All-in-One Starter Kit for ESP32-P4

Preface..1

Lesson01--- GPIO.. 10

Lesson02--- Control the LED...13

Lesson03--- Touch Control LED... 17

Lesson04--- PIR Control LED...20

Lesson05--- Relay.. 24

Lesson06--- Hall Sensor...29

Lesson07--- Button Control LED...33

Lesson08--- Servo...38

Lesson09--- Display.. 42

Lesson10--- Touch Display.. 50

Lesson11--- Ultrasonic..58

Lesson12--- Temp&Humi Sensor...66

Lesson13--- Light Sensor..73

Lesson14--- Six-axis Accelerometer Sensor................................... 79

Lesson15--- Gas Module..85

Lesson16--- Microphone Recording..92

Lesson17--- Speaker...104

Preface

Welcome to the All-in-One Starter Kit for ESP32-P4 User Manual. Let's begin our journey into learning about the All-in-One Starter Kit for ESP32-P4 and how to make the most of its diverse electronic modules!

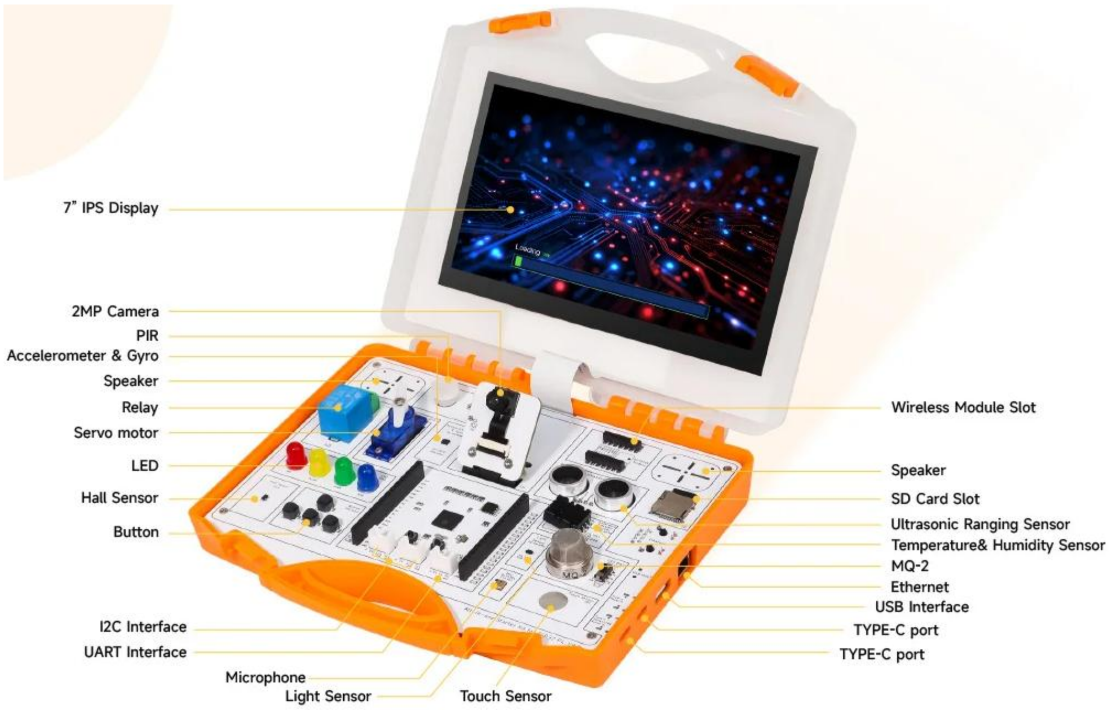
This development board comes with 18 easy-to-follow, engaging, and insightful lessons that will guide you step-by-step through the learning process. Along the way, you'll get hands-on experience with various electronic modules, sharpen your logical thinking, ignite your creativity, and learn how to program these modules to perform specific tasks.

The lessons start with the basics—installing the Arduino software—then introduce the All-in-One Starter Kit for ESP32-P4 development board and its different sensor modules. You'll learn about programming these modules and the languages used to control them, eventually mastering the practical applications of sensors. Each lesson is clearly explained, so even if you're a beginner, you'll quickly pick up Arduino programming.

The All-in-One Starter Kit for ESP32-P4 includes 16 electronic modules, each with unique features and functions, making it an ideal choice for beginners. Through this kit, you'll not only grasp the basic principles of sensors and important concepts like digital and analog signals, analog-to-digital conversion, and programming logic, but you'll also learn to work with more complex electronic modules. Most importantly, you'll get a solid start on learning Arduino programming, which will enhance your problem-solving and logical thinking skills.

For the programming part, we'll use the Arduino software. Arduino is one of the most popular open-source platforms worldwide, known for its simplicity and ease of learning. With a wide variety of hardware (different Arduino boards) and software (Arduino IDE), it's one of the best platforms to get started with programming.

Introduction to the Development Board



The All-in-One Starter Kit for the ESP32-P4 not only includes common sensor modules like temperature and humidity sensors, light sensors, and accelerometers, but also comes with several advanced hardware components, such as a high-definition camera, a 7-inch IPS display, dual speakers, and a microphone. These advanced modules provide developers with a great deal of flexibility and creative space, enabling the easy implementation of complex functions like video processing, audio playback, voice recognition, and display output.

The camera module captures high-definition images and supports advanced features such as image processing and facial recognition, making it ideal for applications like security monitoring and smart recognition. The 7-inch IPS display offers clear and vibrant visuals, perfect for developing touch interfaces, graphic displays, and data visualization projects. The combination of dual speakers and a microphone allows developers to create multimedia applications like voice interaction and sound playback, enhancing project interactivity and user experience.

The integration of these advanced modules makes the ESP32-P4 development board an excellent choice not only for beginners to learn and get started but also for professional developers building smart hardware, IoT applications, and multimedia projects. Whether you're working on embedded development, AI applications, or smart home systems, the All-in-One Starter Kit provides extensive support, helping developers quickly bring their ideas to life and shorten development cycles.

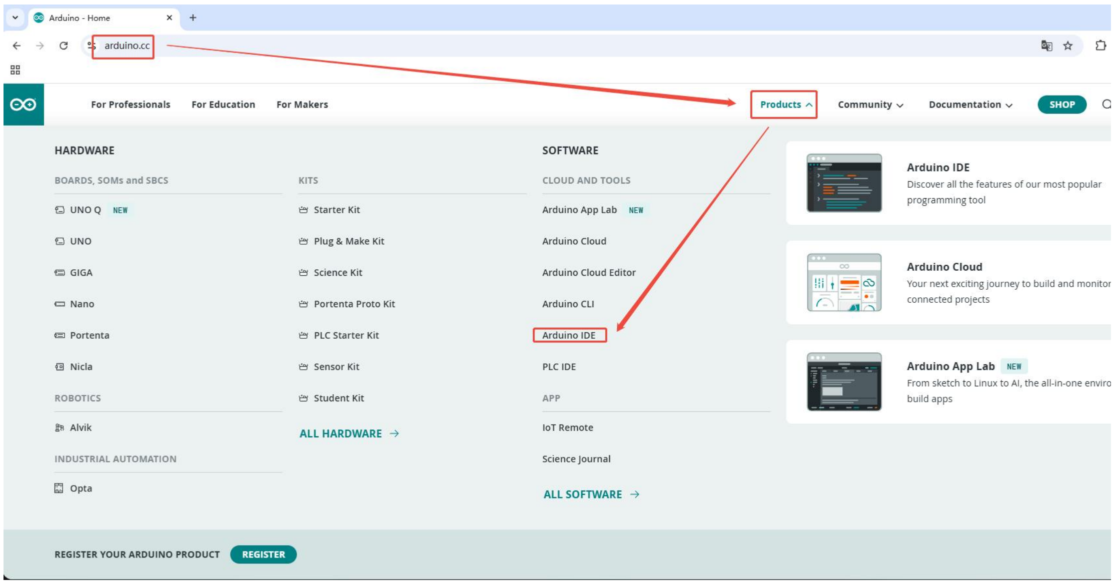
Installing Arduino IDE

Download Arduino in Windows system

Step 1:

Login to Arduino official website, download Arduino

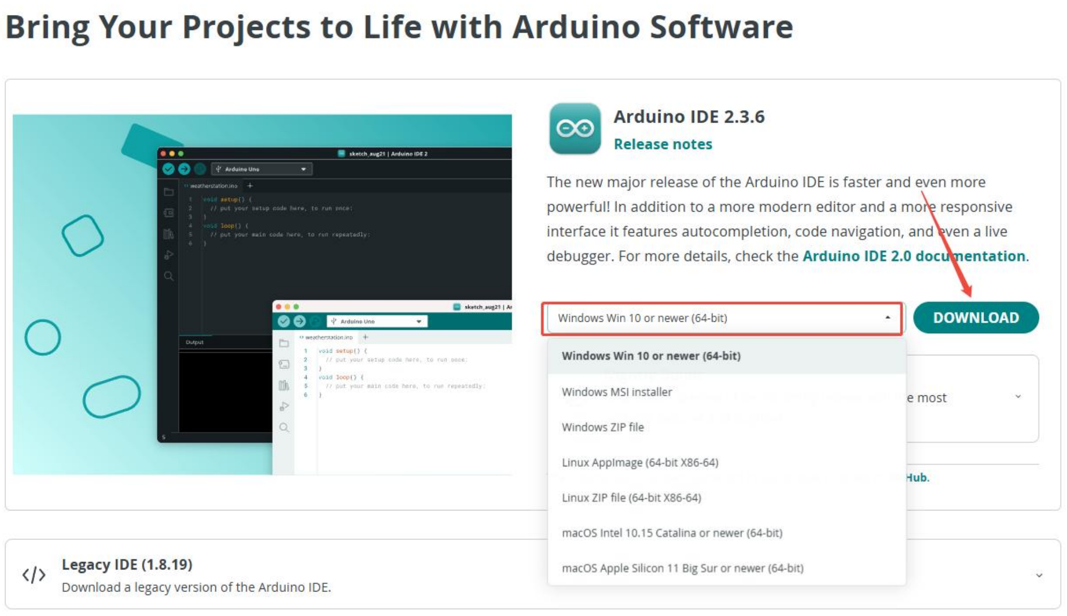
Arduino official website: https://www.arduino.cc/



Step 2:

Select your computer's corresponding system to download, such as Window system.

Note: This tutorial uses version 2.3.6. You can try other versions, but if you encounter any issues during flashing, please switch back to version 2.3.6 and try again.


Bring Your Projects to Life with Arduino Software

Arduino IDE 2.3.6

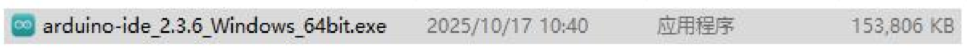
Release notes

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger. For more details, check the Arduino IDE 2.0 documentation.

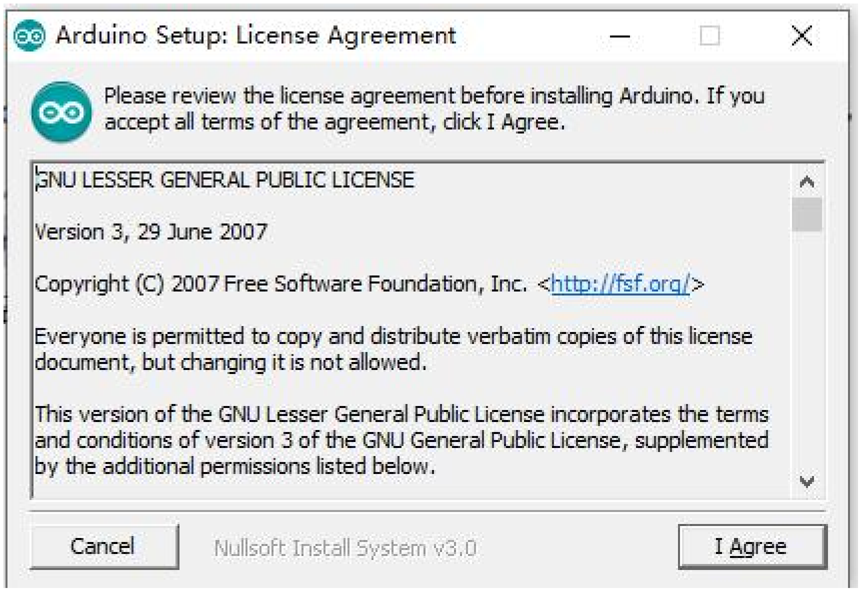
Windows Win 10 or newer (64-bit)
DOWNLOAD
Windows Win 10 or newer (64-bit)
Windows MSI installer
Windows ZIP file
Linux AppImage (64-bit X86-64)
Linux ZIP file (64-bit X86-64)
macOS Intel 10.15 Catalina or newer (64-bit)
macOS Apple Silicon 11 Big Sur or newer (64-bit)
Legacy IDE (1.8.19)
Download a legacy version of the Arduino IDE.

Step 3:

1.When installing Arduino, please locate the executable file with the .exe extension within the folder where you previously saved, which is the Arduino installation package.


arduino-ide_2.3.6_Windows_64bit.exe 2025/10/17 10:40 应用程序 153,806 KB

2. After double-clicking the installation package, this page will appear. Click on "I Agree".


Arduino Setup: License Agreement

Please review the license agreement before installing Arduino. If you accept all terms of the agreement, click I Agree.

GNU LESSER GENERAL PUBLIC LICENSE

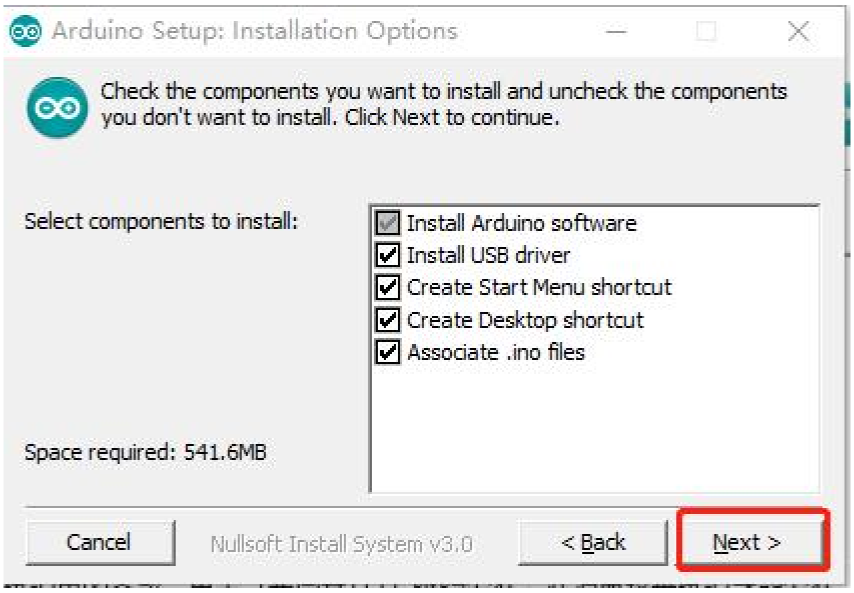
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

3.Check all options by default and click next.


Arduino Setup: Installation Options

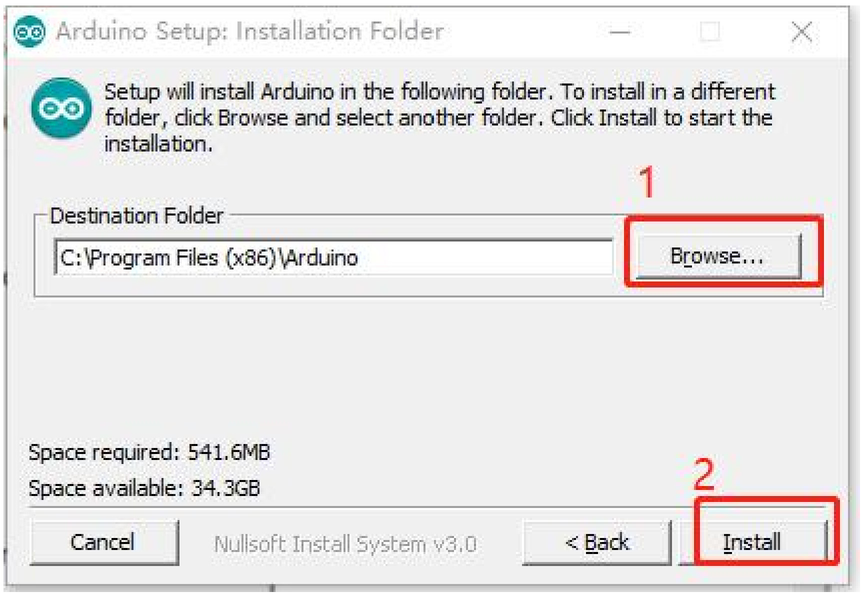
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

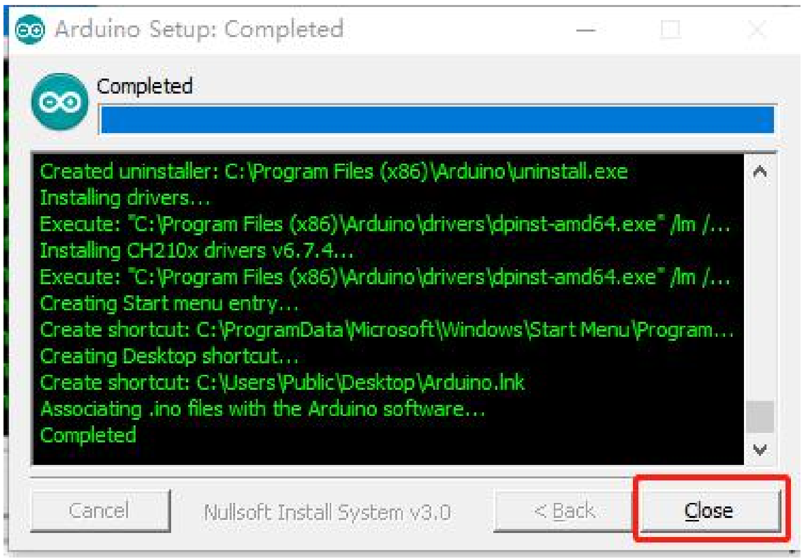
- [✓] Install Arduino software
- [✓] Install USB driver
- [✓] Create Start Menu shortcut
- [✓] Create Desktop shortcut
- [✓] Associate .ino files

Space required: 541.6MB

4.Click on 'Browse' to select the installation location, it is recommended to install it on any drive other than the C: drive. Then click 'Install'.


Arduino Setup: Installation Folder
Setup will install Arduino in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.
1
Destination Folder
C:\Program Files (x86)\Arduino
Browse...
Space required: 541.6MB
Space available: 34.3GB
2
Cancel
Nullsoft Install System v3.0
< Back
Install

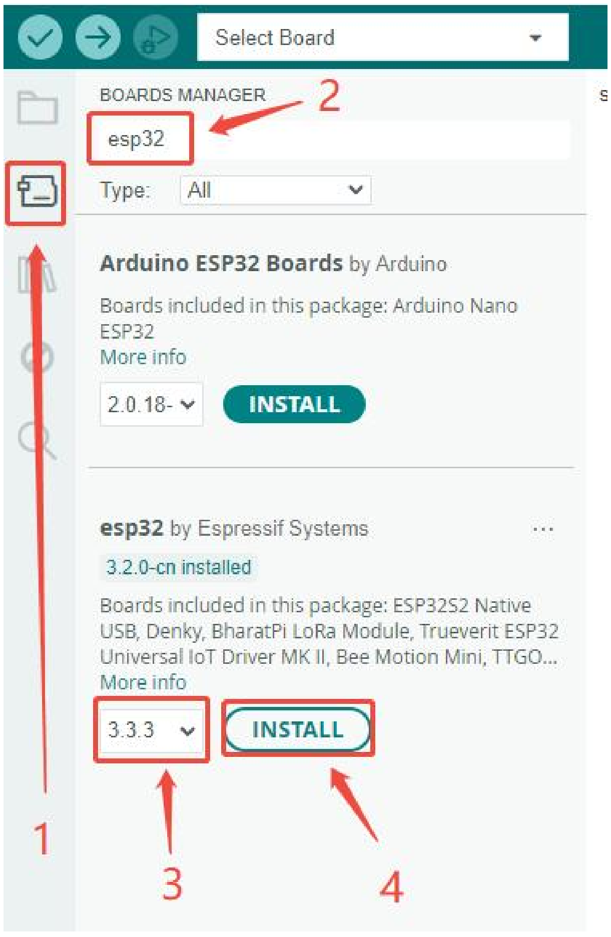
5.Installation Completed



```
Created uninstaller: C:\Program Files (x86)\Arduino\uninstall.exe
Installing drivers...
Execute: "C:\Program Files (x86)\Arduino\drivers\dpinst-amd64.exe" /lm /...
Installing CH210x drivers v6.7.4...
Execute: "C:\Program Files (x86)\Arduino\drivers\dpinst-amd64.exe" /lm /...
Creating Start menu entry...
Create shortcut: C:\ProgramData\Microsoft\Windows\Start Menu\Program...
Creating Desktop shortcut...
Create shortcut: C:\Users\Public\Desktop\Arduino.lnk
Associating .ino files with the Arduino software...
Completed
```

Installing the ESP32-P4 Development Board

①Click the Board Icon on the left side
②Search for "ESP32" in the search bar
③Select Version 3.3.3. Note: While you may try using other versions, if you encounter errors during flashing, switch back to version 3.3.3. This tutorial is based on code developed for version 3.3.3
④Click "INSTALL" to install the board



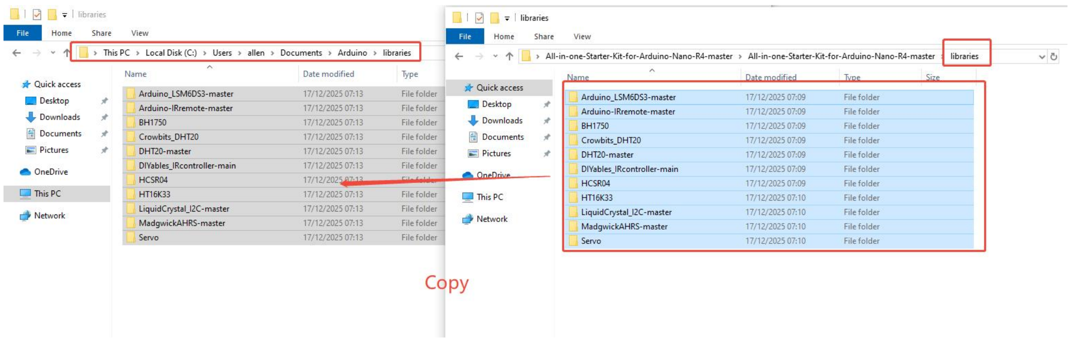
Arduino ESP32 Boards by Arduino
Boards included in this package: Arduino Nano ESP32
More info

esp32 by Espressif Systems
3.2.0-cn installed
Boards included in this package: ESP32S2 Native USB, Denky, BharatPi LoRa Module, Trueverit ESP32 Universal IoT Driver MK II, Bee Motion Mini, TTGO...
More info

During the installation process, a pop-up window will appear. Simply click "Yes" to proceed.

Importing Library Files (Important)

Before starting, we need to download the required library files for the kit and place them in the "/Arduino/libraries" directory on your computer.
Download link:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_libraries
Copy all the library files from the downloaded libraries folder to the following directory on your local computer: **C:\Users\YourUserName\Documents\Arduino\libraries**
Make sure to replace "YourUserName" with your actual Windows username.



This PC › Local Disk (C:) › Users › allen › Documents › Arduino › libraries

Name	Date modified	Type
Arduino_LSM6DS3-master	17/12/2025 07:13	File folder
Arduino-IRremote-master	17/12/2025 07:13	File folder
BH1750	17/12/2025 07:13	File folder
Crowbits_DHT20	17/12/2025 07:13	File folder
DHT20-master	17/12/2025 07:13	File folder
DIYables_IRcontroller-main	17/12/2025 07:13	File folder
HCSR04	17/12/2025 07:13	File folder
HT16K33	17/12/2025 07:13	File folder
LiquidCrystal_I2C-master	17/12/2025 07:13	File folder
MadgwickAHRS-master	17/12/2025 07:13	File folder
Servo	17/12/2025 07:13	File folder

All-in-one-Starter-Kit-for-Arduino-Nano-R4-master › All-in-one-Starter-Kit-for-Arduino-Nano-R4-master › libraries

Name	Date modified	Type	Size
Arduino_LSM6DS3-master	17/12/2025 07:09	File folder	
Arduino-IRremote-master	17/12/2025 07:09	File folder	
BH1750	17/12/2025 07:09	File folder	
Crowbits_DHT20	17/12/2025 07:09	File folder	
DHT20-master	17/12/2025 07:09	File folder	
DIYables_IRcontroller-main	17/12/2025 07:09	File folder	
HCSR04	17/12/2025 07:09	File folder	
HT16K33	17/12/2025 07:10	File folder	
LiquidCrystal_I2C-master	17/12/2025 07:10	File folder	
MadgwickAHRS-master	17/12/2025 07:10	File folder	
Servo	17/12/2025 07:10	File folder	

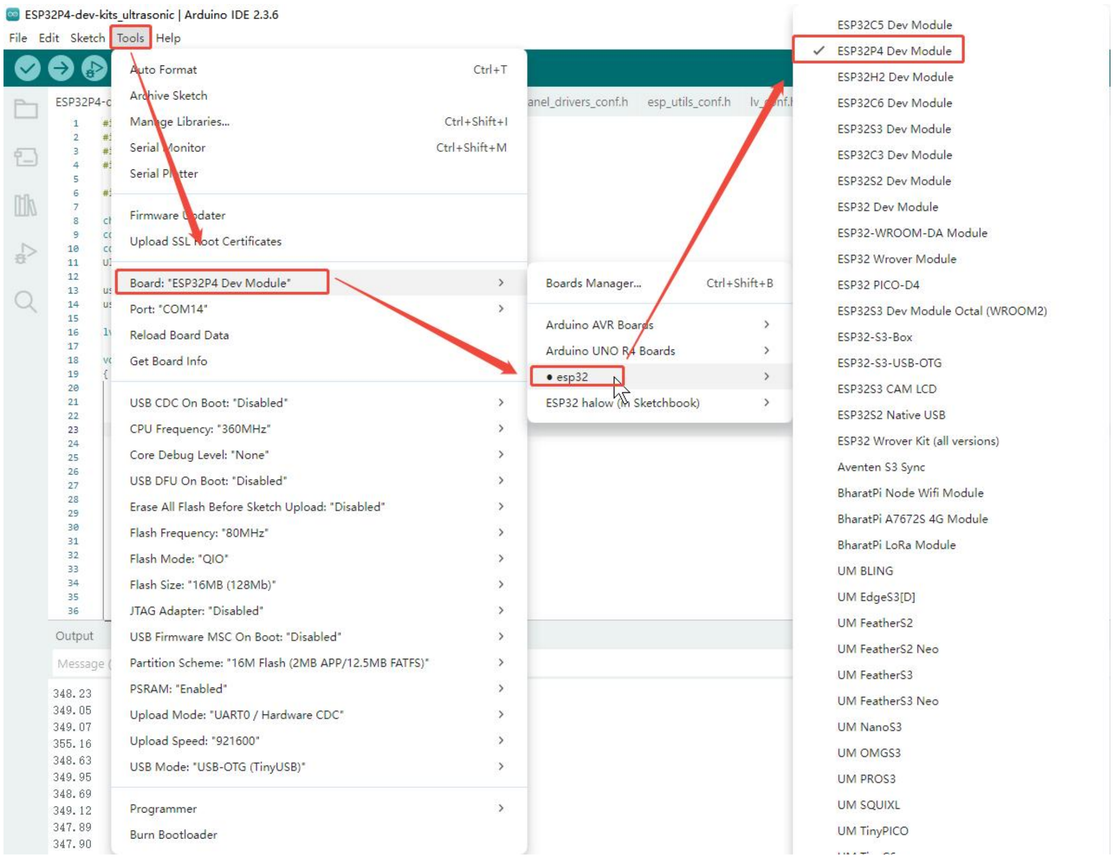
Note: If there is no "libraries" folder in your path, please create one yourself.

Uploading Steps (Important)

The upload steps for all subsequent lessons are the same. Please read carefully, and if you forget any steps, you can always return here and follow the instructions

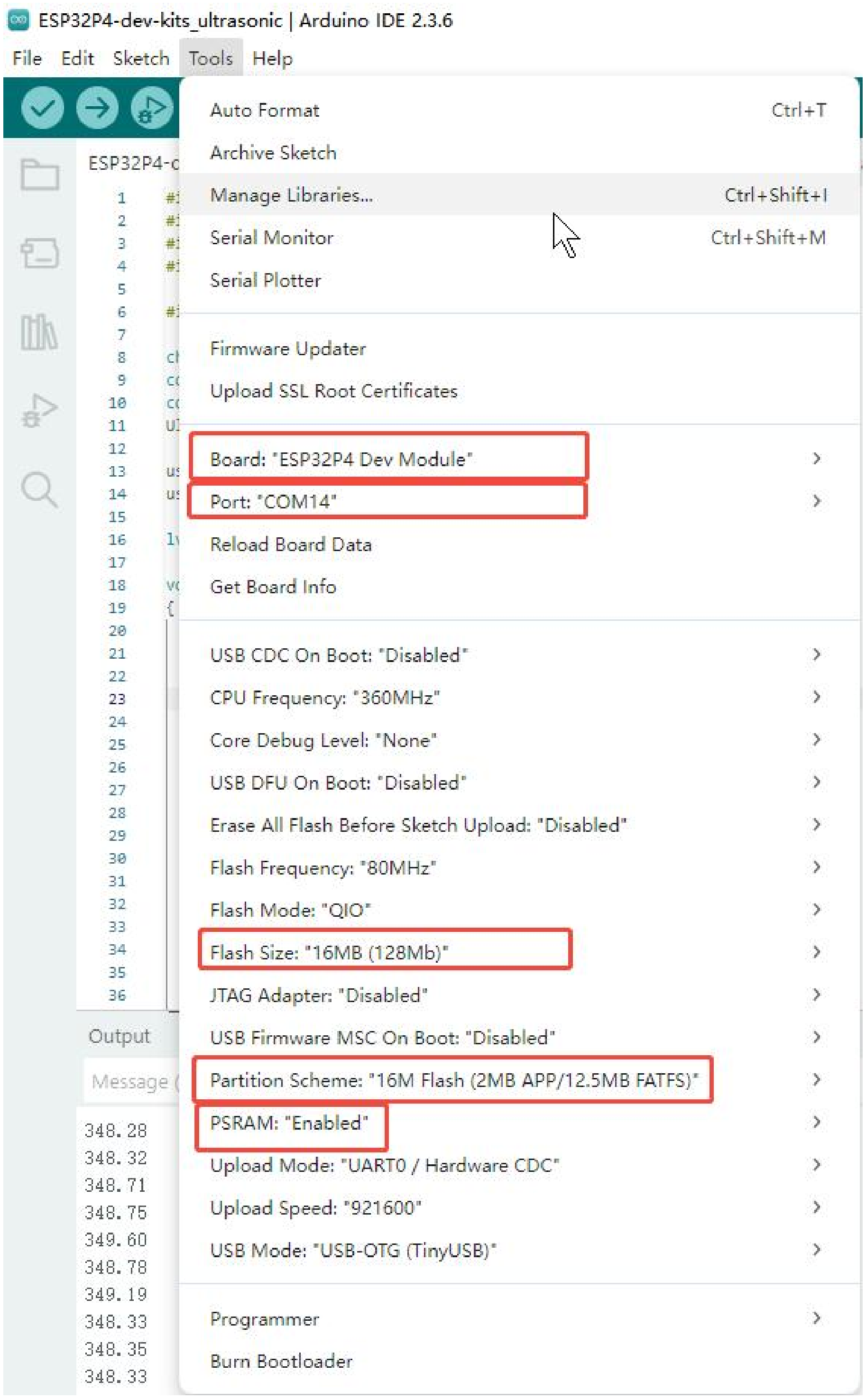
1.Connect the Type-C upload port to your computer

2.After writing the code, select the board to upload:"Tools" -> "Board" -> "ESP32" -> "ESP32P4 Dev Module"


ESP32P4-dev-kits_ultrasonic | Arduino IDE 2.3.6

Board: "ESP32P4 Dev Module"
Port: "COM14"
USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "None"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (2MB APP/12.5MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "USB-OTG (TinyUSB)"

3.Select other important configurations


ESP32P4-dev-kits_ultrasonic | Arduino IDE 2.3.6

File Edit Sketch Tools Help

Auto Format Ctrl+T
Archive Sketch
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter

Firmware Updater
Upload SSL Root Certificates

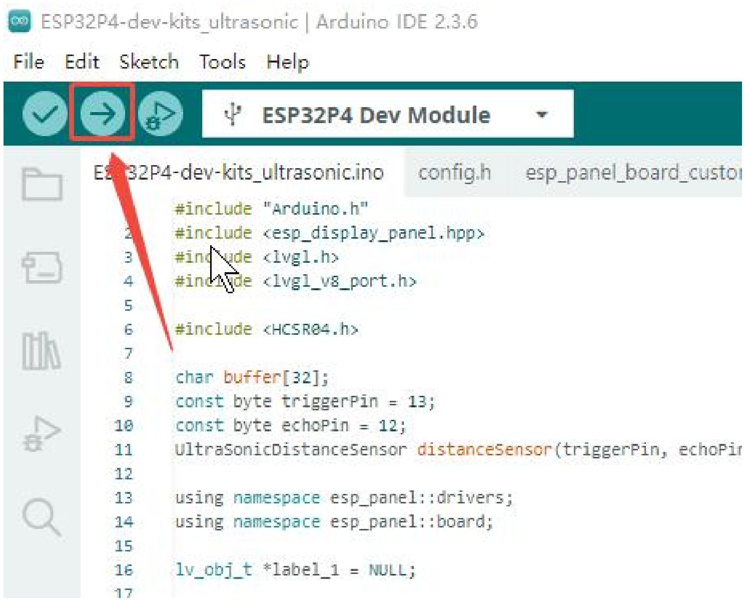
Board: "ESP32P4 Dev Module"
Port: "COM14"
Reload Board Data
Get Board Info

USB CDC On Boot: "Disabled"
CPU Frequency: "360MHz"
Core Debug Level: "None"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "16MB (128Mb)"
JTAG Adapter: "Disabled"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "16M Flash (2MB APP/12.5MB FATFS)"
PSRAM: "Enabled"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "USB-OTG (TinyUSB)"

Programmer
Burn Bootloader

```
348.28
348.32
348.71
348.75
349.60
348.78
349.19
348.33
348.35
348.33
```

4.Click the "Upload" button to flash the code to the development board.



ESP32P4-dev-kits_ultrasonic | Arduino IDE 2.3.6
File Edit Sketch Tools Help
ESP32P4 Dev Module
ESP32P4-dev-kits_ultrasonic.ino
config.h
esp_panel_board_custo

```
#include "Arduino.h"
#include <esp_display_panel.hpp>
#include <lvgl.h>
#include <lvgl_v8_port.h>

#include <HCSR04.h>

char buffer[32];
const byte triggerPin = 13;
const byte echoPin = 12;
UltraSonicDistanceSensor distanceSensor(triggerPin, echoPi

using namespace esp_panel::drivers;
using namespace esp_panel::board;

lv_obj_t *label_1 = NULL;

```

Lesson01---GPIO

Introduction

In this lesson, we will learn how to control GPIO (General Purpose Input/Output) pins. To understand hardware control, the first thing we need to master is the use of GPIO, as hardware control is essentially achieved by manipulating different I/O pins. By correctly configuring and controlling GPIO, we can interact with various hardware devices and perform more complex tasks. Therefore, mastering GPIO control is the foundation of hardware development.

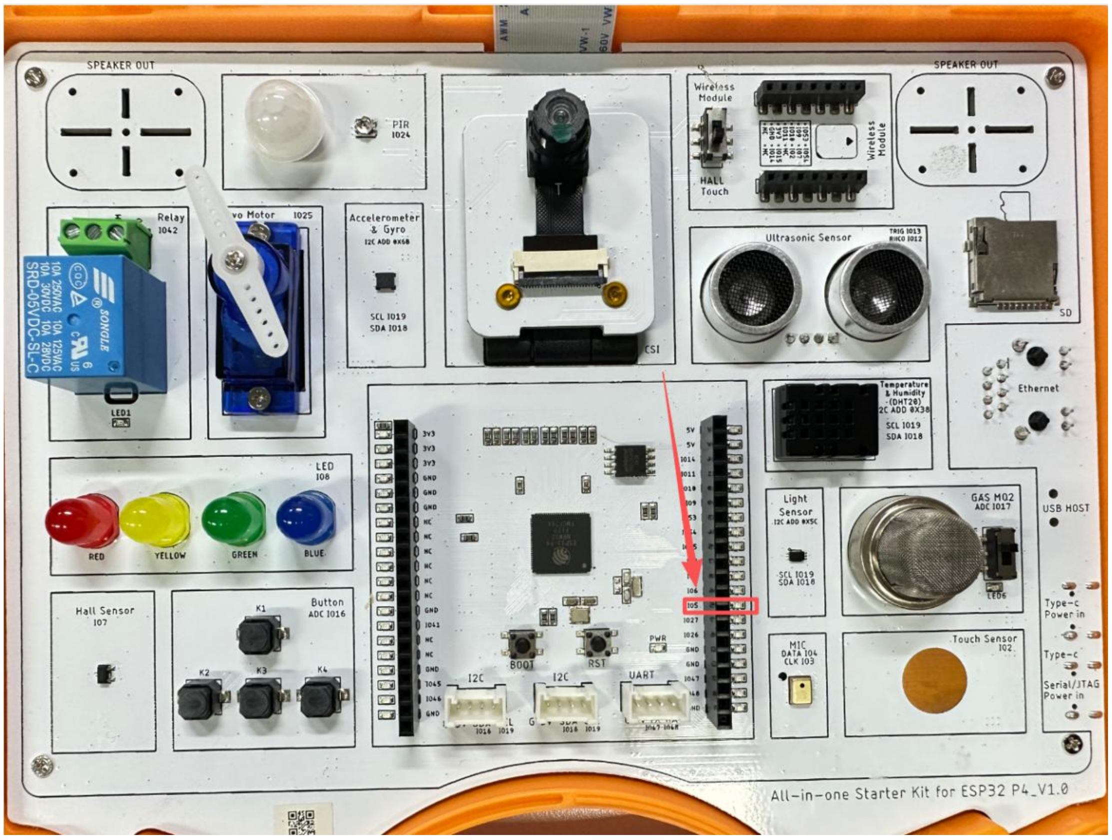
Learning Goals

1.Understand what GPIO is

2.Learn how to control GPIO pins

3.Master the use of the digitalWrite() function to control GPIO pins

Hardware Used in This Lesson:

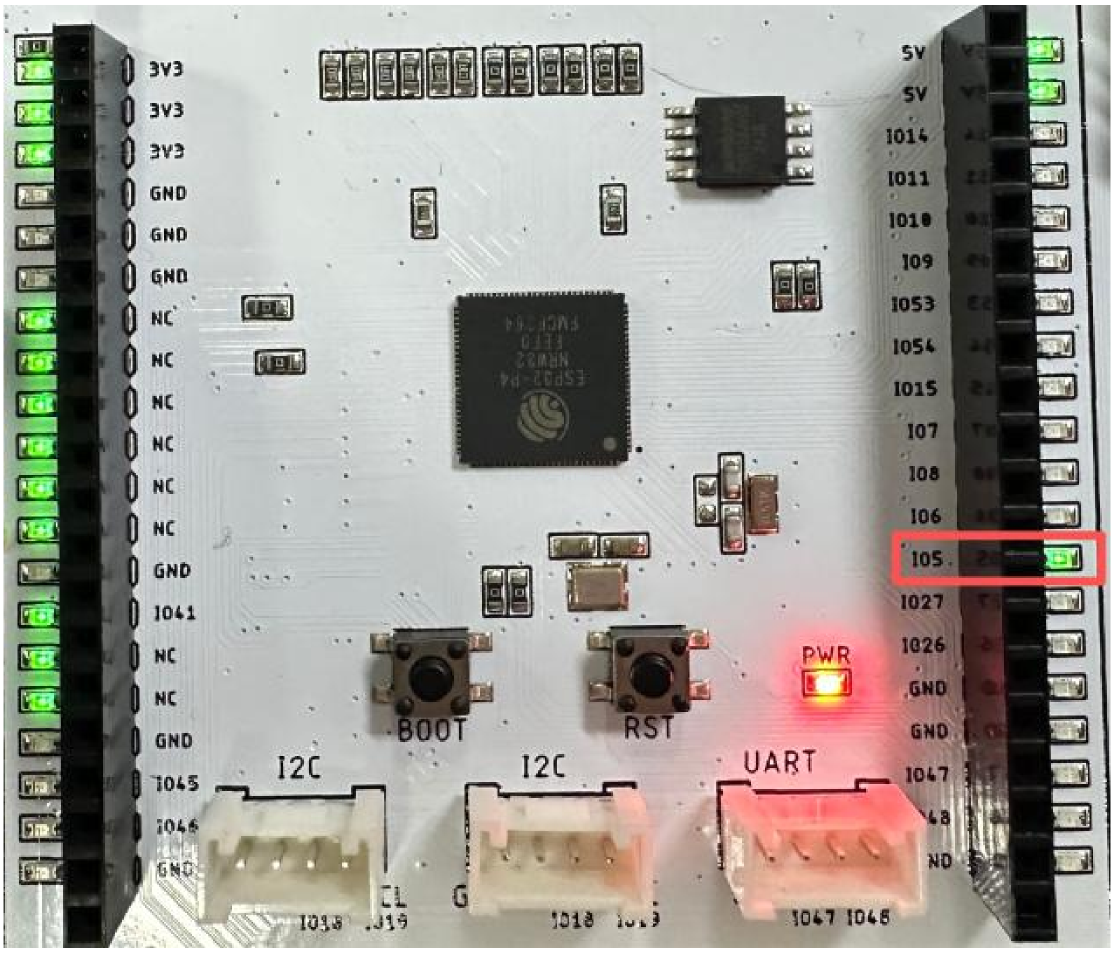


GPIO Working Principle

GPIO (General Purpose Input/Output) is a common interface found on microcontrollers (MCUs) or single-board computers (like Raspberry Pi) used to process digital signal input or output. Its working principle is simple, but it plays a crucial role in hardware control and embedded systems.

GPIO pins can be configured as input or output, depending on the need to interact with external devices.
Input Mode: When a GPIO pin is configured as an input, it is used to receive electrical signals from external devices (such as switches, sensors, etc.). It can read the state of external devices (e.g., high or low voltage levels).
Output Mode: When a GPIO pin is configured as an output, it can control external devices. By changing the output voltage level (high or low), it can switch devices like LEDs, relays, motors, etc. on or off.

Preview of the Result



Upon successful execution, you will see the LED on the All-in-One Starter Kit for P4 flashing—turning on for 0.5 seconds and off for 0.5 seconds.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson01_GPIO

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#define LED_GPIO 5
```

Import the Arduino core library, which contains all the basic functions required for programming. This library provides support for hardware pins, delays, and other operations. Define LED_GPIO as pin 5.

setup() function

```
void setup()
{
    pinMode(LED_GPIO, OUTPUT);// Set the LED pin as an output
}
```

The setup() function is a special function in Arduino programs that runs once at the beginning. It is typically used for initialization. The line of code pinMode(LED_GPIO, OUTPUT); sets the pin LED_GPIO (which is pin 5) to output mode. This means you can control the connected LED by outputting current through this pin.

loop() function

```
void loop() {
    digitalWrite(LED_GPIO, HIGH);   // Turn the LED on (set pin HIGH)
    delay(500);   // Wait for 500 milliseconds (0.5 second)

    digitalWrite(LED_GPIO, LOW);   // Turn the LED off (set pin LOW)
    delay(500);   // Wait for 500 milliseconds (0.5 second)
}
```

The loop() function is the main loop of an Arduino program, and it runs continuously. First, it sets GPIO pin 5 to a high voltage, waits for 0.5 seconds, then sets GPIO pin 5 to a low voltage, and waits another 0.5 seconds. This process repeats, creating a blinking effect.
digitalWrite(LED_GPIO,HIGH):is the method used to write a high or low voltage to a GPIO pin. Here, it writes a high voltage to the LED_GPIO pin.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson02---Control the LED

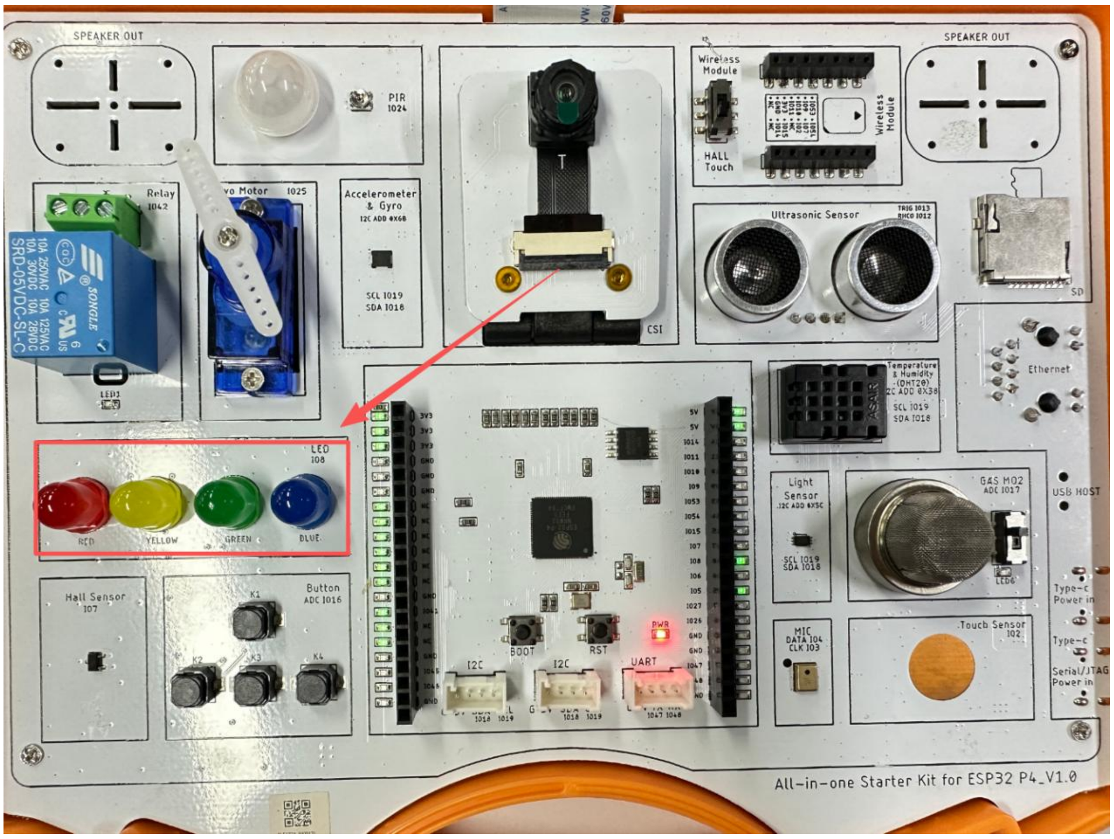
Introduction

In this lesson, we will learn how to control LEDs. In fact, controlling LEDs is very similar to controlling GPIO pins. On our All-in-One Starter Kit for ESP32-P4 development board, although there are four LEDs, their control method is slightly different due to the wiring design. Through a clever circuit design, only two pins are required to control the on/off states of all four LEDs simultaneously. This approach not only simplifies the hardware design but also improves resource efficiency.

Learning Goals

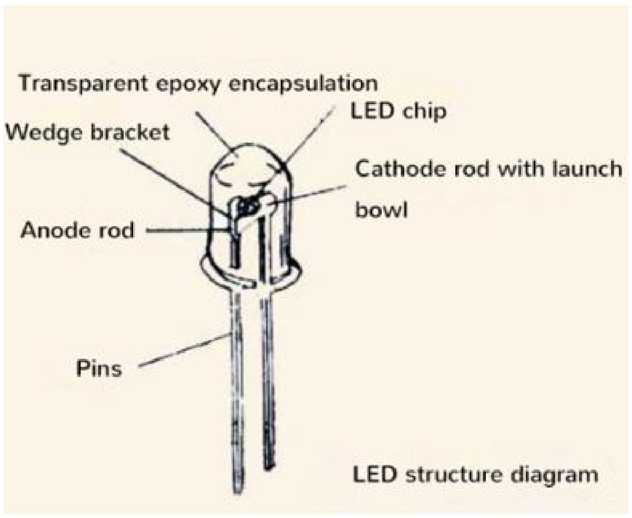
1.Understand how LEDs work
2.Learn how to use the Adafruit NeoPixel library
3.Master how to control LEDs using the digitalWrite() function

Hardware Used in This Lesson:

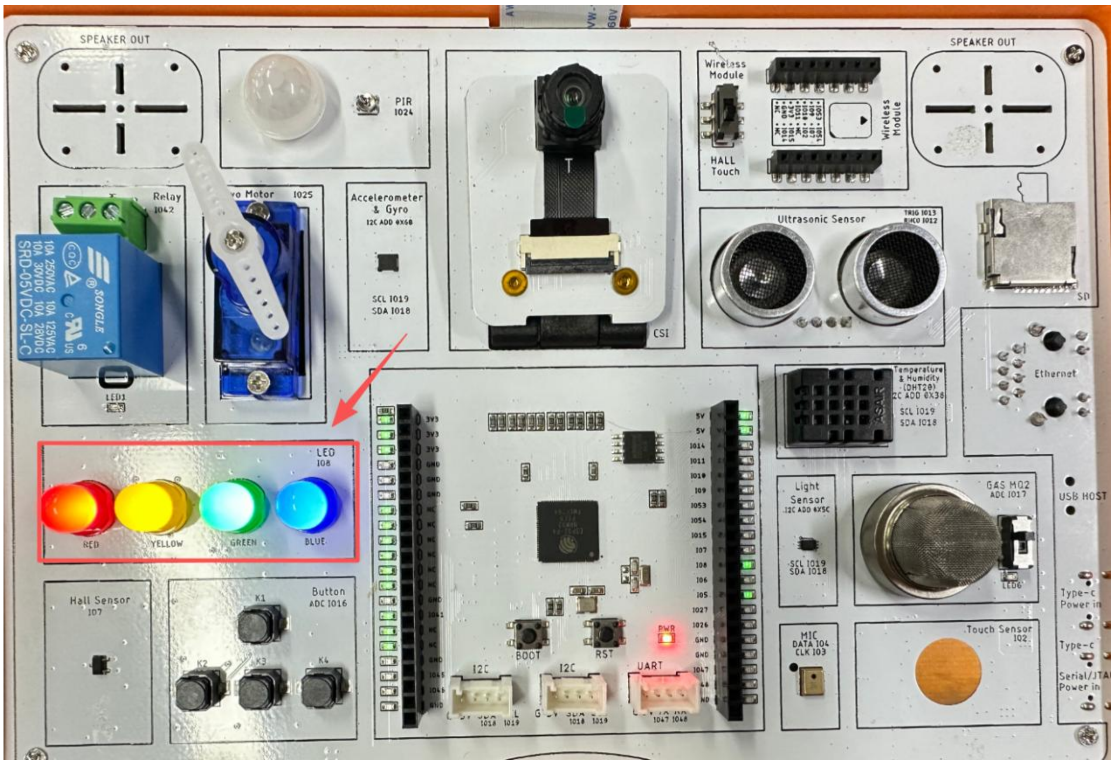


The working principle of LED

An LED, or light-emitting diode, is a solid-state semiconductor device that converts electrical energy directly into visible light. At its core is a semiconductor chip: one side of the chip is mounted to the lead frame and connected to the negative terminal of the power supply, while the other side is connected to the positive terminal. In most cases, the longer lead is the positive (anode) and the shorter lead is the negative (cathode).



Preview of the Result



After successful execution, you will see the LEDs on the All-in-One Starter Kit for P4 blinking—on for 1 second and off for 1 second.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson02_Control_the_LED

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"   // Include the Arduino library
#include <Adafruit_NeoPixel.h>   // Include the Adafruit NeoPixel library
#define LED_NUM      2     // Define the number of LEDs in the strip (2 LEDs)
#define LED_PIN      8     // Define the GPIO pin number for the LED strip (pin 8)
```

Import the Arduino core library and the Adafruit NeoPixel library. We use this library to control the four LEDs, but since the red, yellow, and green LEDs are connected to the same pin, we only need to define two LED counts in total.

Create a NeoPixel object

```
Adafruit_NeoPixel strip(LED_NUM, LED_PIN, NEO_GRB + NEO_KHZ800);   // Create a NeoPixel strip object with 2 LEDs, on pin 8
```

This line of code creates an Adafruit NeoPixel object named strip. The main parameters specify that there are 2 LEDs, the control GPIO pin is 8, the RGB LED color order is set to RGB, and the data transmission frequency is 800 KHz, which is the standard frequency used by NeoPixel LEDs.

setup() Function

```
void setup()
{
  strip.begin();              // Initialize the NeoPixel strip
  strip.show();               // Make sure the strip is turned off initially
  strip.setBrightness(50);   // Set the brightness of the LEDs (range: 0 to 255)
}
```

Initialize the NeoPixel LED strip; this function must be called before controlling the LEDs. Use the show method to update the LED states and display the initialized LED colors. Set the LED brightness—here it is set to 50, with a possible range from 0 to 255.

loop() Function

```
void loop()
{
  // Set the first LED (index 0) to white color (RGB: 255, 255, 255)
  strip.setPixelColor(0, strip.Color(255, 255, 255));
  // Set the second LED (index 1) to white color (RGB: 255, 255, 255)
  strip.setPixelColor(1, strip.Color(255, 255, 255));

  strip.show();              // Update the strip to show the new colors
  delay(1000);                    // Wait for 1 second (1000 milliseconds)

  // Turn off the first LED (index 0)
  strip.setPixelColor(0, strip.Color(0, 0, 0));
  // Turn off the second LED (index 1)
  strip.setPixelColor(1, strip.Color(0, 0, 0));

  strip.show();                  // Update the strip to show the new (off) state
  delay(1000);                    // Wait for 1 second before repeating the loop
}
```

The loop() function is the main loop of an Arduino program and runs continuously. First, it sets GPIO pin 5 to high, waits 0.5 seconds, then sets GPIO pin 5 to low, waits another 0.5 seconds, and repeats this process to create a blinking effect.

strip.setPixelColor(0, strip.Color(255, 255, 255));the parameter 0 directly controls the three LEDs on the development board. The first 255 sets the red LED brightness to 255, the second 255 sets the yellow LED brightness to 255, and the third 255 sets the green LED brightness to 255.

strip.setPixelColor(0, strip.Color(1, 255, 255));the parameter 1 directly controls the blue LED on the development board. To set it to full brightness, all three parameters should be 255; to turn it off, all three parameters should be 0.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson03---Touch Control the LED

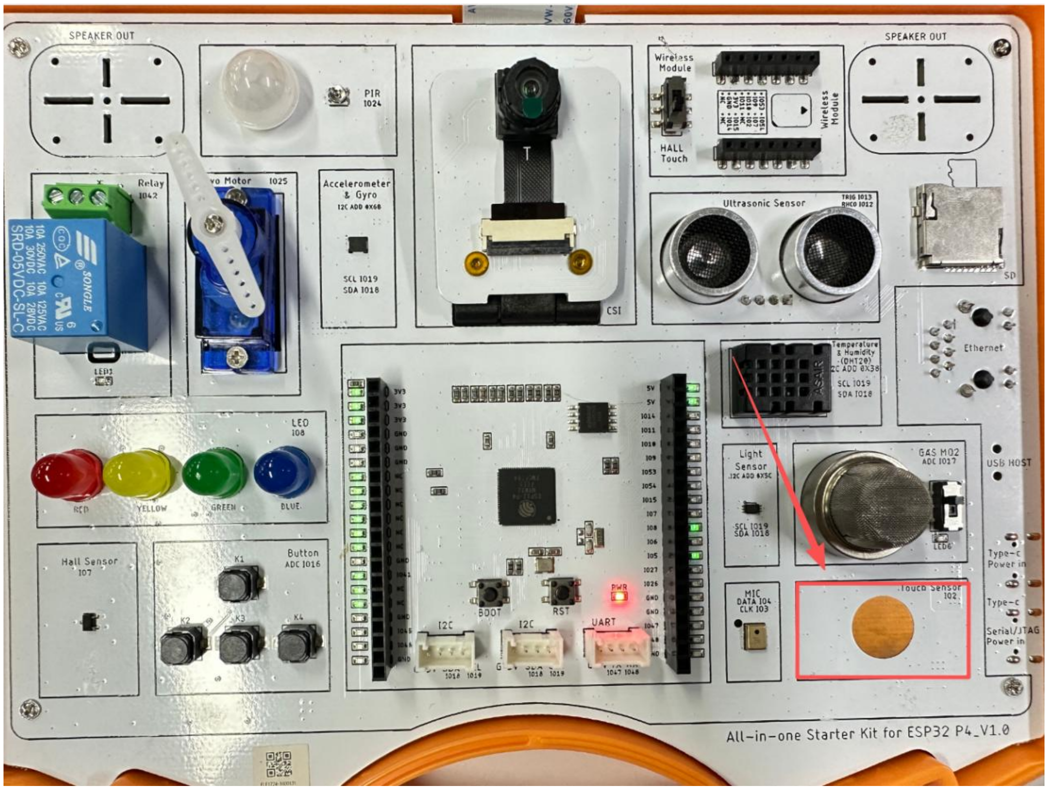
Introduction

In this lesson, we will learn how to use a touch sensor to control an LED. We will read the digital pin to get the current state of the touch sensor. When the touch sensor is detected as pressed, the LED will turn on; if it is not pressed, the LED will turn off. This allows us to implement simple touch-based interactive control.

Learning Goals

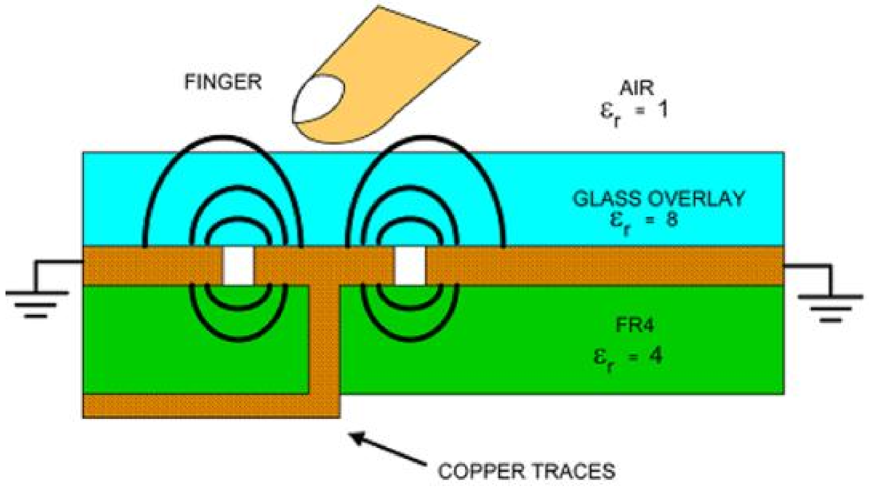
1.Understand how the touch sensor works
2.Learn how to use digitalRead() to read the sensor's voltage level
3.Master the use of if-else statements

Hardware Used in This Lesson:

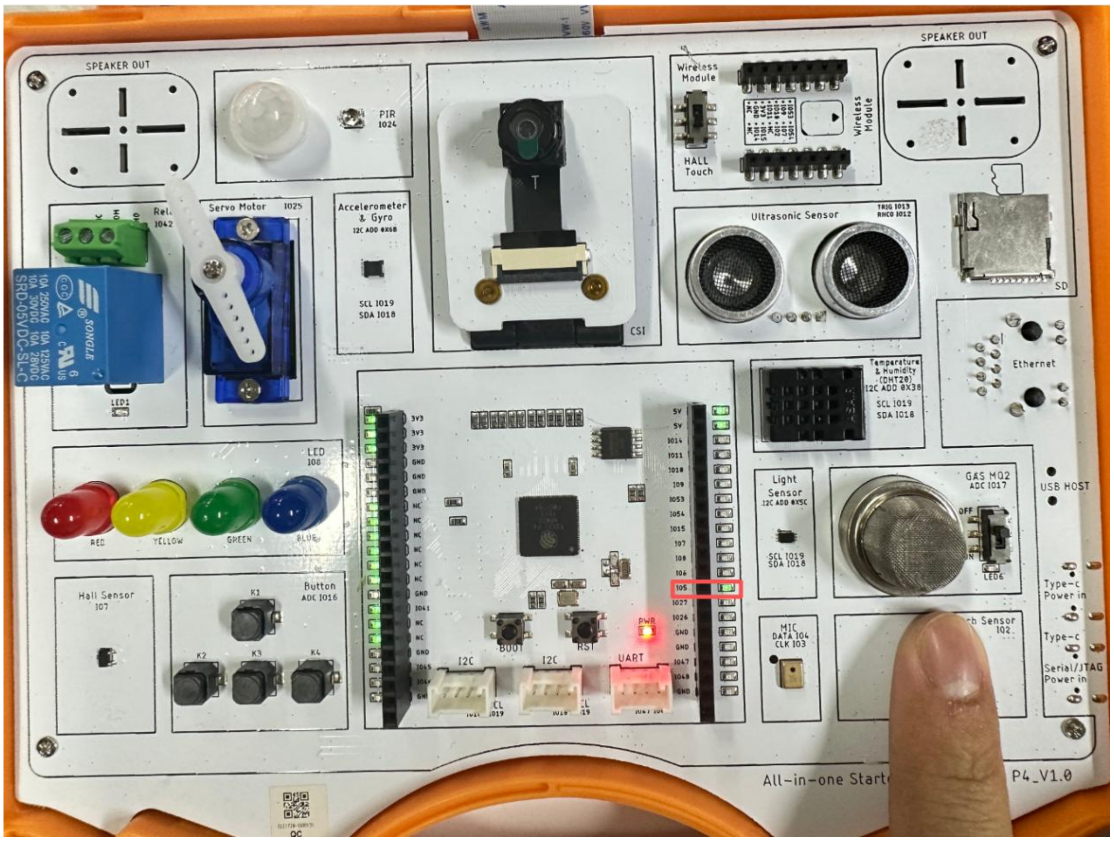


The working principle of Touch Sensor

The touch sensor is a digital input module, which is actually equivalent to a switch. It controls the on-off of the circuit through the conductive film on the sensor, and is widely used in touch switch control circuits. Because our fingers are also conductive, when we touch the conductive film on the touch sensor with our finger, the circuit is turned on, and when the finger leaves the conductive film on the touch sensor, the circuit is disconnected.



Preview of the Result



After successful execution, when the touch sensor is pressed, the LED corresponding to IO5 lights up. When the touch sensor is released, the LED corresponding to IO5 turns off.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson03_Touch_Control_LED

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#define TOUCH_PIN     2     // Define the GPIO pin number for the touch sensor (pin 2)
#define LED_PIN       5    // Define the GPIO pin number for the LED indicator (pin 5)
```

Import the Arduino core library, which provides all the basic functions needed in the program. Define the pin connected to the touch sensor, here set as GPIO2, meaning the touch sensor is connected to pin 2 on the Arduino board. Also define the pin connected to the LED, here set as GPIO5, meaning the LED is connected to pin 5 on the Arduino board.

setup() Function

```
void setup()
{
    pinMode(TOUCH_PIN, INPUT);   // Set the touch sensor pin as an input
    pinMode(LED_PIN, OUTPUT);    // Set the LED pin as an output
}
```

This line of code uses the pinMode() function to set the Touch_PIN as an input, since the touch sensor needs to send its status to the Arduino. The LED_PIN is set as an output because the LED is controlled through this pin.

loop() function

```
void loop()
{
    // If the touch sensor is triggered (state is LOW), turn on the LED
    if (digitalRead(TOUCH_PIN) == 0)
    {
        digitalWrite(LED_PIN, HIGH);   // Turn on the LED
    }
    // If the touch sensor is not triggered (state is HIGH), turn off the LED
    else
    {
        digitalWrite(LED_PIN, LOW);     // Turn off the LED
    }
}
```

If the voltage level read from TOUCH_PIN is 0, it means the touch sensor is pressed, and the LED turns on. Otherwise, the touch sensor is not pressed, and the LED turns off.
digitalRead(): Reads the voltage level of the specified I/O pin.
if-else: This is a conditional statement. When the condition in the if statement is true, the code inside {} under if is executed. If the condition is false, the code inside {} under else is executed.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson04---PIR Control the LED

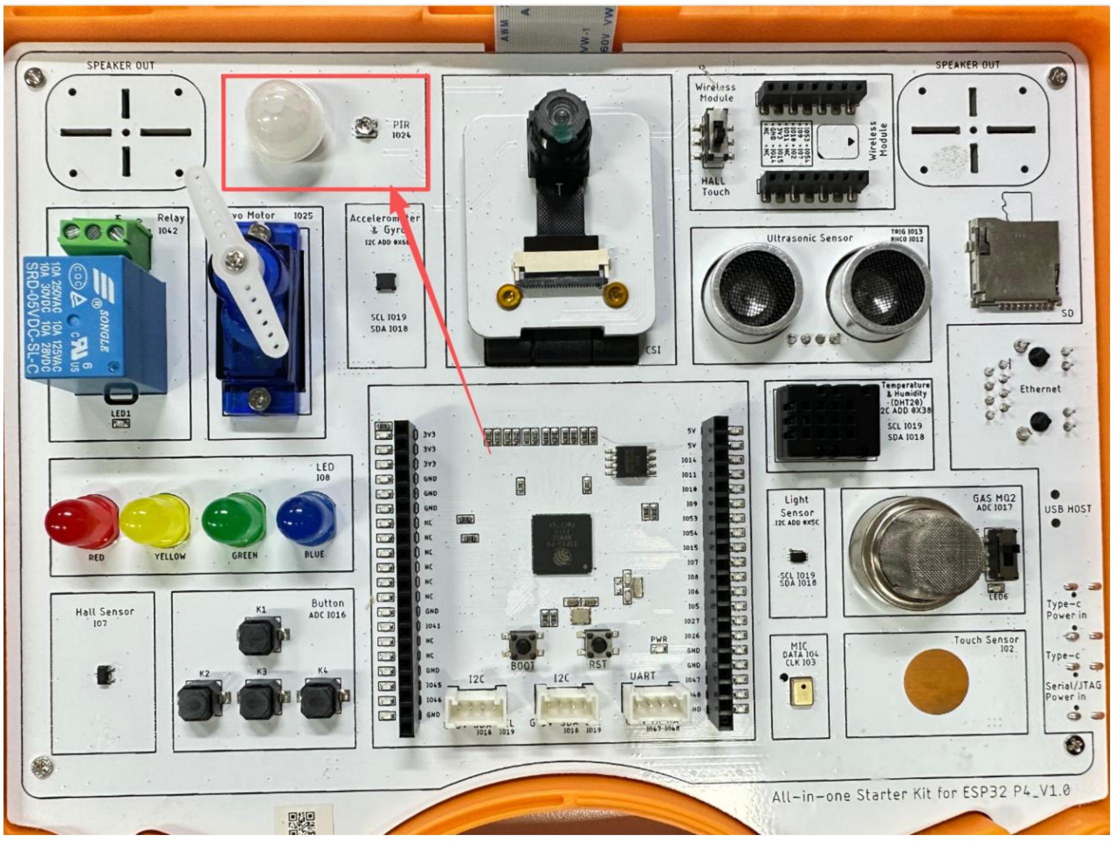
Introduction

In this lesson, we will learn how to use a PIR sensor to control an LED. First, we will understand how the PIR sensor works, then read its current state through a digital pin. When the sensor detects motion, the LED will automatically turn on; otherwise, when no motion is detected, the LED will turn off. This motion-based control method is widely used in smart lighting and security systems.

Learning Goals

1.Understand how the PIR sensor works
2.Learn how to adjust the sensitivity of the PIR sensor
3.Master the use of if-else if statements

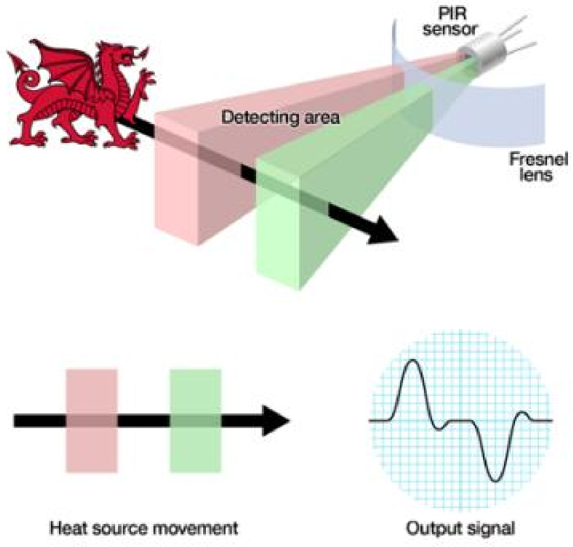
Hardware Used in This Lesson:



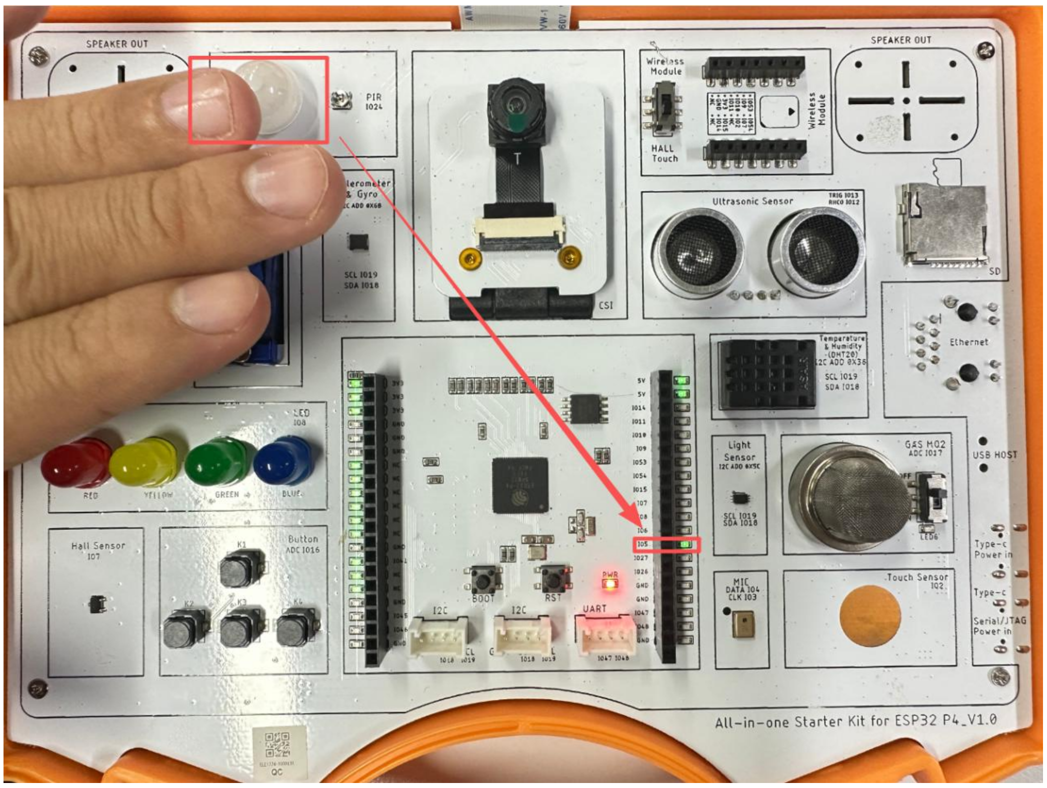
The working principle of PIR Sensor

The PiR motion sensor can detect the change in the amount of infrared radiation reflected on it, which depends on the temperature and surface characteristics of the object in front of the sensor.

When an object such as a person moves within the detection range of the PiR sensor, the temperature at that point in the sensor's field of view will rise from room temperature to body temperature, and this change will return to the sensor and be detected.
PiR sensors are also called motion sensors, which are usually used to detect whether humans have moved in or out of the sensor range. PiR sensors are small in size, cheap, low power consumption, easy to use and will not wear out, so PiR sensors are usually used in electrical appliances.

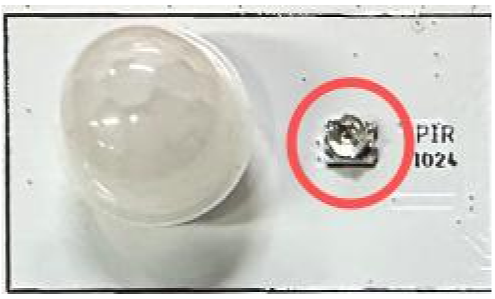


Preview of the Result



After successful execution, when you wave your hand over the PIR sensor, it will be triggered and turn on the LED. The LED will stay on for a short period because the PIR sensor requires some time to reset after being triggered.

Adjusting the Sensitivity of the PIR Sensor:



Turning clockwise increases the sensitivity of the PIR sensor, while turning counterclockwise decreases its sensitivity.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson04_PIR_Control_LED

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"

#define PIR_PIN 24 // Define the GPIO pin number for the PIR sensor (pin 24)
#define LED_PIN 5   // Define the GPIO pin number for the LED indicator (pin 5)
```

Import the Arduino core library, which provides all the basic functions needed in the program. Define the pin connected to the PIR sensor as GPIO24, meaning the PIR sensor is connected to pin 24 on the Arduino board. Also define the pin connected to the LED as GPIO5, meaning the LED is connected to pin 5 on the Arduino board.

setup() function

```
void setup()
{
    pinMode(PIR_PIN, INPUT);   // Set the PIR sensor pin as an input
    pinMode(LED_PIN, OUTPUT); // Set the LED pin as an output
}
```

This line of code uses the pinMode() function to set the PIR_PIN as an input, since the PIR sensor needs to send its status to the Arduino. The LED_PIN is set as an output because the LED is controlled through this pin.

loop() function

```
void loop()
{
    // If the PIR sensor detects motion (PIR_PIN is HIGH), turn off the LED
    if(digitalRead(PIR_PIN) == 1)
    {
        digitalWrite(LED_PIN, LOW);   // Turn off the LED
    }
    // If the PIR sensor does not detect motion (PIR_PIN is LOW), turn on the LED
    else if (digitalRead(PIR_PIN) == 0)
    {
        digitalWrite(LED_PIN, HIGH); // Turn on the LED
    }
}
```

If the voltage level read from PIR_PIN is 1, it means the PIR sensor has not detected anyone, and the LED is off. If the voltage level read from PIR_PIN is 0, it means the PIR sensor has detected someone, and the LED turns on.

if-else if:If the condition is true, the code inside {} under if executes. The else executes when the if condition is false, regardless of any other condition. else if adds an additional condition—its code only executes if the else if condition is met.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson05---Relay

Introduction

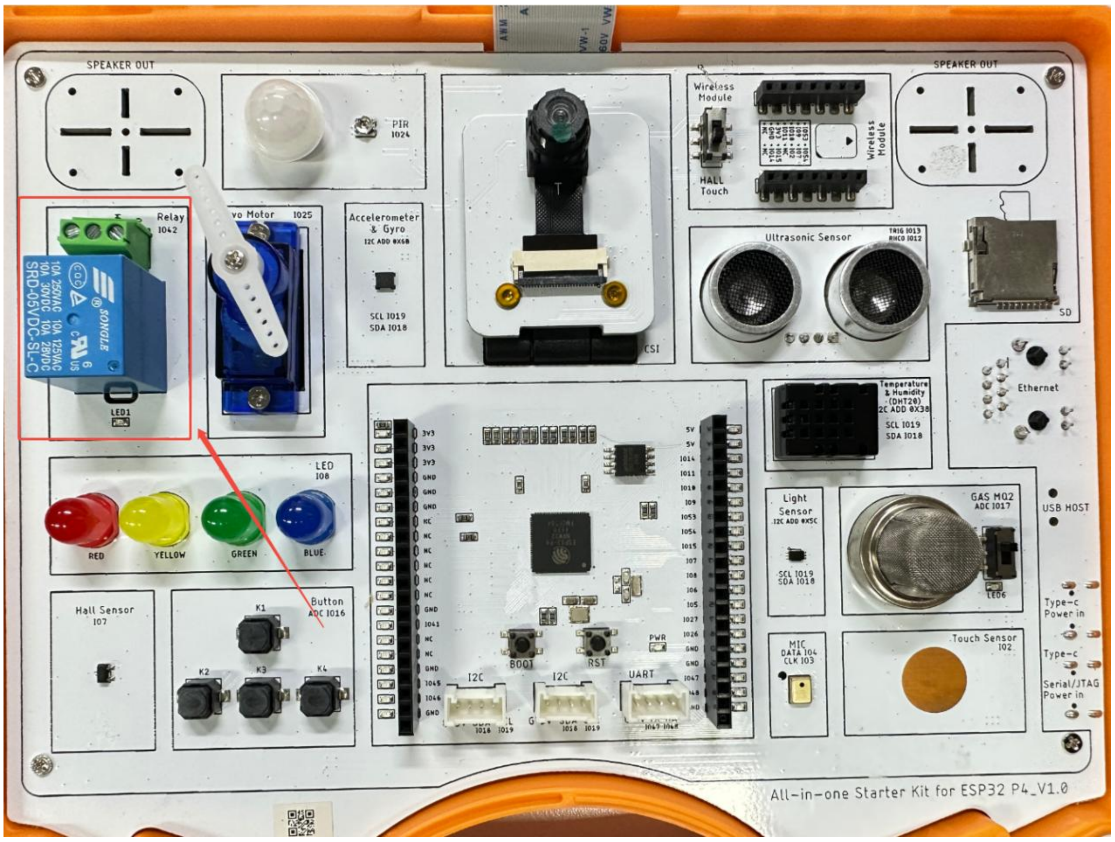
In this lesson, we will learn about relays, focusing on understanding the working principle of a relay. The course will use a simple LED lighting experiment to visually demonstrate and explain the working process of a relay. By the end of this lesson, students will understand the basic principles of relays and master the fundamental methods of using them.

Note: The breadboard, jumper wires, resistors, and LED kit mentioned in the course are not provided.

Learning Goals

1.Understand the function of a relay
2.Master the working principle of a relay

Hardware Used in This Lesson:

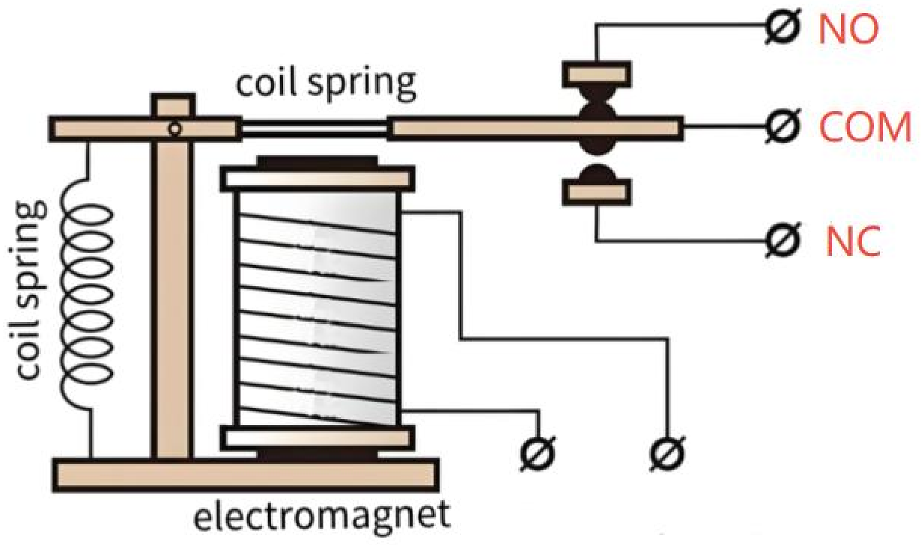


The working principle of Relay

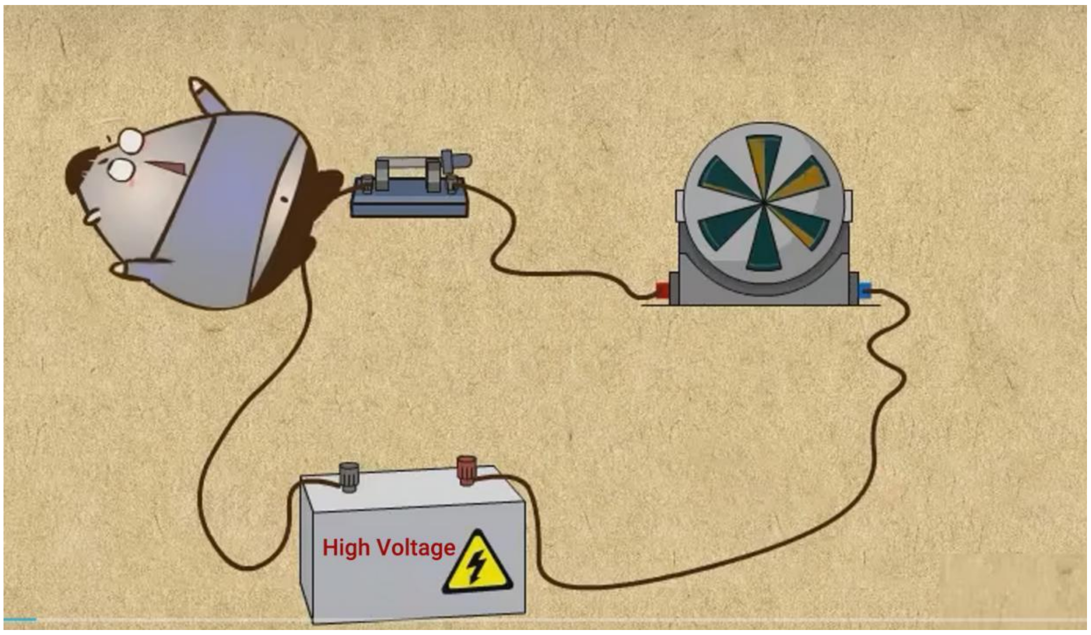
An LED is connected in series with a 330-ohm resistor and attached to the NO (Normally Open) terminal of the relay. The other end of the LED is connected to GND (using the GND pin at the lower right corner of the Arduino Nano R4). The COM (Common) terminal of the relay is connected to 5V (using the 5V pin at the lower right corner of the Arduino Nano R4).

A relay is an electrically controlled switch with a coil and contacts. The common contacts are COM (Common), NC (Normally Closed), and NO (Normally Open). In the default state, when the relay is not powered, the COM is connected to NC and disconnected from NO. When the coil is powered, it generates a magnetic force that moves the contacts, switching COM from NC to NO, so COM is now connected to NO and disconnected from NC.

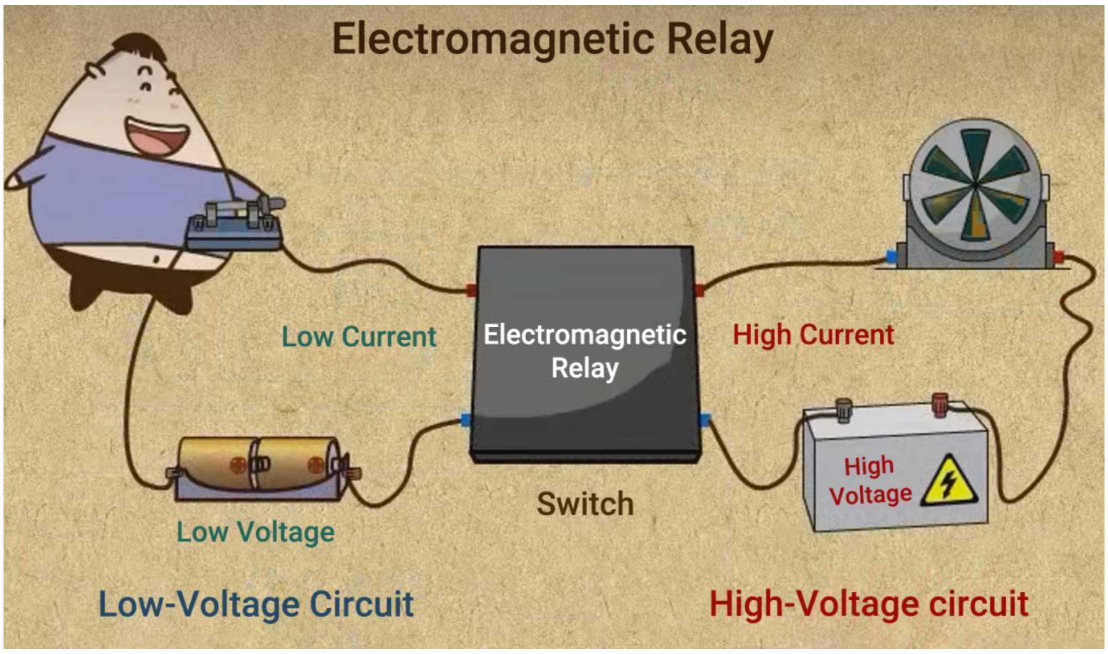
By using this characteristic, a relay can safely control high current or high voltage loads with a small current and low voltage control signal, allowing the circuit to be switched on and off.



Without a relay, directly controlling high-voltage equipment is very dangerous:

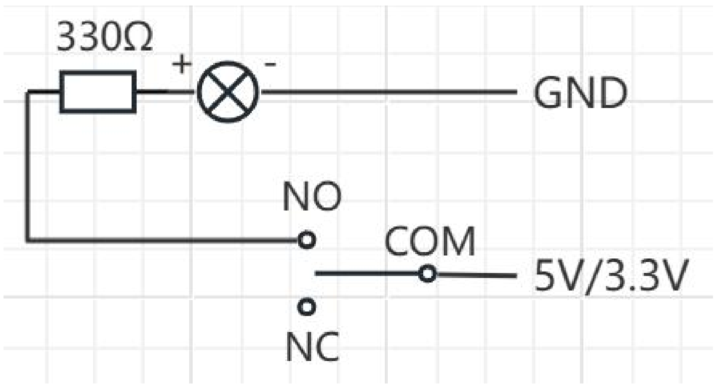


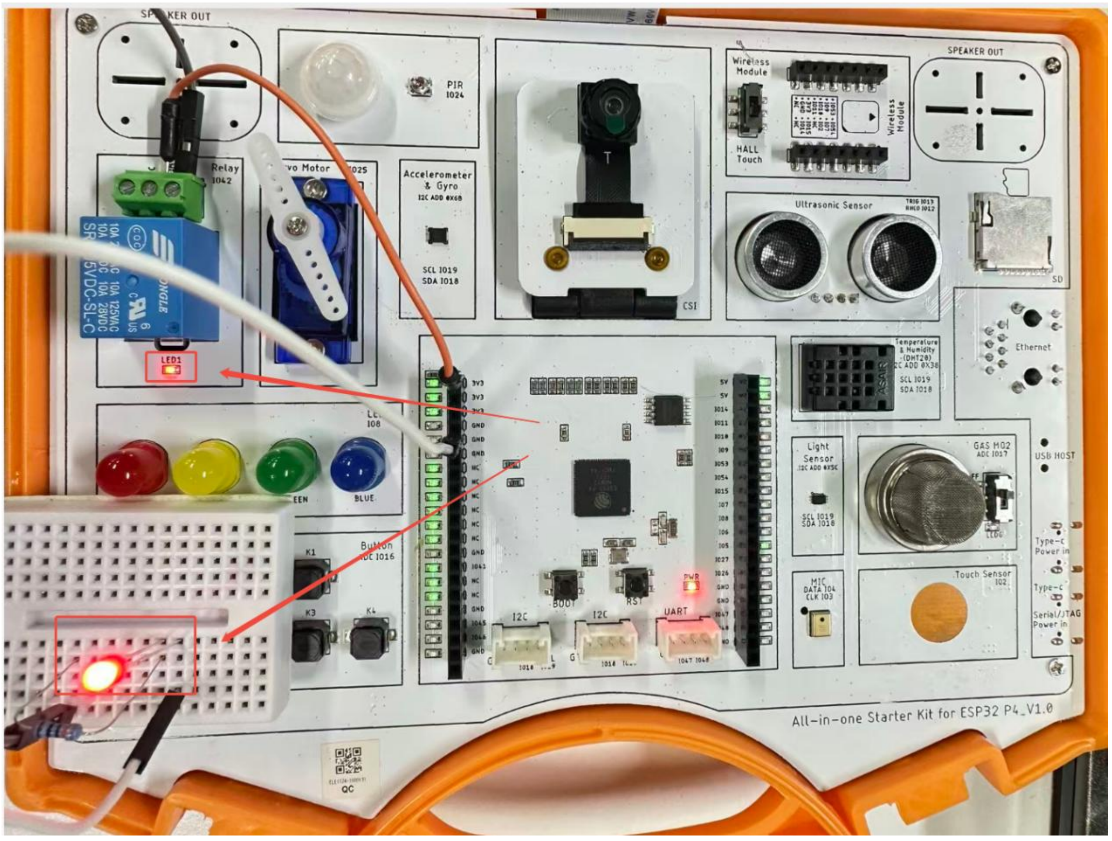
If a relay helps us control high-voltage equipment, the human body is safe:



Preview of the Result

In this lesson, we also built an LED circuit using a breadboard. The circuit setup is shown in the following diagram:





After successful execution, the LED below the relay will turn on for 5 seconds and off for 5 seconds. The circuit on the breadboard we built will also turn on and off every 5 seconds.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson05_Relay

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#define RELAY_PIN 42 // Define the GPIO pin number for controlling the relay (pin 42)
```

Import the Arduino core library, which provides all the basic functions required for the program. Define the pin connected to the relay as GPIO42, meaning the relay is connected to pin 42 on the Arduino board.

setup() function

```
void setup()
{
    pinMode(RELAY_PIN, OUTPUT);   // Set the relay control pin as an output
    digitalWrite(RELAY_PIN, LOW); // Initialize the relay to OFF (LOW)
}
```

We still use the pinMode() function to set RELAY_PIN as output mode and initialize it to a low voltage. This means that after uploading the program, the initial state of the relay is off.

loop() function

```
void loop()
{
    digitalWrite(RELAY_PIN, HIGH); // Turn the relay ON
    delay(5000);                   // Wait for 5000 milliseconds (5 seconds)

    digitalWrite(RELAY_PIN, LOW);  // Turn the relay OFF
    delay(5000);                   // Wait for 5000 milliseconds (5 seconds)
}
```

In the loop function, we set the relay to turn on every 5 seconds and turn off every 5 seconds. On the circuit, you will see the LED light up every 5 seconds and turn off every 5 seconds.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson06---Hall Sensor

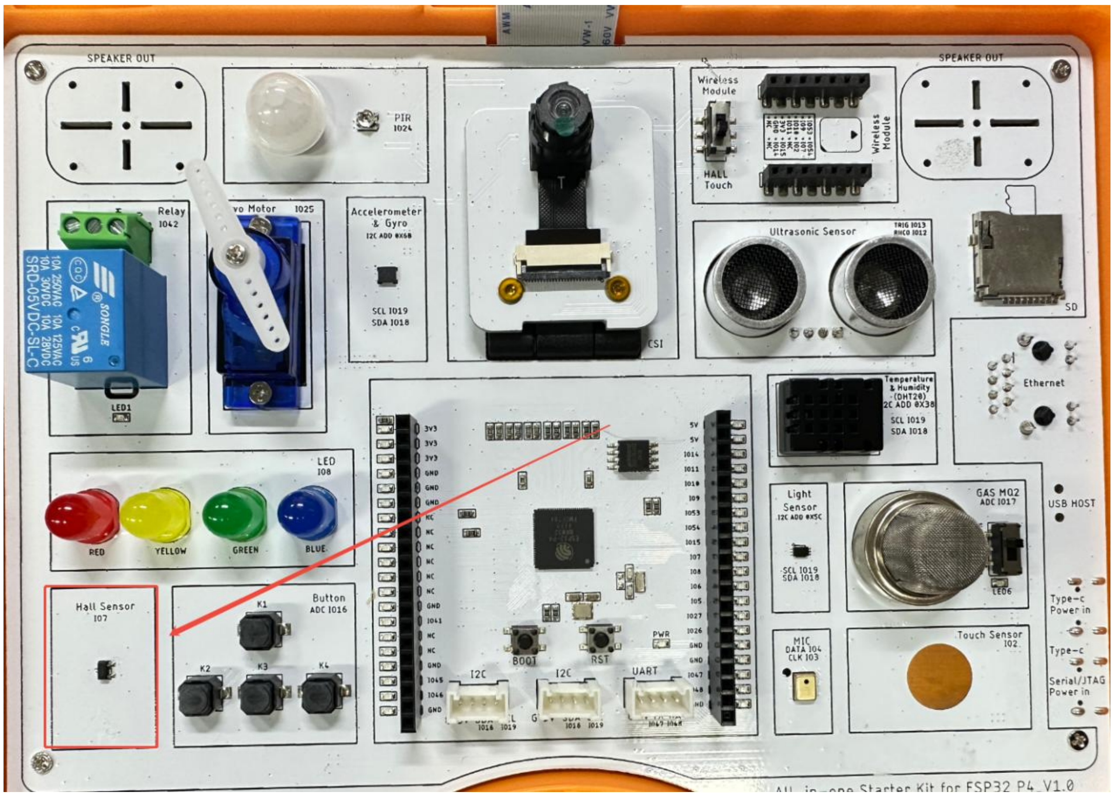
Introduction

In this lesson, we will learn about the Hall sensor, a sensor based on the Hall effect principle. A Hall sensor can detect changes in a magnetic field and convert them into electrical signals. When a magnetic field is present, the sensor generates a voltage signal proportional to the strength of the magnetic field.

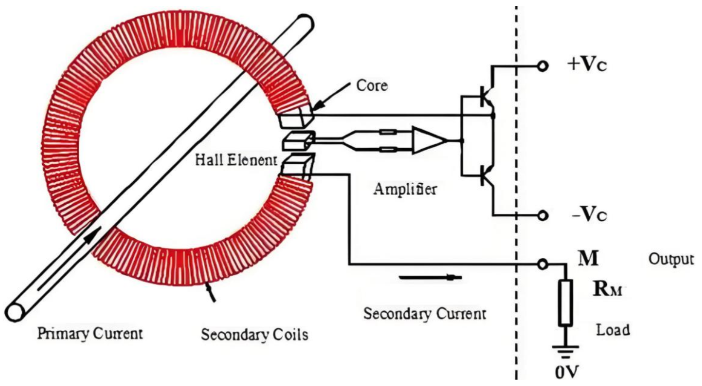
Learning Goals

1.Understand the working principle of the Hall sensor
2.Reinforce the use of if-else if statements
3.Learn how to use code to determine whether a magnet is approaching the Hall sensor

Hardware Used in This Lesson:



The working principle of Hall Sensor



Let's first look at the left side, where there is a red toroidal coil called the secondary compensation coil. Beside it runs a conductor carrying the primary current, which is the current we want to measure.

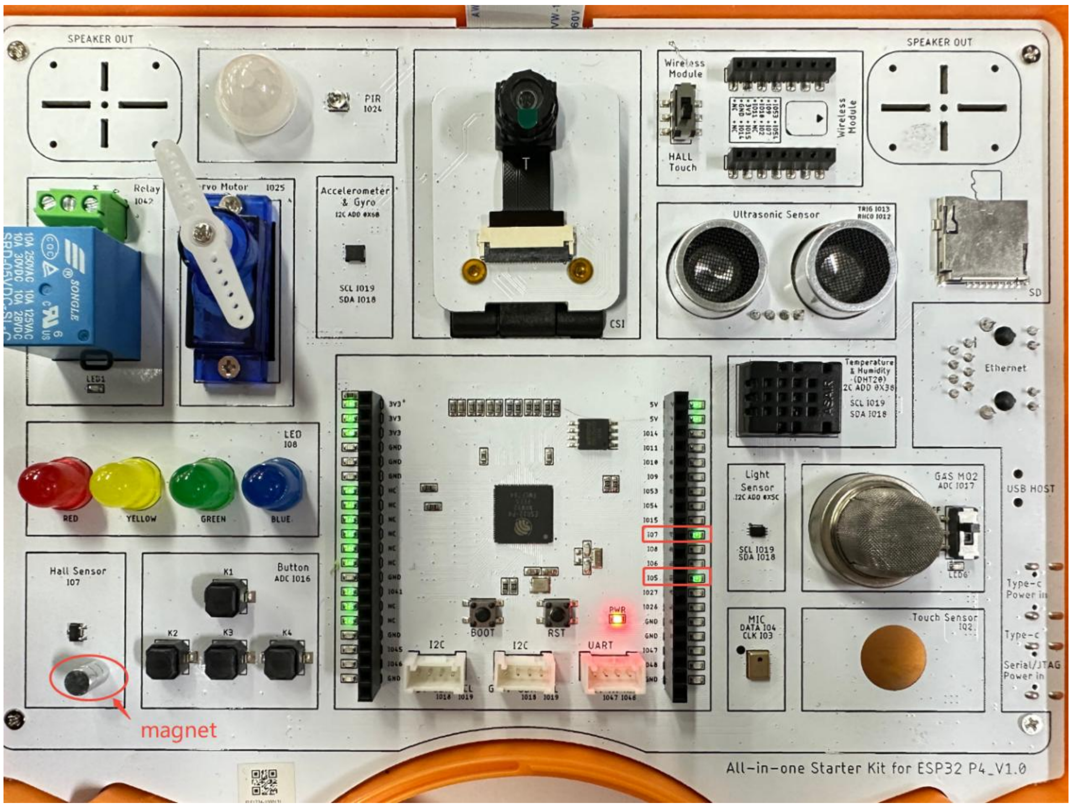
The primary current generates a magnetic field, and this field acts on the central magnetic core. Inside the core sits a Hall element, which plays a crucial role. It can "sense" the changes in the magnetic field within the core. When the magnetic field produced by the primary current reaches the Hall element, it outputs a small electrical signal.

Since this signal is quite weak, it is then fed into an operational amplifier. The amplifier works like a "signal booster", strengthening the Hall element's weak signal.

The amplified signal goes on to control the following circuitry, which generates the secondary compensation current.This current flows through the secondary compensation coil, producing a magnetic field that interacts with the magnetic field created by the primary current.

The two fields counteract each other, keeping the magnetic field inside the core balanced.

Preview of the Result



After running the program, when you bring a magnet close to the Hall sensor, the LEDs corresponding to IO7 and IO5 will both light up. The IO7 LED indicates the status of the Hall sensor, while the IO5 LED is the LED controlled by our code.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson06_Hall_Sensor

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#define HALL_PIN    7    // Define the GPIO pin number for the hall sensor (pin 7)
#define LED_PIN     5    // Define the GPIO pin number for the LED indicator (pin 5)
```

Import the Arduino core library, which provides all the basic functions required for the program. Define the pin connected to the Hall sensor as GPIO7, meaning the Hall sensor is connected to pin 7 on the Arduino board. Also define a controllable LED with pin 5.

setup() function

```
void setup()
{
    pinMode(HALL_PIN, INPUT);    // Set the hall sensor pin as an input
    pinMode(LED_PIN, OUTPUT);    // Set the LED pin as an output
    digitalWrite(LED_PIN, LOW); // Initialize the LED to OFF (LOW)
}
```

The Hall sensor is an input module, so in the initialization function we need to set it to input mode. We also initialize the LED connected to LED_PIN to a low level, so that when the Hall sensor detects a magnet later, it can turn on the LED we want to control.

loop() function

```
void loop()
{
    // If the hall sensor detects a magnetic field (state is HIGH), turn on the LED
    if(digitalRead(HALL_PIN) == 1)
    {
        digitalWrite(LED_PIN, HIGH);   // Turn on the LED
    }
     // If the hall sensor does not detect a magnetic field (state is LOW), turn off the
LED
    else if (digitalRead(HALL_PIN) == 0)
    {
        digitalWrite(LED_PIN, LOW);     // Turn off the LED
        }
}
```

In the loop function, when the value read from the Hall sensor is 1, it means a magnet is approaching, so the LED on pin 5 is set to high and turned on. If the value read from the Hall sensor is 0, it means no magnet is nearby, and the LED is turned off.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson07---Button Control LED

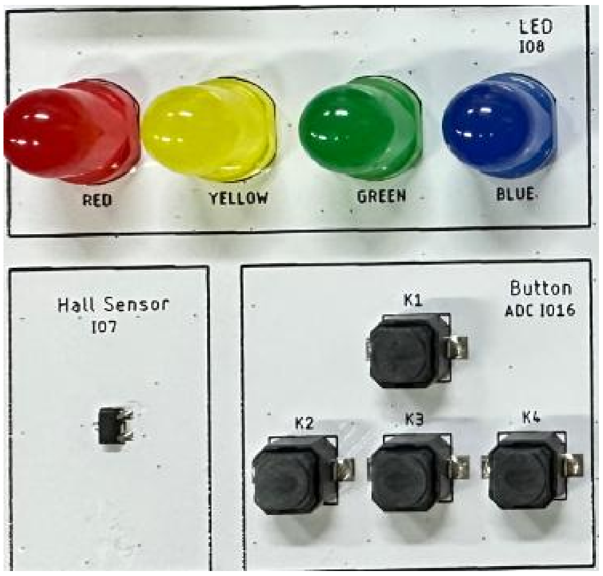
Introduction

In the previous lessons, we learned about the if statement, which allows us to control the flow of a program based on a single condition. In this lesson, we'll build on that knowledge by exploring multi-condition statements. We'll see how Arduino determines and distinguishes which key has been pressed when there are multiple inputs. By the end of this lesson, you will complete a practical project: using three button modules to control the on/off state of three LED lights. This will help you master the basics of multi-condition logic and how to trigger different outcomes based on different conditions.

Learning Goals

1.Understand the functionality and principles of the button module
2.Master the logic of the if-else if-else statement
3.Use the if-else if-else statement to determine which of the three button modules is pressed and light up the corresponding LED
4.Learn how to use logical operators within conditional statements

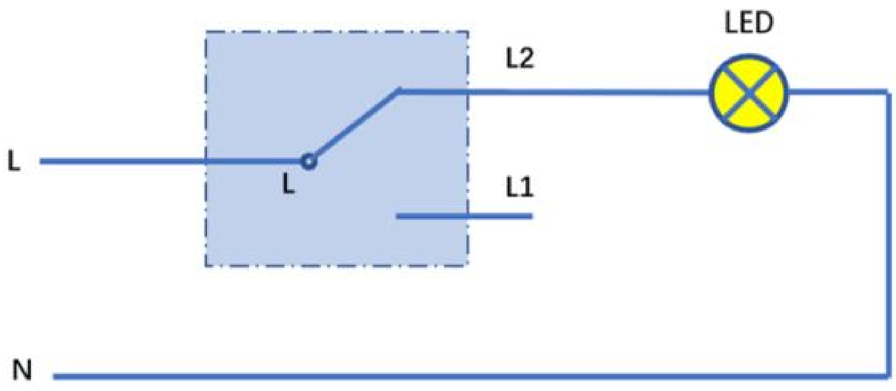
Hardware Used in This Lesson



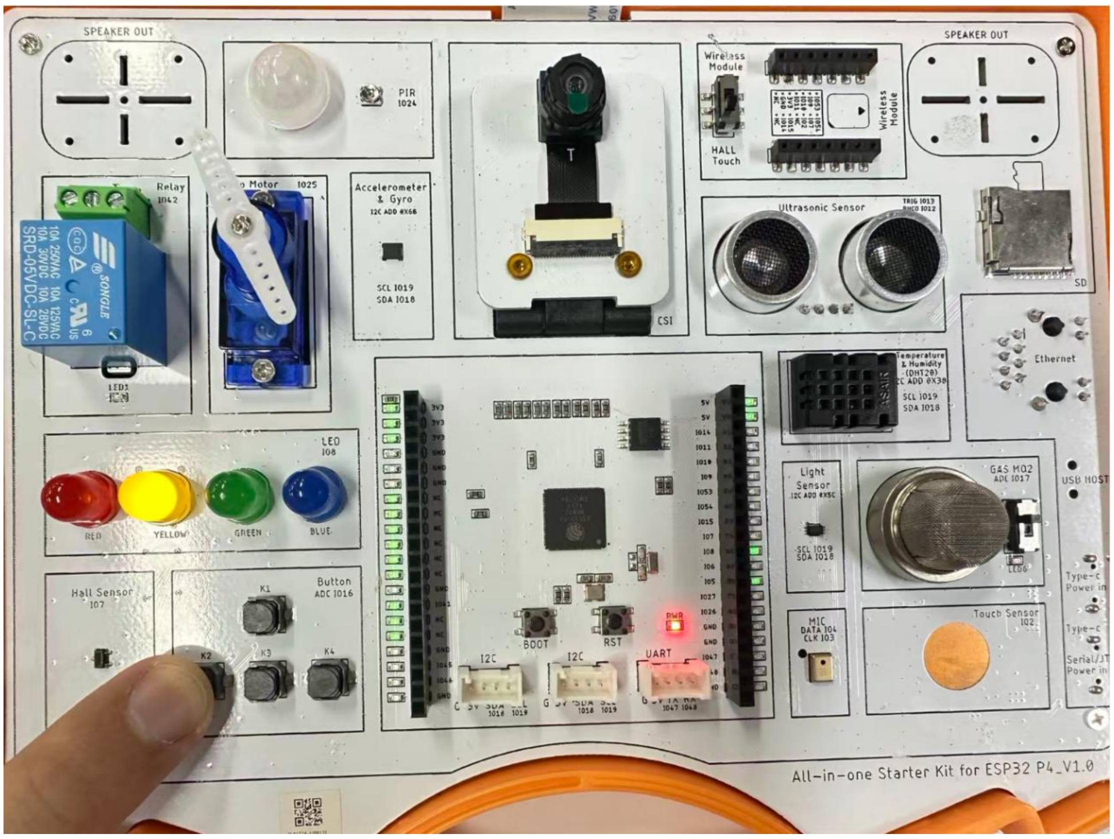
On the All-in-One Starter Kit for the ESP32-P4 development board, we have four buttons:
These four buttons are all connected to the IO16 pin of the P4 microcontroller. IO16 is an analog input pin, which means it can read the analog voltage values on the pin. For our four buttons, pressing different buttons will cause the IO16 pin to return different analog values. By reading these values, the program can determine which button was pressed and perform the corresponding action.

The working principle of Button Module

The button module we used in this lesson is equivalent to a single-pole double-throw switch. As shown in the figure below: When the button is pressed, it is equivalent to the switch hitting the L2 end, and the circuit is turned on; when the button is released, it is equivalent to the switch hitting the L1 end, and the circuit is disconnected.



Preview of the Result



After uploading the code, pressing K1 will turn on the red light, pressing K2 will turn on the yellow light, pressing K3 will turn on the green light, and pressing K4 will turn on the blue light.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson07_Button_Control_LED

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#define LED_NUM       2     // Define the number of LEDs in the NeoPixel strip (2 LEDs)
#define LED_PIN       8     // Define the GPIO pin number for the LED strip (pin 8)
#include <Adafruit_NeoPixel.h>   // Include the Adafruit NeoPixel library
```

Import the Arduino core library and the Adafruit NeoPixel library. We use this library to control the four LEDs, but since the red, yellow, and green LEDs are connected to the same pin, we only need to define two LED counts in total.

setup() function

```
void setup()
{
   Serial.begin(115200);                    // Initialize serial communication at 115200 baud
rate
   strip.begin();                      // Initialize the NeoPixel strip
   strip.show();                       // Ensure the strip is turned off initially
   strip.setBrightness(50);            // Set the brightness of the LEDs (range: 0 to 255)
   analogSetPinAttenuation(16, ADC_11db); // Set the ADC attenuation for the pin 16
to 11db (for a wider input voltage range)
}
```

Serial.begin(115200);:Initializes serial communication with a baud rate of 115200. This is typically used for debugging, although it's not directly used in this particular code.

strip.begin();:Initializes the NeoPixel LED strip.

strip.show();:Ensures that the LED strip is turned off when it's initialized.

strip.setBrightness(50);:Sets the brightness of the LED strip, with a range from 0 (dim) to 255 (bright). Here, it's set to 50, which provides a moderate brightness.

analogSetPinAttenuation(16,ADC_11db);:Sets the ADC (Analog-to-Digital Conversion) attenuation for GPIO pin 16 to 11dB. This expands the input voltage range, allowing it to handle a wider voltage range (for example, 0-3.3V).

loop() function

```
void loop()
{
     uint32_t voltage_mv = analogReadMilliVolts(16);   // Read the voltage from pin 16
in millivolts
     float voltage_v = voltage_mv / 1000.0;              // Convert the voltage to volts
```

uint32_t voltage_mv = analogReadMilliVolts(16);:Reads the voltage value from GPIO pin 16, with the result in millivolts (mV).

analogReadMilliVolts() function is used to obtain the voltage on the pin, and in this code, the voltage is measured in millivolts.
float voltage_v = voltage_mv / 1000.0;:Converts the voltage reading from millivolts to volts. Since the value is in millivolts, dividing by 1000 gives the voltage in volts.

Next, we will use the different voltage ranges to control the color of different LEDs.

```
    // If the voltage is between 2.7V and 3.3V, set the first LED to red color
    if(voltage_v > 2.7 && voltage_v < 3.3)
    {
      strip.setPixelColor(0, strip.Color(255, 0, 0));   // Set the first LED to red
    }
    // If the voltage is less than 2.0V, set the first LED to green color
    else if(voltage_v < 2)
    {
      strip.setPixelColor(0, strip.Color(0, 255, 0));   // Set the first LED to green
    }
    // If the voltage is between 2.4V and 2.7V, set the first LED to blue color
    else if(voltage_v > 2.4 && voltage_v < 2.7)
    {
      strip.setPixelColor(0, strip.Color(0, 0, 255));   // Set the first LED to blue
    }
    // If the voltage is between 2.0V and 2.4V, set the second LED to white color
    else if (voltage_v > 2 && voltage_v < 2.4)
    {
      strip.setPixelColor(1, strip.Color(255, 255, 255));   // Set the second LED to white
    }
    else
    {
      // If the voltage is outside the expected range, turn off both LEDs
      strip.setPixelColor(0, strip.Color(0, 0, 0));   // Turn off the first LED
      strip.setPixelColor(1, strip.Color(0, 0, 0));   // Turn off the second LED
    }

  strip.show();   // Update the LED strip with the new colors
  delay(200);      // Wait for 200 milliseconds before repeating the loop
}
```

if(voltage_v > 2.7 && voltage_v < 3.3):If the voltage is between 2.7V and 3.3V, turn on the red LED.
else if(voltage_v < 2):If the voltage is below 2.0V, turn on the yellow LED.
else if(voltage_v > 2.4 && voltage_v < 2.7):If the voltage is between 2.4V and 2.7V, turn on the green LED.
else if (voltage_v > 2 && voltage_v < 2.4):If the voltage is between 2.0V and 2.4V, turn on the blue LED.
else:If the voltage is outside of the predefined ranges, turn off all LEDs.

Note: The line strip.setPixelColor(0, strip.Color(0, 125, 255)); is used to control the red, yellow, and green LEDs. The three parameters in strip.Color(0, 125, 255) control these LEDs (from left to right). A value of 0 means the LED is off, and 255 means the LED is fully bright.

The line strip.setPixelColor(1, strip.Color(255, 255, 255)); controls the blue LED. Setting all parameters to 255 turns the LED on at full brightness, while setting them to 0 turns the LED off.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson08---Servo

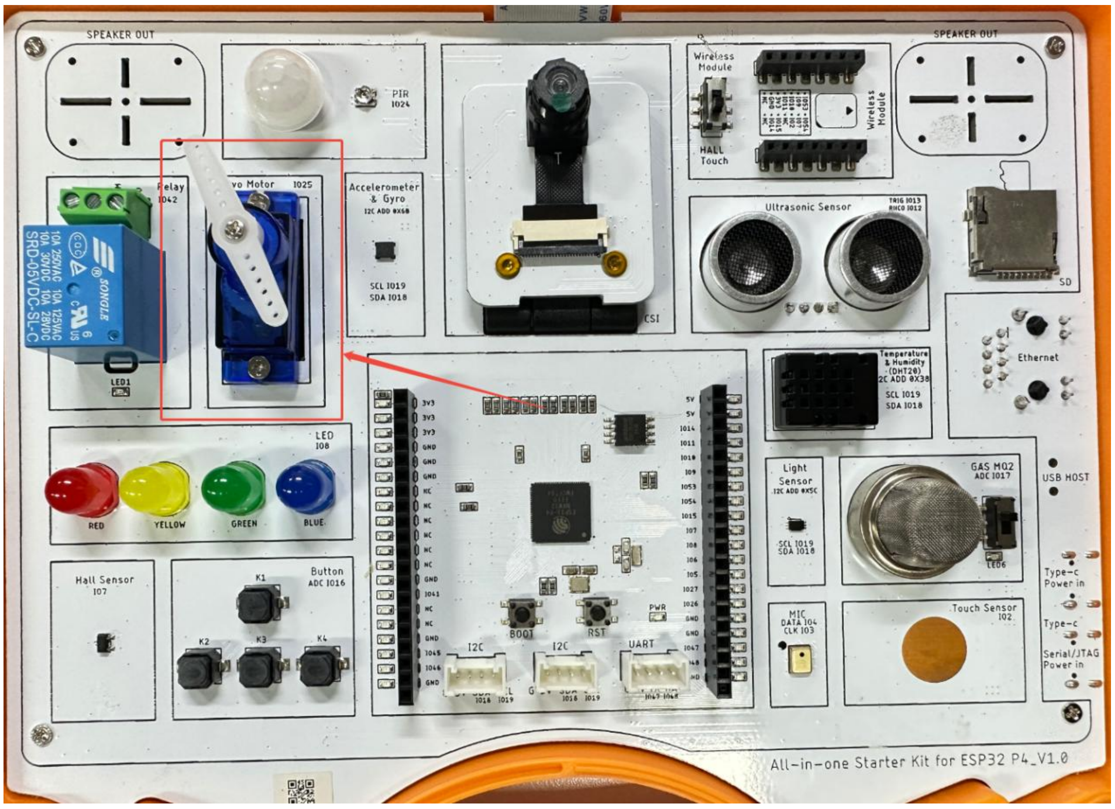
Introduction

In this lesson, we will learn how to use a servo motor module. A servo motor is a module that operates using a PWM (Pulse Width Modulation) signal. We'll focus on understanding what PWM signals are and how they can be used to control the rotation angle of the servo motor.

Learning Goals

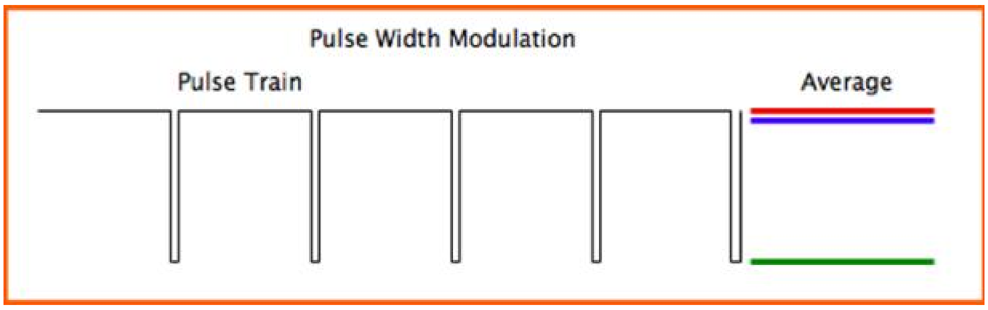
1.Understand what PWM signals are
2.Learn how the servo motor module works
3.Complete a project where the servo motor rotates from 0 to 180 degrees

Hardware Used in This Lesson

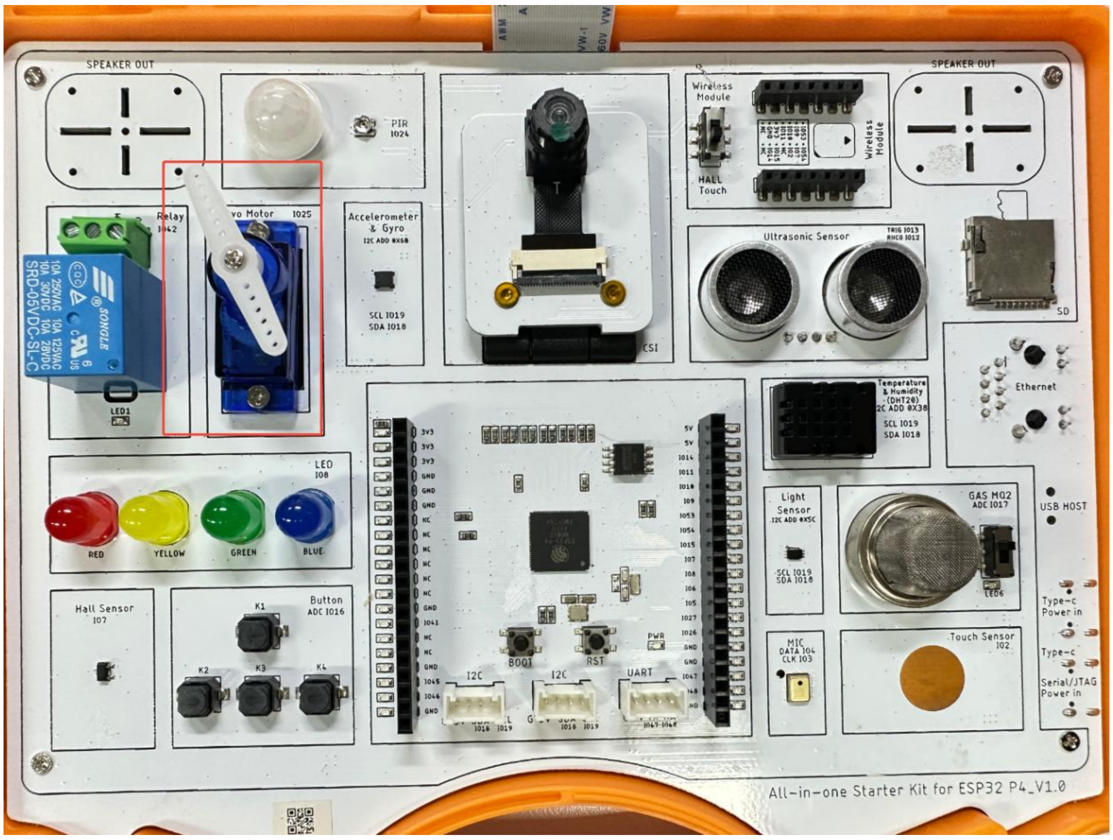


The working principle of Servo Module

The servo motor module is another type of module driven by PWM (Pulse Width Modulation) signals. In previous lessons, we've already learned about PWM pulses, which are a method of controlling the operation of motors or other modules by adjusting the pulse width. The driving principle of a servo motor is that its rotation angle is controlled by the width of the PWM signal: the longer the pulse width, the larger the rotation angle; the shorter the pulse width, the smaller the rotation angle. This allows for precise control of the rotation angle.



Preview of the Result



When the program runs, the servo motor will rotate from 0 to 180 degrees and then back from 180 degrees to 0 degrees.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson08_Servo

Key Explanation:

Include Header Files

```
#include "Arduino.h"
```

```
#include <ESP32Servo.h>
```

Arduino.h：This is the basic Arduino library that provides common functions and types, such as **pinMode()**, **digitalWrite()**, **delay()**, and more.

<ESP32Servo.h>：This is the servo control library for the ESP32, which is the ESP32 version of the Servo library. Since the PWM (Pulse Width Modulation) control on the ESP32 is different from that of the Arduino Uno, a dedicated library is needed for proper servo control.

Create a servo motor object

```
Servo myservo;
```

Declare a Servo type object called myservo to control the servo motor. You will later use this object to call methods that rotate the servo.

Initialize the variables

```
int pos = 0;
```

The variable pos is used to store the current angle of the servo (ranging from 0 to 180 degrees).

setup function

```
void setup()
{
    ESP32PWM::allocateTimer(0);
    ESP32PWM::allocateTimer(1);
    ESP32PWM::allocateTimer(2);
    ESP32PWM::allocateTimer(3);
    myservo.setPeriodHertz(50);
    myservo.attach(25, 1000, 2000);
}
```

Assign a timer to the servo in advance to avoid conflicts. Servo control typically uses a 50Hz PWM frequency (a period of 20ms), meaning a PWM signal is sent every 20ms.

Connect the servo to a GPIO pin

```
myservo.attach(25, 1000, 2000);
```

25 → The GPIO pin on the ESP32 to which the servo signal line is connected.

1000 → The minimum PWM pulse width in microseconds (µs), corresponding to 0° for the servo.

2000 → The maximum PWM pulse width in microseconds (µs), corresponding to 180° for the servo.

For most servos, the PWM pulse width range is from 1000µs to 2000µs, representing the range of 0° to 180°.

loop function

```
void loop()
{
    for (pos = 0; pos <= 180; pos += 1) {
        myservo.write(pos);
        delay(15);
    }
    for (pos = 180; pos >= 0; pos -= 1) {
        myservo.write(pos);
        delay(15);
    }
}
```

Here, we use a for loop to gradually increase the servo's angle from 0, incrementing by 1. This allows the servo to slowly rotate from 0° to 180°. Similarly, we use another for loop to decrease the angle from 180, decrementing by 1, which makes the servo slowly rotate back from 180° to 0°.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson09---Display

Introduction

In this lesson, we will learn how to display text on the screen of the All-in-One Starter Kit for the ESP32-P4. First, we need to understand how the screen works and how to turn on its backlight. Then, we will learn how to set the text position and font size to display content correctly.

Learning Goals

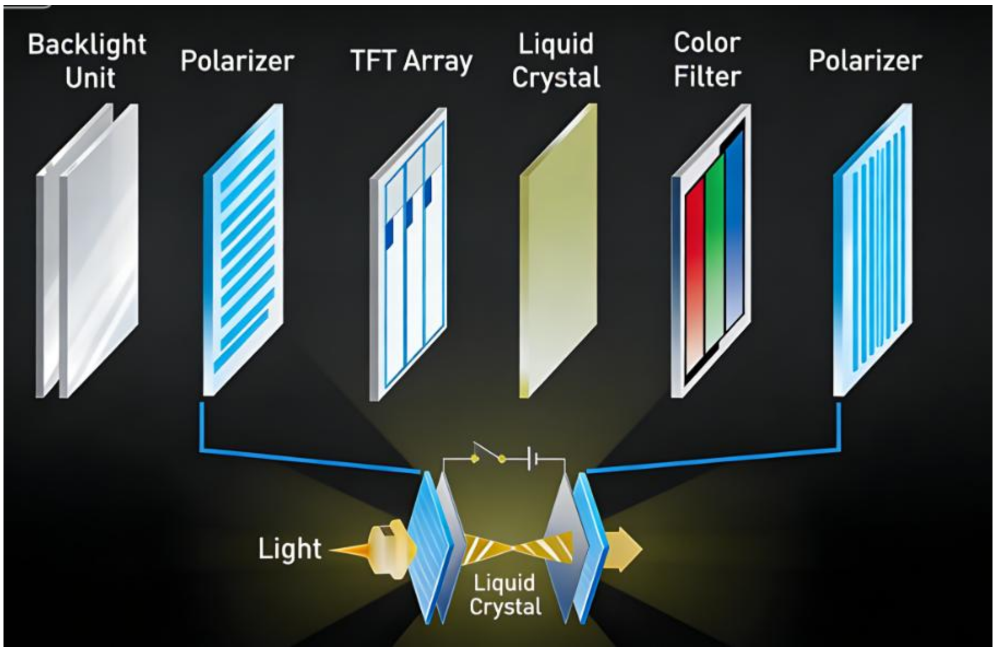
1.Understand how the screen works

2.Complete a project that displays text on the screen

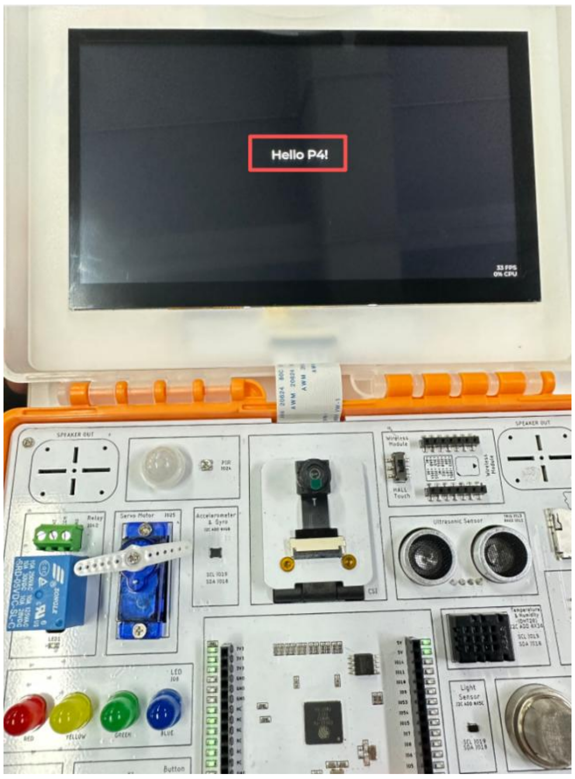
Hardware Used in This Lesson

The working principle of Display

The core principle of an IPS screen lies in its backlight-driven liquid crystal shutter mechanism. Light from the backlight first passes through a polarizing filter, turning it into linearly polarized light. The TFT array then precisely applies an electric field, causing the liquid crystal molecules to rotate in-plane, which controls how much of the polarized light passes through. This modulated light then passes through a color filter to produce red, green, and blue pixel light. Finally, it goes through a second polarizer, converting the electrical signals into visible color images. This in-plane rotation design gives IPS screens wide viewing angles and excellent color reproduction.



Preview of the Result

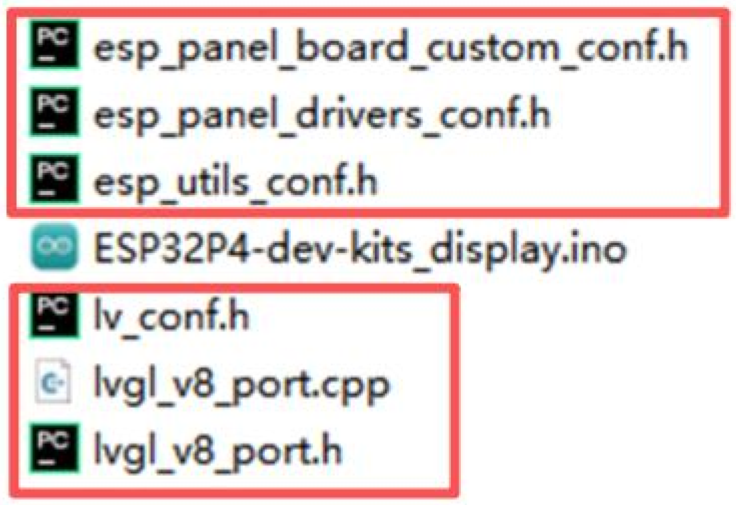


After running the program, the screen will display "Hello P4!"

Complete Code:

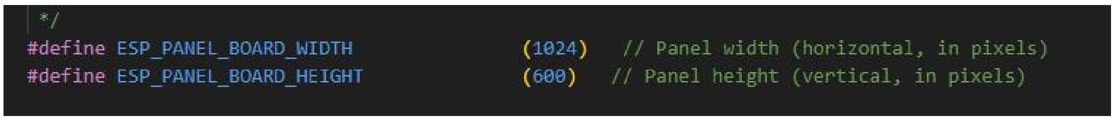
Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson09_Display

In the code you downloaded, there are six configuration files included. Next, we'll go through each one and explain what they do.



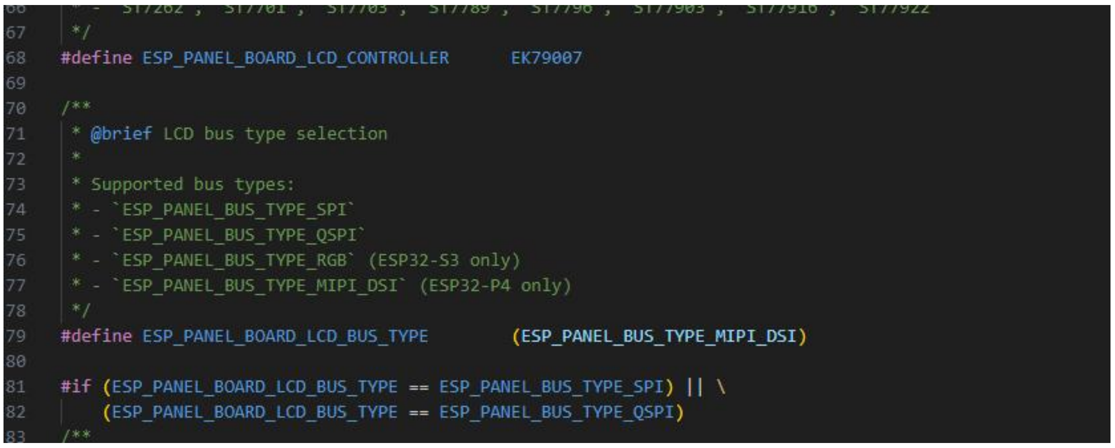
esp_panel_board_custom_conf.h
esp_panel_drivers_conf.h
esp_utils_conf.h
ESP32P4-dev-kits_display.ino
lv_conf.h
lvgl_v8_port.cpp
lvgl_v8_port.h

In the **esp_panel_board_custom_conf.h** file, the screen's width and height are defined.



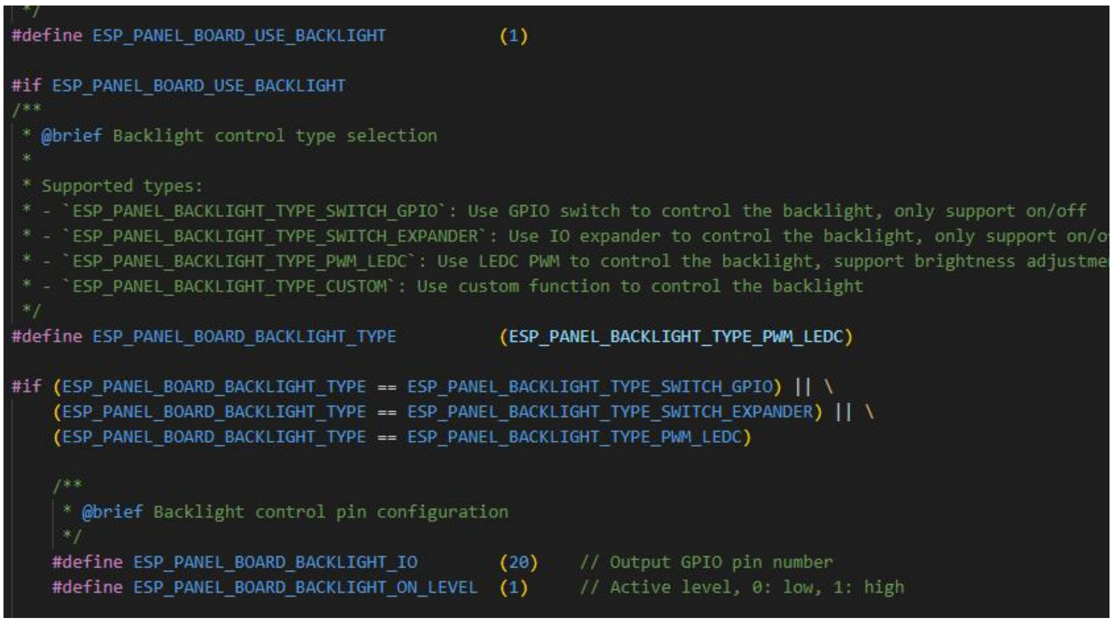
```
 */
#define ESP_PANEL_BOARD_WIDTH               (1024)   // Panel width (horizontal, in pixels)
#define ESP_PANEL_BOARD_HEIGHT              (600)   // Panel height (vertical, in pixels)
```

Screen enable settings, along with the chip model and connection interface.



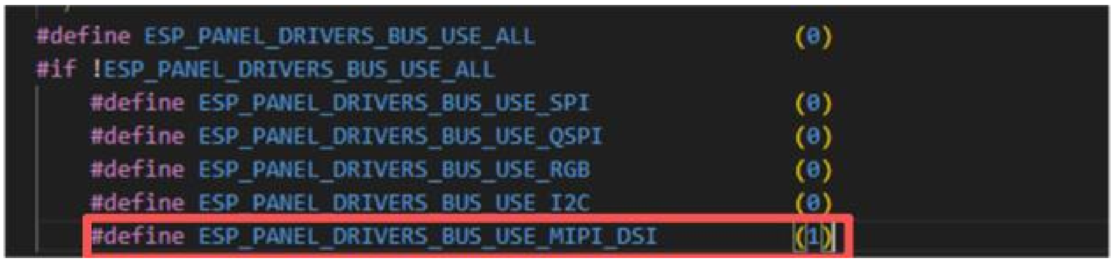
```
 * - `ST7262`, `ST7701`, `ST7703`, `ST7789`, `ST7796`, `ST7903`, `ST7916`, `ST7922`
 */
#define ESP_PANEL_BOARD_LCD_CONTROLLER      EK79007

/**
 * @brief LCD bus type selection
 *
 * Supported bus types:
 * - `ESP_PANEL_BUS_TYPE_SPI`
 * - `ESP_PANEL_BUS_TYPE_QSPI`
 * - `ESP_PANEL_BUS_TYPE_RGB` (ESP32-S3 only)
 * - `ESP_PANEL_BUS_TYPE_MIPI_DSI` (ESP32-P4 only)
 */
#define ESP_PANEL_BOARD_LCD_BUS_TYPE        (ESP_PANEL_BUS_TYPE_MIPI_DSI)

#if (ESP_PANEL_BOARD_LCD_BUS_TYPE == ESP_PANEL_BUS_TYPE_SPI) || \
    (ESP_PANEL_BOARD_LCD_BUS_TYPE == ESP_PANEL_BUS_TYPE_QSPI)
/**
```

Touch functionality enable and touch chip model, touch chip connection method, touch I²C configuration, and backlight settings.



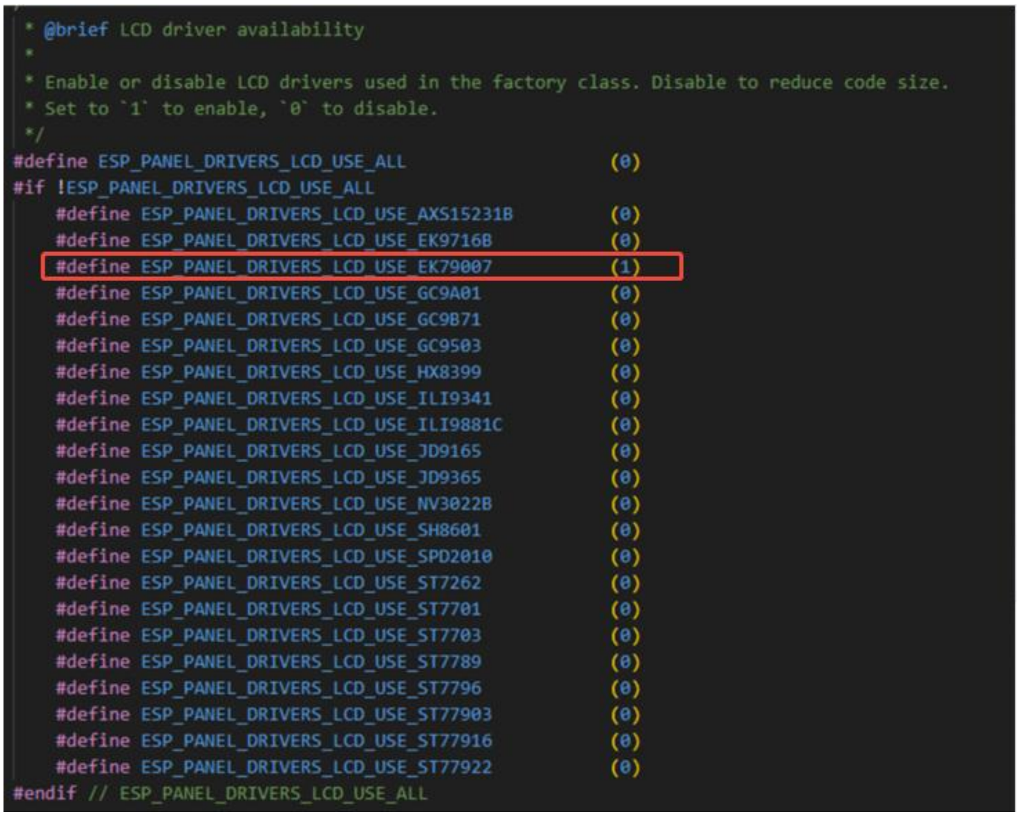
```
 */
#define ESP_PANEL_BOARD_USE_BACKLIGHT          (1)

#if ESP_PANEL_BOARD_USE_BACKLIGHT
/**
 * @brief Backlight control type selection
 *
 * Supported types:
 * - `ESP_PANEL_BACKLIGHT_TYPE_SWITCH_GPIO`: Use GPIO switch to control the backlight, only support on/off
 * - `ESP_PANEL_BACKLIGHT_TYPE_SWITCH_EXPANDER`: Use IO expander to control the backlight, only support on/o
 * - `ESP_PANEL_BACKLIGHT_TYPE_PWM_LEDC`: Use LEDC PWM to control the backlight, support brightness adjustme
 * - `ESP_PANEL_BACKLIGHT_TYPE_CUSTOM`: Use custom function to control the backlight
 */
#define ESP_PANEL_BOARD_BACKLIGHT_TYPE         (ESP_PANEL_BACKLIGHT_TYPE_PWM_LEDC)

#if (ESP_PANEL_BOARD_BACKLIGHT_TYPE == ESP_PANEL_BACKLIGHT_TYPE_SWITCH_GPIO) || \
    (ESP_PANEL_BOARD_BACKLIGHT_TYPE == ESP_PANEL_BACKLIGHT_TYPE_SWITCH_EXPANDER) || \
    (ESP_PANEL_BOARD_BACKLIGHT_TYPE == ESP_PANEL_BACKLIGHT_TYPE_PWM_LEDC)

    /**
     * @brief Backlight control pin configuration
     */
    #define ESP_PANEL_BOARD_BACKLIGHT_IO       (20)    // Output GPIO pin number
    #define ESP_PANEL_BOARD_BACKLIGHT_ON_LEVEL (1)     // Active level, 0: low, 1: high
```

In the **esp_panel_drivers_conf** file, the screen's connection method is configured.



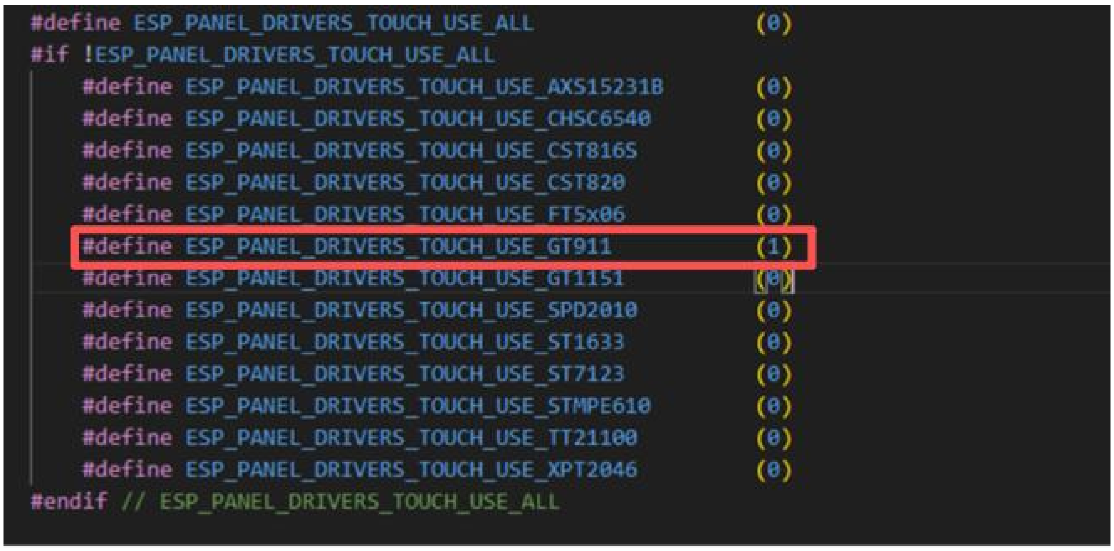
```
#define ESP_PANEL_DRIVERS_BUS_USE_ALL              (0)
#if !ESP_PANEL_DRIVERS_BUS_USE_ALL
    #define ESP_PANEL_DRIVERS_BUS_USE_SPI          (0)
    #define ESP_PANEL_DRIVERS_BUS_USE_QSPI         (0)
    #define ESP_PANEL_DRIVERS_BUS_USE_RGB          (0)
    #define ESP_PANEL_DRIVERS_BUS_USE_I2C          (0)
    #define ESP_PANEL_DRIVERS_BUS_USE_MIPI_DSI     (1)
```

Select the screen chip



```
 * @brief LCD driver availability
 *
 * Enable or disable LCD drivers used in the factory class. Disable to reduce code size.
 * Set to `1` to enable, `0` to disable.
 */
#define ESP_PANEL_DRIVERS_LCD_USE_ALL               (0)
#if !ESP_PANEL_DRIVERS_LCD_USE_ALL
    #define ESP_PANEL_DRIVERS_LCD_USE_AXS15231B     (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_EK9716B       (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_EK79007       (1)
    #define ESP_PANEL_DRIVERS_LCD_USE_GC9A01        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_GC9B71        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_GC9503        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_HX8399        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ILI9341       (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ILI9881C      (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_JD9165        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_JD9365        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_NV3022B       (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_SH8601        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_SPD2010       (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST7262        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST7701        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST7703        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST7789        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST7796        (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST77903       (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST77916       (0)
    #define ESP_PANEL_DRIVERS_LCD_USE_ST77922       (0)
#endif // ESP_PANEL_DRIVERS_LCD_USE_ALL
```

Select the touch chip



```
#define ESP_PANEL_DRIVERS_TOUCH_USE_ALL                  (0)
#if !ESP_PANEL_DRIVERS_TOUCH_USE_ALL
    #define ESP_PANEL_DRIVERS_TOUCH_USE_AXS15231B        (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_CHSC6540         (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_CST816S          (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_CST820           (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_FT5x06           (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_GT911            (1)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_GT1151           (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_SPD2010          (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_ST1633           (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_ST7123           (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_STMPE610         (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_TT21100          (0)
    #define ESP_PANEL_DRIVERS_TOUCH_USE_XPT2046          (0)
#endif // ESP_PANEL_DRIVERS_TOUCH_USE_ALL
```

The files **esp_utils_conf.h**, **lv_conf.h**, **lvgl_v8_port.cpp**, and **lvgl_v8_port.h** contain the driver code needed to run the example.

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#include <esp_display_panel.hpp>
#include <lvgl.h>
#include <lvgl_v8_port.h>
```

Arudino.h:Provides basic functions like pinMode(), digitalWrite(), Serial, and more

esp_display_panel.hpp: The official ESP library for display panels, used to control LCDs, touchscreens, and backlights

lvgl.h:The LVGL graphics library, used to create UI elements such as buttons, text, and images

lvgl_v8_port.h:The LVGL adaptation layer for ESP32, responsible for connecting the LCD and touchscreen

Using a namespace

```
// Use namespaces from the ESP display panel library
using namespace esp_panel::drivers;
using namespace esp_panel::board;
```

Shorten long names so you don't have to type them out every time

setup() function

```
void setup()
{
    // Configure GPIO pins 32 and 33 as output pins
    pinMode(32, OUTPUT);
    pinMode(33, OUTPUT);
    // Set initial pin states
    digitalWrite(32, 0);
    digitalWrite(33, 1);
    // Start serial communication for debugging
    Serial.begin(115200);
    Serial.println("Initializing board");
```

Set GPIO pins 32 and 33 as outputs, then set pin 32 to LOW and pin 33 to HIGH. This configures the LCD circuit and turns on the backlight. Also, set the serial baud rate to 115200 and print "Initializing board."

Create a Board object

```
  // Create a new Board object which handles display, touch, and backlight
    Board *board = new Board();
```

Create a Board class object that manages the LCD, touchscreen, and backlight—essentially acting as a hardware controller for the board.

Initialize and start up the development board

```
     // Initialize board hardware
    board->init();

    // Start the board (initialize peripherals like LCD, touch, etc.)
    board->begin();
```

Initialize the hardware, including GPIO pins, the LCD driver, and the touch driver. Once initialized, start running the peripherals.

Initialize the LVGL graphics library

```
     // Initialize LVGL graphics library with LCD and touch drivers
    lvgl_port_init(board->getLCD(), board->getTouch());
```

Link the LVGL graphics library to the hardware so it can draw the UI on the screen. The touchscreen functionality won't be used in this lesson.

Turn off the backlight

```
     // Turn off backlight initially (brightness = 0)
    board->getBacklight()->setBrightness(0);
```

Start by turning off the screen backlight. This prevents the screen from flashing before the UI fully loads.

Get the current screen object

```
     // Get the currently active LVGL screen object
    lv_obj_t *scr = lv_scr_act();
```

Get the currently active screen, which will hold all the UI elements

Set the screen's background color

```
    // Set screen background color to black
    lv_obj_set_style_bg_color(scr, lv_color_black(), LV_PART_MAIN);
```
Set the screen's background color to black

Set the screen's background opacity

```
     // Make the background fully opaque
    lv_obj_set_style_bg_opa(scr, LV_OPA_COVER, LV_PART_MAIN);
```
Set the background to fully opaque

Lock LVGL to prevent other tasks from modifying the UI

```
     // Lock LVGL to safely modify UI elements
    lvgl_port_lock(-1);
```
Lock LVGL to prevent multi-threading conflicts, since LVGL can run simultaneously on the UI thread, touch thread, and screen refresh thread

Create a text label and set its content

```
     // Create a label object on the screen
    lv_obj_t *label_1 = lv_label_create(scr);

    // Set the text displayed on the label
    lv_label_set_text(label_1, "Hello P4!");
```
Create a Label and set its text to "Hello P4!"

Set the text color and font size

```
    // Set label text color to white
    lv_obj_set_style_text_color(label_1, lv_color_white(), LV_STATE_DEFAULT);

    // Set label font size to Montserrat 30
    lv_obj_set_style_text_font(label_1, &lv_font_montserrat_30, 0);
```
Set the text color to white, use the "Montserrat" font, and set the font size to 30

Set the text position

```
   // Align the label to the center of the screen, slightly upward (-20 pixels)
    lv_obj_align(label_1, LV_ALIGN_CENTER, 0, -20);
```
Center the text on the screen and shift it upward by 20 pixels

Unlock LVGL to allow other tasks to modify the UI

```
    // Unlock LVGL after UI updates are done
    lvgl_port_unlock();
    // Small delay to ensure UI setup is stable
    delay(200);
```
Allow LVGL to continue running.The 0.2-second delay ensures that the UI initialization is complete.

Turn on the screen backlight

```
    // Turn on the backlight to full brightness (100%)
        board->getBacklight()->setBrightness(100);
    }
```

After the UI initialization is complete, turn on the backlight, and the screen will light up.

loop() function

```
    void loop()
    {
        // Main loop does nothing for now, just waits
        delay(1000);
    }
```

In the loop function, do nothing and simply wait for one second. This is because the UI is automatically refreshed by the LVGL task in the initialization function.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson10---Touch Display

Introduction

In the previous lesson, we learned about screen drivers and successfully displayed "Hello P4!" on the screen. In this lesson, we will build on that by using the touch functionality of the screen. We will design four touch buttons, and when each corresponding button is touched, the LED will light up.

Learning Goals

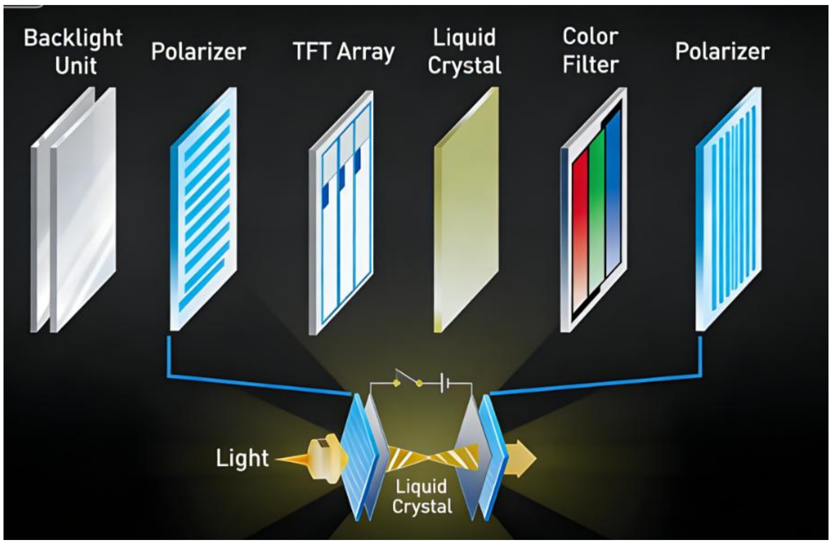
1.Understand the methods for driving the touch screen
2.Complete a project where touching the screen turns on the corresponding LED light

Hardware Used in This Lesson

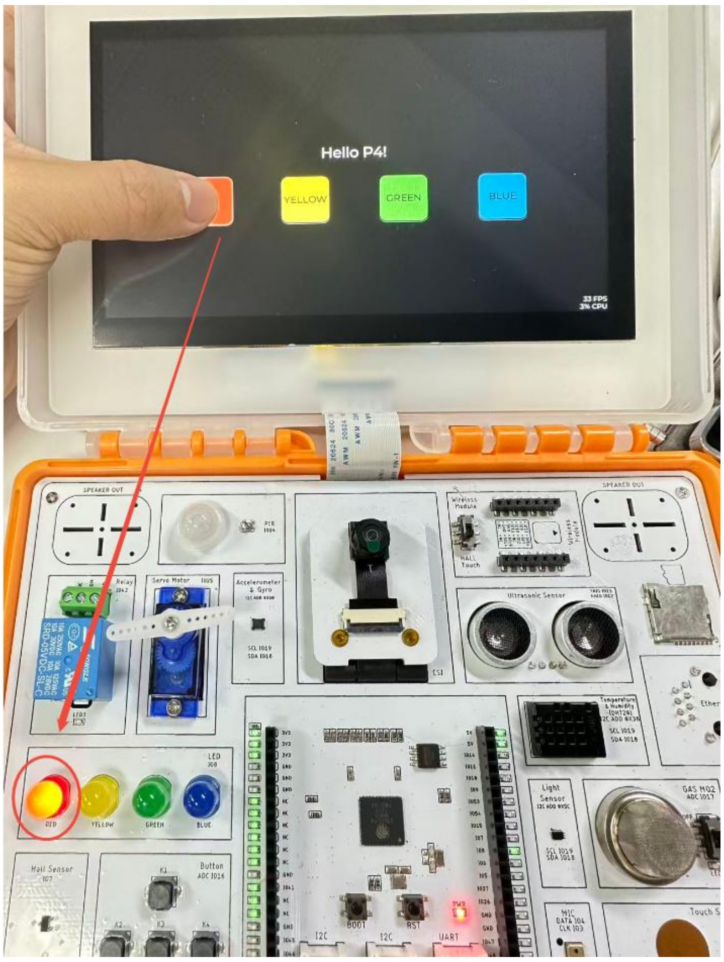
The working principle of Display

The core principle of an IPS screen is its backlight-driven liquid crystal shutter mechanism. Light from the backlight first passes through a polarizing filter, producing linearly polarized light. The

TFT array then applies an electric field with precision, causing the liquid crystal molecules to rotate in-plane, which controls how much of the polarized light passes through. This modulated light then goes through a color filter to generate red, green, and blue pixel light, and finally passes through a second polarizer, converting electrical signals into visible color images. This in-plane rotation design gives IPS screens wide viewing angles and vibrant color reproduction.



Preview of the Result



After running the program, the screen will display "Hello P4!" along with four buttons in red, yellow, green, and blue. Pressing each button will light up the corresponding LED on the All-in-One Starter Kit for ESP32-P4 development board.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson10_Touch_display

The configuration file for this lesson is the same as in Lesson 9, so we won't go over it again here. If you're unsure about the downloaded configuration file, you can review the content from Lesson 9.

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#include <esp_display_panel.hpp>
#include <lvgl.h>
#include <lvgl_v8_port.h>
#include "Adafruit_NeoPixel.h"
#define LED_NUM      2    // Define the number of LEDs in the NeoPixel strip
#define LED_PIN      8    // Define the GPIO pin number for the LED indicator
Adafruit_NeoPixel strip(LED_NUM, LED_PIN, NEO_GRB + NEO_KHZ800);   // Initialize NeoPixel strip
```

Arduino.h: The standard Arduino library, providing basic functions for the ESP32 board, such as pinMode, digitalWrite, delay, and more.
esp_display_panel.hpp: Provides support for ESP32 display panels, typically used to control the screen.
lvgl.h: LVGL is a lightweight graphics library for creating user interfaces on embedded systems.
lvgl_v8_port.h:The LVGL port file that allows LVGL to work with ESP32 hardware.
Adafruit_NeoPixel.h:A library for controlling NeoPixel RGB LED strips.
LED_NUM defines the number of LEDs in the NeoPixel strip, which is 2 in this case.
LED_PIN defines the GPIO pin used to control the LEDs, here it is GPIO 8.
The **Adafruit_NeoPixel** library is used to initialize the LED strip, setting the number of LEDs and the control pin.

Use a namespace

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

Use a namespace to simplify code and avoid having to type long names repeatedly.

Initialize the LDO power supply

```
esp_err_t board_p4_ldo_init()
{
    esp_err_t err = ESP_OK;
    esp_ldo_channel_handle_t ldo3_handle = NULL;
    esp_ldo_channel_config_t ldo3_cfg = {
        .chan_id = 3,               // LDO Channel 3
        .voltage_mv = 2500,       // Set to 2500mV (2.5V)
    };
    Serial.println("Initializing LDO3 to 2.5V...");
    err = esp_ldo_acquire_channel(&ldo3_cfg, &ldo3_handle);
    if (err != ESP_OK) {
        Serial.printf("LDO3 Power Error: %s\n", esp_err_to_name(err));
        return err;
    } else {
        Serial.println("LDO3 Power enabled successfully.");
    }
    // --- Power Configuration (LDO4 for I2C/touch pull up) ---
    esp_ldo_channel_handle_t ldo4_handle = NULL;
    esp_ldo_channel_config_t ldo4_cfg = {
        .chan_id = 4,               // LDO Channel 4
        .voltage_mv = 3300,       // Set to 3300mV (3.3V)
    };
    Serial.println("Initializing LDO4 to 3.3V...");
    err = esp_ldo_acquire_channel(&ldo4_cfg, &ldo4_handle);
    if (err != ESP_OK) {
        Serial.printf("LDO4 Power Error: %s\n", esp_err_to_name(err));
        return err;
    } else {
        Serial.println("LDO4 Power enabled successfully.");
    }
    return ESP_OK;
}
```

Initialize the LDO power channels. Here, two LDO channels are used: LDO3 is set to 2.5V, and LDO4 is set to 3.3V. The LDO provides a stable power supply for certain ESP32 components.

Button event handler

```
// Button event handler function
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 */
    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)   // When the button is pressed
    {
        lv_obj_add_state(btn, LV_STATE_PRESSED); /* Add pressed state to the button */
          // 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);
        switch(led_index)   // Control the LED color based on the button index
        {
            case 0:
                strip.setPixelColor(0, strip.Color(255, 0, 0)); // Red
                break;
            case 1:
                strip.setPixelColor(0, strip.Color(0, 255, 0)); // Green
                break;
            case 2:
                strip.setPixelColor(0, strip.Color(0, 0, 255)); // Blue
                break;
            case 3:
                strip.setPixelColor(1, strip.Color(0, 255, 0)); // Green (second LED)
                break;
        }

    }
    else if (code == LV_EVENT_RELEASED)   // When the button is released
    {
         lv_obj_clear_state(btn, LV_STATE_PRESSED); /* Remove pressed state from the
button */
        // set_single_led_status(led_index, 0x00); /* Turn off LEDs */
        // MAIN_INFO("BTN[%d] released", led_index);
        strip.setPixelColor(0, strip.Color(0, 0, 0));   // Turn off LED 0
        strip.setPixelColor(1, strip.Color(0, 0, 0));   // Turn off LED 1
    }
    strip.show();   // Update the LED strip
}
```

This is a button event handler that manages button press and release events. When a button is pressed, it sets the color of the NeoPixel LED strip based on the button's index (led_index). When the button is released, it turns off the LED.

setup() function

```
    //void setup()
    {
        pinMode(32, OUTPUT);   // Set GPIO 32 as output
        pinMode(33, OUTPUT);   // Set GPIO 33 as output
        digitalWrite(32, 0);   // Set GPIO 32 to low
        digitalWrite(33, 1);   // Set GPIO 33 to high
        Serial.begin(115200); // Start serial communication

        Serial.println("Initializing board");

        board_p4_ldo_init();   // Initialize the LDO power channels

        strip.begin();                 // Initialize the NeoPixel strip
        strip.show();                   // Initialize the LEDs
        strip.setBrightness(50);   // Set the brightness of the LEDs

        Board *board = new Board();   // Initialize the display board
        board->init();   // Initialize the display hardware

        board->begin();   // Start the display board

        lvgl_port_init(board->getLCD(), board->getTouch());   // Initialize LVGL port for touch and
display
        board->getBacklight()->setBrightness(0);   // Turn off the backlight initially

        lv_obj_t *scr = lv_scr_act();   // Get the current screen object

        lv_obj_set_style_bg_color(scr, lv_color_black(), LV_PART_MAIN);   // Set the background
color to black
        lv_obj_set_style_bg_opa(scr, LV_OPA_COVER, LV_PART_MAIN);   // Set the background
opacity

        lv_obj_t *label_1 = lv_label_create(scr);   // Create a label for the text
        lv_label_set_text(label_1, "Hello P4!");   // Set the text for the label
        lv_obj_set_style_text_color(label_1, lv_color_white(), LV_STATE_DEFAULT);   // Set text
color to white
        lv_obj_align(label_1, LV_ALIGN_CENTER, 0, -50);   // Center align the label
        lv_obj_set_style_text_font(label_1, &lv_font_montserrat_30, 0);   // Set the font for the
label

        // Create an array of buttons
        lv_obj_t *buttons[4];
        lv_color_t btn_color[4] = {
            lv_palette_main(LV_PALETTE_RED),        // Red button color
            lv_palette_main(LV_PALETTE_YELLOW),     // Yellow button color
            lv_palette_main(LV_PALETTE_GREEN),      // Green button color
            lv_palette_main(LV_PALETTE_BLUE)        // Blue button color
        };
        const char *button_texts[] = {"RED", "YELLOW", "GREEN", "BLUE"};   // Button text labels
```

In the setup function, tasks like initializing the backlight and setting the font were already covered in the previous lesson, so we won't go over that code again. The main focus here is creating four buttons, assigning each button its text and color, and adding event callbacks so that pressing a button changes the corresponding LED's color.

Create four buttons

```
for (int i = 0; i < 4; i++)
{
    buttons[i] = lv_btn_create(scr);
```

The for loop runs four times, creating one button (buttons[i]) in each iteration.
lv_btn_create(scr):This function creates a button object on the current screen scr. The scr is the active screen, usually obtained in setup() using lv_scr_act().Here, buttons[i] stores the pointer to each button in an array.

Set the size and position of each button.

```
lv_obj_set_size(buttons[i], 100, 100);
lv_obj_set_pos(buttons[i], (162 + (i * 200)), 300);
```

lv_obj_set_size(buttons[i], 100, 100):Sets the width and height of the button to 100 pixels, making each button a square.
lv_obj_set_pos(buttons[i], (162 + (i * 200)), 300):Sets the initial position of the button. The buttons are arranged horizontally along the X-axis, with a fixed Y-axis position of 300 pixels.
162 + (i * 200):The value of i determines the X-coordinate, spacing the buttons evenly.As i goes from 0 to 3, the buttons appear at X positions 162, 362, 562, and 762, all aligned at Y = 300.

Set the corner radius of the button

```
lv_obj_set_style_radius(buttons[i], 15, LV_STATE_DEFAULT);
```

lv_obj_set_style_radius(buttons[i], 15, LV_STATE_DEFAULT):Sets the button's corner radius to 15 pixels, giving it rounded edges.
LV_STATE_DEFAULT specifies that this style applies to the button's default (normal) state.

Create a label on the button and set its text

```
lv_obj_t *label = lv_label_create(buttons[i]);
lv_label_set_text(label, button_texts[i]);
lv_obj_center(label);
```

lv_obj_t *label = lv_label_create(buttons[i]);:Creates a label and attaches it to the button. The label represents the text on the button.
lv_label_set_text(label, button_texts[i]);:Sets the label's text. button_texts[i] comes from an array containing "RED", "YELLOW", "GREEN", and "BLUE", corresponding to each button's color.
lv_obj_center(label);:Centers the label on the button, so the text appears in the middle of the button.

Set the button's background color and the label's text color

```
lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_DEFAULT);
lv_obj_set_style_text_color(buttons[i], lv_color_black(), LV_STATE_DEFAULT);
```

lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_DEFAULT): Sets the button's background color. btn_color[i] is an array of colors, with each button assigned a different color (e.g., the first button is red, the second is yellow, and so on). LV_STATE_DEFAULT applies this style to the button's normal state.

lv_obj_set_style_text_color(buttons[i], lv_color_black(), LV_STATE_DEFAULT): Sets the text color on the button's label to black.

Set the button's pressed-state style, including its background color and text color

```
lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_PRESSED);
lv_obj_set_style_text_color(buttons[i], lv_color_black(), LV_STATE_PRESSED);
```

lv_obj_set_style_bg_color(buttons[i], btn_color[i], LV_STATE_PRESSED): Sets the button's background color when it is pressed. btn_color[i] is the predefined color for each button.

lv_obj_set_style_text_color(buttons[i], lv_color_black(), LV_STATE_PRESSED): Sets the button's text color to black while the button is pressed.

Add an event callback to the button

```
lv_obj_add_event_cb(buttons[i], button_event_handler, LV_EVENT_ALL, (void *)(intptr_t)i);
```

button_event_handler is the event handler function we defined earlier, which is called when the button is pressed, released, or otherwise interacted with.

LV_EVENT_ALL means the callback will respond to all events, including press and release.

(void *)(intptr_t)i passes the button's index i as user data, allowing the event handler to determine which button was pressed and perform different actions, such as changing the LED color.

Unlock the LVGL port and set the backlight brightness

```
lvgl_port_unlock();    // Unlock LVGL port
delay(200);    // Short delay
board->getBacklight()->setBrightness(100);    // Turn the backlight to 100
```

lvgl_port_unlock():Releases the LVGL port, allowing the graphics library to continue updating the user interface.

delay(200):Adds a 200-millisecond delay to allow the display content to stabilize.

board->getBacklight()->setBrightness(100):Sets the backlight brightness to 100, turning the backlight on. The brightness can be adjusted as needed; in this case, it is set to 100.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson11---Ultrasonic

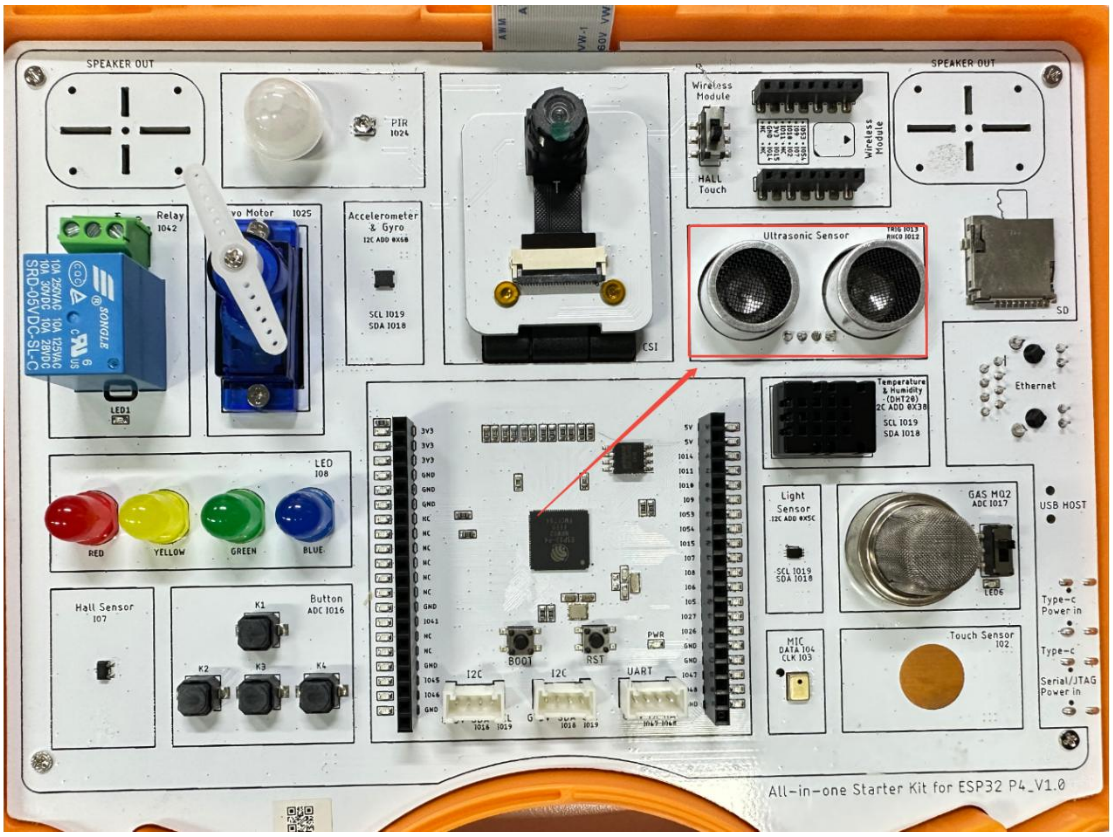
Introduction

In this lesson, we'll learn how to use an ultrasonic sensor. As we all know, ultrasonic modules are commonly used for distance measurement—but have you ever wondered how they actually determine distance? By the end of this lesson, you'll understand the working principle behind ultrasonic distance sensing, and you'll use code to control the hardware to emit sound waves, receive the reflected signal, and calculate the distance based on the response.

Learning Goals

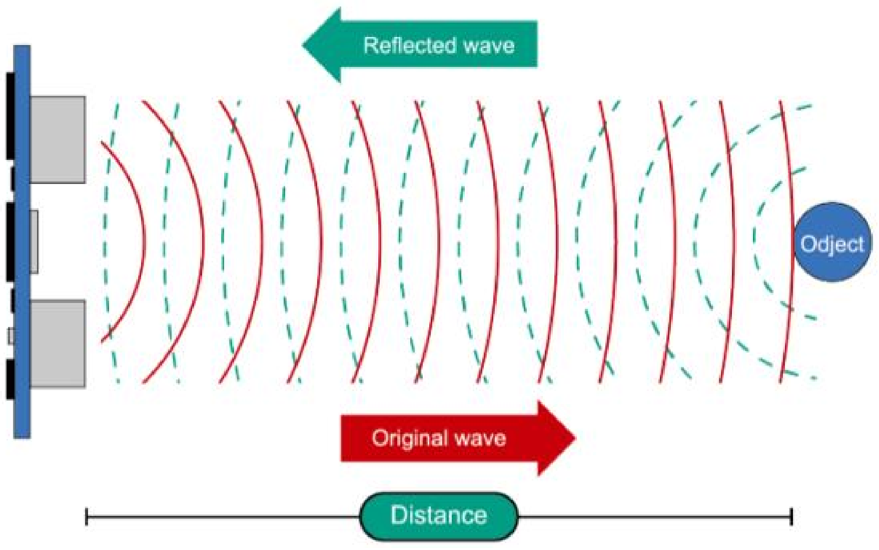
1.Understand the working principle of ultrasonic distance measurement
2.Write and run an example program for ultrasonic distance measurement to achieve real-time distance measurement.

Hardware Used in This Lesson



The working principle of Ultrasonic Module

As shown in the diagram below, when the ultrasonic sensor is operating, it first sends out a burst of ultrasonic waves through the transmitting pin (Trig). When these sound waves hit an obstacle in front of the sensor, they are reflected back and detected by the receiving pin (Echo).



How the Ultrasonic Sensor Works:

1.Transmission Stage

The program triggers the sensor's transmit pin to send a short ultrasonic pulse and records the exact moment this pulse is emitted (the transmission time).

2.Propagation and Reflection

The ultrasonic wave travels through the air. When it hits an object in front of the sensor, it is reflected back.

3.Reception Stage

As soon as the receiving pin detects the returning echo, the program immediately records the current time (the reception time).

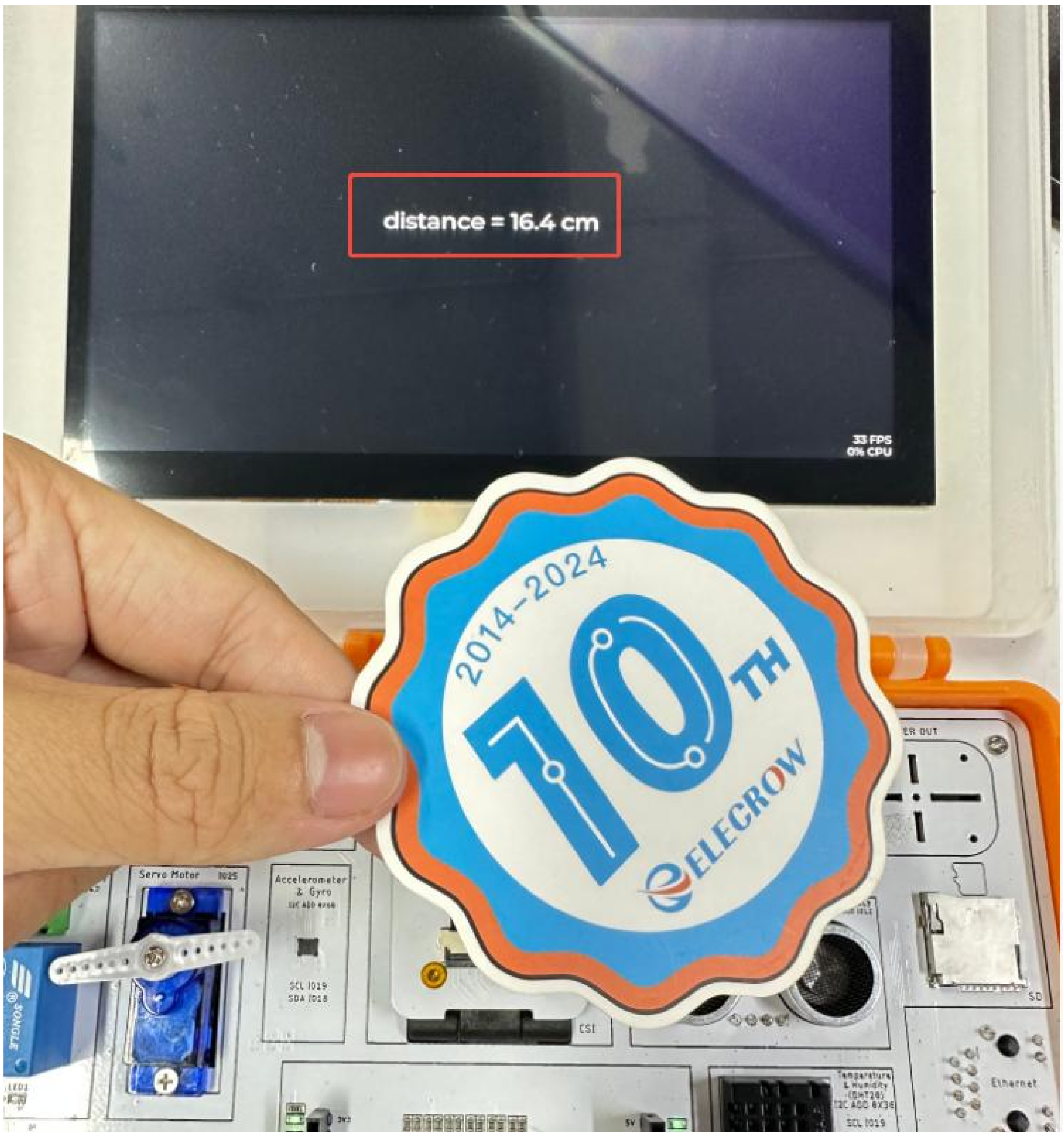
4.Time Difference Calculation

By calculating the difference between the transmission time and the reception time, we obtain the total round-trip travel time of the ultrasonic wave.

Distance Calculation Formula:Distance = (Speed of sound * Total travel time) / 2

It's important to note that the sound wave makes a round trip—traveling to the obstacle and back—so the result must be divided by 2 to get the actual distance to the object. The speed of sound in air is approximately 343 m/s.

Preview of the Result



After running the program, the ultrasonic sensor will continuously measure the distance between the obstacle and the sensor, displaying the results in real-time on the screen.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson11_Ultrasonic

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#include <esp_display_panel.hpp>
#include <lvgl.h>
```

```
#include <lvgl_v8_port.h>
#include <HCSR04.h>
```

Arduino.h: The standard Arduino library, which includes basic functions
esp_display_panel.hpp: The ESP32 display control library, used to manage the display
lvgl.h & lvgl_v8_port.h: LVGL library, used for creating and controlling graphical user interfaces
HCSR04.h: Ultrasonic sensor library, used for distance measurement

Global Variables and Initialization

```
// Create a buffer to hold the formatted distance string
char buffer[32];
// Define the trigger and echo pins for the ultrasonic sensor
const byte triggerPin = 13;
const byte echoPin = 12;
// Initialize the ultrasonic distance sensor
UltraSonicDistanceSensor distanceSensor(triggerPin, echoPin);
// Create a label object for displaying the distance on the screen
lv_obj_t *label_1 = NULL;
```

buffer[32]: A buffer used to store the formatted distance string. It holds the distance value as text for display purposes.
triggerPin & echoPin: These define the trigger and echo pins for the ultrasonic sensor. The triggerPin sends the pulse to the sensor, while the echoPin receives the reflected signal.
distanceSensor: Creates an UltraSonicDistanceSensor object to interact with the ultrasonic sensor. This object handles the communication and distance calculation based on the sensor's signals.
label_1: An LVGL label object used to display the distance value on the screen. The label shows the updated distance value that the ultrasonic sensor detects.

Use a namespace

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

Use a namespace to simplify code and avoid having to type long names repeatedly.

setup function

```
void setup()
{
    // Set up the backlight pins for the display
    pinMode(32, OUTPUT);
    pinMode(33, OUTPUT);
    digitalWrite(32, 0); // Set pin 32 low (turn off backlight)
    digitalWrite(33, 1); // Set pin 33 high (turn on backlight)

    // Initialize serial communication at 115200 baud rate
    Serial.begin(115200);

    // Print a message to indicate the board is initializing
    Serial.println("Initializing board");
```

Set GPIO pins 32 and 33 to output mode, controlling pin 32 to LOW and pin 33 to HIGH. This is used to turn on the screen backlight. Initialize serial communication with a baud rate of 115200, enabling data exchange between the microcontroller and a computer or other devices. Then, send an initialization message via the serial port to indicate that the program has started the initialization process.

Create and initialize a Board object

```
     // Create a Board object and initialize it
    Board *board = new Board();
    board->init();
    board->begin();
```

Board *board = new Board();： This creates a Board object named board to manage hardware resources related to the development board, such as the display, touchscreen, backlight, etc.

board->init(); & board->begin();： These methods initialize and start the hardware and settings within the Board object. init() is typically responsible for configuring the hardware resources, while begin() activates those hardware components.

Initialize the LVGL library

```
        // Initialize the LVGL (LittlevGL) library for display and touch
         lvgl_port_init(board->getLCD(), board->getTouch());
```

lvgl_port_init(board->getLCD(), board->getTouch());： This initializes the LVGL library with the interfaces for the display and touchscreen.

board->getLCD() & board->getTouch() are used to retrieve the objects for the LCD display and the touchscreen respectively. These objects are passed as arguments to initialize the control for both the display and touchscreen.

Turn off the backlight

```
        // Turn off the backlight initially
        board->getBacklight()->setBrightness(0);
```

board->getBacklight()->setBrightness(0); ： This retrieves the backlight control object using board->getBacklight() and then calls setBrightness(0) to turn off the backlight. Setting the brightness to 0 effectively disables the backlight.

Retrieve and configure the display

```
            // Get the active screen (LVGL screen)
           lv_obj_t *scr = lv_scr_act();
           // Set the background color of the screen to black
           lv_obj_set_style_bg_color(scr, lv_color_black(), LV_PART_MAIN);
           lv_obj_set_style_bg_opa(scr, LV_OPA_COVER, LV_PART_MAIN);
```

lv_scr_act()： This function retrieves the currently active screen object in LVGL. It represents the target screen that you want to interact with or modify.

lv_obj_set_style_bg_color(scr, lv_color_black(), LV_PART_MAIN)： This sets the background color of the screen to black. Here, lv_color_black() is a predefined function in LVGL that returns the black color.

lv_obj_set_style_bg_opa(scr, LV_OPA_COVER, LV_PART_MAIN)： This sets the background opacity of the screen to fully opaque (LV_OPA_COVER).

The purpose of these steps is to configure the screen's background color and opacity, so that the screen will initially display with a black background and no transparency.

Lock the LVGL port

```
                  // Lock LVGL port (to prevent concurrent access)
                    lvgl_port_lock(-1);
```

lvgl_port_lock(-1)： This locks the resources of the LVGL library to ensure that during display updates, there won't be any conflicts with other tasks or operations. The purpose of locking the LVGL port is to prevent other tasks from accessing the display or LVGL library simultaneously, avoiding potential data conflicts or inconsistencies.

Create and configure a label

```
  // Create a label on the screen to display the distance
label_1 = lv_label_create(scr);
lv_label_set_text(label_1, "distance = 0.0 cm"); // Default text for the label
lv_obj_set_style_text_color(label_1, lv_color_white(), LV_STATE_DEFAULT); // Set label
text color to white
lv_obj_set_style_text_font(label_1, &lv_font_montserrat_30, 0); // Set font size
lv_obj_align(label_1, LV_ALIGN_CENTER, 0, -20); // Position the label in the center of
the screen
```

label_1 = lv_label_create(scr);： This creates a label object (lv_label_t) on the screen, which will be used to display text.

lv_label_set_text(label_1, "distance = 0.0 cm");： This sets the text of the label to "distance = 0.0 cm", which will be the default text displayed initially.

lv_obj_set_style_text_color(label_1, lv_color_white(), LV_STATE_DEFAULT);： This changes the text color of the label to white.

lv_obj_set_style_text_font(label_1, &lv_font_montserrat_30, 0); ： This sets the font of the label's text to lv_font_montserrat_30, which is a font size of 30.

lv_obj_align(label_1, LV_ALIGN_CENTER, 0, -20);： This aligns the label to the center of the screen and offsets it by -20 pixels in the vertical direction, moving the label slightly upward.

These steps are meant to create and configure a label on the screen, displaying the initial text (e.g., "distance"). The label will be centered, the text will be white, and the font will be set to size 30.

Unlock the LVGL port

```
                  // Unlock LVGL port (allowing access to other tasks)
                       lvgl_port_unlock();
```

lvgl_port_unlock()： This unlocks the LVGL port, allowing other tasks to access the LVGL library.The purpose of this step is to release the resources that were previously locked, enabling other tasks to interact with the LVGL library and perform operations such as graphic updates, screen rendering, or input handling.

Wait for initialization to stabilize, then set the backlight brightness.

```
    // Delay to allow the display to initialize properly
        delay(200);

        // Set the backlight brightness to a higher value (100)
        board->getBacklight()->setBrightness(100);
    }
```

board->getBacklight()->setBrightness(100);： This sets the backlight brightness to 100 (with a maximum value of 255).In this case, the backlight brightness is increased to ensure the screen displays clear and bright images, making it more visible under various lighting conditions.

loop function

```
    // void loop()
    {
        // Measure the distance using the ultrasonic sensor (in cm)
        float distance = distanceSensor.measureDistanceCm();

        // Lock LVGL port to avoid concurrent access
        lvgl_port_lock(-1);
```

distanceSensor.measureDistanceCm()： This method uses the ultrasonic sensor to measure the distance in centimeters. The value returned by this method is stored in the distance variable.

lvgl_port_lock(-1)： This locks the LVGL (LittlevGL) library port to prevent concurrent access issues during execution. Since the display content is frequently updated in the loop() function, locking the port ensures that no other tasks interfere with the display operations, avoiding conflicts while rendering new content on the screen.

Distance measurement and display

```
    // Check if the distance exceeds the threshold (35 cm)
        if(distance > 35)
        {
            // If the distance exceeds the limit, display a warning message
            lv_label_set_text(label_1, "the distance exceeds the limit");
        }
        else
        {
              // Format the measured distance into a string and update the
label text
            snprintf(buffer, sizeof(buffer), "distance = %.1f cm", distance);
            lv_label_set_text(label_1, buffer);
        }
```

if (distance > 35)： Checks if the distance exceeds the threshold (35 cm). If it does, a warning message is displayed.

lv_label_set_text(label_1, "the distance exceeds the limit");： If the distance is greater than 35 cm, it updates the label text on the screen to display a warning message: "the distance exceeds the limit."

else： If the distance is less than or equal to 35 cm, the following actions are performed:

snprintf(buffer, sizeof(buffer), "distance = %.1f cm", distance);： Formats the measured distance into a string and stores it in the buffer. %.1f ensures the output is a floating-point number with one decimal place (e.g., "distance = 12.3 cm").

lv_label_set_text(label_1, buffer);： Displays the formatted distance string on the label label_1.

Print the distance value to the serial port.

```
    // Unlock LVGL port after updating the display
    lvgl_port_unlock();

    // Print the measured distance to the serial monitor
    Serial.println(distance);

    // Add a delay of 500ms before the next loop iteration
    delay(500);
}
```

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson12---Temp&Humi

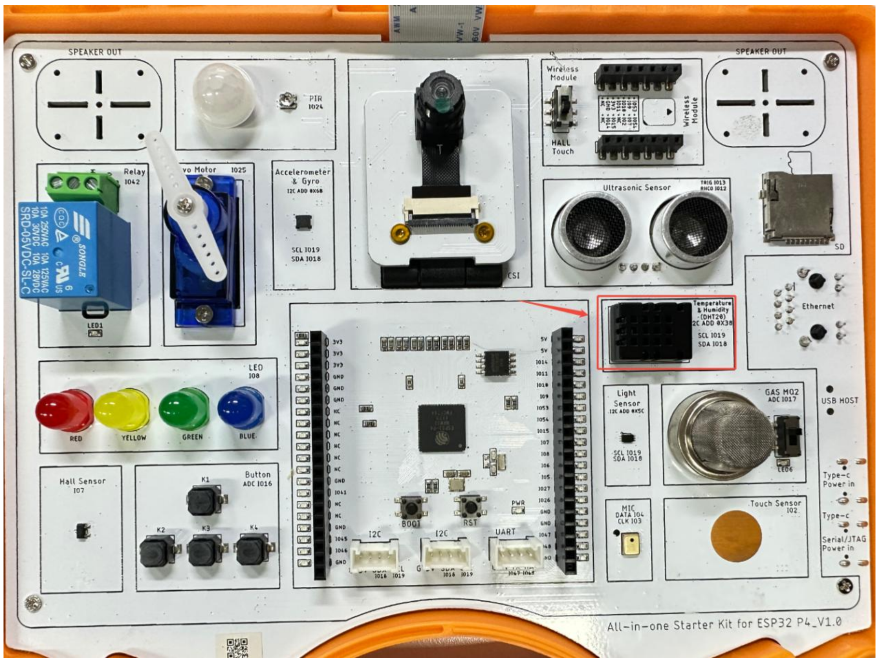
Introduction

In this lesson, we will learn how to use the temperature and humidity sensor on the All-in-One Starter Kit for ESP32-P4 development board. We will use the DHT20 library and its built-in methods to display the temperature and humidity values on the screen.

Learning Goals

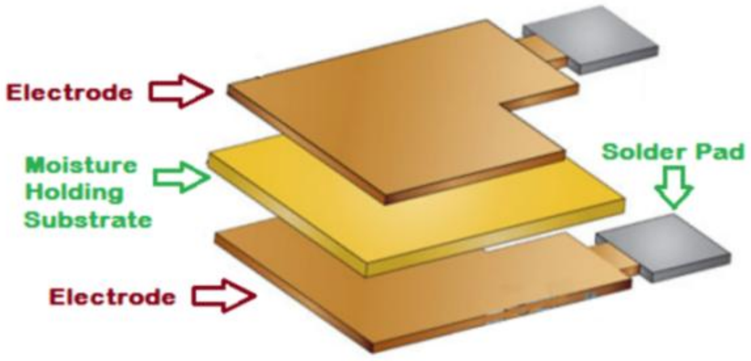
1.Understand the Working Principle of Temperature and Humidity Sensors
2.Implement a Case to Retrieve Data from a Temperature and Humidity Sensor and Display it on the Screen

Hardware Used in This Lesson:

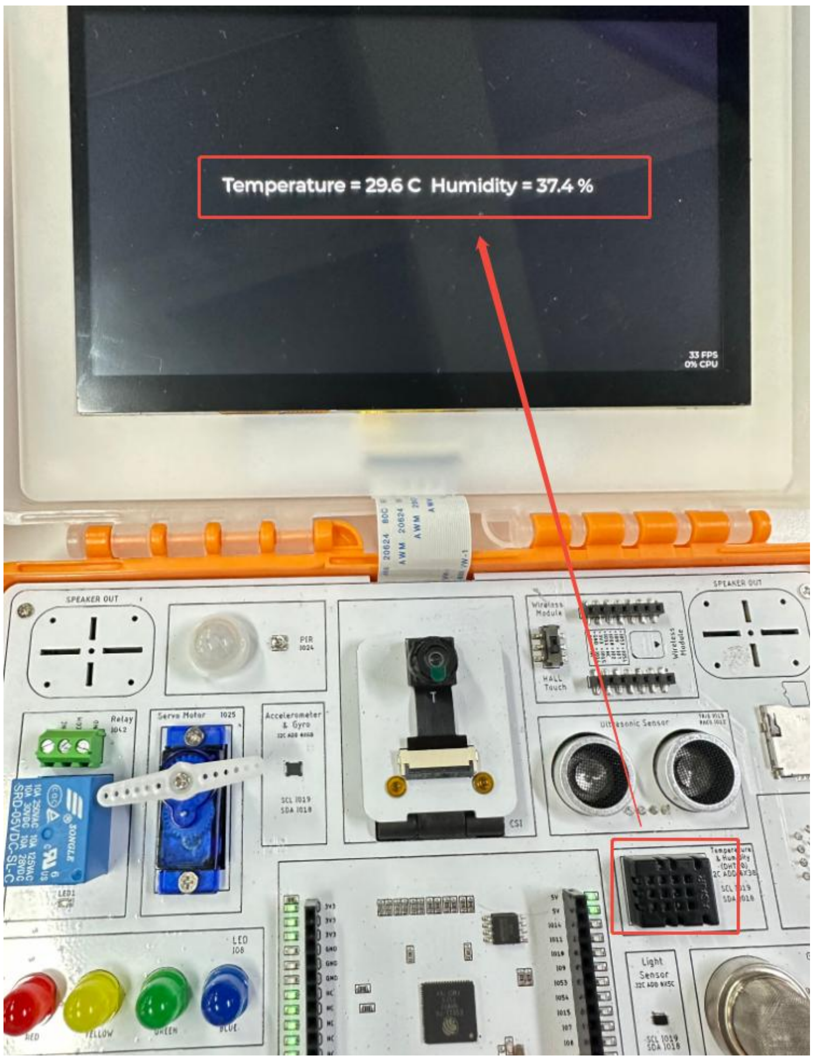


The working principle of Temperature and Humidity Sensor Module

The diagram below shows the structure of the humidity sensor detection unit: its core consists of upper and lower electrodes, with a hygroscopic dielectric substrate in between. The pads are used to achieve electrical connections between the electrodes and the external circuit. During the humidity detection process, water vapor in the environment is either adsorbed or desorbed by the dielectric substrate. This process alters the dielectric constant of the substrate, which in turn causes changes in the capacitance of the capacitor formed by the electrodes and the substrate, thus converting the humidity signal. At the same time, the integrated temperature-sensitive element of the sensor collects the ambient temperature. It not only outputs temperature signals independently but also compensates for the humidity measurement deviation through a temperature compensation mechanism. Finally, the signal conditioning circuit outputs accurate temperature and humidity electrical signals.



Running Effect Diagram:


Temperature = 29.6 C Humidity = 37.4 %

After running the program, you can view the real-time temperature and humidity values on the All-in-One Starter Kit for ESP32-P4 display.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson12_Temp_Humi_sensor

Key Explanation:

Include Header Files / Variable Definitions

```
#include "Arduino.h"
#include <esp_display_panel.hpp>
#include <lvgl.h>
#include <lvgl_v8_port.h>
#include "bsp_i2c.h"
#include "bsp_dht20.h"
```

#include "Arduino.h"： Includes the core Arduino library, which provides essential functions and definitions for the Arduino platform.

#include <esp_display_panel.hpp>： Includes the ESP32 display control library to manage the display panel.

#include <lvgl.h>： Includes LVGL a library for creating and managing graphical user interfaces.

#include <lvgl_v8_port.h>： Includes the porting code for LVGL, enabling it to work in the ESP32 environment。

#include "bsp_i2c.h"： Includes the I2C bus initialization and control library, allowing communication with the DHT20 sensor.

#include "bsp_dht20.h"： Includes the DHT20 sensor library for controlling and reading data from the DHT20 sensor.

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

Using the **esp_panel::drivers** and **esp_panel::board** namespaces makes it easier to access the classes and functions related to ESP32 panel control in the future.

Declare task handles and label objects

```
// Declare task handle for DHT20 reading
TaskHandle_t read_dht20;
// Declare a label for displaying temperature and humidity
lv_obj_t *label_1 = NULL;
```

TaskHandle_t read_dht20;： Declares a task handle for managing the DHT20 data reading task.

lv_obj_t *label_1 = NULL;： Declares an LVGL label object (of type lv_obj_t) to display temperature and humidity information, initially set to NULL.

The function to update the temperature and humidity display

```
// Function to update the DHT20 values on the display
void update_dht20_value(float temperature, float humidity)
{
    if (label_1)   // If label is created
    {
        char buffer[60];
          snprintf(buffer, sizeof(buffer), "Temperature = %.1f C   Humidity = %.1f %%", 
temperature, humidity); // Format the data into a string
        lv_label_set_text(label_1, buffer);   // Set the new text for the label
    }
}
```

update_dht20_value：Defines a function to update the displayed temperature and humidity data.

if (label_1)： Checks whether the label has been created already

snprintf： Formats the temperature and humidity data into a string that includes the relevant information

lv_label_set_text(label_1, buffer)： Sets the formatted string as the text to be displayed on the label.

The task of reading data from the DHT20 sensor

```
// Task to read the DHT20 sensor data
void dht20_read_task(void *param)
{
    static dht20_data_t measurements;   // Store the measurement data
    while (1)
    {
        if (dht20_is_calibrated() == ESP_OK)   // Check if the DHT20 sensor is calibrated
        {
            Serial.println("is calibrated....");
        }
        else
        {
            Serial.println("is NOT calibrated....");
                 if (dht20_begin() != ESP_OK)    // Reinitialize the DHT20 sensor if not 
calibrated
            {
                Serial.println("dht20 init again false");
                     vTaskDelay(100 / portTICK_PERIOD_MS);    // Wait for a bit before 
retrying
                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(label_1, "dht20 read data error");   // Display an error
message if reading fails
                lvgl_port_unlock();
            }
            Serial.println("dht20 read data error");
        }
        else   // If reading is successful
        {
            if (lvgl_port_lock(0))
            {
                                                  update_dht20_value(measurements.temperature,
measurements.humidity);   // Update the displayed values on the screen
                lvgl_port_unlock();
            }
            Serial.printf("Temperature:\t%.1fC\n", measurements.temperature);   // Print
temperature to Serial Monitor
               Serial.printf("Humidity:     \t%.1f%%", measurements.humidity);   // Print
humidity to Serial Monitor
        }

        vTaskDelay(1000 / portTICK_PERIOD_MS);   // Wait for 1 second before reading
again
    }
}
```

dht20_read_task: A task function responsible for continuously reading data from the DHT20 sensor and updating the display.
static dht20_data_t measurements;: Declares a static variable to store the temperature and humidity data.
while (1): An infinite loop that continuously reads sensor data.
if (dht20_is_calibrated() == ESP_OK): Checks if the DHT20 sensor has been calibrated.
dht20_begin(): Initializes the DHT20 sensor.
dht20_read_data(&measurements): Reads temperature and humidity data from the DHT20 sensor and stores it in the measurements variable.
lvgl_port_lock(0) & lvgl_port_unlock(): Locks and unlocks the display update process to prevent concurrency issues with the LVGL library.
vTaskDelay(1000 / portTICK_PERIOD_MS): Delays the task for 1 second to control the frequency of data reading.

Setup function

```
void setup()
```

```
{
    // Set pins for control (e.g., for backlight)
    pinMode(32, OUTPUT);
    pinMode(33, OUTPUT);
    digitalWrite(32, 0);   // Turn off pin 32
    digitalWrite(33, 1);   // Turn on pin 33

    i2c_init();   // Initialize I2C for communication
    dht20_begin();   // Initialize the DHT20 sensor
    Serial.begin(115200);   // Start Serial communication

    Serial.println("Initializing board");

    // Initialize the display board
    Board *board = new Board();
    board->init();
    board->begin();

    // Initialize the LVGL port with the display
    lvgl_port_init(board->getLCD(), nullptr);
    board->getBacklight()->setBrightness(0);   // Turn off the backlight initially
    lv_obj_t *scr = lv_scr_act();   // Get the current screen

    // Set screen background color to black
    lv_obj_set_style_bg_color(scr, lv_color_black(), LV_PART_MAIN);
    lv_obj_set_style_bg_opa(scr, LV_OPA_COVER, LV_PART_MAIN);

    lvgl_port_lock(-1);   // Lock LVGL port before creating the label
    label_1 = lv_label_create(scr);   // Create a label for displaying data
    lv_label_set_text(label_1, "Temperature = 0.0 C   Humidity = 0.0 %");   // Set initial
text
     lv_obj_set_style_text_color(label_1, lv_color_white(), LV_STATE_DEFAULT);   // Set
text color to white
    lv_obj_set_style_text_font(label_1, &lv_font_montserrat_30, 0);   // Set font for text
    lv_obj_align(label_1, LV_ALIGN_CENTER, 0, -20);   // Align label to center of screen
    lvgl_port_unlock();   // Unlock LVGL port after creating label

    // Create a task to handle DHT20 readings in the background
    xTaskCreate(dht20_read_task, "read_dht20", 4096, NULL, configMAX_PRIORITIES -
5, &read_dht20);

    delay(200);   // Delay to ensure initialization is complete
    board->getBacklight()->setBrightness(100);   // Set backlight brightness to 100
}
```

setup(): Initialization code that runs once to configure hardware and software settings.
Configures and initializes I/O pins, the I2C interface, the DHT20 sensor, and serial communication.
Initializes the display and the LVGL library, setting the display's background color and text properties.
Creates and starts the dht20_read_task, which runs in the background to read data from the DHT20 sensor.

loop function

```
void loop()
{
    // The loop function is left empty since we handle everything in the DHT20 task
#if 0
    lvgl_port_lock(-1);

    lvgl_port_unlock();

    delay(500);
#endif
}
```

loop(): This is the main loop function in the Arduino program. Since all tasks are handled within the DHT20 reading task, this function is empty.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson13---Light Sensor

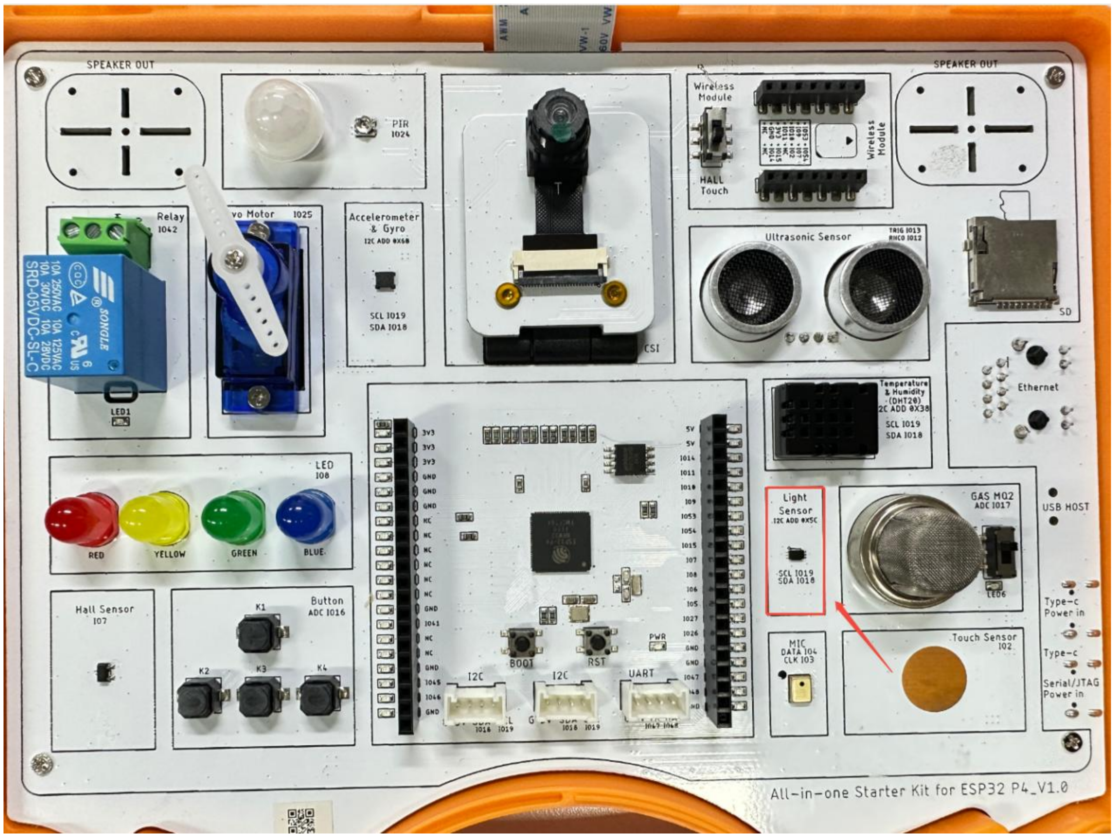
Introduction

In this lesson, we will learn how to use the "Light Sensor" on the "All-in-One Starter Kit for ESP32-P4" development board. We will retrieve data from the light sensor and display it on the board's screen.

Learning Goals

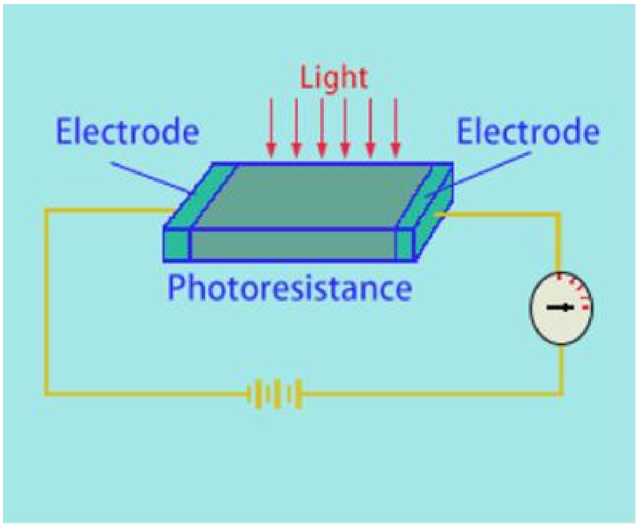
1.Understand how a light-dependent resistor (LDR) sensor works
2.Read the values from the Light sensor and display them on the screen

Hardware Used in This Lesson:



The working principle of Light Sensor

As shown in the diagram, the core of the light sensor is the light-dependent resistor (LDR) in the center. When light hits the surface of the LDR, its resistance changes based on the light intensity — the stronger the light, the lower the resistance. In the circuit formed by the power source, the ammeter, and the LDR, the current will vary according to the change in resistance (lower resistance results in higher current, and higher resistance results in lower current). This way, the light signal is converted into an electrical signal within the circuit, and the ammeter's reading reflects the changes in light intensity. This is the fundamental working principle of the light sensor, where it "controls electricity with light."



Running Effect Diagram:

Once the system is running, the values read by the light sensor will be displayed on the screen of the "All-in-One Starter Kit for ESP32-P4."

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson13_Light_Sensor

Key Explanation:

Include Header Files

```
#include "Arduino.h"
#include <esp_display_panel.hpp>
#include <lvgl.h>
#include <lvgl_v8_port.h>
#include "bsp_i2c.h"
#include "bsp_bh1750.h"
```

#include "Arduino.h"： Includes the core Arduino library, which provides essential functions and definitions for the Arduino platform.

#include <esp_display_panel.hpp>： Includes the ESP32 display control library to manage the display panel.

#include <lvgl.h>： Includes LVGL a library for creating and managing graphical user interfaces.

#include <lvgl_v8_port.h>： Includes the porting code for LVGL, enabling it to work in the ESP32 environment。

#include "bsp_i2c.h"： Includes the I2C bus initialization and control library, allowing communication with the DHT20 sensor.

#bsp_bh1750.h"： Operation to import the BH1750 light sensor library.

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

Using the **esp_panel::drivers** and **esp_panel::board** namespaces makes it easier to access the classes and functions related to ESP32 panel control in the future.

Global variables

```
// Task handle for reading BH1750 sensor data
TaskHandle_t read_bh1750;

// Label to display the light level on the screen
lv_obj_t *label_1 = NULL;
```

TaskHandle_t read_bh1750;： This stores the FreeRTOS task handle, which is used for the task that reads the BH1750 sensor.

lv_obj_t *label_1 = NULL;： This is a pointer to a LVGL label object, which is used to display the lux value.

Function to display the lux value.

```
// Function to update the label with the current lux value from the BH1750 sensor
void update_bh1750_value(float lux)
{
    if (label_1)
    {
        char buffer[60];
        snprintf(buffer, sizeof(buffer), "lux = %.1f", lux); // Format the data into a string
        lv_label_set_text(label_1, buffer);                   // Set the formatted string as
label text
    }
}
```

Format the **float lux** value into a string like "lux = 123.4".

Use **lv_label_set_text()** to update the LVGL label with the new value.

Check if **label_1** is valid before accessing it to avoid null pointer issues.

The task of reading the BH1750 sensor

```
// Task that continuously reads data from the BH1750 sensor and updates the display
void bh1750_read_task(void *param)
{
    float lux = 0;
    while (1)
    {
          lux = bh1750_read_data(); // Read the illuminance data from the BH1750
sensor
        if (lux == (-1))
        {
            if (lvgl_port_lock(0))
            {
                    lv_label_set_text(label_1, "bh1750 read data error"); // Display error
message if reading fails
                lvgl_port_unlock();
            }
        }
        else
        {
            // If data is valid, update the label with the new lux value
            if (lvgl_port_lock(0))
            {
                update_bh1750_value(lux); // Update the BH1750 data on the screen
                lvgl_port_unlock();
            }
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay for 1 second before reading
again
    }
}
```

bh1750_read_data()：Reads the light intensity from the BH1750 sensor. A return value of -1 indicates a failure to read the data

lvgl_port_lock() & lvgl_port_unlock()：These are thread locks for LVGL to prevent concurrent access from multiple tasks, which could cause conflicts or crashes in the graphical library

vTaskDelay()：A FreeRTOS delay function. In this case, it's used to read the BH1750 sensor once every second.

Setup function

```
void setup()
{
    pinMode(32, OUTPUT);
    pinMode(33, OUTPUT);
    digitalWrite(32, 0);
    digitalWrite(33, 1);
```

Set GPIO32 and GPIO33 to output mode, with GPIO32 set to low (0V) and GPIO33 set to high (3.3V). This configuration is used to control the backlight of the screen, turning it on

Initialize I2C and establish communication with the BH1750 sensor, with serial output for debugging purposes.

```
    i2c_init();              // Initialize I2C communication
    bh1750_begin();          // Initialize the BH1750 sensor
    Serial.begin(115200); // Initialize serial communication for debugging
    Serial.println("Initializing board");
```

Create a Board object and initialize the display and its backlight.

```
    Board *board = new Board();
    board->init();
    board->begin();
```

Initialize the LVGL library, bind it to the display, and set the backlight brightness to 0 (off).

```
    // Initialize LVGL graphics library with the board's display
    lvgl_port_init(board->getLCD(), nullptr);
    board->getBacklight()->setBrightness(0); // Set initial brightness to 0
```

Get the current screen object scr, set its background color to black, and make it fully cover the display.

```
    lv_obj_t *scr = lv_scr_act();
    lv_obj_set_style_bg_color(scr, lv_color_black(), LV_PART_MAIN);
    lv_obj_set_style_bg_opa(scr, LV_OPA_COVER, LV_PART_MAIN);
```

Create a label to display the lux value, and set its text color, font size, and position.

```
	lvgl_port_lock(-1);
	label_1 = lv_label_create(scr);
	lv_label_set_text(label_1, "lux = 0.0");
	lv_obj_set_style_text_color(label_1, lv_color_white(), LV_STATE_DEFAULT);
	lv_obj_set_style_text_font(label_1, &lv_font_montserrat_30, 0);
	lv_obj_align(label_1, LV_ALIGN_CENTER, 0, -20);
	lvgl_port_unlock();
```

Set the text color to white, font size to 30, and position the label slightly above the center of the screen. Use **lvgl_port_lock()** to ensure thread-safe LVGL operations.

Create a FreeRTOS task to read the BH1750 sensor once every second

```
	xTaskCreate(bh1750_read_task, "read_bh1750", 4096, NULL,
configMAX_PRIORITIES - 5, &read_bh1750);
	delay(200);
```

Set the task stack size to 4096 bytes, with a priority slightly below the highest level.

Set the backlight to its maximum brightness of 100

```
	board->getBacklight()->setBrightness(100);
```

loop function

```
void loop()
{
	lvgl_port_lock(-1);
	lvgl_port_unlock();
	delay(500);
}
```

The loop() function doesn't perform any operations here; all functionality is handled within FreeRTOS tasks.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson14---Six-axis Module

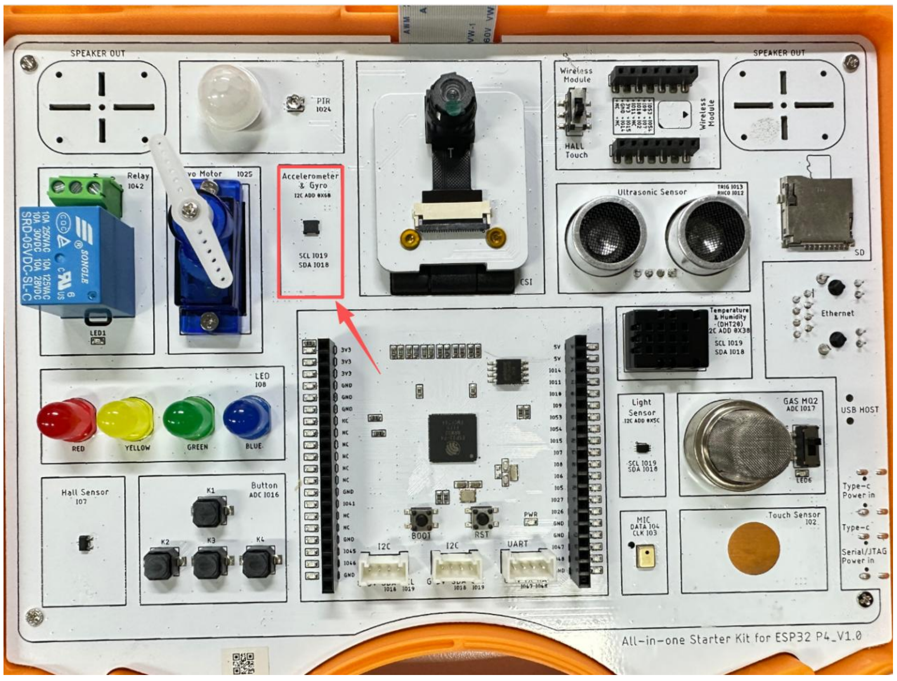
Introduction

In this lesson, we will learn how to use the six-axis accelerometer sensor on the "All-in-One Starter Kit for ESP32-P4" development board. During this lesson, we will utilize the external "**bsp_lsm6ds3tr**" library to read data from both the accelerometer and gyroscope, and display the real-time data on the development board's screen.

Learning Goals

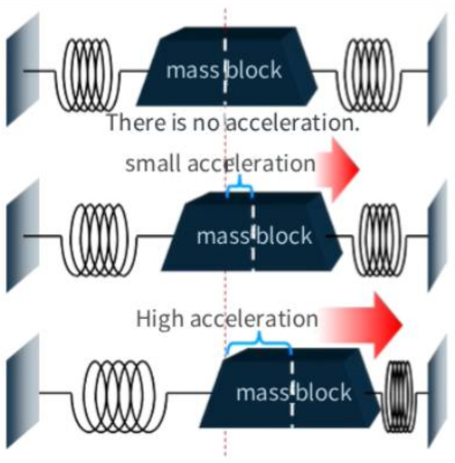
1.Understand the working principle of the six-axis sensor
2.Acquire raw data for acceleration and angular velocity, and display it on the screen

Hardware Used in This Lesson:

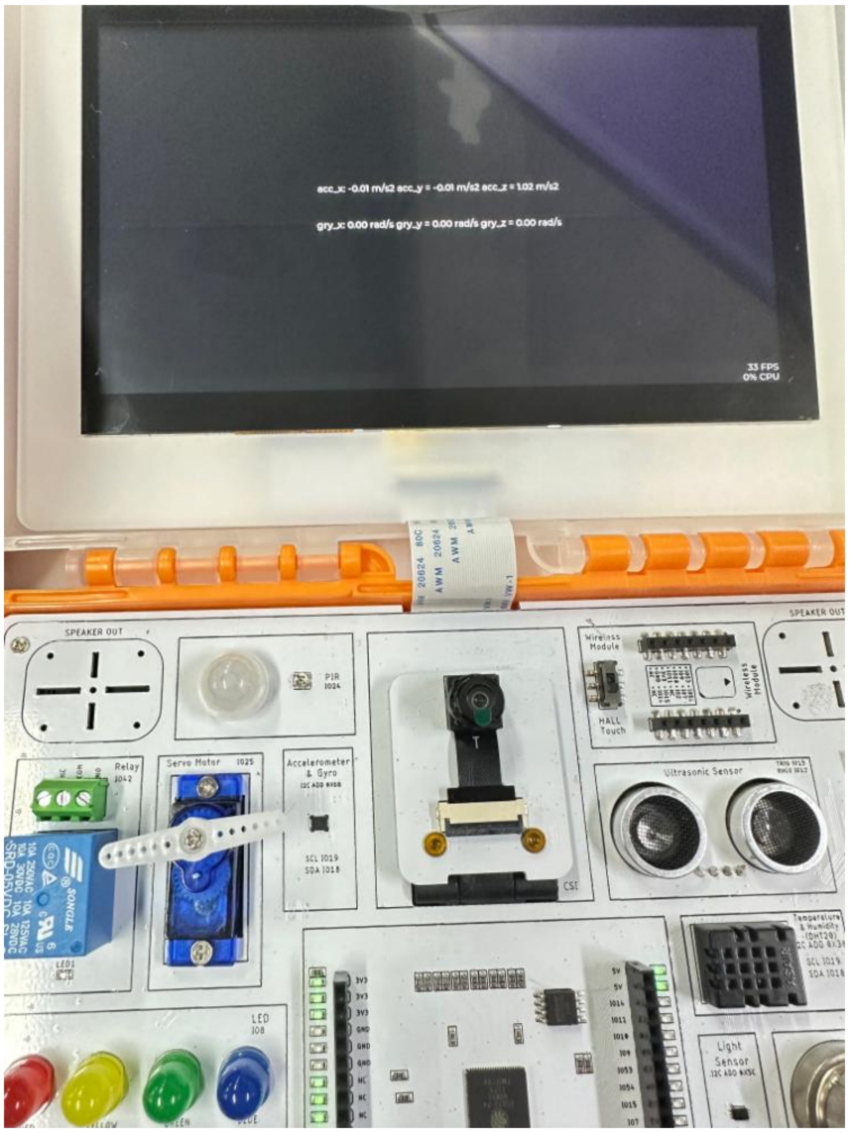


The working principle of Six-axis Module

The diagram below shows the principle of the "single-axis accelerometer" in a six-axis sensor: It relies on an internal mass block and a spring. When the sensor accelerates, the inertia of the mass block causes a deformation in the spring. The amount of deformation corresponds to the magnitude of acceleration (when there is no acceleration, the deformation is consistent; for small accelerations, the deformation is slight; for larger accelerations, the deformation is greater). A six-axis sensor includes three such acceleration axes (one for each X/Y/Z direction, with the same principle as the one shown in the diagram, but with different directions) and three gyroscope axes for detecting rotational angular velocity. The multi-axis structure allows the sensor to detect three-dimensional acceleration and angular velocity.



Running Effect Diagram:



```
acc_x: -0.01 m/s2 acc_y = -0.01 m/s2 acc_z = 1.02 m/s2

gry_x: 0.00 rad/s gry_y = 0.00 rad/s gry_z = 0.00 rad/s
```

After running the program, the data collected by the six-axis sensor will be displayed at the center of the screen.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson14_Six_axis_accelerometer_sensor

Key Explanation:

Include Header Files

```
#include "Arduino.h"
#include <esp_display_panel.hpp>
#include <lvgl.h>
#include <lvgl_v8_port.h>
#include "bsp_i2c.h"
#include "bsp_lsm6ds3tr.h"
```

#include "Arduino.h"： Includes the core Arduino library, which provides essential functions and definitions for the Arduino platform.
#include <esp_display_panel.hpp>： Includes the ESP32 display control library to manage the display panel.
#include <lvgl.h>： Includes LVGL a library for creating and managing graphical user interfaces.
#include <lvgl_v8_port.h>： Includes the porting code for LVGL, enabling it to work in the ESP32 environment.。
#include "bsp_i2c.h"： Includes the I2C bus initialization and control library, allowing communication with the DHT20 sensor.
#bsp_lsm6ds3tr.h"： Operation to import the LSM6DS3 sensor library.

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

Using the **esp_panel::drivers** and **esp_panel::board** namespaces makes it easier to access the classes and functions related to ESP32 panel control in the future.

Global variables

```
TaskHandle_t read_lsm6ds3;   // Task handle for the LSM6DS3 read task
static lv_obj_t *lsm6ds3_acc_data = NULL; // LVGL label object for accelerometer data
static lv_obj_t *lsm6ds3_gry_data = NULL; // LVGL label object for gyroscope data
```

read_lsm6ds3： FreeRTOS task handle used to control the LSM6DS3 sensor reading task.
lsm6ds3_acc_data & lsm6ds3_gry_data： Label objects representing the accelerometer and gyroscope data, created using the LVGL library and displayed on the screen.

Function to update the displayed data

```
// Function to update the LVGL labels with LSM6DS3 sensor data
void update_lsm6ds3_value(lsm6ds3tr data) {
  if ((lsm6ds3_acc_data) && (lsm6ds3_gry_data)) {
    char buffer_acc[80];
    char buffer_gry[80];

    // Format accelerometer data into a string
     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 gyroscope 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);

    // Update the LVGL labels with new sensor data
    lv_label_set_text(lsm6ds3_acc_data, buffer_acc);
    lv_label_set_text(lsm6ds3_gry_data, buffer_gry);
  }
}
```

The purpose of this function is to receive a data structure from the LSM6DS3 sensor (containing both accelerometer and gyroscope data), then format the data into a string and update the corresponding labels. The **snprintf** function is used to format the data so it can be displayed on the screen.

LSM6DS3 reading task

```
// FreeRTOS task to continuously read LSM6DS3 sensor data
void lsm6ds3_read_task(void *param) {
  while (1) {
    // Scan the LSM6DS3 sensor to determine data availability
    if (lsm6ds3_scan() != ESP_OK) {
      if (lvgl_port_lock(0)) {
        // Display read error messages on the LVGL labels
        lv_label_set_text(lsm6ds3_acc_data, "lsm6ds3 read acc data error");
        lv_label_set_text(lsm6ds3_gry_data, "lsm6ds3 read gry data error");
        lvgl_port_unlock();
      }
      //MAIN_ERROR("lsm6ds3 scan false"); // Optional debug log
    } else {
      if (lvgl_port_lock(0)) {
        // Update the labels with the latest sensor values
        update_lsm6ds3_value(my_lsm6ds3);
        lvgl_port_unlock();
      }
    }
```

This is a FreeRTOS task that continuously reads data from the LSM6DS3 sensor. If the read fails, an error message is displayed on the screen. If data is successfully read, the **update_lsm6ds3_value** function is called to update the display.
vTaskDelay is used to pause the task for 1 second (1000ms), preventing the sensor from being read too frequently.

Setup function

```
void setup()
{
        pinMode(32, OUTPUT);
        pinMode(33, OUTPUT);
        digitalWrite(32, 0);
        digitalWrite(33, 1);
```

Set GPIO32 and GPIO33 to output mode, with GPIO32 set to low (0V) and GPIO33 set to high .
This configuration is used to control the backlight of the screen, turning it on

Initialize I2C and establish communication with the LSM6DS3 sensor, with serial output for debugging purposes.

```
  // Initialize I2C and LSM6DS3 sensor
  i2c_init();
  lsm6ds3_begin();

  Serial.begin(115200);
  Serial.println("Initializing board");
```

Create a Board object and initialize the display and its backlight.

```
        Board *board = new Board();
        board->init();
        board->begin();
```

Initialize the LVGL library, bind it to the display, and set the backlight brightness to 0 (off).

```
      // Initialize LVGL graphics library with the board's display
      lvgl_port_init(board->getLCD(), nullptr);
      board->getBacklight()->setBrightness(0); // Set initial brightness to 0
```

Get the current screen object scr, set its background color to black, and make it fully cover the display.

```
        lv_obj_t *scr = lv_scr_act();
        lv_obj_set_style_bg_color(scr, lv_color_black(), LV_PART_MAIN);
        lv_obj_set_style_bg_opa(scr, LV_OPA_COVER, LV_PART_MAIN);
```

Create a label to display the lux value, and set its text color, font size, and position.

```
   lvgl_port_lock(-1); // Lock LVGL for object creation
  // Create and configure accelerometer label
  lsm6ds3_acc_data = lv_label_create(scr);
  lv_obj_set_style_text_color(lsm6ds3_acc_data, lv_color_white(), LV_STATE_DEFAULT);
  lv_obj_align(lsm6ds3_acc_data, LV_ALIGN_CENTER, 0, -30);
  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");
  // Create and configure gyroscope label
  lsm6ds3_gry_data = lv_label_create(scr);
  lv_obj_set_style_text_color(lsm6ds3_gry_data, lv_color_white(), LV_STATE_DEFAULT);
  lv_obj_align(lsm6ds3_gry_data, LV_ALIGN_CENTER, 0, 30);
  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");
  lvgl_port_unlock(); // Unlock LVGL
```

Set the text color to white, font size to 30, and position the label slightly above the center of the screen. Use **lvgl_port_lock()** to ensure thread-safe LVGL operations.

Create a FreeRTOS task to read the LSM6DS3 sensor once every second

```
     // Create the FreeRTOS task to read LSM6DS3 sensor values
       xTaskCreate(lsm6ds3_read_task, "read_lsm6ds3", 4096, NULL,
configMAX_PRIORITIES - 5, &read_lsm6ds3);
     delay(200);
```

The task priority is set lower than other tasks (configured using **configMAX_PRIORITIES - 5**)

Set the backlight to its maximum brightness of 100

```
     board->getBacklight()->setBrightness(100);
```

loop function

```
void loop()
{
     delay(500);
}
```

The loop() function doesn't perform any operations here; all functionality is handled within FreeRTOS tasks.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson15---Gas Module

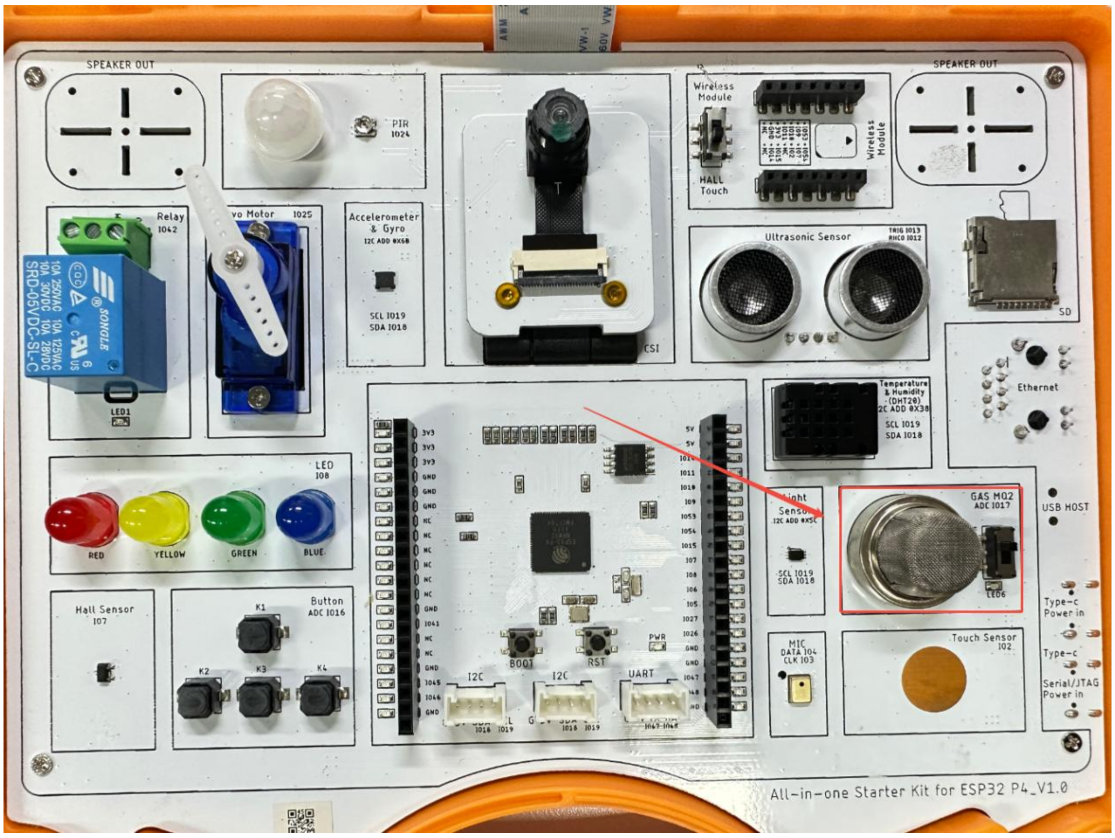
Introduction

In this lesson, we will use the MQ2 gas sensor and an LED light to create a simple "smoke detector." You will integrate the skills you've learned so far: reading the analog values from the sensor, determining whether there are hazardous gases based on a threshold, and controlling the LED to light up. Additionally, we will monitor real-time data changes through the serial port to help you visually understand how the system works.

Learning Goals

1.Understand the working principle of gas sensors
2.Complete a hazardous gas alarm project

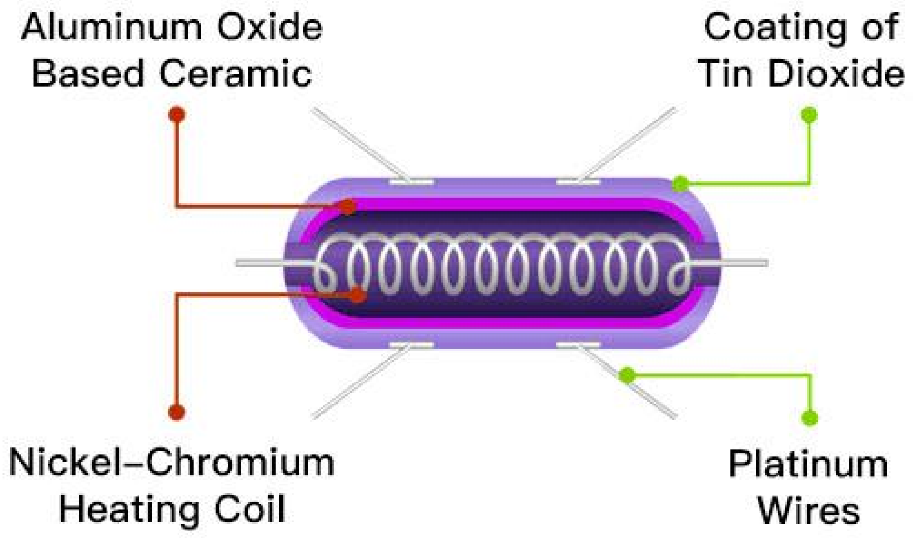
Hardware Used in This Lesson:



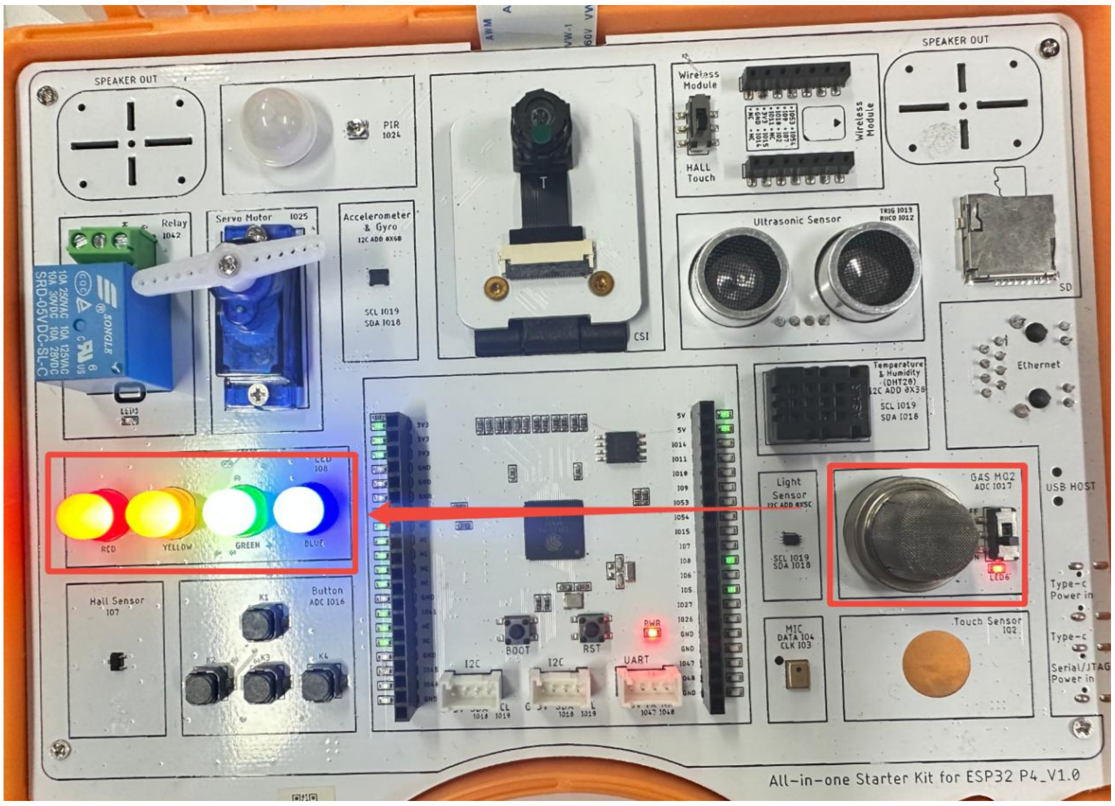
Note: When using the MQ2 gas sensor, remember to toggle the switch to the "on" position.

The working principle of Gas Module

The MQ2 smoke sensor is a device that detects smoke based on the semiconductor gas-sensing effect. Its working process can be described as follows: The core of the sensor consists of an "alumina-based ceramic carrier + tin dioxide coating". The carrier is embedded with a nickel-chromium heating coil inside, and the signal is led out through platinum wire electrodes on the outside. During operation, the nickel-chromium coil is energized to heat up, maintaining the tin dioxide coating at a stable temperature of 200 to 400° C - at this temperature, the surface of the tin dioxide will adsorb oxygen molecules from the air, forming an "oxygen negative ion adsorption layer", keeping the resistivity of the tin dioxide at a relatively high baseline level. When smoke (such as flammable gases, volatile particulate matter) comes into contact with the tin dioxide coating, the reducing molecules in the smoke will react chemically with the surface oxygen negative ions, consuming the adsorbed oxygen ions. This process will change the carrier concentration of the tin dioxide, enhancing its semiconductor conductivity and significantly reducing its resistivity. The platinum wire electrodes will promptly capture this resistance change and convert it into a signal that can be recognized by the external circuit (such as a decrease in resistance value or fluctuation in voltage signal), ultimately corresponding to the level of smoke concentration through the amplitude of the signal change, completing the smoke detection.



Running Effect Diagram:



After running the program, when the gas value detected by the MQ2 sensor exceeds 5000, the LED light will turn on. Otherwise, it will remain off.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson15_gas_module

Key Explanation:

Include Header Files

```
#include <Arduino.h>
#include <Adafruit_NeoPixel.h>
#include <math.h>
```

Arduino.h: Includes the basic library for Arduino programming, used to access fundamental hardware functions (such as pin reading, delays, serial communication, etc.).

Adafruit_NeoPixel.h: A library used to control NeoPixel LED strips.

math.h: Provides mathematical functions, used for calculating gas concentration.

Define Variables

```
#define LED_NUM         2
#define LED_PIN        8
#define GAS_ANALOG_PIN       17
#define VOLTAGE_SAMPLE_COUNT    2
#define CALIBRATION_SAMPLE_COUNT 10
#define VCC                    5.0f
#define LOAD_RESISTOR       4.7f
#define COEFF_A                11.5428f
#define COEFF_B                0.6549f
#define COEFF_C                100.0f
#define ALARM_THRESHOLD_PPM    5000
```

LED_NUM：The number of NeoPixel LEDs, set to 2 LEDs here.
LED_PIN：The GPIO pin connected to the LED, set to pin 8.
GAS_ANALOG_PIN：The analog input pin connected to the gas sensor, set to pin 17.
VOLTAGE_SAMPLE_COUNT：The number of samples taken to calculate the voltage, set to 2 samples.
CALIBRATION_SAMPLE_COUNT：The number of samples used to calibrate R0 (the reference resistance), set to 10 samples.
VCC：The supply voltage for the sensor (typically 5V).
LOAD_RESISTOR：The value of the load resistor (4.7kΩ), which must match the hardware.
COEFF_A、**COEFF_B**、**COEFF_C**：These coefficients are used in the formula to calculate gas concentration (PPM).
ALARM_THRESHOLD_PPM：The alarm threshold, where the LED will light up if the gas concentration (PPM) exceeds this value.

Define a structure gas_data

```
// Global structure storing sensor parameters and latest data
struct {
  float gas_voltage;    // Latest measured sensor voltage (V)
  float RL;              // Load resistor value (kΩ)
  float R0;              // Baseline resistance in clean air (kΩ)
} gas_data;
```

gas_voltage：The current voltage value of the gas sensor.
RL：The value of the load resistor.
R0：The reference resistance value (the value in clean air, used for gas concentration calculation).

NeoPixel LED Initialize

```
// NeoPixel LED strip object
Adafruit_NeoPixel strip(LED_NUM, LED_PIN, NEO_GRB + NEO_KHZ800);
```

Create an **Adafruit_NeoPixel** object, which represents an LED strip connected to a specified pin. This object allows you to control the color and state of the LEDs.

Initialize the gas sensor

```
void gas_init() {
   analogSetPinAttenuation(GAS_ANALOG_PIN, ADC_11db);
   gas_data.RL = LOAD_RESISTOR;
   gas_data.R0 = 0.0;
   gas_data.gas_voltage = 0.0;
}
```

analogSetPinAttenuation：Sets the attenuation (reduces signal strength) for the analog input pin, allowing it to read high voltage signals.
Initialize the parameters in the **gas_data** structure, including the load resistor RL, the gas sensor voltage, and the reference resistance R0.

Read the gas sensor voltage

```
int get_gas_voltage() {
   static int vol_sum = 0;
   static uint8_t vol_reading_cnt = 0;

   int raw_mv = analogReadMilliVolts(GAS_ANALOG_PIN);
   vol_sum += raw_mv;
   vol_reading_cnt++;

   if (vol_reading_cnt == VOLTAGE_SAMPLE_COUNT) {
      float avg_mv = (float)vol_sum / vol_reading_cnt;
      gas_data.gas_voltage = avg_mv / 1000.0f;
      vol_sum = 0;
      vol_reading_cnt = 0;
      return 1;
   }
   return 0;
```

This function reads the analog value from the gas sensor, calculates the average voltage value after a certain number of samples, and stores it in **gas_data.gas_voltage.**

Calibrate R0 (the reference resistance)

```
float get_r0_calibration() {
   static uint8_t vol_reading_cnt = 0;
   static float vol_sum = 0;

   while (1) {
      if (get_gas_voltage()) {
         vol_sum += gas_data.gas_voltage;
         vol_reading_cnt++;
      }

      if (vol_reading_cnt >= CALIBRATION_SAMPLE_COUNT) {
         float Vavg = vol_sum / vol_reading_cnt;
```

```
            gas_data.R0 = ((VCC - Vavg) * gas_data.RL) / Vavg;
            vol_reading_cnt = 0;
            vol_sum = 0;
            break;
        }

        delay(100);
    }

    return gas_data.R0;
}
```

The **get_r0_calibration** function calculates the average voltage value through multiple samples and then computes the reference resistance **R0**. This value is measured in clean air and serves as the baseline for calculating gas concentration.

Calculate the gas concentration

```
    float get_gas_data() {
    float RS = 0;
    float ppm = 0;

    while (!get_gas_voltage()) {
        delay(10);
    }

    RS = ((VCC - gas_data.gas_voltage) * gas_data.RL) / gas_data.gas_voltage;

    ppm = pow(((COEFF_A * gas_data.R0) / RS), COEFF_B) * COEFF_C;

    return ppm;
}
```

This function first obtains the current voltage value through **get_gas_voltage**, then uses the formula to calculate the sensor's resistance **RS**.Using the sensor's resistance and known coefficients (**COEFF_A, COEFF_B, COEFF_C**), it calculates the gas concentration (PPM).

LED Control

```
    void led_on() {
    strip.setPixelColor(0, strip.Color(255, 255, 255));
    strip.setPixelColor(1, strip.Color(255, 255, 255));
    strip.show();
}

void led_off() {
    strip.setPixelColor(0, strip.Color(0, 0, 0));
    strip.setPixelColor(1, strip.Color(0, 0, 0));
```

```
   strip.show();
}
```

led_on and **led_off** control the switching of the two LEDs. When the gas concentration exceeds the set threshold, **led_on** is called to turn on the LED. Otherwise, **led_off** is called to turn off the LED.

The main program consists of the setup and loop functions

```
   void setup() {
   Serial.begin(115200);
   gas_init();
   Serial.println("Calibrating R0 in clean air...");
   float r0 = get_r0_calibration();
   Serial.print("R0 = ");
   Serial.print(r0);
   Serial.println(" kOhm");
}

void loop() {
   static float last_ppm_data = 0;
   static bool led_state = false;
   static unsigned long last_toggle = 0;

   unsigned long now = millis();
   float ppm_data = get_gas_data();
   if (ppm_data >= ALARM_THRESHOLD_PPM) {
      led_on();
   } else {
      led_off();
   }

   Serial.print("Gas ppm = ");
   Serial.println(ppm_data, 2);
   delay(20);
}
```

setup: Initializes the serial communication, sets up the sensor, performs R0 calibration, and prints the results to the serial monitor.

loop: Continuously retrieves gas concentration data. If the concentration exceeds the threshold, it turns on the LED and triggers the alarm; otherwise, it turns off the LED. After each loop, there is a 20-millisecond delay.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson16---Microphone recording

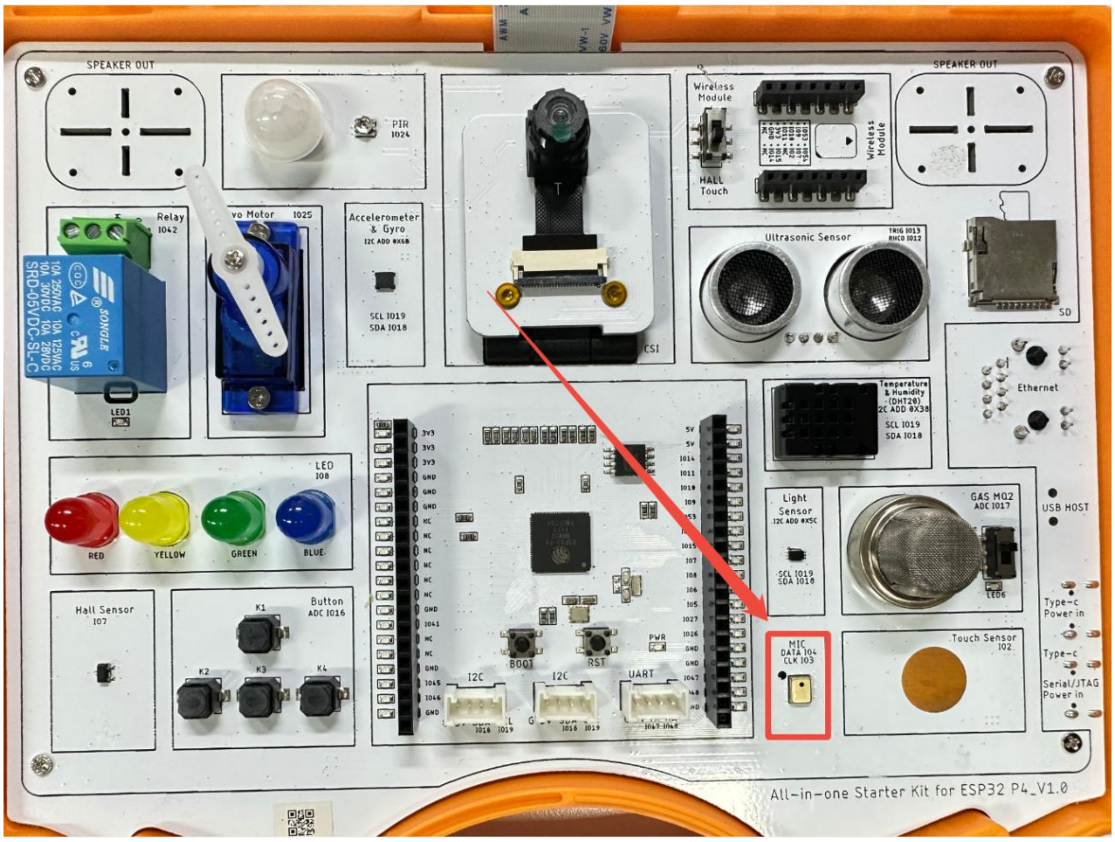
Introduction

In this lesson, we'll explore the microphone module (mic) on the"All-in-One Starter Kit for ESP32-P4"development board. The microphone module is essential for voice interaction. The practical example for this session will be recording audio with the microphone module and saving the recorded audio files to an SD card.

Learning Goals

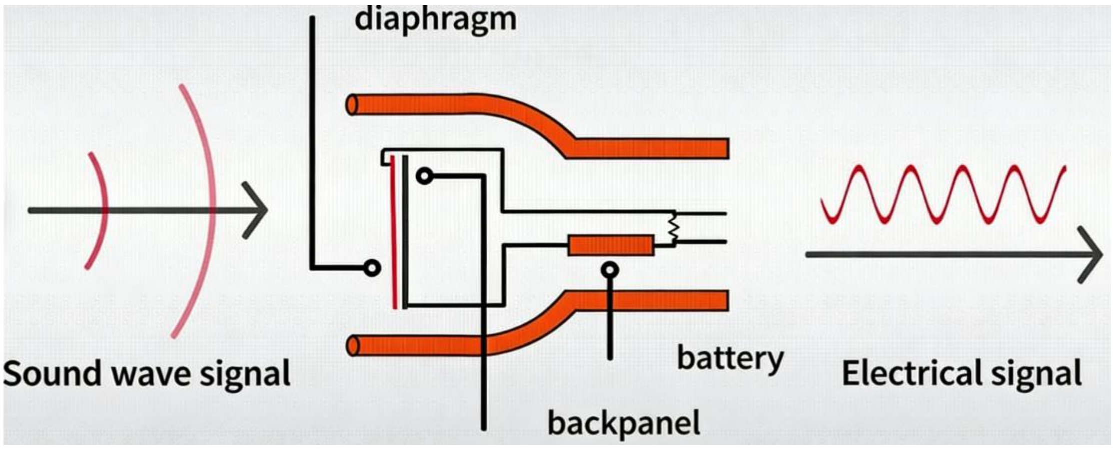
1.Understand how the microphone module works
2.Complete the project of recording audio using the microphone module and saving it to the SD card

Hardware Used in This Lesson:

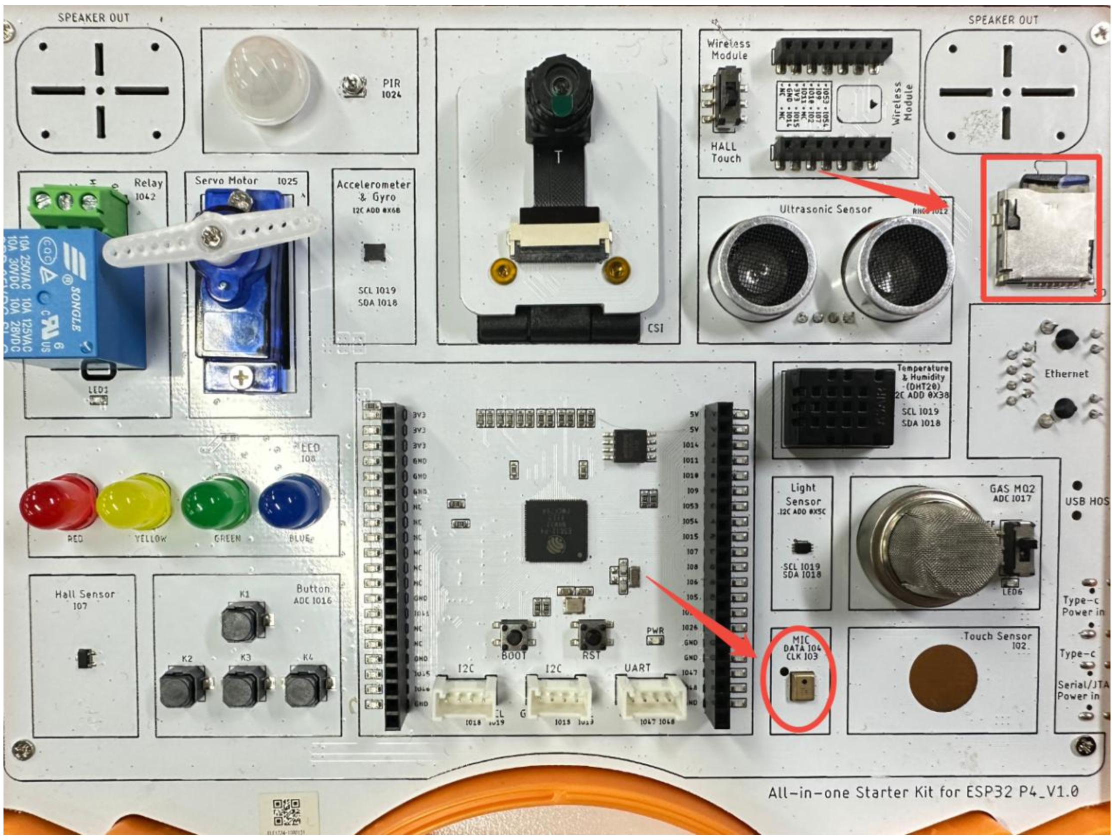


The working principle of MIC Module

This diagram shows the working principle of a microphone module. When sound wave signals (on the left) enter the microphone, they affect the diaphragm inside the microphone. The movement of the diaphragm alters the circuit and electrical signals within the microphone. Specifically, the sound waves cause the diaphragm to vibrate, which changes the resistance or capacitance near the diaphragm. These changes are converted into electrical signals (on the right) through the microphone's battery and backing circuit. This electrical signal can then be amplified and further processed for various audio applications. This process allows the microphone to convert sound into an electrical signal for use by devices.



Running Effect Diagram:



After inserting the SD card and running the program, we speak into the MIC module. It will record for 5 seconds and generate an audio file, which will be stored on the SD card.

Note: The SD card used here is formatted as FAT32. SD cards with other formats may not be recognized.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson16_Microphone_recording

Key Explanation:

Include Header Files

```
#include <Arduino.h>
#include <FS.h>
#include <SD_MMC.h>            // Depending on SD card type, SD.h can also be used
#include <ESP_I2S.h>           // Provides I2SClass definition
#include <esp_ldo_regulator.h>  // ESP32-P4 specific LDO management
#include <sys/unistd.h>          // Include system calls for file handling
#include <sys/stat.h>            // Include functions for file status and permissions
#include <esp_vfs_fat.h>         // Include ESP-IDF FAT filesystem support for SD card
#include <sdmmc_cmd.h>             // Include SDMMC card command definitions and
helpers
#include <driver/sdmmc_host.h>    // Include SDMMC host driver for SD card
communication
```

Arduino.h： Imports the core Arduino library, providing essential functions like setup(), loop(), Serial, delay(), and others.

FS.h： Provides a unified interface for file operations.

SD_MMC.h： The SD card driver library. It uses SDIO for faster speeds and supports both 1-bit and 4-bit modes.

ESP_I2S.h： An I2S wrapper class for the ESP32, providing functions like I2SClass, readBytes(), and begin(). It's used for collecting data from the microphone.

esp_ldo_regulator.h： A dedicated LDO power control for ESP32-P4, used for software control of internal voltage output.

sys/unistd.h、sys/stat.h： Used for low-level file system operations (reading, writing, permissions).

esp_vfs_fat.h： For mounting and working with the FAT file system.

sdmmc_cmd.h、 driver/sdmmc_host.h： Low-level commands and interface controls for SD card management.

GPIO Definiton

```
#define AUDIO_GPIO_CTRL           (30)     // GPIO pin number for audio power
#define AUDIO_POWER_ENABLE        (LOW)     // GPIO set low level to enable audio
power
#define AUDIO_POWER_DISABLE       (HIGH)   // GPIO set high level to disable audio
power
#define MIC_GPIO_CLK              (3)      // GPIO pin number for microphone BCLK (Bit
Clock of PDM)
#define MIC_GPIO_SDIN              (4)      // GPIO pin number for microphone SDIN
```

```
(Serial Data of PDM)
// SD card GPIO with SD_MMC
#define SD_GPIO_MMC_CLK       (43)
#define SD_GPIO_MMC_CMD       (44)
#define SD_GPIO_MMC_D0        (39)
// SD card GPIO with SPI
#define SD_GPIO_SPI_CLK       SD_GPIO_MMC_CLK
#define SD_GPIO_SPI_MOSI      SD_GPIO_MMC_CMD
#define SD_GPIO_SPI_MISO      SD_GPIO_MMC_D0
```

AUDIO_GPIO_CTRL：GPIO pin that controls the power for the audio module, activated by a low level signal.

MIC_GPIO_CLK / MIC_GPIO_SDIN：I2S BCLK (clock) and SDIN (data) for the PDM microphone.

SD_GPIO_MMC_x：SD card interface (using SD_MMC, single-line mode).

Audio Parameters and Buffer

```
#define SAMPLE_RATE        16000          // Sample rate 16kHz
#define RECORD_SECONDS     5              // Recording duration (seconds)
#define BYTE_RATE          (SAMPLE_RATE * 16 / 8) // Bytes per second for 16-bit mono
#define BUFFER_SIZE        16000          // Buffer size for each read (bytes)
// ========== Global I2S object ==========
static I2SClass i2s_mic;                  // I2S object for microphone input
// ========== Buffer ==========
uint8_t i2s_readraw_buff[BUFFER_SIZE];   // Temporary buffer to store I2S data
```

Sample Rate: 16 kHz, 16-bit mono PCM

BYTE_RATE：The number of audio bytes per second = 16000 * 16 / 8 = 32,000 bytes per second

BUFFER_SIZE：The amount of data per second (for convenient read/write in one go)

i2s_mic：The I2S microphone object

i2s_readraw_buff：A temporary buffer for raw audio data

Create SD Structure and Mounting Path

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

static sdmmc_card_t *card;              // Pointer to SD card information
const char sd_mount_point[] = SD_MOUNT_POINT;
```

Power Initialization Function

```
// Initialize ESP32-P4 LDO power rails for audio and peripheral devices
esp_err_t board_p4_ldo_init()
{
    esp_err_t err = ESP_OK;
    esp_ldo_channel_handle_t ldo3_handle = NULL;
    esp_ldo_channel_config_t ldo3_cfg = {
        .chan_id = 3,                // LDO Channel 3
        .voltage_mv = 2500,        // Set to 2500mV (2.5V)
    };

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

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

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

    return ESP_OK;
}
```

This function configures the output voltage of the internal LDO regulator on the ESP32-P4 to supply power to external peripherals. Its main purpose is to handle the "hardware power-up" process; without it, the peripherals won't function at all.

Generate WAV File Header and Write to Memory

```
// ========== Generate WAV header ==========
void generate_wav_header(uint8_t *header, size_t data_size, uint32_t sample_rate) {
    // RIFF chunk descriptor
    memcpy(header, "RIFF", 4);
    uint32_t chunk_size = data_size + 36;
    memcpy(header + 4, &chunk_size, 4);
    memcpy(header + 8, "WAVE", 4);

    // fmt sub-chunk
    memcpy(header + 12, "fmt ", 4);
    uint32_t fmt_size = 16;
    memcpy(header + 16, &fmt_size, 4);
    uint16_t audio_format = 1;              // PCM
    memcpy(header + 20, &audio_format, 2);
    uint16_t num_channels = 1;              // Mono
    memcpy(header + 22, &num_channels, 2);
    memcpy(header + 24, &sample_rate, 4);
    uint32_t byte_rate = sample_rate * num_channels * 16 / 8;
    memcpy(header + 28, &byte_rate, 4);
    uint16_t block_align = num_channels * 16 / 8;
    memcpy(header + 32, &block_align, 2);
    uint16_t bits_per_sample = 16;          // 16-bit samples
    memcpy(header + 34, &bits_per_sample, 2);

    // data sub-chunk
    memcpy(header + 36, "data", 4);
    memcpy(header + 40, &data_size, 4);
}
```

This function generates a standard WAV file header (44 bytes) and writes it to memory. The structure of a WAV file consists of a 44-byte header followed by audio data. Without this header, the player won't know the sample rate, whether it's mono or stereo, or the data length, which can result in the file not opening or playing incorrectly.

Initialize I2S Peripheral

```
// ========== Initialize I2S microphone (PDM mode) ==========
bool init_mic() {
    // Set GPIO pins for PDM receive mode
    i2s_mic.setPinsPdmRx(MIC_GPIO_CLK, MIC_GPIO_SDIN);

    // Begin I2S in PDM receive mode with 16kHz, 16-bit mono
        if (!i2s_mic.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT,
I2S_SLOT_MODE_MONO, I2S_STD_SLOT_LEFT)) {
        Serial.println("PDM input initialization failed!");
        while (1) delay(1000);  // Halt execution on failure
    }

    return true;
}
```

Initialize the I2S Peripheral to allow the ESP32-P4 to read audio data from the PDM microphone and automatically convert it to PCM.

I2S_MODE_PDM_RX：PDM receive mode, meaning data is read from the microphone in PDM format.
16000：Target PCM sample rate.
I2S_DATA_BIT_WIDTH_16BIT：Output data is 16-bit PCM.
I2S_SLOT_MODE_MONO：Mono mode.
I2S_STD_SLOT_LEFT：Use the left channel for data output.

Read audio from the I2S microphone for a specified duration, write it to a WAV file on the SD card, and return the total file size or an error code.

```
// ========== Record audio and save as WAV to SD card ==========
int mic_read_to_sd(const char *filename, size_t rec_seconds, size_t *out_size) {
    if (rec_seconds > 3600 || filename == NULL) {
        return 1; // Invalid parameter
    }
    if (BUFFER_SIZE < 16000) {
        return 1; // Buffer too small
    }

    size_t rec_size = rec_seconds * BYTE_RATE;          // Total audio data size
    uint8_t wav_header[44];
    generate_wav_header(wav_header, rec_size, SAMPLE_RATE);

    // Open SD card file in binary write mode
    FILE *file = fopen(filename, "wb");
    if (!file) {
        Serial.println("Failed to open file for writing");
        return 2;
    }

    // Write WAV header
    if (fwrite(wav_header, 1, 44, file) != 44) {
        fclose(file);
        return 3;
    }

    size_t written_total = 0;
    while (written_total < rec_size) {
        size_t bytes_to_read = min((size_t)BUFFER_SIZE, rec_size - written_total);
        size_t bytes_read = 0;

        // Read bytes from I2S microphone
        bytes_read = i2s_mic.readBytes((char*)i2s_readraw_buff, bytes_to_read);
        if (bytes_read == 0) {
            Serial.println("I2S read failed");
            fclose(file);
            return 4;
        }

        // Seek to correct position in file (after WAV header)
        if (fseek(file, written_total + 44, SEEK_SET) != 0) {
            fclose(file);
            return 5;
        }

        // Write buffer data to file
        if (fwrite(i2s_readraw_buff, 1, bytes_read, file) != bytes_read) {
            fclose(file);
            return 6;
        }

        written_total += bytes_read; // Increment total written bytes
    }

    fclose(file);
    if (out_size != NULL) {
        *out_size = rec_size + 44;   // Return total file size including header
    }
    return 0;
}
```

```
if (rec_seconds > 3600 || filename == NULL) {
     return 1; // Invalid parameter
}
```

If the recording time exceeds 1 hour or the file name is empty, return an error.

```
if (BUFFER_SIZE < 16000) {
     return 1; // Buffer too small
}
```

If the buffer is too small, it will cause the I2S data read to fail, and an error will be returned.

```
size_t rec_size = rec_seconds * BYTE_RATE;
```

Calculate the total number of bytes for the recording:BYTE_RATE = SAMPLE_RATE * 16 / 8.

```
uint8_t wav_header[44];
generate_wav_header(wav_header, rec_size, SAMPLE_RATE);
```

Call the function to generate the WAV file header and create a standard 44-byte WAV file header.

```
FILE *file = fopen(filename, "wb");
if (!file) {
        Serial.println("Failed to open file for writing");
        return 2;
}
```

Open the SD file, and if it fails to open, return 2.

```
if (fwrite(wav_header, 1, 44, file) != 44) {
        fclose(file);
        return 3;
}
```

Write the WAV header to the file, writing 44 bytes. If the write fails, close the file and return 3.

```
size_t written_total = 0;
while (written_total < rec_size) {
```

Continuously read and write audio data.

```
size_t bytes_to_read = min((size_t)BUFFER_SIZE, rec_size - written_total);
size_t bytes_read = 0;
```

Read the data in chunks each time to prevent the last block of data from exceeding the total length.

```
bytes_read = i2s_mic.readBytes((char*)i2s_readraw_buff, bytes_to_read);
if (bytes_read == 0) {
        Serial.println("I2S read failed");
        fclose(file);
        return 4;
}
```

Read PCM data from the microphone, and if the read fails, return 4.

```
if (fseek(file, written_total + 44, SEEK_SET) != 0) {
        fclose(file);
        return 5;
}
```

Adjust the file write position to skip the 44-byte WAV header before writing the audio data.

```
if (fwrite(i2s_readraw_buff, 1, bytes_read, file) != bytes_read) {
        fclose(file);
        return 6;
}
```

Write the data from the buffer to the file, and if the write fails, return 6.

```
written_total += bytes_read; // Increment total written bytes
```

Update the total number of bytes written, accumulating the data until the total reaches rec_size.

```
fclose(file);
if (out_size != NULL) {
        *out_size = rec_size + 44;    // Return total file size including header
}
return 0;
```

Close the file and return the file size

Initialize the SD card (using the SDMMC interface mode) and mount the FAT file system, allowing the ESP32 to interact with the SD card through the file system (e.g., reading/writing files).

```
// ========== Initialize SD card (SD_MMC mode) ==========
esp_err_t sd_init()
{
    esp_err_t err = ESP_OK;
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
                // If SD card file system is not FAT32, mount fails unless
format_if_mount_failed=true
        .format_if_mount_failed = false,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024,
    };

    sdmmc_host_t host =   SDMMC_HOST_DEFAULT(); // Use default SDMMC host
    host.slot = SDMMC_HOST_SLOT_0;
    host.max_freq_khz = 10000;                  // Limit clock frequency

     sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT(); // Default
slot config
    slot_config.clk = (gpio_num_t)SD_GPIO_MMC_CLK;
    slot_config.cmd = (gpio_num_t)SD_GPIO_MMC_CMD;
    slot_config.d0 = (gpio_num_t)SD_GPIO_MMC_D0;
    slot_config.width = 1;   // Use 1-line SDIO
      slot_config.flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP; // Enable internal
pull-ups

    // Mount SD card filesystem
          err = esp_vfs_fat_sdmmc_mount(sd_mount_point, &host, &slot_config,
&mount_config, &card);
    if (err != ESP_OK) {
        if (err == ESP_FAIL) {
            Serial.println("Failed to mount filesystem.");
        } else {
            Serial.printf("Failed to initialize the card (%s).\n", esp_err_to_name(err));
        }
        return err;
    }
    Serial.println("SD card mounted successfully");
    sdmmc_card_print_info(stdout, card); // Print SD card info
    return err;
}
```

format_if_mount_failed = false: If the mounting fails, the SD card will not be formatted. If set to true, it will format the SD card upon mounting failure.
max_files = 5: The file system supports up to 5 files being open simultaneously.
allocation_unit_size = 16 * 1024: The file system's allocation unit size is set to 16KB.
sdmmc_host_t host: This is the SDMMC host configuration structure, which controls the communication with the SD card.
SDMMC_HOST_DEFAULT(): Creates an SDMMC host object with the default configuration.
host.slot = SDMMC_HOST_SLOT_0: Selects the first slot for the SD card.
host.max_freq_khz = 10000: Sets the maximum operating frequency of the SD card to 10 MHz. This frequency can be adjusted based on the SD card's type and quality. If set too high, it could cause instability.
SDMMC_SLOT_CONFIG_DEFAULT(): Uses the default slot configuration.
slot_config.clk = (gpio_num_t)SD_GPIO_MMC_CLK: Sets the GPIO pin for the SD card clock signal.
slot_config.cmd = (gpio_num_t)SD_GPIO_MMC_CMD: Sets the GPIO pin for the SD card command signal.
slot_config.d0 = (gpio_num_t)SD_GPIO_MMC_D0: Sets the GPIO pin for the SD card data pin D0.
slot_config.width = 1: Uses a 1-bit width (SDIO) communication mode, meaning data is transferred over a single line. You can also configure it to 4-bit width, but it requires hardware support.
slot_config.flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP: Enables internal pull-up resistors to ensure stable signals.
esp_vfs_fat_sdmmc_mount: This is the function used to mount the SD card file system and initialize the SD card.
sd_mount_point: Specifies the path where the SD card will be mounted (e.g., /sdcard). This is the file system path through which you can access the files on the SD card.
&host: Passes the host configuration structure, controlling the SD card's clock frequency and slot.
&slot_config: Passes the slot configuration structure, controlling the SD card's interface pins and width.
&mount_config: Mount configuration structure, which specifies the behavior of the file system.
&card: Returns the SD card information structure.

setup function

```
void setup() {
    Serial.begin(115200);                    // Initialize Serial for debugging
```

Serial.begin(115200);: Initializes serial communication with a baud rate of 115200. This line of code is mainly used for debugging, allowing information to be output to the serial monitor to check the program's running status.

```
    board_p4_ldo_init();                    // Initialize LDO power rails
```

Call the function to initialize the LDO (Low Dropout) power rail.

```
    if (sd_init() != ESP_OK) {
            Serial.println("SD card initialization failed!");
            return;
      }
```

Initialize the SD card. If initialization fails, print "SD card initialization failed!" and return.

```
    if (!init_mic()) {
            Serial.println("Mic initialization failed");
            return;
      }
```

Initialize the microphone. If initialization fails, print an error message and return.

```
    // Start recording 5 seconds
      Serial.println("Start recording...");
      size_t total_size;
      int ret = mic_read_to_sd("/sdcard/test.wav", RECORD_SECONDS, &total_size);
      if (ret == 0) {
          Serial.printf("Recording finished, file size: %u bytes\n", total_size);
      } else {
          Serial.printf("Recording failed, error code: %d\n", ret);
      }

      // Stop I2S interface
      i2s_mic.end();
  }
```

Before starting the recording, print "Start recording..." to the serial monitor to indicate the beginning. Call the microphone recording function to save the audio to the SD card at /sdcard/test.wav for 5 seconds. If the return value is 0, the recording was successful—print a success message and the file size to the serial monitor. If the return value is not 0, print a failure message along with the error code. Finally, stop the I2S interface and microphone communication, releasing all related resources to ensure the microphone stops recording.

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.

Lesson17---Speaker

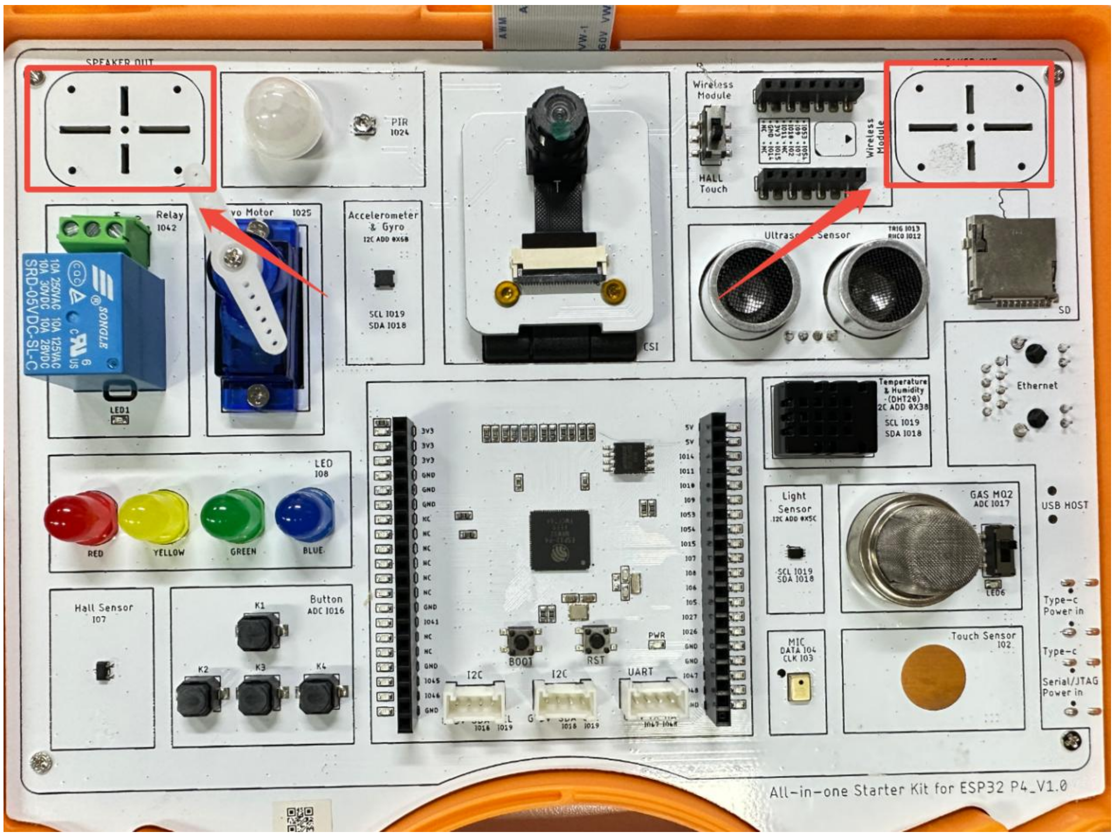
Introduction

In this lesson, we will learn how to drive a speaker on the "ESP32-P4 All-in-One Starter Kit," read, and play audio files stored on an SD card. By the end of this course, you will understand how a speaker works and gain hands-on experience with reading and playing audio files.

Learning Goals

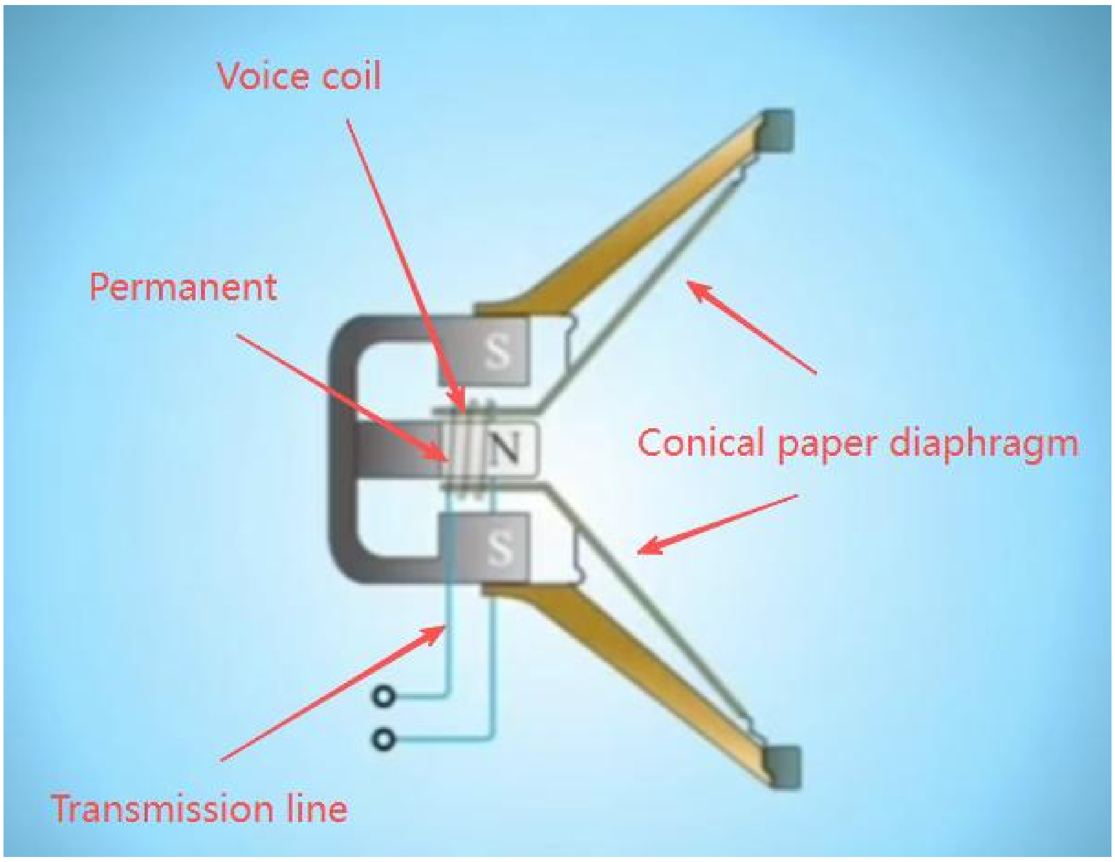
1.Understand how a speaker works
2.Complete a project that reads and plays audio files from an SD card

Hardware Used in This Lesson:

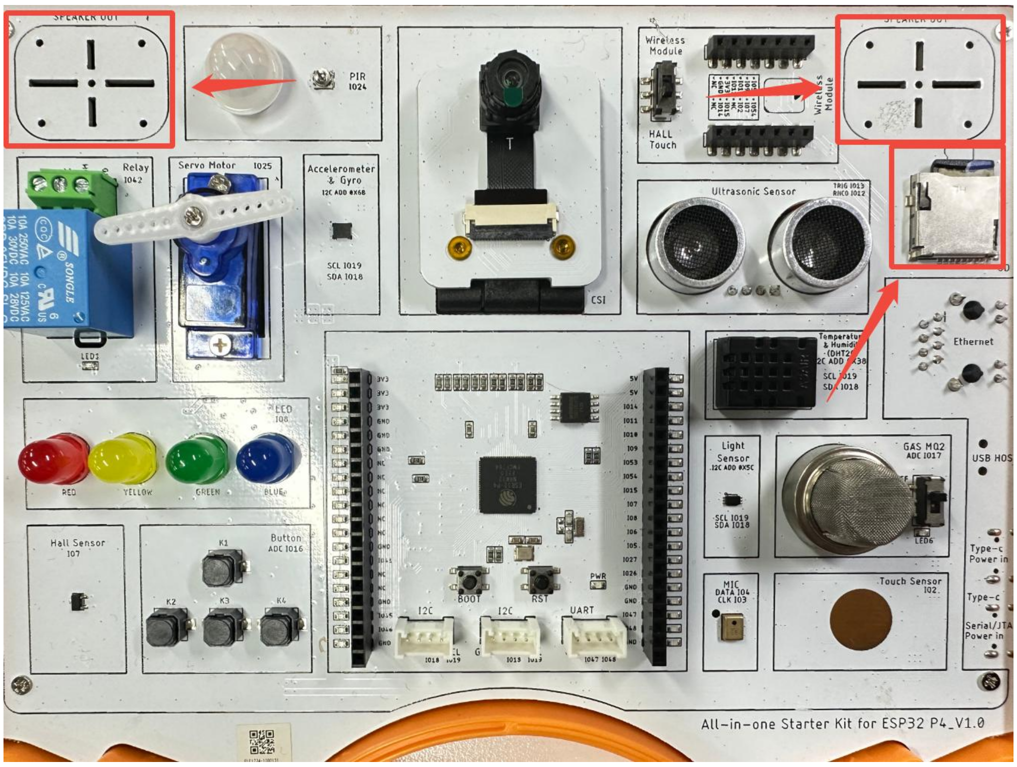


The working principle of Speaker Module

A speaker is mainly composed of a permanent magnet, a voice coil, and a cone-shaped diaphragm. When an audio current flows through the transmission line into the voice coil, the coil experiences an Ampere force within the magnetic field of the permanent magnet, causing it to move back and forth as the current changes.The vibration of the voice coil drives the connected cone diaphragm to move in sync, continuously compressing and expanding the surrounding air. This process converts electrical signals into mechanical vibrations, which are then transformed into sound waves that the human ear can perceive, enabling audio playback.



Running Effect Diagram:



After running the program, the speaker will play the audio files stored on the SD card.

Complete Code:

Click the link below to open the complete code:
https://github.com/Elecrow-RD/All-in-one-Starter-Kit-for-ESP32-P4-with-Common-Board-design/tree/master/example/Arduino_code/Lesson17_Speaker

Key Explanation:

Include Header Files

```
#include <Arduino.h>
#include <ESP_I2S.h>               // ESP32 I2S Library
#include <string.h>                // Include standard string manipulation functions
#include <esp_log.h>               // ESP-IDF logging library
#include <esp_err.h>               // ESP-IDF error codes
#include <esp_ldo_regulator.h>   // ESP32-P4 specific LDO management
#include <sys/unistd.h>            // Include system calls for file handling
#include <sys/stat.h>              // Include functions for file status and permissions
#include <esp_vfs_fat.h>           // Include ESP-IDF FAT filesystem support for SD card
#include <sdmmc_cmd.h>             // Include SDMMC card command definitions and
helpers
#include <driver/sdmmc_host.h>     // Include SDMMC host driver for SD card
communication
#define EXAMPLE_MAX_CHAR_SIZE 64     // Maximum character buffer size for file
read/write operations
#define SD_MOUNT_POINT "/sdcard"    // Default SD card mount point path
```

Arduino.h： Imports the core Arduino library, providing essential functions like setup(), loop(), Serial, delay(), and others.

ESP_I2S.h： Includes the ESP32 I2S library, allowing you to handle audio through the I2S interface on the ESP32.

string.h： Used for manipulating C-style strings or memory blocks.

esp_log.h： Used for debugging and printing log information, supporting different levels (INFO, WARN, ERROR, DEBUG).

esp_err.h： Used for handling error statuses returned by functions.

esp_ldo_regulator.h： In power-sensitive applications, this library allows software-based adjustment of ESP32 power parameters.

sys/unistd.h & sys/stat.h： On the ESP32, these libraries support operations on the SD card or file system.

esp_vfs_fat.h： Eases reading and writing files on the SD card with the ESP32.

sdmmc_cmd.h、driver/sdmmc_host.h： Low-level commands and interface controls for SD card management.

#define EXAMPLE_MAX_CHAR_SIZE 64： Makes it easy to modify buffer size, maintaining code maintainability.

#define SD_MOUNT_POINT "/sdcard"： Centralizes mount point management, making the code easier to port or modify.Centralizes mount point management, making the code easier to port or modify.

Pin definitions

```
// SD card GPIO with SD_MMC interface
#define SD_GPIO_MMC_CLK       (43)
#define SD_GPIO_MMC_CMD       (44)
#define SD_GPIO_MMC_D0        (39)
// SD card GPIO with SPI interface
#define SD_GPIO_SPI_CLK       SD_GPIO_MMC_CLK
#define SD_GPIO_SPI_MOSI      SD_GPIO_MMC_CMD
#define SD_GPIO_SPI_MISO      SD_GPIO_MMC_D0

#define AUDIO_GPIO_CTRL            (6)      // GPIO pin number for audio power control
#define AUDIO_POWER_ENABLE         (LOW)    // GPIO set low level to enable audio power
#define AUDIO_POWER_DISABLE        (HIGH)   // GPIO set high level to disable audio power

#define AUDIO_GPIO_LRCLK           (21)     // GPIO pin number for I2S LRCLK (Left-Right Clock)
#define AUDIO_GPIO_BCLK            (22)     // GPIO pin number for I2S BCLK (Bit Clock)
#define AUDIO_GPIO_SDATA           (23)     // GPIO pin number for I2S SDATA (Serial Data)

#define MIC_GPIO_CLK               (3)      // GPIO pin number for microphone BCLK (Bit Clock of PDM)
#define MIC_GPIO_SDIN              (4)      // GPIO pin number for microphone SDIN (Serial Data of PDM)
```

```
#define AUDIO_ERROR(format, ...) Serial.printf("ERROR: " format "\n", ##__VA_ARGS__)
#define AUDIO_INFO(format, ...) Serial.printf("INFO: " format "\n", ##__VA_ARGS__)
```

These two macros are used to format and print log information to the serial port. **AUDIO_ERROR** is used to print error messages, while **AUDIO_INFO** is used to print regular informational messages.

```
static sdmmc_card_t *card;                    // SD card info pointer
const char sd_mount_point[] = SD_MOUNT_POINT;
```

card： A pointer of type **sdmmc_card_t** used to store information about the SD card. This pointer allows access to the SD card's status and data.

sd_mount_point： The mount path for the SD card, pointing to the previously defined **/sdcard**.

```
static I2SClass i2s_spk;                    // Create an I2SClass speaker instance
```

Create an instance of type **I2SClass** named **i2s_spk**, which is used to handle audio output to the speaker.

WAV file header structure

```
stypedef struct {
      char riff[4];                   // "RIFF"
      uint32_t fileSize;           // File size
      char wave[4];                    // "WAVE"
      char fmt[4];                     // "fmt "
      uint32_t fmtSize;             // fmt chunk size
      uint16_t audioFormat;      // Audio format (1 = PCM)
      uint16_t numChannels;      // Number of channels
      uint32_t sampleRate;       // Sample rate
      uint32_t byteRate;           // Byte rate
      uint16_t blockAlign;       // Block alignment
      uint16_t bitsPerSample; // Bits per sample
      char data[4];                    // "data"
      uint32_t dataSize;           // Data chunk size
} wav_header_t;
```

This structure describes the WAV file header. A WAV file consists of multiple chunks, each containing different audio metadata such as sample rate and bit depth. This structure helps parse the header information of a WAV file.

riff[4]：A character array storing the string "RIFF," identifying the file as a RIFF format.
fileSize：The size of the entire WAV file (excluding the "RIFF" and "WAVE" identifiers).
wave[4]：A character array storing the string "WAVE," indicating this is a WAV audio file.
fmt[4]：A character array storing the string "fmt ", marking the audio format chunk.
fmtSize：Size of the fmt chunk, usually 16 for PCM format.
audioFormat：Audio format, where 1 indicates PCM (Pulse Code Modulation).
numChannels：Number of channels, typically 1 (mono) or 2 (stereo).
sampleRate：Sampling rate, e.g., 44100 Hz means 44,100 samples per second.
byteRate：Byte rate, indicating how many bytes are transmitted per second, usually calculated as sampleRate * numChannels * bitsPerSample / 8.
blockAlign：Block alignment, representing the size of each sample block, usually numChannels * bitsPerSample / 8.
bitsPerSample：Bits per sample, typically 8, 16, or 32.
data[4]：A character array storing the string "data," marking the start of the data chunk.
dataSize：Size of the data chunk, i.e., the size of the audio data excluding the WAV header.

```
if (file == NULL)
{
      AUDIO_ERROR("File pointer is NULL");
      return false;
}
```

If the passed file pointer is NULL, it indicates that the file was not opened correctly. In this case, the function will return false and print an error log.

```
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;
}
```

Use **ftell()** to get the current position of the file pointer and store it in **original_position**. This allows you to restore the file pointer to its original position if you need to backtrack later.

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

Use **fseek()** to move the file pointer to the beginning of the file. **SEEK_SET** means the pointer will be moved from the start of the file.

```
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;
```

Read the header of the WAV file (44 bytes) and store it in the **header** array. The standard header size for a WAV file is 44 bytes.

```
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;
}
```

The first **memcmp()** checks if the first 4 bytes of the file header are "RIFF," which is the RIFF chunk identifier for a WAV file.

The second **memcmp()** checks if bytes 9 to 12 of the file header are "WAVE," which is the format identifier for a WAV file.

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

Check if the 4 bytes starting from byte 13 are "fmt ", which is the identifier for the format information sub-chunk (fmt) in a WAV file.

```
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;
}
```

Read the audio format field (audio_format) from the file header. It should be 1, indicating PCM (Pulse Code Modulation) format. If it's not 1, return an error.

```
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;
}
```

Check the number of channels specified in the file. If it is neither 1 (mono) nor 2 (stereo), return an error.

```
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;
}
```

Check the sample rate field to ensure it matches one of the common sample rates, such as 8000 Hz, 16000 Hz, or 44100 Hz. If it's not one of these, return an error.

```
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;
}
```

Check the bits per sample field to ensure it matches a common bit depth, such as 8, 16, 24, or 32 bits. If it's not one of these values, return an error.

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

Check the identifier field of the data chunk ("data"), which indicates the start of the audio data.

```
uint32_t file_size = *(uint32_t *)(header + 4) + 8; // RIFF block size + 8-byte header
uint32_t data_size = *(uint32_t *)(header + 40);
```

file_size calculates the total size of the file (including the header and data chunk).
data_size retrieves the size of the audio data chunk.

```
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);
```

Print the basic information of the WAV file, including the number of channels, sample rate, bit depth, data size, and file size.

```
fseek(file, original_position, SEEK_SET); // Restore original file position
return true;
```

Restore the file pointer to its original position, ensuring that the caller can continue reading the file.

```
static void set_Audio_ctrl(bool state) {
```

Define a static function **set_Audio_ctrl** to control the power switch of the audio hardware.

```
bool status = !state;
```

Invert the value of **state** to obtain **status**. If **state** is true, then **status** will be false, indicating that the audio hardware is turned off. If **state** is false, then **status** will be true, meaning the audio hardware is turned on.

```
digitalWrite(AUDIO_GPIO_CTRL, status);
```

Use the **digitalWrite()** function to set the voltage level of the GPIO pin **AUDIO_GPIO_CTRL**.

```
if (state) {
        AUDIO_INFO("Audio hardware enabled");
} else {
        AUDIO_INFO("Audio hardware disabled");
}
```

Based on the value of **state**, print information about the audio hardware's status. If **state** is true, it means the audio hardware is enabled, so output "Audio hardware enabled." If **state** is false, it means the audio hardware is disabled, so output "Audio hardware disabled."

```
esp_err_t Audio_play_wav_sd(const char *filename)
{
        esp_err_t err = ESP_OK;
        if (filename == NULL)
                return ESP_ERR_INVALID_ARG;
```

A function named **Audio_play_wav_sd** is defined to play a WAV file from a specified path.

```
FILE *fh = fopen(filename, "rb");
if (fh == NULL)
{
        Serial.printf("Failed to open file\n");
        return ESP_ERR_INVALID_ARG;
}
```

Use the **fopen()** function to open the WAV file in binary read mode.

```
if (!validate_wav_header(fh)) // Validate WAV header
{
        Serial.printf("Invalid WAV file format: %s\n", filename);
        fclose(fh);
        return ESP_ERR_INVALID_ARG;
}
```

Call the **validate_wav_header()** function to verify whether the header format of the WAV file is correct.

```
if (fseek(fh, 44 + 4000, SEEK_SET) != 0) // Skip 44-byte header + 4000 bytes of data
{
        Serial.printf("Failed to seek file\n");
        fclose(fh);
        return ESP_FAIL;
}
```

First, skip the 44-byte WAV file header. Then, skip 4000 bytes of data, which is typically used for silent sections or initialization parts in the WAV file.

```
const size_t SAMPLES_PER_BUFFER = 512; // Number of samples per buffer
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);

int16_t *input_buf = (int16_t*)heap_caps_malloc(INPUT_BUFFER_SIZE, MALLOC_CAP_SPIRAM |
MALLOC_CAP_DMA | MALLOC_CAP_32BIT);
int16_t *output_buf = (int16_t*)heap_caps_malloc(OUTPUT_BUFFER_SIZE,
MALLOC_CAP_SPIRAM | MALLOC_CAP_DMA | MALLOC_CAP_32BIT);
```

Set the size of each buffer to **SAMPLES_PER_BUFFER** samples.
input_buf and **output_buf** are the input and output buffers.
heap_caps_malloc is used to allocate memory from specific memory regions (such as SPI RAM, DMA area, or 32-bit aligned memory).

```
digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_ENABLE); // Enable audio power
```

Use **digitalWrite()** to set the GPIO pin that controls the audio hardware power, turning on the audio hardware power.

```
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] * 2; // Linear volume multiplication
                if (volume_data > 32767) volume_data = 32767;
                else if (volume_data < -32768) volume_data = -32768;
                output_buf[i] = (int16_t)volume_data; // Left channel
        }

        bytes_to_write = samples_read * sizeof(int16_t);
        bytes_written = i2s_spk.write((uint8_t*)output_buf, bytes_to_write); // Write to I2S
        if (bytes_written != bytes_to_write)
```

```
    {
        Serial.printf("I2S write failed: %s, written: %d/%d\n", esp_err_to_name(err),
bytes_written, bytes_to_write);
        break;
    }
    total_samples += samples_read;
}
```

In the **while** loop, read the data chunks of the WAV file until the end of the file.
Each time, read **SAMPLES_PER_BUFFER** audio samples into the **input_buf** buffer.
Convert the mono data into stereo: For each sample, use a simple linear gain to amplify it, then store it into the **output_buf** buffer (left channel data).
Write the data from **output_buf** to the audio hardware via the I2S interface.

```
digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE); // Disable audio power
```

Use **digitalWrite()** to turn off the power of the audio hardware.

```
free(input_buf);
free(output_buf);
fclose(fh);
Serial.printf("Audio playback completed: %d samples\n", total_samples);
return err;
```

Free the memory buffers allocated for the audio data. Close the opened file. Print the number of samples played, indicating that the playback has finished.

```
esp_err_t board_p4_ldo_init()
{
    esp_err_t err = ESP_OK;
```

Define a function **board_p4_ldo_init** to initialize the LDO (Low Dropout Regulator) channel.

```
esp_ldo_channel_handle_t ldo3_handle = NULL;
esp_ldo_channel_config_t ldo3_cfg = {
    .chan_id = 3,              // LDO3 channel
    .voltage_mv = 2500,        // 2.5V
};
```

Define the channel handle **ldo3_handle** and the configuration structure **ldo3_cfg** for LDO3.

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

Call the **esp_ldo_acquire_channel()** function to initialize the LDO3 channel, passing the configuration parameters **ldo3_cfg** and the channel handle **ldo3_handle**. The **esp_ldo_acquire_channel()** function will configure the LDO output voltage and return a success status.

```
esp_ldo_channel_handle_t ldo4_handle = NULL;
esp_ldo_channel_config_t ldo4_cfg = {
        .chan_id = 4,
        .voltage_mv = 3300    // LDO4: 3.3V
};
```

Define the channel handle **ldo4_handle** and the configuration structure **ldo4_cfg** for LDO4.

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

Call the **esp_ldo_acquire_channel()** function to initialize the LDO4 channel, passing the configuration parameters **ldo4_cfg** and the channel handle **ldo4_handle**. The function will configure the LDO output voltage and return a success status.

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

A function named **sd_init()** is defined to initialize the SD card and mount it to the file system.

```
esp_vfs_fat_sdmmc_mount_config_t mount_config = {
        .format_if_mount_failed = false,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024,
};
```

Configure the file system mount parameters as follows:
format_if_mount_failed = false: If the mount fails, the SD card will not be formatted.
max_files = 5: A maximum of 5 files can be opened at the same time.
allocation_unit_size = 16 * 1024: The allocation unit size is set to 16KB.

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

Use the **SDMMC_HOST_DEFAULT()** macro to initialize a default SDMMC host configuration.
slot = SDMMC_HOST_SLOT_0: Select the first SD card slot (typically the built-in SD card slot on the ESP32).
max_freq_khz = 10000: Set the maximum frequency to 10 MHz, which can be adjusted as needed (lower frequencies generally offer better stability, but slower speed).

```
sdmmc_slot_config_t slot_config = SDMMC_SLOT_CONFIG_DEFAULT();
slot_config.clk = (gpio_num_t)SD_GPIO_MMC_CLK;
slot_config.cmd = (gpio_num_t)SD_GPIO_MMC_CMD;
slot_config.d0 = (gpio_num_t)SD_GPIO_MMC_D0;
slot_config.width = 1; // 1-bit SDIO
slot_config.flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP;
```

Configure the SD card slot pin settings to ensure proper connection between the ESP32 and the SD card interface:

clk = (gpio_num_t)SD_GPIO_MMC_CLK: Set the GPIO pin for the SD card clock signal.
cmd = (gpio_num_t)SD_GPIO_MMC_CMD: Set the GPIO pin for the SD card command line (GPIO 44).
d0 = (gpio_num_t)SD_GPIO_MMC_D0: Set the GPIO pin for SD card data line 0 (GPIO 39).
width = 1: Select the SD card to operate in 1-bit SDIO mode (indicating a 1-bit data transfer width).
flags |= SDMMC_SLOT_FLAG_INTERNAL_PULLUP: Enable the internal pull-up resistor to ensure signal stability in certain conditions.

```
err = esp_vfs_fat_sdmmc_mount(sd_mount_point, &host, &slot_config, &mount_config, &card);
if (err != ESP_OK) {
	Serial.printf("Failed to initialize SD card (%s)\n", esp_err_to_name(err));
	return err;
}
```

Use the **esp_vfs_fat_sdmmc_mount()** function to mount the SD card to the file system.
sd_mount_point: The path where the SD card will be mounted, typically /sdcard.
&host: The SD card host configuration, including clock frequency, slot, and other parameters.
&slot_config: The SD card slot configuration, including pin configuration and data width.
&mount_config: The file system mount configuration.
&card: The SD card information structure, used to return information about the SD card.

```
Serial.println("SD card mounted successfully");
sdmmc_card_print_info(stdout, card);
return err;
```

If the SD card is successfully mounted, print "SD card mounted successfully."
Use the **sdmmc_card_print_info()** function to print detailed information about the SD card, such as its capacity, type, and SD card version.

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

The function **get_sd_card**_info is used to print detailed information about the mounted SD card. It utilizes the **sdmmc_card_print_info()** function to output the relevant information about the SD card. This function will print details such as the manufacturer, capacity, type, and other information to the standard output (usually the serial terminal).

```
void speaker_init()
{
    i2s_spk.setPins(AUDIO_GPIO_BCLK, AUDIO_GPIO_LRCLK, AUDIO_GPIO_SDATA); // BCLK,
LRCLK, DOUT
    if (!i2s_spk.begin(I2S_MODE_STD, 16000, I2S_DATA_BIT_WIDTH_16BIT,
I2S_SLOT_MODE_MONO, I2S_STD_SLOT_BOTH)) {
        Serial.println("I2S output mode initialization failed!");
        while (1) delay(1000);
    }
    Serial.println("I2S speaker initialized");
}
```

The **speaker_init()** function initializes the I2S output interface to enable the transfer of audio data to the speaker via the I2S bus.

The previous steps defined the functional functions, and now we move on to the main process function.

```
Serial.begin(115200);
delay(1000);
Serial.println("ESP32-P4 Audio Player Starting...");
```

Initialize serial communication, set the baud rate to 115200, and print the device startup information.

```
board_p4_ldo_init(); // Initialize LDOs
```

Call the **board_p4_ldo_init()** function to initialize the Low Dropout Regulator (LDO) on the ESP32-P4. This function ensures that the device can provide a stable power supply to external hardware.

```
pinMode(AUDIO_GPIO_CTRL, OUTPUT);
digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE); // Power off audio initially
```

pinMode(AUDIO_GPIO_CTRL, OUTPUT): Set the **AUDIO_GPIO_CTRL** pin to output mode, used to control the power switch of the audio hardware.
digitalWrite(AUDIO_GPIO_CTRL, AUDIO_POWER_DISABLE): During initialization, turn off the audio hardware power to ensure that no sound is accidentally played when the system starts.

```
if (sd_init() != ESP_OK) {
    Serial.println("SD card initialization failed!");
    return;
}
```

Call the **sd_init()** function to initialize the SD card and mount the file system. If the SD card initialization fails, print an error message and exit the **setup()** function to prevent any subsequent operations from causing errors.

```
speaker_init(); // Initialize speaker
```

Call the **speaker_init()** function to initialize the I2S interface and speaker hardware, ensuring that the speaker can receive and play audio data.

```
if (Audio_play_wav_sd("/sdcard/test.wav") != ESP_OK) { // Play WAV file
        Serial.println("Audio playback failed!");
}
```

Call the **Audio_play_wav_sd()** function to play the WAV file stored on the SD card. The file path is **/sdcard/test.wav**. If playback fails, print the error message "Audio playback failed!".

Upload Code:

Please follow the upload steps (P7) mentioned in the preface to upload the program.