Advance HMI P4

www.elecrow.com

Table of contents

Lesson 01
Print “Hello World”

Introduction

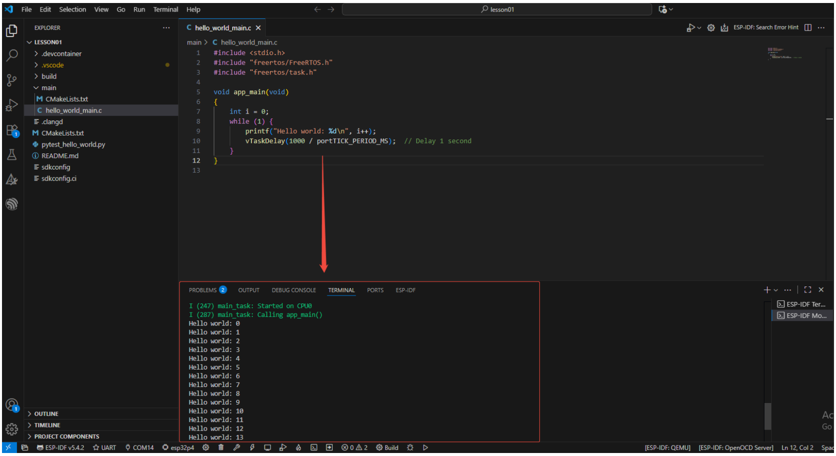
In this class, we will officially learn to write code in the ESP-IDF environment to drive the Advance-P4 development board. The subsequent courses will follow a gradient design from simple to complex, helping you gradually master the ESP-IDF development framework and the usage logic of the ESP32-P4 chip, 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.

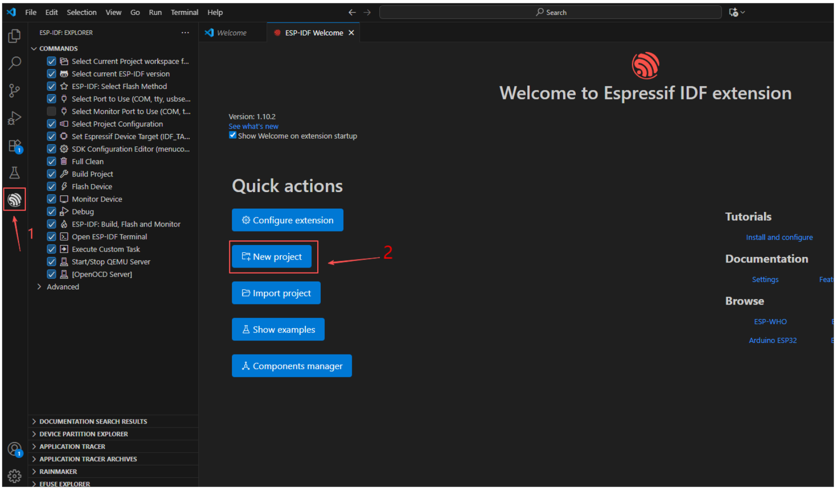
Operation Effect Diagram

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

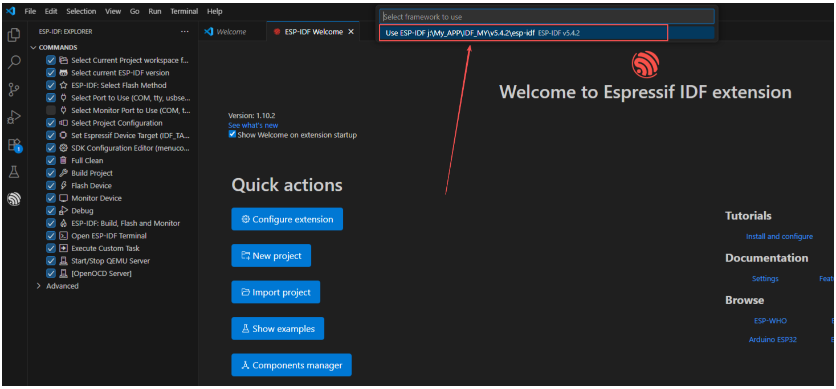


Key Explanations

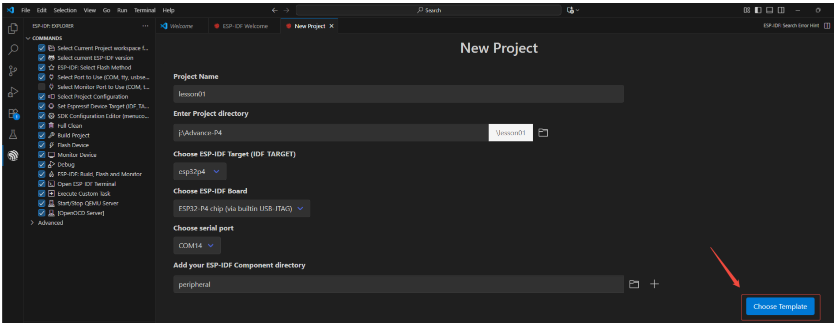
- First, let's talk about how to create a new project in the already installed ESP-IDF.
- Click on the ESP-IDF icon, then click "New project"



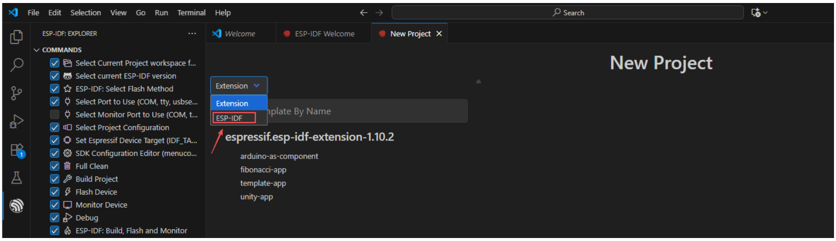
- Then a version of the ESP-IDF environment that you configured in the previous class will pop up.
- Select the 5.4.2 version that you previously set up.



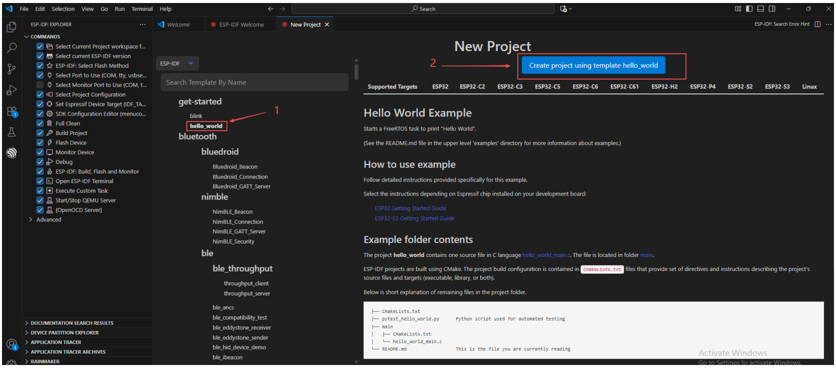
- Then, enter this configuration interface. Here, fill in and set the name, path, target chip, serial port, and the folder name for the subsequent used component files of your newly created project.
- Finally, select the template.



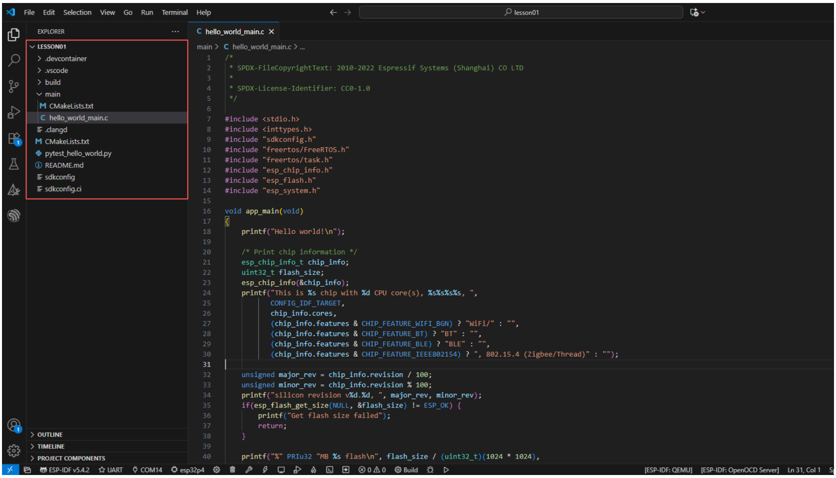
- Choose ESP-IDF



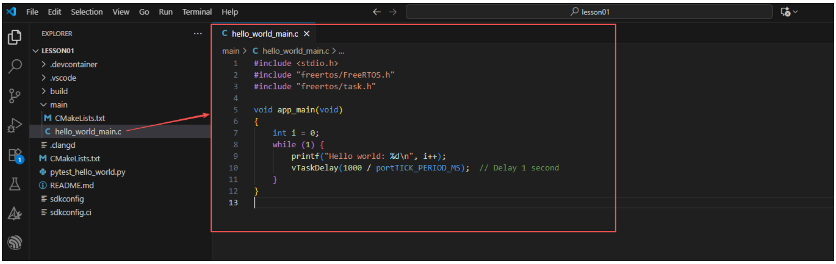
- After selecting "Hello World", click "Confirm Creation" (you can also take a detailed look at the official introduction of this interface).



- 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 > C hello_world_main.c > ...
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
```

- 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)
{
    int i = 0;
    while (1) {
        printf("Hello world: %d\n", i++);
        vTaskDelay(1000 / portTICK_PERIOD_MS);  // Delay 1 second
    }
}
```

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

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

- 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)
{
    int i = 0;
    while (1) {
        printf("Hello world: %d\n", i++);
        vTaskDelay(1000 / portTICK_PERIOD_MS);  // Delay 1 second
    }
}
```

- 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.

main > C hello_world_main.c > ...

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

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

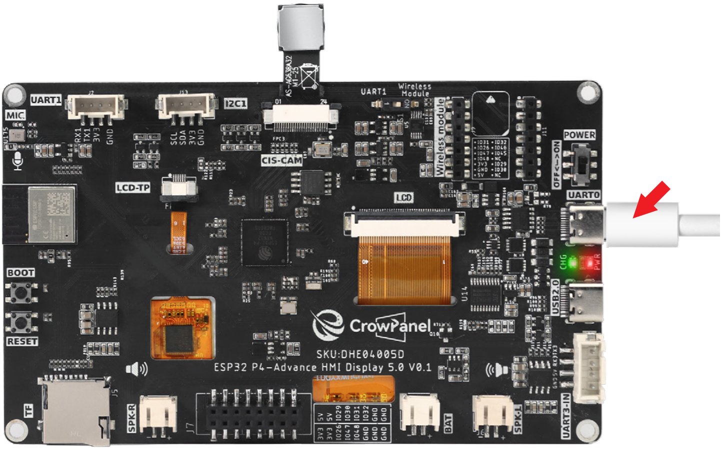
Complete Code

Kindly click the link below to view the full code.

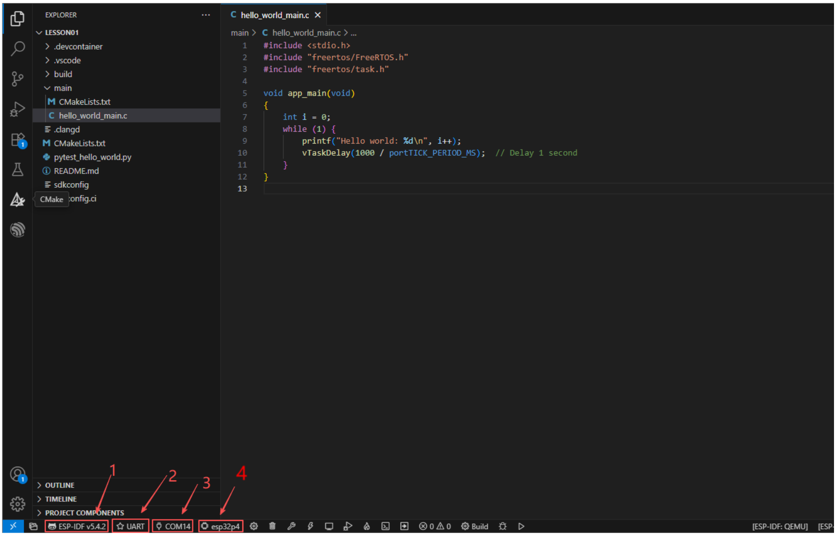
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson01-Print_Hello_World

Programming Steps

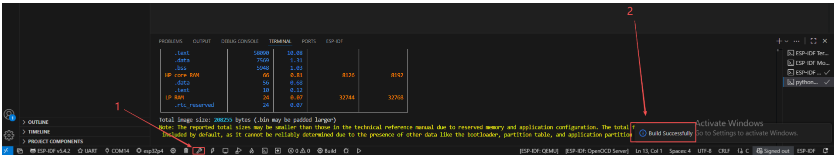
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable. (Connect UART0)



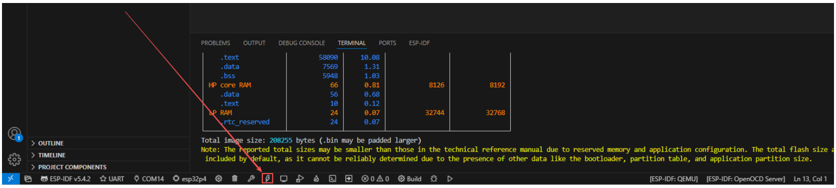
- In order of priority, select the ESP-IDF version 5.4.2 that you are currently using.
- We are using serial flash programming, so select UART.
- Since the serial port number displayed may vary depending on your device, after clicking 3, select the serial port that belongs to your own device.
- Make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



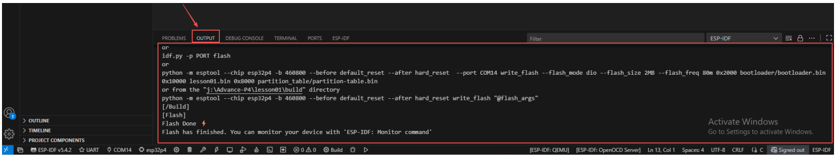
- After configuring 1, 2, 3, and 4 as mentioned above, we will proceed to compile the project to check if there are any issues with the code.
- First, click on 1 in the picture below, which represents the function of compiling the code.
- Wait for a while, after the code is compiled, you will be able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



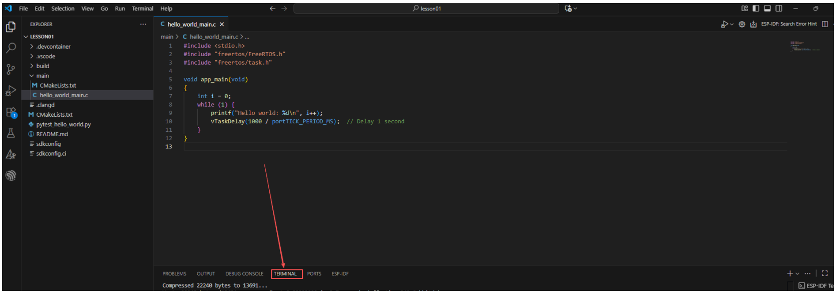
- Then, click the "Burn" button.



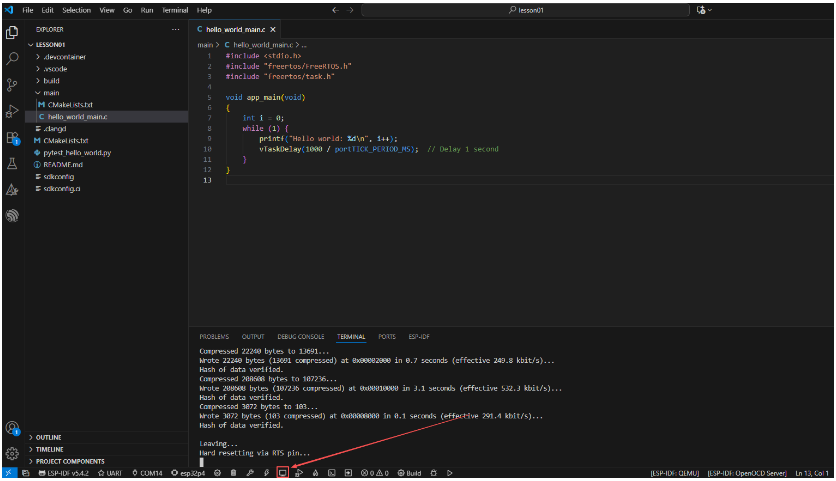
- After waiting for a while, you will be able to see from the displayed information on the output that the code has been uploaded successfully.

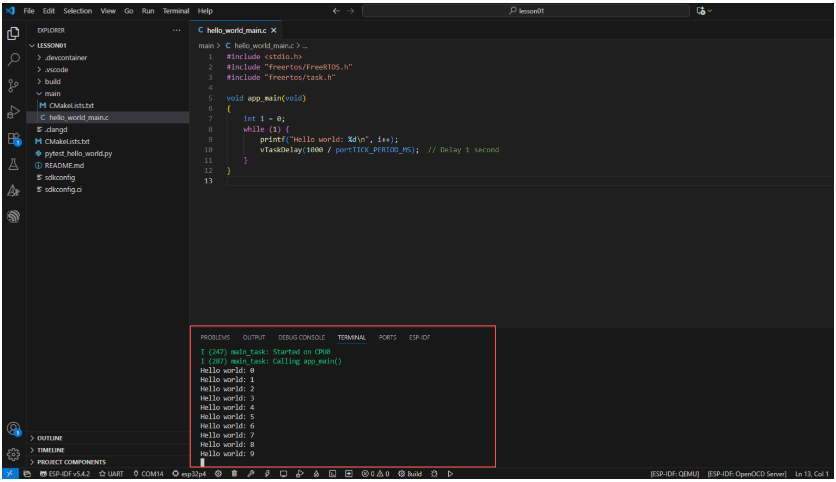


- Of course, you can also see from the upload process displayed on the terminal that your code has been uploaded successfully.



- 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

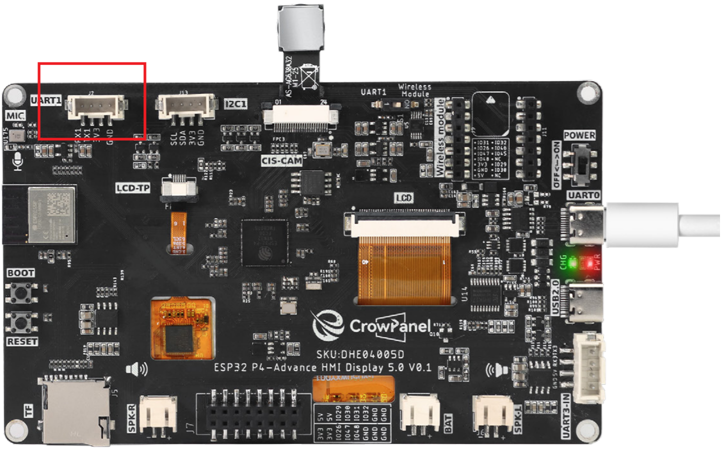
Introduction

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.

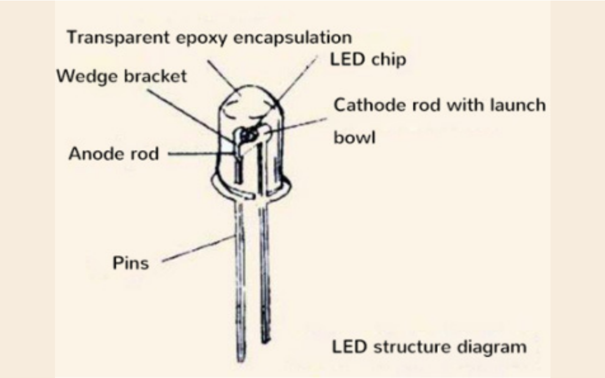
The GPIO functions of ESP32-P4 provide powerful hardware support for developers. In this chapter, we will delve into the application and configuration of GPIO through an example of lighting an LED.

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.



LED structure diagram

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.

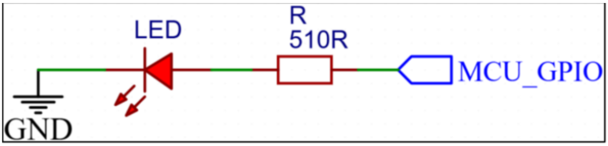
2. Principle of LED Lighting Driver

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.

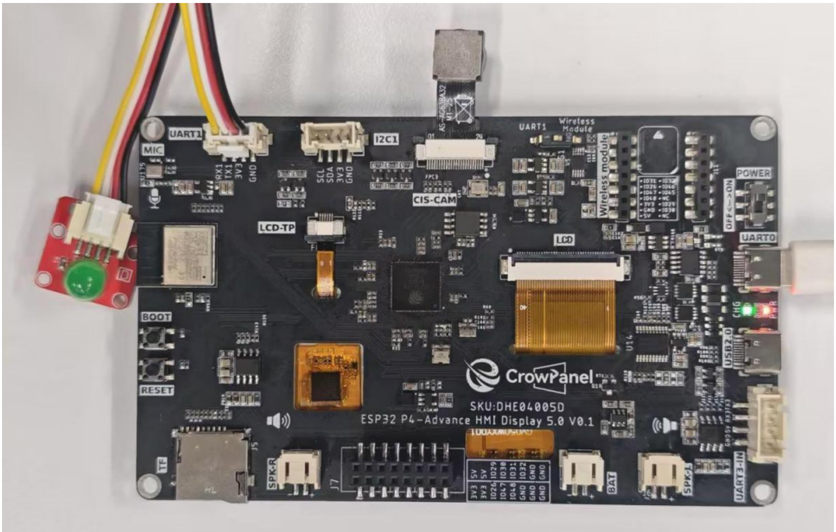
2. Power current configuration. This refers to the voltage and current provided by the microcontroller, and the current output will be applied to the LED. If the LED is driven directly by the GPIO of the microcontroller, its driving capability is relatively weak and may not be able to provide sufficient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

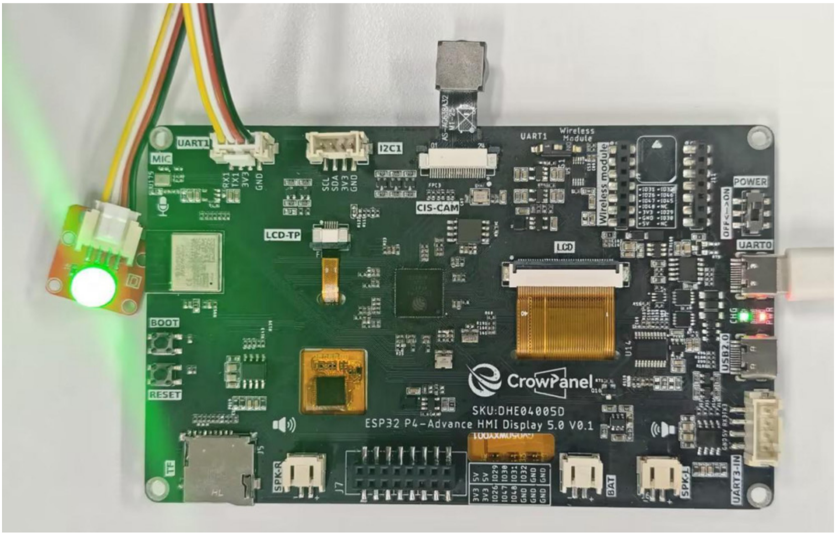


The LED circuit on the ESP32-P4 development board adopts the "current receiving" configuration. This approach avoids the microcontroller directly powering and supplying current to the LED, thereby effectively reducing the load on the microcontroller. This enables the microcontroller to focus more on 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

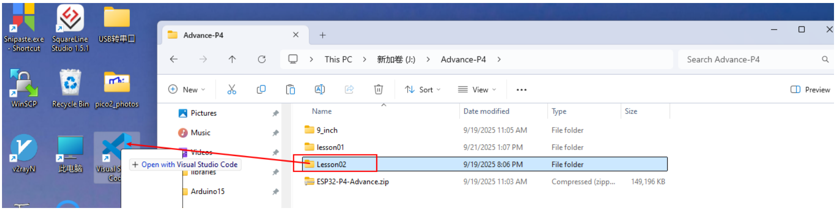
Now let's talk about how the overall code framework is structured and connected after adding the bsp_extra component?

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-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson02-Turn_on_the%20LED

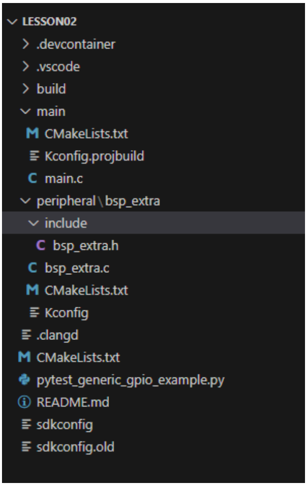
Then, drag the code of this lesson into VS Code and open the project file.



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.

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

#include <string.h>      // Standard C library for string handling functions
#include <stdint.h>      // Standard C library for fixed-width integer types
#include "esp_log.h"     // ESP-IDF logging library for debug/info/error logs
#include "esp_err.h"     // ESP-IDF error code definitions and handling utilities
#include "driver/gpio.h" // ESP-IDF GPIO driver for configuring and controlling pins

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

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

```
/*——————————————————————————————————————————Variable declaration——————————————————————————————————————————*/

#define EXTRA_TAG "EXTRA"                          // Define log tag name "EXTRA" used for identifying log messages
#define EXTRA_INFO(fmt, ...) ESP_LOGI(EXTRA_TAG, fmt, ##__VA_ARGS__)   // Macro for info-level logging with tag "EXTRA"
#define EXTRA_DEBUG(fmt, ...) ESP_LOGD(EXTRA_TAG, fmt, ##__VA_ARGS__)  // Macro for debug-level logging with tag "EXTRA"
#define EXTRA_ERROR(fmt, ...) ESP_LOGE(EXTRA_TAG, fmt, ##__VA_ARGS__)  // Macro for error-level logging with tag "EXTRA"

esp_err_t gpio_extra_init();                       // Function declaration for initializing GPIO
esp_err_t gpio_extra_set_level(bool level);        // Function declaration for setting GPIO output level (high/low)

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

- 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".

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

- The gpio_extra_init() function is used to configure GPIO48 of the ESP32-P4 as an output pin.

```
esp_err_t gpio_extra_init()                         // Function to initialize GPIO48 as output
{
    esp_err_t err = ESP_OK;                         // Variable to store error code, initialized to ESP_OK
    const gpio_config_t gpio_cofig = {              // Define GPIO configuration structure
        .pin_bit_mask = (1ULL << 48),               // Select GPIO48 by setting bit 48 in the mask
        .mode = GPIO_MODE_OUTPUT,                   // Configure GPIO48 as output mode
        .pull_up_en = false,                        // Disable internal pull-up resistor
        .pull_down_en = false,                      // Disable internal pull-down resistor
        .intr_type = GPIO_INTR_DISABLE,             // Disable GPIO interrupt for this pin
    };
    err = gpio_config(&gpio_cofig);                 // Apply the configuration using ESP-IDF API
    return ESP_OK;                                  // Always return ESP_OK (ignores actual error code)
}
```

- 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.

```
esp_err_t gpio_extra_set_level(bool level)      // Function to set output level of GPIO48
{
    gpio_set_level(48, level);                  // Set GPIO48 output to high (1) or low (0) depending on 'level'
    return ESP_OK;                              // Return ESP_OK (does not check for errors)
}
```

- 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:

peripheral > bsp_extra > M CMakeLists.txt

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES 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

main.c | CMakeLists.txt | bsp_extra.c | bsp_extra.h

peripheral > bsp_extra > CMakeLists.txt

```cmake
FILE(GLOB_RECURSE component_sources "*.c")

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:

peripheral > bsp_extra > CMakeLists.txt

```cmake
FILE(GLOB_RECURSE component_sources "*.c")

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.

main.c | CMakeLists.txt | bsp_extra.c | bsp_extra.h

peripheral > bsp_extra > include > bsp_extra.h > ...

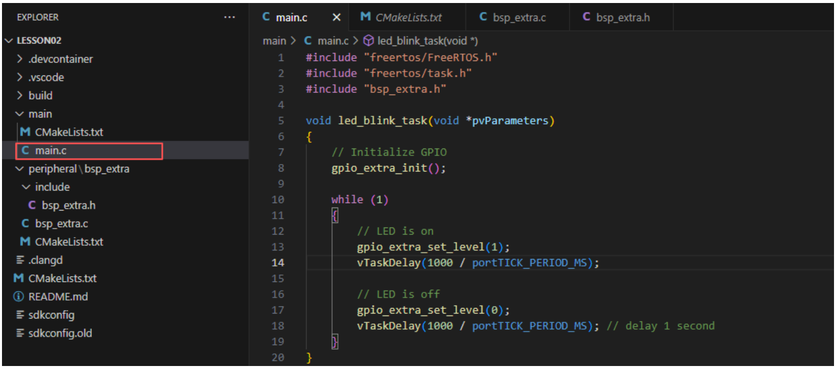
```c
#ifndef _BSP_EXTRA_H_
#define _BSP_EXTRA_H_

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

#include <string.h>      // Standard C library for string handling functions
#include <stdint.h>      // Standard C library for fixed-width integer types
#include "esp_log.h"     // ESP-IDF logging library for debug/info/error logs
#include "esp_err.h"     // ESP-IDF error code definitions and handling utilities
#include "driver/gpio.h" // ESP-IDF GPIO driver for configuring and controlling pins

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

/*——————————————————————————————————————Variable declaration——————————————————————————————————————*/
```

- "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.



- 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().

```
main > main.c > led_blink_task(void *)
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "bsp_extra.h"
```

- 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.

```
void led_blink_task(void *pvParameters)
{
    // Initialize GPIO
    gpio_extra_init();
```

- 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)

```c
void led_blink_task(void *pvParameters)
{
    // Initialize GPIO
    gpio_extra_init();

    while (1)
    {
        // LED is on
        gpio_extra_set_level(1);
        vTaskDelay(1000 / portTICK_PERIOD_MS);

        // LED is off
        gpio_extra_set_level(0);
        vTaskDelay(1000 / portTICK_PERIOD_MS); // delay 1 second
    }
}
```

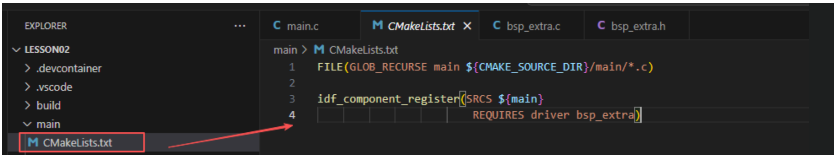
- Then, the delay function from the FreeRTOS library is called to delay for one second.

```c
    while (1)
    {
        // LED is on
        gpio_extra_set_level(1);
        vTaskDelay(1000 / portTICK_PERIOD_MS);

        // LED is off
        gpio_extra_set_level(0);
        vTaskDelay(1000 / portTICK_PERIOD_MS); // delay 1 second
    }
}
```

- 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.

```c
void led_blink_task(void *pvParameters)
    while (1)
    {
        // LED is on
        gpio_extra_set_level(1);
        vTaskDelay(1000 / portTICK_PERIOD_MS);

        // LED is off
        gpio_extra_set_level(0);
        vTaskDelay(1000 / portTICK_PERIOD_MS); // delay 1 second
    }
}

void app_main(void)
{
    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.

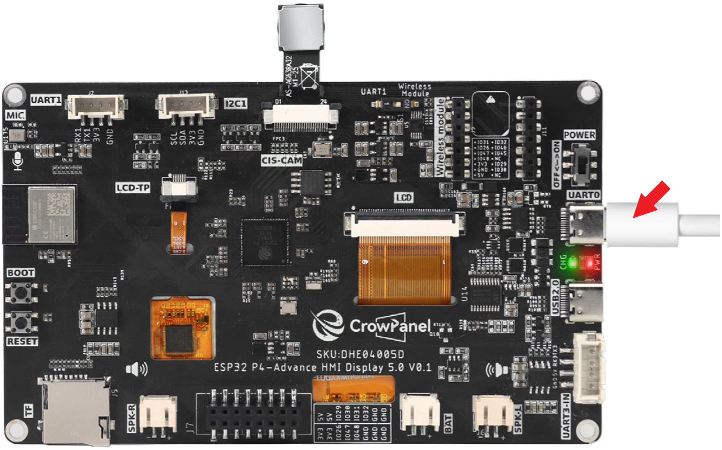
Complete Code

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

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson02-Turn_on_the%20LED

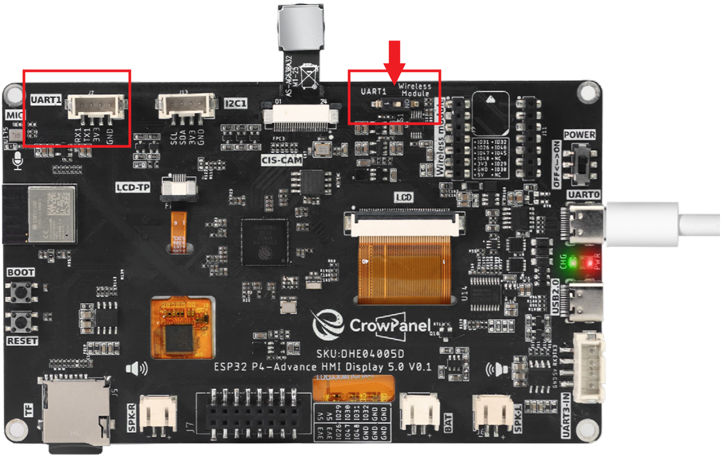
Programming Steps

- 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.

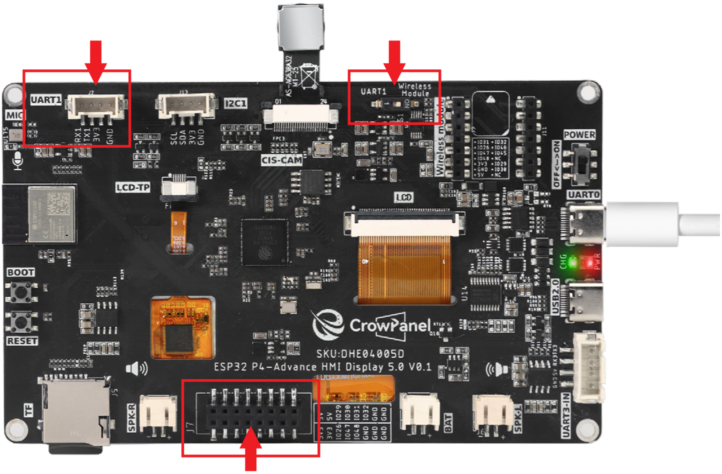


Then, switch the toggle switch on the 5-inch Advance-P4 to the UART1 position.

Only in this way can the UART1 interface be used.



- This is the design on the hardware side.



Switch to UART1 port:

Among the three interfaces shown in the figure, only the UART1 interface can be used at this time.

Alternatively, the expansion header at the bottom can also be used.

That is, either the UART1 interface or the expansion header can be used, but not both.

Switch to Wireless Module port:

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

Alternatively, the expansion header at the bottom can also be used.

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

Summary:

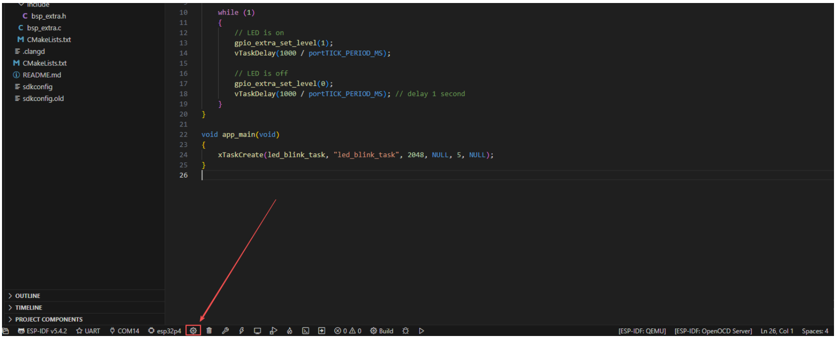
The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.

The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

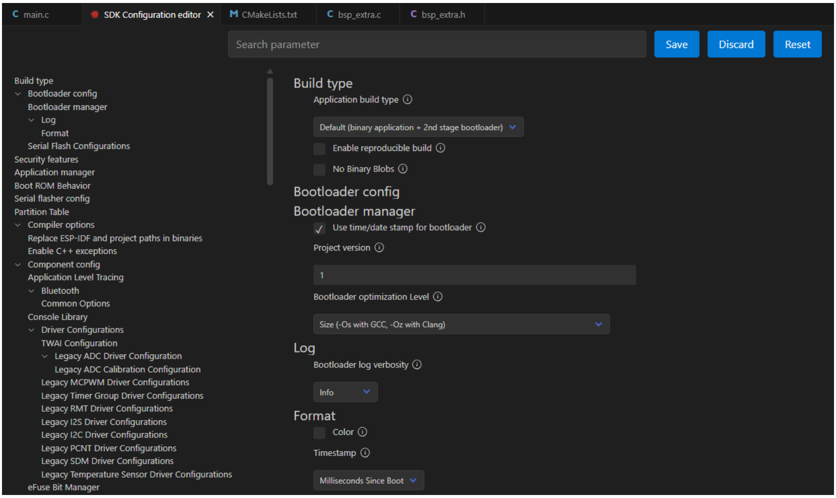
Note: The H2 and C6 wireless modules can be used simultaneously with UART1.

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

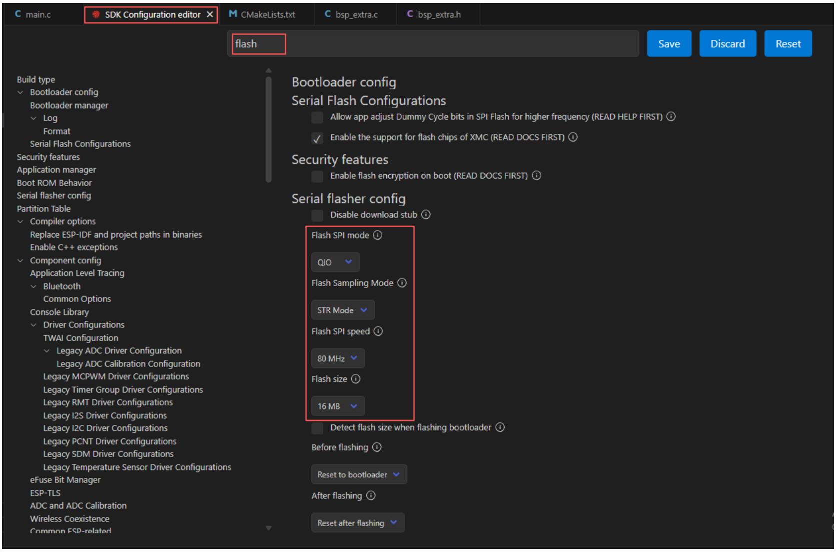
- Here, following the steps in the first section, we first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



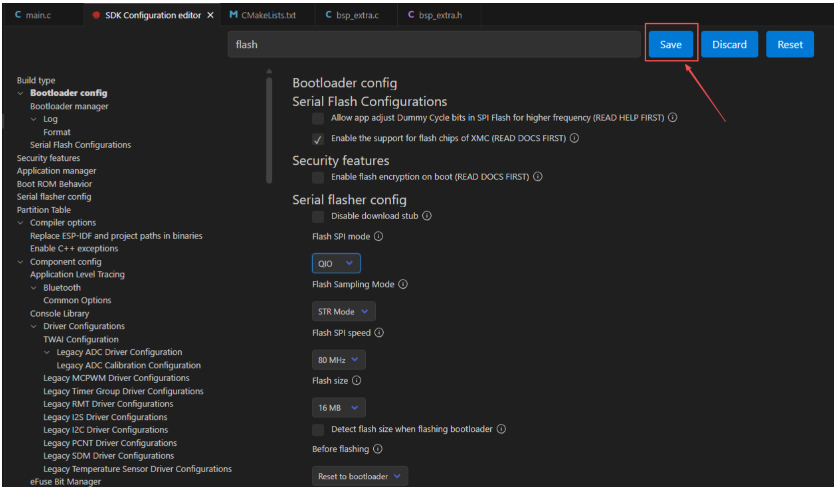
- Wait for a moment for the loading process to complete, and then you can proceed with the relevant SDK configuration.



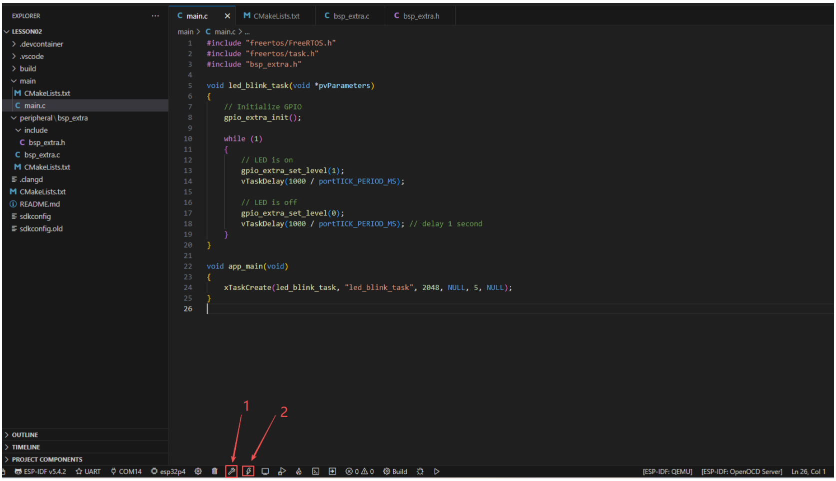
- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



- 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)

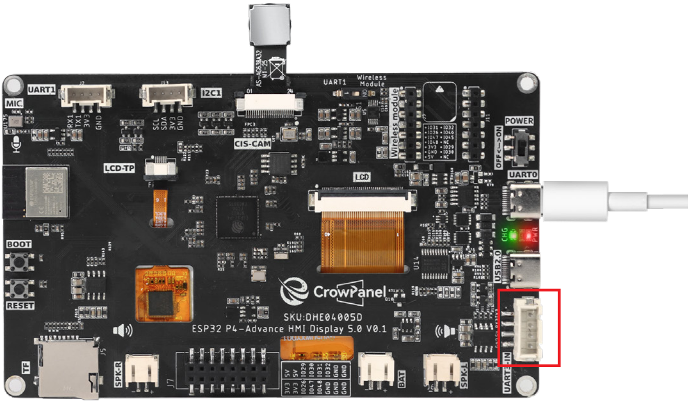


- After waiting for a while, you will be able to see the LED connected to UART1 on your Advance-P4 turning on and off, remaining off for one second, and repeating this process over and over again.

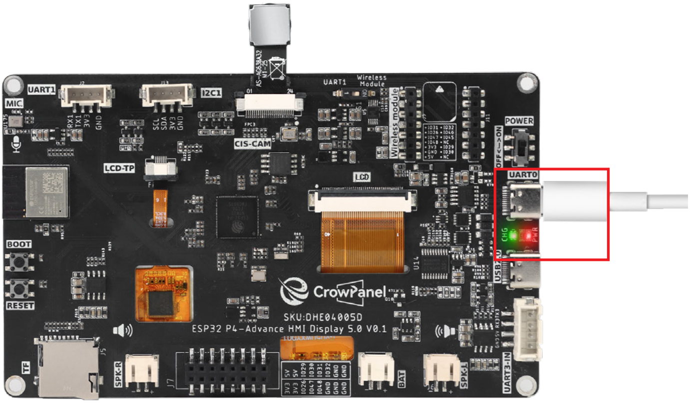
Lesson 03
UART3-IN interface (external power supply)

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.



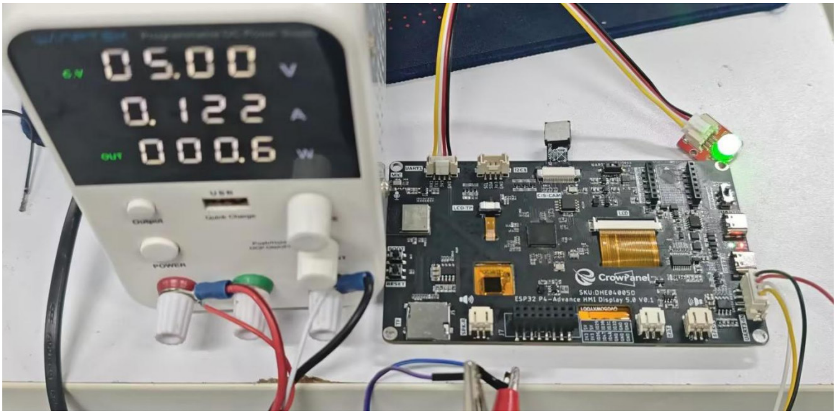
Then we come back to the UART3-IN interface. This interface is similar in function to the UART0 interface we just discussed. It can supply power, but it cannot upload code.

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

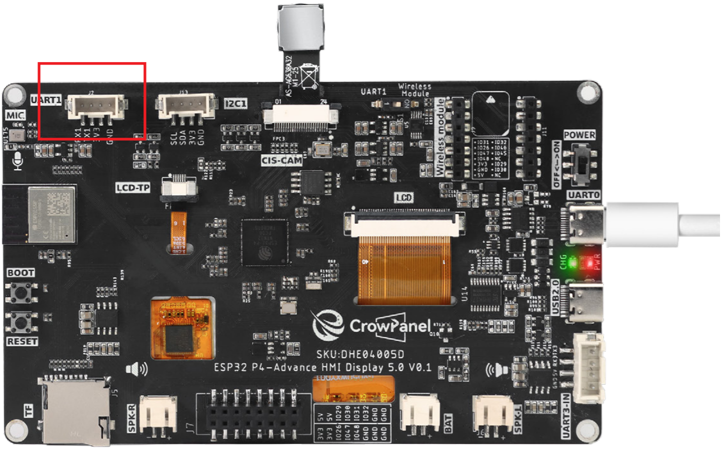
Introduction

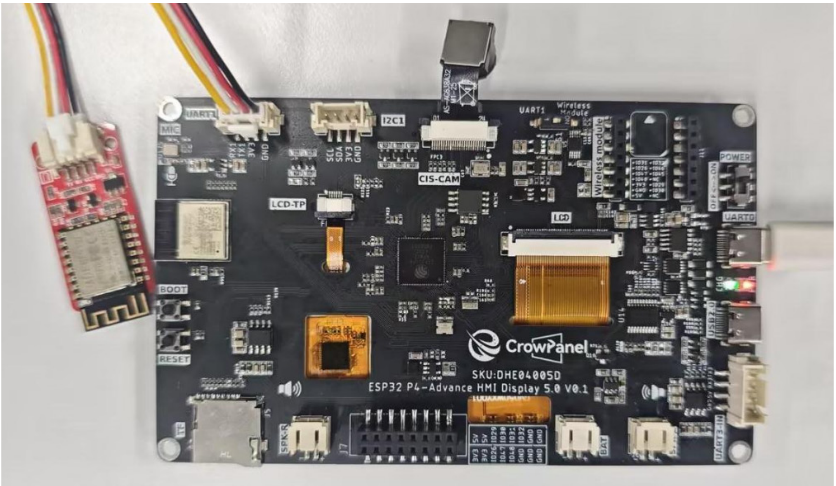
In this class, we will start teaching you how to use the serial port component. We will communicate with the Wi-Fi serial module through the UART1 interface on the Advance-P4.

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

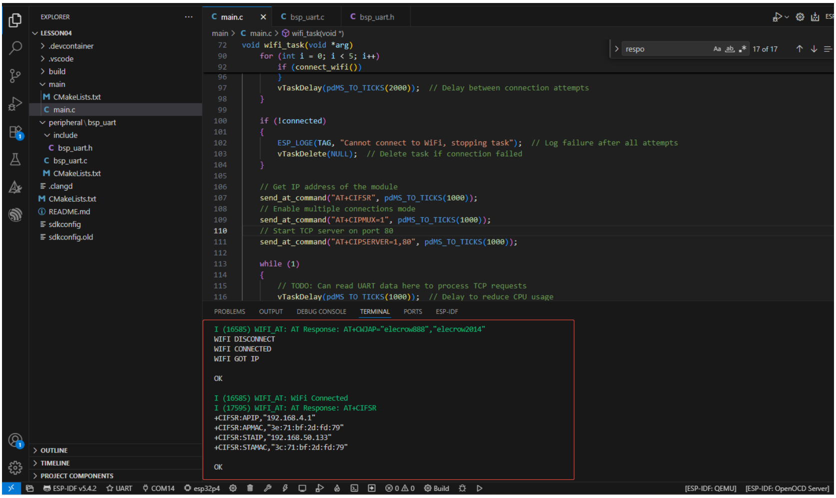
The UART1 interface on the Advance-P4





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

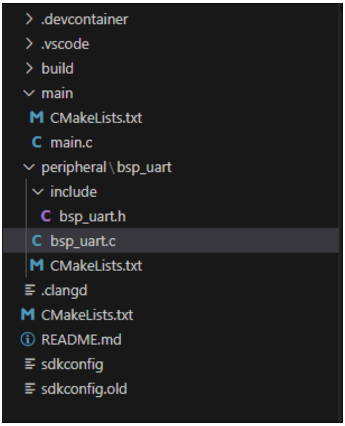
The main focus of this class is on how to use the serial port. Here, we will provide everyone with a new component called bsp_uart. This component is mainly used for initializing the serial port, configuring the serial port, and providing related interface usage. As you know, you can call the interfaces 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-5inch-ESP32-P4-HMI-AI-Display-800x480-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.

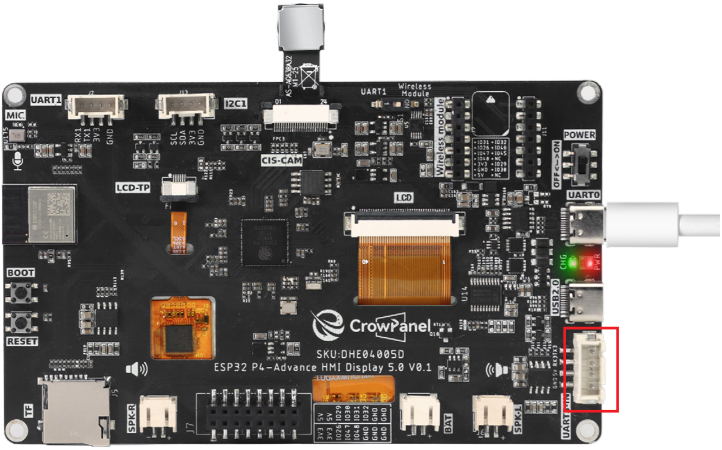
```c
/*------------------------------------Header file declaration------------------------------------*/
#include <string.h>               // Standard C library for string manipulation
#include <stdint.h>               // Standard integer type definitions (e.g., uint8_t, int32_t)
#include "freertos/FreeRTOS.h"    // FreeRTOS core definitions
#include "freertos/task.h"        // FreeRTOS task management APIs
#include "esp_log.h"              // ESP-IDF logging library
#include "esp_err.h"              // ESP-IDF error codes
#include "driver/uart.h"          // ESP-IDF UART driver APIs
/*------------------------------------Header file declaration end------------------------------------*/
```

- 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.)

```c
/*------------------------------------Variable declaration------------------------------------*/
#define UART_TAG "UART"              // Logging tag for UART module
#define UART_INFO(fmt, ...) ESP_LOGI(UART_TAG, fmt, ##__VA_ARGS__)   // Macro for UART info log
#define UART_DEBUG(fmt, ...) ESP_LOGD(UART_TAG, fmt, ##__VA_ARGS__)  // Macro for UART debug log
#define UART_ERROR(fmt, ...) ESP_LOGE(UART_TAG, fmt, ##__VA_ARGS__)  // Macro for UART error log

#define UART_IN_EXTRA_GPIO_TXD 27    // Define GPIO number 27 as UART TXD pin for input extra UART
#define UART_IN_EXTRA_GPIO_RXD 28    // Define GPIO number 28 as UART RXD pin for input extra UART

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

typedef enum
{
    UART_SCAN = 1,                   // UART state: scanning or waiting for input
    UART_DECODE,                     // UART state: decoding received data
    UART_ERR,                        // UART state: error occurred
} uart_state;

int SendData(const char *data);      // Function to send data over UART
esp_err_t uart_init();               // Function to initialize UART

/*------------------------------------Variable declaration end------------------------------------*/
#endif                               // End of header guard
```

- 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.)

```c
#define UART_IN_EXTRA_GPIO_TXD 27    // Define GPIO number 27 as UART TXD pin for input extra UART
#define UART_IN_EXTRA_GPIO_RXD 28    // Define GPIO number 28 as UART RXD pin for input extra UART
```



- 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.

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

- 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.

```
int SendData(const char *data)        // Function to send a string of data through UART2
{
    const int len = strlen(data);     // Get the length of the input string
    const int txBytes = uart_write_bytes(UART_NUM_2, data, len);  // Write string to UART2
    return txBytes;                   // Return number of bytes actually sent
}

esp_err_t uart_init()                 // Function to initialize UART2
{
    esp_err_t err = ESP_OK;           // Variable to store error status, default to ESP_OK
    const uart_config_t uart_config = {   // UART configuration structure
        .baud_rate = 115200,                 // Set baud rate to 115200
        .data_bits = UART_DATA_8_BITS,       // 8 data bits per frame
        .parity = UART_PARITY_DISABLE,       // Disable parity check
        .stop_bits = UART_STOP_BITS_1,       // 1 stop bit
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE, // Disable hardware flow control
        .source_clk = UART_SCLK_DEFAULT,       // Use default UART clock source
    };

    err = uart_driver_install(UART_NUM_2, 1024 * 2, 0, 0, NULL, 0);    // Install UART2 driver with RX buffer size 2048 bytes
    if (err != ESP_OK)             // Check if driver installation failed
    {
        UART_ERROR("extra uart driver install fail");   // Log error if installation failed
        return err;                                    // Return error code
    }
    uart_set_pin(UART_NUM_2, UART1_EXTRA_GPIO_TXD, UART1_EXTRA_GPIO_RXD, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    // Configure UART2 TX and RX pins, keep RTS/CTS unchanged
    err = uart_param_config(UART_NUM_2, &uart_config);  // Apply UART parameter configuration
    if (err != ESP_OK)                                  // Check if configuration failed
        return err;                                     // Return error code

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

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.

```c
int SendData(const char *data)          // Function to send a string of data through UART2
{
    const int len = strlen(data);       // Get the length of the input string
    const int txBytes = uart_write_bytes(UART_NUM_2, data, len);  // Write string to UART2
    return txBytes;                     // Return number of bytes actually sent
}
```

main.c | bsp_uart.c | bsp_uart.h

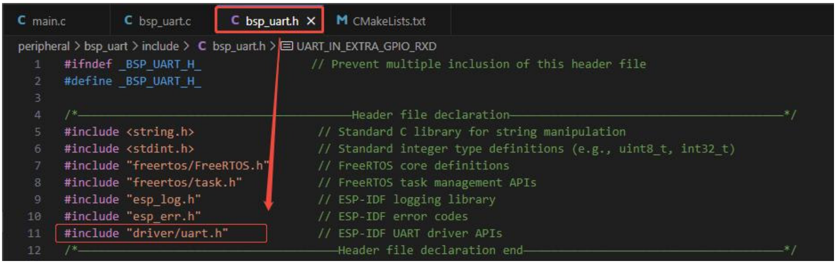
peripheral > bsp_uart > include > bsp_uart.h > UART_IN_EXTRA_GPIO_RXD

```c
#ifndef _BSP_UART_H_                    // Prevent multiple inclusion of this header file
#define _BSP_UART_H_

/*——————————————————————————Header file declaration——————————————————————————*/
#include <string.h>                     // Standard C library for string manipulation
#include <stdint.h>                     // Standard integer type definitions (e.g., uint8_t, int32_t)
#include "freertos/FreeRTOS.h"          // FreeRTOS core definitions
#include "freertos/task.h"              // FreeRTOS task management APIs
#include "esp_log.h"                    // ESP-IDF logging library
#include "esp_err.h"                    // ESP-IDF error codes
#include "driver/uart.h"                // ESP-IDF UART driver APIs
/*——————————————————————————Header file declaration end——————————————————————*/

/*——————————————————————————Variable declaration—————————————————————————————*/
#define UART_TAG "UART"                 // Logging tag for UART module
#define UART_INFO(fmt, ...) ESP_LOGI(UART_TAG, fmt, ##__VA_ARGS__)   // Macro for UART info log
#define UART_DEBUG(fmt, ...) ESP_LOGD(UART_TAG, fmt, ##__VA_ARGS__)  // Macro for UART debug log
#define UART_ERROR(fmt, ...) ESP_LOGE(UART_TAG, fmt, ##__VA_ARGS__)  // Macro for UART error log

#define UART_IN_EXTRA_GPIO_TXD 27       // Define GPIO number 27 as UART TXD pin for input extra UART
#define UART_IN_EXTRA_GPIO_RXD 28       // Define GPIO number 28 as UART RXD pin for input extra UART

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

typedef enum
{
    UART_SCAN = 1,                      // UART state: scanning or waiting for input
    UART_DECODE,                        // UART state: decoding received data
    UART_ERR,                           // UART state: error occurred
} uart_state;
```

- 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.


```cmake
FILE(GLOB_RECURSE component_sources "*.c")

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

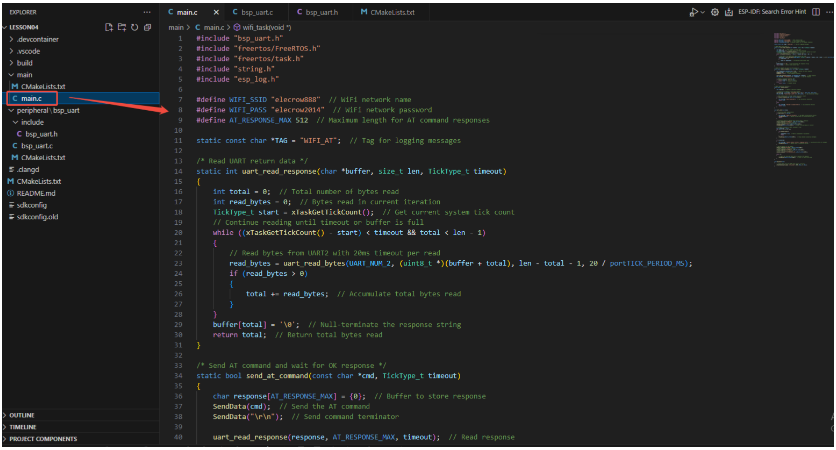
- The reason why this is called "driver" is that we have called it in the "bsp_uart.h" file (for other libraries that are system libraries, there is no need to add anything).



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 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.c > ...

```c
#include "bsp_uart.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "string.h"
#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.

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

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

```c
/* Read UART return data */
static int uart_read_response(char *buffer, size_t len, TickType_t timeout)
{
    int total = 0;  // Total number of bytes read
    int read_bytes = 0;  // Bytes read in current iteration
    TickType_t start = xTaskGetTickCount();  // Get current system tick count
    // Continue reading until timeout or buffer is full
    while ((xTaskGetTickCount() - start) < timeout && total < len - 1)
    {
        // Read bytes from UART2 with 20ms timeout per read
        read_bytes = uart_read_bytes(UART_NUM_2, (uint8_t *)(buffer + total), len - total - 1, 20 / portTICK_PERIOD_MS);
        if (read_bytes > 0)
        {
            total += read_bytes;  // Accumulate total bytes read
        }
    }
    buffer[total] = '\0';  // Null-terminate the response string
    return total;  // Return total bytes read
}
```

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.

```
I (25585) WIFI_AT: AT Response: AT+CIPMUX=1

OK

I (26585) WIFI_AT: AT Response: AT+CIPSERVER=1,80

OK
```

- 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.

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

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

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

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.

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

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

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

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;

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

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

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

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

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

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

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

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

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

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

- 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.

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

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

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

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

```

- 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.

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

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

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

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

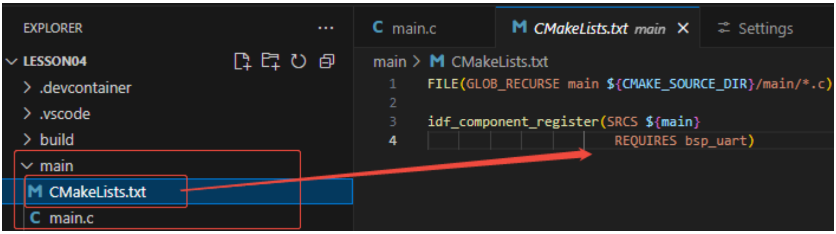
- Finally, it enters an infinite loop, retaining the interface for subsequent processing of TCP requests, and reducing CPU usage through delay, thereby completing the entire process of WiFi network management and services.

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

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

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

- 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.

```c
void app_main(void)
{
    // Create WiFi task with 4096 bytes stack, priority 5
    xTaskCreate(wifi_task, "wifi_task", 4096, NULL, 5, NULL);
}
```

- 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".



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.

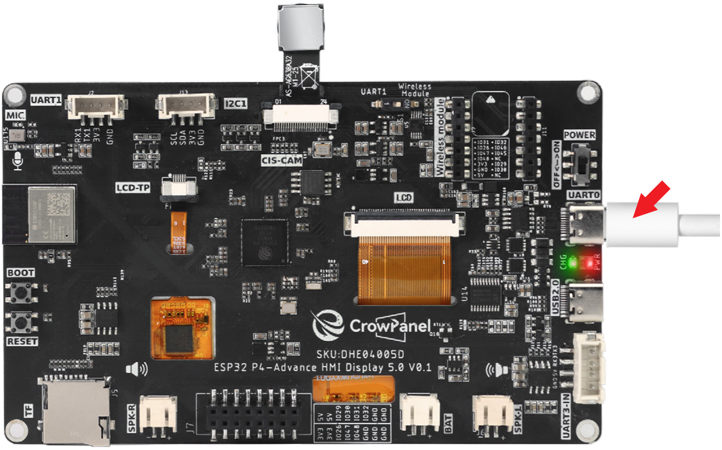
Complete Code

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

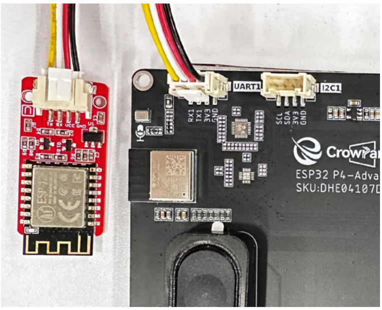
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson04-Serial_port_usage

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.

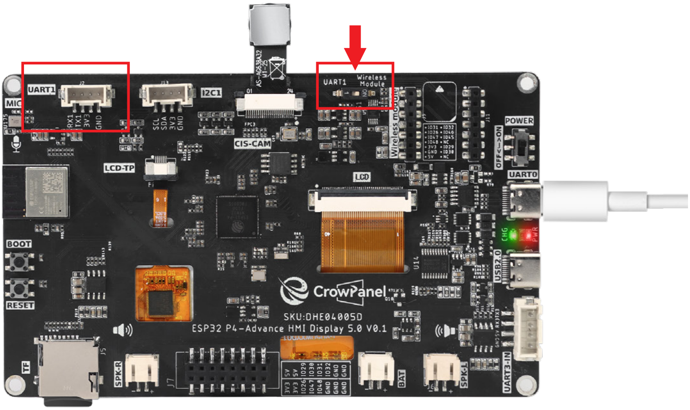


- Then, connect an ESP8266 wifi module to the UART1 interface.
- (Connect the VCC of UART1 interface to the VCC pin of the wifi module)
- (Connect the GND of UART1 interface to the GND pin of the wifi module)
- (Turn the TX of UART1 interface to the RX pin of the wifi module) (Cross connection)
- (Turn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

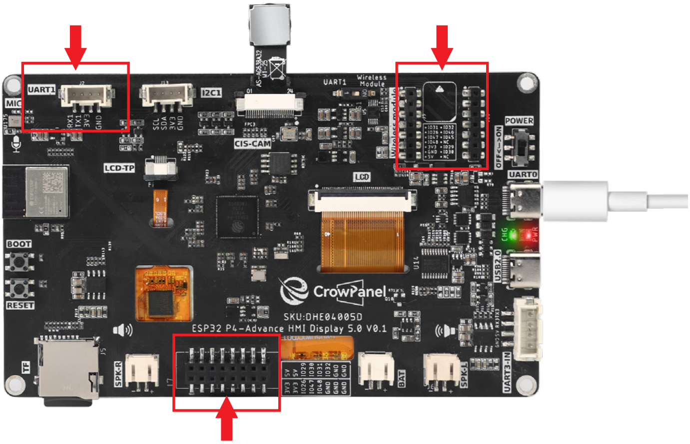


Then, switch the toggle switch on the 5-inch Advance-P4 to the UART1 position.

Only in this way can the UART1 interface be used.



- This is the design on the hardware side.



Switch to UART1 port:

Among the three interfaces shown in the figure, only the UART1 interface can be used at this time.

Alternatively, the expansion header at the bottom can also be used.

That is, either the UART1 interface or the expansion header can be used, but not both.

Switch to Wireless Module port:

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

Alternatively, the expansion header at the bottom can also be used.

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

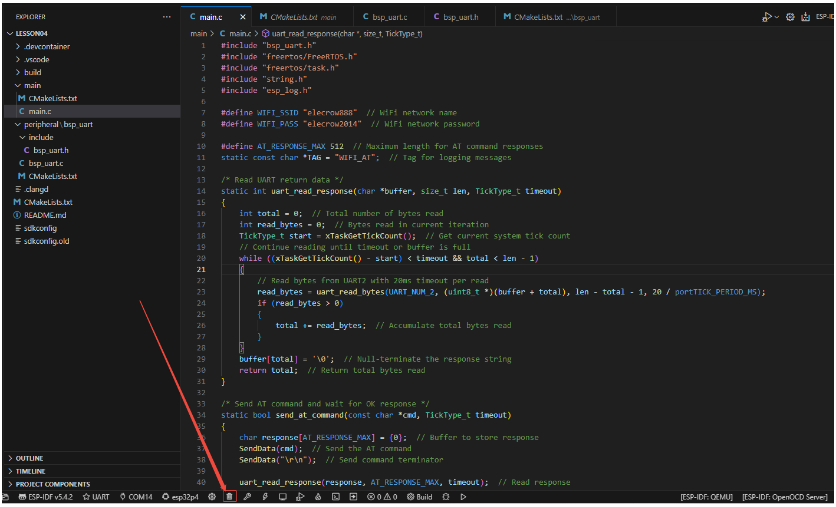
Summary:

The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.

The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

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

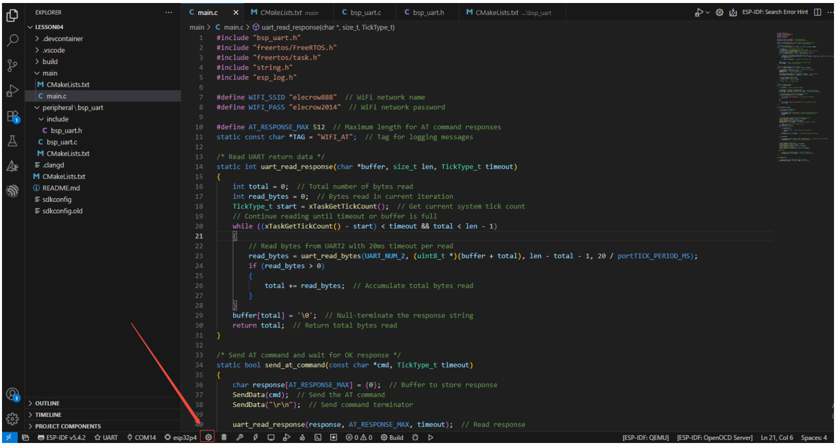
- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



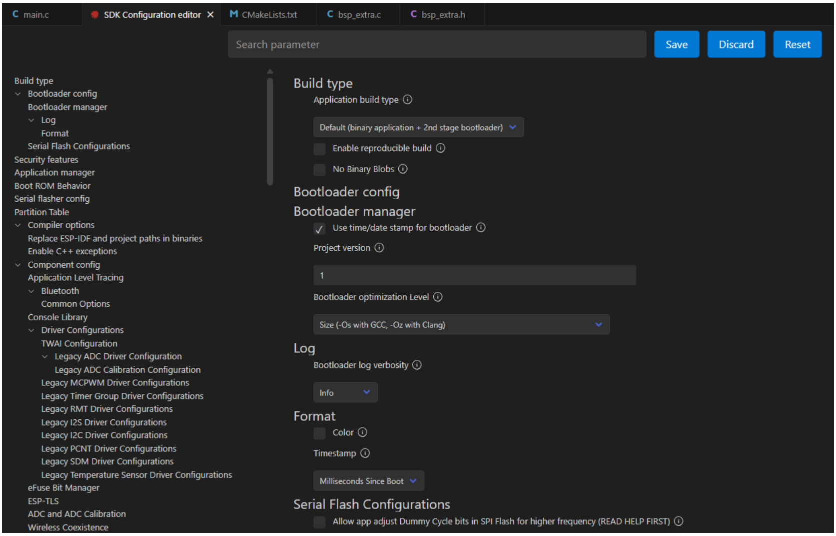
- Here, following the steps in the first section, we first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.

- Then here we need to configure the SDK.

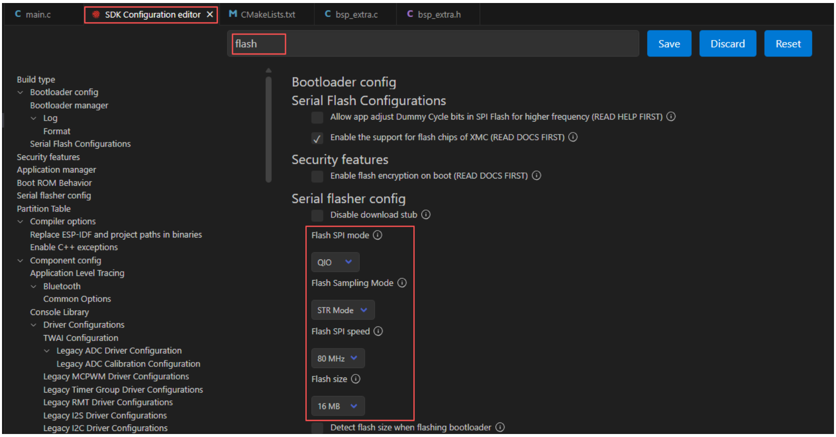
- Click the icon in the picture below.



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



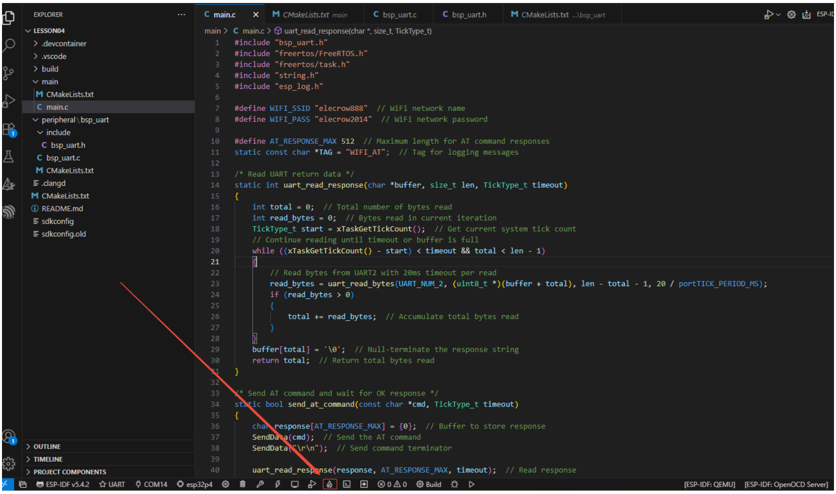
- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



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

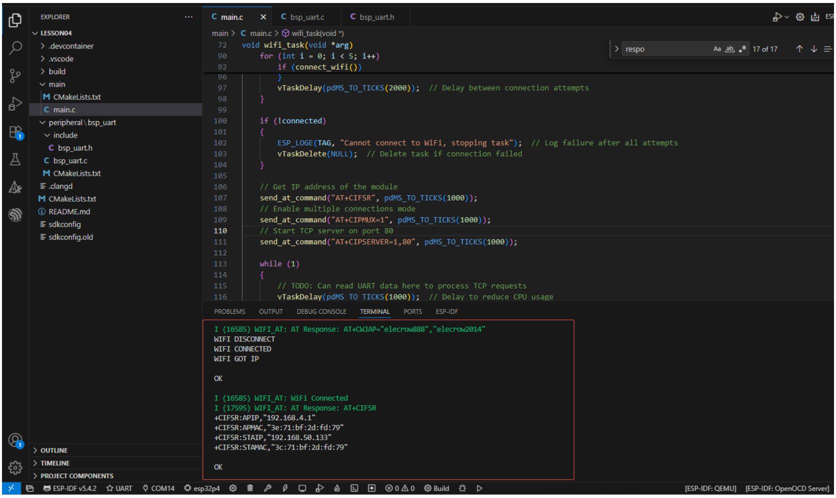
- After that, we will compile and burn the code (which was explained in detail in the first class)

- 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

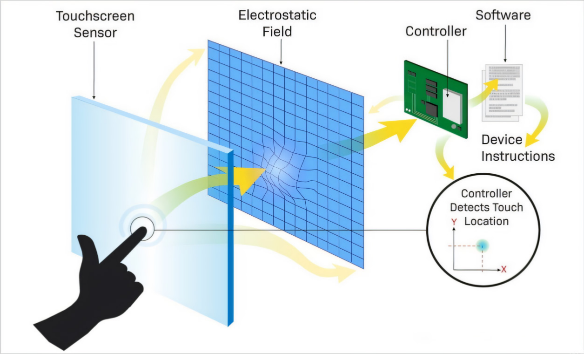
In this class, we will gradually start to use multiple components together. We hope this will help everyone gain a deeper understanding of ESP-IDF and ESP32-P4.

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".

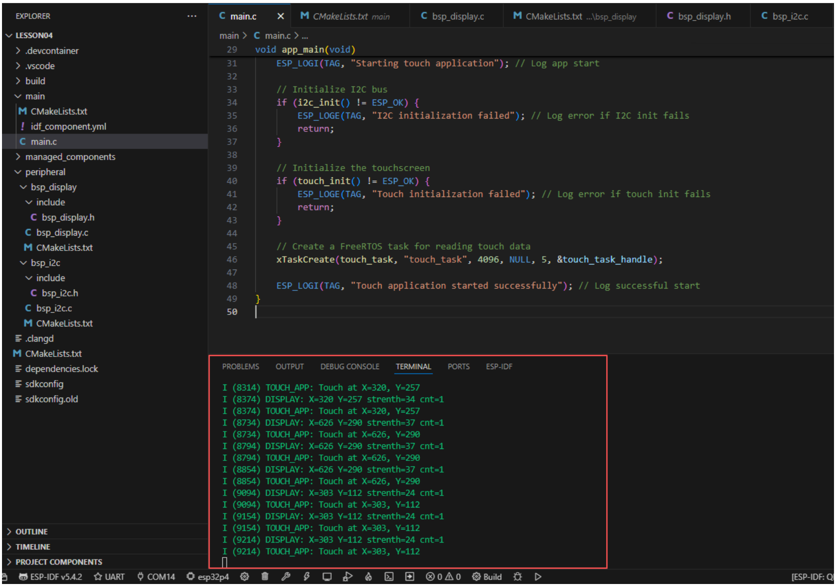
After that, the GT911 sends the calculated touch point coordinate information to the connected main processor (such as an ESP32 microcontroller) according to the pre-set communication protocol (such as I2C, SPI, etc.).

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.



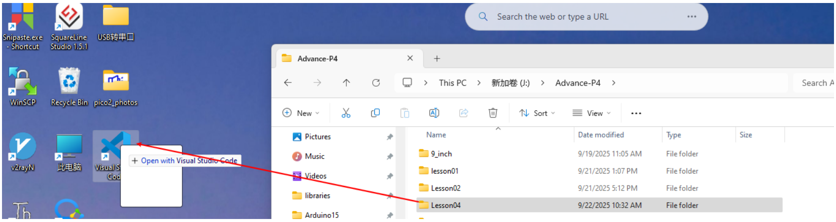
Key Explanations

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

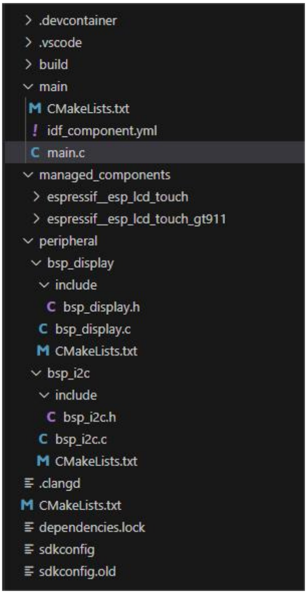
It's actually not difficult. Once you understand how one component is used, the two components are similar. First, click on the Github link below to download the code for this lesson.

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-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".

Screen touch driver code

- The screen touch driver consists of two files: "bsp_display.c" and "bsp_display.h".
- Next, we will first analyze the "bsp_display.h" program.
- "bsp_display.h" is a header file for the display and touch screen driver module, mainly used for:
- Making the functions, macros, and variable declarations implemented in "bsp_display.c" available for use by external programs
- Allowing other .c files to simply include "bsp_display.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_display.h" file for centralized management.

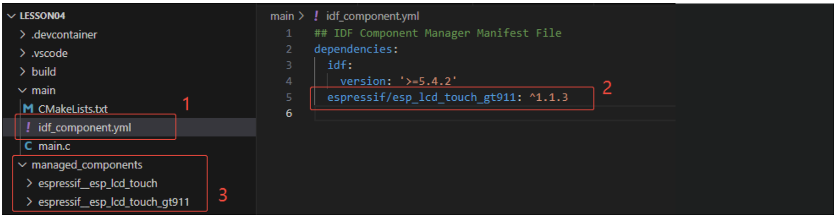
```
/*------------------------------------Header file declaration------------------------------------*/
#include "esp_log.h"
#include "esp_err.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_lcd_touch_gt911.h"
#include "bsp_i2c.h"
/*------------------------------------Header file declaration end------------------------------------*/
```

- Such as esp_lcd_touch_gt911.h

peripheral > bsp_display > include > C bsp_display.h > ...

```
#ifndef _BSP_DISPLAY_H_
#define _BSP_DISPLAY_H_
/*------------------------------------Header file declaration------------------------------------*/
#include "esp_log.h"
#include "esp_err.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_lcd_touch_gt911.h"
#include "bsp_i2c.h"
/*------------------------------------Header file declaration end------------------------------------*/

/*------------------------------------Variable declaration------------------------------------*/
#define DISPLAY_TAG "DISPLAY"
#define DISPLAY_INFO(fmt, ...) ESP_LOGI(DISPLAY_TAG, fmt, ##__VA_ARGS__)
#define DISPLAY_DEBUG(fmt, ...) ESP_LOGD(DISPLAY_TAG, fmt, ##__VA_ARGS__)
#define DISPLAY_ERROR(fmt, ...) ESP_LOGE(DISPLAY_TAG, fmt, ##__VA_ARGS__)

#define V_size 600
#define H_size 1024
```

- 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.



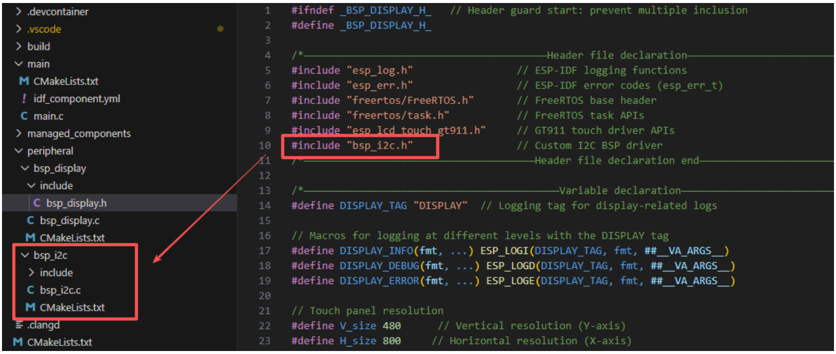
- When the project is compiled in the future, it will download the esp_lcd_touch_gt911 library version 1.1.3. After the download, these network components will be saved in the "managed_components" folder. (This is automatically generated after filling in the version number.)

- Then we will return to the "bsp_display.h" file.

- We can see that the "bsp_i2c.h" file is also included in it.

```
/*————————————————————Header file declaration————————————————————*/
#include "esp_log.h"                // ESP-IDF logging functions
#include "esp_err.h"                // ESP-IDF error codes (esp_err_t)
#include "freertos/FreeRTOS.h"      // FreeRTOS base header
#include "freertos/task.h"          // FreeRTOS task APIs
#include "esp_lcd_touch_gt911.h"    // GT911 touch driver APIs
#include "bsp_i2c.h"                // Custom I2C BSP driver
/*——————————————————Header file declaration end——————————————————*/

/*——————————————————————Variable declaration——————————————————————*/
#define DISPLAY_TAG "DISPLAY"   // Logging tag for display-related logs
```

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



- 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.

```c
/*———————————————————————————————Variable declaration———————————————————————————————*/
#define DISPLAY_TAG "DISPLAY"  // Logging tag for display-related logs

// Macros for logging at different levels with the DISPLAY tag
#define DISPLAY_INFO(fmt, ...) ESP_LOGI(DISPLAY_TAG, fmt, ##__VA_ARGS__)
#define DISPLAY_DEBUG(fmt, ...) ESP_LOGD(DISPLAY_TAG, fmt, ##__VA_ARGS__)
#define DISPLAY_ERROR(fmt, ...) ESP_LOGE(DISPLAY_TAG, fmt, ##__VA_ARGS__)

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

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

// Public API: Get the latest touch coordinates and press state
void get_coor(uint16_t* x, uint16_t* y, bool* press);

// Internal API: Update the stored coordinates and press state (only used inside .c file)
static void set_coor(uint16_t x, uint16_t y, bool press);
```

- 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.

peripheral > bsp_display > M CMakeLists.txt

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES esp_lcd_touch_gt911 bsp_i2c)
```

- 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.

peripheral > bsp_display > M CMakeLists.txt

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES esp_lcd_touch_gt911 bsp_i2c)
```

- 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.

peripheral > bsp_display > M CMakeLists.txt

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES esp_lcd_touch_gt911 bsp_i2c)
```

- 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.)

peripheral > bsp_display > CMakeLists.txt

```
FILE(GLOB_RECURSE component_sources "*.c")

idf_component_register(SRCS ${component_sources}
                        INCLUDE_DIRS "include"
                        REQUIRES esp_lcd_touch_gt911 bsp_i2c)
```

- 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)

main.c | bsp_display.c | CMakeLists.txt | bsp_display.h | bsp_i2c.c | bsp_i2c.h | idf_component.yml

peripheral > bsp_display > include > bsp_display.h > ...

```
#ifndef _BSP_DISPLAY_H_
#define _BSP_DISPLAY_H_
/*----------------------------Header file declaration----------------------------*/
#include "esp_log.h"
#include "esp_err.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_lcd_touch_gt911.h"
#include "bsp_i2c.h"
/*----------------------------Header file declaration end----------------------------*/
```

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.

bsp_display.c | CMakeLists.txt ...\bsp_display | bsp_display.h | bsp_i2c.c | CMakeLists.txt ...\bsp_i2c | bsp_i2c.h | idf_co

peripheral > bsp_i2c > include > bsp_i2c.h > ...

```
#ifndef _BSP_I2C_H_
#define _BSP_I2C_H_

/*----------------------------Header file declaration----------------------------*/
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <rom/ets_sys.h>
#include "esp_timer.h"
#include "driver/i2c_master.h"
#include "esp_log.h"
#include "esp_err.h"
/*----------------------------Header file declaration end----------------------------*/

/*----------------------------Variable declaration----------------------------*/
#define I2C_TAG "I2C"
#define I2C_INFO(fmt, ...) ESP_LOGI(I2C_TAG, fmt, ##__VA_ARGS__)
#define I2C_DEBUG(fmt, ...) ESP_LOGD(I2C_TAG, fmt, ##__VA_ARGS__)
#define I2C_ERROR(fmt, ...) ESP_LOGE(I2C_TAG, fmt, ##__VA_ARGS__)

#define I2C_MASTER_PORT 0
#define I2C_GPIO_SCL 46
#define I2C_GPIO_SDA 45
```

- 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.

peripheral > bsp_i2c > CMakeLists.txt

```
FILE(GLOB_RECURSE component_sources "*.c")

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

- Here, in the "bsp_i2c.h" file, we have utilized "driver/i2c_master.h" and "esp_timer.h".

bsp_display.c | CMakeLists.txt ...\bsp_display | bsp_display.h | bsp_i2c.c | CMakeLists.txt ...\bsp_i2c | bsp_i2c.h | idf_

peripheral > bsp_i2c > include > bsp_i2c.h > ...

```
#ifndef _BSP_I2C_H_
#define _BSP_I2C_H_

/*----------------------------------------Header file declaration----------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <rom/ets_sys.h>
#include "esp_timer.h"
#include "driver/i2c_master.h"
#include "esp_log.h"
#include "esp_err.h"
/*--------------------------------------Header file declaration end--------------------------------------*/
```

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.

```
EXPLORER
LESSON04
  > .devcontainer
  > .vscode
  > build
  v main
      CMakeLists.txt
      idf_component.yml
      main.c
  > managed_components
  v peripheral
    v bsp_display
      v include
          bsp_display.h
        bsp_display.c
        CMakeLists.txt
    v bsp_i2c
      v include
          bsp_i2c.h
        bsp_i2c.c
        CMakeLists.txt
  .clangd
  CMakeLists.txt
```

main.c | bsp_display.c | CMakeLists.txt ...\bsp_display | bsp_display.h | bsp_i2c.c

main > main.c > touch_task(void *)

```
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "bsp_i2c.h"
#include "bsp_display.h"

#define TAG "TOUCH_APP"

TaskHandle_t touch_task_handle = NULL;

void touch_task(void *param)
{
    while (1) {
        if (touch_read() == ESP_OK) {
            uint16_t x, y;
            bool pressed;
            get_coor(&x, &y, &pressed);

            if (pressed) {
                ESP_LOGI(TAG, "Touch at X=%d, Y=%d", x, y);
            }
        }
        vTaskDelay(pdMS_TO_TICKS(50));
    }
```

- 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 > main.c > touch_task(void *)

```c
#include "esp_log.h"           // ESP-IDF logging functions
#include "freertos/FreeRTOS.h" // FreeRTOS base header
#include "freertos/task.h"     // FreeRTOS task APIs
#include "bsp_i2c.h"           // Custom I2C BSP driver
#include "bsp_display.h"       // Custom display/touch BSP driver

#define TAG "TOUCH_APP"        // Logging tag for this application

TaskHandle_t touch_task_handle = NULL; // Handle for the touch reading task

// Task function: continuously reads touch data and logs coordinates
void touch_task(void *param)
{
    while (1) {
        if (touch_read() == ESP_OK) {   // Read touch panel
            uint16_t x, y;
            bool pressed;
            get_coor(&x, &y, &pressed); // Get current touch coordinates and state

            if (pressed) {
                ESP_LOGI(TAG, "Touch at X=%d, Y=%d", x, y); // Log touch coordinates
            }
        }
        vTaskDelay(pdMS_TO_TICKS(50)); // Delay 50ms between reads
    }
}
```

- 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 > main.c > touch_task(void *)

```c
#include "esp_log.h"           // ESP-IDF logging functions
#include "freertos/FreeRTOS.h" // FreeRTOS base header
#include "freertos/task.h"     // FreeRTOS task APIs
#include "bsp_i2c.h"           // Custom I2C BSP driver
#include "bsp_display.h"       // Custom display/touch BSP driver

#define TAG "TOUCH_APP"        // Logging tag for this application

TaskHandle_t touch_task_handle = NULL; // Handle for the touch reading task

// Task function: continuously reads touch data and logs coordinates
void touch_task(void *param)
```

- Infinite loop, reading touchscreen data every 50ms.
- touch_read(): Read GT911 touchscreen data and update internal coordinates.
- get_coor(&x, &y, &pressed): Obtain the current touch coordinates and pressing status.
- If a touch is detected (pressed = true), print the touch coordinates.
- vTaskDelay(pdMS_TO_TICKS(50)): Put the task to sleep for 50ms to avoid frequent polling occupying CPU.

main > main.c > touch_task(void *)

```c
#include "esp_log.h"           // ESP-IDF logging functions
#include "freertos/FreeRTOS.h" // FreeRTOS base header
#include "freertos/task.h"     // FreeRTOS task APIs
#include "bsp_i2c.h"           // Custom I2C BSP driver
#include "bsp_display.h"       // Custom display/touch BSP driver

#define TAG "TOUCH_APP"        // Logging tag for this application

TaskHandle_t touch_task_handle = NULL; // Handle for the touch reading task

// Task function: continuously reads touch data and logs coordinates
void touch_task(void *param)
{
    while (1) {
        if (touch_read() == ESP_OK) {   // Read touch panel
            uint16_t x, y;
            bool pressed;
            get_coor(&x, &y, &pressed); // Get current touch coordinates and state

            if (pressed) {
                ESP_LOGI(TAG, "Touch at X=%d, Y=%d", x, y); // Log touch coordinates
            }
        }
        vTaskDelay(pdMS_TO_TICKS(50)); // Delay 50ms between reads
    }
}
```

- Then comes the main function app_main.
- It first prints the information about the program startup.

```c
// Main application entry point
void app_main(void)
{
    ESP_LOGI(TAG, "Starting touch application"); // Log app start

```

- Call the initialization code in "bsp_i2c.c" to initialize the I2C bus, which is used for communication with the touch screen chip.

```c
// Main application entry point
void app_main(void)
{
    ESP_LOGI(TAG, "Starting touch application"); // Log app start

    // Initialize I2C bus
    if (i2c_init() != ESP_OK) {
        ESP_LOGE(TAG, "I2C initialization failed"); // Log error if I2C init fails
        return;
    }

```

- 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.

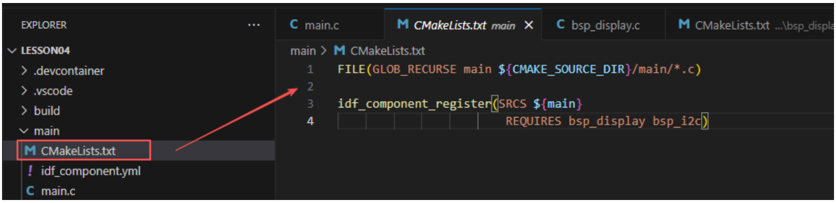
```
main > C main.c > app_main(void)
void touch_task(void *param)
    while (1) {
        if (touch_read() == ESP_OK) {   // Read touch panel
            if (pressed) {
                ESP_LOGI(TAG, "Touch at X=%d, Y=%d", x, y); // Log touch coordinates
            }
        }
        vTaskDelay(pdMS_TO_TICKS(50)); // Delay 50ms between reads
    }
}

// Main application entry point
void app_main(void)
{
    ESP_LOGI(TAG, "Starting touch application"); // Log app start

    // Initialize I2C bus
    if (i2c_init() != ESP_OK) {
        ESP_LOGE(TAG, "I2C initialization failed"); // Log error if I2C init fails
        return;
    }

    // Initialize the touchscreen
    if (touch_init() != ESP_OK) {
        ESP_LOGE(TAG, "Touch initialization failed"); // Log error if touch init fails
        return;
    }
```

- The following code is also familiar to you. You have encountered it in previous courses. The function of this line of code is to create and start a task named "touch_task" in FreeRTOS, allowing it to periodically read touch screen data in an independent thread. At the same time, through the "touch_task_handle" handle, this task can be managed later.

```
// Main application entry point
void app_main(void)
{
    ESP_LOGI(TAG, "Starting touch application"); // Log app start

    // Initialize I2C bus
    if (i2c_init() != ESP_OK) {
        ESP_LOGE(TAG, "I2C initialization failed"); // Log error if I2C init fails
        return;
    }

    // Initialize the touchscreen
    if (touch_init() != ESP_OK) {
        ESP_LOGE(TAG, "Touch initialization failed"); // Log error if touch init fails
        return;
    }

    // Create a FreeRTOS task for reading touch data
    xTaskCreate(touch_task, "touch_task", 4096, NULL, 5, &touch_task_handle);

    ESP_LOGI(TAG, "Touch application started successfully"); // Log successful start
}
```

- 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" and the custom component "bsp_i2c";
- This way, during the build process, ESP-IDF knows to build "bsp_display" and "bsp_i2c" 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.

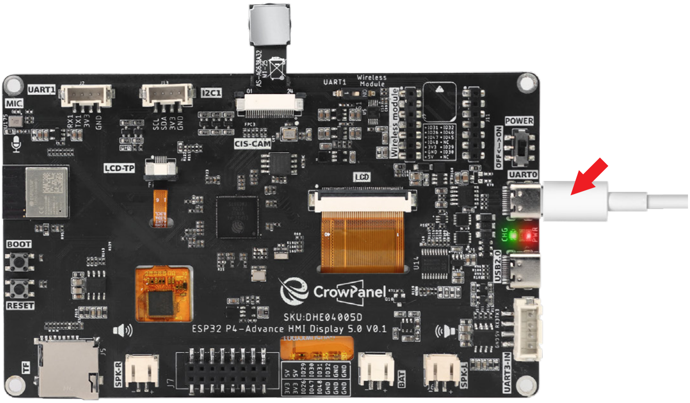
Complete Code

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

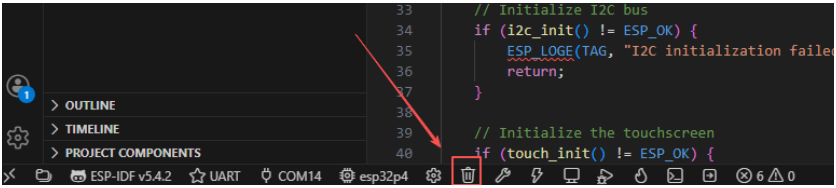
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson05-Touchscreen

Programming Steps

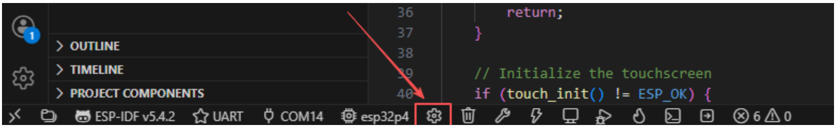
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable.



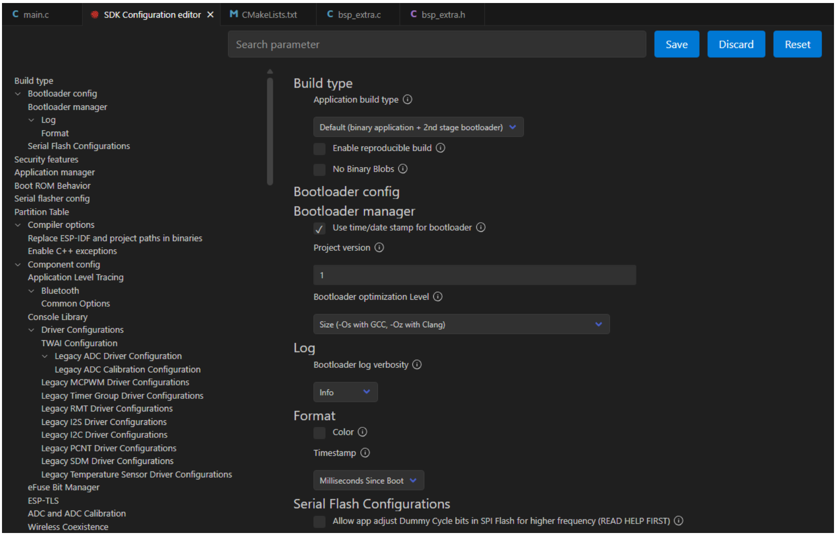
- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



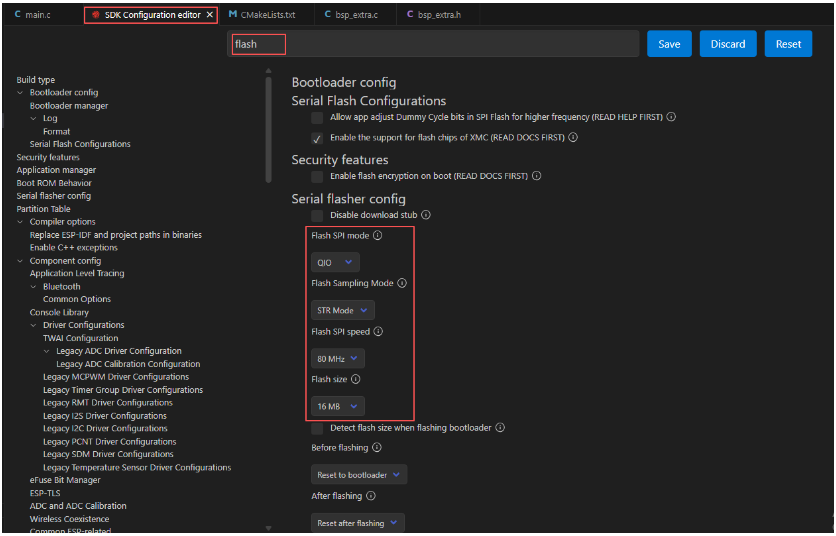
- Then, following the steps in the first section, select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



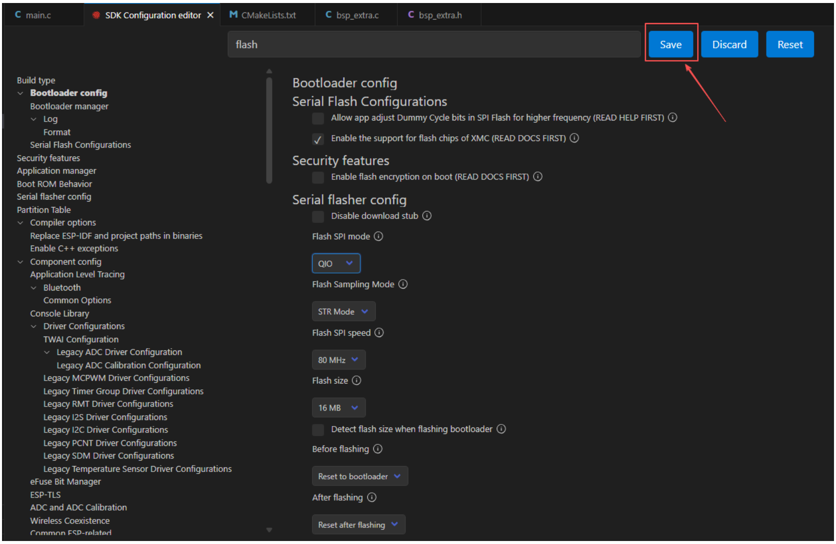
- Wait for a moment for the loading process to complete, and then you can proceed with the related SDK configuration.



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

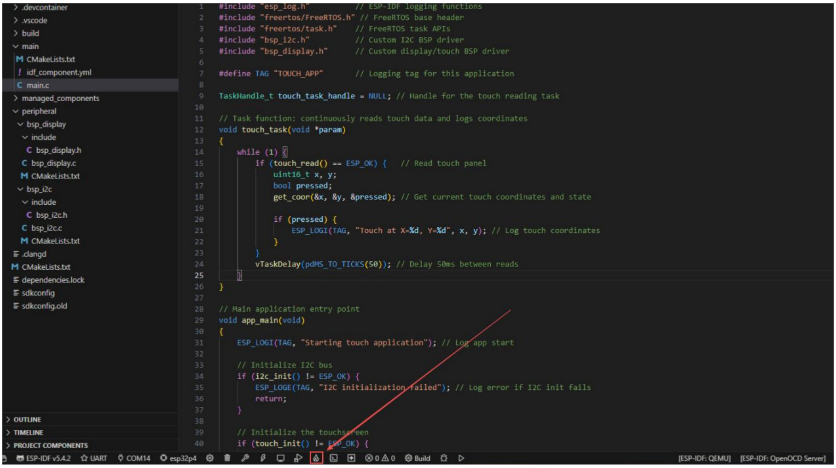


- After the configuration is completed, be sure to save your settings.

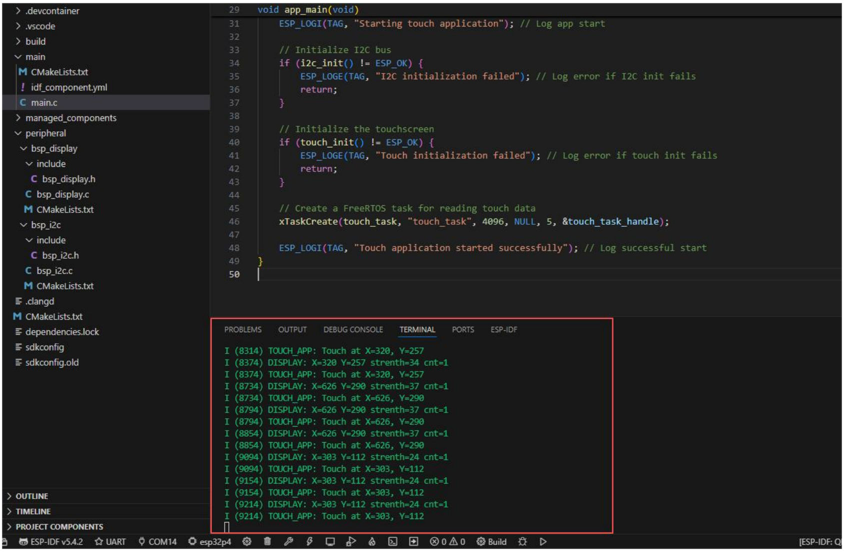


- Then we will compile and burn the code (as detailed in the first class).

- 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 Adcance-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.

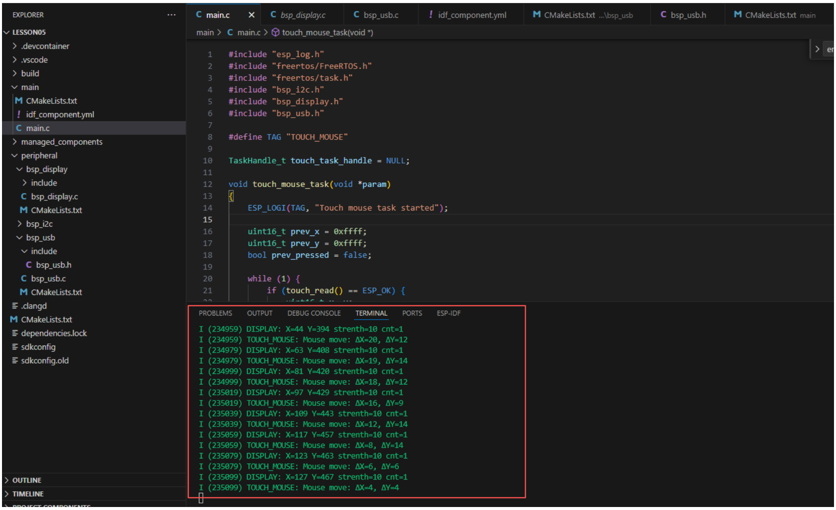
In this class, we will add a new component, bsp_usb, on top of these two components. This will enable us to use the USB2.0 interface on our Advance-P4 to act as a mouse. When you slide on the screen of the Advance-P4, you will be able to see that the mouse on your computer also moves accordingly.

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.



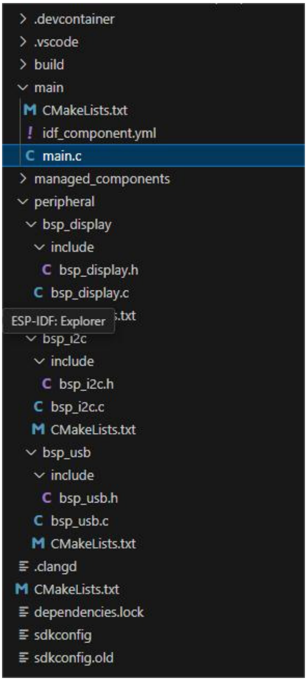
Key Explanations

- 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-5inch-ESP32-P4-HMI-AI-Display-800x480-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.

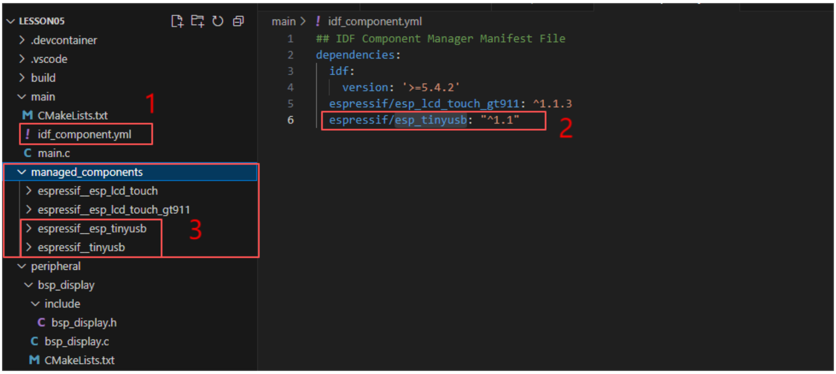
```c
#include <stdint.h>          // Standard integer types
#include "esp_err.h"          // ESP-IDF error handling definitions
#include "esp_log.h"          // ESP-IDF logging functions
#include "freertos/FreeRTOS.h" // FreeRTOS base header
#include "freertos/task.h"    // FreeRTOS task management APIs

#include "tinyusb.h"          // TinyUSB core library
#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)

```c
#include <stdint.h>          // Standard integer types
#include "esp_err.h"          // ESP-IDF error handling definitions
#include "esp_log.h"          // ESP-IDF logging functions
#include "freertos/FreeRTOS.h" // FreeRTOS base header
#include "freertos/task.h"    // FreeRTOS task management APIs

#include "tinyusb.h"          // TinyUSB core library
#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.

peripheral > bsp_usb > include > C bsp_usb.h > HID_ITF_PROTOCOL_MOUSE

```c

#include "tinyusb.h"          // TinyUSB core library
#include "class/hid/hid_device.h" // TinyUSB HID device class definitions

#define USB_TAG "USB"          // Logging tag for USB module
#define USB_INFO(fmt, ...) ESP_LOGI(USB_TAG, fmt, ##__VA_ARGS__)  // Info log macro
#define USB_DEBUG(fmt, ...) ESP_LOGD(USB_TAG, fmt, ##__VA_ARGS__) // Debug log macro
#define USB_ERROR(fmt, ...) ESP_LOGE(USB_TAG, fmt, ##__VA_ARGS__) // Error log macro

// HID protocol definition
#define HID_ITF_PROTOCOL_MOUSE    1  // HID interface protocol: Mouse

// Function to send mouse movement deltas over USB HID
void send_hid_mouse_delta(int8_t delta_x, int8_t delta_y);

// Function to check whether USB is initialized and ready
bool is_usb_ready(void);

// Function to initialize USB subsystem
esp_err_t usb_init(void);

#endif // _BSP_USB_H_

```

- 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:

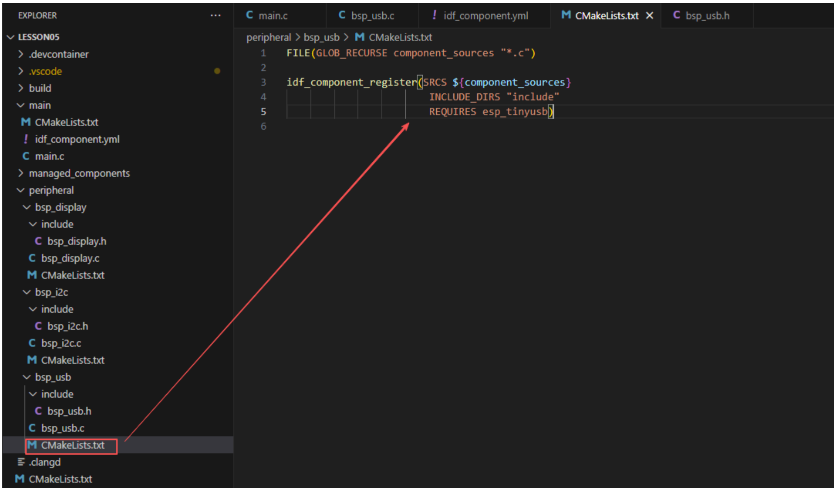
This function constructs tinyusb_config_t (containing string descriptors, configuration descriptors, etc.) and calls tinyusb_driver_install(&tusb_cfg) to install the TinyUSB driver; it is responsible for starting the USB subsystem and exposing the HID device to the operating system (the host).

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.

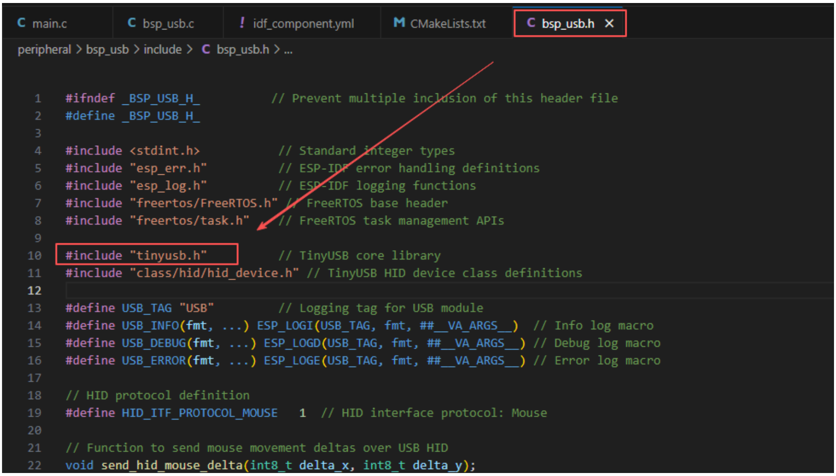
That's all about the bsp_usb component. Just know how to call these interfaces and you're good to go.

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.



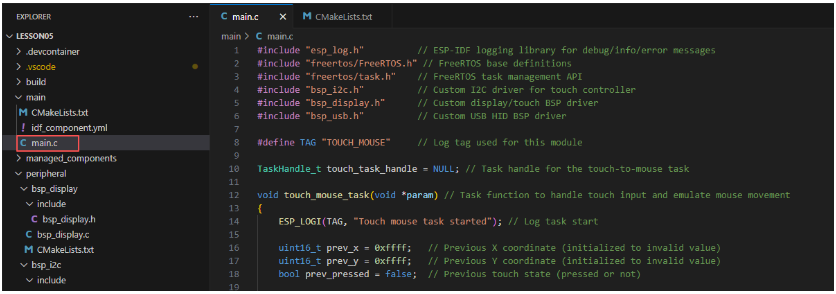
- The reason why it is called esp_tinyusb here is that we called it in the "bsp_usb.h" file (for other libraries that are system libraries, there is no need to add anything).



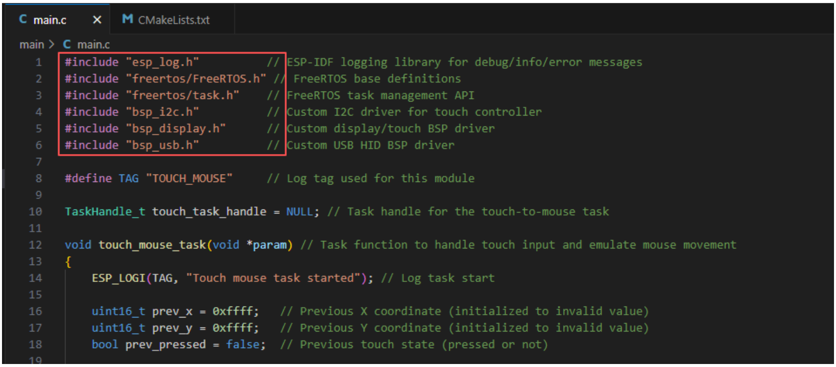
Main function

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.

main > C main.c

```c
#include "esp_log.h"            // ESP-IDF logging library for debug/info/error messages
#include "freertos/FreeRTOS.h" // FreeRTOS base definitions
#include "freertos/task.h"      // FreeRTOS task management API
#include "bsp_i2c.h"            // Custom I2C driver for touch controller
#include "bsp_display.h"        // Custom display/touch BSP driver
#include "bsp_usb.h"            // Custom USB HID BSP driver

#define TAG "TOUCH_MOUSE"       // Log tag used for this module
TaskHandle_t touch_task_handle = NULL; // Task handle for the touch-to-mouse task
```

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.

```c
void touch_mouse_task(void *param) // Task function to handle touch input and emulate mouse movement
{
    ESP_LOGI(TAG, "Touch mouse task started"); // Log task start

    uint16_t prev_x = 0xffff;   // Previous X coordinate (initialized to invalid value)
    uint16_t prev_y = 0xffff;   // Previous Y coordinate (initialized to invalid value)
    bool prev_pressed = false;  // Previous touch state (pressed or not)

    while (1) { // Infinite loop for continuous task execution
        if (touch_read() == ESP_OK) { // Read touch input
            uint16_t x, y;         // Current X, Y coordinates
            bool pressed;          // Current touch state
            get_coor(&x, &y, &pressed); // Retrieve touch coordinates and state

            // Send mouse movement only when USB is ready and screen is being touched
            if (pressed && is_usb_ready()) {
                if (prev_pressed && prev_x != 0xffff && prev_y != 0xffff) {
                    // Calculate movement delta
                    int16_t delta_x = (int16_t)x - (int16_t)prev_x;
                    int16_t delta_y = (int16_t)y - (int16_t)prev_y;

                    // Send mouse movement HID report
                    send_hid_mouse_delta(delta_x, delta_y);
                    ESP_LOGI(TAG, "Mouse move: ΔX=%d, ΔY=%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 press 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 delta_x = x - prev_x, delta_y = y - 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.

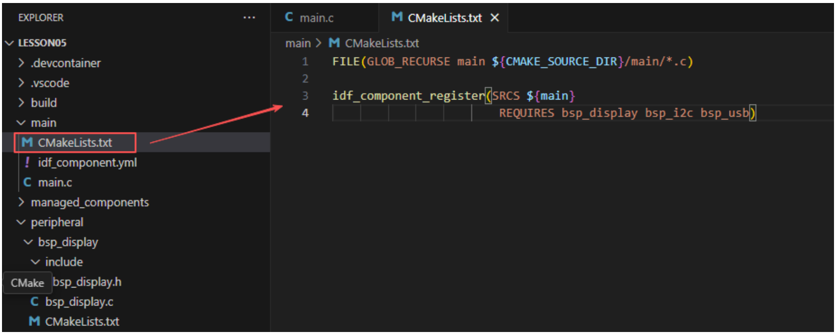
Then comes the main function **app_main**.

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.

```c
void app_main(void) // Main application entry point
{
    ESP_LOGI(TAG, "Starting Touch Mouse application"); // Log application start

    // Initialize I2C bus
    if (i2c_init() != ESP_OK) {
        ESP_LOGE(TAG, "I2C initialization failed"); // Log error if I2C init fails
        return;
    }

    // Initialize touchscreen
    if (touch_init() != ESP_OK) {
        ESP_LOGE(TAG, "Touch initialization failed"); // Log error if touch init fails
        return;
    }

    // Initialize USB HID subsystem
    if (usb_init() != ESP_OK) {
        ESP_LOGE(TAG, "USB initialization failed"); // Log error if USB init fails
        return;
    }

    // Create FreeRTOS task for touch-to-mouse handling
    xTaskCreate(touch_mouse_task, "touch_mouse_task", 4096, NULL, 5, &touch_task_handle);
    if (touch_task_handle == NULL) {
        ESP_LOGE(TAG, "Failed to create touch mouse task"); // Log error if task creation fails
        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.

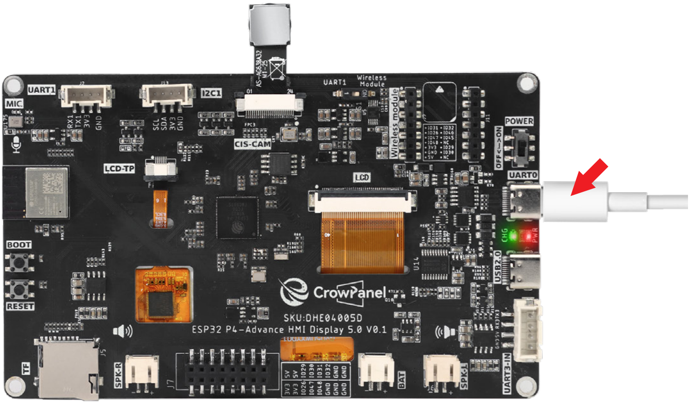
Complete Code

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

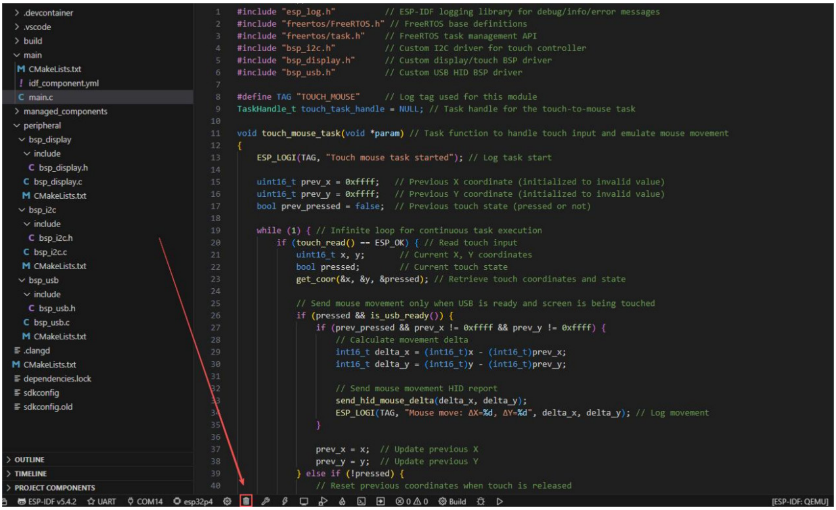
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson06-USB2.0

Programming Steps

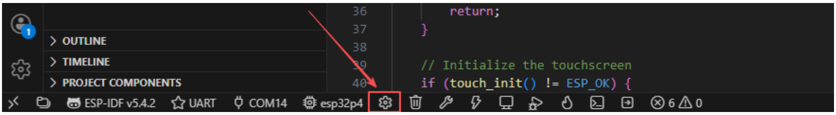
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable.



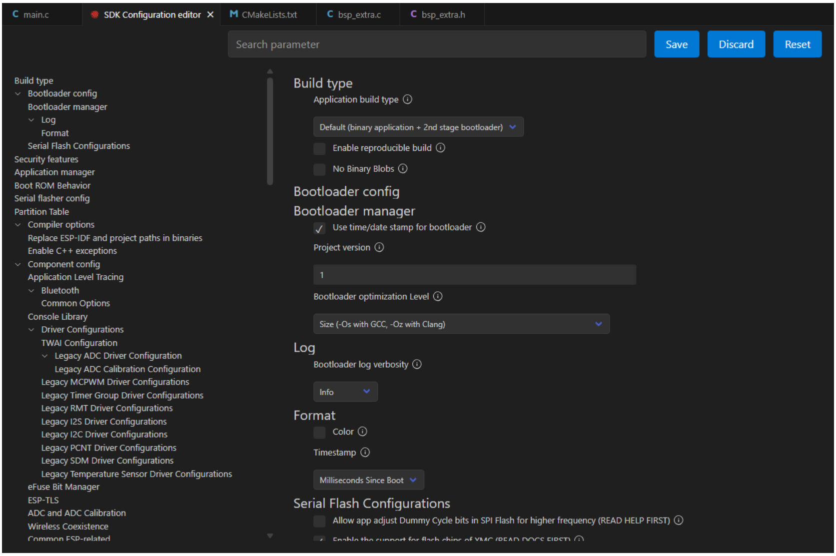
- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



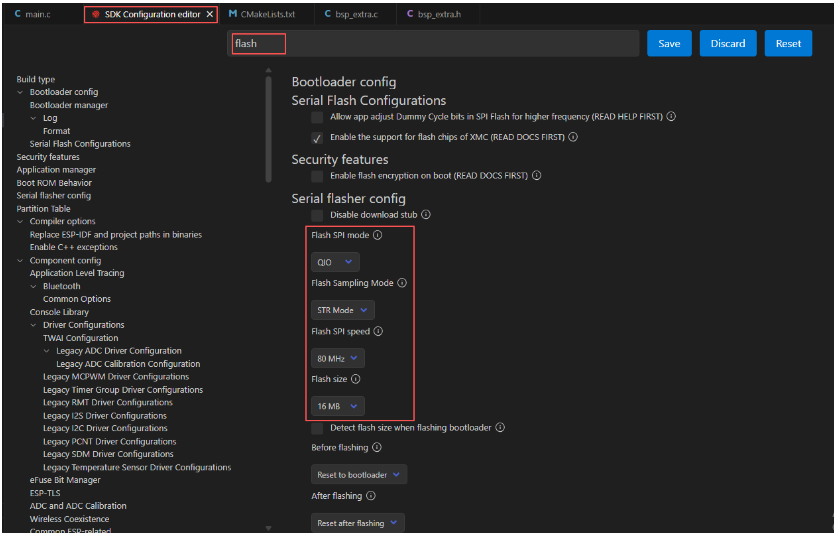
- Here, following the steps in the first section, first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



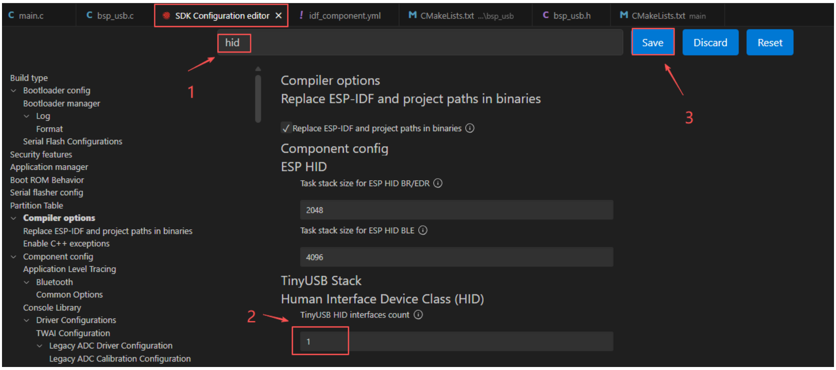
- Wait for a moment for the loading process to complete, and then you can proceed with the related SDK configuration.



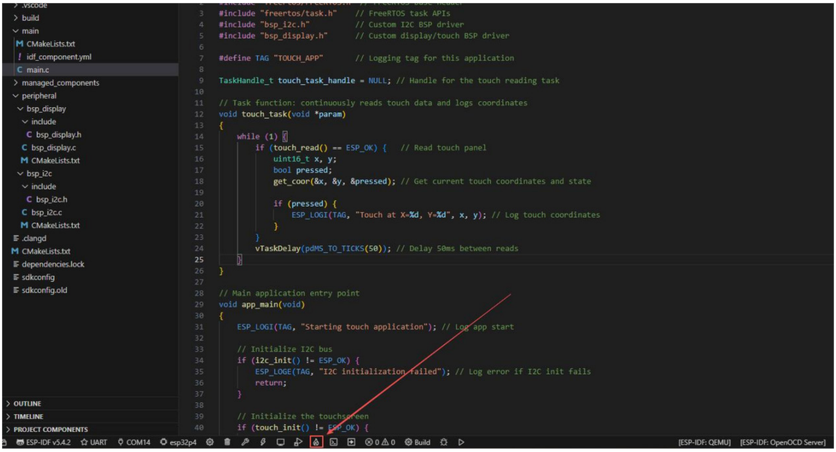
- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



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



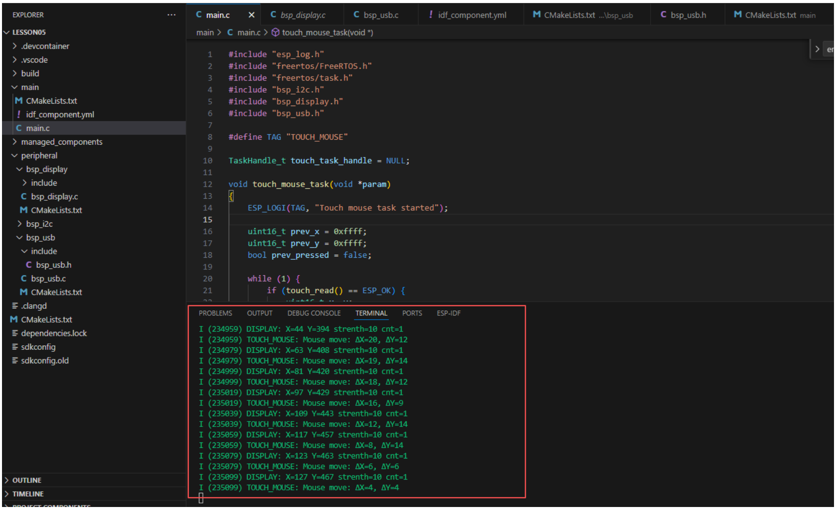
- After the configuration is completed, be sure to save your settings.
- Then we will compile and burn the code (as detailed in the first class).
- Here, we would like to introduce to you another very convenient feature. With just one button press, you can perform the compilation, upload, and open the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


EXPLORER
LESSON05
.devcontainer
.vscode
build
main
CMakeLists.txt
idf_component.yml
main.c
managed_components
peripheral
bsp_display
include
bsp_display.c
CMakeLists.txt
bsp_i2c
bsp_usb
include
bsp_usb.h
bsp_usb.c
CMakeLists.txt
.clangd
CMakeLists.txt
dependencies.lock
sdkconfig
sdkconfig.old
OUTLINE
TIMELINE
main.c
bsp_display.c
bsp_usb.c
idf_component.yml
CMakeLists.txt ...\bsp_usb
bsp_usb.h
CMakeLists.txt main
main > main.c > touch_mouse_task(void *)
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "bsp_i2c.h"
#include "bsp_display.h"
#include "bsp_usb.h"
#define TAG "TOUCH_MOUSE"
TaskHandle_t touch_task_handle = NULL;
void touch_mouse_task(void *param)
{
ESP_LOGI(TAG, "Touch mouse task started");
uint16_t prev_x = 0xffff;
uint16_t prev_y = 0xffff;
bool prev_pressed = false;
while (1) {
if (touch_read() == ESP_OK) {
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
ESP-IDF
I (234959) DISPLAY: X=44 Y=394 strenth=10 cnt=1
I (234959) TOUCH_MOUSE: Mouse move: ΔX=20, ΔY=12
I (234979) DISPLAY: X=63 Y=408 strenth=10 cnt=1
I (234979) TOUCH_MOUSE: Mouse move: ΔX=19, ΔY=14
I (234999) DISPLAY: X=81 Y=420 strenth=10 cnt=1
I (234999) TOUCH_MOUSE: Mouse move: ΔX=18, ΔY=12
I (235019) DISPLAY: X=97 Y=429 strenth=10 cnt=1
I (235019) TOUCH_MOUSE: Mouse move: ΔX=16, ΔY=9
I (235039) DISPLAY: X=109 Y=443 strenth=10 cnt=1
I (235039) TOUCH_MOUSE: Mouse move: ΔX=12, ΔY=14
I (235059) DISPLAY: X=117 Y=457 strenth=10 cnt=1
I (235059) TOUCH_MOUSE: Mouse move: ΔX=8, ΔY=14
I (235079) DISPLAY: X=123 Y=463 strenth=10 cnt=1
I (235079) TOUCH_MOUSE: Mouse move: ΔX=6, ΔY=6
I (235099) DISPLAY: X=127 Y=467 strenth=10 cnt=1
I (235099) TOUCH_MOUSE: Mouse move: ΔX=4, ΔY=4

Lesson 07
Turn on the screen

Introduction

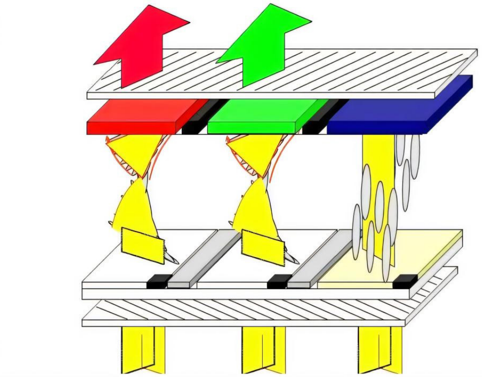
In this class, we will start by teaching you how to turn on the screen. Then, while turning on the screen backlight, we will display "Hellow Elecrow" on the screen. Of course, you can replace it with whatever you want.

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

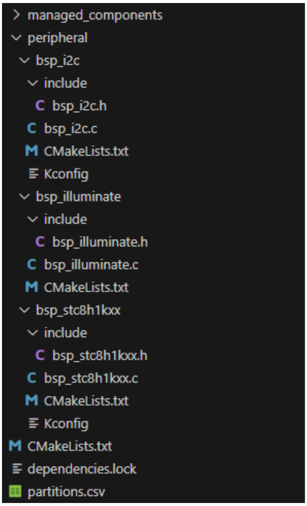
After running the code, you will be able to visually see that "Hello Elecrow" is displayed on the screen of the Advance-P4.

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-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-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.



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

The "bsp_illuminate" folder contains the "bsp_illuminate.c" driver file, and the "include" folder contains the "bsp_illuminate.h" header file.

The "CMakeLists.txt" file will integrate the driver into the build system, enabling the project to utilize the screen display functionality described in "bsp_illuminate.c".

- There are also two other components, bsp_i2c and bsp_stc8h1kxx.
- bsp_i2c is the I2C component. Here, an I2C control expansion chip STC8H1K17 is needed, and then STC8H1K17 will control the backlight of the screen.
- bsp_stc8h1kxx is the control interface required by the expansion chip. This expansion chip can control the backlight of the screen, the power amplifier enable, etc. We will remind you again when we use it later.
- You can see that there is another component called "components" above. This one is a component we downloaded from the internet. After downloading, we modified its code based on it to adapt to the display requirements of our RGB screen. (You can use it directly without any modifications.)

Screen display code

- The driver code displayed on the screen consists of two files: "bsp_illuminate.c" and "bsp_illuminate.h".
- Next, we will first analyze the "bsp_illuminate.h" program.
- "bsp_illuminate.h" is a header file for the screen display module, mainly used for:
- Making the functions, macros, and variable declarations implemented in "bsp_illuminate.c" available for use by external programs.
- Allowing other .c files to simply include "bsp_illuminate.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_illuminate.h" file for unified management.

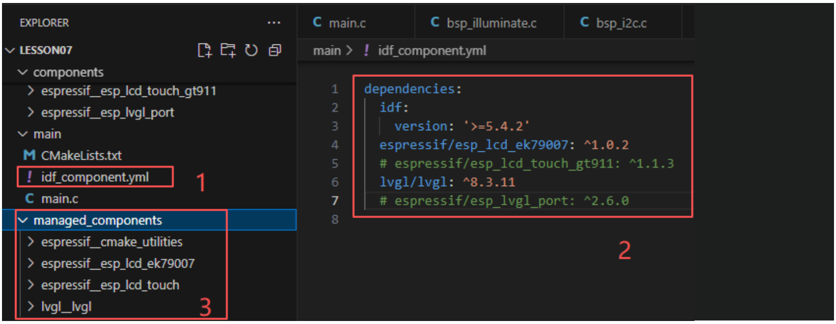
```c
/*——————————————————————————————Header file declaration——————————————————————————————*/
#include "esp_log.h"          //References for LOG Printing Function-related API Functions
#include "esp_err.h"          //References for Error Type Function-related API Functions
#include "esp_ldo_regulator.h" //References for LDO Function-related API Functions
#include "esp_lcd_ek79007.h"   //References for lcd ek79007 Function-related API Functions
#include "esp_lcd_mipi_dsi.h"  //References for lcd mipi dsi Function-related API Functions
#include "esp_lcd_panel_ops.h" //References for lcd panel ops Function-related API Functions
#include "esp_lcd_panel_io.h"  //References for lcd panel io Function-related API Functions
#include "esp_lcd_panel_rgb.h"
#include "esp_lvgl_port.h"     //References for LVGL port Function-related API Functions
#include "driver/gpio.h"       //References for GPIO Function-related API Functions
#include "driver/ledc.h"       //References for LEDC PWM Function-related API Functions
#include "lvgl.h"              //References for LVGL Function-related API Functions
#include "bsp_i2c.h"
```

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

```c
/*——————————————————————————————Header file declaration——————————————————————————————*/
#include "esp_log.h"          //References for LOG Printing Function-related API Functions
#include "esp_err.h"          //References for Error Type Function-related API Functions
#include "esp_ldo_regulator.h" //References for LDO Function-related API Functions
#include "esp_lcd_ek79007.h"   //References for lcd ek79007 Function-related API Functions
#include "esp_lcd_mipi_dsi.h"  //References for lcd mipi dsi Function-related API Functions
#include "esp_lcd_panel_ops.h" //References for lcd panel ops Function-related API Functions
#include "esp_lcd_panel_io.h"  //References for lcd panel io Function-related API Functions
#include "esp_lvgl_port.h"     //References for LVGL port Function-related API Functions
#include "driver/gpio.h"       //References for GPIO Function-related API Functions
#include "driver/ledc.h"       //References for LEDC PWM Function-related API Functions
#include "lvgl.h"              //References for LVGL Function-related API Functions
/*——————————————————————————————Header file declaration end——————————————————————————————*/
```

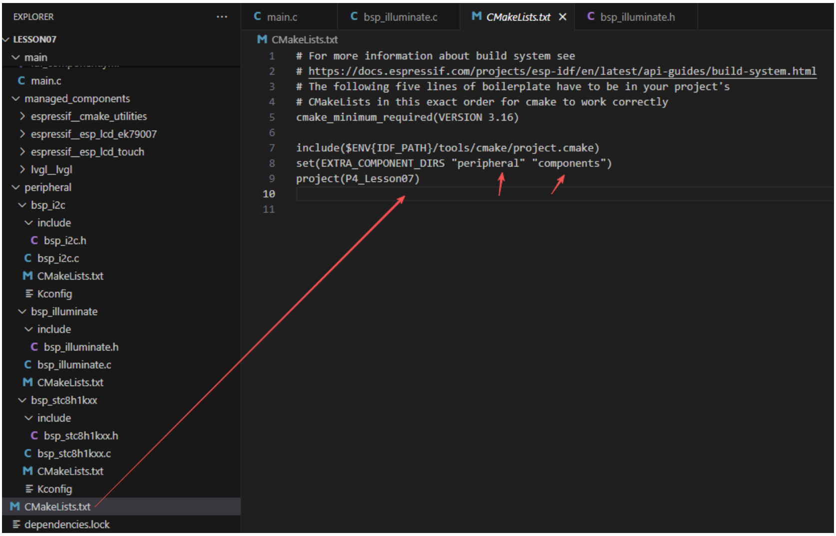
- In this case, we need to fill in the versions of esp_lcd_ek79007, esp_lvgl_port and lvgl in the idf_component.yml file located in the main folder.

- 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 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.)
- Here we have commented out esp_lcd_touch_gt911 and esp_lvgl_port because we have already prepared them in the components folder and made the necessary modifications in their code. Everyone can just use it directly.

- Then, include the components in the outer CMakeLists.txt file.
- (These are the custom components we use)
- Peripheral: Components such as bsp_illuminate that we wrote ourselves
- Components: The network components will be modified after download and adapted to our 5-inch screen.



- 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.)
- It includes:
- LCD parameter definitions: including screen resolution (800×480), color bit depth (16-bit), pixel clock (18 MHz), and synchronization signal timing (HSYNC, VSYNC, etc.).
- Pin definitions: Lists the GPIO pin numbers used for the LCD data lines (D0–D15), clock, and synchronization signals.
- Color constants: Define common colors (red, green, blue, white, black, gray).
- Function declarations: display_init() is used to initialize the LCD display, and set_lcd_blight() is used to set the backlight brightness of the screen.

```
/*----------------------------------------Variable declaration----------------------------------------*/
#define ILLUMINATE_TAG "ILLUMINATE"
#define ILLUMINATE_INFO(fmt, ...) ESP_LOGI(ILLUMINATE_TAG, fmt, ##__VA_ARGS__)
#define ILLUMINATE_DEBUG(fmt, ...) ESP_LOGD(ILLUMINATE_TAG, fmt, ##__VA_ARGS__)
#define ILLUMINATE_ERROR(fmt, ...) ESP_LOGE(ILLUMINATE_TAG, fmt, ##__VA_ARGS__)

#define H_size 800              // Horizontal resolution
#define V_size 480               // Vertical resolution
#define BITS_PER_PIXEL 16        // Number of image display bits of the display screen

#define LCD_GPIO_BLIGHT -1       // LCD Blight GPIO
#define BLIGHT_PWM_Hz 30000      // LCD Blight PWM GPIO

#define LV_COLOR_RED lv_color_make(0xFF, 0x00, 0x00)    // RED
#define LV_COLOR_GREEN lv_color_make(0x00, 0xFF, 0x00) // GREEN
#define LV_COLOR_BLUE lv_color_make(0x00, 0x00, 0xFF)   // BLUE
#define LV_COLOR_WHITE lv_color_make(0xFF, 0xFF, 0xFF)  // WHITE
#define LV_COLOR_BLACK lv_color_make(0x00, 0x00, 0x00)  // BLACK
#define LV_COLOR_GRAY lv_color_make(0x80, 0x80, 0x80)   // GRAY

//////////////////////////////////////////////////////////////////////////////////////////
/////////////////// Please update the following configuration according to your LCD spec /////////////////
//////////////////////////////////////////////////////////////////////////////////////////
// Refresh Rate = 18000000/(1+40+20+800)/(1+10+5+480) = 42Hz
#define RGB_LCD_PIXEL_CLOCK_HZ      (18 * 1000 * 1000)
#define RGB_LCD_H_RES               H_size
#define RGB_LCD_V_RES               V_size
#define RGB_LCD_HSYNC               4
#define RGB_LCD_HBP                 8
#define RGB_LCD_HFP                 8
#define RGB_LCD_VSYNC               4
#define RGB_LCD_VBP                 16
#define RGB_LCD_VFP                 16

#define RGB_PIN_NUM_DISP_EN         -1
#define RGB_PIN_NUM_HSYNC           40
#define RGB_PIN_NUM_VSYNC           41
#define RGB_PIN_NUM_DE              2
#define RGB_PIN_NUM_PCLK            3
```

```c
#define RGB_PIN_NUM_DATA2          6
#define RGB_PIN_NUM_DATA3          5
#define RGB_PIN_NUM_DATA4          4
#define RGB_PIN_NUM_DATA5          14
#define RGB_PIN_NUM_DATA6          13
#define RGB_PIN_NUM_DATA7          12
#define RGB_PIN_NUM_DATA8          11
#define RGB_PIN_NUM_DATA9          10
#define RGB_PIN_NUM_DATA10         9
#define RGB_PIN_NUM_DATA11         19
#define RGB_PIN_NUM_DATA12         18
#define RGB_PIN_NUM_DATA13         17
#define RGB_PIN_NUM_DATA14         16
#define RGB_PIN_NUM_DATA15         15

#if BITS_PER_PIXEL > 16
#define RGB_PIN_NUM_DATA16         CONFIG_RGB_LCD_DATA16_GPIO
#define RGB_PIN_NUM_DATA17         CONFIG_RGB_LCD_DATA17_GPIO
#define RGB_PIN_NUM_DATA18         CONFIG_RGB_LCD_DATA18_GPIO
#define RGB_PIN_NUM_DATA19         CONFIG_RGB_LCD_DATA19_GPIO
#define RGB_PIN_NUM_DATA20         CONFIG_RGB_LCD_DATA20_GPIO
#define RGB_PIN_NUM_DATA21         CONFIG_RGB_LCD_DATA21_GPIO
#define RGB_PIN_NUM_DATA22         CONFIG_RGB_LCD_DATA22_GPIO
#define RGB_PIN_NUM_DATA23         CONFIG_RGB_LCD_DATA23_GPIO
#endif

#if (16 == BITS_PER_PIXEL)
#define RGB_DATA_BUS_WIDTH         16
#define RGB_PIXEL_SIZE             2
#define RGB_LV_COLOR_FORMAT        LV_COLOR_FORMAT_RGB565
#elif (24 == BITS_PER_PIXEL)
#define RGB_DATA_BUS_WIDTH         24
#define RGB_PIXEL_SIZE             3
#define RGB_LV_COLOR_FORMAT        LV_COLOR_FORMAT_RGB888
#endif

esp_err_t display_init();                             // Display Screen Initialization Function
esp_err_t set_lcd_blight(uint32_t brightness); // Set the screen backlight

```

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:

This function is used to initialize the backlight control hardware of the LCD.

At present, no actual PWM configuration has been performed (only a success return is given), but typically here:

Configure the backlight control pin as an output;

Initialize the PWM channel and the timer;

Have the ability to adjust the brightness through duty cycle.

Subsequently, the actual brightness adjustment is achieved through the function set_lcd_blight().

set_lcd_blight(uint32_t brightness) :

This function adjusts the LCD backlight brightness based on the input brightness value (0 - 100):

Call the function stc8_set_pwm_duty() internally to modify the duty cycle of the PWM signal;

The higher the duty cycle, the brighter the backlight will be.

When brightness is 0, the backlight is completely turned off.

This enables us to flexibly adjust the screen brightness within the application.

display_port init(void):

This function completes the hardware initialization of the display interface, and mainly includes the following steps:

Configure RGB interface parameters:

This includes data bit width, DMA alignment, synchronization signal pins (HSYNC, VSYNC, DE), pixel clock, resolution, timing parameters, etc.

Create and register the panel driver:

esp_lcd_new_rgb_panel() creates the LCD panel object, and esp_lcd_panel_init() performs the hardware initialization.

Set up PSRAM frame buffer:

By setting flags.fb_in_psram = true, place the frame buffer in the external PSRAM to avoid insufficient memory.

Summary: This function establishes a communication bridge between the ESP32 and the EK79007 display controller.

lvgl_init() :

This function is the core of the graphical interface system and is responsible for initiating the LVGL graphics library.

The main workflow is as follows:

Initialize LVGL tasks and timers: Create the LVGL running task using lvgl_port_init().

Registering display device: Use disp_cfg to register the previously created LCD panel as an LVGL output device;

Configuration of display parameters: including resolution, color format, refresh mode, etc.

Enable the anti-tearing mechanism to ensure no flickering when the image is refreshed.

Result: LVGL has successfully been bound to the hardware display, and the system can now display the LVGL graphical interface.

display_init() :

This is the main entry function of the entire display system, which is executed in a fixed sequence:

blight_init() → Initializes the backlight control module;

display_port_init() → Initialize the hardware display interface;

lvgl_init() → Initializes the graphics display system.

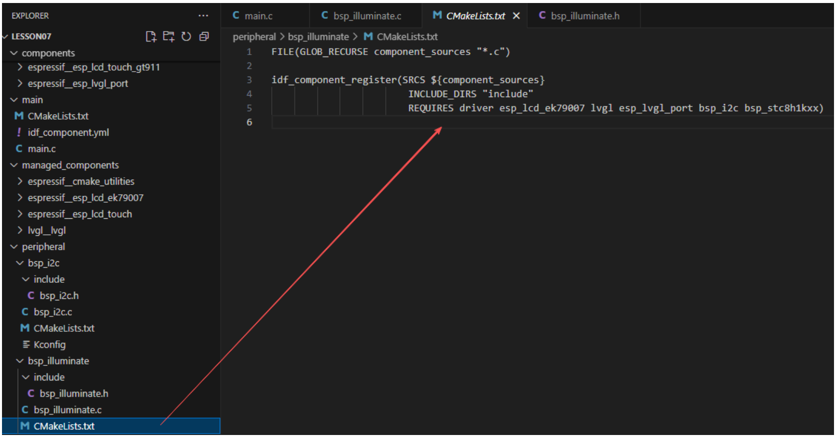
If any step fails, an error message will be immediately returned to ensure the system's security and stability.

Final Result: Once the display_init() function is successfully executed, the entire display system can enter the visual working state, preparing for the display of the LVGL graphical interface.

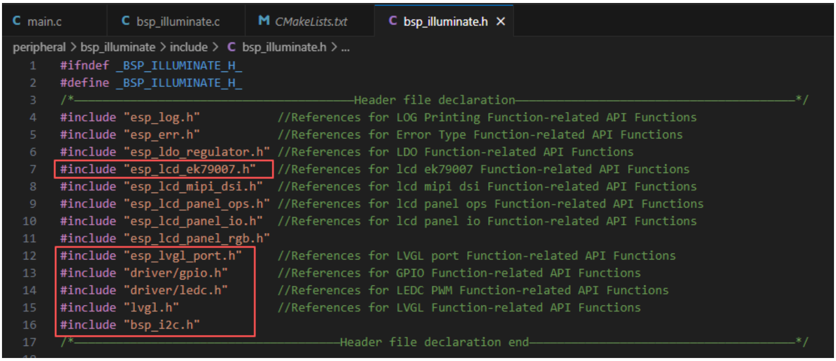
That's all about the components of bsp_illuminate. Just remember how to call these interfaces and you'll be fine.

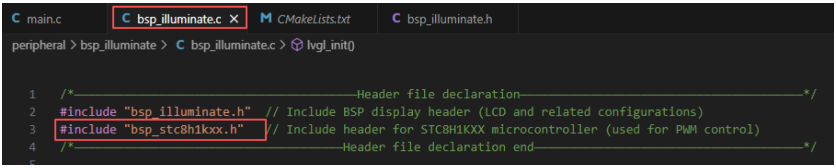
Then, if we need to make a call, we must also configure the "CMakeLists.txt" file located in the "bsp_illuminate" folder.

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, esp_lvgl_port,bsp_i2c and bsp_stc8h1kxx is that we called them in "bsp_illuminate.h" (for other libraries that are system libraries, there is no need to add them)





- After talking about the bsp_stc8h1kxx component for such a long time, now let's take a look at what this component actually achieves.

- The header file bsp_stc8h1kxx.h is mainly used for communication and control with the STC8H1KXX chip via the I2C bus on the ESP32 platform. The file first introduces the necessary system and driver header files, then defines log output macros, basic data type aliases, and a series of register addresses, function enumerations, and structures related to the STC8 chip. It mainly includes three functional modules: battery management (obtaining battery voltage, power, and charging status), GPIO control (reading input pin levels, setting output pin states), and PWM control (adjusting the duty cycle of devices such as LCD backlight).

- The file finally declares the corresponding I2C initialization function and various operation interface functions, enabling upper-layer applications to conveniently implement communication and control of the STC8H1KXX peripheral chip through a unified API.

main.c | bsp_stc8h1kxx.c | bsp_stc8h1kxx.h | bsp_illuminate.c | bsp_illuminate.h

peripheral > bsp_stc8h1kxx > include > bsp_stc8h1kxx.h > ...

```c
#ifndef _BSP_STC8H1KXX_H_           // Prevent recursive inclusion of this header file
#define _BSP_STC8H1KXX_H_

/*———————————————————————————————————Header file declaration———————————————————————————————————*/
#include <string.h>                 // Include standard string library
#include <stdint.h>                 // Include standard integer type definitions
#include "freertos/FreeRTOS.h"      // Include FreeRTOS main header
#include "freertos/task.h"          // Include FreeRTOS task management
#include "esp_log.h"                // Include ESP logging library
#include "esp_err.h"                // Include ESP error handling definitions
#include "driver/uart.h"            // Include UART driver
#include "bsp_i2c.h"                // Include I2C bus driver for STC communication
#include "driver/gpio.h"            // Include GPIO driver
/*———————————————————————————————————Header file declaration end———————————————————————————————————*/

#ifdef __cplusplus
extern "C"
{
#endif

/*———————————————————————————————————Variable declaration———————————————————————————————————*/
#define STC8H1KXX_TAG "STC8H1KXX"                                    // Tag name used for logging
#define STC8H1KXX_INFO(fmt, ...)  ESP_LOGI(STC8H1KXX_TAG, fmt, ##__VA_ARGS__)   // Information log macro
#define STC8H1KXX_DEBUG(fmt, ...) ESP_LOGD(STC8H1KXX_TAG, fmt, ##__VA_ARGS__)   // Debug log macro
#define STC8H1KXX_ERROR(fmt, ...) ESP_LOGE(STC8H1KXX_TAG, fmt, ##__VA_ARGS__)   // Error log macro

typedef uint8_t     u8;                                          // Define 8-bit unsigned integer type
typedef uint16_t    u16;                                         // Define 16-bit unsigned integer type
typedef uint32_t    u32;                                         // Define 32-bit unsigned integer type

/**************** STC I2C register address and control command ****************/
#define STC8_I2C_SLAVE_DEV_ADDR     0x2F                        // STC8 I2C slave device address

typedef enum
{
    STC8_REG_ADDR_BATTERY    = 0x00, // Get battery information
    STC8_REG_ADDR_GET_GPIO   = 0x10, // Get GPIO input level
    STC8_REG_ADDR_SET_GPIO   = 0x18, // Set GPIO output level
```

- Now let's take a look at the content in the bsp_stc8h1kxx.c file.

1. stc8_i2c_init()

This function is used to initialize the I2C communication with the STC8H1KXX chip. It first calls the i2c_dev_register() function to register the device with the I2C slave device address of STC8 (STC8_I2C_SLAVE_DEV_ADDR) in the system, and saves the returned device handle to the global static variable stc8_handle. If the registration fails, it outputs the error message through the log macro and returns ESP_FAIL; if the registration is successful, it returns ESP_OK.

The main function of this function is to establish the communication channel for all subsequent read and write operations based on I2C, and it is the initialization entry of the entire STC8 module driver.

2. stc8_battery_info_get(Battery_info_t *bat_info)

This function is used to read battery information from the STC8H1KXX chip via the I2C bus. The current implementation adopts the method of reading byte by byte. It reads the required number of bytes for the entire Battery_info_t structure starting from the register address STC8_REG_ADDR_BATTERY in a loop. After reading each byte, the data is written into the bat_info buffer. If an error occurs during the reading process, the function will immediately record the error log and return the error code; if all bytes are successfully read, it will return ESP_OK.

Through this function, the upper-level application can obtain complete battery information including battery voltage, percentage of charge, charging status, and LED indication status.

3. stc8_gpio_get_level(int gpio_num, uint8_t* level)

This function is used to obtain the level status of the specified GPIO input pin on the STC8H1KXX chip. The function first checks whether the input gpio_num is within the valid range (i.e., less than STC8_GPIO_IN_MAX). If it is out of range, it returns an error and outputs a log. Then, it reads the level value (high or low) of the corresponding pin from the register address STC8_REG_ADDR_GET_GPIO + gpio_num using the i2c_read_reg() function, and stores it in the passed level pointer. If the reading fails, it prints the error message and returns a failure status; otherwise, it returns ESP_OK.

This function is mainly used to detect the level changes of external input signals on the STC8.

4. stc8_gpio_set_level(int gpio_num, uint8_t level)

This function is used to control the level state of the output GPIO pins on the STC8H1KXX chip. It first checks whether gpio_num is within the valid range of output pins (less than STC8_GPIO_OUT_MAX). If it exceeds this range, it records an error log and returns ESP_FAIL. Then, it uses i2c_write_reg() to write the specified level value (usually 0 or 1) to the corresponding register address STC8_REG_ADDR_SET_GPIO + gpio_num, thereby changing the logic state of the output pin. If the write fails, it outputs an error log and returns an error code; if it succeeds, it returns ESP_OK.

Through this function, various peripheral control functions such as controlling the touchscreen reset, camera reset, audio amplifier enable, and LCD backlight switch can be achieved.

5. stc8_set_pwm_duty(int pwm_num, uint8_t duty)

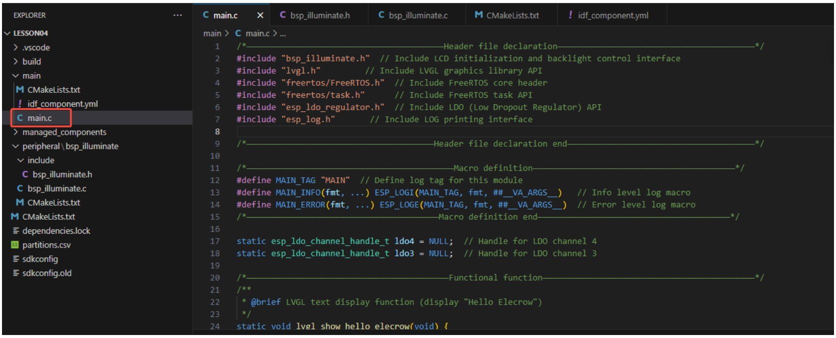
This function is used to set the duty cycle of a certain PWM channel on the STC8H1KXX chip. It first checks if pwm_num is less than STC8_PWM_MAX to ensure the requested channel is valid. If it is invalid, it records an error and returns a failure. Then, the function calls i2c_write_reg() to write the specified duty cycle value duty to the register address STC8_REG_ADDR_SET_PWM + pwm_num, to adjust the duty cycle of the PWM output signal and thereby control the brightness or power of external peripherals such as LCD backlight. If an error occurs during the writing process, the function outputs an error log and returns a failure code; if successful, it returns ESP_OK.

The main purpose of this function is to implement remote brightness or intensity control of the internal PWM module of STC8 through I2C.

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_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.

"bsp_stc8h1kxx.h": We have custom-written the expansion chip components. In this class, we will call the interface that has been written to control the screen backlight.

main > C main.c > system_init(void)

```
/*——————————————————————————————Header file declaration——————————————————————————————*/
#include "bsp_illuminate.h"  // Include LCD initialization and backlight control interface
#include "lvgl.h"           // Include LVGL graphics library API
#include "freertos/FreeRTOS.h"  // Include FreeRTOS core header
#include "freertos/task.h"      // Include FreeRTOS task API
#include "esp_log.h"         // Include LOG printing interface
#include "bsp_stc8h1kxx.h"
/*——————————————————————————————Header file declaration end——————————————————————————————*/
```

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.

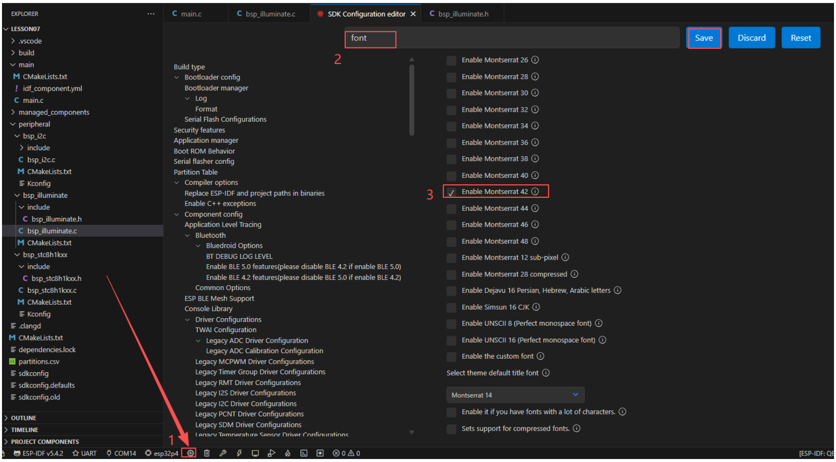
Finally, call lv_obj_center() to center the label, release the LVGL lock lvgl_port_unlock() to allow the LVGL rendering task to continue working.

(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.)

main > C main.c > lvgl_show_hello_elecrow(void)

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

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

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

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

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

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

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

Open the configuration and deactivate the font.



Remember to save it finaly.

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.

```c
/**
 * @brief Initialization failure handler (print error message repeatedly)
 */
static void init_fail_handler(const char *module_name, esp_err_t err) {
    while (1) {  // Infinite loop
        MAIN_ERROR("[%s] init failed: %s", module_name, esp_err_to_name(err));  // Print error with module name
        vTaskDelay(pdMS_TO_TICKS(1000));  // Delay 1 second between logs
    }
}
```

system_init(void):

The "system_init()" function is responsible for completing the initialization tasks of the system, mainly including the initialization of the LCD display screen, backlight, and I2C bus-related hardware. The function first initializes the I2C bus and establishes I2C communication with the STC8H1KXX chip. If the initialization fails, it calls "init_fail_handler()" to enter an infinite error printing loop;

then it initializes the LCD display screen and turns on the LCD backlight, setting the brightness to the maximum value of 100. Similarly, in case of initialization failure, it will enter an error handling loop.

The entire function ensures that the hardware is ready before the upper-level application calls LVGL or controls the backlight, and outputs corresponding log information at each successful step to guarantee the controllability and reliability of the system.

```c
static void system_init(void) {
    esp_err_t err = ESP_OK;  // Error variable initialized to OK

    // 1. Initialize LCD hardware and LVGL (important: must init before enabling backlight)

    err = i2c_init();
    if (err != ESP_OK)
        init_fail_handler("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);

    err = stc8_i2c_init();
    if (err != ESP_OK)
        init_fail_handler("stc8h1kxx", err);
    MAIN_INFO("I2C and stc8 init success");  // Print success log

    err = display_init();
    if (err != ESP_OK) {  // Check error
        init_fail_handler("LCD", err);  // Handle failure
    }
    MAIN_INFO("LCD init success");  // Print success log

    // 2. Turn on LCD backlight (brightness set to 100 = maximum)
    err = set_lcd_blight(100);  // Enable backlight
    if (err != ESP_OK) {  // Check error
        init_fail_handler("LCD Backlight", err);  // Handle failure
    }
    MAIN_INFO("LCD backlight opened (brightness: 100)");  // Print success log
}
```

Then comes the main function app_main.

Function: Program entry point. It prints the start information, calls system_init() to complete the initialization of hardware and display, then calls lvgl_show_hello_elecrow() to draw the text, and finally prints the success 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).

```c
/*——————————————————————————————Main function——————————————————————————————*/
void app_main(void) {
    MAIN_INFO("Start Hello Elecrow Display Demo");  // Print start log

    // 1. System initialization (LCD + Backlight)
    system_init();
    // 2. Show "Hello Elecrow" text
    lvgl_show_hello_elecrow();
    MAIN_INFO("Show 'Hello Elecrow' success");  // Print success log

}
/*——————————————————————————————Main function end——————————————————————————————*/
```

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

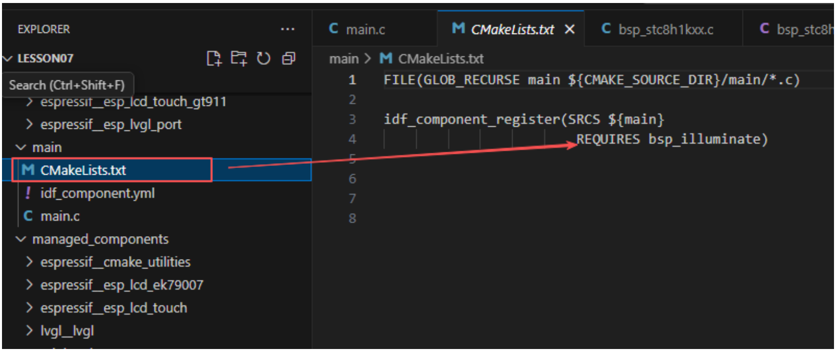
The function of this CMake configuration is as follows:

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".

Then, the "bsp_illuminate" will link the components in the "CMakeLists.txt" file within "bsp_illuminate".

This way, during the build process, ESP-IDF knows to build "bsp_illuminate" 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.

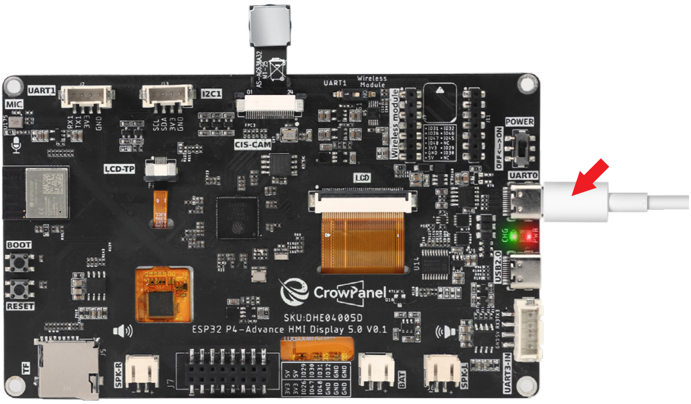
Complete Code

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

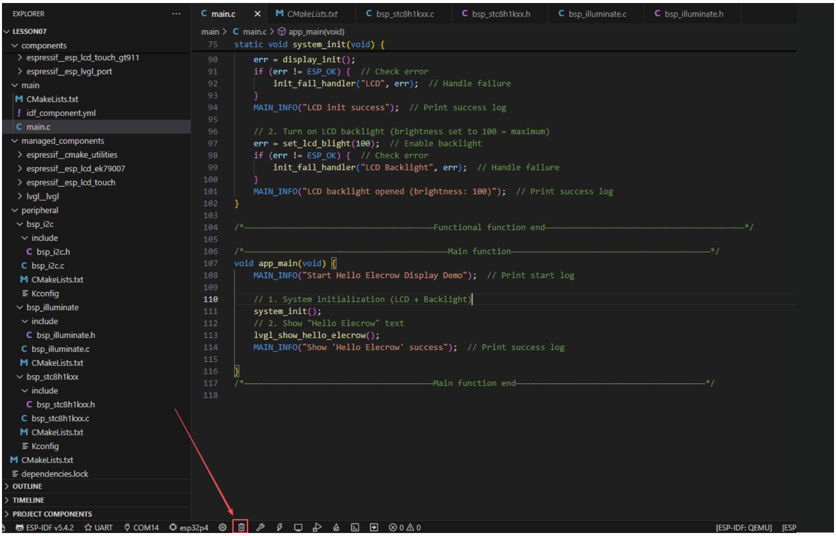
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson07-Turn_on_the_screen

Programming Steps

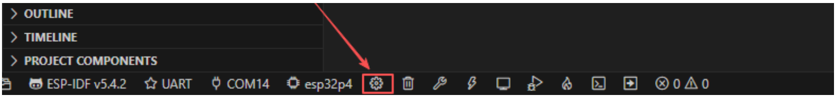
- Now the code is ready. Next, we need to flash the ESP32-P4 so that we can observe the results.
- First, we connect the Advance-P4 device to our computer host via the USB cable.



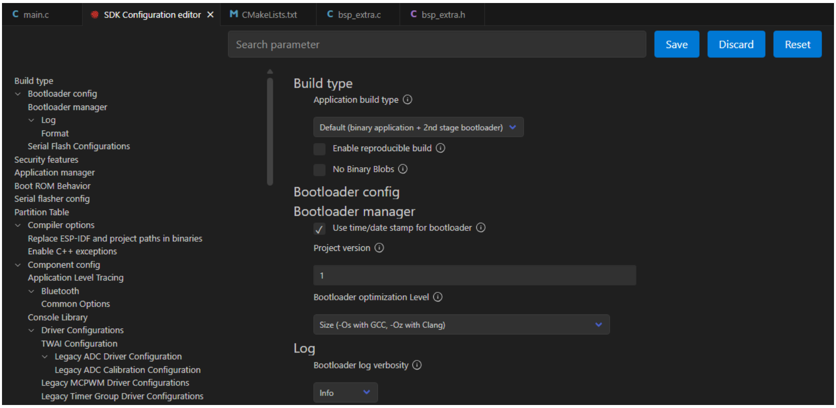
- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



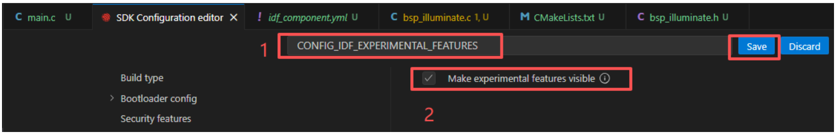
- Here, following the steps in the first section, first select the ESP-IDF version, the code upload method, the serial port, and the chip to be used.
- Then here we need to configure the SDK.
- Click the icon in the picture below.



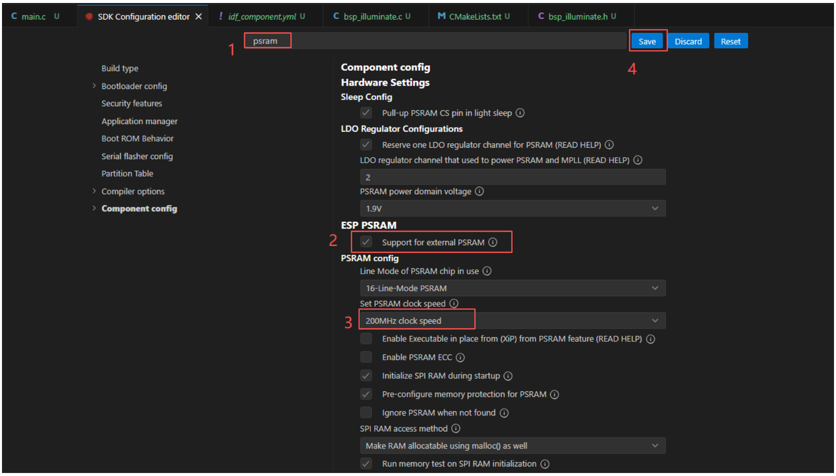
- Wait for a moment for the loading process to complete, and then you can proceed with the related SDK configuration.



After the SDK configuration is enabled, search for "CONFIG_IDF_EXPERIMENTAL_FEATURES" in the search box, check the box, and then save the configuration.

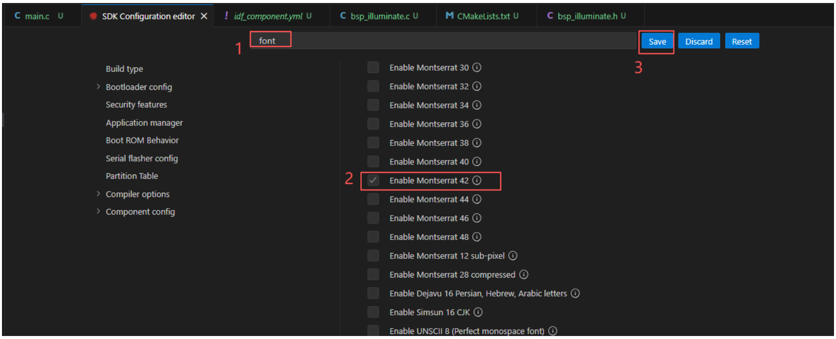


- After setting "CONFIG_IDF_EXPERIMENTAL_FEATURES", then search for PSRAM. Set it in sequence here to enable the 200M PSRAM, so that the screen can display a picture.
- Finally, remember to save the successfully configured configuration.
- Only by enabling the PSRAM option can there be sufficient RAM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

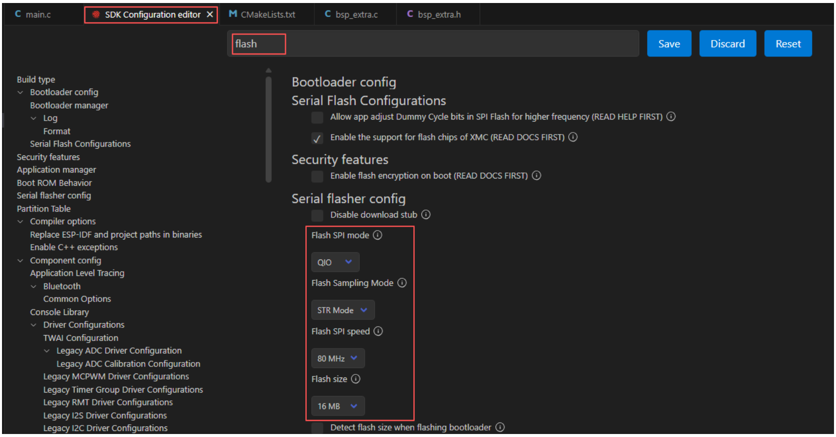


- In order to meet the font size requirements for LVGL as specified in the previous code, here we need to search for "font", open the font, so that we can use the font set by LVGL.

```
        lvgl_port_unlock();  // Unlock LVGL before returning
        return;  // Exit function
    }

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

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

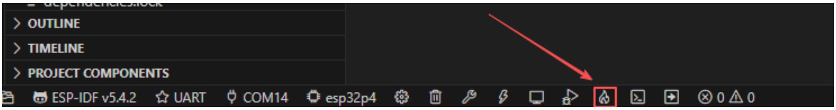


- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)



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.

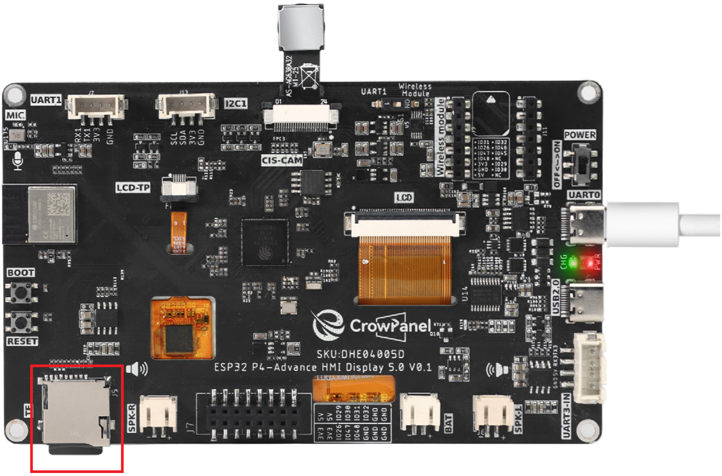
Lesson 08
SD Card File Reading

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.

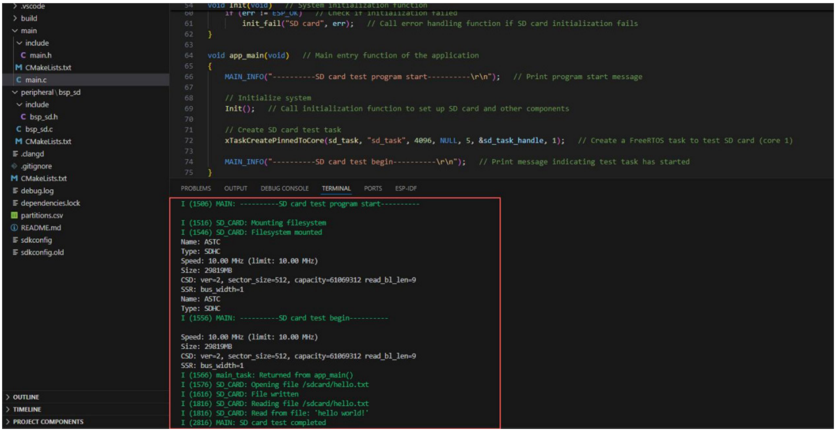
Hardware Used in This Lesson

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.

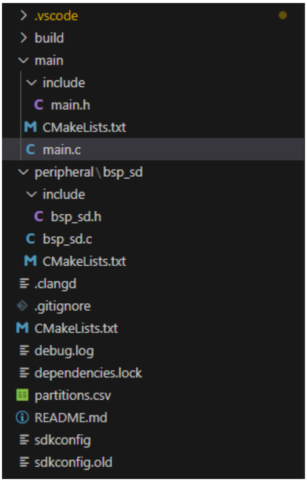


Key Explanations

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

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-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.

```
/*——————————————————————————Header file declaration——————————————————————————*/
#include <string.h>          // Include standard string manipulation functions
#include <sys/unistd.h>      // Include system calls for file handling
#include <sys/stat.h>        // Include functions for file status and permissions
#include "esp_vfs_fat.h"     // Include ESP-IDF FAT filesystem support for SD card
#include "sdmmc_cmd.h"       // Include SDMMC card command definitions and helpers
#include "driver/sdmmc_host.h"// Include SDMMC host driver for SD card communication
/*————————————————————————Header file declaration end————————————————————————*/
```

- 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".)

```c
/*----------------------------------Variable declaration----------------------------------*/
#define SD_TAG "SD_CARD"   // Tag used for logging messages related to SD card operations

#define SD_INFO(fmt, ...) ESP_LOGI(SD_TAG, fmt, ##__VA_ARGS__)   // Macro for info-level SD log output
#define SD_DEBUG(fmt, ...) ESP_LOGD(SD_TAG, fmt, ##__VA_ARGS__)  // Macro for debug-level SD log output
#define SD_ERROR(fmt, ...) ESP_LOGE(SD_TAG, fmt, ##__VA_ARGS__)  // Macro for error-level SD log output

#define EXAMPLE_MAX_CHAR_SIZE 64   // Maximum character buffer size for file read/write operations
#define SD_MOUNT_POINT "/sdcard"   // Default SD card mount point path

esp_err_t create_file(const char *filename);                     // Function to create a new file on SD card
esp_err_t write_string_file(const char *filename, char *data);   // Function to write a string to a file
esp_err_t read_string_file(const char *filename);                // Function to read a string from a file
esp_err_t write_file(const char *filename, char *data, size_t size);   // Function to write raw data to a file
esp_err_t write_file_seek(const char *filename, void *data, size_t size, int32_t seek);  // Function to write data to a specific file offset
esp_err_t read_file(const char *filename, char *data, size_t size);    // Function to read raw data from a file
esp_err_t read_file_size(const char *filename);                  // Function to read file and return its size
void get_sd_card_info(void);                                     // Function to print SD card information
esp_err_t format_sd_card();                                      // Function to format SD card (FAT filesystem)
esp_err_t sd_init();                                             // Function to initialize and mount SD card
/*----------------------------------Variable declaration end----------------------------------*/
#endif   // End of include guard
```

- 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.

peripheral > bsp_sd > C bsp_sd.c > create_file(const char *)

```c
/*----------------------------------Header file declaration----------------------------------*/
#include "bsp_sd.h"
/*----------------------------------Header file declaration end----------------------------------*/

/*----------------------------------Variable declaration----------------------------------*/
static sdmmc_card_t *card;
const char sd_mount_point[] = SD_MOUNT_POINT;
/*----------------------------------Variable declaration end----------------------------------*/

/*----------------------------------Functional function----------------------------------*/
esp_err_t create_file(const char *filename)
{
    SD_INFO("Creating file %s", filename);
    FILE *file = fopen(filename, "wb");
    if (!file)
    {
        SD_ERROR("Failed to create file");
        return ESP_FAIL;
    }
    fclose(file);
    SD_INFO("File created");
    return ESP_OK;
}
```

create_file:

Use fopen(filename, "wb") to create a file in binary write mode;Close the file immediately after successful creation; Return ESP_FAIL if opening fails.

Function: Ensure that an empty file exists on the SD card.

```
esp_err_t create_file(const char *filename)
{
    SD_INFO("Creating file %s", filename);
    FILE *file = fopen(filename, "wb");
    if (!file)
    {
        SD_ERROR("Failed to create file");
        return ESP_FAIL;
    }
    fclose(file);
    SD_INFO("File created");
    return ESP_OK;
}
```

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.

```
esp_err_t write_string_file(const char *filename, char *data)
{
    SD_INFO("Opening file %s", filename);
    FILE *file = fopen(filename, "w");
    if (!file)
    {
        SD_ERROR("Failed to open file for writing string");
        return ESP_FAIL;
    }
    fprintf(file, "%s", data);
    fclose(file);
    SD_INFO("File written");
    return ESP_OK;
}
```

read_string_file:

Open the file for reading;

Use "fgets()" to read a line of text;

Check if there is a newline character "\n", and if so, replace it with a string terminator;

Print the read content.

Function: Read a line of text content from the file and output it to the "log".

```c
esp_err_t read_string_file(const char *filename)
{
    SD_INFO("Reading file %s", filename);
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        SD_ERROR("Failed to open file for reading string");

        return ESP_FAIL;
    }
    char line[EXAMPLE_MAX_CHAR_SIZE];
    fgets(line, sizeof(line), file);
    fclose(file);

    char *pos = strchr(line, '\n');
    if (pos)
    {
        *pos = '\0';
        SD_INFO("Read a line from file: '%s'", line);
    }
    else
        SD_INFO("Read from file: '%s'", line);
    return ESP_OK;
}
```

Note: The maximum number of characters that can be read here is 64. If you need to read more characters, you will need to adjust the size.

```c
esp_err_t read_string_file(const char *filename)
{
    SD_INFO("Reading file %s", filename);
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        SD_ERROR("Failed to open file for reading string");

        return ESP_FAIL;
    }
    char line[EXAMPLE_MAX_CHAR_SIZE];
    fgets(line, sizeof(line), file);
    fclose(file);
```

main.c | SDK Configuration editor | bsp_sd.c | CMakeLists.txt | bsp_sd.h ×

peripheral > bsp_sd > include > bsp_sd.h > SD_MOUNT_POINT

```c
#ifndef _BSP_SD_H_   // Prevent multiple inclusions of this header file
#define _BSP_SD_H_

/*————————————————————————————————————————Header file declaration————————————————————————————————————————*/
#include <string.h>          // Include standard string manipulation functions
#include <sys/unistd.h>      // Include system calls for file handling
#include <sys/stat.h>        // Include functions for file status and permissions
#include "esp_vfs_fat.h"     // Include ESP-IDF FAT filesystem support for SD card
#include "sdmmc_cmd.h"       // Include SDMMC card command definitions and helpers
#include "driver/sdmmc_host.h"// Include SDMMC host driver for SD card communication
/*————————————————————————————————————————Header file declaration end————————————————————————————————————————*/

/*————————————————————————————————————————Variable declaration————————————————————————————————————————*/
#define SD_TAG "SD_CARD"   // Tag used for logging messages related to SD card operations

#define SD_INFO(fmt, ...) ESP_LOGI(SD_TAG, fmt, ##__VA_ARGS__)   // Macro for info-level SD log output
#define SD_DEBUG(fmt, ...) ESP_LOGD(SD_TAG, fmt, ##__VA_ARGS__)  // Macro for debug-level SD log output
#define SD_ERROR(fmt, ...) ESP_LOGE(SD_TAG, fmt, ##__VA_ARGS__)  // Macro for error-level SD log output

#define EXAMPLE_MAX_CHAR_SIZE 64   // Maximum character buffer size for file read/write operations
#define SD_MOUNT_POINT "/sdcard"   // Default SD card mount point path
```

write_file:

Open the file in binary write mode ("wb");

Use "fwrite()" to write the "data" in memory to the file;

If the number of bytes written is not equal to "size", it indicates a write failure;

Finally, close the file.

Function: Suitable for writing binary data or image files.

```
esp_err_t write_file(const char *filename, char *data, size_t size)
{
    size_t success_size = 0;
    FILE *file = fopen(filename, "wb");
    if (!file)
    {
        SD_ERROR("Failed to open file for writing");
        return ESP_FAIL;
    }
    success_size = fwrite(data, 1, size, file);
    if (success_size != size)
    {
        fclose(file);
        SD_ERROR("Failed to write file");
        return ESP_FAIL;
    }
    else
    {
        fclose(file);
        SD_INFO("File written");
    }
    return ESP_OK;
}
```

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.

```c
esp_err_t write_file_seek(const char *filename, void *data, size_t size, int32_t seek)
{
    size_t success_size = 0;
    FILE *file = fopen(filename, "wb");
    if (!file)
    {
        SD_ERROR("Failed to open file for writing");
        return ESP_FAIL;
    }
    if (fseek(file, seek, SEEK_SET) != 0)
    {
        SD_ERROR("Failed to seek file");
        return ESP_FAIL;
    }
    success_size = fwrite(data, 1, size, file);
    if (success_size != size)
    {
        fclose(file);
        SD_ERROR("Failed to write file");
        return ESP_FAIL;
    }
    else
    {
        fclose(file);
        SD_INFO("File written");
    }
    return ESP_OK;
}
```

read_file:

Open the file;

Use "fread()" to read a fixed-size data from the file;

If the number of bytes read does not match the expected value, an error is reported;

Otherwise, close the file and return success.

Function: Read binary files or fixed-length data blocks.

```c
esp_err_t read_file(const char *filename, char *data, size_t size)
{
    size_t success_size = 0;
    FILE *file = fopen(filename, "rb");
    if (!file)
    {
        SD_ERROR("Failed to open file for reading");
        return ESP_FAIL;
    }
    success_size = fread(data, 1, size, file);
    if (success_size != size)
    {
        fclose(file);
        SD_ERROR("Failed to read file");
        return ESP_FAIL;
    }
    else
    {
        fclose(file);
        SD_INFO("File read success");
    }
    return ESP_OK;
}
```

read_file_size:

Read all data blocks in the file in a loop;

Accumulate the "size" to get the total number of bytes of the file;

Output the total size of the file.

Function: Calculate the file size and verify the correctness of writing.

```
esp_err_t read_file_size(const char *read_filename)
{
    size_t read_success_size = 0;
    size_t size = 0;
    FILE *read_file = fopen(read_filename, "rb");
    if (!read_file)
    {
        SD_ERROR("Failed to open file for reading");
        return ESP_FAIL;
    }
    uint8_t buffer[1024];
    while ((read_success_size = fread(buffer, 1, sizeof(buffer), read_file)) > 0)
    {
        size += read_success_size;
    }
    fclose(read_file);
    SD_INFO("File read success,success size =%d", size);
    return ESP_OK;
}
```

read_write_file:

Open the source file (for reading) and the target file (for writing);

Read 1024-byte content from the source file in blocks;

Write the content to the target file;

Check whether the number of written bytes is consistent with the number of read bytes;

Finally, close the files and output the message indicating successful copying.

Function: Implement file copying operation.

```
esp_err_t read_write_file(const char *read_filename, char *write_filename)
{
    size_t read_success_size = 0;
    size_t write_success_size = 0;
    size_t size = 0;
    FILE *read_file = fopen(read_filename, "rb");
    FILE *write_file = fopen(write_filename, "wb");
    if (!read_file)
    {
        SD_ERROR("Failed to open file for reading");
        return ESP_FAIL;
    }
    if (!write_file)
    {
        SD_ERROR("Failed to open file for writing");
        return ESP_FAIL;
    }
    uint8_t buffer[1024];
    while ((read_success_size = fread(buffer, 1, sizeof(buffer), read_file)) > 0)
    {
        write_success_size = fwrite(buffer, 1, read_success_size, write_file);
        if (write_success_size != read_success_size)
        {
            SD_ERROR("inconsistent reading and writing of data");
            return ESP_FAIL;
        }
        size += write_success_size;
    }
    fclose(read_file);
    fclose(write_file);
    SD_INFO("File read and write success,success size =%d", size);
    return ESP_OK;
}
```

sd_init:

Create an "esp_vfs_fat_sdmmc_mount_config_t" configuration structure to set:

- "format_if_mount_failed = false" → Do not automatically format;
- "max_files = 5" → Maximum 5 files can be opened simultaneously;
- "allocation_unit_size = 16 * 1024" → Each cluster size is 16KB;

Initialize "sdmmc_host_t" and "sdmmc_slot_config_t":

- Set clock, command, and data line pins;
- Set bus width (1-line mode);
- Reduce the clock frequency to 10MHz to improve stability;

Call "esp_vfs_fat_sdmmc_mount()" to mount the SD card file system to "/sdcard";

If successful, print card information.

Function: Mount the SD card and establish the "FAT" file system.

```
esp_err_t sd_init()
{
    esp_err_t err = ESP_OK;
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
        .format_if_mount_failed = false,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024,
    };

    sdmmc_host_t host =  SDMMC_HOST_DEFAULT();
    host.slot = SDMMC_HOST_SLOT_0;
    host.max_freq_khz = 10000;

    sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
    slot_config.clk = GPIO_NUM_43;
    slot_config.cmd = GPIO_NUM_44;
    slot_config.d0 = GPIO_NUM_39;
    slot_config.width = 1;  // 1线SDIO
    slot_config.flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP;
    SD_INFO("Mounting filesystem");
    err = esp_vfs_fat_sdmmc_mount(sd_mount_point, &host, &slot_config, &mount_config, &card);
    if (err != ESP_OK) {
        if (err == ESP_FAIL) {
            ESP_LOGE(SD_TAG, "Failed to mount filesystem. "
                    "If you want the card to be formatted, set the EXAMPLE_FORMAT_IF_MOUNT_FAILED menuconfig option.");
        } else {
            ESP_LOGE(SD_TAG, "Failed to initialize the card (%s). "
                    "Make sure SD card lines have pull-up resistors in place.", esp_err_to_name(err));
        }
        return err;
    }
    SD_INFO("Filesystem mounted");
    sdmmc_card_print_info(stdout, card);
    return err;
}
```

get_sd_card_info:

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

```
void get_sd_card_info()
{
    sdmmc_card_print_info(stdout, card);
}
```

format_sd_card:

Call "esp_vfs_fat_sdcard_format()" to format the "FAT" file system;

Output an error message if formatting fails.

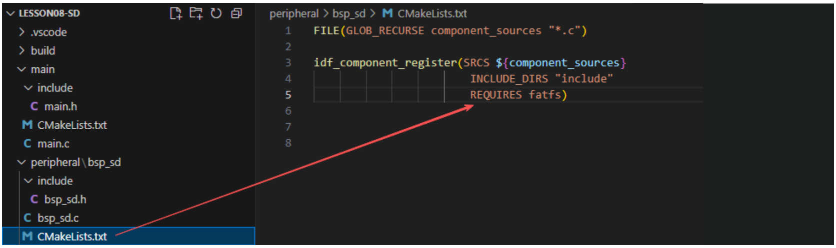
Function: Clear the SD card file system and reformat it.

```
esp_err_t format_sd_card()
{
    esp_err_t err = ESP_OK;
    err = esp_vfs_fat_sdcard_format(sd_mount_point, card);
    if (err != ESP_OK)
    {
        SD_ERROR("Failed to format FATFS (%s)", esp_err_to_name(err));
        return err;
    }
    return err;
}
```

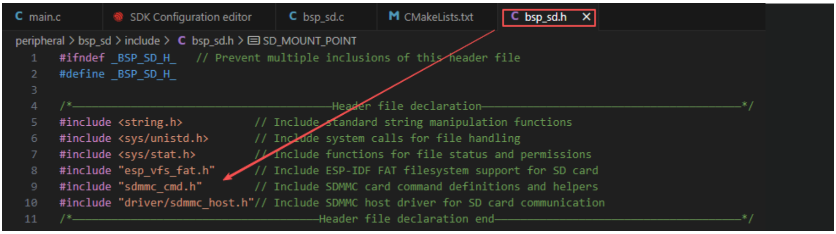
- That concludes our introduction to the "bsp_sd" component. It's sufficient for everyone to understand how to call these interfaces.

- 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.



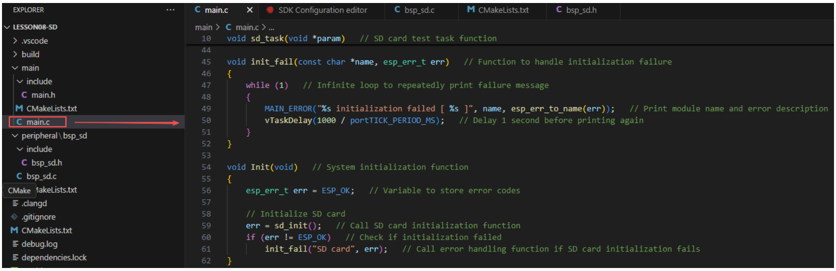
- The reason why "fatfs" is involved here is that we have called it in "bsp_sd.h" (other libraries that are system libraries do not need to be added).



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(), 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.

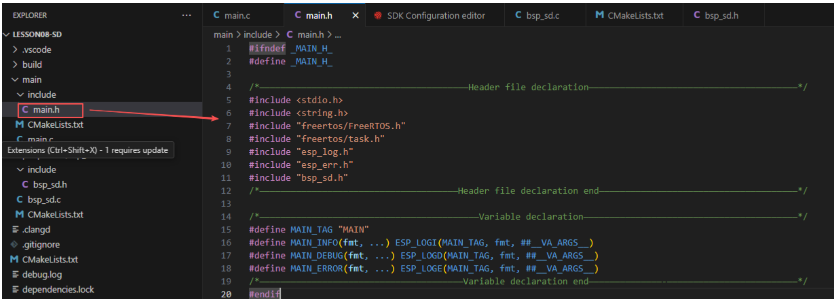
Exception Handling

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".

```
/*----------------------------------------Header file declaration----------------------------------------*/
#include "main.h"   // Include the main header file containing required definitions and declarations
/*----------------------------------------Header file declaration end----------------------------------------*/
```

- 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.

```
/*————————————————————Variable declaration————————————————————*/
TaskHandle_t sd_task_handle;    // Task handle for the SD card test task
/*——————————————————Variable declaration end——————————————————*/
```

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

```
void sd_task(void *param)   // SD card test task function
{
    esp_err_t err = ESP_OK;   // Variable to store function return values (error codes)

    const char *file_hello = SD_MOUNT_POINT "/hello.txt";   // File path for SD card test file
    char *data = "hello world!";   // Data to be written into the file

    // Get SD card information
    get_sd_card_info();   // Print SD card info such as size, type, and speed

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

        vTaskDelay(200 / portTICK_PERIOD_MS);   // Delay 200ms to allow SD card to complete internal operations

        // Read data from file
        err = read_string_file(file_hello);   // Read the content from the written file
        if (err != ESP_OK)   // Check if reading failed
        {
            MAIN_ERROR("Read file failed");   // Print error message if reading fails
        }

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

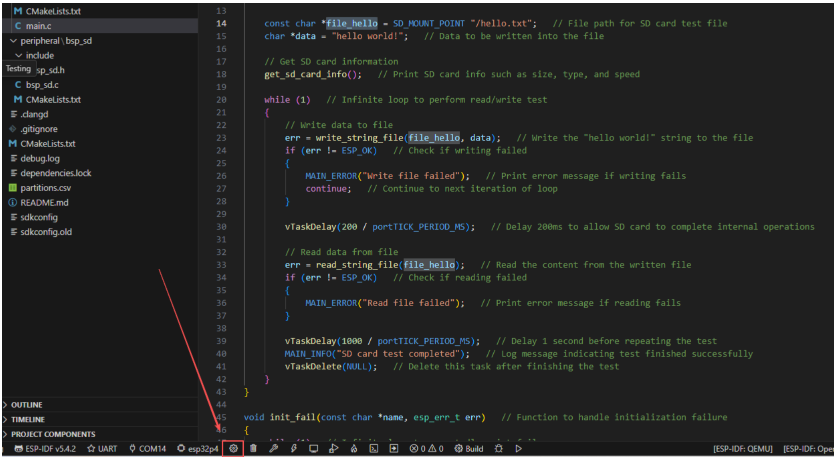
Among them:

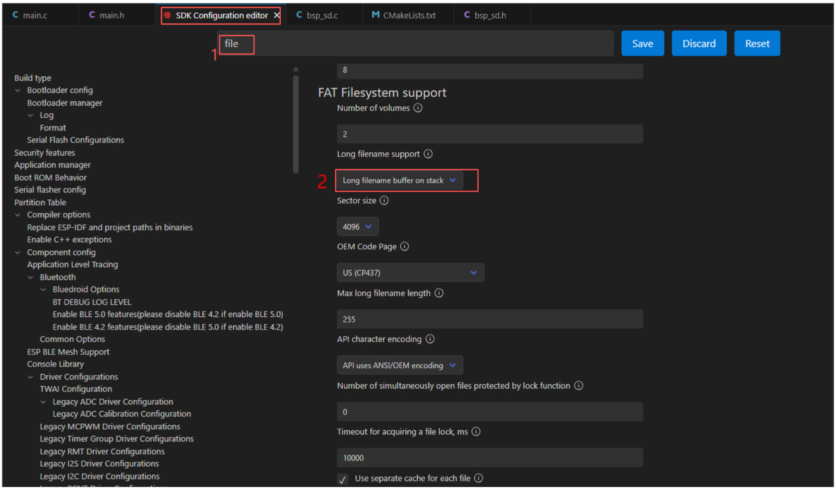
"file_hello" is the file path (usually "/sdcard/hello.txt").

"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.

```c
void init_fail(const char *name, esp_err_t err)    // Function to handle initialization failure
{
    while (1)    // Infinite loop to repeatedly print failure message
    {
        MAIN_ERROR("%s initialization failed [ %s ]", name, esp_err_to_name(err));    // Print module name and error description
        vTaskDelay(1000 / portTICK_PERIOD_MS);    // Delay 1 second before printing again
    }
```

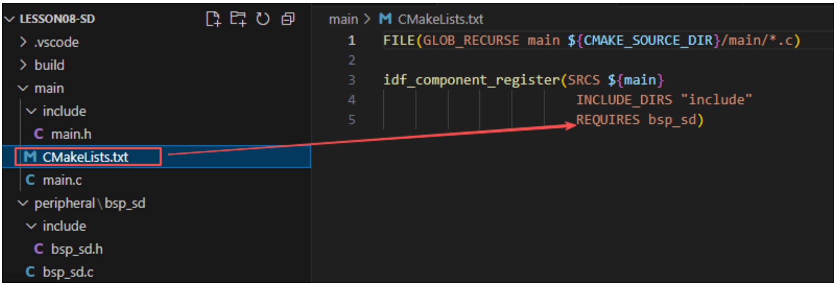
- When the module initialization fails (such as the SD card not being inserted, wrong wiring, etc.), it will cyclically print error logs and block the program.
- The function is to prevent the execution of tasks in an error state from continuing.
- The code here calls "sd_init" from the "bsp_sd" component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

```c
void Init(void)    // System initialization function
{
    esp_err_t err = ESP_OK;    // Variable to store error codes

    // Initialize SD card
    err = sd_init();    // Call SD card initialization function
    if (err != ESP_OK)    // Check if initialization failed
        init_fail("SD card", err);    // Call error handling function if SD card initialization fails
}
```

- 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.

```c
void app_main(void)   // Main entry function of the application
{
    MAIN_INFO("----------SD card test program start----------\r\n");   // Print program start message

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

    // Create SD card test task
    xTaskCreatePinnedToCore(sd_task, "sd_task", 4096, NULL, 5, &sd_task_handle, 1);   // Create a FreeRTOS task to test SD card (core 1)

    MAIN_INFO("----------SD card test begin----------\r\n");   // Print message indicating test task has started
}
/*------------------------------Functional function end------------------------------*/
```

- 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.

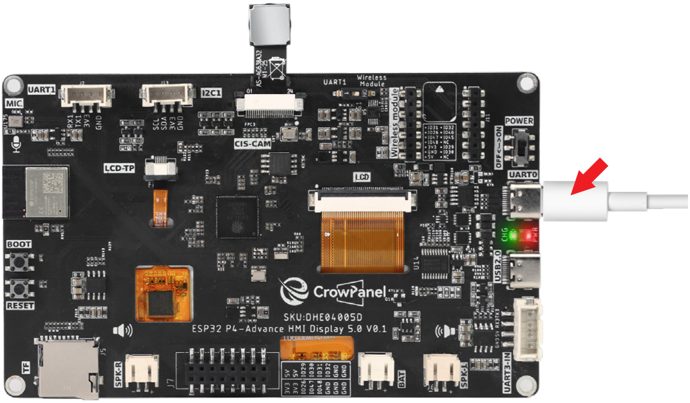
Complete Code

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

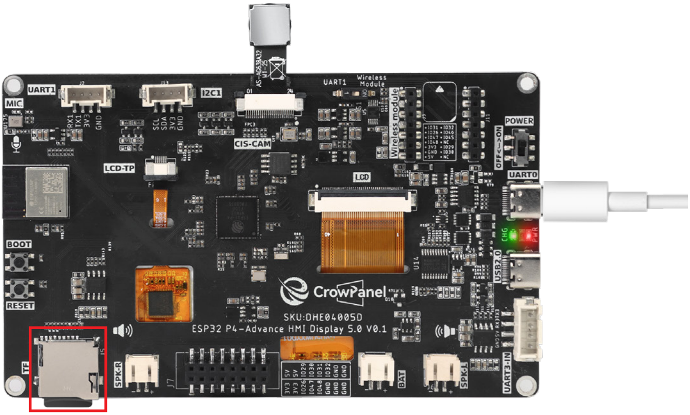
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson08-SD_Card_File_Reading

Programming Steps

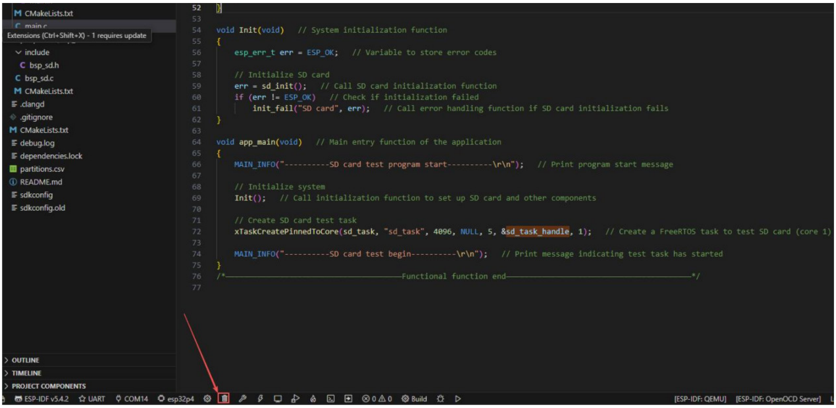
- Now the code is ready. Next, we need to flash it to the ESP32-P4 to observe the actual behavior.
- First, connect the Advance-P4 device to your computer via a USB cable.



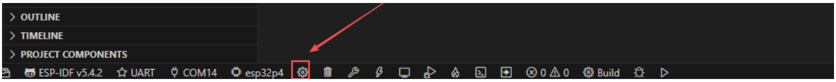
- Then insert the SD card you will use into the SD card slot of the Advance-P4.



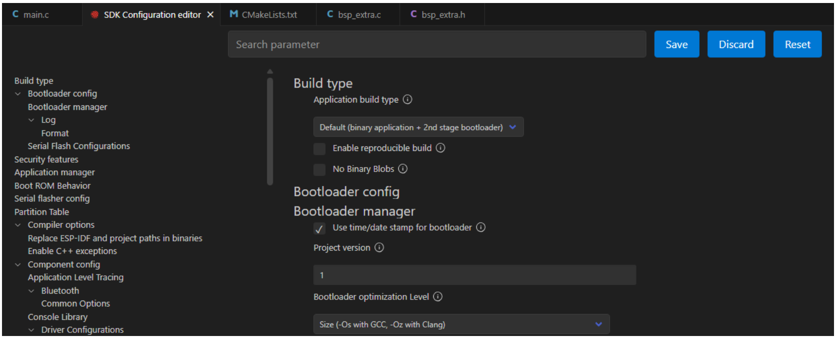
- Before starting the flashing process, first delete all files generated during compilation to restore the project to an "unbuilt" initial state. (This ensures that subsequent compilations are not affected by your previous build residues.)



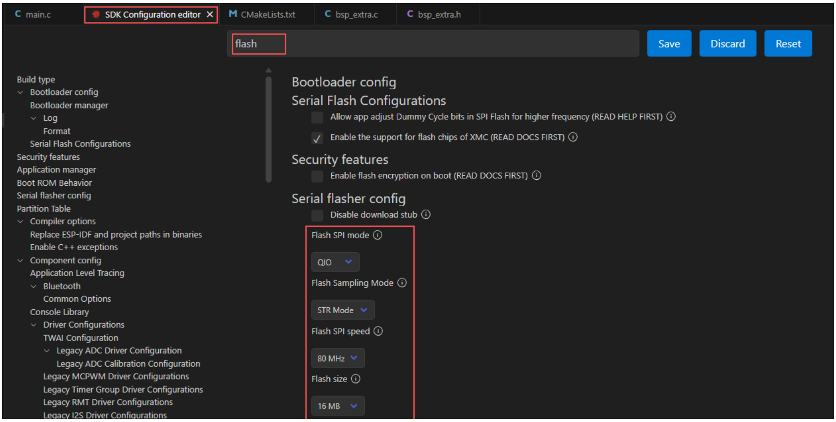
- First, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon shown in the figure.



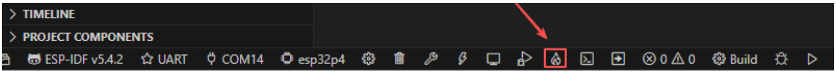
- Wait for a moment while it loads, and then you can proceed with the relevant SDK configurations.



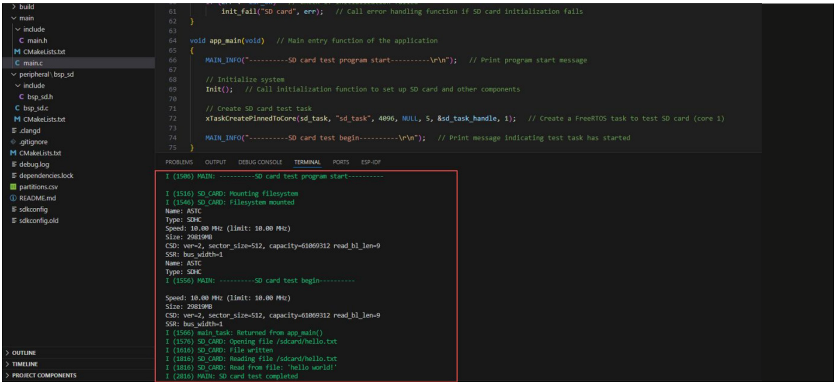
- Subsequently, search for "flash" in the search box(Ensure your flash configuration matches mine).



- 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

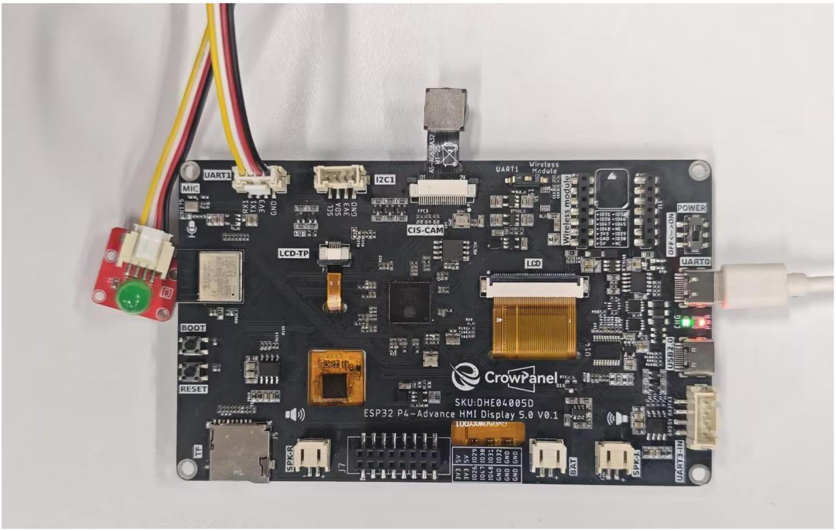
Introduction

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.

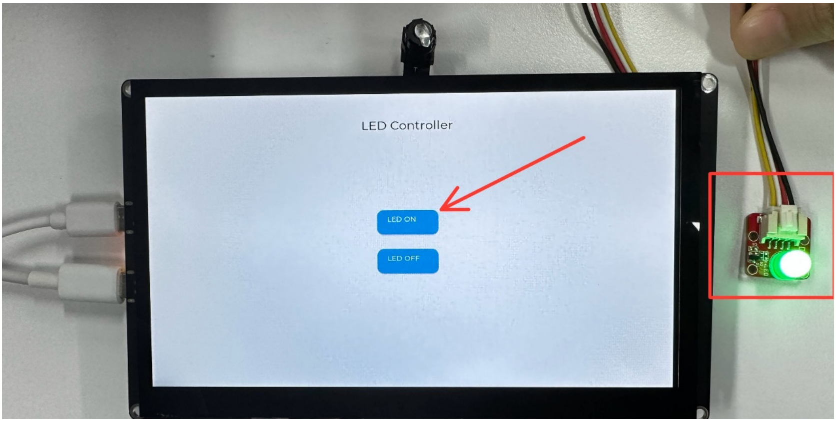
Hardware Used in This Lesson

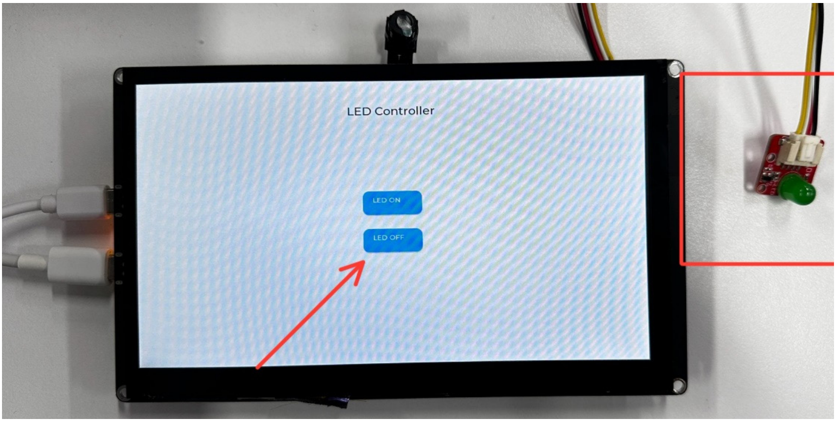
The UART1 interface on the Advance-P4 is connected to an LED.



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.



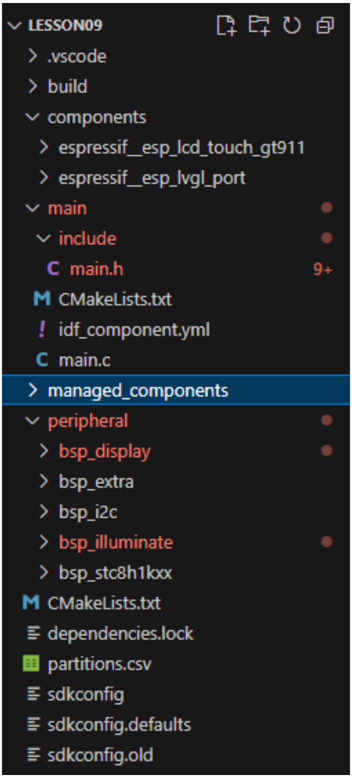


Key Explanations

- 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-5inch-ESP32-P4-HMI-AI-Display-800x480-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.(Lesson05)
- bsp_i2c: Provides I2C driver support required for touch functionality.(Lesson05)
- bsp_extra: Used to control the LED connected to the UART1 interface.(Lesson02)
- bsp_illuminate: Responsible for screen initialization, screen lighting, and LVGL initialization.(Lesson07)
- bsp_stc8h1kxx：Control the expansion chip to control the backlight of the screen.

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.

```
main.c    bsp_illuminate.c 2
peripheral > bsp_illuminate > bsp_illuminate.c > lvgl_init()
static esp_err_t display_port_init(void)  // Initialize LCD port (MIPI DSI + panel config)
        return err;
    err = esp_lcd_panel_init(panel_handle);   // Initialize panel
    if (err != ESP_OK)
        return err;
    return err;  // Return success
}

static esp_err_t lvgl_init()  // Initialize LVGL
{

    esp_err_t err = ESP_OK;
    const lvgl_port_cfg_t lvgl_cfg = {  // LVGL port configuration
        .task_priority = configMAX_PRIORITIES - 4, /* LVGL task priority */
        .task_stack = 8192*2,                      /* LVGL task stack size */
        .task_affinity = -1,                       /* Task pinned to core (-1 = no affinity) */
        .task_max_sleep_ms = 10,                   /* Max sleep in LVGL task */
        .timer_period_ms = 5,                      /* LVGL timer tick period in ms */
    };
    err = lvgl_port_init(&lvgl_cfg);  // Initialize LVGL port
```

- 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);

Significance:

- The LVGL task continuously calls lv_timer_handler() to refresh the UI, process animations, and respond to events.

```
peripheral > bsp_illuminate > C bsp_illuminate.c > ...
static esp_err_t lvgl_init()
    const lvgl_port_display_cfg_t disp_cfg = {
            .monochrome = false,                           // Color display
#if LVGL_VERSION_MAJOR >= 9
            .color_format = LV_COLOR_FORMAT_RGB565,        // Color format for LVGL 9
#endif
            .rotation = {
                .swap_xy = false,                          // No XY swap
                .mirror_x = false,                         // No X mirroring
                .mirror_y = false,                         // No Y mirroring
            },
            .flags = {
                .buff_dma = false,                         // Disable DMA buffer
                .buff_spiram = true,                       // Allocate buffer in PSRAM
                .sw_rotate = false,                        // Disable software rotation
#if LVGL_VERSION_MAJOR >= 9
                .swap_bytes = true,                        // Swap byte order for RGB565
#endif
#if CONFIG_DISPLAY_LVGL_FULL_REFRESH
                .full_refresh = true,                      // Enable full screen refresh
#else
                .full_refresh = false,                     // Disable full screen refresh
#endif
#if CONFIG_DISPLAY_LVGL_DIRECT_MODE
                .direct_mode = true,                       // Enable direct rendering mode
#else
                .direct_mode = true,                       // Direct rendering mode (default)
#endif
            },
    };
    const lvgl_port_display_rgb_cfg_t lvgl_rgb_cfg = {
            .flags = {
#if CONFIG_DISPLAY_LVGL_AVOID_TEAR
                .avoid_tearing = true,                     // Enable tearing avoidance
#else
                .avoid_tearing = true,                     // Enable tearing avoidance (default)
#endif
            },
    };
    my_lvgl_disp = lvgl_port_add_disp_rgb(&disp_cfg, &lvgl_rgb_cfg); // Add LVGL RGB display
```

- This step registers the display screen with LVGL through lvgl_port_add_disp_dsi(), serving as a bridge between "LVGL" and the "screen".
- The initialization content includes:
    - io_handle: The physical communication interface of the screen (such as "MIPI", "SPI", "RGB", etc.)
    - panel_handle: Screen panel driver handle
    - buffer_size: Frame buffer size (used for rendering images)
    - double_buffer: Whether to use double buffering (prevents tearing and improves refresh smoothness)
    - hres/vres: Screen resolution
    - color_format: Color format (e.g., "RGB565")
    - 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.

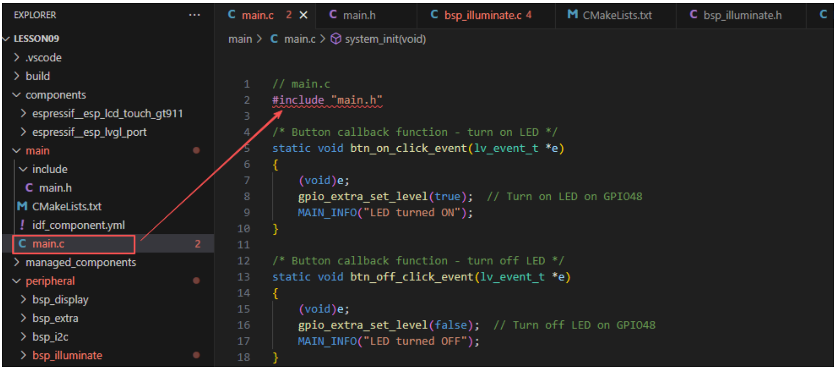
peripheral > bsp_illuminate > C bsp_illuminate.c > lvgl_init()

```c
static esp_err_t lvgl_init()
    const lvgl_port_display_cfg_t disp_cfg = {
        .flags = {
#endif
        },
    };
    const lvgl_port_display_rgb_cfg_t lvgl_rgb_cfg = {
        .flags = {
#if CONFIG_DISPLAY_LVGL_AVOID_TEAR
            .avoid_tearing = true,                    // Enable tearing avoidance
#else
            .avoid_tearing = true,                    // Enable tearing avoidance (default)
#endif
        },
    };
    my_lvgl_disp = lvgl_port_add_disp_rgb(&disp_cfg, &lvgl_rgb_cfg); // Add LVGL RGB display
    if (my_lvgl_disp == NULL)
    {
        err = ESP_FAIL;                               // Set error if display creation fails
        ILLUMINATE_ERROR("LVGL rgb port add fail");   // Log error
    }

    const lvgl_port_touch_cfg_t touch_cfg = {
        .disp = my_lvgl_disp,
        .handle = tp,
    };
    my_touch_indev = lvgl_port_add_touch(&touch_cfg);
    if (my_touch_indev == NULL)
    {
        err = ESP_FAIL;
        ILLUMINATE_ERROR("LVGL touch port add fail");
    }
    return err;                                       // Return success or failure
}
```

- 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.

```c
static void system_init(void) {
    esp_err_t err = ESP_OK;

    // 1. Initialize I2C (required for touch chip)
    MAIN_INFO("Initializing I2C...");
    err = i2c_init();
    if (err != ESP_OK) init_fail_handler("I2C", err);
    MAIN_INFO("I2C init success");

    vTaskDelay(200 / portTICK_PERIOD_MS);

    // 2. Initialize stc8
    err = stc8_i2c_init();
    if (err != ESP_OK)
        init_fail_handler("stc8h1kxx", err);
    MAIN_INFO("stc8 init success");  // Print success log

    // 3. Initialize touch panel (low-level driver)
    MAIN_INFO("Initializing touch panel...");
    err = touch_init();
    if (err != ESP_OK) init_fail_handler("Touch", err);
    MAIN_INFO("Touch panel init success");

    // 4. Initialize LCD hardware and LVGL (must initialize before turning on backlight)
    err = display_init();
    if (err != ESP_OK) init_fail_handler("LCD", err);
    MAIN_INFO("LCD init success");

    // 5. Turn on LCD backlight (brightness set to 100 = max)
    err = set_lcd_blight(100);
    if (err != ESP_OK) init_fail_handler("LCD Backlight", err);
    MAIN_INFO("LCD backlight opened (brightness: 100)");
```

- 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.

```c
static void system_init(void) {
    esp_err_t err = ESP_OK;

    // 1. Initialize I2C (required for touch chip)
    MAIN_INFO("Initializing I2C...");
    err = i2c_init();
    if (err != ESP_OK) init_fail_handler("I2C", err);
    MAIN_INFO("I2C init success");

    vTaskDelay(200 / portTICK_PERIOD_MS);

    // 2. Initialize stc8
    err = stc8_i2c_init();
    if (err != ESP_OK)
        init_fail_handler("stc8h1kxx", err);
    MAIN_INFO("stc8 init success");  // Print success log
```

- Initialize the STC8 expansion chip and connect the I2C control to the STC8 expansion chip. This will enable the expansion chip to control the screen backlight, laying a solid foundation.

```c
// 2. Initialize stc8
err = stc8_i2c_init();
if (err != ESP_OK)
    init_fail_handler("stc8h1kxx", err);
MAIN_INFO("stc8 init success");  // Print success log

// 3. Initialize touch panel (low-level driver)
MAIN_INFO("Initializing touch panel...");
err = touch_init();
if (err != ESP_OK) init_fail_handler("Touch", err);
MAIN_INFO("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.

```c
    // 4. Initialize LCD hardware and LVGL (must initialize before turning on backlight)
    err = display_init();
    if (err != ESP_OK) init_fail_handler("LCD", err);
    MAIN_INFO("LCD init success");

    // 5. Turn on LCD backlight (brightness set to 100 = max)
    err = set_lcd_blight(100);
    if (err != ESP_OK) init_fail_handler("LCD Backlight", err);
    MAIN_INFO("LCD backlight opened (brightness: 100)");
```

- 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)
MAIN_INFO("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
MAIN_INFO("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();
MAIN_INFO("UI created successfully");
```

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.

```c
/* Create LED control UI */
static void create_led_control_ui(void)
{
    // Create main screen
    lv_obj_t *scr = lv_scr_act();
    lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFFFF), LV_PART_MAIN);  // Set white background

    // Create title label
    lv_obj_t *label = lv_label_create(scr);
    lv_label_set_text(label, "LED Controller");
    lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 50);
    // Font size
    lv_obj_set_style_text_font(label, &lv_font_montserrat_24, 0);

    // Create LED ON button
    lv_obj_t *btn_on = lv_btn_create(scr);
    lv_obj_set_size(btn_on, 120, 50);
    lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
    lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);

    // ON button label
    lv_obj_t *label_on = lv_label_create(btn_on);
    lv_label_set_text(label_on, "LED ON");

    // Create LED OFF button
    lv_obj_t *btn_off = lv_btn_create(scr);
    lv_obj_set_size(btn_off, 120, 50);
    lv_obj_align(btn_off, LV_ALIGN_CENTER, 0, 40);
    lv_obj_add_event_cb(btn_off, btn_off_click_event, LV_EVENT_CLICKED, NULL);

    // OFF button label
    lv_obj_t *label_off = lv_label_create(btn_off);
    lv_label_set_text(label_off, "LED OFF");
}
```

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()

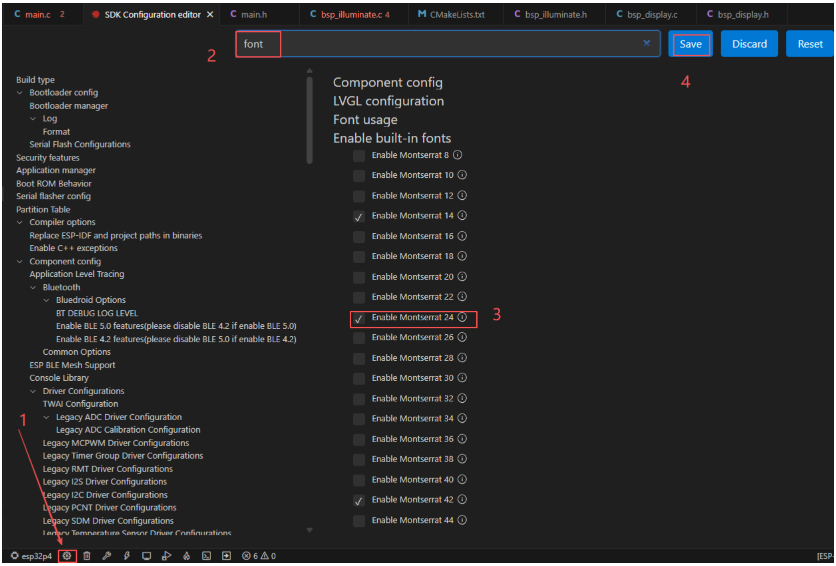
Sets the background color of this screen to white (0xFFFFFF).


```c
/* Create LED control UI */
static void create_led_control_ui(void)
{
    // Create main screen
    lv_obj_t *scr = lv_scr_act();
    lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFFFF), LV_PART_MAIN);  // Set white background

    // Create title label
    lv_obj_t *label = lv_label_create(scr);
    lv_label_set_text(label, "LED Controller");
    lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 50);
    // Font size
    lv_obj_set_style_text_font(label, &lv_font_montserrat_24, 0);

    // Create LED ON button
    lv_obj_t *btn_on = lv_btn_create(scr);
    lv_obj_set_size(btn_on, 120, 50);
    lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
    lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);

    // ON button label
    lv_obj_t *label_on = lv_label_create(btn_on);
    lv_label_set_text(label_on, "LED ON");
```

- 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.

```
/* Create LED control UI */
static void create_led_control_ui(void)
{
    // Create main screen
    lv_obj_t *scr = lv_scr_act();
    lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFFFF), LV_PART_MAIN);  // Set white background

    // Create title label
    lv_obj_t *label = lv_label_create(scr);
    lv_label_set_text(label, "LED Controller");
    lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 50);
    // Font size
    lv_obj_set_style_text_font(label, &lv_font_montserrat_24, 0);

    // Create LED ON button
    lv_obj_t *btn_on = lv_btn_create(scr);
    lv_obj_set_size(btn_on, 120, 50);
    lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
    lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);
```

- Here, if you want to set the LVGL font size to 24, you need to open the SDK configuration and activate this font. Then it will be usable.



- '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".

```c
/* Create LED control UI */
static void create_led_control_ui(void)
{
    // Create main screen
    lv_obj_t *scr = lv_scr_act();
    lv_obj_set_style_bg_color(scr, lv_color_hex(0xFFFFFF), LV_PART_MAIN);  // Set white background

    // Create title label
    lv_obj_t *label = lv_label_create(scr);
    lv_label_set_text(label, "LED Controller");
    lv_obj_align(label, LV_ALIGN_TOP_MID, 0, 50);
    // Font size
    lv_obj_set_style_text_font(label, &lv_font_montserrat_24, 0);

    // Create LED ON button
    lv_obj_t *btn_on = lv_btn_create(scr);
    lv_obj_set_size(btn_on, 120, 50);
    lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
    lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);

    // ON button label
    lv_obj_t *label_on = lv_label_create(btn_on);
    lv_label_set_text(label_on, "LED ON");

    // Create LED OFF button
    lv_obj_t *btn_off = lv_btn_create(scr);
    lv_obj_set_size(btn_off, 120, 50);
    lv_obj_align(btn_off, LV_ALIGN_CENTER, 0, 40);
    lv_obj_add_event_cb(btn_off, btn_off_click_event, LV_EVENT_CLICKED, NULL);

    // OFF button label
    lv_obj_t *label_off = lv_label_create(btn_off);
    lv_label_set_text(label_off, "LED OFF");
}
```

- 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.

```c
    // Create LED ON button
    lv_obj_t *btn_on = lv_btn_create(scr);
    lv_obj_set_size(btn_on, 120, 50);
    lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
    lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);

    // ON button label
    lv_obj_t *label_on = lv_label_create(btn_on);
    lv_label_set_text(label_on, "LED ON");

```

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.

```c
    // Create LED ON button
    lv_obj_t *btn_on = lv_btn_create(scr);
    lv_obj_set_size(btn_on, 120, 50);
    lv_obj_align(btn_on, LV_ALIGN_CENTER, 0, -40);
    lv_obj_add_event_cb(btn_on, btn_on_click_event, LV_EVENT_CLICKED, NULL);
```

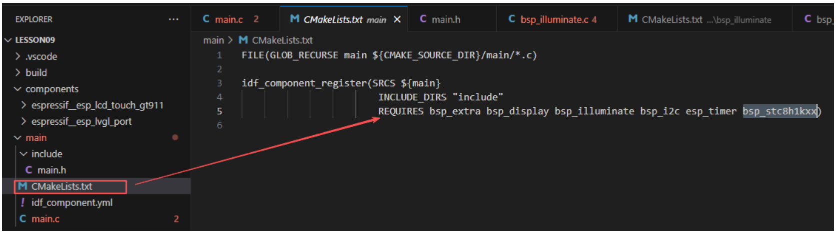
```c
/* Button callback function - turn on LED */
static void btn_on_click_event(lv_event_t *e)
{
    (void)e;
    gpio_extra_set_level(true);  // Turn on LED on GPIO48
    MAIN_INFO("LED turned ON");
}

/* Button callback function - turn off LED */
static void btn_off_click_event(lv_event_t *e)
{
    (void)e;
    gpio_extra_set_level(false);  // Turn off LED on GPIO48
    MAIN_INFO("LED turned OFF");
}
```

- 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).

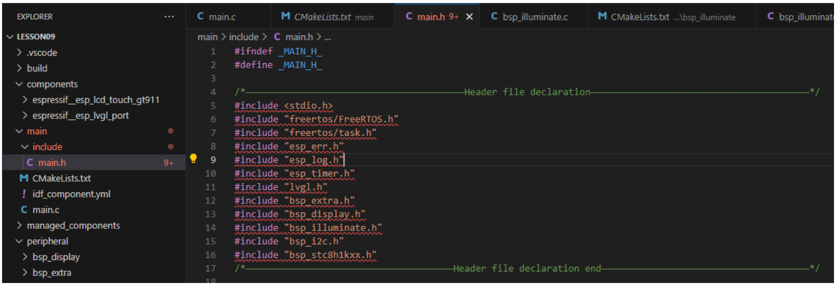
```c
void app_main(void)
{
    MAIN_INFO("Starting LED control application...");

    // System initialization (including LDO, LCD, touch, LED and all hardware)
    system_init();

    MAIN_INFO("System initialized successfully");

    while (1) {
        // Other background tasks can be placed here; maintain low power
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

- 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", "esp_timer" and "bsp_stc8h1kxx".

- This way, during the build process, ESP-IDF knows to build these five components first, then build the "main" component.



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



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.

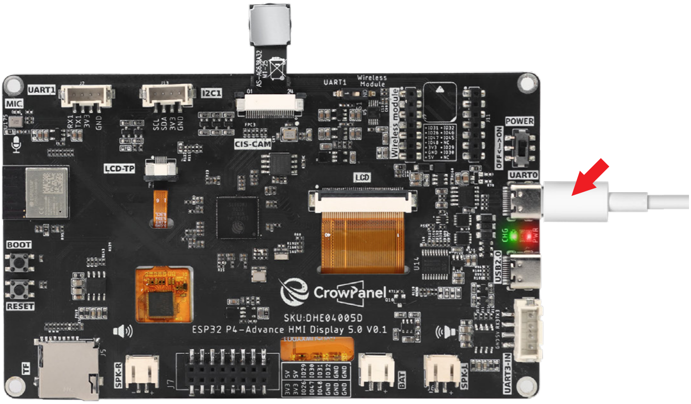
Complete Code

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

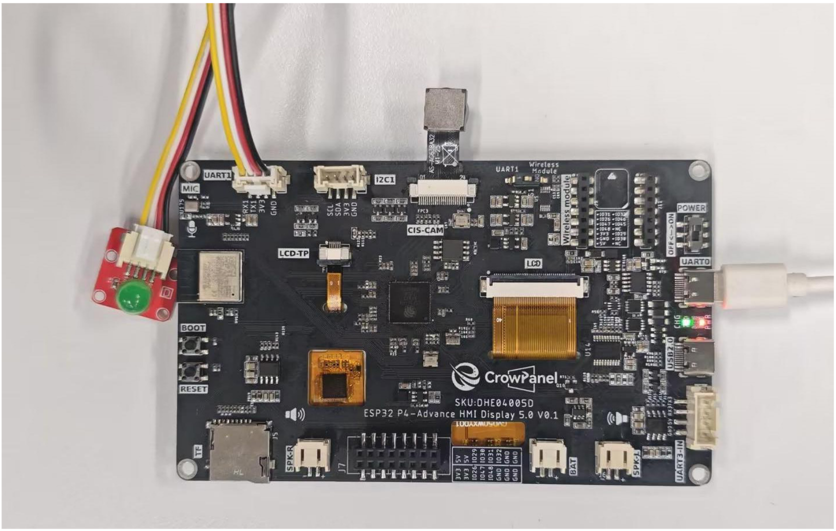
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson09-LVGL_Lighting_Control

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.

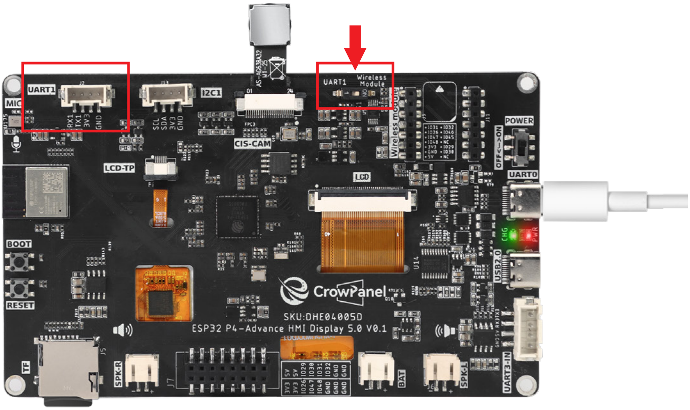


- Also, remember to connect an LED to the UART1 interface.

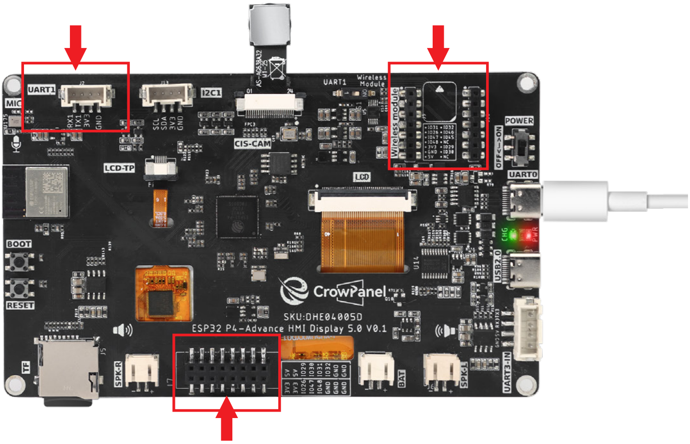


Then, switch the toggle switch on the 5-inch Advance-P4 to the UART1 position.

Only in this way can the UART1 interface be used.



- This is the design on the hardware side.



Switch to UART1 port:

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

Switch to Wireless Module port:

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

Alternatively, the expansion header at the bottom can also be used.

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

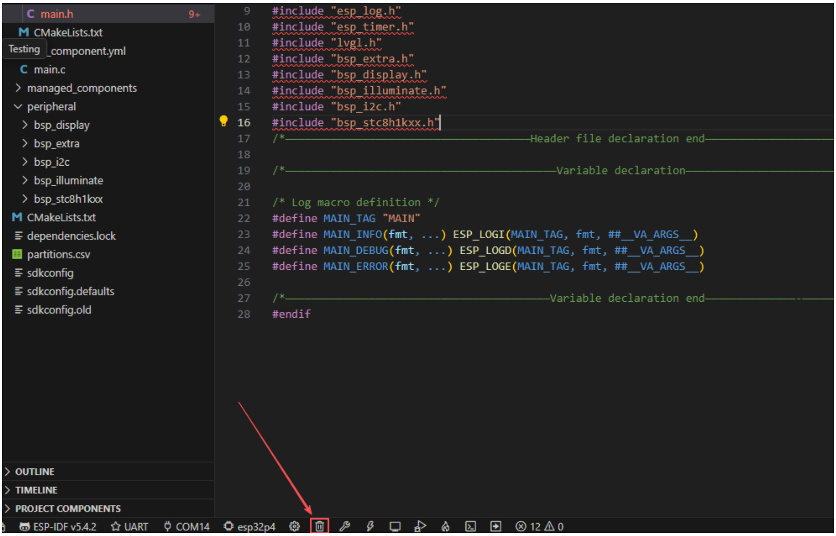
Summary:

The UART1 interface and the Wireless Module can only be used when switched to the corresponding port.

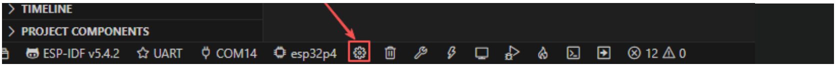
The expansion header at the bottom can be used regardless of the position of the mode switch, but it cannot be used simultaneously with the above interfaces. (When used simultaneously, only one of the three interfaces can be selected.)

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

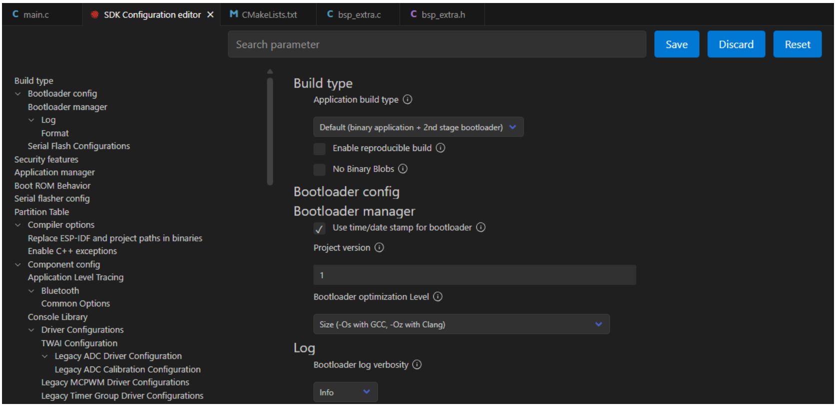
- Before starting the burning process, delete all the compiled files and restore the project to its initial "uncompiled" state. (This ensures that the subsequent compilation will not be affected by your previous actions.)



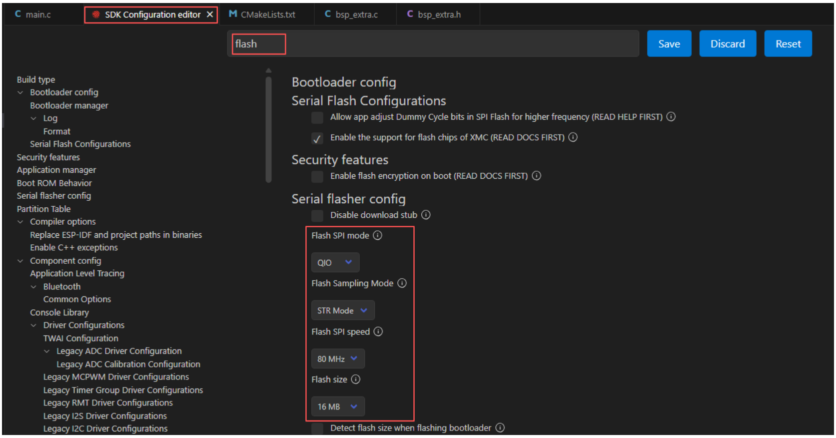
- Here, follow the steps from the first lesson to first select the ESP-IDF version, code upload method, serial port number, and the target chip (ESP32-P4).
- Next, we need to configure the SDK.
- Click on the icon shown in the figure below.



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



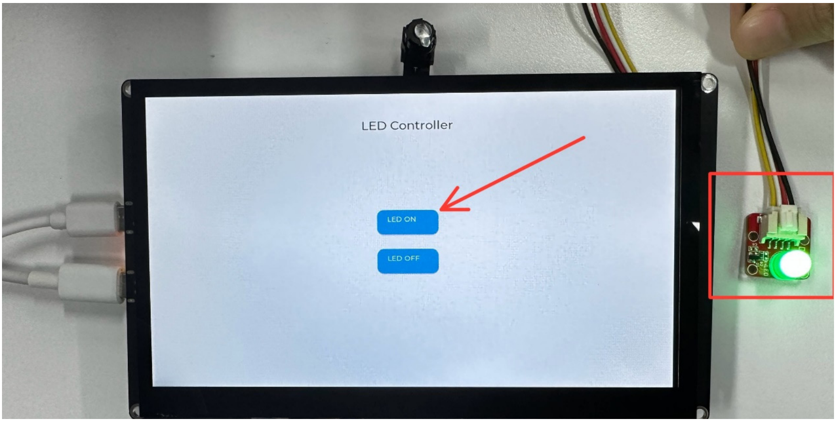
- Then, search for "flash" in the search box. (Make sure your flash settings are the same as mine.)

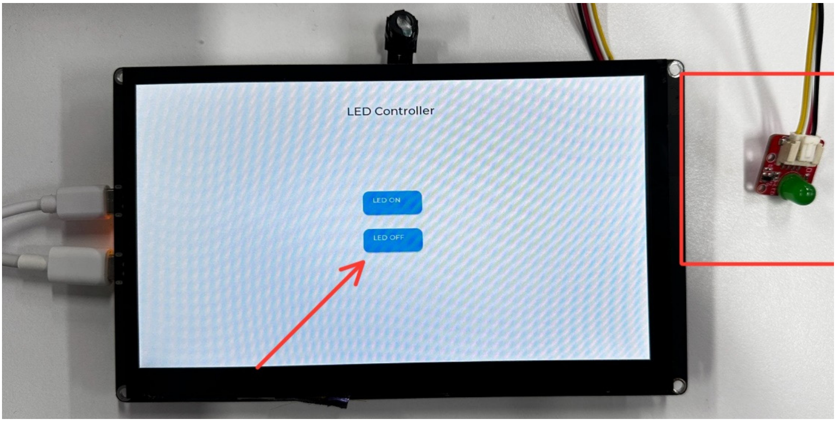


- After completing the configuration, remember to save your settings.
- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we also want to share a very convenient feature: there is a single button that can execute compilation, uploading, and opening the monitor in one go.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


LED Controller
LED ON
LED OFF


LED Controller
LED ON
LED OFF

Lesson 10
Temperature and Humidity

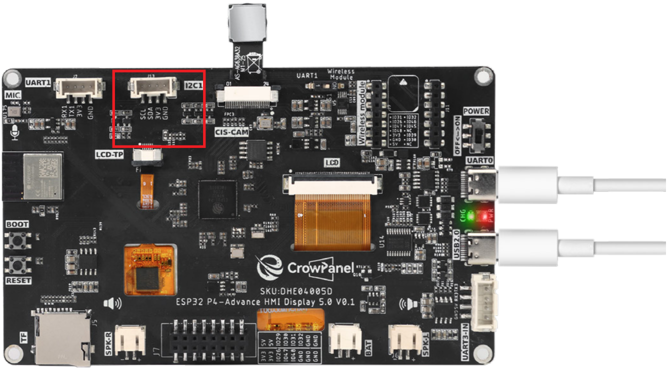
Introduction

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.

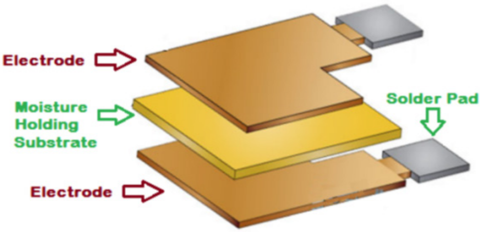
The key learning focus of this lesson is the use of the I2C interface. We will reuse the I2C component and screen display component covered in previous lessons, and additionally introduce a new temperature and humidity component: bsp_dht20.

Hardware Used in This Lesson

I2C Interface on the Advance-P4



Temperature and humidity sensor Schematic Diagram



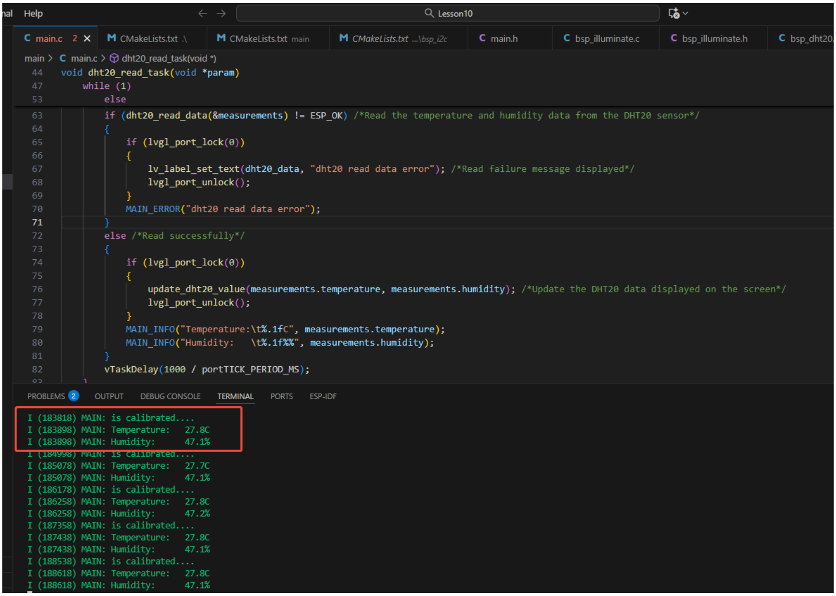
- In the temperature and humidity sensor, humidity detection relies on hygroscopic materials. These materials absorb or release water in response to changes in environmental humidity, thereby altering their own electrical properties (such as resistance, capacitance, etc.). The sensor obtains humidity 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.

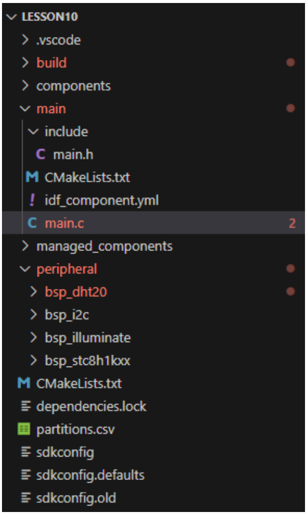
- Next, let's focus on understanding the bsp_dht20 component. (The bsp_i2c component and bsp_dht20 component were explained in detail in previous courses.)

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

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-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.

```
/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include <string.h>           //References for string Function-related API Functions
#include "freertos/FreeRTOS.h" //References for Freertos Function-related API Functions
#include "freertos/task.h"     //References for Freertos Task Function-related API Functions
#include "esp_log.h"           //References for LOG Printing Function-related API Functions
#include "esp_err.h"           //References for Error Type Function-related API Functions
#include "esp_timer.h"         //References for high-precision timers Function-related API Functions
#include "bsp_i2c.h"
/*———————————————————————————————————Header file declaration end——————————————————————————————————*/
```

- Then, we declare the variables we need to use, as well as the functions—whose specific implementations are in "bsp_dht20.c".
- Centralizing these declarations in "bsp_dht20.h" is for the convenience of calling and management. (We will understand their roles when they are used in "bsp_dht20.c".)

```
/*-----------------------------------Variable declaration-----------------------------------*/
#define DHT20_TAG "DHT20"
#define DHT20_INFO(fmt, ...) ESP_LOGI(DHT20_TAG, fmt, ##__VA_ARGS__)
#define DHT20_DEBUG(fmt, ...) ESP_LOGD(DHT20_TAG, fmt, ##__VA_ARGS__)
#define DHT20_ERROR(fmt, ...) ESP_LOGE(DHT20_TAG, fmt, ##__VA_ARGS__)

#define DHT20_I2C_ADDRESS 0x38 // The 7-bit I2C address of DHT20

#define DHT20_MEASURE_TIMEOUT 1000 // Measurement timeout time of DHT20

typedef struct dht20_data
{
    float temperature;  // The measured temperature data
    float humidity;     // the measured humidity data
    uint32_t raw_humid; // Intermediate quantity for humidity data conversion
    uint32_t raw_temp;  // Intermediate quantity for temperature data conversion
} dht20_data_t;

esp_err_t dht20_begin(void);                     // Initialization of DHT20 sensor
esp_err_t dht20_is_calibrated(void);             // The function for determining whether the DHT20 sensor is ready or not
esp_err_t dht20_read_data(dht20_data_t *data); // DHT20 Sensor Temperature and Humidity Data Reading Function

/*-----------------------------------Variable declaration end-----------------------------------*/
#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.

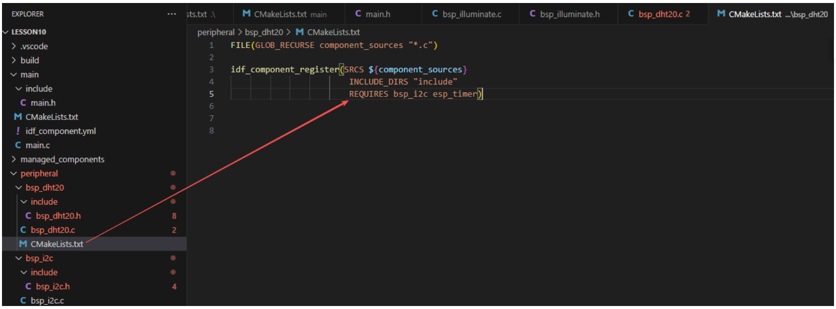
- The dht20_crc8 function: Calculates the checksum of data using the CRC8 algorithm specified in the DHT20 protocol (polynomial 0x31) to verify the integrity of received data. It is used after reading sensor data (e.g., in dht20_read_data) to compare the calculated CRC value with the CRC byte returned 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).

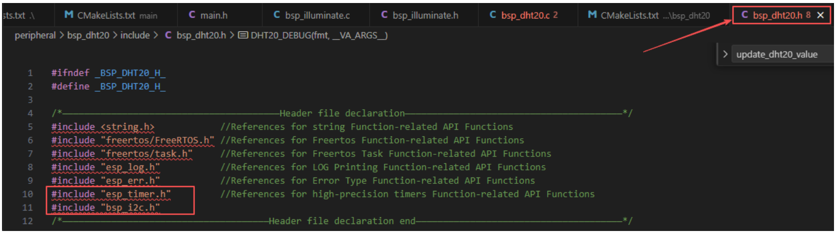
- That concludes our introduction to the bsp_dht20 component—you only need to understand how to call these interfaces.

- 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.



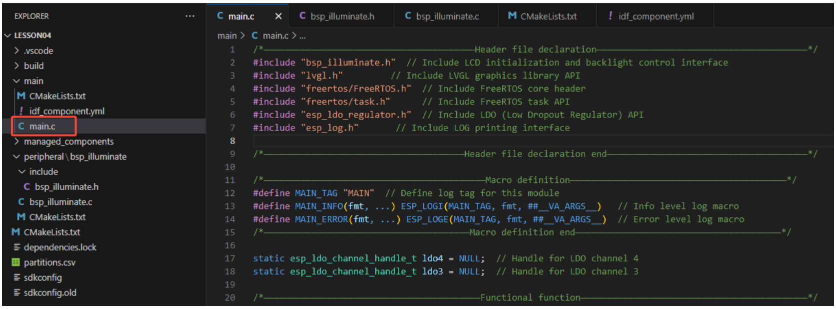
- The reason we include bsp_i2c and esp_timer here is that they are explicitly used in "bsp_dht20.h". (Other system libraries do not need to be added because they are already integrated into the ESP-IDF framework by default.)



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, call the Init function to initialize the following components in sequence: the I2C bus (the foundation for sensor communication), the STC8 expansion chip (which controls the screen backlight here), the DHT20 sensor (to complete registration and status calibration), and the display module. If the initialization fails, report the error in the 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.

```c
#ifndef _MAIN_H_
#define _MAIN_H_

/*—————————————————————————————————————Header file declaration—————————————————————————————————————*/
#include <stdio.h>
#include "string.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "esp_private/esp_clk.h"
#include "esp_ldo_regulator.h"  // Include LDO (Low Dropout Regulator) API
#include "bsp_i2c.h"
#include "bsp_illuminate.h"
#include "bsp_dht20.h"
#include "bsp_stc8h1kxx.h"
/*—————————————————————————————————Header file declaration end—————————————————————————————————*/

/*—————————————————————————————————————Variable declaration—————————————————————————————————————*/

#define MAIN_TAG "MAIN"
#define MAIN_INFO(fmt, ...) ESP_LOGI(MAIN_TAG, fmt, ##__VA_ARGS__)
#define MAIN_DEBUG(fmt, ...) ESP_LOGD(MAIN_TAG, fmt, ##__VA_ARGS__)
#define MAIN_ERROR(fmt, ...) ESP_LOGE(MAIN_TAG, fmt, ##__VA_ARGS__)

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

- Here, it includes libraries for the three components used:
  - bsp_i2c: Since the temperature and humidity sensor communicates via I2C.
  - bsp_illuminate: Used for displaying temperature and humidity values on the screen.
  - bsp_dht20: For initializing the temperature and humidity sensor and obtaining its readings.
  - bsp_stc8h1kxx： By expanding the chip to control the backlight of the screen.

C main.c   C main.h 9+ ×

main > include > C main.h > ...

```c
#ifndef _MAIN_H_
#define _MAIN_H_

/*———————————————————————————————Header file declaration———————————————————————————————*/
#include <stdio.h>
#include "string.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "esp_private/esp_clk.h"
#include "esp_ldo_regulator.h"   // Include LDO (Low Dropout Regulator) API
#include "bsp_i2c.h"
#include "bsp_illuminate.h"
#include "bsp_dht20.h"
#include "bsp_stc8h1kxx.h"
/*——————————————————————————————Header file declaration end——————————————————————————————*/
```

- stdio.h, string.h: Provide basic input/output (e.g., printf) and string processing (e.g., memset, snprintf) functions, supporting operations such as data formatting.
- 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 for FreeRTOS task management, providing APIs for task operations such as creation, deletion, suspension, and delay, enabling the program to implement multi-task concurrent execution.
- esp_log.h: This is the header file for the log printing interface of ESP-IDF, providing log output at different levels (INFO, ERROR, etc.), enabling developers to debug and track program running status.
- esp_private/esp_clk.h: The private interface for ESP32 clock control (such as clock frequency configuration), ensuring stable system timing;

C main.c   C main.h 9+ ×

main > include > C main.h > ...

```c
#ifndef _MAIN_H_
#define _MAIN_H_

/*———————————————————————————————Header file declaration———————————————————————————————*/
#include <stdio.h>
#include "string.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "esp_private/esp_clk.h"
#include "esp_ldo_regulator.h"   // Include LDO (Low Dropout Regulator) API
#include "bsp_i2c.h"
#include "bsp_illuminate.h"
#include "bsp_dht20.h"
#include "bsp_stc8h1kxx.h"
/*——————————————————————————————Header file declaration end——————————————————————————————*/
```

- 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.

```
C main.c  2 ×    C main.h 9+
main > C main.c > ...

/*———————————————————————————————Header file declaration———————————————————————————————*/
#include "main.h"
/*——————————————————————————————Header file declaration end——————————————————————————————*/

/*————————————————————————————————Variable declaration————————————————————————————————*/

TaskHandle_t read_dht20;
static lv_obj_t *dht20_data = NULL;
/*———————————————————————————————Variable declaration end———————————————————————————————*/
```

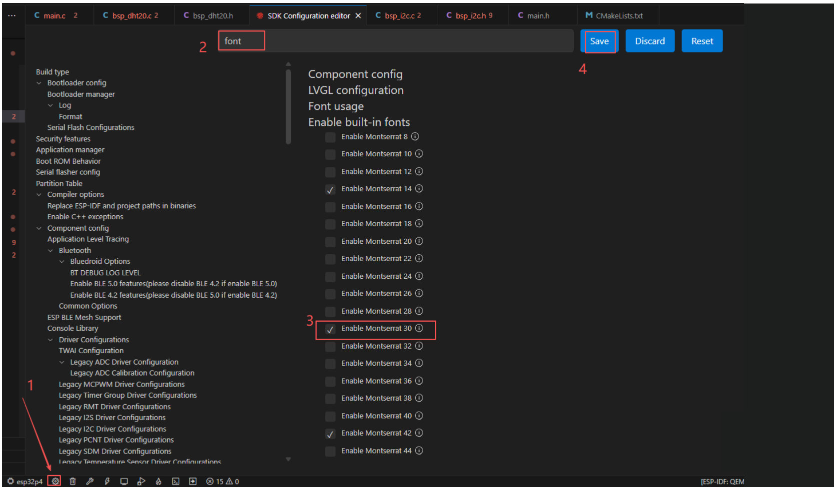
dht20_display():

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 dht20_display()
{
    if (lvgl_port_lock(0))
    {
        dht20_data = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t label_style;
        lv_style_init(&label_style);                                          /*Initialize a style*/
        lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);                     /*Set the style LVGL background color*/
        lv_obj_add_style(dht20_data, &label_style, LV_PART_MAIN);             /*Add a style to an object*/
        lv_obj_set_style_text_color(dht20_data, LV_COLOR_WHITE, LV_PART_MAIN); /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(dht20_data, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_center(dht20_data);                                            /*Align an object to the center on its parent*/
        lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLACK, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN);    /*Set the screen's LVGL background transparency*/
        lv_label_set_text(dht20_data, "Temperature = 0.0 C  Humidity = 0.0 %%"); /*Set a new text for a label*/
        lvgl_port_unlock();
    }
}
```

If you want to use the 30-point font of LVGL, you need to enter the SDK configuration and activate the 30-point font.



void update_dht20_value(float temperature, float humidity):

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

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

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

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_fail(const char *name, esp_err_t err)
{
    static bool state = false;
    while (1)
    {
        if (!state)
        {
            MAIN_ERROR("%s init  [ %s ]", name, esp_err_to_name(err));
            state = true;
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

void Init(void):

The "Init()" function is mainly responsible for the initialization of the entire system. It sequentially completes the initialization of the I2C bus, the I2C interface of the STC8H1KXX microcontroller, the initialization of the DHT20 temperature and humidity sensor, and the initialization of the LCD display screen.

At each step, it checks whether the return value is "ESP_OK". If there is a failure, it calls "init_fail()" to enter the error handling loop. After all peripheral devices are successfully initialized, the function will set the LCD backlight brightness to 100 (i.e., the maximum brightness), and output corresponding log information to ensure that the hardware devices are ready to enter the normal working state.

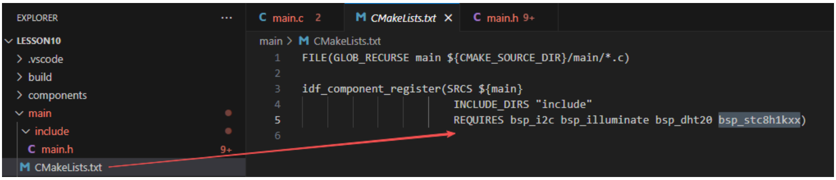
```
void Init(void)
{
    static esp_err_t err = ESP_OK;

    err = i2c_init(); /*I2C Initialization*/
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);

    err = stc8_i2c_init();
    if (err != ESP_OK)
        init_fail("stc8h1kxx", err);
    MAIN_INFO("I2C and stc8 init success");  // Print success log

    err = dht20_begin(); /*DHT20 Initialization*/
    if (err != ESP_OK)
        init_fail("dht20", err);

    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
    MAIN_INFO("LCD init success");  // Print success log

    // Turn on LCD backlight (brightness set to 100 = maximum)
    err = set_lcd_blight(100);  // Enable backlight
    if (err != ESP_OK) {  // Check error
        init_fail("LCD Backlight", err);  // Handle failure
    }
    MAIN_INFO("LCD backlight opened (brightness: 100)");  // Print success log
}
```

- Then there is the main function 'app_main'.
- The "app_main()" function is the entry point of the program, responsible for the overall startup process of the system. It first outputs the log "DHT20 test start", then calls "Init()" to complete the initialization of various hardware modules, including I2C, sensors, and display screens.

- Next, it creates and initializes the LVGL interface for displaying temperature and humidity data through "dht20_display()", and then uses "xTaskCreate()" to start an independent "dht20_read_task" task to periodically read the data from the DHT20 sensor and update the screen display in real time. Finally, it outputs the log "Start the test", indicating that the system has officially entered the working state of data collection and display.

```c
void app_main(void)
{
    MAIN_INFO("----------DHT20 test start----------");
    Init();                                                    /*Set the screen backlight to maximum brightness*/
    dht20_display();                                                            /*Set the screen's LVGL default display page*/
    xTaskCreate(dht20_read_task, "read_dht20", 4096, NULL, configMAX_PRIORITIES - 5, &read_dht20); /*Create a DHT20 data display refresh thread*/
    MAIN_INFO("----------Start the test----------");
}
/*————————————————————————Functional function end————————————————————————*/
```

- Finally, let's take a look at the "CMakeLists.txt" file in the main directory.
- The role of this CMake configuration is as follows:
    - It collects all the .c source files in the main/ directory as the source files of the component.
    - It registers the main component with the ESP-IDF build system and declares that it depends on the custom components: bsp_dht20, bsp_illuminate, bsp_i2c and bsp_stc8h1kxx.
- In this way, during the build process, ESP-IDF knows to build these four 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.

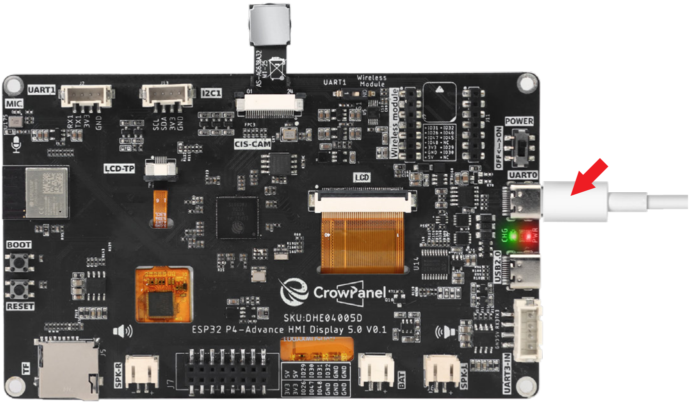
Complete Code

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

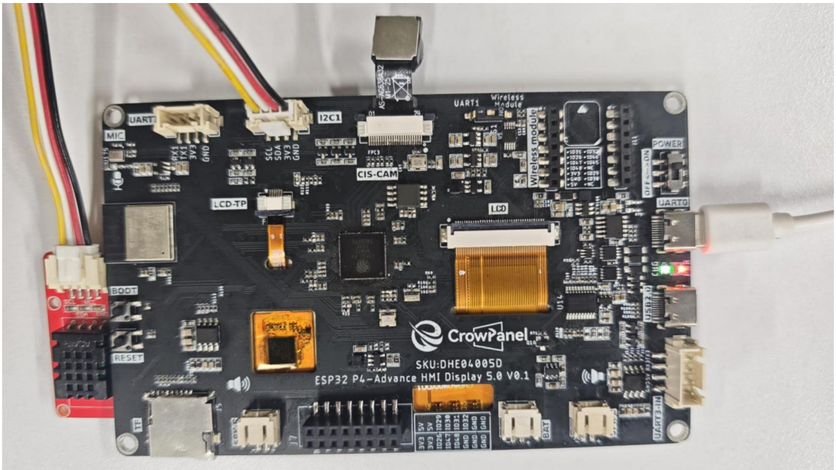
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson10-Temperature_and_Humidity

Programming Steps

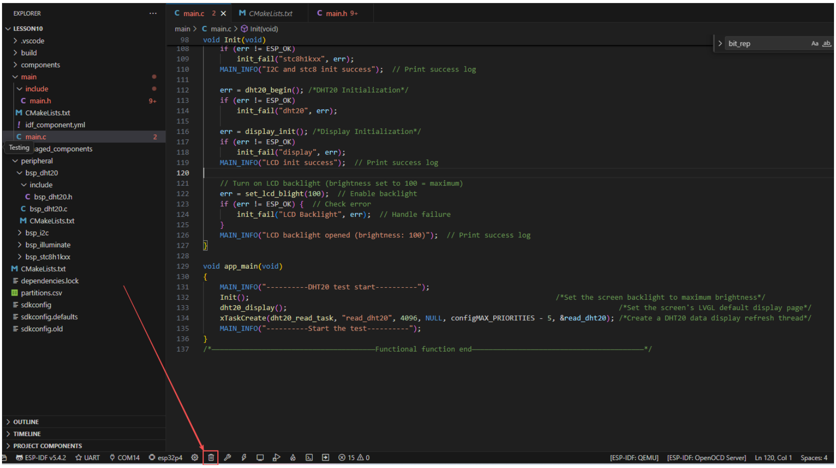
- Now the code is ready. Next, we need to flash it to the ESP32-P4 to see the actual effect.
- First, connect the Advance-P4 device to our computer via a USB cable.



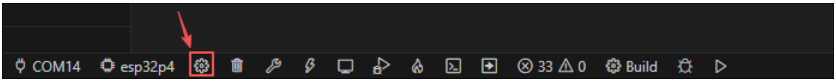
- After connecting the Advance-P4 board, connect the temperature and humidity sensor to the I2C interface.



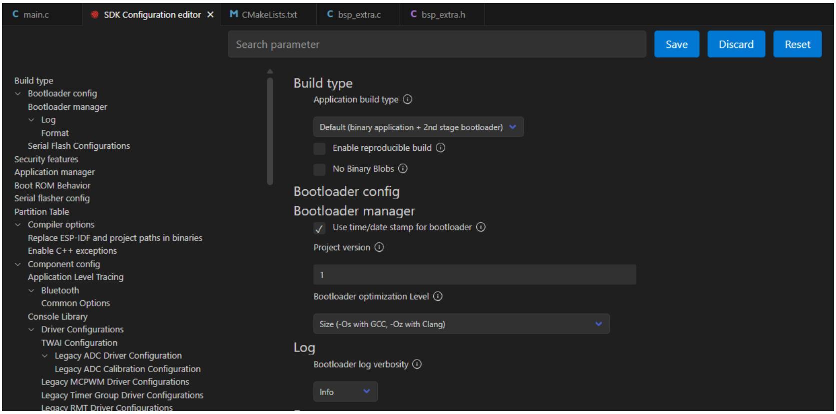
- Before starting the preparation for flashing, first delete all files generated during compilation to restore the project to its initial "unbuilt" state. (This ensures that subsequent compilations are not affected by your previous build artifacts.)



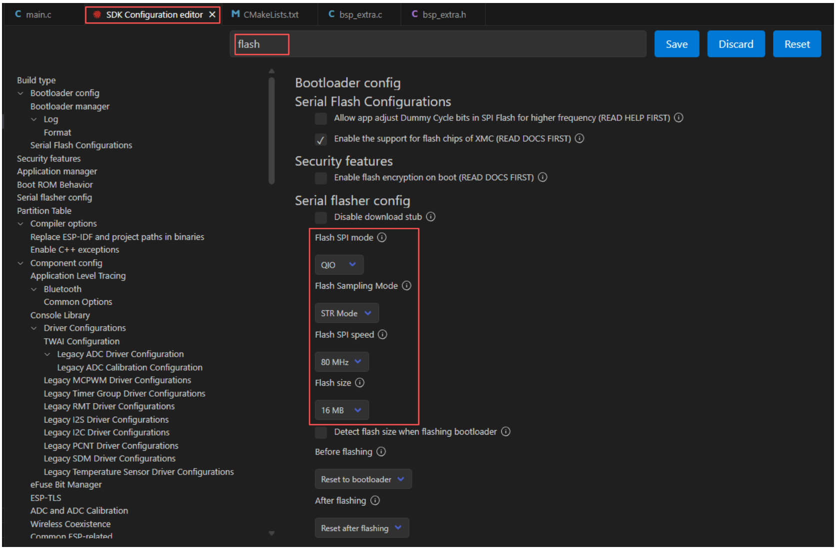
- First, follow the steps in the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon shown in the figure below.



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

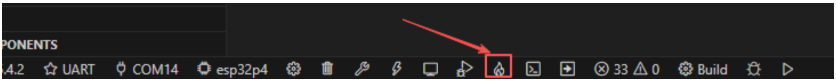


- Then, search for flash in the search box. (Make sure your flash configuration is consistent with mine.)



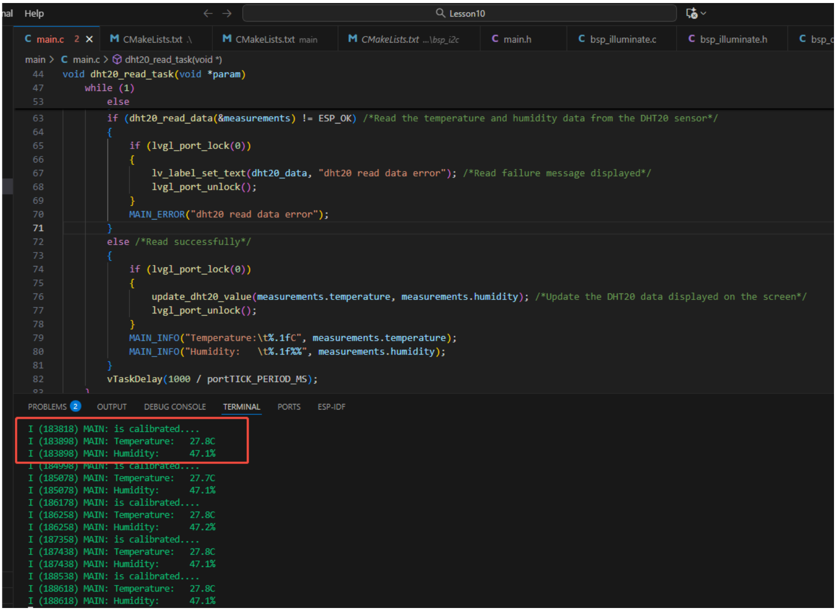
- After the configuration is completed, remember to save your settings.

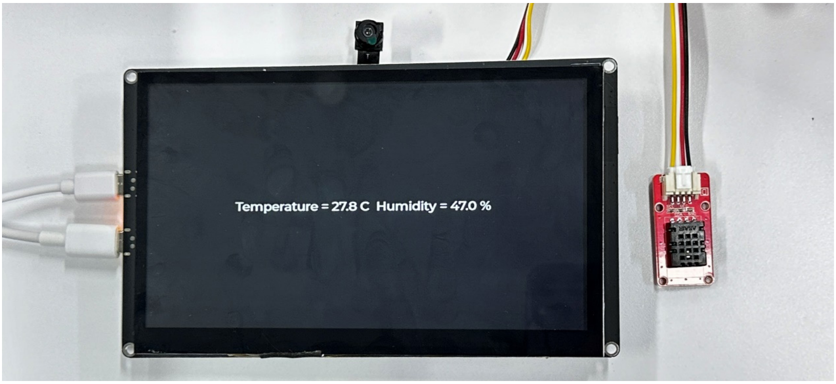
- Next, we will compile and flash the code (detailed in the first lesson). Here, we will also introduce a very convenient feature: you can execute compilation, upload, and open the monitor in one go with a single button (this works on the premise that the entire code is error-free).



- 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.


Help
Lesson10
main.c 2
CMakeLists.txt .\
CMakeLists.txt main
CMakeLists.txt ...\bsp_i2c
main.h
bsp_illuminate.c
bsp_illuminate.h
main > main.c > dht20_read_task(void *)
void dht20_read_task(void *param)
while (1)
else
if (dht20_read_data(&measurements) != ESP_OK) /*Read the temperature and humidity data from the DHT20 sensor*/
{
if (lvgl_port_lock(0))
{
lv_label_set_text(dht20_data, "dht20 read data error"); /*Read failure message displayed*/
lvgl_port_unlock();
}
MAIN_ERROR("dht20 read data error");
}
else /*Read successfully*/
{
if (lvgl_port_lock(0))
{
update_dht20_value(measurements.temperature, measurements.humidity); /*Update the DHT20 data displayed on the screen*/
lvgl_port_unlock();
}
MAIN_INFO("Temperature:\t%.1fC", measurements.temperature);
MAIN_INFO("Humidity: \t%.1f%%", measurements.humidity);
}
vTaskDelay(1000 / portTICK_PERIOD_MS);
PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
ESP-IDF
I (183818) MAIN: is calibrated....
I (183898) MAIN: Temperature: 27.8C
I (183898) MAIN: Humidity: 47.1%
I (185078) MAIN: Temperature: 27.7C
I (185078) MAIN: Humidity: 47.1%
I (186178) MAIN: is calibrated....
I (186258) MAIN: Temperature: 27.8C
I (186258) MAIN: Humidity: 47.2%
I (187358) MAIN: is calibrated....
I (187438) MAIN: Temperature: 27.8C
I (187438) MAIN: Humidity: 47.1%
I (188538) MAIN: is calibrated....
I (188618) MAIN: Temperature: 27.8C
I (188618) MAIN: Humidity: 47.1%


Temperature = 27.8 C Humidity = 47.0 %

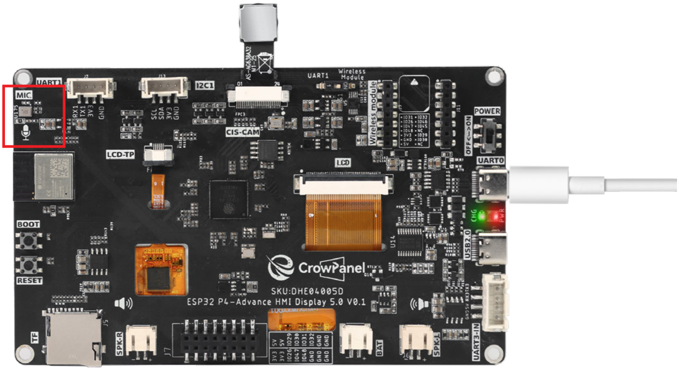
Lesson 11
Playback After Recording

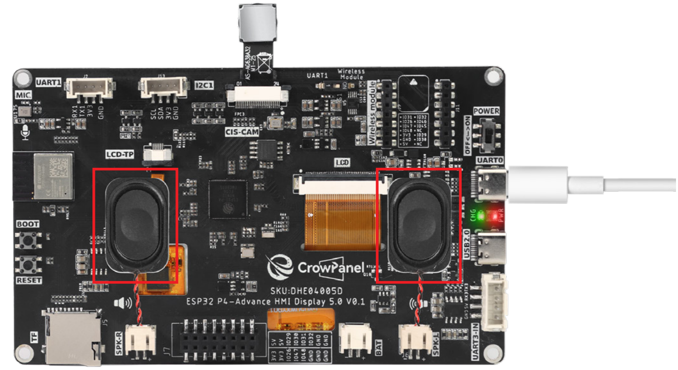
Introduction

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.

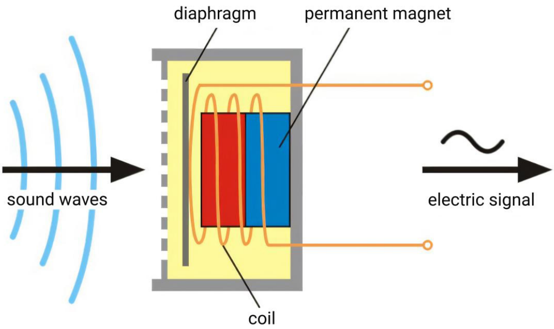
Hardware Used in This Lesson

Microphone and Speaker on the Advance-P4





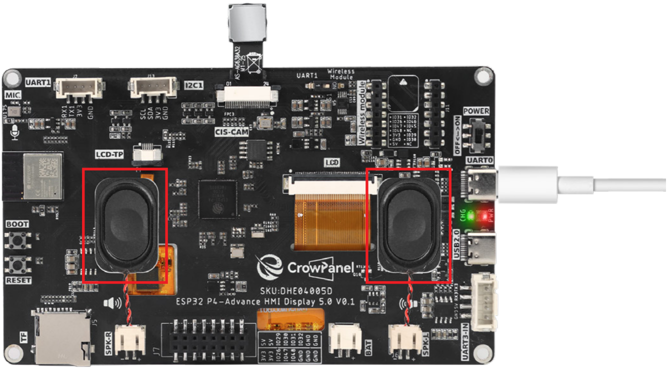
Microphone and Speaker Schematic Diagrams



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.



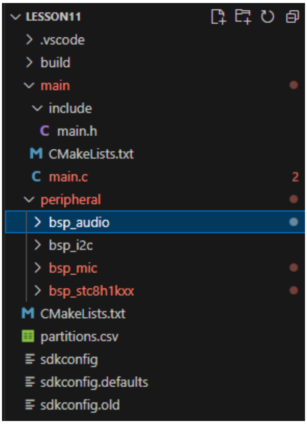
The 5-second recorded audio is now playing.

Key Explanations

- The key focus of this lesson is the use of two components: bsp_mic and bsp_audio.
- However, here we need to use the bsp_stc8h1kxx that was explained in the previous course. Here, we are opening the audio output pins because the expansion chip STC8H1K17 not only controls the screen backlight but also can control the high and low levels of the audio output pins, that is, whether to turn on the speaker sound.
- Also, since the expansion chip STC8H1K17 is controlled by the I2C of the main control ESP32-P4, we also need to initialize the I2C so that we can use the expansion chip to control.
- 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-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-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:

Declare the functions, macros, and variables implemented in "bsp_audio.c" for use by external programs, allowing other .c files to call this module simply by adding #include "bsp_audio.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, all the libraries we need to use are included in the "bsp_audio.h" file for unified management.

```
/*-----------------------------Header file declaration-----------------------------*/
#include <string.h>            // Include string manipulation functions
#include <stdint.h>            // Include standard integer type definitions
#include "freertos/FreeRTOS.h" // Include FreeRTOS core definitions
#include "freertos/task.h"     // Include FreeRTOS task management functions
#include "esp_log.h"           // Include ESP-IDF logging utilities
#include "esp_err.h"           // Include ESP-IDF error handling definitions
#include "driver/gpio.h"       // Include GPIO driver functions
#include "driver/i2s_std.h"    // Include standard I2S driver API
#include "bsp_stc8h1kxx.h"
/*-----------------------------Header file declaration end-----------------------------*/
```

- Then, we declare the variables we need to use, as well as the functions—whose specific implementations are in "bsp_audio.c".
- Having these declarations unified in "bsp_audio.h" is for the convenience of calling and management. (We will learn about their roles when they are used in "bsp_audio.c".)

```
/*-----------------------------Variable declaration-----------------------------*/
#define AUDIO_TAG "AUDIO"  // Define the logging tag for audio module
#define AUDIO_INFO(fmt, ...)    ESP_LOGI(AUDIO_TAG, fmt, ##__VA_ARGS__)  // Info-level log macro for audio module
#define AUDIO_DEBUG(fmt, ...)   ESP_LOGD(AUDIO_TAG, fmt, ##__VA_ARGS__)  // Debug-level log macro for audio module
#define AUDIO_ERROR(fmt, ...)   ESP_LOGE(AUDIO_TAG, fmt, ##__VA_ARGS__)  // Error-level log macro for audio module

#define AUDIO_GPIO_LRCLK     21   // GPIO pin number for LRCLK (Left-Right Clock)
#define AUDIO_GPIO_BCLK      22   // GPIO pin number for BCLK (Bit Clock)
#define AUDIO_GPIO_SDATA     23   // GPIO pin number for SDATA (Serial Data)
#define AUDIO_GPIO_CTRL      -1   // GPIO pin number for audio amplifier control

esp_err_t audio_init();            // Function prototype for audio initialization
esp_err_t audio_ctrl_init();       // Function prototype for audio amplifier control initialization
esp_err_t set_Audio_ctrl(bool state);  // Function prototype to enable or disable the audio amplifier
i2s_chan_handle_t get_audio_handle();  // Function prototype to get the I2S channel handle

/*-----------------------------Variable declaration end-----------------------------*/
#endif  // End of include guard
```

- Let's take a look at the specific function of each function in "bsp_audio.c".
- "bsp_audio.h": This project's custom audio module header file defines macros, GPIO pins, and function declarations.

```
/*-----------------------------Header file declaration-----------------------------*/
#include "bsp_audio.h"
/*-----------------------------Header file declaration end-----------------------------*/
```

- 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.

```
/*------------------------------------------Variable declaration------------------------------------------*/
i2s_chan_handle_t tx_chan;
/*----------------------------------------Variable declaration end----------------------------------------*/
```

- 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.

```
esp_err_t audio_init()
{
    esp_err_t err = ESP_OK;

    i2s_chan_config_t chan_cfg = {
        .id = I2S_NUM_1,
        .role = I2S_ROLE_MASTER,
        .dma_desc_num = 6,
        .dma_frame_num = 256,
        .auto_clear = true,
        .intr_priority = 0,
    };
    err = i2s_new_channel(&chan_cfg, &tx_chan, NULL);
    if (err != ESP_OK)
        return err;
    i2s_std_config_t std_cfg = {
        .clk_cfg = {
            .sample_rate_hz = 16000,
            .clk_src = I2S_CLK_SRC_DEFAULT,
            .mclk_multiple = I2S_MCLK_MULTIPLE_256,
        },
        .slot_cfg = {
            .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT,
            .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO,
            .slot_mode = I2S_SLOT_MODE_STEREO,
            .slot_mask = I2S_STD_SLOT_BOTH,
            .ws_width = I2S_DATA_BIT_WIDTH_16BIT,
            .ws_pol = false,
            .bit_shift = true,
            .left_align = true,
            .big_endian = false,
            .bit_order_lsb = false,
        },
        .gpio_cfg = {
            .mclk = I2S_GPIO_UNUSED,
            .bclk = AUDIO_GPIO_BCLK,
            .ws = AUDIO_GPIO_LRCLK,
            .dout = AUDIO_GPIO_SDATA,
            .din = I2S_GPIO_UNUSED,
```

set_Audio_ctrl:

Function: Controls the on/off state of the audio module.

Details:

The function receives a bool type parameter named state to specify the target state.

Internally, the pin STC8_GPIO_OUT_AUDIO_SD is operated by the stc8_gpio_set_level function:

When state is true, the pin is set to a low level (the level corresponding to !state, which is false, is usually the low level indicating audio output is enabled).

When state is false, the pin is set to a high level (possibly indicating the audio output is disabled).

It is speculated that STC8_GPIO_OUT_AUDIO_SD is the "shutdown" control pin of the audio module, and it controls the enable state of the module through high and low levels.

```
esp_err_t set_Audio_ctrl(bool state)
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, !state);
    return ESP_OK;
}
```

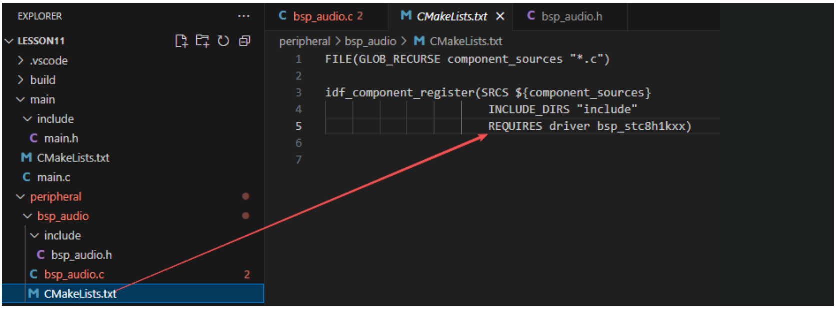
- **get_audio_handle:** This function is used to obtain and return the handle of the current I2S audio output channel, allowing other modules to use this handle for audio data transmission or playback operations.

```
i2s_chan_handle_t get_audio_handle()
{
    return tx_chan;
}
```

- 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).

```c
#ifndef _BSP_AUDIO_H_  // Prevent multiple inclusion of this header file
#define _BSP_AUDIO_H_

/*----------------------------------Header file declaration----------------------------------*/
#include <string.h>            // Include string manipulation functions
#include <stdint.h>            // Include standard integer type definitions
#include "freertos/FreeRTOS.h" // Include FreeRTOS core definitions
#include "freertos/task.h"     // Include FreeRTOS task management functions
#include "esp_log.h"           // Include ESP-IDF logging utilities
#include "esp_err.h"           // Include ESP-IDF error handling definitions
#include "driver/gpio.h"       // Include GPIO driver functions
#include "driver/i2s_std.h"    // Include standard I2S driver API
#include "bsp_stc8h1kxx.h"
/*----------------------------------Header file declaration end----------------------------------*/
```

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".

```c
#ifndef _BSP_MIC_H_
#define _BSP_MIC_H_

/*----------------------------------Header file declaration----------------------------------*/
#include <string.h>
#include <stdint.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/gpio.h"
#include "driver/i2s_pdm.h"
#include "bsp_audio.h"
/*----------------------------------Header file declaration end----------------------------------*/

/*----------------------------------Variable declaration----------------------------------*/
#define MIC_TAG "MIC"
#define MIC_INFO(fmt, ...) ESP_LOGI(MIC_TAG, fmt, ##__VA_ARGS__)
#define MIC_DEBUG(fmt, ...) ESP_LOGD(MIC_TAG, fmt, ##__VA_ARGS__)
#define MIC_ERROR(fmt, ...) ESP_LOGE(MIC_TAG, fmt, ##__VA_ARGS__)

#define MIC_GPIO_CLK        (24)
#define MIC_GPIO_SDIN2      (25)

#define MIC_SAMPLE_RATE     16000
#define BYTE_RATE           ((16000 * (16 / 8)) * 1)

esp_err_t mic_init();
esp_err_t mic_read_to_audio(size_t rec_seconds);

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

- 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.

```
peripheral > bsp_mic > include > C bsp_mic.h > MIC_GPIO_CLK
#ifndef _BSP_MIC_H_
#include <string.h>
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/gpio.h"
#include "driver/i2s_pdm.h"
#include "bsp_audio.h"
/*------------------------------Header file declaration end------------------------------*/

/*------------------------------Variable declaration------------------------------*/
#define MIC_TAG "MIC"
#define MIC_INFO(fmt, ...) ESP_LOGI(MIC_TAG, fmt, ##__VA_ARGS__)
#define MIC_DEBUG(fmt, ...) ESP_LOGD(MIC_TAG, fmt, ##__VA_ARGS__)
#define MIC_ERROR(fmt, ...) ESP_LOGE(MIC_TAG, fmt, ##__VA_ARGS__)

#define MIC_GPIO_CLK        (24)
#define MIC_GPIO_SDIN2      (25)

#define MIC_SAMPLE_RATE     16000
#define BYTE_RATE           ((16000 * (16 / 8)) * 1)

esp_err_t mic_init();
esp_err_t mic_read_to_audio(size_t rec_seconds);
```

- 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.
- It mainly includes two functions: microphone initialization (mic_init) and recording to audio playback (mic_read_to_audio).
- "bsp_mic.h": The header file for the microphone module, which defines macros, pins, and function declarations.
- rx_chan: A global variable representing the I2S receive channel handle, which will be used for all subsequent operations involving reading audio data from the microphone.

```
peripheral > bsp_mic > C bsp_mic.c > mic_init()
/*------------------------------Header file declaration------------------------------*/
#include "bsp_mic.h"
/*------------------------------Header file declaration end------------------------------*/

/*------------------------------Variable declaration------------------------------*/

i2s_chan_handle_t rx_chan;
/*------------------------------Variable declaration end------------------------------*/
```

- mic_init: This function is used to initialize the I2S receive channel (in PDM mode) for the microphone. It configures parameters such as the sampling rate, DMA buffer, GPIO pins, high-pass filter, and mono audio data format, and enables the channel. This allows the system to collect audio signals from the digital microphone.

```
peripheral > bsp_mic > C bsp_mic.c > mic_init()
esp_err_t mic_init()
{
    esp_err_t err = ESP_OK;

    i2s_chan_config_t rx_chan_cfg = {
        .id = I2S_NUM_0,
        .role = I2S_ROLE_MASTER,
        .dma_desc_num = 6,
        .dma_frame_num = 256,
        .auto_clear_after_cb = true,
        .auto_clear_before_cb = true,
        .allow_pd = false,
        .intr_priority = 0,
    };
    err = i2s_new_channel(&rx_chan_cfg, NULL, &rx_chan);
    if (err != ESP_OK)
        return err;
    i2s_pdm_rx_config_t pdm_rx_cfg = {
        .clk_cfg = {
            .sample_rate_hz = MIC_SAMPLE_RATE,
            .clk_src = I2S_CLK_SRC_DEFAULT,
            .mclk_multiple = I2S_MCLK_MULTIPLE_256,
            .dn_sample_mode = I2S_PDM_DSR_8S,
            .bclk_div = 8,
        },
        /* The data bit-width of PDM mode is fixed to 16 */
        .slot_cfg = {
            .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT,
            .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO,
            .slot_mode = I2S_SLOT_MODE_MONO,
            .slot_mask = I2S_PDM_SLOT_LEFT,
            .hp_en = true,
            .hp_cut_off_freq_hz = 35.5,
            .amplify_num = 1,
        },
        .gpio_cfg = {
            .clk = MIC_GPIO_CLK,
            .din = MIC_GPIO_SDIN2,
            .invert_flags = {
                .clk_inv = false,
```

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.)

```c
esp_err_t mic_read_to_audio(size_t rec_seconds)
{
    esp_err_t err = ESP_OK;
    size_t bytes_read = 0;
    size_t bytes_write = 0;
    if (rec_seconds > 60)
    {
        MIC_INFO("Exceeding the maximum recording duration");
        return ESP_FAIL;
    }
    size_t rec_size = rec_seconds * BYTE_RATE;
    i2s_chan_handle_t write_handle = get_audio_handle();
    int16_t *read_buf = heap_caps_malloc(rec_size, MALLOC_CAP_SPIRAM);
    if (NULL == read_buf) {
        MIC_INFO("mic read_buf fail to apply");
        return ESP_FAIL;
    }
    memset(read_buf, 0, rec_size);
    int16_t *write_buf = heap_caps_malloc(rec_size * 2, MALLOC_CAP_SPIRAM);
    if (NULL == write_buf) {
        MIC_INFO("mic write_buf fail to apply");
        return ESP_FAIL;
    }
    memset(write_buf, 0, rec_size * 2);
    MIC_INFO("Start Recording %d of audio data", rec_seconds);
    err = i2s_channel_read(rx_chan, read_buf, rec_size, &bytes_read, portMAX_DELAY);
    if (err != ESP_OK)
    {
        MIC_INFO("read mic data fail");
        return err;
    }
    if (bytes_read != rec_size)
    {
        MIC_INFO("read mic data num error");
        return err;
    }

    int32_t data;
```

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

.h 9 | CMakeLists.txt ...\bsp_audio | bsp_audio.c 1 | bsp_mic.c 2 | CMakeLists.txt main | CMakeLists.txt ...\bsp_mic | bsp_mic.h 9+

peripheral > bsp_mic > include > bsp_mic.h > ...

```c
#ifndef _BSP_MIC_H_
#define _BSP_MIC_H_

/*———————————————————————————Header file declaration———————————————————————————*/
#include <string.h>
#include <stdint.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/gpio.h"
#include "driver/i2s_pdm.h"
#include "bsp_audio.h"
/*———————————————————————————Header file declaration end———————————————————————————*/
```

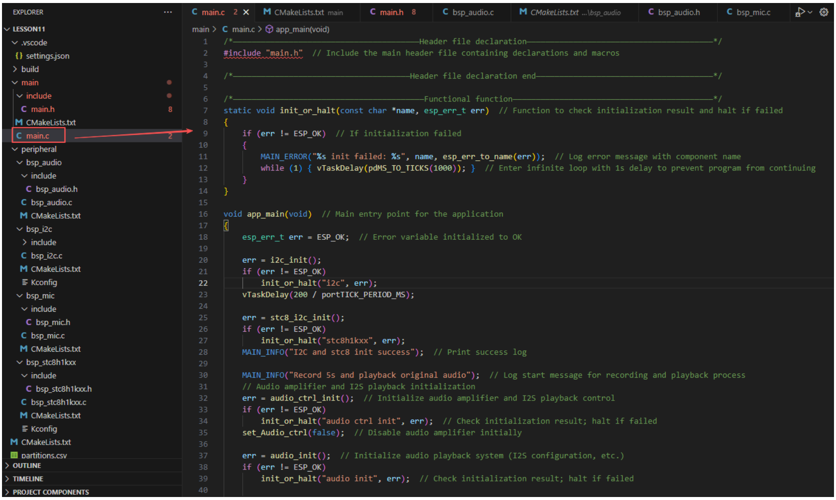
bsp_audio.c 2 | CMakeLists.txt | bsp_audio.h | bsp_mic.c 2 | bsp_mic.h 9+

peripheral > bsp_audio > bsp_audio.c > ...

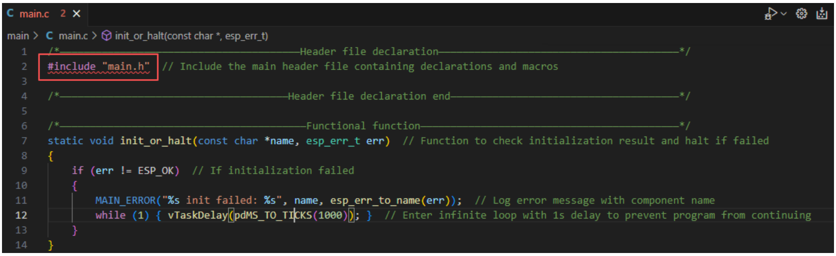
```c
esp_err_t audio_init()
    i2s_std_config_t std_cfg = {
    err = i2s_channel_init_std_mode(tx_chan, &std_cfg);
    if (err != ESP_OK)
        return err;
    err = i2s_channel_enable(tx_chan);
    if (err != ESP_OK)
        return err;
    return err;
}

esp_err_t audio_ctrl_init()
{
    return ESP_OK;
}

esp_err_t set_Audio_ctrl(bool state)
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, !state);
    return ESP_OK;
}
```

Main function

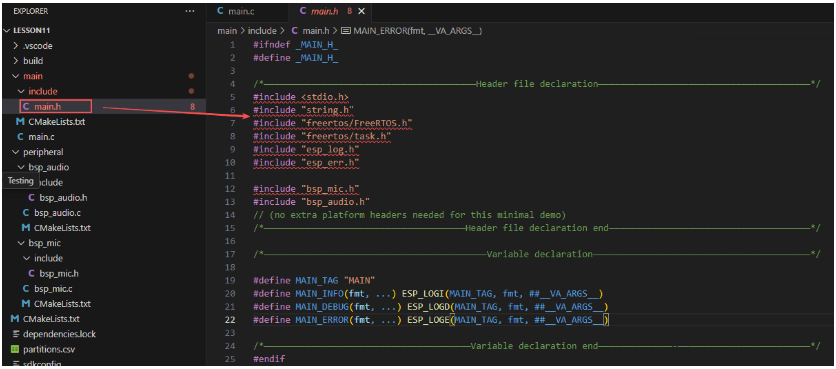
- The main folder is the core directory for program execution, which contains the main function executable file "main.c".
- Add the main folder to the "CMakeLists.txt" file of the build system.



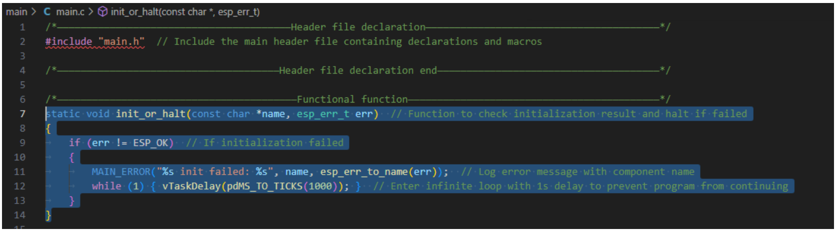
- This is the entry file of the entire application. In ESP-IDF, there is no int main(), and the program starts running from void app_main(void).
- Let's first explain "main.c".
- The app_main function is the main entry point of the entire application, responsible for coordinating the initialization of 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.



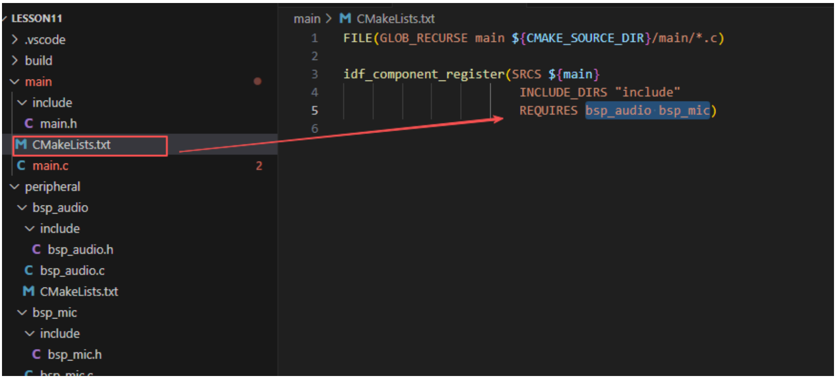
- Next is the main function "app_main".
- The "app_main" function serves as the primary entry point of the entire application, responsible for coordinating the initialization of the audio system and microphone, as well as audio recording and playback.
- It first initializes the audio power amplifier 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.

main.c 2 × | main.h 8 | bsp_audio.c | CMakeLists.txt | bsp_audio.h | bsp_mic.c | bsp_mic.h

main > main.c > app_main(void)

```
void app_main(void)  // Main entry point for the application
    esp_err_t err = ESP_OK;  // Error variable initialized to OK

    err = i2c_init();
    if (err != ESP_OK)
        init_or_halt("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);

    err = stc8_i2c_init();
    if (err != ESP_OK)
        init_or_halt("stc8h1kxx", err);
    MAIN_INFO("I2C and stc8 init success");  // Print success log

    MAIN_INFO("Record 5s and playback original audio");  // Log start message for recording and playback process
    // Audio amplifier and I2S playback initialization
    err = audio_ctrl_init();  // Initialize audio amplifier and I2S playback control
    if (err != ESP_OK)
        init_or_halt("audio ctrl init", err);  // Check initialization result; halt if failed
    set_Audio_ctrl(false);  // Disable audio amplifier initially

    err = audio_init();  // Initialize audio playback system (I2S configuration, etc.)
    if (err != ESP_OK)
        init_or_halt("audio init", err);  // Check initialization result; halt if failed

    // Microphone initialization
    err = mic_init();  // Initialize microphone input
    if (err != ESP_OK)
        init_or_halt("mic", err);  // Check microphone initialization result; halt if failed

    // Record for 5 seconds and playback
    MAIN_INFO("Start 5s recording...");  // Log recording start message
    err = mic_read_to_audio(5);  // Record 5 seconds of audio and play it back
    if (err != ESP_OK)  // If recording or playback failed
        MAIN_ERROR("record/playback error: %s", esp_err_to_name(err));  // Log error message
    else
        MAIN_INFO("Playback done");  // Log success message when playback finishes

    // Keep task alive
    while (1) { vTaskDelay(pdMS_TO_TICKS(1000)); }  // Keep the task alive indefinitely with a 1s delay
}
```

- 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 drivers into the main function.

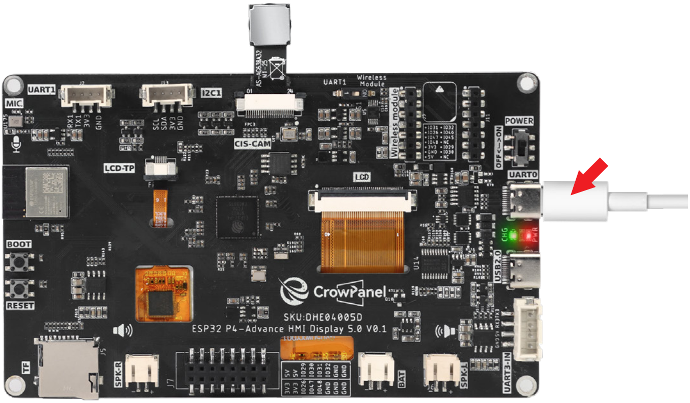
Complete Code

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

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson11-Playback_After_Recording

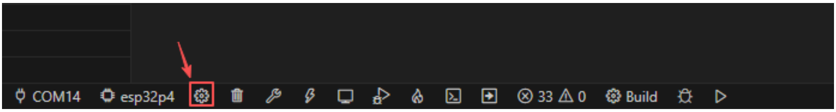
Programming Steps

- 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.

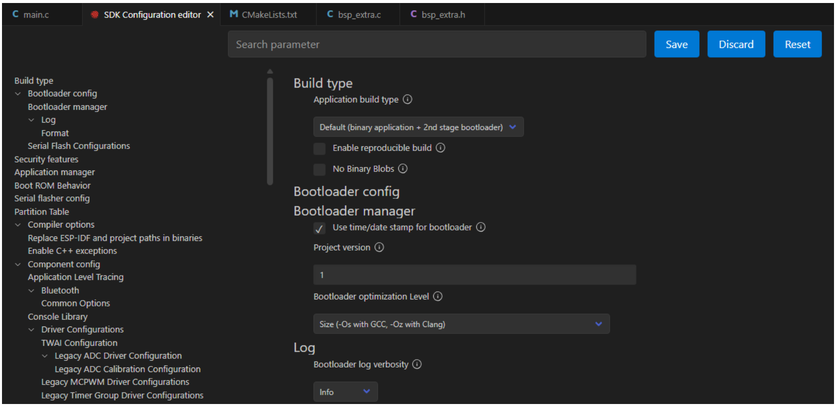


- Before starting the flashing preparation, delete all files generated during compilation to restore the project to its initial "unbuilt" state. (This ensures that subsequent compilations are not affected by your previous operations.)

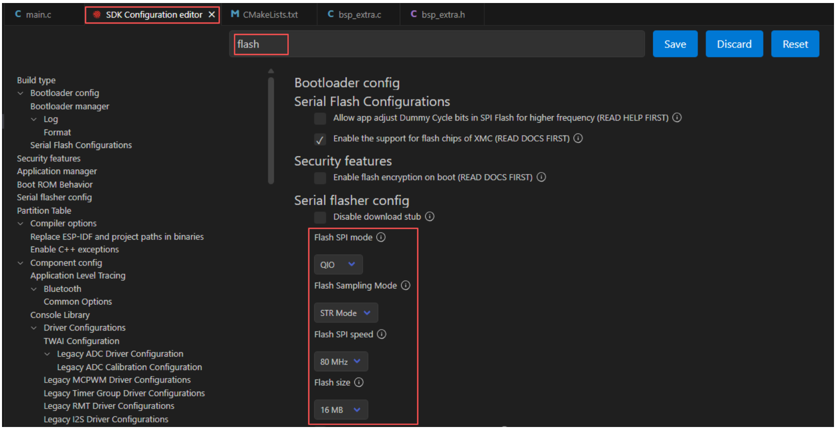
- Here, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip first.
- Next, we need to configure the SDK.
- Click the icon shown in the figure below.



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



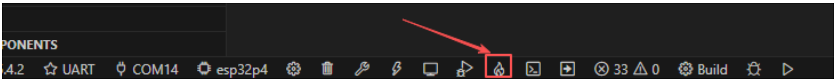
- Subsequently, search for "flash" in the search box. (Make sure your flash configuration is the same as mine.)



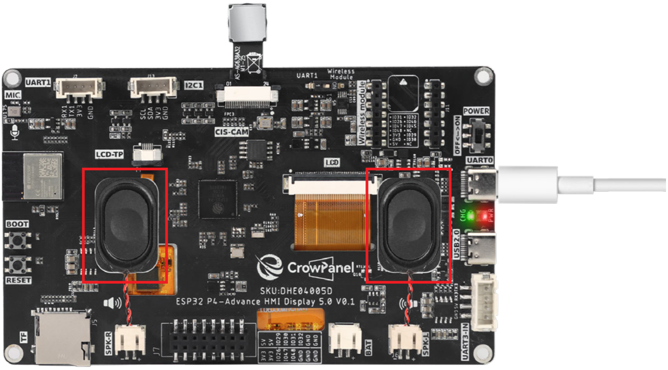
- After completing the configuration, remember to save your settings.

- 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.

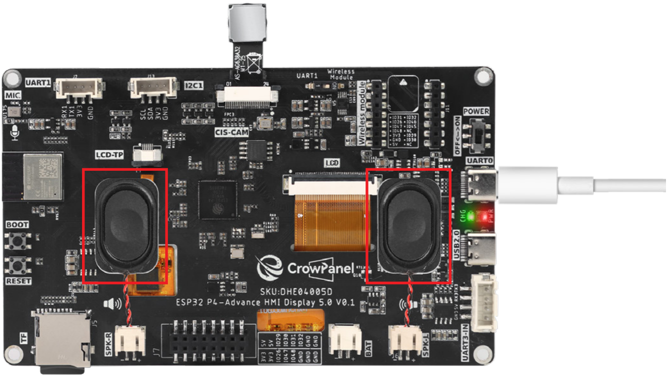
Lesson 12
Playing Local Music from SD Card

Introduction

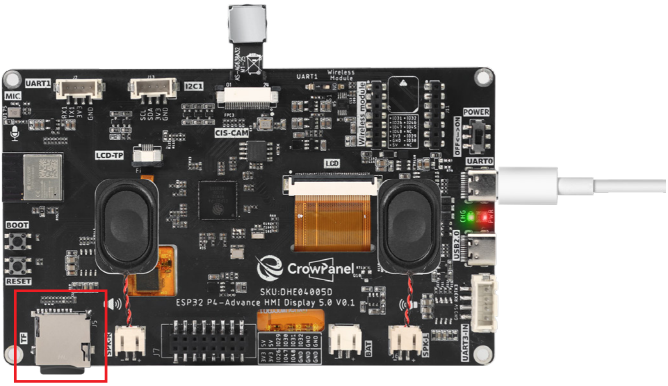
In this lesson, we will use the bsp_sd component and bsp_audio component (which were used in previous lessons) to play WAV audio files stored in the SD card.

Hardware Used in This Lesson

Speaker on the Advance-P4

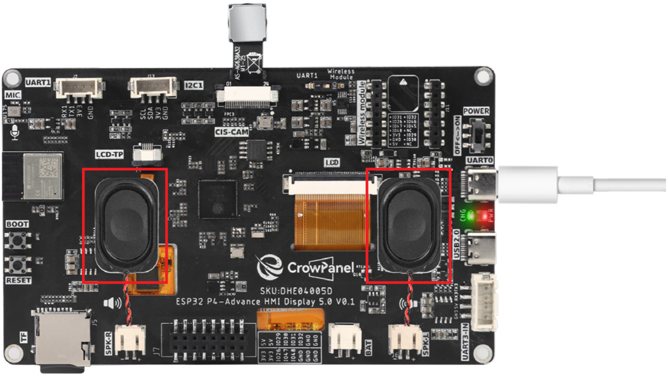


SD Card on the Advance-P4



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

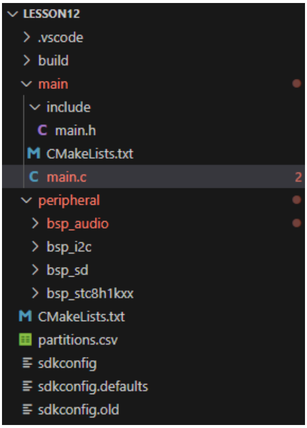
- The key focus of this lesson is the combined use of the two components: bsp_sd and bsp_audio.In fact, for the SD card component, we still use the same interfaces as in the previous component. These interfaces were explained in detail earlier, so they will not be covered again here.
- However, here we need to use the bsp_stc8h1kxx that was explained in the previous course. Here, we are opening the audio output pins because the expansion chip STC8H1K17 not only controls the screen backlight but also can control the high and low levels of the audio output pins, that is, whether to turn on the speaker sound.
- Also, since the expansion chip STC8H1K17 is controlled by the I2C of the main control ESP32-P4, we also need to initialize the I2C so that we can use the expansion chip to control.
- 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-5inch-ESP32-P4-HMI-AI-Display-800x480-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.

- The role of bsp_stc8h1kxx here is to control the shutdown of the power amplifier pins. It has been used many times in previous courses as well.

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.

```c
/*----------------------------Header file declaration----------------------------*/
#include <string.h>          // Include string manipulation functions
#include <stdint.h>          // Include standard integer type definitions
#include "freertos/FreeRTOS.h" // Include FreeRTOS core definitions
#include "freertos/task.h"   // Include FreeRTOS task management functions
#include "esp_log.h"         // Include ESP-IDF logging utilities
#include "esp_err.h"         // Include ESP-IDF error handling definitions
#include "driver/gpio.h"     // Include GPIO driver functions
#include "driver/i2s_std.h"  // Include standard I2S driver API
#include "bsp_stc8h1kxx.h"
/*----------------------------Header file declaration end----------------------------*/
```

- Then, we declare the variables we need to use as well as the functions, whose specific implementations are in "bsp_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".)

```c
/*----------------------------Variable declaration----------------------------*/
#define AUDIO_TAG "AUDIO"  // Define the logging tag for audio module
#define AUDIO_INFO(fmt, ...)    ESP_LOGI(AUDIO_TAG, fmt, ##__VA_ARGS__)  // Info-level log macro for audio module
#define AUDIO_DEBUG(fmt, ...)   ESP_LOGD(AUDIO_TAG, fmt, ##__VA_ARGS__)  // Debug-level log macro for audio module
#define AUDIO_ERROR(fmt, ...)   ESP_LOGE(AUDIO_TAG, fmt, ##__VA_ARGS__)  // Error-level log macro for audio module

#define AUDIO_GPIO_LRCLK    21  // GPIO pin number for LRCLK (Left-Right Clock)
#define AUDIO_GPIO_BCLK     22  // GPIO pin number for BCLK (Bit Clock)
#define AUDIO_GPIO_SDATA    23  // GPIO pin number for SDATA (Serial Data)
#define AUDIO_GPIO_CTRL     -1  // GPIO pin number for audio amplifier control

esp_err_t audio_init();             // Function prototype for audio initialization
esp_err_t audio_ctrl_init();        // Function prototype for audio amplifier control initialization
esp_err_t set_Audio_ctrl(bool state);  // Function prototype to enable or disable the audio amplifier
i2s_chan_handle_t get_audio_handle();  // Function prototype to get the I2S channel handle

esp_err_t Audio_play_wav_sd(const char *fp); // The speaker plays the wav file stored on the SD card

/*----------------------------Variable declaration end----------------------------*/
```

- 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.

```c
/*----------------------------Header file declaration----------------------------*/
#include "bsp_audio.h"
/*----------------------------Header file declaration end----------------------------*/
```

- 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.

```c
/*----------------------------Variable declaration----------------------------*/
i2s_chan_handle_t tx_chan;
/*----------------------------Variable declaration end----------------------------*/
```

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.

The main function of this function is to create and configure an I2S audio transmission channel, enabling ESP32-P4 to output audio in 16kHz, 16-bit, stereo format through specified GPIOs.

peripheral > bsp_audio > C bsp_audio.c > audio_init()

```c
esp_err_t audio_init()
{
    esp_err_t err = ESP_OK;
    i2s_chan_config_t chan_cfg = {
        .id = I2S_NUM_1,          /*Use I2S controller 1*/
        .role = I2S_ROLE_MASTER, /*I2S acts as master (generates clock signals)*/
        .dma_desc_num = 6,       /*Number of DMA descriptors for buffer management*/
        .dma_frame_num = 256,    /*Number of frames per DMA descriptor*/
        .auto_clear = true,      /*Automatically clear DMA buffer on underflow*/
        .intr_priority = 0,      /*Interrupt priority level*/
    }; /*I2S channel configuration for transmitter (audio output)*/
    err = i2s_new_channel(&chan_cfg, &tx_chan, NULL); /*Create new I2S channel (transmit channel only, no receive channel)*/
    if (err != ESP_OK)
        return err;
    i2s_std_config_t std_cfg = {
        .clk_cfg = {
            .sample_rate_hz = 16000,                 /*Audio sample rate: 16kHz*/
            .clk_src = I2S_CLK_SRC_DEFAULT,          /*Default clock source*/
            .mclk_multiple = I2S_MCLK_MULTIPLE_256, /*Master clock multiplier*/
        },                                           /* Clock configuration*/
        .slot_cfg = {
            .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT, /*16-bit audio samples*/
            .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO,  /*Auto-calculate slot width*/
            .slot_mode = I2S_SLOT_MODE_STEREO,          /*Stereo audio (2 channels)*/
            .slot_mask = I2S_STD_SLOT_BOTH,             /*Enable both left and right channels*/
            .ws_width = I2S_DATA_BIT_WIDTH_16BIT,       /*Word select signal width*/
            .ws_pol = false,                            /*Word select polarity (normal)*/
            .bit_shift = true,                          /*Enable bit shift in data frame*/
            .left_align = true,                         /*Left-aligned data in slot*/
            .big_endian = false,                        /*Little-endian byte order*/
            .bit_order_lsb = false,                     /*MSB first bit order*/
        },                                              /* Audio slot/data format configuration*/
        .gpio_cfg = {
            .mclk = I2S_GPIO_UNUSED,   /*Master clock not used*/
            .bclk = AUDIO_GPIO_BCLK,   /*Bit clock pin*/
            .ws = AUDIO_GPIO_LRCLK,    /*Word select (left/right clock) pin*/
            .dout = AUDIO_GPIO_SDATA, /*Serial data output pin*/
            .din = I2S_GPIO_UNUSED,    /*Data input not used (output only)*/
            .invert_flags = {
                .mclk_inv = false, /*Don't invert master clock*/
                .bclk_inv = false, /*Don't invert bit clock*/
                .ws_inv = false,   /*Don't invert word select*/
            },                     /*Signal inversion flags*/
        },                         /*GPIO pin configuration for I2S signals*/
    }; /*Standard I2S configuration for audio playback*/
    err = i2s_channel_init_std_mode(tx_chan, &std_cfg); /*Initialize I2S channel in standard mode for audio output*/
    if (err != ESP_OK)
        return err;
```

Therefore, any audio files you use later must meet this requirement (16kHz sampling rate, 16-bit bit depth, and stereo format, i.e., dual-channel).

```c
i2s_std_config_t std_cfg = {
    .clk_cfg = {
        .sample_rate_hz = 16000,                 /*Audio sample rate: 16kHz*/
        .clk_src = I2S_CLK_SRC_DEFAULT,          /*Default clock source*/
        .mclk_multiple = I2S_MCLK_MULTIPLE_256, /*Master clock multiplier*/
    },                                           /* Clock configuration*/
    .slot_cfg = {
        .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT, /*16-bit audio samples*/
        .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO,  /*Auto-calculate slot width*/
        .slot_mode = I2S_SLOT_MODE_STEREO,          /*Stereo audio (2 channels)*/
        .slot_mask = I2S_STD_SLOT_BOTH,             /*Enable both left and right channels*/
        .ws_width = I2S_DATA_BIT_WIDTH_16BIT,       /*Word select signal width*/
        .ws_pol = false,                            /*Word select polarity (normal)*/
        .bit_shift = true,                          /*Enable bit shift in data frame*/
        .left_align = true,                         /*Left-aligned data in slot*/
        .big_endian = false,                        /*Little-endian byte order*/
        .bit_order_lsb = false,                     /*MSB first bit order*/
    },                                              /* Audio slot/data format configuration*/
```

set_Audio_ctrl:

Function: Controls the on/off state of the audio module.

Details:

The function receives a bool type parameter named state to specify the target state.

Internally, the pin STC8_GPIO_OUT_AUDIO_SD is operated by the stc8_gpio_set_level function:

When state is true, the pin is set to a low level (the level corresponding to !state, which is false, is usually the low level indicating audio output is enabled).

When state is false, the pin is set to a high level (possibly indicating the audio output is disabled).

It is speculated that STC8_GPIO_OUT_AUDIO_SD is the "shutdown" control pin of the audio module, and it controls the enable state of the module through high and low levels.

```
esp_err_t set_Audio_ctrl(bool state)
{
    stc8_gpio_set_level(STC8_GPIO_OUT_AUDIO_SD, !state);
    return ESP_OK;
}
```

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.

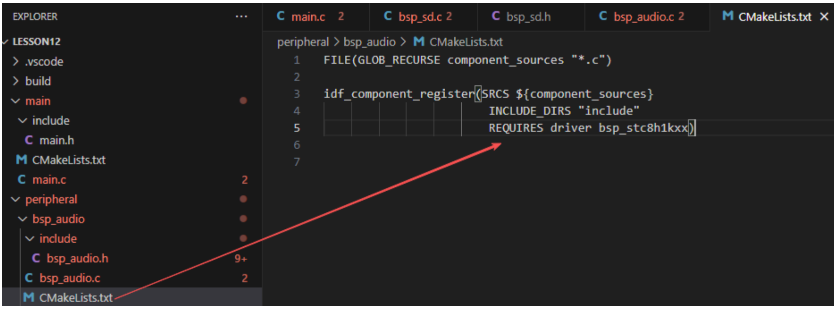
- if (fh == NULL) —— Returns an error if the file fails to open.

- 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 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

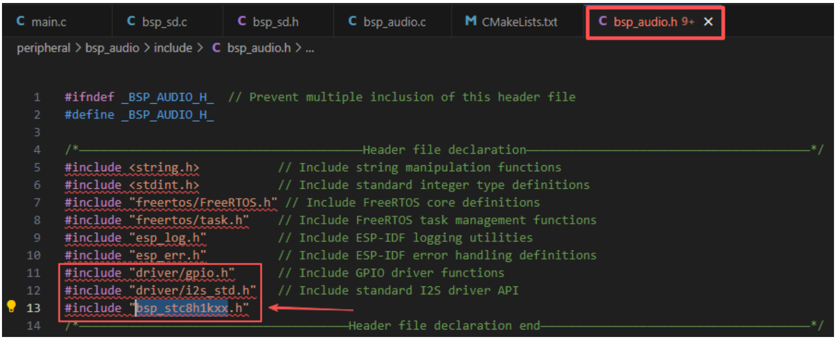
This function reads WAV files from the SD card, plays audio in chunks after validating the format, outputs to the audio hardware through I2S, handles volume and buffer management, and releases all resources after playback.

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_stc8h1kxx" 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

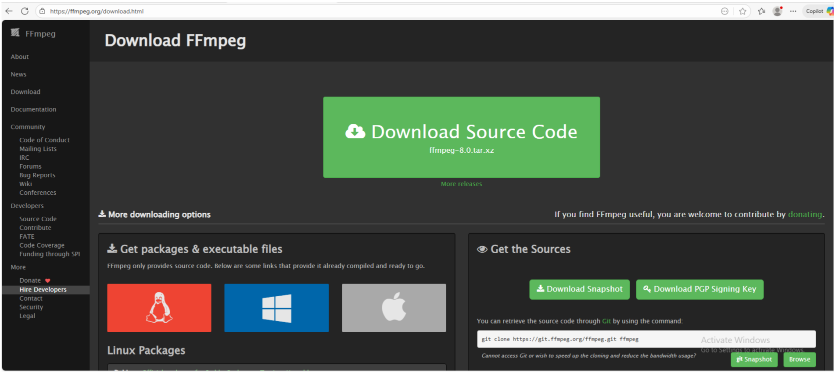
As mentioned above, if you want to play audio based on the code of this lesson, the audio must meet the requirement of being a WAV file with 16kHz sampling rate, 16-bit bit depth, and stereo format (i.e., dual-channel).

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).

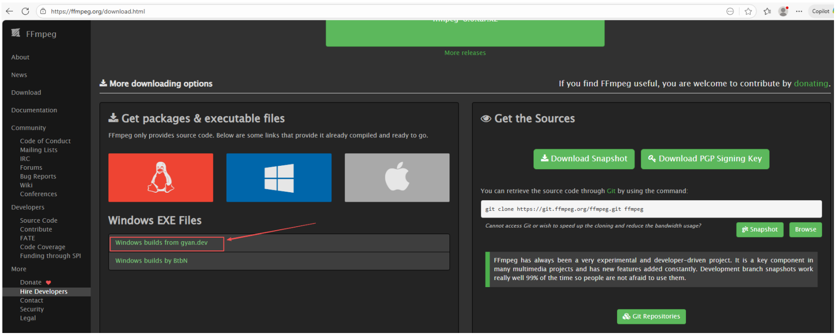
FFmpeg is an open-source toolkit for processing multimedia files such as video and audio. It supports conversion, cutting, and editing of almost all multimedia formats, making it an essential tool for developers and multimedia professionals.

Open the following link to download FFmpeg:

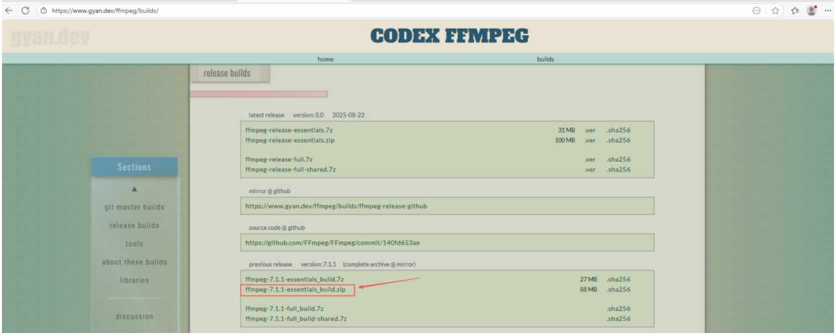
https://ffmpeg.org/download.html



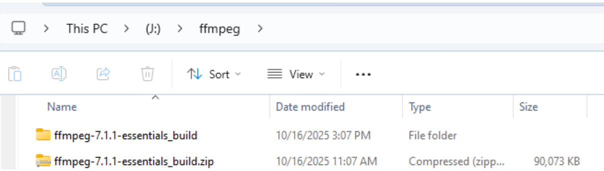
- Taking Windows as an Example: Select the installation package "Windows builds from gyan.dev".



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

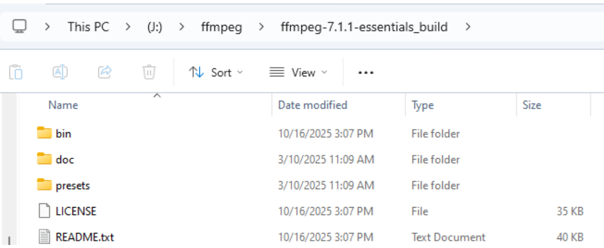


- Once the download is complete, extract the file to get the "FFmpeg" folder.



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.



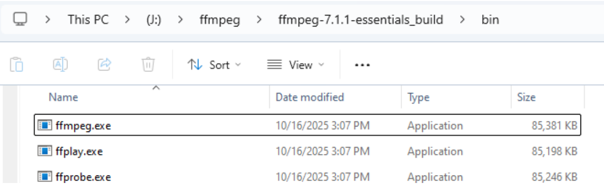
Directory Structure of the Extracted Folder

The extracted folder should contain the following directories:

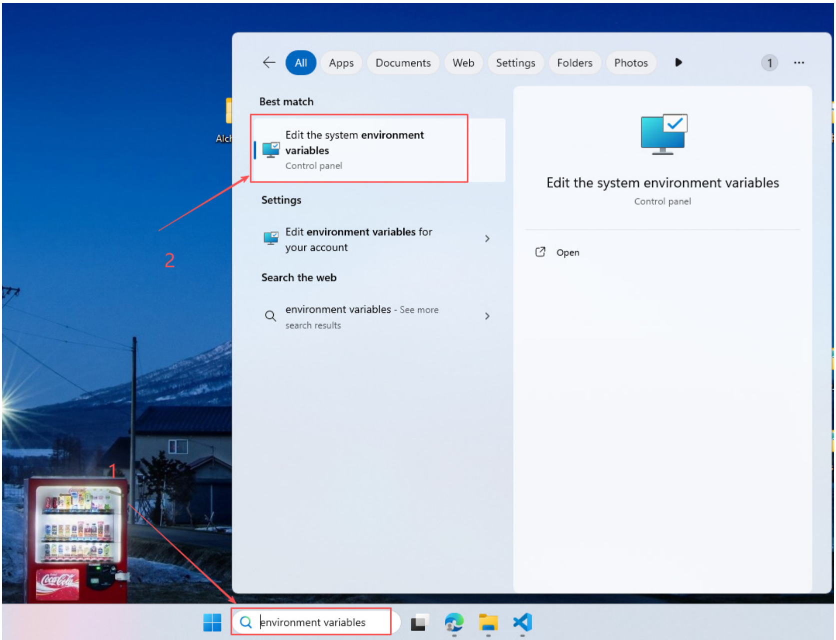
- **"bin":** The folder containing FFmpeg executable files. All commands to run FFmpeg must be executed via the files in this directory.

- **"doc":** Documentation and reference materials.
- **"presets":** Preconfigured formats and encoding schemes.

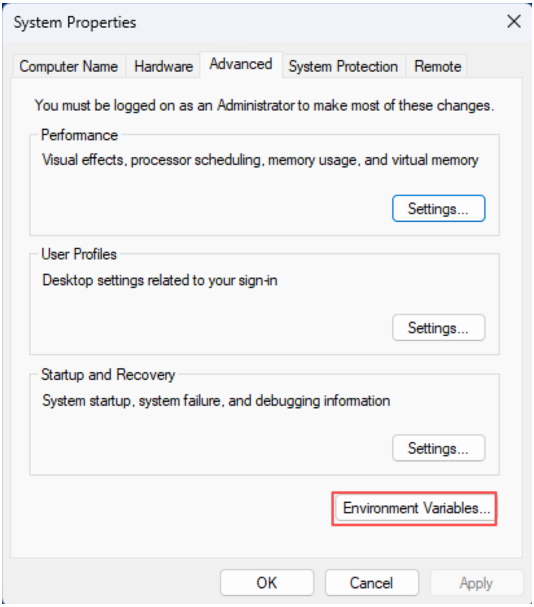
Navigate to the **"bin"** directory, and you will see three core executable files of FFmpeg: **"ffmpeg.exe"**, **"ffplay.exe"**, and **"ffprobe.exe"**.



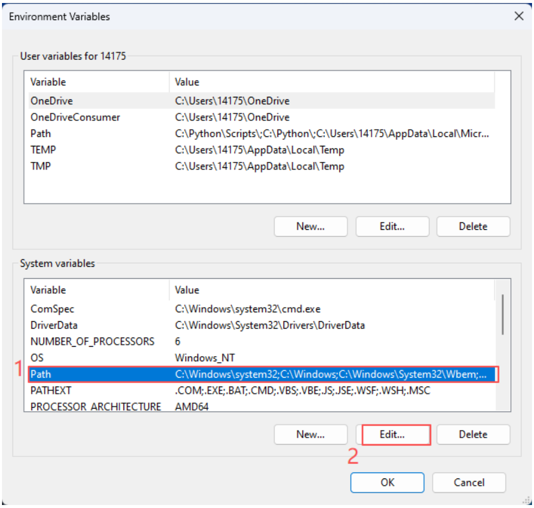
- To conveniently call FFmpeg directly in the command line, you need to add it to the system's environment variables.
- Search for "Environment Variables" in the Start Menu at the bottom left of the desktop, find "Edit the system environment variables", and click to open it.



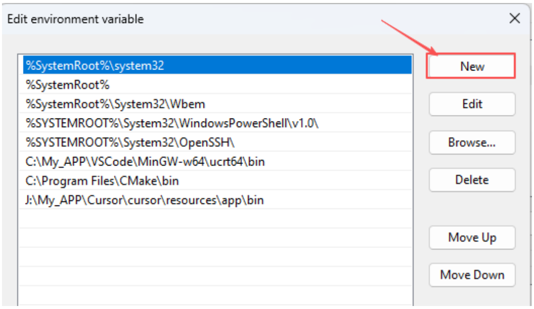
- Click the "Environment Variables" button.



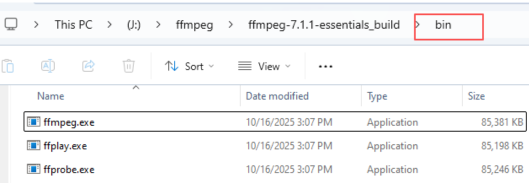
- Locate the "Path" entry under "System Variables" and click "Edit".

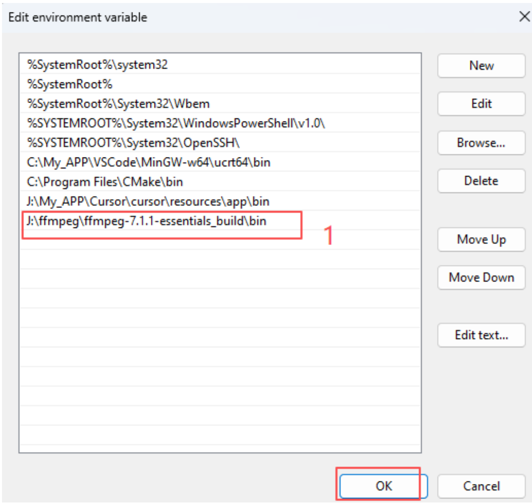


- In the "Edit environment variable" window, click "New".



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

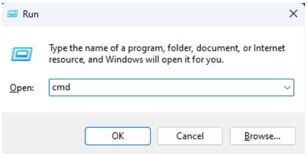




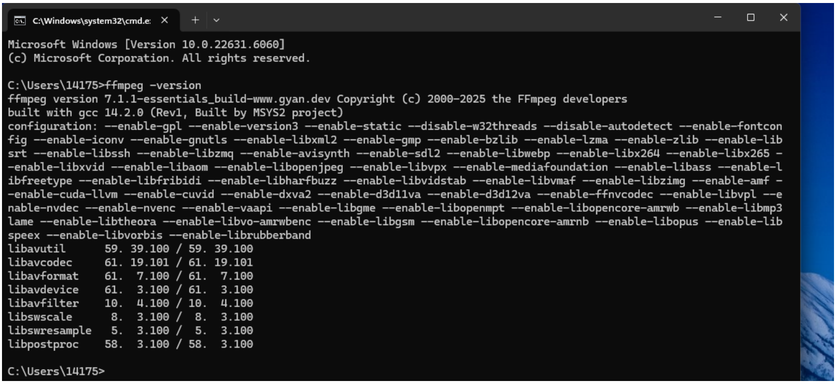
- Remember to save the settings after entering the path.

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

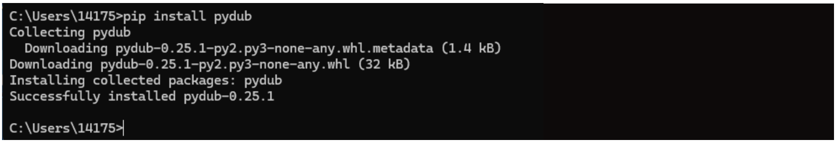
- **Verifying Successful FFmpeg Installation**
- Press the **Win + R** keys, then type "cmd" to open the command line window.



- 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).



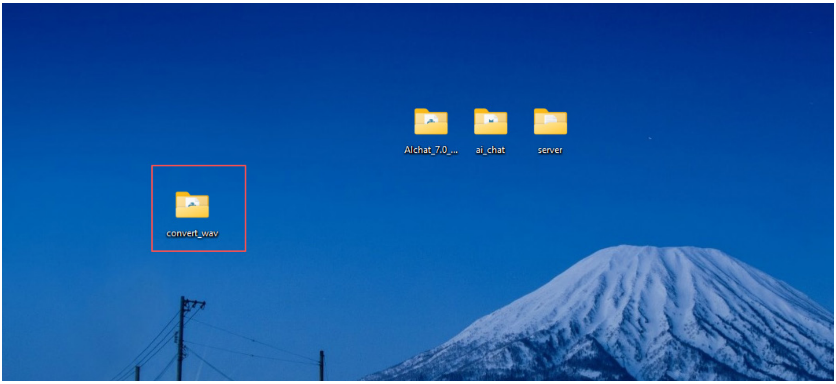
- Then, still in the command window, install the dependency by running: pip install pydub



- After installation, open the script code we prepared for converting MP3 to WAV format (meeting the specifications of 16kHz, 16-bit, and stereo/dual-channel) in the provided code package.
- Click the link below to open the script code:

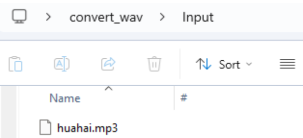
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/convert_wav

- Now I have placed this script on my desktop.

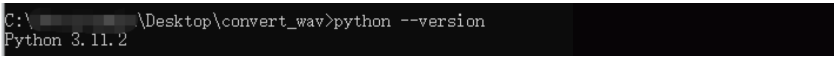


- In the command window, I navigate to this path.

- Then put your MP3 files in the "Input" folder.



- Run this script code. (Ensure your Python environment is Python 3.11.2.)



- Starting from Python 3.13:The official team removed the audioop module (which pydub depends on).Some third-party libraries (such as pyaudio, pygame, pydub) are not yet fully compatible.
- For Python 3.11.x:

  ✅ Stable, mature, and highly compatible;

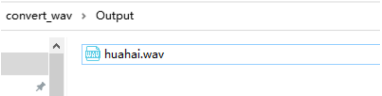
  ✅ Includes audioop;

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

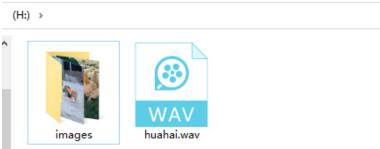
- Run our script:

```
C:\[redacted]\Desktop\convert_wav>python mp3_to_wav.py
[OK] huahai.mp3 -> C:\Users\admin\Desktop\convert_wav\Output\huahai.wav (Conversion: Yes)
□ Batch conversion completed. All files meet ESP32 I2S requirements.
```

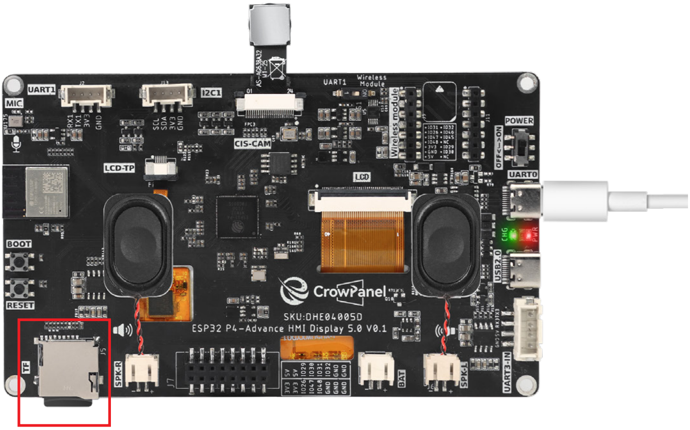
- You will find the generated WAV files in the "Output" folder.



- Then move this file to a USB flash drive.



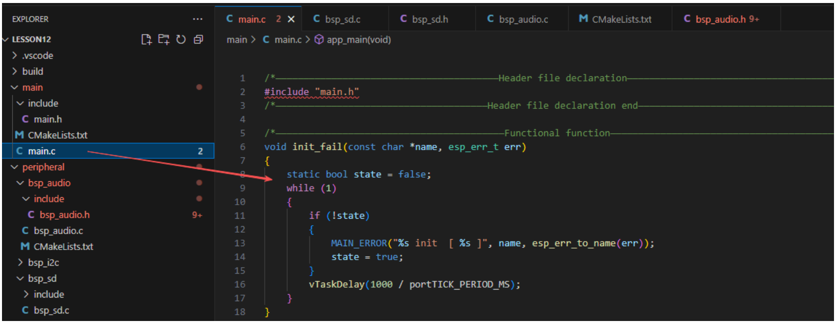
- Finally, remove the SD card and insert it into the **Advance-P4** board.



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.



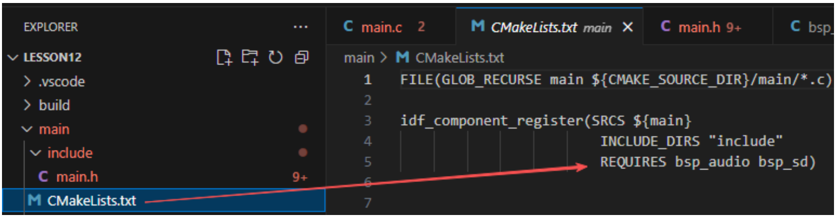
- This is the entry file of the entire application. In ESP-IDF, there is no int main(), and execution starts from void app_main(void).
- Let's first explain main.c.

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.

```
void Init(void)
{
    static esp_err_t err = ESP_OK;

    err = i2c_init();
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);

    err = stc8_i2c_init();
    if (err != ESP_OK)
        init_fail("stc8h1kxx", err);
    MAIN_INFO("I2C and stc8 init success");  // Print success log

    err = sd_init(); /*SD Initialization*/
    if (err != ESP_OK)
        init_fail("sd", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);

    err = audio_ctrl_init(); /*Audio CTRL Initialization*/
    if (err != ESP_OK)
        init_fail("audio ctrl", err);

    set_Audio_ctrl(false);
    err = audio_init(); /*Audio Initialization*/
    if (err != ESP_OK)
        init_fail("audio", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
}

void app_main(void)
{
    MAIN_INFO("----------Start the test--------");
    Init();
```

- 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.

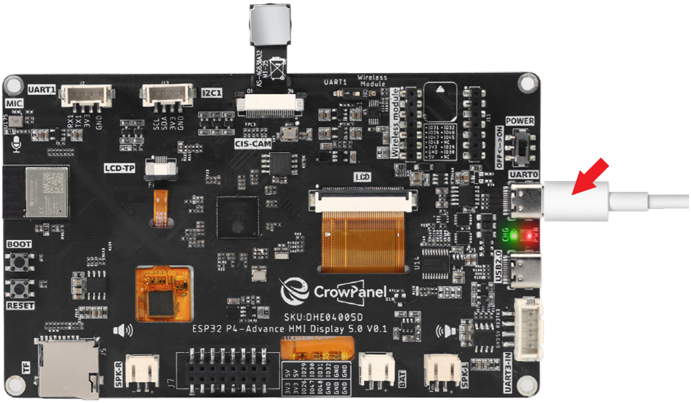
Complete Code

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

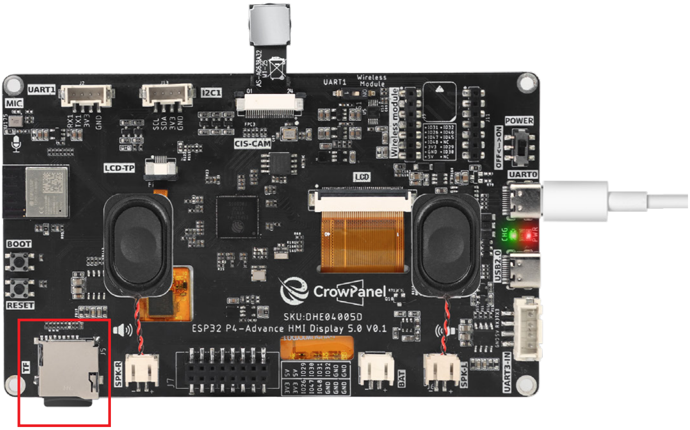
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson12-Playing_Loca_Music_from_SD_Card

Programming Steps

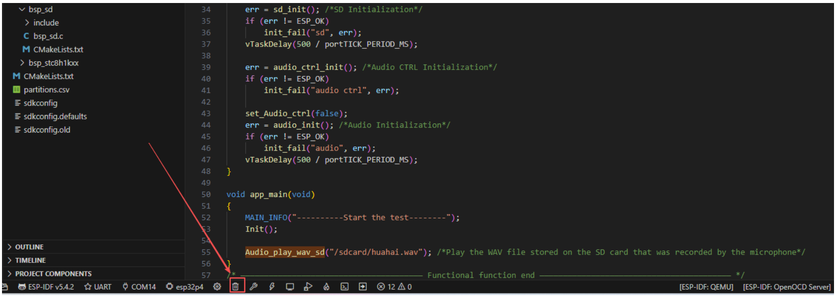
- Now that the code is ready, the next step is to flash it to the ESP32-P4 so we can observe the results.
- First, connect the **Advance-P4** device to your computer using a USB cable.



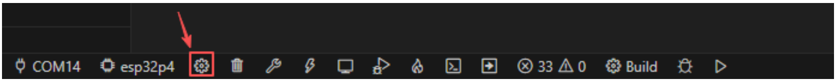
- First, double-check two things: whether the converted WAV audio file has been placed in the SD card, and whether the SD card is inserted into the SD card slot of the **Advance-P4**.



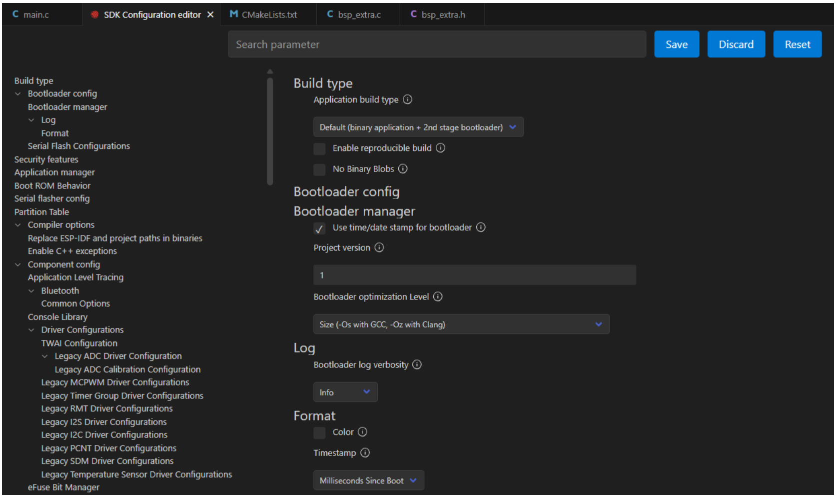
- Before starting the preparation for flashing, delete all files generated by compilation to restore the project to its initial "unbuilt" state. This ensures that subsequent compilations are not affected by your previous operations.



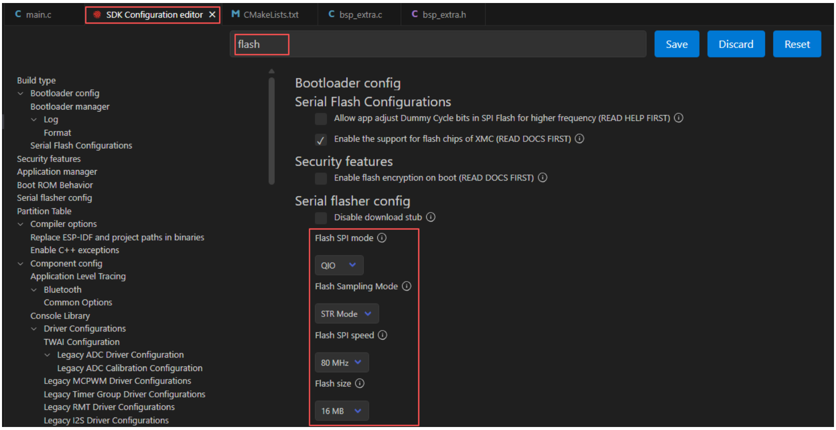
- First, follow the steps in the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon in the figure below.



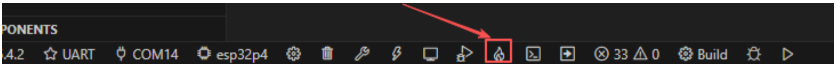
- Wait for a short loading period, then you can proceed with the relevant SDK configuration.



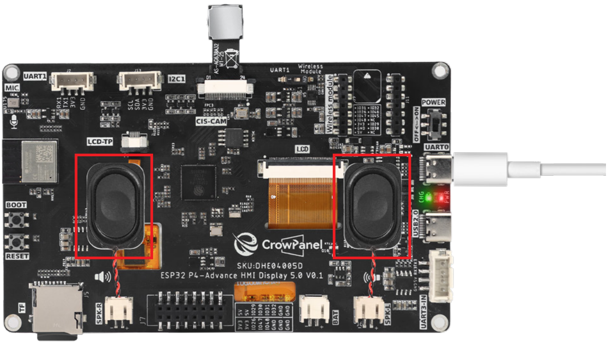
- Next, enter "flash" in the search box. (Make sure your flash configuration matches mine.)



- 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.

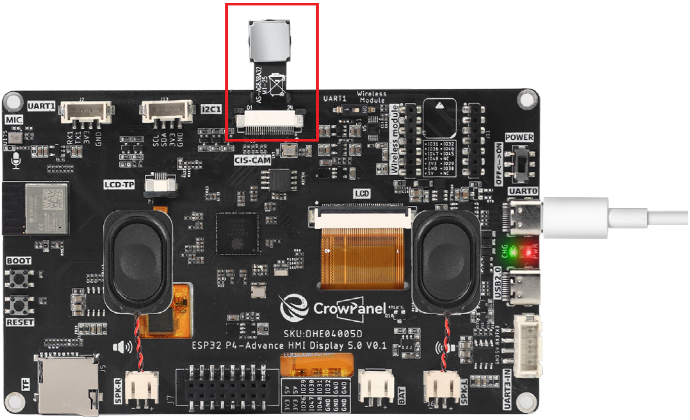
Lesson 13
Camera Real-Time

Introduction

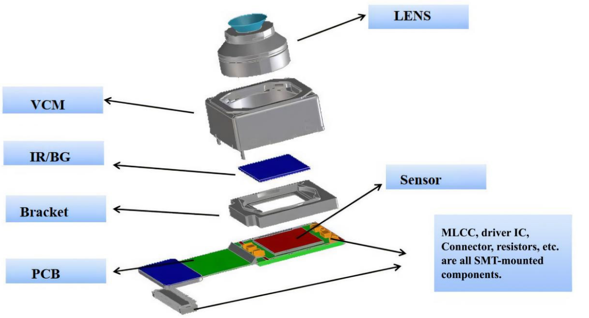
In this lesson, we will start teaching you how to activate the camera, enabling real-time display of the camera feed on the Advance-P4 screen.

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

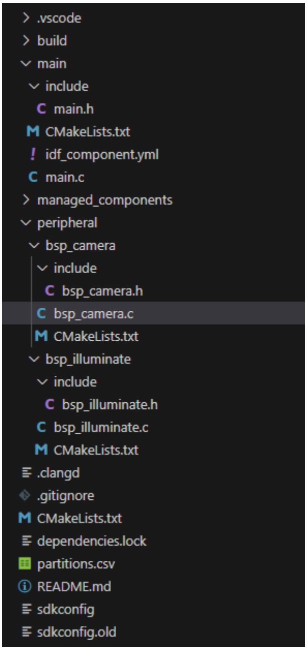
After running the code, you will be able to see the real-time feed from the camera displayed on the screen of the Advance-P4.

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-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-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

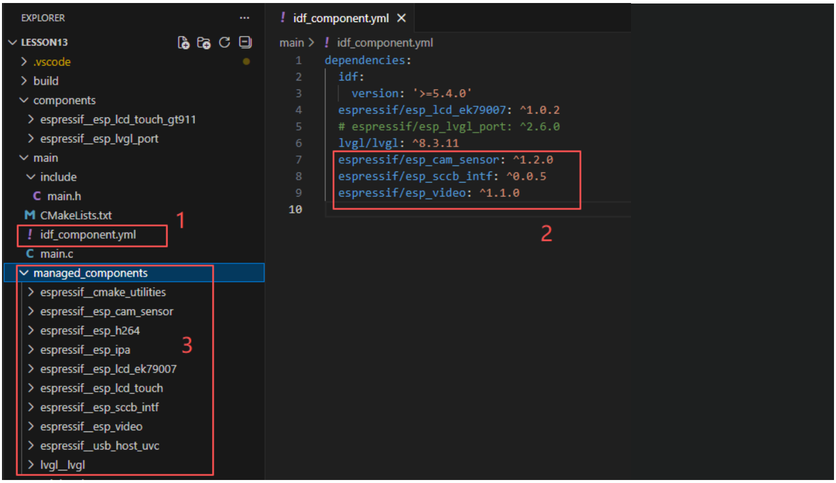
- The camera display code consists of two files: "bsp_camera.c" and "bsp_camera.h".
- Next, we will first analyze the "bsp_camera.h" program.
- "bsp_camera.h" is the header file for camera display, mainly used to:
- Declare the functions, macros, and variables implemented in "bsp_camera.c" for use by external programs;
- Allow other .c files to call this module simply by adding #include "bsp_camera.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_camera.h" file for unified management.

```c
#include <string.h>
#include "esp_log.h"
#include "esp_err.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/i2c_master.h"
#include "driver/isp.h"
#include "esp_etm.h"
#include "esp_async_memcpy.h"
#include "esp_sccb_intf.h"
#include "esp_sccb_i2c.h"
#include "esp_cam_sensor.h"
#include "esp_cam_sensor_detect.h"
#include "esp_cam_ctlr_csi.h"
#include "esp_cam_ctlr.h"
#include "esp_cache.h"
#include "hal/cache_ll.h"
#include "hal/cache_hal.h"
#include "bsp_illuminate.h"
```

- Such as "esp_sccb_intf.h", "esp_sccb_i2c.h", "esp_cam_sensor.h", "esp_cam_sensor_detect.h", and so on (these are all libraries under the network component).

```c
/*------------------------------------Header file declaration------------------------------------*/
#include <string.h>
#include "esp_log.h"
#include "esp_err.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/i2c_master.h"
#include "driver/isp.h"
#include "esp_etm.h"
#include "esp_async_memcpy.h"
#include "esp_sccb_intf.h"
#include "esp_sccb_i2c.h"
#include "esp_cam_sensor.h"
#include "esp_cam_sensor_detect.h"
#include "esp_cam_ctlr_csi.h"
#include "esp_cam_ctlr.h"
#include "esp_cache.h"
#include "hal/cache_ll.h"
#include "hal/cache_hal.h"
#include "bsp_illuminate.h"

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

- 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".)

```
/*————————————————————————————————————————Variable declaration————————————————————————————————————————*/
#define CAMERA_TAG "CAMERA"
#define CAMERA_INFO(fmt, ...) ESP_LOGI(CAMERA_TAG, fmt, ##__VA_ARGS__)
#define CAMERA_DEBUG(fmt, ...) ESP_LOGD(CAMERA_TAG, fmt, ##__VA_ARGS__)
#define CAMERA_ERROR(fmt, ...) ESP_LOGE(CAMERA_TAG, fmt, ##__VA_ARGS__)

#define SCCB_MASTER_PORT 1
#define SCCB_GPIO_SCL 34
#define SCCB_GPIO_SDA 33

esp_err_t camera_init();
void camera_display();
esp_err_t camera_refresh();
void camera_display_refresh();

extern lv_img_dsc_t img_camera;
extern esp_cam_ctlr_trans_t my_trans;
extern esp_cam_ctlr_handle_t cam_handle;

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

- 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_ctlr_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()).

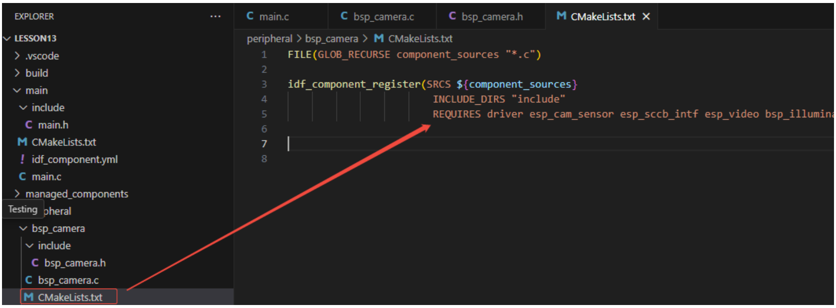
10. camera_display()

Function:

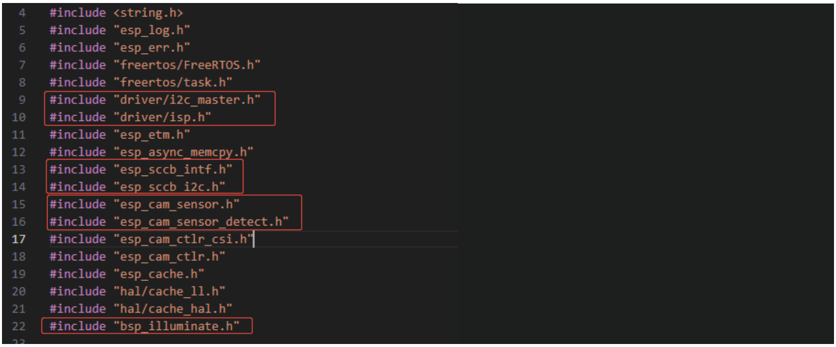
- Creates an image object in LVGL for displaying the camera feed.
- The specific steps are as follows:
  - Create an lv_img object;
  - Set center alignment for the object;
  - Bind the image buffer (RGB565 data captured by the camera);
  - Configure the image source;
  - Unlock LVGL to allow rendering.

Calling Timing:

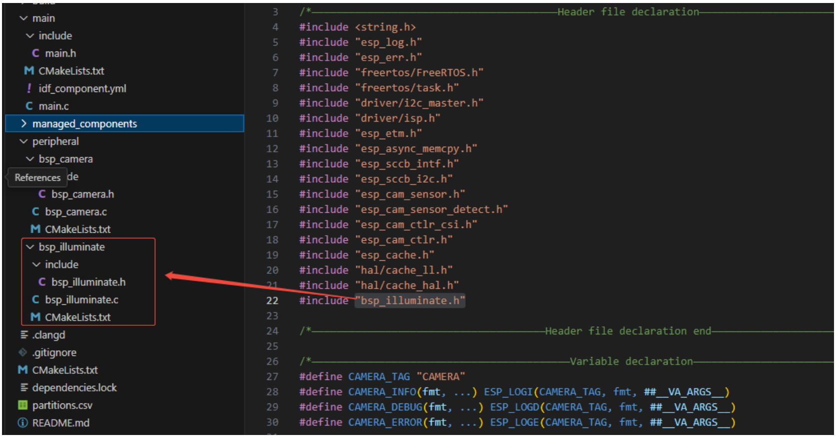
- Called once after the camera is initialized successfully, to create and display the image control (invoked within Init()).
- This concludes our introduction to the bsp_camera component. For your purposes, it is sufficient to know how to call these interfaces.
- If you need to call these interfaces, 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).

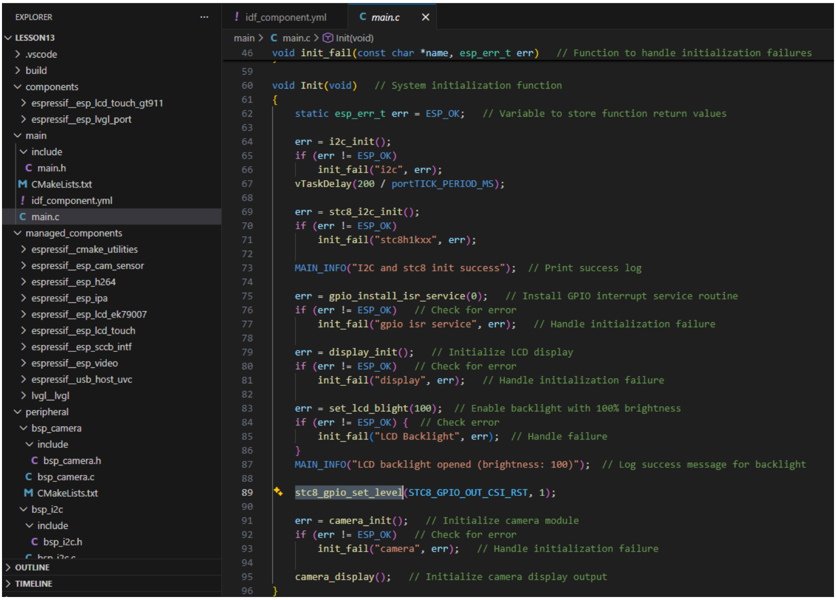


- For example, "bsp_illuminate.h" is a component related to screen display that we explained earlier. Since it was covered in detail before, we won't go into it again here.
- It is used to initialize the screen, turn on the screen backlight, and enable the screen to display relevant content.



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.



- This is the entry file of the entire application. There is no int main() in ESP-IDF; instead, the program starts running from void app_main(void).
- First, let's explain "main.c".
- When the program runs, the general process is as follows:
- During program execution, the system first calls Init() in app_main() to initialize hardware and modules: configure the LDO power supply, GPIO interrupts, LCD display and backlight, and initialize the camera and display buffer.
- After initialization is completed, the program first captures a frame of camera feed, then creates the camera_display_task task and enters a loop: lock LVGL, refresh the camera display, unlock LVGL, and delay for approximately 23ms. This loop continuously updates the frame, enabling real-time camera display.
- Next, let's explain the main code "main.c".

```
/*——————————————————————————————————Header file declaration——————————————————————————————————*/
#include "main.h"   // Include the main header file containing required definitions and declarations
/*——————————————————————————————————Header file declaration end——————————————————————————————*/
```

- It includes the custom main header file "main.h", which typically contains log macros, peripheral initialization declarations, and header files of other interfaces that need to be used.
- Below is the content within "main.h":

```
LESSON13
  > .vscode
  > build
  v main
    v include
      C main.h
    M CMakeLists.txt
    ! idf_component.yml
    C main.c
  > managed_components
  v peripheral
    v bsp_camera
      v include
        C bsp_camera.h
      C bsp_camera.c
      M CMakeLists.txt
    v bsp_illuminate
      v include
        C bsp_illuminate.h
      C bsp_illuminate.c
      M CMakeLists.txt
  .clangd
  .gitignore
  M CMakeLists.txt
  dependencies.lock
  partitions.csv
```

main > include > C main.h > ...

```
#ifndef _MAIN_H_
#define _MAIN_H_

/*——————————————————————————————Header file declaration——————————————————————————————*/
#include <stdio.h>
#include "string.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "esp_private/esp_clk.h"
#include "esp_ldo_regulator.h"
#include "esp_sleep.h"
#include "driver/rtc_io.h"
#include "esp_timer.h"

#include "bsp_camera.h"
#include "bsp_illuminate.h"

/*——————————————————————————————Header file declaration end——————————————————————————*/
/*——————————————————————————————Variable declaration——————————————————————————————————*/
#define MAIN_TAG "MAIN"
#define MAIN_INFO(fmt, ...) ESP_LOGI(MAIN_TAG, fmt, ##__VA_ARGS__)
#define MAIN_DEBUG(fmt, ...) ESP_LOGD(MAIN_TAG, fmt, ##__VA_ARGS__)
#define MAIN_ERROR(fmt, ...) ESP_LOGE(MAIN_TAG, fmt, ##__VA_ARGS__)

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

- Let's continue to look at the content in "main.c".
- lvgl_camera: A handle for the LVGL display task, used to manage the display task.
- Function declarations:

- ldo3 : The ESP32-P4 has an externally controllable low-dropout voltage regulator (LDO) power channel. The camera requires a stable voltage, so an LDO must be applied first: ldo3 -> 2.5V. This is the basic power supply for the camera module to operate normally. In this code, through our practice, the camera needs the support of the LDO to display images correctly.
    - init_fail: Handles initialization failure.
    - Init: Performs system hardware initialization.
    - camera_display_task: Implements the camera display refresh task.

```c
/*---------------------------------------------Variable declaration---------------------------------------------*/

TaskHandle_t lvgl_camera;    // Task handle for LVGL camera display task
TaskHandle_t camera_read;

// static esp_ldo_channel_handle_t ldo4 = NULL;
static esp_ldo_channel_handle_t ldo3 = NULL;

// function declaration
void init_fail(const char *name, esp_err_t err);   // Function declaration for initialization failure handling
void Init(void);   // Function declaration for system initialization
void camera_display_task(void *param);   // Function declaration for camera display task
/*---------------------------------------------Variable declaration end---------------------------------------------*/
```

camera_display_task:

- A FreeRTOS task function used to continuously refresh the camera display.

Core Process:

- Infinite loop while(1).
- Attempt to acquire the LVGL lock via lvgl_port_lock(0).
- If the lock is successfully acquired, call camera_display_refresh() to update the display buffer to the screen.
- Unlock LVGL with lvgl_port_unlock().
- Delay for 23ms (vTaskDelay) to control the refresh rate, approximately 43 FPS.

Once the task is created after program startup, it will continuously refresh the camera display.

```c
void camera_display_task(void *param)   // Task function to continuously refresh the camera display
{
    while (1)   // Infinite loop for periodic refreshing
    {
        // Directly refresh the camera display without the need for status check
        if (lvgl_port_lock(0))   // Lock LVGL port for safe access (timeout = 0)
        {
            camera_display_refresh();   // Refresh camera display content
            lvgl_port_unlock();   // Unlock LVGL port after refresh
        }
        vTaskDelay(23 / portTICK_PERIOD_MS);   // Delay approximately 23 ms between refreshes
    }
}
```

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.

```
void init_fail(const char *name, esp_err_t err)   // Function to handle initialization failures
{
    static bool state = false;   // Flag to avoid repeated error logging
    while (1)   // Stay in infinite loop after failure
    {
        if (!state)   // Print error message only once
        {
            MAIN_ERROR("%s init  [ %s ]", name, esp_err_to_name(err));   // Log initialization failure with error name
            state = true;   // Update state to prevent repeated logs
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);   // Wait 1 second before looping again
    }
}
```

init:

- Hardware initialization function during system startup.

Initialization Steps:

- ldo3 : The function of this code is very simple. It turns on the LDO3 power channel of the ESP32-P4 to provide the correct working voltage for the external camera.
- Initialize the I2C and STC8 chips required for the expansion chip, which will facilitate the subsequent activation of the screen.
- Initialize the LCD display with display_init().
- Turn on the LCD backlight using set_lcd_blight(100).
- Initialize the camera module with camera_init().
- Initialize the camera display with camera_display().

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

```
void Init(void)   // System initialization function
{
    static esp_err_t err = ESP_OK;   // Variable to store function return values

    esp_ldo_channel_config_t ldo3_cof = {
        .chan_id = 3,
        .voltage_mv = 2500,
    };
    err = esp_ldo_acquire_channel(&ldo3_cof, &ldo3);
    if (err != ESP_OK)
        init_fail("ldo3", err);
    // esp_ldo_channel_config_t ldo4_cof = {
    //     .chan_id = 4,
    //     .voltage_mv = 3300,
    // };
    // err = esp_ldo_acquire_channel(&ldo4_cof, &ldo4);
    // if (err != ESP_OK)
    //     init_fail("ldo4", err);

    err = i2c_init();
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);

    err = stc8_i2c_init();
    if (err != ESP_OK)
        init_fail("stc8h1kxx", err);

    MAIN_INFO("I2C and stc8 init success");  // Print success log

    err = set_lcd_blight(100);  // Enable backlight with 100% brightness
    if (err != ESP_OK) {  // Check error
        init_fail("LCD Backlight", err);  // Handle failure
    }
    MAIN_INFO("LCD backlight opened (brightness: 100)");  // Log success message for backlight

    err = display_init();   // Initialize LCD display
    if (err != ESP_OK)   // Check for error
        init_fail("display", err);   // Handle initialization failure

    err = camera_init();   // Initialize camera module
    if (err != ESP_OK)   // Check for error
        init_fail("camera", err);   // Handle initialization failure

    camera_display();   // Initialize camera display output
}
```

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.

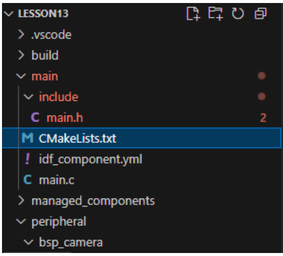
```c
void app_main(void)   // Main application entry point
{
    MAIN_INFO("----------Camera task----------\r\n");   // Print start log message

    Init();   // Call system initialization function

    camera_refresh();   // Refresh camera once before starting display loop

    xTaskCreatePinnedToCore(camera_display_task, "camera_display", 4096, NULL, configMAX_PRIORITIES - 4, &lvgl_camera, 1);
    // Create and start the camera display task on Core 1 with priority (max - 4)

    MAIN_INFO("----------The screen is displaying.----------\r\n");   // Log that the screen is now displaying camera output
}
/*————————————————————————————————————Functional function end————————————————————————————————————*/
```

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".



main > CMakeLists.txt

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_illuminate bsp_camera)

```

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

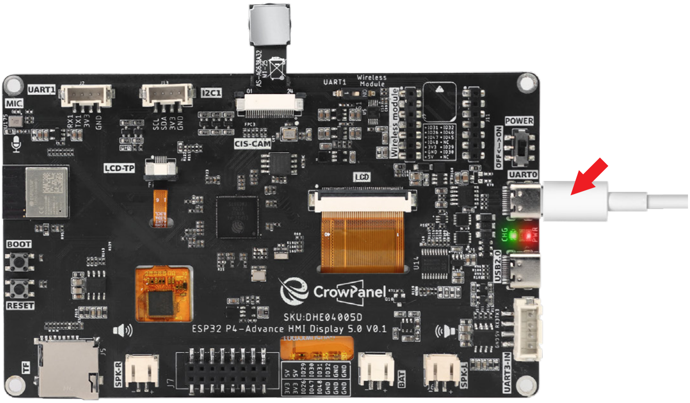
Complete Code

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

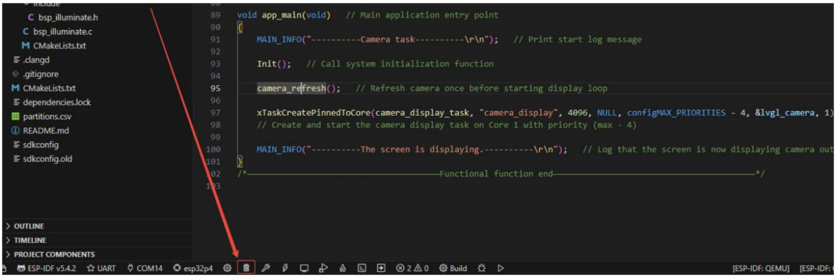
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson13-Camera_Real-Time

Programming Steps

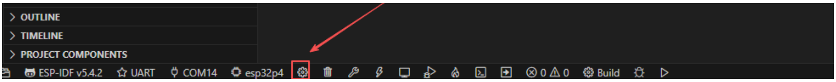
- Now that the code is ready, the next step is to flash it to the ESP32-P4 so we can observe the actual behavior.
- First, connect the Advance-P4 device to your computer via a USB cable.



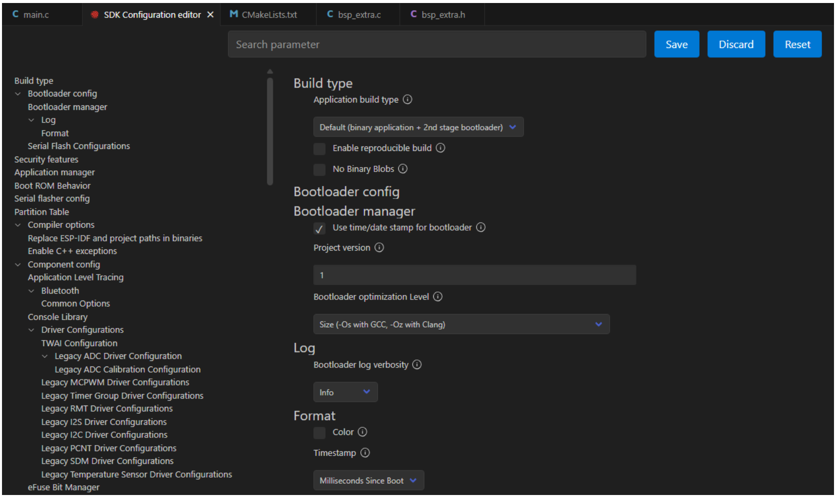
- Before starting the flashing preparation, delete all compiled files to restore the project to its initial "unbuilt" state. (This ensures that subsequent compilations are not affected by your previous build artifacts.)



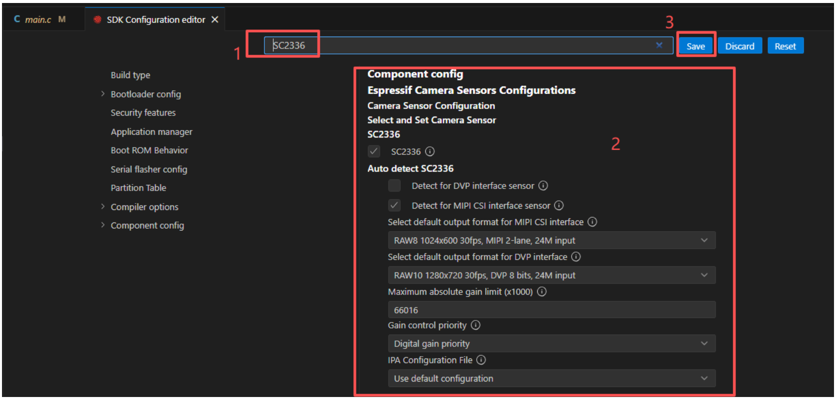
- First, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip correctly.
- Next, we need to configure the SDK.
- Click the icon shown in the image below.



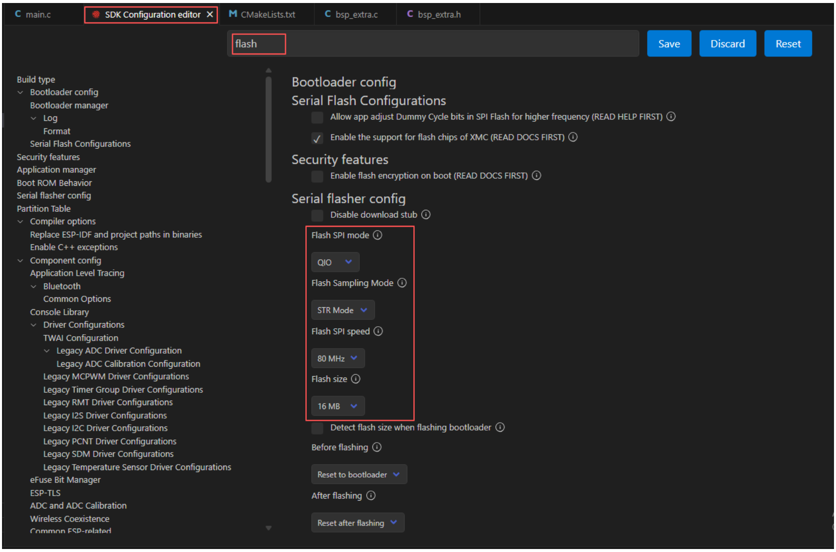
- Wait for a short loading period, and then you can proceed with the relevant SDK configuration.



- For more related configurations regarding screen lighting, please refer to Lesson 7.
- Here you need to configure the camera options to use the camera normally.
- (Just make the same configuration as shown in the picture.)



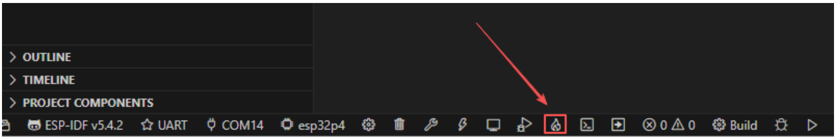
- After that, enter "flash" in the search box to find flash-related settings. (Make sure your flash configuration matches mine exactly.)



- After completing the configuration, remember to save your settings.

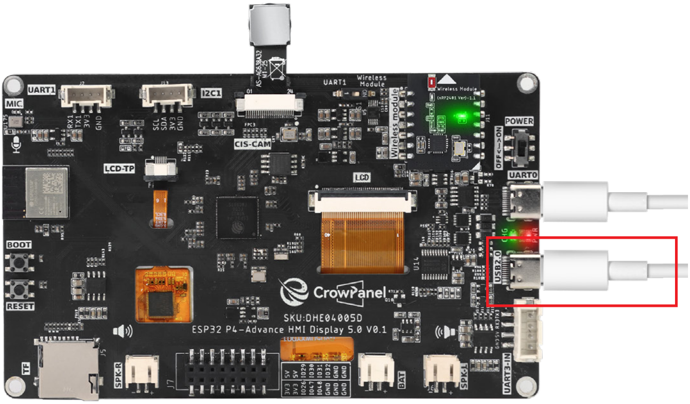
- Next, we will compile and flash the code (detailed in the first lesson).

- 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

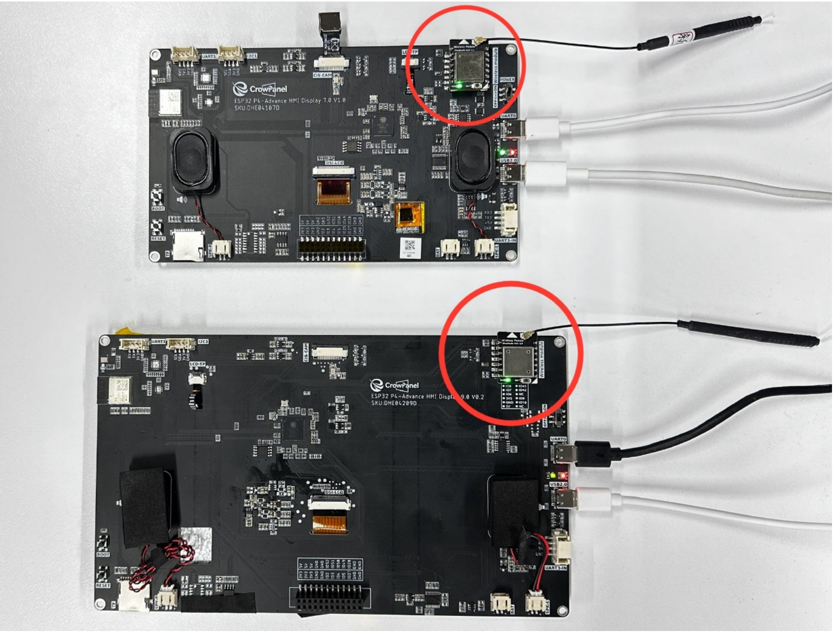
Introduction

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

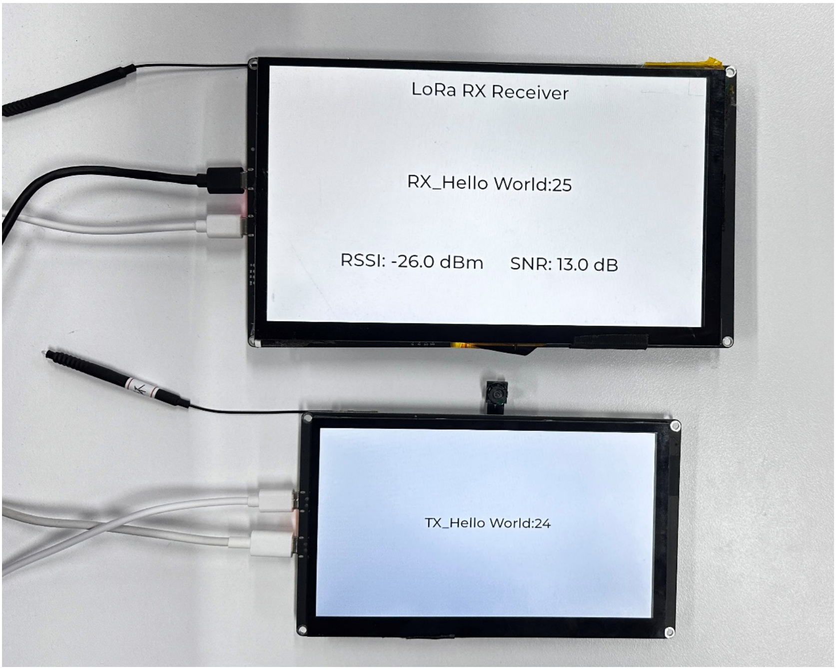


Operation Effect Diagram

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.

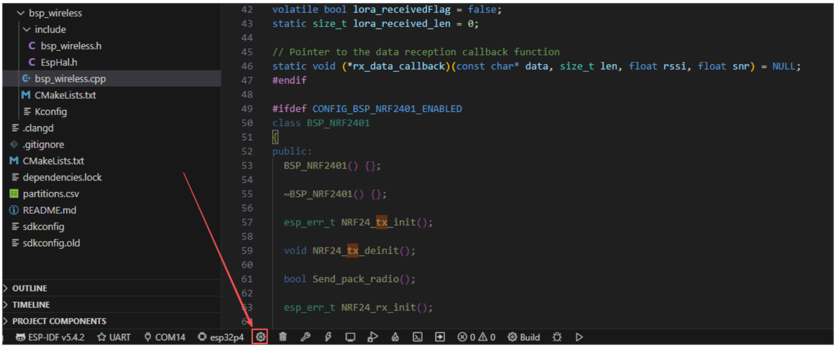


Key Explanations

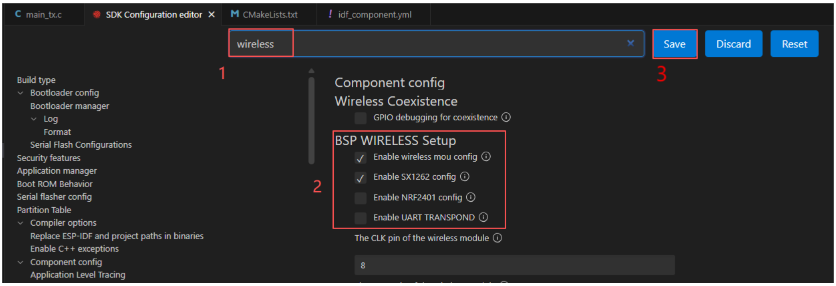
- The main focus of this lesson is to learn how to use the wireless module, including how to initialize the SX1262 LoRa module and send or receive data.
- In this section, we will introduce a new component called bsp_wireless.
- The main functions of this component are as follows:
    - It encodes and 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.



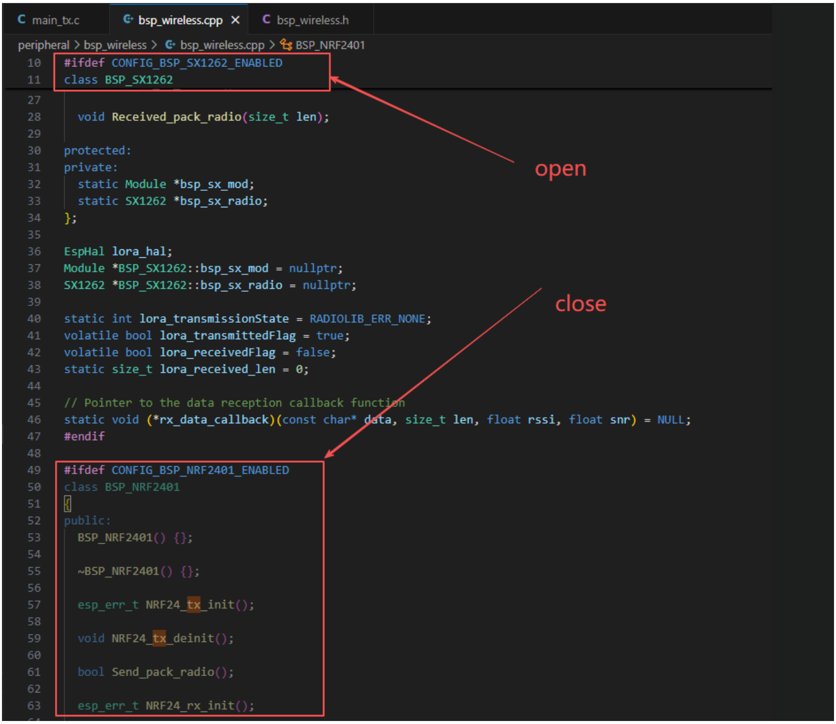
- Search for **"wireless"** and open the configuration you are using.



- 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.



- As shown in the figure, we have enabled the SX1262-related configuration, so the other wireless modules are currently disabled and not in use.
- Within the bsp_wireless component, you only need to know when to call the provided interfaces that we have written.
- Next, let's focus on understanding the bsp_wireless component itself.
- First, click the GitHub link below to download the source code for this lesson.
- Transmitting end code:

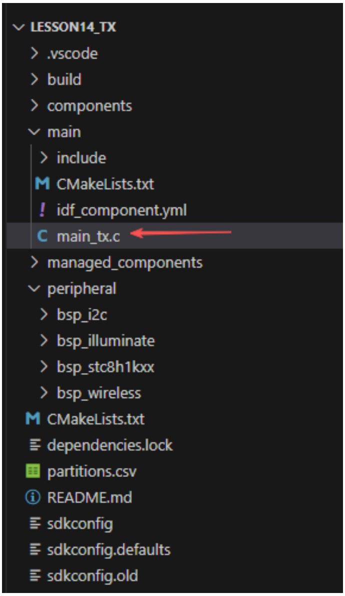
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14_TX_SX1262_Wireless_Module

- Receiving end code:

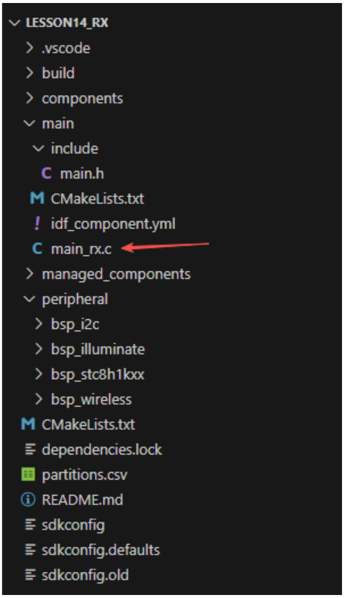
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14_RX_SX1262_Wireless_Module

- 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.
- And the "bsp_stc8h1kxx" component can be used to control the screen backlight.

- **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.

main_rx.c | main.h | bsp_wireless.cpp | **bsp_wireless.h**

peripheral > bsp_wireless > include > bsp_wireless.h > ...

```c
#ifndef _BSP_WIRELESS_H
#define _BSP_WIRELESS_H

/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include <string.h>
#include <stdint.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "driver/uart.h"

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

main_rx.c | main.h | **bsp_wireless.cpp** | bsp_wireless.h

peripheral > bsp_wireless > bsp_wireless.cpp > BSP_SX1262

```cpp
/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include "bsp_wireless.h"
#include <RadioLib.h>
#include "EspHal.h"
#include <stdio.h>
#include <string.h>
/*——————————————————————————————————Header file declaration end——————————————————————————————————*/
```

- 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.

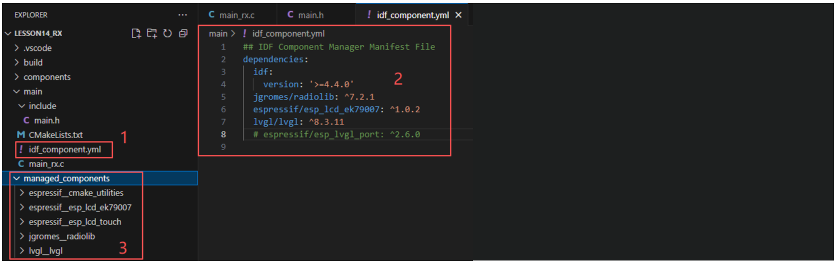
main_rx.c | main.h | bsp_wireless.cpp | bsp_wireless.h | idf_component.yml

peripheral > bsp_wireless > bsp_wireless.cpp > ...

```cpp
/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include "bsp_wireless.h"
#include <RadioLib.h>
#include "EspHal.h"
#include <stdio.h>
#include <string.h>
/*——————————————————————————————————Header file declaration end——————————————————————————————————*/
```

- This requires us to specify the version of jgromes/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.

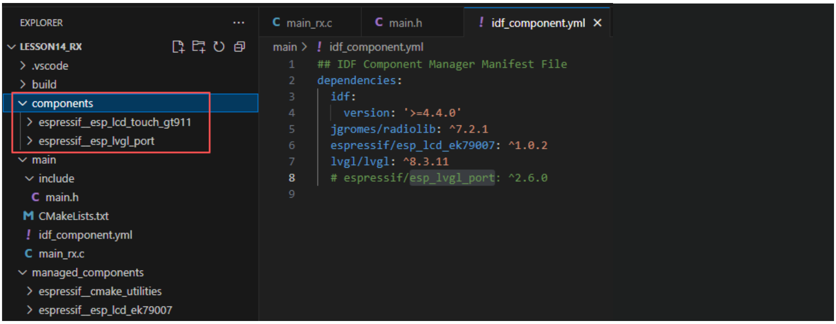


- These three components, which we discussed earlier, are used in the bsp_illuminate component to light up the screen and render information on the interface using LVGL.

main > ! idf_component.yml

```yaml
## IDF Component Manager Manifest File
dependencies:
  idf:
    version: '>=4.4.0'
  jgromes/radiolib: ^7.2.1
  espressif/esp_lcd_ek79007: ^1.0.2
  lvgl/lvgl: ^8.3.11
  # espressif/esp_lvgl_port: ^2.6.0
```

- During the subsequent compilation process, the project will automatically download the following library versions:
    - jgromes/Radiolib version 7.2.1
    - espressif/esp_lcd_ek79007 version 1.0.2
    - lvgl version 8.3.11
- Once downloaded, these online components will be stored in the managed_components folder. (This is automatically generated after specifying the version numbers.)
- It can be seen that the esp_lvgl_port has been commented out by me. This is because in the components, we downloaded this network component and then made relevant modifications based on it to adapt it to our 5-inch Advance-P4 product.



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

```c
#define RADIO_GPIO_CLK 26
#define RADIO_GPIO_MISO 47
#define RADIO_GPIO_MOSI 48

//-------------------------------------

#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY 29
#define SX1262_GPIO_IRQ 31
#define SX1262_GPIO_NRST 32
#define SX1262_GPIO_NSS 30
```

- The pin assignments should not be modified, otherwise the wireless module will not work due to incorrect connections.
- Next, we declare the variables and functions that we will use. The actual implementation of these functions is in bsp_wireless.cpp.
- By placing them all in bsp_wireless.h, it becomes easier to call and manage them. (We will explore their specific functionality when we look at bsp_wireless.cpp.)

```
//---------------------------------------------------------------------------
#ifdef CONFIG_BSP_SX1262_ENABLED

#define SX1262_GPIO_BUSY 29
#define SX1262_GPIO_IRQ 31
#define SX1262_GPIO_NRST 32
#define SX1262_GPIO_NSS 30

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t sx1262_tx_init();
    void sx1262_tx_deinit();
    bool send_lora_pack_radio();

    uint32_t sx1262_get_tx_counter();

    esp_err_t sx1262_rx_init();
    void sx1262_rx_deinit();
    void received_lora_pack_radio(size_t len);
    void sx1262_set_rx_callback(void (*callback)(const char* data, size_t len, float rssi, float snr));
    size_t sx1262_get_received_len(void);
    bool sx1262_is_data_received(void);
#ifdef __cplusplus
}
#endif

#endif
//---------------------------------------------------------------------------
```

- Next, let's take a look at bsp_wireless.cpp to understand the specific function of each function.
- The bsp_wireless component implements LoRa data transmission and reception, communicates with the main controller via the SPI interface, and handles the sending and receiving at the wireless data link layer.
- Here, we won't go into the detailed code. Instead, we will explain the purpose of each function and when to call them.

BSP_SX1262 Class:

This indicates that:

- It is a C++ wrapper class for operating the SX1262 module.
- It mainly provides functions for initialization, de-initialization, and data transmission/reception.
- All hardware operations are performed based on the RadioLib library.
- bsp_sx_mod and bsp_sx_radio are object pointers in memory for the SX1262 module (statically shared).

```
class BSP_SX1262
{
public:
    BSP_SX1262() {};

    ~BSP_SX1262() {};

    esp_err_t Sx1262_tx_init();

    void Sx1262_tx_deinit();

    bool Send_pack_radio();

    esp_err_t Sx1262_rx_init();

    void Sx1262_rx_deinit();

    void Received_pack_radio(size_t len);

protected:
private:
    static Module *bsp_sx_mod;
    static SX1262 *bsp_sx_radio;
};
```

Defines the **core global variables required by the SX1262 LoRa module driver**, used to manage the module instance, status, and data callbacks:

- lora_hal is the low-level SPI hardware abstraction layer object, responsible for SPI communication.
- bsp_sx_mod and bsp_sx_radio point to the generic RadioLib module object and the SX1262 module object, respectively. They encapsulate the specific hardware pins and transmission/reception interfaces. These objects are created during module initialization (e.g., Sx1262_tx_init() or Sx1262_rx_init()) and released or set to standby during de-initialization.
- lora_transmissionState records the status code of the last transmission operation for debugging and error handling.
- lora_transmittedFlag is the transmission completion flag, set by the transmission interrupt callback set_sx1262_tx_flag(), indicating that the module is ready to send a new data packet.

- lora_receivedFlag is the reception completion flag, set by the reception interrupt callback set_sx1262_rx_flag(), indicating that new data is available to read.
- lora_received_len stores the length of the most recently received data.
- rx_data_callback is a function pointer that allows the upper layer to register a callback. When the SX1262 receives data, this callback is automatically triggered, passing the received data, its length, RSSI, and SNR information to the upper-level processing.

```
EspHal lora_hal;
Module *BSP_SX1262::bsp_sx_mod = nullptr;
SX1262 *BSP_SX1262::bsp_sx_radio = nullptr;

static int lora_transmissionState = RADIOLIB_ERR_NONE;
volatile bool lora_transmittedFlag = true;
volatile bool lora_receivedFlag = false;
static size_t lora_received_len = 0;

// Pointer to the data reception callback function
static void (*rx_data_callback)(const char* data, size_t len, float rssi, float snr) = NULL;
#endif
```

Sx1262_tx_init():

The function Sx1262_tx_init() in the BSP_SX1262 class is used to initialize the SX1262 module for data transmission.

- The function first uses lora_hal to configure the SPI pins (RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI) and the SPI clock frequency (8 MHz), then calls spiBegin() to start SPI communication, providing the module with a low-level communication interface.
- Next, it creates a Module object bsp_sx_mod to encapsulate the SX1262 hardware pins (NSS, IRQ, NRST, BUSY) and uses this module object to create the SX1262 instance bsp_sx_radio. By calling begin(), it configures the LoRa parameters (915 MHz frequency, 125 kHz bandwidth, spreading factor 7, coding rate 4/7, sync word, 22 dBm power, pre-gain 8, LNA 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.

```
esp_err_t BSP_SX1262::Sx1262_tx_init()
{
  lora_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI);
  lora_hal.setSpiFrequency(8000000);
  lora_hal.spiBegin();

  bsp_sx_mod = new Module(&lora_hal, SX1262_GPIO_NSS, SX1262_GPIO_IRQ, SX1262_GPIO_NRST, SX1262_GPIO_BUSY);
  bsp_sx_radio = new SX1262(bsp_sx_mod);
  int state = bsp_sx_radio->begin(915.0, 125.0, 7, 7, RADIOLIB_SX126X_SYNC_WORD_PRIVATE, 22, 8, 1.6);
  if (state != RADIOLIB_ERR_NONE)
  {
    SX1262_ERROR("radio tx init failed, code :%d", state);
    lora_hal.spiEnd();
    return ESP_FAIL;
  }
  // bsp_sx_radio->setCurrentLimit(60);
  bsp_sx_radio->setPacketSentAction(set_sx1262_tx_flag);
  return ESP_OK;
}
```

In bsp_sx_radio->begin(), the 915.0 MHz represents the operating center frequency of the SX1262. This can be changed according to the LoRa frequency regulations of different regions:

- China commonly uses 433 MHz or 470–510 MHz
- Europe uses 868 MHz
- The United States and Australia use 915 MHz
- Japan uses 923 MHz

When changing the frequency, the transmitter and receiver must match, otherwise communication will fail. Additionally, ensure that the selected frequency falls within the legally allowed ISM band for that region.

Parameters such as bandwidth and spreading factor can generally remain unchanged, although some frequency bands may have officially recommended values.

Send_pack_radio:

The function Send_pack_radio() in the BSP_SX1262 class is the core function for sending LoRa data packets.

- It first checks the transmission completion flag lora_transmittedFlag. If it is true, it indicates that the previous packet has been sent and the module is ready to send new data.
- If so, the flag is reset to false to prevent duplicate transmissions. The function then checks lora_transmissionState to determine whether the previous transmission was successful and prints the corresponding log.
- Next, it calls bsp_sx_radio->finishTransmit() to complete any remaining operations from the previous transmission, ensuring the module is ready for use. The transmission counter sx1262_tx_counter is incremented, and a text message with the counter is formatted and stored in the static buffer text.

- The function then calculates the message length and calls bsp_sx_radio->startTransmit() to initiate the transmission of the new data packet. It also updates lora_transmissionState to record the status of this transmission. If the transmission fails to start, an error message is printed.
- Finally, the function returns true if the transmission event has been handled, or false if the module is not yet ready to send.

This function is usually called periodically in the main loop or task scheduler to poll and send LoRa data packets, and it must ensure that the previous transmission is complete before sending a new packet.

sx1262_get_tx_counter()

This is a C-style interface used to obtain the value of the SX1262 module's transmitted packet counter sx1262_tx_counter. The function simply returns the global static variable sx1262_tx_counter and does not modify any state. It is typically used in applications to query the number of packets sent, for example, for debugging, statistics, or displaying the transmission count. It can be called at any time and does not depend on the transmission or reception status.

sx1262_tx_init()

This is a C-style wrapper interface for initializing the SX1262 transmission functionality. Inside the function, a BSP_SX1262 object is created, and its method Sx1262_tx_init() is called to complete the LoRa module SPI configuration, module object creation, parameter initialization, and registration of the transmission completion callback. The function returns ESP_OK if initialization is successful, or ESP_FAIL if it fails. This function is typically called once at system startup or before starting data transmission to ensure that the module is in a ready-to-transmit state.

sx1262_tx_deinit()

This is a C-style de-initialization interface for the SX1262 transmission function. Inside the function, a BSP_SX1262 object is created, and its method Sx1262_tx_deinit() is called to shut down the transmission functionality. During de-initialization, it calls finishTransmit() to complete any ongoing transmission, clears the transmission callback, switches the module to standby mode, and closes the SPI interface. This function is generally called when the system is shutting down, the module no longer needs to send data, or it enters low-power mode, releasing resources and ensuring the module safely stops.

send_lora_pack_radio()

This is a C-style interface used to trigger the SX1262 to send a data packet. Inside the function, a BSP_SX1262 object is created, and its method Send_pack_radio() is called. It polls the transmission completion flag lora_transmittedFlag and, when ready, generates a data packet and starts transmission. The function returns true if the transmission event has been handled, or false if the module is not yet ready. It is usually called periodically in the main loop or task scheduler to achieve continuous or scheduled data transmission.

set_sx1262_rx_flag()

This is a static internal function used as the callback for SX1262 reception completion. Inside the function, it sets the global reception flag lora_receivedFlag to true, notifying the system that a new data packet has been received.

It is not called directly. Instead, it is registered by calling bsp_sx_radio->setPacketReceivedAction(set_sx1262_rx_flag), and the SX1262 hardware automatically triggers it each time a reception is completed, driving the data reception processing logic.

Sx1262_rx_init()

The function **Sx1262_rx_init()** in the **BSP_SX1262** class is used to initialize the SX1262 module for reception.

- The function first uses lora_hal to configure the SPI pins (RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI) and the SPI clock frequency (8 MHz), then calls spiBegin() to start SPI communication, providing a low-level interface for the SX1262.
- Next, it creates a Module object bsp_sx_mod and an SX1262 object bsp_sx_radio to encapsulate the hardware pins and transmission/reception interfaces. It then calls begin() to configure the LoRa parameters (915 MHz frequency, 125 kHz bandwidth, spreading factor 7, coding rate 4/7, sync word, 22 dBm power, etc.), completing module initialization. If initialization fails, an error is printed and the function returns a failure status.
- It then registers the reception completion callback via setPacketReceivedAction(set_sx1262_rx_flag), so that the module automatically sets lora_receivedFlag whenever a packet is received.
- The function calls setRxBoostedGainMode(true) to enable boosted gain mode for improved reception sensitivity, then calls startReceive() to start reception mode. If starting reception fails, it prints an error and returns failure.

This function is usually called once at system startup or before starting LoRa data reception to ensure the module is in a receivable state, after which received data can be processed via polling or callback.

```
esp_err_t BSP_SX1262::Sx1262_rx_init()
{
    lora_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI);
    lora_hal.setSpiFrequency(8000000);
    lora_hal.spiBegin();

    bsp_sx_mod = new Module(&lora_hal, SX1262_GPIO_NSS, SX1262_GPIO_IRQ, SX1262_GPIO_NRST, SX1262_GPIO_BUSY);
    bsp_sx_radio = new SX1262(bsp_sx_mod);
    int state = bsp_sx_radio->begin(915.0, 125.0, 7, 7, RADIOLIB_SX126X_SYNC_WORD_PRIVATE, 22, 8, 1.6);
    if (state != RADIOLIB_ERR_NONE)
    {
        SX1262_ERROR("radio rx init failed, code :%d", state);
        return ESP_FAIL;
    }
    bsp_sx_radio->setPacketReceivedAction(set_sx1262_rx_flag);
    bsp_sx_radio->setRxBoostedGainMode(true);
    state = bsp_sx_radio->startReceive();
    if (state != RADIOLIB_ERR_NONE)
    {
        SX1262_ERROR("radio start receive failed, code :%d", state);
        return ESP_FAIL;
    }
    return ESP_OK;
}
```

Here, we are initializing the **receiver module**. Similarly, by keeping the frequency band at **915 MHz**, the module can successfully receive the data sent from the transmitter.

Received_pack_radio:

The function Received_pack_radio(size_t len) in the BSP_SX1262 class is the core function for handling received LoRa data packets.

- The function first checks the reception flag lora_receivedFlag. If it is true, it indicates that a new data packet has arrived. The flag is then reset to false to prevent duplicate processing.
- It then obtains the actual length of the received data via bsp_sx_radio->getPacketLength(). If a valid length is returned, it is used; otherwise, the externally provided len serves as a fallback.
- Next, a buffer data[255] is defined, and bsp_sx_radio->readData() is called to read the received data into the buffer. If reading succeeds, the function prints the received data, RSSI (Received Signal Strength), SNR (Signal-to-Noise Ratio), and frequency offset. If a callback function rx_data_callback has been registered, it passes the data, length, and signal parameters to the upper-level application for processing.

This function is usually called periodically in the main loop or tasks. It executes after the SX1262 reception interrupt sets lora_receivedFlag, allowing the upper-level application to retrieve and process received packets promptly and reliably.

```
void BSP_SX1262::Received_pack_radio(size_t len)   // Function to process received LoRa data packets, parameter len indicates expected leng
{
  if (lora_receivedFlag)   // Check if the receive flag is set (indicating data was received)
  {
    lora_receivedFlag = false;   // Reset the receive flag to avoid repeated processing

    // Get the actual received data length
    size_t actual_len = bsp_sx_radio->getPacketLength();   // Get packet length from SX1262 radio module
    if (actual_len > 0) {   // If a valid packet length is returned
      lora_received_len = actual_len;   // Use the actual received length
    } else {
      lora_received_len = len; //  Use the passed-in length as a fallback
    }

    uint8_t data[255];   // Define a buffer to store the received data
    int state = bsp_sx_radio->readData(data, lora_received_len);   // Read data from the SX1262 module into the buffer
    if (state == RADIOLIB_ERR_NONE)   // If reading succeeded
    {
      SX1262_INFO("Received packet!");   // Log message: packet successfully received
      SX1262_INFO("Valid Data : %.*s", lora_received_len, (char *)data);   // Print the received data as a string
      SX1262_INFO("RSSI:%.2f dBm", bsp_sx_radio->getRSSI());   // Log the received signal strength (RSSI)
      SX1262_INFO("SNR:%.2f dB", bsp_sx_radio->getSNR());   // Log the signal-to-noise ratio (SNR)
      SX1262_INFO("Frequency error:%.2f", bsp_sx_radio->getFrequencyError());   // Log the frequency error information

      //  Call the callback function to notify the upper application
      if (rx_data_callback != NULL) {   // If the callback function has been registered
        rx_data_callback((const char*)data, lora_received_len, bsp_sx_radio->getRSSI(), bsp_sx_radio->getSNR());   // Pass received data, I
      }
    }
    else if (state == RADIOLIB_ERR_CRC_MISMATCH)   // If CRC verification failed (data corrupted)
    {
      SX1262_ERROR("CRC error!");   // Log an error message indicating CRC mismatch
    }
    else   // Other unexpected errors during data reading
    {
      SX1262_ERROR("radio receive failed, code :%d", state);   // Log the specific error code
    }
  }
}
```

sx1262_rx_init()

This is a C-style interface used to initialize the SX1262 module's reception function. Inside the function, a BSP_SX1262 object is created, and its member function Sx1262_rx_init() is called to complete SPI configuration, module initialization, parameter setup, registration of the reception callback, and starting reception mode. The function returns ESP_OK if initialization succeeds, or ESP_FAIL if it fails. This function is typically called once at system startup or before starting LoRa data reception to ensure the module is in a ready-to-receive state.

sx1262_rx_deinit()

This is a C-style de-initialization interface for the SX1262 reception function. Inside the function, a BSP_SX1262 object is created, and its method Sx1262_rx_deinit() is called to shut down the reception functionality. The de-initialization process includes clearing the reception callback, switching the module to standby mode, delaying to ensure safe shutdown, and closing the SPI interface. This function is generally called when the system is shutting down, the module no longer needs to receive data, or it enters low-power mode.

received_lora_pack_radio(size_t len)

This is a C-style interface used to handle received LoRa data packets. Inside the function, a BSP_SX1262 object is created, and its method Received_pack_radio(len) is called. The function processes the data by checking the reception flag, reading the data, printing logs, and invoking the upper-layer callback function.

This function is generally called periodically in the main loop or tasks and executes after lora_receivedFlag is set, ensuring that the upper-level application can timely retrieve and handle received data packets.

sx1262_set_rx_callback(void (*callback)(const char* data, size_t len, float rssi, float snr))

This function is used to register the upper-layer callback rx_data_callback. When the SX1262 module receives a data packet, this callback is automatically triggered, passing the data, length, RSSI, and SNR information to the upper-layer application. This function is typically called once after initializing the reception functionality to bind the data processing logic.

sx1262_get_received_len()

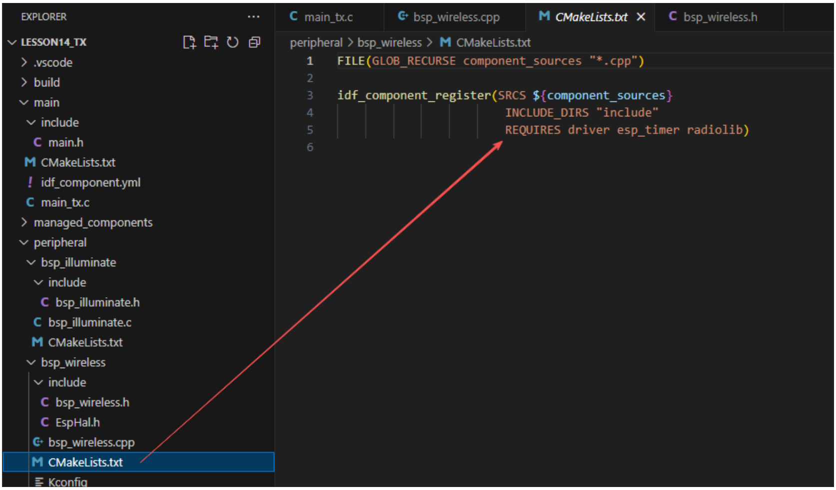
This is a query interface that returns the length of the most recently received data lora_received_len. Internally, the function simply returns the static variable without modifying any state. It is usually called when processing received data or performing debug/statistics, to obtain the actual length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

sx1262_is_data_received()

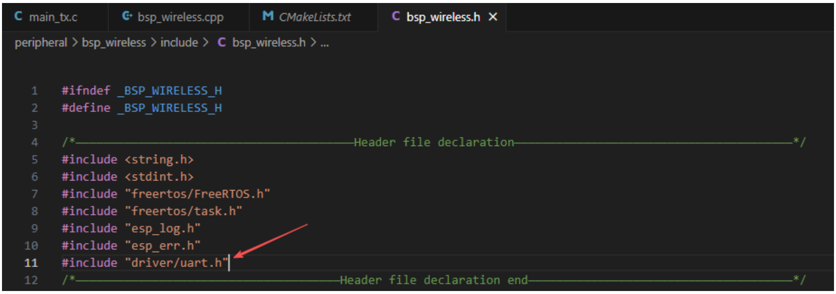
This is a status query interface that returns the reception flag lora_receivedFlag, used to determine whether a new data packet has arrived. The function simply returns the status of the global variable without modifying it. It is typically polled in the main loop or tasks to decide whether to call received_lora_pack_radio()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

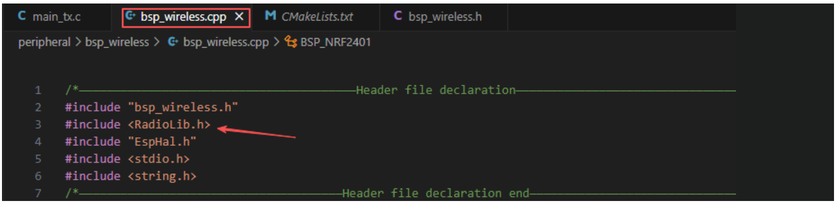
That concludes the introduction to the bsp_wireless component. You only need to know how to call these interfaces.

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.

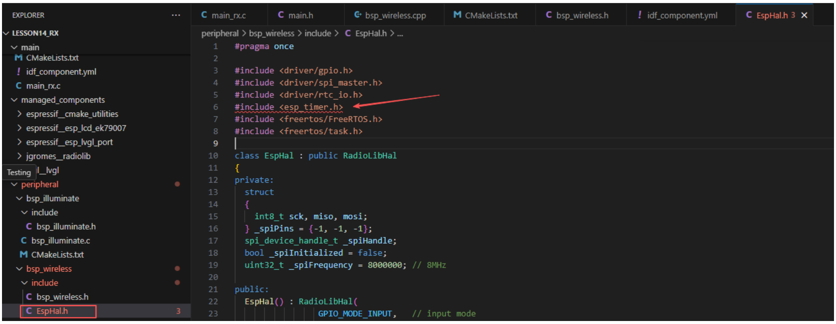


The reason driver, esp_timer, and RadioLib are included here is that we call them in bsp_wireless.h and bsp_wireless.cpp. Other libraries are system libraries and do not need to be explicitly added.





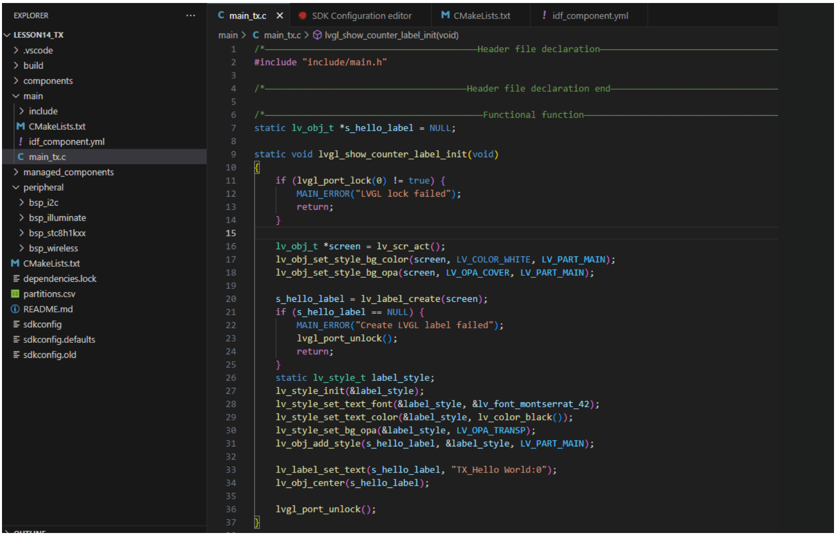
As well as the esp_timer used in the EspHal.h file.



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.



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.

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

After the system starts, app_main() first calls Hardware_Init() to initialize the hardware, including I2C,STC8 expansion chip, the LCD display and LVGL library, and the SX1262 LoRa transmission module, ensuring all hardware resources are ready.

- Then, lvgl_show_counter_label_init() is called to create an LVGL label for displaying the transmission count, centered on the screen. After initialization, the system enters the task scheduling stage.
- The system creates two FreeRTOS tasks:
    - ui_counter_task reads the SX1262 transmission counter every second, updates the display via LVGL, and prints logs.
    - lora_tx_task calls send_lora_pack_radio() every second to send LoRa data packets and prints error messages if transmission fails.
- The two tasks use vTaskDelayUntil() to ensure synchronized execution on a fixed 1-second cycle, enabling coordinated screen display and wireless transmission, achieving the complete process of sending LoRa messages every second and dynamically showing "TX_Hello World:count" on the screen.

Next, let's go through the main code in main_tx.c.

```
/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include "main.h"   // Include the main header file containing required definitions and declarations
/*———————————————————————————————————Header file declaration end———————————————————————————————————*/
```

It includes the custom main header file main.h, which typically contains log macros, peripheral initialization declarations, and headers for other interfaces that need to be used.

Below is the content of main.h:

C main_tx.c | C main.h × | bsp_wireless.cpp | M CMakeLists.txt | C bsp_wireless.h

main > include > C main.h

```
#ifndef _MAIN_H_

/*————————————————————————————————————Header file declaration————————————————————————————————————*/
#include <stdio.h>
#include "string.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "esp_private/esp_clk.h"
#include "esp_ldo_regulator.h"
#include "esp_sleep.h"
#include "driver/rtc_io.h"
#include "driver/gpio.h"
#include "esp_timer.h"

#include "bsp_wireless.h"
#include "bsp_illuminate.h"
#include "lvgl.h"
/*———————————————————————————————————Header file declaration end———————————————————————————————————*/
```

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

lvgl_show_counter_label_init:

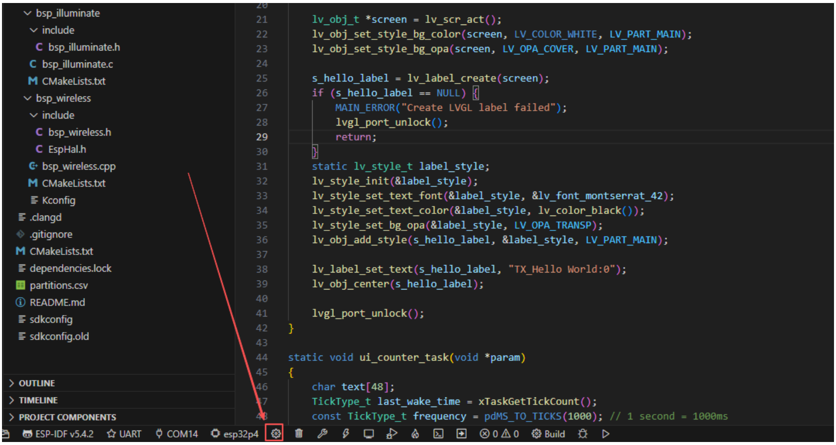
The function **lvgl_show_counter_label_init()** initializes the **counter label** in the LVGL display interface, used to show the **LoRa transmission count.**

- 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.

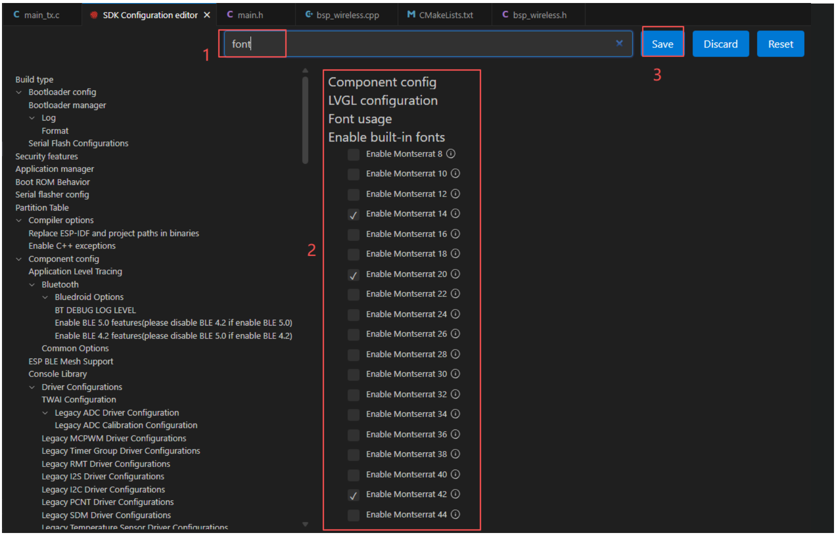
If you want to change the LVGL font size, you need to go into the SDK configuration and enable the desired font.

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, initialize the I2C and the STC8H1K17 expansion chip controlled by I2C, so as to control the subsequent screen backlight and make the screen light up.
- 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 r**eliable 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 I2C, STC8 expansion chip, 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.

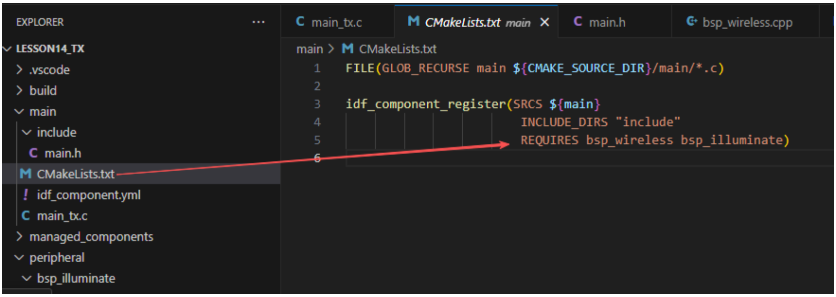
```
void app_main(void)
{
    MAIN_INFO("---------- LoRa TX ----------");
    Hardware_Init();

    lvgl_show_counter_label_init();
    MAIN_INFO("-------- LVGL Show OK ----------");

    // Create tasks and use the same priority to ensure synchronization
    xTaskCreatePinnedToCore(ui_counter_task, "ui_counter", 4096, NULL,
                            configMAX_PRIORITIES - 5, NULL, 0);

    xTaskCreatePinnedToCore(lora_tx_task, "sx1262_tx", 8192, NULL,
                            configMAX_PRIORITIES - 5, NULL, 1);

    MAIN_INFO("Tasks created, starting synchronized transmission...");
}
/*---------------------------------Functional function end---------------------------------*/
```

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.



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:

- 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.

```
/*----------------------------------------Variable declaration----------------------------------------*/
// Define global static variables used throughout the file

static lv_obj_t *s_rx_label = NULL;          // LVGL label object to display received data
static lv_obj_t *s_rssi_label = NULL;        // LVGL label object to display RSSI value
static lv_obj_t *s_snr_label = NULL;         // LVGL label object to display SNR value
static uint32_t rx_packet_count = 0;         // Counter for the number of received LoRa packets
/*--------------------------------------Variable declaration end--------------------------------------*/
```

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.

- 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.
- 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.

- 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.
- 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.
- 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:

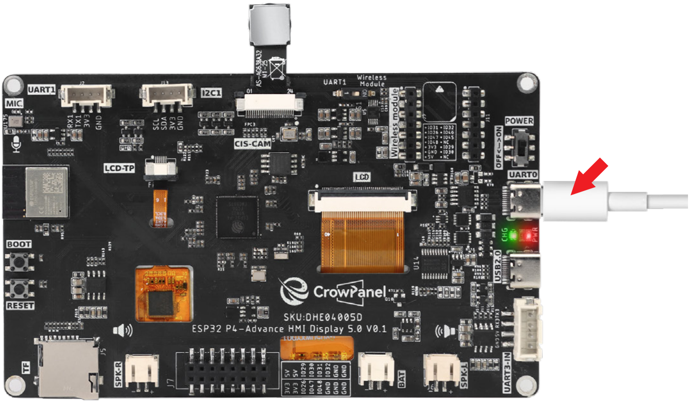
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14_TX_SX1262_Wireless_Module

- Receiving end code:

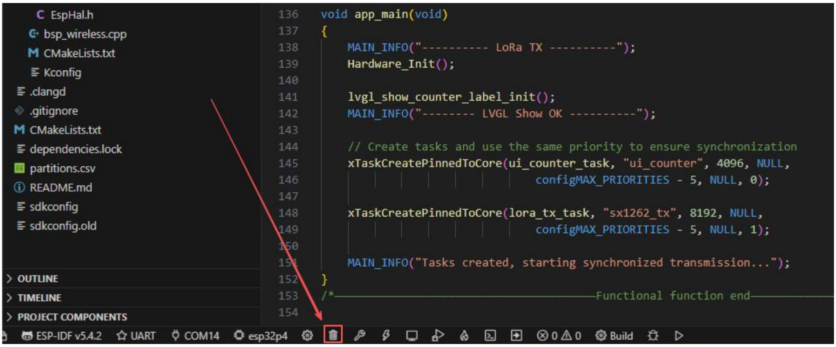
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson14_RX_SX1262_Wireless_Module

Programming Steps

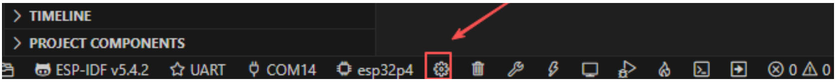
- Now that the code is ready, the next step is to **flash it onto the ESP32-P4** so we can observe the actual operation.
- First, connect the Advance-P4 device to your computer using a USB cable.



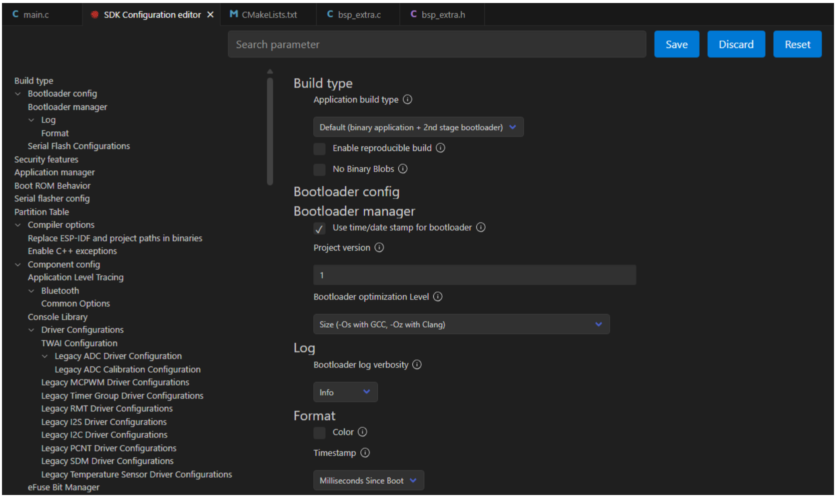
- Before starting the flashing preparation, delete all files generated during compilation to restore the project to its initial "unbuilt" state. This ensures that subsequent compilations are not affected by your previous build results.



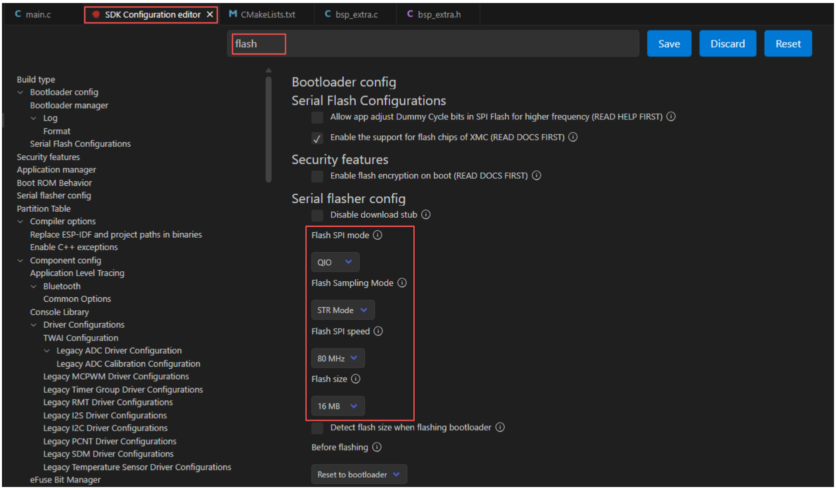
- Next, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip correctly.
- Then, we need to configure the SDK.
- Click on the icon shown in the figure below.



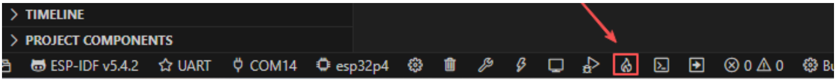
- Wait for a short loading period, and then you can proceed with the relevant SDK configuration.



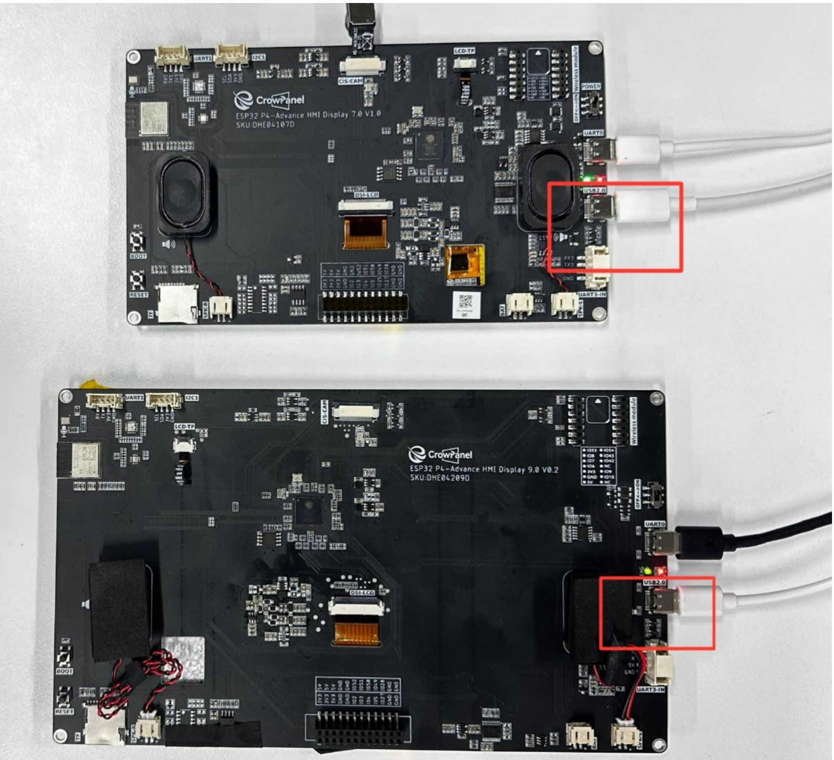
- Then, type "flash" into the search box. (Ensure your flash configuration matches mine.)



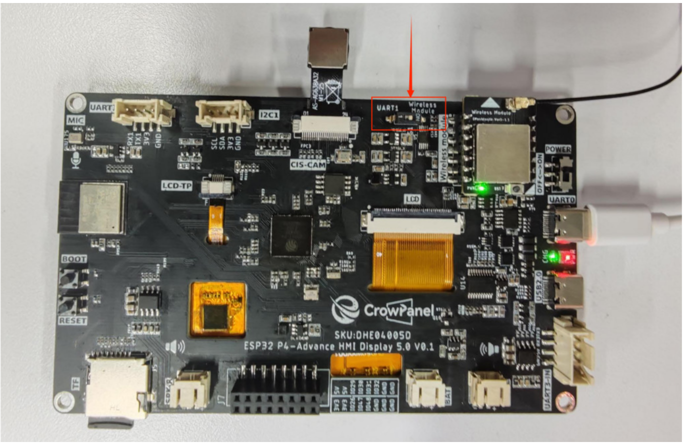
- After completing the configuration, remember to save your settings.
- Next, we will compile and flash the code (detailed in the first lesson).
- Here, we'd like to share a very convenient feature with you: a single button can execute **compilation, uploading, and serial monitor opening** in one go.



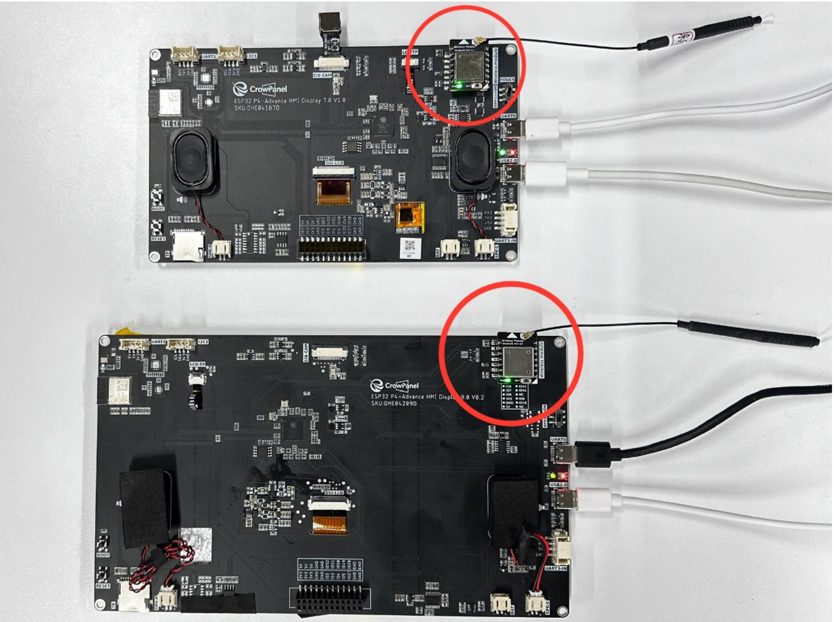
- After waiting for a moment, the code will finish compiling and uploading, and the serial monitor will open automatically.
- At this point, remember to connect your Advance-P4 using an additional Type-C cable via the **USB 2.0 interface.** This is because the maximum current provided by a computer's 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.)



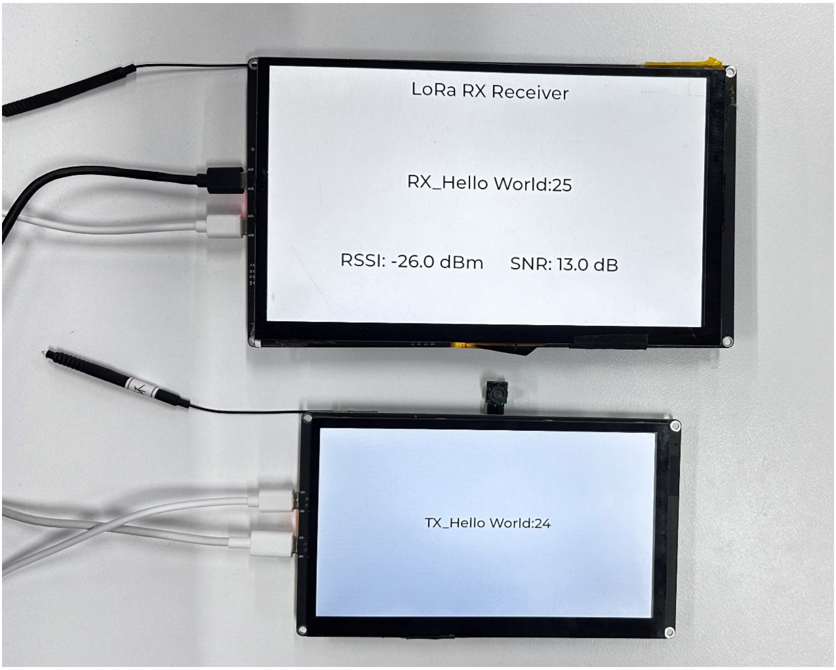
For the 5-inch Advance-P4 product, the jumpers need to be switched on the hardware in order to use the wireless module. (Switch to the side with the wireless module)



- 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

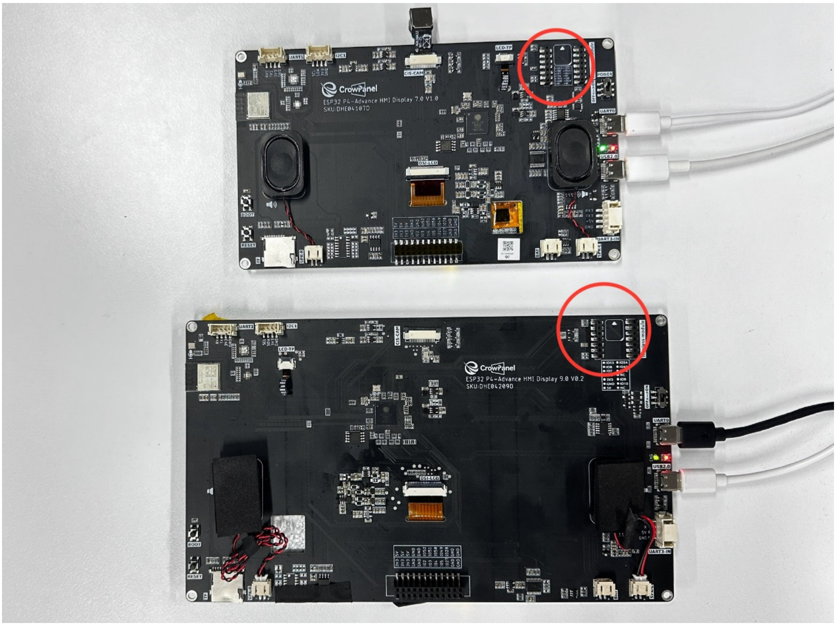
Introduction

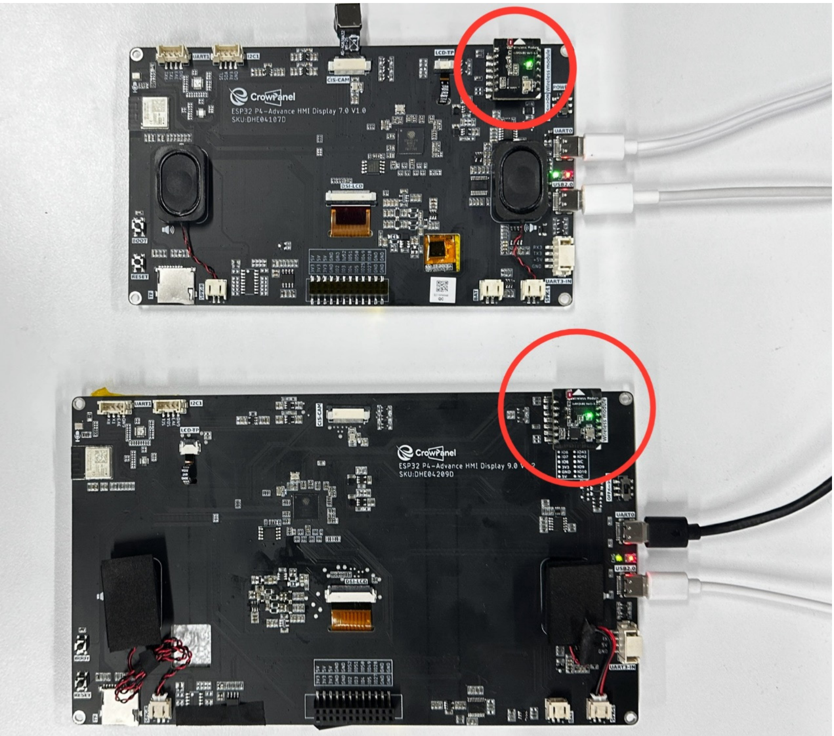
In this lesson, we will start using another wireless module. Since we will implement the transmission and reception functions of the nRF2401 module, two Advance-P4 development boards and two nRF2401 wireless RF (Radio Frequency) communication modules are required.

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

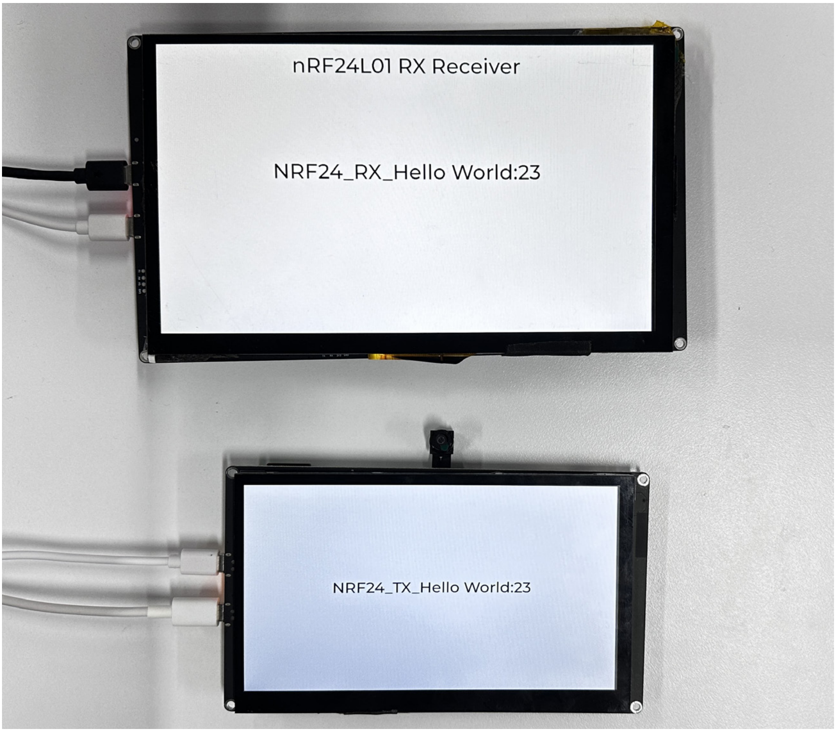




Operation Effect Diagram

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.

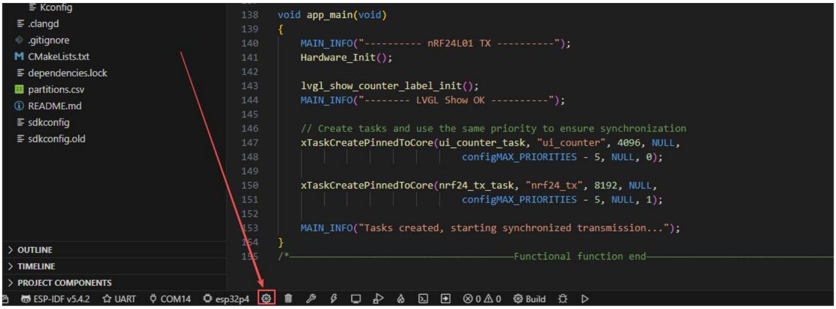


Key Explanations

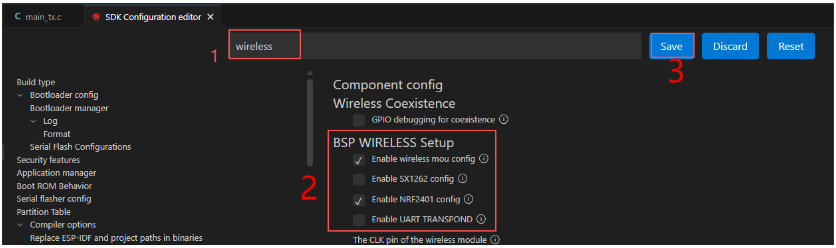
- The focus of this lesson is on how to use the wireless module, including initializing the nRF2401 module and sending or receiving information.
- Here, we will still use the bsp_wireless component from the previous lesson.
- The main functions of this component are as follows:
  - It is responsible 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:

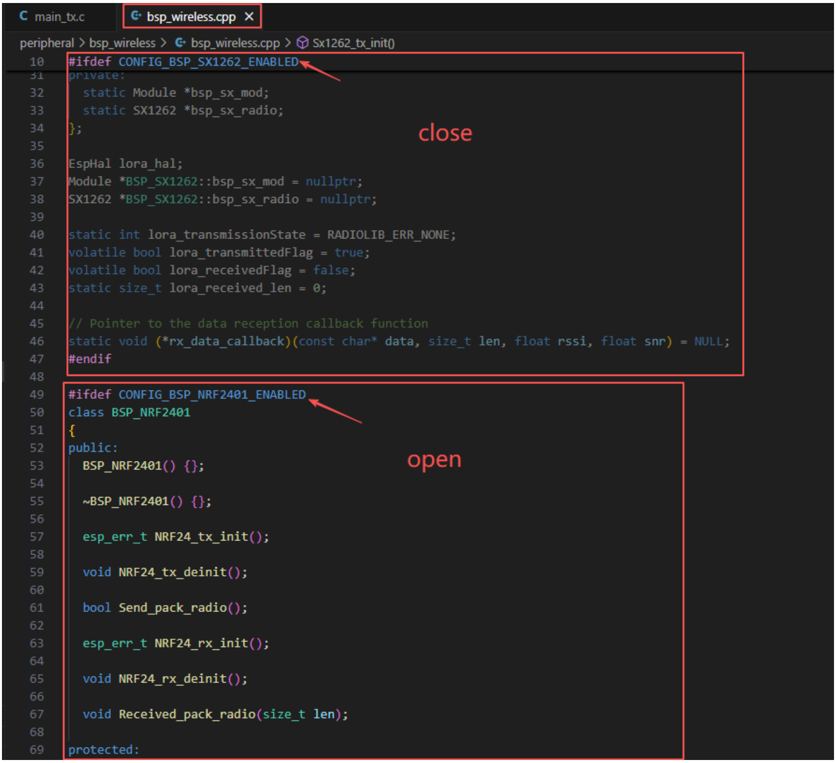
- **Click on the SDK configuration.**



- 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.



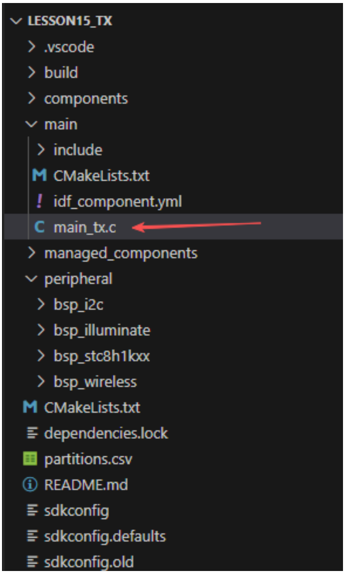
- As shown in the figure, we have enabled the nRF2401 configuration, so the other modules are temporarily disabled and not applicable.
- In the bsp_wireless component, you only need to call the prepared interfaces when needed.
- Next, let's focus on understanding the bsp_wireless component.
- First, click the GitHub link below to download the code for this lesson.
- Transmitting end code:

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15_TX_nRF2401_Wireless_RF_Module

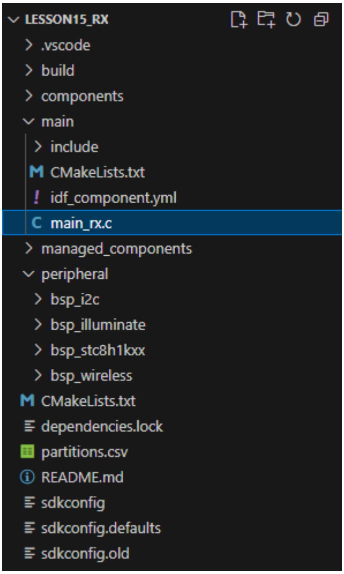
- Receiving end code:

https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-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.

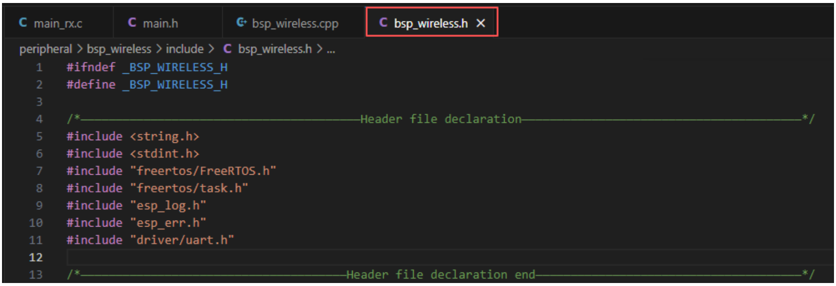
The CMakeLists.txt file integrates the driver into the build system, allowing the project to use the nRF2401 module send and receive functions implemented in bsp_wireless.cpp.

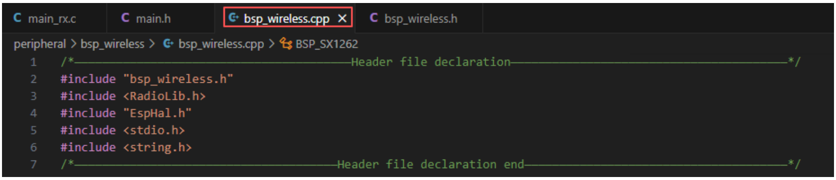
Additionally, there is **bsp_illuminate**, our familiar component used to light up the screen and draw text via LVGL.

And the "bsp_stc8h1kxx" component can be used to control the screen backlight.

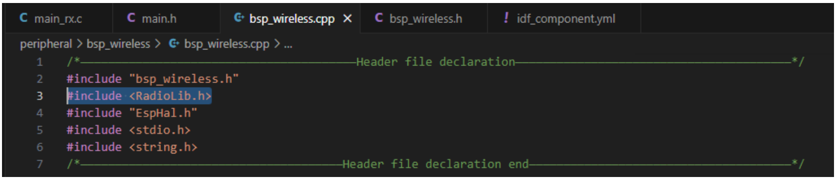
nRF2401 Communication Code

- 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.

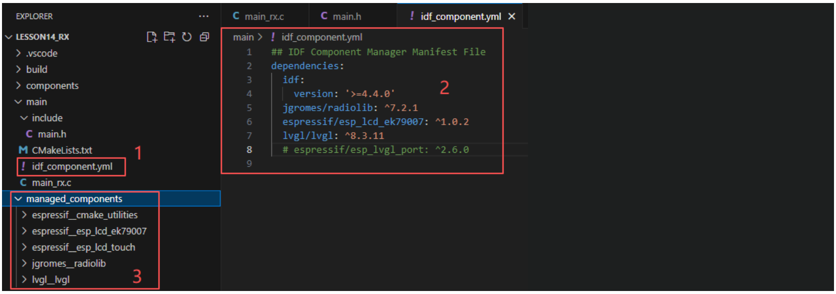




- Since the function implementations in bsp_wireless.cpp use the function wrappers provided in EspHal.h, the header file needs to be included in the .cpp file.
- For example, #include <RadioLib.h> (this is a library under the networking components)



- This inclusion requires us to specify the version of jgromes/radiolib in the idf_component.yml file located in the main folder. Because this is an official library, we rely on it to implement the wireless transmission or reception functionality of the nRF2401 on the Advance-P4.



- 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.

```
main > ! idf_component.yml
## IDF Component Manager Manifest File
dependencies:
  idf:
    version: '>=4.4.0'
  jgromes/radiolib: ^7.2.1
  espressif/esp_lcd_ek79007: ^1.0.2
  lvgl/lvgl: ^8.3.11
  # espressif/esp_lvgl_port: ^2.6.0
```

- 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 and lvgl version 8.3.11.

- Once downloaded, these networking components will be stored in the managed_components folder. (This is automatically generated after specifying the version numbers.)

- It can be seen that the esp_lvgl_port has been commented out by me. This is because in the components, we downloaded this network component and then made relevant modifications based on it to adapt it to our 5-inch Advance-P4 product.

```
EXPLORER
LESSON14_RX
  > .vscode
  > build
  v components
    > espressif__esp_lcd_touch_gt911
    > espressif__esp_lvgl_port
  v main
    v include

main_rx.c    main.h    idf_component.yml

main > ! idf_component.yml
## IDF Component Manager Manifest File
dependencies:
  idf:
    version: '>=4.4.0'
  jgromes/radiolib: ^7.2.1
  espressif/esp_lcd_ek79007: ^1.0.2
  lvgl/lvgl: ^8.3.11
  # espressif/esp_lvgl_port: ^2.6.0
```

- Returning to bsp_wireless.h, this is where we declare the pins used by the wireless module.

```
#define RADIO_GPIO_CLK 26
#define RADIO_GPIO_MISO 47
#define RADIO_GPIO_MOSI 48
```

```
#ifdef CONFIG_BSP_NRF2401_ENABLED

#define NRF24_GPIO_IRQ 29
#define NRF24_GPIO_CE 31
#define NRF24_GPIO_CS 32
```

- The pin definitions should not be modified; otherwise, the wireless module will not function correctly due to incorrect wiring.

- Next, we declare the variables we need to use, as well as the functions. The actual implementations of these functions are in bsp_wireless.cpp. Placing all declarations in bsp_wireless.h is intended to make them easier to call and manage. (We will examine their specific functionality when they are used in bsp_wireless.cpp.)

```
//---------------------------------------------------------------------------

#ifdef CONFIG_BSP_NRF2401_ENABLED

#define NRF24_GPIO_IRQ 29
#define NRF24_GPIO_CE 31
#define NRF24_GPIO_CS 32

#ifdef __cplusplus
extern "C"
{
#endif
    esp_err_t nrf24_tx_init();
    void nrf24_tx_deinit();
    bool send_nrf24_pack_radio();
    uint32_t nrf24_get_tx_counter();

    esp_err_t nrf24_rx_init();
    void nrf24_rx_deinit();
    void received_nrf24_pack_radio(size_t len);
    void nrf24_set_rx_callback(void (*callback)(const char* data, size_t len));
#ifdef __cplusplus
}
#endif
#endif
//---------------------------------------------------------------------------
```

- Next, let's take a look at the specific functionality of each function in bsp_wireless.cpp.
- In the bsp_wireless component, BSP_NRF2401 is a BSP driver wrapper class for the nRF24L01 wireless transceiver module. It provides initialization, execution, de-initialization, and callback mechanisms for sending and receiving.
- This allows the application layer to complete wireless communication simply by calling straightforward C interface functions (such as nrf24_tx_init() or send_nrf24_pack_radio()), without needing to directly manipulate the underlying SPI registers or the RadioLib interface.
- Here, we won't go into a detailed code walkthrough; we will only explain the purpose of each function and the situations in which it should be called.

BSP_NRF2401 Class:

This means:

This code defines a class named BSP_NRF2401 to encapsulate the driver logic for the nRF2401 wireless transceiver module, implementing initialization, sending, and receiving functionalities for wireless communication.

- The class declares initialization and de-initialization functions for both the transmitter and receiver (such as NRF24_tx_init, NRF24_rx_init), as well as data sending and receiving handling functions (Send_pack_radio, Received_pack_radio).
- Two static pointers, bsp_nrf_mod and bsp_nrf_radio, are defined to point to the underlying hardware module object and the radio object, respectively, allowing global sharing.

- nrf_hal is the hardware abstraction layer object, used to manage hardware communication with the chip.
- Two volatile variables are defined: radio24_transmittedFlag indicates whether transmission is complete, and radio24_receivedFlag indicates whether reception is complete.
- nrf24_tx_counter is used to record the number of transmissions.
- Finally, a function pointer nrf24_rx_data_callback is defined to trigger an upper-layer callback when data is received.

Overall, this code establishes the basic control framework for the nRF2401 module, providing a unified interface and state management mechanism for subsequent wireless data transmission and reception.

```
#ifdef CONFIG_BSP_NRF2401_ENABLED
class BSP_NRF2401
{
public:
    BSP_NRF2401() {};

    ~BSP_NRF2401() {};

    esp_err_t NRF24_tx_init();

    void NRF24_tx_deinit();

    bool Send_pack_radio();

    esp_err_t NRF24_rx_init();

    void NRF24_rx_deinit();

    void Received_pack_radio(size_t len);

protected:
private:
    static Module *bsp_nrf_mod;
    static nRF24 *bsp_nrf_radio;
};

EspHal nrf_hal;
Module *BSP_NRF2401::bsp_nrf_mod = nullptr;
nRF24 *BSP_NRF2401::bsp_nrf_radio = nullptr;

volatile bool radio24_transmittedFlag = true;
volatile bool radio24_receivedFlag = false;
static uint32_t nrf24_tx_counter = 0;

// Pointer to the data reception callback function for nRF24L01
static void (*nrf24_rx_data_callback)(const char* data, size_t len) = NULL;
#endif
```

NRF24_tx_init:

Initializes the transmitter of the nRF2401 module by configuring the SPI interface, creating the communication object, setting the wireless parameters, and specifying the transmission channel, enabling the module to send data.

- At the beginning of the function, nrf_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI) sets the SPI communication pins between the nRF2401 and the main controller (Clock, Master In Slave Out, Master Out Slave In).
- setSpiFrequency(8000000) sets the SPI clock frequency to 8 MHz to improve communication speed.
- spiBegin() formally initializes the SPI bus.
- A module object bsp_nrf_mod is then created via new Module(...), binding the SPI interface along with control pins such as Chip Select (CS), Interrupt (IRQ), and Chip Enable (CE), providing a hardware interface for the nRF24 module.
- Next, bsp_nrf_radio = new nRF24(bsp_nrf_mod) creates the specific nRF24 radio object and begins the driver logic.
- Calling begin(2400, 250, 0, 5) completes the core initialization of the wireless module. The parameters represent, in order: operating frequency 2400 MHz (i.e., 2.4 GHz band), data rate 250 kbps, output power level 0 (typically 0 dBm), and communication channel number 5. If initialization fails (return value is not RADIOLIB_ERR_NONE), the error is logged and the function exits.
- Then, a transmit address is defined as uint8_t addr[] = {0x01, 0x02, 0x11, 0x12, 0xFF}, which is a 5-byte transmit pipe address (similar to a "device address" or "channel identifier" in wireless communication), ensuring that the transmitter and receiver communicate over the same address.
- setTransmitPipe(addr) sets this address as the current transmit pipe, allowing the module to send data through this channel. If configured successfully, the function returns ESP_OK, indicating that initialization is complete.

Send_pack_radio:

This function sends a wireless data packet through the nRF2401 module and records and prints the transmission status.

- Specifically, the function first defines a static character array text[32] to store the message to be sent. It then uses snprintf to format the message as "NRF24_TX_Hello World:<transmit_count>", where <transmit_count> comes from nrf24_tx_counter and represents the current number of transmissions.
- The function calculates the message length using strlen and stores it in tx_len.

- Next, it calls bsp_nrf_radio->transmit((uint8_t *)text, tx_len, 0) to send the message through the nRF2401 module. If the return value is RADIOLIB_ERR_NONE, the transmission is successful, and NRF2401_INFO prints the completion message along with the content sent. Otherwise, it prints a transmission failure message and the error code.
- The function finally returns true, indicating that the send operation has been executed.

nrf24_tx_init():

This is a C-language interface function used to initialize the nRF2401 transmitter module. Inside the function, a BSP_NRF2401 object obj is instantiated, and its member function NRF24_tx_init() is called to complete SPI configuration, wireless parameter setup, and transmit pipe address configuration, returning the initialization result.

Purpose: Provides a unified interface for upper-layer or C code to prepare the nRF2401 module for data transmission.

nrf24_tx_deinit():

This is a C-language interface function used to release or shut down the nRF2401 transmitter resources. It creates a BSP_NRF2401 object internally and calls its member function NRF24_tx_deinit(), putting the wireless module into an idle state and closing the SPI bus.

Purpose: Called when the transmission task is finished or the module is no longer in use, safely releasing transmitter resources.

send_nrf24_pack_radio():

This is a C-language interface function used to send a data packet via the nRF2401. Internally, it creates a BSP_NRF2401 object and calls its member function Send_pack_radio() to send the formatted message and print the transmission result.

Purpose: Provides a simple interface for the upper layer to send wireless data without needing to handle the underlying driver details.

nrf24_get_tx_counter():

This is a C-language interface function used to get the current value of the nRF2401 transmit counter nrf24_tx_counter.

Purpose: Allows upper-layer programs to obtain the number of packets sent, useful for statistics or debugging.

nrf24_inc_tx_counter():

This is a C-language interface function used to increment the transmit counter nrf24_tx_counter by 1.

Purpose: Updates the counter after each successful packet transmission, used to record the number of sends or to mark a sequence number in the message.

set_rx_flag():

This is a static internal function called within the receive interrupt or callback, used to set radio24_receivedFlag to true, indicating that the nRF2401 module has received new data.

Purpose: Serves as a receive event flag to notify the upper-layer program that new data is available for processing.

NRF24_rx_init:

This function, **BSP_NRF2401::NRF24_rx_init()**, initializes the receiver side of the nRF2401 module, enabling it to receive wireless data.

- Specifically, the function first sets the SPI communication pins using nrf_hal.setSpiPins(RADIO_GPIO_CLK, RADIO_GPIO_MISO, RADIO_GPIO_MOSI), sets the SPI clock frequency to 8 MHz with setSpiFrequency(8000000), and initializes the SPI bus using spiBegin().
- A module object bsp_nrf_mod is then created via new Module(...), binding the SPI interface and control pins. Next, bsp_nrf_radio = new nRF24(bsp_nrf_mod) creates the nRF24 radio object.
- Calling bsp_nrf_radio->begin(2400, 250, 0, 5) initializes the wireless parameters, where 2400 represents the 2.4 GHz operating frequency, 250 is the data rate in kbps, 0 is the output power level, and 5 is the communication channel. If an error occurs, it logs the failure and returns.
- A receive pipe address is defined as addr[] = {0x01, 0x02, 0x11, 0x12, 0xFF}. The function then calls setReceivePipe(0, addr) to set pipe 0 as the receive address, ensuring the module only receives data sent to this address.
- setPacketReceivedAction(set_rx_flag) registers a receive callback, setting radio24_receivedFlag to notify the upper layer. Finally, startReceive() puts the module into receive mode. If successful, the function returns ESP_OK.

Received_pack_radio:

This function, **BSP_NRF2401::Received_pack_radio(size_t len)**, handles data packets received by the nRF2401 module.

- Specifically, the function first checks the receive flag radio24_receivedFlag. If it is true, it indicates that new data has arrived. The flag is then reset to false to avoid repeated processing.
- A buffer data[len] is defined to store the received data, and bsp_nrf_radio->readData(data, len) is called to read len bytes from the module.
- If the return value is RADIOLIB_ERR_NONE, the data is successfully read. The function uses NRF2401_INFO to print a success message along with the received data, and checks whether the callback function pointer nrf24_rx_data_callback has been registered. If it is registered, the callback is called to notify the upper-layer application.
- If reading fails, NRF2401_ERROR prints the error code. Finally, bsp_nrf_radio->startReceive() is called to re-enter receive mode, waiting for the next data packet.

nrf24_rx_init()

This is a C-language interface function used to initialize the receiver side of the nRF2401 module. Internally, a BSP_NRF2401 object obj is instantiated, and its member function NRF24_rx_init() is called to complete SPI configuration, wireless parameter initialization, receive pipe address setup, and callback registration, returning the initialization result.

Purpose: Provides a unified interface for upper-layer or C-language programs to prepare the nRF2401 module for data reception.

nrf24_rx_deinit()

This is a C-language interface function used to release the nRF2401 receiver resources. Internally, a BSP_NRF2401 object is created, and its member function NRF24_rx_deinit() is called to put the module into an idle state, clear callbacks, and close the SPI bus.

Purpose: Called when the reception task is finished or the module is no longer in use, safely releasing receiver resources.

received_nrf24_pack_radio(size_t len)

This is a C-language interface function used to handle received data packets. Internally, it creates a BSP_NRF2401 object and calls its member function Received_pack_radio(len) to read the data, log the results, and notify the upper-layer application via a callback.

Purpose: Provides an upper-layer interface to trigger the nRF2401 data reception processing flow.

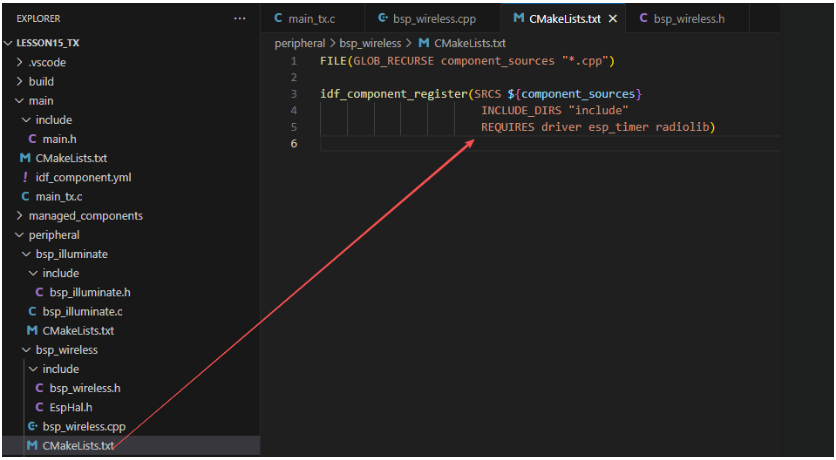
nrf24_set_rx_callback(void (*callback)(const char* data, size_t len))

This is a C-language interface function used to register a callback for received data, notifying the upper-layer application when data arrives. Internally, the passed function pointer is saved to nrf24_rx_data_callback.

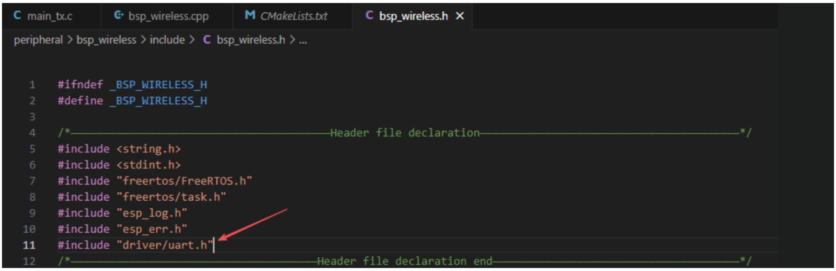
Purpose: Allows the upper-layer program to set a custom callback for immediate processing or response upon receiving nRF2401 data.

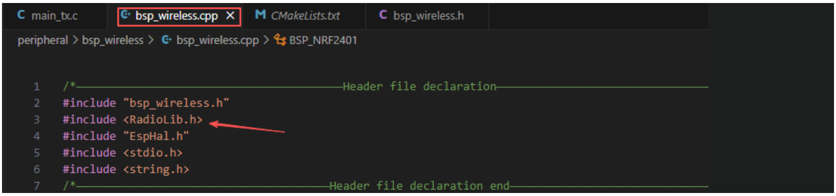
We will conclude the introduction of the bsp_wireless component here. It is enough for everyone to understand how to call these interfaces.

If you want to use 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.

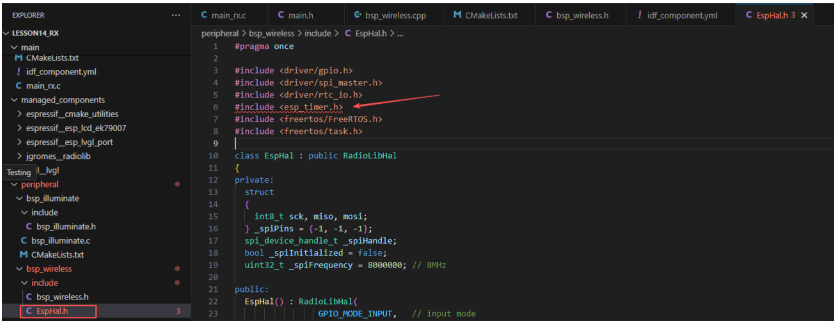


The reason only **driver**, **esp_timer**, and **radiolib** are listed is that we use them in bsp_wireless.h and bsp_wireless.cpp. (Other libraries are system libraries, so they do not need to be added.)





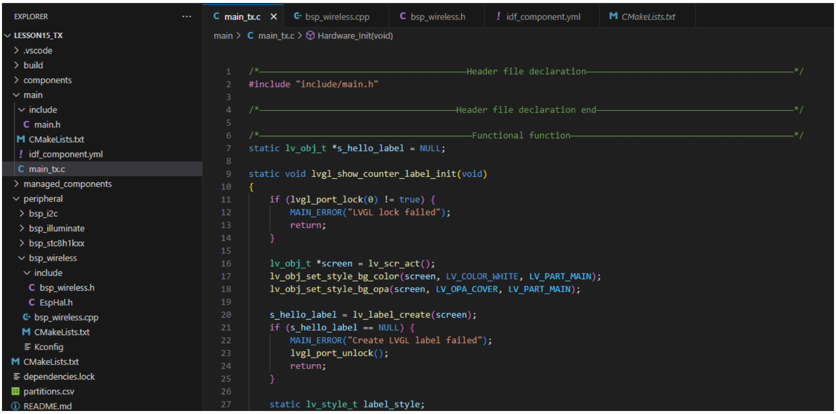
As well as **esp_timer**, which is used in the EspHal.h file.



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.



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 nRF2401.

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 I2C,STC8 expansion chip, 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.

```
/*——————————————————————————————Header file declaration——————————————————————————————*/
#include "main.h"   // Include the main header file containing required definitions and declarations
/*——————————————————————————————Header file declaration end——————————————————————————————*/
```

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**:

```
main_tx.c | main.h | bsp_wireless.cpp | CMakeLists.txt | bsp_wireless.h
main > include > main.h
#ifndef _MAIN_H_

/*——————————————————————————————Header file declaration——————————————————————————————*/
#include <stdio.h>
#include "string.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_err.h"
#include "esp_private/esp_clk.h"
#include "esp_ldo_regulator.h"
#include "esp_sleep.h"
#include "driver/rtc_io.h"
#include "driver/gpio.h"
#include "esp_timer.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:

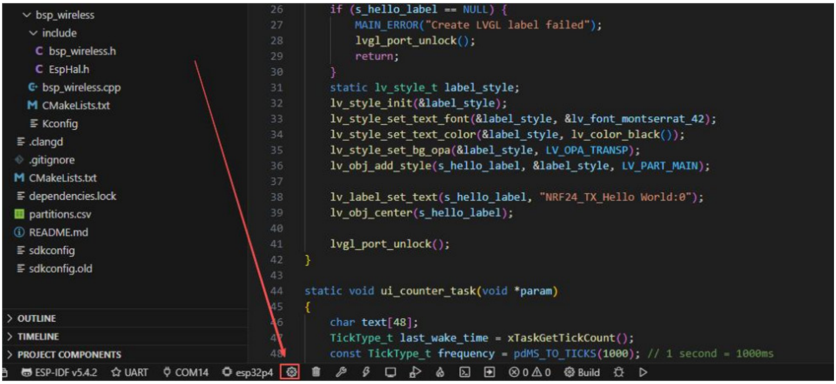
- First, lvgl_port_lock(0) is called to lock LVGL resources, preventing concurrent access.
- The current active screen is obtained via lv_scr_act(), and the background is set to white and fully covering.
- A label is created using lv_label_create(screen) and checked for successful creation; if creation 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.

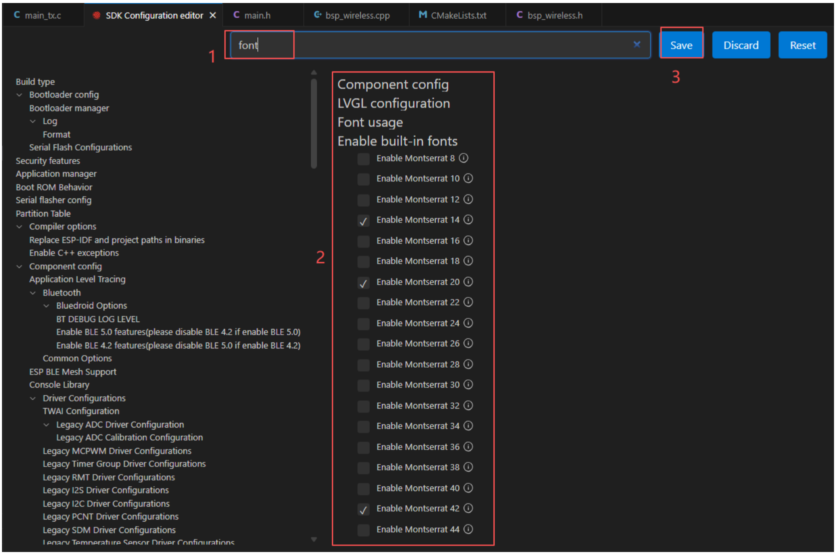
```c
static void lvgl_show_counter_label_init(void)
{
    if (lvgl_port_lock(0) != true) {
        MAIN_ERROR("LVGL lock failed");
        return;
    }

    lv_obj_t *screen = lv_scr_act();
    lv_obj_set_style_bg_color(screen, LV_COLOR_WHITE, LV_PART_MAIN);
    lv_obj_set_style_bg_opa(screen, LV_OPA_COVER, LV_PART_MAIN);

    s_hello_label = lv_label_create(screen);
    if (s_hello_label == NULL) {
        MAIN_ERROR("Create LVGL label failed");
        lvgl_port_unlock();
        return;
    }
    static lv_style_t label_style;
    lv_style_init(&label_style);
    lv_style_set_text_font(&label_style, &lv_font_montserrat_42);
    lv_style_set_text_color(&label_style, lv_color_black());
    lv_style_set_bg_opa(&label_style, LV_OPA_TRANSP);
    lv_obj_add_style(s_hello_label, &label_style, LV_PART_MAIN);

    lv_label_set_text(s_hello_label, "NRF24_TX_Hello World:0");
    lv_obj_center(s_hello_label);

    lvgl_port_unlock();
}
```

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 snprintf.
- Attempt to lock the LVGL resource with lvgl_port_lock(0). If successful and the label exists, call lv_label_set_text to update the display text and release the lock.
- Finally, use vTaskDelayUntil to delay according to absolute time to ensure an accurate one-second cycle, realizing the function of updating the display every second.

Overall, its role is to continuously refresh the transmission count on the interface to achieve real-time display.

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, initialize the I2C and the STC8H1K17 expansion chip controlled by I2C, so as to control the subsequent screen backlight and make the screen light up.
- 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 I2C,STC8 expansion chip, 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.

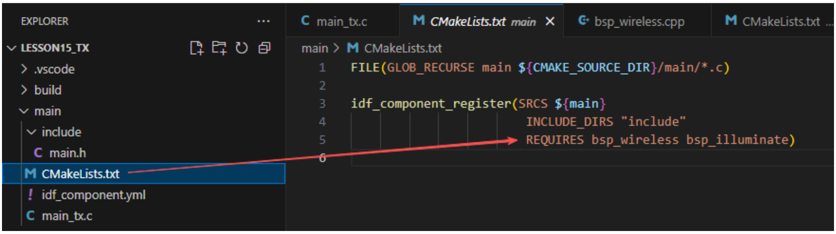
```c
void app_main(void)
{
    MAIN_INFO("---------- nRF24L01 TX ----------");
    Hardware_Init();

    lvgl_show_counter_label_init();
    MAIN_INFO("-------- LVGL Show OK ----------");

    // Create tasks and use the same priority to ensure synchronization
    xTaskCreatePinnedToCore(ui_counter_task, "ui_counter", 4096, NULL,
                            configMAX_PRIORITIES - 5, NULL, 0);

    xTaskCreatePinnedToCore(nrf24_tx_task, "nrf24_tx", 8192, NULL,
                            configMAX_PRIORITIES - 5, NULL, 1);

    MAIN_INFO("Tasks created, starting synchronized transmission...");
}
/*------------------------------------Functional function end------------------------------------*/
```

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.



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 initializesI2C,STC8 expansion chip, 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:

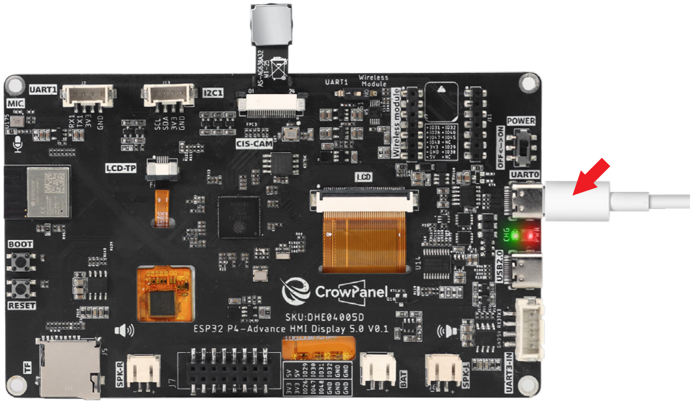
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15_TX_nRF2401_Wireless_RF_Module

- Receiving end code:

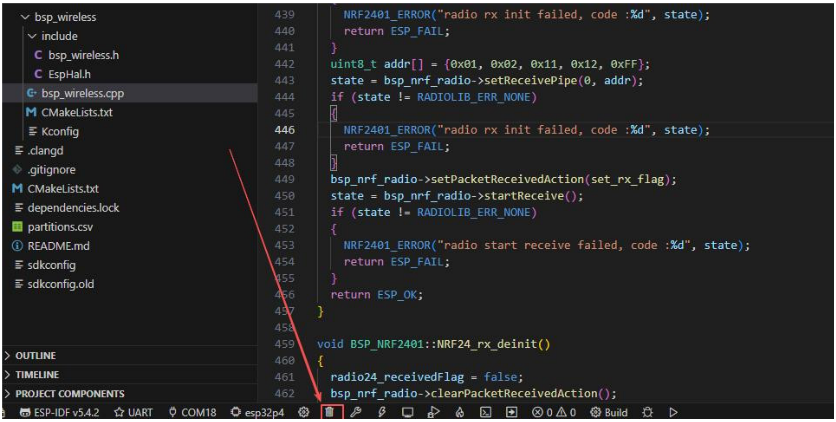
https://github.com/Elecrow-RD/-CrowPanel-Advanced-5inch-ESP32-P4-HMI-AI-Display-800x480-IPS-Touch-Screen/tree/master/example/V1.0/idf-code/Lesson15_RX_nRF2401_Wireless_RF_Module

Programming Steps

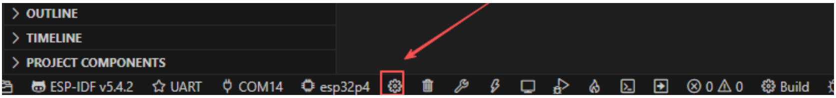
- Now that the code is ready, the next step is to flash it onto the ESP32-P4 so we can observe the actual operation.
- First, connect the Advance-P4 device to your computer using a USB cable.



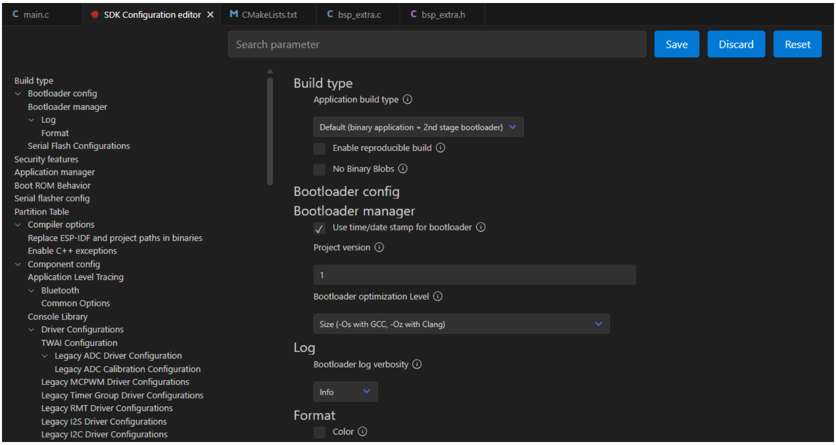
- Before starting the preparation for flashing, first delete all compiler-generated files to restore the project to its initial "unbuilt" state. This ensures that subsequent compilations are not affected by your previous operations.



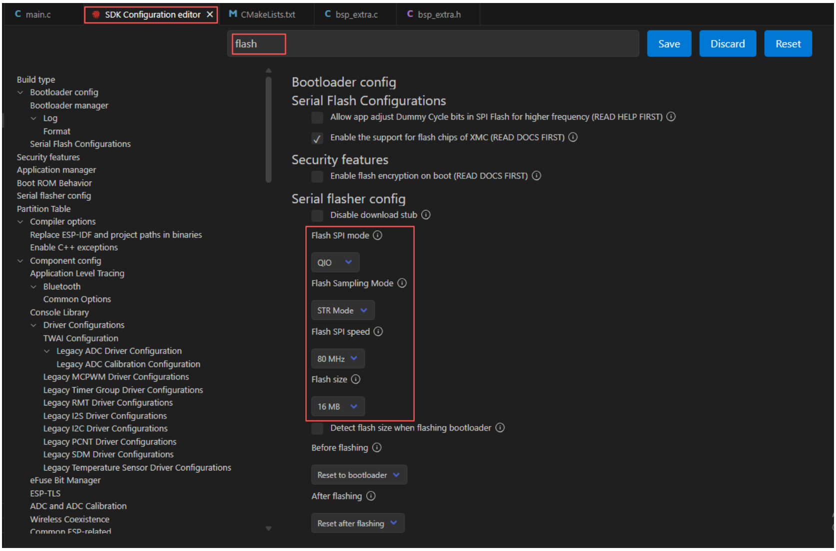
- Here, follow the steps from the first section to select the ESP-IDF version, code upload method, serial port number, and target chip.
- Next, we need to configure the SDK.
- Click the icon shown in the figure below.



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



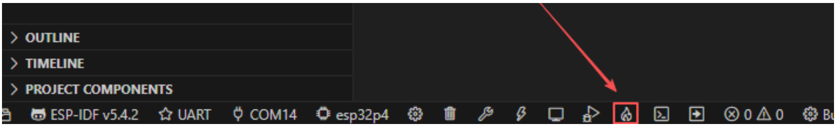
- Then, enter "flash" in the search box to search. (Make sure your flash configuration matches mine.)



- After completing the configuration, remember to save your settings.

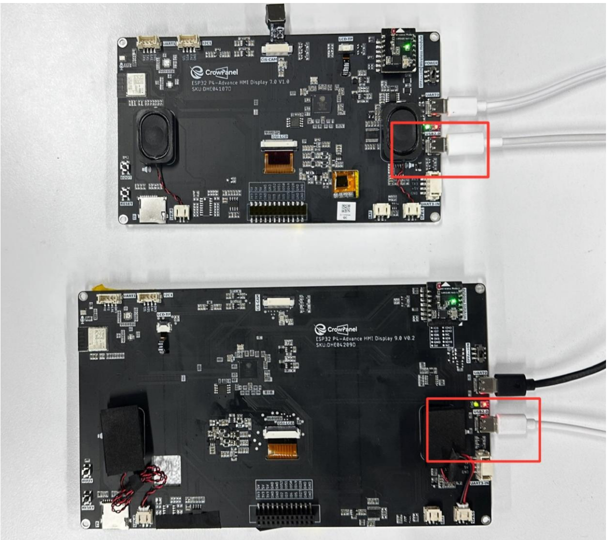
- Next, we will compile and flash the code (detailed in the first lesson).

- 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**.

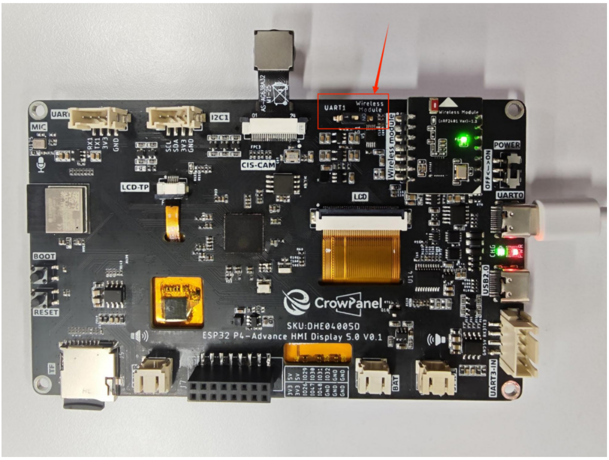


- After waiting for a moment, the code will finish compiling and uploading, and the monitor will open automatically.

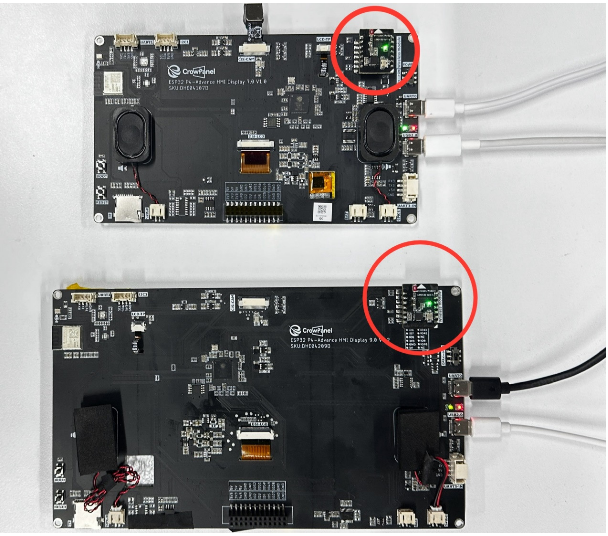
- At this point, remember to connect your Advance-P4 using an additional Type-C cable via the USB 2.0 interface. This is because the maximum current provided by a computer's USB-A port is generally 500mA, and the Advance-P4 requires a sufficient power supply when using multiple peripherals—especially the screen. (It is recommended to connect it to a charger.)



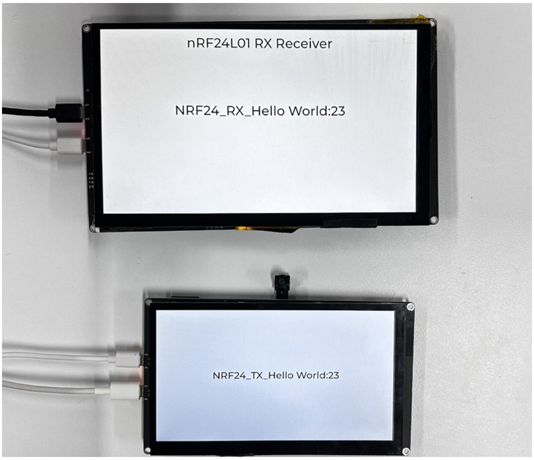
- For the 5-inch Advance-P4 product, the jumpers need to be switched on the hardware in order to use the wireless module. (Switch to the side with the wireless module)



- Insert the nRF2401 wireless RF module into each of the two Advance-P4 development boards.



- After running the code on both boards respectively, you will be able to see on the transmitter's Advance-P4 screen that the nRF2401 module is sending data labeled "NRF24_TX_Hello World:i", where "i" increases by 1 every second.
- Similarly, on the receiver's Advance-P4 screen, you will see that the nRF2401 module is receiving data labeled "NRF24_RX_Hello World:i"; after receiving the message, "i" will also increase by 1 every second.



ELECROW

MAKE YOUR MAKING EASIER