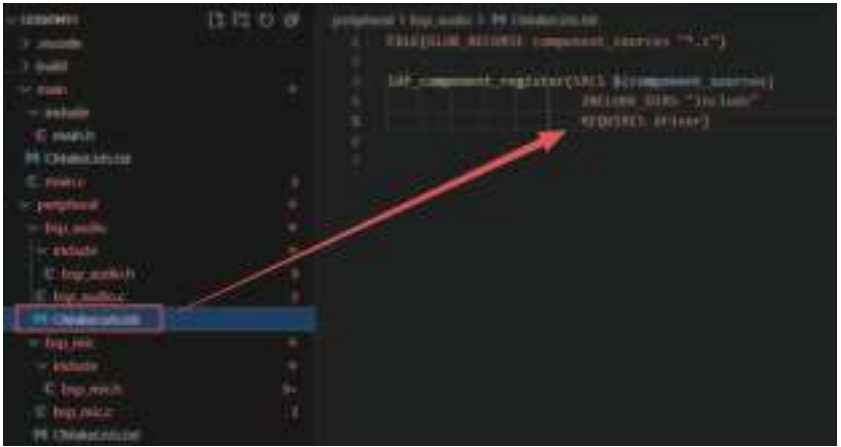
- `get_audio_handle`: This function is used to obtain and return the handle of the current I2S audio output channel, allowing other modules to use this handle for audio data transmission or playback operations.

```

00 i2s_chan_handle_t get_audio_handle()
01 {
02     return ix_chan;
03 }

```

- That concludes our introduction to the "bsp\_audio" component. What's important is that you know how to call these interfaces.
- If you need to use this component, you must also configure the "CMakeLists.txt" file under the "bsp\_audio" folder.
- This file, located in the "bsp\_audio" folder, mainly functions to tell the ESP-IDF build system (CMake): how to compile and register the "bsp\_audio" component.



- The reason why "driver" is included here is that we have called it in "bsp\_audio.h" (other libraries are system libraries and do not need to be added).



### Code for "bsp\_mic"

Let's now look at how audio recording is implemented. Here, we'll directly examine the composition of functions in "bsp\_mic.c".

First, let's look at "bsp\_mic.h".

```
10 #define MIC_GPIO_CLK GPIO_PIN_10
11 #define MIC_GPIO_SDIN2 GPIO_PIN_11
12
13 #define MIC_SAMPLE_RATE 16000
14 #define BYTE_RATE (MIC_SAMPLE_RATE * 2)
15
16 #define MIC_I2S_CFG I2S_CFG_1
17 #define MIC_I2S_MODE I2S_MODE_1
18 #define MIC_I2S_DMA_CHANNEL DMA_CHANNEL_1
19 #define MIC_I2S_DMA_STREAM DMA_STREAM_1
20 #define MIC_I2S_DMA_IRQ DMA_IRQ_1
21 #define MIC_I2S_DMA_PRIORITY DMA_PRIORITY_1
22
23 #define MIC_I2S_CLK_DIV 2
24 #define MIC_I2S_FILTER I2S_FILTER_0
25 #define MIC_I2S_MONO I2S_MONO_1
26 #define MIC_I2S_MONO_MODE I2S_MONO_MODE_1
27 #define MIC_I2S_MONO_CHANNEL I2S_MONO_CHANNEL_1
28 #define MIC_I2S_MONO_STREAM I2S_MONO_STREAM_1
29 #define MIC_I2S_MONO_IRQ I2S_MONO_IRQ_1
30 #define MIC_I2S_MONO_PRIORITY I2S_MONO_PRIORITY_1
31 #define MIC_I2S_MONO_DMA_CHANNEL I2S_MONO_DMA_CHANNEL_1
32 #define MIC_I2S_MONO_DMA_STREAM I2S_MONO_DMA_STREAM_1
33 #define MIC_I2S_MONO_DMA_IRQ I2S_MONO_DMA_IRQ_1
34 #define MIC_I2S_MONO_DMA_PRIORITY I2S_MONO_DMA_PRIORITY_1
35
36 #define MIC_I2S_CFG_1 I2S_CFG_1
37 #define MIC_I2S_MODE_1 I2S_MODE_1
38 #define MIC_I2S_DMA_CHANNEL_1 DMA_CHANNEL_1
39 #define MIC_I2S_DMA_STREAM_1 DMA_STREAM_1
40 #define MIC_I2S_DMA_IRQ_1 DMA_IRQ_1
41 #define MIC_I2S_DMA_PRIORITY_1 DMA_PRIORITY_1
42
43 #define MIC_I2S_CLK_DIV_1 2
44 #define MIC_I2S_FILTER_1 I2S_FILTER_0
45 #define MIC_I2S_MONO_1 I2S_MONO_1
46 #define MIC_I2S_MONO_MODE_1 I2S_MONO_MODE_1
47 #define MIC_I2S_MONO_CHANNEL_1 I2S_MONO_CHANNEL_1
48 #define MIC_I2S_MONO_STREAM_1 I2S_MONO_STREAM_1
49 #define MIC_I2S_MONO_IRQ_1 I2S_MONO_IRQ_1
50 #define MIC_I2S_MONO_PRIORITY_1 I2S_MONO_PRIORITY_1
51 #define MIC_I2S_MONO_DMA_CHANNEL_1 I2S_MONO_DMA_CHANNEL_1
52 #define MIC_I2S_MONO_DMA_STREAM_1 I2S_MONO_DMA_STREAM_1
53 #define MIC_I2S_MONO_DMA_IRQ_1 I2S_MONO_DMA_IRQ_1
54 #define MIC_I2S_MONO_DMA_PRIORITY_1 I2S_MONO_DMA_PRIORITY_1
```

- GPIO pins: MIC\_GPIO\_CLK (clock) and MIC\_GPIO\_SDIN2 (data input) specify the physical pins through which the microphone connects to the MCU. Audio sampling parameters: MIC\_SAMPLE\_RATE defines the sampling rate as 16 kHz, and BYTE\_RATE calculates the amount of audio data generated per second (32,000 bytes), which is used for subsequent audio processing and storage management.

```
10 #define MIC_GPIO_CLK GPIO_PIN_10
11 #define MIC_GPIO_SDIN2 GPIO_PIN_11
12
13 #define MIC_SAMPLE_RATE 16000
14 #define BYTE_RATE (MIC_SAMPLE_RATE * 2)
15
16 #define MIC_I2S_CFG I2S_CFG_1
17 #define MIC_I2S_MODE I2S_MODE_1
18 #define MIC_I2S_DMA_CHANNEL DMA_CHANNEL_1
19 #define MIC_I2S_DMA_STREAM DMA_STREAM_1
20 #define MIC_I2S_DMA_IRQ DMA_IRQ_1
21 #define MIC_I2S_DMA_PRIORITY DMA_PRIORITY_1
22
23 #define MIC_I2S_CLK_DIV 2
24 #define MIC_I2S_FILTER I2S_FILTER_0
25 #define MIC_I2S_MONO I2S_MONO_1
26 #define MIC_I2S_MONO_MODE I2S_MONO_MODE_1
27 #define MIC_I2S_MONO_CHANNEL I2S_MONO_CHANNEL_1
28 #define MIC_I2S_MONO_STREAM I2S_MONO_STREAM_1
29 #define MIC_I2S_MONO_IRQ I2S_MONO_IRQ_1
30 #define MIC_I2S_MONO_PRIORITY I2S_MONO_PRIORITY_1
31 #define MIC_I2S_MONO_DMA_CHANNEL I2S_MONO_DMA_CHANNEL_1
32 #define MIC_I2S_MONO_DMA_STREAM I2S_MONO_DMA_STREAM_1
33 #define MIC_I2S_MONO_DMA_IRQ I2S_MONO_DMA_IRQ_1
34 #define MIC_I2S_MONO_DMA_PRIORITY I2S_MONO_DMA_PRIORITY_1
```

- We'll stop here with the macro definitions in "bsp\_mic.h" for now. During usage, there's no need to modify these - keep the pins unchanged and maintain the microphone's sampling rate. Next, let's look at "bsp\_mic.c".
- Two functions are implemented here to enable microphone recording and playback through audio output, using I2S PDM mode.



## mic\_read\_to\_audio:

This function is used to record audio data from the microphone for a specified number of seconds and play it back in real time. Here's its detailed workflow:

First, it checks if the recording duration exceeds 60 seconds and calculates the required buffer size. Then, it dynamically allocates `read_buf` in SPI RAM to store the original mono audio data received from the I2S interface, and `write_buf` to store the processed stereo data for playback.

The function uses `i2s_channel_read` to block and read microphone data. For each audio sample, it performs volume amplification (multiplied by 10) and clipping processing to prevent overflow. It then copies the mono data to both left and right channels to form stereo data.

Subsequently, it turns on the power amplifier (`set_Audio_ctrl(true)`) and plays the processed audio through the audio output I2S channel. After playback is complete, it turns off the power amplifier and releases the buffer memory, ensuring the entire recording and playback process is safe and reliable.

(Please refer to the provided code for detailed implementation.)

```
40 mic_read_to_audio(read_buf, write_buf, sec, channel)
41
42 {
43     int read_buf_size = 0;
44     int write_buf_size = 0;
45     if (sec > 60)
46     {
47         mic_print("Recording time exceeds 60s!");
48         return ESP_FAIL;
49     }
50     read_buf_size = sec * 16000 * 1;
51     write_buf_size = sec * 16000 * 2;
52     read_buf = (uint8_t *) malloc(read_buf_size);
53     write_buf = (uint8_t *) malloc(write_buf_size);
54     if (read_buf == NULL || write_buf == NULL)
55     {
56         mic_print("malloc failed to apply!");
57         return ESP_FAIL;
58     }
59     memset(read_buf, 0, read_buf_size);
60     memset(write_buf, 0, write_buf_size);
61     if (read_buf == NULL || write_buf == NULL)
62     {
63         mic_print("malloc failed to apply!");
64         return ESP_FAIL;
65     }
66     memset(write_buf, 0, write_buf_size);
67     mic_print("Start recording for %d seconds!", sec);
68     int i = 0;
69     while (i < read_buf_size)
70     {
71         mic_print("read mic data fail!");
72         return 0;
73     }
74     if (write_buf_size != 0)
75     {
76         mic_print("read mic data non-zero!");
77         return 0;
78     }
79     return 0;
80 }
```

- Here, the `set_Audio_ctrl` function from `"bsp_audio.c"` is called to turn on the power amplifier pin, enabling sound playback.

```

100 // ...
101 int main()
102 {
103     // ...
104     set_Audio_ctrl(1);
105     // ...
106 }
  
```

```

100 // ...
101 int main()
102 {
103     // ...
104     set_Audio_ctrl(1);
105     // ...
106 }
  
```

```

Audio playback is enabled
  
```

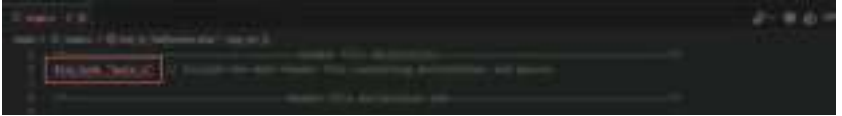
## Main function

- The main folder is the core directory for program execution, which contains the main function executable file `"main.c"`.
- Add the main folder to the `"CMakeLists.txt"` file of the build system.

```

add_subdirectory(main)
  
```

- This is the entry file of the entire application. In ESP-IDF, there is no `int main()`, and the program starts running from `void app_main(void)`.
- Let's first explain "main.c".
- The `app_main` function is the main entry point of the entire application, responsible for coordinating the initialization of the audio system and microphone, as well as handling recording and playback.
- First, there is the reference to "main.h". We store the header files used and macro definitions in "main.h".



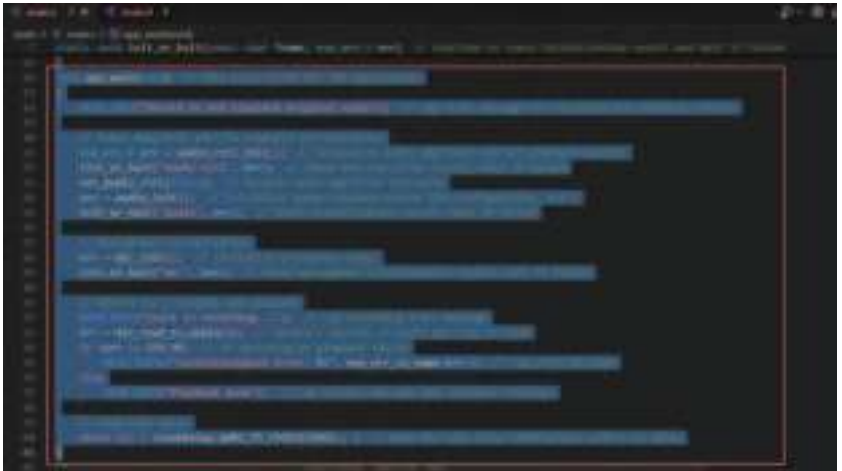
- Include C standard libraries and string manipulation libraries to provide basic functions.
- Include FreeRTOS task and scheduling interfaces for task creation and delay functions.
- Include ESP-IDF logging and error handling interfaces (`esp_log.h`, `esp_err.h`).
- Include header files of the microphone and audio modules to access interfaces such as `mic_init()`, `mic_read_to_audio()`, and `audio_init()`.



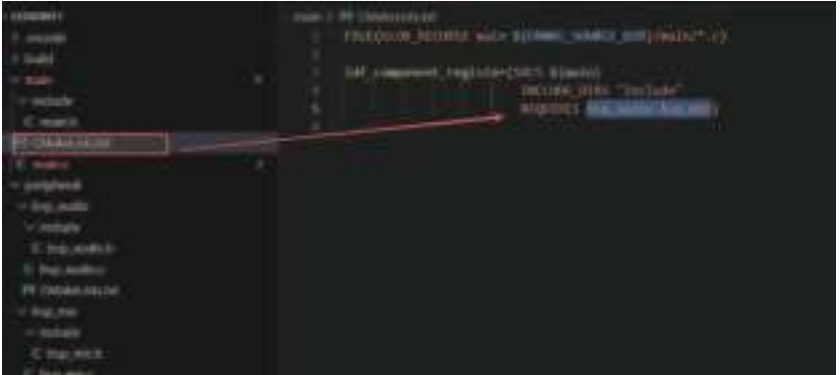
- The function "init\_or\_halt" is designed to uniformly check the return status of each module's initialization. It ensures the system does not continue running when the initialization of critical hardware or peripherals fails, thereby preventing undefined behavior or hardware damage.
- Specifically, it accepts two parameters: the module name "name" and the initialization result "err". If "err" is not equal to "ESP\_OK", it indicates a failed initialization. In this case, the function will print a detailed error log (including the module name and error information) via "MAIN\_ERROR", then enter an infinite loop with a 1-second delay in each loop iteration to prevent the program from proceeding further.



- Next is the main function "app\_main".
- The "app\_main" function serves as the primary entry point of the entire application, responsible for coordinating the initialization of the audio system and microphone, as well as audio recording and playback.
- It first initializes the audio power amplifier and the I2S playback channel, and uses "init\_or\_halt" to check if the initialization is successful. If the initialization fails, the program will get stuck in an infinite loop. Subsequently, it initializes the microphone input channel and also verifies the success of this initialization. After that, the program will record audio for 5 seconds and play it back via I2S. During this process, it prints log information to indicate the recording and playback status, and records error messages when errors occur.
- Finally, the function enters an infinite loop to keep the task alive, ensuring that the main program does not exit and thus maintaining the operating environment of the audio system. On the whole, this function implements a complete sample workflow for audio recording and playback.



- Finally, let's take a look at the "CMakeLists.txt" file in the main directory.
- The role of this CMake configuration is as follows:
  - Collect all .c source files in the main/ directory and use them as the source files of the component;
  - Register the main component to the ESP-IDF build system, and declare that it depends on the custom components "bsp\_audio" and "bsp\_mic".
- In this way, during the build process, ESP-IDF will know to build these two components first, and then build the main component.



Note: In the subsequent courses, we will not create a new "CMakeLists.txt" file from scratch. Instead, we will make minor modifications to this existing file to integrate other driver programs into the main function.

## Complete Code

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson11-Playback\\_After\\_Recording](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson11-Playback_After_Recording)

## Programming Steps

---

- Now that the code is ready, next, we need to flash it to the ESP32-P4 to observe the actual behavior.
- First, connect the Advance-P4 device to your computer via a USB cable.



- Before starting the flashing preparation, delete all files generated during compilation to restore the project to its initial "unbuilt" state. (This ensures that subsequent compilations are not affected by your previous operations.)



- Here, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip first.
- Next, we need to configure the SDK.
- Click the icon shown in the figure below.



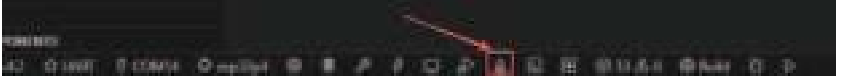
- After waiting for a short loading period, you can proceed with the relevant SDK configurations.



- Subsequently, search for "flash" in the search box. (Make sure your flash configuration is the same as mine.)



- After completing the configuration, remember to save your settings.
- Next, we will compile and flash the code (detailed steps were covered in the first lesson).
- Here, we also want to share a very convenient feature with you: there is a single button that allows you to execute compilation, upload, and monitor opening in one go. (This is on the premise that the entire code is confirmed to be error-free.)



- Wait for a moment, and the code will finish compiling and uploading, with the monitor opening automatically afterward.
- Once the flashing is successful, you can speak near the Advance-P4 device. The Advance-P4 will use its microphone to record the current sound within 5 seconds, and then play it back automatically.



**The 5-second recorded audio is now playing.**

# Lesson 12

## Playing Local Music from SD Card

### Introduction

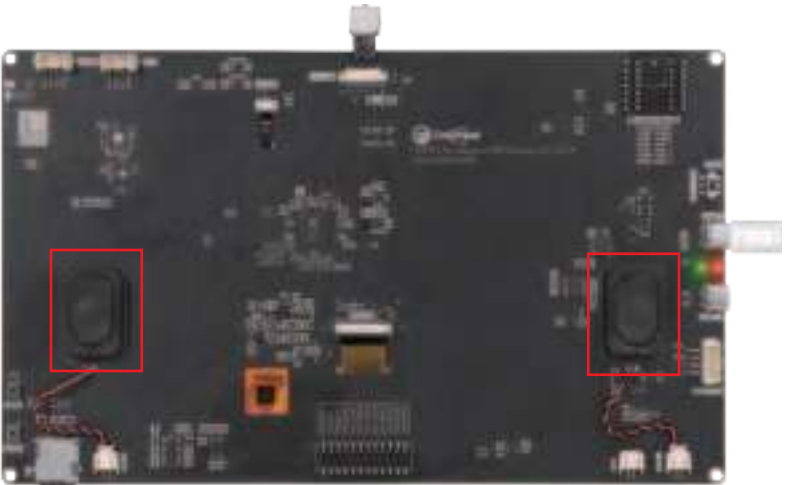
---

In this lesson, we will use the `bsp_sd` component and `bsp_audio` component (which were used in previous lessons) to play WAV audio files stored in the SD card.

### Hardware Used in This Lesson

---

#### Speaker on the Advance-P4



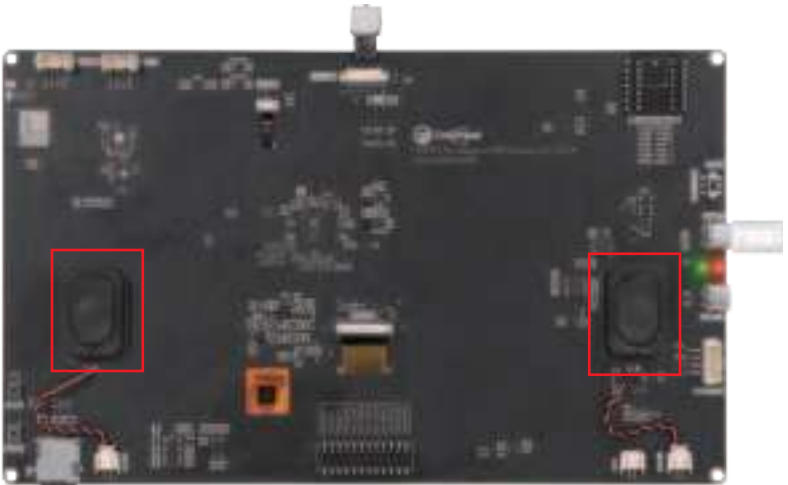
## SD Card on the Advance-P4



## Operation Effect Diagram

---

After running the code, you will be able to hear the WAV audio saved in your SD card playing through the speaker on the Advance-P4.



**The WAV audio file from your SD card is now playing.**

## Key Explanations

- The key focus of this lesson is the combined use of the two components: `bsp_sd` and `bsp_audio`. In fact, for the SD card component, we still use the same interfaces as in the previous component. These interfaces were explained in detail earlier, so they will not be covered again here.
- Next, we will focus on understanding the `bsp_audio` component. This component was used in the previous lesson to play the original sound after 5 seconds of recording. We already gained some knowledge about it back then, but only learned how to turn on the speaker. In this lesson, we will increase the difficulty slightly and learn how to play audio in WAV format.
- First, click on the GitHub link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson12-Playing\\_Loca\\_Music\\_from\\_SD\\_Card](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson12-Playing_Loca_Music_from_SD_Card)

- Then drag the code for this lesson into VS Code and open the project file.
- Once opened, you can see the framework of this project.



In the example for this lesson, new folders named `bsp_sd` and `bsp_audio` are created under the `peripheral\` directory.

Inside the `bsp_audio\` folder, a new include folder and a "CMakeLists.txt" file are created. (The same structure applies to `bsp_sd`.)

The `bsp_audio` folder contains the "bsp\_audio.c" driver file, and the include folder contains the "bsp\_audio.h" header file. (The same file structure applies to `bsp_sd`.)

The "CMakeLists.txt" file integrates the drivers into the build system. This allows the project to utilize the functions defined in "bsp\_audio.c" – including parsing WAV audio and playing WAV audio from the SD card – as well as the functions in "bsp\_sd.c" – such as initializing the SD card and retrieving SD card information.

### bsp\_audio Code

Let's first look at the audio playback component, which includes two files: "bsp\_audio.c" and "bsp\_audio.h".

- Next, we will first analyze the "bsp\_audio.h" program.
- "bsp\_audio.h" is the header file of the audio playback module, mainly used to:
  - Declare the functions, macros, and variables implemented in "bsp\_audio.c" for external programs, so that other .c files can call this module simply by #include "bsp\_audio.h".
  - In other words, it is an interface layer that announces which functions and constants are available to the outside while hiding the internal details of the module.
  - In this component, all the libraries we need to use are placed in the "bsp\_audio.h" file for unified management.

- Then, we declare the variables we need to use as well as the functions, whose specific implementations are in "bsp\_audio.c".
- Putting them uniformly in "bsp\_audio.h" is for the convenience of calling and management. (We will learn about their functions when they are used in "bsp\_audio.c".)

- Now let's look at the specific function of each function in "bsp\_audio.c".
- bsp\_audio.h: A custom audio module header file for this project, which defines macros, GPIO pins, and function declarations.

- A global variable tx\_chan is defined, with the type i2s\_chan\_handle\_t, i.e., an I2S channel handle.
- This handle represents the audio output channel (TX), and all subsequent audio playback operations will be performed through this channel.

```
1 // Initialize the I2S audio output channel of ESP32, enabling it to play
2 // audio in 16kHz, 16-bit, stereo format. It creates an I2S transmission channel, configures
3 // standard audio parameters (such as sampling rate, bit width, left/right channels, GPIO
4 // pins, etc.), and starts the channel to prepare for audio output.
```

## audio\_init()

This function is used to initialize the I2S audio output channel of ESP32, enabling it to play audio in 16kHz, 16-bit, stereo format. It creates an I2S transmission channel, configures standard audio parameters (such as sampling rate, bit width, left/right channels, GPIO pins, etc.), and starts the channel to prepare for audio output.

- `esp_err_t err = ESP_OK;` -- Initializes the error status variable, defaulting to successful operation.
- `i2s_chan_config_t chan_cfg = {...};` -- Configures I2S transmission channel parameters:
  - `id`: Uses I2S controller 1
  - `role`: Master mode (generates clock signals)
  - `dma_desc_num` and `dma_frame_num`: DMA buffer size settings
  - `auto_clear`: Automatically clears DMA buffer underflow
  - `intr_priority`: Interrupt priority
- `i2s_new_channel(&chan_cfg, &tx_chan, NULL);` -- Creates a new I2S transmission channel and saves it to `tx_chan`.
- `i2s_std_config_t std_cfg = {...};` -- Configures standard I2S audio parameters:
  - `clk_cfg`: Clock settings (sampling rate 16kHz, master clock multiplier 256)
  - `slot_cfg`: Audio data format (16-bit, stereo, left-aligned)
  - `gpio_cfg`: GPIO pins corresponding to I2S signals (BCLK, LRCLK, SDATA output) and whether to invert them
- `i2s_channel_init_std_mode(tx_chan, &std_cfg);` -- Initializes the I2S transmission channel in standard mode, making the channel comply with the above clock, data format, and GPIO configurations.
- `i2s_channel_enable(tx_chan);` -- Enables the I2S channel to start working and transmit audio data.
- `return err;` -- Returns the initialization status; if there is an error midway, an error code will be returned in advance.



## audio\_ctrl\_init:

This function is used to initialize the audio power amplifier control pin, configuring it as an output mode to control the on/off state of the power amplifier subsequently.

```
00 exp_err_t audio_ctrl_init()
01 {
02     exp_err_t err = ESP_OK;
03     gpio_config_t audio_gpio_config = {
04         .pin_bit_mask = (1ULL << AUDIO_POWER_CTRL),
05         .mode = GPIO_MODE_OUTPUT,
06         .pull_up_en = false,
07         .pull_down_en = false,
08         .io_type = GPIO_PIN_TYPE,
09     };
10     err = gpio_config(&audio_gpio_config);
11     if (err != ESP_OK)
12         return err;
13     return err;
14 }
```

## set\_Audio\_ctrl:

This function is used to control the on/off state of the audio power amplifier. It turns the power amplifier on or off by setting the level of the power amplifier control pin (active low).

```
00 exp_err_t set_audio_ctrl(bool state)
01 {
02     exp_err_t err = ESP_OK;
03     bool status = !state;
04     err = gpio_set_level(AUDIO_POWER_CTRL, status);
05     return err;
06 }
```

## validate\_wav\_header():

This function is used to check whether the header of an opened WAV file is valid, confirm if the file is in standard PCM WAV format, and verify that it supports common sampling rates, channel counts, and bit depths. After validation, the function restores the file pointer to its original position without altering the file reading state.

- if (file == NULL) — Checks if the file pointer is null; returns false if it is.
- long original\_position = ftell(file); — Obtains the current position of the file pointer for subsequent restoration.
- if (original\_position == -1) — Checks if the file position was obtained successfully.
- fseek(file, 0, SEEK\_SET) — Moves the file pointer to the beginning of the file.
- uint8\_t header[44]; size\_t bytes\_read = fread(header, 1, 44, file); — Reads the first 44 bytes of the WAV file (the standard WAV file header).

- `if (bytes_read != 44) ---` Checks if the WAV header was read completely.
- `memcmp(header, "RIFF", 4) ---` Verifies if the file starts with "RIFF" (the RIFF chunk identifier).
- `memcmp(header + 8, "WAVE", 4) ---` Checks if the format is "WAVE".
- `memcmp(header + 12, "fmt ", 4) ---` Verifies the existence of the fmt subchunk.
- `uint16_t audio_format = *(uint16_t*)(header + 20); ---` Retrieves the audio format field (1 indicates PCM).
- `uint16_t num_channels = *(uint16_t*)(header + 22); ---` Obtains the number of channels (supports 1 or 2 channels).
- `uint32_t sample_rate = *(uint32_t*)(header + 24); ---` Retrieves the sampling rate and verifies if it is a commonly used value.
- `uint16_t bits_per_sample = *(uint16_t*)(header + 34); ---` Obtains the number of bits per sample (supports 8/16/24/32 bits).
- `memcmp(header + 36, "data", 4) ---` Verifies if the data chunk identifier is "data".
- `uint32_t file_size = *(uint32_t*)(header + 4) + 8; uint32_t data_size = *(uint32_t*)(header + 40); ---` Retrieves the total file size and audio data size for printing information.
- `AUDIO_INFO(...)` --- Outputs WAV file information (number of channels, sampling rate, bit depth, data size, and file size).
- `fseek(file, original_position, SEEK_SET); ---` Restores the file pointer to its original position.
- `return true; ---` Returns true if validation passes.

The function's role is to check the validity of the WAV file header, ensuring the file is in standard PCM WAV format, supports common sampling rates, bit depths, and channel counts, and restores the file pointer position after validation.

- The first 44 bytes form the standard PCM WAV header, which describes information such as audio format, number of channels, and sampling rate.
- Before playing or processing a WAV file, it is usually necessary to read and validate this header to ensure the file format meets expectations.
- The `validate_wav_header()` function checks the validity of each field according to this structure.

## Audio\_play\_wav\_sd:

`Audio_play_wav_sd()` is used to read WAV files from the SD card and play audio through the I2S output of ESP32. It validates the WAV file header, skips the header, reads audio data in chunks, processes the volume (amplifies and limits the range), sends the data to the I2S player until the audio playback is completed, and then releases resources.

- `esp_err_t err = ESP_OK;` --- Initializes the error status variable.
- `if (filename == NULL)` --- Checks if the input filename is null; returns a parameter error if it is.
- `FILE *fh = fopen(filename, "rb");` --- Opens the WAV file in read-only binary mode.
- `if (fh == NULL)` --- Returns an error if the file fails to open.
- `if (!validate_wav_header(fh))` --- Calls the previously written WAV header validation function to check if the format is correct.
- `fseek(fh, 44, SEEK_SET)` --- Skips the WAV file header (44 bytes) to prepare for reading audio data.
- Define buffer sizes
  - `SAMPLES_PER_BUFFER = 512` --- Number of samples read each time
  - `INPUT_BUFFER_SIZE, OUTPUT_BUFFER_SIZE` --- Byte sizes of input and output buffers
- `heap_caps_malloc(...)` --- Allocates input and output buffers in SPI RAM; if allocation fails, releases the allocated resources and exits.
- Initializes variables for reading and writing: `samples_read`, `bytes_to_write`, `bytes_written`, `total_samples`, `volume_data`.
- `set_Audio_ctrl(true);` --- Turns on the audio hardware or amplifier.
- `while (1)` --- Loops to read audio data and play:
  - `samples_read = fread(...)` --- Reads audio samples from the file into the input buffer
  - `if (samples_read == 0) break;` --- Exits the loop when the file reading is completed
  - for loop --- Amplifies mono samples by 10 times, limits them to the `int16_t` range, and stores them in the output buffer (can be used for the left channel here, or extended to stereo)
  - `bytes_to_write = samples_read * sizeof(int16_t);` --- Calculates the number of bytes to be written to I2S

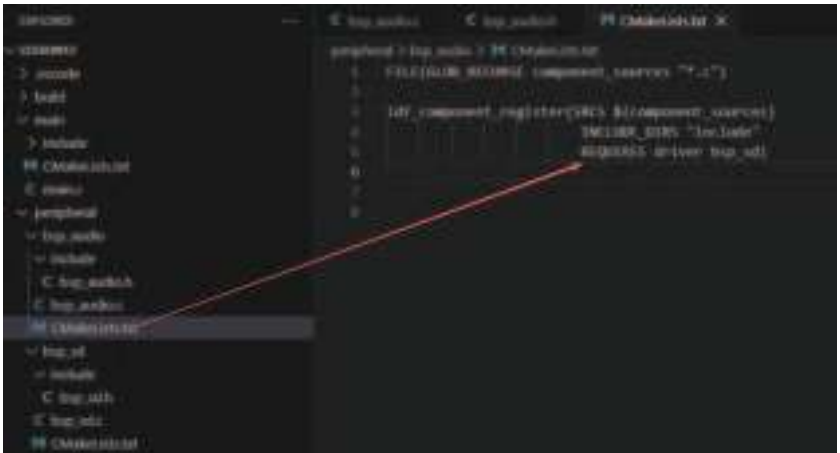
- `i2s_channel_write(tx_chan, output_buf, ...)` -- Writes audio data to the I2S output channel
- Error checking: Prints an error and exits the loop if writing fails
- Accumulates `total_samples` to count the total number of played samples
- Cleans up resources after the loop ends:
  - `set_Audio_ctrl(false);` -- Turns off the audio hardware
  - `free(input_buf); free(output_buf); fclose(fh);` -- Releases buffers and closes the file
- `AUDIO_INFO(...)` -- Prints playback completion information
- `return err;` -- Returns the playback result status

This function reads WAV files from the SD card, plays audio in chunks after validating the format, outputs to the audio hardware through I2S, handles volume and buffer management, and releases all resources after playback.

That's all for the introduction of the `bsp_audio` component. It's sufficient for you to know how to call these interfaces.

To call them, we must also configure the "CMakeLists.txt" under the `bsp_audio` folder.

This file, placed in the `bsp_audio` folder, mainly functions to tell the ESP-IDF build system (CMake) how to compile and register the `bsp_audio` component.



- The reason why "driver" and "bsp\_sd" are included here is that we have called them in "bsp\_audio.h" (other libraries are system libraries, so no need to add them).



- It uses interfaces from the SD component for SD card reading operations, among others.
- As for the bsp\_sd component, it was explained in detail in previous lessons, so it will not be repeated here. We will directly use this component.

### 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.

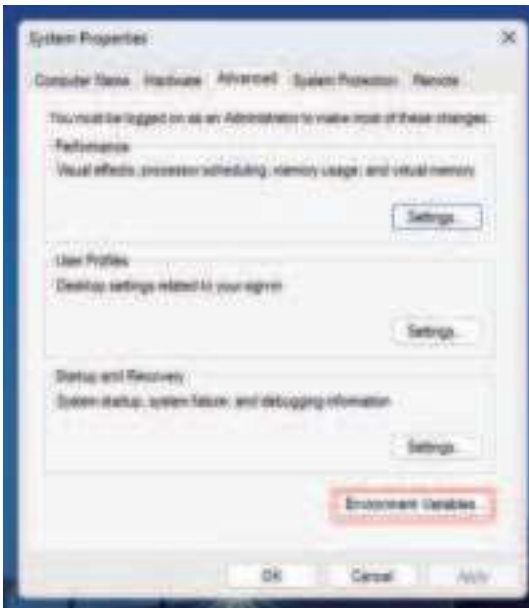
Navigate to the "bin" directory, and you will see three core executable files of FFmpeg: **"ffmpeg.exe"**, **"ffplay.exe"**, and **"ffprobe.exe"**.



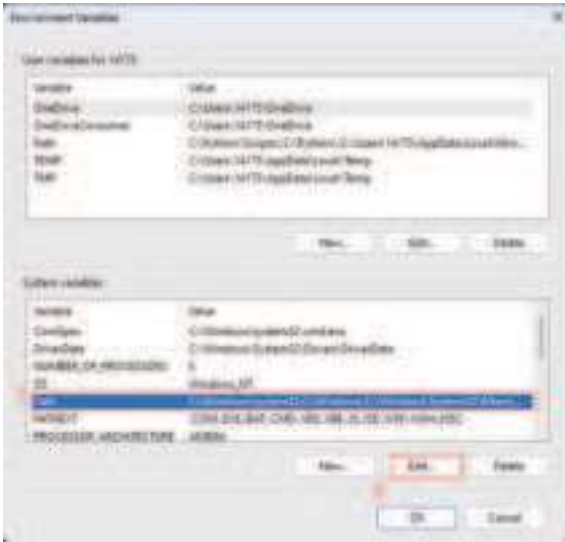
- To conveniently call FFmpeg directly in the command line, you need to add it to the system's environment variables.
- Search for "Environment Variables" in the Start Menu at the bottom left of the desktop, find "Edit the system environment variables", and click to open it.



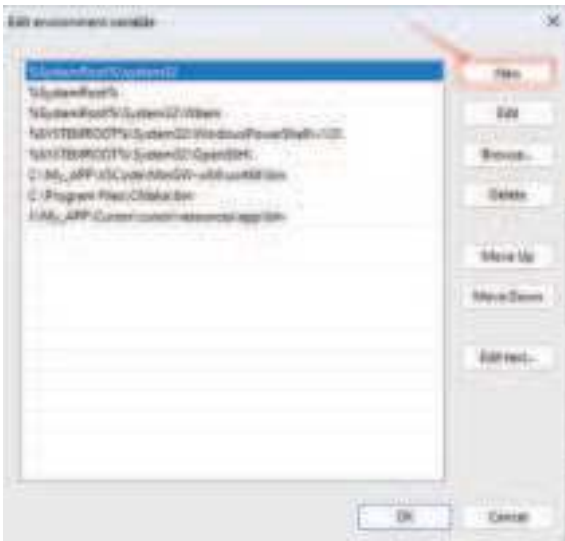
- Click the "Environment Variables" button.



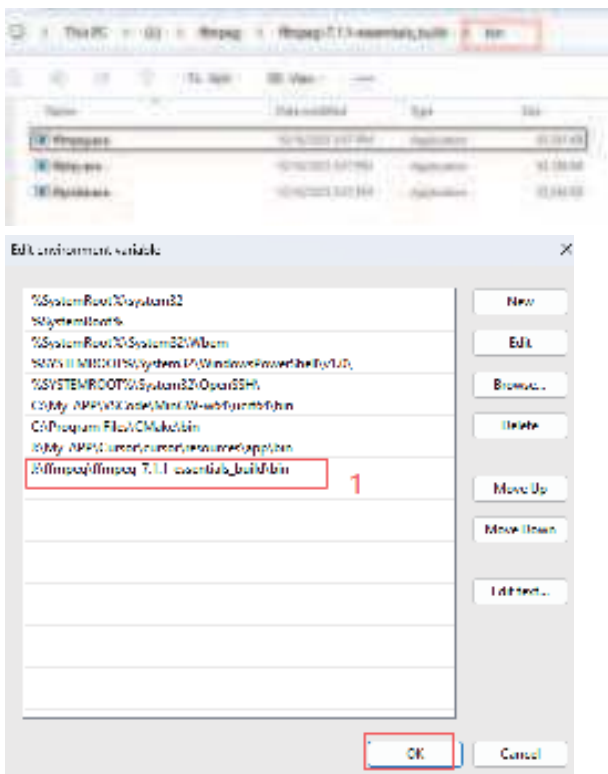
- Locate the "Path" entry under "System Variables" and click "Edit".



- 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).

```

Microsoft Windows [Version 10.0.22H2.19000]
(c) Microsoft Corporation. All rights reserved.

C:\Users\12179>ffmpeg -version
ffmpeg version 7.1.1 Copyright (c) 2000-2025 the FFmpeg developers
built with gcc 14.2.0 (x64) built by mingw-w64
configuration: --enable-shared --enable-static --enable-version3 --enable-zlib --enable-lzma --enable-gnutls --enable-libass --enable-libbluray --enable-libbrotli --enable-libdav1d --enable-libfreetype --enable-libfontconfig --enable-libfribidi --enable-libgsm --enable-libharfbuzz --enable-libjbig --enable-liblame --enable-liblibmtp --enable-libmodplug --enable-libmp3lame --enable-libopus --enable-librtmp --enable-librubiofm --enable-libshine --enable-libsnappy --enable-libsoxr --enable-libspeex --enable-libssh --enable-libsvt-av1 --enable-libtesseract --enable-libtheora --enable-libtwolame --enable-libvidstab --enable-libvorbis --enable-libvpx --enable-libwebp --enable-libx264 --enable-libx265 --enable-libxml2 --enable-libzimg --enable-libzmq --enable-libzstd --enable-libzvbi --enable-libzvonk --enable-openssl --enable-vaapi --enable-vulkan --enable-x11 --enable-xlib --enable-xv --enable-xvcc --enable-xvcc2 --enable-xvcc3 --enable-xvcc4 --enable-xvcc5 --enable-xvcc6 --enable-xvcc7 --enable-xvcc8 --enable-xvcc9 --enable-xvcc10 --enable-xvcc11 --enable-xvcc12 --enable-xvcc13 --enable-xvcc14 --enable-xvcc15 --enable-xvcc16 --enable-xvcc17 --enable-xvcc18 --enable-xvcc19 --enable-xvcc20 --enable-xvcc21 --enable-xvcc22 --enable-xvcc23 --enable-xvcc24 --enable-xvcc25 --enable-xvcc26 --enable-xvcc27 --enable-xvcc28 --enable-xvcc29 --enable-xvcc30 --enable-xvcc31 --enable-xvcc32 --enable-xvcc33 --enable-xvcc34 --enable-xvcc35 --enable-xvcc36 --enable-xvcc37 --enable-xvcc38 --enable-xvcc39 --enable-xvcc40 --enable-xvcc41 --enable-xvcc42 --enable-xvcc43 --enable-xvcc44 --enable-xvcc45 --enable-xvcc46 --enable-xvcc47 --enable-xvcc48 --enable-xvcc49 --enable-xvcc50 --enable-xvcc51 --enable-xvcc52 --enable-xvcc53 --enable-xvcc54 --enable-xvcc55 --enable-xvcc56 --enable-xvcc57 --enable-xvcc58 --enable-xvcc59 --enable-xvcc60 --enable-xvcc61 --enable-xvcc62 --enable-xvcc63 --enable-xvcc64 --enable-xvcc65 --enable-xvcc66 --enable-xvcc67 --enable-xvcc68 --enable-xvcc69 --enable-xvcc70 --enable-xvcc71 --enable-xvcc72 --enable-xvcc73 --enable-xvcc74 --enable-xvcc75 --enable-xvcc76 --enable-xvcc77 --enable-xvcc78 --enable-xvcc79 --enable-xvcc80 --enable-xvcc81 --enable-xvcc82 --enable-xvcc83 --enable-xvcc84 --enable-xvcc85 --enable-xvcc86 --enable-xvcc87 --enable-xvcc88 --enable-xvcc89 --enable-xvcc90 --enable-xvcc91 --enable-xvcc92 --enable-xvcc93 --enable-xvcc94 --enable-xvcc95 --enable-xvcc96 --enable-xvcc97 --enable-xvcc98 --enable-xvcc99 --enable-xvcc100
libavcodec      7.1.1      2025-07-18
libavformat     7.1.1      2025-07-18
libavutil       7.1.1      2025-07-18
libavfilter     7.1.1      2025-07-18
libavresample   0          2017-12-15
libpostproc    7.1.1      2025-07-18
libswresample   7.1.1      2025-07-18
libswscale      7.1.1      2025-07-18
libzimg         3.0.4      2024-07-29
libzmq          4.3.4      2020-07-14
libzstd         1.5.6      2024-07-29
libzvbi         0.11.0     2019-06-11
libzvonk        1.0.0      2024-07-29
openssl         3.0.15     2024-07-29
vaapi           2.19.0     2024-07-29
vulkan          1.3.280    2024-07-29
x11             1.0.0      2024-07-29
xlib            1.0.0      2024-07-29
xv              1.0.0      2024-07-29
xvcc            1.0.0      2024-07-29
xvcc2           1.0.0      2024-07-29
xvcc3           1.0.0      2024-07-29
xvcc4           1.0.0      2024-07-29
xvcc5           1.0.0      2024-07-29
xvcc6           1.0.0      2024-07-29
xvcc7           1.0.0      2024-07-29
xvcc8           1.0.0      2024-07-29
xvcc9           1.0.0      2024-07-29
xvcc10          1.0.0      2024-07-29
xvcc11          1.0.0      2024-07-29
xvcc12          1.0.0      2024-07-29
xvcc13          1.0.0      2024-07-29
xvcc14          1.0.0      2024-07-29
xvcc15          1.0.0      2024-07-29
xvcc16          1.0.0      2024-07-29
xvcc17          1.0.0      2024-07-29
xvcc18          1.0.0      2024-07-29
xvcc19          1.0.0      2024-07-29
xvcc20          1.0.0      2024-07-29
xvcc21          1.0.0      2024-07-29
xvcc22          1.0.0      2024-07-29
xvcc23          1.0.0      2024-07-29
xvcc24          1.0.0      2024-07-29
xvcc25          1.0.0      2024-07-29
xvcc26          1.0.0      2024-07-29
xvcc27          1.0.0      2024-07-29
xvcc28          1.0.0      2024-07-29
xvcc29          1.0.0      2024-07-29
xvcc30          1.0.0      2024-07-29
xvcc31          1.0.0      2024-07-29
xvcc32          1.0.0      2024-07-29
xvcc33          1.0.0      2024-07-29
xvcc34          1.0.0      2024-07-29
xvcc35          1.0.0      2024-07-29
xvcc36          1.0.0      2024-07-29
xvcc37          1.0.0      2024-07-29
xvcc38          1.0.0      2024-07-29
xvcc39          1.0.0      2024-07-29
xvcc40          1.0.0      2024-07-29
xvcc41          1.0.0      2024-07-29
xvcc42          1.0.0      2024-07-29
xvcc43          1.0.0      2024-07-29
xvcc44          1.0.0      2024-07-29
xvcc45          1.0.0      2024-07-29
xvcc46          1.0.0      2024-07-29
xvcc47          1.0.0      2024-07-29
xvcc48          1.0.0      2024-07-29
xvcc49          1.0.0      2024-07-29
xvcc50          1.0.0      2024-07-29
xvcc51          1.0.0      2024-07-29
xvcc52          1.0.0      2024-07-29
xvcc53          1.0.0      2024-07-29
xvcc54          1.0.0      2024-07-29
xvcc55          1.0.0      2024-07-29
xvcc56          1.0.0      2024-07-29
xvcc57          1.0.0      2024-07-29
xvcc58          1.0.0      2024-07-29
xvcc59          1.0.0      2024-07-29
xvcc60          1.0.0      2024-07-29
xvcc61          1.0.0      2024-07-29
xvcc62          1.0.0      2024-07-29
xvcc63          1.0.0      2024-07-29
xvcc64          1.0.0      2024-07-29
xvcc65          1.0.0      2024-07-29
xvcc66          1.0.0      2024-07-29
xvcc67          1.0.0      2024-07-29
xvcc68          1.0.0      2024-07-29
xvcc69          1.0.0      2024-07-29
xvcc70          1.0.0      2024-07-29
xvcc71          1.0.0      2024-07-29
xvcc72          1.0.0      2024-07-29
xvcc73          1.0.0      2024-07-29
xvcc74          1.0.0      2024-07-29
xvcc75          1.0.0      2024-07-29
xvcc76          1.0.0      2024-07-29
xvcc77          1.0.0      2024-07-29
xvcc78          1.0.0      2024-07-29
xvcc79          1.0.0      2024-07-29
xvcc80          1.0.0      2024-07-29
xvcc81          1.0.0      2024-07-29
xvcc82          1.0.0      2024-07-29
xvcc83          1.0.0      2024-07-29
xvcc84          1.0.0      2024-07-29
xvcc85          1.0.0      2024-07-29
xvcc86          1.0.0      2024-07-29
xvcc87          1.0.0      2024-07-29
xvcc88          1.0.0      2024-07-29
xvcc89          1.0.0      2024-07-29
xvcc90          1.0.0      2024-07-29
xvcc91          1.0.0      2024-07-29
xvcc92          1.0.0      2024-07-29
xvcc93          1.0.0      2024-07-29
xvcc94          1.0.0      2024-07-29
xvcc95          1.0.0      2024-07-29
xvcc96          1.0.0      2024-07-29
xvcc97          1.0.0      2024-07-29
xvcc98          1.0.0      2024-07-29
xvcc99          1.0.0      2024-07-29
xvcc100         1.0.0      2024-07-29
C:\Users\12179>

```

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

```

C:\Users\12179>pip install pydub
Collecting pydub
  Downloading pydub-0.25.1-py2.py3-none-any.whl (32.1 kB)
Installing collected packages: pydub
Successfully installed pydub-0.25.1
C:\Users\12179>

```

- 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-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/convert\\_wav](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/convert_wav)

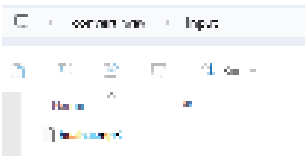
- Now I have placed this script on my desktop.



- In the command window, I navigate to this path.

```
C:\Users\14179>cd Desktop/
C:\Users\14179\Desktop>cd convert_wav
C:\Users\14179\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:\Users\14179\Desktop\convert_wav>python version
Python 3.11.2
```

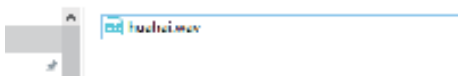
- Starting from Python 3.13: The official team removed the audioop module (which pydub depends on). Some third-party libraries (such as pyaudio, pygame, pydub) are not yet fully compatible.
- For Python 3.11.x:
  - ✓ Stable, mature, and highly compatible;
  - ✓ Includes audioop;
  - ✓ Perfectly compatible with most AI, audio, and data analysis libraries.

- Run our script:

```
C:\Users\user> cd Desktop\convert_wav && python ng3_to_wav.py
[16:03:00] user@ng3: C:\Users\user\Desktop\convert_wav\output> lsahai.wav (Conversion: Yes)
[16:03:00] Patch conversion completed. All files meet EGFSC I2S requirements.
```

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

```
convert_wav > Output
```



- Then move this file to a USB flash drive.

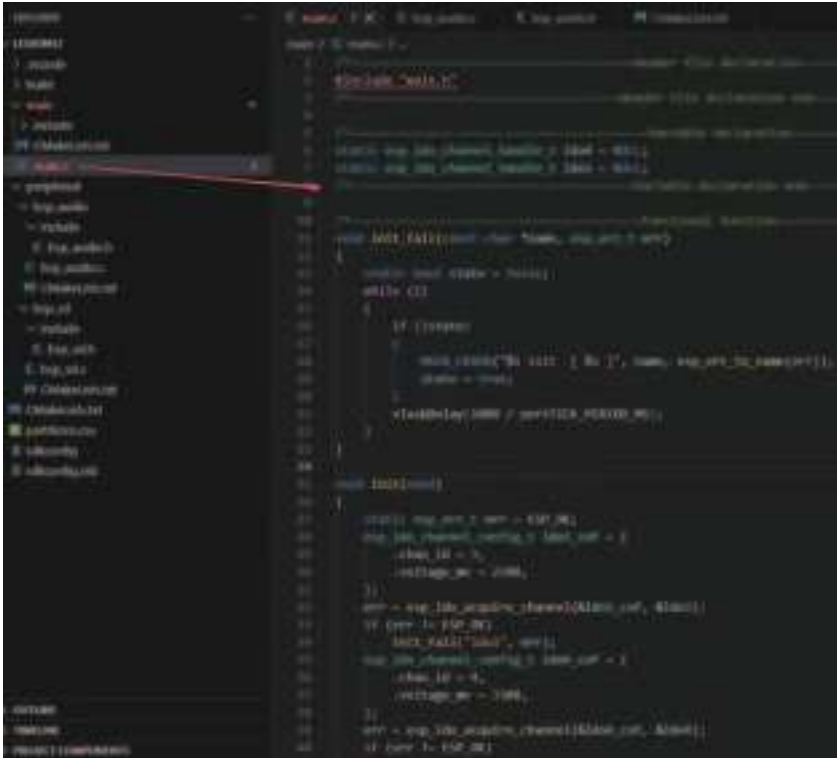


- Finally, remove the SD card and insert it into the **Advance-P4** board.



## Main function

- The main folder is the core directory for program execution, containing the main function executable file main.c.
- Add the main folder to the "CMakeLists.txt" file of the build system.



```
1 // 主函数
2
3 #include <stdio.h>
4 #include <string.h>
5 #include <conio.h>
6 #include <windows.h>
7
8 int main()
9 {
10     // 设置控制台窗口标题
11     SetConsoleTitle(TEXT("ESP-IDF Example"));
12     // 设置控制台窗口大小
13     SetConsoleWindowInfo(GetStdHandle(STD_OUTPUT_HANDLE), TRUE, &cbw);
14     // 设置控制台窗口位置
15     SetConsoleWindowInfo(GetStdHandle(STD_OUTPUT_HANDLE), TRUE, &cbw);
16     // 设置控制台窗口颜色
17     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
18     // 打印消息
19     printf("ESP-IDF Example\n");
20     // 等待用户输入
21     getch();
22     return 0;
23 }
```

- This is the entry file of the entire application. In ESP-IDF, there is no `int main()`, and execution starts from `void app_main(void)`.
- Let's first explain `main.c`.

## Init:

- The `Init()` function is used to initialize the hardware required for the audio playback system, including configuring and obtaining LDO3 (2.5V) and LDO4 (3.3V) channels, initializing the SD card for reading WAV files, initializing the audio controller and turning off the audio hardware, as well as initializing the I2S audio channel to prepare for WAV playback. If any step fails, it will call `init_fail()` to print an error and stop program execution.

```
main > C:\msys2\home\user> gcc main.c -o main.exe
10 void Init(void)
11 {
12     esp_err_t channel_config_3_lsd3_conf = {
13         esp_err_t acquire_channel(Bluetooth_conf, &lhd3);
14         if (err != ESP_OK)
15             Init_fail("lhd3", err);
16     esp_err_t channel_config_4_lsd4_conf = {
17         channel_ID = 4,
18         voltage_mv = 3300,
19     };
20 }
21 esp_err_t acquire_channel(Bluetooth_conf, &lhd4);
22 if (err != ESP_OK)
23     Init_fail("lhd4", err);
24
25 esp_err_t init_i2s(); //I2S initialization
26 if (err != ESP_OK)
27     Init_fail("i2s", err);
28 vTaskDelay(100 / portTICK_PERIOD_MS);
29
30 esp_err_t audio_ctrl_init(); //audio ctrl initialization
31 if (err != ESP_OK)
32     Init_fail("audio ctrl", err);
33
34 esp_audio_ctrl_t(i2s_conf);
35 esp_err_t audio_init(); //audio initialization
36 if (err != ESP_OK)
37     Init_fail("audio", err);
38 vTaskDelay(100 / portTICK_PERIOD_MS);
39
40 void app_main(void)
41 {
42     printf("Hello World!\n");
43     printf("ESP version: %s\n", ESP_GET_VERSION());
44     printf("Start the task.\n");
45     Init();
46     printf("Initialization complete!\n");
47     //Play the WAV file stored on the SD card
48     //functional function end
49 }
```

- After waiting for the SD card and other components to complete initialization, the next step is to execute `Audio_play_wav_sd` from the `bsp_audio` component to play the converted WAV audio files stored in the SD card.
- Finally, let's look at the "CMakeLists.txt" file in the main directory.
- The role of this CMake configuration is as follows:
  - Collect all `.c` source files in the `main/directory` as the component's source files.
  - Register the main component with the ESP-IDF build system and declare its dependencies on the custom components `bsp_audio` and `bsp_sd`.
- This ensures that during the build process, ESP-IDF knows to build these two components first, followed by the main component.



Note: In subsequent courses, we will not create a new "CMakeLists.txt" file from scratch. Instead, we will make minor modifications to this existing file to integrate other drivers into the main function.

## Complete Code

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.linch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson12-Playing\\_Loca\\_Music\\_from\\_SD\\_Card](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.linch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson12-Playing_Loca_Music_from_SD_Card)

## Programming Steps

---

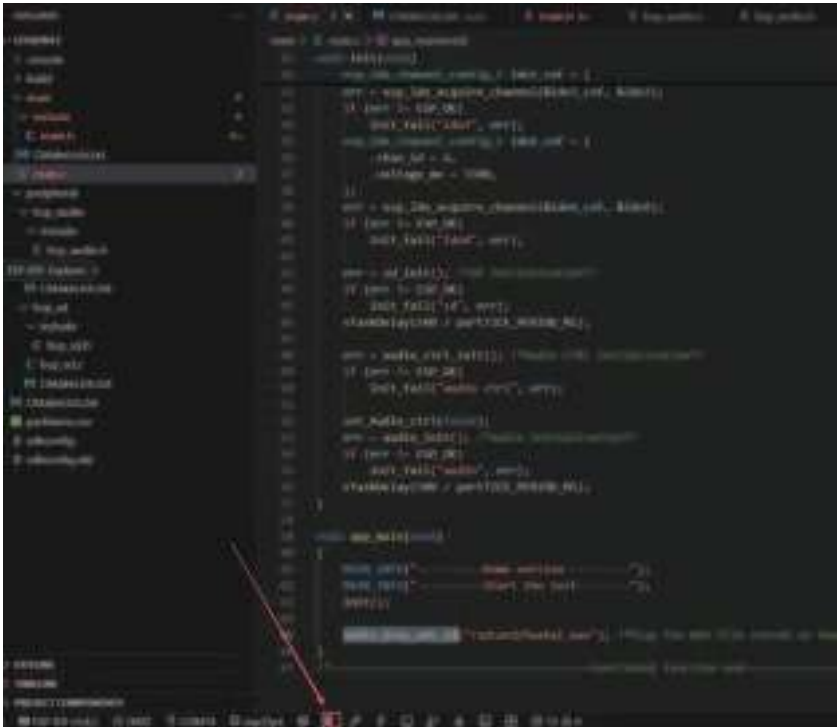
- Now that the code is ready, the next step is to flash it to the ESP32-P4 so we can observe the results.
- First, connect the **Advance-P4** device to your computer using a USB cable.



- 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**.



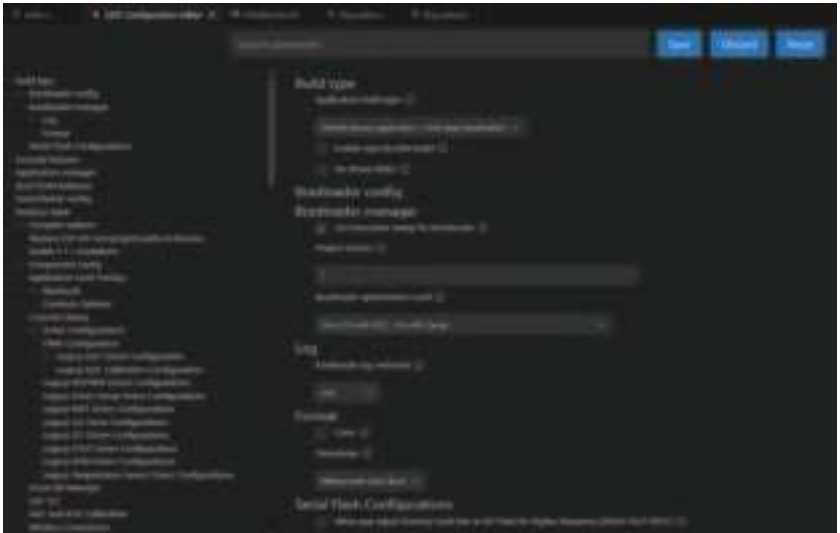
- Before starting the preparation for flashing, delete all files generated by compilation to restore the project to its initial "unbuilt" state. This ensures that subsequent compilations are not affected by your previous operations.



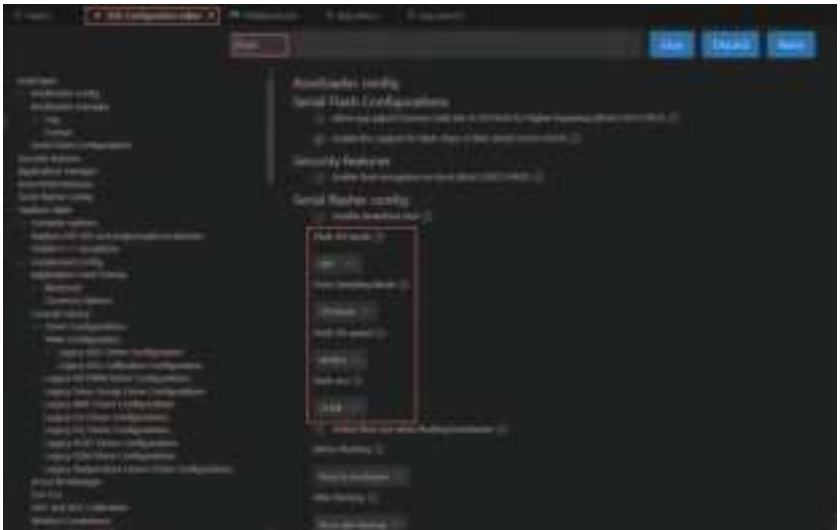
- First, follow the steps in the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon in the figure below.



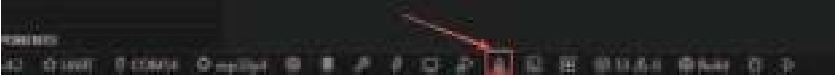
- Wait for a short loading period, then you can proceed with the relevant SDK configuration.



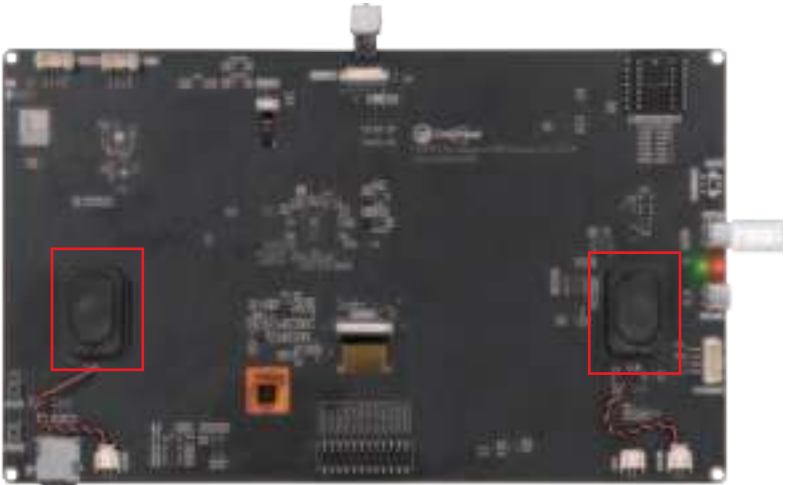
- Next, enter "flash" in the search box. (Make sure your flash configuration matches mine.)



- After completing the configuration, remember to save your settings.
- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we'd like to share a very convenient feature: a single button can execute compilation, upload, and monitor opening in one go (provided the entire code is error-free).



- After waiting for a while, the code will finish compiling and uploading, and the monitor will open automatically.
- Once the code runs, you will hear the speaker on the **Advance-P4** playing the WAV audio stored in your SD card.



The WAV audio file from your SD card is now playing.

# Lesson 13

## Camera Real-Time

### Introduction

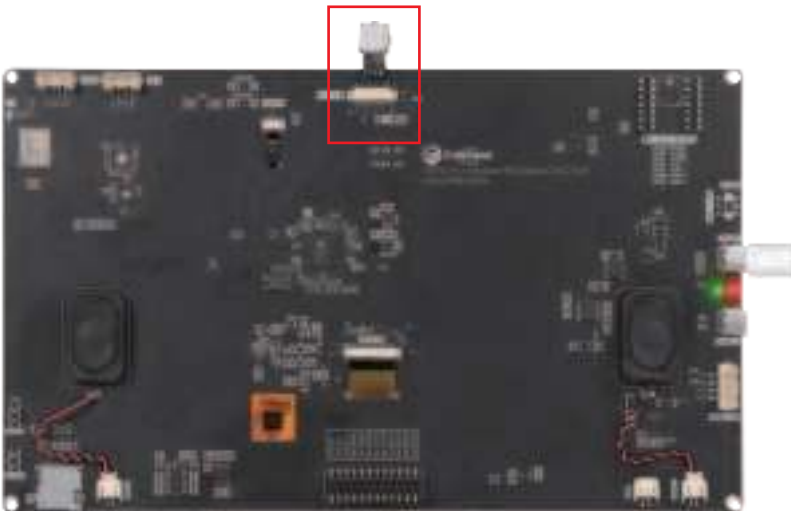
---

In this lesson, we will start teaching you how to activate the camera, enabling real-time display of the camera feed on the Advance-P4 screen.

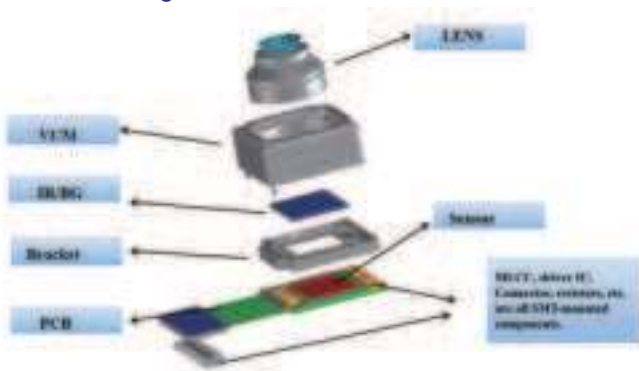
### Hardware Used in This Lesson

---

#### The camera on the Advance-P4



## Camera Schematic Diagram



First, the lens serves as a "collector" of light. Its optical structure can capture light from external scenes and, through its curvature and other design features, converge this light to provide a basic optical signal for subsequent imaging.

Next, the Voice Coil Motor (VCM) plays a key role in autofocus. Based on control signals from the circuit, it uses the principle of electromagnetic induction to drive the lens to move precisely within a certain range. By changing the distance between the lens and the image sensor (Sensor), it adjusts the focal point of the light, ensuring that the object being photographed is clearly imaged on the Sensor. Before the light reaches the Sensor, the IR cut/blue glass filter (IR/BG) filters the light. The IR cut filter blocks infrared light, as infrared light can interfere with visible light imaging and cause color distortion. The blue glass filter not only blocks infrared light but also reduces the entry of stray light, further improving light purity and making the light received by the subsequent Sensor more conducive to forming images with accurate colors and clarity.

Then, the image sensor (Sensor), as a core component, is covered with photosensitive elements such as photodiodes on its surface, which convert the received optical signals into electrical signals. Light of different intensities causes the photosensitive elements to generate electrical signals of different magnitudes, corresponding to information such as brightness and color in the scene.

Finally, components such as Multilayer Ceramic Capacitors (MLCC), driver integrated circuits (driver ICs), connectors, and resistors mounted on the Printed Circuit Board (PCB) form a complete signal processing and transmission system through circuit connections. The driver IC is responsible for preliminary processing of the electrical signals generated by the Sensor, such as amplification and analog-to-digital conversion, converting analog electrical signals into digital signals. Capacitors like MLCC and resistors stabilize voltage, filter noise, and ensure the stable operation of the circuit.

The digital signals processed in this way are then transmitted through connectors to subsequent devices (such as the main control chips of mobile phones and cameras), and finally decoded and rendered into the digital images we see.

## Operation Effect Diagram

---

After running the code, you will be able to see the real-time feed from the camera displayed on the screen of the Advance-P4.



## Key Explanations

---

- Now, the key focus of this lesson is how to use the camera and display the camera feed on the screen.
- Here, we will prepare another new component for you: "bsp\_camera".
- The main functions of this component are as follows:
  - Initialize the camera hardware (including I2C communication, MIPI CSI interface, and ISP (Image Signal Processing)).
  - Implement ISP (Image Signal Processing) workflows such as Auto Exposure (AE), Auto White Balance (AWB), and Color Correction Matrix (CCM).
  - Acquire real-time image data from the camera and display it on the screen (using the LVGL graphics library).
  - Provide functions for refresh control, display control, and buffer control.

- You just need to know when to call the interfaces we have written.
- Next, let's focus on understanding the "bsp\_camera" component.
- First, click the GitHub link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/df-code/Lesson13-Camera\\_Real-Time](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/df-code/Lesson13-Camera_Real-Time)

- Then drag the code for this lesson into VS Code and open the project file.
- Once opened, you can see the framework of this project.



In the example of this lesson, a new folder named "bsp\_camera" has been created under "peripheral\". Within the "bsp\_camera\" folder, a new "include" folder and a "CMakeLists.txt" file have been created.

The "bsp\_camera" folder contains the driver file "bsp\_camera.c", and the "include" folder contains the header file "bsp\_camera.h".

The "CMakeLists.txt" file integrates the driver into the build system, enabling the project to utilize the camera initialization and related display functions written in "bsp\_camera.c".

## Camera Display Code

- The camera display code consists of two files: "bsp\_camera.c" and "bsp\_camera.h".
- Next, we will first analyze the "bsp\_camera.h" program.
- "bsp\_camera.h" is the header file for camera display, mainly used to:



- We need to fill in the versions of "esp\_cam\_sensor", "esp\_cam\_sensor", and "esp\_cam\_sensor" in the "idf\_component.yml" file under the main folder.
- Since these are official libraries, we need to rely on them to implement the camera functionality on our Advance-P4.



- During subsequent compilation, the project will automatically download the esp\_cam\_sensor library version 1.2.0, esp\_cam\_sensor version 0.0.5, and esp\_video version 1.1.0. Once the download is complete, these network components will be stored in the "managed\_components" folder (which is automatically generated after filling in the version numbers).
- Next, we need to declare the variables we will use and the functions whose specific implementations are in "bsp\_camera.c".
- Centralizing these declarations in "bsp\_camera.h" facilitates easier calling and management. (We will explore their specific roles when they are used in "bsp\_camera.c".)



- Let's take a look at the specific functions of each function in "bsp\_camera.c".
- The "bsp\_camera" component provides significant support for everyone to use the camera later. By understanding the role of each function clearly, you can use the camera conveniently.
- We won't explain the code in detail here; we'll only tell you what each function does and under what circumstances to call it.

## 1. example\_isp\_awb\_on\_statistics\_done\_cb()

### Function:

- This is a callback function for the Auto White Balance (AWB) module in the ISP (Image Signal Processor). It is called when the AWB module completes its statistics calculation.
- Currently, it simply returns true to indicate "default processing after statistics completion" and has no actual operational logic.

### Calling Timing:

- Automatically invoked by the underlying ISP driver (when the ISP finishes the white balance statistics for a single frame of image).

## 2. camera\_get\_new\_vb()

### Function:

- Provides a new frame buffer for the Camera Controller.
- When the camera is ready to capture a new frame of image, the driver will call this function to obtain the memory address of the buffer.

### Calling Timing:

- Automatically invoked by the underlying camera driver, when the controller detects that it can capture a new frame of image.

## 3. camera\_get\_finished\_trans()

### Function:

- Used to notify that the transmission of a frame of image has been completed.
- Currently, the function does nothing internally (it simply returns false), meaning no special processing is temporarily required for the completed image.

### Calling Timing:

- Automatically invoked by the camera controller, triggered when the transmission of a frame of data from the camera to memory is completed.

## 4. camera\_sensor\_init()

### Function:

- Initializes the operating parameters and communication interface of the camera sensor itself.
- It mainly includes the following steps:
  - Initialize SCCB (I2C bus) communication;
  - Automatically detect the model of the connected camera;
  - Set resolution, pixel format (RAW8), and frame rate;
  - Set mirroring (horizontal flip), exposure time, and exposure value;
  - Enable video data stream output.

### Calling Timing:

- During the overall camera system initialization (called within camera\_init()).

## 5. camera\_csi\_init()

### Function:

- Initializes the camera's MIPI-CSI interface controller, which is the module responsible for receiving camera data streams.
- It mainly completes the following tasks:
  - Configure CSI controller parameters (resolution, data rate, number of channels, etc.);
  - Register data transmission callbacks (camera\_get\_new\_vb, camera\_get\_finished\_trans);
  - Enable the controller.

### Calling Timing:

- Also during the camera initialization phase (called within camera\_init()).

## 6. `isp_init()`

### Function:

- Initializes the ISP (Image Signal Processor) module.
- The ISP is responsible for processing the raw image data (RAW data) output by the camera to convert it into RGB images.
- This includes:
  - Enabling the main ISP module;
  - Setting color adjustment parameters (brightness, contrast, saturation, hue);
  - Enabling the Auto White Balance (AWB) controller;
  - Enabling the Auto Exposure (AE) controller;
  - Enabling the Color Correction Matrix (CCM).

### Calling Timing:

- During the camera initialization phase (called within `camera_init()`).

## 7. `camera_init()`

### Function:

- This is the "main initialization function" for the entire camera subsystem.
- It is responsible for:
  - Allocating image buffers for the camera (located in external PSRAM);
  - Calling the three core initialization functions mentioned earlier:
    - `camera_sensor_init()` → Initializes the camera sensor;
    - `camera_csi_init()` → Initializes the image reception interface;
    - `isp_init()` → Initializes image signal processing;
  - Starting the camera data stream acquisition.

### Calling Timing:

- When the system powers on (usually called once in `app_main()` or during the device initialization phase).

## 8. camera\_refresh()

### Function:

- Manually triggers the camera to capture a frame of image.
- Essentially, it calls `esp_cam_ctr_receive()` to receive a frame of image data.

### Calling Timing:

- Invoked when the application layer needs to refresh the camera image, such as:
  - The first capture after program startup;
  - Manual refresh by the user;
  - Periodic calls in timed tasks.

## 9. camera\_display\_refresh()

### Function:

- Notifies LVGL to refresh the camera feed display area.
- It calls `lv_obj_invalidate()`, which prompts LVGL to redraw the camera image in the next rendering cycle.

### Calling Timing:

- Invoked after the image content is updated (e.g., within the loop of `camera_display_task()`).

## 10. camera\_display()

### Function:

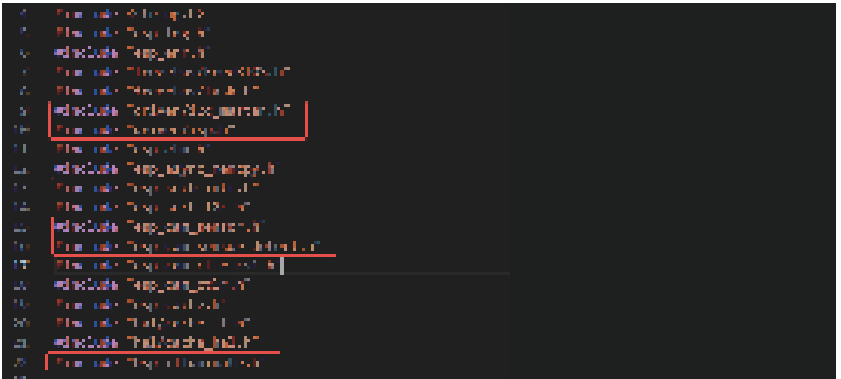
- Creates an image object in LVGL for displaying the camera feed.
- The specific steps are as follows:
  - Create an `lv_img` object;
  - Set center alignment for the object;
  - Bind the image buffer (RGB565 data captured by the camera);
  - Configure the image source;
  - Unlock LVGL to allow rendering.

## Calling Timing:

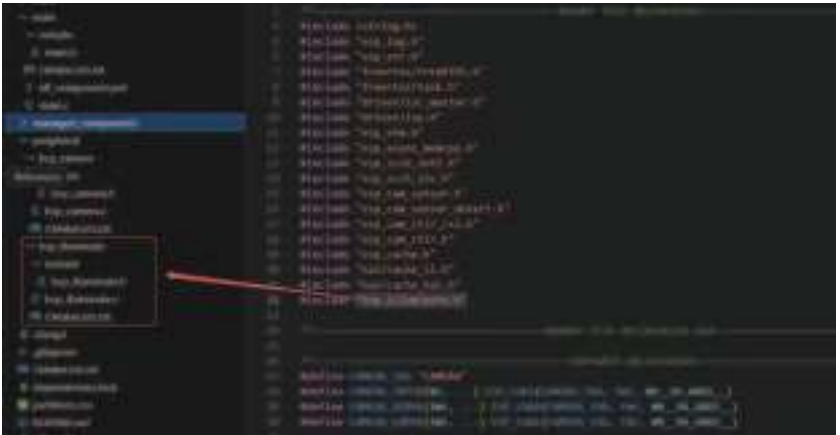
- Called once after the camera is initialized successfully, to create and display the image control (invoked within Init()).
- This concludes our introduction to the bsp\_camera component. For your purposes, it is sufficient to know how to call these interfaces.
- If you need to call these interfaces, you must also configure the "CMakeLists.txt" file under the bsp\_camera folder. This file, located in the bsp\_camera folder, primarily functions to tell the ESP-IDF build system (CMake): how to compile and register the bsp\_camera component.



- The reason for including "driver", "esp\_cam\_sensor", "esp\_sccb\_intf", "esp\_video", and "bsp\_illuminate" is that we have called these in "bsp\_camera.h" (other libraries that are system libraries do not need to be added).



- For example, "bsp\_illuminate.h" is a component related to screen display that we explained earlier. Since it was covered in detail before, we won't go into it again here.
- It is used to initialize the screen, turn on the screen backlight, and enable the screen to display relevant content.



## Main function

- The main folder is the core directory for program execution, which contains the main function executable file main.c.
- Add the main folder to the "CMakeLists.txt" file of the build system.







### **init\_fail:**

- Initialization failure handling function:
  - Uses static bool state to prevent repeated printing.
  - Runs in an infinite loop, printing initialization failure messages.
  - Delays for 1 second per cycle.
- **Function:** Once any hardware initialization fails, the program stops further execution and prints error messages.



### **init:**

- Hardware initialization function during system startup.

### **Initialization Steps:**

- Configure LDO3 (2.5V) and LDO4 (3.3V) to supply power to the LCD.
- Install the GPIO interrupt service via `gpio_install_isr_service`.
- Initialize the LCD display with `display_init()`.
- Turn on the LCD backlight using `set_lcd_blight(100)`.
- Initialize the camera module with `camera_init()`.

**Calling Scenario:** Invoked once within `app_main()` when the program starts.

```

34 void initCamera() { // Camera initialization function
35     // Create and start a task to handle the camera refresh
36
37     // Create the camera task configuration
38     struct TaskConfig { // Task Config
39         int priority = 1; // Priority: 1
40         int stack_size = 1024; // Stack size: 1024
41     };
42     esp_err_t esp_err_acquire_channel(1024, 1024); // Acquire and configure task channel
43     if (esp_err_t != ESP_OK) // Check for error
44         exit_fatal("init", esp_err); // Handle initialization failure
45
46     // Create the camera task configuration
47     struct TaskConfig { // Task Config
48         int priority = 1; // Priority: 1
49         int stack_size = 1024; // Stack size: 1024
50     };
51
52     esp_err_t esp_err_acquire_channel(1024, 1024); // Acquire and configure task channel
53     if (esp_err_t != ESP_OK) // Check for error
54         exit_fatal("init", esp_err); // Handle initialization failure
55
56     esp_err_t esp_err_install_service(1); // Install the camera service function
57     if (esp_err_t != ESP_OK) // Check for error
58         exit_fatal("init", esp_err); // Handle initialization failure
59
60     esp_err_t esp_err_display(1); // Initialize the display
61     if (esp_err_t != ESP_OK) // Check for error
62         exit_fatal("init", esp_err); // Handle initialization failure
63
64     esp_err_t esp_err_set_backlight(100); // Enable backlight with 100% brightness
65     if (esp_err_t != ESP_OK) // Check for error
66         exit_fatal("set backlight", esp_err); // Handle failure
67
68     // Log success message for backlight
69     printf("LED backlight opened (brightness: 100%)\n");
70
71     esp_err_t esp_err_camera_init(); // Initialize camera module
72     if (esp_err_t != ESP_OK) // Check for error
73         exit_fatal("camera", esp_err); // Handle initialization failure
74
75     camera_display(); // Initialize camera display output
76 }

```

## app\_main:

- The program entry point for ESP32 FreeRTOS.

## Process:

- Print the log "Camera task".
- Call `Init()` to initialize the system.
- Call `camera_refresh()` to retrieve a new frame of image data from the camera controller into the buffer, providing the latest frame for subsequent display or processing.
- Create the `camera_display_task` task, attach the display task to Core 1 with a relatively high priority.
- Print the log "The screen is displaying" to indicate that the display has started.

```
10 #endif
11
12 #endif
13
14 #endif
15
16 #endif
17
18 #endif
19
20 #endif
21
22 #endif
23
24 #endif
25
26 #endif
27
28 #endif
29
30 #endif
31
32 #endif
33
34 #endif
35
36 #endif
37
38 #endif
39
40 #endif
41
42 #endif
43
44 #endif
45
46 #endif
47
48 #endif
49
50 #endif
51
52 #endif
53
54 #endif
55
56 #endif
57
58 #endif
59
60 #endif
61
62 #endif
63
64 #endif
65
66 #endif
67
68 #endif
69
70 #endif
71
72 #endif
73
74 #endif
75
76 #endif
77
78 #endif
79
80 #endif
81
82 #endif
83
84 #endif
85
86 #endif
87
88 #endif
89
90 #endif
91
92 #endif
93
94 #endif
95
96 #endif
97
98 #endif
99
100 #endif
```

Finally, let's take a look at the "CMakeLists.txt" file in the main directory.

The role of this CMake configuration is as follows:

- Collect all .c source files in the main/ directory as the source files of the component.
- Register the main component with the ESP-IDF build system, and declare that it depends on the custom component "bsp\_camera" and the custom component "bsp\_illuminate".

In this way, during the build process, ESP-IDF will know to build "bsp\_camera" and "bsp\_illuminate" first, and then build "main".

```
1 # ESP-IDF CMakeLists.txt
2
3 # This file is used to describe the components of the project.
4 # See the documentation for more details.
5
6 # The main component of the project is the main function.
7 # It depends on the custom component "bsp_camera" and the custom component
8 # "bsp_illuminate".
9
10 # Register the main component with the ESP-IDF build system.
11 # The main component is the main function.
12 # It depends on the custom component "bsp_camera" and the custom component
13 # "bsp_illuminate".
14
15 # The main component is the main function.
16 # It depends on the custom component "bsp_camera" and the custom component
17 # "bsp_illuminate".
18
19 # The main component is the main function.
20 # It depends on the custom component "bsp_camera" and the custom component
21 # "bsp_illuminate".
22
23 # The main component is the main function.
24 # It depends on the custom component "bsp_camera" and the custom component
25 # "bsp_illuminate".
26
27 # The main component is the main function.
28 # It depends on the custom component "bsp_camera" and the custom component
29 # "bsp_illuminate".
30
31 # The main component is the main function.
32 # It depends on the custom component "bsp_camera" and the custom component
33 # "bsp_illuminate".
34
35 # The main component is the main function.
36 # It depends on the custom component "bsp_camera" and the custom component
37 # "bsp_illuminate".
38
39 # The main component is the main function.
40 # It depends on the custom component "bsp_camera" and the custom component
41 # "bsp_illuminate".
42
43 # The main component is the main function.
44 # It depends on the custom component "bsp_camera" and the custom component
45 # "bsp_illuminate".
46
47 # The main component is the main function.
48 # It depends on the custom component "bsp_camera" and the custom component
49 # "bsp_illuminate".
50
51 # The main component is the main function.
52 # It depends on the custom component "bsp_camera" and the custom component
53 # "bsp_illuminate".
54
55 # The main component is the main function.
56 # It depends on the custom component "bsp_camera" and the custom component
57 # "bsp_illuminate".
58
59 # The main component is the main function.
60 # It depends on the custom component "bsp_camera" and the custom component
61 # "bsp_illuminate".
62
63 # The main component is the main function.
64 # It depends on the custom component "bsp_camera" and the custom component
65 # "bsp_illuminate".
66
67 # The main component is the main function.
68 # It depends on the custom component "bsp_camera" and the custom component
69 # "bsp_illuminate".
70
71 # The main component is the main function.
72 # It depends on the custom component "bsp_camera" and the custom component
73 # "bsp_illuminate".
74
75 # The main component is the main function.
76 # It depends on the custom component "bsp_camera" and the custom component
77 # "bsp_illuminate".
78
79 # The main component is the main function.
80 # It depends on the custom component "bsp_camera" and the custom component
81 # "bsp_illuminate".
82
83 # The main component is the main function.
84 # It depends on the custom component "bsp_camera" and the custom component
85 # "bsp_illuminate".
86
87 # The main component is the main function.
88 # It depends on the custom component "bsp_camera" and the custom component
89 # "bsp_illuminate".
90
91 # The main component is the main function.
92 # It depends on the custom component "bsp_camera" and the custom component
93 # "bsp_illuminate".
94
95 # The main component is the main function.
96 # It depends on the custom component "bsp_camera" and the custom component
97 # "bsp_illuminate".
98
99 # The main component is the main function.
100 # It depends on the custom component "bsp_camera" and the custom component
101 # "bsp_illuminate".
```

Note: In the subsequent courses, we will not create a new "CMakeLists.txt" file from scratch. Instead, we will make minor modifications to this existing file to integrate other drivers into the main function

## Complete Code

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson13-Camera\\_Real-Time](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-To-uch-Screen/tree/master/example/V1.0/idf-code/Lesson13-Camera_Real-Time)

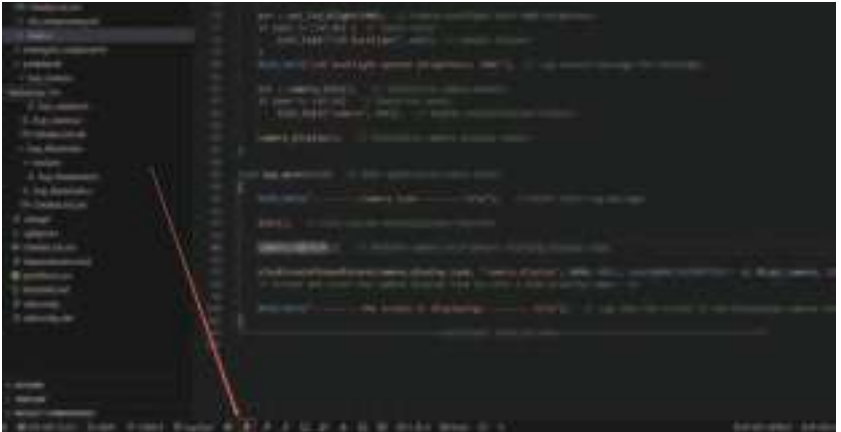
## Programming Steps

---

- Now that the code is ready, the next step is to flash it to the ESP32-P4 so we can observe the actual behavior.
- First, connect the Advance-P4 device to your computer via a USB cable.



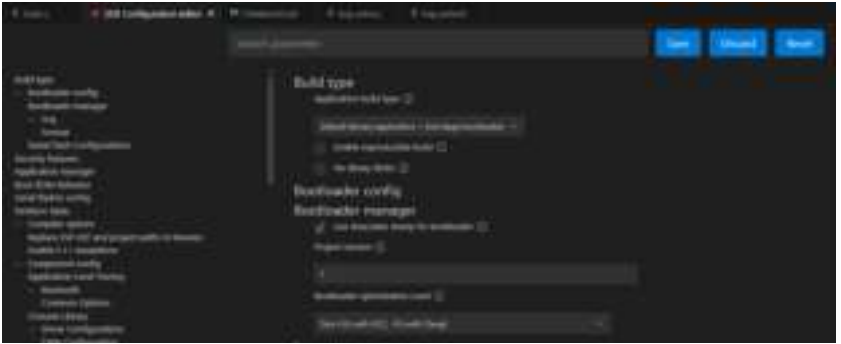
- Before starting the flashing preparation, delete all compiled files to restore the project to its initial "unbuilt" state. (This ensures that subsequent compilations are not affected by your previous build artifacts.)



- First, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip correctly.
- Next, we need to configure the SDK.
- Click the icon shown in the image below.



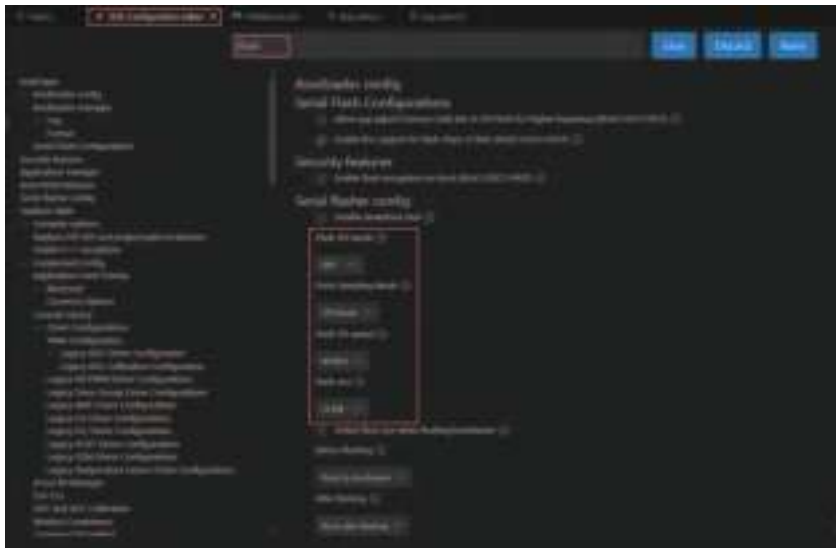
- Wait for a short loading period, and then you can proceed with the relevant SDK configuration.



- Here you need to configure the camera options to use the camera normally.
- (Just make the same configuration as shown in the picture.)



- After that, enter "flash" in the search box to find flash-related settings. (Make sure your flash configuration matches mine exactly.)



- After completing the configuration, remember to save your settings.
- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we will also introduce a very convenient feature: a single button can execute compilation, upload, and monitor activation in one go.



- Wait for a moment until the code compilation and upload are completed, and the monitor will open automatically.
- At this point, please remember to connect your Advance-P4 with an additional Type-C cable via the USB 2.0 port. 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.)



- After running the code, you will be able to see the real-time feed from the camera on the Advance-P4 screen.



# Lesson 14

## 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.

### Hardware Used in This Lesson

---

#### SX1262 Wireless Module on the Advance-P4



## Operation Effect Diagram

---

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.



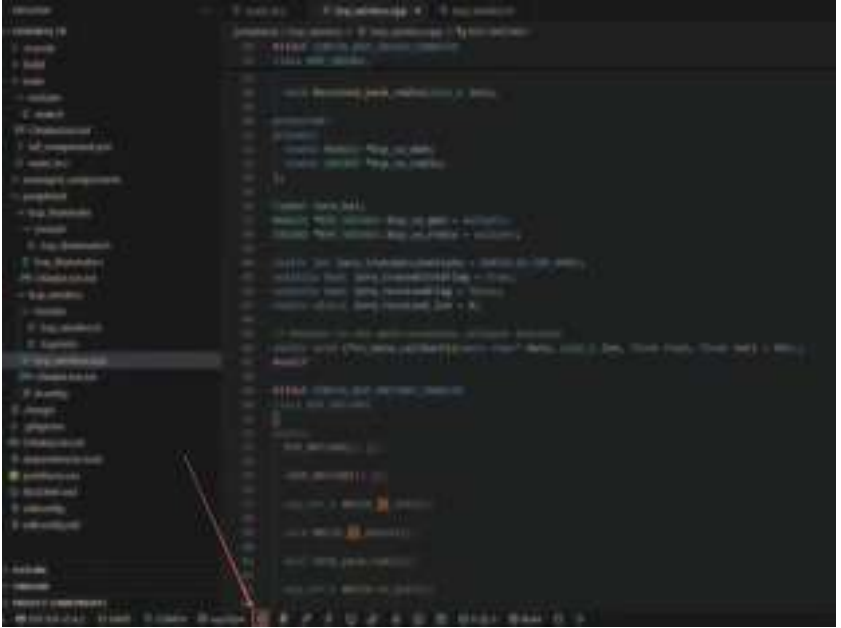
## Key Explanations

---

- 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.
- 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:

- Click SDK Configuration.



- Search for "wireless" and open the configuration you are using.



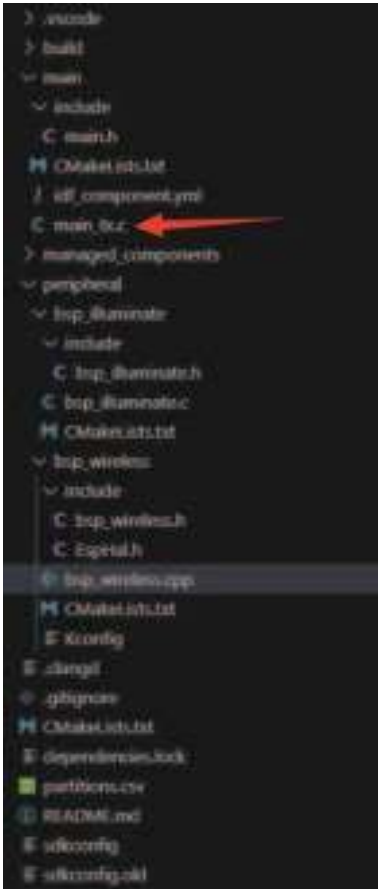
- Since in this case we are using the **SX1262**, only check the option "**Enable SX1262 config**" and uncheck all the others.

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

After making the changes, don't forget to click Save to apply and store the modifications.



- Then, drag the code for this lesson into VS Code and open the project file.
- Once opened, you will see the project structure.
- 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: main\_tx.c for the transmitter and main\_rx.c for the receiver. All other code files are identical. (For convenience, we have prepared both main functions for you to use separately.)

- In this lesson's example, a new folder named `bsp_wireless` has been created under `peripheral\`. Inside the `bsp_wireless\` folder, there is a new include folder and a `CMakeLists.txt` file.
- The `bsp_wireless` folder contains the driver file `bsp_wireless.cpp`, while the include folder contains the header files `bsp_wireless.h` 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.
- The `CMakeLists.txt` file integrates the driver into the build system, allowing the project to use the LoRa module transmission and reception functions implemented in `bsp_wireless.cpp`.
- Additionally, there is `bsp_illuminate`, our familiar component from previous lessons, which we use to light up the screen and render text using `LVGL`.
- 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`.

```

cmake_minimum_required(VERSION 3.10)
project("peripheral")

include_directories(
  include
  ../bsp_wireless/include
)

add_executable(peripheral main.cpp)

target_include_directories(peripheral PRIVATE
  include
  ../bsp_wireless/include
  ../bsp_illuminate/include
  ../bsp_wifi/include
  ../bsp_uart/include
  ../bsp_lcd/include
  ../bsp_led/include
  ../bsp_buzzer/include
)
  
```

```
#include <bsp_illuminate.h>
#include <bsp_wifi.h>
#include <bsp_wifi.h>
#include <bsp_wifi.h>
#include <bsp_wifi.h>
```

- 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.

```
esp8266
esp8266
esp8266
esp8266
esp8266
```

- This requires us to specify the version of `kgromes/radiolib` in the `idf_component.yml` file located in the main 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.

```
idf_component.yml
idf_component.yml
idf_component.yml
```

- These three components, which we discussed earlier, are used in the `bsp_illuminate` component to light up the screen and render information on the interface using LVGL.

```
esp8266
esp8266
esp8266
esp8266
```



- 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).

```

1 // Core global variables
2
3 #include
4
5 #define
6
7 #define
8
9 #define
10
11 #define
12
13 #define
14
15 #define
16
17 #define
18
19 #define
20
21 #define
22
23 #define
24
25 #define
26
27 #define
28
29 #define
30
31 #define
32
33 #define
34
35 #define
36
37 #define
38
39 #define
40
41 #define
42
43 #define
44
45 #define
46
47 #define
48
49 #define
50
51 #define
52
53 #define
54
55 #define
56
57 #define
58
59 #define
60
61 #define
62
63 #define
64
65 #define
66
67 #define
68
69 #define
70
71 #define
72
73 #define
74
75 #define
76
77 #define
78
79 #define
80
81 #define
82
83 #define
84
85 #define
86
87 #define
88
89 #define
90
91 #define
92
93 #define
94
95 #define
96
97 #define
98
99 #define
100
101 #define
102
103 #define
104
105 #define
106
107 #define
108
109 #define
110
111 #define
112
113 #define
114
115 #define
116
117 #define
118
119 #define
120
121 #define
122
123 #define
124
125 #define
126
127 #define
128
129 #define
130
131 #define
132
133 #define
134
135 #define
136
137 #define
138
139 #define
140
141 #define
142
143 #define
144
145 #define
146
147 #define
148
149 #define
150
151 #define
152
153 #define
154
155 #define
156
157 #define
158
159 #define
160
161 #define
162
163 #define
164
165 #define
166
167 #define
168
169 #define
170
171 #define
172
173 #define
174
175 #define
176
177 #define
178
179 #define
180
181 #define
182
183 #define
184
185 #define
186
187 #define
188
189 #define
190
191 #define
192
193 #define
194
195 #define
196
197 #define
198
199 #define
200
201 #define
202
203 #define
204
205 #define
206
207 #define
208
209 #define
210
211 #define
212
213 #define
214
215 #define
216
217 #define
218
219 #define
220
221 #define
222
223 #define
224
225 #define
226
227 #define
228
229 #define
230
231 #define
232
233 #define
234
235 #define
236
237 #define
238
239 #define
240
241 #define
242
243 #define
244
245 #define
246
247 #define
248
249 #define
250
251 #define
252
253 #define
254
255 #define
256
257 #define
258
259 #define
260
261 #define
262
263 #define
264
265 #define
266
267 #define
268
269 #define
270
271 #define
272
273 #define
274
275 #define
276
277 #define
278
279 #define
280
281 #define
282
283 #define
284
285 #define
286
287 #define
288
289 #define
290
291 #define
292
293 #define
294
295 #define
296
297 #define
298
299 #define
300
301 #define
302
303 #define
304
305 #define
306
307 #define
308
309 #define
310
311 #define
312
313 #define
314
315 #define
316
317 #define
318
319 #define
320
321 #define
322
323 #define
324
325 #define
326
327 #define
328
329 #define
330
331 #define
332
333 #define
334
335 #define
336
337 #define
338
339 #define
340
341 #define
342
343 #define
344
345 #define
346
347 #define
348
349 #define
350
351 #define
352
353 #define
354
355 #define
356
357 #define
358
359 #define
360
361 #define
362
363 #define
364
365 #define
366
367 #define
368
369 #define
370
371 #define
372
373 #define
374
375 #define
376
377 #define
378
379 #define
380
381 #define
382
383 #define
384
385 #define
386
387 #define
388
389 #define
390
391 #define
392
393 #define
394
395 #define
396
397 #define
398
399 #define
400
401 #define
402
403 #define
404
405 #define
406
407 #define
408
409 #define
410
411 #define
412
413 #define
414
415 #define
416
417 #define
418
419 #define
420
421 #define
422
423 #define
424
425 #define
426
427 #define
428
429 #define
430
431 #define
432
433 #define
434
435 #define
436
437 #define
438
439 #define
440
441 #define
442
443 #define
444
445 #define
446
447 #define
448
449 #define
450
451 #define
452
453 #define
454
455 #define
456
457 #define
458
459 #define
460
461 #define
462
463 #define
464
465 #define
466
467 #define
468
469 #define
470
471 #define
472
473 #define
474
475 #define
476
477 #define
478
479 #define
480
481 #define
482
483 #define
484
485 #define
486
487 #define
488
489 #define
490
491 #define
492
493 #define
494
495 #define
496
497 #define
498
499 #define
500
501 #define
502
503 #define
504
505 #define
506
507 #define
508
509 #define
510
511 #define
512
513 #define
514
515 #define
516
517 #define
518
519 #define
520
521 #define
522
523 #define
524
525 #define
526
527 #define
528
529 #define
530
531 #define
532
533 #define
534
535 #define
536
537 #define
538
539 #define
540
541 #define
542
543 #define
544
545 #define
546
547 #define
548
549 #define
550
551 #define
552
553 #define
554
555 #define
556
557 #define
558
559 #define
560
561 #define
562
563 #define
564
565 #define
566
567 #define
568
569 #define
570
571 #define
572
573 #define
574
575 #define
576
577 #define
578
579 #define
580
581 #define
582
583 #define
584
585 #define
586
587 #define
588
589 #define
590
591 #define
592
593 #define
594
595 #define
596
597 #define
598
599 #define
600
601 #define
602
603 #define
604
605 #define
606
607 #define
608
609 #define
610
611 #define
612
613 #define
614
615 #define
616
617 #define
618
619 #define
620
621 #define
622
623 #define
624
625 #define
626
627 #define
628
629 #define
630
631 #define
632
633 #define
634
635 #define
636
637 #define
638
639 #define
640
641 #define
642
643 #define
644
645 #define
646
647 #define
648
649 #define
650
651 #define
652
653 #define
654
655 #define
656
657 #define
658
659 #define
660
661 #define
662
663 #define
664
665 #define
666
667 #define
668
669 #define
670
671 #define
672
673 #define
674
675 #define
676
677 #define
678
679 #define
680
681 #define
682
683 #define
684
685 #define
686
687 #define
688
689 #define
690
691 #define
692
693 #define
694
695 #define
696
697 #define
698
699 #define
700
701 #define
702
703 #define
704
705 #define
706
707 #define
708
709 #define
710
711 #define
712
713 #define
714
715 #define
716
717 #define
718
719 #define
720
721 #define
722
723 #define
724
725 #define
726
727 #define
728
729 #define
730
731 #define
732
733 #define
734
735 #define
736
737 #define
738
739 #define
740
741 #define
742
743 #define
744
745 #define
746
747 #define
748
749 #define
750
751 #define
752
753 #define
754
755 #define
756
757 #define
758
759 #define
760
761 #define
762
763 #define
764
765 #define
766
767 #define
768
769 #define
770
771 #define
772
773 #define
774
775 #define
776
777 #define
778
779 #define
780
781 #define
782
783 #define
784
785 #define
786
787 #define
788
789 #define
790
791 #define
792
793 #define
794
795 #define
796
797 #define
798
799 #define
800
801 #define
802
803 #define
804
805 #define
806
807 #define
808
809 #define
810
811 #define
812
813 #define
814
815 #define
816
817 #define
818
819 #define
820
821 #define
822
823 #define
824
825 #define
826
827 #define
828
829 #define
830
831 #define
832
833 #define
834
835 #define
836
837 #define
838
839 #define
840
841 #define
842
843 #define
844
845 #define
846
847 #define
848
849 #define
850
851 #define
852
853 #define
854
855 #define
856
857 #define
858
859 #define
860
861 #define
862
863 #define
864
865 #define
866
867 #define
868
869 #define
870
871 #define
872
873 #define
874
875 #define
876
877 #define
878
879 #define
880
881 #define
882
883 #define
884
885 #define
886
887 #define
888
889 #define
890
891 #define
892
893 #define
894
895 #define
896
897 #define
898
899 #define
900
901 #define
902
903 #define
904
905 #define
906
907 #define
908
909 #define
910
911 #define
912
913 #define
914
915 #define
916
917 #define
918
919 #define
920
921 #define
922
923 #define
924
925 #define
926
927 #define
928
929 #define
930
931 #define
932
933 #define
934
935 #define
936
937 #define
938
939 #define
940
941 #define
942
943 #define
944
945 #define
946
947 #define
948
949 #define
950
951 #define
952
953 #define
954
955 #define
956
957 #define
958
959 #define
960
961 #define
962
963 #define
964
965 #define
966
967 #define
968
969 #define
970
971 #define
972
973 #define
974
975 #define
976
977 #define
978
979 #define
980
981 #define
982
983 #define
984
985 #define
986
987 #define
988
989 #define
990
991 #define
992
993 #define
994
995 #define
996
997 #define
998
999 #define
1000

```

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.



### **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.



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.

```

100 void bsp_sx1262_init(bsp_sx1262_t* bsp_sx1262)
101 {
102     bsp_sx1262->setFrequency(915000000);
103     bsp_sx1262->setPins(0, 1, 2);
104     bsp_sx1262->setMode(MODE_RX);
105     bsp_sx1262->setPacketReceivedAction(set_sx1262_rx_flag);
106     bsp_sx1262->setRxBoostedGainMode(true);
107     bsp_sx1262->startReceive();
108     if (state != STATE_RX_PACKET_RECEIVED) {
109         return FAILURE;
110     }
111     bsp_sx1262->setPacketReceivedAction(set_sx1262_rx_flag);
112     bsp_sx1262->setRxBoostedGainMode(true);
113     state = bsp_sx1262_startReceive();
114     if (state != STATE_RX_PACKET_RECEIVED) {
115         return FAILURE;
116     }
117     return SUCCESS;
118 }
  
```

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.



### `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()**

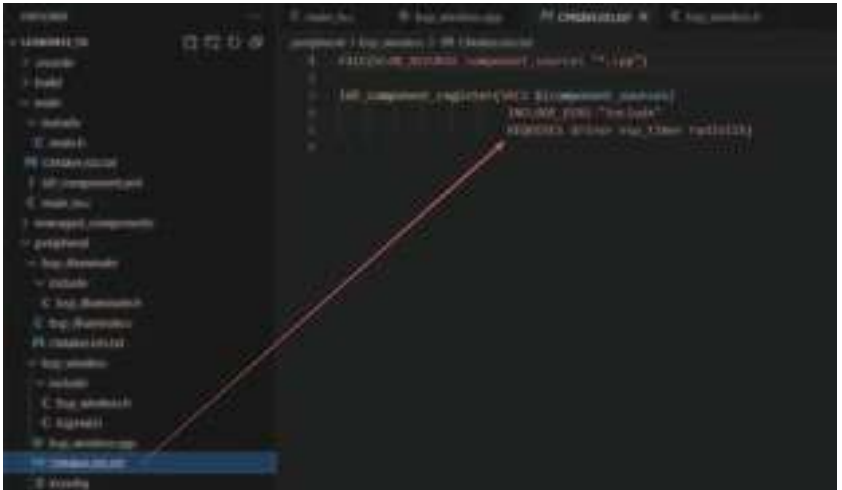
This is a query interface that returns the length of the most recently received data `lora_received_len`. Internally, the function simply returns the static variable without modifying any state. It is usually called when processing received data or performing debug/statistics, to obtain the actual length of the received packet.

## **sx1262\_is\_data\_received()**

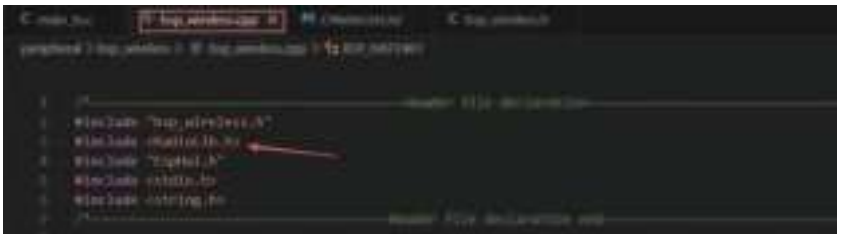
This is a status query interface that returns the reception flag `lora_receivedFlag`, used to determine whether a new data packet has arrived. The function simply returns the status of the global variable without modifying it. It is typically polled in the main loop or tasks to decide whether to call `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.

When calling these functions, you also need to configure the `CMakeLists.txt` file in the `bsp_wireless` folder. This file, located in the `bsp_wireless` directory, mainly tells the ESP-IDF build system (CMake) how to compile and register the `bsp_wireless` component.



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.



As well as the `esp_timer` used in the `EspHal.h` file.



A screenshot of a code editor showing the `esp_timer.h` header file. The code is written in C and includes various function declarations for timer management, such as `esp_timer_create`, `esp_timer_start_once`, and `esp_timer_start_periodic`. The editor has a dark theme and a sidebar on the left showing a project tree.

## Main function

The main folder is the core directory for program execution and contains the main executable file `main_tx.c`.

Add the main folder to the build system's `CMakeLists.txt` file.



A screenshot of a code editor showing the `CMakeLists.txt` file. The code is written in CMake and defines a project named `main_tx`. It includes the `add_executable` function to create the `main_tx` executable and the `target_link_libraries` function to link the `main_tx` target with the `main_tx` library. The editor has a dark theme and a sidebar on the left showing a project tree.

This is the entry file for the entire application. In ESP-IDF, there is no `int main()`; execution starts from `void app_main(void)`.

Let's first go through the transmitter main function file `main_tx.c` to see how it calls interfaces to send LoRa messages.



Let's continue looking at the content of `main_tx.c`.

## lvgl\_show\_counter\_label\_init:

The function `lvgl_show_counter_label_init()` initializes the **counter label** in the LVGL display interface, used to show the **LoRa transmission count**.

- The function first calls `lvgl_port_lock(0)` to acquire the LVGL operation lock, ensuring safe access to LVGL in a multi-task environment. If locking fails, it prints an error and returns.
- It then gets the current active screen object via `lv_scr_act()` and sets the screen background to white, fully covering the display.
- Next, it creates a label object `s_hello_label`. If creation fails, an error is printed, the lock is released, and the function returns.
- It then creates and initializes a style `label_style` for the label, setting the font to **Montserrat size 42**, text color to black, and background to transparent, and applies the style to the label.
- Using `lv_label_set_text()`, the initial text is set to "TX\_Hello World:0", and `lv_obj_center()` centers the label on the screen.
- Finally, `lvgl_port_unlock()` is called to release the LVGL lock, allowing other tasks to safely operate on LVGL.

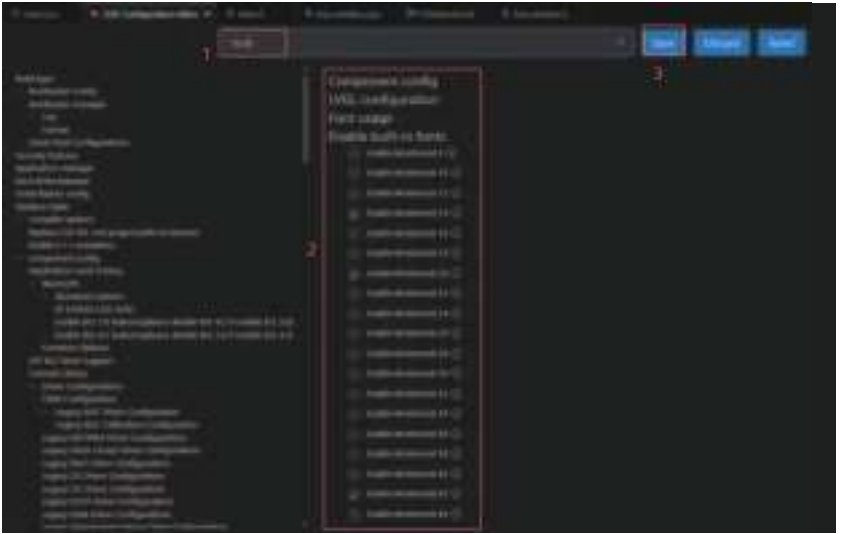
If you want to change the LVGL font size, you need to go into the SDK configuration and enable the desired font.

### Steps:

Click on the SDK Configuration option.



Search for "font" and select the **font size you want to use**. After making changes, **remember to save**.



### ui\_counter\_task:

The function `ui_counter_task()` is a FreeRTOS task that updates the LoRa transmission count label on the LVGL display every second.

- Inside the function, a character array `text[48]` is defined to store the formatted display text. The current system tick count is obtained via `xTaskGetTickCount()` as the task's initial wake time `last_wake_time`, and the task period frequency is set to 1000 milliseconds.
- The task enters an infinite loop. In each iteration, it calls `sx1262_get_tx_counter()` to get the current number of LoRa packets sent, then formats the string as "TX\_Hello World:count" using `snprintf`.
- It then attempts to acquire the LVGL operation lock. If successful and the label object `s_hello_label` is valid, it updates the label text and releases the lock, ensuring safe LVGL access in a multi-task environment.
- Next, it prints the current transmission information using `MAIN_INFO`.
- Finally, `vTaskDelayUntil()` is called with absolute timing to ensure each loop executes precisely every one second.

Overall, this task continuously refreshes the display with the LoRa transmission count while logging, providing real-time visual feedback.

## Hardware\_Init:

The function `Hardware_Init()` is used to initialize hardware modules when the program starts, ensuring that all parts of the system work properly.

- First, it calls `esp_ldo_acquire_channel()` to acquire the LDO3 (2.5V) and LDO4 (3.3V) power channels. If acquisition fails, it calls `init_or_halt()`, repeatedly waiting and printing error messages to ensure stable power.
- Next, it calls `display_init()` to initialize the LCD hardware and the LVGL graphics library, which must be done before turning on the backlight, otherwise the display may behave abnormally.
- Then, it calls `set_lcd_blight(100)` to turn on the LCD backlight and set the brightness to maximum 100, using `init_or_halt()` to check for errors.
- Finally, it calls `sx1262_tx_init()` to initialize the LoRa transmission module. If initialization fails, it is also handled via `init_or_halt()`.

Overall, this function provides a **reliable hardware environment** for screen display, backlight, and the wireless communication module, ensuring that subsequent program functionality runs smoothly. It is typically called in `app_main()` during system startup.

## lora\_tx\_task:

The function `"lora_tx_task()"` is a FreeRTOS task used to periodically send data packets through the LoRa module.

- The function first obtains the current system tick count using `"xTaskGetTickCount()"` as the start time of the task, and sets the transmission period to 1000 milliseconds (1 second).
- In an infinite loop, it calls `"send_lora_pack_radio()"` to attempt sending a LoRa data packet. It determines whether the transmission is successful through the return value, and if the transmission fails, it prints an error log using `"MAIN_ERROR"`.
- Finally, it uses `"vTaskDelayUntil()"` to delay according to absolute time, ensuring that each loop sends data at an accurate interval of 1 second, thus achieving timed and stable wireless data transmission.

This task is usually created after the system starts and runs continuously to continuously broadcast messages to the receiving end.

## app\_main:

The function `"app_main()"` is the entry point of the entire program. After the system starts, it first prints the `"LoRa TX"` log to indicate entering the main process.

Subsequently, it calls "Hardware\_Init()" to complete hardware initialization, including the initialization of LDO power supply, LCD display, and LoRa module.

Then, it invokes "lvgl\_show\_counter\_label\_init()" to create and display a text label for counting on the LCD.

After that, it uses "xTaskCreatePinnedToCore()" to create two FreeRTOS tasks: "ui\_counter\_task" is used to update the LVGL label displaying the transmission count every second, and "lora\_tx\_task" is used to send LoRa data packets every second. Both tasks have the same priority to maintain synchronization.

Finally, it prints a log indicating that the task creation is completed and synchronous transmission starts.



```
... Hardware_Init() ...
... lvgl_show_counter_label_init() ...
... xTaskCreatePinnedToCore() ...
... ui_counter_task created, starting synchronized transmission ...
```

Finally, let's take a look at the "CMakeLists.txt" file in the main directory.

The role of this CMake configuration is as follows:

- Collect all .c source files in the main/ directory as the source files of the component.
- Register the main component to the ESP-IDF build system, and declare that it depends on the custom component bsp\_wireless and the custom component bsp\_illuminate.

In this way, during the build process, ESP-IDF knows to build bsp\_wireless and bsp\_illuminate first, and then build main.



```
target_link_libraries(main PRIVATE bsp_wireless bsp_illuminate)
```

The above is the main function code for the transmitter. Next, let's take a look at the main function code for the receiver.

This section of code defines several static global variables that are crucial in the LoRa reception program:

- First, **static esp\_ldo\_channel\_handle\_t ldo4 = NULL**; is used to store the handle of ESP32's internal LDO channel 4. This channel is responsible for outputting 3.3V voltage to power peripheral devices such as the display screen or wireless module.
- Next, **static esp\_ldo\_channel\_handle\_t ldo3 = NULL**; defines the handle of LDO channel 3. It outputs 2.5V voltage and is often used to power low-voltage modules (e.g., LoRa RF chips).
- 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.



### **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.

### 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.

### **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.

### **app\_main:**

The function `app_main()` serves as the main entry function of the entire LoRa receiver program, responsible for completing core startup tasks such as system initialization, UI interface configuration, and task creation.

1. At the beginning of the function, it outputs a startup log via `MAIN_INFO("-----  
LoRa RX -----")` to indicate that the system has entered LoRa reception mode.
2. It then calls `Hardware_Init()` to initialize all underlying hardware resources, including power management, SPI communication interfaces, and LoRa modules, laying the foundation for subsequent communication.
3. Subsequently, it executes `lvgl_show_rx_interface_init()` to create and initialize the LVGL graphical interface, which is used to display real-time information such as received messages, RSSI, and SNR on the screen.
4. Next, it calls `sx1262_set_rx_callback(rx_data_callback)` to register a data reception callback function. When the LoRa module receives data, the system will automatically trigger this callback to process and display the information.
5. Finally, it creates an independent task `lora_rx_task` under FreeRTOS through `xTaskCreatePinnedToCore()`, which is pinned to core 1 to continuously monitor LoRa signals, enabling asynchronous data reception and real-time response.

This concludes our explanation of the main function code for both the receiver and transmitter ends.

## Complete Code

---

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

- Transmitting end code:

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14\\_TX\\_SX1262\\_Wireless\\_Module](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14_TX_SX1262_Wireless_Module)

- Receiving end code:

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14\\_RX\\_SX1262\\_Wireless\\_Module](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14_RX_SX1262_Wireless_Module)

## Programming Steps

---

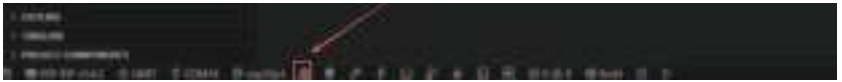
- Now that the code is ready, the next step is to **flash it onto the ESP32-P4** so we can observe the actual operation.
- First, connect the Advance-P4 device to your computer using a USB cable.



- Before starting the flashing preparation, delete all files generated during compilation to restore the project to its initial "unbuilt" state. This ensures that subsequent compilations are not affected by your previous build results.



- Next, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip correctly.
- Then, we need to configure the SDK.
- Click on the icon shown in the figure below.



- Wait for a short loading period, and then you can proceed with the relevant SDK configuration.



- Then, type "flash" into the search box. (Ensure your flash configuration matches mine.)



- After completing the configuration, remember to save your settings.

- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we'd like to share a very convenient feature with you: a single button can execute **compilation, uploading, and serial monitor opening** in one go.



- After waiting for a moment, the code will finish compiling and uploading, and the serial monitor will open automatically.
- 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.



# Lesson 15

## 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.

## Hardware Used in This Lesson

---

### nRF2401 Wireless Module on Advance-P4



## Operation Effect Diagram

---

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.



## Key Explanations

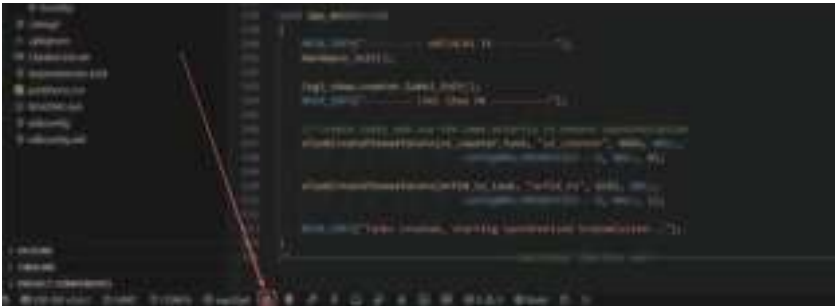
---

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

---

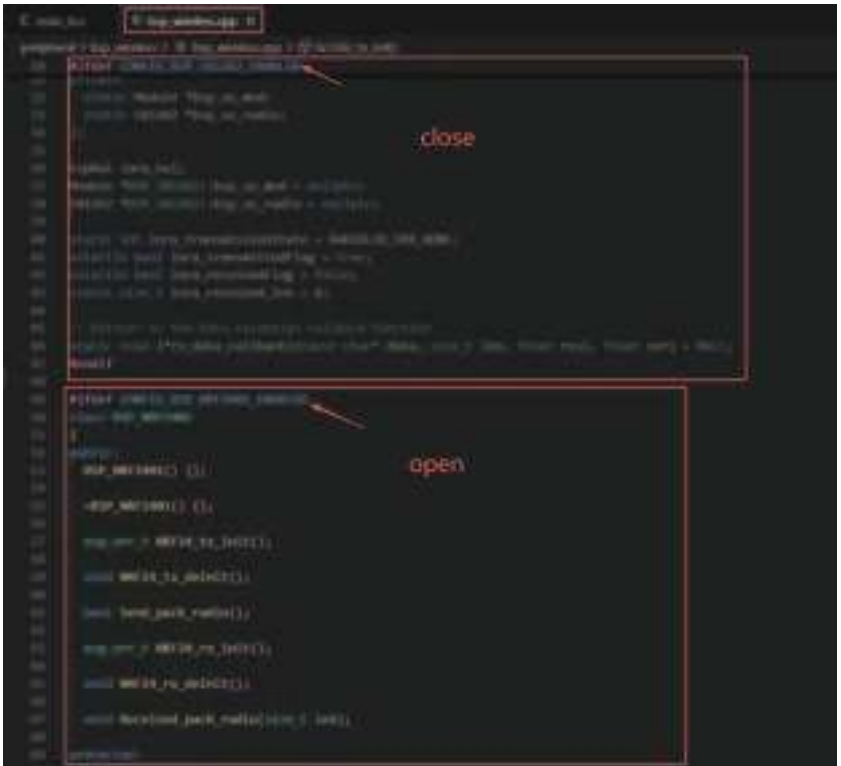
- Click on the SDK configuration.



- Search for "wireless" and open your configuration.



- Since I am using nRF2401 here, I only check "Enable NRF2401 config" and uncheck the others.
- (Enable whichever module you are using.)
- After making changes, click Save to save the configuration.



- 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.
- First, click the GitHub link below to download the code for this lesson.
- Transmitting end code:

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15\\_TX\\_nRF2401\\_Wireless\\_RF\\_Module](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15_TX_nRF2401_Wireless_RF_Module)

- Receiving end code:

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15\\_RX\\_nRF2401\\_Wireless\\_RF\\_Module](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15_RX_nRF2401_Wireless_RF_Module)

- Then, drag the downloaded code into VS Code and open the project files.
- Once opened, you can see the **project structure**:
- This is the **transmitter side**:



- And this is the **receiver side**:



In these two projects, **only the implementations in the main functions `main_tx.c` and `main_rx.c` differ**; all other code files are identical. (For convenience, we have provided **two separate main functions** for use.)

In this lesson's example, under **peripheral \**, a new folder named **bsp\_wireless** is created. Inside the **bsp\_wireless \** folder, there is a new **include** folder and a **CMakeLists.txt** file.

The **bsp\_wireless** folder contains the driver file **bsp\_wireless.cpp**.

The **include** folder contains the header files **bsp\_wireless.h** and **EspHal.h**.

**EspHal.h** converts ESP-IDF C code into the Arduino-style C++ syntax required by the Radiolib component library.

The **CMakeLists.txt** file integrates the driver into the build system, allowing the project to use the nRF2401 module send and receive functions implemented in **bsp\_wireless.cpp**.

Additionally, there is **bsp\_illuminate**, our familiar component used to light up the screen and draw text via LVGL.

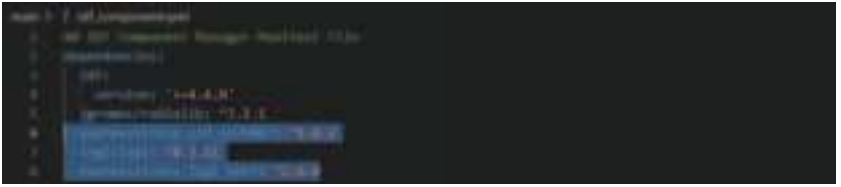
## nRF2401 Communication Code

---

- 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.



- These three components, which we discussed previously and used in the `bsp_illuminate` component, are employed to illuminate the screen and render information on the interface using LVGL.



- During the subsequent compilation process, the project will automatically download the following libraries: `jgromes/radiolib` version 7.2.1, `espressif/esp_lcd_ek79007` version 1.0.2, `lvgl` version 8.3.11, and `espressif/esp_lvgl_port` version 2.6.0. Once downloaded, these networking components will be stored in the `managed_components` folder. (This is automatically generated after specifying the version numbers.)
- Returning to `bsp_wireless.h`, this is where we declare the pins used by the wireless module.



- 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`.)

```

21  #ifdef NRF24_ENABLED
22
23  #define NRF24_TX_PIN 10
24  #define NRF24_RX_PIN 9
25  #define NRF24_ADDR 0x00000000
26
27  #endif
28
29  #include <Arduino.h>
30  #include <SPI.h>
31  #include <NRF24L01.h>
32
33  #define NRF24_TX_PIN 10
34  #define NRF24_RX_PIN 9
35  #define NRF24_ADDR 0x00000000
36
37  #include <Arduino.h>
38  #include <SPI.h>
39  #include <NRF24L01.h>
40
41  #define NRF24_TX_PIN 10
42  #define NRF24_RX_PIN 9
43  #define NRF24_ADDR 0x00000000
44
45  #include <Arduino.h>
46  #include <SPI.h>
47  #include <NRF24L01.h>
48
49  #define NRF24_TX_PIN 10
50  #define NRF24_RX_PIN 9
51  #define NRF24_ADDR 0x00000000
52
53  #include <Arduino.h>
54  #include <SPI.h>
55  #include <NRF24L01.h>
56
57  #define NRF24_TX_PIN 10
58  #define NRF24_RX_PIN 9
59  #define NRF24_ADDR 0x00000000
60
61  #include <Arduino.h>
62  #include <SPI.h>
63  #include <NRF24L01.h>
64
65  #define NRF24_TX_PIN 10
66  #define NRF24_RX_PIN 9
67  #define NRF24_ADDR 0x00000000
68
69  #include <Arduino.h>
70  #include <SPI.h>
71  #include <NRF24L01.h>
72
73  #define NRF24_TX_PIN 10
74  #define NRF24_RX_PIN 9
75  #define NRF24_ADDR 0x00000000
76
77  #include <Arduino.h>
78  #include <SPI.h>
79  #include <NRF24L01.h>
80
81  #define NRF24_TX_PIN 10
82  #define NRF24_RX_PIN 9
83  #define NRF24_ADDR 0x00000000
84
85  #include <Arduino.h>
86  #include <SPI.h>
87  #include <NRF24L01.h>
88
89  #define NRF24_TX_PIN 10
90  #define NRF24_RX_PIN 9
91  #define NRF24_ADDR 0x00000000
92
93  #include <Arduino.h>
94  #include <SPI.h>
95  #include <NRF24L01.h>
96
97  #define NRF24_TX_PIN 10
98  #define NRF24_RX_PIN 9
99  #define NRF24_ADDR 0x00000000
100

```

- 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.

```

10 #ifndef NRF2401_H
11 #define NRF2401_H
12
13 #include "nrf_hal.h"
14
15 #define NRF2401_TX_BUFFER_SIZE 32
16 #define NRF2401_RX_BUFFER_SIZE 32
17
18 typedef struct {
19     uint8_t *tx_buffer;
20     uint8_t *rx_buffer;
21     uint8_t tx_counter;
22     bool transmittedFlag;
23     bool receivedFlag;
24 } nrf2401_t;
25
26 extern nrf2401_t nrf2401;
27
28 void nrf2401_init();
29 void nrf2401_tx_init();
30 void nrf2401_tx_deinit();
31 void nrf2401_rx_init();
32 void nrf2401_rx_deinit();
33 void nrf2401_send_packet(uint8_t *data, uint8_t len);
34 void nrf2401_receive_packet(uint8_t *data, uint8_t len);
35
36 #endif
37
38 #include "nrf2401.h"
39
40 void nrf2401_init() {
41     nrf2401.tx_buffer = (uint8_t *) malloc(NRF2401_TX_BUFFER_SIZE);
42     nrf2401.rx_buffer = (uint8_t *) malloc(NRF2401_RX_BUFFER_SIZE);
43     nrf2401.tx_counter = 0;
44     nrf2401.transmittedFlag = false;
45     nrf2401.receivedFlag = false;
46 }
47
48 void nrf2401_tx_init() {
49     nrf2401.tx_buffer = (uint8_t *) malloc(NRF2401_TX_BUFFER_SIZE);
50 }
51
52 void nrf2401_tx_deinit() {
53     free(nrf2401.tx_buffer);
54 }
55
56 void nrf2401_rx_init() {
57     nrf2401.rx_buffer = (uint8_t *) malloc(NRF2401_RX_BUFFER_SIZE);
58 }
59
60 void nrf2401_rx_deinit() {
61     free(nrf2401.rx_buffer);
62 }
63
64 void nrf2401_send_packet(uint8_t *data, uint8_t len) {
65     if (len > NRF2401_TX_BUFFER_SIZE) {
66         return;
67     }
68     memcpy(nrf2401.tx_buffer, data, len);
69     nrf2401.tx_counter++;
70     nrf2401.transmittedFlag = true;
71 }
72
73 void nrf2401_receive_packet(uint8_t *data, uint8_t len) {
74     if (len > NRF2401_RX_BUFFER_SIZE) {
75         return;
76     }
77     memcpy(data, nrf2401.rx_buffer, len);
78     nrf2401.receivedFlag = true;
79     if (nrf2401_rx_data_callback != NULL) {
80         nrf2401_rx_data_callback(data, len);
81     }
82 }

```

## 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.

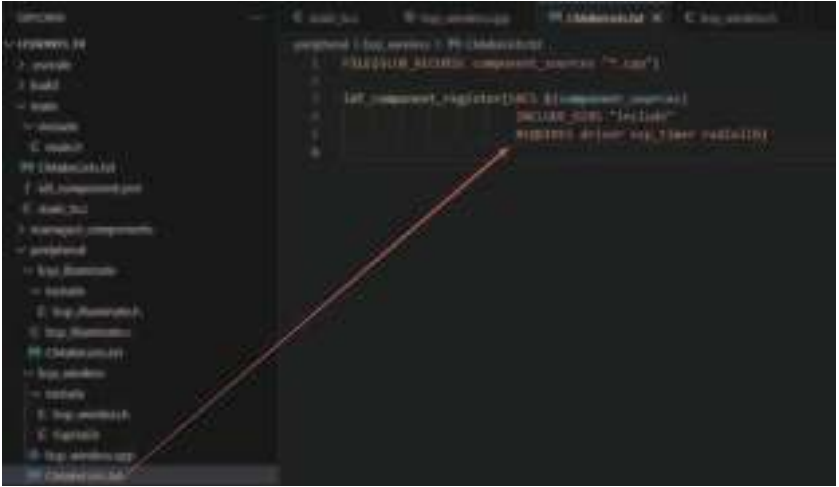
## nrf24\_set\_rx\_callback(void (\*callback)(const char\* data, size\_t len))

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 it, you also need to configure the **CMakeLists.txt** file under the `bsp_wireless` folder. This file, located in the `bsp_wireless` directory, primarily tells the ESP-IDF build system (CMake) how to compile and register the `bsp_wireless` component.



The reason only `driver`, `esp_timer`, and `radiolib` are listed is that we use them in `bsp_wireless.h` and `bsp_wireless.cpp`. (Other libraries are system libraries, so they do not need to be added.)



```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>
#include <esp_timer.h>
#include <string.h>
```

As well as `esp_timer`, which is used in the `EspHdl.h` file.

```
#ifndef ESP_HDL_H
#define ESP_HDL_H

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>
#include <esp_timer.h>
#include <string.h>

class EspHdl {
public:
    EspHdl();
    ~EspHdl();
    void setup();
    void loop();
};

#endif
```

## Main function

The main folder is the core directory for program execution and contains the main executable file `main_tx.c`.

The main folder should be added to the build system in the `CMakeLists.txt` file.

```
cmake_minimum_required(VERSION 3.10)
project("ESP8266")

set(CMAKE_CXX_STANDARD 11)

include_directories(
    ..
)

add_executable(main_tx main_tx.c)
```

This is the entry file of the entire application. In ESP-IDF, there is no `int main()`; instead, execution starts from `void app_main(void)`.

Let's first explain the transmitter's main function file, `main_tx.c`, to see how it calls the interfaces to send information via the nRF24L01.

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

- First, `Hardware_Init()` is called in `app_main()` to initialize the hardware. This includes configuring the LDO power channels, initializing the LCD display and turning on the backlight, and initializing the nRF24L01 wireless module.
- Next, `lvgl_show_counter_label_init()` is called to create and display an LVGL label on the screen for showing the transmit counter.
- The program then creates two FreeRTOS tasks:
  - `ui_counter_task` reads the nRF24L01 transmit counter every second and updates the screen label.
  - `nrf24_tx_task` increments the transmit counter every second and calls `send_nrf24_pack_radio()` to send a wireless data packet, achieving wireless transmission.

The entire process uses task scheduling to keep the display and transmission synchronized, forming a loop system that automatically sends data every second while showing the real-time count on the LCD.

Next, let's explain the main code in `main_tx.c`.

```
#include <stdio.h> // printf, fprintf, sprintf
#include <string.h> // strlen, strcpy, strncpy, strcmp, strncmp, strcat, strncat
#include <math.h> // sqrt, pow, exp, log, exp2, log2, expf, logf, exp2f, log2f, expf, logf, exp2f, log2f
#include <stdint.h> // uint8_t, uint16_t, uint32_t, int8_t, int16_t, int32_t, float, double, long double
```

Includes the custom main header file `main.h`, which typically contains logging macros, declarations for peripheral initialization, and other interface header files that need to be used.

Below is the content of `main.h`:

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdint.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/queue.h>
#include <sys/eventfd.h>
#include <sys/signalfd.h>
#include <sys/timerfd.h>
#include <sys/uio.h>
#include <sys/random.h>
#include <sys/epoll.h>
#include <sys/prctl.h>
#include <sys/personality.h>
#include <sys/xattr.h>
#include <sys/fsuid.h>
#include <sys/fsverity.h>
#include <sys/auxv.h>
#include <sys/swap.h>
#include <sys/swapfile.h>
#include <sys/mount.h>
#include <sys/procfs.h>
#include <sys/syscall.h>
#include <sys/sysinfo.h>
#include <sys/sysmacros.h>
#include <sys/sysmips.h>
#include <sys/sysmips2.h>
#include <sys/sysmips3.h>
#include <sys/sysmips4.h>
#include <sys/sysmips5.h>
#include <sys/sysmips6.h>
#include <sys/sysmips7.h>
#include <sys/sysmips8.h>
#include <sys/sysmips9.h>
#include <sys/sysmips10.h>
#include <sys/sysmips11.h>
#include <sys/sysmips12.h>
#include <sys/sysmips13.h>
#include <sys/sysmips14.h>
#include <sys/sysmips15.h>
#include <sys/sysmips16.h>
#include <sys/sysmips17.h>
#include <sys/sysmips18.h>
#include <sys/sysmips19.h>
#include <sys/sysmips20.h>
#include <sys/sysmips21.h>
#include <sys/sysmips22.h>
#include <sys/sysmips23.h>
#include <sys/sysmips24.h>
#include <sys/sysmips25.h>
#include <sys/sysmips26.h>
#include <sys/sysmips27.h>
#include <sys/sysmips28.h>
#include <sys/sysmips29.h>
#include <sys/sysmips30.h>
#include <sys/sysmips31.h>
#include <sys/sysmips32.h>
#include <sys/sysmips33.h>
#include <sys/sysmips34.h>
#include <sys/sysmips35.h>
#include <sys/sysmips36.h>
#include <sys/sysmips37.h>
#include <sys/sysmips38.h>
#include <sys/sysmips39.h>
#include <sys/sysmips40.h>
#include <sys/sysmips41.h>
#include <sys/sysmips42.h>
#include <sys/sysmips43.h>
#include <sys/sysmips44.h>
#include <sys/sysmips45.h>
#include <sys/sysmips46.h>
#include <sys/sysmips47.h>
#include <sys/sysmips48.h>
#include <sys/sysmips49.h>
#include <sys/sysmips50.h>
#include <sys/sysmips51.h>
#include <sys/sysmips52.h>
#include <sys/sysmips53.h>
#include <sys/sysmips54.h>
#include <sys/sysmips55.h>
#include <sys/sysmips56.h>
#include <sys/sysmips57.h>
#include <sys/sysmips58.h>
#include <sys/sysmips59.h>
#include <sys/sysmips60.h>
#include <sys/sysmips61.h>
#include <sys/sysmips62.h>
#include <sys/sysmips63.h>
#include <sys/sysmips64.h>
#include <sys/sysmips65.h>
#include <sys/sysmips66.h>
#include <sys/sysmips67.h>
#include <sys/sysmips68.h>
#include <sys/sysmips69.h>
#include <sys/sysmips70.h>
#include <sys/sysmips71.h>
#include <sys/sysmips72.h>
#include <sys/sysmips73.h>
#include <sys/sysmips74.h>
#include <sys/sysmips75.h>
#include <sys/sysmips76.h>
#include <sys/sysmips77.h>
#include <sys/sysmips78.h>
#include <sys/sysmips79.h>
#include <sys/sysmips80.h>
#include <sys/sysmips81.h>
#include <sys/sysmips82.h>
#include <sys/sysmips83.h>
#include <sys/sysmips84.h>
#include <sys/sysmips85.h>
#include <sys/sysmips86.h>
#include <sys/sysmips87.h>
#include <sys/sysmips88.h>
#include <sys/sysmips89.h>
#include <sys/sysmips90.h>
#include <sys/sysmips91.h>
#include <sys/sysmips92.h>
#include <sys/sysmips93.h>
#include <sys/sysmips94.h>
#include <sys/sysmips95.h>
#include <sys/sysmips96.h>
#include <sys/sysmips97.h>
#include <sys/sysmips98.h>
#include <sys/sysmips99.h>
#include <sys/sysmips100.h>
```

Let's continue to look at the contents of `main_tx.c`.

## 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:

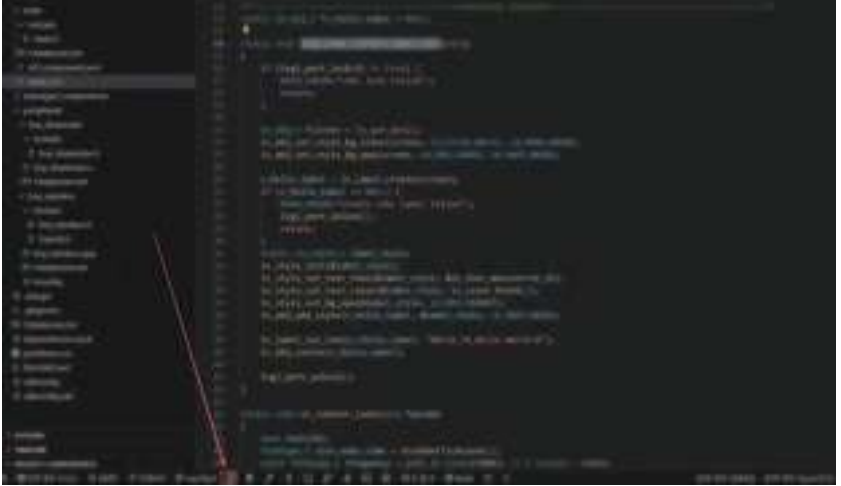
- First, `lvgl_port_lock(0)` is called to lock LVGL resources, preventing concurrent access.
- The current active screen is obtained via `lv_scr_act()`, and the background is set to white and fully covering.
- A label is created using `lv_label_create(screen)` and checked for successful creation; if 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.

```
12 void lvgl_show_counter_label_init(void)
13 {
14     if (!lvgl_port_lock(0)) {
15         lvgl_printf("lvgl: lock failed");
16         return;
17     }
18     lv_obj_t * screen = lv_scr_act();
19     lv_obj_set_style_bg_color(screen, lv_color_black(), LV_PART_MAIN);
20     lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);
21
22     lv_obj_t * label = lv_label_create(screen);
23     if (!lv_obj_t * label == NULL) {
24         lvgl_printf("lvgl: label failed");
25         lvgl_port_unlock();
26         return;
27     }
28     lv_obj_set_style_text_color(label, lv_color_black(), 0);
29     lv_obj_set_style_text_font(label, lv_obj_get_style_text_font(screen), 0);
30     lv_obj_set_style_text_opa(label, lv_obj_get_style_text_opa(screen), 0);
31     lv_obj_set_style_bg_color(label, lv_color_black(), 0);
32     lv_obj_set_style_bg_opa(label, LV_OPA_TRANSP, 0);
33     lv_label_set_text(label, "nRF24_TX_Hello World:0");
34     lv_obj_center(label);
35
36     lvgl_port_unlock();
37 }
```

If you want to change the LVGL font size, you need to enable the fonts in the SDK configuration.

**Steps:** Click on the SDK configuration options



Search for **font**, then select the font size you want to use. After making the changes, be sure to save them.



## 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 `sprintf`.
- 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.

## Hardware\_Init:

The function `Hardware_Init()` is used to initialize hardware modules when the program starts, ensuring all parts of the system can work properly.

- First, it acquires the LDO3 (2.5V) and LDO4 (3.3V) power channels respectively through `esp_ldo_acquire_channel()`. If the acquisition fails, it calls `init_or_halt()` to wait in a loop and print error messages, ensuring stable power supply.
- Then it calls `display_init()` to initialize the LCD hardware and LVGL graphics library, which must be completed before turning on the backlight; otherwise, the display may work abnormally.
- Next, it calls `set_lcd_blight(100)` to turn on the LCD backlight and set the maximum brightness to 100, with errors also checked via `init_or_halt()`.
- Finally, it calls `nrf24_tx_init()` to initialize the nRF2401 wireless transmission module. If initialization fails, it is also handled through `init_or_halt()`.

Overall, its role is to provide a reliable hardware environment for the screen display, backlight, and wireless communication module, ensuring the subsequent functions of the program can run smoothly. It is usually called during system startup in `app_main()`.

## 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.

## app\_main:

`app_main()` is the program entry function, responsible for completing hardware initialization, interface display setup, and launching wireless transmission and interface refresh tasks to implement the synchronized transmission and display functions of the nRF24L01.

The specific workflow is summarized as follows:

- First, it prints the log "----- nRF24L01 TX -----" to indicate program startup.
- It calls `Hardware_Init()` to initialize hardware, including LDO power supplies, LCD display, and the nRF24L01 module.
- It invokes `lvgl_show_counter_label_init()` to create and initialize an LVGL label for displaying the transmission count, and prints the log "----- LVGL Show OK -----".
- Then, it uses `xTaskCreatePinnedToCore` to create two FreeRTOS tasks: `ui_counter_task` (for refreshing the transmission count display on the LCD every second) and `nrf24_tx_task` (for transmitting wireless data packets once per second). Both tasks use the same priority to maintain synchronization.
- Finally, it prints the log "Tasks created, starting synchronized transmission..." to indicate that task creation is complete and the system has started synchronized transmission and interface display.

```
void app_main(void)
{
    esp_log_start("ESP32", LOG_INFO);
    esp_log_start("ESP32", LOG_INFO);

    esp_log_start("ESP32", LOG_INFO);
    esp_log_start("ESP32", LOG_INFO);

    // Create a task and use the task priority to ensure communications
    // start/restart immediately when task is created, esp_log_start("ESP32", LOG_INFO);
    // esp_log_start("ESP32", LOG_INFO);

    // esp_log_start("ESP32", LOG_INFO);
    // esp_log_start("ESP32", LOG_INFO);

    esp_log_start("ESP32", LOG_INFO);
    esp_log_start("ESP32", LOG_INFO);
}
```

Finally, let's take a look at the "CMakeLists.txt" file in the main directory.

The role of this CMake configuration is as follows:

- Collect all .c source files in the main/ directory as the source files of the component.
- Register the main component with the ESP-IDF build system, and declare that it depends on the custom component bsp\_wireless and the custom component bsp\_illuminate.

This ensures that during the build process, ESP-IDF knows to build bsp\_wireless and bsp\_illuminate first, and then build the main component.

```

cmake_minimum_required(VERSION 3.10)
project("main")

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

add_compile_options(-std=c++11)

add_executable(main main.cpp)

target_compile_options(main PRIVATE -std=c++11)

target_link_libraries(main PRIVATE esp32)

```

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.

## **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.

## Hardware\_Init:

This function is identical to the hardware initialization function described earlier. It initializes the LDOs, screen, and nRF2401 module in the same way. The only difference here is that the nRF2401 module is configured in **receiver mode**.

## 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.

## app\_main:

- `app_main()` is the entry function of the nRF24L01 receiver program, used to initialize hardware, the interface, and reception tasks.
- First, the function prints startup information via `MAIN_INFO`, then calls `Hardware_Init()` to initialize hardware peripherals (such as power management, LCD, and the nRF24L01 module).
- Next, it invokes `lvgl_show_rx_interface_init()` to initialize the LVGL display interface and prints a confirmation log.
- Subsequently, it registers the reception callback function using `nrf24_set_rx_callback(rx_data_callback)`—this function is used to process data and update the interface when data is received, and a log is printed for confirmation.
- Finally, it creates the FreeRTOS task `nrf24_rx_task` using `xTaskCreatePinnedToCore()`, which continuously polls for and receives data on the specified core. A log is printed to indicate that the receiver has started.
- This concludes our explanation of the main function code for both the receiver and transmitter of the nRF24L01.

We have now finished explaining the main function code for both the receiver and the transmitter.

## Complete Code

---

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

- Transmitting end code:

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15\\_TX\\_nRF2401\\_Wireless\\_RF\\_Module](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15_TX_nRF2401_Wireless_RF_Module)

- Receiving end code:

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15\\_RX\\_nRF2401\\_Wireless\\_RF\\_Module](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15_RX_nRF2401_Wireless_RF_Module)

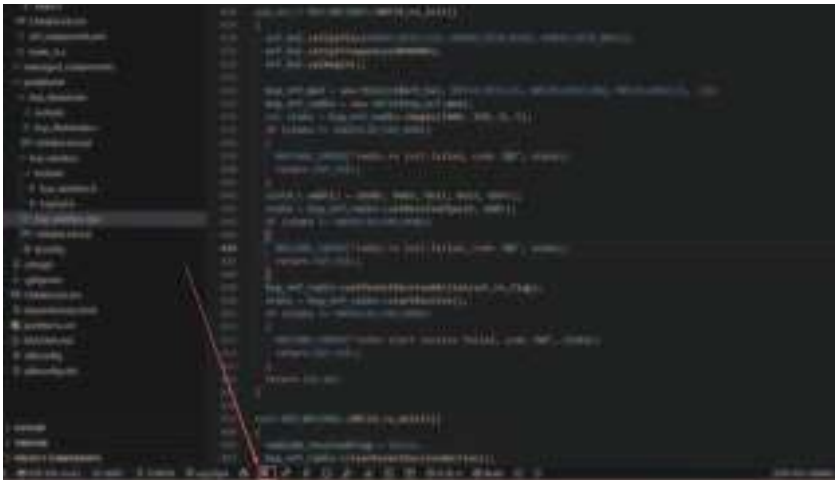
## Programming Steps

---

- Now that the code is ready, the next step is to flash it onto the ESP32-P4 so we can observe the actual operation.
- First, connect the Advance-P4 device to your computer using a USB cable.



- Before starting the preparation for flashing, first delete all compiler-generated files to restore the project to its initial "unbuilt" state. This ensures that subsequent compilations are not affected by your previous operations.



- Here, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon shown in the figure below.



- After waiting for a short loading period, you can proceed with the relevant SDK configurations.



- Then, enter "flash" in the search box to search. (Make sure your flash configuration matches mine.)



- After completing the configuration, remember to save your settings.
- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we also want to share a very convenient feature with you: a single button that can execute compilation, upload, and open the monitor **in one go**.



- After waiting for a moment, the code will finish compiling and uploading, and the monitor will open automatically.
- 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.



# Lesson 16

## Get weather temperature via WiFi

### Introduction

---

In this class, we will learn a new component - the Wi-Fi component. We will also use the display-related components we learned before to achieve Wi-Fi connection together, and then obtain the weather conditions and temperature of your local area, and display the relevant information on the Advance-P4.

### Hardware Used in This Lesson

---

#### The ESP32-C6 Wi-Fi module on the Advance-P4



The ESP32-P4 does not integrate Wi-Fi wireless radio and MAC/PHY modules by itself. Its positioning is as a high-performance application processor, mainly responsible for graphic display, audio and video, multimedia, human-computer interaction, and complex business logic, rather than wireless communication. Therefore, when ESP32-P4 needs to connect to the network, the Wi-Fi function must be realized through an external network coprocessor.

The ESP32-C6 is a complete Wi-Fi 6 + Bluetooth + 802.15.4 wireless SoC that has a mature and stable network protocol stack and radio capabilities. In the actual solution, ESP32-P4 communicates with ESP32-C6 through interfaces such as SPI, UART, and SDIO. The C6 is responsible for the underlying network work such as Wi-Fi scanning, authentication, encryption, and TCP/IP protocol processing, while P4 only needs to send network requests and receive data through the SDIO communication protocol.

This "master control + wireless coprocessor" architecture not only reduces the power consumption and design complexity of P4, but also improves the reliability and scalability of the system. It is a typical design concept adopted by Espressif in high-end human-computer interaction and edge computing products.

## Operation Effect Diagram

---

After running the code, you will be able to see the relevant information about the local weather on the screen of the Advance-P4.



## Key Explanations

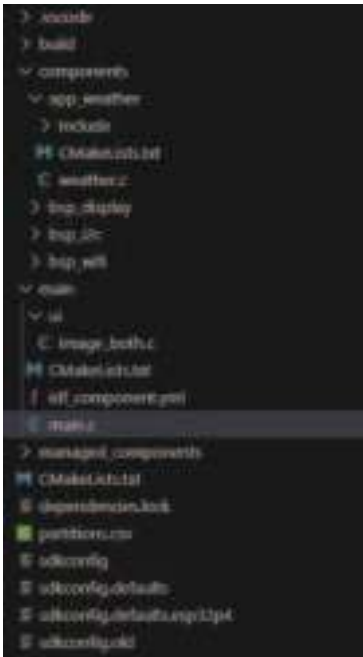
---

- The main focus of this class is on how to connect to Wi-Fi and how to use Wi-Fi to connect to the network to obtain weather information.
- Here, we will prepare a new component called "bsp\_wifi" for you.

- The function of the "bsp\_wifi" component is to encapsulate and manage the native Wi-Fi functionality of ESP-IDF at the board level, and complete the initialization of the Wi-Fi system, creation of STA/AP modes, network interface management, event callback processing, connection and reconnection after disconnection, as well as maintenance of the network connection status. It shields the complex Wi-Fi underlying details from the upper-layer business and only provides "initialization, connection, disconnection, status query" and other capabilities through simple interfaces, so that the application layer can use the network function stably, clearly, and flexibly, without having to directly operate esp\_wifi, esp\_netif and the event system.
- As you all know, you can simply call the interfaces that we have written at the appropriate time.
- Next, we will focus on understanding the bsp\_wifi component.
- First, click on the Github link below to download the code for this lesson.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson16\\_Get%20weather%20via%20WiFi](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson16_Get%20weather%20via%20WiFi)

- Then, drag the code of this lesson into VS Code and open the project file.
- After opening it, you can see the framework of this project.



In the example of this class, a new folder named "bsp\_wifi" was created under the "components\" directory.

Inside the "bsp\_wifi\" folder, a new "include" folder and a "CMakeLists.txt" file were created.

The "bsp\_wifi" folder contains the "bsp\_wifi.c" driver file, and the "include" folder contains the "bsp\_wifi.h" header file.

The "CMakeLists.txt" file will integrate the driver into the build system, enabling the project to utilize the relevant functions for Wi-Fi connection as described in "bsp\_wifi.c".

## Wifi Connection Code

- The code for Wi-Fi connection consists of two files: "bsp\_wifi.c" and "bsp\_wifi.h".
- Next, we will first analyze the "bsp\_wifi.h" program.
- "bsp\_wifi.h" is the header file for the Wi-Fi connection code, mainly used for:
  - Making the functions, macros, and variables implemented in "bsp\_wifi.c" available for use by external programs,
  - Allowing other .c files to simply include "bsp\_wifi.h" to call this module.
- In other words, it is the interface layer, exposing which functions and constants can be used externally while hiding the internal details of the module.
- In this component, all the libraries we need to use are placed in the "bsp\_wifi.h" file for centralized management.

```
4 #include <string.h>
5 #include "freertos/FreeRTOS.h"
6 #include "freertos/task.h"
7
8 #include "esp_wifi.h"
9 #include "esp_event.h"
10 #include "esp_netif.h"
11 #include "esp_mac.h"
12
13 #include "esp_log.h"
14 #include "esp_flash.h"
15
```

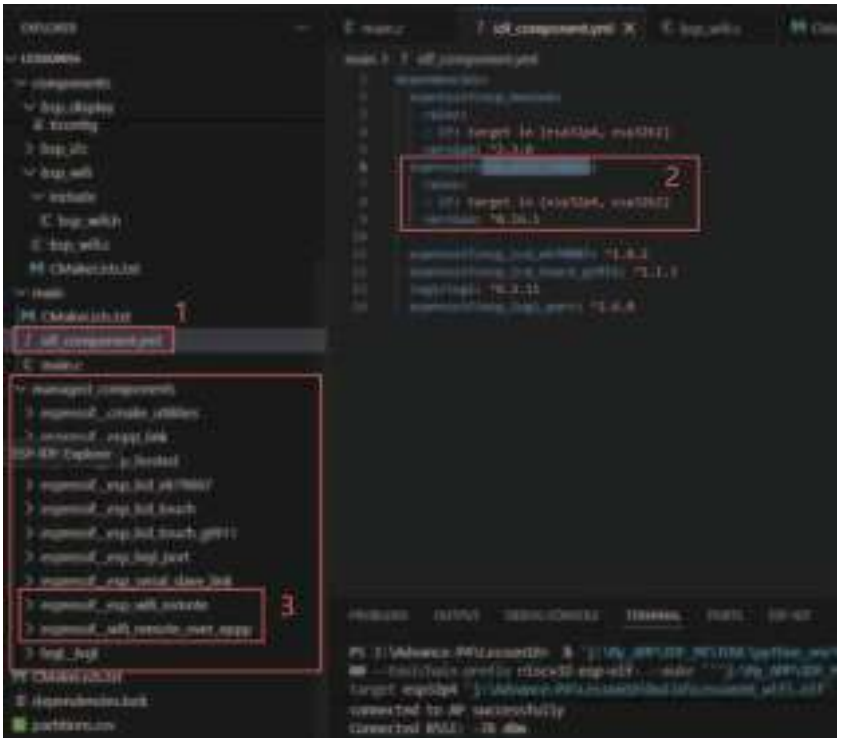
- Such as "esp\_wifi.h" and so on (these are libraries under the network component)

```
components > bsp_wifi > include > C:\bsp_wifi > ...
1 #ifndef __ESP_WIFI_H_
2 #define __ESP_WIFI_H_
3
4 #include <string.h>
5 #include "freertos/FreeRTOS.h"
6 #include "freertos/task.h"
7
8 #include "esp_wifi.h"
9 #include "esp_event.h"
10 #include "esp_netif.h"
11 #include "esp_mac.h"
12
13 #include "esp_log.h"
14 #include "esp_flash.h"
15
```

- Used in bsp\_wifi.c:

```
wifi_usertag_t  
wifi_init_config_t  
wifi_ap_params_t  
  
esp_wifi_init()  
esp_wifi_set_mode()  
esp_wifi_set_config()  
esp_wifi_start()  
esp_wifi_stop()  
esp_wifi_disconnect()  
esp_wifi_disconnect()
```

- In this case, we need to fill in the version of esp\_wifi\_remote in the idf\_component.yml file located in the main folder.
- Since this is an official library, we need to use the official library to implement the wifi-related functions on our Advance-P4.





## 2. EM\_WIFI\_STATE (Wi-Fi connection status)

EM\_WIFI\_STATE is used to describe the actual networking status of the device in the STA mode. It focuses on the most critical issue at the business layer - "Is the current network available?".

This enumeration fully expresses the entire process from not connected, attempting to connect, successfully connecting to the network, to abnormal disconnection through states such as IDLE, CONNECTING, CONNECTED, and DISCONNECTED.

In implementation, this status is not solely based on "whether connected to the router", but rather takes successfully obtaining an IP address as the criterion for truly successful networking, ensuring the availability of the network both logically and functionally.

The status is provided externally through `bsp_wifi_get_state()`, allowing the application layer to safely decide whether to execute HTTP requests, time synchronization, or other network business logic without caring about the details of the underlying events.

Let's take a look at "bsp\_wifi.c" again. We'll discuss the specific functions of each function in this component.

`bsp_wifi`: This component provides significant support for the subsequent use of the Wi-Fi connection function. By understanding the functions clearly, you can easily utilize the Wi-Fi functionality.

Here, we won't go into detail about the code. We will only tell you what each function does and under what circumstances it should be called.

### 1. `void wifi_event_handler(void* arg, esp_event_base_t event_base, int32_t event_id, void* event_data)`

This is the "event hub" of the entire WiFi module, responsible for receiving and processing various asynchronous events sent by the ESP-IDF WiFi and IP subsystems.

For example: WiFi startup, successful connection, disconnection, AP having devices connected, STA obtaining IP, etc.

Inside the function, the event type is determined through `event_base` and `event_id`, and depending on the situation, it updates `wifi_sta_state_`, prints logs, or automatically reconnects WiFi.

It is not manually called by you; instead, it is registered using `esp_event_handler_register()`, and the system will automatically call it when an event occurs. It is a typical "passive execution function".

#### Call scenarios:

Just register it once during the WiFi initialization stage, and it will be automatically triggered throughout the entire WiFi lifecycle. The application layer does not need to care about when to call it.

## 2. void bsp\_wifi\_connect(char\* ssid, char\* password)

The function's purpose is to actively initiate a STA mode WiFi connection.

It will first stop the current WiFi, then configure the SSID, password and security mode of the STA, then start the WiFi and call `esp_wifi_connect()` to initiate the connection request, while setting `wifi_sta_state_` to `WIFI_CONNECTING`.

The actual connection status and whether an IP address has been obtained are not judged here; instead, they are handled by the `wifi_event_handler` for subsequent events.

### Call scenario:

When you have initialized WiFi (`bsp_wifi_init + bsp_wifi_sta_init`) and the user has entered WiFi information or you want to switch networks, call this function.

## 3. void bsp\_wifi\_disconnect(void)

This is a function that actively disconnects the WiFi connection and stops the WiFi module. It first calls `esp_wifi_disconnect()` to break the current connection, and then calls `esp_wifi_stop()` to stop the WiFi driver. This function does not release the network interface or reinitialize WiFi; it merely stops the WiFi.

### Call scenarios:

This function is called when you need to temporarily disconnect from the network, enter a low-power mode, switch WiFi modes (such as STA → AP), or when the user manually clicks the "Disconnect WiFi" button.

## 4. void bsp\_wifi\_init(void)

This is the "overall initialization entry point" of the WiFi subsystem, responsible for all one-time, low-level initialization tasks. It initializes NVS (which the WiFi must use), the network stack (`esp_netif_init`), the event loop (`esp_event_loop_create_default`), and the WiFi driver itself (`esp_wifi_init`). This function only performs initialization and does not start the WiFi or connect to the network.

### Call scenario:

In `app_main()`, this function is called only once throughout the entire program lifecycle, and it must be called before any WiFi-related operations.

## 5. void bsp\_wifi\_sta\_init(void)

`bsp_wifi_init() → bsp_wifi_sta_init() → bsp_wifi_connect()`

This function is used to configure WiFi to the STA (client) mode and start it.

It creates the default STA network interface, registers WiFi and IP event callbacks, sets the WiFi working mode to STA, and starts WiFi.

Note: It does not automatically connect to the router; it merely puts the device into a state where it can connect to WiFi.

#### Calling scenario:

This function is called when you want the ESP32 to act as a client to connect to the router. The usual sequence is: `bsp_wifi_init()` → `bsp_wifi_sta_init()` → `bsp_wifi_connect()`

### 6. void bsp\_wifi\_ap\_init(void)

This function is used to configure the ESP32 to operate in AP (hotspot) mode. It creates the AP network interface, registers WiFi events, sets the AP's SSID, password, channel, maximum connection number and other parameters, then starts WiFi, making the ESP32 a hotspot that can be connected to by other devices. After success, it prints the account information of the AP.

#### Calling scenario:

This function should be called when you want the ESP32 to act as a hotspot (such as in pairing mode, local network control, or communication without a router). Generally, `bsp_wifi_connect()` is not called anymore.

### 7. EM\_WIFI\_STATE bsp\_wifi\_get\_state(void)

This is a status query function, used to return the current connection status of the STA (such as `WIFI_IDLE`, `WIFI_CONNECTING`, `WIFI_CONNECTED`, `WIFI_DISCONNECTED`) to the external module.

It does not perform any operations on its own; it merely reads the internal maintained status variables.

#### Call scenarios:

For occasions such as UI refresh, business logic judgment (such as "request the weather after being connected to the network"), task state machine judgment, etc., to determine whether the WiFi has truly connected to the network.

### 8. void bsp\_wifi\_ap\_sta\_deinit(void)

This is the complete de-initialization function for WiFi STA/AP. It will stop WiFi, cancel event callbacks, destroy the network interface object, and restore the running state to idle. This function is used to completely exit the current WiFi mode, rather than simply disconnecting from the network.

#### Call scenarios:

This function is called when you need to completely disable the WiFi function, switch to STA/AP mode again, or prepare to reinitialize the entire WiFi subsystem.

That's all about the components of `bsp_wifi`. Just make sure you know how to call these interfaces.

Then, if we need to make a call, we must also configure the "CMakeLists.txt" file located in the "bsp\_wifi" folder.

This file is placed in the "bsp\_wifi" folder and its main function is to inform the build system (CMake) of ESP-IDF: how to compile and register the "bsp\_wifi" component.



The reason why it is named as `nvs_flash` and `esp_wifi` is that we called them in the "bsp\_wifi.h" file (for other libraries that are system libraries, there is no need to add them)



## app\_weather component

Now that the components with Wi-Fi connection are available, how can we obtain the local weather conditions?

Here we provide an app called "app\_weather" for obtaining weather information.



It's the same. Let's first take a look at the content of the weather.h file.

```
components > app-weather > include > C-weather > weather.h [weather_created]
1 #ifndef WEATHER_H
2 #define WEATHER_H
3
4 #include <string.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h> // for sockaddr_in
8 #include <netdb.h> // for host type
9
10
11 // Define 200 bytes size for IP.
12 #define MAX_IP_LEN 200
13 #define WEATHER_PORT 8080 // "http://localhost:8080/~/app-weather/"
14
15 // "Server" weather by C language
16 typedef struct {
17     char *ip_address; // IP address to share 200 characters
18 } weather_t;
19
20 //
21 // Server create and initialize a weather struct instance
22 // Return weather_t* returns a pointer to the instance as success, NULL as failure
23 //
24 weather_t* weather_create(int x);
25
26 //
27 // Server destroy resources allocated by the weather struct instance
28 // Return weather_t struct for the instance to be destroyed
29 //
30 void weather_destroy(weather_t* weather);
31
32 //
33 // Server send information to get weather information
34 // Return weather struct pointer
35 // Return char * ip address and port number (string)
36 // Return weather_t struct weather struct pointer (struct)
37 // Return struct weather struct pointer (struct)
38 // Return host struct type (1) as success, NULL (0) as failure
39 //
40 void weather_get_weather(struct sockaddr_in *server, struct hostent *h, struct weather_t* w);
```

Let me focus on this particular URL link.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/time.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 // Define the server address and port
10 #define SERVER_IP "192.168.1.100"
11 #define SERVER_PORT 8080
12
13 // Function to get the local IP address
14 char* get_local_ip() {
15     struct ifaddrs *ifaddrs;
16     struct ifaddrs *ifa;
17     struct sockaddr_in *sin;
18     char ip[INET_ADDRSTRLEN];
19     for (ifa = ifaddrs; ifa != NULL; ifa = ifa->ifa_next) {
20         if (ifa->ifa_flags & IFF_UP & IFF_RUNNING) {
21             if (ifa->ifa_addr->sa_family == AF_INET) {
22                 sin = (struct sockaddr_in *) ifa->ifa_addr;
23                 inet_ntop(sin->sin_addr.s_addr, ip, INET_ADDRSTRLEN, 0);
24                 return ip;
25             }
26         }
27     }
28     return NULL;
29 }
30
31 // Function to get the local time stamp
32 time_t get_local_time() {
33     return time(NULL);
34 }
35
36 // Function to get the local weather conditions
37 char* get_local_weather() {
38     // This function would typically use a weather API to fetch
39     // the local weather conditions based on the local IP address.
40     return "Local weather conditions";
41 }
42
43 // Main function
44 int main() {
45     char local_ip[INET_ADDRSTRLEN];
46     time_t local_time;
47     char* local_weather;
48     local_ip = get_local_ip();
49     local_time = get_local_time();
50     local_weather = get_local_weather();
51     printf("Local IP: %s, Local Time: %s, Local Weather: %s\n", local_ip, local_time, local_weather);
52     return 0;
53 }
```

This link is the server address of our Elecrow company. You can set this link as the fixed URL for accessing the weather information.

Because no matter where you are, when you connect to a Wi-Fi network and access this URL link to obtain the local weather conditions, our URL will, based on the IP of the Wi-Fi network you are connected to, obtain the weather conditions and local time stamp of the location corresponding to your IP.

This is very convenient for you to use. Just directly access this URL.

You can open this URL link to have a look:



(This URL link does not support a large number of accesses)

What we have is a JSON code. You can extract the information you need from it. We will also explain it next.

Then, like the function names listed below, we will proceed to explain in detail in the weather.c file. The declaration in weather.h is to provide an interface that is convenient for use in the .c file.

```
1 // weather module for C language
2 #include <stdio.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7 #include <sys/socket.h>
8 #include <netinet/in.h>
9 #include <arpa/inet.h>
10 #include <netdb.h>
11 #include <curl/curl.h>
12 #include <curl/easy.h>
13 #include <curl/multi.h>
14 #include <curl/types.h>
15 #include <curl/version.h>
16 #include <curl/system.h>
17 #include <curl/curlver.h>
18 #include <curl/curlver.h>
19 #include <curl/curlver.h>
20 #include <curl/curlver.h>
21 #include <curl/curlver.h>
22 #include <curl/curlver.h>
23 #include <curl/curlver.h>
24 #include <curl/curlver.h>
25 #include <curl/curlver.h>
26 #include <curl/curlver.h>
27 #include <curl/curlver.h>
28 #include <curl/curlver.h>
29 #include <curl/curlver.h>
30 #include <curl/curlver.h>
31 #include <curl/curlver.h>
32 #include <curl/curlver.h>
33 #include <curl/curlver.h>
34 #include <curl/curlver.h>
35 #include <curl/curlver.h>
36 #include <curl/curlver.h>
37 #include <curl/curlver.h>
38 #include <curl/curlver.h>
39 #include <curl/curlver.h>
40 #include <curl/curlver.h>
41 #include <curl/curlver.h>
42 #include <curl/curlver.h>
43 #include <curl/curlver.h>
44 #include <curl/curlver.h>
45 #include <curl/curlver.h>
46 #include <curl/curlver.h>
47 #include <curl/curlver.h>
48 #include <curl/curlver.h>
49 #include <curl/curlver.h>
50 #include <curl/curlver.h>
51 #include <curl/curlver.h>
52 #include <curl/curlver.h>
53 #include <curl/curlver.h>
54 #include <curl/curlver.h>
55 #include <curl/curlver.h>
56 #include <curl/curlver.h>
57 #include <curl/curlver.h>
58 #include <curl/curlver.h>
59 #include <curl/curlver.h>
60 #include <curl/curlver.h>
61 #include <curl/curlver.h>
62 #include <curl/curlver.h>
63 #include <curl/curlver.h>
64 #include <curl/curlver.h>
65 #include <curl/curlver.h>
66 #include <curl/curlver.h>
67 #include <curl/curlver.h>
68 #include <curl/curlver.h>
69 #include <curl/curlver.h>
70 #include <curl/curlver.h>
71 #include <curl/curlver.h>
72 #include <curl/curlver.h>
73 #include <curl/curlver.h>
74 #include <curl/curlver.h>
75 #include <curl/curlver.h>
76 #include <curl/curlver.h>
77 #include <curl/curlver.h>
78 #include <curl/curlver.h>
79 #include <curl/curlver.h>
80 #include <curl/curlver.h>
81 #include <curl/curlver.h>
82 #include <curl/curlver.h>
83 #include <curl/curlver.h>
84 #include <curl/curlver.h>
85 #include <curl/curlver.h>
86 #include <curl/curlver.h>
87 #include <curl/curlver.h>
88 #include <curl/curlver.h>
89 #include <curl/curlver.h>
90 #include <curl/curlver.h>
91 #include <curl/curlver.h>
92 #include <curl/curlver.h>
93 #include <curl/curlver.h>
94 #include <curl/curlver.h>
95 #include <curl/curlver.h>
96 #include <curl/curlver.h>
97 #include <curl/curlver.h>
98 #include <curl/curlver.h>
99 #include <curl/curlver.h>
100 #include <curl/curlver.h>
```

## weather.c

Entering the code for the main implementation, let's take a look at what each function does and how to call them.

### 1. weather\_create

The function `weather_create` is responsible for creating and initializing an instance of the weather module, which is an alternative implementation of a C++ constructor in the C language. It allocates memory for the `weather_t` structure itself, allocates space for the internal `json_response` buffer (used to store JSON data returned by HTTP), and clears this buffer to ensure the safety and reliability of subsequent string concatenation.

It is invoked before the program needs to use the "weather" functions. Typically, it is called once after the system initialization is completed and the Wi-Fi connection is successful to obtain a valid pointer to the weather object.

### 2. weather\_destroy

The role of `weather_destroy` is to release all resources allocated by `weather_create`, including the JSON buffer and the `weather_t` structure itself, to prevent memory leaks. This is the equivalent implementation of a "destructor" in the C language.

It should be invoked when the weather functionality is no longer required by the program—for example, before system shutdown, before task exit, or when the module is dynamically unloaded. It must be used in pairs with `weather_create` to form complete resource lifecycle management.

### 3. http\_event\_handle

`http_event_handler` is the event callback function of the ESP-IDF HTTP client, used to receive data returned by the server during the HTTP request process. Whenever the underlying TCP/HTTP receives a segment of data, this function will be automatically called, and by using `evt->user_data`, it retrieves the current `weather_t` instance and safely concatenates the received data into the `json_response` buffer.

The invocation timing is not directly called by user code, but is triggered internally by `esp_http_client_perform()`, which is an indispensable part of the HTTP communication mechanism.

### 4. weather\_http\_get\_json

The function of `weather_http_get_json` is to initiate an HTTP request and fully obtain the weather JSON data. It encapsulates the entire process of HTTP client configuration, initialization, execution, and resource cleanup. Inside the function, the old JSON buffer will be cleared, the request URL and callback function will be configured, and then the HTTP request will be blocked and waited for completion. Finally, the original JSON string returned by the server will be saved in `weather->json_response`.

This function is called when it is necessary to "obtain the latest raw weather data". However, it is an internal implementation function and is only used by this module. It will not be directly called by the outside.

### 5. weather\_analyse\_weather\_json

The role of `weather_analyse_weather_json` is to parse the JSON string returned by the server and extract the data that the business layer actually cares about, including temperature, weather description, and timestamp. It uses the `cJSON` library to parse the JSON structure and performs validity checks on each field, ensuring that the data format is correct before outputting the data to the variables provided by the caller.

It is invoked after successfully obtaining HTTP JSON data, and is used to convert "string data" into "directly usable business data". This function also belongs to the internal functions of the module.

### 6. weather\_get\_weather

`weather_get_weather` is the only core external interface function of the entire weather module, responsible for providing an "end-to-end weather retrieval service" to external callers. Internally, it first invokes `weather_http_get_json` to fetch the latest weather data from the network, then calls `weather_analyse_weather_json` to parse the JSON and output the results. Callers do not need to care about the details of HTTP, JSON, or memory management.

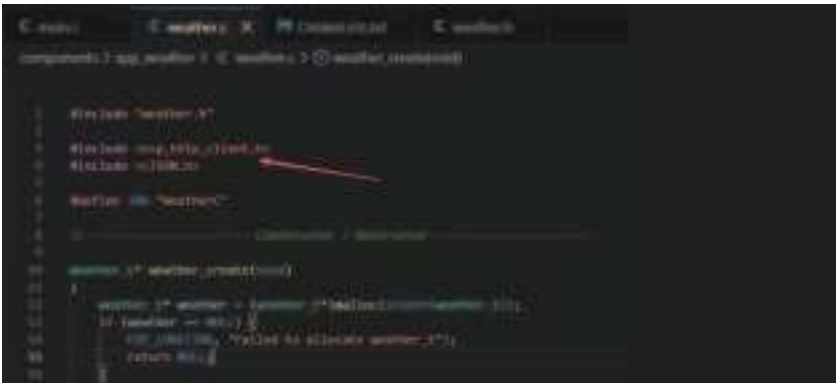
It is invoked when the Wi-Fi connection is successfully established and the application layer requires real-time weather information—for example, scenarios such as UI refresh, periodic weather data updates, etc.

This concludes our explanation of the weather.c file. Although the code implementation is complete, we still need to link the file via CMakeLists.

Therefore, in the CMakeLists.txt file under the app\_weather folder, we need to add the relevant components that are used.



The reason why esp\_http\_client and json are used is that these two libraries were called in the weather.c file.



Let me clarify here that it is recommended to place these two library files only in the "weather.c" file. Since there is no need to expose these two files in the "weather.h" file here, they can be used directly in the "weather.c" file.

## Main function

The main folder is the core directory for program execution, and it contains the executable file main.c for the main function.

Add the main folder to the "CMakeLists.txt" file of the build system.



How to convert the image you want to display on Advance-P4 into a .c file that can be displayed?

First, open this link provided by the LVGL official website:

<https://lvgl.io/tools/imageconverter>

This is a tool provided by LVGL itself for converting images into C arrays.



Then, make the selection as indicated in the figure below.

Step 1: The LVGL-related components in our project all use the v8 version. Therefore, we choose LVGL v8.

Step 2: Select the image you want to convert. This image must maintain a resolution of 1024\*600, as our screen resolution is 1024\*600.

Step 3: Make sure to select CF\_TRUE\_COLOR\_ALPHA. Since our screen is 16-bit, LVGL will automatically convert the 32-bit pixels into the 16-bit pixels of the screen configuration, and the transparency effect will still exist.

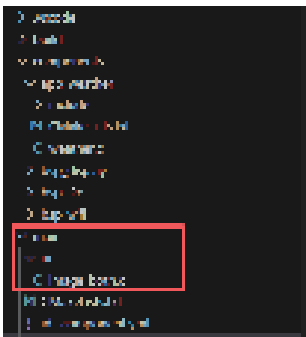
Step 4: Choose the method of converting to a C array.

Step 5: Perform the conversion and obtain the corresponding .c file.

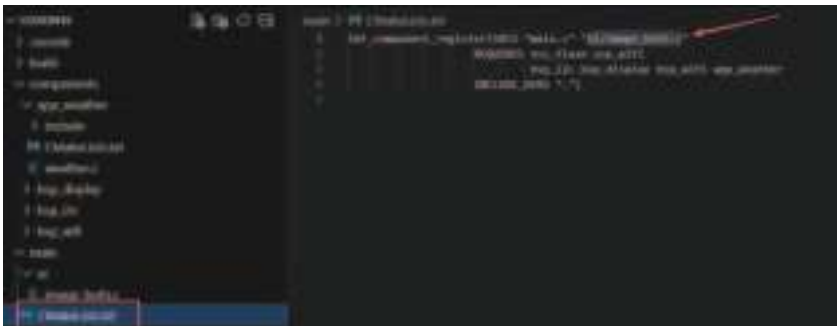




Then, place the converted .c file in the "ui" folder under the "main" folder of the project.



And add this statement in the CMakeLists.txt file under the main folder. You need to specify the exact location of this image C file.



In this way, our code can find the "image.c" file during compilation.

For the main file body, let's first explain main.c.

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

First, complete the initialization of the underlying hardware and system resources, then establish a network connection and obtain weather and time information, and finally initialize and display the graphical interface.

After the program starts, it first configures and enables the LDO3 power channel of the ESP32-P4 to provide 2.5V voltage to the screen; then initializes the basic hardware modules such as NVS, I2C, touch, and display to ensure a complete and usable system operating environment.

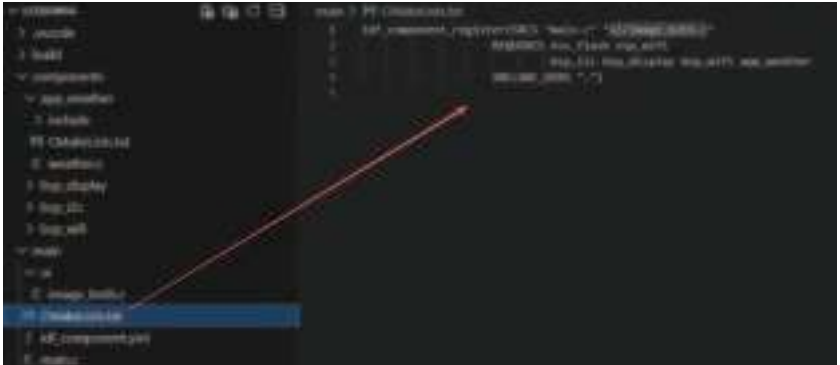
Next, start the Wi-Fi subsystem and connect to the specified hotspot in STA mode. In the loop, wait for the network connection to be successful, then send an HTTP request through the weather module to obtain real-time weather data and timestamp, and synchronize the timestamp to the system clock.

After completing the data preparation, the program enters the initialization stage of the LVGL graphical interface, creates the main interface object and date, temperature, and weather text labels, displays the obtained data on the screen, and finally turns on the backlight to complete the interface presentation.

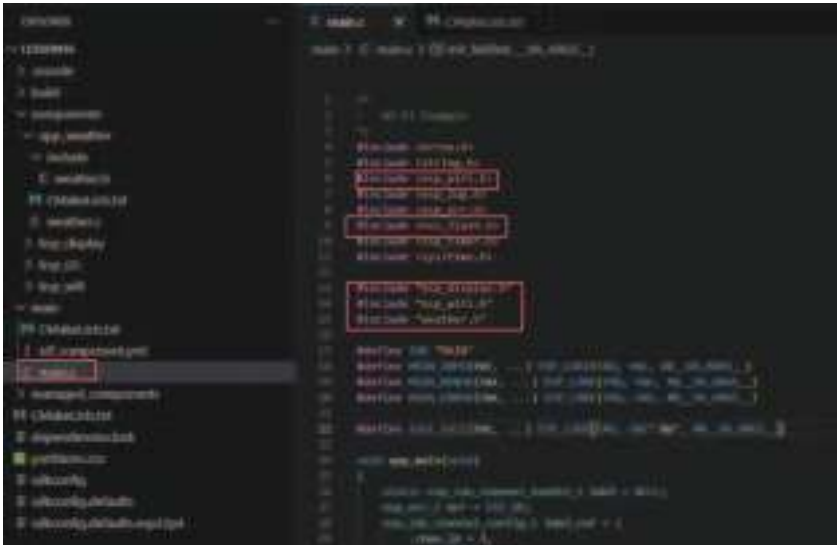
Now, let's explain the main code file "main.c".

```
main > C: main > C:\app\main\main.c
1 //
2 // Wi-Fi Example
3 //
4 #include <string.h>
5 #include <string.h>
6 #include <esp_wifi.h>
7 #include <esp_log.h>
8 #include <esp_err.h>
9 #include <esp_flash.h>
10 #include <esp_timer.h>
11 #include <esp_system.h>
12
13 #include "esp_wifi.h"
14 #include "esp_wifi.h"
15 #include "wifi.h"
16
17 #define SSID "SSID"
18 #define PASS "PASS"
19 #define SSID_LEN (sizeof(SSID) - 1)
20 #define PASS_LEN (sizeof(PASS) - 1)
21 #define SSID_MAX_LEN (sizeof(SSID) - 1)
22 #define PASS_MAX_LEN (sizeof(PASS) - 1)
```

Here, the components required for screen display (bsp\_display), the component for connecting to Wi-Fi (bsp\_wifi), and the component for obtaining weather information (app\_weather) are all included here.



In the CmakeLists.txt file under the main folder, these components are chosen because of the header files included in main.c.



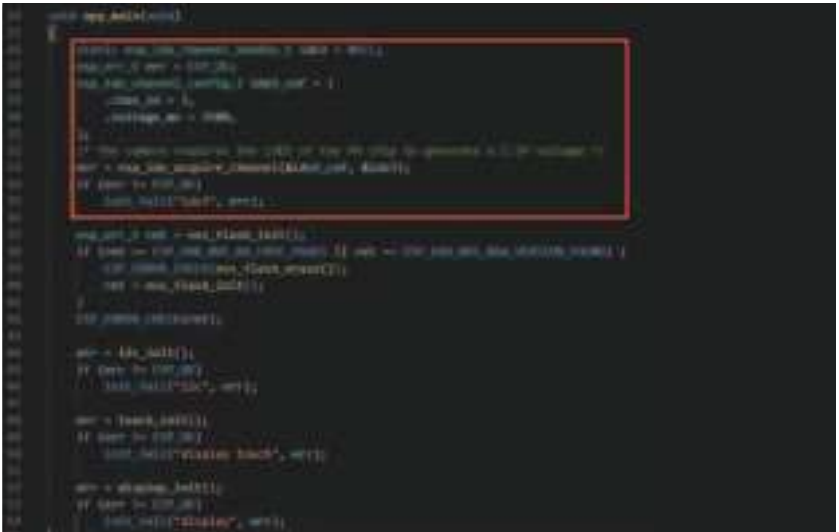
## Let's continue to look at the content in main.c.

### 1. Initialization of power supply and chip-level resources (bottom-level guarantee)

At the beginning of `app_main`, the LDO3 power channel of ESP32-P4 is configured and allocated. Its output is set to 2.5V, which is a prerequisite for providing a stable power supply to external peripherals such as the onboard camera.

This step belongs to the initialization of chip-level resources. If this step is not completed first, subsequent peripherals may experience non-functioning or unstable operation.

By checking the return value of `esp_ldo_acquire_channel`, hardware-level errors can be detected at the earliest stage of the system, avoiding "running the program with defects".



```
void app_main(void)
{
    // Configure LDO3 power channel
    esp_ldo_acquire_channel(LDO3_CHAN, &ldo3);
    if (ldo3 == NULL)
    {
        printf("LDO3 channel acquisition failed\n");
        return;
    }
    // Set LDO3 output voltage to 2.5V
    esp_ldo_set_voltage(ldo3, ESP_LDO_VOLTAGE_2500);
    if (ldo3 == NULL)
    {
        printf("LDO3 voltage setting failed\n");
        return;
    }
    // Enable LDO3
    esp_ldo_enable(ldo3);
    if (ldo3 == NULL)
    {
        printf("LDO3 enable failed\n");
        return;
    }
    // Check LDO3 status
    esp_ldo_get_status(ldo3);
    if (ldo3 == NULL)
    {
        printf("LDO3 status check failed\n");
        return;
    }
    // ... other initialization code ...
}
```

### 2. NVS Initialization (A prerequisite for Wi-Fi / System Services)

Subsequently, NVS (Non-Volatile Storage) is initialized. This is a mandatory module for components such as Wi-Fi and network protocol stack in ESP-IDF.

In the code, handling for "insufficient space or version mismatch" is performed for erasing and re-creating, ensuring that NVS is in a usable state.

This step must be completed before Wi-Fi initialization, as failure to do so will prevent Wi-Fi from starting normally, and thus it must be placed before Wi-Fi initialization.



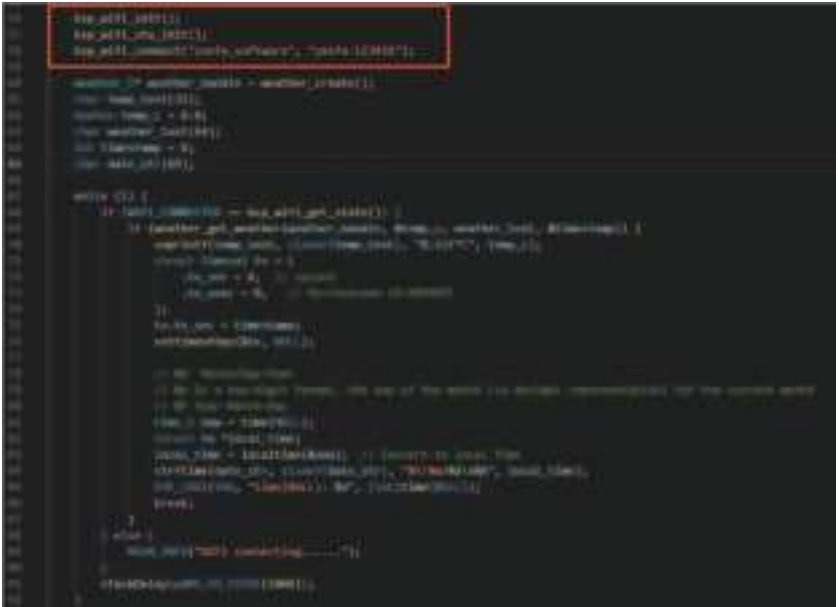
## 4. Wi-Fi subsystem startup and networking

You can enter your Wi-Fi name and password here to connect to your Wi-Fi.

After the hardware is prepared, the program starts the Wi-Fi BSP module and completes the following three steps:

- Initialize the Wi-Fi driver and event system
- Start Wi-Fi in STA (Station) mode
- Connect to the specified wireless router

Here, the state polling (`bsp_wifi_get_state`) method is used instead of blocking waiting, leaving room for subsequent expansion of multi-tasking or real-time UI refresh.



```
bsp_wifi_init();
bsp_wifi_set_ssid(ssid);
bsp_wifi_set_password(password, "wpa123456");

// weather module
weather_module = weather_init();

// main loop
while (1) {
    if (bsp_wifi_get_state() == WIFI_STA_CONNECTED) {
        // Do something when connected to the Wi-Fi network
        // ...
    }
    // Do something when not connected to the Wi-Fi network
    // ...
}
```

## 5. Weather module creation and network data acquisition

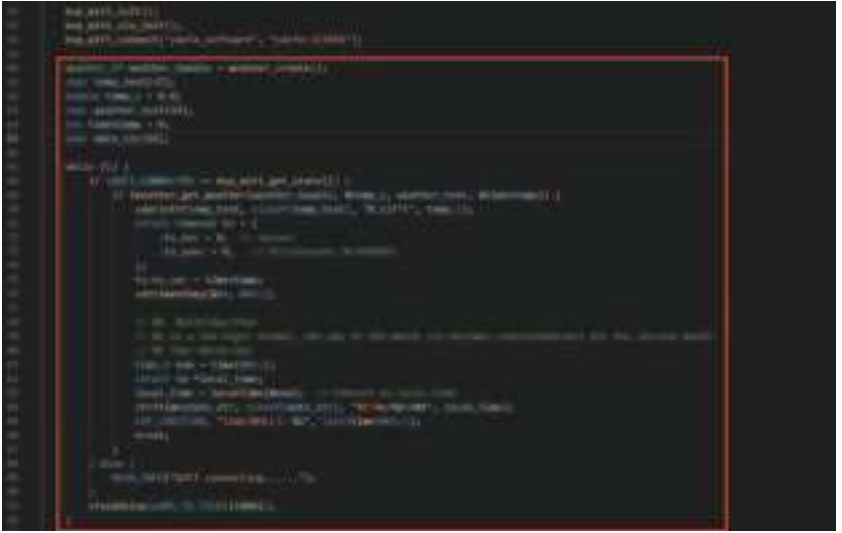
After Wi-Fi is activated, the program creates an instance of the weather module, which is a typical "C language object-oriented encapsulation" approach.

Then, it enters a loop to continuously check if Wi-Fi is connected.

Once the connection is successful, it immediately initiates an HTTP request through the `weather_get_weather` function to obtain weather data.

Once the data is retrieved successfully, not only is the temperature and weather text parsed, but the timestamp returned by the server is also written to the system clock, synchronizing the device time with the network time.

This step lays the foundation for subsequent time display and logging.



## 6. LVGL Interface Creation and Data Display

After the data is prepared, the program enters the stage of building the LVGL graphical interface.

By using `lvgl_port_lock` to ensure thread safety, the main interface container is created, and three label controls for date, temperature, and weather are created one by one.

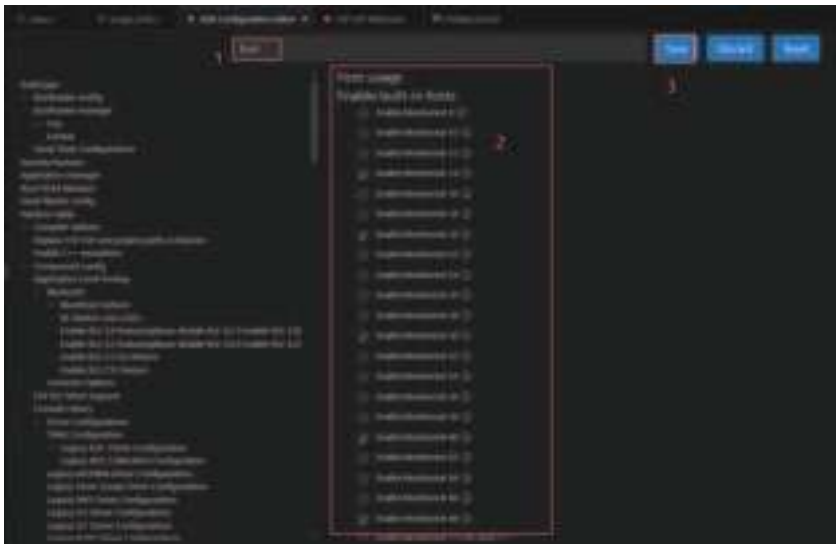
The relevant data obtained previously is directly bound to the UI.

At this point, the data flow has completed a complete closed loop from the network → parsing → system time → UI display.





After opening the SDK, search for "font", select the font you want to use, and then save it.



## 7. Turning on the backlight and entering the stable operation state of the system

Finally, after the interface is constructed, turn on the LCD backlight to allow users to see the complete UI.

At this point, the system has completed initialization and entered a stable operation state.

Subsequently, functions such as periodic weather updates, touch interaction, or multi-page switching can be further expanded.

```
141 set_lcd_backlight(100); // At this point, turn on the screen backlight
142
143
```

## Complete Code

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

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson16\\_Get%20weather%20via%20WiFi](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson16_Get%20weather%20via%20WiFi)

## 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.



- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



- Here, following the steps in the first section, we first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



- Wait for a moment for the loading process to complete, and then you can proceed with the relevant SDK configuration.



- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



After the configuration is completed, remember to save your settings.

- Then we will compile and burn the code (this was explained in detail in the first class).
- Here, we would like to introduce to you a very convenient feature. With just one button press, you can perform the compilation, upload, and open the monitor all at once.



- After waiting for a while, the code compilation and upload were completed, and the monitor also opened.

- At this point, please remember to use another Type-C cable to connect your Advance-P4 through the USB2.0 interface. This interface provides a maximum current of about 500mA for the computer's USB-A interface. When the Advance-P4 is using more external devices, especially the screen, it needs a sufficient current source. (It is recommended to use a charger for connection.)



- After running the code, you will be able to see the local weather conditions you have obtained on the screen of the Advance-P4.



# Lesson 17

## WiFi Mode

### Introduction

---

In this section, we will conduct a systematic study on the Wi-Fi core functions of the ESP32 series chips (including models such as ESP32-C6): Firstly, we will deeply analyze the underlying fundamental principles of Wi-Fi communication, including the technical logic and application scenarios of the three core working modes: Station (STA), SoftAP (AP), and STA+AP coexistence.

### ESP32 Wi-Fi Function Overview

---

The ESP32 series of chips, with their highly integrated wireless communication capabilities, have become the core preferred solution in fields such as the Internet of Things (IoT), smart home, and industrial automation.

Most models of this series are equipped with high-performance wireless modules, among which 2.4 GHz Wi-Fi is the basic standard, which can meet the needs of most low-power, short-range data transmission scenarios.

The new models represented by ESP32-C6 have further expanded the boundaries of wireless capabilities, not only supporting 5 GHz dual-band Wi-Fi, but also being compatible with the new Wi-Fi 6 (802.11ax) protocol, while also being compatible with traditional standards such as 802.11b/g/n.

They can provide higher transmission rates, lower network latency, and more stable concurrent connection experiences for multiple devices.

It is worth noting that some special models in the ESP32 family (such as the P series and H series) have simplified the Wi-Fi function modules to focus on exclusive scenarios such as high-performance computing and industrial-level control, in exchange for more abundant computing resources and more stable industrial-level performance.

The details of the wireless functions supported by each model can be accurately selected according to the official document "ESP32 Product Overview" released by Espressif.

The Wi-Fi function of the ESP32 series chips offers a wide range of working modes, which can flexibly adapt to various development requirements:

- **Station (STA) mode:** The chip acts as a Wi-Fi client, actively connecting to existing wireless networks such as routers and hotspots, obtaining a dynamic IP address and joining the local network, enabling data communication with devices in the external network or within the local network;
- **SoftAP (soft AP) mode:** The chip creates an independent Wi-Fi hotspot by itself, allowing devices such as mobile phones, computers, and other single-chip microcontrollers to connect. This builds a point-to-point or small local area network without the need for an external router, suitable for local device debugging, offline data interaction, etc.;
- **STA + AP coexistence mode:** The chip runs both of these modes simultaneously. It can connect to the external network to achieve data upload and remote control, and also act as a local hotspot for debugging devices to connect, balancing the dual needs of remote operation and local development.

In terms of security and performance, the Wi-Fi function of ESP32 also performs exceptionally well:

In terms of security, it supports mainstream encryption protocols such as WPA2, WPA3, and enterprise-level authentication, which can effectively ensure the security of data transmission and prevent network attacks and data leakage;

In terms of performance, its maximum transmission rate can reach 150 Mbps (with some high-end models having even higher rates), and it is equipped with multiple refined power-saving modes, which can dynamically adjust power consumption according to actual business needs, extend the battery-powered device's battery life, and some models also support multi-antenna diversity technology, further enhancing the stability and anti-interference ability of wireless signals in complex environments.

<https://products.espressif.com/static/Esspressif%20SoC%20Product%20Portfolio.pdf>



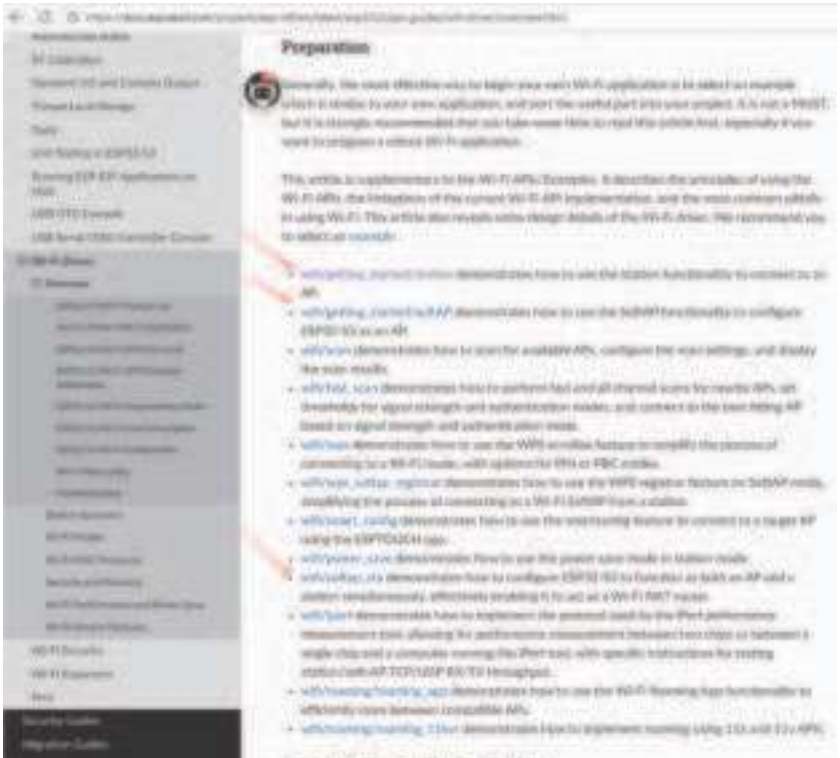
# General steps for Wi-Fi programming

When developing Wi-Fi applications using the ESP32 series chips based on the ESP-IDF framework, regardless of whether the goal is to configure the Station (STA) client mode or the SoftAP (AP) access point mode, the core programming process follows three main stages: initialization, configuration, connection, and event handling.

The operation logic of each stage is highly universal and standardized.

If you want to deeply understand the complete process of specific scenarios (such as disconnection and reconnection in STA mode, and device access management in AP mode), you can refer to the detailed guidelines in the official ESP-IDF documentation for "Wi-Fi Station Development Process" and "Wi-Fi AP Development Process".

<https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-guides/wifi-driver/overview.html>



## 1. Initialization phase: Establish the basic environment for Wi-Fi operation

The core objective of this phase is to complete the initialization of the underlying protocol stack, hardware drivers, and system event framework, laying a solid foundation for the operation of Wi-Fi functions. The specific operations are as follows:

### Initialization of protocol stack and network interface:

Firstly, initialize the lightweight TCP/IP protocol stack (LWIP), which is the core underlying component for implementing network data transmission; simultaneously, initialize the esp-netif network interface management module, which is responsible for uniformly managing various network interfaces of ESP32 (such as STA, AP, Ethernet, etc.), serving as the key bridge between upper-layer applications and the underlying hardware.

### Building event-driven framework:

Create the system's default event loop (esp\_event) and register Wi-Fi-related event handling callbacks. The Wi-Fi status changes (such as successful connection, obtaining IP, disconnection, etc.) are triggered in the form of asynchronous events, and the event loop is the core mechanism for implementing status monitoring and automated processing.

### Creation of default network interface:

Create corresponding default network interface instances (such as ESP\_NETIF\_DEFAULT\_WIFI\_STA or ESP\_NETIF\_DEFAULT\_WIFI\_AP) based on the target mode (STA/AP), bind the network interface with the Wi-Fi hardware, and ensure that data can be transmitted and received through the specified interface.

### Wi-Fi driver initialization:

Call the esp\_wifi\_init() interface to complete the initialization of the Wi-Fi hardware driver. This operation will start the Wi-Fi task thread inside ESP32, complete the adaptation of the radio module, protocol stack, and hardware, and ensure that the Wi-Fi hardware is in a configurable and operational state.

## 2. Configuration Phase: Customizing Wi-Fi Operating Parameters

After initialization is completed, the core operating parameters of Wi-Fi need to be configured according to actual business requirements to match the target network environment or functional requirements:

### Connection Parameter Configuration:

Define the wifi\_config\_t structure variable and fill in the core parameters corresponding to the mode - in STA mode, the SSID (network name), password, authentication method (such as WPA2-PSK) of the target Wi-Fi need to be configured; in AP mode, the hotspot name, password, maximum number of connected devices, authentication method, and channel need to be configured.

### Mode Setting:

Call the `esp_wifi_set_mode()` interface to clearly set the Wi-Fi operating mode (such as `WIFI_MODE_STA`, `WIFI_MODE_AP`, or `WIFI_MODE_APSTA`), and this operation will lock the core working logic of Wi-Fi.

### Configuration Parameters Take Effect:

Call the `esp_wifi_set_config()` interface to apply the parameters configured in the `wifi_config_t` to the corresponding network interface (STA/AP), so that the parameters are bound with the hardware driver and protocol stack.

## 3. Connection and Event Handling Phase:

### Initiating Wi-Fi and Handling State Changes

After configuration is completed, start the Wi-Fi function and handle various asynchronous states through event callbacks to ensure the stable operation of the Wi-Fi function:

#### Starting Wi-Fi function:

Call the `esp_wifi_start()` interface to start the Wi-Fi module. At this point, the ESP32 will initialize the radio module according to the configured mode and enter the "waiting to connect (STA)" or "hotspot broadcasting (AP)" state.

#### Initiating network connection (only in STA mode):

Call the `esp_wifi_connect()` interface to send a connection request to the configured target Wi-Fi network; in AP mode, this step is not required and the hotspot will be automatically broadcasted and waiting for device access.

#### Asynchronous event handling:

By registering the `WIFI_EVENT` and `IP_EVENT` event callback functions, handle various critical states - such as "connection success/failure" "obtaining IP address" "connection disconnection" in STA mode, and "device access/disconnection" in AP mode.

Event handling is the core to ensure the robustness of the Wi-Fi function, for example, adding reconnection logic in the "connection disconnection" event and triggering data transmission tasks in the "obtaining IP" event.

After understanding the above professional knowledge, next we will guide you to have a better understanding of these three modes.

## Station (STA) mode

### 1. Professional Definition

The Station (STA) mode refers to the device operating in the IEEE 802.11 Station role, actively scanning (Active Scan) or passively scanning (Passive Scan) to search for and associate with an Access Point (AP), then obtaining network parameters (IP / Gateway / DNS) through the DHCP protocol, and finally joining an Infrastructure Network.

You don't need to memorize all the key words at once, but being familiar with them is very important.

## 2. What exactly did the STA mode do in the engineering aspect?

Under the STA mode, the ESP32-C6 actually completed a complete set of processes:

- Channel scanning (Channel Scan), 2.4GHz (Channels 1-13)
- Authentication WPA2 / WPA3
- Association
- 4-Way Handshake for Key Agreement
- DHCP Request
- Completion of IP Layer Network Establishment

In your code, you only wrote one sentence:

```
bsp_wifi_connect(ssid, password);
```

But the underlying layer has already executed the entire Wi-Fi protocol stack.

## 3. Understanding STA


You can think of STA as:

- "I'm just an ordinary user, and I want to access the internet"

Its characteristics are very clear:

 No Wi-Fi

 Regardless of others

 Only concerned with:

- Whether I have connected
- Whether I have an IP address
- Whether I can access the server





## SoftAP (AP) mode

### 1. Professional Definition

The SoftAP (Software Access Point) mode refers to the device simulating a Wi-Fi access point (Access Point) through software, periodically sending Beacon frames to broadcast the SSID, and acting as a DHCP Server and local gateway to construct an independent wireless local area network (WLAN).

### 2. What does SoftAP do in engineering

In AP mode, the ESP32-C6 will:

Regular broadcast:

- SSID
- Encryption method
- Channel information

Accept client association requests

For each connected device:

- Assign IP
- Maintain connection table

That is to say:

- At this point, the ESP32 "behaves like a router"

### 3. Understanding SoftAP

You can think of SoftAP as:

- A "router without broadband"

It enables mobile phones to connect;

It allows for communication between devices;

But it does not access the internet by default.

### 4. Why are embedded devices particularly fond of using APs?

Because it solves a real engineering problem:

- When the device is powered on for the first time, it has no idea how to handle the Wi-Fi password?

Answer:

- Enable AP for the device
- Connect the mobile phone to the device
- Enter the actual Wi-Fi information
- Save to NVS
- Restart → STA mode

## 5. The actual applications in the architecture

In the P4 + C6 project, the AP mode is typically used for:

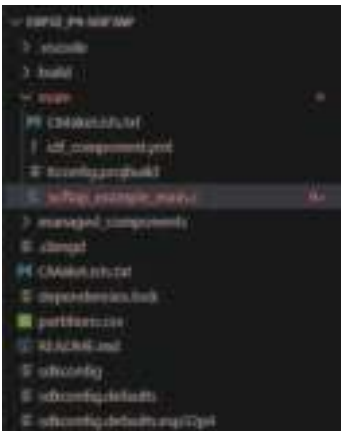
- Distribution network
- Local web control
- Offline debugging

Now that we have understood the SoftAP mode, let's use it to enhance our understanding.

Click the link below to download the code for the SoftAP mode.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson7-Wi-Fi\\_function](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson7-Wi-Fi_function)

After downloading, open the SoftAP mode code using VSCode. The framework of this code can be seen:



Let's be straightforward and say how to use it:

Click on `softap_example_main.c`, then enter the main code.



Since now we are treating Advance-P4 as a router, we can use a mobile phone or a laptop to connect to the Wi-Fi that is set in the code.

You can fill in the name and password of the router Wi-Fi in the code.



Then you can start preparing to compile your project.

Simply click the "One-click Upload" button, and your code will be compiled and uploaded all at once. After the upload is complete, open the serial port monitor.





```
E: 2020/11/25/15:56:11.411
I: 2020/11/25/15:56:11.412
E: 2020/11/25/15:56:11.413
I: 2020/11/25/15:56:11.414
E: 2020/11/25/15:56:11.415
I: 2020/11/25/15:56:11.416
E: 2020/11/25/15:56:11.417
I: 2020/11/25/15:56:11.418
E: 2020/11/25/15:56:11.419
I: 2020/11/25/15:56:11.420
E: 2020/11/25/15:56:11.421
I: 2020/11/25/15:56:11.422
E: 2020/11/25/15:56:11.423
I: 2020/11/25/15:56:11.424
E: 2020/11/25/15:56:11.425
I: 2020/11/25/15:56:11.426
E: 2020/11/25/15:56:11.427
I: 2020/11/25/15:56:11.428
E: 2020/11/25/15:56:11.429
I: 2020/11/25/15:56:11.430
E: 2020/11/25/15:56:11.431
I: 2020/11/25/15:56:11.432
E: 2020/11/25/15:56:11.433
I: 2020/11/25/15:56:11.434
E: 2020/11/25/15:56:11.435
I: 2020/11/25/15:56:11.436
E: 2020/11/25/15:56:11.437
I: 2020/11/25/15:56:11.438
E: 2020/11/25/15:56:11.439
I: 2020/11/25/15:56:11.440
E: 2020/11/25/15:56:11.441
I: 2020/11/25/15:56:11.442
E: 2020/11/25/15:56:11.443
I: 2020/11/25/15:56:11.444
E: 2020/11/25/15:56:11.445
I: 2020/11/25/15:56:11.446
E: 2020/11/25/15:56:11.447
I: 2020/11/25/15:56:11.448
E: 2020/11/25/15:56:11.449
I: 2020/11/25/15:56:11.450
E: 2020/11/25/15:56:11.451
I: 2020/11/25/15:56:11.452
E: 2020/11/25/15:56:11.453
I: 2020/11/25/15:56:11.454
E: 2020/11/25/15:56:11.455
I: 2020/11/25/15:56:11.456
E: 2020/11/25/15:56:11.457
I: 2020/11/25/15:56:11.458
E: 2020/11/25/15:56:11.459
I: 2020/11/25/15:56:11.460
E: 2020/11/25/15:56:11.461
I: 2020/11/25/15:56:11.462
E: 2020/11/25/15:56:11.463
I: 2020/11/25/15:56:11.464
E: 2020/11/25/15:56:11.465
I: 2020/11/25/15:56:11.466
E: 2020/11/25/15:56:11.467
I: 2020/11/25/15:56:11.468
E: 2020/11/25/15:56:11.469
I: 2020/11/25/15:56:11.470
E: 2020/11/25/15:56:11.471
I: 2020/11/25/15:56:11.472
E: 2020/11/25/15:56:11.473
I: 2020/11/25/15:56:11.474
E: 2020/11/25/15:56:11.475
I: 2020/11/25/15:56:11.476
E: 2020/11/25/15:56:11.477
I: 2020/11/25/15:56:11.478
E: 2020/11/25/15:56:11.479
I: 2020/11/25/15:56:11.480
E: 2020/11/25/15:56:11.481
I: 2020/11/25/15:56:11.482
E: 2020/11/25/15:56:11.483
I: 2020/11/25/15:56:11.484
E: 2020/11/25/15:56:11.485
I: 2020/11/25/15:56:11.486
E: 2020/11/25/15:56:11.487
I: 2020/11/25/15:56:11.488
E: 2020/11/25/15:56:11.489
I: 2020/11/25/15:56:11.490
E: 2020/11/25/15:56:11.491
I: 2020/11/25/15:56:11.492
E: 2020/11/25/15:56:11.493
I: 2020/11/25/15:56:11.494
E: 2020/11/25/15:56:11.495
I: 2020/11/25/15:56:11.496
E: 2020/11/25/15:56:11.497
I: 2020/11/25/15:56:11.498
E: 2020/11/25/15:56:11.499
I: 2020/11/25/15:56:11.500
```

However, this Wi-Fi has no network available for access. You can think of it as a router without an Ethernet cable. You can try to connect it, but you won't be able to access the internet.

### STA + AP coexistence mode

#### 1. Professional Definition

The STA + AP coexistence mode (Station + SoftAP Concurrency) refers to the scenario on a single-radio-frequency (Single RF) Wi-Fi SoC where:

- A Station Interface (STA)
- A SoftAP Interface (AP)

are simultaneously running through a virtual network interface (Virtual Wi-Fi Interface, VIF), and the MAC layer scheduler (MAC Scheduler) switches the transmission and reception between the two roles through time-division multiplexing (TDM) in a time-sharing manner.

#### 2. Underlying Facts

TA + AP ≠ Two Wi-Fi modules

Rather:

- One set of RF
- One set of PHY
- One set of MAC
- Two logical identities

The ESP32-C6 "pretends" to be two devices at the protocol level.

### 3. How is the protocol and system layer implemented?

#### Virtual Interface (Virtual Interface, VIF)

In ESP-IDF, STA + AP is actually: Wi-Fi Driver

- netif\_sta (STA interface)
- netif\_ap (AP interface)

Each interface has:

- Independent MAC state machine
- Independent IP stack (lwIP)
- Independent DHCP behavior

However, they share the same radio frequency hardware.

### 4. What is Time Division Multiplexing (TDM) doing?

Since there is only one antenna, the ESP32-C6 will:

- Sometimes "listen to the router"
- Sometimes "send Beacons to the mobile phone"
- Sometimes receive data from the cloud
- Sometimes respond to the phone's requests

You can think of it as: The Wi-Fi chip rapidly switches between two identities.

This switching is transparent to the upper-layer applications.

### 5. Channel Binding

In the STA + AP mode, the channel of the AP must be the same as the channel of the router that the STA is connected to. Reason:

- Single RF cannot operate simultaneously on two channels.
- The STA has a higher priority than the AP.

Therefore:

- The router is on Channel 6.
- The SoftAP can only be on Channel 6.

## 6. Understanding STA + AP

Think of the device as a person:

- STA:

Using a mobile phone to call the company headquarters (the external world)

- AP:

At the same time, using a walkie-talkie, and everyone nearby can call you

One action, two communication methods.

## 7. The fundamental difference from the standalone STA/AP mode

The standalone STA mode only allows the ESP32 to act as a Wi-Fi client and connect to an existing wireless network outside, with the core requirement being "accessing the network to enable external communication";

The standalone AP mode only enables the ESP32 to become a Wi-Fi hotspot itself, with the core requirement being "building a local local area network for other devices to access";

While the STA+AP coexistence mode simultaneously enables these two working modes on a single chip, its essence is to balance the "accessing an external network to achieve remote/external network communication" and "building a local hotspot to enable local device interaction" capabilities.

Although it experiences a slight decrease in throughput due to radio frequency time division multiplexing, it can simultaneously meet the dual scenarios of remote operation, local debugging/ networking configuration, etc., expanding the functional boundaries of a single mode.

Now that we have understood the Sta+AP mode, let's use it to reinforce our understanding.

Click the link below to download the code for the Sta+AP mode.

[https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson7-Wi-Fi\\_function](https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson7-Wi-Fi_function)

After downloading, open the Sta+AP mode code using VSCode.

The framework of this code can be seen:





You can fill in the router's Wi-Fi name and password in the code.



Then you can start preparing to compile your project.

Simply click the "One-click Upload" button, and your code will be compiled and uploaded all at once. After the upload is complete, open the serial port monitor.



After the upload is completed, you will be able to see the relevant information indicating that the STA mode has successfully connected to the WiFi you are using.

```
1 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
2 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
3 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
4 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
5 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
6 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
7 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
8 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
9 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
10 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
11 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
12 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
13 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
14 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
15 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
16 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
17 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
18 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
19 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
20 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
21 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
22 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
23 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
24 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
25 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
26 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
27 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
28 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
29 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
30 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
31 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
32 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
33 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
34 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
35 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
36 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
37 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
38 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
39 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
40 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
41 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
42 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
43 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
44 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
45 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
46 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
47 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
48 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
49 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
50 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0% 2000: 0%
```

Then you can also use your mobile phone or laptop to connect to the Wi-Fi of this router.







## Why are these two configurations enabled?

In ESP-IDF, enabling `CONFIG_LWIP_IP_FORWARD=y` and `CONFIG_LWIP_IPV4_NAPT=y` is the core purpose to enable the Advance-P4 (the Wi-Fi module based on ESP32-C6) to have the ability of cross-interface network data forwarding and network address translation in the STA+AP coexistence mode, ultimately achieving the core requirement of "external devices connected to the Advance-P4 AP hotspot can access the external network through the STA link of Advance-P4".

The specific functions of these two configurations are as follows:

### 1. `CONFIG_LWIP_IP_FORWARD=y`: Enable IP layer data forwarding

LWIP is the lightweight TCP/IP protocol stack used by ESP32-C6. The core of this configuration is to enable the IP forwarding function of the protocol stack - allowing the network protocol stack of ESP32-C6 to allow data packets to be forwarded between different network interfaces (i.e., the STA interface and the AP interface).

Without this configuration: The STA interface (connected to the user's WiFi) and the AP interface (its own hotspot) are completely isolated. Data packets can only flow within their respective interfaces, and devices connected to the AP cannot access the network on the STA side;

After enabling: ESP32-C6 can forward the external network request data packets received from the AP interface to the STA interface, and then access the external network through the user's router; at the same time, it can also forward the data packets returned from the external network from the STA interface back to the AP interface and give them to the devices connected to the hotspot.

In simple terms, this configuration is the basis for achieving "cross-interface communication", equivalent to opening the "data packet channel between the STA and AP interfaces" for ESP32-C6.

## 2. CONFIG\_LWIP\_IPV4\_NAPT=y: Enables IPv4 Network Address Translation (NAT)

This configuration is based on IP forwarding and aims to solve the problem of "address compatibility between devices in different subnets". The core is to implement Network Address Port Translation:

Devices connected to the Advance-P4 hotspot receive a private IP assigned by the AP (such as 192.168.4.x), while the STA interface of the Advance-P4 obtains a private IP from the user router from another subnet; The external network can only recognize the IP of the STA interface but cannot directly recognize the IP of the AP-side device;

After enabling NAPT: The ESP32-C6 will convert the IP + port of the AP-side device + port into its own STA interface's IP + random port and send the request to the external network; When the external network returns data, it will convert the STA interface's IP + port back to the AP-side device's IP + port, achieving data transmission back.

In simple terms, this configuration enables the AP-side device to "borrow" the IP of the STA interface to access the external network, solving the problem of address mapping between different subnets, and is the key for the AP-side device to access the internet normally.

### The final effect of combining the two configurations

After enabling these two configurations, the Advance-P4 gains the core capabilities of a "small router": devices such as mobile phones/computers connected to its AP hotspot can communicate locally with the Advance-P4, and can also access the local area network of the user's WiFi through the STA link of the Advance-P4, and even access the external network (such as accessing the internet, connecting to servers); conversely, if either of them is disabled, the AP-side device will be unable to access the internet normally (only able to communicate locally).



MAKE YOUR MAKING EASIER