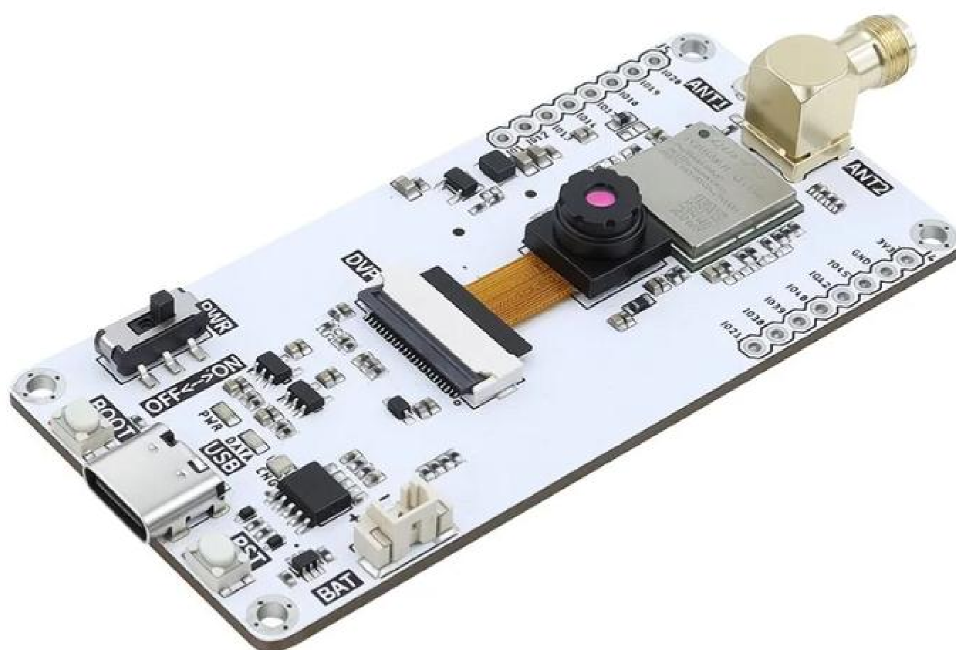
ESP32 WiFi-HaLow module Tutorial

Wi-Fi HaLow (based on the IEEE 802.11ah) is a low-power, long-range wireless communication technology designed by the Wi-Fi Alliance specifically for IoT scenarios. Compared to traditional Wi-Fi technology, it operates in the Sub-1 GHz unlicensed frequency band (mainly the 900MHz band). This frequency offers stronger diffraction and penetration, enabling it to effectively pass through buildings, vegetation, and other obstacles, thereby overcoming the inherent limitations of traditional Wi-Fi in coverage and environmental penetration.

The ESP32 Wi-Fi HaLow camera module is a product that breaks through the limitations of traditional wireless communication distance. Based on the ESP32-S3 and IEEE 802.11ah (Wi-Fi HaLow) technology, it is designed specifically for long-range, high-speed IoT applications. It solves the pain points of traditional Wi-Fi, such as poor wall penetration and short coverage, while also overcoming the shortcomings of LoRa, such as low speed and inability to transmit video. Theoretically, it supports a transmission distance of over 1 kilometer and speeds up to 32 Mbps, making it ideal for IoT applications requiring reliable, high-performance connectivity.

The module integrates a 2-megapixel camera (supporting automatic exposure/gain, image scaling and compression, etc.), an SD card slot, a Type-C interface, and triple wireless connectivity (Wi-Fi + Bluetooth + Wi-Fi HaLow). It features a low-power design and strong signal penetration. Supporting the Arduino development environment, it is an ideal platform for makers and IoT engineers to explore long-range wireless communication.



Lesson02-Usage of "GPIO" 1

Lesson03-Button_Control_LED... 13

Lesson04-DHT20 22

Lesson05-Camera Web Server... 32

In the first lesson, we will set up the development environment for the ESP32 WiFi-HaLow module, primarily installing ESP-IDF for code development and execution. For detailed installation methods and step-by-step instructions, please refer to the Wiki page.

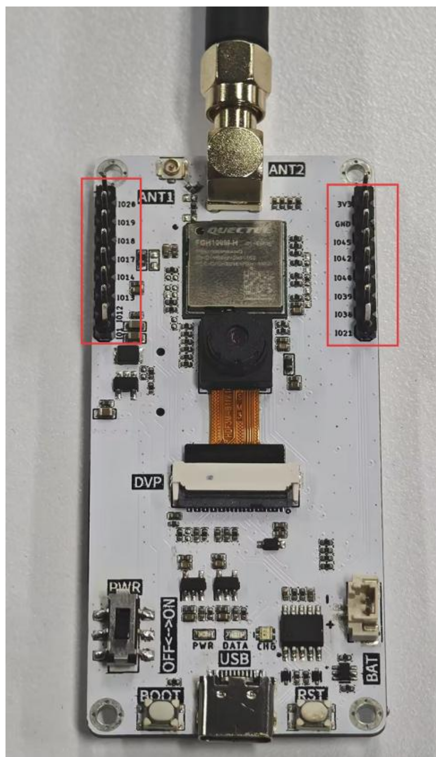
Lesson02 - Usage of "GPIO"

Introduction

In this lesson, we will learn the basic usage methods of "GPIO" pins on the ESP32 WiFi-HaLow module. Through a classic LED blinking experiment, this course demonstrates how to control high and low level output of "GPIO" pins on the development board, so as to realize periodic turning on and off of the LED connected to the corresponding "GPIO" pin.

Hardware Used in This Lesson

The "GPIO" (General Purpose Input Output) ports on ESP32-S3 are important interfaces for the microcontroller to interact with external circuits. They can be used to control LEDs, read key signals, drive display screens, connect sensors and implement various communication functions.



Introduction to GPIO

It is necessary to specifically explain the usage of GPIO pins here. Currently, only

GPIO19 and GPIO20 can be used as regular GPIO pins, while GPIO42 and GPIO45 can be used as IIC communication interfaces for functional development and peripheral expansion. Besides these, the remaining GPIO pins have been occupied by onboard hardware such as cameras and memory cards, and thus cannot be used as regular I/O.

The ESP32-S3 is equipped with abundant "GPIO" pins and adopts a highly flexible "IO Matrix" mechanism. Most peripheral functions can be dynamically mapped to different pins. For example, functions such as "UART", "SPI", "I2C", "PWM" and "ADC" are no longer fixed to specific pins, making hardware design more flexible.
Each "GPIO" can be independently configured as input mode, output mode or input-output mode, and supports pull-up, pull-down, level interrupt and other functions. Some "GPIO" pins also have capabilities including "ADC" analog input, touch detection, "PWM" output and high-speed communication.

In output mode, "GPIO" can output high and low levels to drive external devices; in input mode, it can detect the state of external circuits, such as whether a key is pressed. The typical operating voltage of ESP32-S3 "GPIO" is 3.3V, so attention should be paid to level compatibility when directly connecting peripheral devices.
In addition, certain "GPIO" pins have special purposes, such as "Strapping Pin" for boot configuration, "Flash/PSRAM" communication pins and "USB" function pins.

Therefore, it is necessary to avoid incorrectly occupying key pins in actual development. Overall, the "GPIO" system of ESP32-S3 features rich functions, flexible mapping and strong expansion capability, serving as an important foundation for its wide application in the fields of Internet of Things, smart hardware and embedded control.

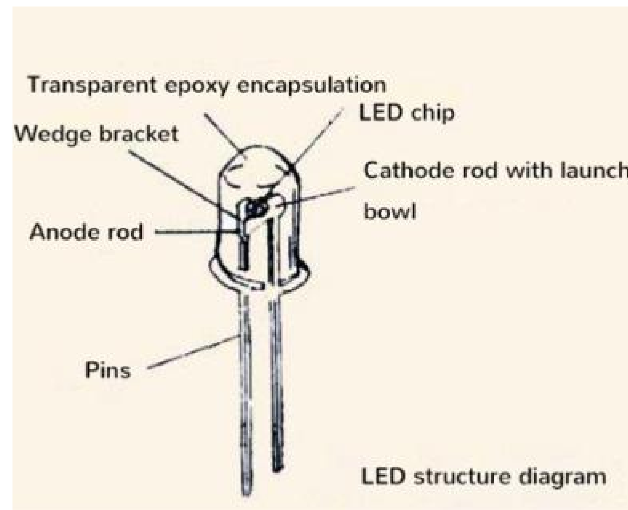
Introduction to LED

LED is a highly efficient and durable miniature semiconductor device that emits light when an electric current passes through it. It has the advantages of high energy conversion efficiency, low heat generation, and environmental friendliness. They are commonly used in indicator lights, display screens, and lighting equipment. LEDs have fast response times and a wide range of color options, making them widely used in electronic products. In the ESP32-P4 lighting demonstration, GPIO control simplifies and makes it intuitive to switch the LEDs, helping users better understand their practical applications.

① The principle of LED light emission

LED devices are light-emitting components based on solid-state semiconductor technology. When a forward current is applied to a semiconductor material with a PN junction, the recombination of charge carriers within the semiconductor releases energy in the form of photons, thereby generating light. Therefore, LEDs are cold light sources, unlike lighting based on filament, which generates heat and thus avoids

problems such as burning out. The following chart illustrates the operating principle of LED devices.



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

1.Firstly, carriers are injected under forward bias;

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

3.Finally, the energy released during the recombination process is radiated outward in the form of light.

In summary, when current passes through the PN junction, electrons are driven to the P region by the electric field. There, they combine with holes, releasing excess energy and generating photons, thereby achieving the light-emitting function of the PN junction.

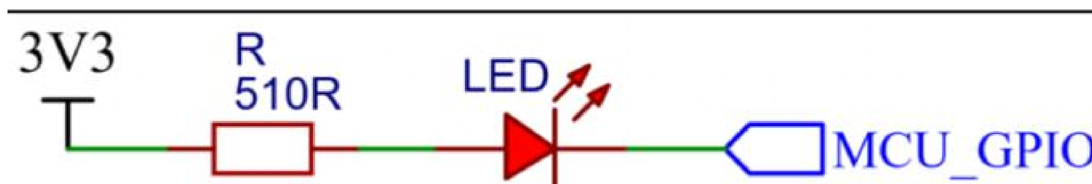
Note: The color of the light emitted by an LED is determined by the band gap width of the semiconductor material used. Different materials will produce light of different wavelengths, thus being able to generate light output of various colors. This efficient light-emitting mechanism has made light-emitting diodes widely adopted in lighting and indication applications.

② Principle of LED Lighting Driver

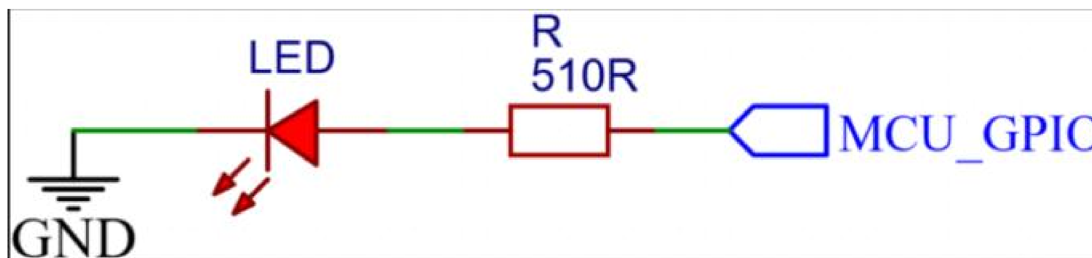
LED driving refers to providing appropriate current and voltage to LEDs through a stable power supply to ensure their normal lighting. The main driving methods for LEDs are constant current driving and constant voltage driving, among which constant current driving is more favored as it can limit the current. Due to the fact that LED lights are very sensitive to current fluctuations, exceeding their rated current may cause damage. Therefore, constant current driving ensures the operation of LEDs by maintaining a stable current flow. Next, we will study these two LED driving methods.

1) Current injection connection. This refers to the working current of the LED being provided externally, and the current is injected into our microcontroller.

The risk here is that the fluctuations of the external power supply can easily cause the microcontroller pins to burn out.



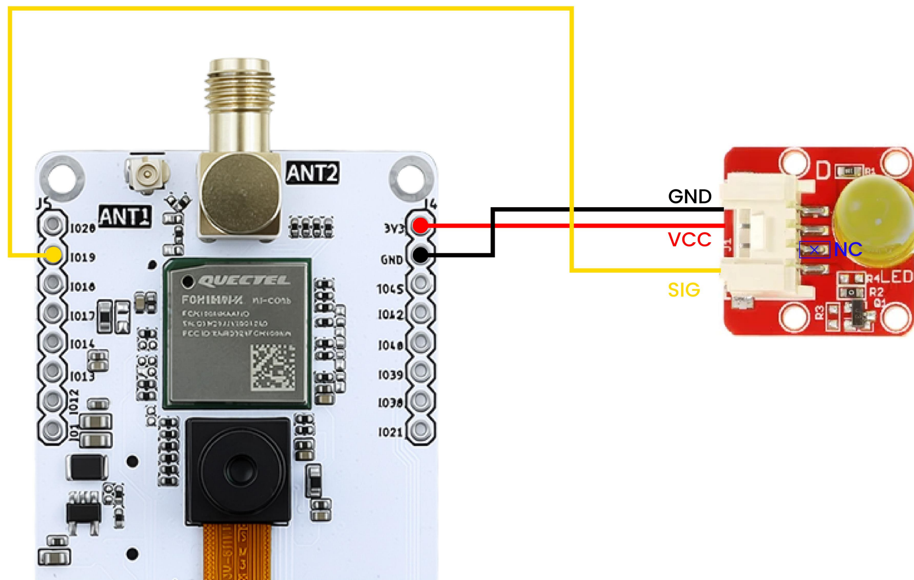
2) Power current configuration. This refers to the voltage and current provided by the microcontroller, and the current output will be applied to the LED. If the LED is driven directly by the GPIO of the microcontroller, its driving capability is relatively weak and may not be able to provide sufficient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

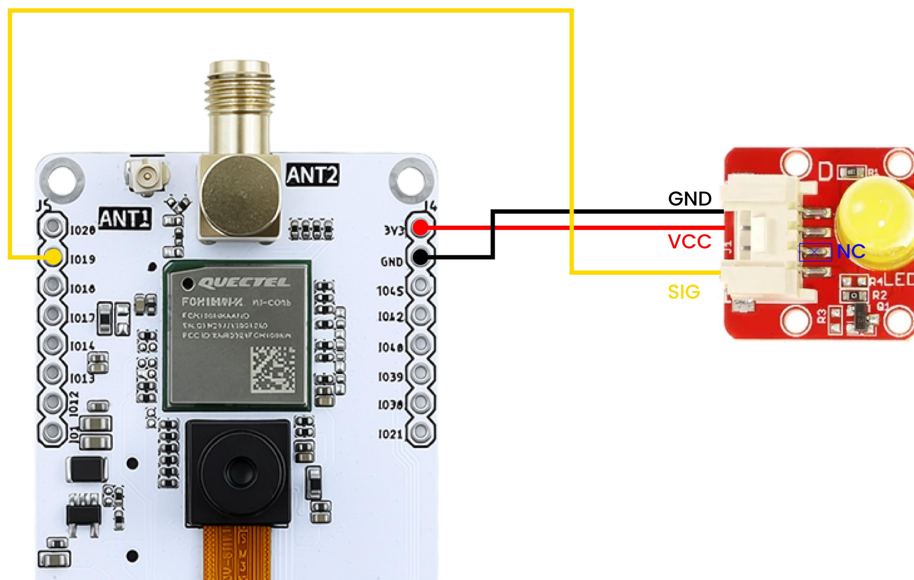


The LED circuit on the ESP32-P4 development board adopts the "current receiving" configuration. This approach avoids the microcontroller directly powering and supplying current to the LED, thereby effectively reducing the load on the microcontroller. This enables the microcontroller to focus more on performing other core tasks, thereby enhancing the performance and stability of the entire system.

Operation Effect Diagram

After running the code, you will see that the LED connected to GPIO19 will light up for one second, then go off for one second, and this process of flashing will repeat.





The LED used here is Elecrow's Crowtail-LED 2.0. You can visit the link below for purchase if needed.

Purchase Link:

https://www.elecrow.com/crowtail-led-p-1224.html

You can also use other common LED devices.

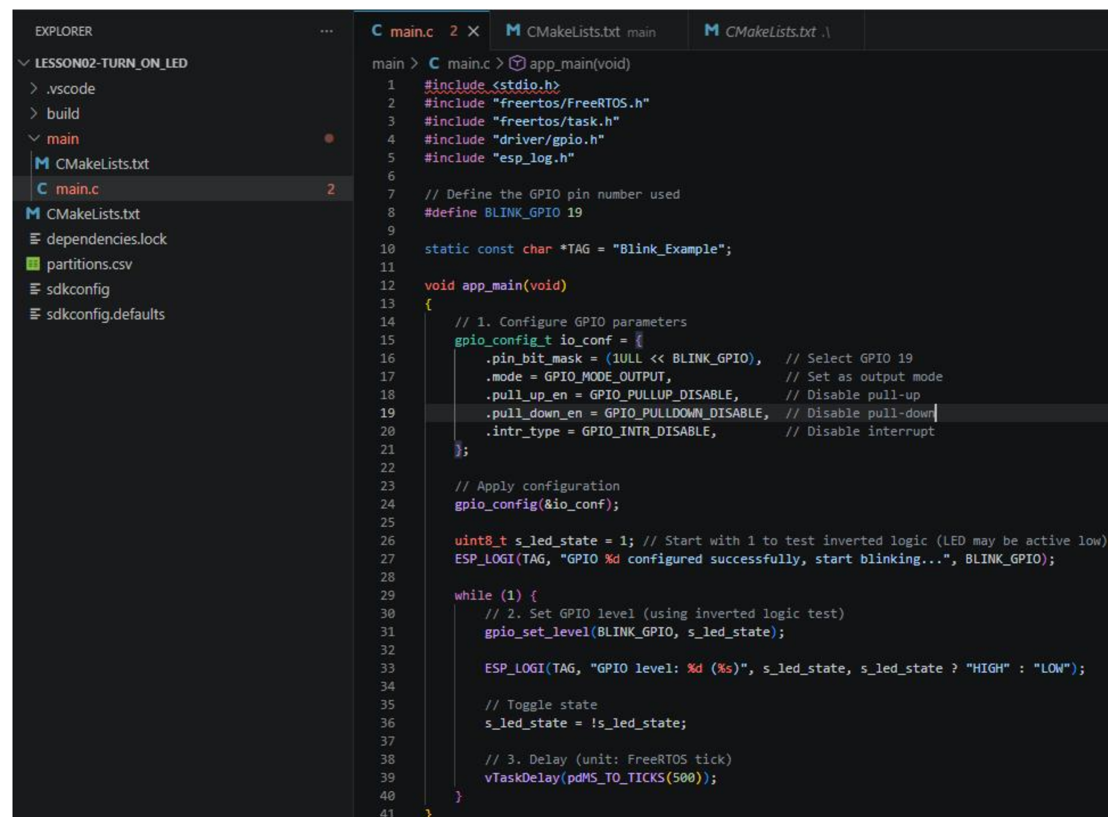
Connection Notes:

- Connect the VCC pin of ESP32 WiFi-HaLow module to the VCC pin of the LED module
- Connect the GND pin of ESP32 WiFi-HaLow module to the GND pin of the LED module
- Connect the "GPIO19" pin of ESP32 WiFi-HaLow module to the SIG pin of the LED module

Key Explanations

Next, we will analyze the control logic of the code in this lesson.
First, click the Github link below to download the supporting code of this lesson.
https://github.com/Elecrow-RD/ESP32_Wi-Fi_HaLow_Module_with_2MP_Camera_32Mbps_High_Speed/tree/master/example/V1.0/ESP-IDF_Code/Lesson02-Turn_On_LED

Open the downloaded code with VS-Code installed in the previous lesson.



Then let’s learn about the code architecture of this lesson.

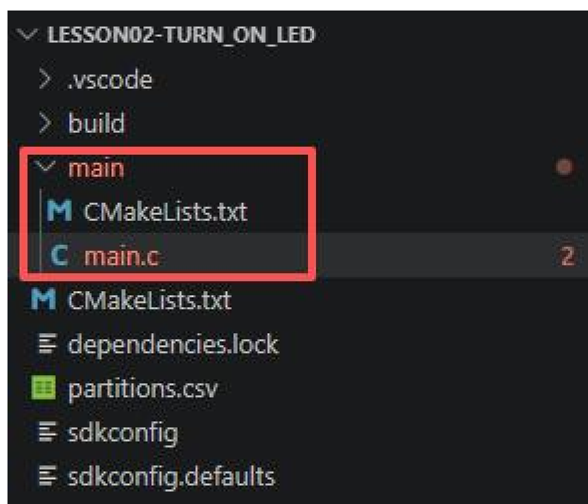
The CMakeLists.txt in the root directory is the overall project build script responsible for top-level configuration; files such as partitions.csv and sdkconfig are used for

partition table and chip parameter configuration.

The main subdirectory is the core application layer, containing its independent CMakeLists.txt and main.c. As the program entry file, main.c stores LED initialization, "GPIO" configuration and lighting control logic, serving as the core carrier of the experiment code. The whole project follows the modular build specification of "ESP-IDF", realizing separation of project configuration, build scripts and business logic code.

The code structure of this lesson is relatively simple, with the core logic concentrated in the main folder.

Let's take a look at the main contents in the **main** folder.



main.c is the standard code for realizing periodic high and low level flipping of "GPIO19" based on "FreeRTOS" on ESP32. Its core function is to make the LED connected to "GPIO19" blink at an interval of 500ms and output pin status logs through the serial port.

```c
main > C main.c > app_main(void)
1   #include <stdio.h>
2   #include "freertos/FreeRTOS.h"
3   #include "freertos/task.h"
4   #include "driver/gpio.h"
5   #include "esp_log.h"
6
7   // Define the GPIO pin number used
8   #define BLINK_GPIO 19
9
10  static const char *TAG = "Blink_Example";
11
12  void app_main(void)
13  {
14      // 1. Configure GPIO parameters
15      gpio_config_t io_conf = {
16          .pin_bit_mask = (1ULL << BLINK_GPIO),   // Select GPIO 19
17          .mode = GPIO_MODE_OUTPUT,               // Set as output mode
18          .pull_up_en = GPIO_PULLUP_DISABLE,      // Disable pull-up
19          .pull_down_en = GPIO_PULLDOWN_DISABLE,  // Disable pull-down
20          .intr_type = GPIO_INTR_DISABLE,         // Disable interrupt
21      };
22
23      // Apply configuration
24      gpio_config(&io_conf);
25
26      uint8_t s_led_state = 1; // Start with 1 to test inverted logic (LED may be active low)
27      ESP_LOGI(TAG, "GPIO %d configured successfully, start blinking...", BLINK_GPIO);
28
29      while (1) {
30          // 2. Set GPIO level (using inverted logic test)
31          gpio_set_level(BLINK_GPIO, s_led_state);
32
33          ESP_LOGI(TAG, "GPIO level: %d (%s)", s_led_state, s_led_state ? "HIGH" : "LOW");
34
35          // Toggle state
36          s_led_state = !s_led_state;
37
38          // 3. Delay (unit: FreeRTOS tick)
39          vTaskDelay(pdMS_TO_TICKS(500));
40      }
41  }
```

The header file introduction statements at the beginning of the program prepare for invoking basic functions and hardware drivers:

- stdio.h: Standard C input and output library, providing basic data processing support;
- freertos/FreeRTOS.h and freertos/task.h: Core header files of "FreeRTOS" real-time operating system, implementing task scheduling, delay and other system-level functions, which are core dependencies for ESP32 program operation;
- driver/gpio.h: Dedicated "GPIO" driver library for ESP32, enabling the program to configure and control chip pins;
- esp_log.h: ESP32 log printing library, used to output debugging information and status prompts to the serial port, facilitating developers to monitor program operation. Without these header files, the program cannot implement core functions such as pin control, delay and log output.

```
1  #include <stdio.h>
2  #include "freertos/FreeRTOS.h"
3  #include "freertos/task.h"
4  #include "driver/gpio.h"
5  #include "esp_log.h"
```

This part is the constant definition of the program. #define BLINK_GPIO 19 defines the LED-connected "GPIO19" pin as a macro constant. This method realizes unified management of pin numbers. If you need to replace the LED connection pin later, only this line needs to be modified without changing other parts of the program, improving code maintainability.

static const char *TAG = "Blink_Example" defines the tag for log output. As the prefix identifier of logs, TAG will be attached during serial port log printing, allowing developers to quickly distinguish log information of different modules and functions for convenient debugging and fault troubleshooting.

```
7   // Define the GPIO pin number used
8   #define BLINK_GPIO 19
9
10  static const char *TAG = "Blink_Example";
```

app_main() is the unique program entry function of ESP32, equivalent to the main() function in standard C language. After the chip is powered on and started, all codes inside this function will be executed preferentially. All pin initialization, LED control logic and cyclic blinking functions are implemented inside this function, which is the core execution carrier of the entire program.

```
12  void app_main(void)
13  {
14      // 1. Configure GPIO parameters
15      gpio_config_t io_conf = {
16          .pin_bit_mask = (1ULL << BLINK_GPIO),     // Select GPIO 19
17          .mode = GPIO_MODE_OUTPUT,                  // Set as output mode
18          .pull_up_en = GPIO_PULLUP_DISABLE,         // Disable pull-up
19          .pull_down_en = GPIO_PULLDOWN_DISABLE,     // Disable pull-down
20          .intr_type = GPIO_INTR_DISABLE,            // Disable interrupt
21      };
22
23      // Apply configuration
24      gpio_config(&io_conf);
25
26      uint8_t s_led_state = 1; // Start with 1 to test inverted logic (LED may be active low)
27      ESP_LOGI(TAG, "GPIO %d configured successfully, start blinking...", BLINK_GPIO);
28
29      while (1) {
30          // 2. Set GPIO level (using inverted logic test)
31          gpio_set_level(BLINK_GPIO, s_led_state);
32
33          ESP_LOGI(TAG, "GPIO level: %d (%s)", s_led_state, s_led_state ? "HIGH" : "LOW");
34
35          // Toggle state
36          s_led_state = !s_led_state;
37
38          // 3. Delay (unit: FreeRTOS tick)
39          vTaskDelay(pdMS_TO_TICKS(500));
40      }
41  }
```

This part is the core of hardware initialization. The gpio_config_t structure is used to configure all parameters of the "GPIO19" pin:

- pin_bit_mask: Precisely selects the "GPIO19" pin;
- mode: Sets the pin to output mode to actively output high and low levels for LED control;
- pull_up_en & pull_down_en: Disable internal pull-up and pull-down resistors respectively, as the LED hardware circuit does not require auxiliary support from internal resistors;
- intr_type: Disable pin interrupt function to meet the simple blinking requirement of the LED.

Finally, call gpio_config(&io_conf) to write the configuration parameters into the chip hardware, complete the initialization of "GPIO19" pin, and prepare for subsequent LED control.

```
// 1. Configure GPIO parameters
gpio_config_t io_conf = {
    .pin_bit_mask = (1ULL << BLINK_GPIO),    // Select GPIO 19
    .mode = GPIO_MODE_OUTPUT,                // Set as output mode
    .pull_up_en = GPIO_PULLUP_DISABLE,       // Disable pull-up
    .pull_down_en = GPIO_PULLDOWN_DISABLE,   // Disable pull-down
    .intr_type = GPIO_INTR_DISABLE,          // Disable interrupt
};

// Apply configuration
gpio_config(&io_conf);
```

This part completes status initialization and startup prompt before program operation. uint8_t s_led_state = 1 defines a variable to store the LED level state with an initial value of 1, adapting to the hardware logic that most development board LEDs light up at low level, facilitating the test of pin level flipping effect.

Meanwhile, the ESP_LOGI log function prints prompt information to inform developers that the "GPIO19" pin is configured successfully and the LED blinking process has started, which can intuitively confirm the normal completion of the initialization process.

```
// Apply configuration
gpio_config(&io_conf);

uint8_t s_led_state = 1; // Start with 1 to test inverted logic (LED may be active low)
ESP_LOGI(TAG, "GPIO %d configured successfully, start blinking...", BLINK_GPIO);
```

This is the core code to realize continuous LED blinking. while(1) creates an infinite loop to ensure continuous program operation.

Inside the loop:

1.Call gpio_set_level to output the level state stored in the variable to "GPIO19" pin, directly controlling the on and off of the LED;

2.Print the current pin level value and state through the log function to monitor the real-time working status of the pin;

3.Execute s_led_state = !s_led_state to invert the level state, realizing switching between high level and low level, which is the key logic for LED blinking;

4.Call vTaskDelay(pdMS_TO_TICKS(500)) to achieve a 500-millisecond delay, keeping the LED in the current state for a fixed period. After the delay, the program re-enters the loop to repeat level output, log printing, state inversion and delay execution, finally realizing the LED blinking effect at an interval of 500 milliseconds.

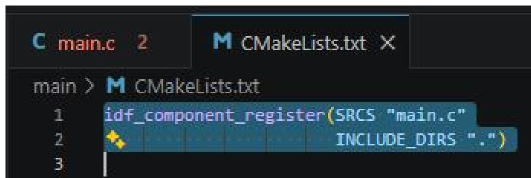
```
while (1) {
    // 2. Set GPIO level (using inverted logic test)
    gpio_set_level(BLINK_GPIO, s_led_state);

    ESP_LOGI(TAG, "GPIO level: %d (%s)", s_led_state, s_led_state ? "HIGH" : "LOW");

    // Toggle state
    s_led_state = !s_led_state;

    // 3. Delay (unit: FreeRTOS tick)
    vTaskDelay(pdMS_TO_TICKS(500));
}
```

And then there is the **CMakeLists.txt** file under the **main** folder.



The CMakeLists.txt script in the main folder is the core configuration instruction for registering the main program component in the "ESP-IDF" framework.

idf_component_register is a key function recognized by the build system:

- SRCS "main.c": Informs the compiler that the source file to be compiled is main.c in the current directory;
- INCLUDE_DIRS ".": Specifies the header file search path as the current directory, enabling the compiler to correctly locate referenced header files in the code.

The entire configuration simply completes the registration of the main program component, associates business code with the "ESP-IDF" build system, and ensures accurate compilation, linking and firmware generation executable on ESP32 during project building.

Complete Code

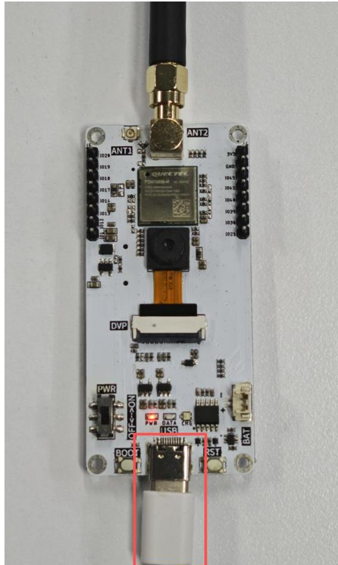
Kindly click the link below to view the full code implementation.

https://github.com/Elecrow-RD/ESP32_Wi-Fi_HaLow_Module_with_2MP_Camera_32Mbps_High_Speed/tree/master/example/V1.0/ESP-IDF_Code/Lesson02-Turn_On_LED

Programming Steps

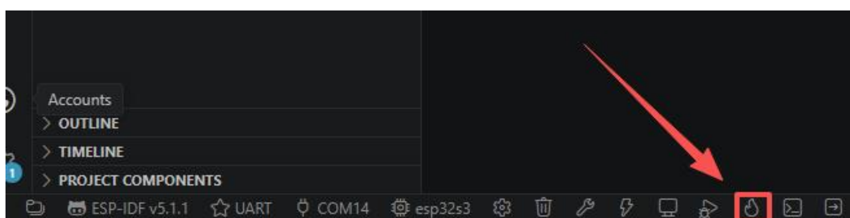
The code explanation is completed above. Next, we will upload the code to the development board.

First, connect the ESP32 WiFi-HaLow module to the computer host via a USB cable.



Follow the steps in Lesson 1 to select the corresponding "ESP-IDF" version V5.1.1, "UART" interface, the used serial port number, and the main control chip ESP32S3 of the ESP32 WiFi-HaLow module.

Click the integrated button for compilation, uploading and opening the serial port monitor to complete all operations with one click.



After waiting for a moment, you can see the LED connected to "GPIO19" start blinking.

The main purpose of this lesson is to guide you to master the basic use of "GPIO" pins on the ESP32 WiFi-HaLow module. After class, you can use other pins and functions to realize different application functions by referring to the provided schematic diagram.

Lesson03---Button_Control_LED

Introduction

In this lesson, we will learn how to use the GPIO pins of the ESP32 WiFi-HaLow module, focusing on solving the practical wiring issue of simultaneously driving two GPIO peripherals (an LED and a button) when the development board only provides one set of VCC and GND interfaces.

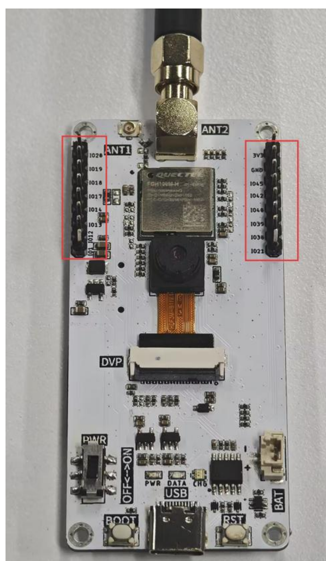
At this point, we need to use a breadboard, as it provides you with a safe, quick-plug, and flexible solderless connection platform. It can not only stably connect the GPIO19, GPIO20, and GND pins of the development board together with scattered components such as LEDs and buttons into the same circuit, but also avoid issues such as poor pin contact, wire short circuits, and unstable component fixation that are likely to occur when Dupont wires are directly connected. In addition, it is extremely convenient for later wiring modifications and component replacements.

Since the ESP32 WiFi-HaLow module only provides one set of 3V3 (VCC) and GND pins, when simultaneously driving the Crowtail LED and Crowtail Button, the power terminals of both devices need to be connected in parallel to share the same VCC and GND, providing unified power supply for them to ensure that both peripherals can obtain stable power and work properly.

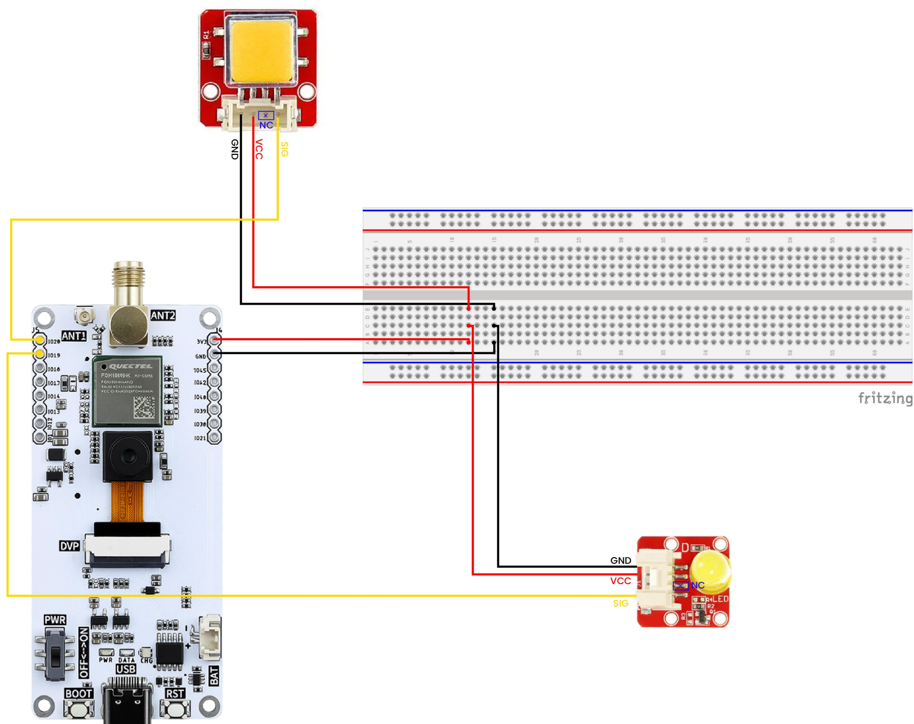
(The breadboard is not included and needs to be provided by yourself.)

Hardware Used in This Lesson

In this lesson, we will use pins 19 and 20 located at the upper left corner of the ESP32 WiFi-HaLow module.
Pin 19 is still connected to the LED from the previous lesson, while pin 20 is connected to the button.



You can refer to the following connection method:



Wiring Method:
First, connect the 3V3 (VCC) and GND pins of the ESP32 WiFi-HaLow module to the breadboard to establish a common power rail;
Next, connect the VCC and GND pins of the Crowtail LED and Crowtail Button respectively in parallel to the corresponding power rails on the breadboard to achieve shared power supply for both devices;
Finally, connect the SIG pin of the Crowtail LED to pin 19 of the ESP32 WiFi-HaLow module, and connect the SIG pin of the Crowtail Button to pin 20 of the ESP32 WiFi-HaLow module.

The LED used here is Elecrow's Crowtail-LED 2.0, and the button is also Elecrow's Crowtail-Button 2.0. If you would like to purchase them, you can visit the following links.

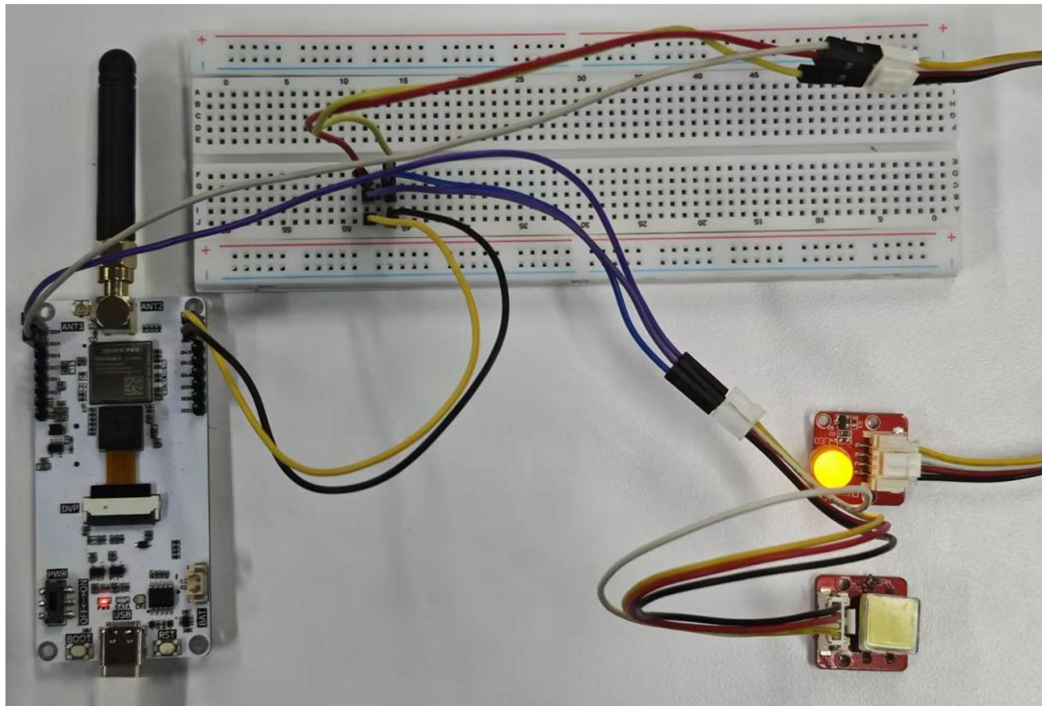
Purchase Links:

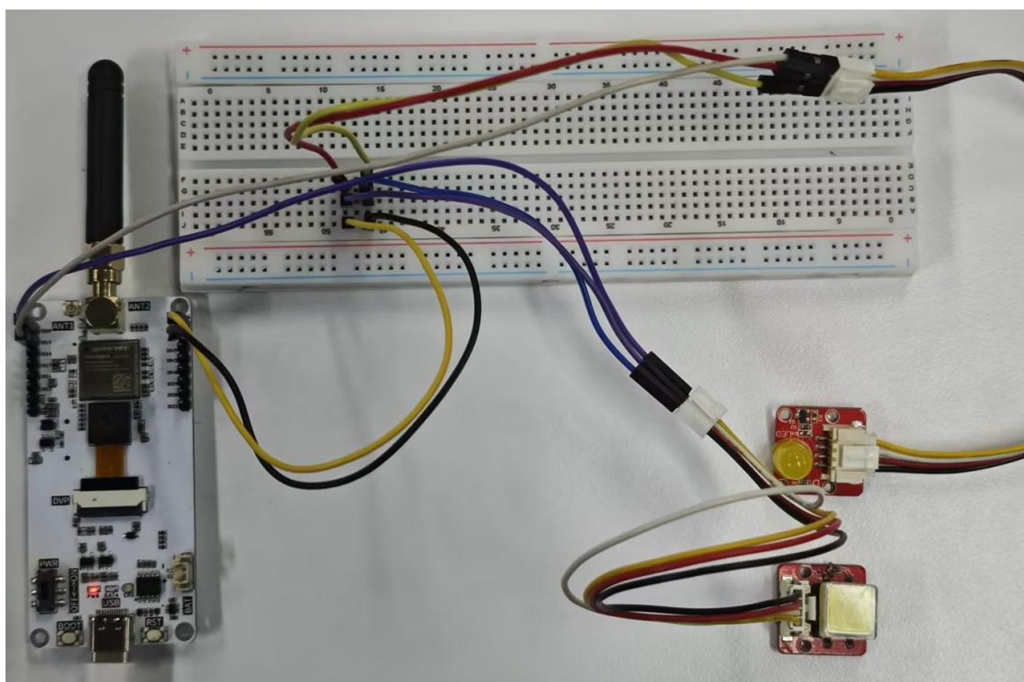
https://www.elecrow.com/crowtail-led-p-1224.html

https://www.elecrow.com/crowtailbutton-p-122.html

Operation Effect Diagram

After running the code, when you press the button, the LED will light up; and if you press the button again, the LED will go off.

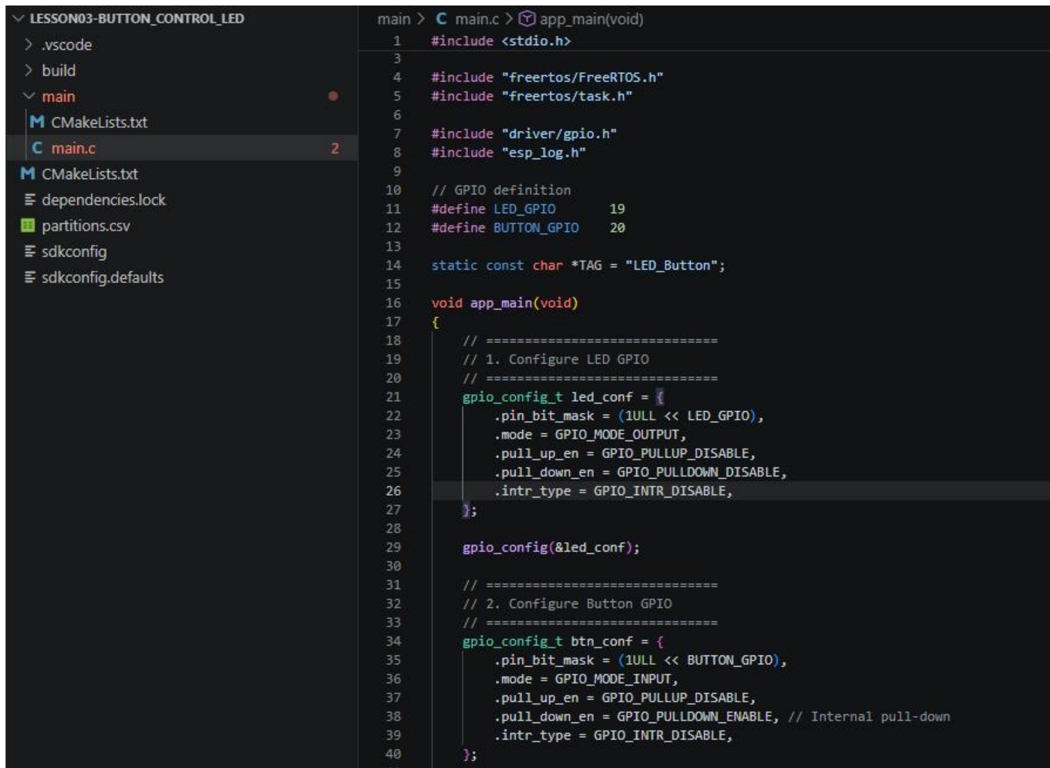




Key Explanations

Next, let us take a look at the control logic used in the code of this lesson.
Let us explore it together with this question in mind.
First, click the GitHub link below to download the code for this lesson.
https://github.com/Elecrow-RD/ESP32_Wi-Fi_HaLow_Module_with_2MP_Camera_32Mbps_High_Speed/tree/master/example/V1.0/ESP-IDF_Code/Lesson03-Button_Control_LED

Then, use the previously installed VS Code to open the code for this lesson.

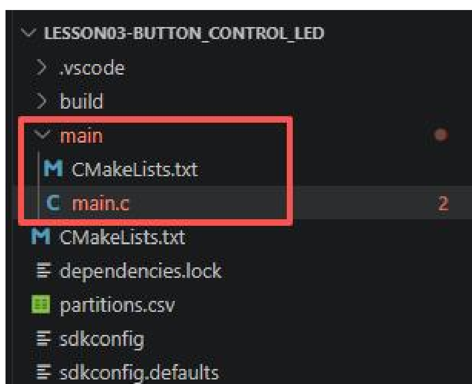


Next, let us take a look at the code architecture of this lesson together.

The CMakeLists.txt file in the root directory is the overall project build script responsible for top-level configuration; files such as partitions.csv and sdkconfig are used for partition tables and chip configuration;

The main subdirectory is the core application layer, containing its own independent CMakeLists.txt and main.c. The main.c file serves as the program entry point and stores the LED initialization, GPIO configuration, and LED control logic. It is the core code carrier of this experiment. The overall structure follows the modular build specifications of ESP-IDF, achieving separation between project configuration, build scripts, and business logic code.

Let us take a look at the main contents in the main folder.



First, look at main.c. This is a standard button-controlled LED switching program under the ESP-IDF framework: press the button → toggle the LED state (switch between ON and OFF), with button debounce and stable detection. It is the most classic practical program for GPIO input and output.

main > C main.c > app_main(void)

```
#include <stdio.h>
#include <stdbool.h>

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#include "driver/gpio.h"
#include "esp_log.h"

// GPIO definition
#define LED_GPIO       19
#define BUTTON_GPIO    20

static const char *TAG = "LED_Button";

void app_main(void)
{
    // ==============================
    // 1. Configure LED GPIO
    // ==============================
    gpio_config_t led_conf = {
        .pin_bit_mask = (1ULL << LED_GPIO),
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE,
    };

    gpio_config(&led_conf);

    // ==============================
    // 2. Configure Button GPIO
    // ==============================
    gpio_config_t btn_conf = {
        .pin_bit_mask = (1ULL << BUTTON_GPIO),
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_ENABLE, // Internal pull-down
        .intr_type = GPIO_INTR_DISABLE,
    };

    gpio_config(&btn_conf);

    ESP_LOGI(TAG, "System initialized");
```

The header file include statements at the beginning of the code provide the necessary functional support for the entire program, where stdio.h provides basic input and output functions, and stdbool.h is used to support Boolean variables;

freertos/FreeRTOS.h and freertos/task.h are the core files of the FreeRTOS real-time operating system, providing system functions such as task scheduling and delays; driver/gpio.h is the GPIO driver library of the ESP32, used for configuring and controlling pin levels;

esp_log.h is used for serial log output, making it convenient for developers to view the program running status. These header files together form the basic operating environment of the program, and none of them are dispensable.

```
1   #include <stdio.h>
2   #include <stdbool.h>
3
4   #include "freertos/FreeRTOS.h"
5   #include "freertos/task.h"
6
7   #include "driver/gpio.h"
8   #include "esp_log.h"
```

This part of the code specifies the LED control pin as GPIO19 and the button detection pin as GPIO20 through macro definitions. Using macro definitions makes subsequent pin modifications more convenient, as only one place needs to be changed for the changes to take effect globally;

At the same time, the log tag LED_Button is defined, which will automatically be included in serial output logs, helping developers quickly identify the source of logs and improving the efficiency of program debugging and troubleshooting.

```
10  // GPIO definition
11  #define LED_GPIO       19
12  #define BUTTON_GPIO    20
13
14  static const char *TAG = "LED_Button";
```

This part of the code performs complete GPIO initialization configuration for GPIO19. Through a structure, the pin is set to output mode, the internal pull-up, pull-down resistors, and interrupt functions are disabled, and the pin is explicitly designated for outputting high and low levels to control the LED ON/OFF state.

After the configuration is completed, the gpio_config function is called to write the parameters into the hardware, making GPIO19 a stable and controllable output pin and preparing it for driving the LED.

```
16  void app_main(void)
17  {
18      // ==============================
19      // 1. Configure LED GPIO
20      // ==============================
21      gpio_config_t led_conf = {
22          .pin_bit_mask = (1ULL << LED_GPIO),
23          .mode = GPIO_MODE_OUTPUT,
24          .pull_up_en = GPIO_PULLUP_DISABLE,
25          .pull_down_en = GPIO_PULLDOWN_DISABLE,
26          .intr_type = GPIO_INTR_DISABLE,
27      };
28
29      gpio_config(&led_conf);
```

This part of the code configures GPIO20 as an input mode, enables the internal pull-down resistor, and disables the pull-up resistor and interrupt function. In this way, when the button is not pressed, the pin remains at a stable low level, and when pressed, it changes to a high level, effectively avoiding false triggering caused by floating levels and making button detection more stable and reliable.

```
34      gpio_config_t btn_conf = {
35          .pin_bit_mask = (1ULL << BUTTON_GPIO),
36          .mode = GPIO_MODE_INPUT,
37          .pull_up_en = GPIO_PULLUP_DISABLE,
38          .pull_down_en = GPIO_PULLDOWN_ENABLE, // Internal pull-down
39          .intr_type = GPIO_INTR_DISABLE,
40      };
41
42      gpio_config(&btn_conf);
```

After completing the GPIO configuration, the program outputs an initialization complete log and defines a Boolean variable led_state to record the current ON/OFF status of the LED, with the initial value set to OFF. This variable will continue to update when the button is triggered, thereby controlling the LED to switch between ON and OFF.

```
46      // LED state
47      bool led_state = false;
48
```

The program enters an infinite loop to ensure continuous operation. In the loop, it continuously reads the level status of GPIO20 to monitor whether the button is pressed in real time. This loop detection method is the most commonly used button scanning method in embedded development and can respond to external operations promptly.

```
49      while (1)
50      {
51          // Read button state
52          int button_level = gpio_get_level(BUTTON_GPIO);
```

When the button level is detected as high, the program first performs a 20-millisecond debounce delay. Mechanical buttons generate level bouncing at the moment they are pressed, and direct detection can easily cause multiple false triggers. After the delay, the button level is checked again, and only if it remains high will it be determined as a valid press, ensuring the accuracy of button detection.

```
54          // Detect button press
55          if (button_level == 1)
56          {
57              // ==============================
58              // Debounce
59              // ==============================
60              vTaskDelay(pdMS_TO_TICKS(20));
61
62              // Confirm button still pressed
63              if (gpio_get_level(BUTTON_GPIO) == 1)
64              {
```

After confirming a valid button press, the program inverts the LED state to achieve ON/OFF switching, and outputs the new state to GPIO19 through the gpio_set_level function to control the hardware LED accordingly. At the same time, the current LED status is printed through serial logs so that developers can visually observe the execution effect of the program.

```
62            // Confirm button still pressed
63            if (gpio_get_level(BUTTON_GPIO) == 1)
64            {
65                // Toggle LED state
66                led_state = !led_state;
67
68                // Set LED output
69                gpio_set_level(LED_GPIO, led_state);
70
71                ESP_LOGI(TAG,
72                         "LED %s",
73                         led_state ? "ON" : "OFF");
```

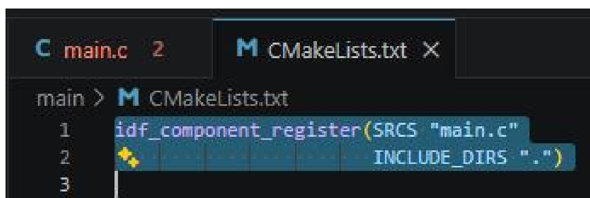
To avoid multiple triggers from a single button press, the program continuously waits for the button to be released. It only exits the waiting state after detecting the level has returned to low, and then performs another 20-millisecond release debounce delay, further ensuring that the button action is executed only once and improving system stability.

```
75                // Wait for button release
76                while (gpio_get_level(BUTTON_GPIO) == 1)
77                {
78                    vTaskDelay(pdMS_TO_TICKS(10));
79                }
80
81                // Release debounce
82                vTaskDelay(pdMS_TO_TICKS(20));
83            }
84        }
```

A short 10-millisecond delay is added at the end of the main loop, which can effectively reduce CPU usage and prevent system resources from being excessively consumed by the infinite loop, making the entire program run more smoothly and stably without affecting the button response speed.

```
85
86            vTaskDelay(pdMS_TO_TICKS(10));
87        }
88    }
```

Next is the CMakeLists.txt file under the main folder.



This CMakeLists.txt script located in the main folder is the core configuration instruction used to register the main program component in the ESP-IDF framework.

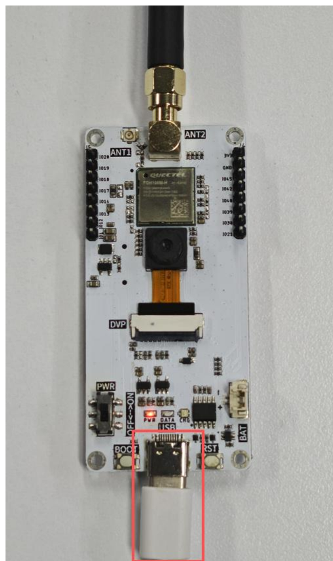
idf_component_register is the key function for the build system to identify components, where SRCS "main.c" explicitly tells the compiler that the source file to be compiled is main.c in the current directory,and INCLUDE_DIRS "." specifies the search path for header files as the current directory, allowing the compiler to correctly locate the header files referenced in the code.
The entire configuration concisely completes the registration of the main program component, linking the business logic code with the ESP-IDF build system to ensure that the project can accurately compile and link the main program file during compilation, generating firmware that can run on the ESP32.

Complete Code

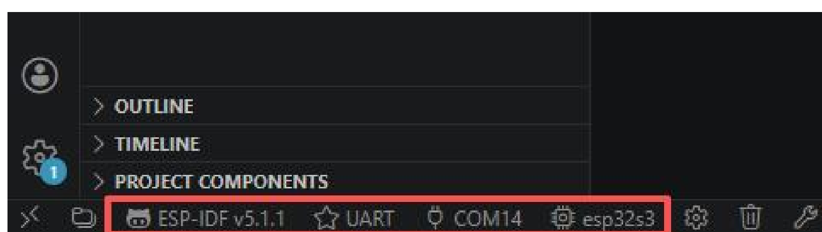
Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/ESP32_Wi-Fi_HaLow_Module_with_2MP_Camera_32Mbps_High_Speed/tree/master/example/V1.0/ESP-IDF_Code/Lesson03-Button_Control_LED

Programming Steps

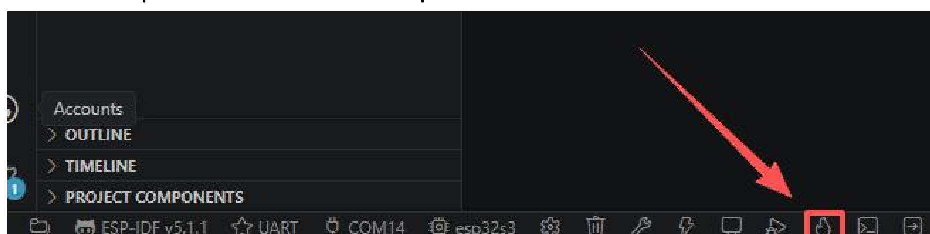
At this point, the code explanation is complete. Next, let us upload the code.
First, connect the ESP32 WiFi-HaLow module to the computer host using a USB cable.



Here, following the steps in the first lesson, first select the ESP-IDF version V5.1.1, UART, the serial port number you are using, and the main control chip ESP32S3 of the ESP32 WiFi-HaLow module.



Then click the button for compiling, uploading, and opening the serial monitor, and all these operations can be completed with one click.



After waiting for a moment, you will be able to use the button to control the LED ON and OFF.

Lesson 04---DHT20

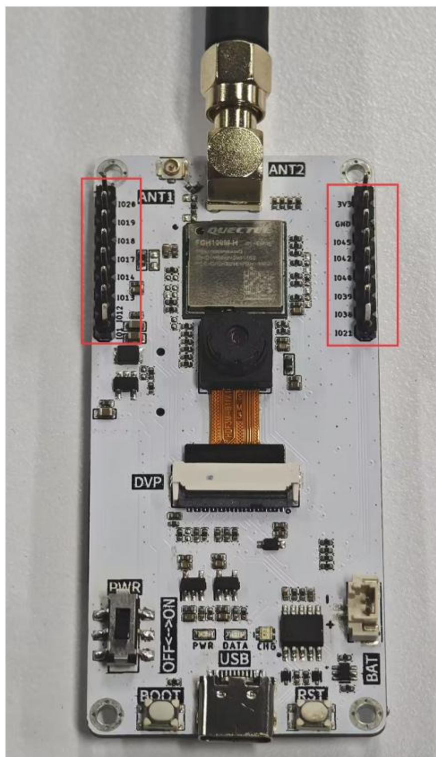
Introduction

In this lesson, we will learn how to use the I2C communication interface of the ESP32 WiFi-HaLow module, focusing on mastering the wiring and programming logic for connecting an I2C protocol peripheral (DHT20 temperature and humidity sensor) via the clock line (SCL) and data line (SDA).

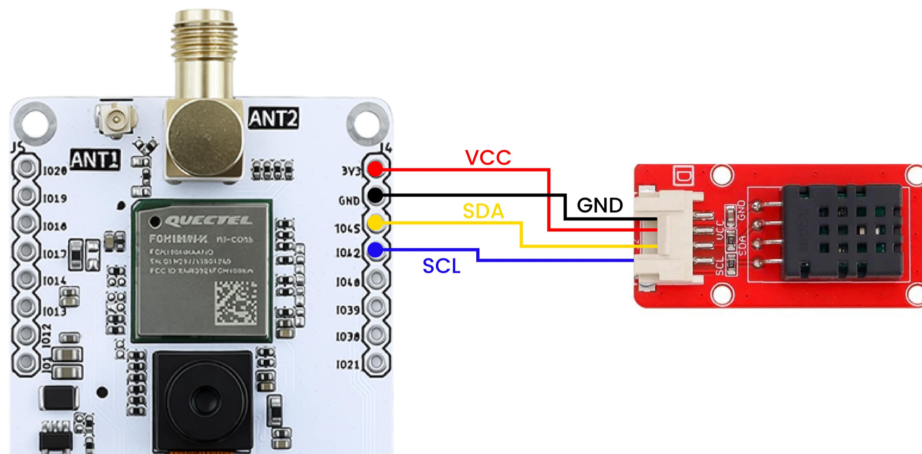
I2C is the most commonly used serial communication protocol in embedded development. It only requires two signal lines (clock + data) to achieve bidirectional communication between the host and peripheral devices, which greatly saves chip pin resources compared to traditional GPIO point-to-point control.
The DHT20 is a standard I2C interface temperature and humidity sensor. We will read the sensor data through the I2C pin of the ESP32 and print it out via the serial port.

Hardware Used in This Lesson

In this lesson, we will use pins 42 and 45 in the upper right corner of the ESP32 WiFi-HaLow module. As shown in the schematic, pin 42 is the I2C clock line of the ESP32 WiFi-HaLow module , and pin 45 is the I2C data line .



You can refer to the following connection methods:



Connection method:

In this lesson, we will use the dedicated I2C pin of the ESP32 WiFi-HaLow module to connect the DHT20 temperature and humidity sensor:
GPIO42: I2C Clock Line (SCL, Serial Clock Line)
GPIO45: I2C Data Line (SDA, Serial Data Line)
Module 3V3 (VCC) / GND: Powers the sensor
Peripherals: DHT20 I2C temperature and humidity sensor
in:
The ESP32 WiFi-HaLow module's pin 42 I2C clock line is connected to the DHT20 module's SCL pin;
The ESP32 WiFi-HaLow module's pin 45 I2C data line connects to the DHT20 module's SDA pin;
The VCC pin of the ESP32 WiFi-HaLow module is connected to the VCC pin of the DHT20 module;
The VCC pin of the ESP32 WiFi-HaLow module is connected to the VCC pin of the DHT20 module;

This DHT20 temperature and humidity sensor is the Elecrow Crowtail-DHT20 Temperature and Humidity Sensor. You can purchase it via the following link.

Purchase link:

https://www.elecrow.com/crowtail-dht20.html

Operation Effect Diagram

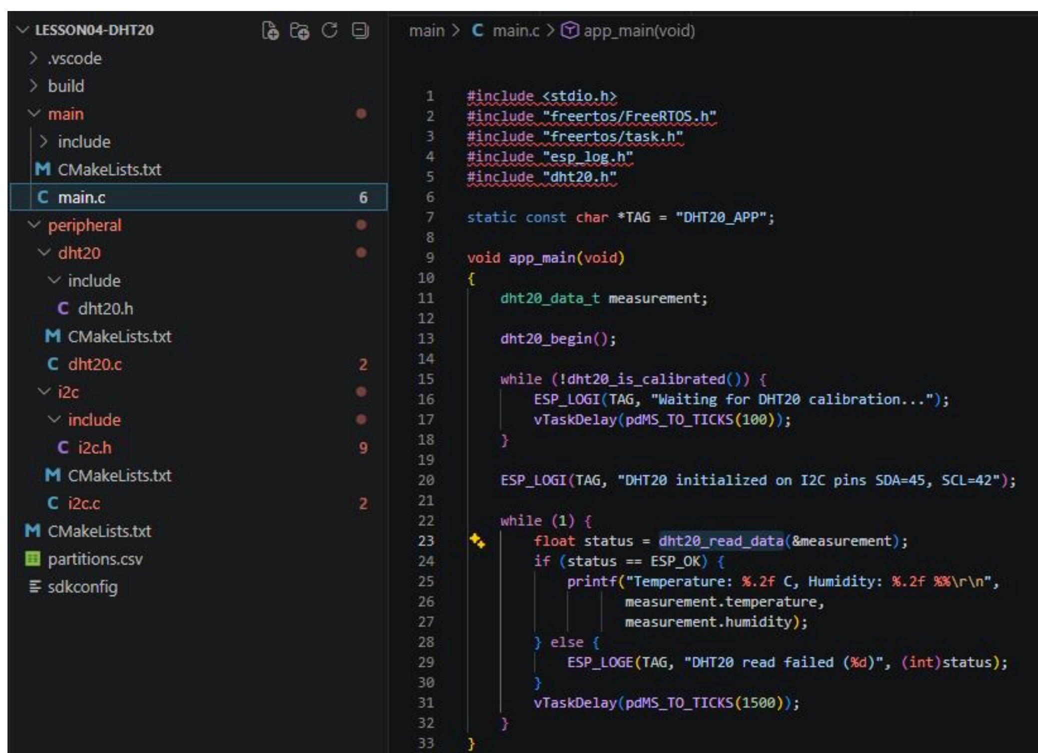
After the code is burned and run, the ESP32 will read the temperature and humidity data of the DHT20 every 1.5 seconds via the I2C bus and print it in real time via the serial monitor.

```
I (291) sleep: Configure to isolate all GPIO pins in sleep state
I (298) sleep: Enable automatic switching of GPIO sleep configuration
I (305) app_start: Starting scheduler on CPU0
I (310) app_start: Starting scheduler on CPU1
I (310) main_task: Started on CPU0
I (320) main_task: Calling app_main()
I (320) I2C: Initializing I2C Bus.......
I (330) I2C: I2C Bus Initialized, ESP_OK
I (740) DHT20_APP: DHT20 initialized on I2C pins SDA=45, SCL=42
Temperature: 26.57 C, Humidity: 62.95 %
Temperature: 26.59 C, Humidity: 63.05 %
Temperature: 26.57 C, Humidity: 63.01 %
Temperature: 26.58 C, Humidity: 62.95 %
Temperature: 26.58 C, Humidity: 62.88 %
Temperature: 26.57 C, Humidity: 62.83 %
Temperature: 26.57 C, Humidity: 62.82 %
```

Key Explanations

Next, let's take a look at the control logic in the code of this lesson.
Let's explore this question together.
First, click the Github link below to download the code for this lesson.
https://github.com/Elecrow-RD/ESP32_Wi-Fi_HaLow_Module_with_2MP_Camera_32Mbps_High_Speed/tree/master/example/V1.0/ESP-IDF_Code/Lesson04-DHT20

Then open the code from this lesson using the installed VS-Code.



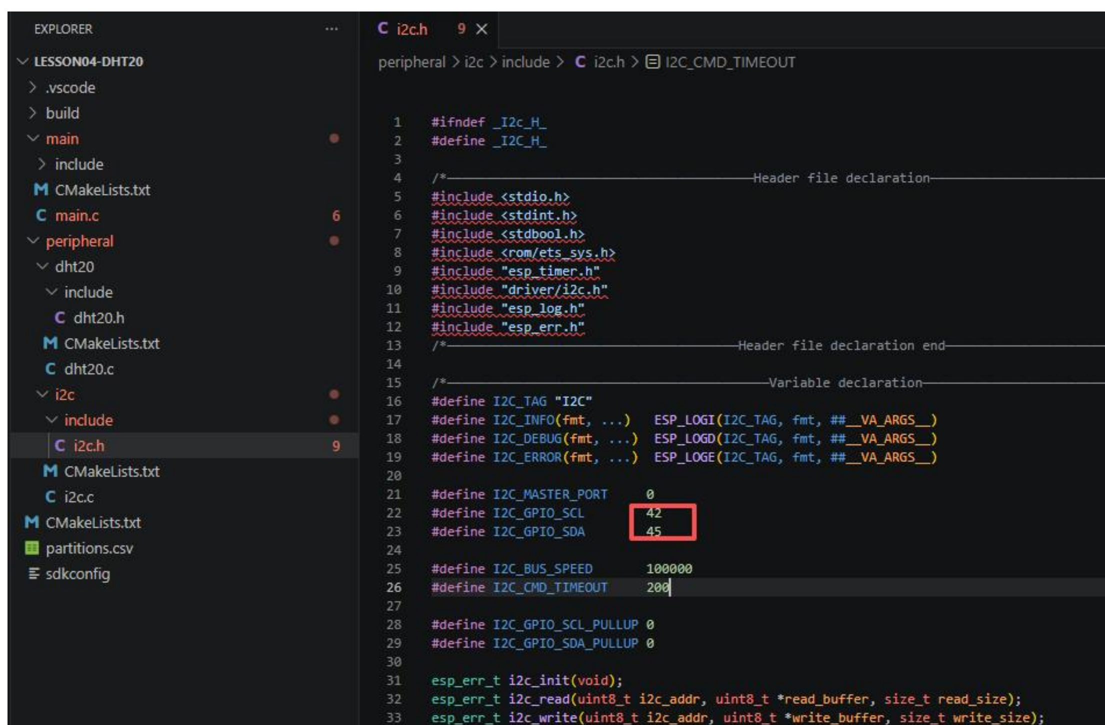
Next, let's take a look at the code architecture of this lesson.

The ESP-IDF project in this lesson uses a modular, layered architecture, achieving a clear decoupling between business logic and underlying drivers:
The root directory contains project-level build configuration files (" CMakeLists.txt " , " partitions.csv " , and " sdkconfig "), which are responsible for the overall compilation rules and chip partition configuration;

" main " directory is the application entry point, where " main.c " contains the main program logic and is also registered as a main component via " CMakeLists.txt " ;

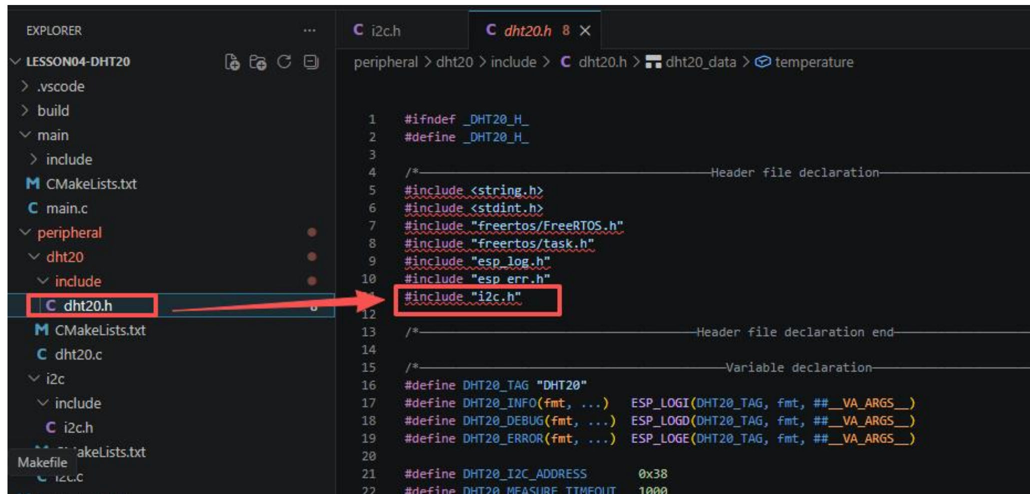
The underlying drivers related to peripherals are uniformly placed in the " peripheral " directory. The " i2c " module encapsulates the initialization and communication operations of the I2C bus, while the " dht20 " module implements the protocol parsing and data reading of the temperature and humidity sensor based on the general I2C interface. Each sub-module contains independent header files, source files, and build scripts, forming a reusable and easily maintainable componentized structure.

Let's first look at the two components in the peripheral: dht20 and i2c.
If you want to use I2C-related devices, you only need to use the I2C component we provide, and then modify the corresponding I2C pins in i2c.h. Of course, you can also add your own code here. The I2C component is the most basic I2C driver code we provide.

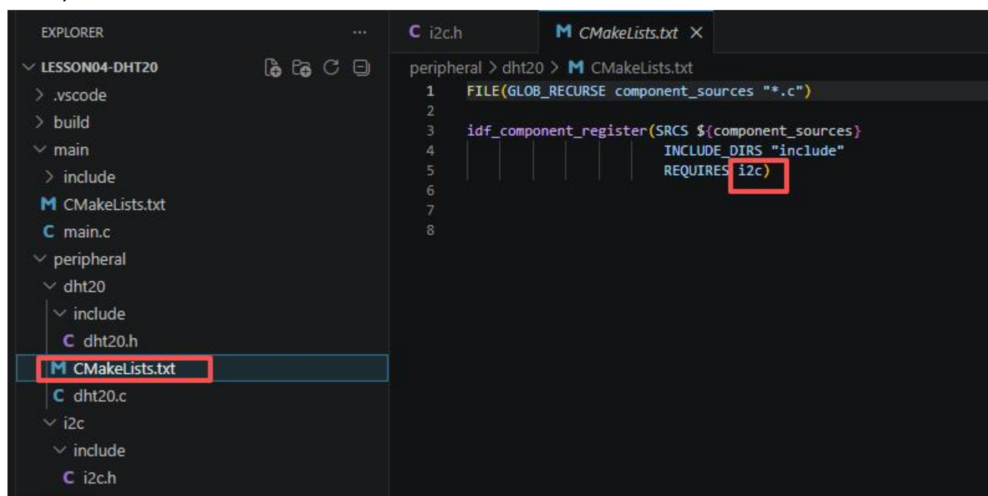


For example, our DHT20 sensor is an I2C device, and we need to use I2C to drive the device.

Therefore, in the dht20.h file of the dht20 component, we referenced the i2c.h file, so that we can use the program interface in the i2c.c code to drive your I2C device.

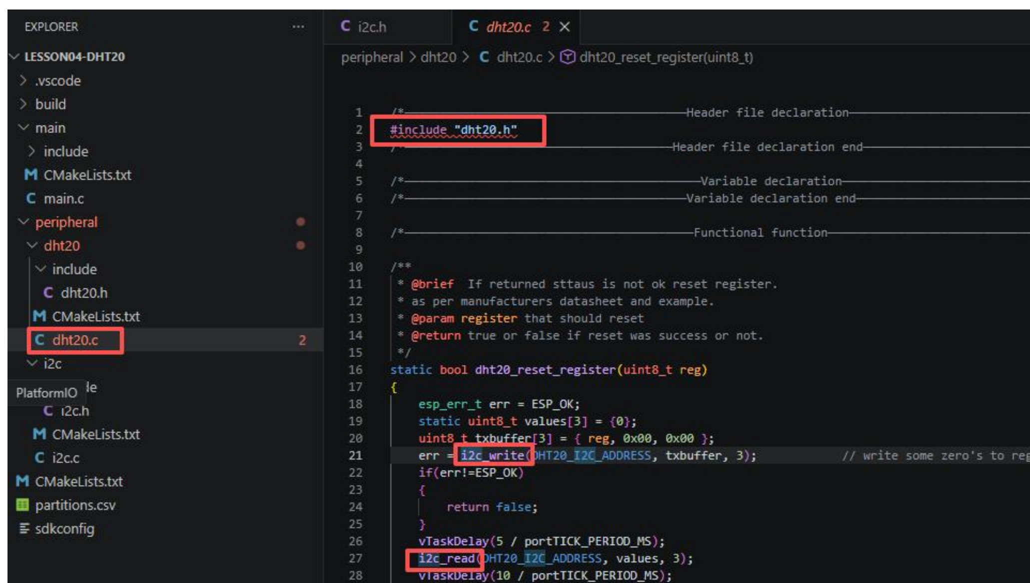


Also, remember to add i2c to the CMakeLists.txt file in dht20.



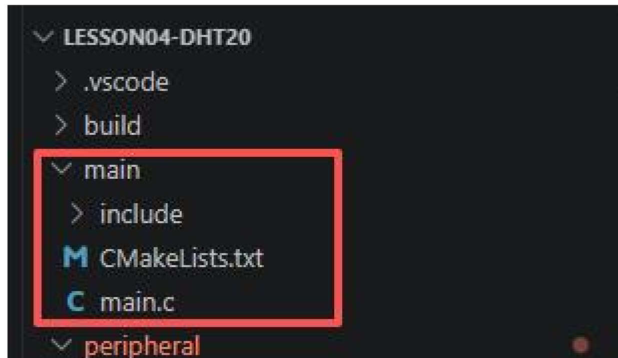
This allows you to use the I2C interface in dht20.c to write code that retrieves temperature and humidity data.

For example:



Therefore, you can simply use the two DHT20 and I2C components that we have written. We have already provided the relevant program interfaces in DHT20.h and I2C.h.
Next, let's take a look at how it's called in the main folder.

Let's take a look at the main contents of the main folder.



Let's look at main.c first. This code is a standard program based on the ESP-IDF framework that reads data from a DHT20 temperature and humidity sensor via the I2C interface. Its core functions are to initialize the sensor, wait for calibration, cyclically collect temperature and humidity values, and output the results via serial port log.

main > C main.c > app_main(void)

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "dht20.h"

static const char *TAG = "DHT20_APP";

void app_main(void)
{
    dht20_data_t measurement;

    dht20_begin();

    while (!dht20_is_calibrated()) {
        ESP_LOGI(TAG, "Waiting for DHT20 calibration...");
        vTaskDelay(pdMS_TO_TICKS(100));
    }

    ESP_LOGI(TAG, "DHT20 initialized on I2C pins SDA=45, SCL=42");

    while (1) {
        float status = dht20_read_data(&measurement);
        if (status == ESP_OK) {
            printf("Temperature: %.2f C, Humidity: %.2f %%\r\n",
                   measurement.temperature,
                   measurement.humidity);
        } else {
            ESP_LOGE(TAG, "DHT20 read failed (%d)", (int)status);
        }
        vTaskDelay(pdMS_TO_TICKS(1500));
    }
}
```

The header file inclusion statement at the beginning of the code provides all the basic functional support required by the program. stdio.h is the standard input/output library, used for data printing;
freertos/FreeRTOS.h and freertos/task.h are the core files of the FreeRTOS real-time operating system, providing task latency and system scheduling capabilities;
esp_log.h is the dedicated logging library for ESP32, used for outputting information and error messages;
dht20.h is the driver library for the DHT20 sensor, which encapsulates all low-level operations such as sensor initialization, calibration, and data reading. These header files together support the program in completing sensor communication and data processing.

```
1    #include <stdio.h>
2    #include "freertos/FreeRTOS.h"
3    #include "freertos/task.h"
4    #include "esp_log.h"
5    #include "dht20.h"
```

This line of code defines the log output label DHT20_APP as an identifier prefix for program runtime information. All logs output through ESP_LOG will have this label attached, allowing developers to quickly distinguish and locate log information related to temperature and humidity acquisition during serial port debugging, greatly improving debugging efficiency.

```
7    static const char *TAG = "DHT20_APP";
```

app_main is the only entry point function of the ESP32 program. It is executed automatically after the chip is powered on. All core logic such as sensor initialization, calibration waiting, data reading, and loop printing is implemented in this function, which is the core of the entire temperature and humidity acquisition program.

```
9    void app_main(void)
10   {
11       dht20_data_t measurement;
12
13       dht20_begin();
14
15       while (!dht20_is_calibrated()) {
16           ESP_LOGI(TAG, "Waiting for DHT20 calibration...");
17           vTaskDelay(pdMS_TO_TICKS(100));
18       }
19
20       ESP_LOGI(TAG, "DHT20 initialized on I2C pins SDA=45, SCL=42");
21
22       while (1) {
23           float status = dht20_read_data(&measurement);
24           if (status == ESP_OK) {
25               printf("Temperature: %.2f C, Humidity: %.2f %%\r\n",
26                       measurement.temperature,
27                       measurement.humidity);
28           } else {
29               ESP_LOGE(TAG, "DHT20 read failed (%d)", (int)status);
30           }
31           vTaskDelay(pdMS_TO_TICKS(1500));
32       }
33   }
```

First, define a variable of type dht20_data_t called measurement to store the read temperature and humidity data. Then, call the dht20_begin() function to start the DHT20 sensor and complete the underlying preparation work such as I2C communication initialization and pin configuration (default SDA=45, SCL=42), laying the foundation for subsequent sensor communication.

```
 9   void app_main(void)
10   {
11       dht20_data_t measurement;
12
13       dht20_begin();
```

This loop continuously checks whether the sensor has completed calibration. dht20_is_calibrated() is used to determine the sensor calibration status. If the sensor is not calibrated, it prints a waiting log in a loop, checking once every 100 milliseconds, until the sensor completes automatic calibration before exiting the loop, ensuring that the data read afterward is accurate and valid.

```
while (!dht20_is_calibrated()) {
    ESP_LOGI(TAG, "Waiting for DHT20 calibration...");
    vTaskDelay(pdMS_TO_TICKS(100));
}
```

After the sensor calibration is completed, the initialization success message is printed in the log, clearly informing the developer that the DHT20 has been initialized on the specified I2C pins (SDA=45, SCL=42) and the program is ready to enter the data acquisition stage.

```
ESP_LOGI(TAG, "DHT20 initialized on I2C pins SDA=45, SCL=42");
```

The program enters an infinite loop to ensure continuous temperature and humidity data collection. Within the loop, the dht20_read_data(&measurement) function is called to read sensor data and store the results in a predefined variable. This is the core operation for obtaining temperature and humidity values.

```
22       while (1) {
23           float status = dht20_read_data(&measurement);
```

When the read status is ESP_OK, it means that the data acquisition was successful. The program uses printf to print the temperature and humidity in a format that retains two decimal places, intuitively displaying the current temperature and humidity values of the environment, making it convenient for developers to view the real-time monitoring results.

```
if (status == ESP_OK) {
    printf("Temperature: %.2f C, Humidity: %.2f %%\r\n",
           measurement.temperature,
           measurement.humidity);
```

If the read status is not ESP_OK, it means that the sensor communication has failed. The program will print the read failure message and the corresponding error code in the error log to help developers quickly troubleshoot wiring faults, sensor abnormalities or communication problems.

After each reading is completed, there is a delay of 1500 milliseconds (1.5 seconds), which not only meets the sampling frequency requirements of the DHT20 sensor and avoids data abnormalities caused by frequent readings, but also reduces system resource consumption, allowing the program to complete temperature and humidity monitoring stably and continuously.

```
            } else {
                ESP_LOGE(TAG, "DHT20 read failed (%d)", (int)status);
            }
            vTaskDelay(pdMS_TO_TICKS(1500));
        }
    }
```

Then there's the CMakeLists.txt file in the main folder.

```
main > CMakeLists.txt
1
2   file(GLOB_RECURSE SRC_UI ${CMAKE_SOURCE_DIR} "main/*.c")
3
4   idf_component_register(
5       SRCS "main.c"
6       INCLUDE_DIRS "." "include"
7       REQUIRES dht20
8   )
9
```

This code is the CMake build configuration script for the main directory under the ESP-IDF framework. The file(GLOB_RECURSE) directive is used to automatically match and collect all .c source files in the main folder.
"idf_component_register" is the core component registration function. It specifies the main program file "main.c" to be compiled via "SRCS", and sets the compiler's search path for header files to the current directory and the "include" folder via "INCLUDE_DIRS".

The "REQUIRES dht20" statement declares that the project depends on the DHT20 sensor driver component . Since DHT20 includes I2C, which is specified in the CMakeLists.txt file within DHT20, we can also access it through a search engine. This has been explained above.

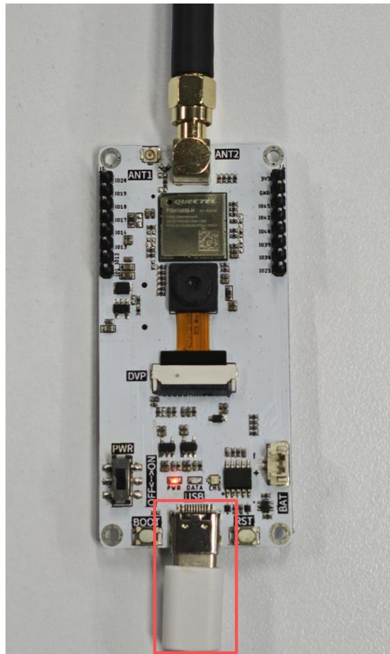
The overall configuration clearly tells the compiler which files need to be compiled, where to read header files, and which external drivers need to be linked, ensuring that the entire temperature and humidity acquisition project can be compiled, linked, and generate firmware that can run on ESP32.

Complete Code

Kindly click the link below to view the full code implementation.
https://github.com/Elecrow-RD/ESP32_Wi-Fi_HaLow_Module_with_2MP_Camera_32Mbps_High_Speed/tree/master/example/V1.0/ESP-IDF_Code/Lesson04-DHT20

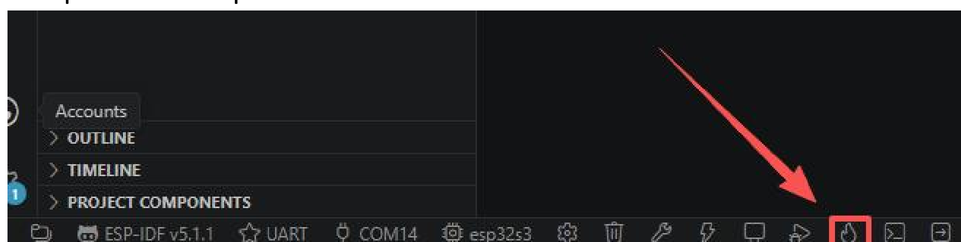
Programming Steps

That concludes the code explanation. Now, let's upload the code.
First, we connect the ESP32 WiFi-HaLow module to our computer via a USB cable.



Here, following the steps in Section 1 , first select the ESP-IDF version V5.1.1, UART, your serial port number, and the main control chip ESP32S3 for the ESP32 WiFi-HaLow module.

Then click the button to compile, upload, and open the serial monitor, and you can complete these operations with one click.



After a short wait, you will see the DHT20 temperature and humidity sensor connected to pins 42 and 45 acquiring the current temperature and humidity data.

The main purpose of this lesson is to guide you in using the GPIO pins on the ESP32 WiFi-HaLow module. After class, you can also use other pins and functions to implement different functions, just refer to the schematic diagram we provided.

Lesson05 --- Camera Web Server

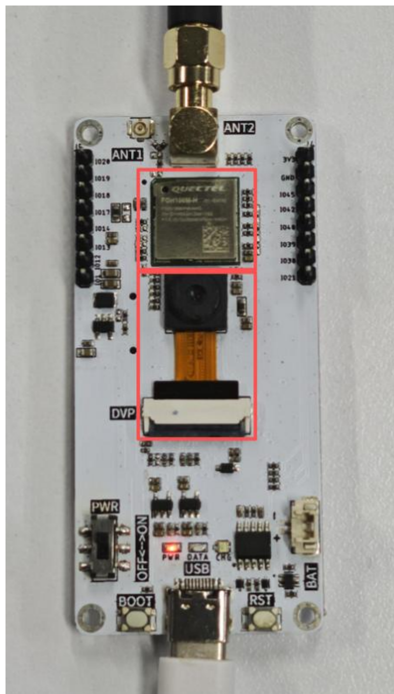
Introduction

This course focuses on the complete deployment process of Camera Web Server, teaching you step-by-step how to build a low-power, long-range, and highly wall-penetrating wireless video transmission system using the ThinkNode-G4 WiFi HaLow gateway.

You will learn: gateway initialization and backend login, HaLow network parameter configuration, connecting to a home router, setting up and managing WiFi, and finally enabling the ESP32 WiFi-HaLow module to access the network and activate the Camera Web Server. You can view the live feed simply by entering the IP address assigned to the ESP32 WiFi-HaLow module by the ThinkNode-G4 WiFi HaLow gateway in your browser, without any additional software.

Hardware Used in This Lesson

In this lesson, we will use the camera and WiFi HaLow wireless communication chip on the ESP32 WiFi-HaLow module. After the camera captures an image, the image data will be transmitted locally through WiFi HaLow wireless communication, without requiring an external network server. Devices within the same network can access the web page to view the real-time camera image. In addition, WiFi HaLow supports long-range wireless communication, allowing camera data to be received over distances of approximately 1–2 kilometers under suitable conditions, making it ideal for remote monitoring and long-distance IoT applications.



Wi-Fi HaLow, based on the IEEE 802.11ah standard, is a low-power, long-range wireless communication technology designed by the Wi-Fi Alliance specifically for Internet of Things (IoT) scenarios. Compared with traditional Wi-Fi technologies, it operates in the Sub-1 GHz unlicensed frequency band (mainly the 900 MHz band), and this frequency selection gives it revolutionary advantages in propagation characteristics. Low-frequency radio waves have stronger diffraction capabilities and penetration, enabling them to effectively pass through buildings, vegetation, and other obstacles, thus solving the inherent limitations of traditional Wi-Fi in terms of coverage and environmental penetration.

Since the ESP32 Wi-Fi HaLow module uses Wi-Fi HaLow technology for wireless communication, it requires a Wi-Fi HaLow gateway to receive and forward the transmitted data. In this project, we use the ThinkNode-G4 Wi-Fi HaLow gateway as the communication bridge. Therefore, besides the ESP32 Wi-Fi HaLow module, you also need to prepare a ThinkNode-G4 Wi-Fi HaLow gateway.

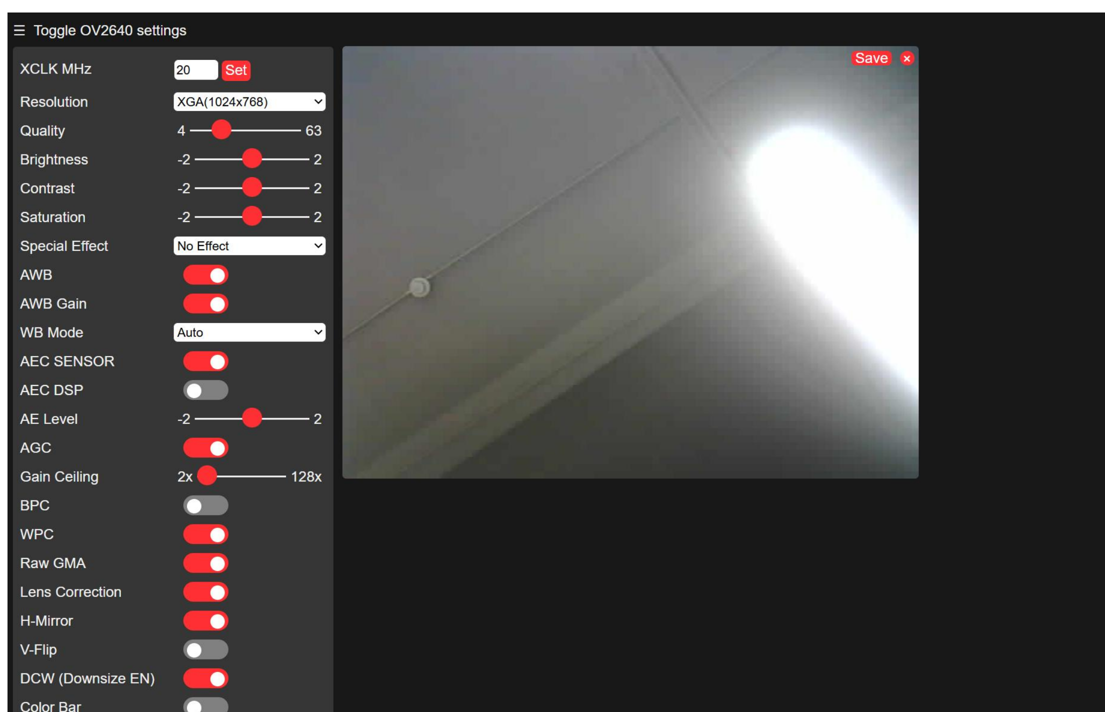
The Wi-Fi HaLow Gateway from ELECROW is an innovative Wi-Fi HaLow gateway designed to meet the long-distance/high-speed data transmission requirements of IoT applications. This gateway adopts Wi-Fi HaLow (IEEE 802.11ah) technology and operates in the sub-1 GHz unlicensed frequency band. Compared with traditional Wi-Fi standards, it has stronger penetration and a larger coverage area. It is equipped with powerful hardware, including advanced RF functions, high-performance MCU, and flexible interfaces, allowing seamless integration with existing networks. It can be easily configured and upgraded over-the-air (OTA) through a Web UI and supports a large number of devices to connect simultaneously, making it an excellent solution for smart manufacturing, smart cities, and more.

Operation Effect Diagram

After successfully burning the code onto the ESP32 WiFi-HaLow module and running it, the ThinkNode-G4 WiFi HaLow gateway will assign an IP address to the ESP32 WiFi-HaLow module.

```
WLAN STA connected
Link is up. Time: 8940 ms, IP: 192.168.12.126, Netmask: 255.255.255.0, Gateway: 192.168.12.1
I (9918) s3 ll_cam: DMA Channel=2
I (9918) cam_hal: cam init ok
I (9918) sccb: pin_sda 45 pin_scl 42
I (9918) sccb: sccb_i2c_port=1
I (9938) camera: Camera PID=0x26 VER=0x42 MIDL=0x7f MIDH=0xa2
I (9938) camera: Detected OV2640 camera
I (9938) camera: Detected camera at address=0x30
I (10008) cam_hal: PSRAM DMA mode disabled
I (10008) cam_hal: buffer_size: 16384, half_buffer_size: 1024, node_buffer_size: 1024, node_cnt: 16, total_cnt: 60
I (10018) cam_hal: Allocating 61440 Byte frame buffer in PSRAM
I (10018) cam_hal: Allocating 61440 Byte frame buffer in PSRAM
I (10028) cam_hal: cam config ok
I (10028) ov2640: Set PLL: clk_2x: 0, clk_div: 0, pclk_auto: 0, pclk_div: 8
I (10098) camera_httpd: Starting web server on port: '80'
I (10108) camera_httpd: Starting stream server on port: '81'
```

Then, by opening the IP address in a web browser, we can see the live feed from the camera on the ESP32 WiFi-HaLow module .



Key Explanations

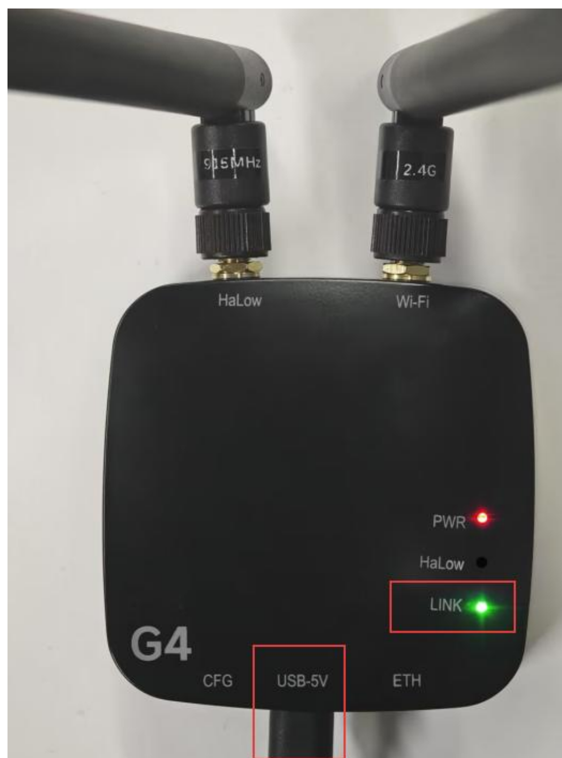
Next, we'll set up the environment together so that we can see the camera feed from the ESP32 WiFi-HaLow module on a web browser.

Configure the ThinkNode-G4 WiFi HaLow Gateway

First, let's configure the ThinkNode-G4 WiFi HaLow gateway.

The ThinkNode-G4 WiFi HaLow gateway is configured to build a low-power, long-distance, and strong wall-penetrating wireless bridge, allowing the ESP32 WiFi-HaLow module to output video streams through the Camera Web Server. At the same time, it establishes a network connection with the home router, enabling devices such as computers and mobile phones to access the camera IP directly through a browser within the same local area network to view real-time footage, achieving stable long-distance network monitoring and video transmission.

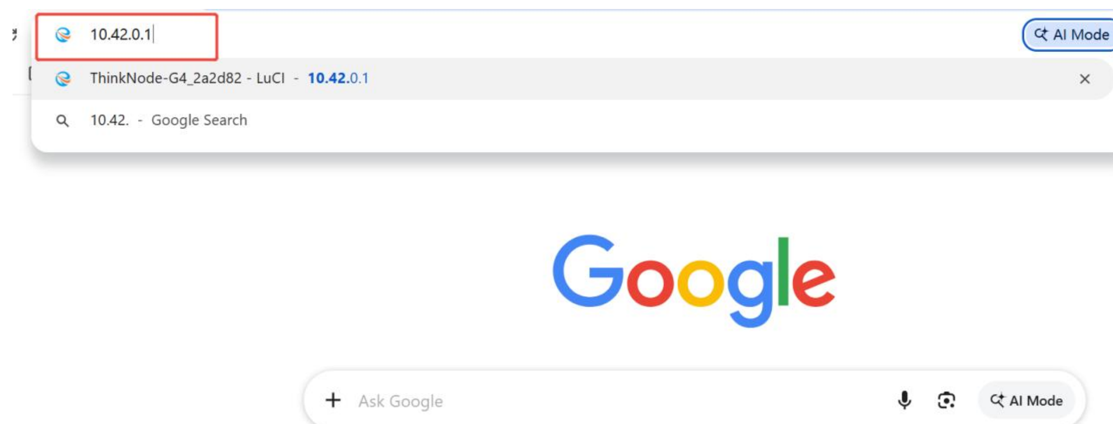
First, power on the ThinkNode - G4 WiFi HaLow gateway by plugging in the USB port. Then wait at least 90 seconds until you see the LINK light on the gateway device start flashing before proceeding to the next step .



Search for and connect to the WiFi network named ThinkNode-G4_xxxxxx on your computer. The password is elecrow.com.

After successfully connecting, open your browser and enter 10.42.0.1.

The ThinkNode-G4 WiFi HaLow gateway comes with a default WiFi configuration set up at the factory, specifically for initial setup. 10.42.0.1 is the gateway's default management IP address; only by connecting to this WiFi can users access the backend.

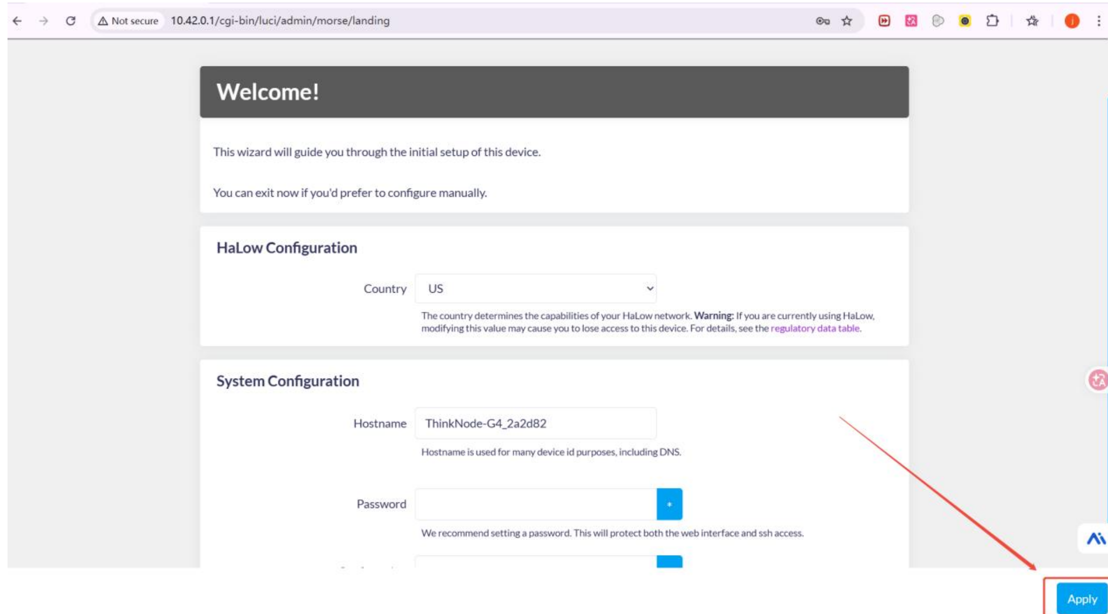


After logging in, you will arrive at this page. Initially, we did not set a password, so you can log in directly for the first time.

Note: If you set a password later and want to run G4 in its initial state again, you need to press and hold the CFG button on the device for 10 seconds to return to the initial password-free state.

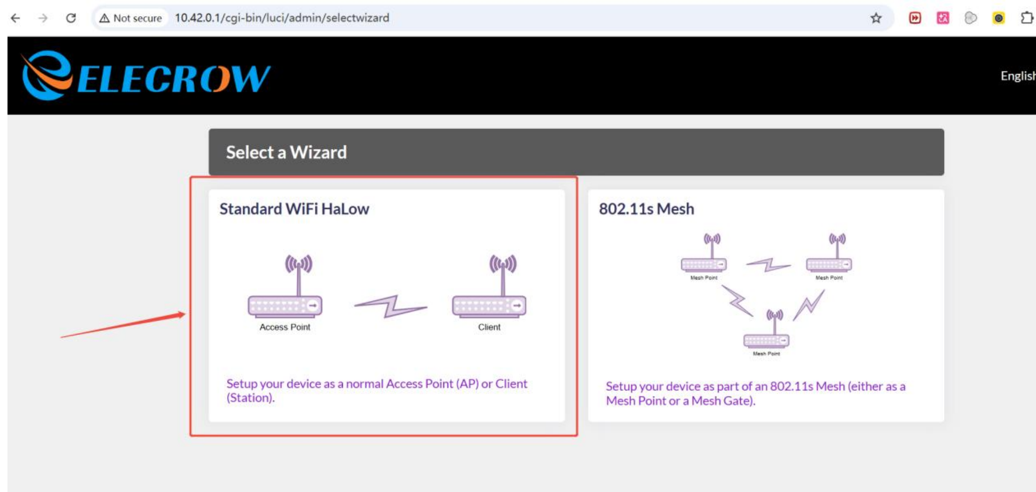
Now click Log In to enter the settings interface.

Skip the welcome page without changing the country or hostname.



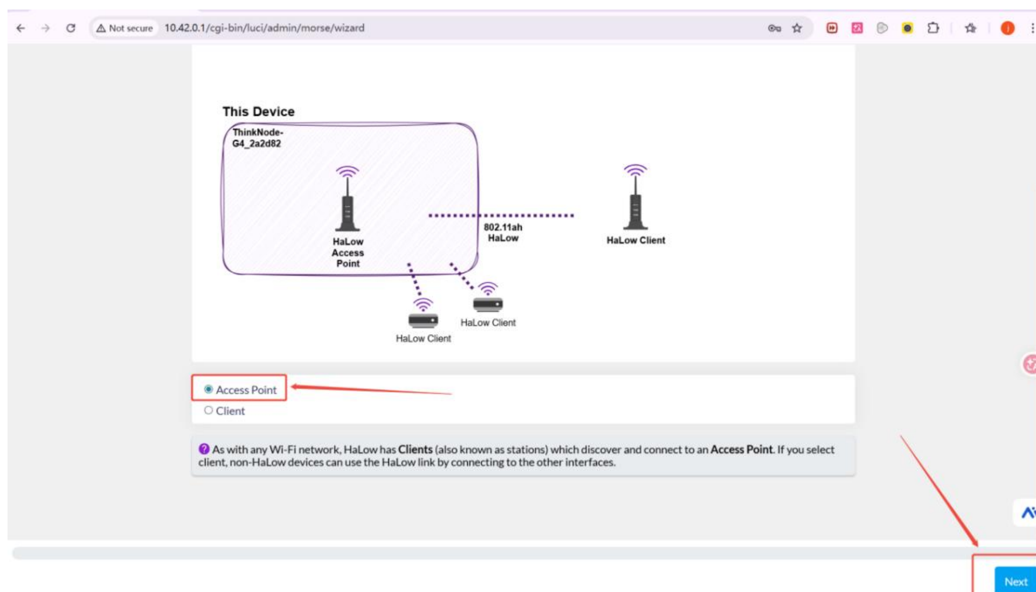
Select Standard WiFi HaLow mode and set it as an Access Point (AP).

Standard WiFi HaLow: Standard peer-to-peer/LAN mode, most stable with lowest latency, suitable for video transmission.



AP mode: Allows the gateway to actively transmit HaLow signals, acting as a "central base station".

In the mode selection interface, select the Standard WiFi HaLow mode and set the gateway to Access Point (AP) mode. This mode is the optimal choice for adapting the HaLow module and camera transmission. In AP mode, the gateway will actively transmit HaLow wireless signals and act as the central base station of the entire network, allowing the ESP32 WiFi-HaLow module as a client to connect stably. This is the core prerequisite for realizing communication between the module and the gateway.



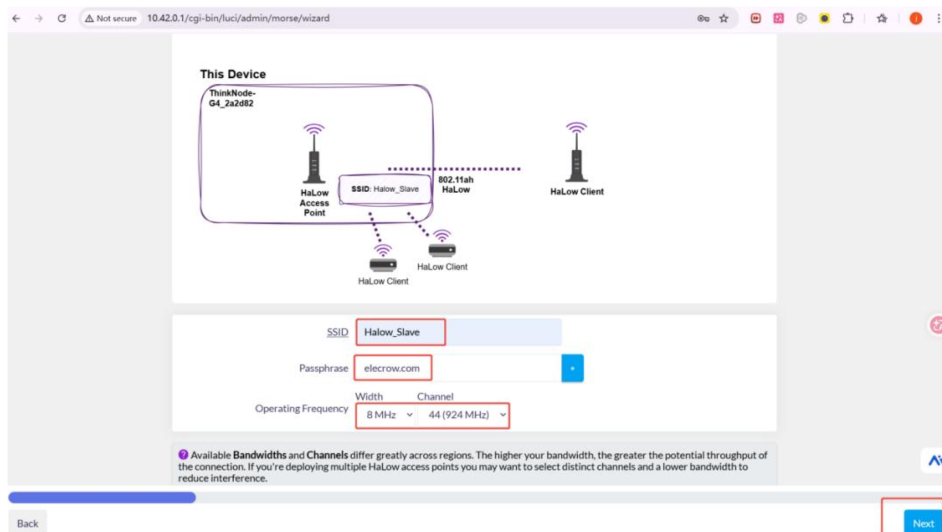
For you: Your ESP32 WiFi-HaLow module is a client; it requires a HaLow AP as a central hub for the module to connect to the gateway. This is the core prerequisite for the camera to connect to the gateway.

Next, set the account, password, channel, and bandwidth for the Wi-Fi Hallow section.

SSID / Password: Enables the ESP32 WiFi-HaLow module to accurately identify and securely connect to this network.

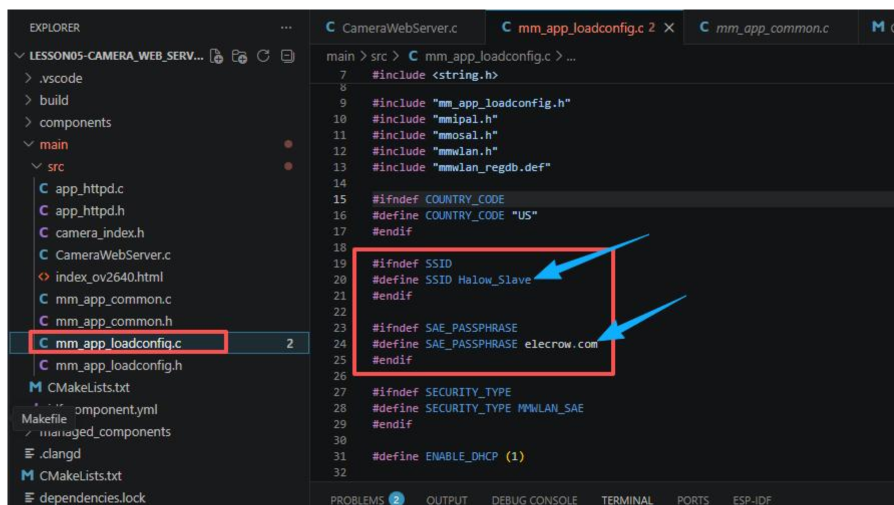
8MHz: The commonly used bandwidth of the HaLow standard, which balances penetration ability, transmission distance and speed, and is sufficient for transmitting camera video.

Channel 44 (924MHz): sub-GHz band, less interference, strong wall penetration, suitable for stable transmission at fixed locations such as cameras.



This means that the parameters must be exactly the same as the code configuration in your ESP32 WiFi-HaLow module; otherwise, the module will not be able to connect to the gateway, and the camera will not be able to go online.

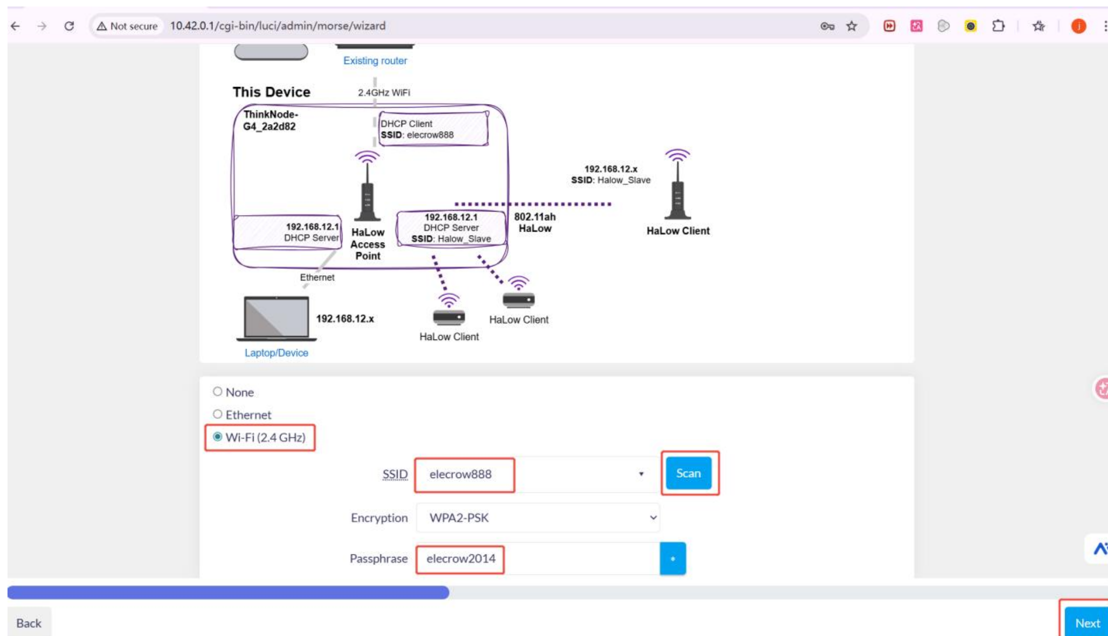
Code configuration:



Select WiFi (2.4GHz), enter your home router's SSID: elecrow888, password: elecrow2014, and encryption: WPA2-PSK.

Reason: The G4 itself does not have direct internet access capability; it must connect to your main router via the 2.4G network to obtain public/local area network access. This step enables the gateway to have "internet access capability".

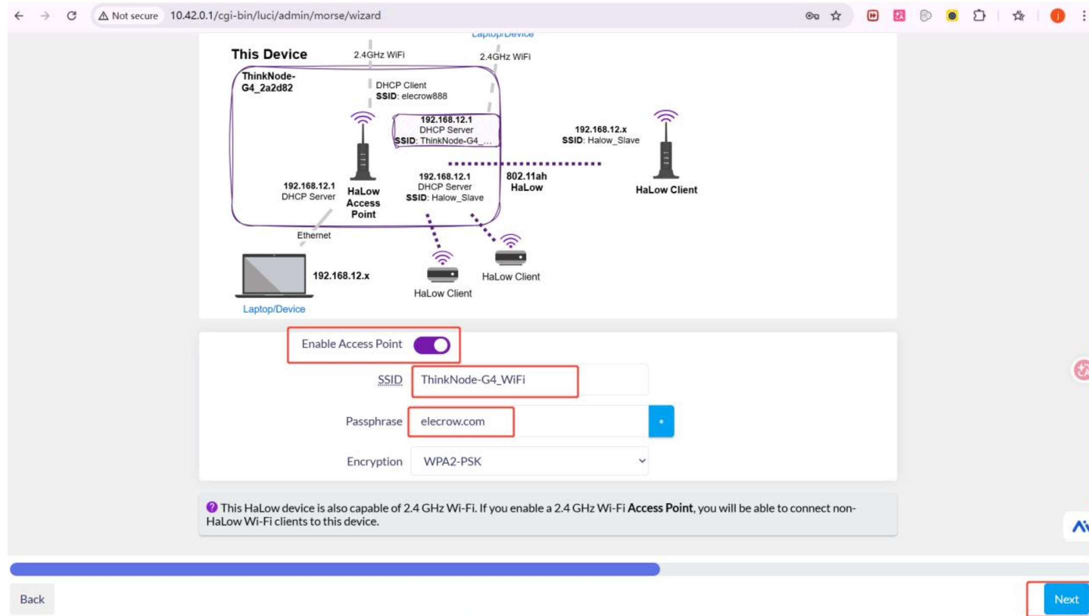
What this means for you: Only when the gateway is connected to the router can your computer access the camera feed on the ESP32 WiFi-HaLow module within the same local area network.



Enable the ThinkNode-G4 WiFi HaLow gateway's own 2.4G AP for easy connection management .

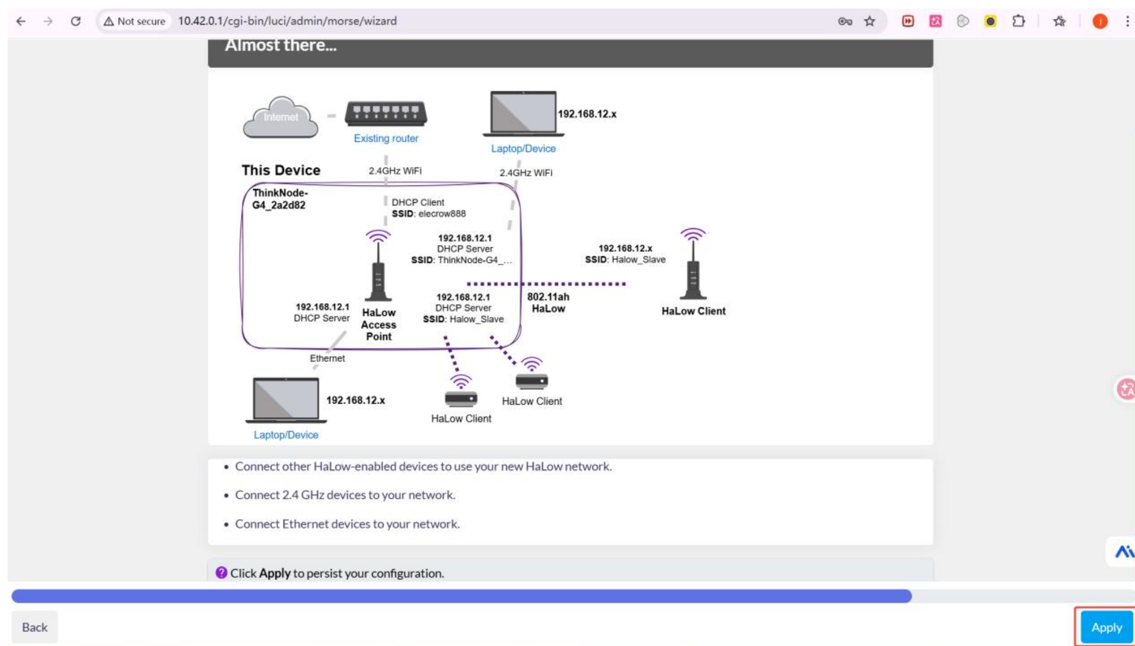
Set the SSID to ThinkNode-G4_WiFi, the password to elecrow.com, and the encryption method to WPA2-PSK. After configuration, the gateway's default WiFi configuration will be turned off. This custom 2.4G AP will serve as a stable management and access point, allowing mobile phones and computers to connect to the gateway for extended periods. It will also ensure that all devices are on the same local area network segment, providing network support for accessing camera footage.

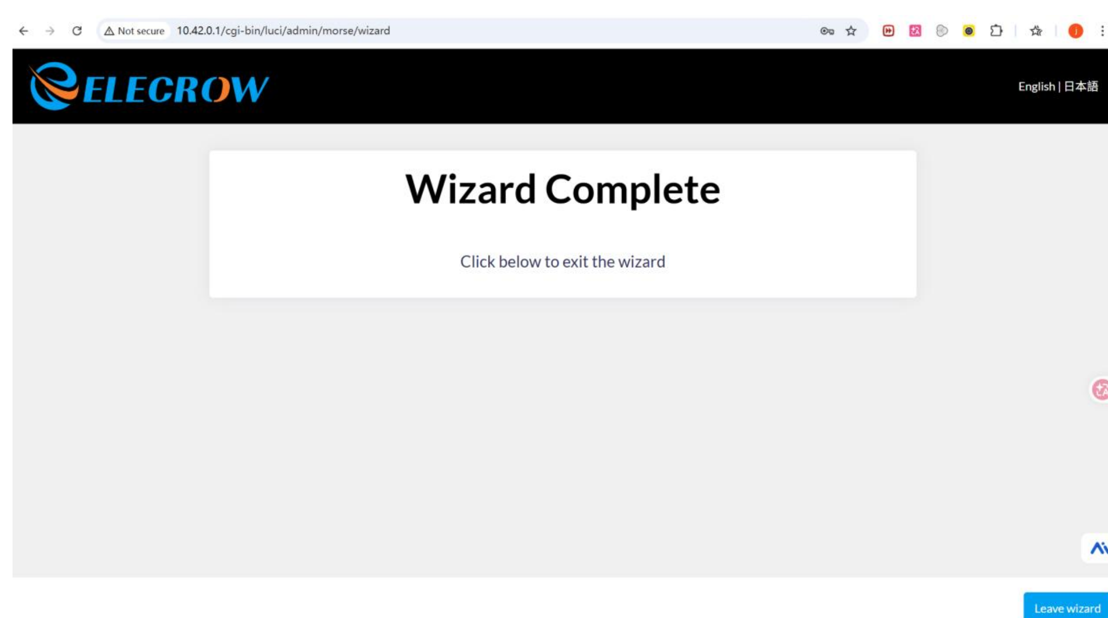
Its significance to you: You will need to connect to this WiFi network to view camera footage and debug modules later. It is the "operation entry point" between you and the entire HaLow network.



After completing all parameter settings, click Apply to save the configuration and wait for the gateway to restart automatically until the LINK light turns solid green.

The Apply operation will write all settings, including HaLow parameters, 2.4G network connectivity, and local AP settings, into the system and make them effective. A green LINK light indicates that the HaLow service, network connection, and LAN access are all normal, and the gateway enters a ready state for normal operation.





After the LINK light turns green, connect to the new WiFi and manage the gateway using the new IP address 192.168.12.1.

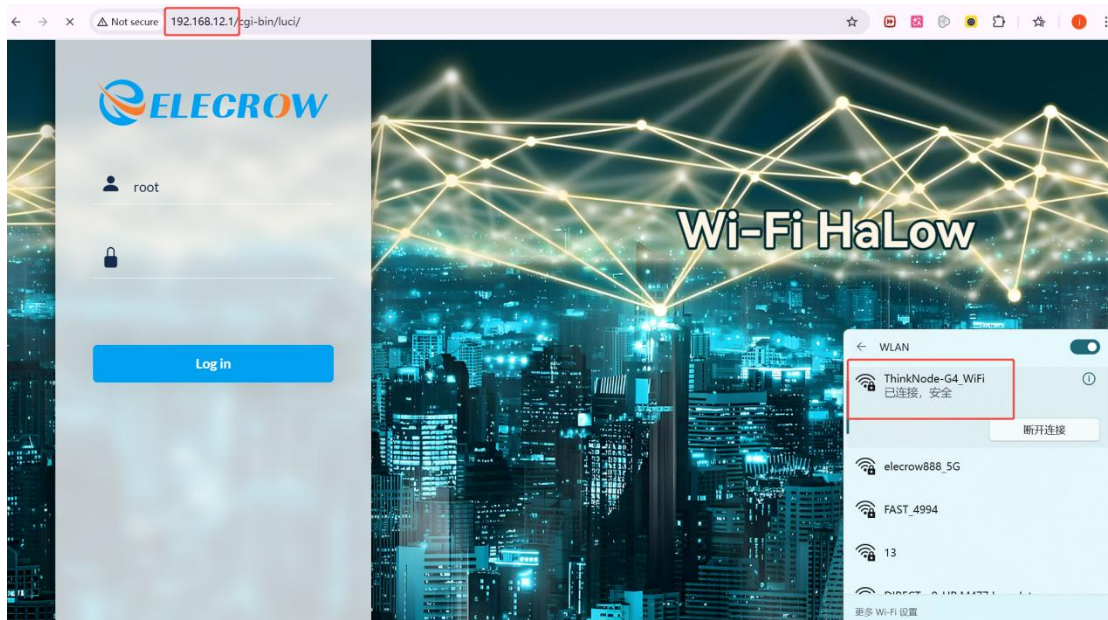
Instructions: Disconnect the old WiFi, connect to ThinkNode-G4_WiFi, password elecrow.com; access 192.168.12.1 in your browser.

Reason: After configuration, the gateway switches to the internal network segment. 192.168.12.1 is the new management address, and all devices (including your camera module) will be in this network segment.

What it means to you:

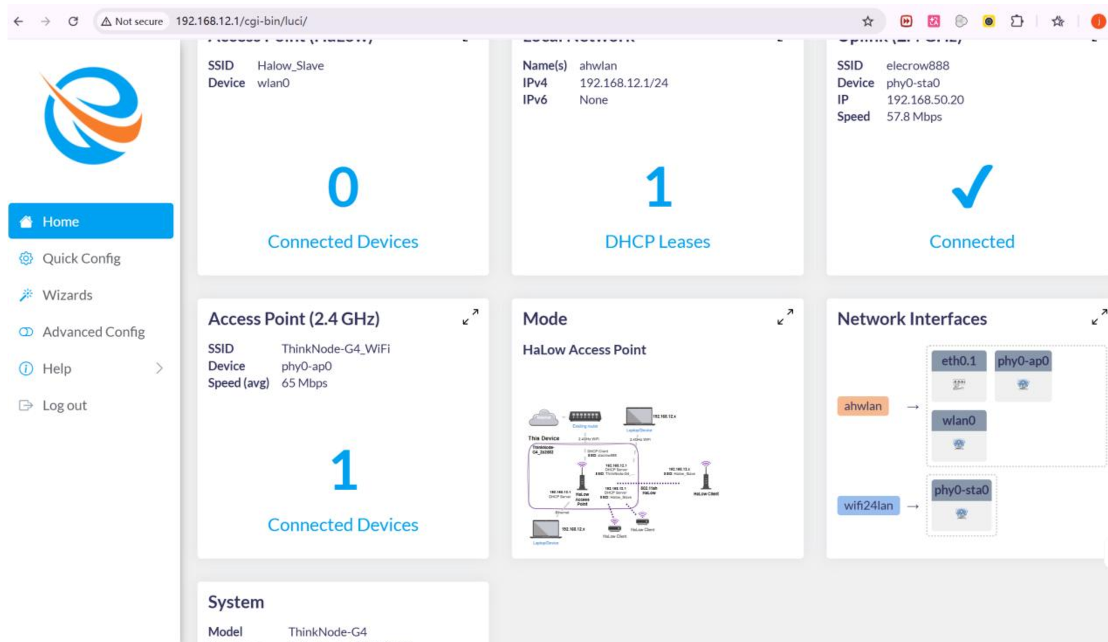
- You are in this network segment
- The gateway is in this network segment
- HaLow camera modules are also in this network segment.

Only when all three are interconnected can you open the live feed from your webcam in your browser.



1. Your HaLow camera module → connects to the gateway's HaLow AP (Halow_Slave) in Client mode.
2. The module obtains an IP address from the 192.168.12.x network segment.
3. Your computer is connected to the G4's 2.4G (ThinkNode-G4_WiFi), and is on the same network segment.
4. You can see the live feed by simply entering the camera module's IP address into your browser.

Once you enter, you will see the current interface.



Complete Code

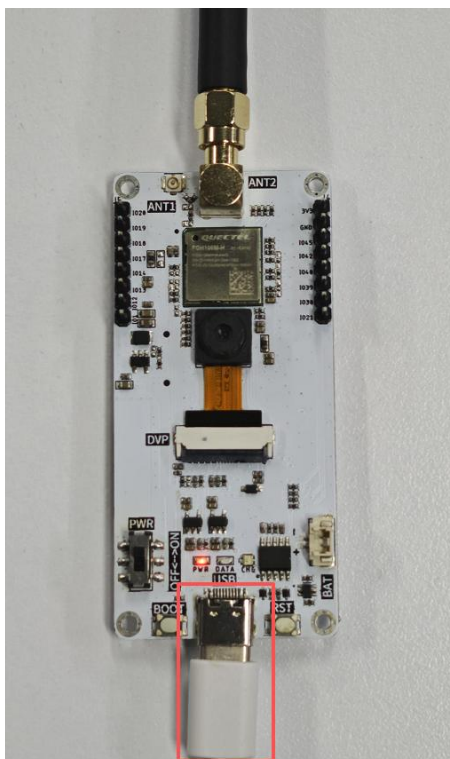
Kindly click the link below to view the full code implementation.

(We won't go into detail about the camera code here; just know how to use it. It involves modifying the Wi-Fi username and password in mm_app_loadconfig.c within the main folder. This has been mentioned in previous and later sections.)
https://github.com/Elecrow-RD/ESP32_Wi-Fi_HaLow_Module_with_2MP_Camera_32Mbps_High_Speed/tree/master/example/V1.0/ESP-IDF_Code/Lesson05-Camera_Web_Server

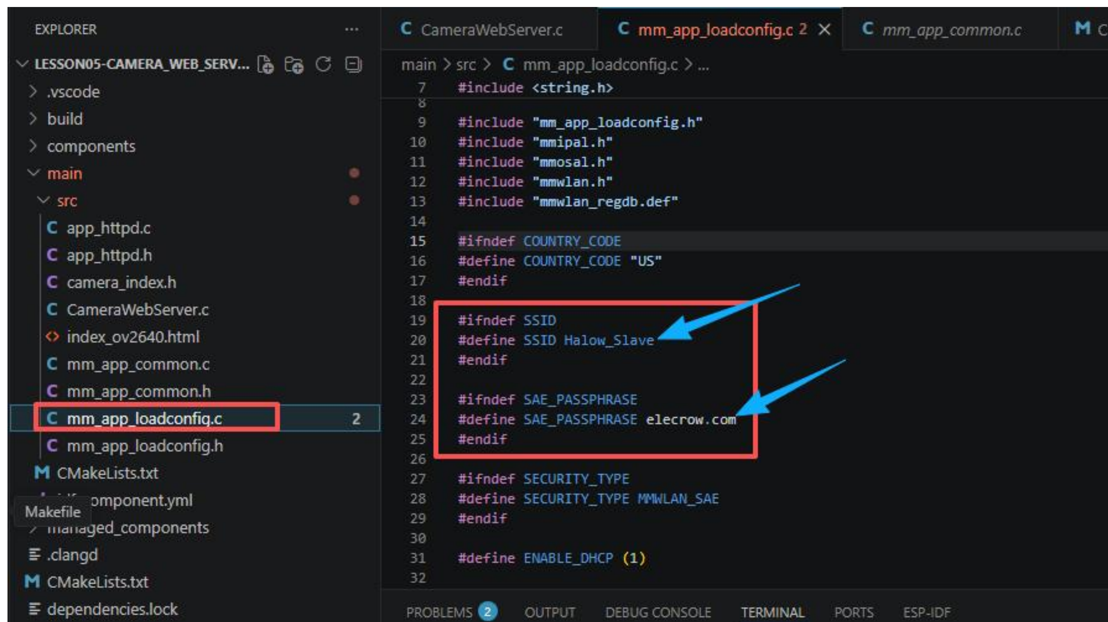
Programming Steps

That completes the configuration of the ThinkNode-G4 WiFi HaLow gateway. Next, we will handle the client upload code.

First, we connect the ESP32 WiFi-HaLow module to our computer via a USB cable.

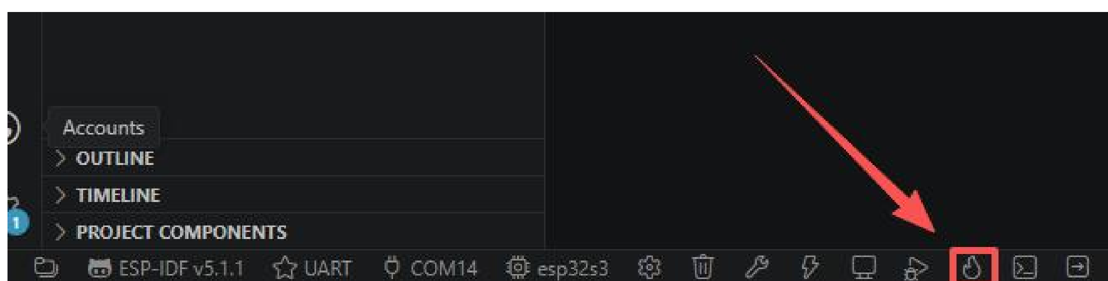


Make sure you change the Wi-Fi Halow username and password you set in the code.



Finally, following the steps in Section 1, first select the ESP-IDF version V5.1.1, UART, your serial port number, and the main control chip ESP32S3 for the ESP32 WiFi-HaLow module.

Then click the button to compile, upload, and open the serial monitor, and you can complete these operations with one click.

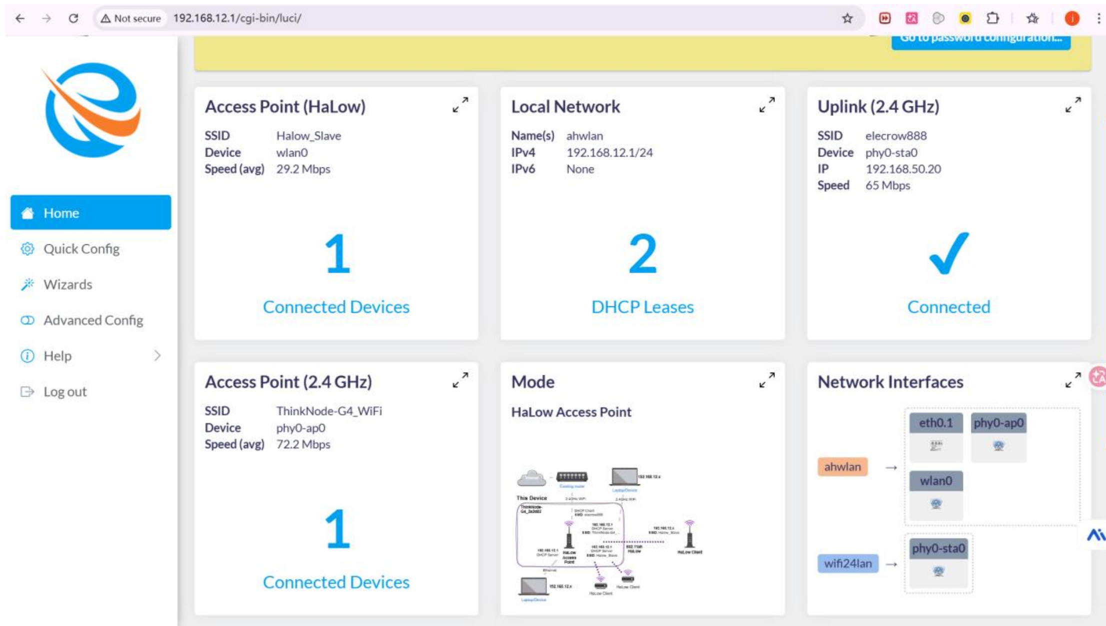


Now, set the IP address and password of the gateway you are connecting to in the code. After the code upload is complete, you will be able to see the IP address assigned to your WiFiHalow module by the gateway in the serial monitor. In my case, it is 192.168.12.126.

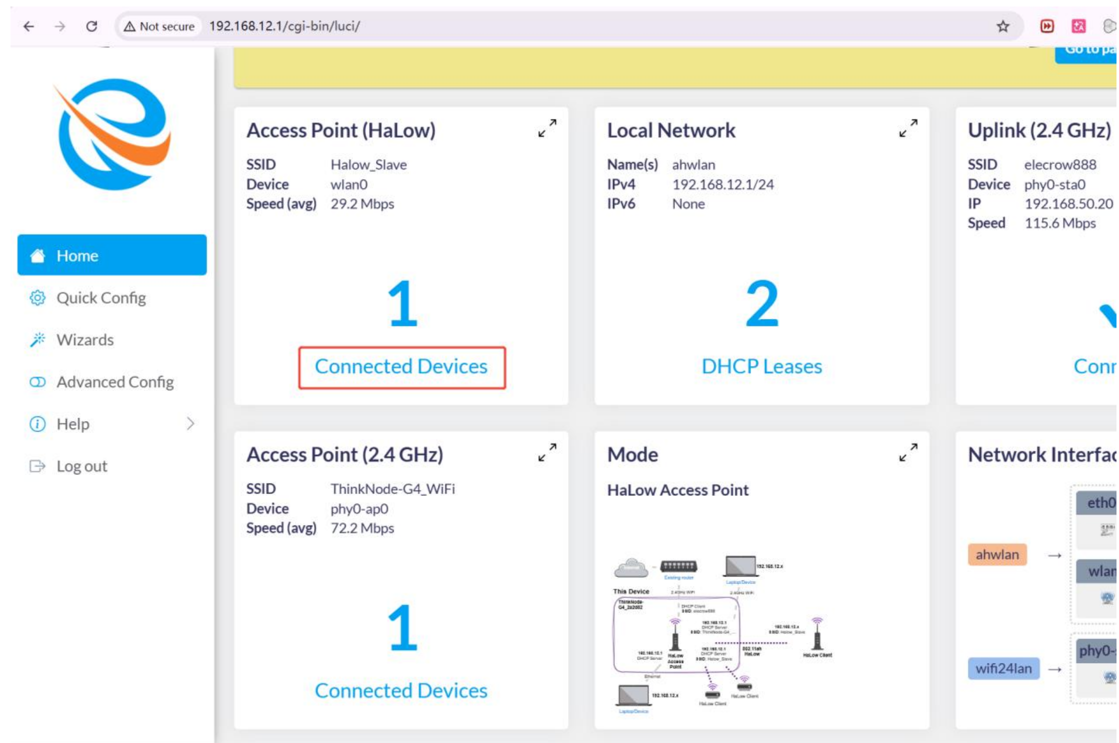
```
Morse LwIP interface initialised. MAC address 3c:22:7f:71:dd:af
Morse firmware version 1.13.1, morselib version 2.6.4-esp32, Morse chip ID 0x306

Attempting to connect to Halow_Slave with passphrase elecrow.com
This may take some time (~30 seconds)
WLAN STA connecting
WLAN STA connected
Link is up. Time: 8940 ms, IP: 192.168.12.126, Netmask: 255.255.255.0, Gateway: 192.168.12.1
I (9918) s3 ll_cam: DMA Channel=2
I (9918) cam_hal: cam init ok
I (9918) sccb: pin_sda 45 pin_scl 42
I (9918) sccb: sccb_i2c_port=1
I (9938) camera: Camera PID=0x26 VER=0x42 MIDL=0x7f MIDH=0xa2
I (9938) camera: Detected OV2640 camera
I (9938) camera: Detected camera at address=0x30
I (10008) cam_hal: PSRAM DMA mode disabled
I (10008) cam_hal: buffer_size: 16384, half_buffer_size: 1024, node_buffer_size: 1024, node_cnt: 16, total_cnt: 60
I (10018) cam_hal: Allocating 61440 Byte frame buffer in PSRAM
I (10018) cam_hal: Allocating 61440 Byte frame buffer in PSRAM
I (10028) cam_hal: cam config ok
I (10028) ov2640: Set PLL: clk_2x: 0, clk_div: 0, pclk_auto: 0, pclk_div: 8
I (10098) camera_httpd: Starting web server on port: '80'
I (10108) camera_httpd: Starting stream server on port: '81'
```

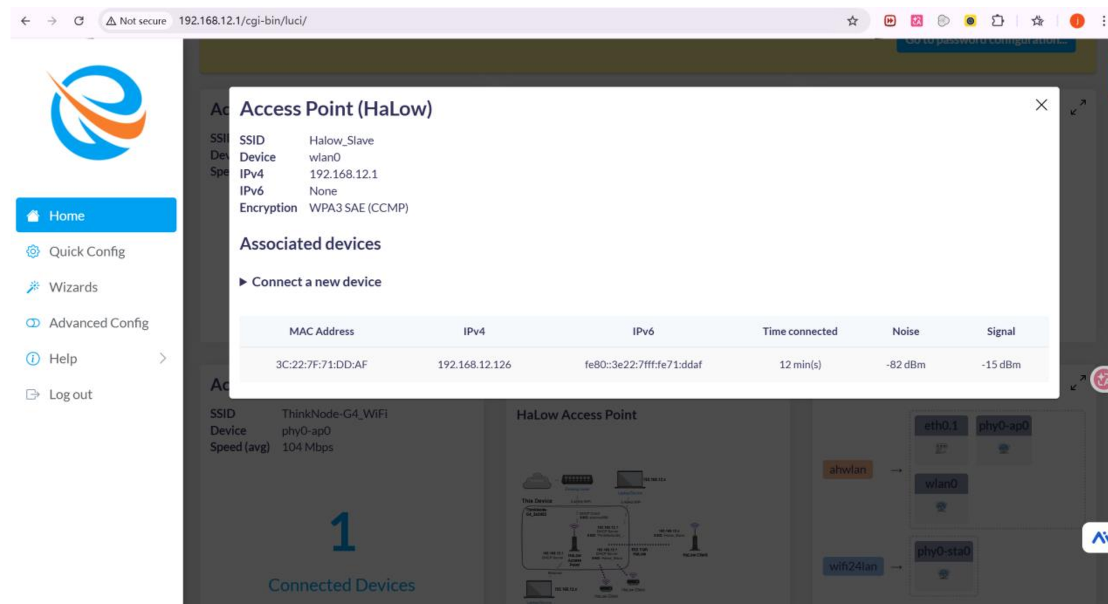
And you can see the Halow devices you've connected to from the web-based management interface.



Click Connected Devices

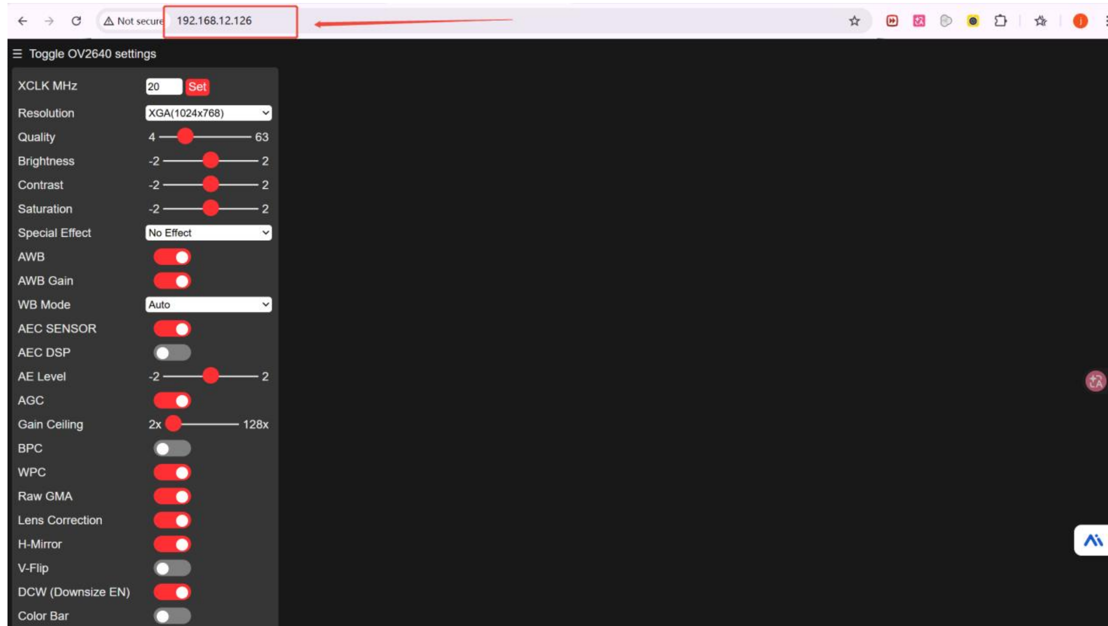


You can see information about the devices currently connected to the gateway.

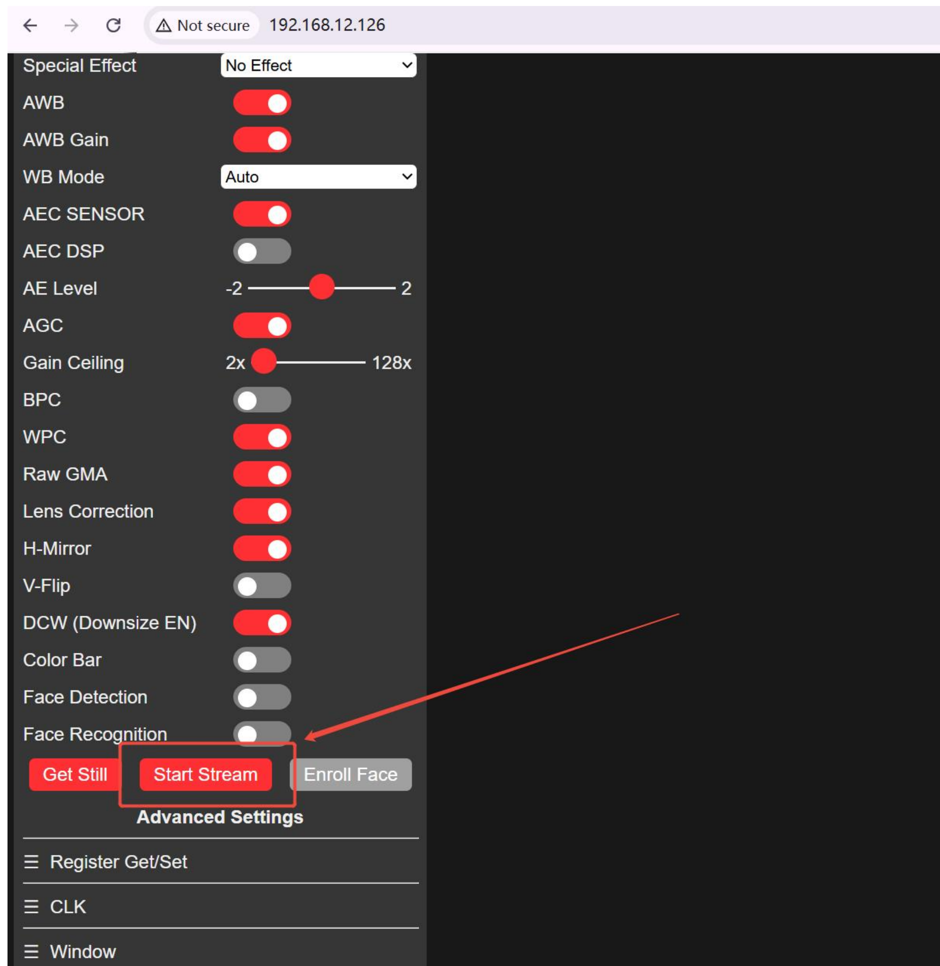


Furthermore, you can see the Halow indicator light on the gateway illuminated, further proving that my ESP32 WiFi-HaLow module has been connected to the Halow gateway.

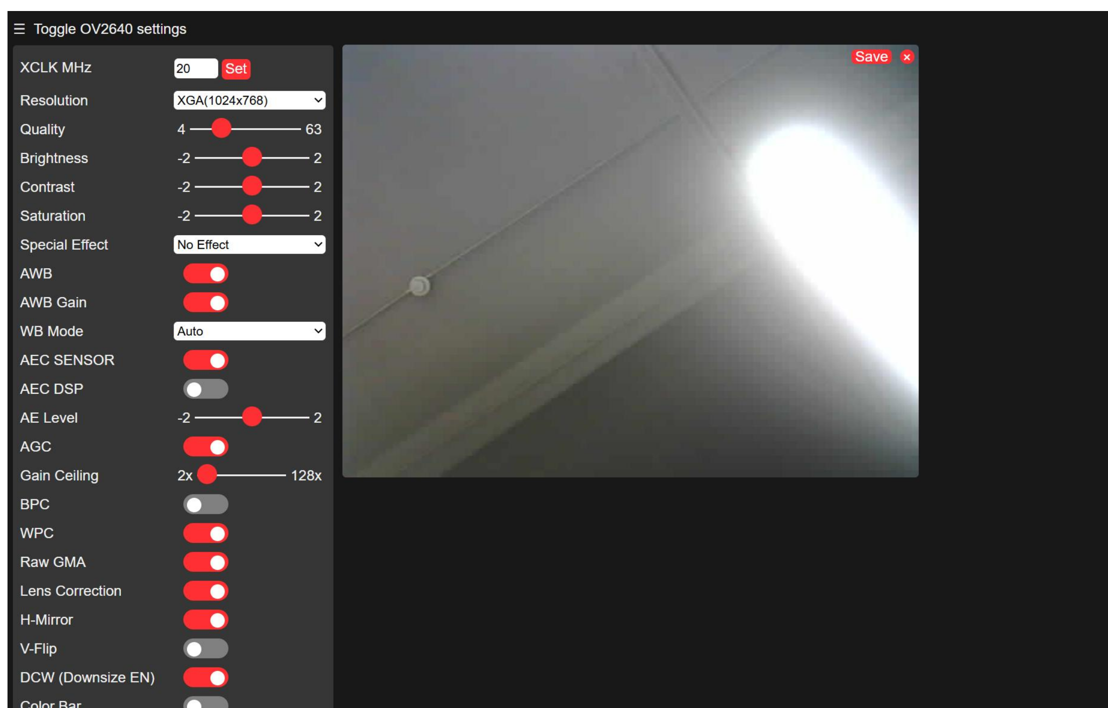
So after I obtained the IP address of this ESP32 WiFi-HaLow module , I was able to find this page by searching for it in my browser.



Then click Start Stream,



You'll then see the camera on the ESP32 WiFi-HaLow module transmitting the image.



Finally, we'll briefly explain the functions of the options on the left side of this web interface to help you navigate them.

XCLK MHz

The camera's external clock frequency (20MHz in this case) serves as the base clock for the entire sensor's operation. Impact: A frequency that is too low may cause stuttering or screen tearing; a frequency that is too high will increase power consumption and may even exceed the sensor's limits. It's generally not recommended to change this frequency arbitrarily unless you are confidently performing low-level driver optimization.

Resolution

Here, XGA (1024×768) is selected, which is the pixel size of the image. Impact: Higher resolution means more image detail, but also a larger data transfer volume, potentially leading to a lower frame rate and increased latency; lower resolution results in a better frame rate and smoother playback, but some detail will be lost. Common options also include QVGA (320×240) and VGA (640×480).

Quality (Image Quality/Compression Quality)

The slider here ranges from 4 to 63. A smaller value results in a lower JPEG compression rate, better image quality, and a larger file size; a larger value results in more severe compression, lower image quality, and a smaller data size. Purpose: To balance image quality and transmission speed/bandwidth usage.

Brightness

The slider ranges from -2 to 2, controlling the overall brightness of the image. Uses: Increase the brightness if the light is too dim, and decrease it if the light is too bright. However, if the image is overexposed or completely black, simply adjusting the brightness won't fix it; you'll need to adjust it in conjunction with the exposure parameters.

Contrast

The slider range is -2 to 2, controlling the contrast between bright and dark areas of the image. Applications: Moving the slider upwards increases black-and-white contrast, resulting in a "harder" image; moving it downwards decreases contrast, resulting in a "greasier" image, suitable for hazy scenes.

Saturation

The slider ranges from -2 to 2, controlling the intensity of colors in the image. Uses: Moving the slider upwards makes the colors more vibrant; moving it downwards makes the colors lighter; adjusting it to the lowest setting will result in a black and white image.

Special Effect

The drop-down menu here selects "No Effect". Common options include filters such as black and white, retro, negative, sketch, and high contrast, which only change the style of the image and do not affect the original data.

AWB (Auto White Balance)

Switch control. When turned on, the camera automatically recognizes the color temperature of the ambient light, restoring white objects to white and preventing the image from appearing yellowish/bluish. When turned off, it fixes the current color temperature, suitable for scenes with fixed light sources.

AWB Gain (White Balance Gain)

White balance gain control, used in conjunction with AWB, affects the sensitivity of white balance

adjustments. When enabled, white balance adjustments become more aggressive; when disabled, the gain is fixed, resulting in a more stable color temperature.

WB Mode (White Balance Mode)

Here, Auto is selected. The dropdown menu usually offers preset modes such as Sunny, Cloudy, Fluorescent, and Incandescent. Purpose: In Auto mode, the camera adjusts automatically; the fixed mode is suitable for specific light sources, such as the Incandescent mode for more accurate images in indoor incandescent lighting scenarios.

AEC SENSOR (Auto Exposure Control)

Automatic exposure control switch at the sensor level. When turned on, the sensor automatically adjusts the exposure time according to the brightness of the scene; when turned off, the exposure time and the brightness of the scene are fixed, which is suitable for scenes with stable lighting.

AEC DSP (Auto Exposure Control - Automatic Exposure on DSP)

Automatic exposure control switch at the DSP (Digital Signal Processing) level. When enabled, the DSP will further fine-tune the exposure to make the image brightness more stable; when disabled, only the sensor-level exposure control is used, and the response will be slower.

AE Level (Auto Exposure Level)

The slider range is -2 to 2, controlling the target brightness for automatic exposure. Uses: Moving it upwards (+) makes the overall image brighter; moving it downwards (-) makes the overall image darker. It's equivalent to setting a "preferred brightness" for automatic exposure.

AGC (Auto Gain Control)

Automatic gain control, also known as automatic brightness adjustment, means the camera will automatically amplify the signal to brighten the image in low light conditions; however, higher gain will result in more noise (snow-like noise). When turned off, the gain is fixed, resulting in less noise but darker scenes.

Gain Ceiling

This range is 2x to 128x, limiting the maximum gain that AGC can achieve. Purpose: Limiting the maximum gain can prevent excessive image noise; however, if the upper limit is too low, low-light scenes will be too dark. For example, 2x is a very low gain, resulting in a clean but dark image; 128x is the maximum gain, resulting in a bright image but with a lot of noise.

BPC (Bad Pixel Correction)

Dead pixel correction switch. When turned on, the camera will automatically identify and repair dead pixels on the sensor (fixed bright/black spots on the screen); when turned off, dead pixels will appear directly in the image.

WPC (White Pixel Correction)

The white point correction switch is specifically designed to repair bright and dead pixels in the image. When enabled, it replaces abnormally bright dead pixels with the average value of surrounding pixels, reducing white points in the image.

Raw GMA (Raw Gamma Correction)

The gamma correction switch on the original image affects the grayscale levels and tonal transitions of the image. When enabled, it optimizes the dynamic range of the image, making the details in the highlights and shadows more balanced; when disabled, the gamma curve of the image is fixed, and the contrast may be too harsh or too gray.

Lens Correction

Lens distortion correction switch. When turned on, it corrects barrel/pincushion distortion (the edges of the image bulge outwards or concave inwards), making straight lines straighter; when turned off, it maintains the lens's original distortion, suitable for some special shooting effects.

H-Mirror (Horizontal Mirror)

The screen can be flipped horizontally or mirrored left and right. When turned on, the screen will flip horizontally, which is suitable for scenarios where the camera is mounted upside down or where a mirrored image is needed (such as when used as a mirror).

V-Flip (Vertical Flip)

The screen flip switch reverses the orientation. When turned on, the screen will flip vertically, suitable for scenarios where the camera is installed upside down, ensuring the image is oriented correctly.

DCW (Downsize EN, Downsize Enable)

Image downsampling/scaling switch.

When enabled, the camera can output images at a lower resolution than the sensor's native resolution, such as reducing 1024×768 to 640×480, thus reducing data volume and increasing frame rate; when disabled, it can only output the native resolution.

Color Bar (Color Bar Test Chart)

Color bar test pattern switch.

When enabled, the screen will display a standard color bar pattern. This is used to test whether the camera's color channels are working properly and whether the display device can display colors correctly. It's a debugging function and shouldn't be used normally.

Get Still: Get a still image, take a snapshot of the current scene and save/download it.

Start Stream: Enables video streaming, which is the live feed you see in the first image; if you turn it off, the screen will go black, and the second image shows the state when streaming is off.

Enroll Face: This feature, in conjunction with Face Recognition above , is used to add facial data and enable facial recognition.

Face Detection: Face detection switch. When turned on, the location of faces will be marked on the screen; when turned off, no face detection will occur.

Face Recognition: Face recognition switch. When turned on, it compares the recorded facial data to identify people in the image; when turned off, it only detects faces and does not identify identities.