MQTT_Video_Stream

Introduction

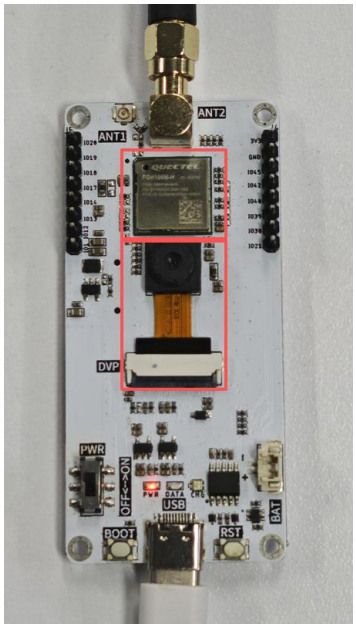
In this lesson, we will use the ESP32 WiFi-HaLow module to connect to the ThinkNode-G4 WiFi HaLow Gateway through WiFi HaLow wireless communication technology. This gateway can not only serve as an access node for the WiFi HaLow network, but also has the capability to connect to the Internet, enabling data communication between devices and cloud servers.

During system operation, the ESP32 WiFi-HaLow module will first connect to the WiFi HaLow network established by the ThinkNode-G4 WiFi HaLow Gateway. After the network connection is successful, the camera on the ESP32 module will continuously capture real-time image data and publish these image data to the MQTT server through the MQTT protocol.

MQTT is a lightweight IoT communication protocol that is highly suitable for data transmission between low-power devices and remote servers. In this lesson, the ESP32 WiFi-HaLow module acts as the MQTT Publisher, continuously sending camera image data to the MQTT server; while a computer or other clients subscribed to the corresponding MQTT Topic act as MQTT Subscribers, capable of receiving and displaying the camera images transmitted by the ESP32 module in real time.

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 to implement remote real-time transmission of camera images.



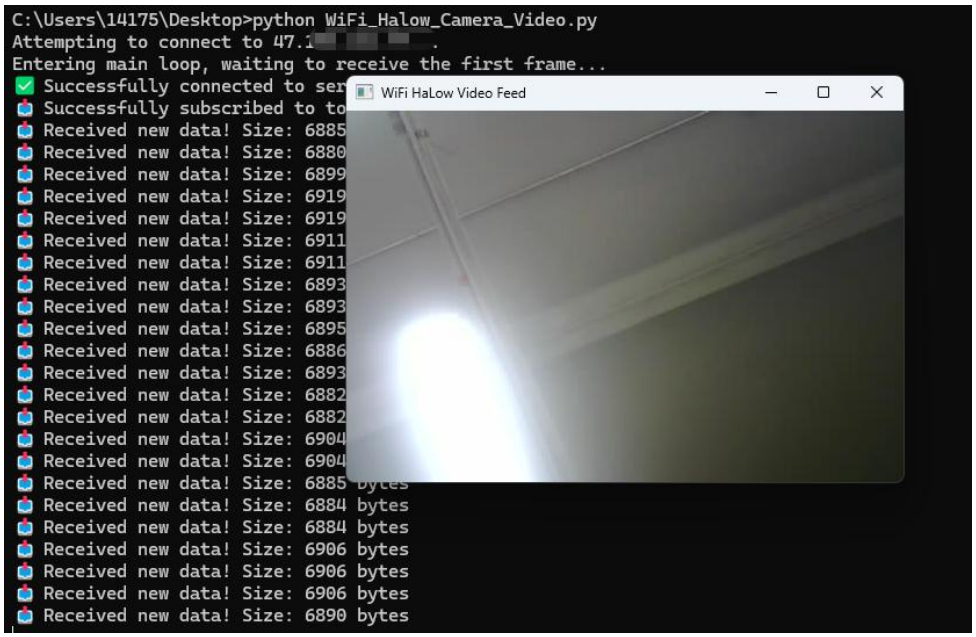
Wi-Fi HaLow (based on the IEEE 802.11ah standard) is a low-power, long-range wireless communication technology specially designed by the Wi-Fi Alliance for IoT scenarios. Compared with traditional Wi-Fi technology, it operates in the license-free Sub-1 GHz frequency band (mainly the 900 MHz band). This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

Since this project uses WiFi HaLow wireless communication, the ESP32 WiFi-HaLow module cannot directly connect to a regular router like standard WiFi devices. Instead, it needs to connect to the ThinkNode-G4 WiFi HaLow Gateway that supports WiFi HaLow technology.
This gateway can establish a WiFi HaLow network and provide Internet access capability, allowing the ESP32 WiFi-HaLow module to successfully access the MQTT server and complete data transmission.

The Wi-Fi HaLow Gateway is an innovative WiFi HaLow gateway developed by ELECROW, designed to meet the long-range and high-speed data transmission requirements of IoT applications. The gateway adopts Wi-Fi HaLow (IEEE 802.11ah) technology and operates in the license-free sub-1 GHz frequency band. Compared with traditional WiFi standards, it provides stronger penetration capability and wider coverage. It is equipped with powerful hardware, including advanced RF functions, a high-performance MCU, and flexible interfaces, enabling seamless integration with existing networks. It can be easily configured and upgraded via OTA through the Web UI, and supports a large number of simultaneous device connections, making it an excellent solution for smart manufacturing, smart cities, and other applications.

Operation Effect Diagram

After the camera captures real-time images, the ESP32 WiFi-HaLow module will upload and publish the image data to the MQTT server through the MQTT protocol. Any client device subscribed to the corresponding MQTT Topic, such as a computer or other IoT devices, can receive these image data in real time and view the camera images transmitted by the ESP32 WiFi-HaLow module.



Key Explanations

Next, let us set up the environment together so that we can remotely view the camera images of the ESP32 WiFi-HaLow module through MQTT.
First, let us configure the ThinkNode-G4 WiFi HaLow Gateway.

The purpose of configuring the ThinkNode-G4 WiFi HaLow Gateway is to establish a low-power, long-range, and strong wall-penetration wireless communication network based on WiFi HaLow, enabling the ESP32 WiFi-HaLow module to stably access the Internet and achieve data communication with the MQTT server.

In this project, the ESP32 WiFi-HaLow module will capture real-time image data through the camera and use the MQTT protocol to publish the image data to the MQTT server. The ThinkNode-G4 WiFi HaLow Gateway is responsible for establishing a communication bridge between the WiFi HaLow network and the regular Internet. On one hand, it provides WiFi HaLow wireless access for the ESP32 WiFi-HaLow module; on the other hand, it connects to a home router through an Ethernet cable or regular WiFi to achieve external network communication.

After the gateway configuration is completed, the ESP32 WiFi-HaLow module will be able to successfully connect to the MQTT server through the WiFi HaLow network and continuously upload real-time camera image data. At the same time, computers or other client devices subscribed to the corresponding MQTT Topic can also receive and view the camera image data in real time within the same network environment, thereby implementing long-distance wireless image monitoring and data transmission functions based on MQTT.

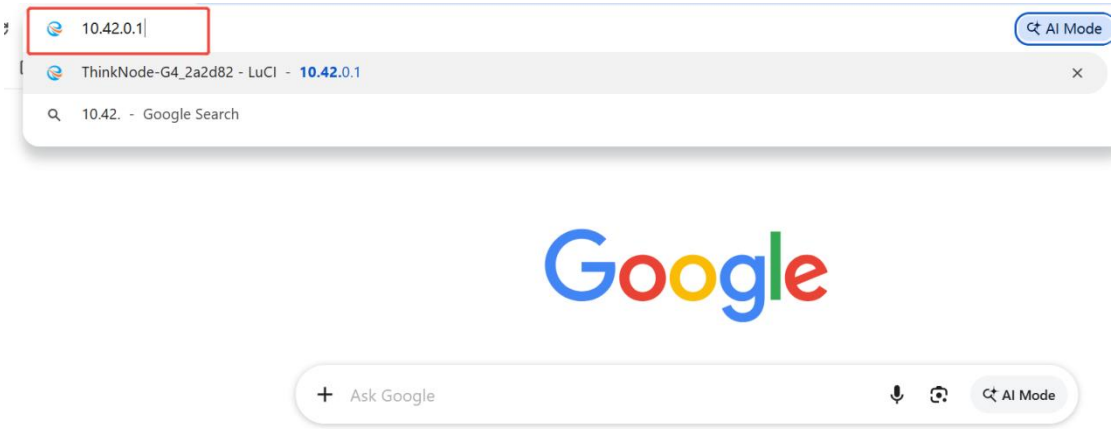
Configure the ThinkNode-G4 WiFi HaLow Gateway

First, power on the ThinkNode-G4 WiFi HaLow Gateway by plugging in the USB interface, then wait for at least 90 seconds. Proceed to the next step only after the LINK indicator on the gateway device starts flashing.

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

After the connection is successful, open a browser and enter 10.42.0.1.
When the ThinkNode-G4 WiFi HaLow Gateway leaves the factory, it enables a default configuration WiFi specifically for the initial setup;

10.42.0.1 is the default management IP of the gateway, and you can only access the backend after connecting to this WiFi.

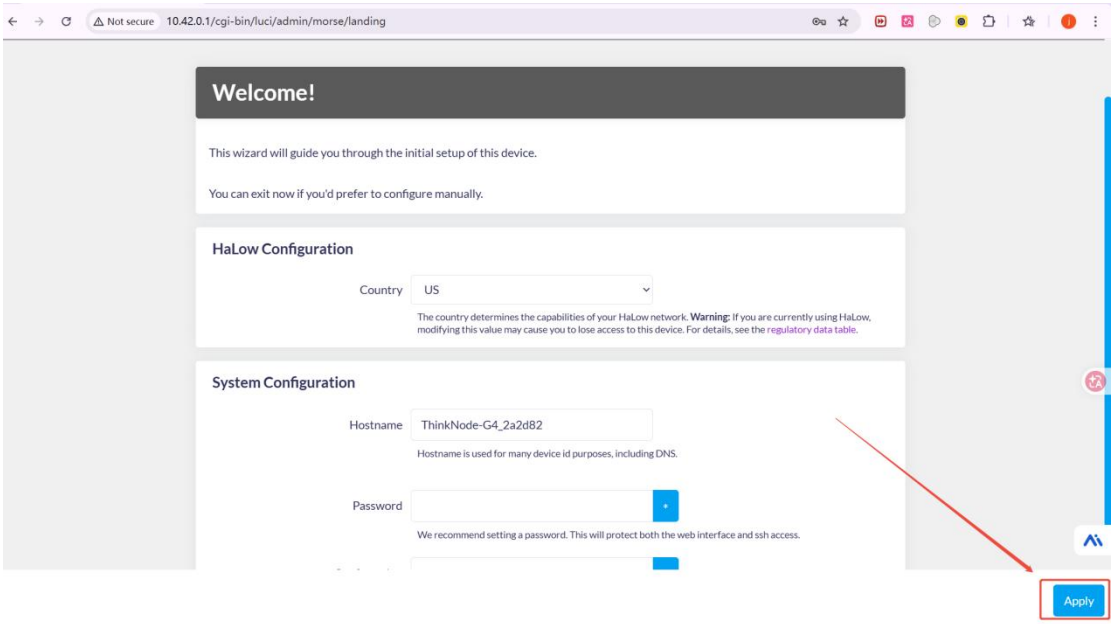


After entering, you will arrive at this page. At the beginning, no password is set, so you can enter directly for convenient first-time quick login.

Reminder: If you set a password later and want to restore the G4 to its initial state again, you need to press and hold the CFG button on the device for 10 seconds before you can enter the initial password-free state.

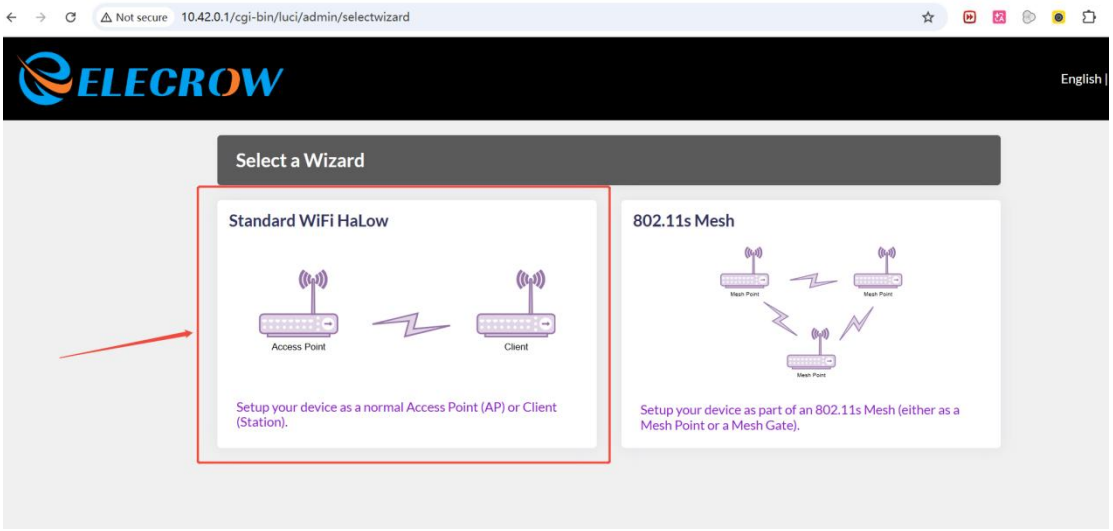
Now click Log In to enter the settings interface.

After entering the welcome page, skip it directly without modifying the country or hostname.



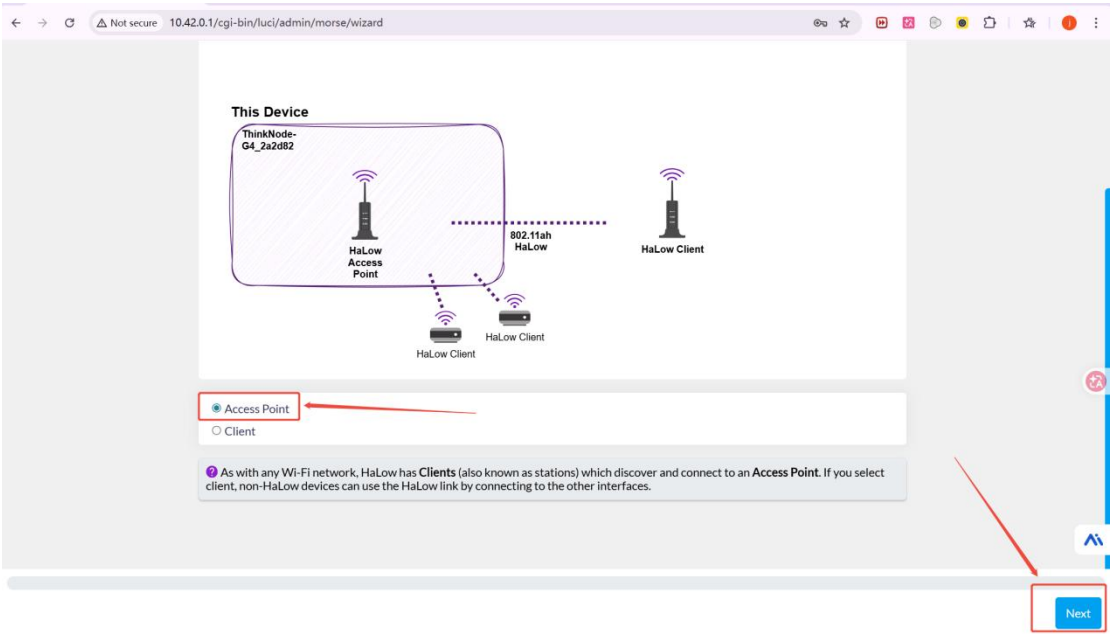
Select Standard WiFi HaLow mode and set it as Access Point (AP).

Standard WiFi HaLow: Standard point-to-point / local area network mode with the best stability and lowest latency, suitable for video transmission.



AP Mode: Allows the gateway to actively broadcast HaLow signals and act as the "central base station."

On the mode selection page, select Standard WiFi HaLow mode and set the gateway to Access Point (AP) mode. This mode is the optimal choice for HaLow modules and camera transmission. In AP mode, the gateway will actively broadcast 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 communication between the module and the gateway.

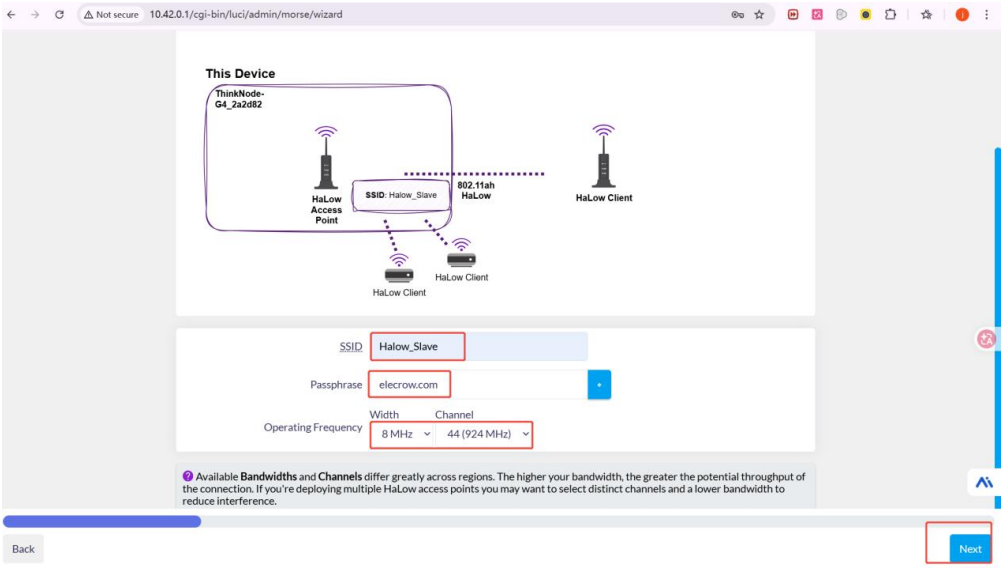


Your ESP32 WiFi-HaLow module is a client, so there must be a HaLow AP acting as the center in order for the module to connect to the gateway. This is the core prerequisite for allowing the camera to connect to the gateway.

Next, configure the WiFi HaLow account, password, channel, and bandwidth settings.

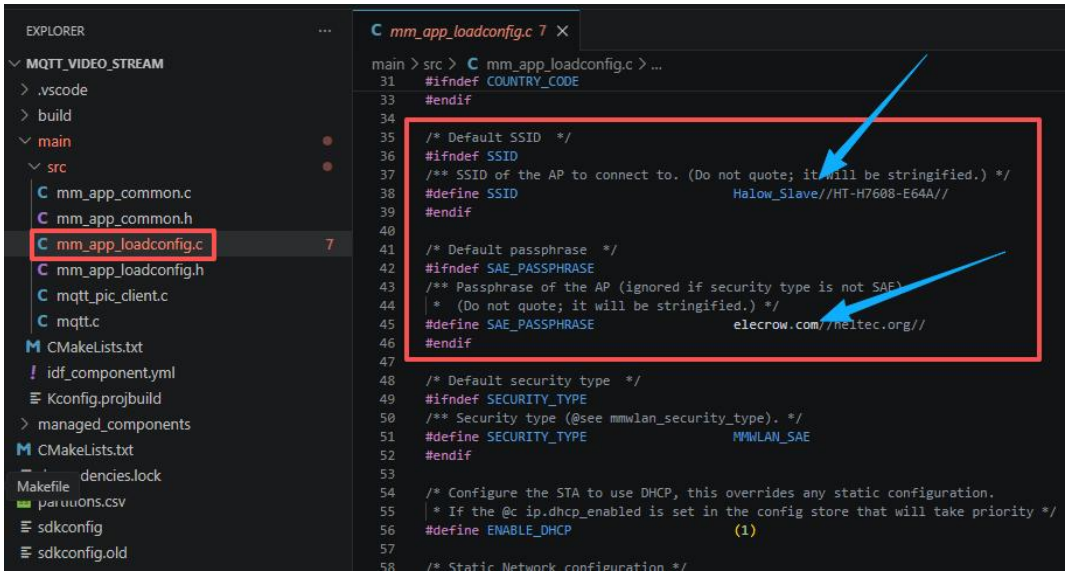
SSID / Password: Allows the ESP32 WiFi-HaLow module to accurately identify and securely connect to this network.

8MHz: A commonly used bandwidth in the HaLow standard that balances penetration capability, transmission distance, and data rate, and is sufficient for transmitting camera video.



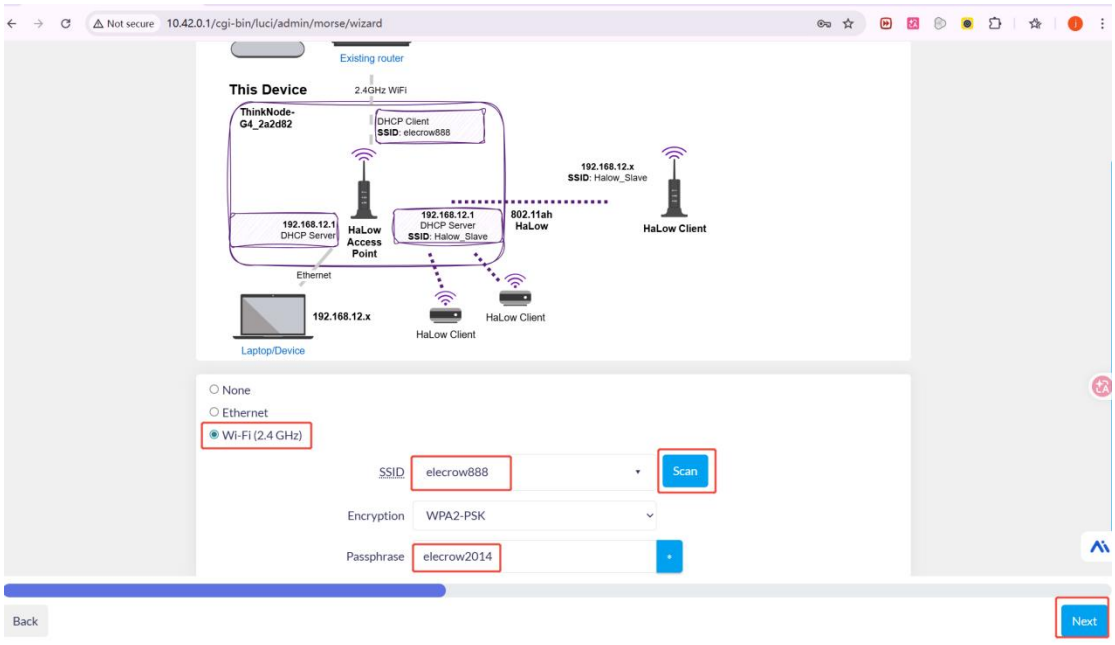
The parameters must be exactly the same as the code configuration in your ESP32 WiFi-HaLow module; otherwise, the module will not be able to connect to the gateway, and the camera will not be able to come online.

Code configuration:



Select WiFi (2.4GHz), enter your home router SSID: elecrow888, password: elecrow2014, and set the encryption method to WPA2-PSK.

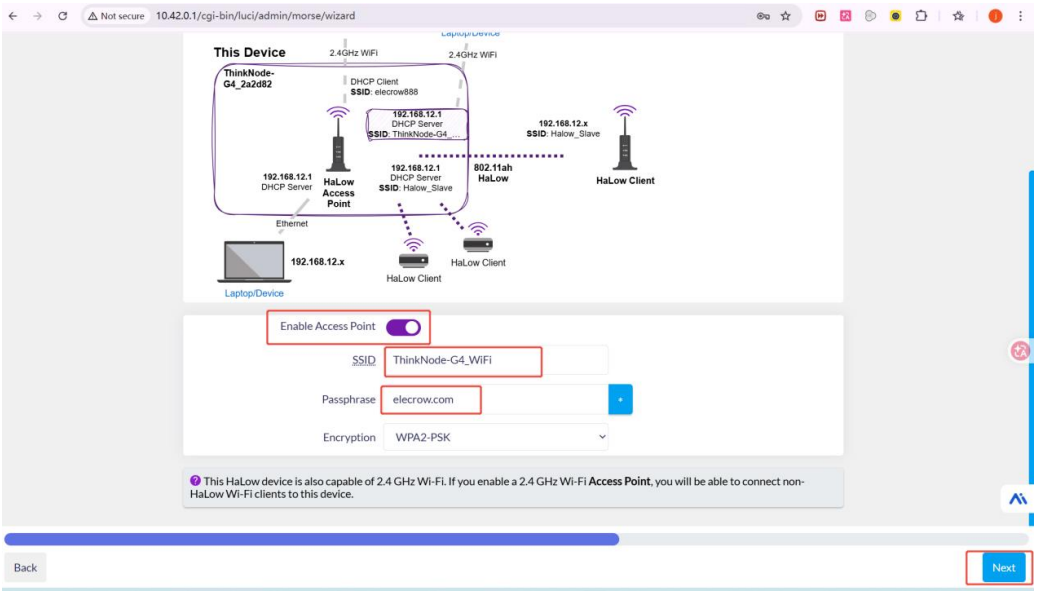
Reason: The G4 itself does not have direct Internet access capability. It must connect to your main router through 2.4G WiFi in order to obtain Internet access. This step gives the gateway "Internet connectivity."



Enable the 2.4G AP wireless hotspot of the ThinkNode-G4 WiFi HaLow Gateway itself to facilitate subsequent gateway management and configuration. Here, we set the AP SSID to "ThinkNode-G4_WiFi", the password to "elecrow.com", and select "WPA2-PSK" as the encryption method. After the configuration is completed, the gateway's default configuration WiFi will be disabled, and the new custom 2.4G AP will serve as a stable management entry point for future use, allowing computers to connect to the gateway for long-term network management and device debugging.

At the same time, this 2.4G AP also allows computers, mobile phones, and other MQTT client devices to be in the same local area network as the gateway, providing the network foundation for subsequent MQTT data communication. After the ESP32 WiFi-HaLow module connects to the ThinkNode-G4 WiFi HaLow Gateway through the WiFi HaLow network, the module will be able to access the MQTT server through the gateway and continuously upload real-time image data captured by the camera.

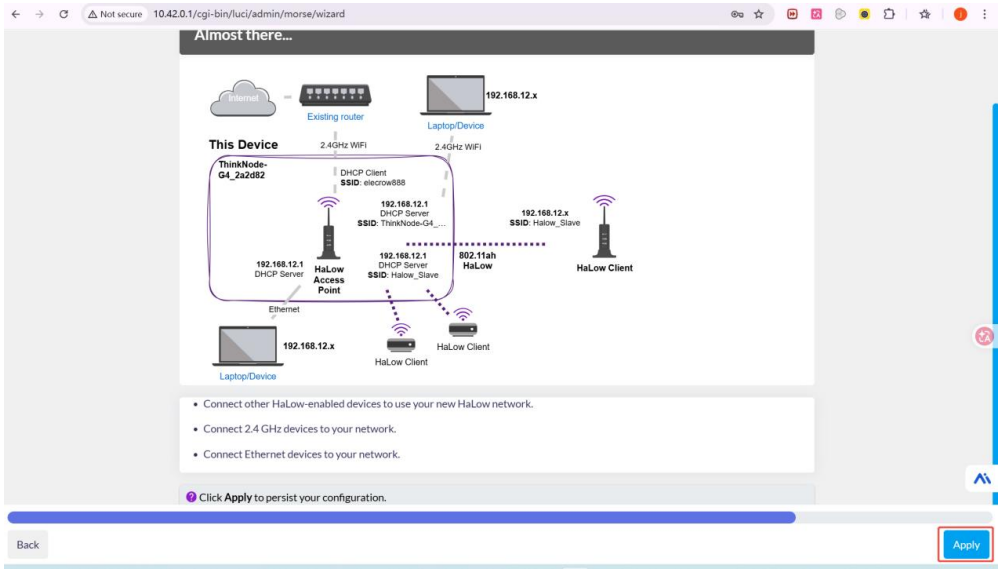
MQTT clients within the same network only need to subscribe to the corresponding MQTT topic to receive the camera image data sent by the ESP32 WiFi-HaLow module in real time, thereby achieving MQTT-based long-distance wireless real-time image transmission and monitoring functionality.

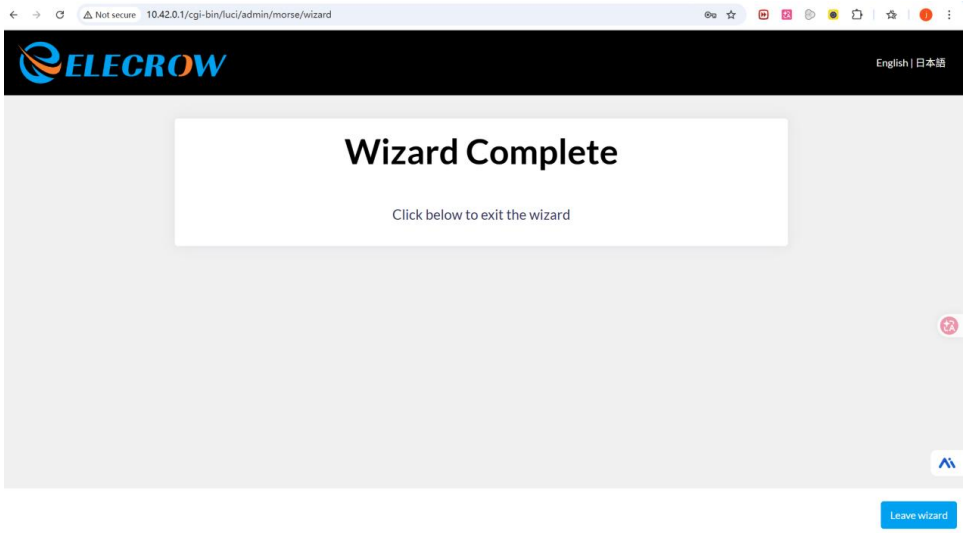


After completing all parameter settings, click Apply to save the configuration, and wait for the gateway to restart automatically until the LINK indicator turns solid green.

The Apply operation will write and activate all settings, including HaLow parameters, 2.4G Internet connection, and local AP configuration. A solid green LINK indicator means that the HaLow service, network connection, and local area network communication are all functioning properly, and the gateway has entered a ready

state for normal operation.





After the LINK indicator turns green, proceed to the next step, indicating that the ThinkNode-G4 WiFi HaLow Gateway has successfully completed network initialization and entered a manageable state.

At this point, disconnect the current old WiFi and reconnect to the newly created 2.4G management network: ThinkNode-G4_WiFi, with the password elecrow.com. After the connection is successful, open a browser and access the gateway's new management address 192.168.12.1 to enter the configuration interface.

The reason for this design is that after the gateway completes the configuration of the WiFi HaLow network and the upstream router or Internet connection, it will automatically switch to the internal management subnet 192.168.12.0/24, and 192.168.12.1 is the default management address within this local area network, used for subsequent device management and MQTT-related configuration operations.

In the MQTT architecture, the significance of this network structure is:

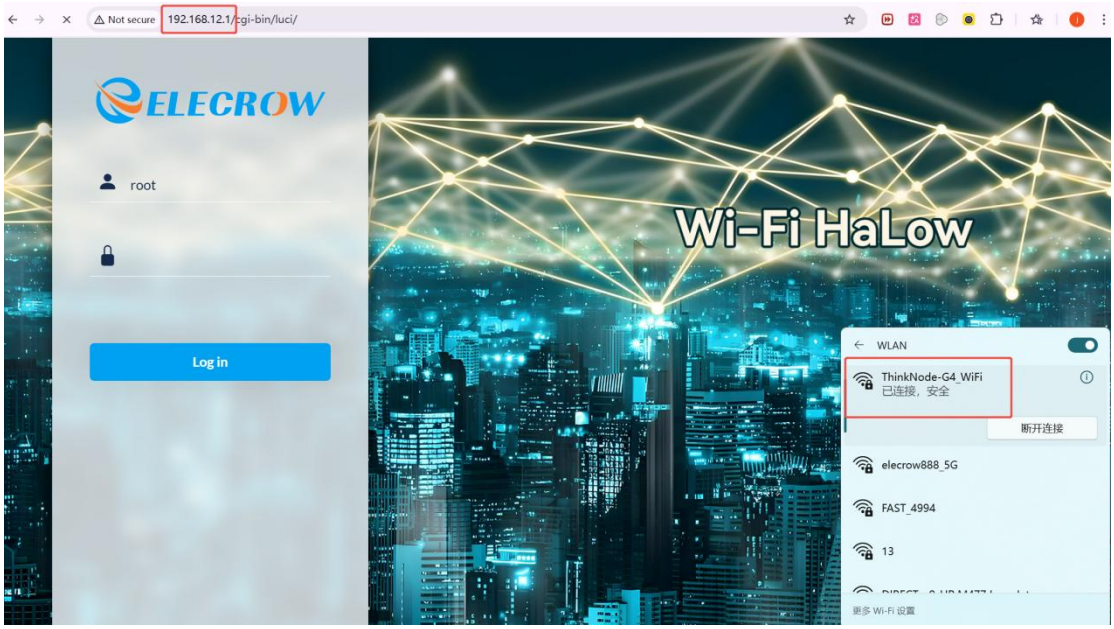
- Gateway as the **WiFi HaLow ↔ Internet bridging node**
- ESP32 WiFi-HaLow camera module as the **MQTT Publisher**
- MQTT server as **the data forwarding center (Broker)**
- Computer or mobile phone as **the MQTT Subscriber**

Therefore, when you enter the 192.168.12.1 management interface, you are essentially configuring a complete MQTT data channel, allowing the ESP32 WiFi-HaLow module to enter the same local area network through the gateway and further connect to the MQTT server to upload real-time camera data.

Ultimately, in this architecture:

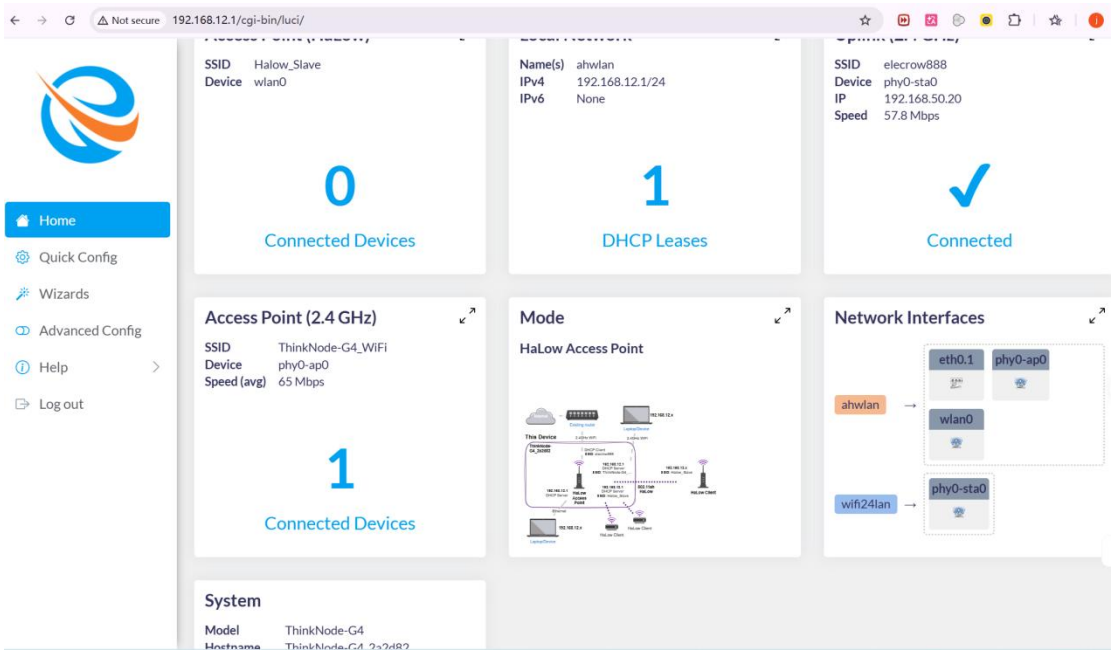
- The gateway is responsible for network connectivity and data forwarding
- The camera module is responsible for publishing image data
- The MQTT server is responsible for message distribution
- The client is responsible for subscribing to and displaying the real-time video stream

These three components (device, gateway, and client) work together within the same network link to achieve a stable MQTT real-time video data transmission system.



1. Your HaLow camera module → connects to the gateway's HaLow AP (Halow_Slave) in Client mode.
2. The module obtains an IP address in the 192.168.12.x subnet.
3. Your computer connects to the G4's 2.4G network (ThinkNode-G4_WiFi), which is also within the same subnet.

After your computer connects to this WiFi, enter the URL 192.168.12.1 in the browser, and you will be able to see the current management interface.

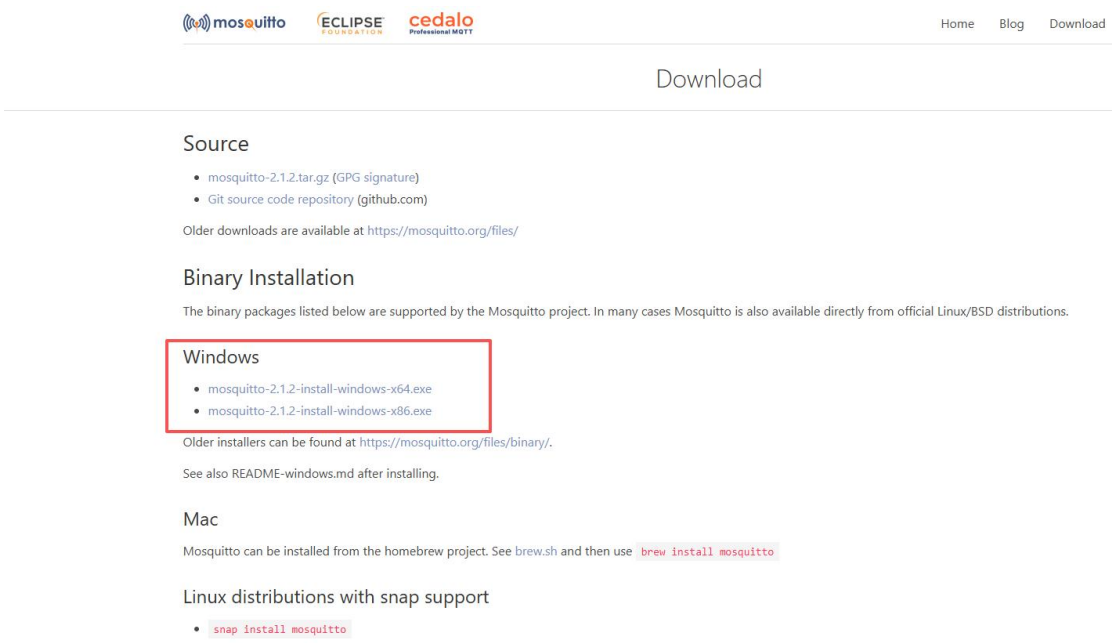


Configure MQTT

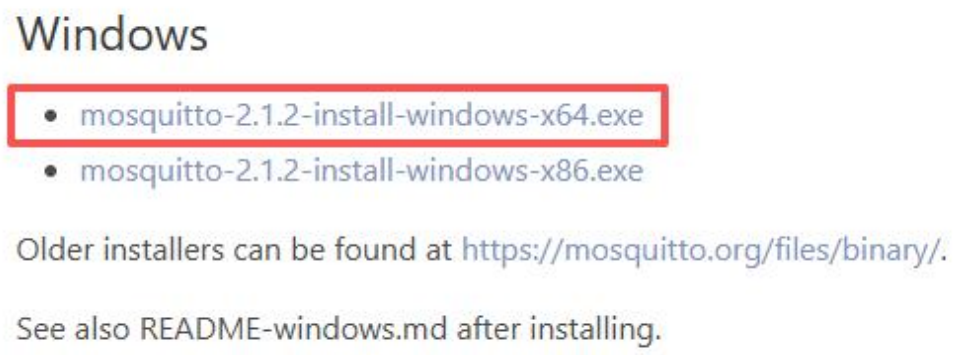
➢ Mosquitto Download

Visit the official Mosquitto download page: https://mosquitto.org/download

Under the Windows category, select the version suitable for your system.



Here, my computer uses a 64-bit operating system, so I selected the x64 version.



➢ Installation

Double-click to run the downloaded .exe installer.

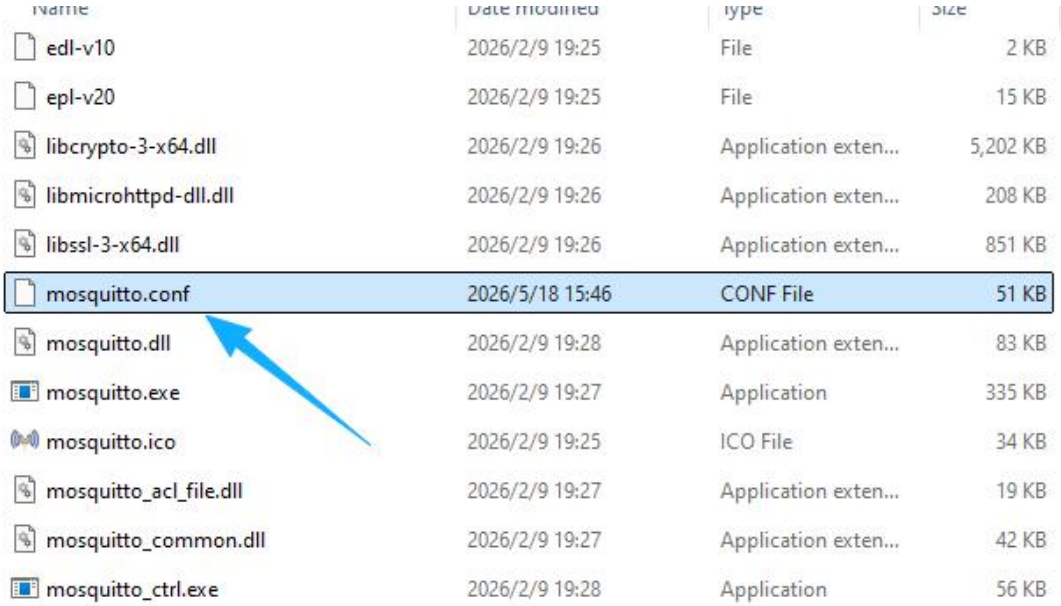
Follow the prompts and click "Next". When selecting components, the "**Service**" option will be checked by default (this will register Mosquitto as a Windows system service and enable automatic startup on boot).

Remember your installation path. By default, it is usually: C:\Program Files\mosquitto.

➢ Modify Configuration

Enter the installation directory: C:\Program Files\mosquitto

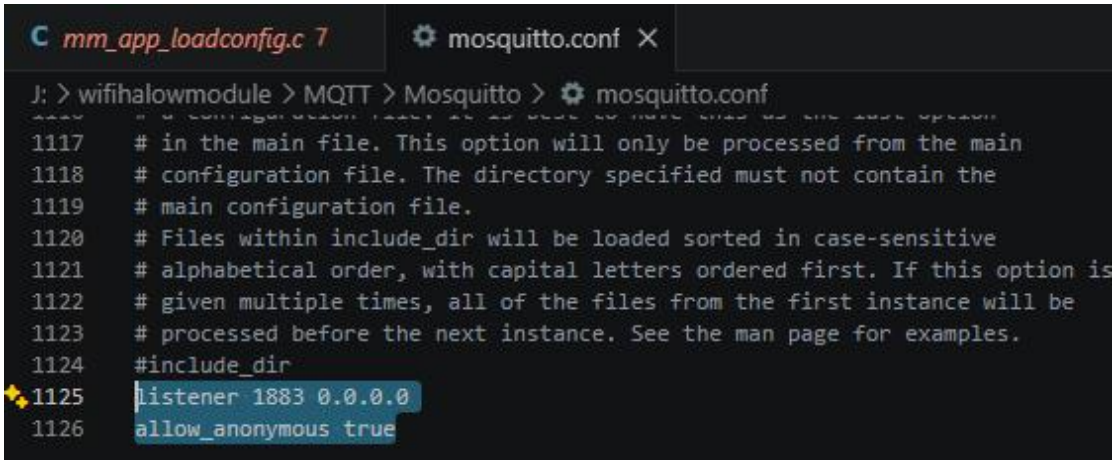
Find the mosquitto.conf file.



Open Notepad or another text editor as **administrator**.

At the end of the file, add the following two lines:

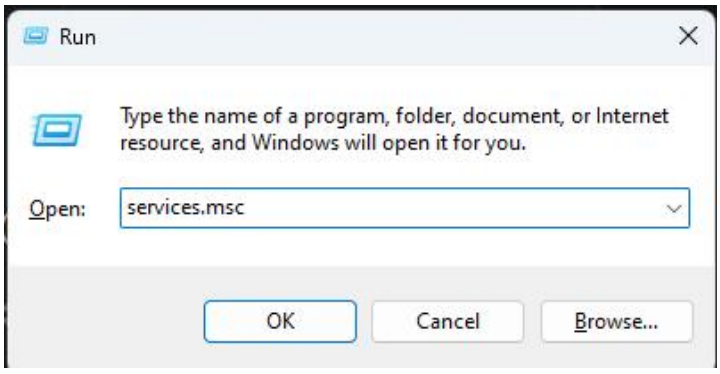
```
listener 1883 0.0.0.0

allow_anonymous true
```



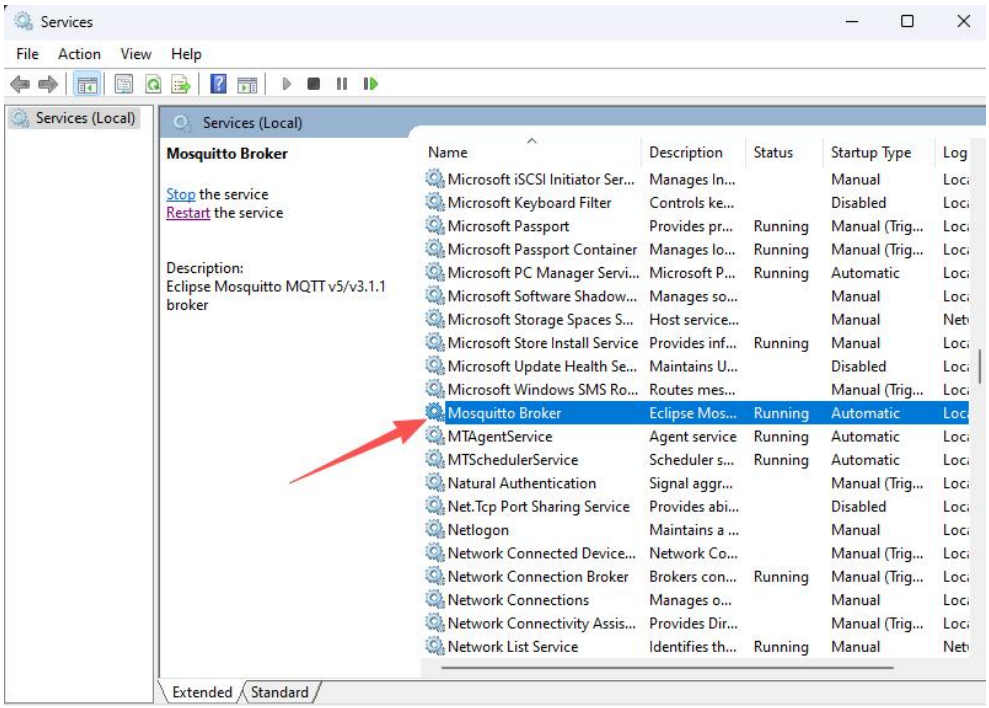
Save and close the file.

- Startup

Press the shortcut key Win + R, enter services.msc, and press Enter to open the Windows "Services" manager.



Find the service named **Mosquitto Broker**.

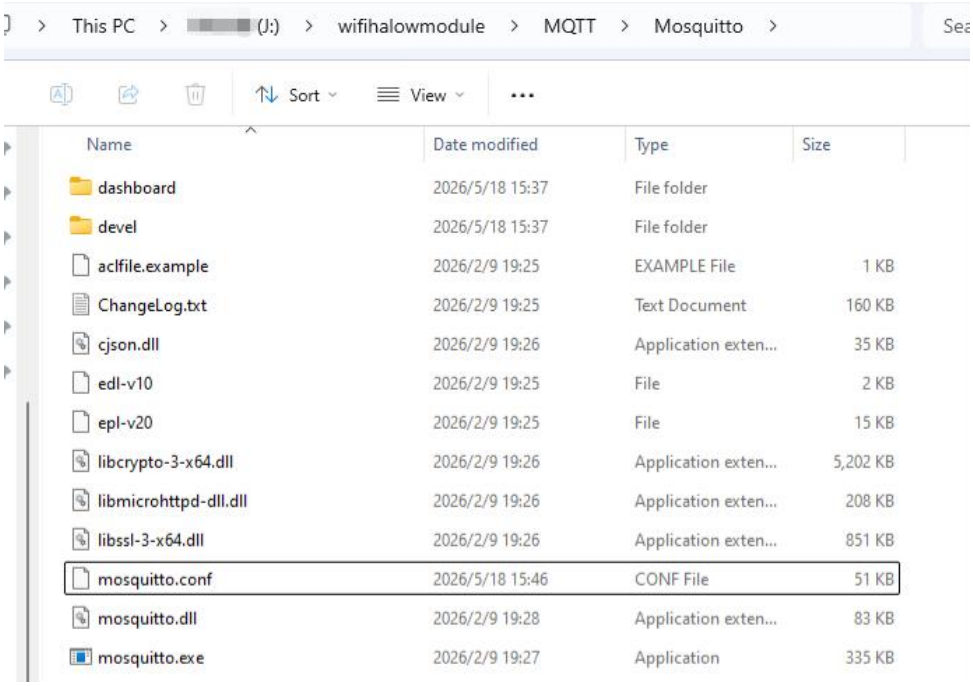


Right-click it and select "Restart" (or "Start") to ensure that the configuration you just modified takes effect.

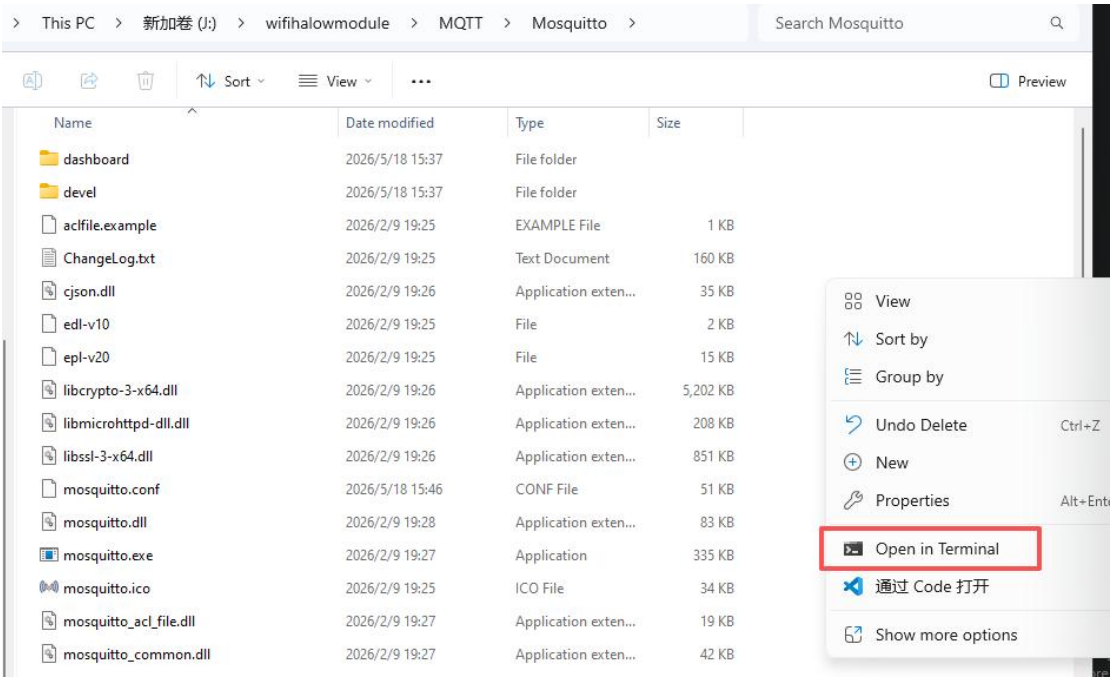
➢ Verification

You can open two CMD/PowerShell windows to test whether the service is working properly.

First, enter the file directory where you installed Mosquitto.



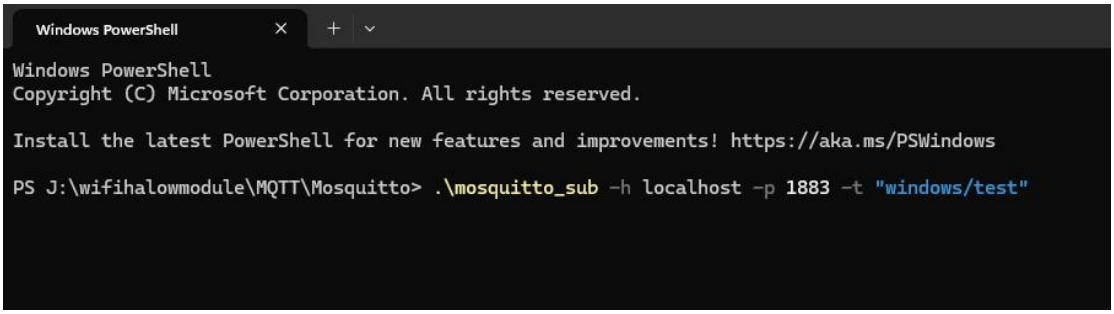
Then right-click within this file path and open two terminals.



Enter the following commands respectively

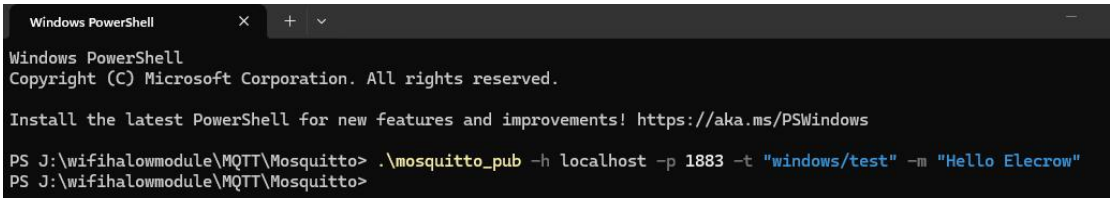
Window 1 (Subscriber)

.\mosquitto_sub -h localhost -p 1883 -t "windows/test"

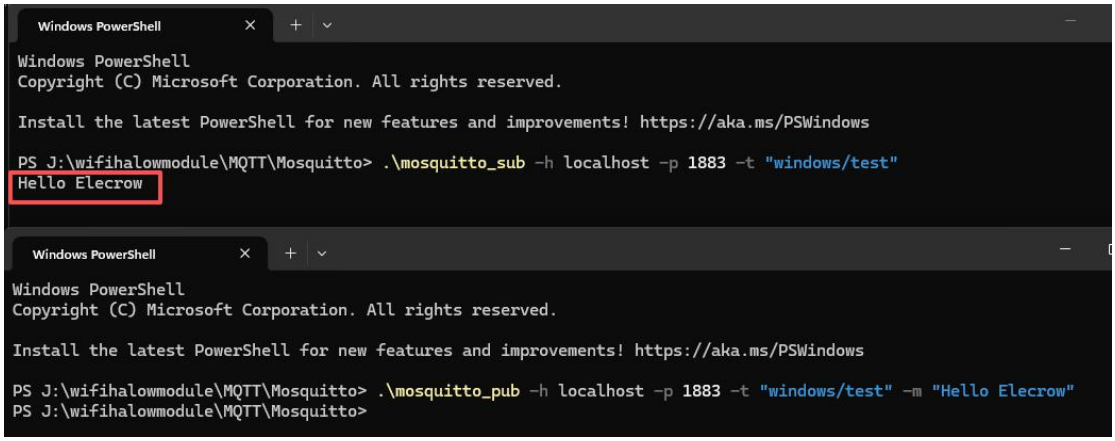


Window 2 (Publisher)

mosquitto_pub -h localhost -p 1883 -t "windows/test" -m "Hello Elecrow"



If Window 1 displays "Hello Elecrow", it means the deployment was successful!



The use of MQTT

Now that MQTT communication has been successfully established, let us further analyze the data interaction process between the ESP32 WiFi-HaLow module and the computer side. This section will mainly explain the program implementation of the ESP32 WiFi-HaLow module as the MQTT Publisher, as well as the code logic of the computer side as the MQTT Subscriber.

First, let us look at the program implementation of the ESP32 WiFi-HaLow module as the MQTT Publisher. (Here we will only explain the key parts of the code.)

Click the link below to download the publisher-side code for the ESP32 WiFi-HaLow module.

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

First, look at the **mqtt_pic_client.c** file in the project.

1. camera_publish_task(void *pvParameters)

This function is a FreeRTOS task function and serves as the core functional implementation unit of the project, specifically responsible for continuously capturing camera images and publishing them through MQTT;

After the task starts, it continuously runs in a loop. It first calls the camera interface to capture a frame image and checks whether the image format is JPEG (only JPEG format upload is supported)

```
48      camera_fb_t *frame = esp_camera_fb_get();
49
50      if (!frame)
51      {
52          ESP_LOGE(TAG, "Camera capture failed");
53          vTaskDelay(pdMS_TO_TICKS(PUBLISH_INTERVAL_MS));
54          continue;
55      }
56
57      if (frame->format == PIXFORMAT_JPEG)
58      {
59          image_data_buf = frame->buf;
60          image_data_buf_len = frame->len;
61      }
```

After verification passes, it calls the MQTT publish interface to send the image data to the specified topic. After publishing is completed, it releases the camera frame buffer

```
67      if (res == ESP_OK)
68      {
69          int msg_id = esp_mqtt_client_publish(client, CAMERA_TOPIC, (const char *)image_data_buf,
70                                               image_data_buf_len, 0, 0);
71          ESP_LOGI(TAG, "Published camera frame, topic=%s, msg_id=%d, size=%u bytes",
72                   CAMERA_TOPIC, msg_id, (unsigned int)frame->len);
73      }
74      else
75      {
76          ESP_LOGW(TAG, "Failed to capture image");
77      }
78
79      esp_camera_fb_return(frame);
80      vTaskDelay(pdMS_TO_TICKS(PUBLISH_INTERVAL_MS));
81  }
82  }
```

Finally, it delays according to the configured time interval and waits for the next capture and publishing cycle. The task parameter receives the MQTT client handle for data uploading.

2. mqtt_event_handler(void *handler_args, esp_event_base_t base, int32_t event_id, void *event_data)

This function is the MQTT client event callback handler. It is automatically called by the ESP-IDF MQTT library and is used to handle all MQTT-related events;

The function parses different event IDs to handle various scenarios such as successful MQTT connection, disconnection, topic subscription/unsubscription, message publishing completion, receiving subscribed data, and error occurrences. The core logic is to create the camera publishing task only once after the MQTT connection is successfully established, while other events are only used for log printing or error information parsing. It serves as the core interaction layer between the MQTT client and the application layer.

3. mqtt_app_start(void)

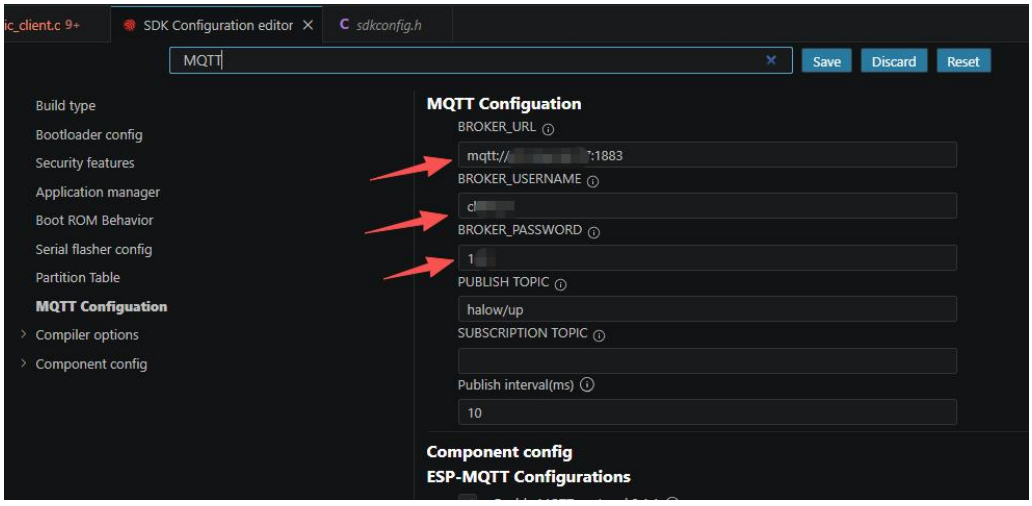
This function is the initialization and startup entry function for the MQTT client. It is responsible for configuring connection parameters such as the MQTT server address, username, and password.

```c
void mqtt_app_start(void)
{
    esp_mqtt_client_config_t mqtt_cfg = {
        .broker.address.uri = CONFIG_BROKER_URL,
        .credentials.username = CONFIG_BROKER_USERNAME,
        .credentials.authentication.password = CONFIG_BROKER_PASSWORD,
    };

    esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);

    /* The last argument may be used to pass data to the event handler, in this example
     * mqtt_event_handler */
    esp_mqtt_client_register_event(client, ESP_EVENT_ANY_ID, mqtt_event_handler, NULL);
    esp_mqtt_client_start(client);
}
```

It calls the ESP-IDF MQTT library to initialize the client handle, registers the above event callback function to handle all MQTT events, and finally starts the MQTT client to connect to the server. It is the startup entry point of the entire MQTT functionality. Once called by the main function, it completes the initialization and connection of the MQTT service.

Here you need to open menuconfig to configure connection parameters such as the MQTT server address, username, and password.

After opening it, search for MQTT and configure the related MQTT parameters here.



These MQTT configuration items shown in the ESP-IDF configuration interface are used to allow the camera client on your ESP32 WiFi-HaLow module to correctly connect and communicate with the specified public MQTT server.

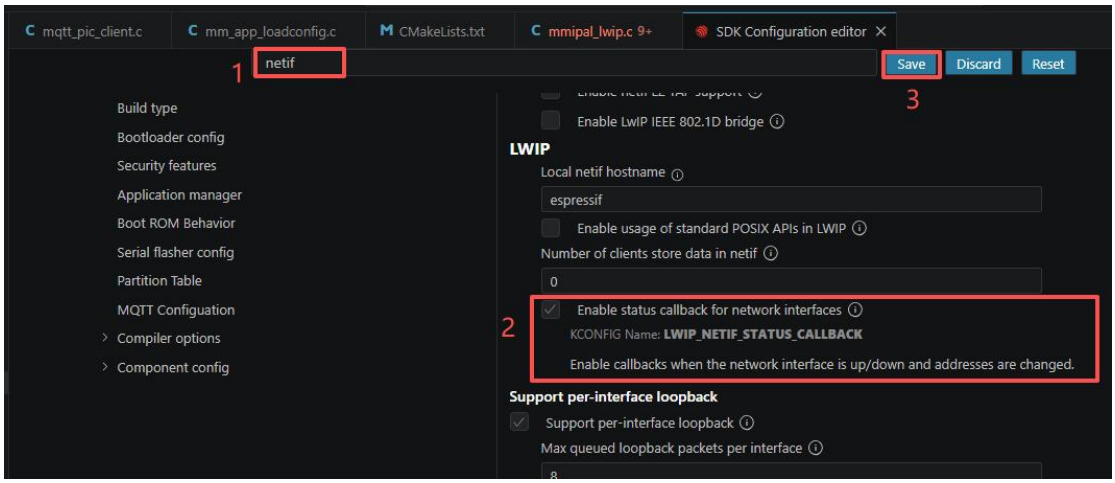
Among them, BROKER_URL specifies the public IP address and port of the server, while BROKER_USERNAME and BROKER_PASSWORD are used for authentication;

PUBLISH TOPIC (halow/up) defines the target topic for image data publishing, while Publish interval(ms) (10ms) controls the image publishing frequency. These parameters will be referenced by your code during compilation and are the foundation for the normal operation of the camera image uploading function. In addition, this server is a public server, and your device data will be transmitted directly to the server through the Internet.

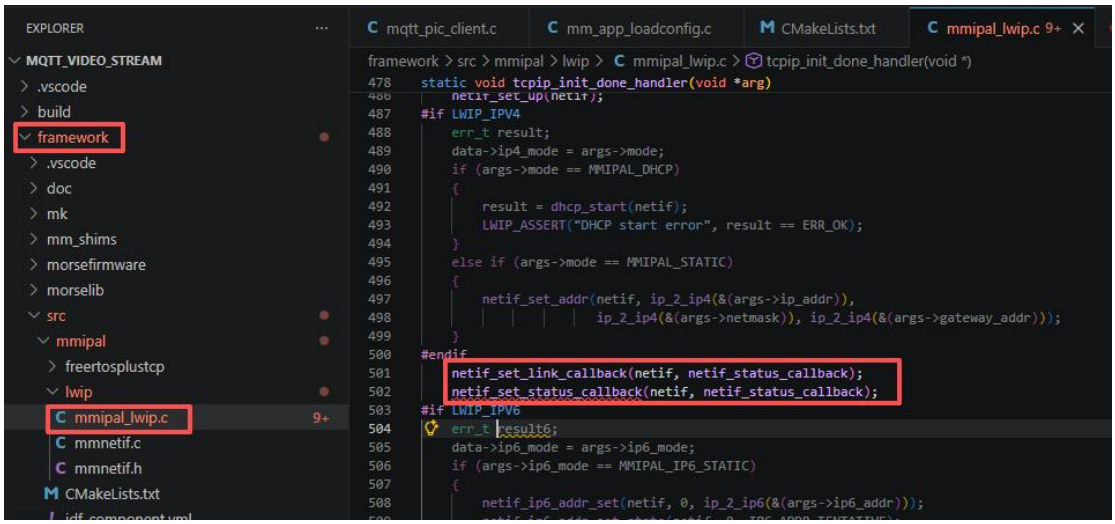
Here everyone needs to prepare and configure a public MQTT server by themselves. After completing the MQTT service application and deployment, you need to fill in the corresponding server information in the program configuration, including the MQTT server address (IP or domain name), port number, client ID, username, password, and the MQTT topic to be used.

Only when these configuration parameters are correctly filled in can the ESP32 WiFi-HaLow module successfully connect to the MQTT server and stably publish the real-time image data captured by the camera, allowing clients subscribed to the corresponding topic to correctly receive and display the image content.

Next, we also need to enable this in menuconfig.

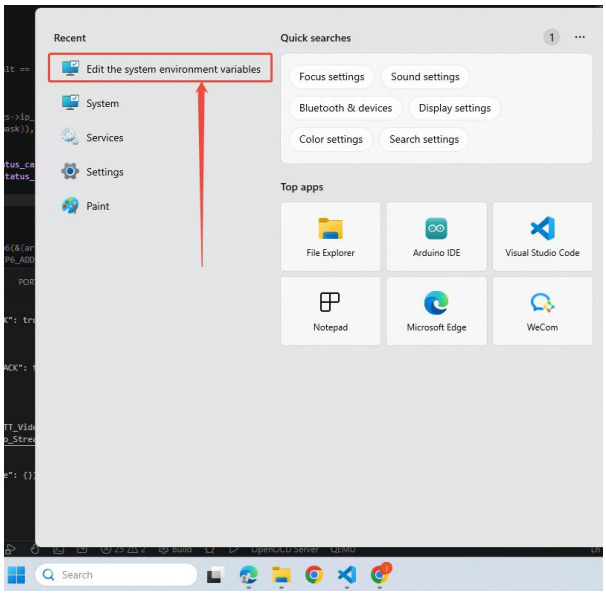


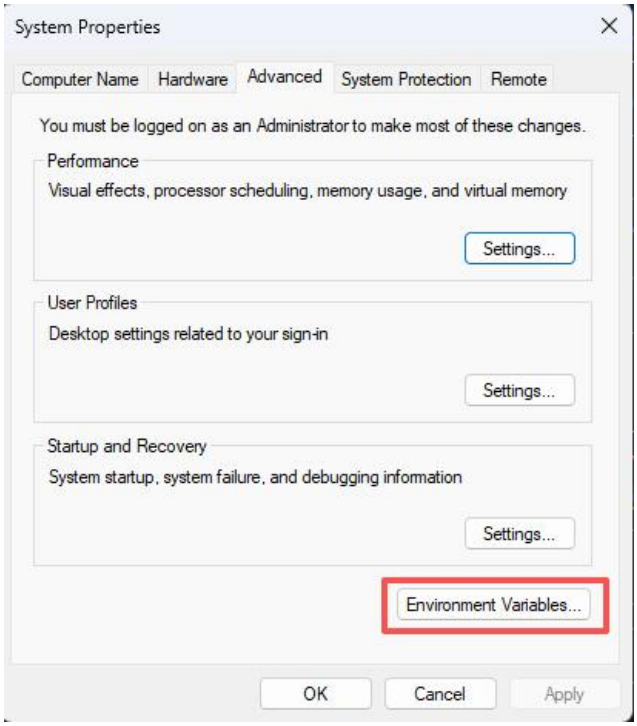
Otherwise, errors will occur in the code inside the framework directory of the project.



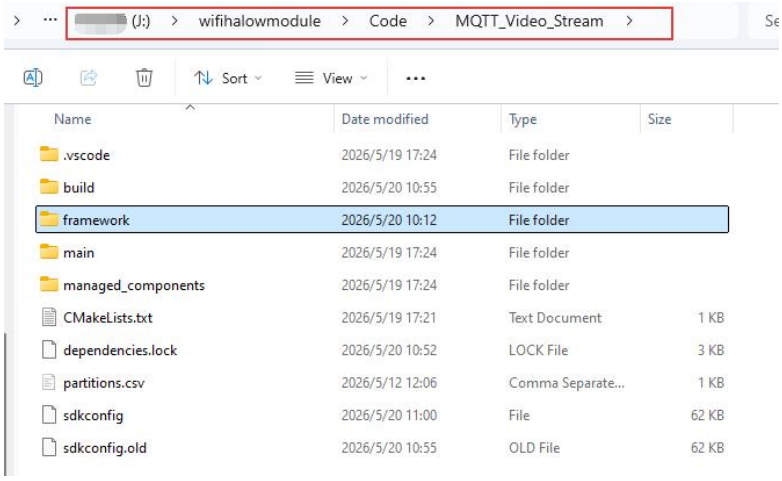
The framework folder in the project is the core MM-IoT-SDK framework directory provided by Morse Micro. It provides complete support for the underlying hardware, protocol stack, and system adaptation for your ESP32 + HaLow WiFi module project, and serves as the foundation that allows the project to run properly.

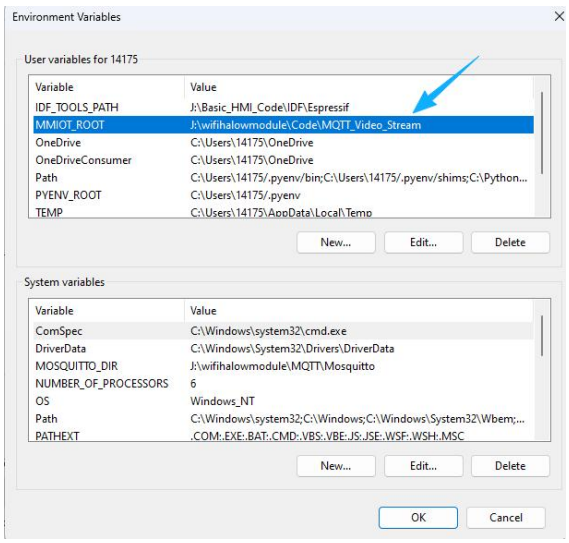
Therefore, we need to add the path of this framework to the environment variables on the computer.



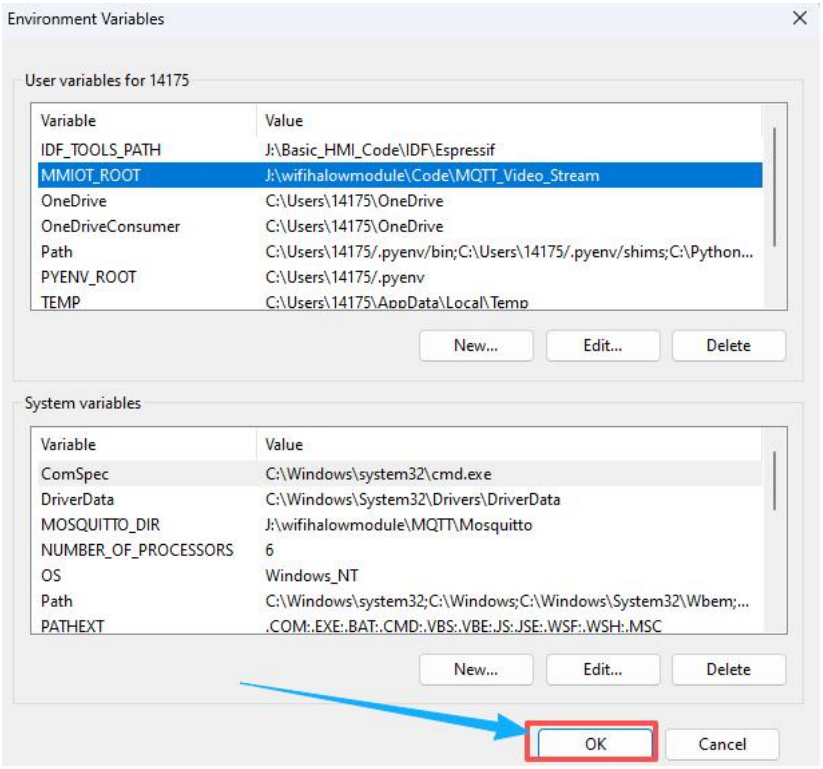


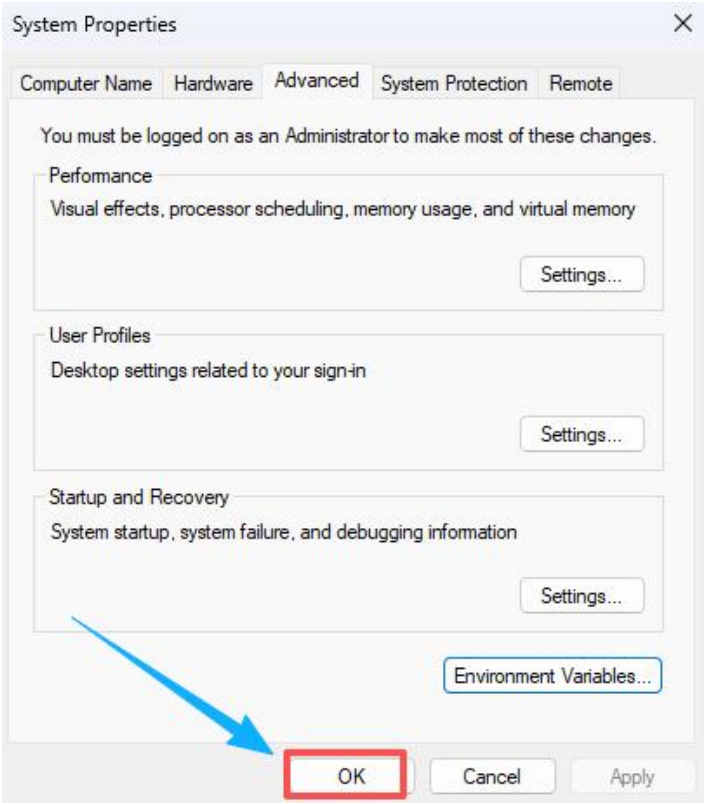
Then add the path of the framework in the upper section.





After completing the addition, remember to save the settings step by step.





After completing the system environment variable configuration, you need to restart the computer so that the operating system can reload the updated environment variable information. This ensures that the computer can correctly recognize the

new system configuration and avoids program runtime errors or related tools failing to work properly due to environment variables not taking effect in time.

Its core function can be summarized in one sentence:
This framework folder is the underlying dependency and runtime support layer of your project. It contains Morse Micro's HaLow WiFi protocol stack, hardware abstraction layer, system adaptation code, and compilation/build scripts.

It is responsible for communicating with your WiFi HaLow module hardware, handling the low-power WiFi protocol stack, and providing stable API interfaces to the upper-layer main directory (your MQTT camera code), allowing you to directly develop application functions without writing drivers and protocols from scratch.

Breakdown of the functions of the key subdirectories:
morselib / mm_shims: Morse Micro's core libraries and system adaptation layer, encapsulating the HaLow protocol stack, WLAN drivers, and operating system (FreeRTOS/ESP-IDF) adaptation interfaces. Functions such as app_wlan_init() and app_wlan_start() in your code call the APIs here to connect to the HaLow network.

morsefirmware: Contains the firmware files for the HaLow module. These files are packaged into the final firmware image during compilation to provide underlying runtime support for the WiFi module.

src: The core source code directory of the framework, containing low-level implementations such as protocol stacks, hardware drivers, and event handling.

mk: Makefile/CMake scripts related to compilation and building. It integrates with the ESP-IDF build system to ensure that the framework code can be correctly compiled and linked into your project.

doc: Documentation and instruction files related to the framework, including API references, development guides, and more.

In simple terms, the camera + MQTT code in the main directory is the "application layer," while the framework directory is the "foundation" that makes everything run properly. It is responsible for handling low-level complex tasks such as WiFi connections, hardware drivers, and protocol stacks, allowing you to focus on implementing the business logic for image capture and MQTT publishing.

At this point, the configuration in menuconfig is completed. Remember to save the configuration.

Next, look at the **mqtt.c** file in the project.

1. init_camera(void)

This function is the ESP32 camera hardware initialization function. It is responsible for configuring all core parameters of the camera, including GPIO pins, clock frequency, image format (JPEG), resolution, buffer size, and more. It calls the ESP-IDF camera driver interface to complete the hardware initialization; after successful initialization, it obtains the camera sensor object and sets parameters such as image flipping and saturation according to different sensor models (OV3660, OV2640, etc.). Finally, it returns the initialization status, which is the prerequisite for the camera to function properly.

2. app_main(void)

This function is the main entry function of the ESP32 program (equivalent to the standard C main function) and serves as the execution starting point of the entire project; the function sequentially completes NVS flash initialization (used to store WiFi/MQTT configurations), creates the default event loop, initializes and starts the WLAN WiFi connection, initializes the camera hardware (and exits directly if initialization fails), and finally starts the MQTT client, connecting the three core modules: WiFi, camera, and MQTT, to complete the overall system initialization and functional startup.

mqtt_pic_client.c focuses on MQTT communication + camera image uploading, including four major functions: error handling, core tasks, event callbacks, and MQTT startup;

mqtt.c focuses on system initialization, including two major functions: camera hardware initialization and the main program entry;

Overall process: The program starts running from the app_main main entry point. It first completes NVS flash initialization and system event loop creation, then calls the HaLow WiFi-related interfaces to initialize the WiFi module and establish the network connection. After the network connection is successful, it performs camera hardware initialization, configures parameters such as pins, image format, and resolution, and starts the camera.

Then it starts the MQTT client and allows it to automatically connect to the public MQTT server. After the MQTT client successfully connects to the server, a FreeRTOS task for camera image capture and publishing is created and started in the event callback.

This task continuously runs in a loop at the configured time interval: obtaining a frame of JPEG image data from the camera, publishing the image data to the specified halow/up topic through the MQTT protocol, releasing the camera buffer after publishing, and then delaying before waiting for the next capture and upload cycle. Throughout the entire process, the images captured by the ESP32 camera are continuously uploaded to the public MQTT server through the HaLow WiFi network.

Next, let us take a look at the implementation of the PC side as the MQTT Subscriber. This section will mainly explain how the computer connects to the MQTT server, subscribes to the specified MQTT topic, and receives and processes the camera image data published by the ESP32 WiFi-HaLow module.

Please click the link below to download the example code for the PC-side MQTT Subscriber.

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

This Python code is the receiving side of the ESP32 camera image transmission system. Its core function is to act as an MQTT client to connect to the public server you configured, subscribe to the image topic published by the ESP32, receive binary JPEG data in real time, decode it, display it as a video stream using OpenCV, and provide a safe exit mechanism.

This section first imports the dependency libraries required for the project: paho.mqtt is used to implement MQTT client communication, numpy is used to convert the received binary data into array format, cv2 (OpenCV) is used for image decoding and window display, and sys is used for safe program termination;

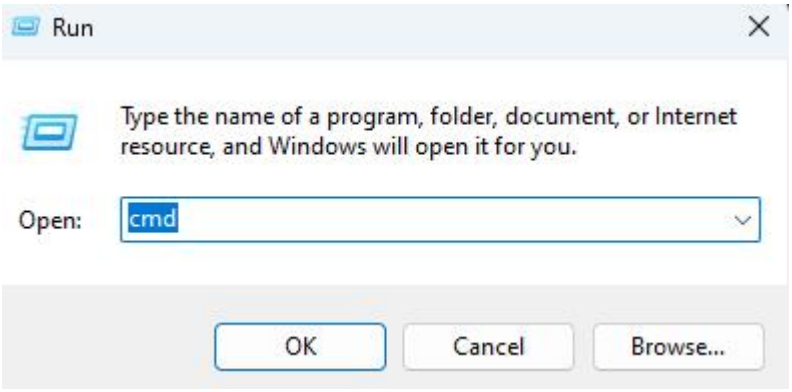
```
1  import paho.mqtt.client as mqtt
2  from paho.mqtt.enums import CallbackAPIVersion
3  import numpy as np
4  import cv2
5  import sys
```

If Python is not installed on your computer, please go to the official Python website to download and install it.

Python official website: python.org

- Go to the official Python website to download the installation package. It is recommended to choose Python 3.10 or above stable versions for the best compatibility.
- Launch the installer. On the first page, make sure to check Add python.exe to PATH so that the system environment variables can be automatically configured without manual setup, then click to complete the installation using the default installation path.
- After the installation is completed, open the cmd command line and enter python --version and pip --version. If the version numbers are displayed correctly, it means the installation and configuration were successful.

Here you need to press Win + R to open CMD.



Then install them one by one.

```
pip install paho-mqtt
pip install numpy
pip install opencv-python
```

paho-mqtt： Used to implement MQTT client connection, topic subscription, and receiving server data.

numpy： Used to convert the binary image data stream into array format.

opencv-python： Used to perform JPEG image decoding and real-time local window image display.

Then the MQTT server address, subscription topic, username, and password are defined. These configurations correspond exactly to the publisher side on your ESP32, ensuring that the server can be connected correctly and data can be received.

```
7   # --- Configuration Area ---
8   MQTT_BROKER = '                77"  # Replace with the actual MQTT server address
9   MQTT_TOPIC = "halow/up"
10  MQTT_USER = "cl      "        # Unified use of available client_2
11  MQTT_PASSWORD = "123"
```

Here three global variables are defined: WINDOW_NAME is the title of the OpenCV display window; latest_img is used to cache the latest received image frame to avoid lag caused by frequent window refreshing; window_created is a status flag used to prevent window state detection from being called before the window is fully created, thereby preventing program errors.

```
13  WINDOW_NAME = "WiFi HaLow Video Feed"
14  latest_img = None
15  window_created = False       # New: Mark whether the window has actually been created
16
```

This is the callback function triggered after the MQTT client connects to the server. It will be automatically triggered once the client successfully establishes a connection with the server: it first checks the connection status code. If the status code is 0, it means the connection is successful, and it will immediately subscribe to the image topic halow/up published by your ESP32 device to prepare for receiving data; if the

connection fails, it will print the error code to help troubleshoot network or configuration issues.

```python
def on_connect(client, userdata, flags, rc, properties=None):
    status_code = rc.value if hasattr(rc, 'value') else rc
    if status_code == 0:
        print(f"✅ Successfully connected to server: {MQTT_BROKER}")
        client.subscribe(MQTT_TOPIC)
        print(f"📩 Successfully subscribed to topic: {MQTT_TOPIC}, continuously listening for data...")
    else:
        print(f"❌ Failed to connect to server, error code: {status_code}")
```

This is the core data processing function of the entire receiver side. It is automatically executed whenever the subscribed topic receives a new message: it first uses np.frombuffer to convert the received binary data into a numpy array, and then uses cv2.imdecode to decode the JPEG-format array into an OpenCV-recognizable image format; if decoding succeeds, the image will be stored in the global variable latest_img for display in the main loop; if decoding fails (for example, due to an incomplete data packet), a warning message will be printed to prevent the program from crashing.

```python
def on_message(client, userdata, msg):
    global latest_img
    print(f"📩 Received new data! Size: {len(msg.payload)} bytes")
    try:
        nparr = np.frombuffer(msg.payload, np.uint8)
        img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
        if img is not None:
            latest_img = img
        else:
            print("⚠️ Data received but OpenCV failed to decode (packet might be incomplete)")
    except Exception as e:
        print(f"❌ Exception occurred during data processing: {e}")
```

This part completes the creation and configuration of the MQTT client: it first creates a client instance using the new callback API, binds the previously defined on_connect and on_message callback functions, and then sets the server username and password; afterward, it attempts to connect to port 1883 of the public MQTT server. After the connection is successful, client.loop_start() is called to start a background thread that continuously listens for server messages, preventing the display logic in the main loop from being blocked.

```python
# Create client
client = mqtt.Client(callback_api_version=CallbackAPIVersion.VERSION2)
client.on_connect = on_connect
client.on_message = on_message
client.username_pw_set(MQTT_USER, MQTT_PASSWORD)

try:
    print(f"Attempting to connect to {MQTT_BROKER}...")
    client.connect(MQTT_BROKER, 1883, 60)
    client.loop_start()
```

The main loop is the core of the program's display and control logic: when latest_img is not empty, cv2.imshow is called to display the latest image and mark that the window has been created; meanwhile, cv2.waitKey(30) refreshes the window every 30 milliseconds and checks keyboard input. Pressing the q key will exit the program; after the window has been created, the program will also detect whether the

window has been manually closed. If the window is closed, the loop will automatically exit to prevent the program from freezing in the background.

```
    print("Entering main loop, waiting to receive the first frame...")
    while True:
        if latest_img is not None:
            cv2.imshow(WINDOW_NAME, latest_img)
            window_created = True # Mark window as existing only after successfully displaying the image

        # 1. Keyboard exit detection
        if cv2.waitKey(30) & 0xFF == ord('q'):
            print("\nDetected 'q' key press, exiting...")
            break

        # 2. Core improvement: Detect X button only after the window has been created
        if window_created:
            try:
                if cv2.getWindowProperty(WINDOW_NAME, cv2.WND_PROP_VISIBLE) < 1:
                    print("\nDetected window manually closed, exiting...")
                    break
            except cv2.error:
                # Catch potential unexpected internal OpenCV errors
                pass
```

Finally comes the exception handling and resource release section: if an uncaught exception occurs during program execution, an error message will be printed; regardless of whether the program exits normally or terminates unexpectedly, the finally block will always execute to stop the MQTT background thread, disconnect from the server, and destroy the OpenCV window, ensuring that all resources are safely released and preventing memory leaks or port occupation.

```
except Exception as e:
    print(f"An error occurred while running the program: {e}")

finally:
    print("Safely releasing resources and exiting...")
    client.loop_stop()
    client.disconnect()
    cv2.destroyAllWindows()
    sys.exit(0)
```

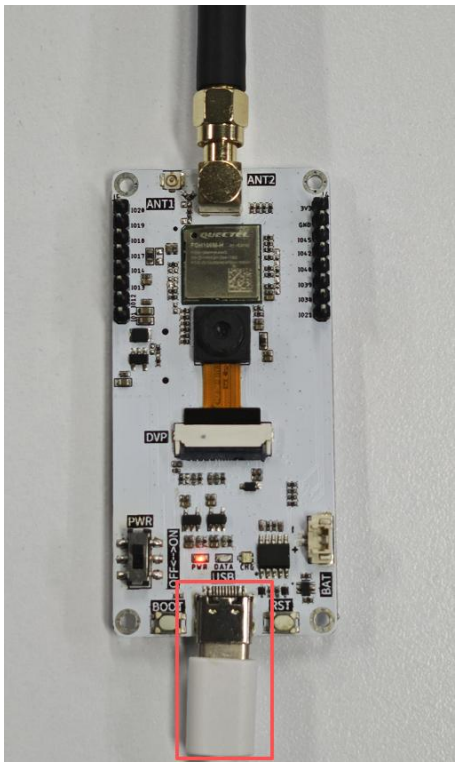
Complete Code

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

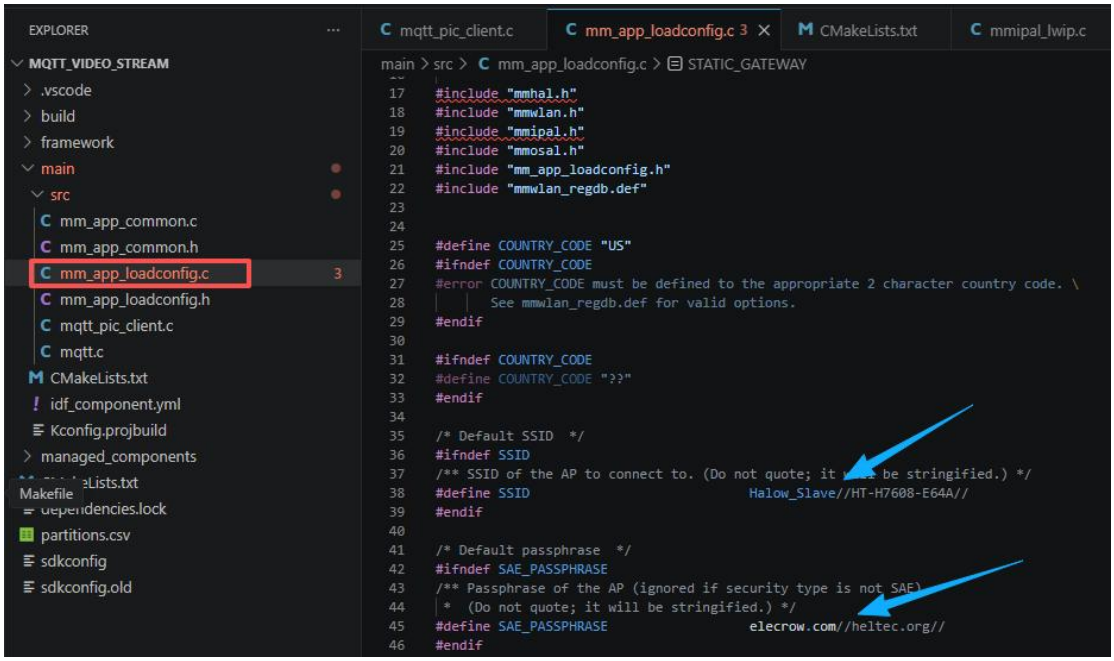
Programming Steps

At this point, we have completed the explanation of all configurations and related code. Next, we will run the code and observe the results.

First, connect the ESP32 WiFi-HaLow module to your computer using a USB cable.



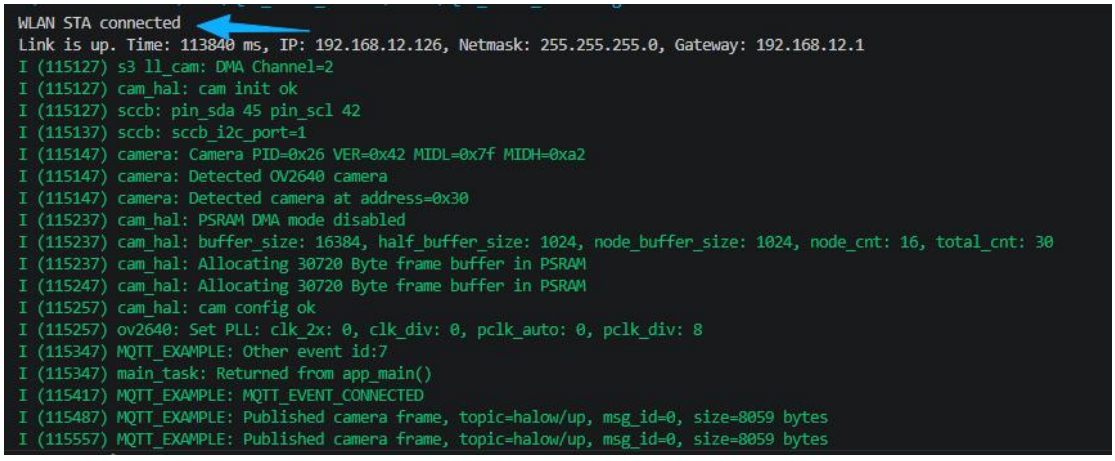
Make sure to modify the WiFi HaLow account and password in the code to match your own settings.



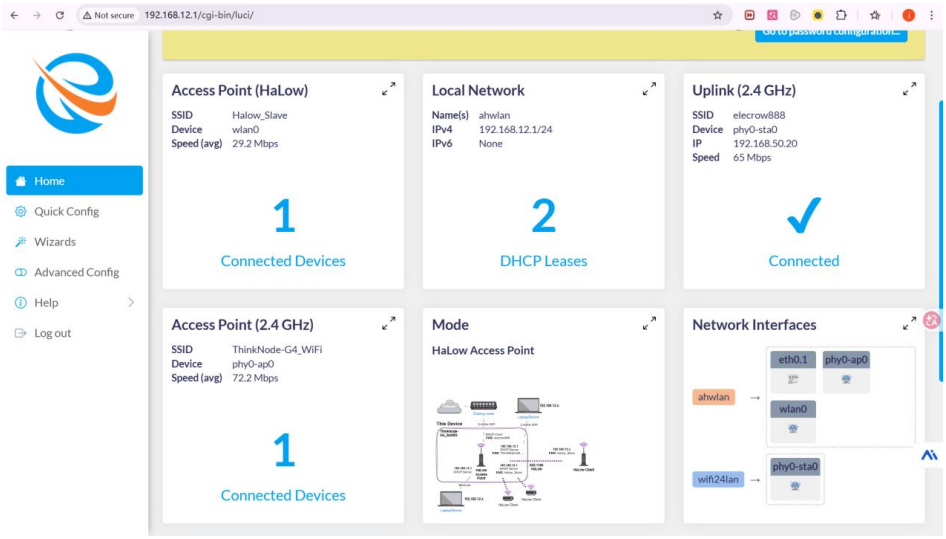
Finally, following the steps in the first lesson, select the ESP-IDF version V5.1.1, UART, the serial port you are using, and the main control chip ESP32S3 of the ESP32 WiFi-HaLow module.

Then click this button for compile, upload, and open serial monitor to complete all these operations with one click.

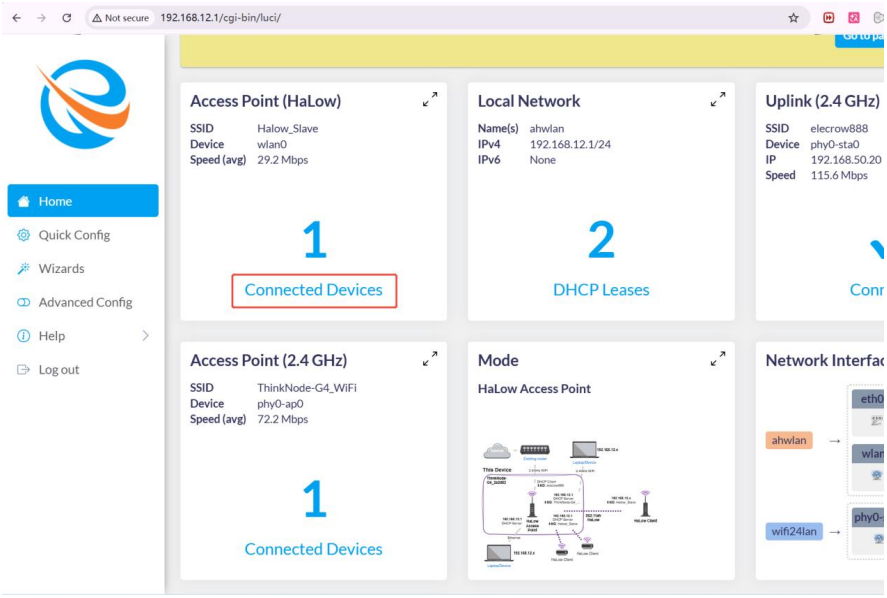
After the flashing process is completed, the ESP32 WiFi-HaLow module will actively connect to the HaLow gateway.



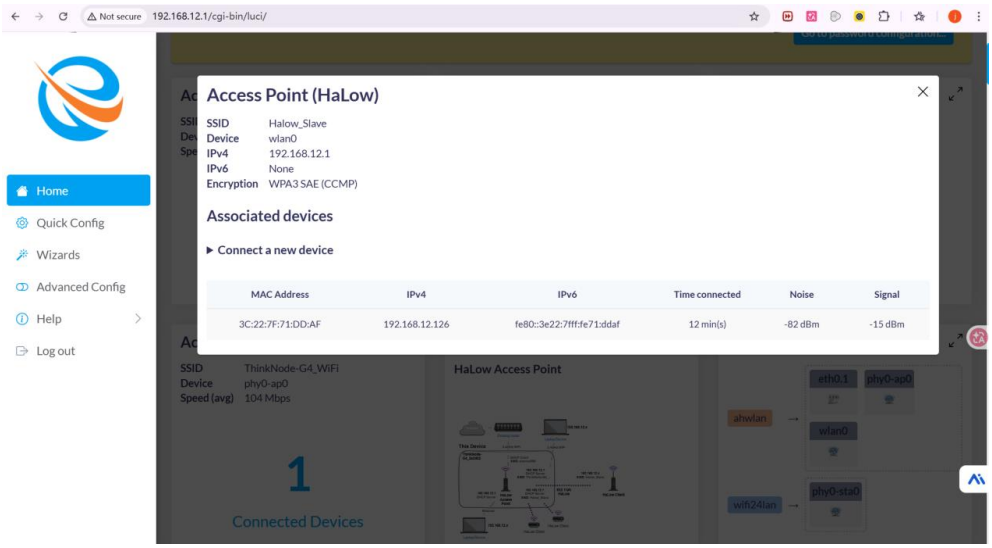
You can also see the connected HaLow devices from the web management interface.



Click Connected Devices.



You can view the information of the devices currently connected to the gateway.



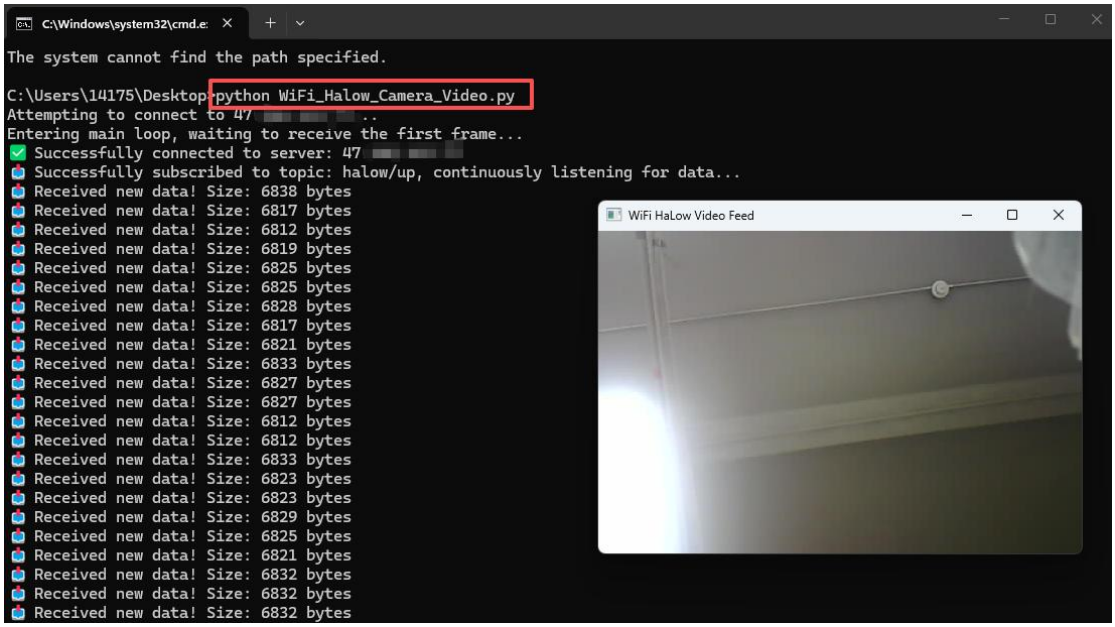
You can also see the HaLow indicator on the gateway light up, which further proves that the ESP32 WiFi-HaLow module has successfully connected to the HaLow gateway.

After you enter the correct MQTT server IP address, the ESP32 WiFi-HaLow module will start uploading the camera images to the MQTT server.

```
I (10557) camera: Detected OV2640 camera
I (10557) camera: Detected camera at address=0x30
I (10647) cam_hal: PSRAM DMA mode disabled
I (10647) cam_hal: buffer_size: 16384, half_buffer_size: 1024, node_buffer_size: 1024, node_cnt: 16, total_cnt: 30
I (10647) cam_hal: Allocating 30720 Byte frame buffer in PSRAM
I (10657) cam_hal: Allocating 30720 Byte frame buffer in PSRAM
I (10667) cam_hal: cam config ok
I (10667) ov2640: Set PLL: clk_2x: 0, clk_div: 0, pclk_auto: 0, pclk_div: 8
I (10747) MQTT_EXAMPLE: Other event id:7
I (10747) main_task: Returned from app_main()
I (12547) MQTT_EXAMPLE: MQTT_EVENT_CONNECTED
I (17247) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6965 bytes
I (17337) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6965 bytes
I (17607) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6789 bytes
I (17667) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6812 bytes
I (17767) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6812 bytes
I (17847) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6796 bytes
I (17947) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6791 bytes
I (18017) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6787 bytes
I (18117) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6787 bytes
I (18207) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6785 bytes
I (18277) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6784 bytes
I (18417) MQTT_EXAMPLE: Published camera frame, topic=halow/up, msg_id=0, size=6768 bytes
```

Next, start the subscriber-side code.
I have now placed the code on the desktop, so I will first enter the file path where it is located.

Then start the code.



You will then be able to see the images captured by the camera on the ESP32 WiFi-HaLow module.

Since this project uses MQTT to transmit camera data, the subscriber side is not restricted by geographical location. No matter where the computer, mobile phone, or other client devices are located, as long as they can access the same public MQTT server and subscribe to the same MQTT topic, they will be able to receive the data published by the ESP32 WiFi-HaLow module in real time, thereby viewing the real-time images captured by the remote camera.