ELECROW

All-in-one Starter Kit for ESP32-P4

- Rapid AI deployment
- 2MP camera | 7-inch display
- Open-source Hardware

20+
Lessons

16
Built-in modules

www.elecrow.com

Table of Contents

Introduction

Welcome to the User Manual for the All-in-one Starter Kit for ESP32-P4. Let's begin our journey into the world of the ESP32-P4 development board and its intelligent sensor modules.

This development board is equipped with 19 courses, carefully designed to be progressively challenging, engaging, and thought-provoking. These courses will guide you step-by-step through essential concepts and hands-on practices. Here, you will become familiar with various electronic modules, strengthen your logical thinking skills, enhance your creative design abilities, and implement the functionality of these modules through programming using the ESP-IDF framework.

The learning process begins with setting up the ESP-IDF development environment, followed by an introduction to the ESP32-P4 development board and its wide range of connected modules. You will then explore how to program each module, understand its communication protocols, and apply your knowledge in practical applications. Each step is clearly explained, making it easy for beginners to quickly grasp embedded development using C language within the IDF environment.

The All-in-one Starter Kit for ESP32-P4 includes 16 electronic modules, each with distinct features and functions, making it an ideal choice for beginners who wish to explore both hardware and software development. For example, the temperature and humidity sensor allows you to monitor environmental data, while the relay and motor modules help you control real-world devices through code.

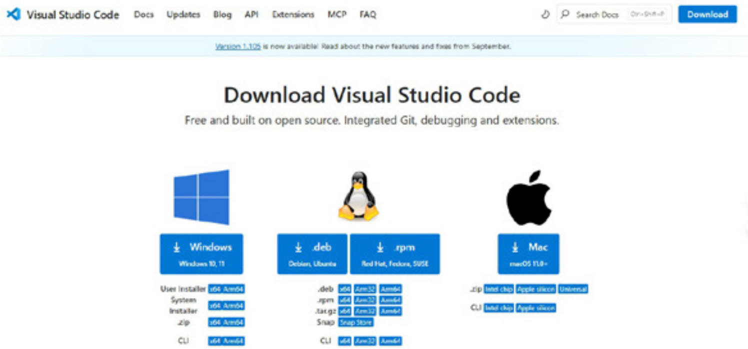
In summary, by working with this development board, you will gain a solid understanding of sensors and actuators, learn important concepts such as digital and analog signals, PWM, ADC, DAC, and communication interfaces like UART, I²C, and SPI. You will also master how to integrate network capabilities such as Wi-Fi and Bluetooth, and even apply simple AI functions in your projects. Most importantly, through ESP-IDF programming, you will develop a deep understanding of embedded systems and enhance your logical and problem-solving skills.

For the programming software, we will utilize the ESP-IDF development framework. ESP-IDF is Espressif's official and powerful open-source platform, offering developers full control over hardware and enabling professional-grade embedded programming. It is one of the best tools for learning real-world IoT and AI application development.

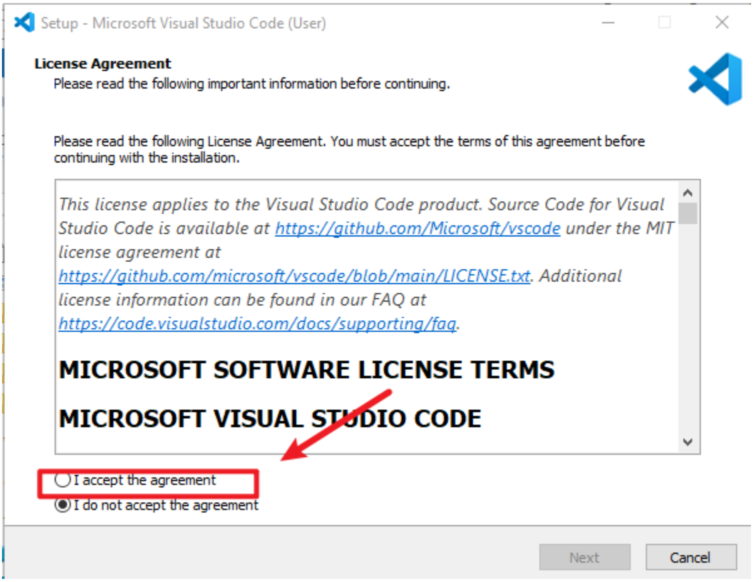
ESP-IDFEnvironment Setup Guide

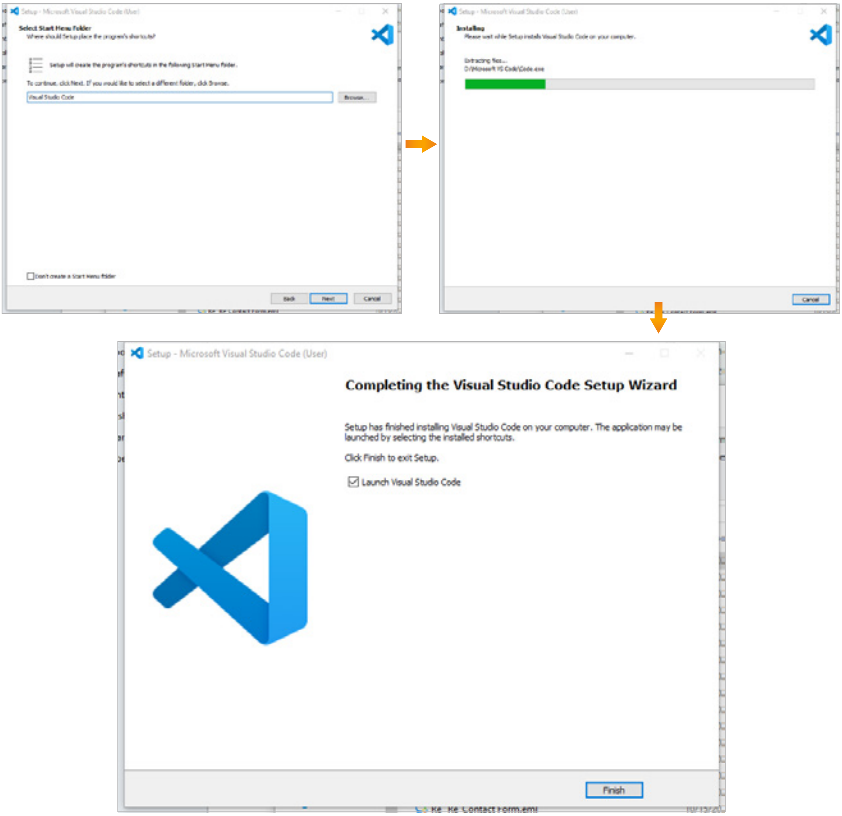
Vs code Installation

• First, download Visual Studio Code from https://code.visualstudio.com/. Select the version compatible with your computer's operating system and download it.

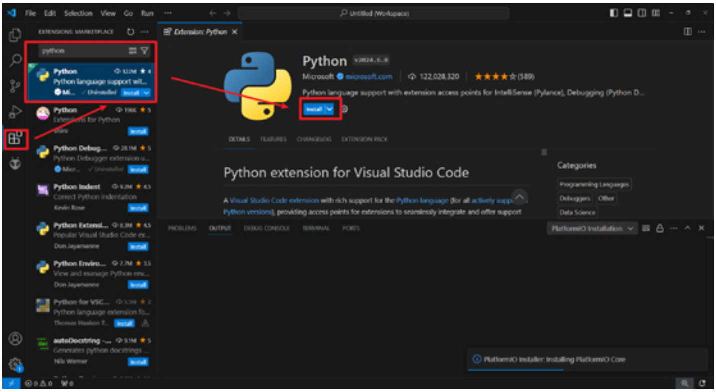


1. Double-click to install Visual Studio Code software, and simply proceed with the default installation throughout.

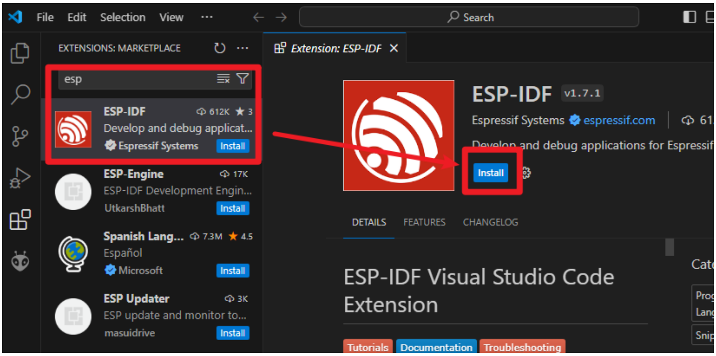




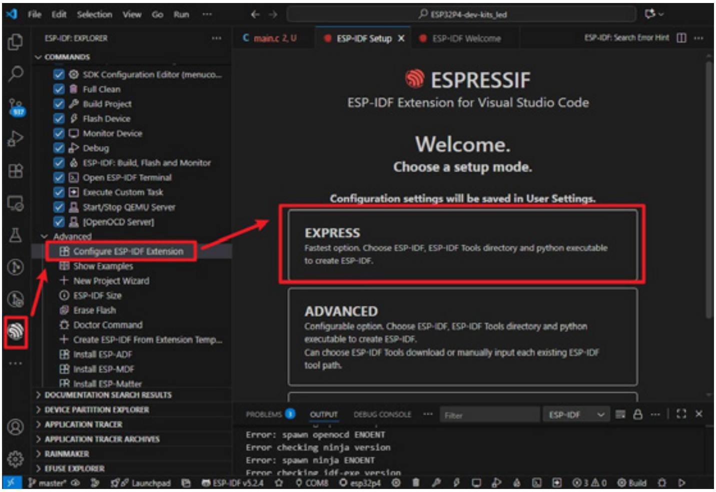
2. Open Visual Studio Code, click on 'Extensions', search for Python, and install it.



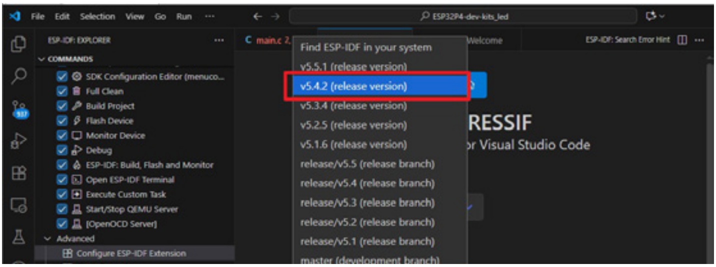
3. Search for ESP-IDF and install it.



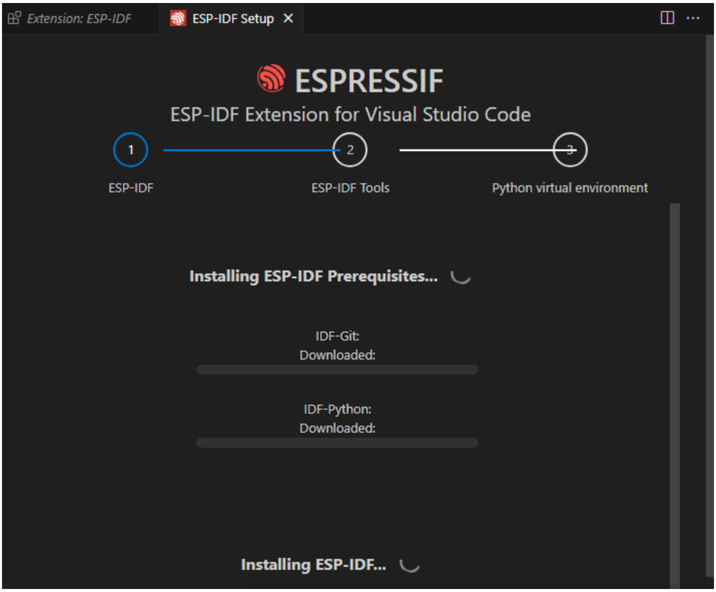
4. Install the ESP-IDF tools.



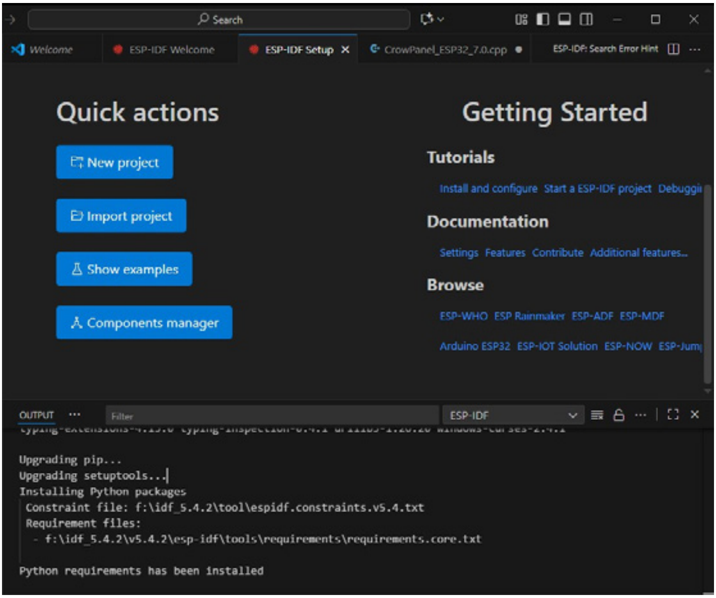
5. Select version 5.4.2 and configure the storage location.



6. Awaiting installation.



7. Installation successful.

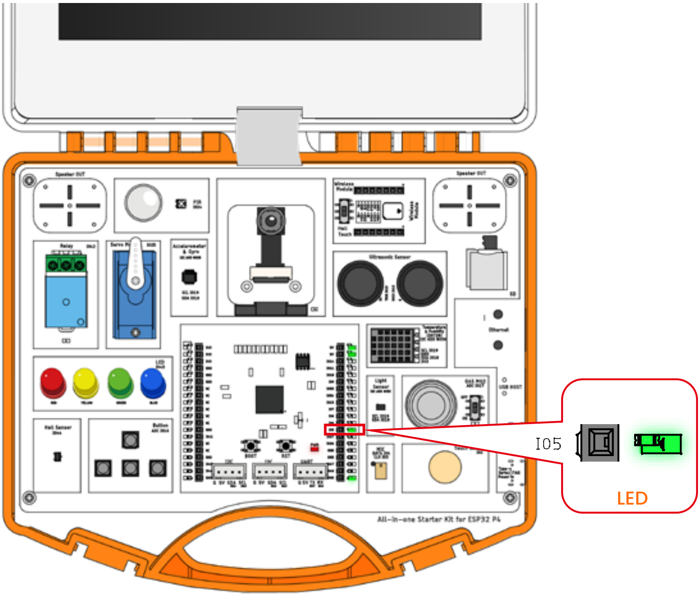


Lesson 1 - GPIO LED Control

Introduction

This chapter's tutorial introduces the GPIO output applications of the ESP32-P4, using a light-up example to help understand its fundamental functionality. As a classic test case, the light-up demonstration provides readers with a straightforward yet comprehensive grasp of the ESP32-P4's applications, laying the groundwork for more complex projects to follow.

Project Demonstration Effect



This chapter is divided into the following subsections

- 1.1 Introduction to GPIO and LEDs
- 1.2 Hardware Design
- 1.3 Programme Design
- 1.4 Download and Verification

1.1 GPIO and LED Introduction

1.1.1 GPIO Introduction

The ESP32-P4 chip provides 55 general-purpose input/output (GPIO) functions, offering flexibility and adaptability across a wide range of applications. Key features of these GPIOs include:

① Versatility: Each GPIO pin can function not only as an input or output, but also be configured via IO MUX for various roles (refer to Chapter 2 for details), such as PWM, ADC, I2C, SPI, and more. This enables the ESP32-P4 to accommodate diverse peripheral connections.

② High current output: The ESP32-P4's GPIO pins support up to 40mA current output, enabling 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 each GPIO's input/output mode, pull-up/pull-down parameters, and other settings to meet specific application requirements.

④ Interrupt Support: GPIO pins support interrupt functionality, capable of triggering interrupts upon signal changes. This is suitable for real-time response applications such as button detection and sensor triggering.

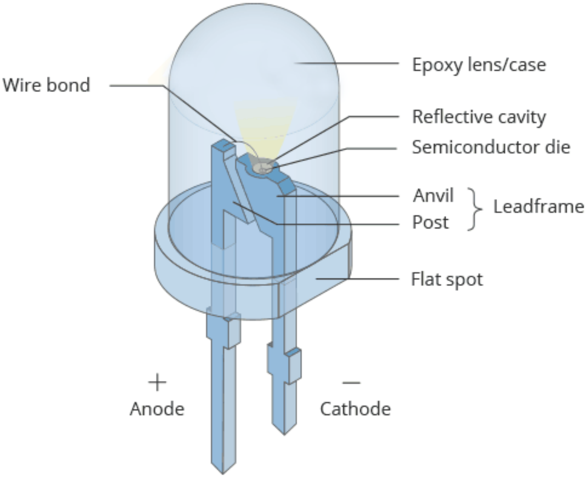
⑤ Status Indication: GPIO pins can function as LED indicators, enabling status visualisation through simple high/low level switching. This facilitates user debugging and monitoring of system operation. The GPIO capabilities of the ESP32-P4 provide robust hardware support for developers. In this chapter, we shall explore GPIO applications and configuration in depth through a light-up example.

1.1.2 LED Introduction

LEDs (light-emitting diodes) are highly efficient, long-lasting miniature semiconductor devices that emit light when an electric current passes through them. They offer advantages such as high energy conversion efficiency, low heat generation, and environmental friendliness. Commonly used in indicator lights, displays, and lighting equipment, LEDs provide rapid response times and a wide range of colours, making them widely applicable in electronic products. In the ESP32-P4 lighting demonstration, GPIO control simplifies and intuitively facilitates LED switching, aiding users in grasping its practical applications.

① Principle of LED Light Emission

LED devices are light-emitting components based on solid-state semiconductor technology. When a forward current is applied across a semiconductor material with a PN junction, the recombination of charge carriers within the semiconductor releases energy in the form of photons, thereby producing light. Consequently, LEDs are cold light sources that do not generate heat from filament-based illumination, eliminating issues such as burnout. The diagram below illustrates the operating principle of an LED device.



Operating Principle of LED Devices

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

Firstly, carriers are injected under the forward bias;

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

Finally, the energy released during recombination is radiated outward in the form of light. Simply put, when current flows through the PN junction, electrons migrate towards the P-region under the influence of the electric field. There, they recombine with holes, releasing excess energy and generating photons, thereby enabling the PN junction's luminescent function.

Note: The colour of light emitted by an LED is determined by the bandgap width of the semiconductor material used. Different materials produce light of varying wavelengths, enabling diverse colour outputs. This highly efficient luminescence mechanism has led to the widespread adoption of LEDs in both illumination and indicator applications.

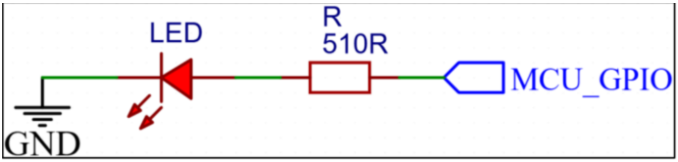
② Principles of LED Lighting Drivers

LED driving refers to supplying LEDs with suitable current and voltage via a stable power source to ensure proper illumination. The primary LED driving methods are constant current and constant voltage, with constant current driving being favoured for its ability to limit current. As LED lamps are highly sensitive to current fluctuations, exceeding their rated current may cause damage. Consequently, constant current driving safeguards LED operation by maintaining stable current flow. Next, we shall examine the two LED drive methods.

1) Current injection connection. This refers to the LED's operating current being supplied externally, injecting current into our MCU.

The risk here is that fluctuations in the external power supply may easily cause the MCU's pins to burn out.

2) Sink current configuration. This denotes the MCU supplying voltage and current, outputting current to the LED. If the MCU's GPIO is used to directly drive the LED, its drive capability is relatively weak and may fail to provide sufficient current to drive the LED.



The LED circuit on the DNESP32P4 development board employs the sink current configuration. This approach avoids the MCU directly supplying voltage and current to drive the LED, thereby effectively reducing the load on the MCU. This allows the MCU to focus more on executing other core tasks, thereby enhancing the overall system performance and stability.

LED lighting in everyday life:

③ LED Voltage Drop and Drive Current

The LED circuit on the P4 development board employs the circuit shown earlier to drive the LED lamp. What, then, is the current flowing through the LED in this circuit? Before addressing this question, we must first grasp a fundamental concept: the reference voltage drop value of an LED. Below are the reference voltage drop values for surface-mount LEDs:

1) Red: Voltage drop 1.82–1.88V, current 5–8mA.

2) Green: Voltage drop 1.75–1.82V, current 3–5mA.

3) Blue: Voltage drop 3.1–3.3V, current 8–10mA.

Using the aforementioned SMD LED voltage drop reference values, the LED current can be calculated via Kirchhoff's voltage law.

The calculation process is as follows:

(3.3 – 1.8) / 510R = 2.9mA

Ignoring the diode's own resistance, the current flowing through the LED is 2.9mA. Although this current value falls outside the standard current reference range for surface-mount LEDs, 2.9mA is still sufficient to illuminate the red LED.

In numerous circuits, regardless of the LED colour mounted on the board, current-limiting resistors of identical values are typically employed. This practice primarily stems from considerations of standardising components and simplifying design. Utilising uniform resistor values reduces production and maintenance complexity while facilitating inventory management. Furthermore, standardised resistor values streamline the circuit design process, enabling designers to work more efficiently during both design and debugging phases.

1.2 Hardware design

1.2.1 Routine Functionality

Within a 500-millisecond cycle, the logic state of LED5 will toggle.

1.2.2 Hardware resources

1）LED

- IO5

In the diagram above, LED5 is controlled by GPIO5 on the ESP32-P4, which determines whether it is illuminated or not. Concurrently, the PWR indicator displays the power status, illuminating when the power supply is connected.

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.

Within the **ESP32P4-dev-kits_gpio** example, a new folder named bsp_led has been created under the path **ESP32P4-dev-kits_gpio\peripheral**. Within the **bsp_led** path, a new include folder, a **CMakeLists.txt** file, and a **Kconfig** file have been created. The **bsp_led** folder houses the **bsp_led.c** driver file, the include folder contains the **bsp_led.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling the project to utilise LED driver functionality. The **Kconfig** file loads the entire driver alongside GPIO pin definitions into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 LED Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The LED driver source code comprises two files: **bsp_led.c** and **bsp_led.h.**

Below we shall first analyse the **bsp_led.h** programme: it contains relevant definitions for the LED pins and function declarations.

/* Header file references*/

```
/*——————————————————————————————————————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 "driver/gpio.h"//References for GPIO Function-related API Functions
/*————————————————————————————————————Header file declaration end————————————————————————————————————*/
```

/* Pin Definitions and Function Declarations */

```
/*---------------------------------------------Variable declaration---------------------------------------------*/
#define LED_TAG "LED"
#define LED_INFO(fmt, ...) ESP_LOGI(LED_TAG, fmt, ##__VA_ARGS__)
#define LED_DEBUG(fmt, ...) ESP_LOGD(LED_TAG, fmt, ##__VA_ARGS__)
#define LED_ERROR(fmt, ...) ESP_LOGE(LED_TAG, fmt, ##__VA_ARGS__)

#ifdef CONFIG_BSP_LED_ENABLED

#define LED_GPIO CONFIG_LED_GPIO //LED GPIO
esp_err_t led_init();//Initialize the GPIO Pin of the LED
esp_err_t set_led_state(bool state);//Set the GPIO Pin Output Level State of the LED
void led_toggle();//Toggle the GPIO Pin Output Level State of the LED

#endif
```

Next, we shall analyse the code in **bsp_led.c**: the initialisation configuration and functional code for the LED pins.

/* Initialisation function led_init */

```
esp_err_t led_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << LED_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT_OUTPUT,   /* GPIO mode: set input/output mode                     */
        .pull_up_en = false,              /* GPIO pull-up                                         */
        .pull_down_en = false,            /* GPIO pull-down                                       */
        .intr_type = GPIO_INTR_DISABLE,   /* GPIO interrupt type                                  */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the led_init function, the various member variables of the gpio_config_t structure are first parameterised. Subsequently, the gpio_config function is invoked to complete the GPIO initialisation using these configuration parameters.

It is worth noting that configuring the pins as input/output mode (`GPIO_MODE_INPUT_OUTPUT`) is primarily because ESP-IDF does not provide relevant level inversion functions. Therefore, to implement level inversion for the LED pin,

the current pin level must first be read, followed by setting its opposite level. This achieves the desired inversion. If configured as output mode (GPIO_MODE_OUTPUT), the gpio_get_level function cannot be used to obtain the pin level, thereby preventing level inversion functionality.

/* Level Toggle Function led_toggle */

```
void led_toggle()
{
    gpio_set_level(LED_GPIO, !(gpio_get_level(LED_GPIO))); /*Set the Corresponding Output Level of GPIO*/
}
```

Within the **led_toggle** function, the LED pin is configured to output a level-inverted signal based on the current input level.

Kconfig file

The primary function of this file is to add the required configuration to the sdkconfig file, enabling certain parameter settings to be modified via a graphical interface. Here, the number 5 refers to GPIO_NUM_5, the pin connected to the LED.

```
menu "BSP LED Setup"

    config BSP_LED_ENABLED
        bool "Enable LED"
        default n

    if BSP_LED_ENABLED
        config LED_GPIO
            int "GPIO For LED"
            default 5
    endif
endmenu
```

CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_led** driver. To successfully call the contents of the bsp_led folder within the main function, it is necessary to create and configure the **CMakeLists.txt** file located within the **bsp_led** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the bsp_led driver functionality.

Note: In subsequent lessons, we shall not create a new **CMakeLists.txt** file from scratch. Instead, we shall make minor modifications to this existing file to incorporate additional drivers into the build system.

1.3.2 main

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the header file main.h within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The main.h file primarily references required header files: functions utilising the **bsp_led** driver necessitate inclusion of the **bsp_led.h** header file.

Below is an analysis of the main.c programme: system initialisation and execution of LED-specific functions.

```
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
```

This code resides within the init function, which is employed to store initialisation functions requiring invocation and to evaluate the outcome of such initialisation. Should the returned status not be ESP_OK, the code will display an error message and cease further execution.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_LED_ENABLED
    while (1)
    {
        led_toggle();                     /*Toggle the GPIO Pin Output Level State of the LED*/
        vTaskDelay(pdMS_TO_TICKS(500)); /*Delay 500ms*/
    }
#endif
```

Within the **app_main** function, establish a loop that repeatedly executes the following: every 500 milliseconds, toggle the LED pin's logic level (to achieve the LED flashing effect).

1.3.3 CMkaLists.txt file

To successfully call the contents of the bsp_led folder within the main function, it is necessary to create and configure the **CMakeLists.txt** file within the main folder. The configuration should be as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led)
```

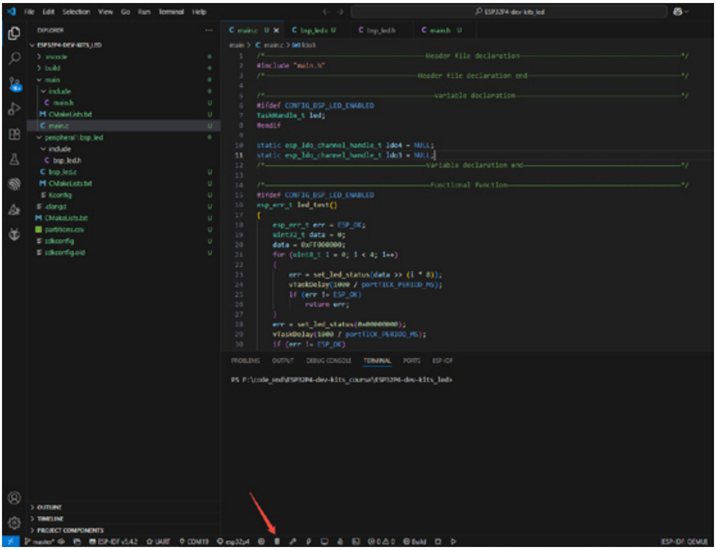
First, the directories for source files and header files are defined, along with the required driver library—specifically, the driver library for linking **bsp_led**. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the main function to utilise the **bsp_led** driver functionality.

> ***Note:*** In subsequent lessons, we shall not create a new **CMakeLists.txt** file from scratch. Instead, we shall make minor modifications to this existing file to incorporate additional drivers into the main function.

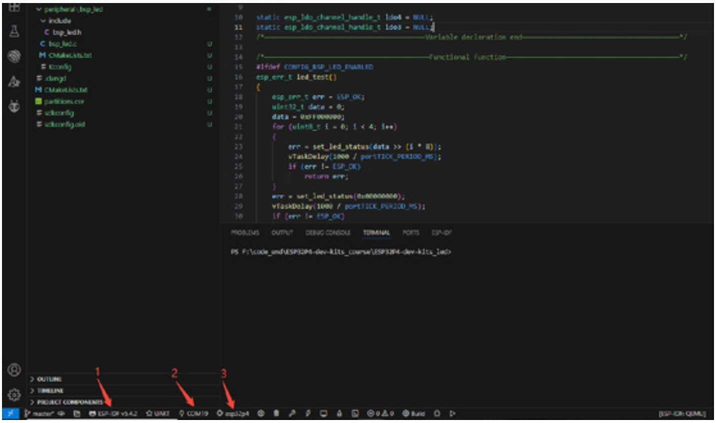
1.4 Programming procedure

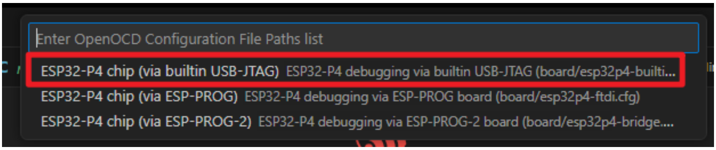
Connect the P4 device to the computer via USB

1.4.1 After cloning the code via Git (link to be confirmed), clear any local compilation information.



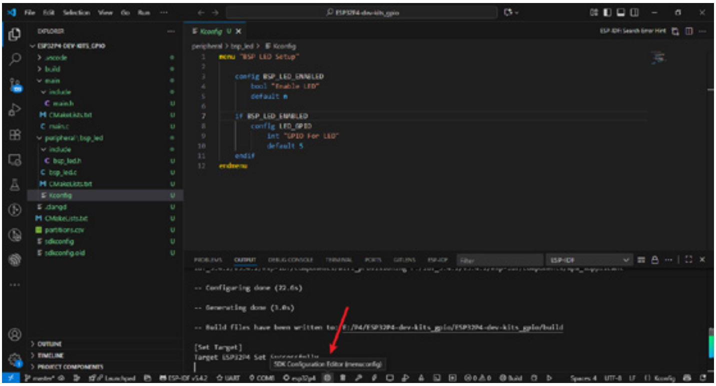
1.4.2 Configure the IDF environment and chip model for compilation, and set the serial port number for programming.



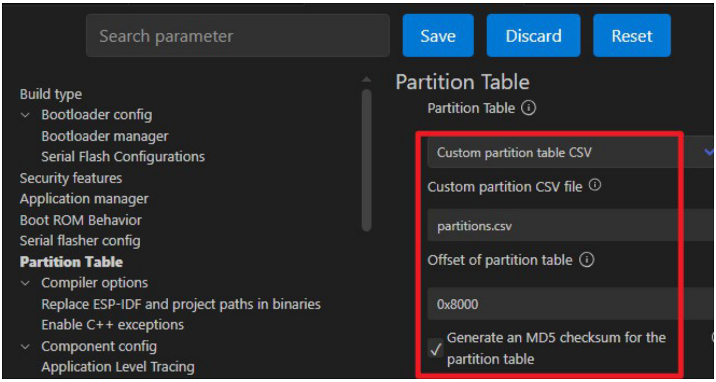


1.4.3 Configure the required settings via the SDKConfig (set bootloader to burn CVS files and select burn speed, enable PSRAM and select speed, activate initialisation for the course-specific BSP_LED component; if unavailable, search directly for the relevant content in the search box). Subsequently execute the function: compile, burn, and open the monitor.

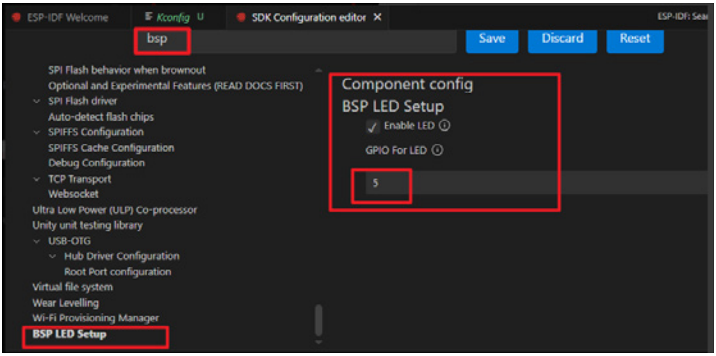
① Click the SDK_config below to access the settings.



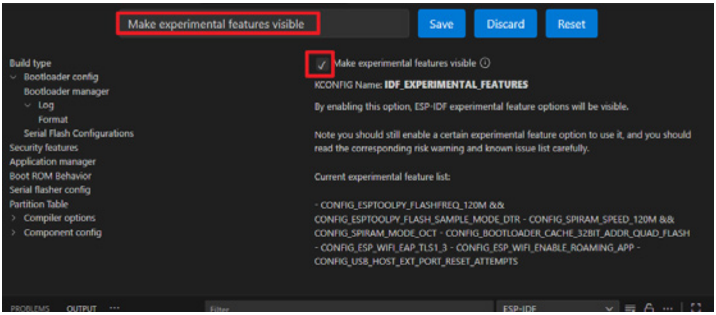
② Scroll down to the Partition Table section and configure the settings as shown in the diagram.



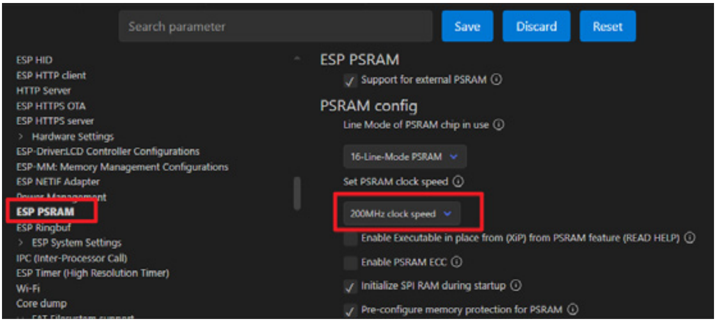
③ You can search directly in the search box for BSP. Set the LED light pins.



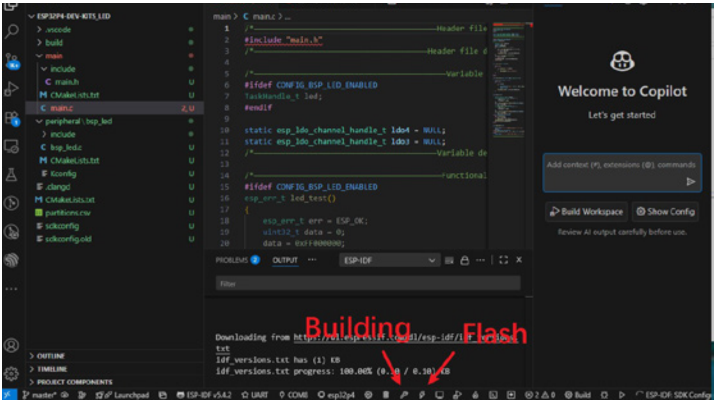
④ You may search directly in the search box for Make experimental features visible and tick the option shown in the image.



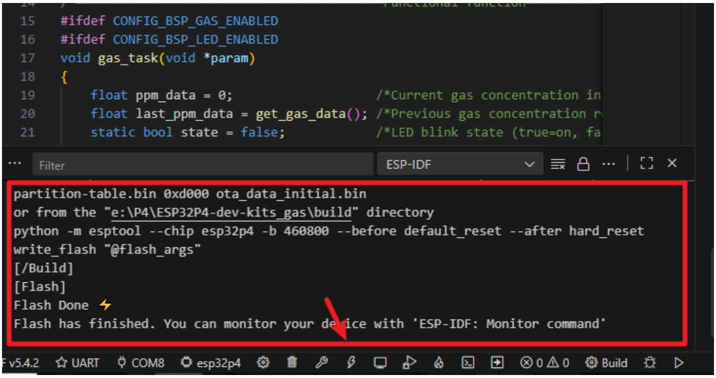
⑤ You may search directly in the search box or locate ESP PSRAM and set it to 200MHz.



1.4.4 Click Compile. Once compilation is successful, click Download.



Download successful notification：

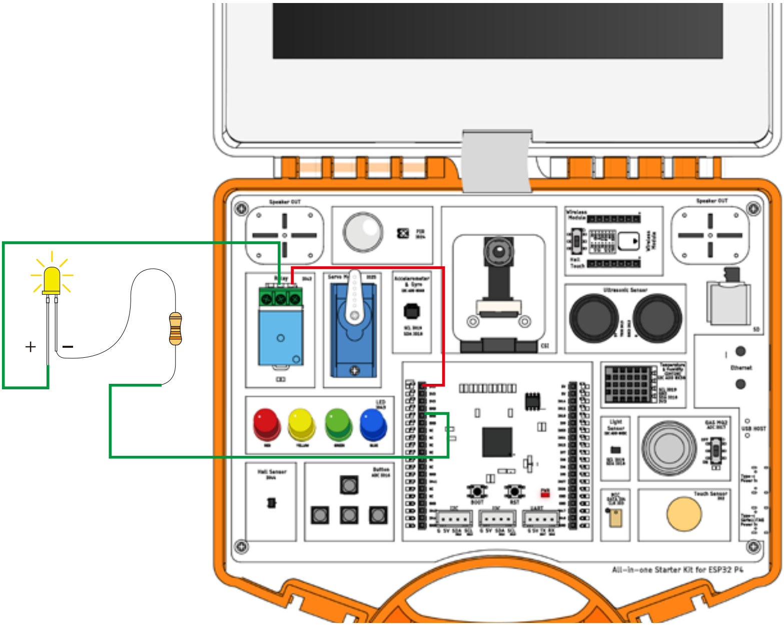


Lesson 2 - GPIO Relay Control

Introduction

This chapter's tutorial introduces the GPIO output applications of the ESP32-P4. Through a relay control example, it aids in understanding the practical application of GPIO in load control. As a common switching device, relays enable isolation and control between microcontrollers and high-voltage, high-current equipment, forming an essential foundation for learning smart hardware projects.

Project Demonstration Effect



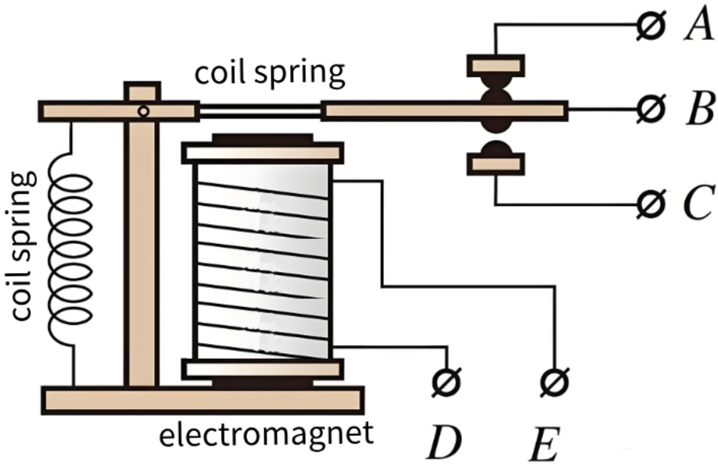
This chapter is divided into the following subsections

- 1.1 Introduction to Relay
- 1.2 Hardware Design
- 1.3 Programme Design
- 1.4 Download and Verification

1.1 Relay Introduction

1.1.1 Relay Introduction

A relay is an electromagnetic control switch that uses a small current to control a larger current. It is widely employed for circuit isolation and the control of electrical equipment. In the ESP32-P4's basic LED lighting example, we utilised an LED; in this section, we shall instead control a relay to illuminate external appliances such as a small fan or light bulb.



Component descriptions in the diagram

Electromagnet coil: Generates a magnetic field when current flows through it.

Armature (iron plate connected to the spring): Pulled down by the magnetic field, altering the contact state.

Return Spring: Retracts the armature to its original position when the electromagnet is de-energised.

Contact Assembly:

A—B: Normally Closed (NC) contact, closed in the default state.

B—C: Normally Open (NO) contact, open in the default state.issues such as burnout. The diagram below illustrates the operating principle of an LED device.

D, E: The two coil terminals of the electromagnet, used for connecting the control current.

① How Relay Works

Relays typically comprise a **coil**, **armature**, and **contacts**. When the GPIO outputs current to drive the relay coil, the coil generates a magnetic field that attracts the armature, thereby altering the contact's open or closed state.

Working Principle

1. Initial State (No Current)

Coils D and E are not energized → The electromagnet has no magnetic force. The spring pulls the armature back, keeping it in the upper position.

At this time:

Contacts B—C are open (not conducting).

Contacts A–B are closed (conductive).

2. Active State (Powered)

When control current is applied across terminals D and E, the electromagnet generates a magnetic field. Magnetic force pulls the armature downward. The armature moves the movable contact B downward:

B–C are connected (closed).

A–B are separated (open).

3. Return State (De-energized)

When the control current is interrupted, the electromagnet loses its magnetic force. The spring pulls the armature back to its original position.

The contacts return to their initial state:

B–C is closed.

A–B is open.

This characteristic of controlling high currents with low currents enables microcontrollers to safely operate 220V AC equipment.

② Relay drive method

As relay coils typically require 5V/tens of milliamperes, and the ESP32-P4's GPIO output capability is insufficient for direct driving, a transistor driver circuit or relay module is required:

GPIO → Transistor → Relay Coil → VCC (5V)

Concurrently, a diode is connected in parallel (in reverse bias across the coil terminals) to absorb the reverse electromotive force generated when the coil de-energises, thereby protecting the components.

③ Relay's principal parameters

Common relay parameters include:

Rated voltage: Typically 3.3V, 5V or 12V (control side voltage).(We use 5V here)

Coil current: Ranges from tens to hundreds of milliamperes.

Contact rating: Determines controllable load capacity (e.g., 15A 250V AC).

Normally Open (NO)/Normally Closed (NC) contacts:

NO: Open by default, closes upon energisation.

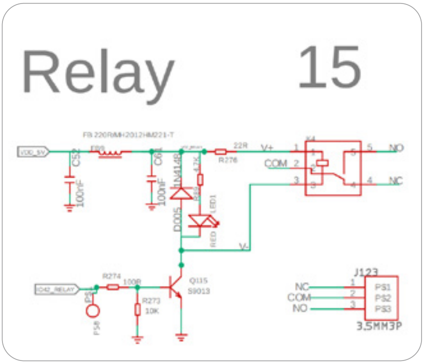
NC: Closed by default, opens upon energisation.

Within smart home systems, relays are commonly employed to control devices such as lighting, air conditioning units, and water pumps.

1.2 Hardware design

1.2.1 Functionality

Within a 5000-millisecond cycle, the relay's electrical state will toggle, switching on once every five seconds to control the illumination of the external LED lamp.



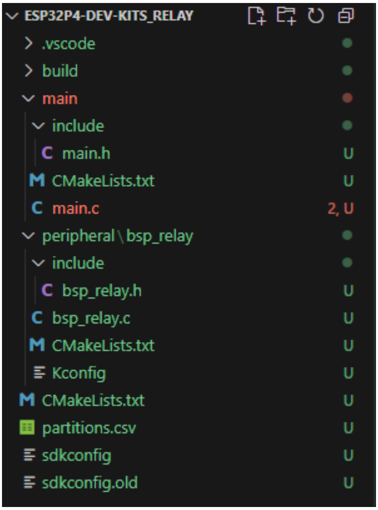
1.2.2 Hardware resources

In the diagram above, the relay is controlled by GPIO42 on the ESP32-P4.

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_relay** example, a new folder named **bsp_relay** has been created under the **ESP32P4-dev-kits_relay\peripheral** directory. Within the **bsp_relay** directory, a new include folder, a **CMakeLists.txt** file, and a Kconfig file have been created. The **bsp_relay** folder houses the **bsp_relay.c** driver file, the include folder contains the **bsp_relay.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling project engineering to utilise the Relay driver functionality. The Kconfig file loads the entire driver alongside GPIO pin definitions into the sdkconfig file within the IDF platform (configurable via the graphical interface).

1.3.1 Relay Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The Relay driver source code comprises two files: **bsp_relay.c** and **bsp_relay.h**.

Below we shall first analyse the **bsp_relay.h** programme: it contains relevant definitions for the Relay pins and function declarations.

/* Header file references */

```
/*——————————————————————————————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 "driver/gpio.h"//References for GPIO Function-related API Functions
/*——————————————————————————————Header file declaration end——————————————————————————*/
```

/* Pin definitions and function declarations */

```
#ifdef CONFIG_BSP_RELAY_ENABLED

#define RELAY_GPIO CONFIG_RELAY_GPIO   // Relay GPIO
esp_err_t relay_init();                // Initialize the GPIO Pin of the Relay
esp_err_t set_relay_state(bool state); // Set the GPIO Pin Output Level State of the Relay
void relay_toggle();                   // Toggle the GPIO Pin Output Level State of the Relay

#endif
```

Next, we shall analyse the code in **bsp_relay.c**: the initialisation configuration and functional code for the Relay pin.

/* Initialisation function relay_init */

```
esp_err_t relay_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << RELAY_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT_OUTPUT,     /* GPIO mode: set input/output mode                     */
        .pull_up_en = false,                /* GPIO pull-up                                         */
        .pull_down_en = false,              /* GPIO pull-down                                       */
        .intr_type = GPIO_INTR_DISABLE,     /* GPIO interrupt type                                  */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the relay_init function, the individual member variables of the gpio_config_t structure were first configured with parameters. Subsequently, the gpio_config function was invoked to complete the initialisation of the GPIO using these configuration parameters.

It is worth noting that configuring the pin as input-output mode (GPIO_MODE_INPUT_OUTPUT) is primarily because ESP-IDF does not provide a dedicated level inversion function. Therefore, to achieve level inversion for the Relay pin, one must first read the pin's current level and then set its opposite level to effect the inversion. If configured as output mode (GPIO_MODE_OUTPUT), the **gpio_get_level** function cannot be used to obtain the pin level, thereby preventing level inversion functionality.

/* Level inversion function relay_toggle */

```
void relay_toggle()
{
    gpio_set_level(RELAY_GPIO, !(gpio_get_level(RELAY_GPIO))); /*Set the Corresponding Output Level of GPIO*/
}
```

Within the **relay_toggle** function, the relay pin is configured to toggle its output state based on the current input level.

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the sdkconfig file, enabling certain parameter settings to be modified via a graphical interface. Here, 42 corresponds to GPIO_NUM_42.

```
menu "BSP RELAY Setup"

    config BSP_RELAY_ENABLED
        bool "Enable RELAY"
        default n

    if BSP_RELAY_ENABLED
        config RELAY_GPIO
            int "GPIO For RELAY"
            default 42
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_relay** driver. To successfully invoke the contents of the **bsp_relay** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_relay** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the **bsp_relay** driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable **main.c** and the header file **main.h** within the include folder. Add the main folder to the CMakeLists.txt file of the build system.

The **main.h** file primarily references required header files: functions utilising the **bsp_relay** driver necessitate inclusion of the **bsp_relay.h** header file.

Below is an analysis of the main.c programme: system initialisation and execution specific to the relay functionality.

```
#ifdef CONFIG_BSP_RELAY_ENABLED
    err = relay_init(); /*Relay Driver Initialization*/
    if (err != ESP_OK)
        init_fail("relay", err);
#endif
```

This code resides within the init function, which is employed to store initialisation functions requiring invocation and to evaluate the outcome of such initialisation. Should the returned status not be **ESP_OK**, the code will display an error message and cease further execution.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_RELAY_ENABLED
    while (1)
    {
        relay_toggle();                 /*Toggle the GPIO Pin Output Level State of the Relay*/
        vTaskDelay(pdMS_TO_TICKS(5000)); /*Delay 5000ms*/
    }
#endif
}
```

Within the app_main function, establish a loop that repeatedly executes the following: every 5000 milliseconds, toggle the level of the relay pin (to achieve relay switching functionality). This differs from the previous lesson's approach to controlling LED flashing intervals because the mechanical structure of the relay switch has a finite lifespan and cannot withstand frequent activation.

1.3.5 CMkaLists.txt file

To successfully call the contents of the bsp_relay folder within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_relay)
```

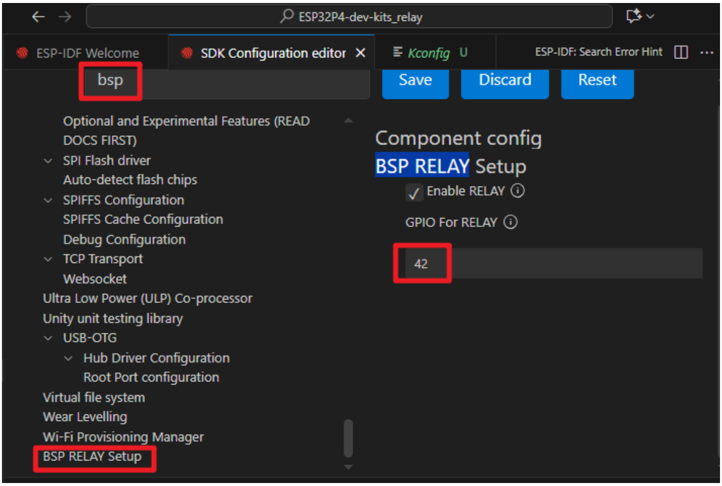
First, the directories for source files and header files are defined, along with the required driver library—namely, the driver library for linking **bsp_relay**. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the main function to utilise the bsp_relay driver functionality.

1.4 Programming procedure

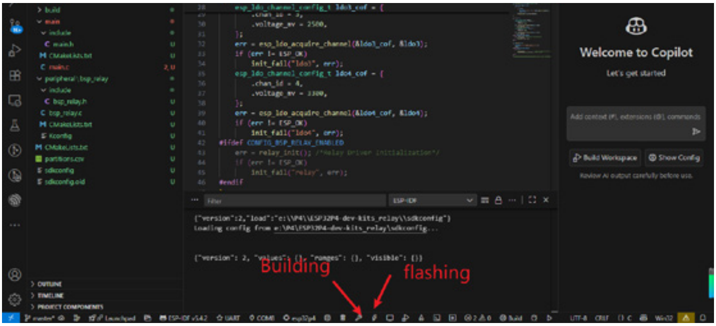
Connect the P4 device to the computer via USB

1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, requiring only the relay pin to be reconfigured.



1.4.3 Click Compile. Upon successful compilation, click Download.

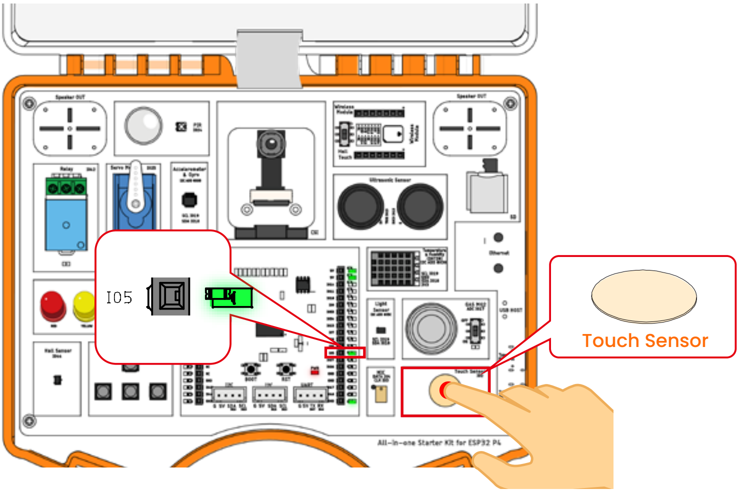


Lesson 3 - Touch Button Toggle

Introduction

This chapter's tutorial introduces the GPIO input application for the ESP32-P4. Through a touch sensor example, it aids in understanding GPIO input detection functionality. As a common human-machine interface, touch sensors enable input detection without mechanical buttons, serving as a crucial case study for learning intelligent interactive applications.

Project Demonstration Effect



This chapter is divided into the following subsections

1.1 Touch Sensor Introduction

1.1.1 Touch Sensor Introduction

Touch sensors are input devices based on the capacitive effect. When a human finger approaches or makes contact, it causes a change in capacitance, which is detected by the chip. They are widely used in smart home systems, consumer electronics, and human-machine interaction applications.

① Principle of Operation of Touch Sensors

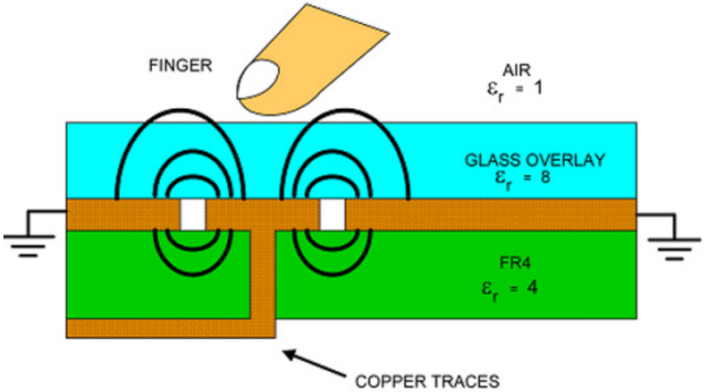
Touch detection relies on capacitance variation:

When no finger is present, the capacitance between the electrode and ground remains stable.

As a finger approaches the electrode, the body's conductivity increases the capacitance value.

The touch detection circuit within the ESP32-P4 periodically measures this capacitance value and compares it against a preset threshold.

Once the capacitance value exceeds this threshold, a ‘touch event’ is detected.



② Touch Sensor Drive Method

The ESP32-P4 incorporates an integrated touch sensor module, enabling developers to utilise it without requiring additional circuitry. Its detection process comprises the following steps:

1. Initialise the GPIO to touch input mode.

2. Read the voltage level of this GPIO to determine whether it has been touched.

Compared to traditional mechanical buttons, touch sensors offer advantages including:

- Absence of mechanical wear

- Rapid response speed

- Convenient waterproof design

③ Key Parameters of Touch Sensors

Common touch sensor parameters include:

Response time: Tens of milliseconds, significantly faster than traditional mechanical buttons.

Sensitivity: Adjustable via software-set thresholds.

Durability: No mechanical components, ensuring extended service life.

Adaptability: Capable of detecting touch through non-metallic materials such as glass and acrylic.

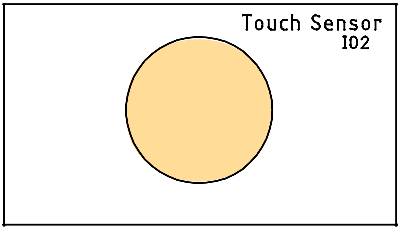
Touch sensors are also widely used in everyday life.

1.2 Hardware design

1.2.1 Routine Functionality

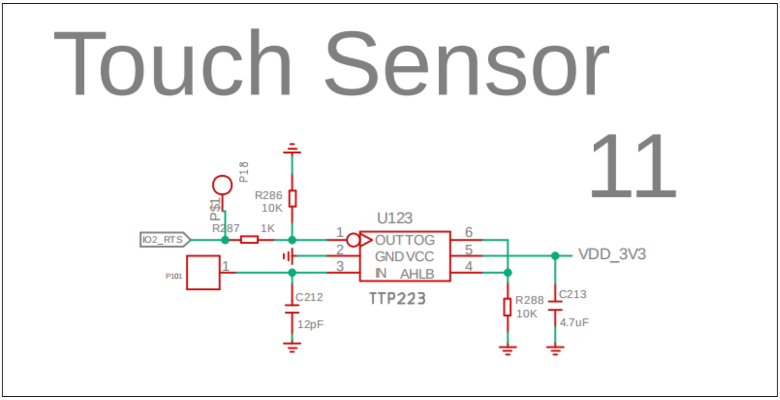
In this experiment, the ESP32-P4 utilises either its onboard or externally attached touch electrodes (metal plates/conductive areas), requiring no additional components. Upon detecting a touch, the LED controlled by GPIO5 illuminates.

1.2.2 Hardware resources



GPIO 2 is connected to the metal touchpad (or the copper foil area on the PCB).

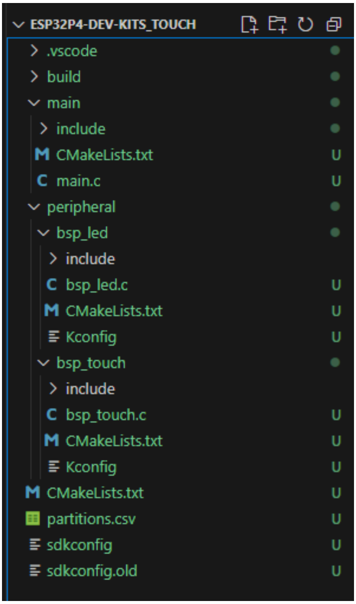
The capacitance changes when a human finger approaches or touches it, and the ESP32-P4 detects these variations in capacitance value.



1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_touch** example, a new folder named **bsp_touch** has been created under the **ESP32P4-dev-kits_touch\peripheral** directory. Within the **bsp_touch** path, a new include folder, a **CMakeLists.txt** file, and a Kconfig file have been established. The **bsp_touch** folder houses the **bsp_touch.c** driver file, the include folder contains the **bsp_touch.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling the project to utilise the Relay driver functionality. The **Kconfig** file loads the entire driver configuration, including GPIO pin definitions, into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 Touch Driver Code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The Touch driver source code comprises two files: **bsp_touch.c** and **bsp_touch.h**.

Below we shall first analyse the **bsp_touch.h** programme: it contains relevant definitions for the touch pins and function declarations.

/* Header file references */

```
/*————————————————————————————————————————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 "driver/gpio.h"//References for GPIO Function-related API Functions
/*——————————————————————————————————————Header file declaration end——————————————————————————————————————*/
```

/* Pin definitions and function declarations */

```
#ifdef CONFIG_BSP_INDEPENDENT_TOUCH_ENABLED

#define INDEPENDENT_TOUCH_GPIO CONFIG_INDEPENDENT_TOUCH_GPIO // Touch GPIO
esp_err_t independent_touch_init();                          // Initialize the GPIO Pin of the Touch
int get_touch_state();                                       // Get the current corresponding level of the GPIO for the Touch button

#endif
```

Next, we shall analyse the code in **bsp_touch.c**: the initialisation configuration and function code for the touch pins.

/* Initialisation function touch_init */

```
esp_err_t independent_touch_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << INDEPENDENT_TOUCH_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT,                        /* GPIO mode: set input/output mode                     */
        .pull_up_en = false,                            /* GPIO pull-up                                         */
        .pull_down_en = false,                          /* GPIO pull-down                                       */
        .intr_type = GPIO_INTR_DISABLE,                 /* GPIO interrupt type                                  */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the **touch_init** function, the member variables of the **gpio_config_t** structure are first configured with parameters. Subsequently, the **gpio_config** function is invoked to complete the initialisation of the GPIO using these configuration parameters. Here, the GPIO mode is set to input mode to read the level status of the I/O ports.

/* Function **get_touch_state** for acquiring touch button status */

```
int get_touch_state()
{
    return gpio_get_level(INDEPENDENT_TOUCH_GPIO); /*Read the current corresponding level of GPIO*/
}
```

Within the `get_touch_state` function, the current level status of the touch button pin is read: a high level indicates the touch button is pressed, while a low level indicates the touch button is released.

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the sdkconfig file, enabling certain parameter settings to be modified via a graphical interface. Here, the number 2 refers to **GPIO_NUM_2**.

```
menu "BSP INDEPENDENT TOUCH Setup"

    config BSP_INDEPENDENT_TOUCH_ENABLED
        bool "Enable INDEPENDENT TOUCH"
        default n

    if BSP_INDEPENDENT_TOUCH_ENABLED
        config INDEPENDENT_TOUCH_GPIO
            int "GPIO For INDEPENDENT TOUCH"
            default 2
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_touch** driver. To successfully invoke the contents of the **bsp_touch** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_touch** folder. The configuration details are as follows:

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

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

Within this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the bsp_touch driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable **main.c** and the header file **main.h** within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The **main.h** file primarily references required header files: functions utilising the **bsp_touch** driver necessitate inclusion of the **bsp_touch** header file, while those employing the **bsp_led** driver require the **bsp_led** header file.

Below is an analysis of the **main.c** programme: system initialisation and execution of LED and touch functionality.

```
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_INDEPENDENT_TOUCH_ENABLED
    err = independent_touch_init(); /*Touch Driver Initialization*/
    if (err != ESP_OK)
        init_fail("independent touch", err);
#endif
```

This code resides within the init function, which is employed to store initialisation functions requiring invocation and to evaluate the outcome of such initialisation. Should the returned status not be ESP_OK, the code will display an error message and cease further execution.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_INDEPENDENT_TOUCH_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED

    bool state = false;      /*Touch button status*/
    bool last_state = false; /*Previous state of the touch button*/
    while (1)
    {
        state = get_touch_state();             /*Get the current touch button status*/
        if ((state) && (state != last_state)) /*If the previous state of the touch button was released, and the current state of the touch button is pressed*/
        {
            set_led_state(true); /*Set the Corresponding Output Level of GPIO*/
        }
        else if ((!state) && (state != last_state)) /*If the previous state of the touch button was pressed, and the current state of the touch button is released*/
        {
            set_led_state(false); /*Set the Corresponding Output Level of GPIO*/
        }
        last_state = state;                    /*Save the previous touch button state*/
        vTaskDelay(10 / portTICK_PERIOD_MS); /*Delay 10ms*/
    }

#endif
#endif
}
```

Within the **app_main** function, initialise the current state variable and past state variable for the touch button. Subsequently, establish a loop to repeatedly execute the following: assess the state every 10 milliseconds. The function for obtaining the touch button status retrieves the current state and compares it with the previous state. If the state has changed, the function within the **bsp_led** driver that sets the LED status is executed. This function takes a parameter to set the LED level (low level illuminates the LED, high level extinguishes it). The current touch button state is then assigned to preserve the past state. The specific functionality is as follows: pressing the touch button turns the LED off, while releasing it turns the LED on.

1.3.5 CMkaLists.txt file

To successfully call the contents of the **bsp_led** and **bsp_touch** folders within the main function, it is necessary to configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

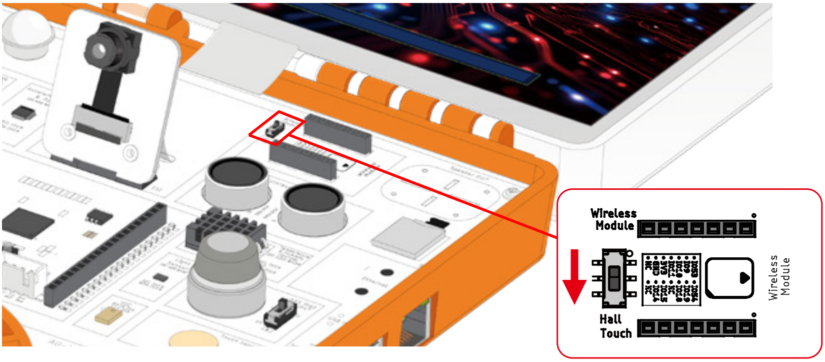
```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_led bsp_touch)
```

First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link **bsp_led** and **bsp_touch**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver capabilities.

1.4 Programming procedure

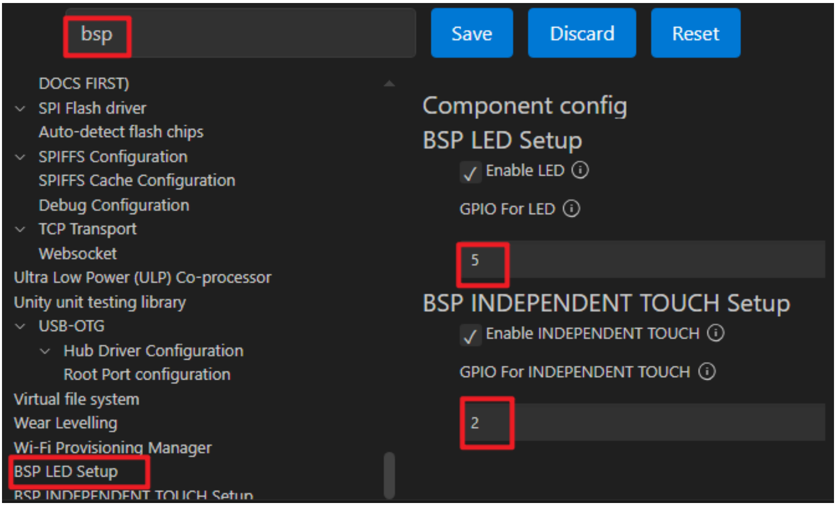
Connect the P4 device to the computer via USB

When using touch and Hall sensors, the toggle switch near the wireless module must be set to the Hall and touch position.

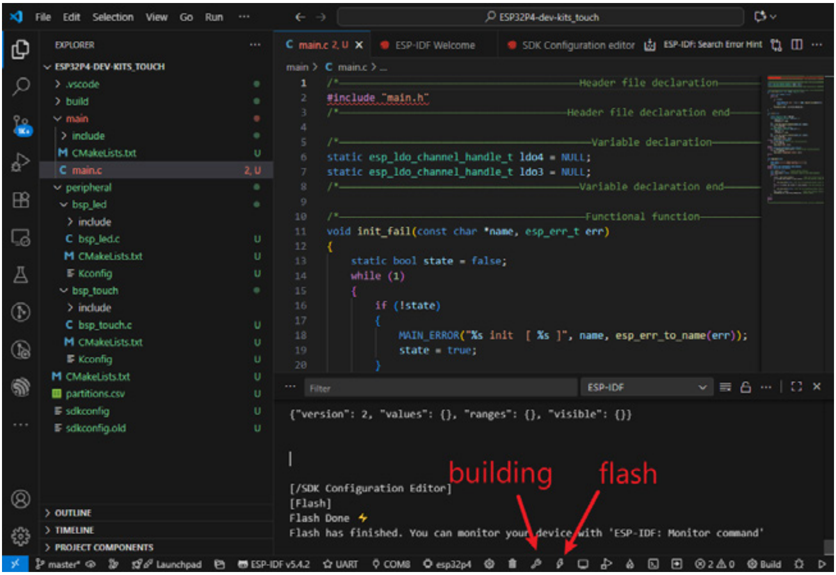


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, requiring only the LED light and touch pins to be reconfigured.



1.4.3 Click Compile. Once compilation is successful, click Download.

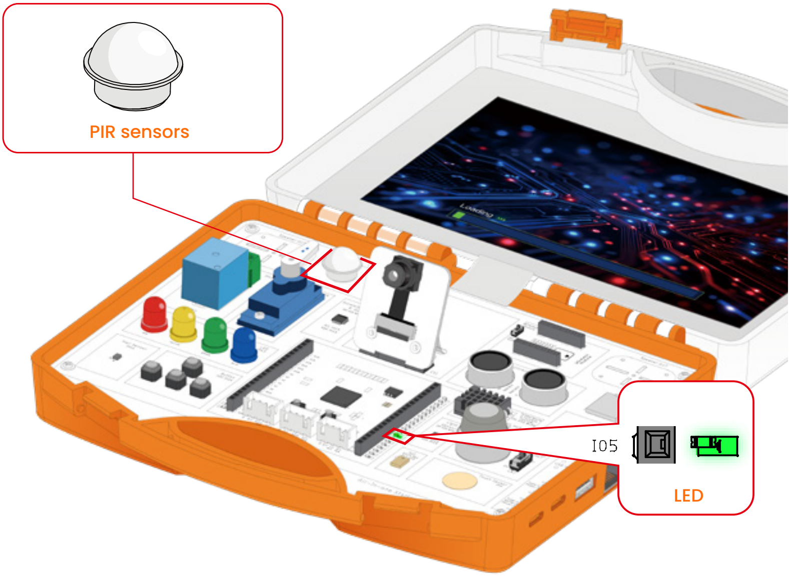


Lesson 4 - PIR Motion Control

Introduction

This tutorial demonstrates the GPIO input application of the ESP32-P4. Through a PIR (Passive Infrared) sensor detection example, it helps users understand the GPIO input detection functionality. As a common human presence detection device, the PIR sensor enables automatic detection of human activity in the environment, making it a crucial case study for learning smart security and automation control.

Project Demonstration Effect



This chapter is divided into the following subsections

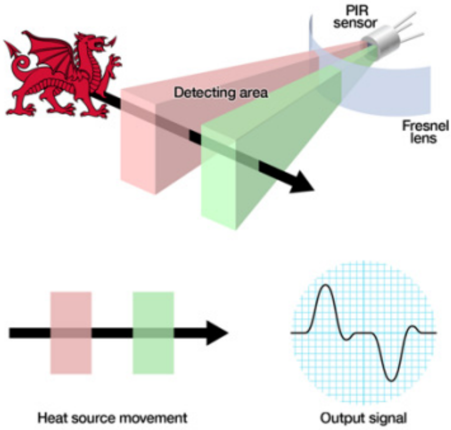
- 1.1 Introduction to the PIR Sensor
- 1.2 Hardware Design
- 1.3 Program Design
- 1.4 Download and Verification

1.1 Introduction to PIR Sensors

1.1.1 Introduction to PIR Sensors

PIR (Passive Infrared Sensor) is a detection device based on infrared radiation sensing, primarily used for detecting human motion. Since the human body emits infrared radiation with wavelengths between 8 and 14 micrometers, PIR sensors detect changes in this radiation to determine whether human activity is present.

① How PIR Sensors Work



The PIR consists internally of a pyroelectric infrared sensor and a Fresnel lens:

When the ambient infrared distribution is stable, the sensor output remains at a low level.

When a human body enters the sensing area, the infrared radiation emitted by the body is focused onto the sensor through a Fresnel lens, causing a change in the input charge.

This charge change is processed by an amplification and comparison circuit, resulting in the output of a high-level pulse.

Therefore, PIR does not directly detect temperature but rather detects dynamic changes in infrared radiation.

② Driving Principle of PIR Sensors

PIR modules typically incorporate signal amplification and comparison circuits, outputting digital signals (high/low levels) for direct connection to the ESP32-P4's GPIO:

Human presence detected → Outputs high level (GPIO detects 1).

No human presence detected → Outputs low level (GPIO detects 0).

This straightforward interface eliminates the need for users to design complex analog circuits.

③ Key Parameters of PIR Sensors

Common PIR modules (such as HC-SR501) have the following key specifications:

Operating voltage: 3.3V to 5V (directly compatible with ESP32-P4 interfaces).

Standby current: Approximately 50 μA (low power consumption).

Detection range: Typically 3 to 7 meters, adjustable.

Sensing angle: Approximately 100°–120°.

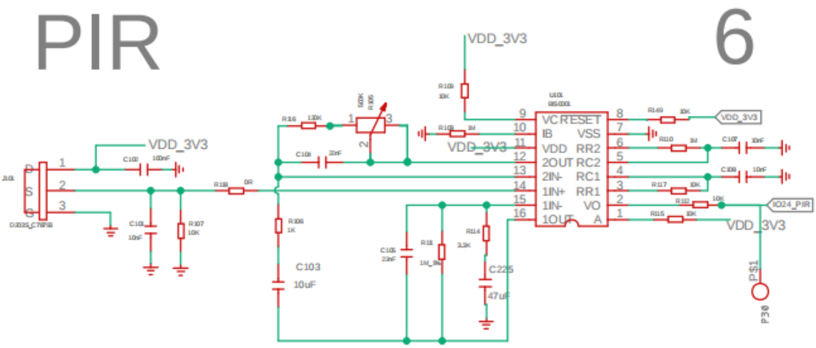
Output format: High/low level, TTL compatible.

Many everyday applications utilize PIR sensors, such as smart lighting systems.

1.2 Hardware Design

In this experiment, the PIR module's VCC is connected to 3.3V, GND is grounded, and the OUT pin is connected to GPIO24 of the ESP32-P4.

When human activity is detected, OUT outputs a high level. The LED controlled by GPIO5 illuminates.

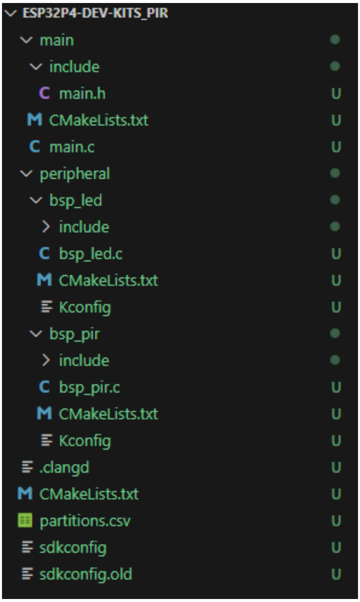


The relay is controlled by the GPIO24 pin of the ESP32-P4.

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



In the **ESP32P4-dev-kits_pir** example, a new folder named **bsp_pir** was created under the **ESP32P4-dev-kits_pir\peripheral** directory. Within the **bsp_pir** directory, a new include folder, a **CMakeLists.txt** file, and a **Kconfig** file were created. The **bsp_pir** folder stores the **bsp_pir.c** driver file, the include folder holds the **bsp_pir.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling project access to PIR driver functionality. The **Kconfig** file loads the entire driver configuration, including GPIO pin definitions, into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 PIR Driver Code

Here we will only explain the core code. For detailed source code, please refer to the corresponding source code for this experiment in the code materials.

The PIR driver source code consists of two files: bsp_pir.c and bsp_pir.h.

Below, we will first analyze the bsp_pir.h program: it defines the PIR pin and declares functions.

/* Header file references*/

```
/*------------------------------------Header file declaration------------------------------------*/
#include "esp_attr.h"    //API Function References Related to Memory Configuration
#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 "driver/gpio.h" //References for GPIO Function-related API Functions
/*------------------------------------Header file declaration end------------------------------------*/
```

/* Pin Definitions and Function Declarations*/

```
#ifdef CONFIG_BSP_PIR_ENABLED

#define PIR_GPIO CONFIG_PIR_GPIO // PIR GPIO
esp_err_t pir_init();            // Initialize the GPIO Pin of the PIR
esp_err_t get_pir_state();       // Get the status for the PIR

#endif
```

Next, we'll analyze the code in bsp_pir.c: the initialization configuration and function code for the PIR pin.

/* Initialization function pir_init */

```
esp_err_t pir_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << PIR_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT,          /* GPIO mode: set input/output mode                    */
        .pull_up_en = false,              /* GPIO pull-up                                        */
        .pull_down_en = false,            /* GPIO pull-down                                      */
        .intr_type = GPIO_INTR_POSEDGE,   /* GPIO interrupt type                                 */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    err = gpio_isr_handler_add(PIR_GPIO, PIR_ISR, (void *)PIR_GPIO); /*Add ISR handler for the corresponding GPIO pin*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the pir_init function, the parameters for each member variable of the gpio_config_t structure are first configured. Next, the gpio_config function is called to complete GPIO initialization using these configuration parameters. Finally, the gpio_isr_handler_add function registers the interrupt callback function and binds it to the corresponding GPIO. Here, the GPIO mode is set to input mode to read the IO port level state, and the interrupt type is selected as rising edge interrupt.

/* PIR pin interrupt callback function PIR_ISR */

```
static void IRAM_ATTR PIR_ISR(void *arg) /*Rising Edge Interrupt Handler for the PIR Pin*/
{
    if ((arg == (void *)PIR_GPIO) && (!PIR_flag))
        PIR_flag = true;
}
```

Within the PIR_ISR function, the interrupt flag is set by checking whether the pin number that triggered the interrupt and the status of the interrupt flag PIR_flag have been cleared. (static bool PIR_flag = false; The interrupt flag type should be defined as a global variable.)

/* Function get_pir_state to retrieve PIR status */

```
esp_err_t get_pir_state()
{
    if (PIR_flag)
    {
        if (gpio_get_level(PIR_GPIO)) /*Read the current corresponding level of GPIO*/
            return ESP_OK;
        else
        {
            PIR_flag = false;
            return ESP_FAIL;
        }
    }
    else
        return ESP_FAIL;
    return ESP_FAIL;
}
```

1.3.2 Kconfig file

The primary function of this file is to add the required configurations to the sdkconfig file, enabling certain parameter adjustments to be made through a graphical interface. Here, 24 refers to GPIO_NUM_24.

```
menu "BSP PIR Setup"

    config BSP_PIR_ENABLED
        bool "Enable PIR config"
        default n

    if BSP_PIR_ENABLED
        config PIR_GPIO
            int "GPIO For PIR"
            default 24
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example primarily relies on the bsp_pir driver. To successfully call the contents of the bsp_pir folder within the main function, you must configure the CMakeLists.txt file located in the bsp_pir folder. The configuration details are as follows:

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

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

In this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver libraries. Then, these settings are registered with the build system using the idf_component_register command, enabling the project to utilize the bsp_pir driver functionality.

1.3.4 main folder

The main folder serves as the core directory for program execution. It contains the main function executable main.c and the main.h header file located within the include folder. Add the main folder to the CMakeLists.txt file of the build system.

The main.h file primarily references required header files: functions utilizing the bsp_pir driver require the bsp_pir header file, while functions using the bsp_led driver require the bsp_led header file.

Below is an analysis of the main.c program: system initialization and execution of LED and PIR functionality.

```c
    err = gpio_install_isr_service(0); /*Install the GPIO driver's ETS_GPIO_INTR_SOURCE ISR handler service, which allows per-pin GPIO interrupt handlers*/
    if (err != ESP_OK)
        init_fail("gpio isr service", err);
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_PIR_ENABLED
    err = pir_init(); /*Pir Driver Initialization*/
    if (err != ESP_OK)
        init_fail("pir", err);
#endif
```

This code resides within the init function, which is used to store initialization functions that need to be called and to evaluate their return values. If the return status is not ESP_OK, the code will print an error message and halt further execution. It is worth noting that we have added the gpio_install_isr_service function to register an interrupt group for all GPIO interrupts.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_PIR_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    bool state = false;             /*PIR status*/
    bool last_state = false;        /*Previous state of the PIR*/
    if (get_pir_state() == ESP_OK) /*First judgment of PIR state*/
        set_led_state(false);
    else
        set_led_state(true);
    while (1)
    {
        if (get_pir_state() == ESP_OK) /*Get the current status feedback from the PIR*/
            state = true;
        else
            state = false;
        if ((state) && (state != last_state))       /*If the previous PIR state was "object stationary" and the current PIR state is "object moving"*/
            set_led_state(false);                   /*Set the Corresponding Output Level of GPIO*/
        else if ((!state) && (state != last_state)) /*If the previous PIR state was "object moving" and the current PIR state is "object stationary"*/
            set_led_state(true);                    /*Set the Corresponding Output Level of GPIO*/
        last_state = state;                         /*Save the previous PIR state*/
        vTaskDelay(10 / portTICK_PERIOD_MS);        /*Delay 10ms*/
    }
#endif
#endif
```

In the app_main function, initialize the PIR current state variable and past state variable. Then, use the first PIR state to determine the initial LED state. Finally, create a loop that repeats the following: check the state every 10ms delay. The PIR status retrieval function determines the current state. This state is compared with the previous state. If the state changes, the function within the bsp_led driver that sets the LED state is executed. This function takes a parameter to set the LED level (low level turns the LED on, high level turns it off). The current PIR state is then assigned to the past state variable for retention. The specific functionality is: the LED illuminates when an object is detected moving and turns off when the object remains stationary.

1.3.5 CMkaLists.txt file

To successfully call the contents of the bsp_led and bsp_pir folders within the main function, you must configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_led bsp_pir)
```

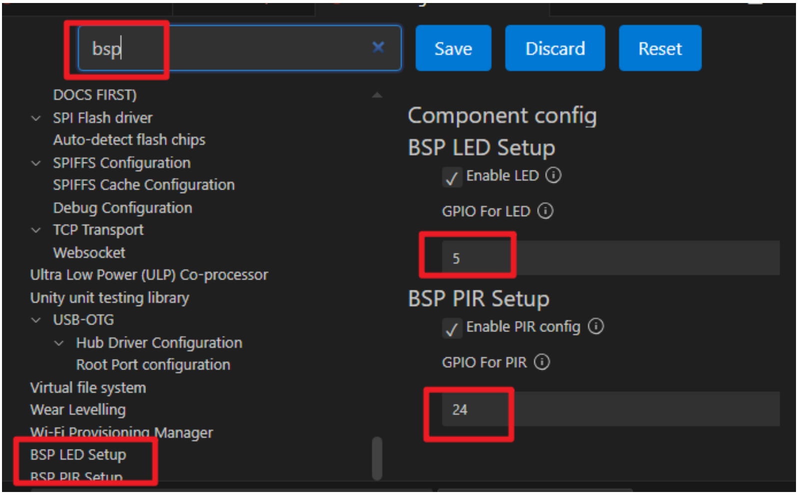
First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link bsp_led and bsp_pir. Then, these settings are registered with the build system using the idf_component_register command, enabling the main function to utilize these driver features.

1.4 Programming procedure

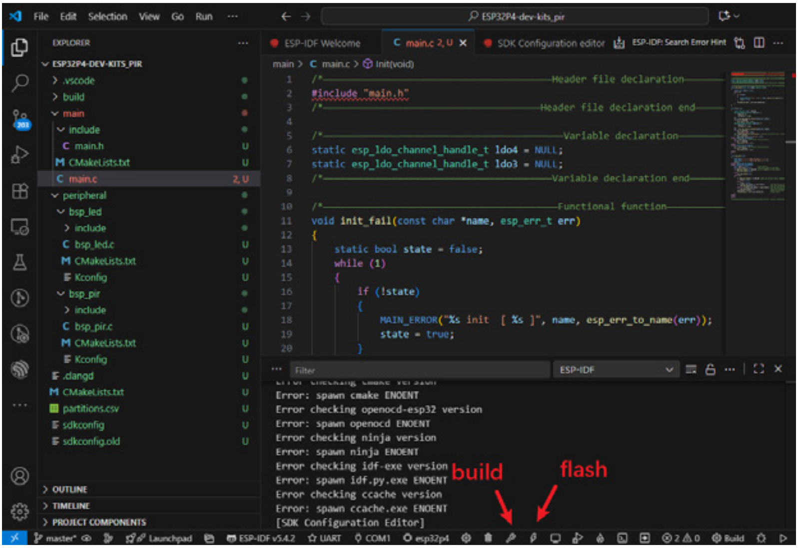
Connect the P4 device to the computer via USB

1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, requiring only the pir and led pins to be reconfigured.



1.4.3、Click Compile. Once compilation is successful, click Download.

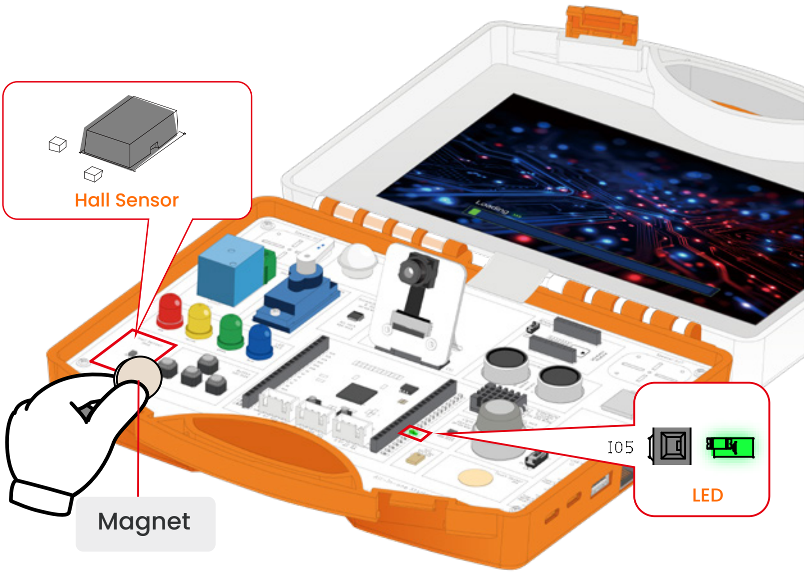


Lesson 5 - Hall Sensor Detect

Introduction

This chapter's tutorial introduces the GPIO input applications of the ESP32-P4, using a Hall sensor example to help you understand its basic functionality. As a common magnetic field detection device, the Hall sensor can directly reflect changes in external magnetic fields, making it widely used in scenarios such as position detection, speed measurement, and current sensing. The learning examples in this chapter will provide readers with a clear understanding of the ESP32-P4's GPIO input capabilities, laying the foundation for more complex sensor applications in subsequent sections.

Project Demonstration Effect



This chapter is divided into the following subsections

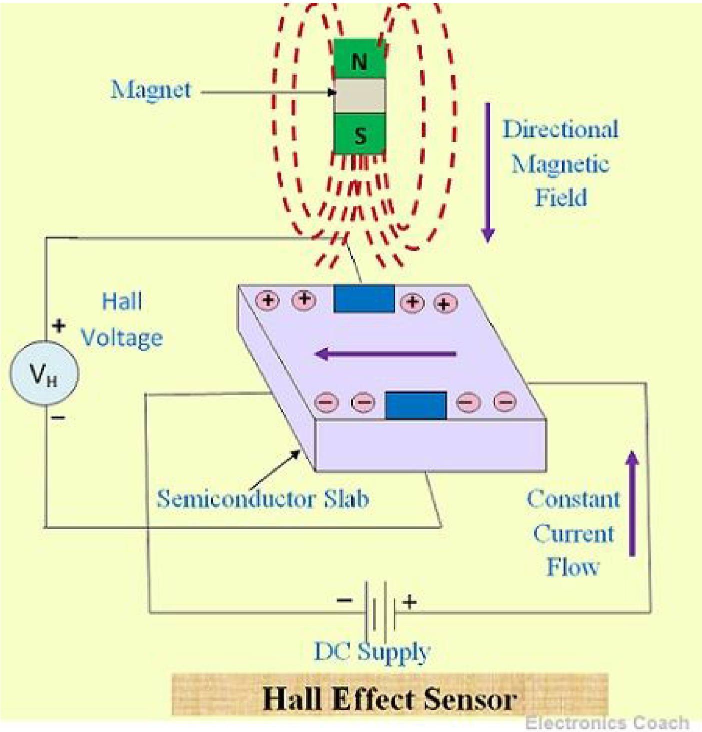
- 1.1 Introduction to Hall Sensors
- 1.2 Hardware Design
- 1.3 Software Design
- 1.4 Download and Verification

1.1 Introduction to hall Sensors

1.1.1 Introduction to hall Sensors

The Hall sensor is a magnetic field detection device that utilizes the Hall effect. When a magnetic field passes through the semiconductor material, it generates a voltage signal across the material's terminals. The ESP32-P4 integrates an analog Hall sensor internally, enabling direct detection of magnetic field strength changes without requiring external components.

1.1.2 Driving Principle of PIR Sensors



① Working Principle of Hall Sensors

The Hall effect refers to the phenomenon where, when a current flows through a conductor or semiconductor, a magnetic field perpendicular to the current direction induces a potential difference between the two ends perpendicular to both the current and magnetic field directions. This is known as the Hall voltage.

Simply put:

No magnetic field → Output voltage approaches zero.

Magnetic field approaches → Output voltage varies with magnetic field strength.

By reading this voltage value via an ADC or dedicated interface, magnetic field information can be obtained.

② Hall Sensor Drive Principle

The ESP32-P4's built-in Hall sensor requires no additional hardware. Magnetic field strength values can be obtained via the SDK's hall_sensor_read() API.

When a magnet approaches the chip, the reading changes noticeably;

When the magnet moves away or is absent, the reading approaches the baseline value;

Users can set thresholds to determine magnetic field detection, enabling functions such as position detection and rotational speed measurement.

③ Key Parameters of Hall Sensors

The critical specifications of Hall sensors include:

Sensitivity: Determines the sensor's responsiveness to magnetic fields.

Operating Voltage: The ESP32's built-in Hall sensor operates directly at 3.3V.

Response Speed: Hall sensors are fast-response devices suitable for high-speed rotor detection.

Temperature Stability: Temperature variations may affect readings; software filtering or calibration methods can compensate.

In experiments with the ESP32-P4 development board, we can verify magnetic field sensing performance without an external Hall chip by directly calling the interfaces provided by the SDK.

1.2 Hardware Design

When using the built-in Hall sensor on the ESP32-P4, no external circuitry is required. Simply position a small magnet near the chip.

To expand with an external Hall sensor (such as the A3144), you need to:

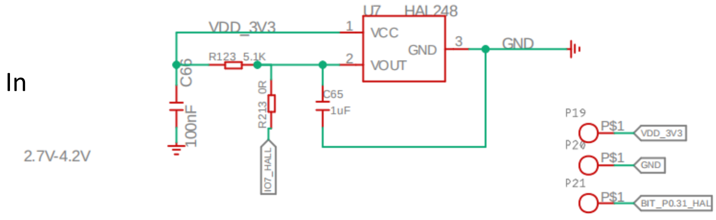
Connect VCC to 3.3V,

Connect GND to ground,

Connect the output pin to a GPIO input pin with a pull-up resistor.

This setup enables magnetic field detection.

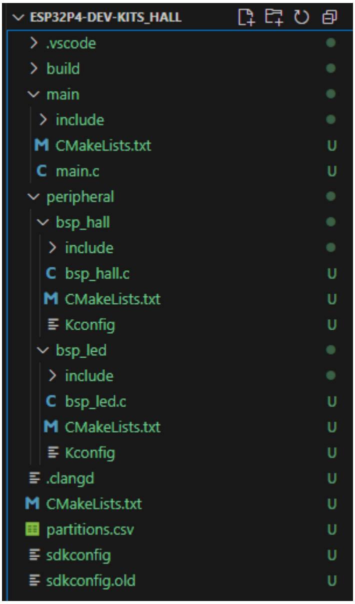
HALL SENSOR 9



1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



In the **ESP32P4-dev-kits_hall** example, a new **bsp_hall** folder was created under the **ESP32P4-dev-kits_hall\peripheral** directory. Within the **bsp_hall** directory, a new include folder, **CMakeLists.txt** file, and **Kconfig** file were added. The **bsp_hall** folder stores the **bsp_hall.c** driver file, the include folder holds the **bsp_hall.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling the project to utilize the HALL driver functionality. The **Kconfig** file loads the entire driver configuration, including GPIO pin definitions, into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 HALL Driver Code

Here we will only explain the core code. For detailed source code, please refer to the corresponding source code for this experiment in the code materials.

The HALL driver source code consists of two files: **bsp_hall.c** and **bsp_hall.h**.

Below, we will first analyze the program in **bsp_hall.h**: it defines the HALL pin and declares functions.

/* Header file references */

```
/*————————————————————————————————Header file declaration————————————————————————————————*/
#include "esp_attr.h"    //API Function References Related to Memory Configuration
#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 "driver/gpio.h" //References for GPIO Function-related API Functions
/*————————————————————————————————Header file declaration end————————————————————————————————*/
```

/* Pin Definitions and Function Declarations */

```
#ifdef CONFIG_BSP_HALL_ENABLED

#define HALL_GPIO CONFIG_HALL_GPIO // HALL GPIO
esp_err_t hall_init();             // Initialize the GPIO Pin of the HALL
bool get_hall_status();            // Get the status for the HALL

#endif
```

Next, we'll analyze the code in **bsp_hall.c**: the initialization configuration and function code for the HALL pin.

/* Initialization function hall_init */

```
esp_err_t hall_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << HALL_GPIO, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_INPUT,           /* GPIO mode: set input/output mode                     */
        .pull_up_en = false,               /* GPIO pull-up                                         */
        .pull_down_en = false,             /* GPIO pull-down                                       */
        .intr_type = GPIO_INTR_ANYEDGE,    /* GPIO interrupt type                                  */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    err = gpio_isr_handler_add(HALL_GPIO, HALL_ISR, (void *)HALL_GPIO); /*Add ISR handler for the corresponding GPIO pin*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

In the hall_init function, the member variables of the **gpio_config_t** structure are first configured with parameters. Subsequently, the **gpio_config** function is called to initialize the GPIO using these configuration parameters. Finally, the **gpio_isr_handler_add** function is used to register the interrupt callback function and bind it to the corresponding GPIO. Here, the GPIO mode is set to input mode to read the IO port level status. The interrupt type is configured as full-edge (triggering the interrupt callback function on either a rising or falling edge).

/* HALL pin interrupt callback function HALL_ISR */

```c
static void IRAM_ATTR HALL_ISR(void *arg) /*Hall Pin's Full-Edge Interrupt Handler*/
{
    if ((arg == (void *)HALL_GPIO) && (!gpio_get_level(HALL_GPIO)))
        hall_state = true;
    else if ((arg == (void *)HALL_GPIO) && (gpio_get_level(HALL_GPIO)))
        hall_state = false;
}
```

In the **HALL_ISR** function, the interrupt flag is set by checking the pin number that triggered the interrupt and the current pin level state. (true indicates a falling edge, meaning the magnet is approaching; false indicates a rising edge, meaning the magnet is moving away.) (static bool hall_state = false; The interrupt flag type definition should be a global variable.)

/* Function get_hall_status to retrieve the HALL status */

```c
bool get_hall_status() /*Get the state of the Hall sensor*/
{
    return hall_state;
}
```

In the **get_hall_status** function, return the interrupt flag.

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the **sdkconfig** file, enabling certain parameter settings to be modified through a graphical interface. Here, the number 7 represents GPIO_NUM_7.

```
menu "BSP HALL Setup"

    config BSP_HALL_ENABLED
        bool "Enable HALL"
        default n

    if BSP_HALL_ENABLED
        config HALL_GPIO
            int "GPIO For HALL"
            default 7
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example primarily relies on the **bsp_hall** driver. To successfully call the contents of the **bsp_hall** folder within the main function, you must configure the **CMakeLists.txt** file located in the **bsp_hall** folder. The configuration details are as follows:

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

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

In this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver libraries. Then, these settings are registered with the build system using the **idf_component_register** command, enabling the project to utilize the bsp_hall driver functionality.

1.3.4 main folder

The main folder serves as the core directory for program execution. It contains the main function executable main.c and the main.h header file located within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The main.h file primarily references required header files: functions utilizing the **bsp_hall** driver require the **bsp_hall** header file, while functions using the **bsp_led** driver require the **bsp_led** header file.

Below is an analysis of the main.c program: system initialization and execution of LED and Hall sensor functions.

```c
    err = gpio_install_isr_service(0); /*Install the GPIO driver's ETS_GPIO_INTR_SOURCE ISR handler service, which allows per-pin GPIO interrupt handlers*/
    if (err != ESP_OK)
        init_fail("gpio isr service", err);
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_HALL_ENABLED
    err = hall_init(); /*HALL Driver Initialization*/
    if (err != ESP_OK)
        init_fail("hall", err);
#endif
```

This code resides within the init function, which is used to store initialization functions that need to be called and to evaluate their return values. If the return status is not ESP_OK, the code will print an error message and halt further execution. It is worth noting that we have added the **gpio_install_isr_service** function to register an interrupt group for all GPIO interrupts.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_HALL_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    bool state = false;      /*HALL status*/
    bool last_state = false; /*Previous state of the HALL*/
    if (get_hall_status())   /*First judgment of HALL state*/
        set_led_state(false);
    else
        set_led_state(true);
    while (1)
    {
        state = get_hall_status();                      /*Get the current status feedback from the HALL*/
        if ((state) && (state != last_state))           /*If the previous Hall sensor state was "magnet moving away" and the current Hall sensor state is "magnet approaching"*/
            set_led_state(false);                       /*Set the Corresponding Output Level of GPIO*/
        else if ((!state) && (state != last_state))     /*If the previous Hall sensor state was "magnet approaching" and the current Hall sensor state is "magnet moving away"*/
            set_led_state(true);                        /*Set the Corresponding Output Level of GPIO*/
        last_state = state;                             /*Save the previous HALL state*/
        vTaskDelay(10 / portTICK_PERIOD_MS);            /*Delay 10ms*/
    }
#endif
#endif
}
```

In the **app_main** function, initialize the HALL current state variable and past state variable. Then use the first HALL state to determine the initial state of the LED. Finally, create a loop that repeats the following: check once every 10ms delay. The function for obtaining the HALL state retrieves the current state and compares it with the previous state. If the state changes, it executes the function in the **bsp_led** driver that sets the LED state. This function takes a parameter to set the LED level (low level turns the LED on, high level turns it off). The current HALL state is then assigned to the past state variable for preservation. The specific functionality is: The LED illuminates when the magnet approaches the Hall sensor. The LED turns off when the magnet moves away from the Hall sensor.

1.3.5 CMkaLists.txt file

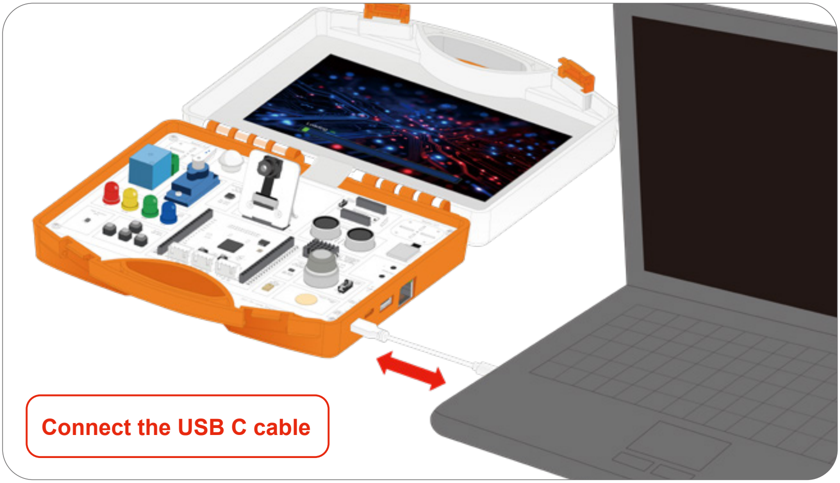
To successfully call the contents of the **bsp_led** and **bsp_hall** folders within the main function, you must configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led bsp_hall)
```

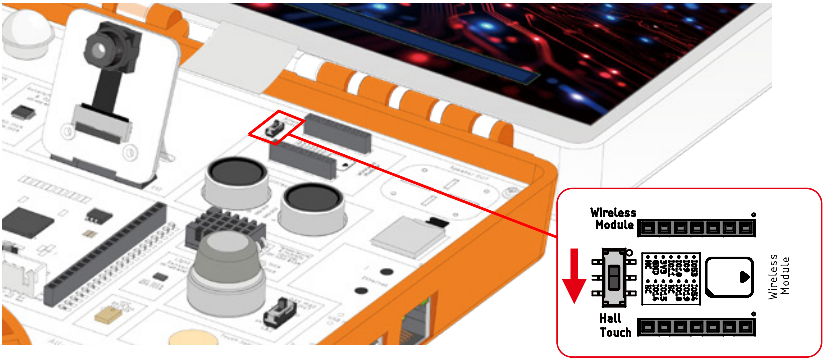
First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link **bsp_led** and **bsp_hall**. Then, these settings are registered with the build system using the **idf_component_register** command, enabling the main function to utilize these driver features.

1.4 Programming procedure

Connect the P4 device to the computer via USB

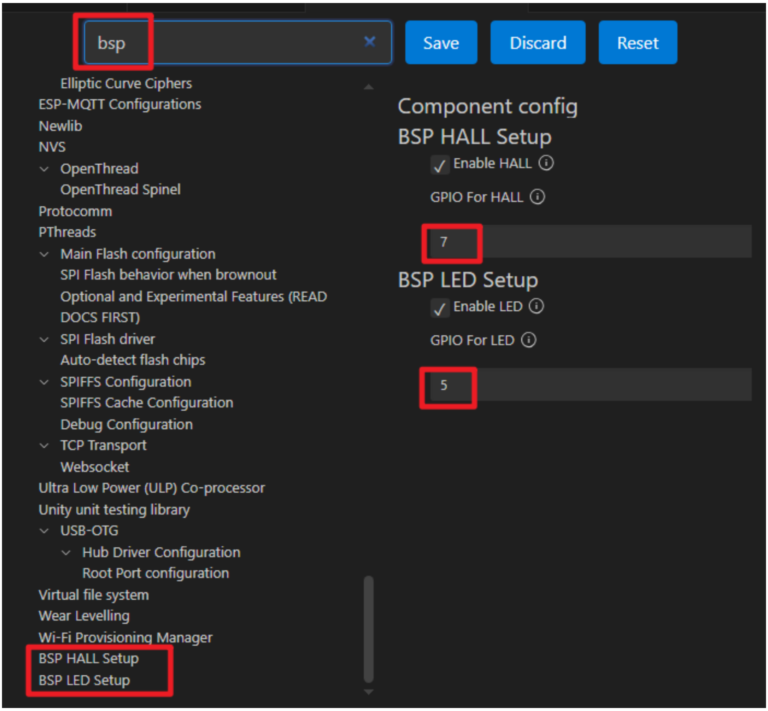


When using touch and Hall sensors, the toggle switch near the wireless module must be set to the Hall and touch position.

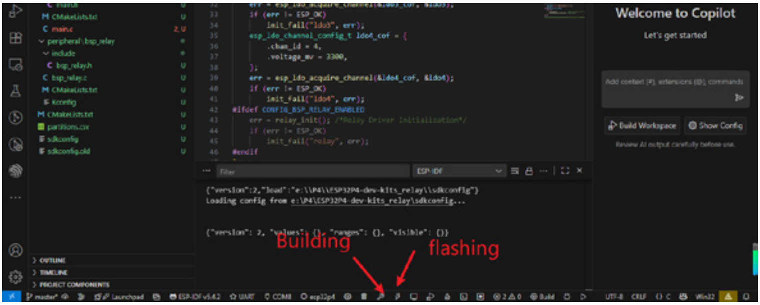


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, requiring the hall and led pins to be reconfigured.



1.4.3 Click Compile. Upon successful compilation, click Download.

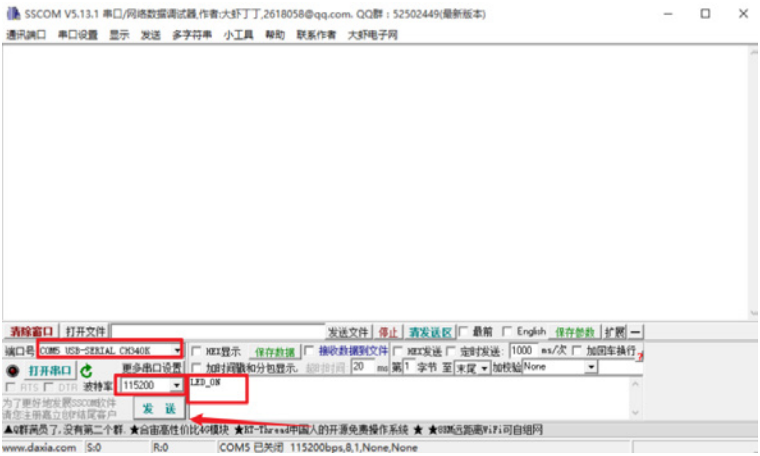


Lesson 6 - Serial LED Control

Introduction

This chapter's tutorial introduces the UART application of the ESP32-P4, utilising serial communication routines to aid understanding of its fundamental functionality. As one of the most prevalent communication methods in embedded development, UART enables developers to rapidly implement data exchange between development boards, PCs, and peripheral modules, laying the groundwork for more complex communication projects.

Project Demonstration Effect



Serial Port Tool Download Address：

https://drive.google.com/drive/folders/1gNItP4DU5yNUgdyyoi10XiRMDNgYmBAf?usp=sharing

This chapter is divided into the following subsections

- 1.1 Introduction to UART
- 1.2 Hardware Design
- 1.3 Programme Design
- 1.4 Download and Verification

1.1 UART Introduction

1.1.1 UART Introduction

UART (Universal Asynchronous Receiver/Transmitter) is a common serial communication protocol that employs asynchronous data transmission. It requires no additional clock signal and accomplishes data transfer using only two signal lines: TX (transmit) and RX (receive).

The ESP32-P4 chip incorporates multiple UART controllers, featuring the following characteristics:
① Multi-channel support: The ESP32-P4 provides up to five UART interfaces, enabling simultaneous communication with multiple peripherals.
② Flexible baud rate: UART supports baud rate configurations ranging from 300bps to 5Mbps, accommodating diverse application scenarios.
③ Hardware FIFO: The UART incorporates internal FIFO buffering, reducing CPU load

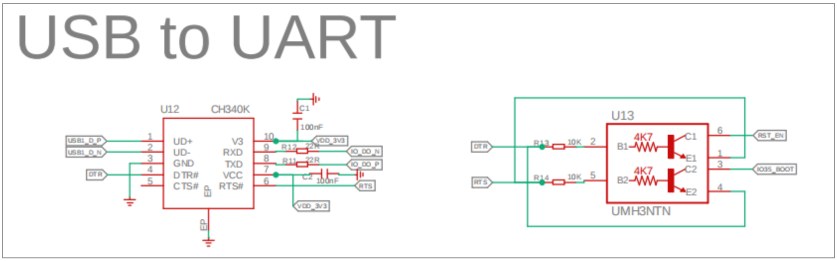
for communication data processing and enhancing efficiency.
④ Interrupt support: The UART interface supports interrupt events such as transmission completion and reception completion, making it suitable for real-time communication.
⑤ Strong compatibility: The UART protocol is straightforward and widely employed in applications including GPS modules, Bluetooth modules, sensors, and debugging printouts.

1.1.2 UART Operating Principle

1.1.2 UART Operating Principle
UART communication transmits data in bit units, typically employing an 8-bit data format with 1 start bit and 1 stop bit. Some applications additionally incorporate a parity bit.
The data frame format is illustrated below:
Start bit | Data bits (D0–D7) | Parity bit (optional) | Stop bit
Brief operational sequence:
1) When the host transmits data to the slave, the TX pin outputs a signal level;
2) The RX pin receives this signal and decodes each bit within the agreed baud rate time interval;
3) The complete byte is ultimately reconstructed, enabling point-to-point communication.
Unlike controlling LEDs via GPIO, UART places greater emphasis on data format and timing. Consequently, it is essential to ensure that the baud rate at both the transmitting and receiving ends is consistent; otherwise, garbled data will occur.

1.2 Hardware design

In this example, we utilise the UART0 of the ESP32-P4 development board to communicate with a PC. The hardware connections are as follows:
TXD0 → USB-to-serial converter chip → Computer serial terminal software (e.g., SecureCRT, XCOM)
RXD0 → USB-to-serial converter → computer serial terminal software
GND ↔ Common ground
The ESP32-P4's UART0 is typically pre-connected to the on-board USB-to-serial converter chip. Users require no additional wiring; communication is achieved using a single Type-C data cable.



Schematic diagram

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.

Within the **ESP32P4-dev-kits_uart** example, a new folder named **bsp_uart** has been created under the **ESP32P4-dev-kits_uart\peripheral** directory. Within the **bsp_uart** path, a new include folder, a **CMakeLists.txt** file, and a Kconfig file have been established. The **bsp_uart** folder houses the **bsp_uart.c** driver file, the include folder contains the **bsp_uart.h** header file, and the CMakeLists.txt file integrates the driver into the build system, enabling the project to utilise the HALL driver functionality. The Kconfig file loads the entire driver configuration, including GPIO pin definitions, into the sdkconfig file within the IDF platform (configurable via the graphical interface).

1.3.1 UART Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The UART driver source code comprises two files: bsp_uart.c and bsp_uart.h.

Below we shall first analyse the bsp_uart.h programme: it contains relevant definitions for the UART pins and function declarations.

/* Header file references */

```c
esp_err_t uart_init()
{
    esp_err_t err = ESP_OK;
    const uart_config_t uart_config = {
        .baud_rate = 115200,                  /*UART baud rate*/
        .data_bits = UART_DATA_8_BITS,        /*UART byte size*/
        .parity = UART_PARITY_DISABLE,        /*UART parity mode*/
        .stop_bits = UART_STOP_BITS_1,        /* UART stop bits*/
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE, /*UART HW flow control mode (cts/rts)*/
        .source_clk = UART_SCLK_DEFAULT,      /*UART source clock selection*/
    };
    err = uart_driver_install(UART_NUM_0, 1024 * 2, 0, 0, NULL, 0); /*Install UART driver and set the UART to the default configuration*/
    if (err != ESP_OK)
    {
        UART_ERROR("uart driver install fail");
        return err;
    }
    err = uart_param_config(UART_NUM_0, &uart_config); /*Set UART configuration parameters*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the **uart_init** function, the various member variables of the **uart_config_ structure** are first configured with parameters. Subsequently, the **uart_driver_install** function is invoked to register the corresponding serial port controller and buffer configuration. Finally, the **uart_param_config** function is employed to assign the configuration parameters to the relevant UART controller.

It is worth noting that here we are configuring UART_NUM_0, which is the default serial port programmed during the initial burn. This utilises the default pins GPIO_NUM_37 and GPIO_NUM_38, so no additional configuration is required. Should you wish to use a different pin, you may call the uart_set_pin function to set the corresponding pin number.

1.3.2 Kconfig file

The primary function of this file is to incorporate the requisite configuration into the sdkconfig file, enabling certain parameter adjustments to be made via a graphical interface. Pin configuration is not required here; this serves solely as a macro definition configuration for enabling features.

```
menu "BSP UART Setup"

    config BSP_UART_ENABLED
        bool "Enable UART"
        default n

endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the bsp_uart driver. To successfully call the contents of the bsp_uart folder within the main function, it is necessary to configure the CMakeLists.txt file located within the bsp_uart folder. The configuration details are as follows:

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

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

Within this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the bsp_uart driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the header file main.h within the include folder. Add the main folder to the CMakeLists.txt file of the build system.

The main.h file primarily references required header files: functions utilising the bsp_uart driver necessitate inclusion of the bsp_uart header file, while those employing the bsp_led driver require the bsp_led header file.

Below is an analysis of the main.c programme: system initialisation and execution of LED and UART functionality.

```
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_UART_ENABLED
    err = uart_init(); /*Uart Driver Initialization*/
    if (err != ESP_OK)
        init_fail("uart", err);
#endif
```

This code resides within the init function, which is employed to store initialisation functions requiring invocation and to evaluate the outcome of such initialisation. Should the returned status not be ESP_OK, the code will display an error message and cease further execution.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_UART_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    int Rx_Bytes = 0;                          /*Data length read from the serial port buffer*/
    char *data = (char *)malloc(512 + 1); /*Storage space for data read from the serial port buffer*/
    while (1)
    {
        Rx_Bytes = uart_read_bytes(UART_NUM_0, data, 512, pdMS_TO_TICKS(1000)); /*UART read bytes from UART buffer*/
        if (Rx_Bytes > 0)
        {
            data[Rx_Bytes] = 0; /*Append \0 at the end of the string*/
            MAIN_INFO("read data: %s, length: %d", data, Rx_Bytes);
            if (strcmp(data, CMD_LED_ON) == 0) /*Compare whether the string matches the preset command*/
            {
                MAIN_INFO("Received the LED_ON command");
                set_led_state(false); /*Set the Corresponding Output Level of GPIO*/
            }
            else if (strcmp(data, CMD_LED_OFF) == 0) /*Compare whether the string matches the preset command*/
            {
                MAIN_INFO("Received the LED_OFF command");
                set_led_state(true); /*Set the Corresponding Output Level of GPIO*/
            }
            else
            {
                MAIN_ERROR("Unknown Command: %s", data);
            }
        }
    }
#endif
#endif
}
```

Within the `app_main` function, initialise the variable for the number of bytes received via the serial port and the pointer for the received data, allocating a specific amount of memory space. Subsequently, establish a loop to repeatedly execute:

(1) Use the uart_read_bytes function to read 512 bytes of data, with a timeout set to 1 second. This means data within the buffer is read within 1 second; if the timeout is exceeded, the read operation automatically terminates and returns the number of bytes read.

(2) When the number of bytes read is greater than zero (indicating data presence in the buffer), append a null character "\0" to the end of the data based on the total bytes read. Subsequently, compare the data against a predefined command string. If the strcmp function returns zero, the strings are identical. The corresponding command then controls the LED (LED_ON string turns it on, LED_OFF string turns it off). If the received string does not match the preset, the serial port prints the error message LOG.

1.3.5 CMkaLists.txt file

To successfully call the contents of the bsp_led and bsp_uart folders within the main function, it is necessary to configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led bsp_uart)
```

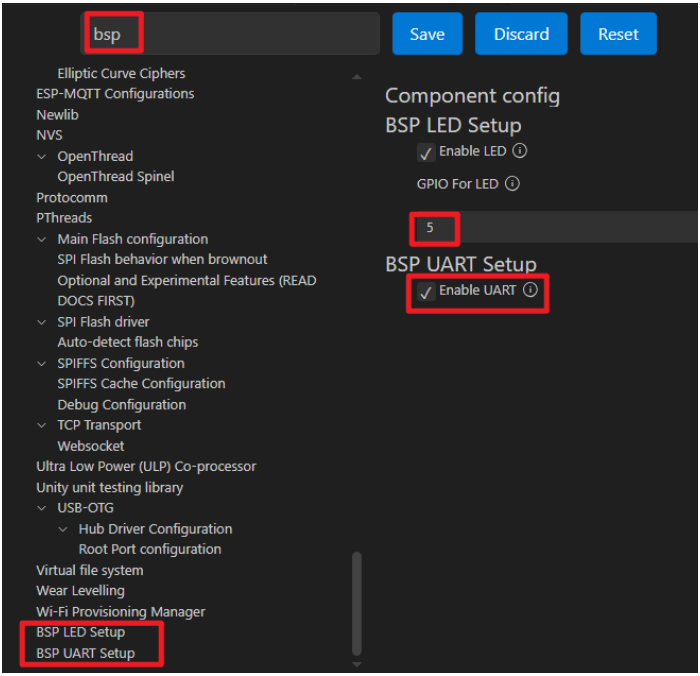
First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link bsp_led and bsp_uart. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the main function to utilise these driver capabilities.

1.4 Programming procedure

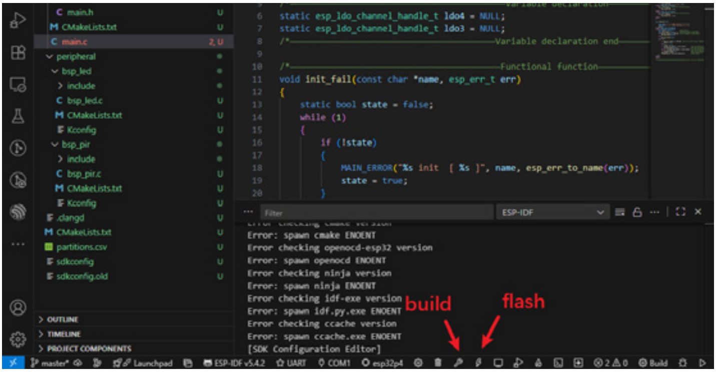
Connect the P4 device to the computer via USB

1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, Simply reconfigure the LED pins and enable the UART interface.



1.4.3 Click Compile. Once compilation is successful, click Download.

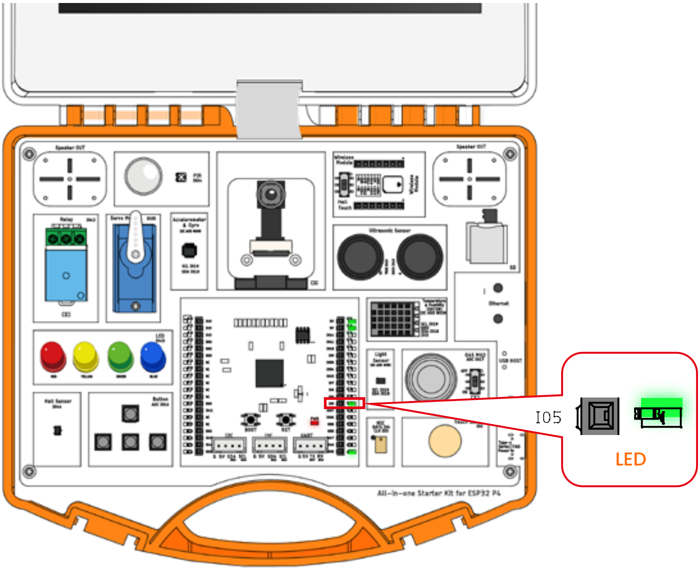


Lesson 7 - Timer LED Blink

Introduction

This chapter's tutorial introduces the Timer application for the ESP32-P4, using an example routine to flash an LED at timed intervals to help understand its fundamental functionality. As a core peripheral in embedded systems, the timer can precisely generate time interval signals and is widely used in scenarios such as task scheduling, event triggering, and PWM control. In this chapter, we shall control LED flashing via the timer to help readers master its fundamental usage, laying the groundwork for more complex projects.

Project Demonstration Effect



This chapter is divided into the following subsections

- 1.1 Timer Introduction
- 1.2 Hardware Design
- 1.3 Programme Design
- 1.4 Download and Verification

1.1 Timer Introduction

1.1.1 Timer Introduction

The ESP32-P4 chip integrates multiple General Purpose Timers with the following characteristics:

① Multiplexed Timers: Supports multiple independent timer groups, enabling simultaneous execution of multiple timing tasks;

② High Precision: Timers operate based on the hardware clock, achieving microsecond-level accuracy;

③ Interrupt Functionality: Timers can trigger interrupts upon reaching preset values to execute specific tasks;

④ Flexible configuration: Supports both periodic and one-shot modes to accommodate diverse application requirements;

⑤ Extensive applications: Commonly employed for LED blinking, task scheduling, event counting, timeout detection, PWM, and similar scenarios.

At its core, a timer functions as a hardware counter that increments or decrements according to a preset clock frequency. Upon reaching the configured value, it triggers an event (such as an interrupt), thereby executing user-defined tasks.

1.1.2 Timer Working Principle

The operation of the timer can be divided into the following steps:

1) Configure the clock source and division factor to determine the counting frequency;

2) Set the timing period (i.e., how many count pulses trigger an event);

3) Start the timer to begin counting;

4) When the counter reaches the preset value, trigger an interrupt;

5) Execute the task within the interrupt service routine (e.g., toggle the LED state).

The process is illustrated below:

System clock → **Frequency divider** → **Counter accumulation** → **Matching value** → **Trigger interrupt** → **Task execution**

1.2 Hardware design

This experiment continues to utilise the on-board LED on the ESP32-P4 development board as the output test subject.

GPIO pin: The on-board LED is connected to GPIO5.

The circuit structure remains identical to the previous chapter, requiring no additional hardware connections.

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.

Within the **ESP32P4-dev-kits_timer** example, a new folder named **bsp_timer** has been created under the **ESP32P4-dev-kits_timer\peripheral** directory. Within the **bsp_timer** directory, a new include folder, a **CMakeLists.txt** file, and a **Kconfig** file have been created. The **bsp_timer** folder houses the **bsp_timer.c** driver file, the include folder contains the **bsp_timer.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling the project to **utilise timer** driver functionality. The **Kconfig** file loads the entire driver alongside GPIO pin definitions into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 Timer Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The Timer driver source code comprises two files: **bsp_timer.c** and **bsp_timer.h**.

Below we shall first analyse the **bsp_timer.h** programme: it declares functions for the Timer.

/* Header file references */

```
/*————————————————————————————————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_timer.h" //References for high-precision timers Function-related API Functions
/*——————————————————————————————Header file declaration end——————————————————————————————*/
```

/* Function declarations and macro definition declarations */

```
#ifdef CONFIG_BSP_TIMER_ENABLED
esp_err_t timer_init();                           /*Initialize the high-precision timer*/
esp_err_t start_timer(uint64_t time);             /*Start timer*/
esp_err_t stop_timer(esp_timer_handle_t timer);   /*Stop timer*/
bool get_timer_flag();                            /*Get the timer flag*/
void reset_timer_flag();                          /*Reset the timer flag*/
#endif
```

Next, we shall analyse the **bsp_timer.c** programme: initialising and configuring the timer, along with various callable API functions.

/* timer_init */

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

    /*Timer configuration passed to esp_timer_create()*/
    const esp_timer_create_args_t timer_args = {
        .callback = &periodic_timer_callback, /* Callback function to execute when timer expires*/
        .arg = NULL,                          /*Argument to pass to callback*/
        .name = "Mytimer"                     /*Timer name*/
    };

    err = esp_timer_create(&timer_args, &periodic_timer); /*Create an esp_timer instance*/
    if (err != ESP_OK)
        return err;

    return err;
}
```

Within the timer_init function, the parameters for each member variable of the **esp_timer_create_args_t** structure are first configured. (The most crucial element here is the callback execution function, which will be detailed later.) Subsequently, the **esp_timer_create** function is invoked to establish the timer controller, receiving a handle via the **esp_timer_handle_t** structure. This handle facilitates subsequent operations on the controller.

It is worth noting that there is no need to call the **esp_timer_init()** function here to initialise the timer controller. This functionality runs automatically upon chip power-up. Re-initialisation of the controller is only required when other timers have been used.

/* periodic_timer_callback */

```
void periodic_timer_callback(void *arg) /*Timer Interrupt Handler*/
{
    if (timer_flag == false)
        timer_flag = true;
}
```

Within the timer callback function, we set the timer flag **timer_flag** to a true value, indicating that the timer's set duration has elapsed. Other functions may read this flag to determine whether the required timing period has been reached. Subsequently, the flag is cleared to ensure the timer can be triggered again.

/* Retrieve timer flag get_timer_flag and reset timer flag reset_timer_flag */

```c
bool get_timer_flag() /*Get the timer flag*/
{
    return timer_flag;
}

void reset_timer_flag() /*Reset the timer flag*/
{
    timer_flag = false;
}
```

/* Start the timer function start_timer */

```c
esp_err_t start_timer(uint64_t time)
{
    esp_err_t err = ESP_OK;
    err = esp_timer_start_periodic(periodic_timer, time); /*Start a periodic timer(period: us)*/
    if (err != ESP_OK)
    {
        TIMER_ERROR("Start a periodic timer error");
        return err;
    }
    return err;
}
```

Within this function, the **esp_timer_start_periodic** function is invoked to initiate the periodic timer (i.e., for cyclical execution). The input parameter specifies the timing interval in microseconds. Adjusting this function's value enables timing at different durations.

1.3.2 Kconfig file

The primary function of this file is to incorporate the requisite configuration into the **sdkconfig** file, enabling certain parameter adjustments to be made via a graphical interface. Pin configuration is not required here; this serves solely as a macro definition configuration for enabling functionality.

```
menu "BSP Timer Setup"

    config BSP_TIMER_ENABLED
        bool "Enable Timer"
        default n

endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_timer** driver. To successfully invoke the contents of the **bsp_timer** folder within the main function, it is necessary to configure the CMakeLists.txt file located within the **bsp_timer** folder. The configuration details are as follows:

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

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

Within this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the bsp_timer driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the header file main.h within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The main.h file primarily references required header files: functions utilising the **bsp_timer** driver necessitate inclusion of the **bsp_timer** header file, while those employing the **bsp_led** driver require the **bsp_led** header file.

Below is an analysis of the main.c programme: system initialisation and execution of LED and timer functions.

```
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*LED Driver Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
#endif
#ifdef CONFIG_BSP_TIMER_ENABLED
    err = timer_init(); /*Timer Driver Initialization*/
    if (err != ESP_OK)
        init_fail("timer", err);
#endif
```

This code resides within the init function, which is employed to store initialisation functions requiring invocation and to evaluate the outcome of such initialisation. Should the returned status not be **ESP_OK**, the code will display an error message and cease further execution.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_TIMER_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    start_timer(3000000); /*Start a periodic timer -3s*/
    while (1)
    {
        if (get_timer_flag()) /*Get the timer flag*/
        {
            led_toggle();
            reset_timer_flag(); /*Reset the timer flag*/
        }
        vTaskDelay(pdTICKS_TO_MS(10));
    }
#endif
#endif
}
```

Within the `**app_main**` function, the `**start_timer**` function is employed to initiate a periodic timer with a cycle of 3 seconds.
Subsequently, a loop is established. Within this loop, the timer flag is checked every 10 milliseconds. Should the return value be true, the LED flipping function is executed (refer to the code from Lesson One). Following execution, the timer flag is cleared to prepare for the next iteration of the check.

1.3.5 CMkaLists.txt file

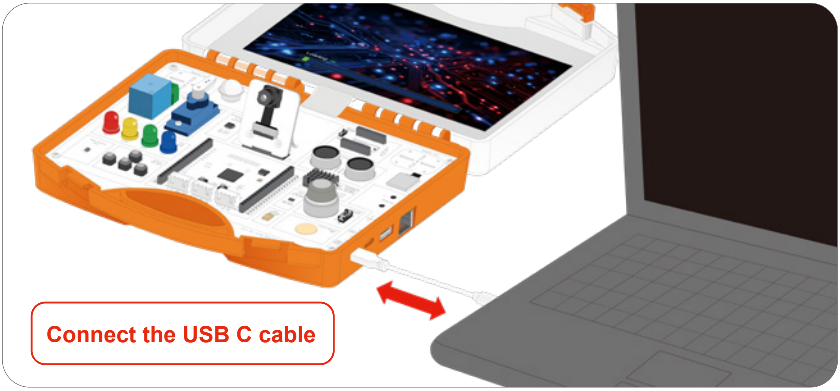
To successfully call the contents of the **bsp_led** and **bsp_timer** folders within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led bsp_timer)
```

First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link **bsp_led** and **bsp_timer**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver functionalities.

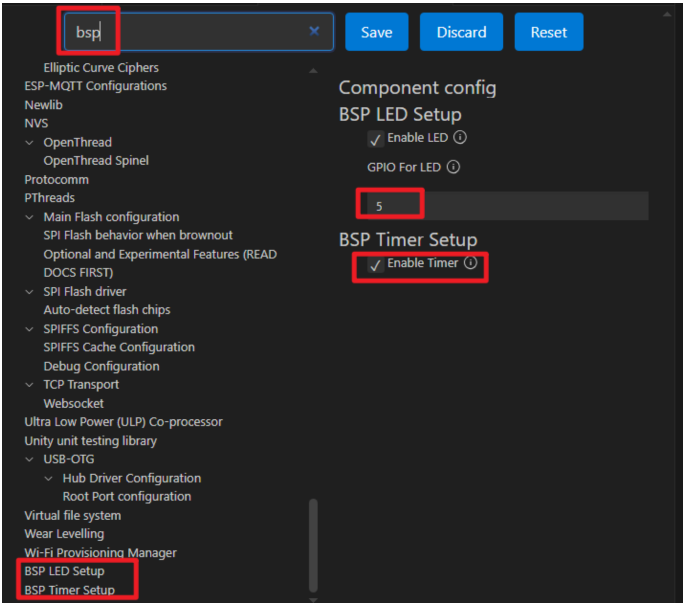
1.4 Programming procedure

Connect the P4 device to the computer via USB

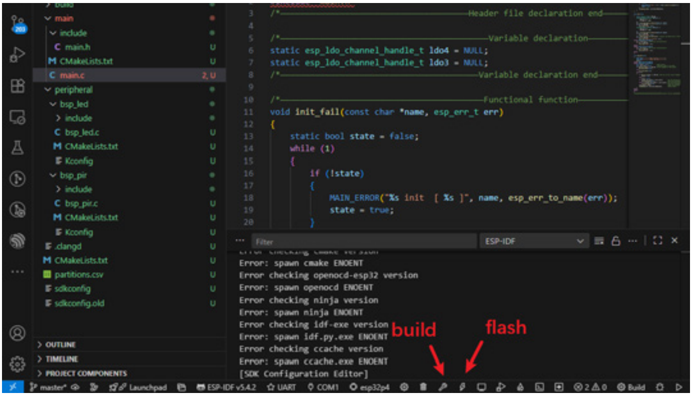


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, Simply reconfigure the LED pins and enable the Timer interface.



1.4.3 Click Compile. Once compilation is successful, click Download.



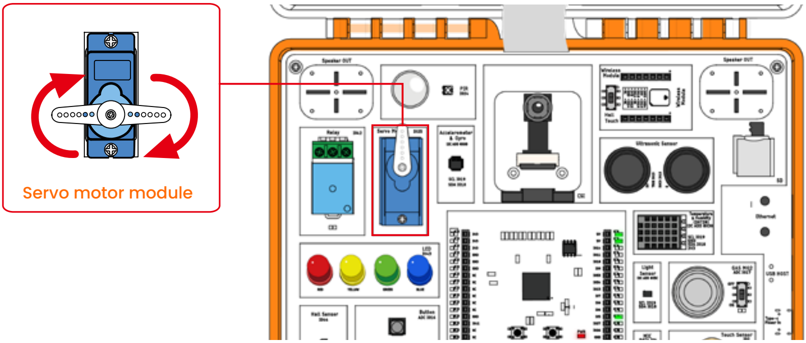
Lesson 8 - PWM Servo Control

Introduction

This chapter's tutorial introduces the PWM output application of the ESP32-P4, using a servo control example to help you understand the fundamental functions of PWM.

Servos, as common actuators, are indispensable components in robots, remote-controlled models, and automated devices. Through this chapter, readers will learn how to generate PWM waveforms using the ESP32-P4's GPIO pins to drive a 180-degree servo, laying the groundwork for more complex motion control projects.

Project Demonstration Effect



This chapter is divided into the following subsections

1.1 Servo Motors and PWM Introduction

1.1.1 PWM Introduction

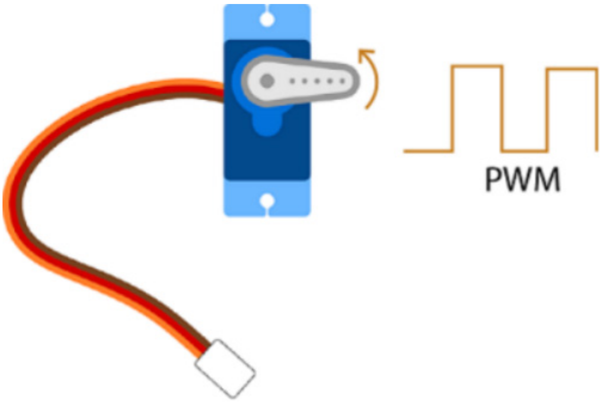
Pulse Width Modulation (PWM) is a common method of digital signal control that regulates the average voltage of an output signal by controlling the ratio of the high-level duration to the cycle period (duty cycle). The ESP32-P4 provides a rich array of PWM channels suitable for controlling devices such as motors, servos, and backlighting.

Key features:

① Flexibility: Programmable duty cycle and frequency, suitable for diverse applications ranging from LED dimming to motor speed control.
② High precision: ESP32-P4's PWM incorporates high-resolution timers enabling smooth control.
③ Multi-channel capability: Simultaneously drives multiple servos or motors to fulfil complex motion requirements.

1.1.2 Introduction to Servo Motors

A servo is an angle-controlled motor comprising a DC motor, gears, and position feedback circuitry.



Standard 180° servo: Angle-controlled, typically 0° to 180°.

This V1.1 tutorial employs a 180-degree servo, controlled as follows:

PWM cycle: Typically 20ms (50Hz).

Duty cycle: Determines the rotation angle.

approximately 0.5 milliseconds 0°.

approximately 1.5 milliseconds 90°.

approximately 2.5 milliseconds 180°.

1.1.3 Principles of Servo Motor Drive

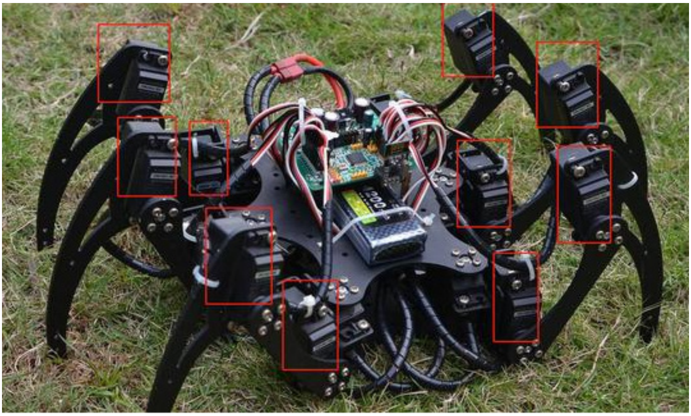
Servo control is essentially achieved through PWM (Pulse Width Modulation):

The ESP32-P4 outputs a PWM wave with a period of 20 milliseconds;

different pulse widths (0.5–2.5 ms) correspond to different rotation angles; the servo's internal circuitry adjusts the motor's operation based on this signal, thereby controlling the rotation angle.

Unlike LED control (which involves simple on/off states), a servo motor requires a continuous PWM signal to maintain its position.

Servo motors are widely used in a variety of applications that require angle control.



1.2 Hardware Design

In this example, we utilise an ESP32-P4 development board paired with a 180-degree servo motor.

The connection configuration is as follows:

Servo VCC → 5V power supply (or development board 5V)

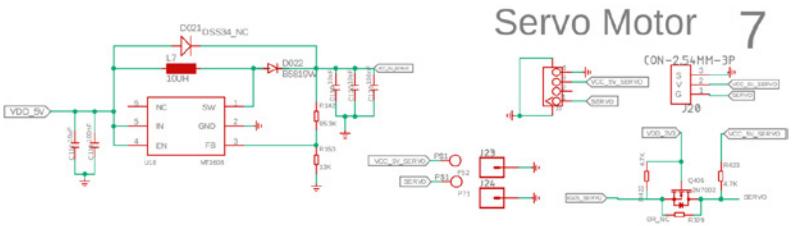
Servo GND → Development board GND

Servo signal pin → An available PWM pin on the ESP32-P4 (e.g., GPIO25)

Important notes:

Servos operate at 5V, while the ESP32-P4 outputs 3.3V signals. However, the vast majority of servos are compatible.

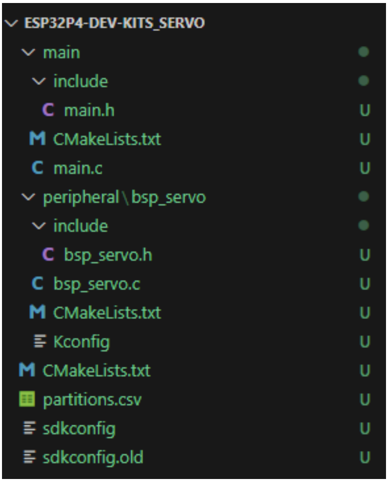
Should multiple servos operate simultaneously, an additional power supply is required to prevent insufficient current from the development board's USB port.



1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_servo** example, a new folder named **bsp_servo** has been created under the **ESP32P4-dev-kits_servo\peripheral** directory. Within the **bsp_servo** directory, a new include folder, a **CMakeLists.txt** file, and a Kconfig file have been created. The bsp_servo folder houses the **bsp_servo.c** driver file, the include folder contains the **bsp_servo.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling the project to utilise servo motor control functionality. The Kconfig file loads the entire driver configuration, including GPIO pin definitions, into the sdkconfig file within the IDF platform (configurable via the graphical interface).

1.3.1 SERVO Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The SERVO driver source code comprises two files: **bsp_servo.c** and **bsp_servo.h**.

Below we shall first analyse the **bsp_servo.h** programme: it contains relevant definitions for servo pins and function declarations.

/* Header file references */

```c
/*------------------------------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/ledc.h"
/*------------------------------Header file declaration end------------------------------*/
```

/* Function declarations and macro definition declarations */

```c
#ifdef CONFIG_BSP_SERVO_ENABLED

#define SERVO_GPIO CONFIG_SERVO_GPIO
#define SERVO_MIN_PULSEWIDTH_US 500  // The pulse width corresponding to 0 degree
#define SERVO_MAX_PULSEWIDTH_US 2500 // The pulse width corresponding to 180 degree
#define SERVO_MAX_DEGREE 180

esp_err_t servo_init();
esp_err_t set_servo_angle(int degree);
bool parse_angle_command(const char *str, int *out_angle);
#endif
/*------------------------------Variable declaration end------------------------------*/
#endif
```

Next, we shall analyse the **bsp_servo.c** programme: initialising and configuring the servo pins, and calling the setup function.

/* Initialisation function servo_init */

```c
esp_err_t servo_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << SERVO_GPIO,
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = false,
        .pull_down_en = false,
        .intr_type = GPIO_INTR_DISABLE,
    };
    err = gpio_config(&gpio_cofig);
    if (err != ESP_OK)
        return err;
    const ledc_timer_config_t servo_timer_config = {
        .clk_cfg = LEDC_USE_PLL_DIV_CLK,
        .duty_resolution = LEDC_TIMER_14_BIT,
        .freq_hz = 50,
        .speed_mode = LEDC_LOW_SPEED_MODE,
        .timer_num = LEDC_TIMER_1,
    };
    const ledc_channel_config_t channel_config = {
        .gpio_num = SERVO_GPIO,
        .speed_mode = LEDC_LOW_SPEED_MODE,
        .channel = LEDC_CHANNEL_1,
        .intr_type = LEDC_INTR_DISABLE,
        .timer_sel = LEDC_TIMER_1,
        .duty = calculate_servo_duty(0),
        .hpoint = 0,
    };
    err = ledc_timer_config(&servo_timer_config);
    if (err != ESP_OK)
        return err;
```

Within the servo_init function, the member variables of the gpio_config_t structure were first configured with parameters. Subsequently, the gpio_config function is invoked to complete the initialisation of the GPIO using these configuration parameters. Following this, the ledc_timer_config_t structure is configured, which sets the parameters for the LEDC timer (frequency, resolution, etc.). Thereafter, the ledc_channel_config_t structure is configured; this structure serves to bind the GPIO port being used to the corresponding LEDC timer and LEDC channel. Finally, the `ledc_timer_config` and `ledc_channel_config` functions are invoked to complete the initialisation of the LEDC controller and its channels.

/* Function to set the servo angle: set_servo_angle*/

```
esp_err_t set_servo_angle(int degree)
{
    esp_err_t err = ESP_OK;
    uint32_t duty = calculate_servo_duty(degree);
    err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1, duty);
    if (err != ESP_OK)
        return err;
    err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_1);
    if (err != ESP_OK)
        return err;
    return err;
}
```

This function takes one parameter: a servo angle (0–180°). It is used to control a standard 180° servo motor using ESP32 LEDC PWM.

The input angle is first converted into a PWM duty cycle by calculate_servo_duty(), mapping the angle to the corresponding pulse width required by the servo (typically 0.5–2.5 ms within a 50 Hz signal).

Then ledc_set_duty() is used to set the PWM duty cycle on the selected LEDC channel, and ledc_update_duty() applies the change to the hardware.

Together, these steps generate the correct PWM signal to drive the servo to the specified angle.

It is worth noting that we are using a 180-degree servo, not a 360-degree servo.

Therefore, the PWM waveform we output controls the servo’s rotation angle, not its rotation speed.

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the sdkconfig file, enabling certain parameter settings to be modified via a graphical interface. Here, 25 refers to GPIO_NUM_25.

```
menu "BSP SERVO Setup"
    config BSP_SERVO_ENABLED
        bool "Enable SERVO config"
        default n

    if BSP_SERVO_ENABLED
        config SERVO_GPIO
            int "GPIO For SERVO"
            default 25
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_servo** driver. To successfully call the contents of the **bsp_servo** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_servo** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to utilise the **bsp_servo** driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the header file main.h within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The main.h file primarily references required header files: functions utilising the bsp_servo driver necessitate inclusion of the **bsp_servo** header file.

Below is an analysis of the **main.c** programme: system initialisation and execution of servo functionality.

```
#ifdef CONFIG_BSP_SERVO_ENABLED
    err = servo_init(); /*Servo Driver Initialization*/
    if (err != ESP_OK)
        init_fail("servo", err);
#endif
```

This code resides within the init function, which is employed to store initialisation functions requiring invocation and to evaluate the outcome of such initialisation. Should the returned status not be ESP_OK, the code will display an error message and cease further execution.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_SERVO_ENABLED
    while (1)
    {
        set_servo_angle(180); /*Set the servo motor 180 degrees*/
        vTaskDelay(2000 / portTICK_PERIOD_MS);
        set_servo_angle(0); /*Set the servo motor 0 degrees*/
        vTaskDelay(2000 / portTICK_PERIOD_MS);
    }
#endif
}
```

Create a loop within the **app_main** function. Within this loop, create a nested loop. This nested loop will sequentially execute the following steps: First, the servo rotates to the 180° position; after a two-second delay, it rotates to the 0° position and pauses for two seconds. This process repeats continuously.

1.3.5 CMkaLists.txt file

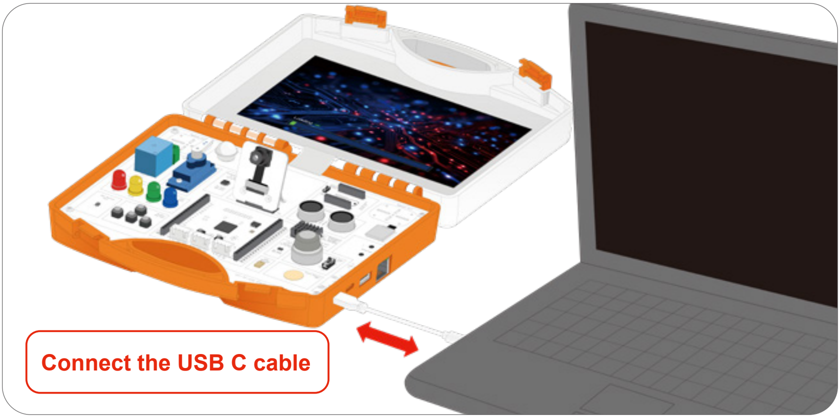
To successfully call the contents of the **bsp_servo** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_servo)
```

First, the directories for source files and header files are defined, along with the required driver library—specifically, the driver library for linking **bsp_servo**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver capabilities.

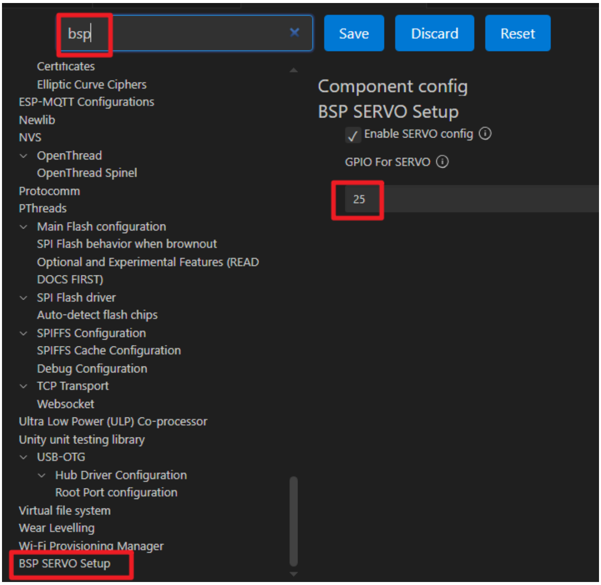
1.4 Programming procedure

Connect the P4 device to the computer via USB

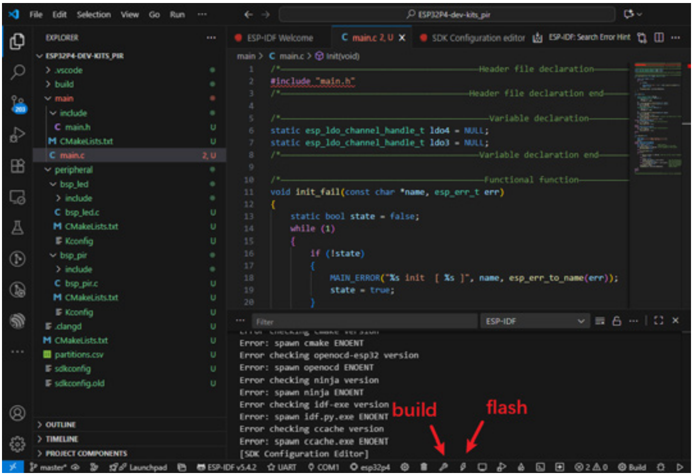


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, requiring only the servo pin to be reconfigured.



1.4.3 Click Compile. Once compilation is successful, click Download.

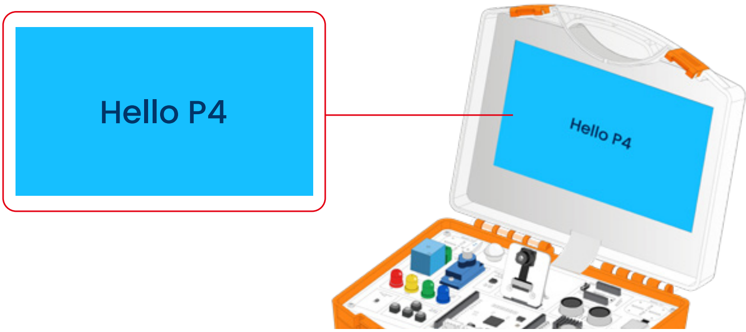


Lesson 9 – LCD Display Hello

Introduction

This chapter's tutorial introduces graphical display applications for the ESP32-P4. Through example routines utilising LVGL (Light and Versatile Graphics Library) combined with the MIPI DSI interface, it aids in understanding its fundamental capabilities. Lighting up the screen and rendering basic graphics serve as classic test cases, enabling readers to gain a straightforward yet comprehensive grasp of display applications on the ESP32-P4. This lays the groundwork for more complex GUI projects in subsequent stages.

Project Demonstration Effect



This chapter is divided into the following subsections

- 1.1 Introduction to LVGL and MIPI DSI
- 1.2 Hardware Design
- 1.3 Software Design
- 1.4 Download and Verification

1.1 Introduction to LVGL and MIPI DSI

1.1.1 LVGL Introduction

LVGL is an open-source embedded GUI development framework characterised by its lightweight, cross-platform, and high-performance features, widely utilised across various MCU and SoC platforms.

Its principal characteristics include:

① Extensive control library: Provides GUI components such as buttons, progress bars, charts, and images for rapid construction of human-machine interfaces;

② Hardware acceleration support: LVGL integrates with the ESP32-P4's 2D accelerator and DMA to significantly enhance rendering speed;

③ Multitasking support: Compatible with RTOS for seamless interface switching and complex logic handling;

④ Customisability: Themes, styles, and fonts are configurable to adapt to diverse product requirements;

⑤ Cross-platform compatibility: Supports deployment from low-end MCUs to high-performance chips, facilitating portability and scalability.

Through LVGL, developers can rapidly implement sophisticated interface designs without requiring direct implementation of underlying graphics algorithms.

1.1.2 MIPI DSI Introduction

MIPI DSI (Display Serial Interface) is a high-speed serial display interface protocol widely employed in smartphone and tablet displays. The ESP32-P4 incorporates an integrated MIPI DSI controller, enabling direct driving of high-resolution displays.

Its key features include:

① High-speed serial communication: Supports data rates exceeding 500 Mbps to 1 Gbps, suitable for high-definition displays;

② Multi-channel support: Selectable 1–4 lane transmission modes to accommodate varying resolution and refresh rate requirements;

③ Low-power design: Utilises differential signal transmission for reduced power consumption and enhanced interference resistance;

④ Excellent compatibility: Supports multiple common MIPI DSI display modules (e.g., 480×800, 720p, 1080p);

⑤ Command and video modes: Supports both command mode (initialisation register writing) and video mode (continuous pixel stream).

Within the ESP32-P4 display example, LVGL serves as the graphics rendering engine while MIPI DSI functions as the hardware transmission interface. Their integration enables rapid screen illumination and interface display.

1.2 Hardware design

Hardware-wise, the ESP32-P4 development board connects to the LCD screen via the MIPI DSI interface. The screen typically incorporates its own power management integrated circuit (PMIC) and backlight control circuitry.

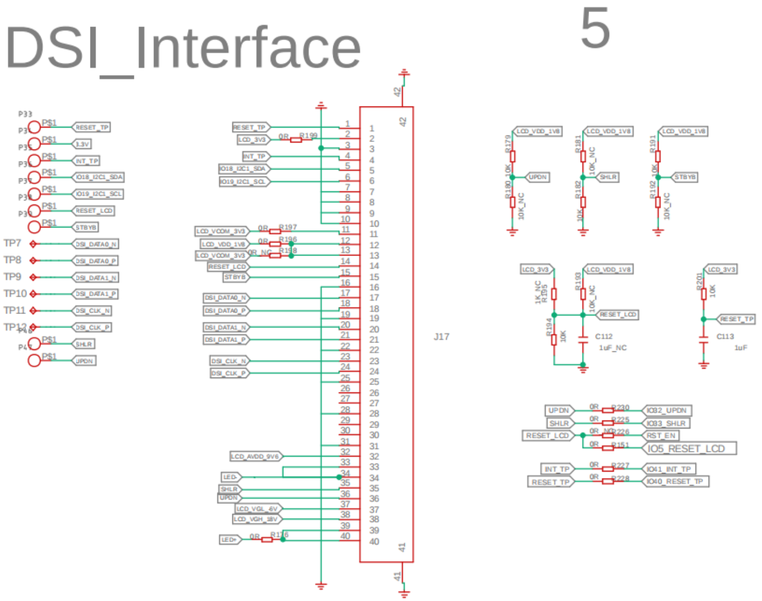
The primary connection relationships are as follows:

ESP32-P4 MIPI DSI Lane0~3 → Display MIPI channels

I2C/SPI → Display touch controller (e.g., GT911, FT5x06)

PWM → Backlight brightness adjustment

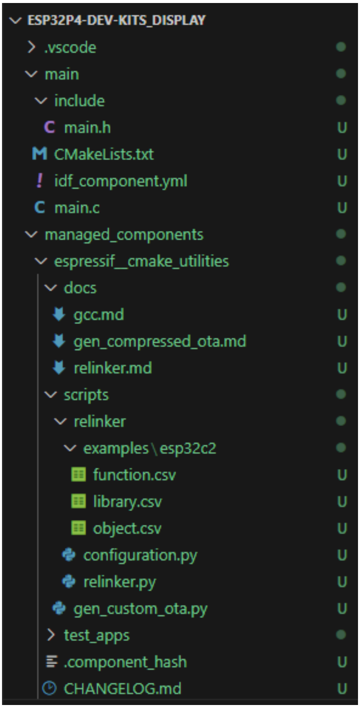
The ESP32-P4 requires only the correct clock supply and initialisation commands to drive the display.



1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_display** example, a new folder named bsp_display has been created under the **ESP32P4-dev-kits_display\peripheral** directory. Within the **bsp_display** path, a new include folder, a **CMakeLists.txt** file, and a **Kconfig** file have been established. The **bsp_display** folder houses the **bsp_display.c** driver file, the include folder contains the **bsp_display.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling the project to utilise the display driver functionality. The **Kconfig** file loads the entire driver configuration, including GPIO pin definitions, into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 Display driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The display driver source code comprises two files: **bsp_display.c** and **bsp_display.h**.

Below we shall first analyse the bsp_display.h programme: it contains relevant definitions for the display pins and function declarations.

/* Header file references */

```
/*----------------------------------------------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----------------------------------------------*/
```

/* Function declarations and macro definitions */

```
#ifdef CONFIG_BSP_DISPLAY_ENABLED

#define V_size CONFIG_V_SIZE                   // Vertical resolution
#define H_size CONFIG_H_SIZE                   // Horizontal resolution
#define BITS_PER_PIXEL CONFIG_BITS_PER_PIXEL // Number of image display bits of the display screen

#define LCD_GPIO_RESET CONFIG_LCD_GPIO_RESET   // LCD Reset GPIO
#define LCD_GPIO_BLIGHT CONFIG_LCD_GPIO_BLIGHT // LCD Blight GPIO
#define LCD_GPIO_UPDN CONFIG_LCD_GPIO_UPDN     // LCD UPDN GPIO
#define LCD_GPIO_SHLR CONFIG_LCD_GPIO_SHLR     // LCD SHLR GPIO
#define BLIGHT_PWM_Hz CONFIG_BLIGHT_PWM_Hz     // LCD Blight PWM GPIO

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

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

Next, we shall analyse the **bsp_display.c** programme: initialising and configuring the display pins, and calling the setup function.

/* Backlight initialisation function blight_init */

```
#ifdef CONFIG_BSP_DISPLAY_ENABLED
static esp_err_t blight_init(void)
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = ((1ULL << LCD_GPIO_RESET) | (1ULL << LCD_GPIO_BLIGHT) | (1ULL << LCD_GPIO_UPDN) | (1ULL << LCD_GPIO_SHLR)), /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_OUTPUT,                                                                                                 /* GPIO mode: set input/output mode                     */
        .pull_up_en = false,                                                                                                      /* GPIO pull-up                                         */
        .pull_down_en = false,                                                                                                    /* GPIO pull-down                                       */
        .intr_type = GPIO_INTR_DISABLE,                                                                                           /* GPIO interrupt type                                  */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;

    const ledc_timer_config_t timer_config = {
        .clk_cfg = LEDC_USE_PLL_DIV_CLK,        /* Configure LEDC source clock from ledc_clk_cfg_t*/
        .duty_resolution = LEDC_TIMER_11_BIT, /*LEDC channel duty resolution*/
        .freq_hz = BLIGHT_PWM_Hz,               /*LEDC timer frequency (Hz)*/
        .speed_mode = LEDC_LOW_SPEED_MODE,      /*LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode*/
        .timer_num = LEDC_TIMER_0,              /* The timer source of channel (0 - LEDC_TIMER_MAX-1)*/
    };
    const ledc_channel_config_t channel_config = {
        .gpio_num = LCD_GPIO_BLIGHT,         /*the LEDC output gpio_num*/
        .speed_mode = LEDC_LOW_SPEED_MODE, /*LEDC speed speed_mode, high-speed mode (only exists on esp32) or low-speed mode*/
        .channel = LEDC_CHANNEL_0,           /*LEDC channel (0 - LEDC_CHANNEL_MAX-1)*/
        .intr_type = LEDC_INTR_DISABLE,      /*configure interrupt, Fade interrupt enable  or Fade interrupt disable*/
        .timer_sel = LEDC_TIMER_0,           /*Select the timer source of channel (0 - LEDC_TIMER_MAX-1)*/
        .duty = 0,                           /*EDC channel duty, the range of duty setting is [0, (2**duty_resolution)] */
        .hpoint = 0,                         /*LEDC channel hpoint value, the range is [0, (2**duty_resolution)-1]*/
    };
    err = ledc_timer_config(&timer_config); /*LEDC timer configuration Configure LEDC timer with the given source timer/frequency(Hz)/duty_resolution*/
    if (err != ESP_OK)
        return err;
    err = ledc_channel_config(&channel_config); /*LEDC channel configuration Configure LEDC channel with the given channel/output gpio_num/interrupt/source timer/frequency(Hz)/LEDC dut*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the **blight_init** function, the member variables of the **gpio_config_t** structure are first parameterised. Subsequently, the **gpio_config** function is invoked to complete the initialisation of the GPIO using these configuration parameters. (Here, all pins required for subsequent operations are initialised). Subsequently, the **ledc_timer_config_t** structure is configured, setting parameters for the LEDC timer (frequency, resolution, etc.). Following this, the **ledc_channel_config_t** structure is configured to bind the GPIO port in use with the corresponding **LEDC timer** and **LEDC channel**. Finally, the **`ledc_timer_config`** and **`ledc_channel_config`** functions are invoked to complete the initialisation of the LEDC controller and its channels. (This registers the PWM interface for backlight control.)

/* Function to set backlight brightness: **`set_lcd_brightness`** */

```
/*brightness - (0 - 100) */
esp_err_t set_lcd_blight(uint32_t brightness)
{
    esp_err_t err = ESP_OK;
    if (brightness != 0)
    {
        err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, ((brightness * 18) + 200)); /*LEDC set duty*/
        if (err != ESP_OK)
            return err;
        err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0); /*LEDC update channel parameters*/
        if (err != ESP_OK)
            return err;
    }
    else
    {
        err = ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, 0);
        if (err != ESP_OK)
            return err;
        err = ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0);
        if (err != ESP_OK)
            return err;
    }
    return err;
}
```

This function takes one parameter: the backlight brightness percentage (0-100). It executes the **`ledc_set_duty`** and **`ledc_update_duty`** functions based on the input brightness percentage. These functions set the duty cycle for the current PWM channel and update the current settings for execution.
/* Display port driver function: **`display_port_init`** */

```
static esp_err_t display_port_init(void)
{
    esp_err_t err = ESP_OK;
    lcd_color_rgb_pixel_format_t dpi_pixel_format; /*LCD color pixel format in RGB color space*/
    /*MIPI DSI bus configuration structure*/
    esp_lcd_dsi_bus_config_t bus_config = {
        .bus_id = 0,                                /*Select which DSI controller, index from 0*/
        .num_data_lanes = 2,                        /*Number of data lanes, if set to 0, the driver will fallback to use maximum number of lanes*/
        .phy_clk_src = MIPI_DSI_PHY_CLK_SRC_DEFAULT, /*MIPI DSI PHY clock source*/
        .lane_bit_rate_mbps = 900,                  /*Lane bit rate in Mbps*/
    };
    err = esp_lcd_new_dsi_bus(&bus_config, &mipi_dsi_bus); /*Create MIPI DSI bus handle*/
    if (err != ESP_OK)
        return err;
    /*Panel IO configuration structure, for MIPI DSI command interface*/
    esp_lcd_dbi_io_config_t dbi_config = {
        .virtual_channel = 0, /*Virtual channel ID, index from 0*/
        .lcd_cmd_bits = 8,    /*Bit-width of LCD command*/
        .lcd_param_bits = 8,  /*Bit-width of LCD parameter*/
    };
    err = esp_lcd_new_panel_io_dbi(mipi_dsi_bus, &dbi_config, &mipi_dbi_io); /*Create LCD panel IO, for MIPI DSI DBI interface*/
    if (err != ESP_OK)
        return err;
    /*Select according to the number of data display digits*/
    if (BITS_PER_PIXEL == 24)
        dpi_pixel_format = LCD_COLOR_PIXEL_FORMAT_RGB888;
    else if (BITS_PER_PIXEL == 18)
        dpi_pixel_format = LCD_COLOR_PIXEL_FORMAT_RGB666;
    else if (BITS_PER_PIXEL == 16)
        dpi_pixel_format = LCD_COLOR_PIXEL_FORMAT_RGB565;
```

First, configure the esp_lcd_dsi_bus_config_t structure to set up the MIPI DSI interface. As we are using a 2-lane display, the num_data_lanes parameter is set to 2. Use the esp_lcd_new_dsi_bus function to create a new DSI bus handle. Next, configure the esp_lcd_dbi_io_config_t structure and create an IO interface handle for the MIPI DSI DBI interface using the esp_lcd_new_panel_io_dbi function. Finally, select the screen colour rendering type via different bit depth settings (we default to RGB565).

```c
/*MIPI DSI DPI panel configuration structur*/
const esp_lcd_dpi_panel_config_t dpi_config = {
    .dpi_clk_src = MIPI_DSI_DPI_CLK_SRC_DEFAULT, /*MIPI DSI DPI clock source*/
    .dpi_clock_freq_mhz = 51,                    /* DPI clock frequency in MHz*/
    .virtual_channel = 0,                        /*Virtual channel ID, index from 0*/
    .pixel_format = dpi_pixel_format,            /*Pixel format that used by the MIPI LCD device*/
    .num_fbs = 1,                                /*Number of screen-sized frame buffers that allocated by the drivered */
    .video_timing = {
        .h_size = H_size,          /*Horizontal resolution*/
        .v_size = V_size,          /*Vertical resolution*/
        .hsync_back_porch = 160,   /*Horizontal back porch, number of pixel clock between hsync and start of line active data*/
        .hsync_pulse_width = 70,   /*Horizontal sync width, in pixel cloc*/
        .hsync_front_porch = 160,  /*Horizontal front porch, number of pixel clock between the end of active data and the next hsync*/
        .vsync_back_porch = 23,    /*Vertical back porch, number of invalid lines between vsync and start of frame*/
        .vsync_pulse_width = 10,   /*Vertical sync width, in number of lines*/
        .vsync_front_porch = 12,   /*Vertical front porch, number of invalid lines between the end of frame and the next vsync*/
    },
    .flags.use_dma2d = true, /*Use DMA2D to copy user buffer to the frame buffer when necessary*/
};
/*LCD panel vendor configuration.*/
ek79007_vendor_config_t vendor_config = {
    .mipi_config = {
        .dsi_bus = mipi_dsi_bus,      /*MIPI-DSI bus configuration*/
        .dpi_config = &dpi_config, /*MIPI-DPI panel configuration */
    },
};
/*Configuration structure for panel device*/
const esp_lcd_panel_dev_config_t panel_config = {
    .reset_gpio_num = LCD_GPIO_RESET,              /* GPIO used to reset the LCD panel, set to -1 if it's not used*/
    .rgb_ele_order = LCD_RGB_ELEMENT_ORDER_RGB, /*Set RGB element order, RGB or BGR*/
    .bits_per_pixel = BITS_PER_PIXEL,              /*Color depth, in bpp*/
    .vendor_config = &vendor_config,               /*vendor specific configuration, optional, left as NULL if not used*/
};
err = esp_lcd_new_panel_ek79007(mipi_dbi_io, &panel_config, &panel_handle); /*Create LCD panel for model EK79007*/
if (err != ESP_OK)
    return err;
err = esp_lcd_panel_reset(panel_handle); /*Reset LCD panel*/
if (err != ESP_OK)
    return err;
err = esp_lcd_panel_init(panel_handle); /*Initialize LCD panel*/
if (err != ESP_OK)
    return err;
return err;
}
```

We configure the **esp_lcd_dpi_panel_config_t** structure to set parameters specific to our display. We then configure the **ek79007_vendor_config_t structure**, incorporating our registered display configuration and MIPI DSI bus interface settings. Subsequently, we configure the **esp_lcd_panel_dev_config_t** structure, which sets our colour bit depth, display mode, and display reset pin.

The **esp_lcd_new_panel_ek79007** function creates a new control interface for our display driver chip. Should the screen be replaced, this function must be reconfigured with the corresponding display driver chip. It registers a handle for this specific display driver chip. Subsequent calls to **esp_lcd_panel_reset** reset this handle, and finally, **esp_lcd_panel_init initialises** the handle, configuring the display interface initialisation. It is worth noting that the configuration of the `**video_timing**` structure involves specific parameters for the display. These must be set according to the display's data manual, with resolution and other parameters corresponding precisely to the specifications outlined therein.

/* lvgl port driver function lvgl_init */

```c
static esp_err_t lvgl_init()
{
    esp_err_t err = ESP_OK;
    const lvgl_port_cfg_t lvgl_cfg = {
        .task_priority = configMAX_PRIORITIES - 4, /* LVGL task priority */
        .task_stack = 8192,                        /* LVGL task stack size */
        .task_affinity = -1,                       /* LVGL task pinned to core (-1 is no affinity) */
        .task_max_sleep_ms = 10,                   /* Maximum sleep in LVGL task */
        .timer_period_ms = 5,                      /* LVGL timer tick period in ms */
    };
    err = lvgl_port_init(&lvgl_cfg); /*Initialize LVGL portation*/
    if (err != ESP_OK)
    {
        DISPLAY_ERROR("LVGL port initialization failed");
    }
```

First, configure the **lvgl_port_cfg_t** structure, which sets parameters such as stack control and priority for the lvgl thread. Then, use the **lvgl_port_init** function to initialise and create the **lvgl** execution thread.

```c
    const lvgl_port_display_cfg_t disp_cfg = {
        .io_handle = mipi_dbi_io,                                            /*LCD panel IO handle*/
        .panel_handle = panel_handle,                                        /*LCD panel handle*/
        .control_handle = panel_handle,                                      /*LCD panel control handle*/
        .buffer_size = (H_size * V_size * ((BITS_PER_PIXEL + 7) / 8)), /*Size of the buffer for the screen in pixels*/
        .double_buffer = true,                                               /*True, if should be allocated two buffers*/
        .hres = H_size,                                                      /*LCD display horizontal resolution*/
        .vres = V_size,                                                      /*LCD display vertical resolution*/
        .monochrome = false,                                                 /*True, if display is monochrome and using 1bit for 1px*/
#if LVGL_VERSION_MAJOR >= 9
        .color_format = LV_COLOR_FORMAT_RGB565,
#endif
        .rotation = {
            .swap_xy = false,  /*LCD Screen swapped X and Y (in esp_lcd driver)*/
            .mirror_x = false, /*LCD Screen mirrored X (in esp_lcd driver)*/
            .mirror_y = false, /*LCD Screen mirrored Y (in esp_lcd driver)*/
        },
        .flags = {
            .buff_dma = false,    /*Allocated LVGL buffer will be DMA capable*/
            .buff_spiram = true, /*Allocated LVGL buffer will be in PSRAM*/
            .sw_rotate = true,    /* Use software rotation (slower) or PPA if available*/
#if LVGL_VERSION_MAJOR >= 9
            .swap_bytes = true, /*Swap bytes in RGB656 (16-bit) before send to LCD driver*/
#endif
#if CONFIG_DISPLAY_LVGL_FULL_REFRESH
            .full_refresh = true,
#else
            .full_refresh = false, /*Always make the whole screen redrawn*/
#endif
#if CONFIG_DISPLAY_LVGL_DIRECT_MODE
            .direct_mode = true,
#else
            .direct_mode = false, /*Use screen-sized buffers and draw to absolute coordinates*/
#endif
        },
    };
    const lvgl_port_display_dsi_cfg_t lvgl_dpi_cfg = {
        .flags = {
#if CONFIG_DISPLAY_LVGL_AVOID_TEAR
            .avoid_tearing = true,
#else
            /*Use internal MIPI-DSI buffers as a LVGL draw buffers to avoid tearing effect, enabling this option requires over two LCD buffers and may reduce the frame rate*/
            .avoid_tearing = false,
#endif
        },
    };
```

First, configure the **`lvgl_port_display_cfg_t`** structure by setting the handles obtained from the display port driver function within the **`lvgl`** structure. The **`buffer_size`** parameter defines the display's refresh buffer, which can be configured based on refresh rate requirements—such as full-screen or half-screen buffering. The **`double_buffer`** parameter determines whether double buffering is enabled, which significantly improves display frame rates.

Key point: The **buff_dma** field within the flags structure determines whether the buffer is allocated within the DMA region. Configuring this to true offers high display efficiency and rapid refresh rates. However, chip DMA space resources are typically scarce, making this option generally impractical. Moreover, space constraints often prevent full-screen configuration, rendering it suitable only for low-resolution displays.

buff_spiram configures the buffer within the PSRAM region; this option is mutually exclusive with **buff_dma**. Setting this to true enables **full-screen buffering**, providing sufficient space and meeting speed requirements, making it suitable for **high-resolution** displays.

After configuration, initialise and register the display's LVGL configuration using the **lvgl_port_add_disp_dsi** function, which returns a control handle.

/* Screen initialisation function display_init */

```
esp_err_t display_init()
{
    esp_err_t err = ESP_OK;
    err = blight_init(); /*Backlight initialization function*/
    if (err != ESP_OK)
        return err;
    err = display_port_init(); /*Display screen interface initialization function*/
    if (err != ESP_OK)
        return err;
    err = lvgl_init(); /*Screen interface registration LVGL function*/
    if (err != ESP_OK)
    {
        DISPLAY_ERROR("Display init fail");
        return err;
    }
    gpio_set_level(LCD_GPIO_UPDN, 0); /*Set the display screen vertical mirroring*/
    gpio_set_level(LCD_GPIO_SHLR, 1); /*Set the display screen horizontal mirroring*/
    set_lcd_blight(0);                /*Set the backlight to turn off*/
    return err;
}
```

This function calls the **backlight initialisation**, display **driver initialisation**, and **lvgl initialisation**, combining these initialisation functions into a single initialisation process. It then calls the **gpio_set_level** function to set the vertical mirror pin to low level and the horizontal mirror pin to high level. These two mirroring effects are configured according to the display requirements of the screen in use. Finally, it sets the screen backlight brightness to 0% (i.e., turns off the backlight).

1.3.2 Kconfig file

The primary function of this file is to incorporate the requisite configurations into the **sdkconfig file**, enabling certain parameter adjustments to be made via a graphical interface. These parameters comprise: screen colour bit depth: **16-bit for RGB565 and 24-bit for RGB888**; **screen** reset pin, **backlight** control pin, **vertical** mirror pin, **horizontal** mirror pin, and **backlight PWM** frequency. The numerical pin designations correspond to the **GPIO_NUM** sequence.

```
menu "BSP DISPLAY Setup"
    config BSP_DISPLAY_ENABLED
        bool "Enable display functions"

    if BSP_DISPLAY_ENABLED
        config H_SIZE
            int "Horizontal Resolution"
            default 1024

        config V_SIZE
            int "Vertical Resolution"
            default 600

        config BITS_PER_PIXEL
            int "Color depth"
            default 16

        config LCD_GPIO_RESET
            int "GPIO For LCD RESET"
            default 5

        config LCD_GPIO_BLIGHT
            int "GPIO For LCD BACKLIGHT"
            default 20

        config LCD_GPIO_UPDN
            int "GPIO For LCD UPDN"
            default 32

        config LCD_GPIO_SHLR
            int "GPIO For LCD SHLR"
            default 33

        config BLIGHT_PWM_Hz
            int "the PWM output HZ"
            default 30000

        config DISPLAY_LVGL_FULL_REFRESH
            bool "Enable fullscreen refresh"

        config DISPLAY_LVGL_DIRECT_MODE
            bool "Use screen-sized buffers"

        config DISPLAY_LVGL_AVOID_TEAR
            bool "Enable to avoid fragmentation"
    endif

endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_display** driver. To successfully invoke the contents of the **bsp_display** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_display** folder. The configuration details are as follows:

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

idf_component_register(SRCS ${component_sources}
                        INCLUDE_DIRS "include"
                        REQUIRES driver esp_lcd_ek79007 lvgl esp_lvgl_port)
```

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver libraries (the driver library for the display driver chip **ek79007**, and the **lvgl** driver library). Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to **utilise the bsp_display** driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the header file main.h within the include folder. Add the main folder to the CMakeLists.txt file of the build system.

The main.h file primarily references required header files: functions utilising the bsp_display driver necessitate inclusion of the bsp_display header file.

Below is an analysis of the main.c programme: system initialisation and execution of display-specific functions.

```
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
```

This code resides within the **init** function, which serves to store initialisation functions requiring invocation and assess their return outcomes. Should the return status deviate from **ESP_OK**, the code will output an error message and cease further execution.

/* Screen initialisation and display function **display_test** */

```
/*-----------------------------------Functional function-----------------------------------*/
#ifdef CONFIG_BSP_DISPLAY_ENABLED
void display_test()
{
    if (lvgl_port_lock(0))
    {
        lv_obj_t *label = lv_label_create(lv_scr_act()); /*Create a label object*/
        lv_label_set_text(label, "Hello P4!");           /*Set a new text for a label. Memory will be allocated to store the text by the label.*/

        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(label, &label_style, LV_PART_MAIN);                  /*Add a style to an object*/
        lv_obj_center(label);                                                 /*Align an object to the center on its parent*/
        lv_obj_set_style_text_color(label, LV_COLOR_BLACK, LV_PART_MAIN);     /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(label, &lv_font_montserrat_42, LV_PART_MAIN); /*Set the style LVGL text font*/

        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*/
        lvgl_port_unlock();
    }
}
#endif
```

This function primarily configures the initial screen display content: it sets the background colour and text display via the lvgl control.

The **lv_label_set_text** function sets the text displayed on the control.

The **lv_obj_set_style_text_color** function sets the text display colour.

The **lv_obj_set_style_text_font** function sets the text font size.

The **lv_obj_set_style_bg_color** function sets the background colour.

The **lv_obj_set_style_bg_opa** function sets the background transparency.

It is worth noting that when calling Lvgl functions outside of Lvgl threads, a mutual exclusion lock must be acquired. The **lvgl_port_lock** function acquires the mutual exclusion lock, while the **lvgl_port_unlock** function releases it.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    Init();
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100);   /*Set the screen backlight to maximum brightness*/
    if (lvgl_port_lock(0)) /*Take LVGL mutex*/
    {
        display_test();     /*Set the screen's LVGL default display page*/
        lvgl_port_unlock(); /*Give LVGL mutex*/
    }
    while (1)
    {
        if (lvgl_port_lock(0))
        {
            lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_RED, 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*/
            lvgl_port_unlock();
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
        if (lvgl_port_lock(0))
        {
            lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_YELLOW, LV_PART_MAIN);
            lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN);
            lvgl_port_unlock();
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
        if (lvgl_port_lock(0))
        {
            lv_obj_set_style_bg_color(lv_scr_act(), LV_COLOR_BLUE, LV_PART_MAIN);
            lv_obj_set_style_bg_opa(lv_scr_act(), LV_OPA_COVER, LV_PART_MAIN);
            lvgl_port_unlock();
        }
        vTaskDelay(pdMS_TO_TICKS(2000)); /*Delay 2000ms*/
    }
#endif
    MAIN_INFO("----------Start the test----------");
}
```

Within the **app_main** function, the **backlight brightness** is first set to 100%, followed by **initialising** the screen display content. A loop is then created, within which the lvgl function is executed every two seconds to set the background colour. The background colour is cycled through (in the sequence red-yellow-blue). These colours may be modified as required, with colour definitions specified in the **bsp_display.h** header file.

1.3.5 CMkaLists.txt file

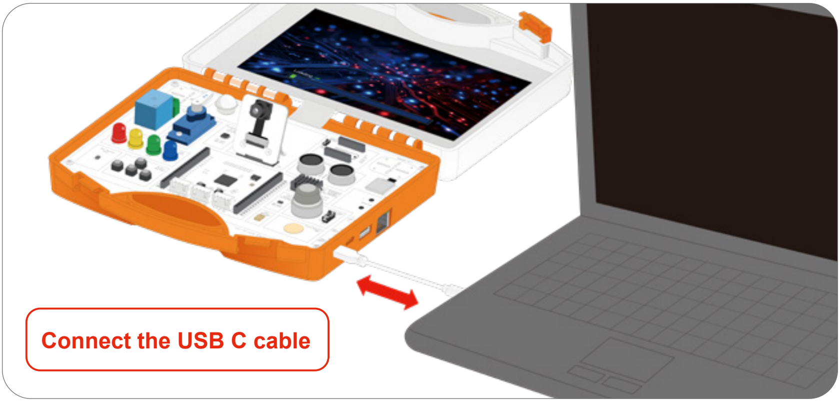
To successfully call the contents of the **bsp_display** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_display)
```

First, the directories for source files and header files are defined, along with the required driver library—specifically, the driver library for linking **bsp_display**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver capabilities.

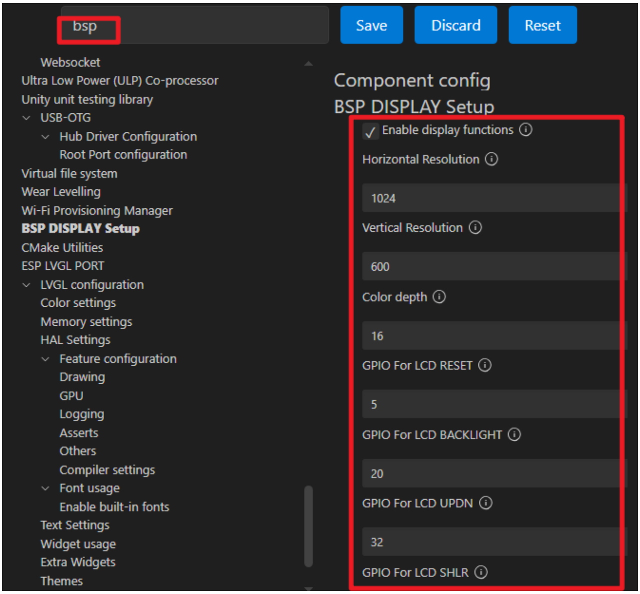
1.4 Programming procedure

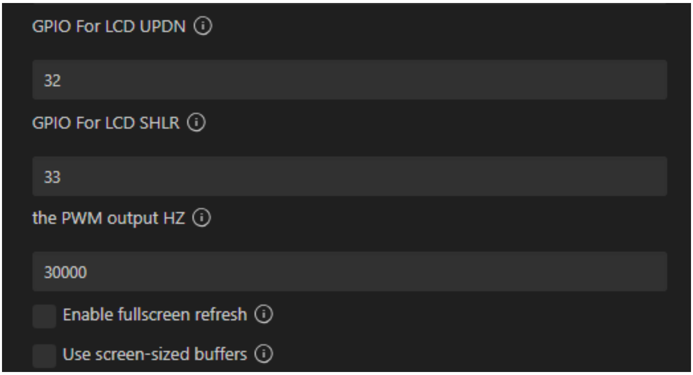
Connect the P4 device to the computer via USB

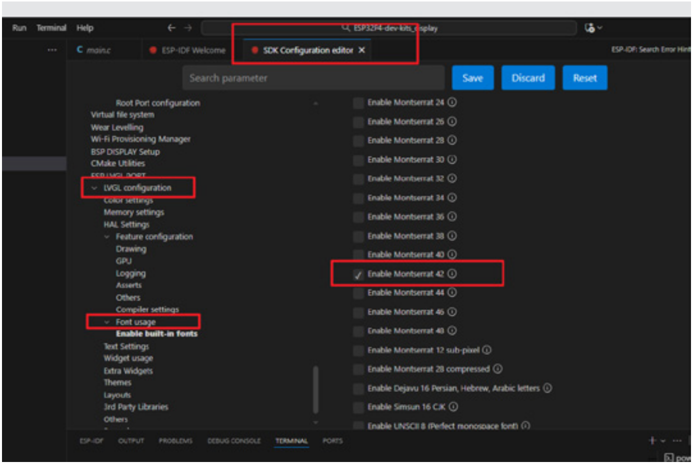


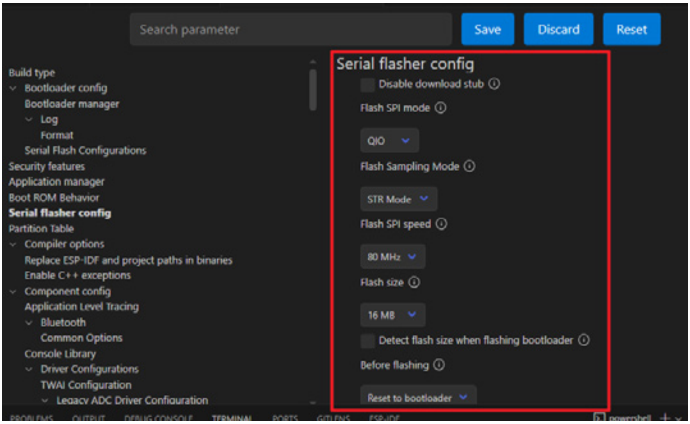
1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the DSI pins.

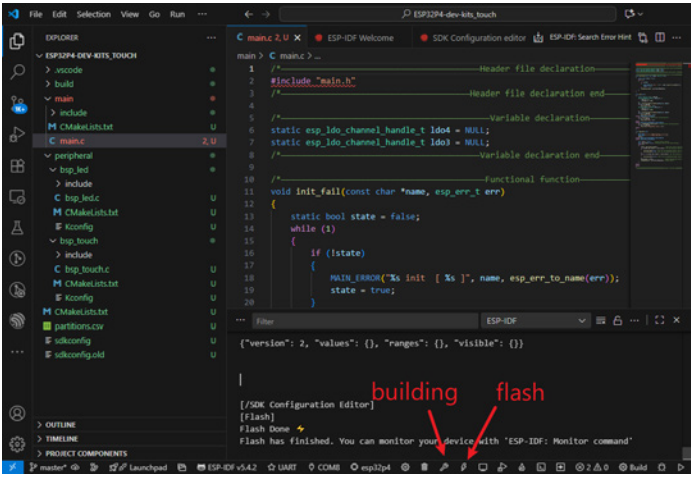



GPIO For LCD UPDN
32
GPIO For LCD SHLR
33
the PWM output HZ
30000
Enable fullscreen refresh
Use screen-sized buffers


Run
Terminal
Help
ESP32P4-dev-kits_display
main.c
ESP-IDF Welcome
SDK Configuration editor
Search parameter
Save
Discard
Reset
Root Port configuration
Virtual file system
Wear Levelling
Wi-Fi Provisioning Manager
BSP DISPLAY Setup
CMake Utilities
LVGL configuration
Memory settings
HAL Settings
Feature configuration
Drawing
GPU
Logging
Asserts
Others
Compiler settings
Font usage
Enable built-in fonts
Text Settings
Widget usage
Extra Widgets
Themes
Layouts
3rd Party Libraries
Others
Enable Montserrat 24
Enable Montserrat 26
Enable Montserrat 28
Enable Montserrat 30
Enable Montserrat 32
Enable Montserrat 34
Enable Montserrat 36
Enable Montserrat 38
Enable Montserrat 40
Enable Montserrat 42
Enable Montserrat 44
Enable Montserrat 46
Enable Montserrat 48
Enable Montserrat 12 sub-pixel
Enable Montserrat 28 compressed
Enable Dejavu 16 Persian, Hebrew, Arabic letters
Enable Simsun 16 CJK
ESP-IDF
OUTPUT
PROBLEMS
DEBUG CONSOLE
TERMINAL
PORTS


Search parameter
Save
Discard
Reset
Build type
Bootloader config
Bootloader manager
Log
Format
Serial Flash Configurations
Security features
Application manager
Boot ROM Behavior
Serial flasher config
Partition Table
Compiler options
Replace ESP-IDF and project paths in binaries
Enable C++ exceptions
Component config
Application Level Tracing
Bluetooth
Common Options
Console Library
Driver Configurations
TWAI Configuration
Legacy ADC Driver Configuration
Serial flasher config
Disable download stub
Flash SPI mode
QIO
Flash Sampling Mode
STR Mode
Flash SPI speed
80 MHz
Flash size
16 MB
Detect flash size when flashing bootloader
Before flashing
Reset to bootloader

1.4.3 Click Compile. Once compilation is successful, click Download.

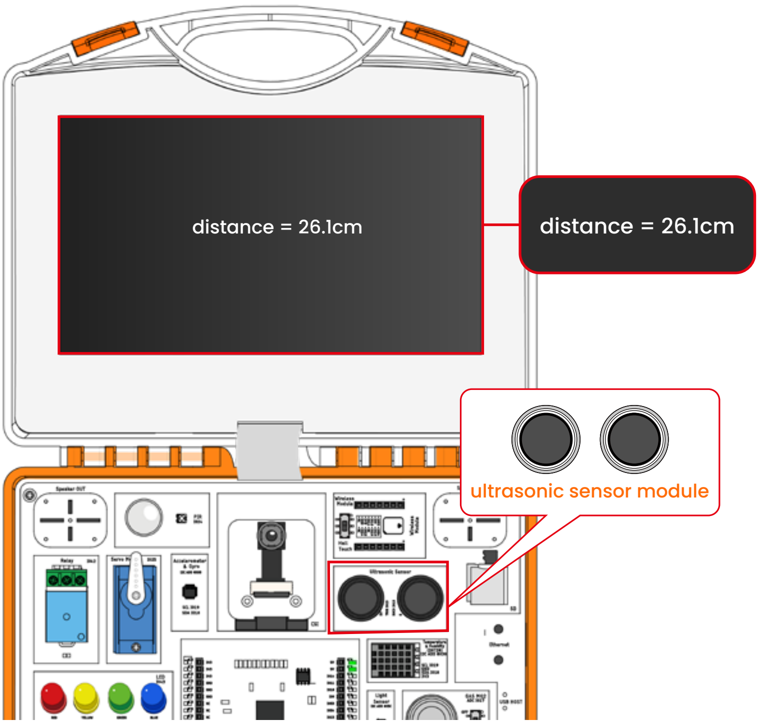


Lesson 10 – Ultrasonic Distance Display

Introduction

This chapter's tutorial introduces the interface application between the ESP32-P4 and an ultrasonic sensor, utilising a distance measurement routine to aid understanding of its fundamental functionality. As a common sensor application case, ultrasonic ranging provides readers with an intuitive grasp of how the ESP32-P4 interacts with peripherals, laying the groundwork for more complex intelligent detection and control projects.

Project Demonstration Effect

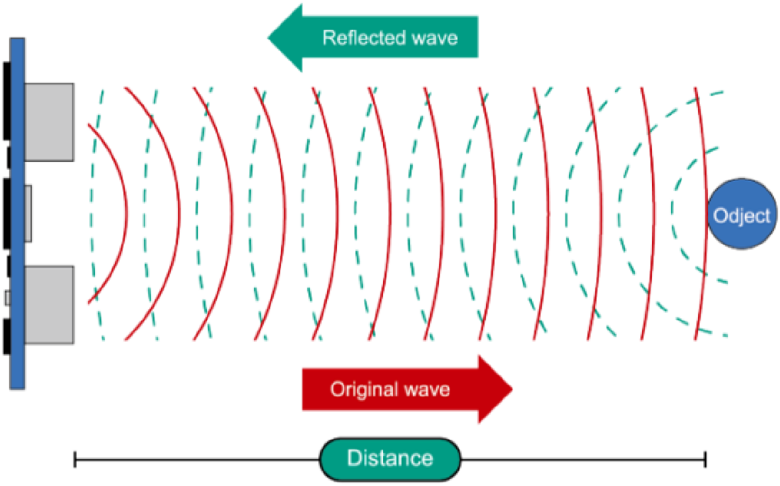


This chapter is divided into the following subsections

1.1 Introduction to Ultrasonic Sensors

1.1.1 The Working Principle of Ultrasonic Sensors

Ultrasonic sensors are devices that utilise the principle of sound wave reflection for non-contact distance measurement, commonly employed in scenarios such as obstacle detection, liquid level measurement, and robotic obstacle avoidance. Taking the widely used HC-SR04 module as an example, it accomplishes measurement through the sequence: emitting an ultrasonic wave → receiving the echo → calculating the distance.



Emission: When the ESP32-P4 applies a high level exceeding 10 μs to the Trigger pin, the sensor emits a 40 kHz ultrasonic pulse.

Propagation and Reflection: The ultrasonic wave reflects back upon encountering an obstacle.

Reception: The sensor's Echo pin outputs a high-level pulse whose duration is proportional to the round-trip time of the ultrasonic wave.

Distance Calculation: The target distance is calculated using the formula:

Distance (cm) = Time (μs) × Speed of Sound (340 m/s) ÷ 2 × 10,000 Distance (cm) = \frac{Time (μs) × Speed of Sound (340 m/s)}{2 × 10,000} Distance (cm) = 2 × 10,000 × Time (μs) × Speed of Sound (340 m/s)

For example, if the Echo high-level signal persists for 2 ms, the target distance is approximately 34 cm.

1.1.2 Module Pin Description

Taking the HC-SR04 as an example, it typically features four pins:

VCC: Power supply voltage 5V (some models support 3.3V)

GND: Ground

Trigger (Trig): Trigger input pin, requires a 10 μs high level

Echo: Echo output terminal, where the high-level width is proportional to the distance

⚠ **Caution**: The ESP32-P4's GPIO operates at 3.3V logic levels, whereas some ultrasonic modules output Echo at 5V. Therefore, voltage reduction protection is required via a voltage divider resistor or a level-shifting module.

1.1.3 Factors Affecting Distance Measurement

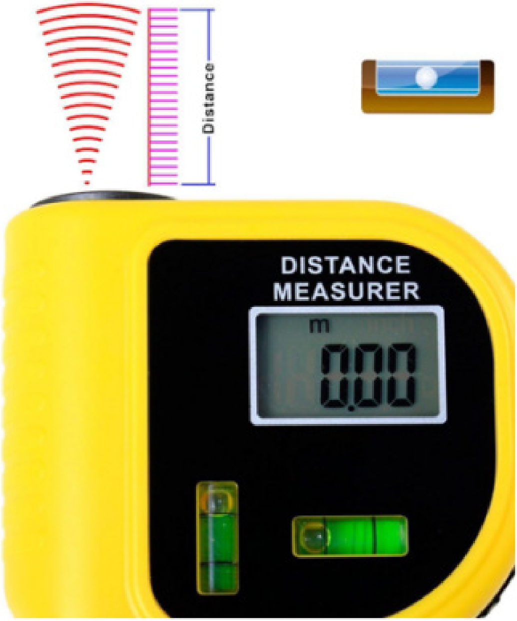
Ambient temperature: The speed of sound varies with temperature (approximately 340 m/s at 20°C, increasing by roughly 0.6 m/s per 1°C rise).

Measurement angle: The sensor's emission cone angle is typically around 15°, requiring the target to remain within this range for accurate distance measurement.

Object Material: Soft or sound-absorbing materials (such as fabric or sponge) may cause echo attenuation or measurement failure.

Measuring Range: The typical effective measurement range for the HC-SR04 is 2 cm to 400 cm.

Ultrasound finds extensive application in various scenarios of daily life.



Ultrasonic Rangefinder

1.2 Hardware design

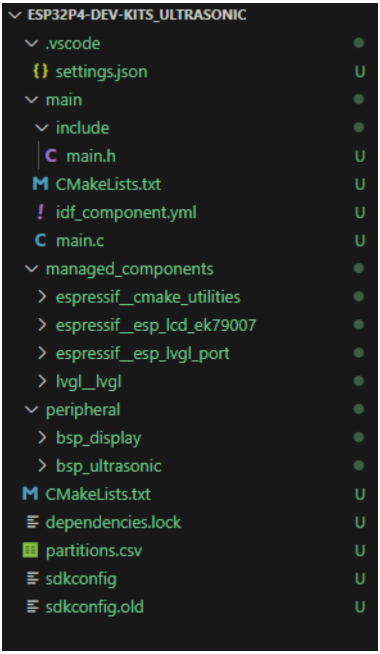
The following table shows the connection method for the ESP32-P4 and HC-SR04:

Ultrasonic Sensor	ESP32-P4	Specifications
VCC	5V	Power Supply
GND	GND	Common Ground
Trig	GPIO10	Trigger Signal Output
Echo	GPIO11	Echo Signal Input (Requires Voltage Dividing)

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_ultrasinic** example project, a new folder named bsp_ultrasinic has been created under the **ESP32P4-dev-kits_ultrasinic\peripheral** directory. Within the **bsp_ultrasinic** path, a new include folder, **CMakeLists.txt** file, and **Kconfig file** have been established. The **bsp_ultrasinic** folder houses the **bsp_ultrasinic.c** driver file, the include folder contains the **bsp_ultrasinic.h** header file, and the **CMakeLists.txt** file integrates the driver into the build system, enabling the project to utilise ultrasonic driver functionality. The **Kconfig** file loads the entire driver and GPIO pin definitions into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 Ultrasonic Driver Code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The ultrasonic driver source code comprises two files: **bsp_ultrasonic.c** and **bsp_ultrasonic.h**.

Below we shall first analyse the **bsp_ultrasonic.h** programme: it contains relevant definitions for the ultrasonic pins and function declarations.

/* Header file references */

```
/*---------------------------------------Header file declaration---------------------------------------*/
#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 "esp32p4/rom/ets_sys.h" //References for system timers Function-related API Functions
#include "driver/gpio.h"         //References for GPIO Function-related API Functions
#include "driver/mcpwm_cap.h"    ///References for MCPWM capture timer Function-related API Functions
/*---------------------------------------Header file declaration end---------------------------------------*/
```

/* Function declarations and macro definition declarations */

```
#ifdef CONFIG_BSP_ULTRASONIC_ENABLED

#define ULTRASONIC_GPIO_ECHO CONFIG_ULTRASONIC_GPIO_ECHO // ULTRASONIC ECHO GPIO
#define ULTRASONIC_GPIO_TRIG CONFIG_ULTRASONIC_GPIO_TRIG // ULTRASONIC TRIG GPIO

typedef struct
{
    uint32_t ultrasonic_time;       // Total Ultrasonic Transmission Time
    uint32_t ultrasonic_start_time; // Ultrasonic transmission start time
    uint32_t ultrasonic_stop_time;  // Ultrasonic transmission end time
    float ultrasonic_distance;      // Distance measured by ultrasonic
} ultrasonic_struct;

esp_err_t ultrasonic_init();                                  // Ultrasonic Initialization function
esp_err_t ultrasonic_callback_register(TaskHandle_t handle); // Ultrasonic task callback registration function
esp_err_t send_ultrasonic_start();                           // Send the ultrasonic start signal function
float get_ultrasonic_distance(uint32_t time);                // Convert the collected time data into distance function

#endif
```

Next, we shall analyse the **bsp_ultrasonic.c** programme: initialising and configuring the ultrasonic pins, calling the settings, and executing the callback function.

/* Ultrasonic initialisation function ultrasonic_init */

```
esp_err_t ultrasonic_init()
{
    esp_err_t err = ESP_OK;
    mcpwm_cap_timer_handle_t cap_timer = NULL; /*Type of MCPWM capture timer handle*/
    mcpwm_capture_timer_config_t cap_conf = {
        .clk_src = MCPWM_CAPTURE_CLK_SRC_DEFAULT, /*MCPWM capture timer clock source*/
        .group_id = 0,                            /*Specify from which group to allocate the capture timer*/
    }; /*MCPWM capture timer configuration structure*/
    err = mcpwm_new_capture_timer(&cap_conf, &cap_timer); /*Create MCPWM capture timer*/
    if (err != ESP_OK)
        return err;
    mcpwm_capture_channel_config_t cap_ch_conf = {
        .gpio_num = ULTRASONIC_GPIO_ECHO, /*GPIO used capturing input signal*/
        .prescale = 1,                    /*Prescale of input signal, effective frequency = cap_input_clk/prescale*/
        .flags.neg_edge = true,           /*Whether to capture on negative edge*/
        .flags.pos_edge = true,           /*Whether to capture on positive edge*/
        .flags.pull_up = true,            /*Whether to pull up internally*/
    }; /*MCPWM capture channel configuration structure*/
    err = mcpwm_new_capture_channel(cap_timer, &cap_ch_conf, &cap_chan); /*Create MCPWM capture channel*/
    if (err != ESP_OK)
        return err;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << ULTRASONIC_GPIO_TRIG, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_OUTPUT,                     /* GPIO mode: set input/output mode                     */
        .pull_up_en = false,                          /* GPIO pull-up                                         */
        .pull_down_en = false,                        /* GPIO pull-down                                       */
        .intr_type = GPIO_INTR_DISABLE,               /* GPIO interrupt type                                  */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    err = mcpwm_capture_timer_enable(cap_timer); /*Enable MCPWM capture timer*/
    if (err != ESP_OK)
        return err;
    err = mcpwm_capture_timer_start(cap_timer); /*Start MCPWM capture timer*/
    if (err != ESP_OK)
        return err;
    ultrasonic_data.ultrasonic_start_time = 0;
    ultrasonic_data.ultrasonic_stop_time = 0;
    ultrasonic_data.ultrasonic_time = 0;
    return err;
}
```

Within the **ultrasinic_init** function, the member variables of the **mcpwm_capture_timer_config_t structure** are first configured with parameters. Subsequently, the **mcpwm_new_capture_timer** function is invoked to create a new mcpwm timer. The **mcpwm_capture_channel_config_t structure** is then configured. The g**pio_num parameter** corresponds to the GPIO pin for the timer input signal, the prescale parameter corresponds to the prescaler coefficient, **flags.neg_edge** indicates whether to capture the falling edge of the signal, **flags.pos_edge** indicates whether to capture the rising edge, and **flags.pull_up** indicates whether to enable internal pull-up. After configuration, the `**mcpwm_new_capture_channel**` function registers the new timer capture channel. Subsequently, the `**gpio_config_t**` structure is configured, and the `**gpio_config**` function sets up the signal output pin. Finally, the capture timer is enabled using `**mcpwm_capture_timer_enable**`, and the capture timer is started with the `**mcpwm_capture_timer_start**` function.

/* Ultrasonic callback registration function **ultrasonic_callback_register** */

```
esp_err_t ultrasonic_callback_register(TaskHandle_t handle)
{
    esp_err_t err = ESP_OK;
    /*Group of supported MCPWM capture event callbacks*/
    mcpwm_capture_event_callbacks_t cbs = {
        .on_cap = hc_sr04_echo_callback, /*Callback function that would be invoked when capture event occurred*/
    };
    err = mcpwm_capture_channel_register_event_callbacks(cap_chan, &cbs, handle); /*Set event callbacks for MCPWM capture channel*/
    if (err != ESP_OK)
        return err;
    err = mcpwm_capture_channel_enable(cap_chan); /*Enable MCPWM capture channel*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

This function takes one argument, which is the method for conveying results via task notifications after data acquisition within the callback function. The parameter passed is the **FreeRTOS** thread handle. This function configures the `**mcpwm_capture_event_-callbacks_t**` structure, registers the callback function via `**mcpwm_capture_chan-nel_register_event_callbacks**`, and finally enables the capture timer channel by calling `**mcpwm_capture_channel_enable**`.

/* Ultrasonic echo callback execution function `**hc_sr04_echo_callback**` */

```
static bool hc_sr04_echo_callback(mcpwm_cap_channel_handle_t cap_chan, const mcpwm_capture_event_data_t *edata, void *user_data)
{
    TaskHandle_t task_to_notify = (TaskHandle_t)user_data;
    BaseType_t high_task_wakeup = pdFALSE;
    if (edata->cap_edge == MCPWM_CAP_EDGE_POS)
    {
        ultrasonic_data.ultrasonic_start_time = edata->cap_value;
        ultrasonic_data.ultrasonic_stop_time = ultrasonic_data.ultrasonic_start_time;
    }
    else
    {
        ultrasonic_data.ultrasonic_stop_time = edata->cap_value;
        ultrasonic_data.ultrasonic_time = ultrasonic_data.ultrasonic_stop_time - ultrasonic_data.ultrasonic_start_time;
        xTaskNotifyFromISR(task_to_notify, ultrasonic_data.ultrasonic_time, eSetValueWithOverwrite, &high_task_wakeup);
    }
    return high_task_wakeup == pdTRUE;
}
```

First, the callback function assigns the captured rising edge signal time as the ultrasonic start time, then assigns the captured falling edge signal time as the ultrasonic end time. Subtracting these yields the total ultrasonic transit time, which is then transmitted via task notification.

/* Function send_ultrasonic_start for sending start signal before ultrasonic transmission */

```
esp_err_t send_ultrasonic_start()
{
    esp_err_t err = ESP_OK;
    err = gpio_set_level(ULTRASONIC_GPIO_TRIG, 0); /*Set the Corresponding Output Level of GPIO*/
    if (err != ESP_OK)
        return err;
    ets_delay_us(20);
    err = gpio_set_level(ULTRASONIC_GPIO_TRIG, 1);
    if (err != ESP_OK)
        return err;
    ets_delay_us(10);
    err = gpio_set_level(ULTRASONIC_GPIO_TRIG, 0);
    if (err != ESP_OK)
        return err;
    return err;
}
```

Call the gpio_set_level function to set the TRIG pin to output a **low-level---high-level---low-level** waveform with a 10μs interval. This is the timing requirement for the ultrasonic sensor's start signal.

/* Ultrasonic time-to-distance conversion function get_ultrasonic_distance */

```
float get_ultrasonic_distance(uint32_t time)
{
    static float last_ultrasonic_distance = 0;
    ultrasonic_data.ultrasonic_distance = time * 0.01715;                 /*Ultrasonic Conversion Formula*/
    if (ultrasonic_data.ultrasonic_distance != last_ultrasonic_distance) /*If the currently converted data is inconsistent with the previous data*/
    {
        last_ultrasonic_distance = ultrasonic_data.ultrasonic_distance;
        return ultrasonic_data.ultrasonic_distance;
    }
    last_ultrasonic_distance = ultrasonic_data.ultrasonic_distance;
    return -1; /*If the currently converted data is consistent with the previous data*/
}
#endif
```

This function converts the input value “time”, measured in microseconds, into distance using the following formula: ultrasonic_data.ultrasonic_distance = time * 0.01715; The conversion formula is time multiplied by 0.01715. The specific formula is as follows:

Distance = High-level Duration × Speed of Sound (340 m/s) ÷ 2. The speed of sound unit can be converted as follows: 340 m/s = 0.0343 cm/μs. Dividing 0.0343 cm/μs by 2 yields 0.01715, hence the formula conversion to time × 0.01715.

It is worth noting that after converting the distance, we perform a check: if the newly calculated distance matches the previous one, we return -1.

1.3.2 Kconfig file

The primary function of this file is to incorporate the required configurations into the sdkconfig file, enabling certain parameter adjustments to be made via a graphical interface. Here, 12 corresponds to GPIO_NUM_12, and 13 corresponds to GPIO_NUM_13.

```
menu "BSP ULTRASONIC Setup"
    config BSP_ULTRASONIC_ENABLED
        bool "Enable ULTRASONIC config"
        default n

    if BSP_ULTRASONIC_ENABLED
        config ULTRASONIC_GPIO_ECHO
            int "GPIO For ULTRASONIC ECHO"
            default 12

        config ULTRASONIC_GPIO_TRIG
            int "GPIO For ULTRASONIC TRIG"
            default 13
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_ultrasonic** driver. To successfully call the contents of the **bsp_ultrasonic** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_ultrasonic** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver libraries (**esp_timer** for the capture timer and **esp_rom** for microsecond-level timing). Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to **utilise the bsp_ultrasonic** driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable **main.c** and the **main.h** header file within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The **main.h** file primarily references required header files: functions utilising the **bsp_display** driver necessitate inclusion of the **bsp_display** header file, while those employing the **bsp_ultrasonic** driver require the **bsp_ultrasonic** header file.

Below is an analysis of the main.c programme: system initialisation alongside execution of display functionality and ultrasonic capabilities.

```
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
#ifdef CONFIG_BSP_ULTRASONIC_ENABLED
    err = ultrasonic_init(); /*ultrasoni Initialization*/
    if (err != ESP_OK)
        init_fail("ultrasonic", err);
#endif
```

This code resides within the init function, which serves to store initialisation functions requiring invocation and assess the outcome of such initialisation. Should the return status not be **ESP_OK**, the code will output an error message and cease further execution.

/* Screen initialisation and display function **ultrasonic_display** */

```
void ultrasonic_display()
{
    if (lvgl_port_lock(0))
    {
        distance_label = 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(distance_label, &label_style, LV_PART_MAIN);                 /*Add a style to an object*/
        lv_obj_set_style_text_color(distance_label, LV_COLOR_WHITE, LV_PART_MAIN);    /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(distance_label, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_center(distance_label);                                                /*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(distance_label, "distance = 0.0 cm");                       /*Set a new text for a label*/
        lvgl_port_unlock();
    }
}
```

This function primarily configures the initial screen display content: setting background colours and text display via lvgl controls.

The lv_label_set_text function sets the text displayed on the control.

The lv_style_set_bg_opa function sets the background colour of the style.

The lv_obj_set_style_text_color function sets the text display colour. The lv_obj_set_-style_text_font function sets the text font size.

The lv_obj_set_style_bg_color function sets the background colour.

lv_obj_set_style_bg_opa function sets background transparency

Note: When calling lvgl functions outside lvgl thread functions, a mutex lock must be acquired. **lvgl_port_lock** function acquires the mutex lock, **lvgl_port_unlock** function releases it.

/* Screen data refresh display function **update_distance_value** */

```
void update_distance_value(float new_distance_cm)
{
    if (distance_label)
    {
        char buffer[32];
        snprintf(buffer, sizeof(buffer), "distance = %.1f cm", new_distance_cm); /*Format the data into a string*/
        lv_label_set_text(distance_label, buffer);                               /*Set a new text for a label*/
    }
}
```

This function employs the snprintf function to format the acquired float-type data into a string, subsequently refreshing the displayed content via the lv_label_set_text function.

It is worth noting that the snprintf function appends a terminating character to the end of the formatted string. Furthermore, lv_label_set_text recognises strings by identifying this terminating character. Consequently, utilising the snprintf function for string formatting constitutes a preferable approach.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_ULTRASONIC_ENABLED
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100);                                          /*Set the screen backlight to maximum brightness*/
    ultrasonic_display();                                         /*Set the screen's LVGL default display page*/
    xTaskCreate(ultrasonic_task, "ultrasonic", 4096, NULL, configMAX_PRIORITIES - 5, &ultrasonic); /*Create an ultrasonic data display refresh thread*/
#endif
#endif
}
```

Within the **app_main** function, the backlight brightness is first set to 100%, followed by initialising the screen display content. Finally, a **FreeRTOS** thread is created to handle data processing and screen refresh operations.

/* Ultrasonic data processing and screen refresh thread: **ultrasonic_task** */

```
void ultrasonic_task(void *param)
{
    static esp_err_t err = ESP_OK;
    uint32_t time = 0;
    float distance = 0;
    err = ultrasonic_callback_register(ultrasonic); /*Register the ultrasonic timing capture callback function (Parameter: Task Handle)*/
    if (err != ESP_OK)
    {
        MAIN_INFO("ultrasonic callback register fail");
    }
 while (1)
 {
     err = send_ultrasonic_start(); /*Send the ultrasonic start signal*/
     if (err != ESP_OK)
     {
         continue;
         MAIN_INFO("send ultrasonic start fail");
     }
     if (xTaskNotifyWait(0x00, ULONG_MAX, &time, pdMS_TO_TICKS(1000)) == pdTRUE) /*Wait for the task notification and obtain the transmitted data value
     {
         float pulse_width_us = time * (1000000.0 / esp_clk_apb_freq()); /*1s = 1000000us*/
         if (pulse_width_us > 35000)                                      /*If the measured time data exceeds 35 seconds, it is out-of-range data*/
         {
             MAIN_INFO("the distance exceeds the limit");
             MAIN_INFO("the pulse_width_us =%f", pulse_width_us);
             if (lvgl_port_lock(0))
             {
                 lv_label_set_text(distance_label, "the distance exceeds the limit"); /*Set a new text for a label*/
                 lvgl_port_unlock();
             }
         }
         else
         {
             distance = get_ultrasonic_distance(pulse_width_us); /*Convert the collected time data into distance*/
             if (distance == -1)                                 /*If the value is -1, the distance data is consistent with the previous one*/
             {
                 MAIN_INFO("not new distance");
             }
             else
             {
                 MAIN_INFO("distance=%.2f cm", distance);
                 if (lvgl_port_lock(0))
                 {
                     update_distance_value(distance); /*Update the distance data displayed on the screen*/
                     lvgl_port_unlock();
                 }
             }
         }
     }
     vTaskDelay(1000 / portTICK_PERIOD_MS);
 }
```

Within the ultrasonic data processing and screen display refresh thread, variables are first initialised, and **ultrasonic_callback_register** is invoked to register the callback function by passing the current thread handle. Subsequently, a while loop is established. Within this loop, the function to send the ultrasonic start signal is invoked. Should this fail, the loop is re-entered via continue. Upon successful execution, the thread awaits the reception task notification. Within this notification, the transmitted time data is received and converted into units of microseconds. The system then assesses whether the elapsed time exceeds the ultrasonic measurement maximum of 35 seconds. If exceeded, the lvgl function is invoked to control the display refresh, showing the string **'the distance exceeds the limit'**. If the limit is not exceeded, the **`get_ultrasonic_distance`** function is called to convert the data into distance values. The function's return value is checked: if it is -1, the data matches the previous reading and the display is not updated. If it is not -1, the **`update_distance_value**(distance)` function is called to refresh the display with the latest data. The final 1-second delay represents the interval at which ultrasonic distance measurement is performed.

1.3.4 CMkaLists.txt file

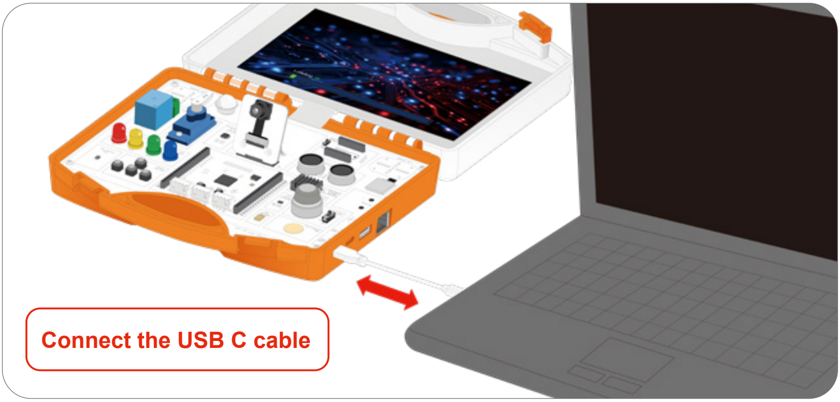
To successfully call the contents of the **bsp_display and bsp_ultrasonic** folders within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_display bsp_ultrasonic)
```

First, the directories for source files and header files are defined, along with the required driver **libraries—specifically**, the driver libraries for linking **bsp_display** and **bsp_ultrasonic**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver functionalities.

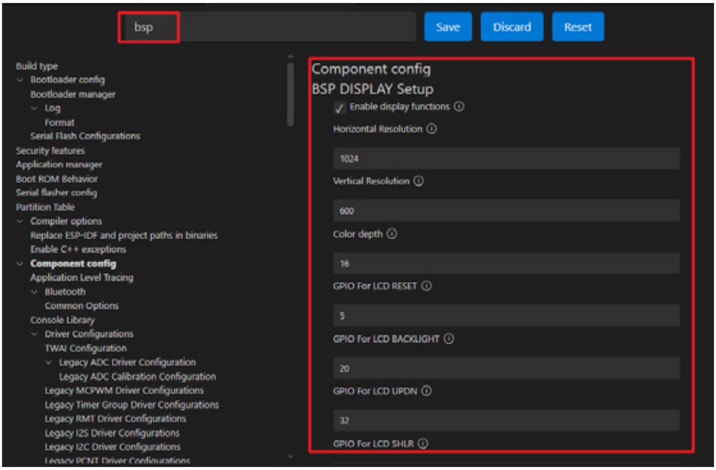
1.4 Programming procedure

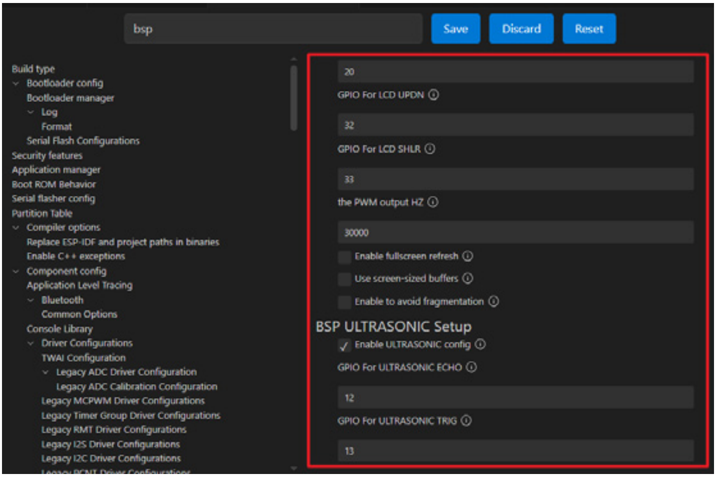
Connect the P4 device to the computer via USB



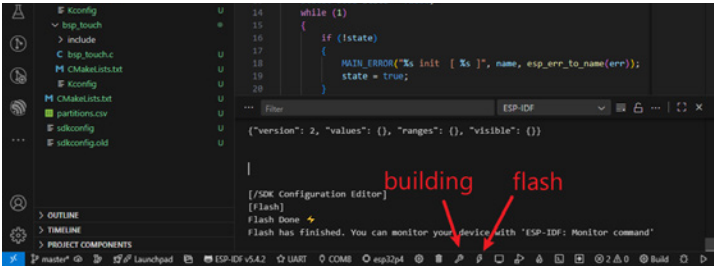
1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the screen and ultrasonic pins.





1.4.3 Click Compile. Once compilation is successful, click Download.

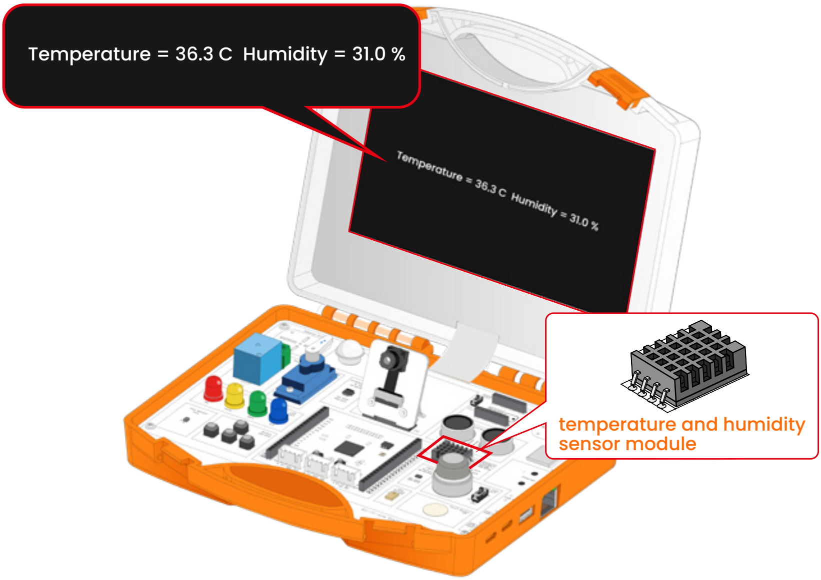


Lesson 11 - DHT20 Temp Humidity

Introduction

This chapter's tutorial introduces the interface application between the ESP32-P4 and the DHT20 digital temperature and humidity sensor. Through example routines for reading temperature and humidity data, it assists in understanding how to utilise digital sensors. As a common environmental monitoring case, temperature and humidity acquisition provides readers with an intuitive understanding of the interaction between the ESP32-P4 and peripherals, laying the groundwork for subsequent complex IoT and smart home projects.

Project Demonstration Effect



This chapter is divided into the following subsections

- 1.1 Introduction to the DHT20 Temperature and Humidity Sensor
- 1.2 Hardware Design
- 1.3 Programme Design
- 1.4 Download and Verification

1.1 DHT20 Temperature and Humidity Sensor Introduction

1.1.1 DHT20 Introduction

The DHT20 is a digital temperature and humidity sensor incorporating an integrated capacitive humidity sensor and temperature measurement element, alongside a built-in 12-bit ADC and digital signal processing circuitry. Unlike conventional analogue sensors, the DHT20 communicates with the host microcontroller via an I²C bus, enabling direct output of calibrated and compensated temperature and humidity data.

Its key features include:

① Digital output: I²C communication eliminates noise and drift issues associated with analogue acquisition;

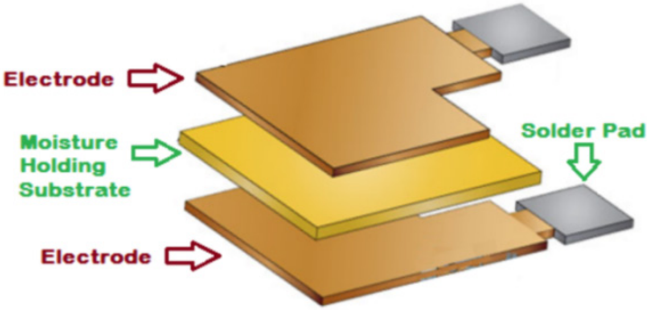
② High precision: Typical humidity accuracy ±3% RH, temperature accuracy ±0.5 ℃;

③ Low power consumption: Typical operating current < 1 mA, suitable for battery-powered devices;

④ Calibration compensation: Factory-calibrated for immediate measurement upon power-up, requiring no additional calibration;

⑤ Rapid response: Typical humidity response time less than 10 seconds.

1.1.2 Working Principle



DHT20 internal components:

Capacitive humidity sensor: Comprising a humidity-sensitive polymer film and electrodes. When air humidity changes, the dielectric constant of the film alters, causing a corresponding change in capacitance.

Temperature sensor: Utilises a high-precision temperature-sensitive element (e.g., a silicon sensor with temperature drift compensation).

Signal processing circuit: Transmits temperature and humidity signals to an analogue-to-digital converter (ADC), which outputs standardised digital values via internal compensation algorithms.

① Humidity measurement principle

The capacitive sensor outputs a capacitance signal that varies with humidity. This is converted to a digital value by the ADC and then scaled to relative humidity (RH%) using a calibration curve.

② Temperature Measurement Principle

The resistance or voltage of the temperature-sensitive element varies with temperature. After ADC conversion, a digital temperature value (in degrees Celsius) is obtained.

③ Data Calculation Formula

According to the DHT20 data sheet, the raw data read is a 20-bit binary number, which must be converted to the actual physical quantity:

$$RH(\%) = \frac{S_{RH}}{2^{20}} \times 100$$

$$T(℃) = \frac{S_T}{2^{20}} \times 200 - 50$$

1.1.3 Pin Description

The DHT20 module typically features a 4-pin interface:

VCC: Supply voltage 2.0V–5.5V (the ESP32-P4 development board's 3.3V power supply can provide direct power)

GND: Ground

SDA: I^2C data line

SCL: I^2C clock line

The default I^2C address is 0x38, supporting both standard mode (100 kHz) and fast mode (400 kHz).

1.1.4 Applications and Influencing Factors

Application scenarios: Widely employed in smart homes, environmental monitoring, weather stations, warehouse surveillance, and similar contexts.

Influencing factors:

Rapid temperature fluctuations may cause humidity measurement delays;

Prolonged operation in high-humidity environments necessitates attention to sensor saturation issues;

Airflow velocity impacts sensor response time.

1.2 Hardware design

The ESP32-P4 is connected to the DHT20 via the I²C bus. The wiring configuration is as follows:

ESP32-P4 GPIOxx (SDA) → DHT20 SDA

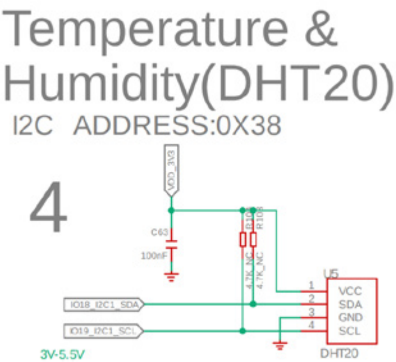
ESP32-P4 GPIOyy (SCL) → DHT20 SCL

3.3V → VCC

GND → GND

(The specific pins for SDA and SCL may be selected according to the actual pin definitions of the development board, and pull-up resistors of 4.7kΩ to 10kΩ must be added.)

The supply voltage is 3.3V, with a typical operating current of 0.5mA, and can be directly powered by the ESP32-P4 development board.

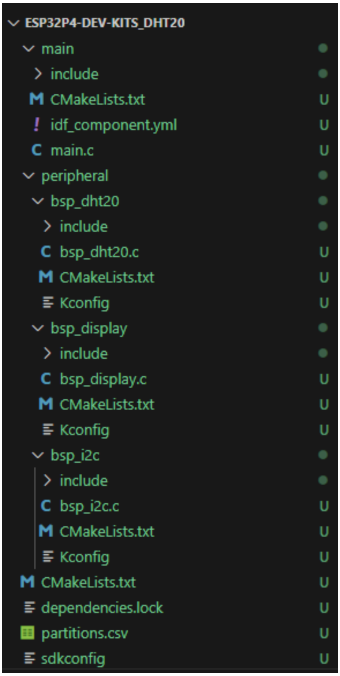


Schematic diagram

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_dht20** example, new folders named **bsp_i2c** and **bsp_dht20** were created under the **ESP32P4-dev-kits_dht20\peripheral** directory. Within the **bsp_dht20** and **bsp_i2c** paths, new include folders, **CMakeLists.txt** files, and Kconfig files were established. The **bsp_i2c** folder houses the **bsp_i2c.c** driver file, while the bsp_dht20 folder contains the **bsp_dht20.c** driver file. The respective include folders house the .h header files, while the **CMakeLists.txt** file integrates the drivers into the build system, enabling project utilisation of their functionality. The Kconfig file, meanwhile, loads the entire driver along with GPIO pin definitions into the **SDKConfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 I2C Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The I2C driver source code comprises two files: **bsp_i2c.c** and **bsp_i2c.h**.

Below we shall first analyse the bsp_i2c.h programme: it contains relevant definitions for the I2C pins and function declarations.

/* Header file references */

```
/*——————————————————————————————————————————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 "driver/i2c_master.h" //References for I2C Master Function-related API Functions
/*——————————————————————————————————————————Header file declaration end——————————————————————————————————————*/
```

/* Function declarations and macro definition declarations */

```c
#ifdef CONFIG_BSP_I2C_ENABLED

#define I2C_MASTER_PORT CONFIG_I2C_PORT_NUM // I2C port number
#define I2C_GPIO_SCL CONFIG_I2C_GPIO_SCL    // I2C SCL GPIO
#define I2C_GPIO_SDA CONFIG_I2C_GPIO_SDA    // I2C SDA GPIO

esp_err_t i2c_init(void);                                                                                                          // Initialize I2C
i2c_master_dev_handle_t i2c_dev_register(uint16_t dev_device_address);                                                             // The I2C slave device registration function
esp_err_t i2c_read(i2c_master_dev_handle_t i2c_dev, uint8_t *read_buffer, size_t read_size);                                       // I2C Read Function
esp_err_t i2c_write(i2c_master_dev_handle_t i2c_dev, uint8_t *write_buffer, size_t write_size);                                    // I2C Write Function
esp_err_t i2c_write_read(i2c_master_dev_handle_t i2c_dev, uint8_t read_reg, uint8_t *read_buffer, size_t read_size, uint16_t delayms); // I2C Write and Read Function
esp_err_t i2c_read_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t *read_buffer, size_t read_size);                 // I2C Read Register Function
esp_err_t i2c_write_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t data);                                          // I2C Write Register Function
#endif
```

Next, we shall analyse the **bsp_i2c.c** programme: initialising and configuring the I2C pins, and exposing the API interface functions.

/* I2C initialisation function i2c_init */

```c
#ifdef CONFIG_BSP_I2C_ENABLED
esp_err_t i2c_init(void)
{
    static esp_err_t err = ESP_OK;
    /*I2C master bus specific configurations*/
    i2c_master_bus_config_t conf = {
        .i2c_port = I2C_MASTER_PORT,           /* I2C port number*/
        .sda_io_num = I2C_GPIO_SDA,            /* GPIO number of I2C SDA signal*/
        .scl_io_num = I2C_GPIO_SCL,            /*GPIO number of I2C SCL signal*/
        .clk_source = I2C_CLK_SRC_DEFAULT,     /*Clock source of I2C master bus*/
        .glitch_ignore_cnt = 7,                /* If the glitch period on the line is less than this value, it can be filtered out, typically value is 7*/
#ifdef CONFIG_I2C_GPIO_PULLUP
        .flags.enable_internal_pullup = true, /*Enable internal pullups*/
#else
        .flags.enable_internal_pullup = false,
#endif
    };
    err = i2c_new_master_bus(&conf, &i2c_bus_handle); /*Allocate an I2C master bus*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the i2c_init function, the member variables of the **i2c_master_bus_config_t** structure are first configured with parameters. Subsequently, the **i2c_new_master_bus** function is invoked to establish a new I2C bus controller. The parameters for the **i2c_master_bus_config_t** structure members are as follows:

i2c_port: I2C bus controller port selection

sda_io_num: I2C bus SDA data line

scl_io_num: I2C bus SCL clock line

clk_source: I2C bus clock source selection

glitch_ignore_cnt: Glitches shorter than this duration are ignored; typically set to 7

flags.enable_internal_pull: Enable internal pull-up resistors

/* I2C slave device registration function **i2c_dev_register** */

```c
i2c_master_dev_handle_t i2c_dev_register(uint16_t dev_device_address)
{
    esp_err_t err = ESP_OK;
    i2c_master_dev_handle_t dev_handle = NULL; /*Type of I2C master bus device handle*/
    i2c_device_config_t cfg = {
        .dev_addr_length = I2C_ADDR_BIT_LEN_7, /*Select the address length of the slave device*/
        .device_address = dev_device_address,  /*I2C device raw address. (The 7/10 bit address without read/write bit)*/
        .scl_speed_hz = 400000,                /*I2C SCL line frequency*/
    };
    err = i2c_master_bus_add_device(i2c_bus_handle, &cfg, &dev_handle); /*Add I2C master BUS device*/
    if (err == ESP_OK)
        return dev_handle;
    return 0;
}
```

This function takes one parameter: the 7-bit address of the I2C slave device to be registered. Using the device address configuration structure, the address is bound to the slave device via the `**i2c_master_bus_add_device**` function, returning a device handle (usable for subsequent read operations, write operations, etc.).

/* I2C read function `**i2c_read**` */

```
esp_err_t i2c_read(i2c_master_dev_handle_t i2c_dev, uint8_t *read_buffer, size_t read_size)
{
    return i2c_master_receive(i2c_dev, read_buffer, read_size, 1000); /*Perform a read transaction on the I2C bus*/
}
```

I2C read operation: input the I2C device handle, read the receive buffer, and specify the read quantity

/* I2C write function **i2c_write** */

```
esp_err_t i2c_write(i2c_master_dev_handle_t i2c_dev, uint8_t *write_buffer, size_t write_size)
{
    return i2c_master_transmit(i2c_dev, write_buffer, write_size, 1000); /*Perform a write transaction on the I2C bus*/
}
```

I2C write operation: input the I2C device handle, write the array, and the number of bytes to write

/* I2C read register function **i2c_read_reg** */

```
esp_err_t i2c_read_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t *read_buffer, size_t read_size)
{
    return i2c_master_transmit_receive(i2c_dev, &reg_addr, 1, read_buffer, read_size, 1000); /*Perform a write-read transaction on the I2C bus*/
}
```

I2C register read operation: input the I2C device handle, register address, read buffer, and read count.

/* I2C register write function **i2c_write_reg** */

```
esp_err_t i2c_write_reg(i2c_master_dev_handle_t i2c_dev, uint8_t reg_addr, uint8_t data)
{
    uint8_t write_buf[2] = {reg_addr, data};                             /*Register address and data*/
    return i2c_master_transmit(i2c_dev, write_buf, sizeof(write_buf), 1000); /*Perform a write transaction on the I2C bus*/
}
```

I²C write register operation: input the I²C device handle, specify the register address, and write the data (single value).

1.3.2 Kconfig file

The primary function of this file is to incorporate the required configuration into the sdkconfig file, enabling certain parameter settings to be modified via a graphical interface. Here, **18** corresponds to **GPIO_NUM_18**, and **19** corresponds to **GPIO_NUM_19**.

```
menu "BSP I2C Setup"
    config BSP_I2C_ENABLED
        bool "Enable I2C functions"

    if BSP_I2C_ENABLED
        config I2C_PORT_NUM
            int "I2C Master Port"
            default 0

        config I2C_GPIO_SCL
            int "GPIO For I2C SCL"
            default 19

        config I2C_GPIO_SDA
            int "GPIO For I2C SDA"
            default 18

        config I2C_GPIO_PULLUP
            bool "Enable I2C Pullup resistor"

    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_i2c** driver. To successfully call functions from the **bsp_i2c** folder within other functions, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_i2c** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the *idf_component_register* command, enabling the project to utilise the **bsp_i2c** driver functionality.

1.3.4 DHT20 Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The DHT20 driver source code comprises two files: **bsp_dht20.c** and **bsp_dht20.h**.

Below we shall first analyse the **bsp_dht20.h** programme: it defines relevant pins for the temperature and humidity sensor and declares functions.

/* Header file references */

```
/*------------------------------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------------------------------*/
```

/* Function declarations and macro definition declarations */

```
#ifdef CONFIG_BSP_DHT20_SENSOR_ENABLED

#define DHT20_I2C_ADDRESS CONFIG_DHT20_I2C_ADDRESS // 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
#endif
```

Next, we shall analyse the **bsp_dht20.c** programme: initialising the DHT20 sensor configuration and exposing API interface functions.

/* DHT20 initialisation function **dht20_begin** */

```c
esp_err_t dht20_begin(void)
{
    esp_err_t err = ESP_OK;
    dht20_handle = i2c_dev_register(DHT20_I2C_ADDRESS); /*DHT20 sensor address registration*/
    if (dht20_handle != NULL)
    {
        if (dht20_reset_sensor() >= 255) /* Check status register, if status not eq 0x18 reset the necearry registers*/
            err = ESP_FAIL;
    }
    else
    {
        err = ESP_FAIL;
        return err;
    }
    return err;
}
```

Within the `**dht20_begin**` function, the `**DHT20**` sensor is first registered on the I²C bus using the `**i2c_dev_register**` function, which returns an operation handle. Subsequently, if the returned device handle is not null, the `**dht20_reset_sensor**` function is invoked to reset the sensor, ensuring it is in a state where temperature and humidity data can be read.

/* DHT20 sensor reset function **dht20_reset_sensor** */

```c
static uint8_t dht20_reset_sensor(void)
{
    static uint8_t rst_count = 0;
    uint8_t status = dht20_status(); /*Obtain the status of the DHT20 sensor*/
    DHT20_DEBUG("Sensor status: %s - 0x%02X", print_byte(status), status);
    while ((status & 0x18) != 0x18) /*If the DHT20 sensor status code is not 0x18*/
    {
        DHT20_DEBUG("Sensor status: %s - 0x%02X", print_byte(status), status);
        rst_count++;
        if (dht20_reset_register(0x1B) != ESP_OK) /*DHT20 sensor register reset function*/
        {
            rst_count++;
        }
        if (dht20_reset_register(0x1C) != ESP_OK)
        {
            rst_count++;
        }
        if (dht20_reset_register(0x1E) != ESP_OK)
        {
            rst_count++;
        }
        if (rst_count >= 255) /*Exceed the retry count*/
            return rst_count;
        DHT20_DEBUG("Registers resetted [%d] times!", rst_count);
        status = dht20_status(); /*Obtain the status of the DHT20 sensor*/
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    return rst_count;
}
```

This function first calls the `**dht20_status**` function to obtain the current status of the DHT20 sensor. If the current status code is not **0x18**, it calls the `**dht20_reset_register**` function to reset the 0x1B, 0x1C, and 0x1E registers and assess whether the reset was successful. Following the reset, it re-calls the `**dht20_status**` function to examine the sensor's current status code. Should the retry count exceed 254, the function terminates abruptly and returns the accumulated retry count.

/* Function `**dht20_status**` for reading DHT20 device status codes */

```
uint8_t dht20_status(void)
{
    esp_err_t err = ESP_OK;
    static uint8_t txbuf = 0x71;
    static uint8_t rxdata;
    err = i2c_write(dht20_handle, &txbuf, 1); /*I2C write command*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(100 / portTICK_PERIOD_MS);
    err = i2c_read(dht20_handle, &rxdata, 1); /*I2C read status*/
    if (err != ESP_OK)
        return err;
    return (rxdata);
}
```

This function uses the i2c_write function to send 0x71 to the DHT20 sensor, then reads the sensor to retrieve its status code.

/* DHT20 sensor reset register function **dht20_reset_register** */

```
static esp_err_t dht20_reset_register(uint8_t reg)
{
    esp_err_t err = ESP_OK;
    static uint8_t values[3] = {0};
    uint8_t txbuffer[3] = {reg, 0x00, 0x00};    /*Register address*/
    err = i2c_write(dht20_handle, txbuffer, 3); /*I2C write command*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(5 / portTICK_PERIOD_MS);
    err = i2c_read(dht20_handle, values, 3); /*I2C read status*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(10 / portTICK_PERIOD_MS);

    memset(txbuffer, 0, sizeof(txbuffer));
    txbuffer[0] = (0xB0 | reg);
    txbuffer[1] = values[1];
    txbuffer[2] = values[2];

    err = i2c_write(dht20_handle, txbuffer, 3);
    if (err != ESP_OK)
        return err;
    vTaskDelay(5 / portTICK_PERIOD_MS);
    return err;
}
```

First configure the data array for writing, then call the **i2c_write** function to write data. After a 5ms delay, call i2c_read to read the sensor and obtain its value. Assign the acquired data to the first and second elements of the data array. The first element of the data array uses the bitwise OR operator to set the high-order bit of the input register address to 1, then rewrites the sensor.

/* Function to determine whether the DHT20 status code is normal: **dht20_is_calibrated** */

```
esp_err_t dht20_is_calibrated(void)
{
    esp_err_t err = ESP_OK;
    uint8_t status_byte = dht20_status();
    if ((status_byte & 0x18) != 0x18) /*If the DHT20 sensor status code is not 0x18*/
    {
        err = ESP_FAIL;
    }
    return err;
}
```

read the receive buffer, and call the dht20_status function with the read count to obtain the sensor status code. If the status code is not equal to 0x18, return ESP_FAIL.

/* DHT20 read temperature and humidity data function dht20_read_data */

```c
esp_err_t dht20_read_data(dht20_data_t *data)
{
    esp_err_t err = ESP_OK;
    data->humidity = 0.0f;
    data->raw_humid = 0;
    data->temperature = 0.0f;
    data->raw_temp = 0;
    static uint8_t txbuf[3] = {0xAC, 0x33, 0x00}; /*Start measuring command + 2 bytes as parameters*/
    static uint8_t status_byte[1] = {0};          /*retrieve measuring status*/
    static uint8_t rxdata[7] = {0};               /*retrieve measurement data (temp + humid)*/
    err = i2c_write(dht20_handle, txbuf, 3);
    if (err != ESP_OK)
        return err;
    vTaskDelay(80 / portTICK_PERIOD_MS); /*minimum delay as per datahseet*/
    DHT20_DEBUG("Reading registers.....");
    err = i2c_read(dht20_handle, status_byte, 1); /*read status byte to check sensor is done measuring*/
    if (err != ESP_OK)
        return err;
    unsigned long start_time = esp_timer_get_time() / 1000;
    while ((status_byte[0] >> 7) != 0) /*while meassurment is ongoing*/
    {
        if ((esp_timer_get_time() / 1000) - start_time >= DHT20_MEASURE_TIMEOUT)
        {
            return ESP_ERR_TIMEOUT; /*Timeout*/
        }
        portYIELD(); /*give scheduler some time for other tasks while we're still measuring*/
    }
```

```c
    err = i2c_read(dht20_handle, rxdata, 7); /*sensor idle let's read all data*/
    if (err != ESP_OK)
        return err;
    DHT20_DEBUG("Byte1: %s", print_byte(rxdata[0]));
    DHT20_DEBUG("Byte2: %s", print_byte(rxdata[1]));
    DHT20_DEBUG("Byte3: %s", print_byte(rxdata[2]));
    DHT20_DEBUG("Byte4: %s", print_byte(rxdata[3]));
    DHT20_DEBUG("Byte5: %s", print_byte(rxdata[4]));
    DHT20_DEBUG("Byte6: %s", print_byte(rxdata[5]));
    DHT20_DEBUG("CRC Byte: %s", print_byte(rxdata[6]));

    uint8_t get_crc = dht20_crc8(rxdata, 6); /*CRC calculation*/
    DHT20_DEBUG("Data byte 7: 0x%02X, calculated crc8: 0x%02X", rxdata[6], get_crc);

    if (rxdata[6] == get_crc) /*Compare CRC calues and continue if they match*/
    {
        uint32_t raw_humid = rxdata[1];
        raw_humid <<= 8;
        raw_humid += rxdata[2];
        raw_humid <<= 4;
        raw_humid += rxdata[3] >> 4;
        data->raw_humid = raw_humid;
        data->humidity = (float)(raw_humid / 1048576.0f) * 100.0f; /*convert RAW to Humidity in %*/

        DHT20_DEBUG("Humidity raw: %lu - Converted: %.1f %%", data->raw_humid, data->humidity);

        uint32_t raw_temp = (rxdata[3] & 0x0F);
        raw_temp <<= 8;
        raw_temp += rxdata[4];
        raw_temp <<= 8;
        raw_temp += rxdata[5];
        data->raw_temp = raw_temp;
        data->temperature = (float)(raw_temp / 1048576.0f) * 200.0f - 50.0f; /*convert RAW to Celsius C*/

        DHT20_DEBUG("Temperature raw: %lu - Converted: %.2fC.", data->raw_temp, data->temperature);
    }
    else
    {
        DHT20_ERROR("CRC Checksum failed !!!");
        return ESP_ERR_INVALID_CRC;
    }
    return 0.0f;
}
```

First, initialise the variables. Call the **`i2c_write`** function to write the measurement initiation command to the sensor. After a delay of 80 milliseconds, call the **`i2c_read`** function to read the status byte and determine whether the sensor has completed the measurement. If the read time exceeds the set maximum measurement duration, exit and return a timeout error. If the measurement is complete, call the `i2c_read` function again to read seven bytes of data (the sensor's full measurement includes a CRC checksum). Convert the checksum for the first six bytes of the read data using the **`dht20_crc8`** function (CRC8 check sum polynomial: **$CRC[7:0] = 1 + x^4 + x^5 + x^8$)**. The calculated check sum is compared with the read check sum. If they match, the data is valid. Finally, the read data is converted using the following formula:

The first byte of the read data is the status code. The upper 4 bits of the second, third, and fourth bytes constitute the humidity data. To convert this humidity data: - Shift the second byte data left by 8 bits. - Add the third byte data. - Shift the result left by 4 bits. - Add the third byte data shifted right by 4 bits. Divide the converted humidity data by 2^{20} × 100% (e.g., if the second byte data is 0x18, third byte data is 0x22, fourth byte data is 0x11, conversion: ((((0x18<<8)|0x22)<<4) | (0x11>>4))=0x18221 (0x18221/(2^20))*100% = 9.43% (rounded to the nearest whole number)

Similarly, the temperature data comprises the lower four bits of the fourth byte, the fifth byte, and the sixth byte. To convert the temperature data:

- Add 0x0F to the fourth byte and extract the fourth bit.

- Shift this extracted bit left by 8 positions and add it to the fifth byte. After addition, shift the result left by 8 bits and add the sixth byte's data. Divide the converted temperature data by 2^{20} * 200, then subtract 50 from the result. (For example, if the fourth byte is 0x16, fifth byte is 0xF5, and sixth byte is 0xF2, the conversion yields:

(((((0x16 & 0x0F) << 8) | 0xF5) << 8) | 0xF2) = 0x6F5F2 (((0x6F5F2 / 2^20) * 200) - 50) = 37 degrees (rounded to the nearest whole degree))

1.3.5 Kconfig file

The primary function of this file is to add the required configuration to the **sdkconfig** file, enabling certain parameter settings to be modified via a graphical interface. Here, **0x38** represents the 7-bit address for the DHT20 sensor.

```
menu "BSP DHT20 Setup"

    config BSP_DHT20_SENSOR_ENABLED
        bool "Enable DHT20 Temperature and Humidity sensor"
        depends on BSP_I2C_ENABLED

    if BSP_DHT20_SENSOR_ENABLED
        config DHT20_I2C_ADDRESS
            hex "DHT20 I2C Address"
            default 0x38
    endif
endmenu
```

1.3.6 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_dht20** driver. To successfully call functions from the **bsp_dht20** folder within other functions, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_dht20** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver library (**bsp_i2c**). Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to **utilise the bsp_dht20** driver functionality.

1.3.7 main folder

The main folder serves as the core directory for programme execution, containing the main function executable **main.c** and the **main.h** header file within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The **main.h** file primarily references required header files: functions utilising the **bsp_display** driver necessitate inclusion of the **bsp_display** header file, while those employing the **bsp_dht20** driver require the **bsp_dht20** header file.

Below is an analysis of the main.c programme: system initialisation and execution of functions for **I2C**, **DHT20** sensor, and display functionality.

```
#ifdef CONFIG_BSP_I2C_ENABLED
    err = i2c_init();/*I2C Initialization*/
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
#ifdef CONFIG_BSP_DHT20_SENSOR_ENABLED
    err = dht20_begin();/*DHT20 Initialization*/
    if (err != ESP_OK)
        init_fail("dht20", err);
#endif
#endif
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init();/*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
```

This code resides within the **init function**, which is used to store initialisation functions requiring invocation and to evaluate their return status. Should the return status not be **ESP_OK**, the code will print an error message and halt further execution.

/* Screen initialisation and display function dht20_display */

```c
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();
    }
}
```

This function primarily configures the initial screen display content: setting background colours and text display via lvgl controls.

The **lv_label_set_text** function sets the text displayed on the control.

The **lv_style_set_bg_opa** function sets the background colour of the style.

The **lv_obj_set_style_text_color** function sets the text display colour.

The **lv_obj_set_style_text_font** function sets the text font size.

The **lv_obj_set_style_bg_color** function sets the background colour.

lv_obj_set_style_bg_opa function sets background transparency

Note: When calling lvgl functions outside lvgl thread functions, a mutex lock must be acquired. lvgl_port_lock function acquires the mutex lock, lvgl_port_unlock function releases it.

/* Screen data refresh display function update_dht20_value */

```c
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*/
    }
}
```

This function employs the snprintf function to format the two acquired float-type data into a string, subsequently refreshing the display content using the **lv_label_set_text** function.

It is worth noting that the snprintf function appends a terminating character to the end of the formatted string. As **lv_label_set_text** recognises strings by identifying this terminating character, utilising snprintf for string formatting constitutes a preferable approach.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    init();
#ifdef CONFIG_BSP_DHT20_SENSOR_ENABLED
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100);                                                                  /*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*/
#endif
#endif
    MAIN_INFO("----------Start the test----------");
}
```

Within the app_main function, the backlight brightness is first set to 100%, followed by initialising the screen display content. Finally, a FreeRTOS thread is created to handle data processing and screen refresh operations.

/* DHT20 temperature and humidity data processing and screen refresh thread **dht20_read_task */**

```c
void dht20_read_task(void *param)
{
    static dht20_data_t measurements;
    while (1)
    {
        if (dht20_is_calibrated() == ESP_OK) /*The function for determining whether the DHT20 sensor is ready or not*/
        {
            MAIN_INFO("is calibrated....");
        }
        else
        {
            MAIN_INFO("is NOT calibrated....");
            if (dht20_begin() != ESP_OK) /*Reinitialize the DHT20 sensor*/
            {
                MAIN_ERROR("dht20 init again false");
                vTaskDelay(100 / portTICK_PERIOD_MS);
                continue;
            }
        }
        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);
    }
}
```

Within the DHT20 temperature and humidity data processing and screen display refresh thread, variables are first initialised. Subsequently, a while loop is established. Within this loop, the `**dht20_is_calibrated**` function is invoked to determine the sensor status code. Should the result not be `**ESP_OK**`, the DHT20 sensor is reinitialised. If initialisation fails, the `**continue**` statement is executed, returning the loop. If ESP_OK is returned, the **dht20_read_data** function is invoked to retrieve temperature and humidity data. Upon successful acquisition, the **update_dht20_value** function refreshes the screen display. Should retrieval fail, the screen displays **'dht20 read data error'**. The concluding 1-second delay ensures data is refreshed once per second.

1.3.8 CMkaLists.txt file

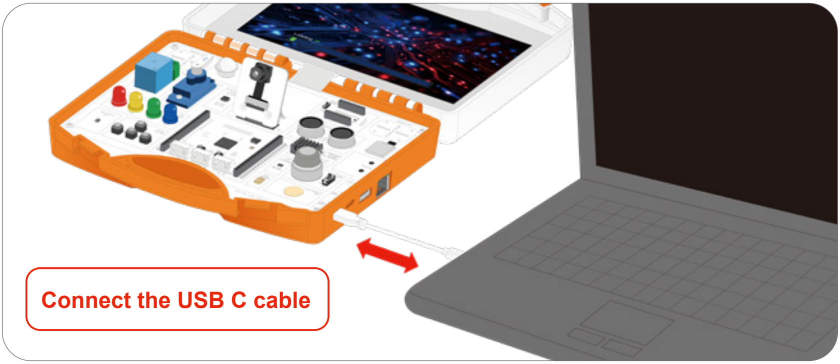
To successfully call the contents of the **bsp_display** and **bsp_dht20** folders within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_i2c bsp_display bsp_dht20)
```

First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries for linking **bsp_display**, **bsp_dht20**, and **bsp_i2c**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver functionalities.

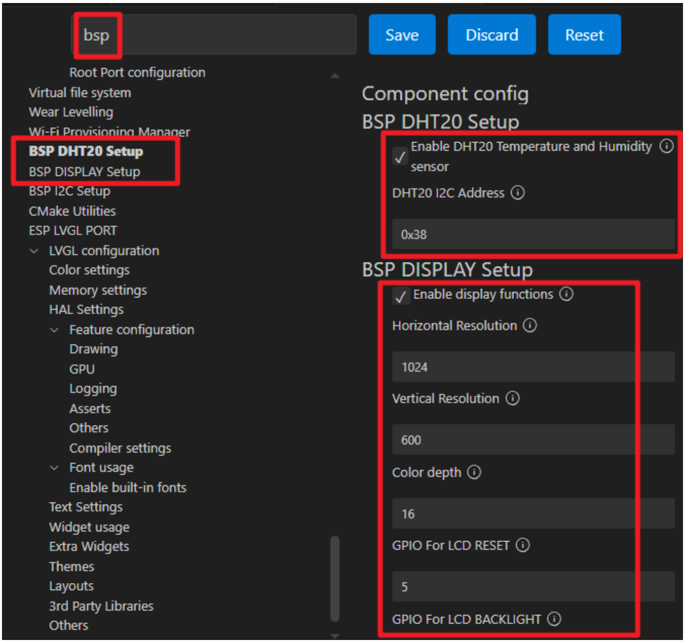
1.4 Programming procedure

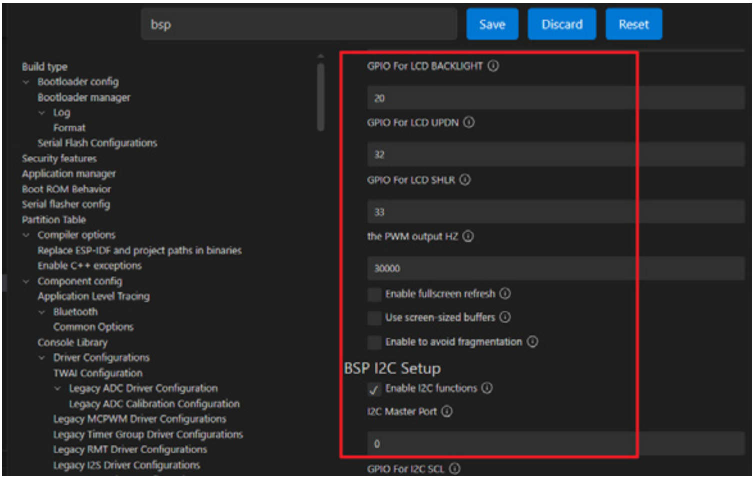
Connect the P4 device to the computer via USB

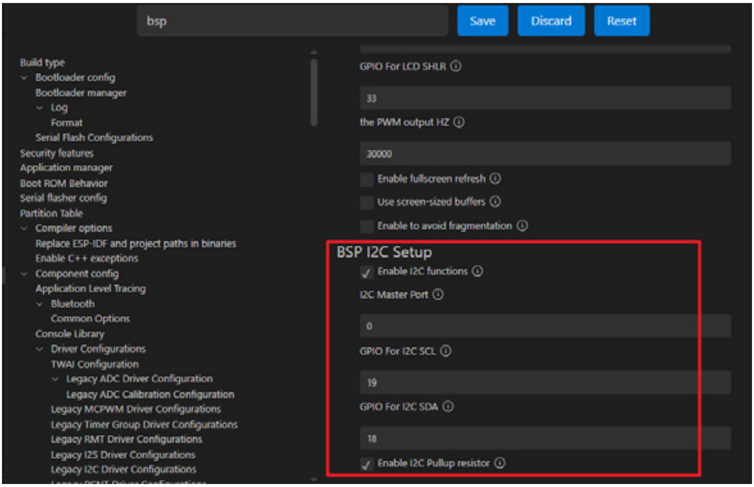


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the DSI and dht20 pins.







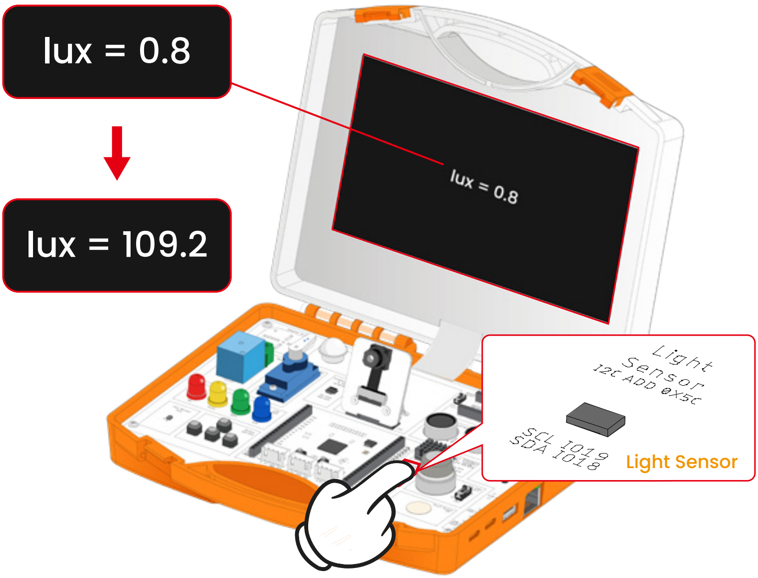
1.4.3 Click Compile. Once compilation is successful, click Download.

Lesson 12 - BH1750 Light Sensor

Introduction

This chapter's tutorial introduces the application of the ESP32-P4's I2C peripheral. Through experiments with the BH1750 light intensity sensor, it aids in understanding the fundamental principles of I2C communication and ambient light data acquisition. The BH1750 is a commonly used digital lux meter capable of directly outputting light intensity values in lux units, making it highly suitable for projects such as smart lighting, automatic dimming, and environmental monitoring. This chapter progresses step-by-step, laying the groundwork for subsequent applications involving additional I²C devices.

Project Demonstration Effect



This chapter is divided into the following subsections

- 1.1 Introduction to BH1750 and the I²C Bus
- 1.2 Hardware Design
- 1.3 Software Design
- 1.4 Download and Verification

1.1 BH1750 Introduction

1.1.1 BH1750 Introduction

The BH1750 is a digital light intensity sensor manufactured by ROHM, capable of converting light intensity into digital signals and outputting them via the I²C bus. Compared to traditional analogue photoresistors, the BH1750 offers advantages such as high precision, strong anti-interference capabilities, and rapid response, making it highly suitable for embedded systems.

Key features include:

① I²C digital output: Eliminates the need for analogue-to-digital conversion (ADC), enabling direct retrieval of light intensity values (in lux) via I²C.

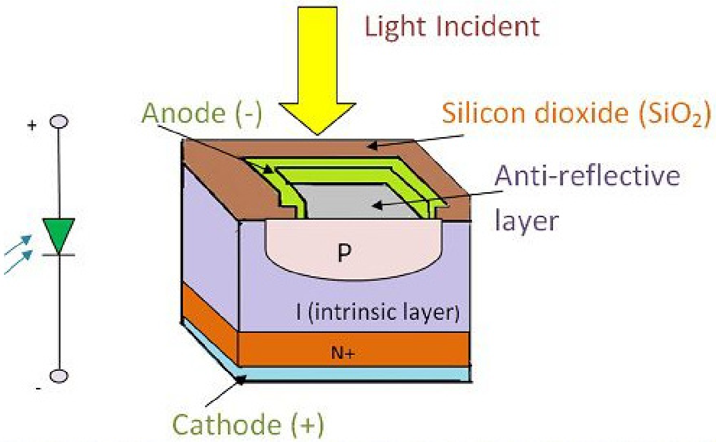
② Wide sensitivity range: 1 to 65,535 lux, accommodating scenarios from night-time illumination to intense sunlight.

③ Low-power design: Operating current of approximately 0.12mA, conserving energy.

④ Automatic range switching: Switches between high-resolution mode (1 lx) and low-resolution mode (4 lx).

⑤ Compact structure: Small package size facilitates easy integration into various control boards.

1.1.2 Principles of Light Measurement



Photodiode Symbol & Construction of Photodiode

The BH1750 detects ambient light intensity via its integrated photodiode, utilising an analogue-to-digital converter (ADC) to convert light signals into digital values for output. Its internal structure primarily comprises the following components:

Photodiode: Converts light energy into electrical signals;

Integrating circuit: Performs time integration on the current signal to obtain the average light intensity;

Analogue-to-digital converter (ADC): Converts electrical signals into digital signals;

Register and I²C interface: Stores measurement results and transmits them to the host controller via I²C.

The BH1750 outputs data in lux (Lx), representing the luminous flux received per unit area. In other words, higher values indicate brighter environments.

1.1.3 Operating Modes of BH1750

BH1750 offers multiple measurement modes, allowing flexible selection according to the application scenario:

modes	Function Description	Typical measurement time	resolution
H-Resolution Mode	High-resolution mode	120ms	1 lx
H-Resolution Mode2	High-precision mode	120ms	0.5 lx
L-Resolution Mode	Low-resolution mode	16ms	4 lx

Mode selection is achieved by writing different command words (such as 0x10, 0x11, 0x13) to the BH1750, enabling highly streamlined communication.

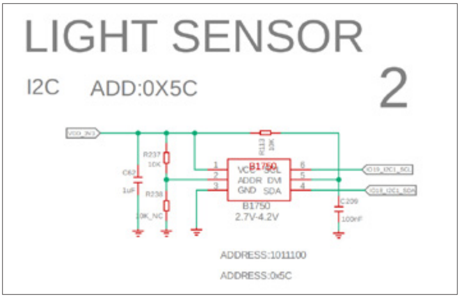
1.1.4 I2C Address and Wiring Instructions

The default I²C address for the BH1750 module is 0x5C.

2.2 Hardware design

In the experiment, the BH1750 module is connected to the ESP32-P4 development board as follows:

ESP32-P4	BH1750
3V3	VCC
GND	GND
IO18 (SDA)	SDA
IO19 (SCL)	SCL

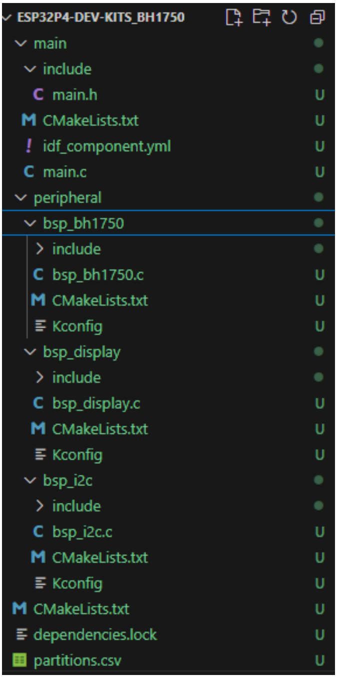


Schematic diagram

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_bh1750** example, a new folder named **bsp_bh1750** has been created under the **ESP32P4-dev-kits_bh1750\peripheral** directory. Within the **bsp_bh1750** path, a new include folder, **CMake-Lists.txt** file, and **Kconfig** file have been established. The **bsp_bh1750** folder houses the **bsp_bh1750.c** driver file. The include folder contains the **bsp_bh1750.h** header file, while the **CMakeLists.txt** file integrates the driver into the build system, enabling project utilisation of its functionality. The Kconfig file loads the entire driver alongside GPIO pin definitions into the sdkconfig file within the IDF platform (configu-rable via the graphical interface).

1.3.1 BH1750 Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The BH1750 driver source code comprises two files: **bsp_bh1750.c** and **bsp_bh1750.h**.

Below we shall first analyse the bsp_bh1750.h programme: it contains relevant definitions and function declarations corresponding to the light sensor pins.

/* Header file references */

```
/*——————————————————————————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 "bsp_i2c.h"
/*————————————————————————Header file declaration end————————————————————————*/
```

/* Function declarations and macro definitions */

```c
#ifdef CONFIG_BSP_BH1750_SENSOR_ENABLED

#define BH1750_I2C_ADDRESS CONFIG_BH1750_I2C_ADDRESS // The 7-bit I2C address of BH1750

#define BH1750_PWR_DOWN 0x00 // Power down
#define BH1750_PWR_ON 0x01   // Power on
#define BH1750_RST 0x07      // reset register
#define BH1750_CON_H 0x10    // 1lx, 120ms
#define BH1750_CON_H2 0x11   // 0.5lx, 120ms
#define BH1750_CON_L 0x13    // 4lx, 16ms
#define BH1750_ONE_H 0x20    // one test : 1lx, 120ms,
#define BH1750_ONE_H2 0x21   // one test : 0.5lx, 120ms,
#define BH1750_ONE_L 0x23    // one test : 4lx, 16ms,

esp_err_t bh1750_begin(void); // Initialization of BH1750 sensor
float bh1750_read_data();     // BH1750 Sensor lux Data Reading Function

#endif
```

Next, we shall analyse the **bsp_bh1750.c** programme: initialising and configuring the BH1750 sensor whilst exposing API interface functions.

/* BH1750 initialisation function **bh1750_begin** */

```c
esp_err_t bh1750_begin(void)
{
    esp_err_t err = ESP_OK;
    uint8_t tx_buf;
    tx_buf = BH1750_PWR_ON;
    bh1750_handle = i2c_dev_register(BH1750_I2C_ADDRESS); /*BH1750 sensor address registration*/
    if (bh1750_handle == NULL)
        return ESP_FAIL;
    err = i2c_write(bh1750_handle, &tx_buf, 1); /*Set the BH1750 sensor to start working*/
    if (err != ESP_OK)
        return err;
    tx_buf = BH1750_CON_H;
    err = i2c_write(bh1750_handle, &tx_buf, 1); /*Start measurement at 1lx resolution,Measurement Time is typically 120ms*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the `**bh1750_begin**` function, the BH1750 sensor is first registered on the I²C bus using the `**i2c_dev_register**` function, which returns an operation handle. Subsequently, if the returned device handle is not null, the `**i2c_write**` function is invoked to enable the sensor, ensuring it is in a state where illuminance data can be read. The test register is then written to enable measurements at 120ms intervals.

/* BH1750 illuminance data read function **bh1750_read_data** */

```c
float bh1750_read_data()
{
    esp_err_t err = ESP_OK;
    float lux;
    uint8_t sensorData[2] = {0};
    err = i2c_read(bh1750_handle, sensorData, 2); /*Read sensor data*/
    if (err != ESP_OK)
        return -1;
    lux = (sensorData[0] << 8 | sensorData[1]) / 1.2; /*Convert illuminance data*/
    return lux;
}
```

First, initialise the variables. Call the i2c_read function to read two bytes of data from the sensor. Convert the two read bytes using the following formula:

Shift the first byte 8 bits to the left, OR it with the second byte, then divide the result by 1.2

((X1<<8) | X2)/1.2

(Example: If the first byte is 0x02 and the second byte is 0x34, then:

((0x02<<8) | 0x34)/1.2 = 470)

1.3.2 Kconfig file

The primary function of this file is to incorporate the requisite configuration into the sdkconfig file, thereby enabling certain parameter adjustments to be made via a graphical interface. Here, 0x5C denotes the 7-bit address for the BH1750 sensor.

```
menu "BSP BH1750 Setup"

    config BSP_BH1750_SENSOR_ENABLED
        bool "Enable BH1750 illumination sensor"
        depends on BSP_I2C_ENABLED

    if BSP_BH1750_SENSOR_ENABLED
        config BH1750_I2C_ADDRESS
            hex "BH1750 I2C Address"
            default 0x5C
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the bsp_bh1750 driver. To successfully call the contents of the bsp_bh1750 folder from other functions, it is necessary to configure the CMakeLists.txt file within the bsp_bh1750 folder. The configuration details are as follows:

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

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

Within this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver library (bsp_i2c). Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the bsp_bh1750 driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the main.h header file within the include folder. Add the main folder to the CMakeLists.txt file within the build system.

The main.h file primarily references required header files: functions utilising the bsp_display driver necessitate inclusion of the bsp_display header file, while those employing the bsp_bh1750 driver require the bsp_bh1750 header file.

Below is an analysis of the main.c programme: system initialisation and execution of functions for I2C, the bh1750 sensor, and display functionality.

```
#ifdef CONFIG_BSP_I2C_ENABLED
    err = i2c_init(); /*I2C Initialization*/
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
#ifdef CONFIG_BSP_BH1750_SENSOR_ENABLED
    err = bh1750_begin(); /*BH1750 Initialization*/
    if (err != ESP_OK)
        init_fail("bh1750", err);
#endif
#endif
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
```

This code resides within the init function, which is employed to store initialisation functions requiring invocation and to evaluate their return outcomes. Should the return status deviate from ESP_OK, the code will output an error message and cease further execution.

/* Screen initialisation and display function **bh1750_display** */

```c
void bh1750_display()
{
    if (lvgl_port_lock(0))
    {
        bh1750_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(bh1750_data, &label_style, LV_PART_MAIN);                    /*Add a style to an object*/
        lv_obj_set_style_text_color(bh1750_data, LV_COLOR_WHITE, LV_PART_MAIN);       /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(bh1750_data, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_center(bh1750_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(bh1750_data, "lux = 0.0");                                  /*Set a new text for a label*/
        lvgl_port_unlock();
    }
}
```

This function primarily configures the initial screen display content: setting background colours and text display via lvgl controls.

The **lv_label_set_text** function sets the text displayed on the control.

The **lv_style_set_bg_opa** function sets the background colour of the style.

The **lv_obj_set_style_text_color** function sets the text display colour.

The **lv_obj_set_style_text_font** function sets the text font size.

The **lv_obj_set_style_bg_color** function sets the background colour.

The **lvgl_obj_set_style_bg_opa** function sets the background transparency.

It is worth noting: When calling lvgl functions outside of lvgl thread functions, a mutex lock must be acquired.

The **lvgl_port_lock** function acquires the mutex lock, while the **lvgl_port_unlock** function releases it.

/* Screen data refresh display function update_bh1750_value */

```c
void update_bh1750_value(float lux)
{
    if (bh1750_data)
    {
        char buffer[40];
        snprintf(buffer, sizeof(buffer), "lux = %.1f", lux); /*Format the data into a string*/
        lv_label_set_text(bh1750_data, buffer);              /*Set a new text for a label*/
    }
}
```

This function employs the snprintf function to format the acquired **float-type** data into a string, subsequently refreshing the displayed content via the **lv_label_set_text** function.

It is worth noting that the snprintf function appends a terminating character to the end of the formatted string. Furthermore, **lv_label_set_text recognises** strings by identifying this terminating character. Consequently, utilising the snprintf function for string formatting constitutes a preferable approach.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_BH1750_SENSOR_ENABLED
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100);                                                              /*Set the screen backlight to maximum brightness*/
    bh1750_display();                                                                 /*Set the screen's LVGL default display page*/
    xTaskCreate(bh1750_read_task, "read_bh1750", 4096, NULL, configMAX_PRIORITIES - 5, &read_bh1750); /*Create a BH1750 data display refresh thread*/
#endif
#endif
}
```

Within the app_main function, the backlight brightness is first set to 100%, followed by initialising the screen display content. Finally, a FreeRTOS thread is created to handle data processing and screen refresh operations.

/* BH1750 illuminance data processing and screen refresh thread bh1750_read_task */

```c
void bh1750_read_task(void *param)
{
    float lux = 0;
    while (1)
    {
        lux = bh1750_read_data(); /*Read the illuminance data of the BH1750 sensor*/
        if (lux == (-1))
        {
            if (lvgl_port_lock(0))
            {
                lv_label_set_text(bh1750_data, "bh1750 read data error"); /*Read failure message displayed*/
                lvgl_port_unlock();
            }
        }
        else
        {
            MAIN_INFO("lux:%.2f", lux);
            if (lvgl_port_lock(0))
            {
                update_bh1750_value(lux); /*Update the Bh1750 data displayed on the screen*/
                lvgl_port_unlock();
            }
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Within the BH1750 illuminance data processing and screen display refresh thread, variables are first initialised. Subsequently, a while loop is established. Within this loop, the bh1750_read_data function is invoked to acquire illuminance data. Upon successful retrieval, the update_bh1750_value function is employed to refresh the screen display. Should retrieval fail, the screen displays “bh1750 read data error”. The final 1-second delay ensures data is refreshed and retrieved once every second.

1.3.5 CMkaLists.txt file

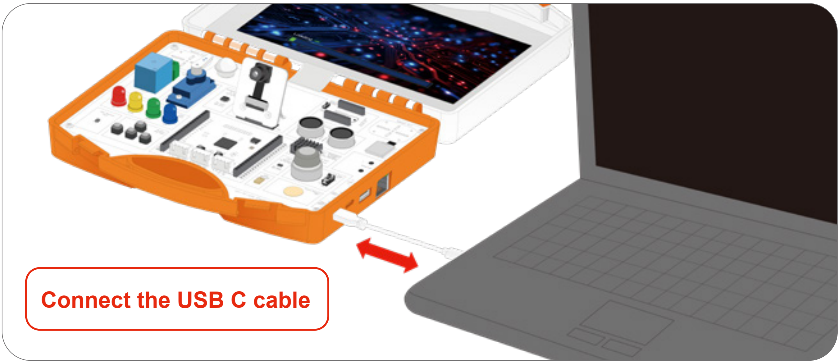
To successfully call the contents of the bsp_display and bsp_bh1750 folders within the main function, it is necessary to configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_i2c bsp_display bsp_bh1750)
```

First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries for linking bsp_display, bsp_bh1750, and bsp_i2c. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the main function to utilise these driver functionalities.

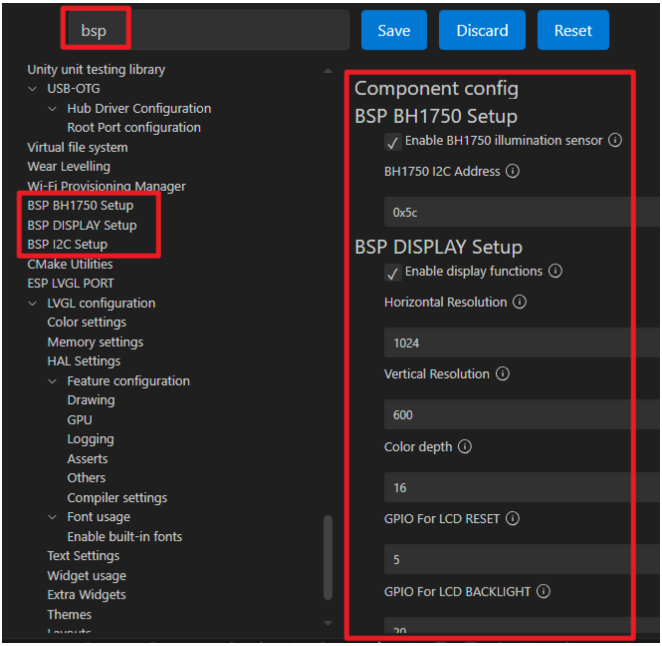
1.4 Programming procedure

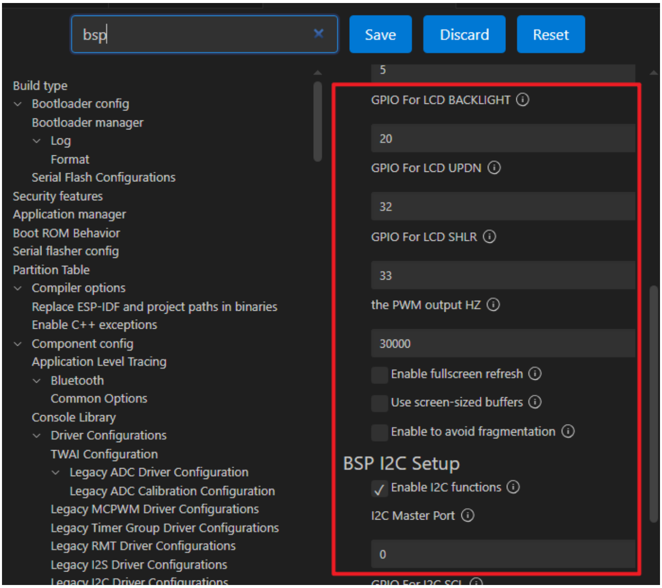
Connect the P4 device to the computer via USB

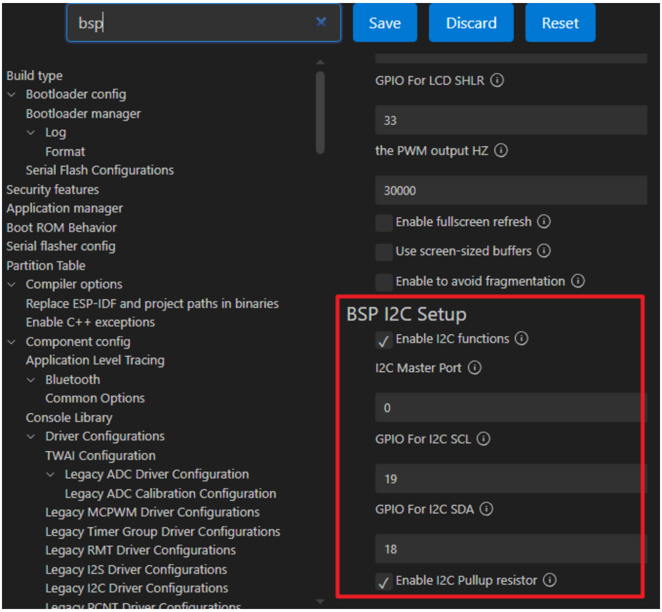


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the pins.







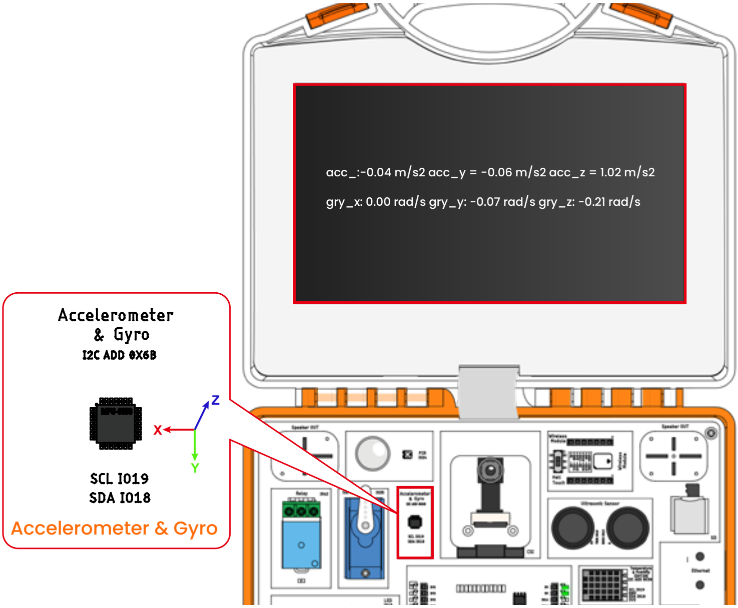
1.4.3 Click Compile. Once compilation is successful, click Download.

Lesson 13 - LSM6DS3 Gyroscope Display

Introduction

This chapter's tutorial demonstrates the interface application between the ESP32-P4 and the LSM6DS3TR gyroscope sensor. Through an attitude angular velocity measurement example, it aids in understanding the fundamental capabilities of six-axis inertial sensors. As a common motion detection component, the gyroscope enables readers to gain an intuitive grasp of the ESP32-P4's applications in motion control, attitude estimation, and wearable devices, laying the groundwork for more complex intelligent interaction projects.

Project Demonstration Effect



This chapter is divided into the following subsections

- 1.1 Introduction to the LSM6DS3TR Sensor
- 1.2 Hardware Design
- 1.3 Software Design
- 1.4 Download and Verification

1.1 LSM6DS3TR Sensor Introduction

1.1.1 LSM6DS3TR Introduction

The LSM6DS3TR is a six-axis inertial measurement unit (IMU) developed by STMicro-electronics, featuring integrated:

A triaxial accelerometer (±2g / ±4g / ±8g / ±16g selectable)

A triaxial gyroscope (±125 dps / ±250 dps / ±500 dps / ±1000 dps / ±2000 dps select-able)

It incorporates a digital signal processing unit (DSP) and communicates directly with the host via I²C or SPI interfaces.

Key features include:

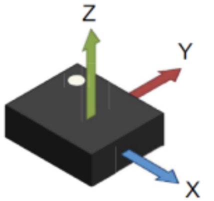
① High performance: Accelerometer noise density as low as 90 μg/√Hz; gyroscope noise density typically 4 mdps/√Hz;

② Low power consumption: Typical operating current just 0.9 mA (accelerometer + gyroscope);

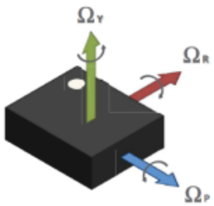
③ Embedded functionality: Supports FIFO buffering, gait detection, activity recognition, and free-fall detection;

④ Flexible communication: Supports I²C (100kHz/400kHz) and SPI (up to 10 MHz);

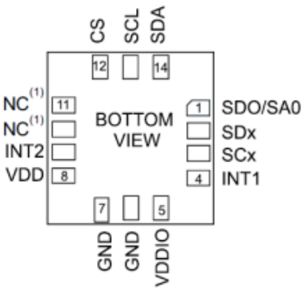
⑤ Wide applicability: Suitable for drones, mobile phones, fitness trackers, VR/AR headsets, robots, and more.



Direction of detectable acceleration (top view)



Direction of detectable angular rate (top view)



1.1.2 How Gyroscopes Work

A gyroscope is a device for measuring angular velocity, implemented internally in the LSM6DS3TR using MEMS micro-electro-mechanical systems. Its core principle operates as follows:

When the internal micro-mechanical structure vibrates, it experiences Coriolis force;

Upon angular velocity changes within the device, the vibrating structure undergoes minute displacement;

This displacement is converted into an electrical signal via capacitive sensing;

The internal ADC and DSP convert the signal into a digital output.

$$F_n = 2mv\omega$$

Where:

- F_n: Coriolis force
- m: Mass
- v: Vibration velocity
- ω: Angular velocity

Thus, the gyroscope can measure angular velocity values around the X, Y, and Z axes in real time (unit: dps, i.e. degrees per second)

1.1.3 Principle of Operation of an Accelerometer

The LSM6DS3TR also incorporates a triaxial accelerometer for measuring linear acceleration:

When the device experiences acceleration, its internal mass block shifts;

This displacement alters the capacitive structure;

The resulting signal is converted into a digital value by the ADC and output.

Using the triaxial accelerometer, an object's motion state and orientation (such as horizontal tilt angle) can be calculated.

1.1.4 Data Output and Conversion

The LSM6DS3TR registers store raw measurement values (16-bit two's complement binary).

Gyroscope Data Calculation Formula

ω (dps) = (Raw ÷ 32768) × FS_{gyro}

Where:

- Raw: Raw register value
- FS_{gyro}: Full-scale range, e.g., ±250 dps

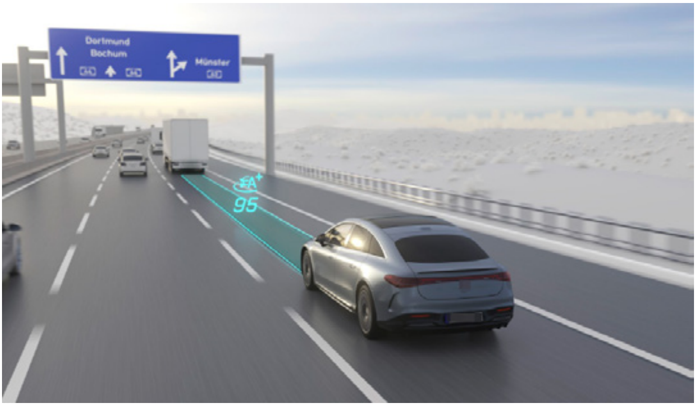
Accelerometer Data Calculation Formula

a (g) = (Raw ÷ 32768) × FS_{acc}

Where:

- FS_{acc}: Full-scale range, e.g., ±2 g

Gyroscopes find extensive application, being present in everyday items such as drones and automobiles.



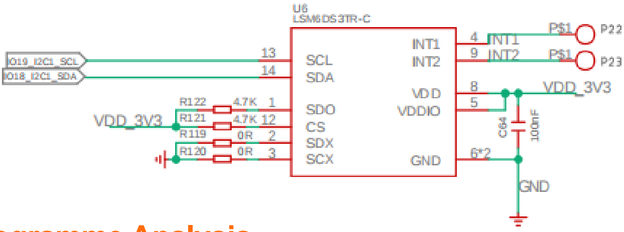
1.2 Hardware design

The typical wiring configuration for the ESP32-P4 and LSM6DS3TR is as follows (I²C mode):
ESP32-P4 SDA → LSM6DS3TR SDA
ESP32-P4 SCL → LSM6DS3TR SCL
3.3V → VCC
GND → GND
The default I²C address is 0x6B
It is recommended to connect a 4.7kΩ pull-up resistor to each of the SDA and SCL lines.

Accelerometer & Gyro

I2C ADD:0X6B

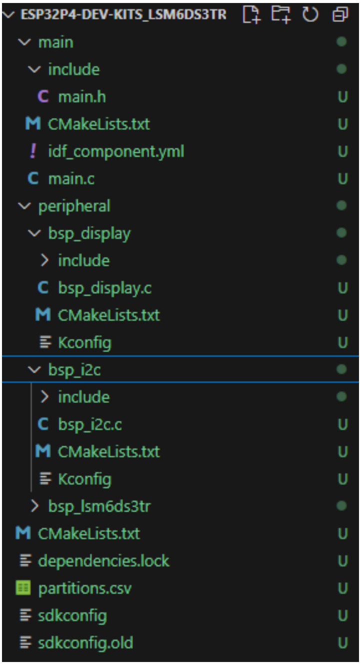
8



1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_lsm6ds3tr** example, a new folder named **bsp_lsm6ds3tr** has been created under the **ESP32P4-dev-kits_lsm6ds3tr\peripheral** directory. Within the **bsp_lsm6ds3tr** path, a new include folder, **CMakeLists.txt** file, and **Kconfig** file have been established. The **bsp_lsm6ds3tr** folder houses the **bsp_lsm6ds3tr.c** driver file. The include folder contains the **bsp_lsm6ds3tr.h** header file, while the **CMakeLists.txt** file integrates the driver into the build system, enabling project utilisation of its functionality. The **Kconfig** file loads the entire driver alongside GPIO pin definitions into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 LSM6DS3TR-C Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The LSM6DS3TR driver source code comprises two files: bsp_lsm6ds3tr.c and bsp_lsm6ds3tr.h.

We shall first examine the bsp_lsm6ds3tr.h file: it contains relevant definitions for the accelerometer and gyroscope sensor pins, along with function declarations.

/* Header file references */

```c
/*——————————————————————————————Header file declaration——————————————————————————————*/
#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 "bsp_i2c.h"
/*——————————————————————————————Header file declaration end——————————————————————————————*/
```

/* Function declarations and macro definition declarations */

```c
#ifdef CONFIG_BSP_LSM6DS3_SENSOR_ENABLED

#define LSM6DS3_I2C_ADDRESS CONFIG_LSM6DS3_I2C_ADDRESS // The 7-bit I2C address of LSM6DS3TR
#define WHO_AM_I_ADDRESS 0x0F                          // This register is a read-only register. Its value is fixed at 69h
#define CTRL1_XL_ADDRESS 0x10                          // Linear acceleration sensor control register 1
#define CTRL2_G_ADDRESS 0x11                           // Angular rate sensor control register 2
#define CTRL3_C_ADDRESS 0x12                           // Control register 3
#define CTRL4_C_ADDRESS 0x13                           // Control register 4
#define CTRL6_C_ADDRESS 0x15                           // Angular rate sensor control register 6
#define CTRL7_G_ADDRESS 0x16                           // Angular rate sensor control register 7
#define CTRL8_XL_ADDRESS 0x17                          // Linear acceleration sensor control register 8
#define STATUS_REG_ADDRESS 0x1E                        // Status register
#define OUTX_L_G_ADDRESS 0x22                          // Angular rate sensor pitch axis (X) angular rate output register(L_Bit)
#define OUTX_H_G_ADDRESS 0x23                          // Angular rate sensor pitch axis (X) angular rate output register(H_Bit)
#define OUTY_L_G_ADDRESS 0x24                          // Angular rate sensor roll axis (Y) angular rate output register(L_Bit)
#define OUTY_H_G_ADDRESS 0x25                          // Angular rate sensor roll axis (Y) angular rate output register(H_Bit)
#define OUTZ_L_G_ADDRESS 0x26                          // Angular rate sensor yaw axis (Z) angular rate output register(L_Bit)
#define OUTZ_H_G_ADDRESS 0x27                          // Angular rate sensor yaw axis (Z) angular rate output register(H_Bit)
#define OUTX_L_XL_ADDRESS 0x28                         // Linear acceleration sensor X-axis output register(L_Bit)
#define OUTX_H_XL_ADDRESS 0x29                         // Linear acceleration sensor X-axis output register(H_Bit)
#define OUTY_L_XL_ADDRESS 0x2A                         // Linear acceleration sensor Y-axis output register(L_Bit)
#define OUTY_H_XL_ADDRESS 0x2B                         // Linear acceleration sensor Y-axis output register(H_Bit)
#define OUTZ_L_XL_ADDRESS 0x2C                         // Linear acceleration sensor Z-axis output register(L_Bit)
#define OUTZ_H_XL_ADDRESS 0x2D                         // Linear acceleration sensor Z-axis output register(H_Bit)
```

```c
/*Output data rate and power mode selection*/
#define LSM6DS3TRC_RATE_0 0x00
#define LSM6DS3TRC_RATE_1HZ6 0xB0
#define LSM6DS3TRC_RATE_12HZ5 0x10
#define LSM6DS3TRC_RATE_26HZ 0x20
#define LSM6DS3TRC_RATE_52HZ 0x30
#define LSM6DS3TRC_RATE_104HZ 0x40
#define LSM6DS3TRC_RATE_208HZ 0x50
#define LSM6DS3TRC_RATE_416HZ 0x60
#define LSM6DS3TRC_RATE_833HZ 0x70
#define LSM6DS3TRC_RATE_1660HZ 0x80
#define LSM6DS3TRC_RATE_3330HZ 0x90
#define LSM6DS3TRC_RATE_6660HZ 0xA0
/*Accelerometer full-scale selection*/
#define LSM6DS3TRC_ACC_FSXL_2G 0x00
#define LSM6DS3TRC_ACC_FSXL_16G 0x04
#define LSM6DS3TRC_ACC_FSXL_4G 0x08
#define LSM6DS3TRC_ACC_FSXL_8G 0x0C
/*Gyroscope full-scale selection*/
#define LSM6DS3TRC_GYR_FSG_245 0x00
#define LSM6DS3TRC_GYR_FSG_500 0x04
#define LSM6DS3TRC_GYR_FSG_1000 0x08
#define LSM6DS3TRC_GYR_FSG_2000 0x0C
/*Anti-aliasing filter bandwidth selection*/
#define LSM6DS3TRC_ACC_BW0XL_1500HZ 0x00
#define LSM6DS3TRC_ACC_BW0XL_400HZ 0x01
/*Accelerometer bandwidth selection*/
#define LSM6DS3TRC_CTRL4_DEN_XL_EN_DISABLE 0x00
#define LSM6DS3TRC_CTRL4_DEN_XL_EN_ENABLE 0x80
/*Gyroscope sleep mode enable*/
#define LSM6DS3TRC_CTRL4_SLEEP_ENABLE 0x40
#define LSM6DS3TRC_CTRL4_SLEEP_DISABLE 0x00
/*All interrupt signals available on INT1 pad enable*/
#define LSM6DS3TRC_CTRL4_DEN_DRDY_INT1_DISBALE 0x00
#define LSM6DS3TRC_CTRL4_DEN_DRDY_INT1_ENABLE 0x20
/*Data-ready mask enable*/
#define LSM6DS3TRC_CTRL4_DRDY_MASK_DISABLE 0x00
#define LSM6DS3TRC_CTRL4_DRDY_MASK_ENABLE 0x08
```

```c
/*Disable I2C interface*/
#define LSM6DS3TRC_CTRL4_I2C_DISABLE 0x04
#define LSM6DS3TRC_CTRL4_I2C_ENABLE 0x00
/* Enable gyroscope digital LPF1. The bandwidth can be selected through */
#define LSM6DS3TRC_CTRL4_LPF1_SELG_ENABLE 0x02
#define LSM6DS3TRC_CTRL4_LPF1_SELG_DISABLE 0x00

#define LSM6DS3TRC_CTRL6_C_EDGE_TRIGGER 0x80 // Gyroscope data edge-sensitive trigger enable

#define LSM6DS3TRC_CTRL6_C_LEVEL_TRIGGER 0x40 // Gyroscope data level-sensitive trigger enable
#define LSM6DS3TRC_CTRL6_C_LEVEL_LATCHED 0x60 // Gyroscope level-sensitive latched enable.
#define LSM6DS3TRC_CTRL6_C_LEVEL_FIFO 0xC0    // LSM6DS3TRC_CTRL6_C_EDGE_TRIGGER|LSM6DS3TRC_CTRL6_C_LEVEL_TRIGGER
/* High-performance operating mode disable for accelerometer*/
#define LSM6DS3TRC_CTRL6_C_XL_HM_MODE_ENABLE 0x00
#define LSM6DS3TRC_CTRL6_C_XL_HM_MODE_DISABLE 0x10
/* Gyroscope's low-pass filter (LPF1) bandwidth selection */
#define LSM6DS3TRC_CTRL6_C_FTYPE_1 0x00
#define LSM6DS3TRC_CTRL6_C_FTYPE_2 0x01
#define LSM6DS3TRC_CTRL6_C_FTYPE_3 0x02
#define LSM6DS3TRC_CTRL6_C_FTYPE_4 0x03
/*High-performance operating mode*/
#define LSM6DS3TRC_CTRL7_G_HM_MODE_ENABLE 0x00
#define LSM6DS3TRC_CTRL7_G_HM_MODE_DISABLE 0x80
/*Gyroscope digital high-pass filter enable*/
#define LSM6DS3TRC_CTRL7_G_HP_EN_DISABLE 0x00
#define LSM6DS3TRC_CTRL7_G_HP_EN_ENABLE 0x40
/*Gyroscope digital HP filter cutoff selection*/
#define LSM6DS3TRC_CTRL7_G_HPM_16MHZ 0x00
#define LSM6DS3TRC_CTRL7_G_HPM_65MHZ 0x10
#define LSM6DS3TRC_CTRL7_G_HPM_260MHZ 0x20
#define LSM6DS3TRC_CTRL7_G_HPM_1HZ04 0x30
/*Source register rounding function enable*/
#define LSM6DS3TRC_CTRL7_G_ROUNDING_STATUS_DISABLE 0x04
#define LSM6DS3TRC_CTRL7_G_ROUNDING_STATUS_ENABLE 0x00
```

```c
/*Accelerometer ODR register setting*/
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_50 0x88
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_100 0xA8
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_9 0xC8
#define LSM6DS3TRC_ACC_LOW_PASS_ODR_400 0xE8

#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_50 0x04
#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_100 0x24
#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_9 0x44
#define LSM6DS3TRC_ACC_HIGH_PASS_ODR_400 0x64

#define LSM6DS3TRC_STATUS_GYROSCOPE 0x02     // Gyroscope new data available
#define LSM6DS3TRC_STATUS_ACCELEROMETER 0x01 // Accelerometer new data available

typedef struct
{
    float acc_x;
    float acc_y;
    float acc_z;
    float gry_x;
    float gry_y;
    float gry_z;
} lsm6ds3tr; // lsm6ds3tr data structure

extern lsm6ds3tr my_lsm6ds3;
esp_err_t lsm6ds3_begin(void);            // Initialization of Lsm6ds3tr sensor
esp_err_t lsm6ds3_get_acc(uint8_t fsxl); // lsm6ds3 acquires acceleration data
esp_err_t lsm6ds3_get_gry(uint8_t fsg);  // lsm6ds3 acquires gyroscope data
esp_err_t lsm6ds3_scan();                 // The lsm6ds3 scans to determine whether the currently read data is from the gyroscope or the accelerometer
#endif
```

Here, numerous macros define the various register addresses and operation commands for the sensor.

Next, we shall analyse the program in bsp_lsm6ds3tr.c: initialising the sensor configuration and exposing the API interface functions.

/* LSM6DS3TR initialisation function lsm6ds3_begin */

```
esp_err_t lsm6ds3_begin(void)
{
    esp_err_t err = ESP_OK;
    lsm6ds3_handle = i2c_dev_register(LSM6DS3_I2C_ADDRESS); /*lsm6ds3 sensor address registration*/
    if (lsm6ds3_handle != NULL)
    {
        err = lsm6ds3_getchipID(); /*lsm6ds3 obtains ID information*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_reset(); /*Reset the sensor of lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_BDU(true); /*Set BDU for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_acc_rate(LSM6DS3TRC_RATE_12HZ5); /*Set the rate of the accelerometer for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_gyr_rate(LSM6DS3TRC_RATE_12HZ5); /*Set the rate of the gyroscope for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_acc_fullscale(LSM6DS3TRC_ACC_FSXL_2G); /*Set the full-scale of the accelerometer for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_gyr_fullscale(LSM6DS3TRC_GYR_FSG_2000); /*Set the full-scale of the gyroscope for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_acc_bandwidth(LSM6DS3TRC_ACC_BW0XL_400HZ, LSM6DS3TRC_ACC_LOW_PASS_ODR_100); /*Set the bandwidth of the accelerometer for lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_gry_register4(LSM6DS3TRC_CTRL4_LPF1_SELG_ENABLE); /*Set the gyroscope register 4 of lsm6ds3*/
        if (err != ESP_OK)
            return err;
```

```
        err = lsm6ds3_set_gry_register6(LSM6DS3TRC_CTRL6_C_FTYPE_1); /*Set the gyroscope register 6 of lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_set_gry_register7(LSM6DS3TRC_CTRL7_G_HP_EN_ENABLE | LSM6DS3TRC_CTRL7_G_HPM_260MHZ); /*Set the gyroscope register 7 of lsm6ds3*/
        if (err != ESP_OK)
            return err;
        err = lsm6ds3_getchipID(); /*lsm6ds3 obtains ID information*/
        if (err != ESP_OK)
            return err;
    }
    else
    {
        err = ESP_FAIL;
        return err;
    }
    return err;
}
```

Within the **lsm6ds3_begin** function, the lsm6ds3tr sensor is first registered on the I2C bus using the **i2c_dev_register** function, which returns an operation handle. Subsequently, if the returned device handle is non-null, the various register configurations for initialising the **lsm6ds3tr** sensor are invoked. The specific functionality of each function will be analysed in detail below.

/* Function to read the device ID of the LSM6DS3TR: **lsm6ds3_getchipID** */

```
static esp_err_t lsm6ds3_getchipID(void)
{
    esp_err_t err = ESP_OK;
    uint8_t buf = 0;
    err = i2c_read_reg(lsm6ds3_handle, WHO_AM_I_ADDRESS, &buf, 1); /*I2C Read read-only register*/
    if (err != ESP_OK)
        return err;
    if (buf != 0x6A) /*Check if the id is 0x6A*/
    {
        err = ESP_FAIL;
        return err;
    }
    return err;
}
```

First, use the `**i2c_read_reg**` function to read the ID register address and retrieve the ID data. Determine whether it is 0x6A (the default value for this register must be 0x6A). If so, return `**ESP_OK**`.

/* lsm6ds3_reset function to reset the LSM6DS3TR device */

```
esp_err_t lsm6ds3_reset(void)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    buf[0] = 0x80;
    err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*Reboot memory content*/
    if (err != ESP_OK)
        return err;
    vTaskDelay(15 / portTICK_PERIOD_MS);
    err = i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;
    buf[0] |= 0x01;
    err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*Software reset*/
    if (err != ESP_OK)
        return err;
    while (buf[0] & 0x01) /*(0: normal mode; 1: reset device)*/
    {
        i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
        vTaskDelay(15 / portTICK_PERIOD_MS);
    }
    return err;
}
```

First, employ the **`i2c_write_reg`** function to write 0x80 to the controller's register address 3. This step resets the memory contents. After a 15ms delay, read the data from register address 3; the result should be 0x01, indicating a low bit 1. Subsequently, use the **`i2c_write_reg`** function to write this value, thereby enabling the software reset function. Finally, establish a loop to determine whether the software reset function has completed (the low bit is reset to 0).

/* **lsm6ds3_set_BDU** function for configuring BDU on **lsm6ds3tr** */

```
esp_err_t lsm6ds3_set_BDU(bool state)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;
    if (state == true)
    {
        buf[0] |= 0x40;
        err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*1: output registers not updated until MSB and LSB have been read*/
        if (err != ESP_OK)
            return err;
    }
    else
    {
        buf[0] &= 0xBF;
        err = i2c_write_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf[0]); /*0: continuous update*/
        if (err != ESP_OK)
            return err;
    }

    err = i2c_read_reg(lsm6ds3_handle, CTRL3_C_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;

    return err;
}
```

First, use the **`i2c_read_reg`** function to read the controller's register address 3. This step preserves the original settings, altering only the BDU parameter configuration. Based on the input Boolean parameter, if true, employ the **`i2c_write_reg`** function to set BDU control bit 1. This prevents the output register from updating before reading the most significant bit and least significant bit. If the parameter is false, the `i2c_write_reg` function is employed to set the BDU control bit to 0, ensuring the output register continues to update.

/* Function `lsm6ds3_set_acc_rate` for configuring the accelerometer output rate on the LSM6DS3TR */

```
esp_err_t lsm6ds3_set_acc_rate(uint8_t rate)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf, 1); /*I2C Read register status*/
    if (err != ESP_OK)
        return err;
    buf[0] |= rate;
    err = i2c_write_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf[0]); /*Accelerometer Output data rate*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

First, use the `i2c_read_reg` function to read the register address of Controller 1. This step preserves the original settings, altering only the parameter configuration for the accelerometer's output rate. Based on the input parameters,

the configured data is written to the Controller 1 register address via the `i2c_write_reg` function.

/* Function lsm6ds3_set_acc_fullscale for configuring the accelerometer's maximum range on the LSM6DS3TR */

```
esp_err_t lsm6ds3_set_acc_fullscale(uint8_t value)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= value;
    err = i2c_write_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf[0]); /*Accelerometer full-scale selection*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

First, use the `i2c_read_reg` function to read the register address of Controller 1. This step preserves the original settings, altering only the configuration parameter for the accelerometer's maximum range. Based on the input parameters,

the specified data is written to the register address of Controller 1 via the `i2c_write_reg` function.

/* Function lsm6ds3_set_gyr_rate for configuring gyroscope output rate on LSM6DS3TR */

```
esp_err_t lsm6ds3_set_gyr_rate(uint8_t rate)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= rate;
    err = i2c_write_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf[0]); /*Gyroscope Output data rate*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

First, use the `i2c_read_reg` function to read the controller 2 register address. This step preserves the original settings, altering only the gyroscope output rate configuration. Based on the input parameters, the specified data is written to the controller 2 register address via the `i2c_write_reg` function.

/* Function lsm6ds3_set_gyr_fullscale for setting the gyroscope full-scale range on the LSM6DS3TR */

```
esp_err_t lsm6ds3_set_gyr_fullscale(uint8_t value)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= value;
    err = i2c_write_reg(lsm6ds3_handle, CTRL2_G_ADDRESS, buf[0]); /*Gyroscope full-scale selection*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

First, use the `i2c_read_reg` function to read the controller 2 register address. This step preserves the original settings, altering only the gyroscope maximum range parameter configuration. Based on the input parameters,

the configured data is written to the controller 2 register address via the `i2c_write_reg` function.

/* Function lsm6ds3_set_acc_bandwidth for configuring the accelerometer bandwidth on the LSM6DS3TR */

```
esp_err_t lsm6ds3_set_acc_bandwidth(uint8_t BW0XL, uint8_t ODR)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= BW0XL;
    err = i2c_write_reg(lsm6ds3_handle, CTRL1_XL_ADDRESS, buf[0]); /*Anti-aliasing filter bandwidth selection*/
    if (err != ESP_OK)
        return err;

    err = i2c_read_reg(lsm6ds3_handle, CTRL8_XL_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= ODR;
    err = i2c_write_reg(lsm6ds3_handle, CTRL8_XL_ADDRESS, buf[0]); /*Accelerometer ODR register setting*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Firstly, this function has two parameters which jointly determine the accelerometer's bandwidth settings. These are located in

Controller 1 register and Controller 8 register respectively.

/* lsm6ds3_set_gry_register4 function for setting gyroscope controller 4 register on lsm6ds3tr */

```
esp_err_t lsm6ds3_set_gry_register4(uint8_t cmd)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};

    err = i2c_read_reg(lsm6ds3_handle, CTRL4_C_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= cmd;
    err = i2c_write_reg(lsm6ds3_handle, CTRL4_C_ADDRESS, buf[0]); /*Enable gyroscope digital LPF1 (0: disabled; 1: enabled)*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

This function enables or disables the filter by setting the LPF1 filter parameter in controller register 4 (0: disabled, 1: enabled).

/* lsm6ds3_set_gry_register6: Function to configure gyroscope controller register 6 for the LSM6DS3TR */

```c
esp_err_t lsm6ds3_set_gry_register6(uint8_t cmd)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL6_C_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= cmd;
    err = i2c_write_reg(lsm6ds3_handle, CTRL6_C_ADDRESS, buf[0]); /*this bit must be set to '0' for the correct operation of the device.*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

This function sets controller register 6, which must be configured to 0 according to the data sheet requirements.

/* lsm6ds3_set_gry_register7: Function to configure gyroscope controller register 7 for the LSM6DS3TR */

```c
esp_err_t lsm6ds3_set_gry_register7(uint8_t cmd)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[1] = {0};
    err = i2c_read_reg(lsm6ds3_handle, CTRL7_G_ADDRESS, buf, 1);
    if (err != ESP_OK)
        return err;
    buf[0] |= cmd;
    err = i2c_write_reg(lsm6ds3_handle, CTRL7_G_ADDRESS, buf[0]); /*Gyroscope digital high-pass filter enable and Gyroscope digital HP filter cutoff selection 260MHZ*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

This function enables or disables the gyroscope high-pass filter and selects its cutoff frequency by configuring parameters in controller register 7.

/* lsm6ds3_get_status function retrieves status register from lsm6ds3tr */

```c
static uint8_t lsm6ds3_get_status(void)
{
    uint8_t buf[1] = {0};
    i2c_read_reg(lsm6ds3_handle, STATUS_REG_ADDRESS, buf, 1); /*I2C Read register status*/
    return buf[0];
}
```

By calling the i2c_read_reg function to read the value of the sensor status register and return it (determining whether the current state is for reading accelerometer data updates or gyroscope data updates)

/* lsm6ds3_data_read function for reading the LSM6DS3TR output register */

```c
/*I2C reads the registers, performing a cyclic reading process*/
static esp_err_t lsm6ds3_data_read(uint8_t reg_addr, uint8_t *rev_data, uint8_t length)
{
    esp_err_t err = ESP_OK;
    while (length) /*Read the length*/
    {
        err = i2c_read_reg(lsm6ds3_handle, reg_addr++, rev_data++, 1); /*I2C Read register data*/
        if (err != ESP_OK)
            return err;
        length--;
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    return err;
}
```

By inputting parameters as conditions for loop execution. Within the loop, the i2c_read_reg function is invoked to read values from the sensor's output register (after each read, increment the register address and increment the receive buffer address to achieve cyclic reading; subsequently decrement the read length data to determine when the loop should terminate).

/* Function lsm6ds3_get_acc for reading accelerometer parameters from lsm6ds3tr */

```c
esp_err_t lsm6ds3_get_acc(uint8_t fsxl)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[6];
    int16_t acc[3];
    err = lsm6ds3_data_read(OUTX_L_XL_ADDRESS, buf, 6); /*Read the value stored in the accelerometer register*/
    if (err != ESP_OK)
        return err;
    else
    {
        acc[0] = buf[1] << 8 | buf[0];
        acc[1] = buf[3] << 8 | buf[2];
        acc[2] = buf[5] << 8 | buf[4];
        switch (fsxl) /*Based on setting different full-scale parameters, convert the parameters returned by the accelerometer*/
        {
        case LSM6DS3TRC_ACC_FSXL_2G:
            my_lsm6ds3.acc_x = ((float)acc[0] * 0.061f) / 1000.0f;
            my_lsm6ds3.acc_y = ((float)acc[1] * 0.061f) / 1000.0f;
            my_lsm6ds3.acc_z = ((float)acc[2] * 0.061f) / 1000.0f;
            break;
        case LSM6DS3TRC_ACC_FSXL_4G:
            my_lsm6ds3.acc_x = ((float)acc[0] * 0.122f) / 1000.0f;
            my_lsm6ds3.acc_y = ((float)acc[1] * 0.122f) / 1000.0f;
            my_lsm6ds3.acc_z = ((float)acc[2] * 0.122f) / 1000.0f;
            break;
        case LSM6DS3TRC_ACC_FSXL_8G:
            my_lsm6ds3.acc_x = ((float)acc[0] * 0.244f) / 1000.0f;
            my_lsm6ds3.acc_y = ((float)acc[1] * 0.244f) / 1000.0f;
            my_lsm6ds3.acc_z = ((float)acc[2] * 0.244f) / 1000.0f;
            break;
        case LSM6DS3TRC_ACC_FSXL_16G:
            my_lsm6ds3.acc_x = ((float)acc[0] * 0.488f) / 1000.0f;
            my_lsm6ds3.acc_y = ((float)acc[1] * 0.488f) / 1000.0f;
            my_lsm6ds3.acc_z = ((float)acc[2] * 0.488f) / 1000.0f;
            break;
        }
    }
    return err;
}
```

The lsm6ds3_data_read function retrieves accelerometer data. The input parameters determine which conversion method should be applied to the read data (based on the set maximum range parameter). The specific conversion formula can be found in the sensor data manual.

/* lsm6ds3_get_gry function for reading gyroscope parameters from lsm6ds3tr */

```c
esp_err_t lsm6ds3_get_gry(uint8_t fsg)
{
    esp_err_t err = ESP_OK;
    uint8_t buf[6];
    int16_t gry[3];
    err = lsm6ds3_data_read(OUTX_L_G_ADDRESS, buf, 6); /*Read the values stored in the gyroscope register*/
    if (err != ESP_OK)
        return err;
    else
    {
        gry[0] = buf[1] << 8 | buf[0];
        gry[1] = buf[3] << 8 | buf[2];
        gry[2] = buf[5] << 8 | buf[4];
        switch (fsg) /*Based on setting different full-scale parameters, convert the parameters returned by the gyroscope*/
        {
        case LSM6DS3TRC_GYR_FSG_245:
            my_lsm6ds3.gry_x = ((float)gry[0] * 8.750f) / 1000.0f;
            my_lsm6ds3.gry_y = ((float)gry[1] * 8.750f) / 1000.0f;
            my_lsm6ds3.gry_z = ((float)gry[2] * 8.750f) / 1000.0f;
            break;
        case LSM6DS3TRC_GYR_FSG_500:
            my_lsm6ds3.gry_x = ((float)gry[0] * 17.50f) / 1000.0f;
            my_lsm6ds3.gry_y = ((float)gry[1] * 17.50f) / 1000.0f;
            my_lsm6ds3.gry_z = ((float)gry[2] * 17.50f) / 1000.0f;
            break;
        case LSM6DS3TRC_GYR_FSG_1000:
            my_lsm6ds3.gry_x = ((float)gry[0] * 35.00f) / 1000.0f;
            my_lsm6ds3.gry_y = ((float)gry[1] * 35.00f) / 1000.0f;
            my_lsm6ds3.gry_z = ((float)gry[2] * 35.00f) / 1000.0f;
            break;
        case LSM6DS3TRC_GYR_FSG_2000:
            my_lsm6ds3.gry_x = ((float)gry[0] * 70.00f) / 1000.0f;
            my_lsm6ds3.gry_y = ((float)gry[1] * 70.00f) / 1000.0f;
            my_lsm6ds3.gry_z = ((float)gry[2] * 70.00f) / 1000.0f;
            break;
        }
    }
    return err;
}
```

The gyroscope data is read via the lsm6ds3_data_read function. The input parameters determine which conversion method should be applied to the read data (based on the set maximum range parameter). The specific conversion formula can be found in the sensor data manual.

/* lsm6ds3_scan function for scanning and reading the lsm6ds3tr */

```c
esp_err_t lsm6ds3_scan()
{
    esp_err_t err = ESP_FAIL;
    uint8_t status;
    status = lsm6ds3_get_status();                  /*I2C Read register status*/
    if (status & LSM6DS3TRC_STATUS_ACCELEROMETER) /*Accelerometer new data available status*/
    {
        err = lsm6ds3_get_acc(LSM6DS3TRC_ACC_FSXL_2G);
        if (err != ESP_OK)
            return err;
    }

    if (status & LSM6DS3TRC_STATUS_GYROSCOPE) /*Gyroscope new data available status*/
    {
        err = lsm6ds3_get_gry(LSM6DS3TRC_GYR_FSG_2000);
        if (err != ESP_OK)
            return err;
    }

    return err;
}
```

Retrieve the current sensor update status via the lsm6ds3_get_status function, and determine whether to read the accelerometer or gyroscope data based on the status.

1.3.2 Kconfig file

The primary function of this file is to incorporate the requisite configuration into the sdkconfig file, enabling certain parameter adjustments to be made via a graphical interface. Here, 0x6B denotes the 7-bit address for the LSM6DS3TR-C sensor.

```
menu "BSP LSM6DS3 Setup"
    config BSP_LSM6DS3_SENSOR_ENABLED
        bool "Enable LSM6DS3TR sensor"
        depends on BSP_I2C_ENABLED

    if BSP_LSM6DS3_SENSOR_ENABLED
        config LSM6DS3_I2C_ADDRESS
            hex "LSM6DS3TR I2C Address"
            default 0x6B
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the bsp_lsm6ds3tr driver. To successfully call the contents of the bsp_lsm6ds3tr folder from other functions, it is necessary to configure the CMakeLists.txt file within the bsp_lsm6ds3tr folder. The configuration details are as follows:

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

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

Within this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver library (bsp_i2c). Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the bsp_lsm6ds3tr driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the main.h header file within the include folder. Add the main folder to the CMakeLists.txt file within the build system.

The main.h file primarily references required header files: functions utilising the bsp_display driver necessitate inclusion of the bsp_display header file, while those employing the bsp_lsm6ds3tr driver require the bsp_lsm6ds3tr header file.

Below is an analysis of the main.c programme: system initialisation and execution of functions for I2C, the lsm6ds3tr-c sensor, and display functionality.

```
#ifdef CONFIG_BSP_I2C_ENABLED
    err = i2c_init(); /*I2C Initialization*/
    if (err != ESP_OK)
        init_fail("i2c", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
#ifdef CONFIG_BSP_LSM6DS3_SENSOR_ENABLED
    err = lsm6ds3_begin(); /*Lsm6ds3tr Initialization*/
    if (err != ESP_OK)
        init_fail("lsm6ds3tr", err);
#endif
#endif
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
```

This code resides within the init function, which serves to store initialisation functions requiring invocation and assess the outcome of such initialisation. Should the return status deviate from ESP_OK, the code shall output an error message and cease further execution.

/* Screen initialisation and display function lsm6ds3_display */

```
void lsm6ds3_display()
{
    if (lvgl_port_lock(0))
    {
        lsm6ds3_acc_data = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t acc_label_style;
        lv_style_init(&acc_label_style);                                                   /*Initialize a style*/
        lv_style_set_bg_opa(&acc_label_style, LV_OPA_TRANSP);                              /*Set the style LVGL background color*/
        lv_obj_add_style(lsm6ds3_acc_data, &acc_label_style, LV_PART_MAIN);                /*Add a style to an object*/
        lv_obj_set_style_text_color(lsm6ds3_acc_data, LV_COLOR_WHITE, LV_PART_MAIN);       /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(lsm6ds3_acc_data, &lv_font_montserrat_30, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_align(lsm6ds3_acc_data, LV_ALIGN_CENTER, 0, -30);                          /*Change the alignment of an object*/
        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(lsm6ds3_acc_data, "acc_x: 0.00 m/s2 acc_y = 0.00 m/s2 acc_z = 0.00 m/s2"); /*Set a new text for a label*/

        lsm6ds3_gry_data = lv_label_create(lv_scr_act()); /*Create a label object*/
        static lv_style_t gry_label_style;
        lv_style_init(&gry_label_style);                                                     /*Initialize a style*/
        lv_style_set_bg_opa(&gry_label_style, LV_OPA_TRANSP);                                /*Set the style LVGL background color*/
        lv_obj_add_style(lsm6ds3_gry_data, &gry_label_style, LV_PART_MAIN);                  /*Add a style to an object*/
        lv_obj_set_style_text_color(lsm6ds3_gry_data, LV_COLOR_WHITE, LV_PART_MAIN);         /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(lsm6ds3_gry_data, &lv_font_montserrat_30, LV_PART_MAIN);  /*Set the style LVGL text font*/
        lv_obj_align(lsm6ds3_gry_data, LV_ALIGN_CENTER, 0, 30);                             /*Change the alignment of an object*/
        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(lsm6ds3_gry_data, "gry_x: 0.00 rad/s gry_y - 0.00 rad/s gry_z - 0.00 rad/s"); /*Set a new text for a label*/

        lvgl_port_unlock();
    }
}
```

This function primarily configures the initial screen display content: it sets the background colour and text display via the lvgl control.

The **lv_label_set_text** function sets the text displayed on the control.

The **lv_style_set_bg_opa** function sets the background colour of the style.

The **lv_obj_set_style_text_color** function sets the text display colour.

The **lv_obj_set_style_text_font** function sets the text font size.

The **lv_obj_set_style_bg_color** function sets the background colour.

The **lv_obj_set_style_bg_opa** function sets the background transparency.

lv_obj_align function sets the control's alignment and offset

Note: When calling lvgl functions outside lvgl thread functions, a mutex lock must be acquired. **lvgl_port_lock** function acquires the mutex lock, **lvgl_port_unlock** function releases it.

/* Screen data refresh display function update_lsm6ds3_value */

```c
void update_lsm6ds3_value(lsm6ds3tr data)
{
    if ((lsm6ds3_acc_data) && (lsm6ds3_gry_data))
    {
        char buffer_acc[80];
        char buffer_gry[80];
        snprintf(buffer_acc, sizeof(buffer_acc), "acc_x: %.2f m/s2 acc_y = %.2f m/s2 acc_z = %.2f m/s2", data.acc_x, data.acc_y, data.acc_z);    /*Format the data into a string*/
        snprintf(buffer_gry, sizeof(buffer_gry), "gry_x: %.2f rad/s gry_y = %.2f rad/s gry_z = %.2f rad/s", data.gry_x, data.gry_y, data.gry_z); /*Format the data into a string*/
        lv_label_set_text(lsm6ds3_acc_data, buffer_acc);                                                                                        /*Set a new text for a label*/
        lv_label_set_text(lsm6ds3_gry_data, buffer_gry);                                                                                        /*Set a new text for a label*/
    }
}
```

This function employs the snprintf function to format the acquired float-type data into a string, subsequently refreshing the displayed content via the lv_label_set_text function.

It is worth noting that the snprintf function appends a terminating character to the end of the formatted string. As lv_label_set_text recognises strings by identifying this terminating character, utilising snprintf for string formatting constitutes a preferable approach.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_LSM6DS3_SENSOR_ENABLED
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100);                                                                          /*Set the screen backlight to maximum brightness*/
    lsm6ds3_display();                                                                            /*Set the screen's LVGL default display page*/
    xTaskCreate(lsm6ds3_read_task, "read_lsm6ds3", 4096, NULL, configMAX_PRIORITIES - 5, &read_lsm6ds3); /*Create a lsm6ds3tr data display refresh thread*/
#endif
#endif
}
```

Within the app_main function, the backlight brightness is first set to 100%, followed by initialising the screen display content. Finally, a FreeRTOS thread is created to handle data processing and screen refresh operations.

/* lsm6ds3_read_task: Sensor data processing and screen refresh thread for the LSM6DS3TR-C sensor */

```c
void lsm6ds3_read_task(void *param)
{
    while (1)
    {
        if (lsm6ds3_scan() != ESP_OK) /*The lsm6ds3 scans to determine whether the currently read data is from the gyroscope or the accelerometer*/
        {
            if (lvgl_port_lock(0))
            {
                lv_label_set_text(lsm6ds3_acc_data, "lsm6ds3 read acc data error"); /*Read failure message displayed*/
                lv_label_set_text(lsm6ds3_gry_data, "lsm6ds3 read gry data error"); /*Read failure message displayed*/
                lvgl_port_unlock();
            }
            MAIN_ERROR("lsm6ds3 scan false");
        }
        else
        {
            if (lvgl_port_lock(0))
            {
                update_lsm6ds3_value(my_lsm6ds3); /*Update the lsm6ds3 data displayed on the screen*/
                lvgl_port_unlock();
            }
        }
        MAIN_INFO("acc_x: %.2f m/s2", my_lsm6ds3.acc_x);
        MAIN_INFO("acc_y: %.2f m/s2", my_lsm6ds3.acc_y);
        MAIN_INFO("acc_z: %.2f m/s2", my_lsm6ds3.acc_z);
        MAIN_INFO("gry_x: %.2f rad/s", my_lsm6ds3.gry_x);
        MAIN_INFO("gry_y: %.2f rad/s", my_lsm6ds3.gry_y);
        MAIN_INFO("gry_z: %.2f rad/s", my_lsm6ds3.gry_z);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Within the LSM6DS3TR-C sensor data processing and screen display refresh thread, a while loop is first established. Within this loop, the lsm6ds3_scan function is invoked to update and acquire sensor data. Upon successful acquisition, the update_lsm6ds3_value function refreshes the screen display data. Should acquisition fail, the screen displays the error messages ‘LSM6DS3 read acc data error’ and ‘LSM6DS3 read gry data error’. The final 1-second delay ensures data is refreshed and retrieved once every second.

1.3.5 CMkaLists.txt file

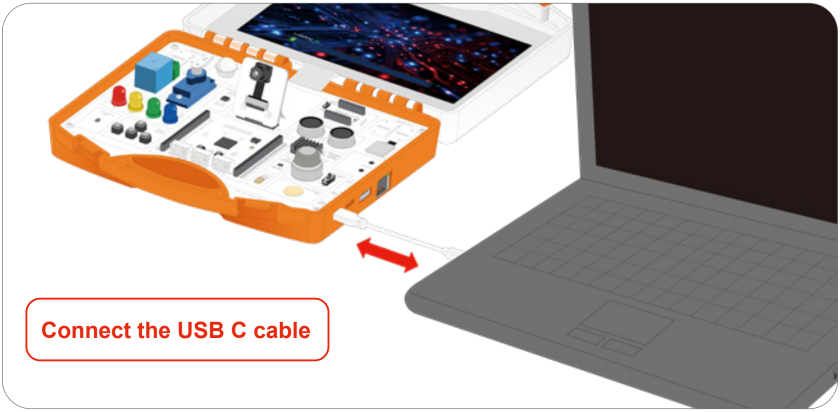
To successfully call the contents of the bsp_display and bsp_lsm6ds3tr folders within the main function, it is necessary to configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_i2c bsp_display bsp_lsm6ds3tr)
```

First, the directories for source files and header files are defined, along with the required driver libraries: bsp_display, bsp_lsm6ds3tr, and bsp_i2c. These settings are then registered with the build system via the idf_component_register command, enabling the main function to utilise these driver functionalities.

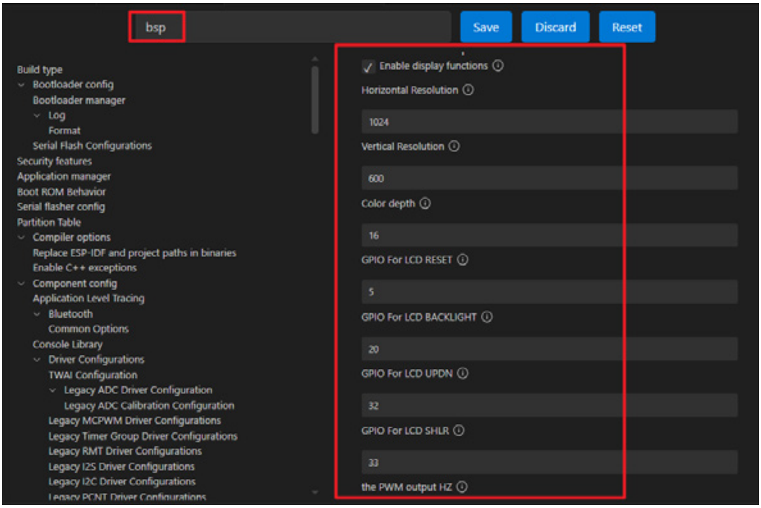
1.4 Programming procedure

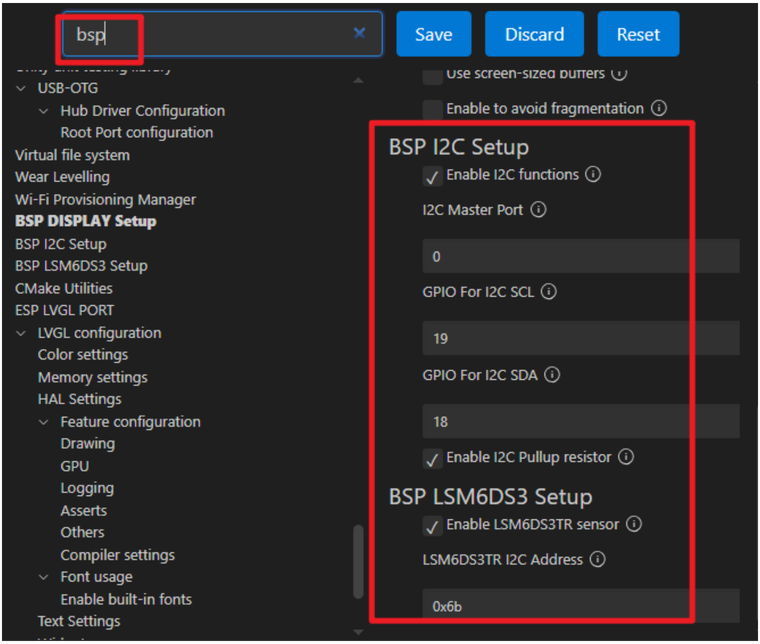
Connect the P4 device to the computer via USB



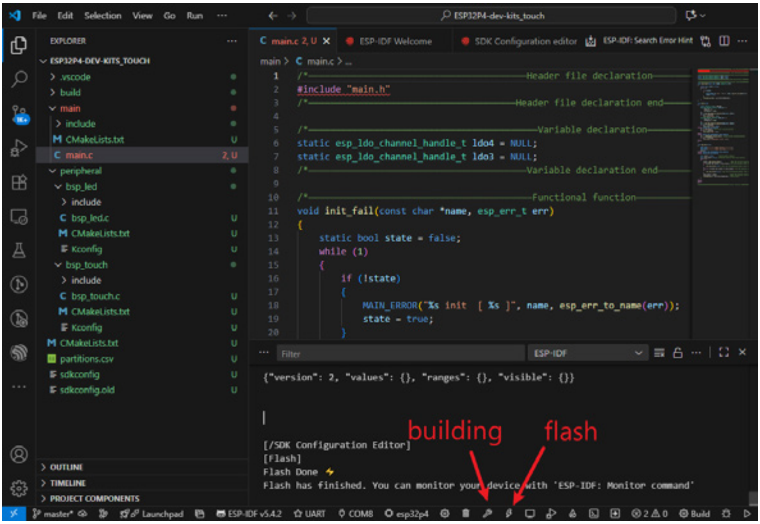
1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the DSIplay, IIC, and LSM6DS3 pins.





1.4.3 Click Compile. Once compilation is successful, click Download.



Lesson 14 - WS2814 RGBW Control

Introduction

This chapter's tutorial demonstrates the ESP32-P4's control application for WS2814 RGB LED strips. Through examples of illuminating and transitioning lighting effects, it aids in understanding its fundamental capabilities. Serving as an advanced test case, illuminating the LED strip provides readers with an intuitive and in-depth understanding of the ESP32-P4's peripheral driver capabilities, laying the groundwork for more complex projects such as LED strip matrix displays and ambient lighting control.

Project Demonstration Effect

This chapter is divided into the following subsections

- 1.1 Introduction to WS2814 and RGB LEDs
- 1.2 Hardware Design
- 1.3 Software Design
- 1.4 Download and Verification

1.1 WS2814 Introduction

1.1.1 WS2814 Introduction

WS2814 is a control chip featuring an integrated constant-current driver and signal decoding functionality, typically encapsulated with LEDs to form programmable light beads. Its key characteristics include:

① Single-wire control: WS2814 is controlled via a single data line, supporting cascading connections where multiple LEDs can be linked sequentially, significantly simplifying circuit wiring.

② Constant-current drive: Each channel incorporates an internal constant current source, ensuring uniform brightness and preventing uneven illumination caused by voltage fluctuations.

③ 16-bit grey scale: WS2814 supports 256 brightness levels (8-bit), enabling rich colour display effects.

④ Fault tolerance: Supports resume-from-break functionality. Should an LED fail, subsequent LEDs continue operating normally, enhancing reliability.

⑤ Voltage compatibility: WS2814 typically operates at 5V, with current consumption dependent on LED colour and brightness.

The ESP32-P4 can drive WS2814 via the RMT peripheral or PWM + precise timing, simplifying complex colour light control.

WS2814 is commonly used in RGB light strips.

1.2 Hardware design

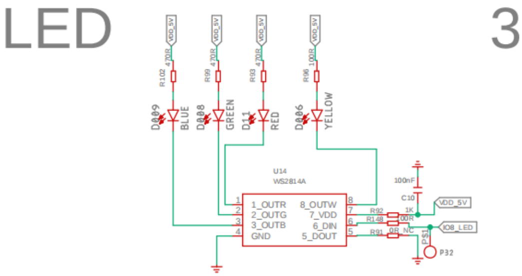
This experiment employs an ESP32-P4 development board with four WS2814 RGB LEDs connected in series. The circuit connections are as follows:

ESP32-P4 GPIO 8 → WS2814 DIN (Data Input)

5V Power Supply → WS2814 VCC

GND → WS2814 GND

WS2814 DOUT → Next WS2814 DIN (Cascade)



1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.

Within the **ESP32P4-dev-kits_led** example, a new folder named **bsp_led** has been created under the **ESP32P4-dev-kits_led\peripheral** directory. Within the **bsp_led** path, an include folder, a **CMakeLists.txt** file, and a **Kconfig** file have been established. The **bsp_led** folder is designated for storing the **bsp_led.c** driver file. The `**include**` folder holds the `**bsp_led.h**` header file, while the `**CMakeLists.txt**` file integrates the driver into the build system, enabling project utilisation of its functionality. The `**Kconfig**` file registers the entire driver alongside GPIO pin definitions within the `**sdkconfig**` file on the IDF platform (configurable via the graphical interface).

1.3.1 RMT LED Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The LED driver source code comprises two files: bsp_led.c and bsp_led.h.

Below we shall first analyse the bsp_led.h programme: it defines the RMT output channel pins and declares the functions for LED usage.

/* Header file references */

```c
/*——————————————————————————————————Header file declaration——————————————————————————————————*/
#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 "driver/rmt_tx.h"     //References for RMT TX Function-related API Functions
/*——————————————————————————————————Header file declaration end——————————————————————————————*/
```

/* Function declarations and macro definitions */

```c
#ifdef CONFIG_BSP_LED_ENABLED

#define LED_GPIO CONFIG_LED_GPIO              // LED GPIO
#define RMT_LED_STRIP_RESOLUTION_HZ 10000000 // Channel clock resolution, in Hz

typedef enum
{
    LED_BYTE_RED = 0,
    LED_BYTE_YELLOW,
    LED_BYTE_GREEN,
    LED_BYTE_BLUE,
    LED_BYTE_DONE,
} led_byte_order_t;

esp_err_t led_init();                                                        // RMT LED Initialization Function
esp_err_t set_led_status(uint32_t bit);                                      // Set led display status
esp_err_t set_single_led_status(led_byte_order_t color, uint8_t brightness); // Set single led display status
#endif
```

This structure defines the colours for the LEDs connected to the four pins of the WS2814A, facilitating subsequent control via functions.

Next, we shall examine the bsp_led.c programme: initialising the RMT controller and exposing API interface functions.

/* RMT controller initialisation function led_init */

```c
esp_err_t led_init()
{
    esp_err_t err = ESP_OK;
    rmt_tx_channel_config_t tx_chan_config = {
        .clk_src = RMT_CLK_SRC_DEFAULT,                 /*Clock source of RMT TX channel*/
        .gpio_num = LED_GPIO,                           /*GPIO number used by RMT TX channel*/
        .mem_block_symbols = 64,                        /*The size of the memory block bytes(64 * 4 = 256)*/
        .resolution_hz = RMT_LED_STRIP_RESOLUTION_HZ,   /*10 MHz tick resolution(1 tick = 0.1 µs)*/
        .trans_queue_depth = 4,                         /*max num*/
    };
    err = rmt_new_tx_channel(&tx_chan_config, &led_chan); /*Create a RMT TX channel*/
    if (err != ESP_OK)
    {
        LED_ERROR("create new rmt tx channel fail");
        return err;
    }
    const rmt_simple_encoder_config_t simple_encoder_cfg = {
        .callback = encoder_callback, /*Callback to call for encoding data into RMT items*/
    };
    err = rmt_new_simple_encoder(&simple_encoder_cfg, &led_encoder); /*Create RMT simple callback encoder*/
    if (err != ESP_OK)
    {
        LED_ERROR("create new rmt simple encoder fail");
        return err;
    }
    err = rmt_enable(led_chan); /*Enable the RMT channel*/
    if (err != ESP_OK)
    {
        LED_ERROR("enable rmt fail");
        return err;
    }
    return err;
}
```

Within the `led_init` function, configuration is first performed for the `rmt_tx_channel_-config_t` structure. The parameters specified are: the clock source for the TX channel, the GPIO pin number utilised by the TX channel, the size of the memory block, the channel clock resolution, and the maximum depth of the internal transmission queue. Subsequently, a new RMT output channel is created, the callback function for RMT signal processing is registered within the RMT decoder, and finally, the RMT controller is enabled.

It is worth noting that here we set the channel clock resolution to 10 MHz. Since the WS2814A control signals operate at the nanosecond level, a resolution of 10 MHz is set (meaning one tick equals 0.1 microseconds).

/* Signal definition for RMT output to WS2814A chip */

```c
/*The layout of RMT symbol stored in memory*/
static const rmt_symbol_word_t signal_zero = {
    .level0 = 1,     /*Level of the first part*/
    .duration0 = 4, /*Duration of level0*/
    .level1 = 0,     /*Level of the second part*/
    .duration1 = 9, /*Duration of level1*/
};

static const rmt_symbol_word_t signal_one = {
    .level0 = 1,     /*Level of the first part*/
    .duration0 = 6, /*Duration of level0*/
    .level1 = 0,     /*Level of the second part*/
    .duration1 = 7, /*Duration of level1*/
};

static const rmt_symbol_word_t signal_reset = {
    .level0 = 0,        /*Level of the first part*/
    .duration0 = 1500, /*Duration of level0*/
    .level1 = 0,        /*Level of the second part*/
    .duration1 = 1500, /*Duration of level1*/
};
```

First, parsing the contents of the structure:

Level0 and duration0 define the level type and duration of the signal's first portion.

Level1 and duration1 define the level type and duration of the signal's second portion.

According to the signal requirements of the WS2814A chip: - A signal with a high level lasting 400ns and a low level lasting 900ns represents signal value 1. - A signal with a high level lasting 600ns and a low level lasting 700ns represents signal value 0. - A low level signal lasting over 280ns constitutes a reset signal.

/* RMT controller decoding callback function **encoder_callback** */

```c
/*Callback to call for encoding data into RMT items*/
static size_t encoder_callback(const void *data, size_t data_size, size_t symbols_written, size_t symbols_free, rmt_symbol_word_t *symbols, bool *done, void *arg)
{
    /* Check if there's enough space in the symbols buffer for a full byte (8 symbols)*/
    if (symbols_free < 8) /*The maximum amount of symbols that can be written into the `symbols` buffer*/
    {
        return 0; /*Not enough space, return 0 to indicate no symbols were written*/
    }
    /*Calculate current position in the data stream based on symbols already written*/
    /*Each byte requires 8 symbols, so divide by 8 to get the byte position*/
    size_t data_pos = symbols_written / 8; /*Current position in encoded stream, in symbols*/
    uint8_t *data_bytes = (uint8_t *)data; /*Cast void pointer to byte array for easier access */
    if (data_pos < data_size)              /*Check if we still have data bytes to encode*/
    {
        size_t symbol_pos = 0; /*Position counter within the current symbols buffer*/
        for (int bitmask = 0x80; bitmask != 0; bitmask >>= 1)
        {
            /*Encode each bit of the current byte (MSB first)*/
            /*Bitmask starts at 0x80 (10000000 binary) and shifts right each iteration*/
            if (data_bytes[data_pos] & bitmask) /*Check if the current bit is set (1) or clear (0)*/
            {
                symbols[symbol_pos++] = signal_one; /*use the symbol defined for logic high*/
            }
            else
            {
                symbols[symbol_pos++] = signal_zero; /*use the symbol defined for logic low*/
            }
        }
        return symbol_pos;
    }
    else
    {
        symbols[0] = signal_reset; /*All data bytes have been encoded, send reset symbol to indicate end of transmission*/
        *done = 1;                 /*Write reset symbol to mark transmission end*/
        return 1;
    }
}
```

First, determine whether the current symbol buffer possesses sufficient space to accommodate a character (8 symbols). If this condition is not met, exit immediately. If space requirements are satisfied, ascertain the current position within the data stream based on the symbols already written. (Each byte requires 8 symbols; the number of bytes written can be determined by dividing the number of symbols written by 8). If the current data has not yet completed encoding, execute the signal encoding configuration (determine whether signal 0 or signal 1 should be output at the current position). Upon completion of data encoding, output the reset signal to refresh the LED display.

/* LED function for setting a single LED: set_single_led_status */

```c
esp_err_t set_single_led_status(led_byte_order_t color, uint8_t brightness)
{
    esp_err_t err = ESP_OK;
    /*Configure RMT transmission for single shot operation (no repetition)*/
    rmt_transmit_config_t tx_config = {
        .loop_count = 0,
    };
    /*Set the specified color component to the desired brightness and turn off all other color components*/
    switch (color)
    {
    case LED_BYTE_RED:
        led_strip_pixels[0] = 0x00;       /*Yellow component off*/
        led_strip_pixels[1] = brightness; /*Red component on*/
        led_strip_pixels[2] = 0x00;       /*Green component off*/
        led_strip_pixels[3] = 0x00;       /*Blue component off*/
        break;
    case LED_BYTE_GREEN:
        led_strip_pixels[0] = 0x00;
        led_strip_pixels[1] = 0x00;
        led_strip_pixels[2] = brightness;
        led_strip_pixels[3] = 0x00;
        break;
    case LED_BYTE_BLUE:
        led_strip_pixels[0] = 0x00;
        led_strip_pixels[1] = 0x00;
        led_strip_pixels[2] = 0x00;
        led_strip_pixels[3] = brightness;
        break;
    case LED_BYTE_YELLOW:
        led_strip_pixels[0] = brightness;
        led_strip_pixels[1] = 0x00;
        led_strip_pixels[2] = 0x00;
        led_strip_pixels[3] = 0x00;
        break;
    default:
        break;
    }
```

```c
    err = rmt_transmit(led_chan, led_encoder, led_strip_pixels, sizeof(led_strip_pixels), &tx_config); /*Transmit data by RMT TX channel*/
    if (err != ESP_OK)
    {
        LED_INFO("transmit rmt cmd fail");
        return err;
    }
    err = rmt_tx_wait_all_done(led_chan, portMAX_DELAY); /*Wait for all pending TX transactions done*/
    if (err != ESP_OK)
    {
        LED_INFO("transmit rmt fail");
        return err;
    }
    return err;
}
```

Firstly, this function has only one input parameter: bit - the 32-bit data parameter controlling the output of the four LEDs on the WS2814A (as detailed above, the specific values control the brightness of the lights). The data bits 0-7 correspond to the blue light, 8-15 to the green light, 16-23 to the amber light, and 24-31 to the red light.

The input data undergoes a shift operation to split it into individual 8-bit segments. Similar to the function controlling a single LED, the corresponding data is configured into the output array, then the send function is invoked to transmit it. (Use case: inputting 0x00000000 extinguishes all lights).

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the sdkconfig file, enabling certain parameter settings to be modified via a graphical interface. Here, the number 8 corresponds to **GPIO_NUM_8**.

```
menu "BSP LED Setup"
    config BSP_LED_ENABLED
        bool "Enable LED config"
        default n

    if BSP_LED_ENABLED
        config LED_GPIO
            int "GPIO For LED"
            default 8
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_led** driver. To successfully call the contents of the **bsp_led** folder from other functions, it is necessary to configure the **CMakeLists.txt** file within the **bsp_led** folder. The configuration details are as follows:

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

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

Within this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver libraries. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the project to utilise the bsp_led driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable **main.c** and the header file **main.h** within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The **main.h** file primarily references required header files: functions utilising the **bsp_led** driver necessitate inclusion of the **bsp_led** header file.

Below is an analysis of the main.c programme: system initialisation and execution specific to the RMT LED functionality.

```
            init_fail("io64", err);
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*RMT LED Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
    set_led_status(0x00000000); /*All the LEDs are off*/
#endif
```

This code resides within the init function, which serves to store initialisation functions requiring invocation and assess their return outcomes. Should the return status deviate from **ESP_OK**, the code will output an error message and halt further execution. Following a 200ms delay, the set_led_status function is executed to clear all LED displays. This step ensures that upon power-up, all LEDs are in an extinguished state.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_LED_ENABLED
    xTaskCreate(led_task, "led", 4096, NULL, configMAX_PRIORITIES - 5, &led); /*Create a led display refresh thread*/
#endif
}
```

Within the app_main function, create a FreeRTOS thread to execute the LED running light effect refresh.

/* LED running light thread led_task */

```
void led_task(void *param)
{
    led_byte_order_t led_byte = LED_BYTE_RED; /*Initialize the default display color*/
    while (1)
    {
        /*Set the lighting of a single lamp*/
        set_single_led_status(led_byte++, 0xFF); /*led_byte++:cycled through all available colors,0xFF = full brightness (255)*/
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        if (led_byte == LED_BYTE_DONE) /*Check if we've cycled through all available colors*/
        {
            led_byte = LED_BYTE_RED; /*Reset to Red color to restart the cycle*/
        }
    }
}
```

Within the LED running light thread, first establish a while loop. Within this loop, invoke the **`set_single_led_status`** function to activate the desired LED. The sequence defined within our LED structure corresponds to the left-to-right arrangement of LEDs on the development board. Therefore, incrementing the enumeration variable sequentially achieves the effect of illuminating each LED in turn (extinguishing previously lit LEDs). This sequence executes once every second. Upon reaching the final enumerated LED value, the sequence resets to the initial red LED state.

1.3.5 CMkaLists.txt file

To successfully call the contents of the bsp_led folder within the main function, it is necessary to configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_led)
```

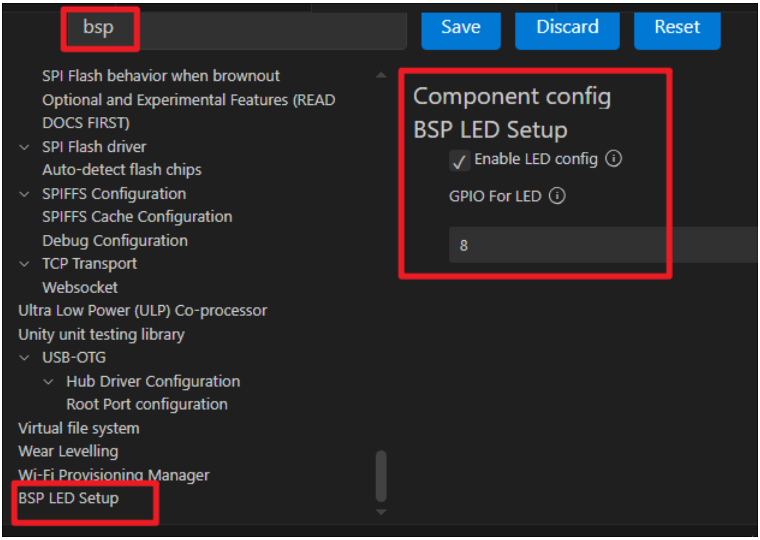
First, the directories for source files and header files are defined, along with the required driver library—specifically, the driver library for linking bsp_led. Subsequently, these settings are registered with the build system via the idf_component_register command, enabling the main function to utilise these driver capabilities.

1.4 Programming procedure

Connect the P4 device to the computer via USB

1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the led pins.



1.4.3 Click Compile. Once compilation is successful, click Download.

Lesson 15 – ADC Button Control

Introduction

This tutorial chapter introduces the ADC input application of the ESP32-P4. Through a routine controlling four LEDs via four buttons, it aids in understanding the fundamental functions of analogue signal detection and multiplexed control. As a common human-machine interaction method, button inputs combined with the ADC voltage divider detection technique enable multi-key input whilst occupying only a single ADC pin, significantly enhancing pin utilisation. This example provides readers with an intuitive grasp of the ESP32-P4's ADC applications, laying the groundwork for more complex sensor and interactive projects.

Project Demonstration Effect

This chapter is divided into the following subsections

1.1 ADC and Button Introduction

1.1.1 ADC Introduction

The ESP32-P4 chip incorporates a multi-channel ADC (analogue-to-digital converter) capable of converting analogue voltages ranging from 0 to 3.3V into digital values. Its principal features include:

① Multi-channel input: Supports multiple ADC channels, enabling connection to several sensors or input signals.

② High resolution: The ESP32-P4's ADC supports 12-bit resolution, with conversion results ranging from 0 to 4095.

③ Flexible sampling: Configurable sampling period and attenuation mode to accommodate different input voltage ranges.

④ Versatile applications: Commonly used for button voltage divider detection, potentiometers, temperature sensors, battery voltage monitoring, and similar scenarios.

In this chapter's experiment, we shall utilise the ADC's voltage detection capability to identify inputs from different buttons via a resistor voltage divider circuit, thereby controlling the corresponding LED lights accordingly.

1.1.2 Button Overview

Button Operating Principle

Buttons are the most common type of switch device, conducting when pressed and breaking when released. To conserve I/O pins, a resistor voltage-dividing method can be employed, connecting multiple buttons to the same ADC channel:

Each button is connected in series with a resistor of a different value;

When different buttons are pressed, the ADC acquires different voltages;

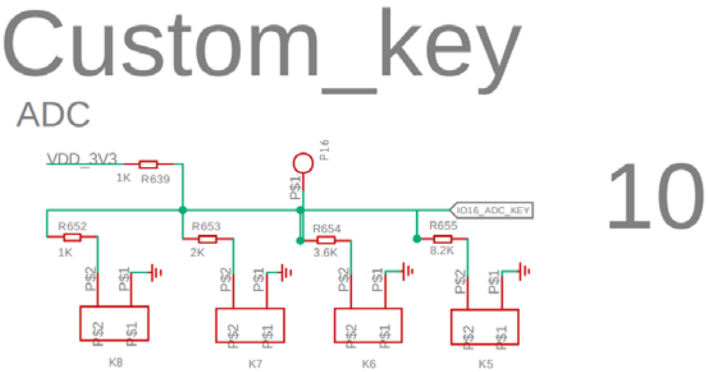
The programme determines which button has been pressed based on the voltage range.

1.2 Hardware design

The hardware comprises one ADC input pin and four buttons:

ADC input pin: GPIO16

Button circuitry: Each of the four buttons is connected to the ADC pin via a voltage divider circuit using resistors of differing values.

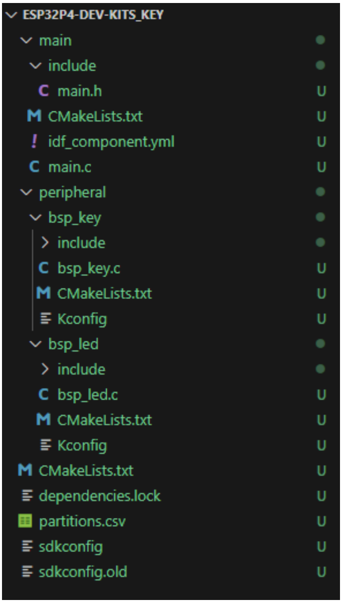


Schematic diagram

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_key** example, a new folder named **bsp_key** has been created under the **ESP32P4-dev-kits_key\peripheral** directory. Within the **bsp_key** path, an include folder, a **CMakeLists.txt** file, and a **Kconfig** file have been established. The **bsp_key** folder is designated for storing the **bsp_key.c** driver file. The `**include**` folder holds the `**bsp_key.h**` header file, while the `**CMakeLists.txt**` file integrates the driver into the build system, enabling project utilisation of its functionality. The `**Kconfig**` file loads the entire driver configuration, including GPIO pin definitions, into the `**sdkconfig**` file within the IDF platform (configurable via the graphical interface).

1.3.1 ADC Button Driver Code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The ADC key driver source code comprises two files: bsp_key.c and bsp_key.h.

Below we shall first analyse the bsp_key.h programme: it defines the relevant ADC key pins and declares the functions used.

/* Header file references */

```
/*------------------------------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 "driver/gpio.h" //References for GPIO Function-related API Functions
#include "iot_button.h"  //References for IOT Button Function-related API Functions
#include "button_adc.h"  //References for ADC Button Function-related API Functions
/*-----------------------------Header file declaration end-----------------------------*/
```

/* Function declarations and macro definitions */

```
#ifdef CONFIG_BSP_KEY_ENABLED

#define KEY_GPIO CONFIG_KEY_GPIO // The GPIO used for the ADC button channel

typedef enum
{
    Key_idle = 0,
    Key_left,
    Key_middle,
    Key_right,
    Key_up,
    Key_release,
} key_state; // Key Press State Structure

esp_err_t key_init(adc_oneshot_unit_handle_t handle); // ADC Key Initialization Function
esp_err_t key_register_cb();                          // Key press status registration callback function
extern key_state key_status;
#endif
```

This structure defines the press and release effects for the four ADC buttons, facilitating subsequent control via functions.

Next, we shall analyse the bsp_key.c programme: initialising and configuring the ADC buttons, utilising callback functions, and introducing API interface functions.

/* ADC button initialisation function key_init */

```
esp_err_t key_init(adc_oneshot_unit_handle_t handle)
{
    esp_err_t err = ESP_OK;
    /*Configure button timing parameters*/
    button_config_t btn_cfg = {
        .long_press_time = 3000, /*Trigger time(ms) for long press*/
        .short_press_time = 200, /*Trigger time(ms) for short press*/
    };
    /*Configure ADC parameters for 4 buttons on the same ADC channel*/
    button_adc_config_t btn_adc_cfg[4] = {
        /*Button 0: Voltage range 1500-1900*/
        {
            .adc_handle = &handle,          /*Pointer to ADC unit handle*/
            .unit_id = ADC_UNIT_1,          /*Use ADC unit 1*/
            .adc_channel = ADC_CHANNEL_0,   /*Use ADC channel 0*/
            .button_index = 0,              /*Identifier for button 0*/
            .min = 1500,                    /*Minimum ADC value for button 0 detection*/
            .max = 1900,                    /*Maximum ADC value for button 0 detection*/
        },
        /*Button 1: Voltage range 2000-2320*/
        {
            .adc_handle = &handle,
            .unit_id = ADC_UNIT_1,
            .adc_channel = ADC_CHANNEL_0,
            .button_index = 1,
            .min = 2000, /*Minimum ADC value for button 1 detection*/
            .max = 2320, /*Maximum ADC value for button 1 detection*/
        },
        /*Button 2: Voltage range 2400-2650*/
        {
            .adc_handle = &handle,
            .unit_id = ADC_UNIT_1,
            .adc_channel = ADC_CHANNEL_0,
            .button_index = 2,
            .min = 2400, /*Minimum ADC value for button 2 detection*/
            .max = 2650, /*Maximum ADC value for button 2 detection*/
        },
        /*Button 3: Voltage range 2700-3000*/
        {
            .adc_handle = &handle,
            .unit_id = ADC_UNIT_1,
            .adc_channel = ADC_CHANNEL_0,
            .button_index = 3,
            .min = 2700, /*Minimum ADC value for button 3 detection*/
            .max = 3000, /*Maximum ADC value for button 3 detection*/
        },
    };
```

```
    for (size_t i = 0; i < 4; i++)
    {
        err = iot_button_new_adc_device(&btn_cfg, &btn_adc_cfg[i], &btns[i]); /*Create a new ADC button device*/
        if (err != ESP_OK)
            return err;
        if (btns[i] == NULL) /*Verify that the button handle was created successfully*/
            return err;
    }
    return err;
}
```

Within the key_init function, configuration begins with the button_config_t structure, where parameters specify the long-press recognition duration and short-press recognition duration. Subsequently, the button_adc_config_t structure is configured, with parameters including the ADC controller handle in use, the ADC controller number, the ADC acquisition channel, the key recognition sequence number, the minimum ADC voltage value for the key, and the maximum ADC voltage value for the key. A for loop is employed to configure each of the four buttons sequentially, obtaining the returned handles.

It is important to note: when setting the voltage ranges for the ADC buttons, overlapping ranges must be avoided, as this will prevent proper button recognition.

/* ADC button event registration function key_register_cb() */

```c
esp_err_t key_register_cb()
{
    esp_err_t err = ESP_OK;
    for (size_t i = 0; i < 4; i++)
    {
        err = iot_button_register_cb(btns[i], BUTTON_PRESS_DOWN, NULL, button_event_cb, (void *)i); /*Register the button event callback function*/
        if (err != ESP_OK)
            return err;
        err = iot_button_register_cb(btns[i], BUTTON_PRESS_UP, NULL, button_event_cb, (void *)i);
        if (err != ESP_OK)
            return err;
    }
    return err;
}
```

This function employs a for loop to register all four button callbacks. The iot_button_register_cb function is used to register button event callbacks (utilising the button handles obtained during initialisation).

It is worth noting that this experiment only utilises button press and release events. To employ other events, consult the enumeration type. Furthermore, ADC buttons cannot support combination key events, as the ADC values generated by pressing multiple buttons simultaneously are unpredictable.

/*ADC button event callback function button_event_cb*/

```c
static void button_event_cb(void *arg, void *data)
{
    button_event_t event = iot_button_get_event(arg); /*Get button event*/
    switch (event)
    {
    case BUTTON_PRESS_DOWN: /*Press the button*/
        switch ((int)data)  /*Determine the key sequence identifier*/
        {
        case 0:
            key_status = Key_left;
            KEY_INFO("left btn : BUTTON_PRESS_DOWN");
            break;
        case 1:
            key_status = Key_right;
            KEY_INFO("right btn : BUTTON_PRESS_DOWN");
            break;
        case 2:
            key_status = Key_middle;
            KEY_INFO("middle btn : BUTTON_PRESS_DOWN");
            break;
        case 3:
            key_status = Key_up;
            KEY_INFO("up btn : BUTTON_PRESS_DOWN");
            break;
        }
        break;
    case BUTTON_PRESS_UP: /*Release the button*/
        switch ((int)data)
        {
        case 0:
            KEY_INFO("left btn : BUTTON_PRESS_UP");
            break;
        case 1:
            KEY_INFO("right btn : BUTTON_PRESS_UP");
            break;
        case 2:
            KEY_INFO("middle btn : BUTTON_PRESS_UP");
            break;
        case 3:
            KEY_INFO("up btn : BUTTON_PRESS_UP");
            break;
        }
        key_status = Key_release;
        break;
    default:
        break;
    }
```

First, the callback function executes upon detecting a key press event. Within the key press event handler, the key identifier can be determined to identify which key triggered the event. For each of the four keys, the key state variable is assigned accordingly. The key release state requires no identification of the triggering key.

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the **sdkconfig** file, enabling certain parameter settings to be modified via a graphical interface. Here, 16 refers to **GPIO_NUM_16**.

```
menu "BSP KEY Setup"

    config BSP_KEY_ENABLED
        bool "Enable KEY"
        default n

    if BSP_KEY_ENABLED
        config KEY_GPIO
            int "GPIO For KEY"
            default 16
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_key** driver. To successfully call the contents of the **bsp_key** folder from other functions, it is necessary to configure the **CMakeLists.txt** file within the **bsp_key** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver library (the button library). Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to utilise the **bsp_key** driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the header file main.h within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The main.h file primarily references required header files: functions utilising the **bsp_led** driver necessitate inclusion of the **bsp_led** header file, while those employing the **bsp_key** driver require the **bsp_key** header file.

Below is an analysis of the main.c programme: system initialisation and execution of ADC key functions.

```c
    err = adc_oneshot_new_unit(&init_cfg, &adc_handle); /*Create a handle to a specific ADC unit*/
    if (err != ESP_OK)
        init_fail("new adc oneshot", err);
```

```c
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*RMT LED Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
    set_led_status(0x00000000); /*All the LEDs are off*/
#endif
#ifdef CONFIG_BSP_KEY_ENABLED
    err = key_init(adc_handle); /*ADC Key Initialization*/
    if (err != ESP_OK)
        init_fail("key", err);
    err = key_register_cb(); /*Key press status registration callback*/
    if (err != ESP_OK)
        init_fail("key register cb", err);
#endif
```

This code resides within the init function, which serves to store initialisation functions requiring invocation and assess their return status. Should the return status not be ESP_OK, the code will output an error message and halt further execution. After a 200ms delay, the `set_led_status` function is executed to clear all LED displays. This step ensures all LEDs are off upon power-up. Calling the `key_register_cb` function registers the key press event callback.

It is worth noting that prior to calling the key initialisation, we must first create a new handle pointing to the ADC controller. This handle is passed into the key initialisation function for configuration.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_KEY_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    xTaskCreate(key_task, "key", 4096, NULL, configMAX_PRIORITIES - 5, &key); /*Create a thread for turning on the LED light by pressing a button*/
#endif
#endif
}
```

Within the app_main function, create a FreeRTOS thread to execute the detection and corresponding LED effect for key presses.

/* ADC key press execution thread key_task */

```c
void key_task(void *param)
{
    esp_err_t err = ESP_OK;
    while (1)
    {
        /*Process key events based on the current key_status*/
        switch (key_status)
        {
        case Key_idle:
            /*No key pressed*/
            vTaskDelay(10 / portTICK_PERIOD_MS);
            break;
        case Key_left:
            if (set_single_led_status(LED_BYTE_RED, 0xFF) != ESP_OK) /*Left button pressed - set LED to red*/
            {
                MAIN_ERROR("Failed to set LED status");
            }
            key_status = Key_idle;
            break;
        case Key_right:
            if (set_single_led_status(LED_BYTE_BLUE, 0xFF) != ESP_OK) /*Right button pressed - set LED to blue*/
            {
                MAIN_ERROR("Failed to set LED status");
            }
            key_status = Key_idle;
            break;
        case Key_middle:
            if (set_single_led_status(LED_BYTE_YELLOW, 0xFF) != ESP_OK) /*middle button pressed - set LED to yellow*/
            {
                MAIN_ERROR("Failed to set LED status");
            }
            key_status = Key_idle;
            break;
```

```c
        case Key_up:
            if (set_single_led_status(LED_BYTE_GREEN, 0xFF) != ESP_OK) /*up button pressed - set LED to green*/
            {
                MAIN_ERROR("Failed to set LED status");
            }
            key_status = Key_idle;
            break;
        case Key_release:                              /*Key released - turn off all LEDs*/
            if (set_led_status(0x00000000) != ESP_OK) /*All the LEDs are off*/
            {
                MAIN_ERROR("Failed to set LED status");
            }
            key_status = Key_idle;
            break;
        default:
            MAIN_ERROR("Unknown key status: %d", key_status);
            key_status = Key_idle;
            break;
        }
    }
}
```

Within the ADC key execution thread, establish a while loop. Within this loop, call the key status variable to determine the pressed state of different keys, illuminating corresponding LEDs accordingly. Should a key be released, extinguish all LEDs. (When no key operation occurs or after executing key effects, set the variable to the idle state and introduce a 10ms delay, i.e., the scanning recognition interval is 10ms.)

1.3.5 CMkaLists.txt file

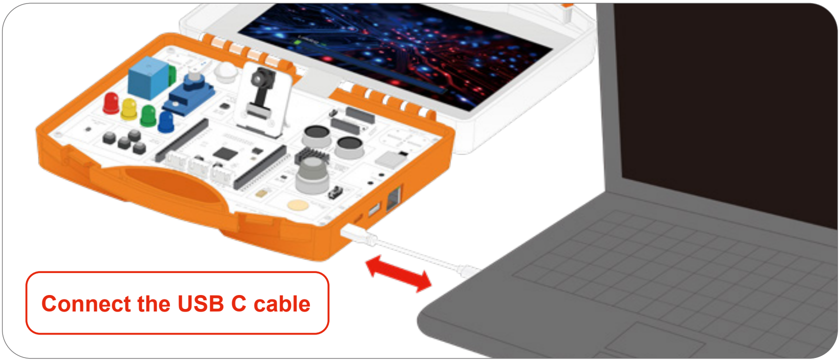
To successfully call the contents of the **bsp_key** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_led bsp_key)
```

First, the directories for source files and header files are defined, along with the required driver library—specifically, the driver library for linking **bsp_key**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling main to utilise these driver functionalities.

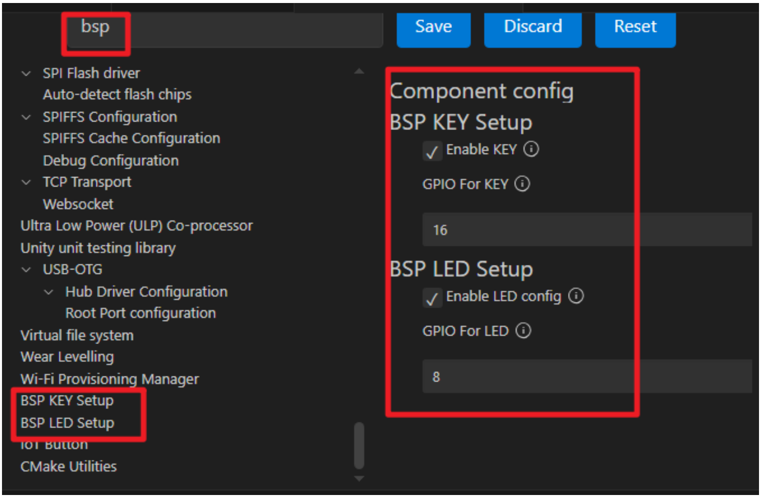
1.4 Programming procedure

Connect the P4 device to the computer via USB

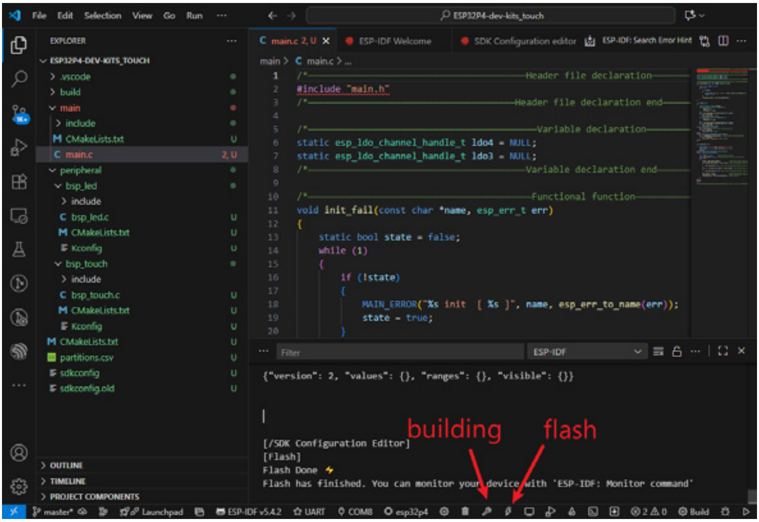


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the key and led pins.



1.4.3 Click Compile. Once compilation is successful, click Download.



Lesson 16 - Smoke Sensor Alert

Introduction

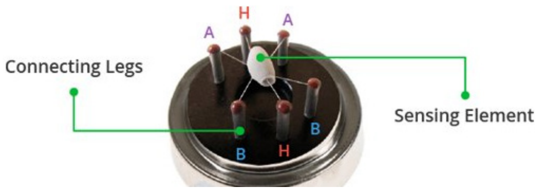
This chapter's tutorial demonstrates the application of the ESP32-P4's ADC inputs by connecting an MQ-2 smoke sensor to detect smoke concentration levels in the environment. The MQ-2, a common gas sensor, detects combustible gases including LPG, butane, methane, alcohol, hydrogen, and smoke. This experiment helps readers grasp the fundamental principles of ADC analogue acquisition and environmental sensing, laying the groundwork for subsequent IoT monitoring and security alarm projects.

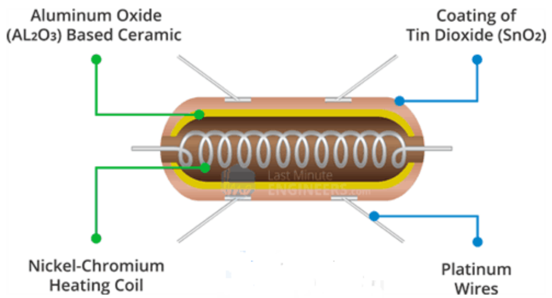
Project Demonstration Effect

This chapter is divided into the following subsections

- 1.1 Introduction to the MQ-2 Sensor
- 1.2 Hardware Design
- 1.3 Programme Design
- 1.4 Download and Verification

1.1 MQ-2 Sensor Introduction





1.1.1 Principles of the MQ-2 Gas Sensor

The MQ-2 sensor comprises a heater and a gas-sensitive resistor (SnO_2 semiconductor material) within its structure. Its operating principle is as follows:

① Under the influence of the heater, oxygen molecules adsorb onto the surface of the gas-sensitive material, causing a change in electrical resistance;

② When combustible gases or smoke are present in the air, oxygen molecules react with the target gas, releasing electrons;

③ These electrons re-enter the semiconductor, reducing the resistance of the gas-sensitive resistor and thereby altering the output voltage;

④ By reading the voltage value via an ADC, changes in gas concentration can be calculated.

1.1.2 Relationship between Simulated Voltage and Smoke Concentration

Clean air: Output voltage is low (small ADC value);

Presence of smoke or combustible gases: Output voltage increases (larger ADC value);

Through calibration and curve fitting, ADC values can be mapped to relative concentrations (ppm).

1.2 Hardware design

Sensor input: The analogue output pin of the MQ-2 connects to GPIO17 (ADC channel) on the ESP32-P4;

Power supply: The MQ-2 module operates at 5V, with an output voltage of 0–3.3V;

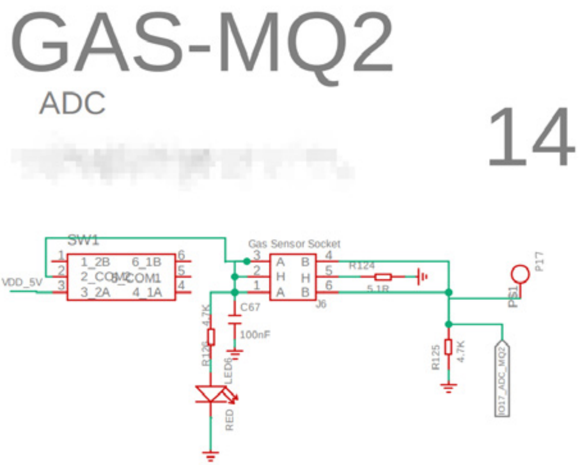
Indicator output (optional): An LED may illuminate when smoke concentration exceeds a specified threshold.

Hardware connection diagram:

VCC → 5V

GND → GND

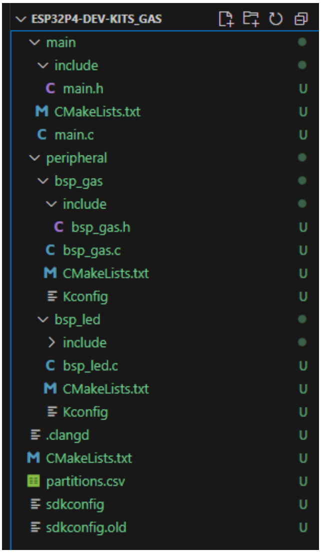
AOUT → GPIO17（ADC input）



1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



Within the **ESP32P4-dev-kits_gas** example, a new folder named **bsp_gas** has been created under the **ESP32P4-dev-kits_gas\peripheral** directory. Within the **bsp_gas** directory, a new include folder, **CMakeLists.txt** file, and Kconfig file have been established. The **bsp_gas** folder is designated for storing the **bsp_gas.c** driver file. The `**include**` folder holds the `**bsp_gas.h**` header file, while the `**CMakeLists.txt**` file integrates the driver into the build system, enabling project utilisation of its functionality. The `**Kconfig**` file loads the entire driver and GPIO pin definitions into the `**sdkconfig**` file within the IDF platform (configurable via the graphical interface).

1.3.1 ADC GAS Driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The MQ2 smoke sensor driver source code comprises two files: **bsp_gas.c** and **bsp_gas.h**.

Below we shall first analyse the **bsp_gas.h** programme: it defines the relevant pins for the smoke sensor and declares the functions used.

/* Header file references */

```c
/*———————————————————————————Header file declaration———————————————————————————*/
#include <math.h>
#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_adc/adc_oneshot.h"     //References for ADC Oneshot Function-related API Functions
#include "esp_adc/adc_cali.h"        //References for ADC Cail Function-related API Functions
#include "esp_adc/adc_cali_scheme.h" //References for ADC Cail Scheme Function-related API Functions
/*———————————————————————————Header file declaration end———————————————————————————*/
```

/* Function declarations and macro definition declarations */

```c
#ifdef CONFIG_BSP_GAS_ENABLED

#define GAS_CHANNEL CONFIG_GAS_CHANNEL // ADC acquisition channel

typedef struct
{
    adc_oneshot_unit_handle_t gas_handle;
    int gas_voltage;
    float gas_vaule;
    float RL;
    float R0;
} gas_config;

esp_err_t gas_init(adc_oneshot_unit_handle_t handle); // GAS ADC Initialization Function
float get_gas_vaule();                                // Obtain the ADC acquisition voltage value of GAS
float get_r0_calibration();                           // Obtain the R0 value (load resistance data under clean air conditions) through data conversion
float get_gas_data();                                 // Obtain the gas concentration value of gas(ppm)
#endif
```

This structure defines parameters for the smoke sensor's load resistor, ADC handle, and acquired values, facilitating subsequent function control.

Next, we shall analyse the bsp_gas.c programme: initialising ADC channel configuration, calculating voltage values, converting ppm parameters, and exposing API interface functions.

/* ADC GAS initialisation function gas_init */

```
esp_err_t gas_init(adc_oneshot_unit_handle_t handle)
{
    esp_err_t err = ESP_OK;
    adc_oneshot_chan_cfg_t adc_config = {
        .bitwidth = SOC_ADC_DIGI_MAX_BITWIDTH, /*ADC conversion result bits*/
        .atten = ADC_ATTEN_DB_12,              /*ADC attenuation*/
    };
    gas_data.gas_handle = handle;                                          /*Obtain the handle of the ADC controller*/
    err = adc_oneshot_config_channel(gas_data.gas_handle, GAS_CHANNEL, &adc_config); /*Set ADC oneshot mode required configurations*/
    if (err != ESP_OK)
    {
        GAS_ERROR("create adc oneshot channel fail");
        return err;
    }
    adc_cali_curve_fitting_config_t cali_config = {
        .unit_id = ADC_UNIT_1,                 /*Use ADC unit 1*/
        .chan = GAS_CHANNEL,                   /*Use ADC channel 1*/
        .atten = ADC_ATTEN_DB_12,              /*ADC attenuation*/
        .bitwidth = SOC_ADC_DIGI_MAX_BITWIDTH, /*ADC conversion result bits*/
    };
    err = adc_cali_create_scheme_curve_fitting(&cali_config, &gas_cali_handle); /*Create a Curve Fitting calibration scheme*/
    if (err != ESP_OK)
    {
        GAS_ERROR("create adc scheme curve  fitting fail");
        return err;
    }
    gas_data.RL = 4.7;
    return err;
}
```

Within the **gas_init** function, configuration of the **adc_oneshot_chan_cfg_t** structure is first performed, setting the ADC read bit width. Subsequently, the **adc_oneshot_config_channel** function is invoked to initialise the ADC channel. Subsequently, the **adc_cali_curve_fitting_config_t** structure is configured. This structure similarly sets parameters for the ADC handle, specifying the relevant channel configuration curve-fitting calibration scheme. The specific creation function is **adc_cali_create_scheme_curve_fitting**. (The final RL=4.7 is the default; in the hardware design, this resistor is indeed 4.7kΩ.)

Within the **gas_init** function, configuration is first performed for the **adc_oneshot_chan_cfg_t** structure. A noteworthy aspect of the configuration values is that the ESP32P4 chip currently only supports the curve-fitting calibration scheme.

/* Function **get_gas_voltage** for obtaining the current GAS voltage value via ADC */

```
esp_err_t get_gas_voltage()
{
    static int vol_sum = 0;
    static uint8_t vol_reading_cnt = 0;
    esp_err_t err = ESP_OK;
    int vaule = 0;
    while (1)
    {
        err = adc_oneshot_read(gas_data.gas_handle, GAS_CHANNEL, &vaule); /*Get one ADC conversion raw result*/
        if (err != ESP_OK)
            return err;
        vol_sum += vaule; /*Accumulate the reading for averaging*/
        vol_reading_cnt++;
        if (vol_reading_cnt == 2) /*Process when we have collected 2 samples*/
        {
            err = adc_cali_raw_to_voltage(gas_cali_handle, vol_sum / vol_reading_cnt, &gas_data.gas_voltage); /*Convert ADC raw data to calibrated voltage*/
            if (err != ESP_OK)
            {
                GAS_ERROR("GAS adc voltage cali fail");
                return err;
            }
            vol_sum = 0;
            vol_reading_cnt = 0;                                          /*Reset accumulators for next reading cycle*/
            gas_data.gas_vaule = ((float)gas_data.gas_voltage) / 1000; /*Convert millivolts to volts (divide by 1000)*/
            break;                                                        /*Exit the while loop after successful processing*/
        }
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    return err;
}
```

```
float get_gas_vaule()
{
    return gas_data.gas_vaule; /*Return the voltage value collected by the ADC*/
}
```

Within the **gas_init** function, configuration is first performed for the **adc_oneshot_chan_cfg_t** structure.

This function first invokes **adc_oneshot_read** to acquire the sampled voltage value from the current ADC channel. Through an accumulative approach, it sums every two acquired sample values before performing an average calculation. The resulting voltage value is then passed as an input parameter to the curve-fitting calibration function for calibration. Finally, the calibrated voltage value is converted from millivolts (mv) to volts (V). Should voltage parameters be required for debugging, they may be retrieved using the **get_gas_value** function.

/* Gas load resistor calibration function (clean air conditions) **get_r0_calibration** */

```
float get_r0_calibration()
{
    static uint8_t vol_reading_cnt = 0;
    static float vol_sum = 0;
    while (1)
    {
        if (get_gas_voltage() == ESP_OK) /*Attempt to read current gas sensor voltage*/
        {
            vol_sum += gas_data.gas_vaule; /*Accumulate voltage value*/
            vol_reading_cnt++;             /*Increment reading counter*/
        }
        if (vol_reading_cnt >= 10) /*Check if we have collected enough samples (10 readings)*/
        {
            /*
             * Calculate R0 using the voltage divider formula:
             * R0 = ((Vcc - Vavg) * RL) / Vavg
             * Where Vavg = vol_sum / vol_reading_cnt
             *
             * This formula derives from:
             * Rs = (Vcc - Vout) * RL / Vout
             * Where Rs is sensor resistance, and in clean air Rs = R0
             */
            gas_data.R0 = ((5 - (vol_sum / vol_reading_cnt)) * gas_data.RL) / (vol_sum / vol_reading_cnt);
            vol_reading_cnt = 0; /*Reset accumulators for potential future calibrations*/
            vol_sum = 0;
            break;
        }
        vTaskDelay(100 / portTICK_PERIOD_MS);
    }
    return gas_data.R0;
}
```

This function utilises the **get_gas_voltage** function to obtain the voltage value currently fed back from the gas sensor pin. Through average filtering, it acquires ten samples and calculates the mean value for conversion. Subsequently, the current load resistance value is derived using the voltage divider formula:

RS = (Vcc - Vout) * RL / Vout.

It should be noted that this method is intended for use in clean air conditions, calculating the initial load resistance value applicable to clean air.

/*gas function to obtain the current air ppm value **get_gas_data*/**

```c
float get_gas_data()
{
    float RS = 0;                    /*Current sensor resistance in ohms*/
    float ppm = 0;                   /*Gas concentration in Parts Per Million*/
    if (get_gas_voltage() == ESP_OK) /*Read current gas sensor voltage*/
    {
        /*
         * Calculate current sensor resistance (RS) using voltage divider formula:
         * RS = ((Vcc - Vout) * RL) / Vout
         * Where:
         * - Vcc = 5V (supply voltage)
         * - Vout = gas_data.gas_vaule (measured voltage across load resistor)
         * - RL = gas_data.RL (load resistor value in ohms)
         *
         * This formula derives from: Vout = Vcc * RL / (RS + RL)
         */
        RS = ((5 - gas_data.gas_vaule) * gas_data.RL) / gas_data.gas_vaule;
        /*
         * Convert resistance ratio to PPM using calibration curve:
         * ppm = ((11.5428 * R0) / RS)^0.6549 * 100
         * Where:
         * - R0 = gas_data.R0 (baseline resistance in clean air)
         * - RS = current sensor resistance
         *
         * This empirical formula is specific to the gas sensor type and characteristics.
         * The constants (11.5428 and 0.6549) are derived from the sensor's datasheet
         * or calibration data for a specific gas (e.g., MQ-2 for LPG, MQ-135 for air quality).
         */
        ppm = pow(((11.5428 * gas_data.R0) / RS), 0.6549f) * 100;
    }
    return ppm;
}
```

This function utilises the **get_gas_voltage** function to obtain the voltage value currently fed back from the gas sensor pin. It calculates the RS resistance value using the voltage divider formula (listed above), then applies the ppm conversion formula to the processed RS resistance value. (adjustable based on specific experiments; here employing the default gas sensor concentration calibration formula). Here, R0 denotes the sensor's resistance value in clean air, RS represents the sensor's resistance value in the current gas environment, 11.5428 is the sensor's characteristic parameter (manufacturer-calibrated), and 0.6549 is the sensor response curve parameter. The expression (11.5428*R0)/RS represents the ratio of the current gas concentration relative to the reference state. The Pow function is the C language power function calculation function. As the gas sensor response is non-linear, an exponential function is required for fitting.

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the **sdkconfig** file, enabling certain parameter settings to be modified via a graphical interface. Here, “1” denotes ADC sampling channel 1.

```
menu "BSP GAS Setup"

    config BSP_GAS_ENABLED
        bool "Enable GAS"
        default n

    if BSP_GAS_ENABLED
        config GAS_CHANNEL
            int "ADC CHANNEL For GAS"
            default 1
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_gas** driver. To successfully call functions from the **bsp_gas** folder within other functions, you must configure the **CMakeLists.txt** file located in the **bsp_gas** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver library (**esp_adc** library). Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to utilise the **bsp_gas** driver functionality.

1.3.4 main folder

The main folder serves as the core directory for programme execution, containing the main function executable **main.c** and the header file **main.h** within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The main.h file primarily references required header files: functions utilising the **bsp_led** driver necessitate inclusion of the **bsp_led** header file, while those employing the **bsp_gas** driver require the **bsp_gas** header file.

Below is an analysis of the **main.c** programme: system initialisation and initialisation specific to the ADC functionality.

```
    err = adc_oneshot_new_unit(&init_cfg, &adc_handle);
    if (err != ESP_OK)
        init_fail("new adc oneshot", err);
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*RMT LED Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
    set_led_status(0x00000000); /*All the LEDs are off*/
#endif
#ifdef CONFIG_BSP_GAS_ENABLED
    err = gas_init(adc_handle); /*GAS ADC Initialization*/
    if (err != ESP_OK)
        init_fail("gas", err);
    float r0 = 0;
    r0 = get_r0_calibration(); /*Obtain the R0 value (load resistance data under clean air conditions) through data conversion*/
    MAIN_INFO("r0 = %.2f K", r0);
    vTaskDelay(5000 / portTICK_PERIOD_MS);
#endif
```

This code resides within the init function, which is used to store initialisation functions requiring invocation and to evaluate their return status. Should the return status not be ESP_OK, the code will print an error message and halt further execution. After a 200ms delay, the `set_led_status` function is executed to clear all LED displays. This step ensures all LEDs are in an off state upon power-up. Calling the `get_r0_calibration` function sets the default initial state to clean air conditions upon power-up, calibrating the value of the R0 resistor.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_GAS_ENABLED
#ifdef CONFIG_BSP_LED_ENABLED
    xTaskCreate(gas_task, "gas", 4096, NULL, configMAX_PRIORITIES - 5, &gas); /*Create a thread for smoke alarm recognition and processing*/
#endif
#endif
}
```

Within the **app_main** function, create a **FreeRTOS** thread to execute the gas concentration monitoring and visual alarm functions of the smoke sensor.

/* Gas concentration monitoring thread gas_task */

```c
void gas_task(void *param)
{
    float ppm_data = 0;                     /*Current gas concentration in PPM*/
    float last_ppm_data = get_gas_data(); /*Previous gas concentration reading*/
    static bool state = false;              /*LED blink state (true=on, false=off)*/
    uint32_t now_time = 0;                  /*Current time in milliseconds*/
    uint32_t last_time = 0;                 /*Last LED toggle time for blinking*/
    while (1)
    {
        now_time = esp_timer_get_time() / 1000;          /*Get current time in milliseconds from system timer*/
        MAIN_INFO("gas_vaule = %.2f", get_gas_vaule()); /*Log raw gas sensor voltage value for debugging*/
        ppm_data = get_gas_data();                       /*Read current gas concentration in PPM*/
        if (ppm_data >= 1000)                            /*Alarm condition: Gas concentration exceeds 1000 PPM threshold*/
        {
            if (now_time >= last_time)
            {
                if (now_time - last_time >= 500) /*Check if 500ms has elapsed since last LED toggle for blinking effect*/
                {
                    state = !state; /*Toggle LED state*/
                    if (state)
                        set_led_status(0xFFFFFFFF); /*Turn LED on*/
                    else
                        set_led_status(0x00000000); /*Turn LED off*/
                    last_time = now_time;           /*Update last toggle time*/
                }
            }
            else
            {
                last_time = now_time; /*Handle timer overflow case (unlikely in normal operation)*/
            }
        }
        else if ((ppm_data < 1000) && (last_ppm_data >= 1000)) /*Alarm cancellation condition: Concentration drops below threshold after being above it*/
        {
            set_led_status(0x00000000); /*Turn off LED when gas concentration returns to safe levels*/
        }
        last_ppm_data = ppm_data; /*Store current reading for next iteration comparison*/
        MAIN_INFO("ppm = %.2f", ppm_data);
        vTaskDelay(20 / portTICK_PERIOD_MS);
    }
}
```

Within the gas concentration monitoring thread, variables are first initialised and the initial smoke concentration value is acquired for comparison. A while loop is then established, within which the **esp_timer_get_time** function is invoked to obtain the current system time, and the **get_gas_data** function is called to retrieve the current gas concentration value. If the current gas concentration exceeds **1000ppm**, the LED lights flash once every 500ms (all four lights illuminate simultaneously). This is achieved by subtracting the previous system time from the current system time; if the difference is greater than or equal to 500ms, the LED state variable `state` is inverted. LED control then follows this state variable. If the current gas concentration is below **1000ppm** and the previous reading exceeded 1000ppm, the alarm threshold remains untriggered, and the LEDs are extinguished. Finally, the current gas concentration is assigned to the previous reading for subsequent evaluations. The 20ms delay represents a 20-millisecond interval between each recognition and judgement cycle.

1.3.5 CMkaLists.txt

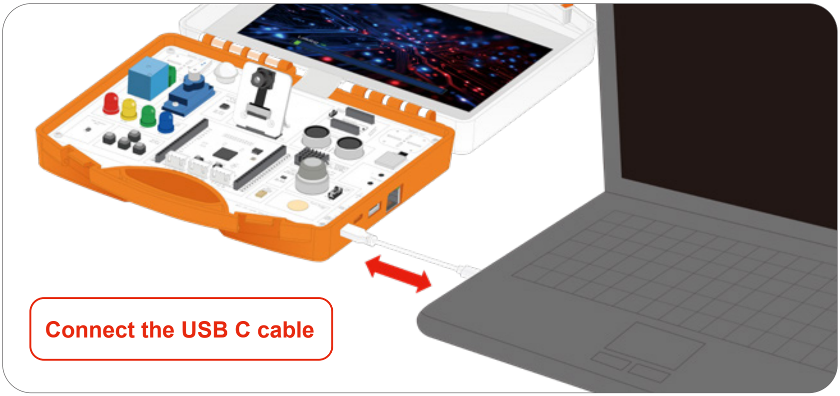
To successfully call the contents of the **bsp_gas** folder within the main function, it is necessary to configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES esp_timer bsp_led bsp_gas)
```

First, the directories for source files and header files are defined, along with the required driver libraries—namely, the driver libraries for linking **bsp_gas**, **bsp_led**, and **esp_timer**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver functionalities.

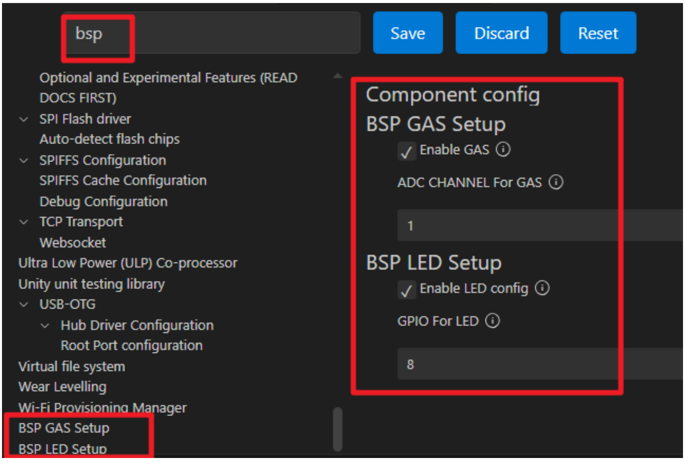
1.4 Programming procedure

Connect the P4 device to the computer via USB

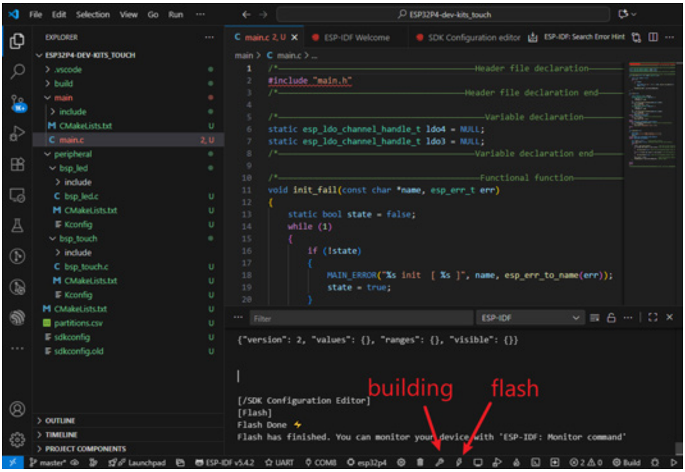


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the adc channel for gas and led pins.



1.4.3 Click Compile. Once compilation is successful, click Download.

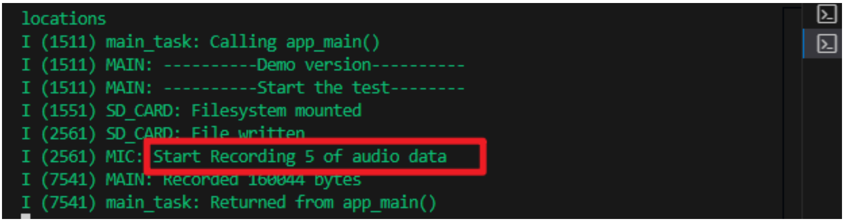


Lesson 17 – I2S Audio Record

Introduction

This chapter's tutorial introduces the I2S-PDM microphone capture and audio storage application for the ESP32-P4, using a recording example to help understand the fundamental functionality of the I2S bus.

As a typical audio application case, recording storage enables readers to quickly grasp the ESP32-P4's capabilities in speech and audio processing, laying the groundwork for more complex projects such as speech recognition and audio playback.



This chapter is divided into the following subsections

1.1 Introduction to I²S and PDM Microphones

1.1.1 I²S Introduction

The ESP32-P4 chip incorporates multiple I2S peripheral interfaces for audio data acquisition and playback. Its key features include:

① Support for multiple audio protocols: including standard I2S, left-aligned, right-aligned, and PDM (Pulse Density Modulation) modes.

② High sampling rate support: Configurable sampling rates from 8kHz to 192kHz, suitable for voice capture and high-fidelity audio processing.

③ DMA transfer: Enables direct audio data transfer via DMA, reducing CPU overhead and enhancing real-time performance.

④ Multi-channel support: Simultaneously processes mono, stereo, and other data formats, facilitating stereo recording or playback.

⑤ Flexible configuration: Operates as either host or slave, supporting both data acquisition and output.

In this example, we shall utilise the ESP32-P4's I2S interface in PDM microphone mode to implement voice data acquisition.

1.1.2 PDM Microphone Overview

PDM (Pulse Density Modulation) is a digital audio output method commonly found in MEMS microphones.

Characteristics of PDM microphones:

① Miniaturisation: Compact size and low power consumption, suitable for embedded devices.

② Digital interface: Outputs directly as a digital pulse stream, eliminating the need for analogue amplifiers.

③ High integration: Most PDM microphones incorporate built-in analogue-to-digital conversion modules.

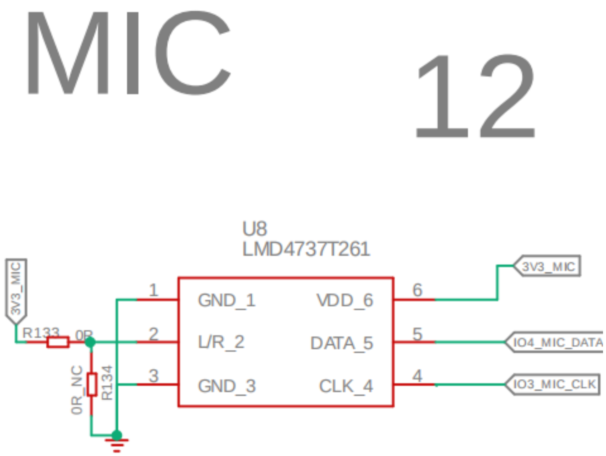
④ Flexible sampling: Can be captured via I²S-PDM interface, filtered, and restored to standard PCM data.

Operating Principle:
When sound waves impinge upon the microphone diaphragm, internal capacitive sensors convert the acoustic energy into electrical signals. These are then processed by a Σ-Δ modulator to generate a high-frequency pulse density modulation (PDM) signal. The ESP32-P4 samples these pulses via the I2S-PDM interface and employs a digital filter to reconstruct them into PCM audio data.

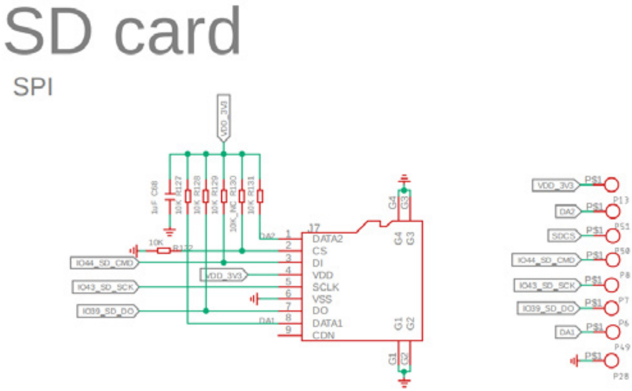
1.1.3 Introduction to WAV Files
WAV files are a common lossless audio format. The file header stores parameters such as the audio's sampling rate, number of channels, and bit depth, while the data section contains the PCM data captured from the samples.
Its primary structure is as follows:
RIFF block: Identifies the file as WAV format
fmt block: Describes the audio format (sampling rate, bit width, channels, etc.)
data block: Stores the actual audio data
By writing the PCM data captured by the PDM microphone to an SD card and encapsulating it as a WAV file, we can achieve standard audio recording functionality.

1.2 Hardware design

In this experiment, the typical connection between the ESP32-P4, PDM digital microphone, and SD card module is as follows:
PDM microphone → ESP32-P4 I2S interface



SD card module → ESP32-P4 SPI interface



In this manner, the ESP32-P4 acquires audio data via I2S and writes it to the SD card via SPI to form WAV files.

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.

Within the **ESP32P4-dev-kits_mic** example, new folders bsp_mic and **bsp_sd** were created under the **ESP32P4-dev-kits_mic\peripheral** path. Within the **bsp_mic** and **bsp_sd** paths, new include folders, **CMakeLists.txt** files, and **Kconfig** files were established. The **bsp_mic** folder houses the **bsp_mic.c** driver file, while the **bsp_sd** folder contains the **bsp_sd.c** driver file. Their respective include folders store .h header files. The **CMakeLists.txt** file integrates the drivers into the build system, enabling project utilisation of driver functionality. The **Kconfig** file loads the entire driver configuration, including GPIO pin definitions, into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 SD Card Driver Code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The SD card driver source code comprises two files: **bsp_sd.c** and **bsp_sd.h**.

Below we shall first analyse the bsp_sd.h programme: it defines the relevant SD card pins and declares the functions used.

/* Header file references */

```c
/*------------------------------------Header file declaration------------------------------------*/
#include "esp_vfs_fat.h"                //References for vfs fat Function-related API Functions
#include "sdmmc_cmd.h"                  //References for sdmmc cmd Function-related API Functions
#include "driver/sdmmc_host.h"          //References for sdmmc host Function-related API Functions
#include "esp_private/sdmmc_common.h"   //References for sdmmc common Function-related API Functions
/*------------------------------------Header file declaration end------------------------------------*/
```

/* Function declarations and macro definitions */

```c
#ifdef CONFIG_BSP_SD_ENABLED

#define EXAMPLE_MAX_CHAR_SIZE 64        // Maximum byte count
#define SD_MOUNT_POINT "/sdcard"        // path where partition should be registered
#define SD_GPIO_CLK CONFIG_SD_GPIO_CLK  // SD SDIO CLK GPIO
#define SD_GPIO_CMD CONFIG_SD_GPIO_CMD  // SD SDIO CMD GPIO
#define SD_GPIO_D0 CONFIG_SD_GPIO_D0    // SD SDIO D0 GPIO

esp_err_t sd_init();                                                                // SD Initialization Function
esp_err_t create_file(const char *filename);                                        // Create a binary file
esp_err_t write_string_file(const char *filename, char *data);                      // Create and write to a text file
esp_err_t read_string_file(const char *filename);                                   // Read to a text file
esp_err_t write_file(const char *filename, char *data, size_t size);                // Create and write to a binary file
esp_err_t write_file_seek(const char *filename, void *data, size_t size, int32_t seek); // Create and write to a binary file that can offset the writing position
esp_err_t read_file(const char *filename, char *data, size_t size);                 // Read to a binary file
esp_err_t read_file_size(const char *filename);                                     // Read the number of bytes in a binary file
esp_err_t format_sd_card();                                                         // Format the SD card
void get_sd_card_info();                                                            // Obtain information about the SD card device
#endif
```

Next, we shall analyse the **bsp_sd.c** programme: it initialises and configures the SD card control bus, mounts it on the file system, and exposes API interface functions.

/* SD card initialisation and file system mounting function sd_init */

```c
esp_err_t sd_init()
{
    esp_err_t err = ESP_OK;
    /*Configuration for FAT filesystem mounting*/
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
#ifdef CONFIG_FORMAT_IF_MOUNT_FAILED
        .format_if_mount_failed = true, /*Format SD card if mount fails*/
#else
        .format_if_mount_failed = false, /*Do not format if mount fails*/
#endif
        .max_files = 5,                   /*Maximum number of open files*/
        .allocation_unit_size = 16 * 1024, /*Cluster size for FAT filesystem (16KB)(Must be a power of 2)*/
    };
    /*SDMMC host controller configuration*/
    sdmmc_host_t host = {
        /*Supported bus modes (1-bit, 4-bit, 8-bit with DDR)*/
        .flags = SDMMC_HOST_FLAG_8BIT |
                 SDMMC_HOST_FLAG_4BIT |
                 SDMMC_HOST_FLAG_1BIT |
                 SDMMC_HOST_FLAG_DDR |
                 SDMMC_HOST_FLAG_DEINIT_ARG,
        .slot = SDMMC_HOST_SLOT_0,                  /*Use SDMMC slot 0*/
        .max_freq_khz = SDMMC_FREQ_DEFAULT,         /*Default frequency (20MHz)*/
        .io_voltage = 3.3f,                         /*I/O voltage level (3.3V)*/
        .driver_strength = SDMMC_DRIVER_STRENGTH_B, /*Default drive strength*/
        .current_limit = SDMMC_CURRENT_LIMIT_200MA, /*200mA current limit*/
        /*Host controller function pointers*/
        .init = &sdmmc_host_init,
        .set_bus_width = &sdmmc_host_set_bus_width,
        .get_bus_width = &sdmmc_host_get_slot_width,
        .set_bus_ddr_mode = &sdmmc_host_set_bus_ddr_mode,
        .set_card_clk = &sdmmc_host_set_card_clk,
        .set_cclk_always_on = &sdmmc_host_set_cclk_always_on,
        .do_transaction = &sdmmc_host_do_transaction,
        .deinit_p = &sdmmc_host_deinit_slot,
        .io_int_enable = &sdmmc_host_io_int_enable,
        .io_int_wait = &sdmmc_host_io_int_wait,
        .command_timeout_ms = 0, /*Use default timeout*/
        .get_real_freq = &sdmmc_host_get_real_freq,
        .input_delay_phase = SDMMC_DELAY_PHASE_0, /*No input delay*/
        .set_input_delay = &sdmmc_host_set_input_delay,
        .dma_aligned_buffer = NULL, /*No DMA buffer*/
        .pwr_ctrl_handle = NULL,    /*No power control*/
        .get_dma_info = &sdmmc_host_get_dma_info,
        .is_slot_set_to_uhs1 = &sdmmc_host_is_slot_set_to_uhs1,
    };
```

Within the `**sd_init**` function, configuration begins with the `**esp_vfs_fat_sdmmc_mount_config_t**` structure to set parameters for mounting the SD card's file system. Subsequently, the `**sdmmc_host_t**` structure is configured to manage SDMMC host controller settings (refer to code comments for specific parameter details). The `**sdmmc_slot_config_t**` structure configures the SD card to utilise single-wire mode on the SDIO bus. Finally, the **esp_vfs_fat_sdmmc_mount** function is invoked to initialise the SD card and mount the file system. Should mounting fail, the error type is logged.

It is worth noting that on the ESP32P4 chip, when operating in Wi-Fi/Bluetooth host mode, the selected SDIO bus slot must differ from that used in Wi-Fi/Bluetooth host mode.

/* Function **get_sd_card_info** retrieves information about the currently mounted SD card */

```c
void get_sd_card_info()
{
    bool print_scr = false;                     /*Flag to control SCR register printing*/
    bool print_csd = false;                     /*Flag to control CSD register printing*/
    const char *type;                           /*String describing card type*/
    SD_INFO("Name: %s\n", card->cid.name); /*Display card name from Card Identification register*/
    if (card->is_sdio)                          /*Determine card type and set appropriate print flags*/
    {
        type = "SDIO";    /*SD Input/Output card*/
        print_scr = true; /*SDIO cards have SCR register*/
        print_csd = true; /*SDIO cards have CSD register*/
    }
    else if (card->is_mmc)
    {
        type = "MMC";     /*MultiMediaCard*/
        print_csd = true; /*MMC cards have CSD register*/
    }
    else
    {
        if ((card->ocr & SD_OCR_SDHC_CAP) == 0) /*Standard SD card type detection*/
            type = "SDSC";                       /*Standard Capacity SD card (up to 2GB)*/
        else
        {
            if (card->ocr & SD_OCR_S18_RA)
                type = "SDHC/SDXC (UHS-I)"; /*High/Extended Capacity with UHS-I support*/
            else
                type = "SDHC"; /*High Capacity SD card (4GB-32GB)*/
        }
        print_csd = true; /*All SD cards have CSD register*/
    }
    SD_INFO("Type: %s\n", type);
    if (card->real_freq_khz == 0) /*Display current and maximum operating frequencies*/
        SD_INFO("Speed: N/A\n");  /*Frequency information not available*/
    else
    {
        /*Convert frequency to appropriate units (kHz or MHz)*/
        const char *freq_unit = card->real_freq_khz < 1000 ? "kHz" : "MHz";
        const float freq = card->real_freq_khz < 1000 ? card->real_freq_khz : card->real_freq_khz / 1000.0;
        const char *max_freq_unit = card->max_freq_khz < 1000 ? "kHz" : "MHz";
        const float max_freq = card->max_freq_khz < 1000 ? card->max_freq_khz : card->max_freq_khz / 1000.0;
        /*Display current speed, maximum capability, and DDR mode if applicable*/
        SD_INFO("Speed: %.2f %s (limit: %.2f %s)%s\n",
                freq, freq_unit,
                max_freq, max_freq_unit,
                card->is_ddr ? ", DDR" : ""); /*Double Data Rate indicator*/
    }
```

```c
    /*Calculate and display total card capacity in Megabytes*/
    SD_INFO("Size: %lluMB\n", ((uint64_t)card->csd.capacity) * card->csd.sector_size / (1024 * 1024));
    /*Display Card Specific Data register information if enabled*/
    if (print_csd)
    {
        SD_INFO("CSD: ver=%d, sector_size=%d, capacity=%d read_bl_len=%d\n",
                (int)(card->is_mmc ? card->csd.csd_ver : card->csd.csd_ver + 1),
                card->csd.sector_size,     /*Size of each sector in bytes*/
                card->csd.capacity,        /*Total number of sectors*/
                card->csd.read_block_len); /*Maximum read block length*/
        /*Display extended information based on card type*/
        if (card->is_mmc)
        {
            SD_INFO("EXT CSD: bus_width=%" PRIu32 "\n",
                    (uint32_t)(1 << card->log_bus_width)); /*MMC bus width (1,4,8 bits)*/
        }
        else if (!card->is_sdio)
        {
            /*Display SD Card Status Register bus width information*/
            SD_INFO("SSR: bus_width=%" PRIu32 "\n",
                    (uint32_t)(card->ssr.cur_bus_width ? 4 : 1)); /*4-bit or 1-bit mode*/
        }
    }
    /*Display SD Configuration Register information if enabled*/
    if (print_scr)
    {
        SD_INFO("SCR: sd_spec=%d, bus_width=%d\n",
                card->scr.sd_spec,    /*SD physical layer specification version*/
                card->scr.bus_width); /*Supported bus widths*/
    }
}
```

This function call initialises the handle “card” obtained from the SD card, reads relevant information about the SD card, and prints the log information.

/* SD card formatting function format_sd_card */

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

This function formats the specified SD card using esp_vfs_fat_sdcard_format.

/* SD card file operations */

create_file: Creates a binary file.

write_string_file: Writes to a text file.

read_string_file: Reads from a text file.

write_file: Writes a binary file.

write_file_seek: Writes a binary file (with an offset, allowing writing beyond the file's start position).

read_file: Reads a binary file.

read_file_size: Reads the data size of a binary file.

read_write_file: Reads a binary file and writes its contents to another binary file.

1.3.2 Kconfig file

The primary function of this file is to add the required configuration to the sdkconfig file, enabling certain parameter settings to be modified via a graphical interface. The numbers here correspond to the respective GPIO pin numbers.

It is important to note that if the FORMAT_IF_MOUNT_FAILED configuration is enabled, the SD card will be formatted should initialisation and mounting fail.

```
menu "BSP SD Setup"

    config BSP_SD_ENABLED
        bool "Enable SD CARD"
        default n

    if BSP_SD_ENABLED

        config FORMAT_IF_MOUNT_FAILED
            bool "Format the card if mount failed"
            default n
            help
                If this config item is set, format_if_mount_failed will be set to true and the card will be formatted if
                the mount has failed.

        config SD_GPIO_CLK
            int "SDIO CLK GPIO number"
            default 43

        config SD_GPIO_CMD
            int "SDIO CMD GPIO number"
            default 44

        config SD_GPIO_D0
            int "SDIO D0 GPIO number"
            default 39

    endif

endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_sd** driver. To successfully call functions from the **bsp_sd** folder within other functions, it is necessary to configure the **CMakeLists.txt** file located within the **bsp_sd** folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver library (the fatfs library). Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to utilise the **bsp_sd** driver functionality.

1.3.4 Microphone driver code

Here we shall focus solely on the core code; for detailed source code, please refer to the corresponding source files for this experiment within the code materials.

The microphone driver source code comprises two files: **bsp_mic.c** and **bsp_mic.h**.

Below we shall first analyse the **bsp_mic.h** programme: it defines the microphone pins and declares the relevant functions.

/* Header file references */

```
/*----------------------------------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 "driver/i2s_pdm.h" //References for I2S PDM Function-related API Functions
#ifdef CONFIG_BSP_SD_ENABLED
#include "bsp_sd.h"
#endif
/*--------------------------------Header file declaration end--------------------------------*/
```

/* Function declarations and macro definitions */

```
#ifdef CONFIG_BSP_MIC_ENABLED

#define MIC_GPIO_CLK CONFIG_MIC_GPIO_CLK           // Microphone CLK GPIO
#define MIC_GPIO_DATA CONFIG_MIC_GPIO_DATA         // Microphone DATA GPIO
#define MIC_SAMPLE_RATE CONFIG_MIC_SAMPLE_RATE     // Microphone recording sampling rate
#define BYTE_RATE (MIC_SAMPLE_RATE * (16 / 8)) * 1 // The data volume recorded by the microphone in one second

esp_err_t mic_init(); // Microphone Initialization Function
#ifdef CONFIG_BSP_SD_ENABLED
esp_err_t mic_readwav_to_sd(const char *filename, size_t rec_seconds, size_t *out_size); // Microphone recording saved to SD card
#endif
#endif
```

Next, we shall analyse the **bsp_mic.c** programme: configuring the microphone for I2S PDM mode and exposing the API interface functions.

/* Microphone initialisation function mic_init */

```c
esp_err_t mic_init()
{
    esp_err_t err = ESP_OK;
    /*I2S channel configuration for receiver (microphone)*/
    i2s_chan_config_t rx_chan_cfg = {
        .id = I2S_NUM_0,                /*Use I2S controller 0*/
        .role = I2S_ROLE_MASTER,        /*I2S acts as master (generates clock signals)*/
        .dma_desc_num = 6,              /*Number of DMA descriptors (affects buffer management)*/
        .dma_frame_num = 256,           /*Number of frames per DMA descriptor*/
        .auto_clear_after_cb = false,   /*Don't auto-clear DMA buffer after callback*/
        .auto_clear_before_cb = false,  /*Don't auto-clear DMA buffer before callback*/
        .allow_pd = false,              /*Don't allow power down during operation*/
        .intr_priority = 0,             /*Interrupt priority (0 = default)*/
    };
    /*Create new I2S channel (receive channel only, no transmit channel)*/
    err = i2s_new_channel(&rx_chan_cfg, NULL, &rx_chan);
    if (err != ESP_OK)
        return err;
    /*PDM (Pulse Density Modulation) receiver configuration*/
    i2s_pdm_rx_config_t pdm_rx_cfg = {
        /*Clock configuration*/
        .clk_cfg = {
            .sample_rate_hz = MIC_SAMPLE_RATE,        /*Sampling frequency (e.g., 16kHz, 44.1kHz)*/
            .clk_src = I2S_CLK_SRC_DEFAULT,           /*Default clock source (usually APLL or PLL)*/
            .mclk_multiple = I2S_MCLK_MULTIPLE_256,   /*Master clock multiple (256 × sample rate)*/
            .dn_sample_mode = I2S_PDM_DSR_8S,         /*Downsampling mode (8s for PDM to PCM conversion)*/
            .bclk_div = 8,                            /*Bit clock divider*/
        },
        /*Slot configuration (data format)*/
        .slot_cfg = {
            .data_bit_width = I2S_DATA_BIT_WIDTH_16BIT, /*Data bit width (16-bit for PDM)*/
            .slot_bit_width = I2S_SLOT_BIT_WIDTH_AUTO,  /*Auto-calculate slot bit width*/
            .slot_mode = I2S_SLOT_MODE_MONO,            /*Mono audio (single channel)*/
            .slot_mask = I2S_PDM_SLOT_LEFT,             /*Use left slot for mono data*/
            .hp_en = true,                              /*Enable high-pass filter*/
            .hp_cut_off_freq_hz = 35.5,                 /*High-pass filter cutoff frequency (35.5Hz)*/
            .amplify_num = 1,                           /*Amplification factor (1 = no amplification)*/
        },
        /*GPIO pin configuration*/
        .gpio_cfg = {
            .clk = MIC_GPIO_CLK,  /*GPIO pin for PDM clock signal*/
            .din = MIC_GPIO_DATA, /*GPIO pin for PDM data input*/
            .invert_flags = {
                .clk_inv = false, /*Don't invert clock polarity*/
            },
        },
    };
```

```c
    err = i2s_channel_init_pdm_rx_mode(rx_chan, &pdm_rx_cfg); /*Initialize I2S channel in PDM receiver mode*/
    if (err != ESP_OK)
        return err;
    err = i2s_channel_enable(rx_chan); /*Enable the I2S channel to start receiving audio data*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

Within the **mic_init** function, configuration begins with the **i2s_chan_config_t** structure, setting parameters for the I2S controller used by the microphone (here employing I2S controller 0). Subsequently, the i2s_new_channel function is invoked to register a receive channel on I2S controller 0. The `i2s_pdm_rx_config_t` structure configures PDM format reception. We can alter the data bit width by modifying the `data_bit_width` and `slot_bit_width` parameters within the `slot_cfg` field. Modifying the `slot_mode` and `slot_mask` within `slot_cfg` changes the microphone's reception mode to stereo, mono, etc. Finally, call the `i2s_channel_init_pdm_rx_mode` function to initialise the receive channel using PDM format, then call `i2s_channel_enable` to activate the receive channel.

Note: In this course, we are reading data in mono mode, specifically from the left channel.

/* Function `generate_wav_header` for configuring the WAV format file header */

```c
static void generate_wav_header(char *wav_header, uint32_t wav_size, uint32_t sample_rate)
{
    uint32_t file_size = wav_size + 44 - 8;
    uint32_t byte_rate = BYTE_RATE;
    const char set_wav_header[] = {
        'R', 'I', 'F', 'F',                                                 /*ChunkID*/
        file_size, file_size >> 8, file_size >> 16, file_size >> 24,        /*ChunkSize*/
        'W', 'A', 'V', 'E',                                                 /*Format*/
        'f', 'm', 't', ' ',                                                 /*Subchunk1ID*/
        0x10, 0x00, 0x00, 0x00,                                             /*Subchunk1Size (16 for PCM)*/
        0x01, 0x00,                                                         /*AudioFormat (1 for PCM)*/
        0x01, 0x00,                                                         /*NumChannels (1 channel)*/
        sample_rate, sample_rate >> 8, sample_rate >> 16, sample_rate >> 24, /*SampleRate*/
        byte_rate, byte_rate >> 8, byte_rate >> 16, byte_rate >> 24,        /*ByteRate*/
        0x02, 0x00,                                                         /*BlockAlign*/
        0x10, 0x00,                                                         /*BitsPerSample (16 bits)*/
        'd', 'a', 't', 'a',                                                 /*Subchunk2ID*/
        wav_size, wav_size >> 8, wav_size >> 16, wav_size >> 24,            /*Subchunk2Size*/
    };
    memcpy(wav_header, set_wav_header, sizeof(set_wav_header));
}
```

This function configures the WAV file header (the standard header for WAV files) by inputting the sampling rate and total data quantity.

/* Microphone recording and saving function **mic_readwav_to_sd** */

```c
esp_err_t mic_readwav_to_sd(const char *filename, size_t rec_seconds, size_t *out_size)
{
    esp_err_t err = ESP_OK;
    size_t flash_wr_size = 0; /*Total bytes written to SD card (audio data only)*/
    size_t bytes_read = 0;    /*Bytes read from I2S in each iteration*/
    if (rec_seconds > 3600 || filename == NULL || out_size == NULL)
    {
        MIC_INFO("Exceeding the maximum recording duration");
        return ESP_ERR_INVALID_ARG;
    }
    if (sizeof(i2s_readraw_buff) < 16000)
    {
        MIC_INFO("I2S buffer too small");
        return ESP_ERR_INVALID_SIZE;
    }
    size_t rec_size = rec_seconds * BYTE_RATE; /*Calculate total audio data size (excluding WAV header)*/
    char wav_header[44];
    generate_wav_header(wav_header, rec_size, MIC_SAMPLE_RATE); /*Generate WAV file header (44 bytes standard WAV header)*/
    err = write_file(filename, wav_header, 44);                 /*Write WAV header to file (creates file and writes header)*/
    if (err != ESP_OK)
        return err;
    MIC_INFO("Start Recording %d of audio data", rec_seconds);
    while (flash_wr_size < rec_size) /*Record audio data in chunks and append to file*/
    {
        size_t bytes_to_read = MIN(16000, rec_size - flash_wr_size);
        err = i2s_channel_read(rx_chan, i2s_readraw_buff, bytes_to_read, &bytes_read, portMAX_DELAY); /*Read 16000 bytes (16KB) of raw audio data from I2S microphone*/
        if (err != ESP_OK)
        {
            MIC_INFO("read mic data fail");
            return err;
        }
        err = write_file_seek(filename, i2s_readraw_buff, bytes_read, flash_wr_size + 44); /*Write audio data chunk to file at correct position (after 44-byte header)*/
        if (err != ESP_OK)
        {
            MIC_INFO("write sd of audio data fail");
            return err;
        }
        flash_wr_size += bytes_read; /*Update progress counter*/
    }
    *out_size = rec_size + 44; /*Set output parameter to total file size (audio data + WAV header)*/
    return ESP_OK;
}
```

This function has three input parameters:

filename - The filename stored on the SD card (including the file system path)

rec_seconds - The duration of the recording (in seconds; consider recording time based on SD card memory, with a maximum of 3600 seconds)

out_size - Total data size to be received from the recording

The function first validates the input parameters and checks whether the recording buffer is correctly configured (requiring a buffer size sufficient for at least 1 second of recording). It then calculates the total data required for recording based on the specified duration. A WAV file header buffer is created, configured using the `**generate_wav_header**` function, and written to the corresponding SD card file. Finally, a loop is established. If the condition that the amount of data read is less than the required amount to be read is met, the loop executes. The `**min**` function calculates how many data points to read in the current iteration. The `**i2s_channel_read**` function is called to read the audio data captured by the microphone, retrieving 16,000 data points each time. Upon successful reading, the data is written to the file (skipping the first 44 data points corresponding to the WAV header).

1.3.5 Kconfig file

The primary function of this file is to incorporate the requisite configuration into the **sdkconfig** file, enabling certain parameter adjustments to be made via a graphical interface. The GPIO configuration number here corresponds to the respective GPIO pin number. **MIC_SAMPLING_RATE** denotes the sampling rate.

```
menu "BSP MIC Setup"

    config BSP_MIC_ENABLED
        bool "Enable MIC"
        default n

    if BSP_MIC_ENABLED
        config MIC_GPIO_CLK
            int "GPIO For MIC CLK"
            default 3

        config MIC_GPIO_DATA
            int "GPIO For MIC DATA"
            default 4

        config MIC_SAMPLE_RATE
            int "PDM sample rate"
            default 16000
    endif
endmenu
```

1.3.6 CMkaLists.txt file

The functionality of this example routine relies primarily on the **bsp_mic** driver. To successfully call the contents of the **bsp_mic** folder from other functions, it is necessary to configure the **CMakeLists.txt** file within the bsp_mic folder. The configuration details are as follows:

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

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

Within this **CMakeLists.txt** file, the directories for source files and header files are first defined, along with the required driver library (**bsp_sd** library). Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the project to **utilise** the **bsp_mic** driver functionality.

1.3.7 main folder

The main folder serves as the core directory for programme execution, containing the main function executable main.c and the header file main.h within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The **main.h** file primarily references required header files: functions utilising the **bsp_sd** driver necessitate inclusion of the **bsp_sd** header file, while those employing the **bsp_mic** driver require the **bsp_mic** header file.

Below is an analysis of the main.c programme: system initialisation and initialisation of SD card functionality and microphone functionality.

```
#ifdef CONFIG_BSP_SD_ENABLED
    err = sd_init(); /*SD Initialization*/
    if (err != ESP_OK)
        init_fail("sd", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
#endif
#ifdef CONFIG_BSP_MIC_ENABLED
    err = mic_init(); /*Microphone Initialization*/
    if (err != ESP_OK)
        init_fail("mic", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
#endif
```

This code resides within the init function, which serves to store initialisation functions requiring invocation and assess their return outcomes. Should the return status deviate from **ESP_OK**, the code will output an error message and halt further execution. The 500ms delay configured for microphone initialisation serves to filter out noise generated during the I2S initialisation of the receive channel.

```
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_MIC_ENABLED
#ifdef CONFIG_BSP_SD_ENABLED
    size_t Recorded_size = 0;
    mic_readwav_to_sd("/sdcard/test.wav", 5, &Recorded_size); /*Microphone recording saved to SD card(5s)*/
    MAIN_INFO("Recorded %d bytes", Recorded_size);
#endif
#endif
}
```

Within the **app_main** function, directly invoke the **mic_readwav_to_sd** function to record 5 seconds of audio and save it to the SD card. The file name defaults to **test.wav** (this may be modified as required; should extended filenames be necessary, enable Long filename support within the SDK configuration).

1.3.8 CMkaLists.txt file

To successfully call the contents of the **bsp_sd** folder within the main function, you must configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                        INCLUDE_DIRS "include"
                        REQUIRES bsp_mic bsp_sd)
```

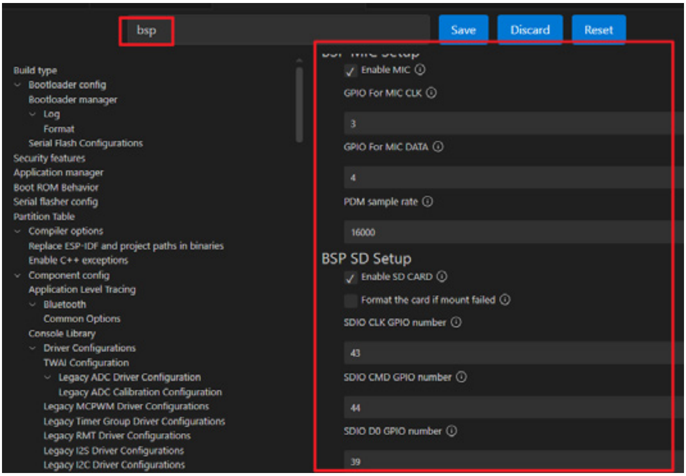
First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link **bsp_sd** and **bsp_mic**. Subsequently, these settings are registered with the build system via the **idf_component_register** command, enabling the main function to utilise these driver functionalities.

1.4 Programming procedure

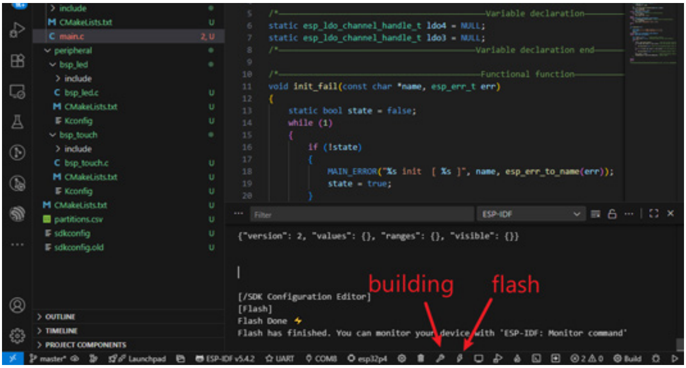
Connect the P4 device to the computer via USB

1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the led pins.



1.4.3 Click Compile. Once compilation is successful, click Download.

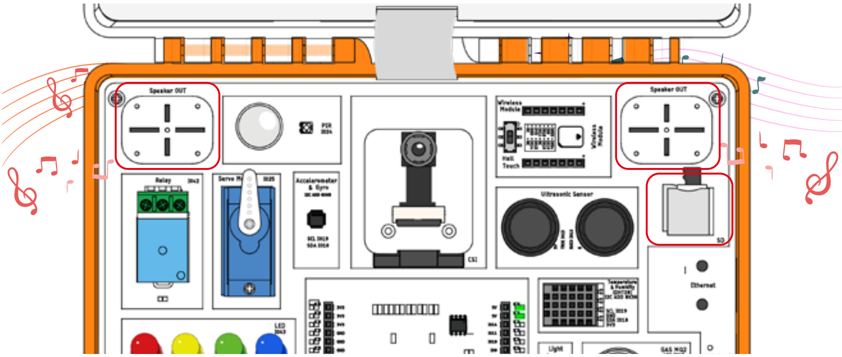


Lesson 18 – I2S Audio Playback

Introduction

This chapter's tutorial introduces the I2S audio interface application for the ESP32-P4. By playing WAV files from an SD card, it helps readers understand the principles of audio data stream transmission and the basic usage of the I2S peripheral. Audio playback is one of the most common functions for ESP32 series chips in multimedia and voice projects. By studying this chapter, readers will learn how to configure the I2S peripheral, parse WAV files, and implement sound output. This lays the foundation for more complex projects involving speech recognition, audio synthesis, or music playback.

Project Demonstration Effect



This chapter is divided into the following subsections

1.1 Introduction to I2S and WAV Files

1.1.1 I2S Introduction

I2S (Inter-IC Sound) is a serial bus interface standard for transmitting digital audio data between audio devices. The ESP32-P4 chip integrates an I2S controller supporting various audio formats, including master/slave modes, stereo/mono, and 16/24/32-bit sampling. It is widely used in scenarios such as speakers, voice assistants, recording, and Bluetooth audio.

Key features of the I2S interface include:

① Standardized Interface: Supports the I2S-STD standard data format, ensuring compatibility with common peripherals like audio decoders, DACs, and amplifier chips.

② High-Fidelity Transmission: Utilizes clock-synchronous audio data transfer to deliver high-quality audio output at 44.1kHz, 48kHz, and even higher sampling rates.

③ Hardware FIFO Buffering: Incorporates an internal FIFO to reduce CPU load, supports DMA continuous transfer, ensuring smooth, stutter-free audio playback.

④ Multi-Mode Support: Supports both Master and Slave modes, enabling flexible adaptation to different audio circuit architectures.

⑤ Programmable Configuration: Software-configurable parameters including data bit width, left/right channel timing, sample rate, and clock polarity to match diverse audio chips.

The ESP32-P4's I2S functionality provides a robust hardware foundation for audio applications. In this chapter, we will explore I2S implementation and configuration through practical playback of WAV files from an SD card.

1.1.2 Playing WAV Files Guide

WAV files are a common uncompressed audio format that stores raw audio data using

PCM (Pulse Code Modulation). Due to its simple structure and lossless audio quality, WAV is frequently used for audio playback experiments in embedded systems.

① Sampling Rate and Audio Quality

The sampling rate determines audio clarity. Common sampling rates include 8kHz (voice), 16kHz (telephony), and 44.1kHz (music). Higher sampling rates yield better sound quality but also larger data sizes.

② Playback Principle

The I2S playback process for WAV files involves these steps:

Read the WAV file header from the SD card and parse parameters;

Initialize the I2S interface, setting the matching sample rate and bit width;

Read audio data blocks in a loop and transfer them to the I2S FIFO via DMA;

The I2S peripheral automatically outputs left and right channel data, driving an external DAC or amplifier to produce sound.

1.2 Hardware Design

The audio output circuit of the ESP32-P4 development board consists of the following components:

I2S data lines:

BCLK (bit clock)

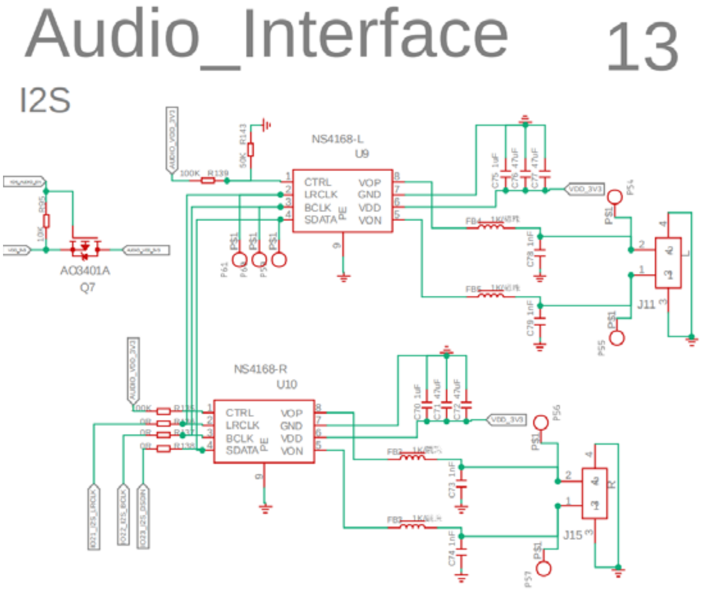
WS (left/right channel selection)

DATA (Audio Data Line)

External DAC / Audio Amplifier Module: e.g., MAX98357A or ES8388, used to convert digital signals into analog audio output;

SD Card Module: Communicates with the ESP32-P4 via SPI interface for storing WAV audio files;

Speaker or Headphone Interface: Connects to the audio output terminal.

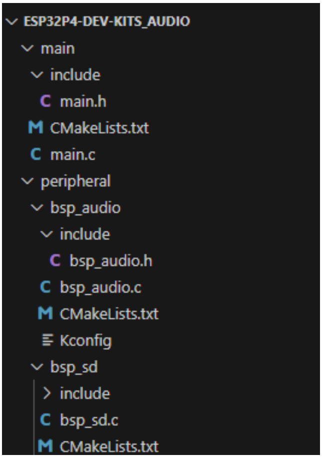


Schematic Diagram

1.3 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



In the **ESP32P4-dev-kits_audio** example, a new **bsp_audio** folder was created under the **ESP32P4-dev-kits_audio\peripheral** directory. Within the **bsp_audio** directory, an include folder, a **CMakeLists.txt** file, and a **Kconfig** file were created. The **bsp_audio** folder stores the **bsp_audio.c** driver file, the include folder **holds .h** header files, and the CMakeLists.txt file integrates the driver into the build system, enabling project access to its functionality. The Kconfig file loads the entire driver along with GPIO pin definitions into the **sdkconfig** file within the IDF platform (configurable via the graphical interface).

1.3.1 Speaker Driver Code

Here we will only explain the core code. For detailed source code, please refer to the corresponding source code for this experiment in the code materials.

The speaker driver source code consists of two files: **bsp_audio.c** and **bsp_audio.h**.

Below, we will first analyze the **bsp_audio.h** program: it defines the speaker output pins and declares the functions used.

/* Header file references */

```
/*------------------------------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 "driver/gpio.h"    //References for GPIO Function-related API Functions
#include "driver/i2s_std.h" //References for I2S STD Function-related API Functions
#ifdef CONFIG_BSP_SD_ENABLED
#include "bsp_sd.h"
#endif
/*------------------------------Header file declaration end------------------------------*/
```

/* Function declarations and macro definitions */

```
#ifdef CONFIG_BSP_AUDIO_ENABLED

#define AUDIO_GPIO_LRCLK CONFIG_AUDIO_GPIO_LRCLK // Audio LRCLK GPIO
#define AUDIO_GPIO_BCLK CONFIG_AUDIO_GPIO_BCLK   // Audio BCLK GPIO
#define AUDIO_GPIO_SDATA CONFIG_AUDIO_GPIO_SDATA // Audio DATA GPIO
#define AUDIO_GPIO_CTRL CONFIG_AUDIO_GPIO_CTRL   // Audio CTRL GPIO

esp_err_t audio_init();                 // Audio Initialization Function
esp_err_t audio_ctrl_init();            // Audio ctrl Initialization Function
esp_err_t set_Audio_ctrl(bool state); // Control mute
#ifdef CONFIG_BSP_SD_ENABLED
esp_err_t Audio_play_wav_sd(const char *fp); // The speaker plays the wav file stored on the SD card
#endif
#endif
```

Next, we'll analyze the **bsp_audio.c** program:

Initialize and configure the speaker for I2S standard mode.

Initialize and configure the NS4168 chip's channel selection pins.

Expose API interface functions.

/* I2S standard mode initialization function audio_init */

```
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*/
```

```c
    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;

    err = i2s_channel_enable(tx_chan); /*Enable the I2S channel to start audio transmission*/
    if (err != ESP_OK)
        return err;

    return err;
}
```

In the **audio_init** function, we first configure the **i2s_chan_config_t** structure to set parameters for the I2S controller used by the speaker (here we use I2S controller 1). Then, using the **i2s_new_channel** function, we register a transmit channel on I2S controller 1. The `**i2s_std_config_t**` structure configures standard-mode transmission. We can modify the data bit width by adjusting the `**data_bit_width**` and `**slot_bit_width**` parameters within the `**slot_cfg**` field. To enable stereo or mono output, modify the `**slot_mode**` and `**slot_mask**` settings within the `**slot_cfg**` field. Finally, call `**i2s_channel_init_std_mode**` to initialize the transmit channel using standard mode, then call `**i2s_channel_enable**` to activate the transmit channel.

Note: Our development board features two speakers, so we configure it for stereo mode with dual-channel output.

/* audio_ctrl_init: Initialization function for the CTRL pin of the ns4168 chip */

```c
esp_err_t audio_ctrl_init()
{
    esp_err_t err = ESP_OK;
    const gpio_config_t gpio_cofig = {
        .pin_bit_mask = 1ULL << AUDIO_GPIO_CTRL, /* GPIO pin: set with bit mask, each bit maps to a GPIO */
        .mode = GPIO_MODE_OUTPUT,                /* GPIO mode: set input/output mode                    */
        .pull_up_en = false,                     /* GPIO pull-up                                        */
        .pull_down_en = false,                   /* GPIO pull-down                                      */
        .intr_type = GPIO_INTR_DISABLE,          /* GPIO interrupt type                                 */
    };
    err = gpio_config(&gpio_cofig); /*Configure GPIO*/
    if (err != ESP_OK)
        return err;
    return err;
}
```

This function calls the GPIO initialization structure to configure the CTRL pin as an output mode.

/* Speaker mute control function **set_Audio_ctrl** */

```c
esp_err_t set_Audio_ctrl(bool state)
{
    esp_err_t err = ESP_OK;
    bool status = !state;                          /*Invert the state (the sound is on when the voltage is low, and off when the voltage is high)*/
    err = gpio_set_level(AUDIO_GPIO_CTRL, status); /*Set the Corresponding Output Level of GPIO*/
    return err;
}
```

This function calls **gpio_set_level** to output high/low level control. The status variable determines the output level. The inversion operation is due to the MOSFET used in the circuit design. A low level enables the MOSFET, while a high level controls the NS4168 chip to activate sound (dual channel). When the MOSFET is off (high level), it remains in a low state, muting the sound.

/*wav file header validation function **validate_wav_header*/**

```c
bool validate_wav_header(FILE *file)
{
    if (file == NULL)
    {
        AUDIO_ERROR("File pointer is NULL");
        return false;
    }
    long original_position = ftell(file); /*Store current file position to restore later*/
    if (original_position == -1)
    {
        AUDIO_ERROR("Cannot get current file position");
        return false;
    }
    if (fseek(file, 0, SEEK_SET) != 0) /*Rewind to beginning of file*/
    {
        AUDIO_ERROR("Cannot seek to file beginning");
        return false;
    }
    uint8_t header[44]; /*Read and validate WAV header*/
    size_t bytes_read = fread(header, 1, 44, file);
    if (bytes_read != 44)
    {
        AUDIO_ERROR("Cannot read complete WAV header (%d bytes)", bytes_read);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header, "RIFF", 4) != 0) /*Validate RIFF chunk descriptor*/
    {
        AUDIO_ERROR("Invalid RIFF header");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header + 8, "WAVE", 4) != 0) /*Validate WAVE format*/
    {
        AUDIO_ERROR("Invalid WAVE format");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header + 12, "fmt ", 4) != 0) /*Validate fmt subchunk*/
    {
        AUDIO_ERROR("Invalid fmt subchunk");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
```

```c
    uint16_t audio_format = *(uint16_t *)(header + 20); /*Check audio format (should be 1 for PCM)*/
    if (audio_format != 1)
    {
        AUDIO_ERROR("Unsupported audio format: %d (only PCM supported)", audio_format);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    uint16_t num_channels = *(uint16_t *)(header + 22); /*Check number of channels (support mono and stereo)*/
    if (num_channels != 1 && num_channels != 2)
    {
        AUDIO_ERROR("Unsupported number of channels: %d", num_channels);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    uint32_t sample_rate = *(uint32_t *)(header + 24); /*Check sample rate (support common rates)*/
    if (sample_rate != 8000 && sample_rate != 16000 && sample_rate != 22050 && sample_rate != 44100 && sample_rate != 48000)
    {
        AUDIO_ERROR("Uncommon sample rate: %lu Hz", sample_rate);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    uint16_t bits_per_sample = *(uint16_t *)(header + 34); /*Check bits per sample (support 8, 16, 24, 32)*/
    if (bits_per_sample != 8 && bits_per_sample != 16 && bits_per_sample != 24 && bits_per_sample != 32)
    {
        AUDIO_ERROR("Unsupported bits per sample: %d", bits_per_sample);
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    if (memcmp(header + 36, "data", 4) != 0) /*Validate data subchunk*/
    {
        AUDIO_ERROR("Invalid data subchunk");
        fseek(file, original_position, SEEK_SET);
        return false;
    }
    /*Get file size from RIFF header for additional validation*/
    uint32_t file_size = *(uint32_t *)(header + 4) + 8; // RIFF块大小 + 8字节头部
    uint32_t data_size = *(uint32_t *)(header + 40);

    AUDIO_INFO("WAV File Info: %d channels, %lu Hz, %d bits, %lu bytes data, %lu bytes total", num_channels, sample_rate, bits_per_sample, data_size, file_size);
    /*Restore original file position*/
    fseek(file, original_position, SEEK_SET);
    return true;
}
```

This function reads the first 44 bytes of a WAV file, checks for the standard WAV header, and prints the file information.

/* Function **Audio_play_wav_sd** to play WAV files stored on the SD card through the speaker */

```c
esp_err_t Audio_play_wav_sd(const char *filename)
{
    esp_err_t err = ESP_OK;
    if (filename == NULL)
        return ESP_ERR_INVALID_ARG;
    FILE *fh = fopen(filename, "rb");
    if (fh == NULL)
    {
        AUDIO_ERROR("Failed to open file");
        return ESP_ERR_INVALID_ARG;
    }
    if (!validate_wav_header(fh)) /*Validate WAV header*/
    {
        AUDIO_ERROR("Invalid WAV file format: %s", filename);
        fclose(fh);
        return ESP_ERR_INVALID_ARG;
    }
    if (fseek(fh, 44, SEEK_SET) != 0) /*Skip 44-byte WAV header*/
    {
        AUDIO_ERROR("Failed to seek file");
        fclose(fh);
        return ESP_FAIL;
    }
    /*Buffer configuration*/
    const size_t SAMPLES_PER_BUFFER = 512;
    const size_t INPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * sizeof(int16_t);
    const size_t OUTPUT_BUFFER_SIZE = SAMPLES_PER_BUFFER * 2 * sizeof(int16_t);
    /* Allocate buffers*/
    int16_t *input_buf = heap_caps_malloc(INPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM);
    int16_t *output_buf = heap_caps_malloc(OUTPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM);
    if (input_buf == NULL || output_buf == NULL)
    {
        AUDIO_ERROR("Failed to allocate audio buffers");
        if (input_buf)
            free(input_buf);
        if (output_buf)
            free(output_buf);
        fclose(fh);
        return ESP_ERR_NO_MEM;
    }
```

```c
    size_t samples_read = 0;
    size_t bytes_to_write = 0;
    size_t bytes_written = 0;
    size_t total_samples = 0;
    int32_t volume_data = 0;
    set_Audio_ctrl(true); /*Enable audio hardware*/
    while (1)
    {
        samples_read = fread(input_buf, sizeof(int16_t), SAMPLES_PER_BUFFER, fh);
        if (samples_read == 0)
            break;
        for (size_t i = 0; i < samples_read; i++) /*convert mono to stereo*/
        {
            volume_data = input_buf[i] * 10; /*Linear multiplication*/
            if (volume_data > 32767)
                volume_data = 32767;
            else if (volume_data < -32768)
                volume_data = -32768;
            output_buf[i * 2] = (int16_t)volume_data;     /*Left channel*/
            output_buf[i * 2 + 1] = (int16_t)volume_data; /*Right channel*/
        }
        bytes_to_write = samples_read * 2 * sizeof(int16_t);
        bytes_written = 0;
        err = i2s_channel_write(tx_chan, output_buf, bytes_to_write, &bytes_written, portMAX_DELAY); /*I2S write data*/
        if (err != ESP_OK || bytes_written != bytes_to_write)
        {
            AUDIO_ERROR("I2S write failed: %s, written: %d/%d", esp_err_to_name(err), bytes_written, bytes_to_write);
            break;
        }
        total_samples += samples_read;
    }
    /*Cleanup*/
    set_Audio_ctrl(false);
    free(input_buf);
    free(output_buf);
    fclose(fh);
    AUDIO_INFO("Audio playback completed: %d samples", total_samples);
    return err;
}
```

1. This function first calls **validate_wav_header** to verify the wav file header's validity. It then uses fseek to offset the file start position (skipping the wav header and initial 44 bytes), creating two buffers: one for storing data read from the SD card, and another for transmitting data in I2S standard mode. The **set_Audio_ctrl** function is invoked to open the audio channel (dual-channel). The `fread` function reads data from the SD card file into the buffer, which is then assigned to the transmission buffer. Since microphone recordings are mono data, and I2S standard mode transmits audio data in the format left channel, right channel, left channel, the read data (identical data) is transmitted for both channels.

2. The *10 operation applied to the read data is due to the low sound pressure recorded by the microphone. Here, a linear amplification method is used to amplify the audio data (this method also amplifies background noise; it is not used for microphone-recorded audio). Since the set audio data width is 16 bits, under linear amplification, clipping processing is required on the audio waveform to ensure the amplified data does not exceed the bit width.

3. Send the amplified data via the **`i2s_channel_write`** function. Calculate the remaining data to be read (to determine when to exit; each read operation processes 512 data points) and the total data sent (to verify if all data was transmitted during debugging; if incomplete, log an error message and exit the loop).

4. Upon transmission completion (or failure), configure the audio control pin to mute mode and release buffers and file operations (to prevent stack overflow).

1.3.2 Kconfig file

The primary function of this file is to add the required configurations to the sdkconfig file, enabling certain parameter adjustments to be made through a graphical interface. The numbers here correspond to the respective GPIO pin numbers.

```
menu "BSP AUDIO Setup"

    config BSP_AUDIO_ENABLED
        bool "Enable AUDIO"
        default n

    if BSP_AUDIO_ENABLED

        config AUDIO_GPIO_LRCLK
            int "GPIO For AUDIO LRCLK"
            default 21

        config AUDIO_GPIO_BCLK
            int "GPIO For AUDIO BCLK"
            default 22

        config AUDIO_GPIO_SDATA
            int "GPIO For AUDIO SDATA"
            default 23

        config AUDIO_GPIO_CTRL
            int "GPIO For AUDIO CTRL"
            default 6
    endif
endmenu
```

1.3.3 CMkaLists.txt file

The functionality of this example primarily relies on the bsp_audio driver. To successfully call functions from the bsp_audio folder within other functions, you must configure the CMakeLists.txt file located in the bsp_audio folder. The configuration is as follows:

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

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

In this CMakeLists.txt file, the directories for source files and header files are first defined, along with the required driver library (bsp_sd library). Then, these settings are registered into the build system using the idf_component_register command, enabling the project to utilize the bsp_audio driver functionality.

1.3.4 main folder

The main folder serves as the core directory for program execution. It contains the main function executable main.c and the main.h header file located within the include folder. Add the main folder to the **CMakeLists.txt** file of the build system.

The main.h file primarily references required header files: functions utilizing the **bsp_sd** driver require the **bsp_sd** header file, while functions using the **bsp_audio** driver require the **bsp_audio** header file.

Below is an analysis of the main.c program: System initialization and initialization for SD card functionality and speaker functionality.

```c
#ifdef CONFIG_BSP_SD_ENABLED
    err = sd_init(); /*SD Initialization*/
    if (err != ESP_OK)
        init_fail("sd", err);
    vTaskDelay(500 / portTICK_PERIOD_MS);
#endif
#ifdef CONFIG_BSP_AUDIO_ENABLED
    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);
#endif
```

This code resides within the init function, which stores initialization functions to be called and evaluates their return status. If the return status is not ESP_OK, the code prints an error message and halts execution. Here, the configuration first sets up the audio control pins and performs a mute operation. The 500ms initialization delay is implemented to filter out noise generated during I2S initialization when sending channels.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test--------");
    Init();
#ifdef CONFIG_BSP_AUDIO_ENABLED
#ifdef CONFIG_BSP_SD_ENABLED
    Audio_play_wav_sd("/sdcard/test.wav"); /*Play the WAV file stored on the SD card that was recorded by the microphone*/
#endif
#endif
}
```

In the app_main function, directly call the Audio_play_wav_sd function to play the WAV file stored on the SD card. The filename is test.wav (can be modified as needed; if long filenames are required, enable long filename support in sdkconfig).

1.3.5 CMkaLists.txt file

To successfully call the contents of the bsp_sd folder and bsp_audio folder within the main function, you must configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

```
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                    INCLUDE_DIRS "include"
                    REQUIRES bsp_audio bsp_sd)
```

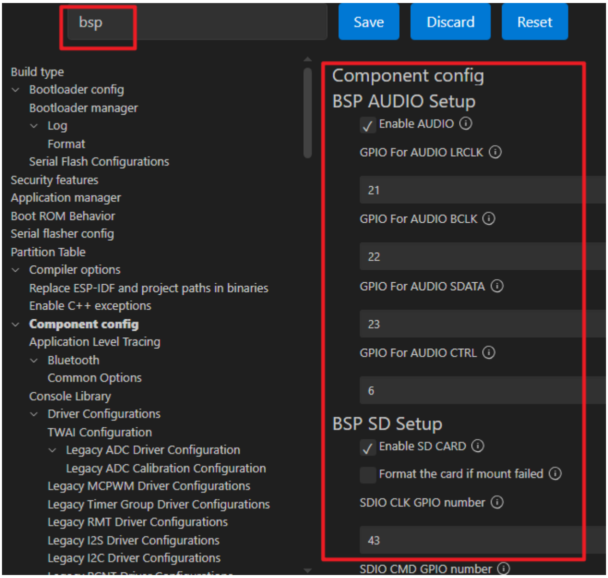
First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link **bsp_sd** and **bsp_audio**. Then, these settings are registered with the build system using the **idf_component_register** command, enabling the main function to utilize these driver features.

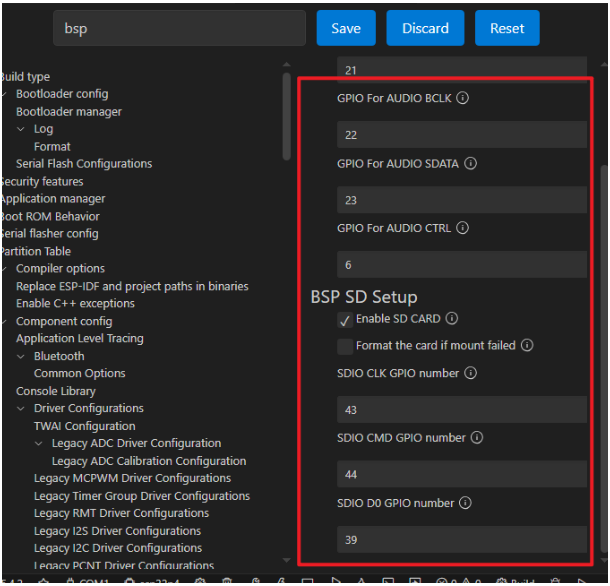
1.4 Programming procedure

Connect the P4 device to the computer via USB

1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the led pins.





1.4.3 Click Compile. Once compilation is successful, click Download.

Lesson 19 - LVGL Touch LED Control

Introduction

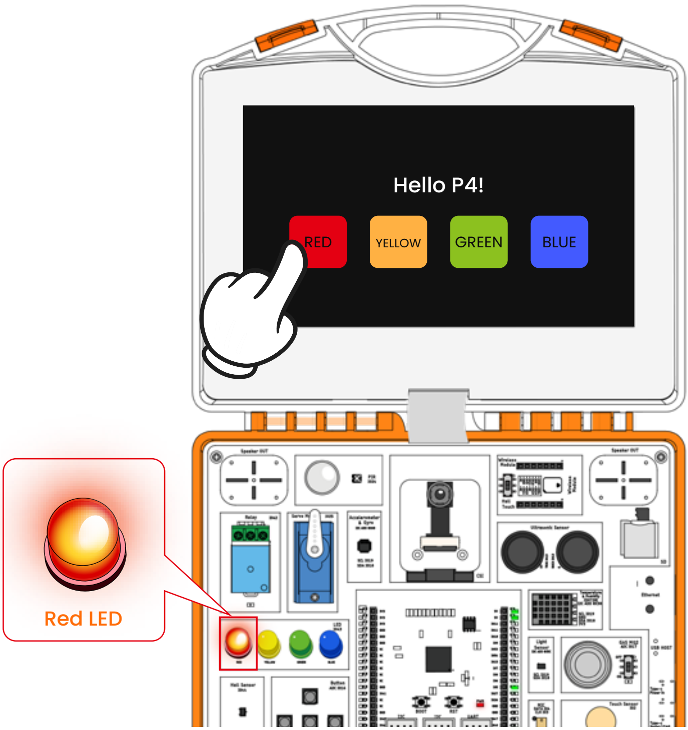
Building upon the previous chapters “GPIO Output Control of LED Lights” and “LVGL Display,” this chapter further expands the control application scenarios for the P4 display. Through this chapter, readers will learn how to implement human-machine interaction control on the ESP32-P4 development board using the LVGL graphical interface library:

Control the on/off state of LED lights by tapping buttons on the touchscreen;

Master LVGL's basic controls and event response mechanisms;

Understand the combined application of GPIO and GUI.

Project Demonstration Effect



This project integrates graphical interface development with underlying hardware control, representing a significant advancement from “turning on a light” to “touch-based smart control.” It lays the foundation for subsequent smart home and human-computer interaction projects.

1.1Project Objectives

Understand the fundamental components and operational mechanisms of the LVGL graphical interface framework;

Master the usage of button controls (lv_btn) and label controls (lv_label);

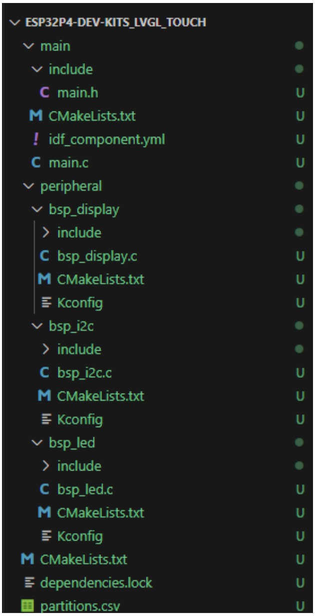
Learn to control GPIO outputs via touch events to implement LED switching;

Master the design logic of event callback functions and methods for synchronized status display.

1.2 Programme Analysis

https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design

Open the project file in VS Code as per the previous instructions.



The bsp_display folder has been added to the **ESP32P4-dev-kits_lvgl_touch** example. Modifications were made to the **bsp_display.c** file located in the **bsp_display** directory, along with updates to the **CMakeLists.txt** file and Kconfig files. New code related to screen touch functionality has been introduced.

1.2.1 Screen Touch Driver Code

Here we will only explain the core code. For detailed source code, please refer to the corresponding source code for this experiment in the code materials.

The modifications to the screen touch driver source code involve two files: bsp_display.c and bsp_display.h.

Below, we will first analyze the bsp_display.h program: Add relevant definitions and function declarations for the screen touch pins.

/* Header file references */

```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 "esp_lcd_touch_gt911.h" //References for Touch GT911 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
#ifdef CONFIG_BSP_I2C_ENABLED
#include "bsp_i2c.h"
#endif
/*---------------------------------Header file declaration end---------------------------------*/
```

/* Function declarations and macro definitions */

```c
#ifdef CONFIG_BSP_TOUCH_ENABLED

#define Touch_GPIO_RST CONFIG_TOUCH_GPIO_RST // TOUCH Reset GPIO
#define Touch_GPIO_INT CONFIG_TOUCH_GPIO_INT // TOUCH INT GPIO
esp_err_t touch_init(void);                  // Touch Initialization Function

#endif
```

Additional content regarding screen touch functionality has been added to the existing framework.

Next, we will analyze the bsp_display.c program: New initialization configuration and setting function calls for the display's touch pins have been implemented.

/* Screen touch initialization function touch_init */

```c
esp_err_t touch_init(void)
{
    esp_err_t err = ESP_OK;
    esp_lcd_panel_io_i2c_config_t io_config = {
        .dev_addr = ESP_LCD_TOUCH_IO_I2C_GT911_ADDRESS_BACKUP, /*Primary I2C device address attempt*/
        .control_phase_bytes = 1,                              /*1 byte for control phase*/
        .dc_bit_offset = 0,                                    /*No DC bit offset (not used for touch)*/
        .lcd_cmd_bits = 16,                                    /*16-bit command width*/
        .flags = {
            .disable_control_phase = 1, /*Disable control phase for touch controllers*/
        },
        .scl_speed_hz = 400000, /*I2C clock speed: 400kHz (Fast Mode)*/
    }; /*I2C configuration for touch panel controller (GT911)*/
    esp_lcd_touch_config_t tp_cfg = {
        .x_max = H_size,                  /*Maximum X coordinate (horizontal resolution)*/
        .y_max = V_size,                  /*Maximum Y coordinate (vertical resolution)*/
        .rst_gpio_num = Touch_GPIO_RST, /*Reset GPIO pin number*/
        .int_gpio_num = Touch_GPIO_INT, /*Interrupt GPIO pin number*/
        .levels = {
            .reset = 0,     /*Active low reset level*/
            .interrupt = 0, /*Active low interrupt level*/
        },
        .flags = {
            .swap_xy = false,  /*Don't swap X and Y coordinates*/
            .mirror_x = false, /*Don't mirror X axis*/
            .mirror_y = false, /*Don't mirror Y axis*/
        },
    }; /*Touch panel configuration parameters*/
    /*Initialize I2C communication with touch controller*/
    err = esp_lcd_new_panel_io_i2c((i2c_master_bus_handle_t)i2c_bus_handle, &io_config, &tp_io_handle);
    if (err != ESP_OK)
        return err;
    err = esp_lcd_touch_new_i2c_gt911(tp_io_handle, &tp_cfg, &tp); /*Initialize GT911 touch controller with primary address*/
    if (err != ESP_OK)                                             /*If primary address fails, try backup I2C address*/
    {
        io_config.dev_addr = ESP_LCD_TOUCH_IO_I2C_GT911_ADDRESS; /*Backup I2C device address*/
        err = esp_lcd_new_panel_io_i2c((i2c_master_bus_handle_t)i2c_bus_handle, &io_config, &tp_io_handle);
        if (err != ESP_OK)
            return err;

        err = esp_lcd_touch_new_i2c_gt911(tp_io_handle, &tp_cfg, &tp); /*Initialize GT911 touch controller with backup address*/
        if (err != ESP_OK)
            return err;
    }
    return err;
}
```

Within the `touch_init` function, parameter configuration is first performed for each member variable of the `esp_lcd_panel_io_i2c_config_t` structure. This includes configuring parameters such as the I²C 7-bit address and command bit width for the GT911 touch control chip. Subsequently, the `esp_lcd_touch_config_t` structure is invoked for configuration. This involves defining parameters related to screen resolution, reset pins, interrupt pins, and their activation signals. The esp_lcd_new_panel_io_i2c function is then invoked to assign the handle obtained earlier from the I2C driver to the touch screen. Finally, the esp_lcd_touch_new_i2c_gt911 function initializes the GT911 touch chip configuration and returns the control handle.

Note: When the current GT911 address configuration fails, we switch to an alternative address for reconfiguration (GT911 has two 7-bit addresses determined by the INT pin level at power-up).

/* Added touch control to lvgl_init function */

```
#ifdef CONFIG_BSP_TOUCH_ENABLED
    const lvgl_port_touch_cfg_t touch_cfg = {
        .disp = my_lvgl_disp,
        .handle = tp, /*LCD touch IO handle*/
    };
    my_touch_indev = lvgl_port_add_touch(&touch_cfg); /*Add LCD touch as an input device*/
    if (my_touch_indev == NULL)
    {
        err = ESP_FAIL;
        DISPLAY_ERROR("LVGL touch port add fail");
    }
#endif
    return err;
```

The code added at the end of this function configures the IO handle obtained during screen touch initialization with the previously configured lvgl display handle. It then uses the lvgl_port_add_touch function to add touch functionality to the lvgl.

1.2.2 Kconfig file

The primary function of this file is to add the required configurations to the sdkconfig file, enabling certain parameter adjustments to be made through a graphical interface. The newly added touch-related configuration parameters define the interrupt pin and reset pin.

```
config BSP_TOUCH_ENABLED
bool "Enable touch functions"
depends on BSP_I2C_ENABLED
```

```
if BSP_TOUCH_ENABLED
    config TOUCH_GPIO_RST
        int "GPIO For Touch RST"
        default 40

    config TOUCH_GPIO_INT
        int "GPIO For Touch INT"
        default 41
endif
```

1.2.3 CMkaLists.txt file

The functionality of this example primarily relies on the bsp_display driver. To successfully call the contents of the bsp_display folder within the main function, you must configure the CMakeLists.txt file located in the bsp_display folder. The configuration details are as follows:

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

idf_component_register(SRCS ${component_sources}
                        INCLUDE_DIRS "include"
                        REQUIRES driver esp_lcd_ek79007 esp_lcd_touch_gt911 lvgl esp_lvgl_port bsp_i2c)
```

In this CMakeLists.txt file, we first define the directories for source files and header files, along with the required driver libraries (the driver library for the display driver chip ek79007, the lvgl driver library, the newly added driver library for the touch driver chip gt911, and our bsp_i2c driver library). Then, using the idf_component_register command, we register these settings with the build system so the project can utilize the bsp_display driver functionality.

1.2.4 main folder

The main folder serves as the core directory for program execution. It contains the main function executable main.c and the main.h header file located within the include folder. Add the main folder to the CMakeLists.txt file of the build system.

The main.h file primarily references required header files: functions utilizing the bsp_display driver require the bsp_display header file, while those using the bsp_led driver require the bsp_led header file.

Below is an analysis of the main.c program: System initialization and execution of functions for display, touch, and LED capabilities.

```
#ifdef CONFIG_BSP_LED_ENABLED
    err = led_init(); /*RMT LED Initialization*/
    if (err != ESP_OK)
        init_fail("led", err);
    vTaskDelay(200 / portTICK_PERIOD_MS);
    set_led_status(0x00000000); /*All the LEDs are off*/
#endif
#ifdef CONFIG_BSP_DISPLAY_ENABLED
#if defined(CONFIG_BSP_TOUCH_ENABLED) && defined(CONFIG_BSP_I2C_ENABLED)
    err = touch_init(); /*touch Initialization*/
    if (err != ESP_OK)
        init_fail("display touch", err);
#endif
    err = display_init(); /*Display Initialization*/
    if (err != ESP_OK)
        init_fail("display", err);
#endif
}
```

This code resides within the init function, which stores initialization functions to be called and evaluates their return values. If the return status is not ESP_OK, the code prints an error message and halts execution.

/* Screen initialization and display function display_test */

```c
void display_test()
{
    if (lvgl_port_lock(0))
    {
        test_screen = lv_obj_create(NULL);
        lv_obj_set_style_bg_color(test_screen, LV_COLOR_BLACK, LV_PART_MAIN); /*Set the screen's LVGL background color*/
        lv_obj_set_style_bg_opa(test_screen, LV_OPA_COVER, LV_PART_MAIN);     /*Set the screen's LVGL background transparency*/
        lv_obj_t *label = lv_label_create(test_screen); /*Create a label object*/
        lv_label_set_text(label, "Hello P4!");          /*Set a new text for a label. Memory will be allocated to store the text by the label.*/
        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(label, &label_style, LV_PART_MAIN);                 /*Add a style to an object*/
        lv_obj_align(label, LV_ALIGN_CENTER, 0, -100);                       /*Change the alignment of an object*/
        lv_obj_set_style_text_color(label, LV_COLOR_WHITE, LV_PART_MAIN);    /*Set the style LVGL text color*/
        lv_obj_set_style_text_font(label, &lv_font_montserrat_42, LV_PART_MAIN); /*Set the style LVGL text font*/
        lv_obj_t *buttons[4];
        lv_color_t btn_color[4] = {LV_COLOR_RED, LV_COLOR_YELLOW, LV_COLOR_GREEN, LV_COLOR_BLUE};
        const char *button_texts[] = {"RED", "YELLOW", "GREEN", "BLUE"};
        for (int i = 0; i < 4; i++)
        {
            buttons[i] = lv_btn_create(test_screen);                     /*Create button object*/
            lv_obj_set_size(buttons[i], 100, 100);                       /*Set the size of an object*/
            lv_obj_set_pos(buttons[i], (162 + (i * 200)), 300);          /*Set the position of an object relative to the set alignment*/
            lv_obj_set_style_radius(buttons[i], 15, LV_STATE_DEFAULT);   /*Set rounded corners for display*/
            lv_obj_t *label = lv_label_create(buttons[i]);                          /*Create a label object*/
            lv_label_set_text(label, button_texts[i]);                             /*Set a new text for a label. Memory will be allocated to store the text by the label.*/
            lv_obj_set_style_text_font(label, &lv_font_montserrat_20, LV_PART_MAIN); /*Set the style LVGL text font*/
            lv_obj_center(label);                                                  /*Align an object to the center on its parent*/
            lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_DEFAULT);        /*Set the style LVGL background color*/
            lv_obj_set_style_text_color(buttons[i], LV_COLOR_BLACK, LV_STATE_DEFAULT);    /*Set the style LVGL text color*/
            lv_obj_set_style_border_width(buttons[i], 2, LV_STATE_DEFAULT);               /*Set the border width of the object*/
            lv_obj_set_style_border_color(buttons[i], LV_COLOR_BLACK, LV_STATE_DEFAULT);  /*Set the border color of the object*/
            lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_PRESSED);       /*Set the style LVGL background color - Pressed State*/
            lv_obj_set_style_text_color(buttons[i], LV_COLOR_BLACK, LV_STATE_PRESSED);   /*Set the style LVGL text color - Pressed State*/
            lv_obj_set_style_border_color(buttons[i], LV_COLOR_GRAY, LV_STATE_PRESSED);  /*Set the border color of the object - Pressed State*/

            lv_obj_add_event_cb(buttons[i], button_event_handler, LV_EVENT_ALL, (void *)(intptr_t)i); /*Add event handling and pass the LED index as user data*/
        }
        lv_scr_load(test_screen); /*Set the current active screen*/
        lvgl_port_unlock();
    }
}
```

This function primarily configures the initial screen display content: sets background color and text display via lvgl controls. (Modifies the position of the original “Hello P4” display, adds four button controls, and configures them accordingly)

lv_label_set_text function sets the text content displayed by the control

lv_style_set_bg_opa function sets the background color of the style

lv_obj_set_style_text_color function sets the text display color lv_obj_set_style_text_font function sets the text font size

lv_obj_set_style_bg_color function sets the background color

lv_obj_set_style_bg_opa function sets background transparency

lv_obj_align function sets control alignment

lv_btn_create function creates button controls (with pressed, released, clicked effects)

lv_obj_set_size function sets control size

lv_obj_set_pos function sets control x/y coordinates

lv_obj_set_style_radius function sets the control's display radius

lv_obj_set_style_border_width function sets the pixel width of the control's border

lv_obj_set_style_border_color function sets the control's border color

Here we use a for loop to uniformly configure the display of buttons, including text, color, border, position, etc. Refer to the code comments for specific effects. The final key step is adding a callback function to handle Lvgl key events. (The last parameter passed during registration is the callback function's argument; here we pass the key index for subsequent LED control.)

Note: When calling lvgl functions outside the lvgl thread function, a mutex lock must be acquired. Use lvgl_port_lock to acquire the lock and lvgl_port_unlock to release it.

/* Button event callback handler function button_event_handler */

```c
static void button_event_handler(lv_event_t *e)
{
    lv_event_code_t code = lv_event_get_code(e);              /*Get the event code of an event*/
    lv_obj_t *btn = lv_event_get_target(e);                   /*Get the object originally targeted by the event. It's the same even if the event is bubbled*/
    int led_index = (int)(intptr_t)lv_event_get_user_data(e); /*Get the user_data passed when the event was registered on the object*/

    if (code == LV_EVENT_PRESSED)
    {
        lv_obj_add_state(btn, LV_STATE_PRESSED); /*Add one or more states to the object*/
        set_single_led_status(led_index, 0xFF);  /*Light up according to the color corresponding to the key press*/
        MAIN_INFO("BTN[%d] pressed", led_index);
    }
    else if (code == LV_EVENT_RELEASED)
    {
        lv_obj_clear_state(btn, LV_STATE_PRESSED); /*Add one or more states to the object*/
        set_single_led_status(led_index, 0x00);
        MAIN_INFO("BTN[%d] released", led_index);
    }
}
```

Here, we retrieve the lv_event_t data obtained from the callback. We use the lv_event_get_code function to determine the event type, the lv_event_get_target function to identify which key was pressed, and the lv_event_get_user_data function to obtain the corresponding input parameter configured for that key.

Based on the event, we determine whether it's a press event. If so, we set the key to a pressed state and control the corresponding LED to light up according to the input parameter. During a release event, we clear the key's pressed state and control the corresponding LED to turn off based on the input parameter.

```c
void app_main(void)
{
    MAIN_INFO("----------Demo version----------");
    MAIN_INFO("----------Start the test----------");
    Init();
#ifdef CONFIG_BSP_DISPLAY_ENABLED
    set_lcd_blight(100);    /*Set the screen backlight to maximum brightness*/
    if (lvgl_port_lock(0)) /*Take LVGL mutex*/
    {
        display_test();     /*Set the screen's LVGL default display page*/
        lvgl_port_unlock(); /*Give LVGL mutex*/
    }
#endif
}
```

In the app_main function, first enable the backlight with brightness set to 100%, then initialize the screen display content. All touch button operations are executed within callback functions.

1.3.5 CMkaLists.txt file

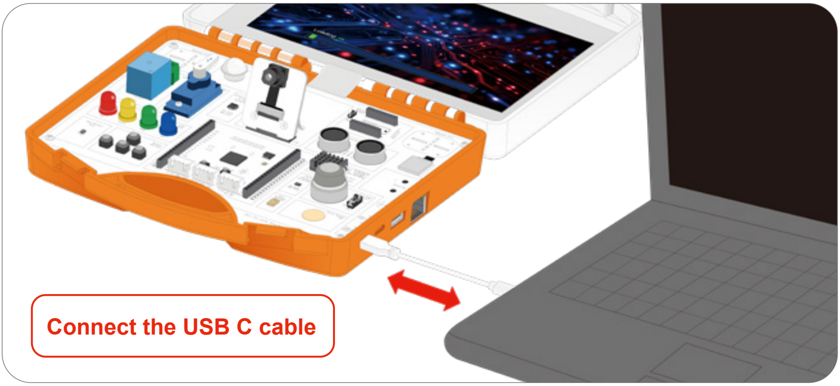
To successfully call the contents of the bsp_display folder within the main function, you must configure the CMakeLists.txt file located in the main folder. The configuration details are as follows:

```cmake
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_display bsp_led)
```

First, the directories for source files and header files are defined, along with the required driver libraries—specifically, the driver libraries needed to link bsp_display and bsp_led (bsp_i2c is already linked in the bsp_display folder). Then, these settings are registered with the build system using the idf_component_register command, enabling main to utilize these driver functions.

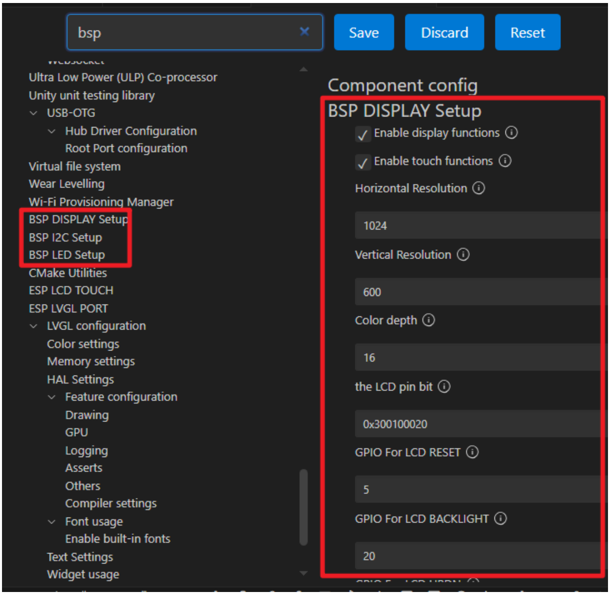
1.4 Programming procedure

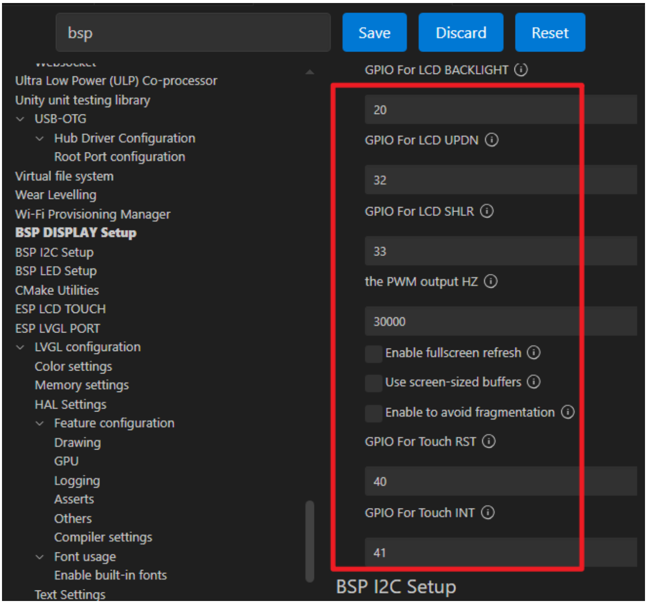
Connect the P4 device to the computer via USB

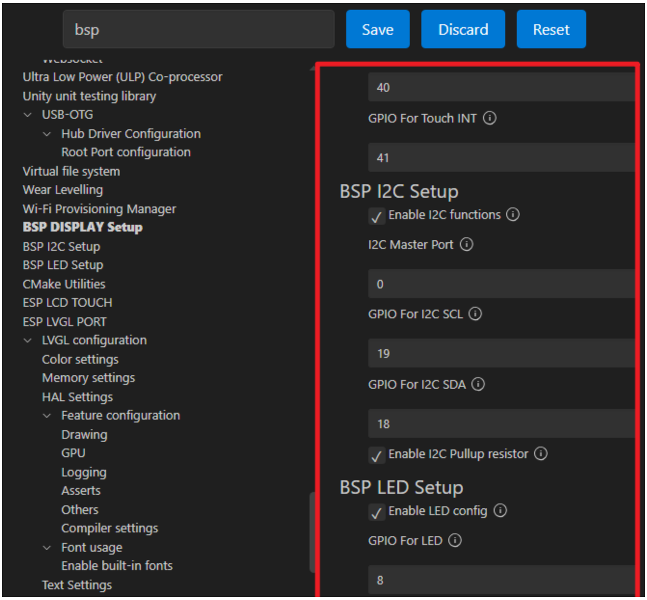


1.4.1 After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the IDF environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

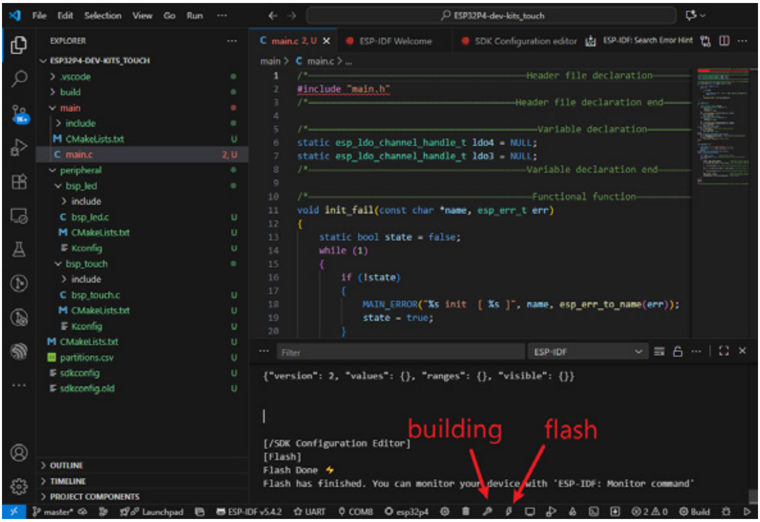
1.4.2 The subsequent SDKConfig configuration is largely identical to Lesson 1, simply reconfigure the led pins.




bsp
Save
Discard
Reset
Ultra Low Power (ULP) Co-processor
Unity unit testing library
USB-OTG
Hub Driver Configuration
Root Port configuration
Virtual file system
Wear Levelling
Wi-Fi Provisioning Manager
BSP DISPLAY Setup
BSP I2C Setup
BSP LED Setup
CMake Utilities
ESP LCD TOUCH
ESP LVGL PORT
LVGL configuration
Color settings
Memory settings
HAL Settings
Feature configuration
Drawing
GPU
Logging
Asserts
Others
Compiler settings
Font usage
Enable built-in fonts
Text Settings
GPIO For LCD BACKLIGHT
20
GPIO For LCD UPDN
32
GPIO For LCD SHLR
33
the PWM output HZ
30000
Enable fullscreen refresh
Use screen-sized buffers
Enable to avoid fragmentation
GPIO For Touch RST
40
GPIO For Touch INT
41
BSP I2C Setup


bsp
Save
Discard
Reset
Ultra Low Power (ULP) Co-processor
Unity unit testing library
USB-OTG
Hub Driver Configuration
Root Port configuration
Virtual file system
Wear Levelling
Wi-Fi Provisioning Manager
BSP DISPLAY Setup
BSP I2C Setup
BSP LED Setup
CMake Utilities
ESP LCD TOUCH
ESP LVGL PORT
LVGL configuration
Color settings
Memory settings
HAL Settings
Feature configuration
Drawing
GPU
Logging
Asserts
Others
Compiler settings
Font usage
Enable built-in fonts
Text Settings
40
GPIO For Touch INT
41
BSP I2C Setup
Enable I2C functions
I2C Master Port
0
GPIO For I2C SCL
19
GPIO For I2C SDA
18
Enable I2C Pullup resistor
BSP LED Setup
Enable LED config
GPIO For LED
8

1.4.3 Click Compile. Once compilation is successful, click Download.



Lesson 20 – Getting Started with ESP32 P4 Camera

Introduction

This tutorial introduces the application of the MIPI camera interface on the ESP32-P4 platform. By driving the SC2336 camera module, it enables real-time image capture and display, helping readers understand the data transmission principles of MIPI CSI cameras and the basic methods for using camera drivers.

Camera functionality is a crucial component of multimedia applications on the ESP32-P4 and is widely used in scenarios such as smart vision, AI recognition, video surveillance, face detection, QR code recognition, and machine vision. By studying this chapter, readers will master core concepts including MIPI camera interface initialization, image data acquisition, camera parameter configuration, and real-time image display, laying the foundation for subsequent complex projects such as AI image recognition, video processing, and edge vision computing.

Project Demonstration Effect

Capture images in real time using the SC2336 camera and display the camera feed on the screen in real time.

This chapter is divided into the following subsections

1.1 Introduction to the SC2336 Camera and MIPI Interface

1.1.1 Introduction to the MIPI Camera Interface

MIPI (Mobile Industry Processor Interface) is a high-speed serial interface standard widely used for communication between cameras and displays in mobile devices. The camera component typically employs the MIPI CSI (Camera Serial Interface) protocol.

The ESP32-P4 integrates a MIPI CSI camera controller, enabling direct connection to high-performance image sensors for high-speed image capture and transmission.

The MIPI camera interface offers the following features:

① High-speed data transmission: Compared to traditional DVP parallel cameras, MIPI CSI uses high-speed differential signal transmission, supporting higher resolutions and frame rates.

② Fewer pins: MIPI uses differential signal transmission, requiring only a small number of signal lines to achieve high-speed image transmission, which simplifies PCB layout design.

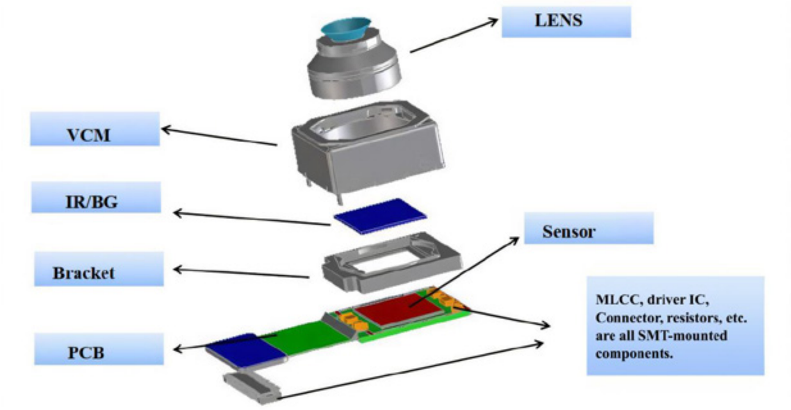
③ Low power consumption: The MIPI interface is specifically optimized for mobile devices and features low power consumption.

④ High-resolution support: It supports image capture at 720p, 1080p, and even higher resolutions.

⑤ Strong interference resistance: Differential signals offer strong resistance to interference, making them suitable for high-speed image data transmission.

The MIPI CSI functionality of the ESP32-P4 provides a robust hardware foundation for AI vision and multimedia applications. In this section, we will use the SC2336 camera module to learn about MIPI camera configuration and the image acquisition process.

1.1.2 SC2336 Camera Overview



The SC2336 is a 2-megapixel CMOS image sensor that supports a maximum output resolution of 1920×1080 and offers excellent low-light performance, making it suitable for applications such as smart vision, security surveillance, and AI image recognition.

The main features of the SC2336 camera are as follows:

① 2-megapixel image capture, supporting Full HD (1920×1080) resolution.

② 100° wide-angle lens, offering a wide field of view, suitable for visual recognition and environmental monitoring.

③ Supports MIPI CSI interface, utilizing high-speed MIPI data transmission.

④ Excellent low-light performance, delivering good image quality even in dimly lit environments.

⑤ Supports multiple output formats, including RGB and YUV.

1.1.3 Key Points of Camera Image Acquisition Principles

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:

· Be responsible for initializing the **SCCB (I2C)** communication bus of the camera and the **CSI** hardware interface of ESP32, and complete the startup and configuration of the camera's low-level driver.

· The camera is opened and configured based on the standard V4L2 protocol, automatically set to the **RGB565** format compatible with LVGL, and supports functions such as image flipping.

· Manage multi-frame image cache, support double buffering mechanism, automatically allocate aligned memory space, and complete the enqueue, dequeue and cyclic usage of the cache.

· Provide a frame data callback interface. Every time a frame of image is collected, it will automatically notify the upper layer, achieving the separation of video acquisition from business logic.

· Create an independent video acquisition task. Through event groups, safely control the start, stop and display switch of the acquisition process, ensuring the stable operation of the system.

· Directly connect to the LVGL canvas to achieve real-time display of the camera image. It adopts zero-copy refreshing and supports interface display/hide switching.

· Provide simple and user-friendly APIs externally, allowing all functions such as camera initialization, startup, and display control to be completed with just a simple call.

1.2 Program Analysis

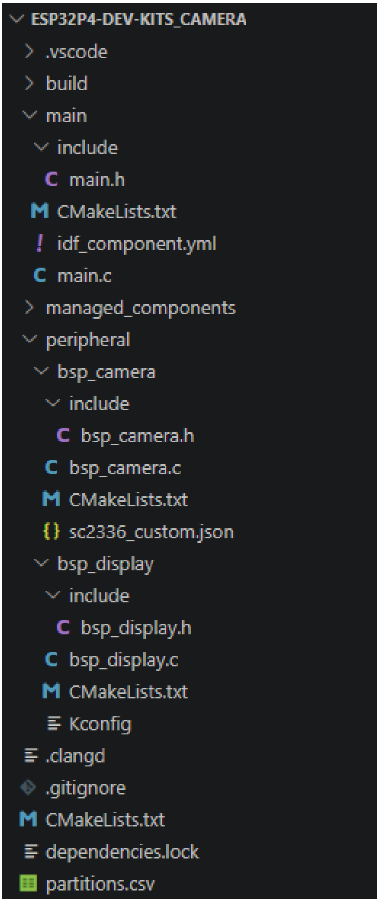
You just need to know when to call the interfaces we have written.

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/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/V1.1/idf-code/20ESP32P4-dev-kits_camera/ESP32P4-dev-kits_camera

Open the project file in VS Code as per the previous instructions.



The **bsp_display** and **bsp_camera** folders have been added to the **ESP32P4-dev-kits_CAMERA** example. The **bsp_display.c** file in the **bsp_display** directory has been modified, and the **CMakeLists.txt** and **Kconfig** files have been updated. There is also camera-related code in the **bsp_camera** folder.

1.2.1 Camera Driver Code

This section covers only the core code. For the detailed source code, please refer to the source code for this experiment in the code materials.

Here, we will focus solely on the camera; for information on the screen display, please refer to previous lessons. Modifications to the camera driver source code involve two files: **bsp_camera.c** and **bsp_camera.h**.

Next, we will first analyze the **bsp_camera.h** file: adding the relevant definitions and function declarations for the screen display.

/* Header file inclusions */

```
peripheral > bsp_camera > include > C bsp_camera.h
#ifndef _BSP_CAMERA_H_
#define _BSP_CAMERA_H_
/*-----------------------------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 "esp_sccb_intf.h"
#include "esp_sccb_i2c.h"
#include "esp_cam_sensor.h"
#include "esp_cam_sensor_detect.h"
#include "linux/videodev2.h"
#include "esp_video_device.h"
#include "esp_video_init.h"
#include "esp_video_ioctl.h"
#include <sys/errno.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/param.h>
#include <fcntl.h>
#include "esp_cache.h"
#include "esp_heap_caps.h"
#include "esp_private/esp_cache_private.h"
#include "bsp_display.h"
/*-----------------------------Header file declaration end-----------------------------*/
```

/*macro definitions */

```
/*-----------------------------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     46
#define SCCB_GPIO_SDA     45
```

All camera-related content has been added to the existing framework.

Next, we will analyze the **bsp_camera.c** program: the initialization and configuration functions for the camera have been implemented.

/* camera initialization function **camera_video_init** */

```c
/*——————————————————Variable declaration end——————————————————*/

/*——————————————————Functional function——————————————————*/
esp_err_t camera_video_init()
{
    esp_err_t err = ESP_OK;
    i2c_master_bus_config_t sccb_conf = {
        .i2c_port = SCCB_MASTER_PORT,
        .sda_io_num = SCCB_GPIO_SDA,
        .scl_io_num = SCCB_GPIO_SCL,
        .clk_source = I2C_CLK_SRC_DEFAULT,
        .flags.enable_internal_pullup = true,
    };
    CAMERA_INFO("Initializing SCCB Bus.......");
    err = i2c_new_master_bus(&sccb_conf, &sccb_bus_handle);
    esp_video_init_csi_config_t csi_config = {
        .sccb_config = {
            .init_sccb = true,
            .i2c_config = {
                .port = SCCB_MASTER_PORT,
                .scl_pin = SCCB_GPIO_SCL,
                .sda_pin = SCCB_GPIO_SDA,
            },
            .freq = 100000,
        },
        .reset_pin = -1,
        .pwdn_pin = -1,
    };
    csi_config.sccb_config.init_sccb = false;
    csi_config.sccb_config.i2c_handle = sccb_bus_handle;
```

In the `**camera_video_init**` function, the SCCB (Camera Control Bus) is first configured using the `**i2c_master_bus_config_t**` structure. This primarily configures the I2C port number, SDA and SCL pins, clock source, and internal pull-up functionality. Essentially, the SCCB is a camera register configuration interface implemented based on the I2C protocol, primarily used for register communication with the SC2336 camera. Subsequently, the `**i2c_new_master_bus**` function is called to initialize the SCCB master bus and obtain the corresponding I2C bus handle.

Next, the program initializes and configures the MIPI CSI camera interface using the `**esp_video_init_csi_config_t**` structure. The `**sccb_config**` member is used to configure the camera's SCCB communication parameters, including whether to initialize SCCB, the I2C port number, the SDA/SCL pins, and the communication frequency. Here, the communication frequency is set to **100 kHz**.

The program then sets `**init_sccb**` to `**false**`, indicating that the subsequent video driver will not reinitialize the SCCB bus but will instead directly use the previously created I2C bus handle. Next, `**i2c_handle**` is set to point to the previously created `**sccb_bus_handle**`, thereby binding the camera driver to the SCCB bus.

Afterward, the video initialization parameters are further configured via the `**esp_video_init_config_t**` structure, and the CSI configuration structure is passed to the `**.csi**` member.

Finally, the `**esp_video_init**` function is called to complete the entire MIPI CSI camera driver initialization, including:

Camera control interface initialization, MIPI CSI interface initialization, video driver registration, and camera device binding

If initialization fails, an error message is returned; if initialization succeeds, **ESP_OK** is returned.

Note:

Here, **reset_pin** and **pwdn_pin** are set to -1, indicating that the current hardware design does not use separate camera reset and power control pins; therefore, the camera boots up by default upon power-on.

/* Added **video_open** function */

```
int video_open()
{
    struct v4l2_format camera_format;
    struct v4l2_capability capability;

#if CONFIG_ENABLE_CAM_SENSOR_PIC_VFLIP || CONFIG_ENABLE_CAM_SENSOR_PIC_HFLIP
    struct v4l2_ext_controls controls;
    struct v4l2_ext_control control[1];
#endif

    int fd = open(ESP_VIDEO_MIPI_CSI_DEVICE_NAME, O_RDONLY | O_NONBLOCK, 0);
    if (fd < 0)
    {
        CAMERA_ERROR("Open video failed");
        return -1;
    }
    if (ioctl(fd, VIDIOC_QUERYCAP, &capability))
    {
        CAMERA_ERROR("failed to get capability");
        goto exit_0;
    }
    CAMERA_INFO("version: %d.%d.%d", (uint16_t)(capability.version >> 16), (uint8_t)(capability.version >
    CAMERA_INFO("driver:  %s", capability.driver);
    CAMERA_INFO("card:    %s", capability.card);
    CAMERA_INFO("bus:     %s", capability.bus_info);
    memset(&camera_format, 0, sizeof(struct v4l2_format));
    camera_format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    if (ioctl(fd, VIDIOC_G_FMT, &camera_format) != 0)
```

The primary purpose of the `**video_open**` function is to open the MIPI camera device and complete the initial configuration of the video format, preparing for subsequent image capture.

At the beginning of the function, the `**v4l2_format**` and `**v4l2_capability**` structures are first defined to retrieve the camera format and device information. The MIPI CSI camera device is then opened using the `**open**` function; if the open operation fails, an error is returned.

Next, `VIDIOC_QUERYCAP` is called to retrieve camera driver information, including the driver version, device name, and bus information.

Then, `**VIDIOC_G_FMT**` is used to obtain the current camera image format and read the image width and height parameters.

The program checks whether the current output format is `**RGB565**`. If not, it uses `**VIDIOC_S_FMT**` to reset it to the **RGB565** format to facilitate real-time display on the LCD.

Finally, based on the configuration, it determines whether to enable:

* Horizontal flip

* Vertical flip

If initialization succeeds, the camera device handle is returned; if it fails, the device is closed and an error is returned.

1.2.2 CMkaLists.txt file

The functionality of this example relies primarily on the **bsp_camera** driver. To successfully call the contents of the **bsp_camera** folder from the main function, you must configure the **CMakeLists.txt** file located in the **bsp_camera** folder. The configuration details are as follows:

peripheral > bsp_camera > CMakeLists.txt

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

idf_component_register(SRCS ${component_sources}
                       INCLUDE_DIRS "include"
                       REQUIRES driver esp_cam_sensor esp_sccb_intf esp_video bsp_display)

```

In this **CMakeLists.txt** file, we first use `**FILE(GLOB_RECURSE** component_sources "*.c")` to automatically search for all .c source files in the current directory and its subdirectories, and store them in the `**component_sources**` variable.

Next, we configure the source and header file directories for the current component using `**idf_component_register**`, where `**INCLUDE_DIRS** "**include**"` specifies the header file path.

Additionally, we add the driver libraries required by the current project to the `**REQUIRES**` section, including:

· driver: Basic driver library

· **esp_cam_sensor** camera driver library

· **esp_sccb_intf** camera SCCB communication driver library

· **esp_video video** capture driver library

· **bsp_display** display driver library

Finally, these configurations are registered with the ESP-IDF build system via `**idf_component_register**`, enabling the project to properly call camera and display-related functions.

1.2.3 main folder

The `**main**` folder is the core directory for program execution. It contains the main function executable file `**main.c**` and the header file `**main.h**`, located in the `include` folder. Please add the `**main**` folder to the `**CMakeLists.txt**` file in your build system.

The **main.h** file primarily includes the necessary header files: functions that use the bsp_display driver require the **bsp_display** header file, while functions that use the bsp_camera driver require the **bsp_camera** header file.

The following is an analysis of the main.c program: system initialization and the execution of functions for display and camera capabilities.

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

/*------------------------------------Variable declaration-----------------------------------------*/
static int video_node = -1;

static esp_ldo_channel_handle_t ldo4 = NULL;   // Handle for LDO channel 4 (used to control power output)
static esp_ldo_channel_handle_t ldo3 = NULL;   // Handle for LDO channel 3 (used to control power output)

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

/*------------------------------------Functional function------------------------------------------*/
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
            state = true;   // Update state to prevent repeated logs
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);   // Wait 1 second before looping again
    }
}
```

In the `**Init**` function, the system's hardware resources are first initialized, and each module is checked to ensure it has started up correctly.

At the beginning of the program, the LDO3 and LDO4 power channels are configured separately using the `**esp_ldo_channel_config_t**` structure. Specifically:

· LDO3 outputs a 2.5V voltage

· LDO4 outputs 3.3V

Next, `**esp_ldo_acquire_channel**` is called to acquire and enable the corresponding LDO channels, providing power to peripherals such as the display and camera.

Then, the GPIO interrupt service is installed using `**gpio_install_isr_service**` to support subsequent touch or external interrupt functions.

Afterward, `**display_init**` is called to initialize the LCD display, and `**set_lcd_blight(100)**` is used to turn on the screen backlight while setting the brightness to 100%.

Finally, the program calls `**camera_video_init**` to initialize the MIPI camera driver and uses `**camera_work()**` to start the camera video capture function. If initialization fails, `**init_fail**` is called to output an error message and terminate execution.

This function primarily handles:

- Power initialization
- GPIO interrupt initialization
- LCD display initialization
- Backlight initialization
- MIPI camera initialization
- and other basic system configuration tasks.

```
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 = {   // LDO channel 3 configuration
        .chan_id = 3,    // Channel ID: 3
        .voltage_mv = 2500,   // Output voltage: 2.5V
    };
    err = esp_ldo_acquire_channel(&ldo3_cof, &ldo3);   // Acquire and configure LDO3 channel
    if (err != ESP_OK)   // Check for error
        init_fail("ldo3", err);   // Handle initialization failure

    esp_ldo_channel_config_t ldo4_cof = {   // LDO channel 4 configuration
        .chan_id = 4,    // Channel ID: 4
        .voltage_mv = 3300,   // Output voltage: 3.3V
    };
    err = esp_ldo_acquire_channel(&ldo4_cof, &ldo4);   // Acquire and configure LDO4 channel
    if (err != ESP_OK)   // Check for error
        init_fail("ldo4", err);   // Handle initialization failure

    err = gpio_install_isr_service(0);   // Install GPIO interrupt service routine
    if (err != ESP_OK)   // Check for error
        init_fail("gpio isr service", err);   // Handle initialization failure

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

    err = set_lcd_blight(100);  // Enable backlight with 100% brightness
```

/* app_main*/

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

    Init();   // Call system initialization function

    set_camera_img_display(true);   // Enable camera image display

    MAIN_INFO("----------The screen is displaying.----------\r\n");   // Log that the screen is now displaying camer
}
```

In the `**app_main**` function, the camera task startup log is first output via `**MAIN_INFO**` to indicate that the system has begun running the camera functionality.

Next, the `**Init()**` function is called to complete system initialization, including:

A:Power initialization;

B:LCD display initialization;

C:Backlight initialization;

D:MIPI camera initialization.

After initialization is complete, the `**set_camera_img_display(true)**` function is called to enable the camera image display feature, allowing the camera's captured footage to be displayed in real time on the LCD screen.

Finally, a log message is output again to indicate that the screen has begun displaying the camera image.

This function is primarily used for:

A:Booting the system;

B:Initializing hardware;

C:Enabling the camera's real-time display feature;

D:It serves as the main entry point for the entire camera program.

1.2.4 CMkaLists.txt file

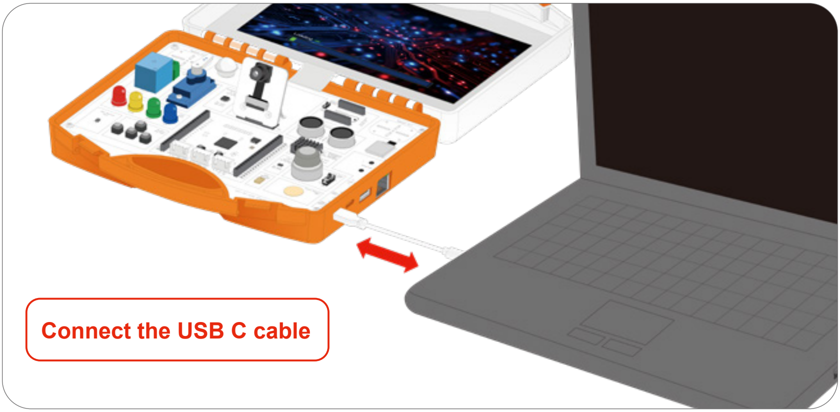
To successfully call the contents of the **bsp_display** and **bsp_camera** folders from the main function, you must configure the **CMakeLists.txt** file located in the main folder. The configuration details are as follows:

```
main > CMakeLists.txt
FILE(GLOB_RECURSE main ${CMAKE_SOURCE_DIR}/main/*.c)

idf_component_register(SRCS ${main}
                       INCLUDE_DIRS "include"
                       REQUIRES bsp_display bsp_camera)
```

First, define the directories for source files and header files, as well as the required driver libraries—specifically, the driver libraries needed to link **bsp_display** and **bsp_camera**. Then, use the **idf_component_register** command to register these settings with the build system, thereby enabling the main function to call these driver functions.

1.4 Programming procedure

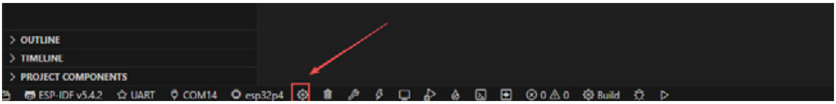
Connect the P4 device to the computer via USB



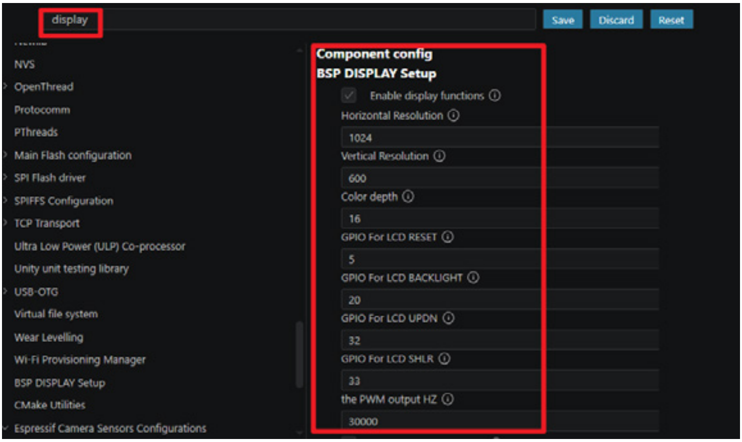
1.4.1.After cloning the code via Git (link to be confirmed), clear all local compilation information. Configure the **IDF** environment and chip model for compilation as per Lesson One, and set the serial port number for programming.

1.4.2. Next, we need to configure the **SDK**.

Click the icon shown in the image below.

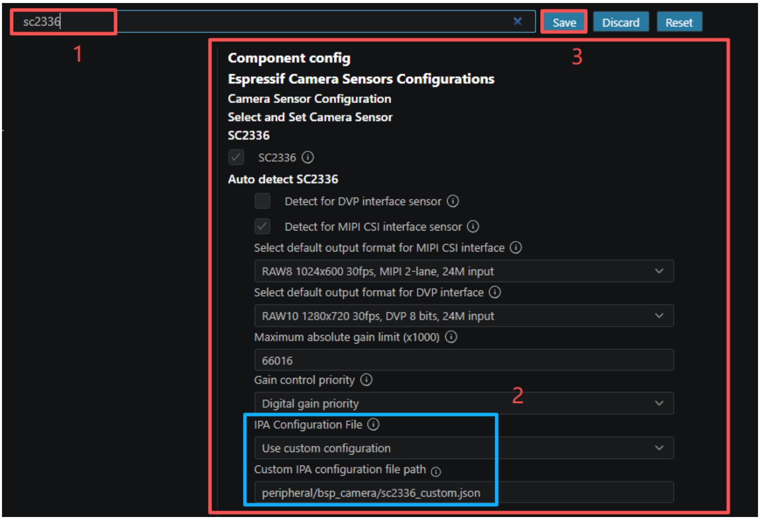


Wait for a short loading period, and then you can proceed with the relevant **SDK** configuration.The subsequent **SDKConfig** settings are essentially the same as in Lesson 1; you only need to enable “**display**”.

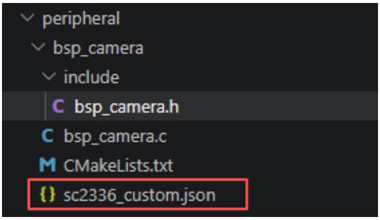


Here you need to configure the camera options to use the camera normally.

(Just make the same configuration as shown in the picture.)

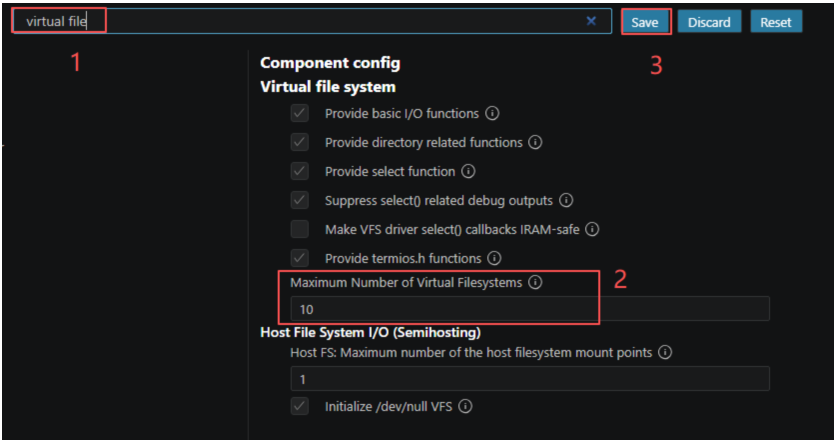


For the second step mentioned above, here we will use the configuration file we have prepared for this purpose. Only in this way can the camera screen display be normal, as we have mentioned before.



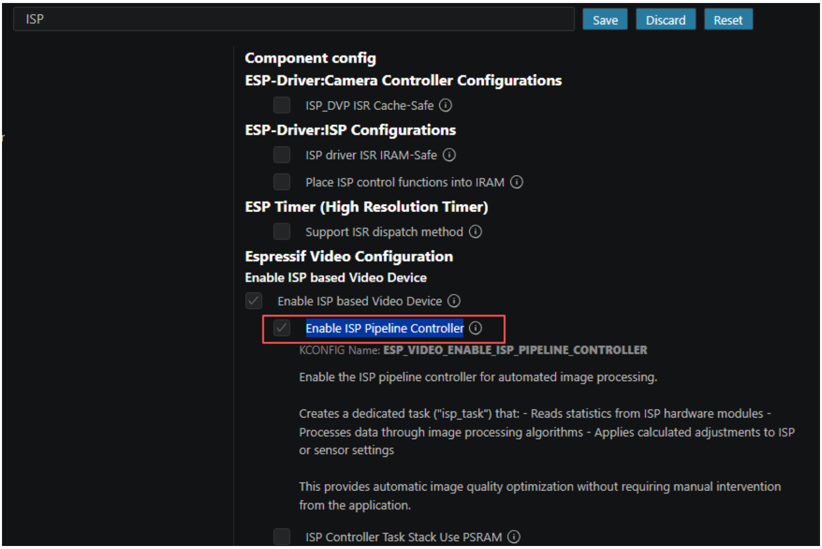
Then, search for **"virtual file"** and set this value to 10; it can also be any other number.

In the SDK configuration of ESP-IDF, **"Maximum Number of Virtual Filesystems"** is used to set the maximum number of file **system/device** nodes that the VFS framework can simultaneously mount. Since your project not only uses the V4L2 video device node like "**/dev/video0**", but may also mount other VFS devices such as serial ports, logs, SD cards or Flash file systems, the default configuration value is usually insufficient to accommodate all these mount points. Therefore, setting this value to 10 is to reserve sufficient VFS slots to avoid problems such as device registration failure, cameras not opening normally, or unstable system operation due to insufficient mount points. This ensures that all the devices and file systems that need to be accessed through VFS in the project can be registered and used normally.

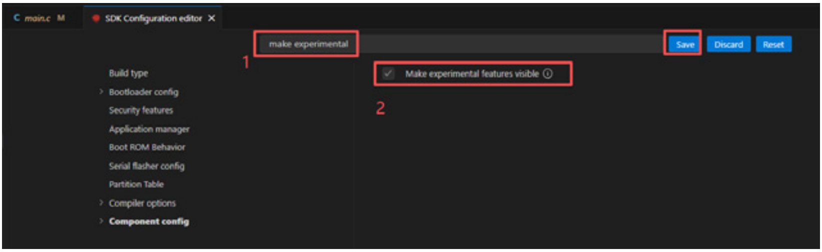


Then, search for "**ISP**" and open this one in the picture.

In ESP-IDF, Enable ISP based Video Device and Enable ISP Pipeline Controller are the core configurations for the camera ISP (Image Signal Processor) functions. Once enabled, the system will activate the video device driver based on the hardware ISP and create an independent **isp_task** background task. This task will automatically read the exposure, white balance, and other statistics from the ISP hardware, run image processing algorithms, and dynamically adjust the camera or **ISP** parameters to achieve image quality optimization such as automatic exposure and automatic white balance. This not only ensures that the output image from the camera has more stable and natural colors and brightness, but also eliminates the need for manual writing of complex image tuning logic at the application layer. These are essential configurations for using **SC2336** and other **MIPI** cameras on **ESP32-P4** to obtain high-quality images.

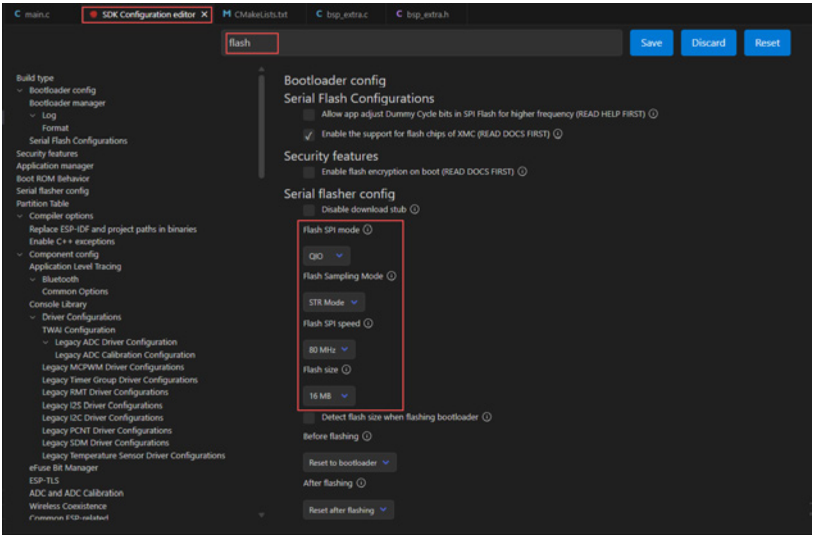


Then, enable the experimental feature option. Only then will you be able to use the **200MHz PSRAM**.

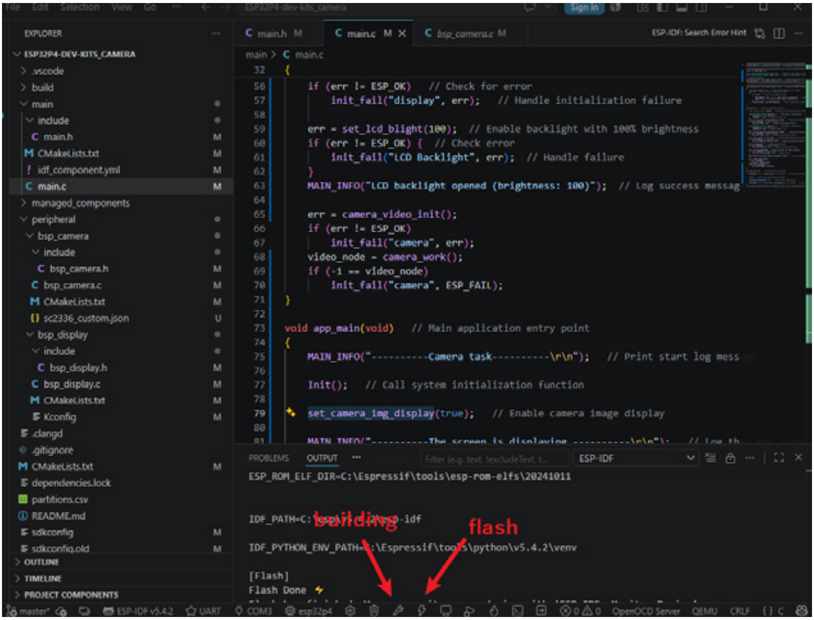


After that, enter "**flash**" in the search box to find flash-related settings.

(Make sure your flash configuration matches mine exactly.)



1.4.3.Click Compile. Once compilation is successful, click flashing.



ELECROW

MAKE YOUR MAKING EASIER