CrowPanel 1.46inch-HMI ESP32 Rotary Display

ESPHome Tutorial

This ESPHome tutorial focuses on the core setup of the CrowPanel 1.46-inch HMI ESP32 Rotary Display, guiding you step-by-step through the essentials: installing ESPHome, powering on the screen, controlling display brightness with the rotary knob, adding an LVGL interface, and linking the rotary encoder to adjust an LVGL slider in real time. These foundational skills will get your smart home control panel up and running, leaving the door open for you to explore more advanced features on your own.

Tutorial Directory

Lesson01---ESPHome Platform Setup

1.Course Introduction

In this lesson, we will teach you how to use and operate the rotary display through ESPHome. Next, we will learn how to install the ESPHome environment so that you can edit code on it and implement related functions for the rotary display.

2.Learning Objectives

Briefly understand the software required to drive the rotary display using ESPHome

Learn how to install the required software

3.Software Introduction

Home Assistant (often called "HA" for short) is a free, open-source smart home management platform—think of it as the central "control brain" for all your smart devices. It works on computers, small servers, or even dedicated smart home hubs, and lets you manage every smart product you own (lights, thermostats, sensors, your rotary screen, etc.) in one unified interface—regardless of the brand or technology the device uses. For your rotary screen, HA acts as the "command center": it can receive inputs from the screen (like knob rotations or touch taps), control other devices based on those inputs (e.g., turn up the lights when you twist the knob), and send data (like room temperature or music volume) to the screen for display.

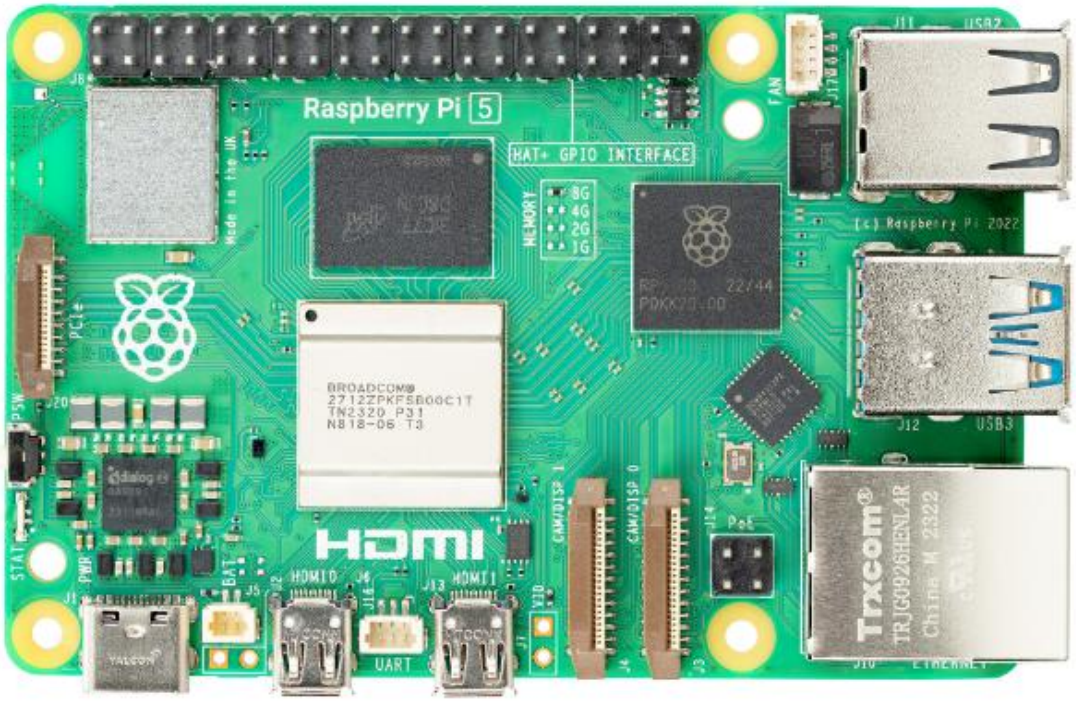
ESPHome is a free, open-source tool built to configure smart hardware (specifically devices powered by ESP32/ESP8266 chips—your rotary screen uses an ESP32). The key benefit? You don't need to write complex code to make the screen work. Instead, you use simple, plain-text configurations (like filling in a template) to define what the rotary screen does: e.g., "show 'Hello World' on the display," "adjust the thermostat when the knob turns," or "wake the screen when touched." ESPHome takes these configurations, turns them into instructions the ESP32 can understand, and "loads" them onto your rotary screen—while also automatically syncing the screen's status (what it's displaying, how the knob is being used) with Home Assistant.

Together, Home Assistant and ESPHome turn your rotary screen into a flexible, customizable smart device: ESPHome gives the screen its "basic skills" (display, knob/touch response), and HA connects those skills to your entire smart home—so

the screen can both control your devices and show you what's happening with them, all without requiring technical expertise.

4.How to Install the Software

First, we need to prepare:
①a Raspberry Pi 5;



②a 64GB SD card;

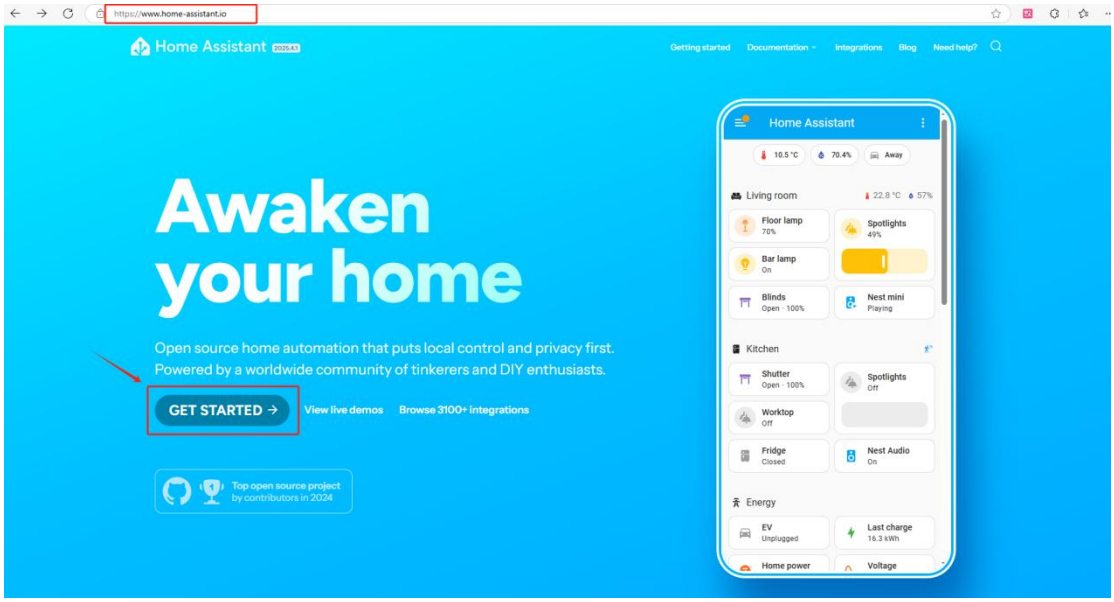
③a CrowPanel 1.46inch-HMI ESP32 Rotary Display 360*360 IPS Round Touch Knob Screen

Product Link:

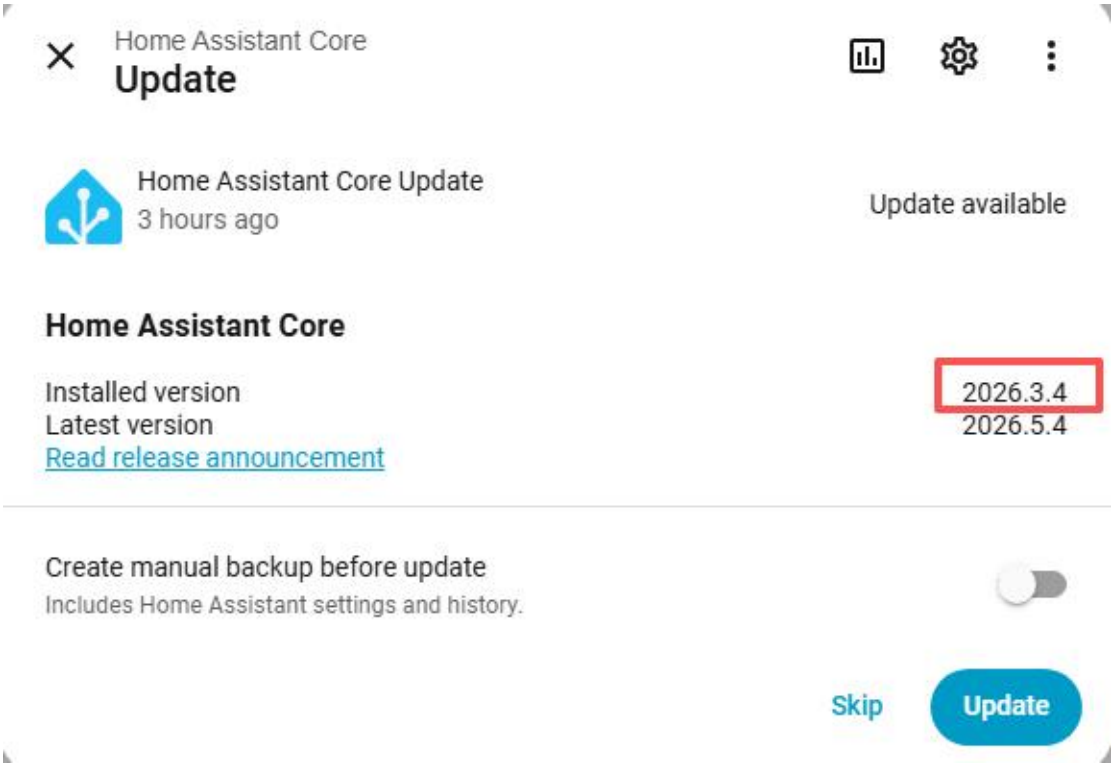
https://www.elecrow.com/crowpanel-1-46inch-hmi-esp32-rotary-display-360-360-ips-round-touch-knob-screen.html

Download HomeAssitant

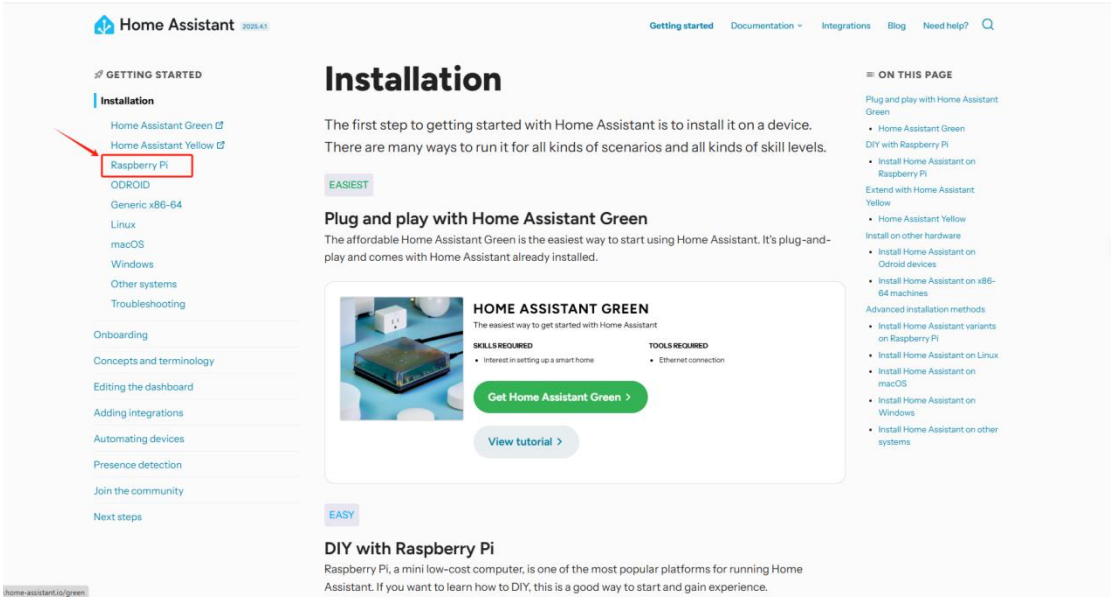
Open the official website of HomeAssitant：https://www.home-assistant.io/



The version we are using here is 2026/3/4



Select a Raspberry Pi and install the Home Assistant system according to this installation guide.(Install according to the official installation guide.)

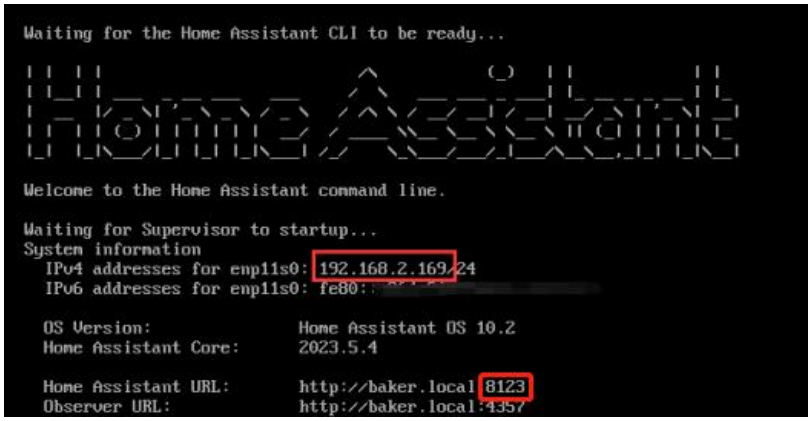


After the installation is complete, insert the SD card with the Home Assistant system into the Raspberry Pi and connect the Ethernet cable.

Note: Make sure that the Wi-Fi network your Crowpanel HMI ESP32 Rotary Display will connect to is on the same local area network (LAN) as the Raspberry Pi.

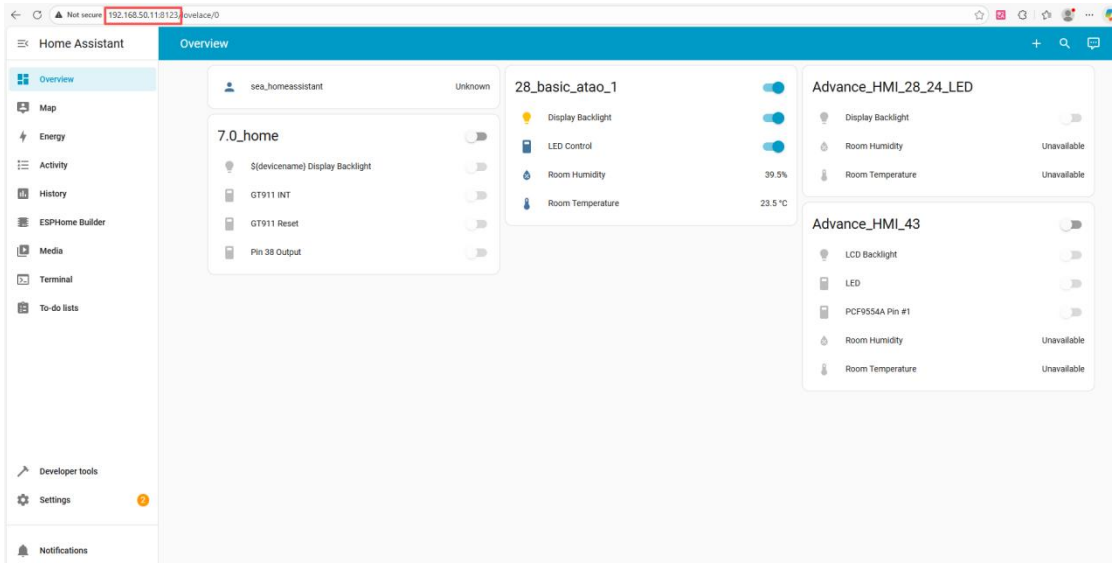
The following devices need to be on the same LAN:
① Your computer
② Crowpanel HMI ESP32 Rotary Display
③ Raspberry Pi with the Home Assistant system

Note: The Raspberry Pi must be connected to a screen to view information. Once everything is ready, power on the Raspberry Pi and wait for it to load. The screen will display the following image.
(You need to connect a display screen to the Raspberry Pi in order to see these information.)

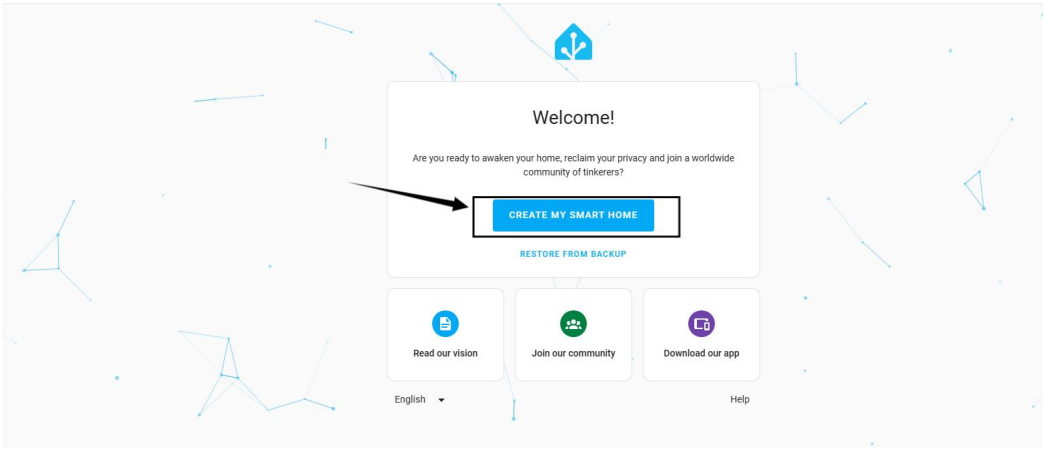


Remember the IP address and the Home Assistant URL.

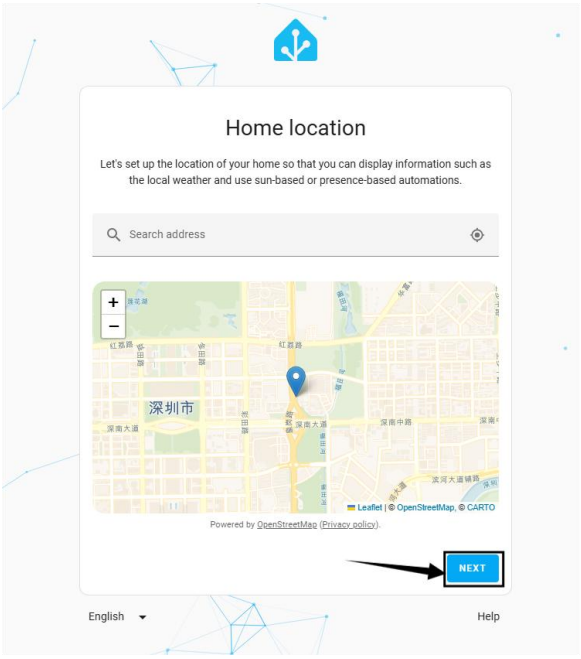
In my case, the IP address is: 192.168.50.11
So I enter **192.168.50.11:8123** in the browser.



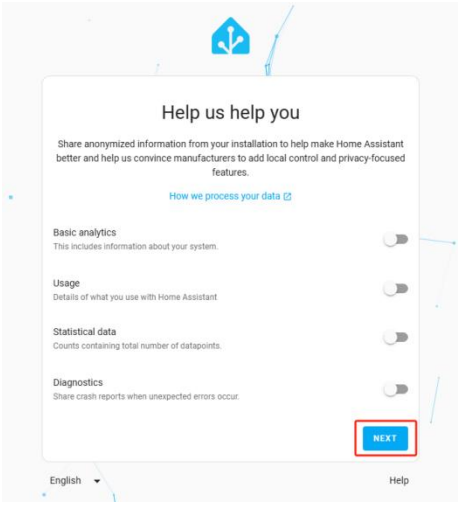
Next, start configuring the Home Assistant system.



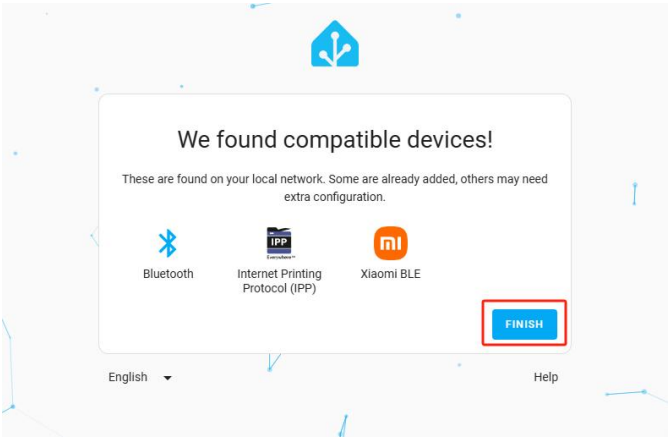
You can either manually set your location or allow it to be detected automatically.



By default, click 'Next'.

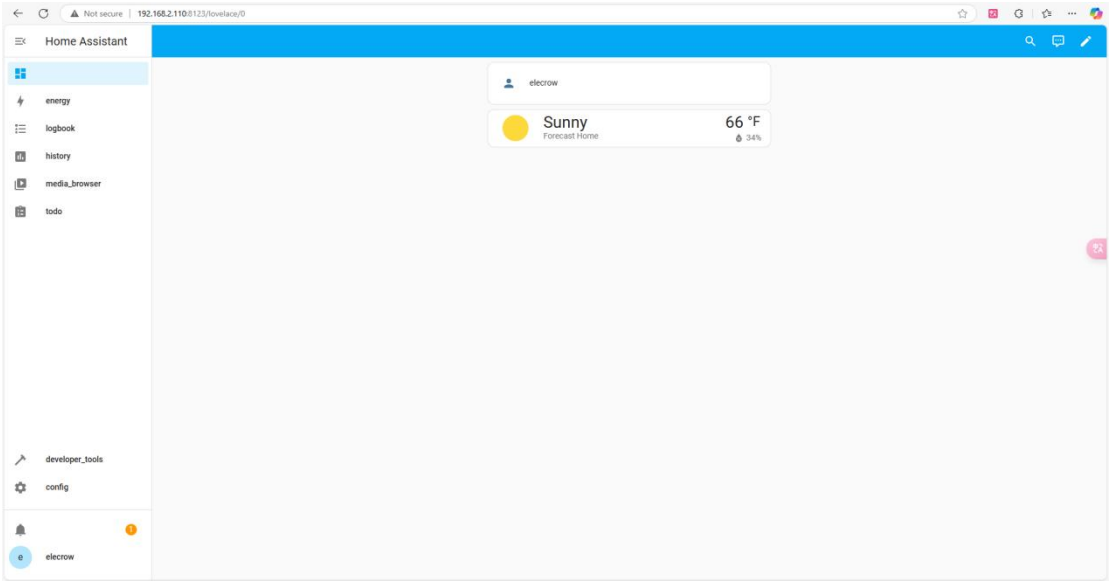


You can either add smart devices now or click **Finish** to add them later.
Here, we will add the devices later.

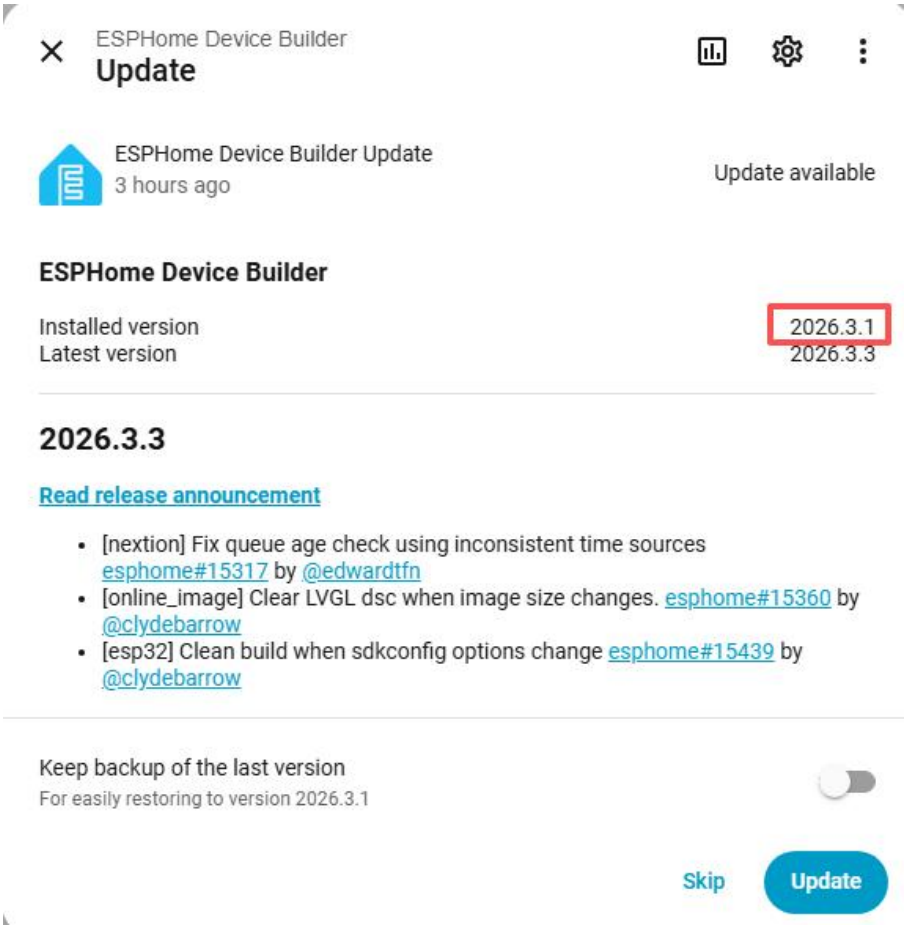


Add ESPHome

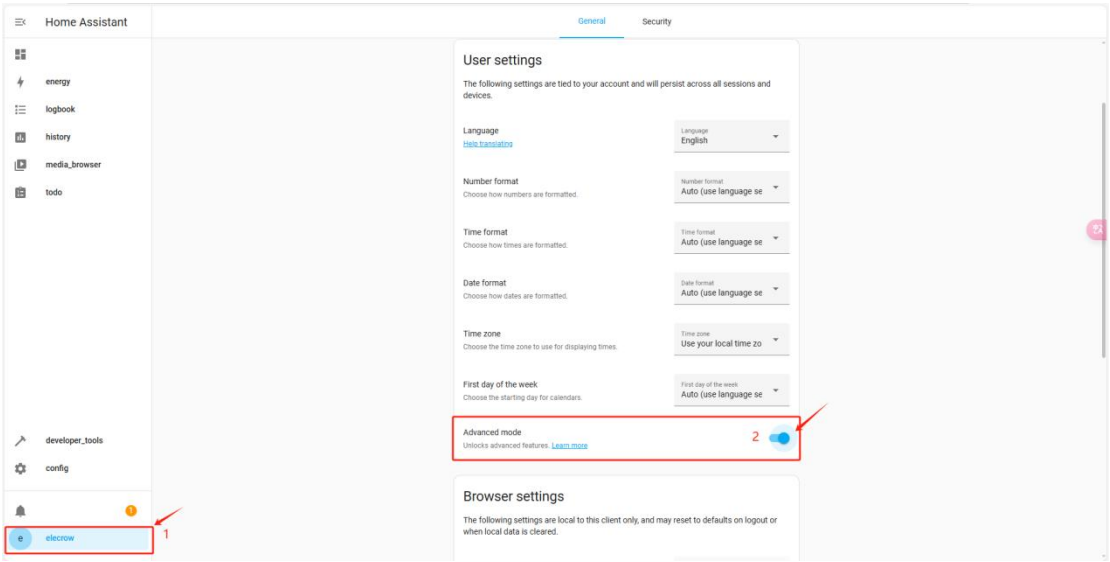
This will bring you to the main interface of Home Assistant.



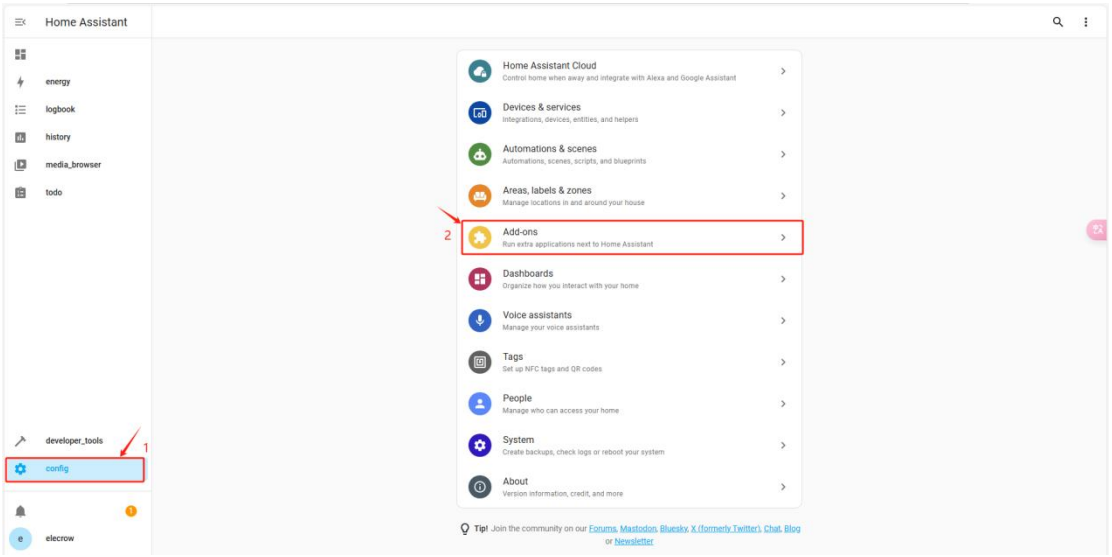
The version of ESPHome we are using is 2026/3/1.



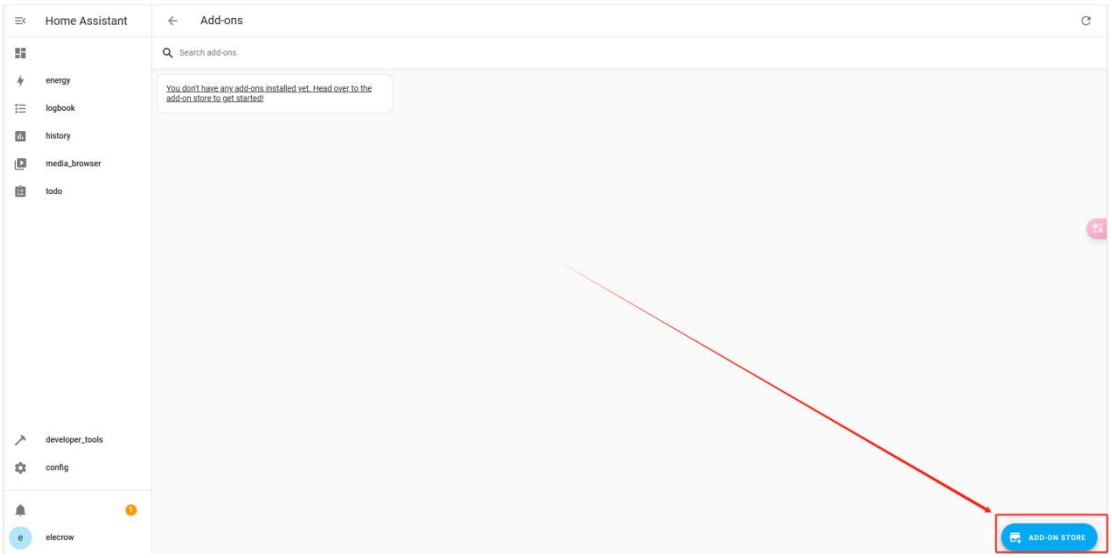
Click on the username and enable "Advanced Mode."

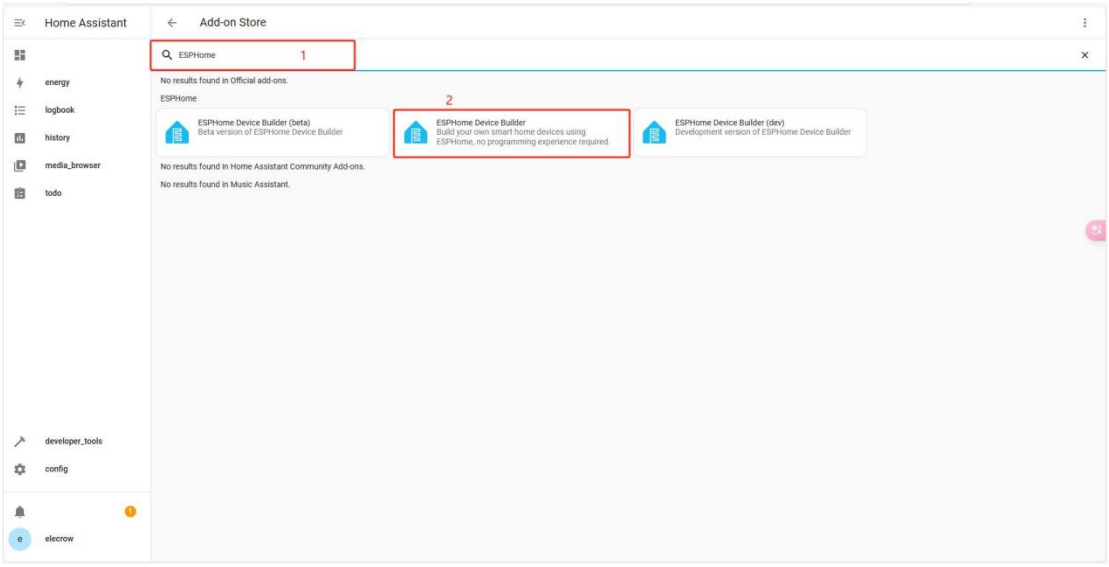


Click on "Settings" and then "Add-ons."

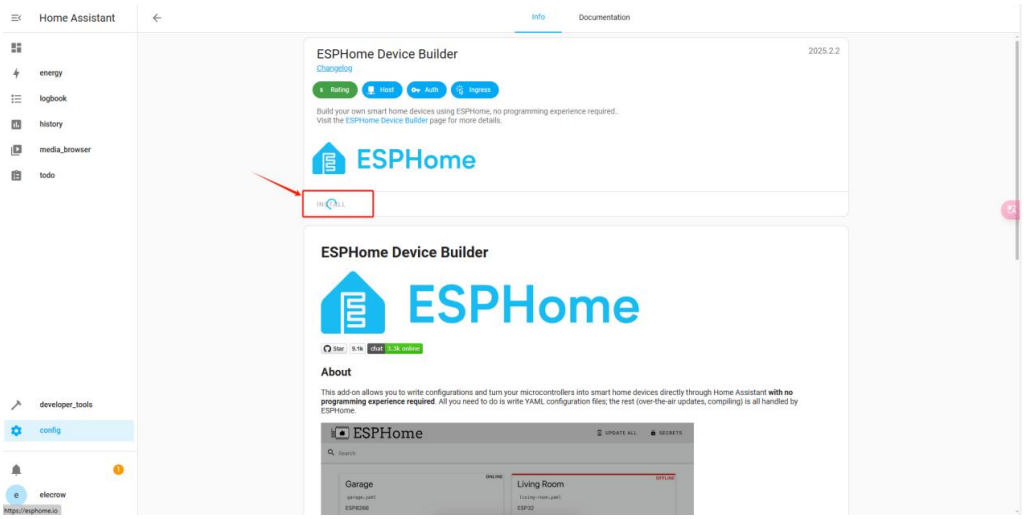


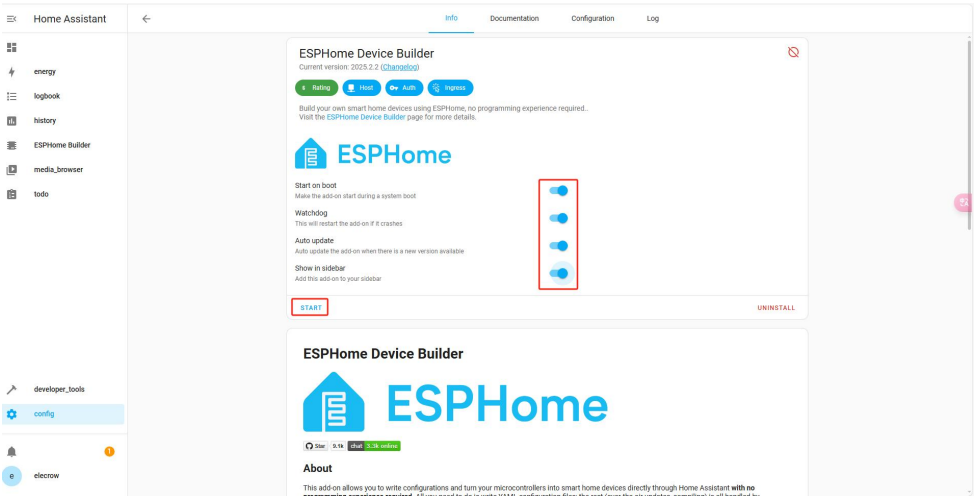
Click on "ADD-ON STORE" and type **ESPHome** to install it.





The installation is in progress, as shown in the image.





In this way, we have successfully installed ESPHome. From now on, we can use ESPHome to edit code and implement the functions we want to achieve.

Lesson02---Turn on the screen

1.Course Introduction

In this lesson, we will use ESPHome to write code to light up the CrowPanel 1.46inch-HMI ESP32 Rotary Display and display "Hello World" on the screen.

2.Learning Objectives

Learn how to drive the rotary display
Light up the screen and print "Hello World" on the display

3.Project Operation Effect Diagram

4.Case Code Download Link and Key Code Explanation

Click the Github link below to download the complete code:

https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/blob/master/example/V1.0/ESPHome/Lesson_Code/rotary-screen-146-display.yaml

Next, let us explain the key parts of this code and how the screen is driven for display on the ESPHome platform.

First, let us establish a core understanding.

YAML code is not “programming code” in the traditional sense (such as Python or C++). Instead, it is a YAML configuration file used by ESPHome — you can think of it as an “instruction manual” written for the ESP32: through configuration content, it tells the ESP32 which hardware to connect, what functions need to be implemented, how to connect to the network, and how to communicate with Home Assistant.

The core idea of YAML is not to write complex program logic, but to describe device behavior and functionality through a structured configuration method. ESPHome reads these YAML configuration contents, automatically generates the underlying code, and then compiles and runs it on the ESP32. Therefore, developers do not need to manually write a large amount of low-level code to quickly implement functionality.

The most important rule in YAML is that indentation is used to represent hierarchical relationships. Content at the same level must be aligned to the left, while child-level content must be indented more spaces than the parent level (ESPHome usually uses two spaces, and Tab keys cannot be used). The YAML parser determines the ownership and hierarchy of data based on indentation, so incorrect indentation may cause the configuration file to fail to run properly.

The most common syntax in YAML is the "Key: Value" structure. The left side of the colon represents the configuration item name, while the right side represents the specific content. For example, Wi-Fi names, passwords, and device names are all configuration values. If a configuration item contains multiple similar entries, "-" can be used to represent a list structure. Comments begin with "#", and comment content is only for developers to read and will not be executed by the program.

In addition, YAML has very strict formatting requirements, especially regarding capitalization, spaces, and indentation. For example, missing one space, adding an extra Tab, or using the wrong hierarchy may cause ESPHome compilation to fail. Therefore, when writing YAML configuration files, special attention must be paid to formatting standards.

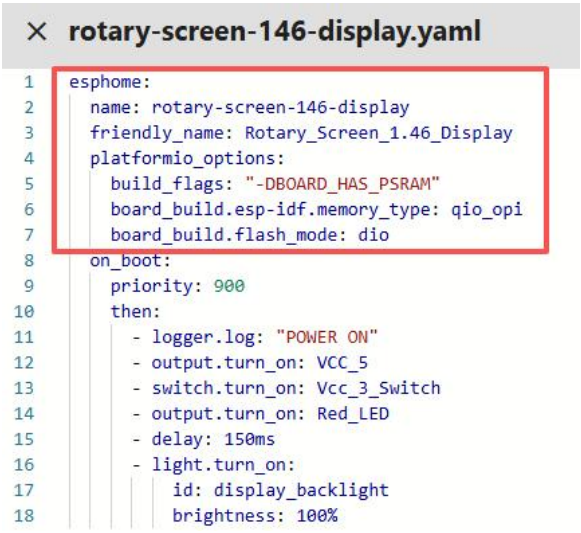
Simply put, ESPHome YAML files are not directly used to “write code,” but instead use a more understandable configuration method to describe the developer’s requirements, which ESPHome then automatically converts into a program that can run on the ESP32.

(1) Basic Information & Compilation Configuration (esphome Block)

This section of code is the core device initialization part of the ESPHome configuration file. It is mainly responsible for defining the basic device information and the startup process after the ESP32 powers on.

```
esphome:
  name: rotary-screen-146-display
  friendly_name: Rotary_Screen_1.46_Display
  platformio_options:
    build_flags: "-DBOARD_HAS_PSRAM"
    board_build.esp-idf.memory_type: qio_opi
    board_build.flash_mode: dio
  on_boot:
    priority: 900
    then:
      - logger.log: "POWER ON"
      - output.turn_on: VCC_5
      - switch.turn_on: Vcc_3_Switch
      - output.turn_on: Red_LED
      - delay: 150ms
      - light.turn_on:
          id: display_backlight
          brightness: 100%
```

name: rotary-screen-146-display is used to specify the internal name of the device. This name will be used during ESPHome compilation, network identification, and Home Assistant integration; friendly_name: Rotary_Screen_1.46_Display is the user-friendly display name shown to users, making it more intuitive in the Home Assistant interface.



platformio_options: is used to pass additional parameters to the underlying compilation system.

- Among them, build_flags: "-DBOARD_HAS_PSRAM" tells the compiler that the current ESP32 board is equipped with PSRAM (external high-speed memory). Since the LVGL graphical interface and larger display screens consume a significant amount of memory, enabling PSRAM support can help avoid insufficient memory issues;
- board_build.esp-idf.memory_type: qio_opi is used to specify the communication mode for Flash and PSRAM, improving data access efficiency;
- board_build.flash_mode: dio sets the Flash to use dual-line data transmission mode.

on_boot: represents the initialization process that is automatically executed every time the ESP32 powers on or restarts.

```
8   on_boot:
9     priority: 900
10    then:
11      - logger.log: "POWER ON"
12      - output.turn_on: VCC_5
13      - switch.turn_on: Vcc_3_Switch
14      - output.turn_on: Red_LED
15      - delay: 150ms
16      - light.turn_on:
17          id: display_backlight
18          brightness: 100%
```

priority: 900 represents a relatively high startup priority, ensuring that these operations are executed as early as possible.

then: defines the tasks executed sequentially during startup:

- First, logger.log: "POWER ON" outputs the log message "POWER ON" to the serial port, making it easier for developers to confirm that the device has started successfully during debugging;
- Next, output.turn_on: VCC_5 enables the output pin named VCC_5, which provides power to the UART and IIC interfaces;
- Then, switch.turn_on: Vcc_3_Switch enables the 3.3V screen power control switch, powering the display;
- output.turn_on: Red_LED turns on the red status indicator LED, allowing users to visually confirm that the device has entered the running state; delay: 150ms means pausing for 150 milliseconds to wait for the power supply to stabilize, preventing abnormal startup of peripherals such as the display before the power becomes stable;

- Finally, light.turn_on: controls the backlight device named display_backlight to turn on, and sets the brightness to maximum through brightness: 100%, allowing the screen to fully light up.

The display_backlight here actually controls the screen backlight pin GPIO46 when enabled, which will be described in detail later in the output section.

```
84  light:
85    - platform: monochromatic
86      output: backlight
87      id: display_backlight
88      restore_mode: ALWAYS_ON
```

```
61  output:
62    - platform: ledc
63      id: backlight
64      pin: GPIO46
65    - platform: gpio
66      id: VCC_5
67      pin: GPIO2
68    - platform: gpio
69      id: Red_LED
70      pin: GPIO40
```

The Vcc_3_Switch here corresponds to GPIO1. We need to set this pin high so that current can pass through the screen, allowing the display to power on.

In other words, if you only enable the screen backlight pin GPIO46, the screen still will not light up. GPIO1 must also be enabled at the same time so that current can flow through the display, allowing it to turn on. Both are required for the screen to light up properly.

```
 8    on_boot:
 9      priority: 900
10      then:
11        - logger.log: "POWER ON"
12        - output.turn_on: VCC_5
13        - switch.turn_on: Vcc_3_Switch
14        - output.turn_on: Red_LED
15        - delay: 150ms
16        - light.turn_on:
17            id: display_backlight
18            brightness: 100%

72  switch:
73    - platform: gpio
74      name: "VCC_3 Control"
75      id: Vcc_3_Switch
76      pin:
77        number: GPIO1
78        mode:
79          output: true
80      restore_mode: ALWAYS_ON
81      on_turn_on:
82        - logger.log: "GPIO1 is HIGH"
```

The VCC_5 here corresponds to GPIO2. This pin is used to provide power to the UART interface and the I2C interface on the 1.46-inch rotary display.

```
on_boot:
  priority: 900
  then:
    - logger.log: "POWER ON"
    - output.turn_on: VCC_5
    - switch.turn_on: Vcc_3_Switch
    - output.turn_on: Red_LED
    - delay: 150ms
    - light.turn_on:
        id: display_backlight
        brightness: 100%
```

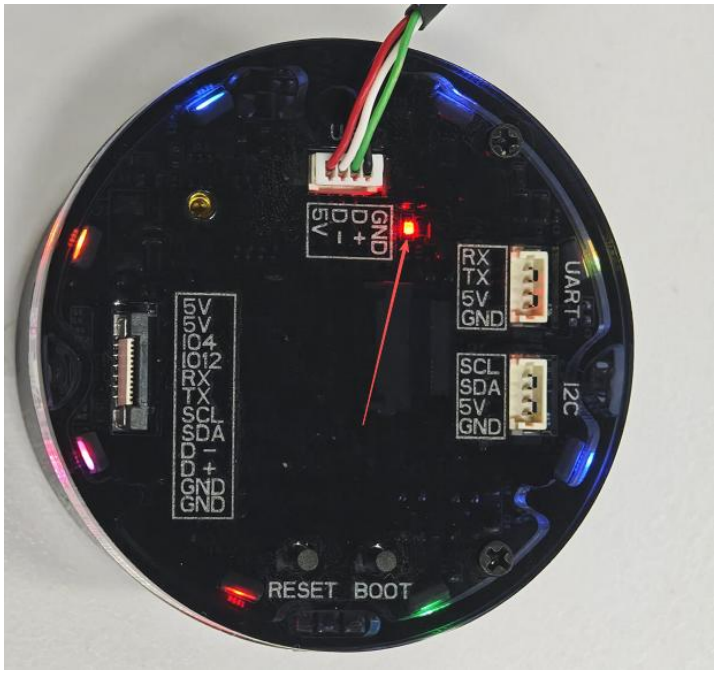
```
output:
  - platform: ledc
    id: backlight
    pin: GPIO46
  - platform: gpio
    id: VCC_5
    pin: GPIO2
  - platform: gpio
    id: Red_LED
    pin: GPIO40
```

In other words, this pin must be set high before the UART interface and I2C interface can be used, allowing current to flow through them.

The Red_LED here corresponds to GPIO40, which is the pin used to turn on the power indicator LED.

```
  on_boot:
    priority: 900
    then:
      - logger.log: "POWER ON"
      - output.turn_on: VCC_5
      - switch.turn_on: Vcc_3_Switch
      - output.turn_on: Red_LED
      - delay: 150ms
      - light.turn_on:
          id: display_backlight
          brightness: 100%

output:
  - platform: ledc
    id: backlight
    pin: GPIO46
  - platform: gpio
    id: VCC_5
    pin: GPIO2
  - platform: gpio
    id: Red_LED
    pin: GPIO40
```

When the 1.46-inch rotary display is powered on, this pin will light up and only serves as a status indicator LED.



(2)Tell ESPHome Which Chip You Are Using

```
esp32:
  board: esp32-s3-devkitc-1
  flash_size: 16MB
  framework:
    type: esp-idf
    sdkconfig_options:
      CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240: y
      CONFIG_ESP32S3_DATA_CACHE_64KB: y
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y
      CONFIG_SPIRAM_RODATA: y
```

This section of code is mainly used to configure the ESP32 hardware platform and the underlying runtime environment. It determines how ESPHome ultimately compiles the program, as well as the performance and memory usage of the ESP32 chip during operation.

"board: esp32-s3-devkitc-1" is used to specify the current development board model. Here, it means the program will be compiled according to the hardware resources of the ESP32-S3 DevKitC development board, including GPIO mapping, Flash configuration, and low-level drivers;

"flash_size: 16MB" indicates that the current development board is equipped with 16MB Flash memory. Flash can be regarded as the “long-term storage space” of the microcontroller, where program code, font files, image resources, and some configuration data are stored.

"framework:" indicates the selection of the underlying development framework,

- Among them, "type: esp-idf" specifies ESP-IDF as the runtime framework. It is the official native development environment provided by Espressif. Compared with the Arduino framework, it provides more complete hardware support for the ESP32-S3 and usually performs better in scenarios involving LVGL graphical interfaces, large-memory displays, and high-performance applications.
- The following "sdkconfig_options:" is used to modify some internal system configuration parameters of ESP-IDF. "CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240: y" means setting the CPU frequency of the ESP32-S3 to 240MHz, allowing the processor to run at the maximum clock speed to improve interface refresh speed and overall performance;
- "CONFIG_ESP32S3_DATA_CACHE_64KB: y" means enabling a 64KB Data Cache. The cache functions like a “high-speed temporary warehouse,” allowing the CPU

to read frequently used data faster, thereby improving display and processing efficiency;

- "CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y" means allowing the CPU to fetch program instructions from external PSRAM (expanded memory), reducing the pressure on internal RAM;
- "CONFIG_SPIRAM_RODATA: y" means allowing Read Only Data, such as strings, font data, and image resources, to be stored in PSRAM instead of occupying all the ESP32 internal memory.

The core purpose of the entire configuration is actually very simple: to maximize the performance advantages of the ESP32-S3 + PSRAM + large-capacity Flash combination, making the LVGL graphical interface, 360×360 circular display, and complex UI run more smoothly while avoiding insufficient memory issues.

(3)psram: Preparing a "Memory Warehouse" for Large Displays and LVGL

```
psram:
  mode: octal
  speed: 80MHz
```

This section is the external PSRAM configuration.

The internal RAM built into the ESP32-S3 is relatively small, and a circular display + ILI9xxx + LVGL almost certainly requires PSRAM.

mode: octal indicates the use of 8-line OPI mode (high speed and large bandwidth), while speed: 80MHz is a commonly used stable configuration for the ESP32-S3 with displays.

If this section is not configured, LVGL frame buffers, font caches, and drawing buffers may fail to allocate memory. The most typical symptom is: the firmware flashes successfully, but the screen remains completely black.

(4)logger: — The "Vital Probe" of the Serial Port

```
logger:
```

This section enables the serial logging function.

It does not directly control any hardware, but it is extremely important for beginners:

- You can check whether the ESP has started successfully
- You can verify whether it has entered on_boot

- You can see whether SPI / Display / LVGL has been initialized

Without logger, if the screen stays black, it is almost impossible to know where the problem occurred.

(5)api: — Communicating with Home Assistant

```
api:
  encryption:
    key: "XSCr5HHOMRzn55Ch8+HfGYhnEevFL9OAxm8YJbkTr/U="
```

This section is the communication interface between ESPHome and Home Assistant.

Even if you are currently only displaying “Hello World,” the core design of ESPHome is still for IoT devices. The api allows you to remotely control the device, update the interface, and send data in the future.

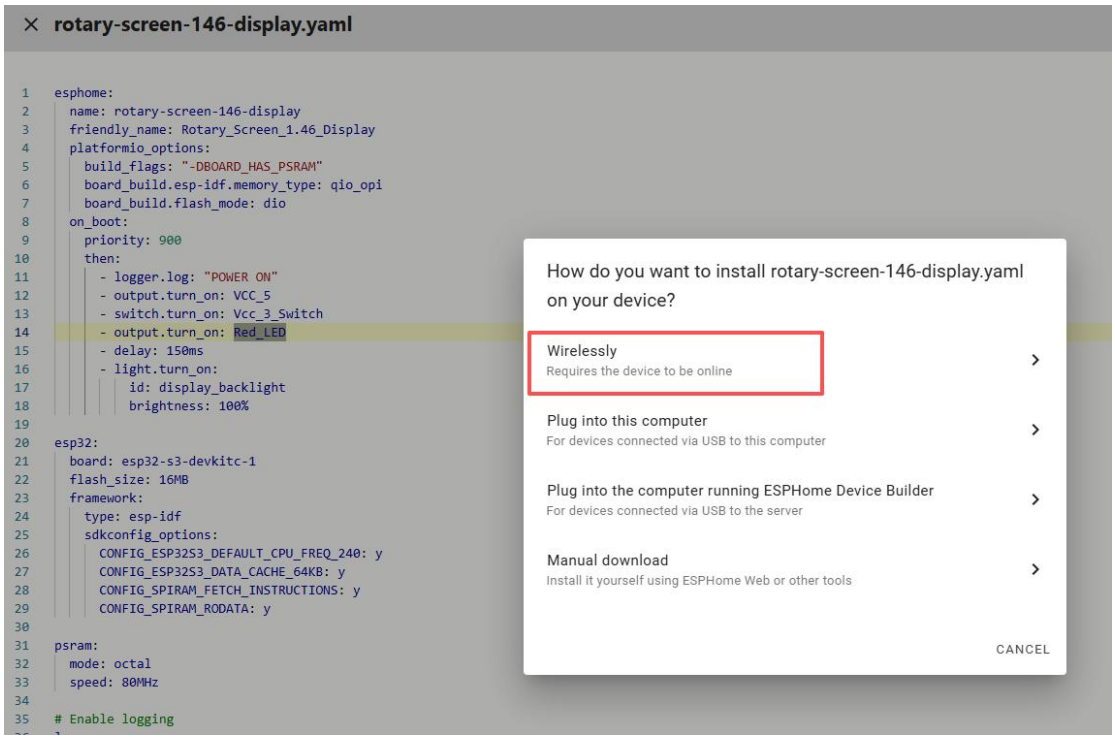
The encryption key is used to prevent the device from being casually controlled within the local network.

(6)ota: — Wireless Firmware Updates Without Repeatedly Plugging in USB

```
ota:
  - platform: esphome
    password: "b0a0f8876c261e9413c3036c357d0675"
```

This section allows you to flash firmware directly over Wi-Fi in the future. This step corresponds to the OTA method used later when flashing firmware. After the device successfully connects to the network, you will be able to remotely upload programs through Wi-Fi.

(The prerequisite is that you have already uploaded the bin file using Manual download first.)



(7)wifi: + captive_portal: — Network Connection and Emergency AP

```
wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  ap:
    ssid: "Rotary-Screen-146-Display"
    password: "U51T8AKI6joe"
captive_portal:
```

This section is the Wi-Fi connection configuration.

Under normal circumstances, the device connects to your home router; if the connection fails, it will automatically enable an Access Point (AP), allowing you to reconnect and reconfigure the device using your phone.

captive_portal is a “web configuration interface” used to prevent the device from becoming inaccessible after flashing.

(8)output: — The “Executor” for GPIO and PWM

```
output:
  - platform: ledc
    id: backlight
    pin: GPIO46
  - platform: gpio
    id: VCC_5
    pin: GPIO2
  - platform: gpio
    id: Red_LED
    pin: GPIO40


switch:
  - platform: gpio
    name: "VCC_3 Control"
    id: Vcc_3_Switch
    pin:
      number: GPIO1
      mode:
        output: true
    restore_mode: ALWAYS_ON
    on_turn_on:
      - logger.log: "GPIO1 is HIGH"
```

This section of code is mainly responsible for the functional definition of ESP32 output pins (GPIO) and the power control logic. Through the output: and switch: modules, the ESP32 GPIO pins are configured as output devices capable of controlling the backlight, power supply, and status indicator LEDs.

First, the output: section is used to create low-level output objects.

platform: ledc indicates the use of the built-in LED PWM (Pulse Width Modulation) controller of the ESP32. Among them, id: backlight assigns the name backlight to this output function, allowing other configurations to directly reference it later through this ID; pin: GPIO46 indicates that GPIO46 is used to control the screen backlight. LEDC is used here instead of a normal GPIO because the backlight usually requires not only simple ON/OFF control, but also brightness adjustment. PWM can adjust brightness by rapidly changing the duty cycle of high and low voltage signals. For example, a 50% duty cycle makes the screen appear half-bright, while 100% represents full brightness.

Next, platform: gpio defines a normal digital output. id: VCC_5 is bound to GPIO2, and this pin is used to provide power to the UART interface and I2C interface on the 1.46-inch rotary display.

Another id: Red_LED uses GPIO40 to control the onboard red status indicator LED. The program can indicate the current device status by turning the LED on or off.

The following switch: section further encapsulates a normal GPIO output into a “controllable switch.” platform: gpio indicates that the switch is still implemented using GPIO at the hardware level. name: "VCC_3 Control" specifies the display name shown in Home Assistant or ESPHome, while id: Vcc_3_Switch is the internal name used by the program.

Inside pin:, number: GPIO1 indicates that the actual control pin is GPIO1, while mode: output: true means this pin operates in output mode and can actively output high or low voltage levels. restore_mode: ALWAYS_ON means that every time the device restarts or recovers from power loss, this switch will automatically turn on by default without requiring manual control again. In other words, after the ESP32 powers on, GPIO1 will automatically output a high level. Finally, on_turn_on: defines the action automatically executed when the switch is turned on. Here, logger.log: "GPIO1 is HIGH" outputs a message to the serial log, allowing developers to confirm that GPIO1 has been successfully pulled high.

Combined with the previous on_boot startup logic, when the device executes switch.turn_on: Vcc_3_Switch during startup, it is essentially causing GPIO1 to output a high level, thereby enabling power to the display.

(9)light: — Packaging PWM as a “Light”

```
light:
  - platform: monochromatic
    output: backlight
    id: display_backlight
    restore_mode: ALWAYS_ON
```

This section of code is used to encapsulate the previously defined screen backlight PWM output into a “light component” that can directly control brightness, making it easier for subsequent programs to uniformly manage the screen brightness.

- platform: monochromatic indicates the creation of a monochrome light device, which is typically used to control devices that only require brightness adjustment without color changes, such as monochrome LEDs or display backlights;

- output: backlight indicates that this light component is actually connected to the previously defined backlight output channel, which is the screen backlight pin controlled by GPIO46 through PWM (Pulse Width Modulation). Therefore, adjusting the brightness of this light essentially means adjusting the screen brightness;
- id: display_backlight defines an internal name for this light component, allowing subsequent programs to perform operations such as turning it on, turning it off, or adjusting brightness through this ID;
- restore_mode: ALWAYS_ON indicates that every time the device restarts or powers on again, the backlight will automatically turn on by default, preventing the situation where the system has already started but the screen remains off.

(10)spi: — The High-Speed Highway for Screen Communication

```
spi:
  id: spi_bus
  mosi_pin: 11
  clk_pin: 10
```

This section of code is mainly used to configure the SPI (Serial Peripheral Interface) communication bus of the ESP32. SPI is a commonly used communication method for high-speed data exchange between a microcontroller and external devices such as displays, sensors, and Flash chips.

- id: spi_bus defines an internal name for the current SPI bus, allowing other components later in the project (such as the display driver) to use this SPI bus for communication through this ID;
- mosi_pin: 11 indicates that GPIO11 of the ESP32 is configured as the MOSI (Master Out Slave In) pin, which is responsible for sending data from the ESP32 to external devices, such as transmitting color data and drawing commands to the display;
- clk_pin: 10 indicates that GPIO10 is configured as the clock signal pin (Clock). During SPI communication, every bit of transmitted data is synchronized through clock pulse signals, allowing the master and slave devices to maintain data synchronization.

Combined with the overall project, this section is essentially establishing a data communication channel between the ESP32 and the circular display. The LCD controller configured later in the display driver section will receive the display data sent by the ESP32 through this SPI bus, thereby enabling the display of text, images, and LVGL graphical interfaces.

(11)display: — The Actual Screen Driver

```
display:
  - platform: ili9xxx
    id: round_display
    model: CUSTOM
    cs_pin: GPIO9
    dc_pin: GPIO3
    reset_pin: GPIO14
    dimensions:
      width: 360
      height: 360
      offset_height: 0
    data_rate: 40MHz
    color_order: RGB
    invert_colors: false
    init_sequence:
      - [0xDE, 0x00]
      - [0xDF, 0x98, 0x55]
      - [0xB2, 0x1F]
      - [0xB7, 0x00, 0x1D, 0x00, 0x45]
      - [0xBB, 0x1B, 0x64, 0xC4, 0x1E, 0x3E, 0xF5]
      - [0xBC, 0x03, 0x20, 0xF3, 0xC0]
      - [0xC0, 0x22, 0xA1]
      - [0xC3, 0x00, 0x02, 0x2A, 0x0B, 0x08, 0x48, 0x08, 0x04, 0x62, 0x30, 0x30]
      - [0xC4, 0x40, 0x00, 0xAD, 0x68, 0x43, 0x07, 0x04, 0x16, 0x43, 0x07, 0x04]
      - [0xD3, 0x28, 0x13]
      - [0xD9, 0x00, 0x00, 0xFF, 0x00, 0xF0, 0x00]
      - [0xDE, 0x01]
      - [0xB7, 0x13, 0xE7, 0x64, 0x39, 0x06, 0x36, 0x19, 0x1C]
      - [0xBE, 0x00]
      - [0xC1, 0x00, 0x4A, 0x80]
      - [0xC2, 0x00, 0x16, 0xDA, 0xE7]
      - [0xC7, 0x00, 0x00, 0x00, 0x38, 0x08, 0x08, 0x00, 0x01]
      - [0xC8, 0x00, 0x00, 0x00, 0x00, 0x15, 0x3B]
      - [0xC9, 0x00, 0x16, 0x06, 0x04, 0x0A]
      - [0xCA, 0x08, 0x35, 0x16, 0x1F, 0x1F]
```

```
    - [0xCB, 0x01, 0x16, 0x07, 0x05, 0x0B]
    - [0xCC, 0x09, 0x35, 0x16, 0x1F, 0x1F]
    - [0xCD, 0x01, 0x16, 0x09, 0x0B, 0x05]
    - [0xCE, 0x07, 0x15, 0x1F, 0x16, 0x1F]
    - [0xCF, 0x00, 0x16, 0x08, 0x0A, 0x04]
    - [0xD0, 0x06, 0x15, 0x1F, 0x16, 0x1F]
    - [0xD1, 0x02, 0x30]
    - [0xD2, 0x02, 0x03, 0x52, 0xDF, 0xDD]
    - [0xD3, 0x3B, 0x04, 0x48]
    - [0xD5, 0x10, 0x10, 0x07, 0x07, 0x0F, 0x94, 0x26]
    - [0xD6, 0x00, 0x00, 0x40]
    - [0xD7, 0x00, 0x00, 0x20]
    - [0xDE, 0x02]
    - [0xB6, 0x1C]
    - [0xDE, 0x00]
    - [0x4D, 0x00]
    - [0x4E, 0x00]
    - [0x4F, 0x00]
    - [0x4C, 0x01]
    - [0x4C, 0x00]
    - [0x36, 0x00]
    - [0x35]
    - [0x2A, 0x00, 0x00, 0x01, 0x67]
    - [0x2B, 0x00, 0x00, 0x01, 0x67]
    - [0x3A, 0x05]
    - [0x11,120]
    - [0x29,20]
```

This section of code is mainly configuring the driver parameters and initialization process of the circular LCD display connected to the ESP32. It determines how the ESP32 communicates with the screen and the working state of the screen after startup.

- platform: ili9xxx indicates the use of the ILI9XXX series display driver component, while id: round_display defines an internal name for the display. The LVGL graphical interface will later output content to the screen through this name;
- model: CUSTOM indicates that this project is not using a built-in standard ESPHome display model, but instead uses custom driver parameters, so the developer must manually provide initialization commands;

- cs_pin: GPIO9 is the Chip Select pin, which tells the SPI bus which device is currently being communicated with;
- dc_pin: GPIO3 is the Data/Command control pin, used to distinguish whether the transmitted data is a display command or image data;
- reset_pin: GPIO14 is the hardware reset pin, which allows the LCD controller chip to return to its initial state during startup.
- dimensions: specifies the logical resolution of the screen. Here, width: 360 and height: 360 indicate a circular display with a resolution of 360×360, while offset_height: 0 means the display area does not require additional offset adjustment.
- data_rate: 40MHz sets the SPI communication speed to 40MHz. A higher data transfer speed can improve the refresh efficiency of the LVGL interface;
- color_order: RGB specifies that the color data arrangement uses RGB format;
- invert_colors: false indicates that color inversion is disabled.

The most complex part is init_sequence:, which is a series of low-level register initialization commands sent to the LCD driver chip. Each hexadecimal value is actually configuring internal parameters of the display controller, such as voltage settings, Gamma curve calibration, timing parameters, color format, scanning direction, display area, and driver modes. For example, 0x11 usually indicates exiting sleep mode so the display can start operating, while the following 120 means waiting for 120ms; 0x29 usually indicates officially enabling display output, followed by 20 which means waiting for 20ms; 0x2A and 0x2B are used to define the display area range; 0x3A configures the color format; and registers such as 0xB7, 0xC3, and 0xD5 are used to adjust internal LCD driver parameters.

These initialization commands are usually provided by the display manufacturer, and developers generally should not modify them manually, because any incorrect parameter may cause issues such as a white screen, abnormal colors, incorrect color rendering, incorrect display orientation, or even a completely blank display. You can simply use them directly.

(12)font: — Text Is Not Drawn Out of Thin Air

```
font:
  - file: "gfonts://Roboto"
    id: roboto24
    size: 24
```

This section defines the font resources.

When LVGL displays text, the fonts must be compiled into the firmware in advance.

Here, you are using the Roboto font from Google Fonts with a size of 24 pixels, which is suitable for a 360×360 circular display.

(13)lvgl: — The “Brain” of the Graphical Interface

```
lvgl:
  displays:
    - round_display

  default_font: roboto24
  disp_bg_color: white

  style_definitions:
    - id: hello_style
      text_font: roboto24
      text_color: 0x000000
      bg_opa: TRANSP
      align: center

  pages:
    - id: hello_page
      widgets:
        - label:
            id: hello_label
            text: "Hello World!"
            styles: hello_style
            align: CENTER
```

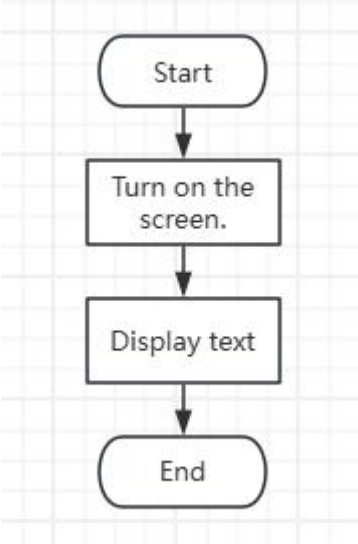
This entire "lvgl:" configuration section is used to enable and build a basic graphical interface system, and display a line of “Hello World!” text at the center of the screen. First, "displays" binds LVGL to the previously initialized "round_display" screen, telling LVGL that “all interface content should be drawn on this display”;

Next, "default_font" and "disp_bg_color" are used to set the default font and background color of the entire interface, preventing situations where black text on a black background makes the screen appear to be completely black;

Then, inside "style_definitions", a reusable text style called "hello_style" is defined. It explicitly specifies the use of the 24-pixel Roboto font, black text color, transparent background, and centered alignment, ensuring that the text remains clearly visible;

Finally, inside "pages", a page named "hello_page" is created, and a "label" text widget is placed on this page with the content “Hello World!”. The previously defined style is applied and the text is centered on the screen, thereby completing a minimal visible and verifiable Hello World interface used to confirm that both the display and LVGL are functioning correctly.

5.Code Logic Flowchart



6.Flashing Steps

Next, we will teach everyone how to use ESPHome to write code for the first time, including the complete operation workflow. Please follow us step by step.

Here we emphasize once again:

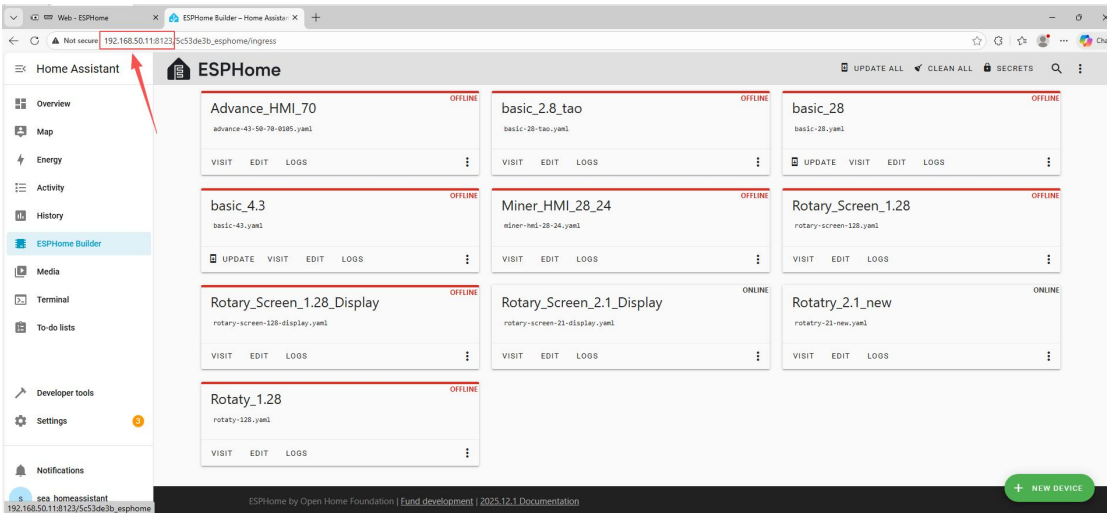
The following devices need to be on the same LAN:

① Your computer

② Crowpanel HMI ESP32 Rotary Display

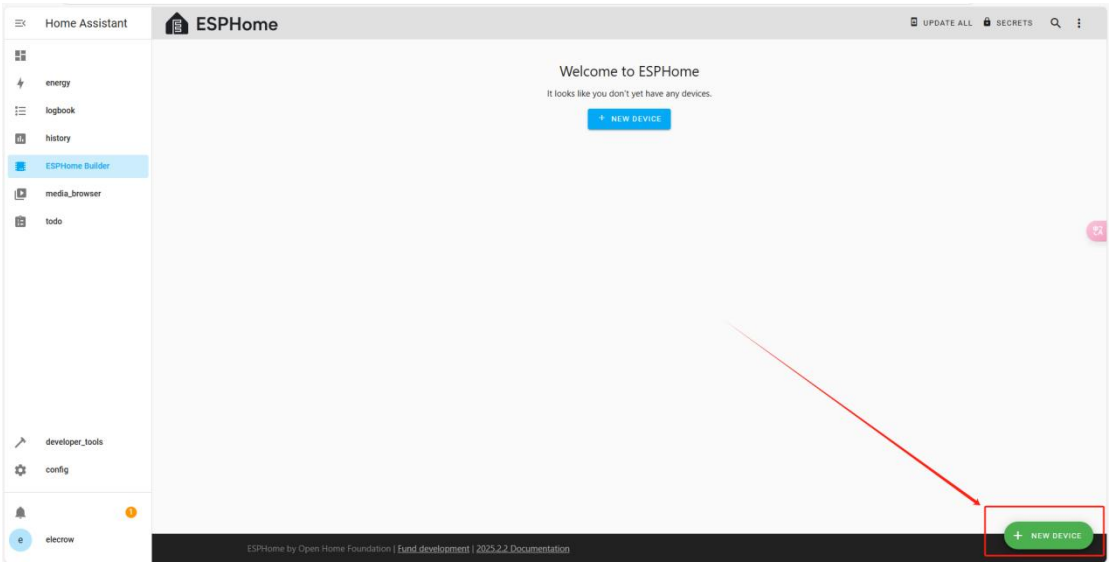
③ Raspberry Pi with the Home Assistant system

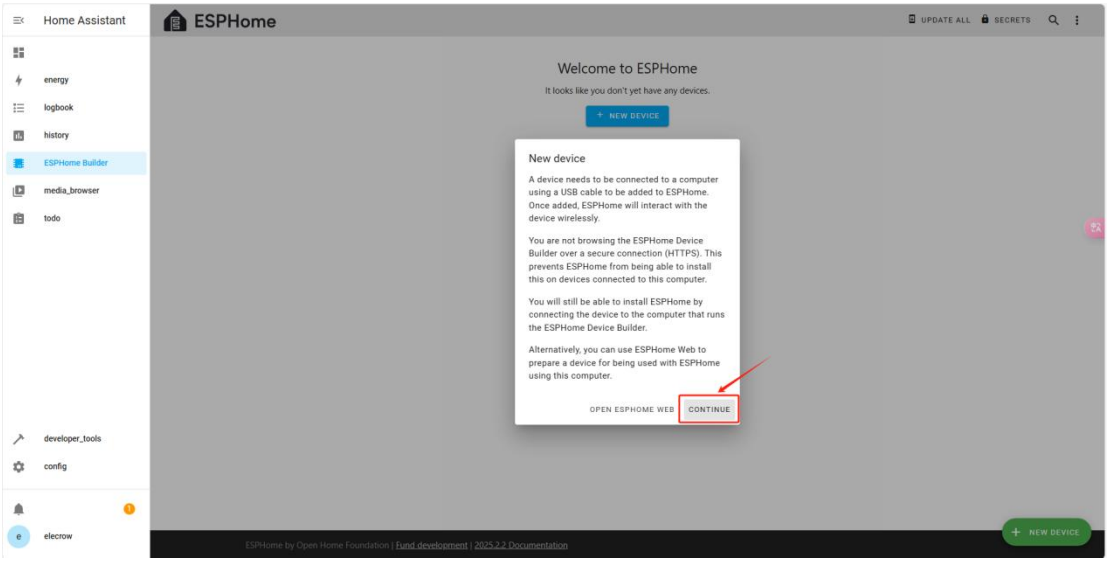
Here, the Raspberry Pi with the Home Assistant system acts as the server in our project. Therefore, whenever we access the IP address of the Raspberry Pi with the Home Assistant system, we are actually entering Home Assistant. In other words, you must first enter this page before you can proceed with the following operations.



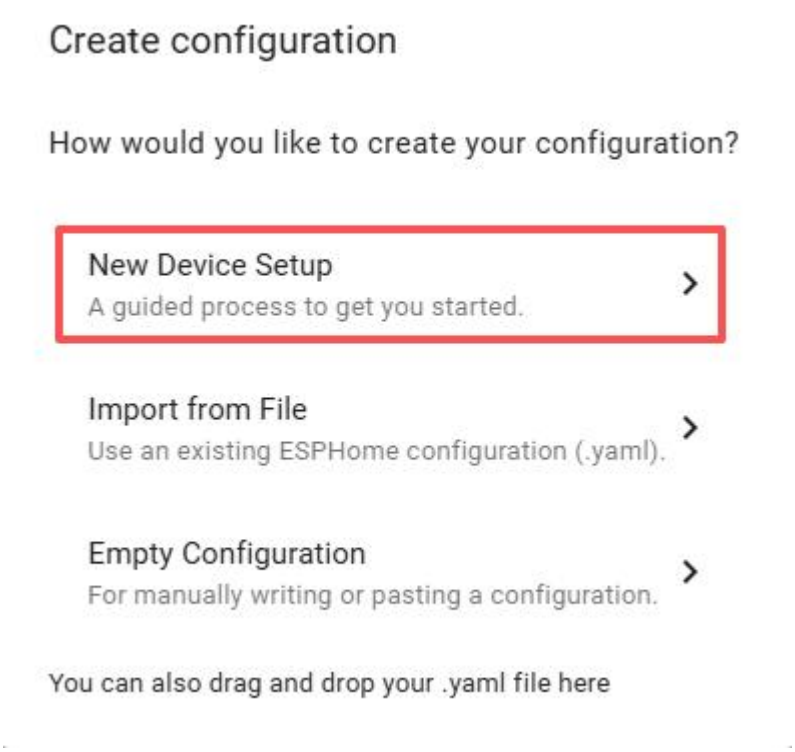
The following operations assume that you have already completed the installation steps in the previous lesson and have entered the main page of ESPHome.

Once the installation is complete, we can start adding devices. Click on **+ New Device** -> **Continue**.



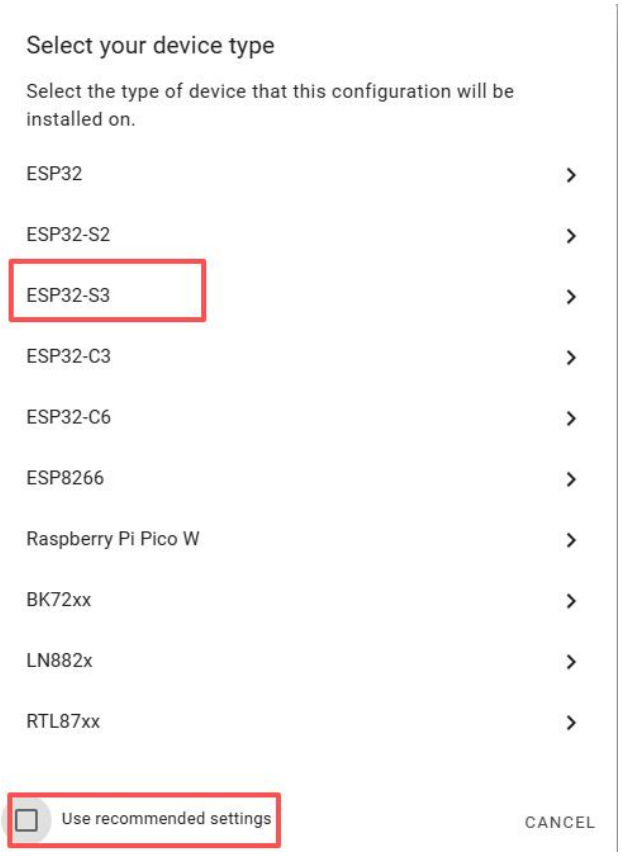


Click “New Device Setup”

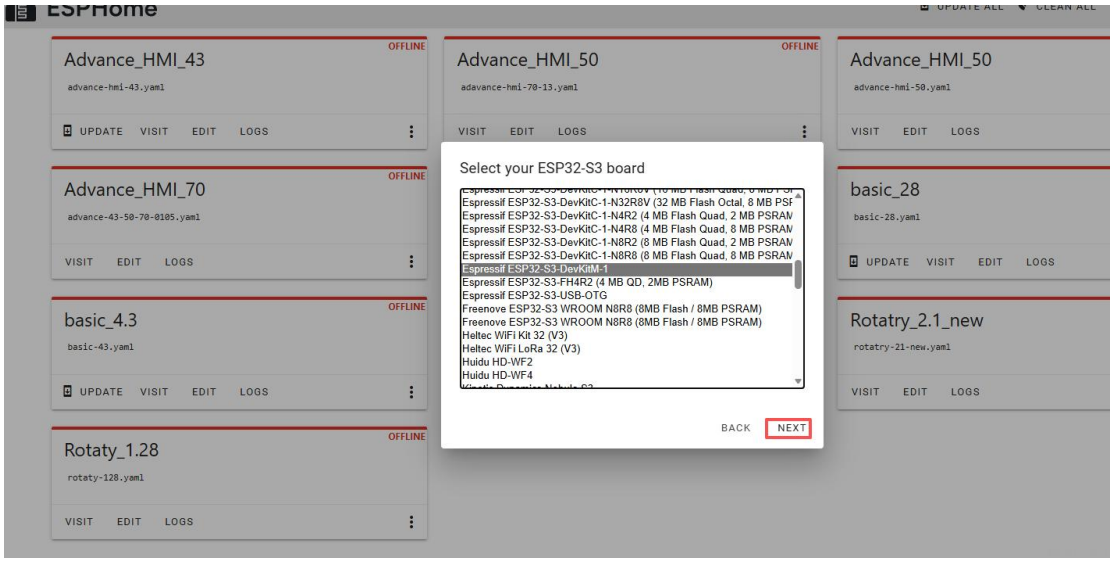


Enter a name and click **Next**.
(You can use any custom name. Do not include any strange symbols such as @, #, etc.)

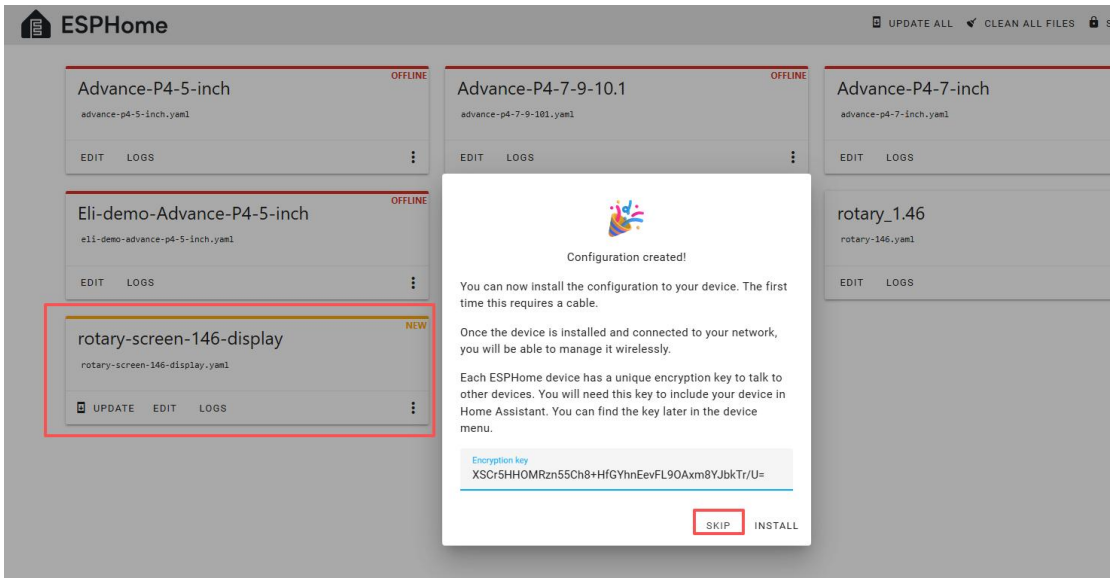
Here, do not check "Use recommended settings."
Select the main chip of the CrowPanel 1.46inch-HMI ESP32 Rotary Display , **ESP32-S3**.



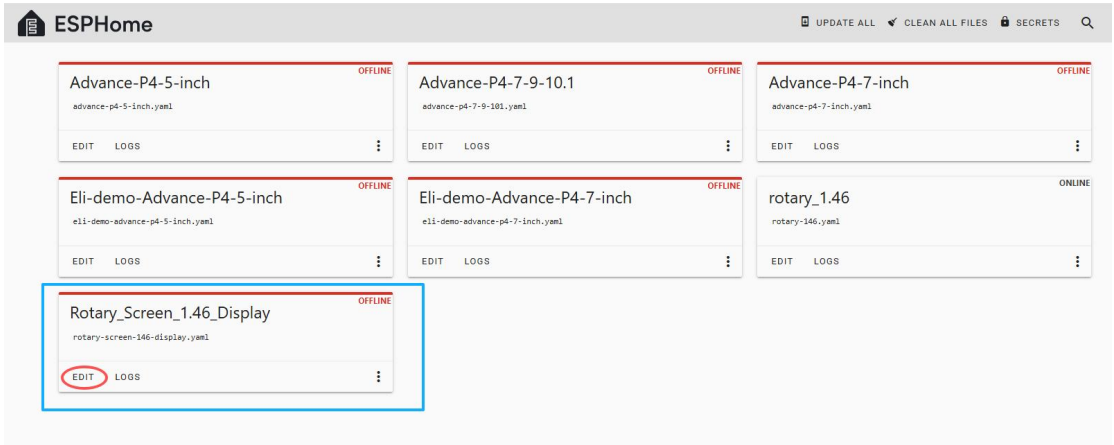
Next, choose any option (since we will replace it in the code later).



Here, click **"SKIP"**.



Then, return to the main interface, find the **Rotary_Screen_1.46_Display** you just created, click **"EDIT"**, and enter the code editor.



This code is automatically generated based on the previous steps. Next, we will make replacements in it, which will help optimize the code for more efficient operation.

Automatically generated code:

rotary-screen-146-display.yaml

```yaml
esphome:
  name: rotary-screen-146-display
  friendly_name: Rotary_Screen_1.46_Display

esp32:
  board: esp32-s3-devkitc-1
  framework:
    type: esp-idf

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "XSCr5HHOMRzn55Ch8+HfGYhnEevFL9OAxm8YJbkTr/U="

ota:
  - platform: esphome
    password: "b0a0f8876c261e9413c3036c357d0675"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "Rotary-Screen-146-Display"
    password: "U51T8AKI6joe"

captive_portal:
```

This is the complete code. You can click on the code link below to obtain it.

https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/blob/master/example/V1.0/ESPHome/Lesson_Code/rotary-screen-146-display.yaml

Next, you can replace the relevant content in **esphome** and **ESP32** as needed.And add the item **"PSRAM".**

× **rotary-screen-146-display.yaml**

```yaml
esphome:
  friendly_name: Rotary_Screen_1.46_Display
  platformio_options:
    build_flags: "-DBOARD_HAS_PSRAM"
    board_build.esp-idf.memory_type: qio_opi
    board_build.flash_mode: dio
  on_boot:
    priority: 900
    then:
      - logger.log: "POWER ON"
      - output.turn_on: VCC_5
      - switch.turn_on: Vcc_3_Switch
      - output.turn_on: Red_LED
      - delay: 150ms
      - light.turn_on:
          id: display_backlight
          brightness: 100%

esp32:
  board: esp32-s3-devkitc-1
  flash_size: 16MB
  framework:
    type: esp-idf
    sdkconfig_options:
      CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240: y
      CONFIG_ESP32S3_DATA_CACHE_64KB: y
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y
      CONFIG_SPIRAM_RODATA: y

psram:
  mode: octal
  speed: 80MHz

# Enable logging
logger:

debug:
  update_interval: 30s

# Enable Home Assistant API
api:
  encryption:
    key: "XSCr5HHOMRzn55Ch8+HfGYhnEevFL9OAxm8YJbkTr/U="

```

Then, based on your existing code, follow the format shown in the figure to add or replace these codes.
(Other configurations remain unchanged)

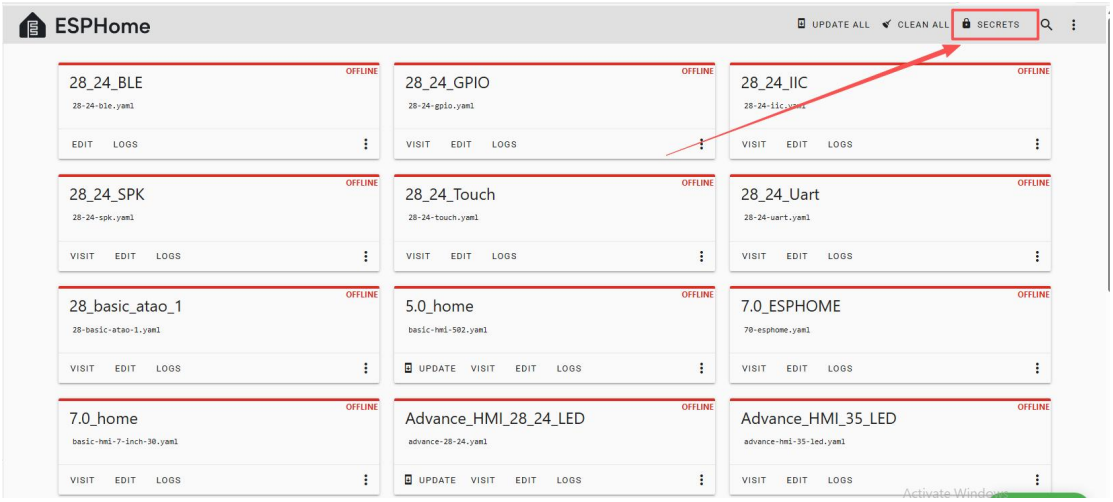
Remember to replace your own Wi-Fi name and password.

Note: This Wi-Fi connection must be in the same local network as your computer and Raspberry Pi!

× **rotary-screen-146-display.yaml**

```yaml
# Enable Home Assistant API
api:
  encryption:
    key: "XSCr5HHOMRzn55Ch8+HfGYhnEevFL9OAxm8YJbkTr/U="

ota:
  - platform: esphome
    password: "b0a0f8876c261e9413c3036c357d0675"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "Rotary-Screen-146-Display"
    password: "U51T8AKI6joe"

captive_portal:

```

Here, our Wi-Fi account and password are configured on the ESPHome main page.



So we can write it like this in the code, which means we are using the Wi-Fi network named "elecrow888".

Of course, you can also directly write the Wi-Fi SSID and password in detail inside the code. If you do not write them explicitly, the system will use the default Wi-Fi settings configured above.

rotary-screen-146-display.yaml

```
# Enable Home Assistant API
api:
  encryption:
    key: "XSCr5HHOMRzn55Ch8+HfGYhnEevFL9OAxm8YJbkTr/U="

ota:
  - platform: esphome
    password: "b0a0f8876c261e9413c3036c357d0675"

wifi:
  ssid: "elecrow888"
  password: "elecrow2014"

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "Rotary-Screen-146-Display"
    password: "U51T8AKI6joe"

captive_portal:
```

Note: The following items are unique to the device you created, so they should not be exactly the same as mine. You can simply keep the original values generated during your device creation process.

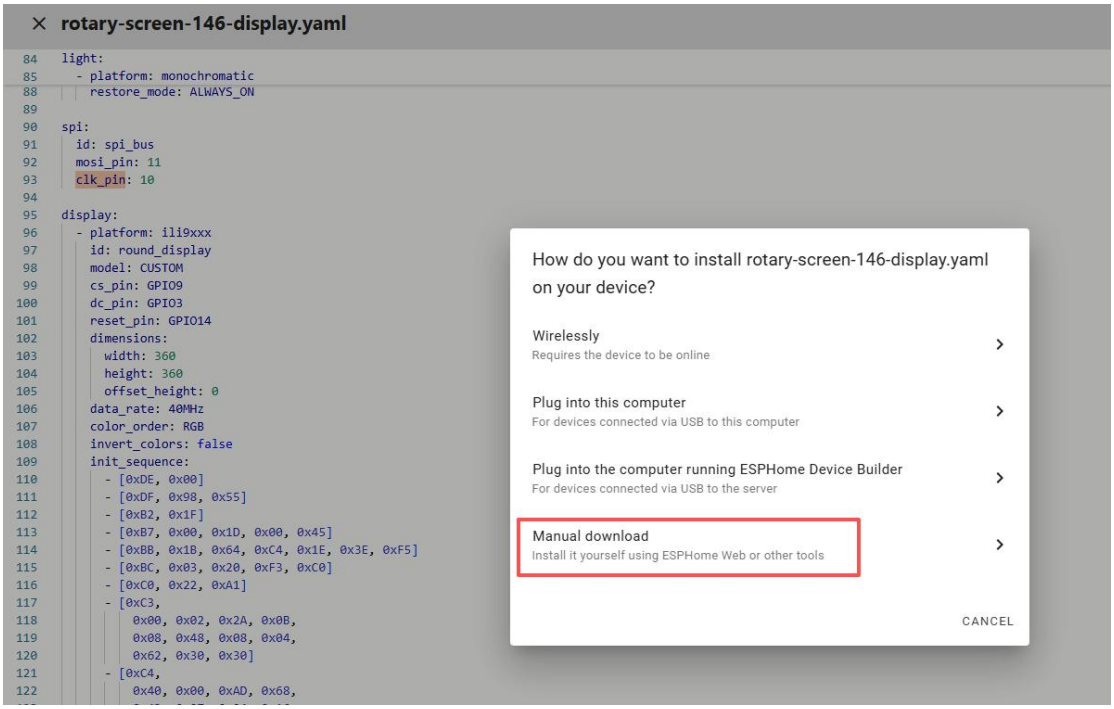
For the other functional code sections, you can copy mine directly.

rotary-screen-146-display.yaml

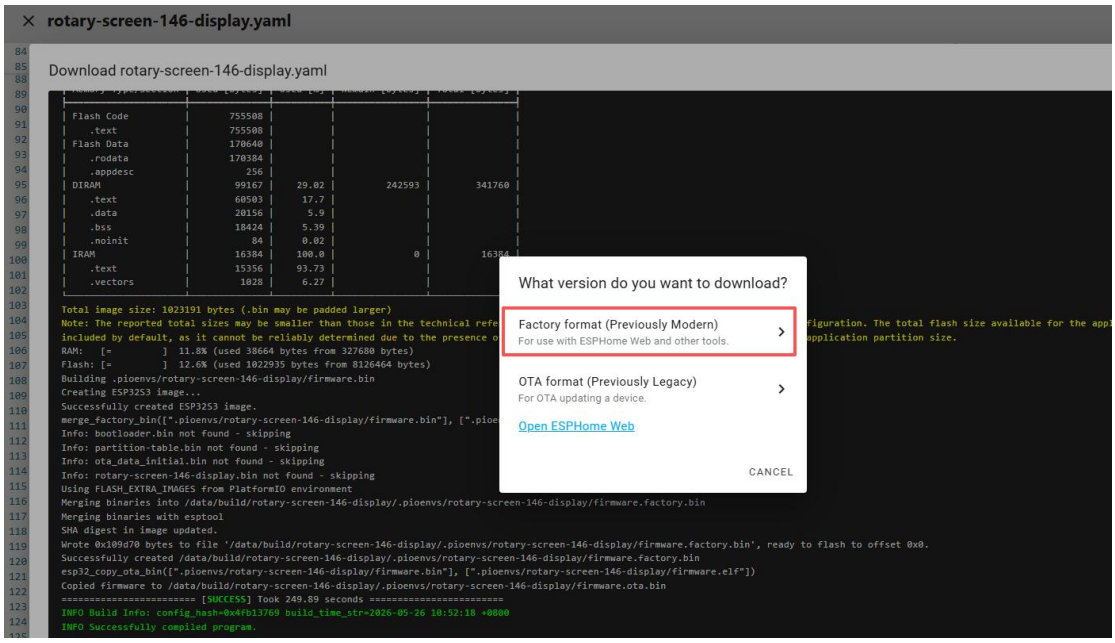
```
# Enable logging
logger:

debug:
  update_interval: 30s

# Enable Home Assistant API
api:
  encryption:
    key: "XSCr5HHOMRzn55Ch8+HfGYhnEevFL9OAxm8YJbkTr/U="

ota:
  - platform: esphome
    password: "b0a0f8876c261e9413c3036c357d0675"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "Rotary-Screen-146-Display"
    password: "U51T8AKI6joe"

captive_portal:
```

Once the code replacement is complete, click **"INSTALL"** in the top right corner.

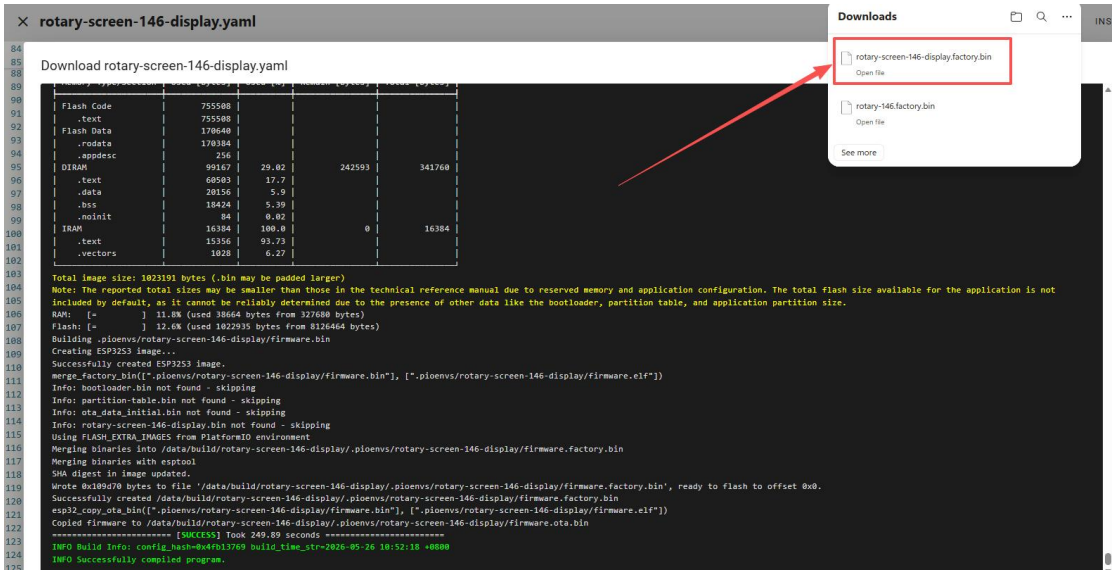
Select **"Manual download"**.



Wait for a few minutes until the installation is complete.
Then, select **"Factory format"**.

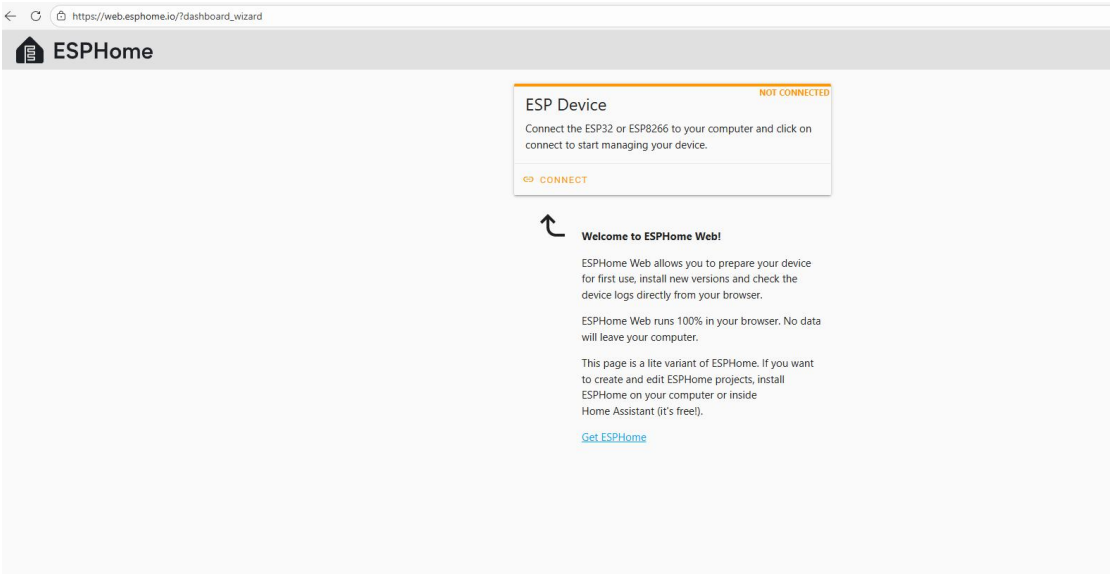


Once the download is complete, you will see the **.bin** file.



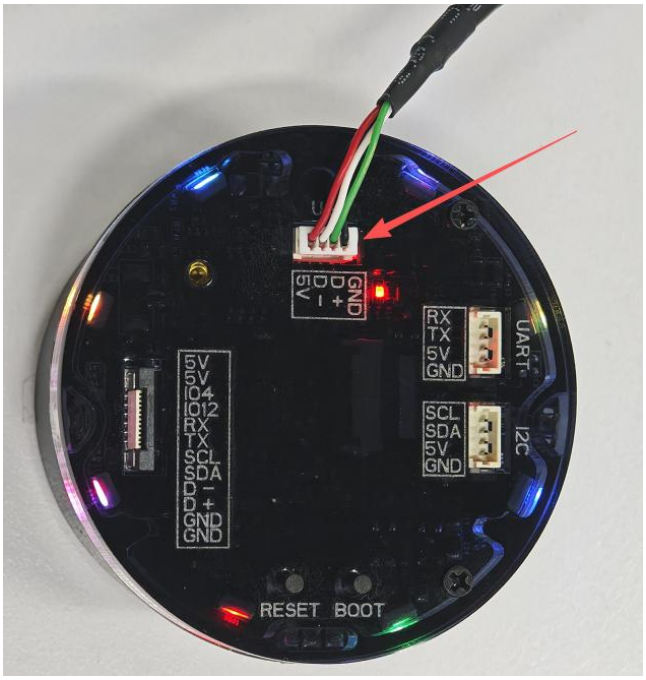
Remember the path of this **.bin** file.
Open the following website: https://web.esphome.io/?dashboard_wizard

After opening this website, you will arrive at this interface:

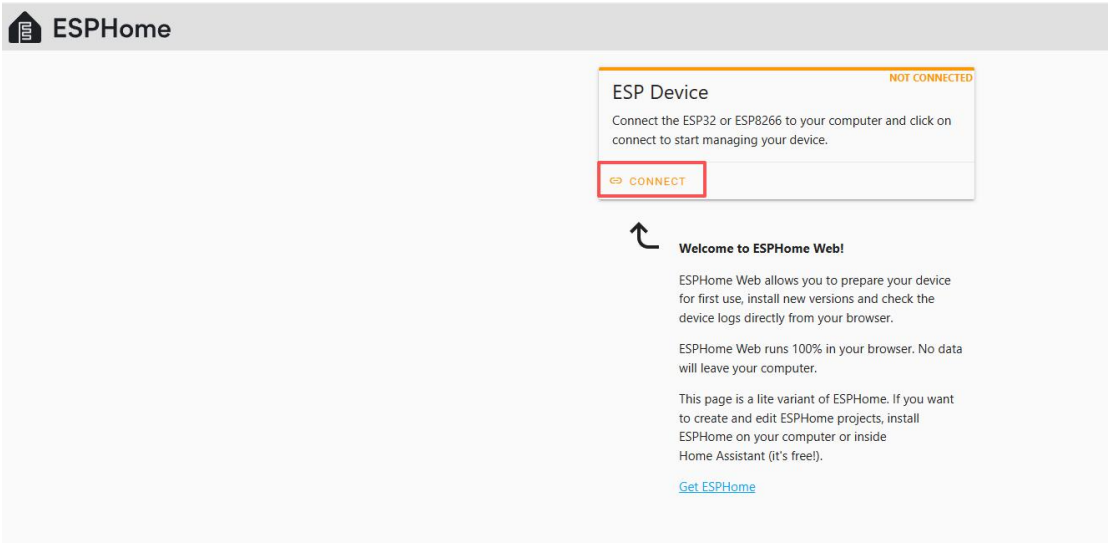


Next, we will flash this **.bin** file into the CrowPanel 1.46inch-HMI ESP32 Rotary Display .

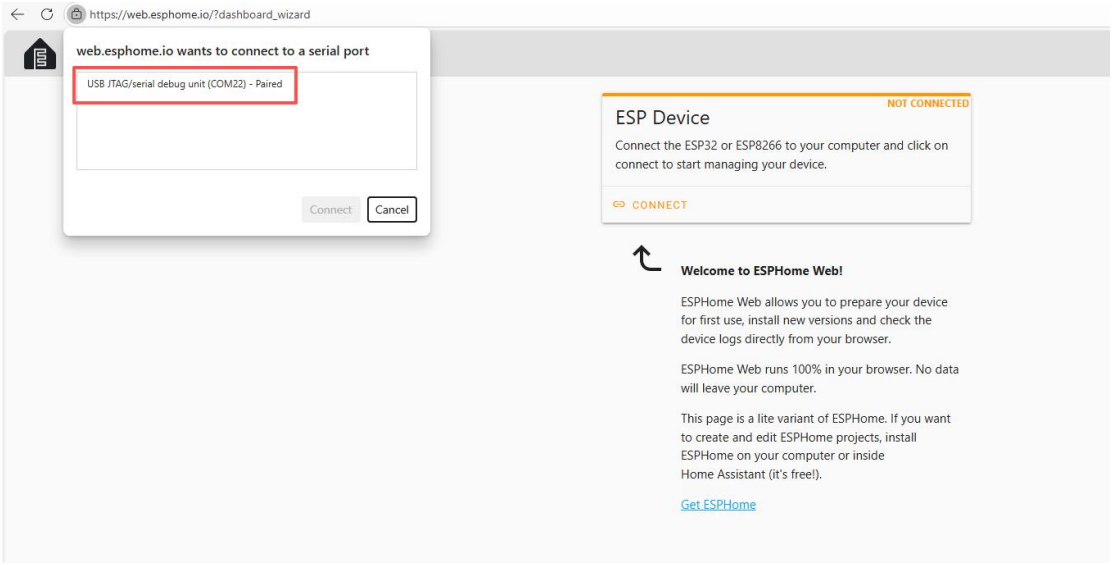
Connect the CrowPanel 1.46inch-HMI ESP32 Rotary Display to your computer.



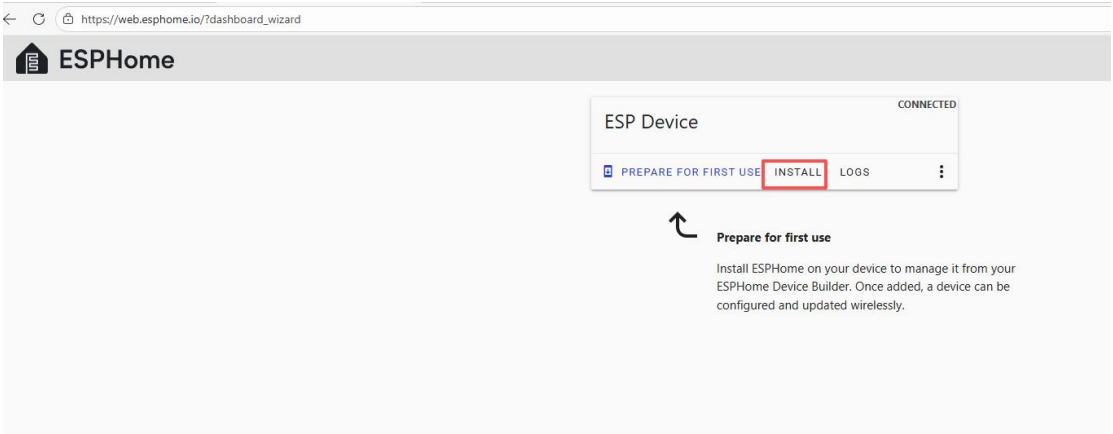
Click **"Connect"**



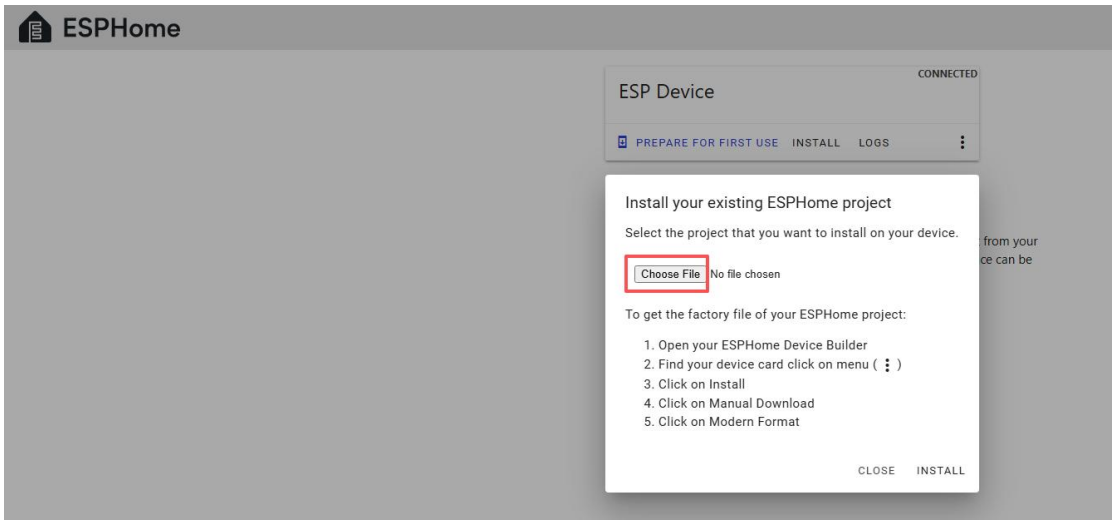
select the COM port, and connect it.



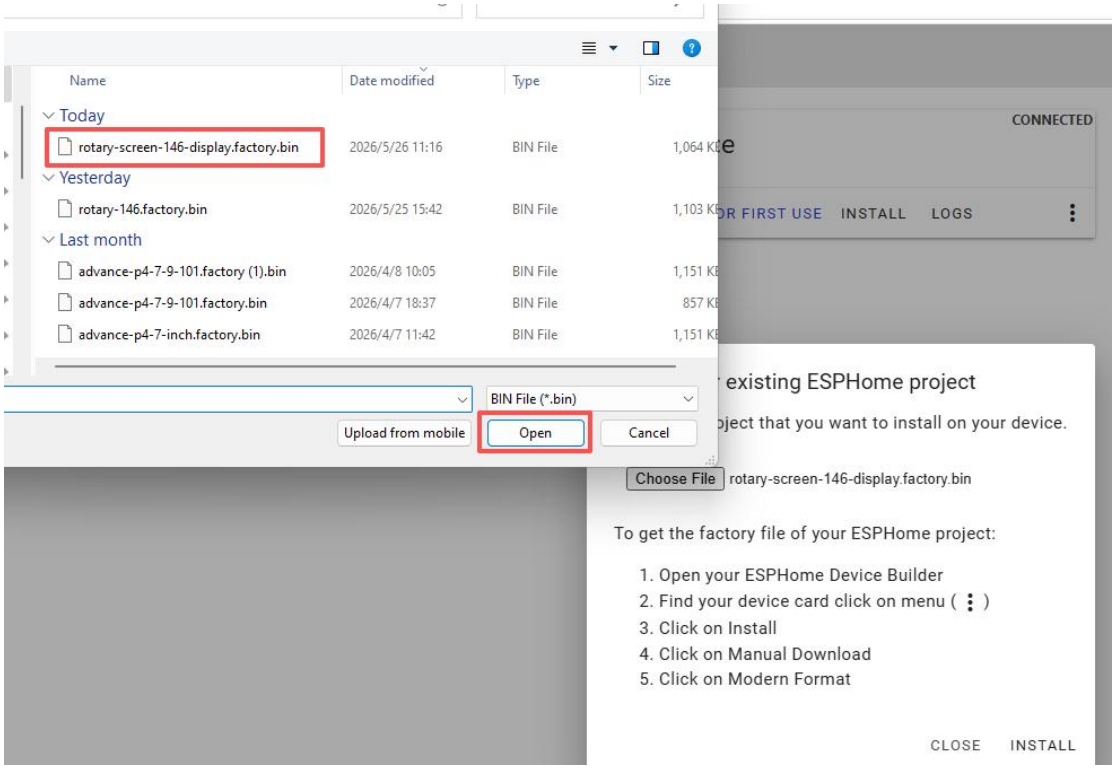
After connecting the CrowPanel 1.46inch-HMI ESP32 Rotary Display , click **"Install"**.



Add the **.bin** file you just downloaded, then click **"Install"**.

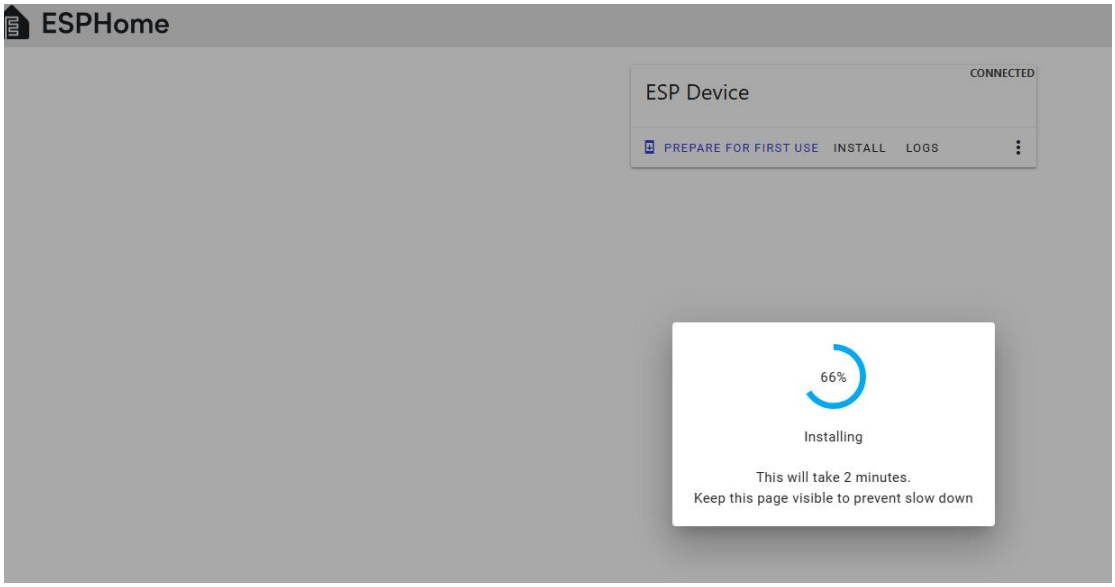


Then select the bin file that you just downloaded.

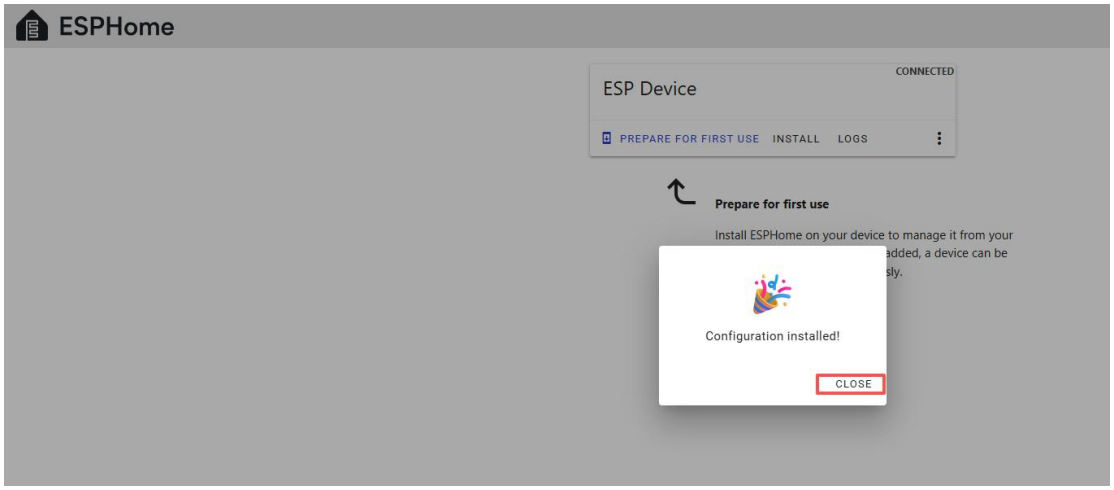


Click "INSTALL"

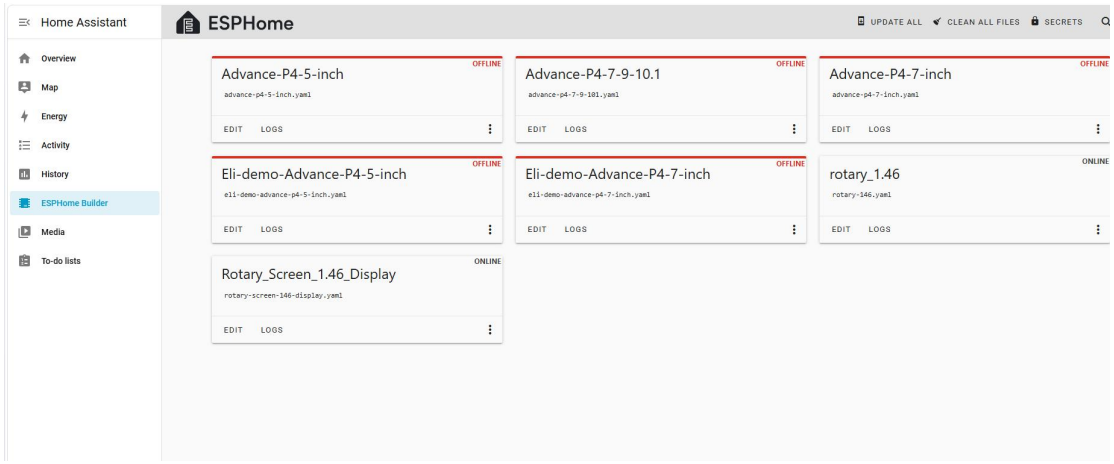
Wait for a few minutes.



After the installation is complete, click **"Close"**.

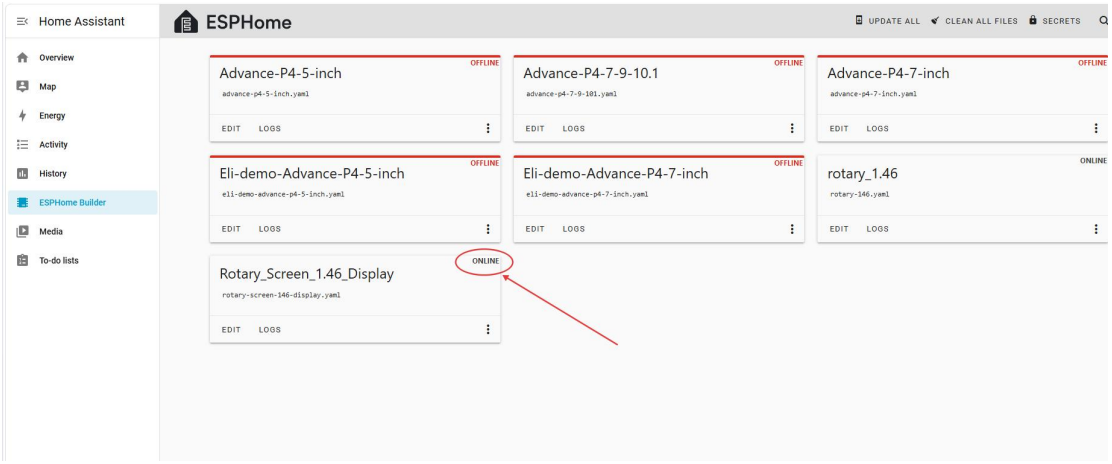


After successfully flashing the **.bin** file, return to the ESPHome page in Home Assistant.



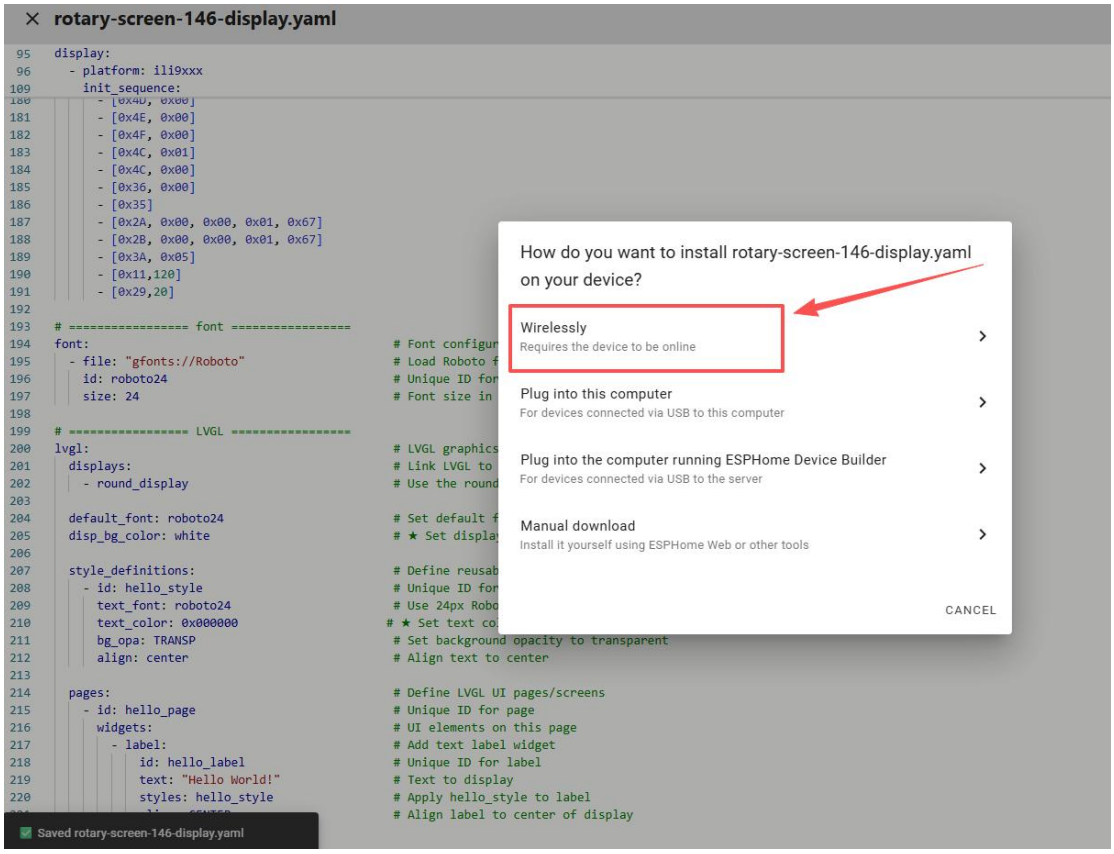
Press the RESET button on the CrowPanel 1.46inch-HMI ESP32 Rotary Display .

Restart the ESP32 display, and you should see the device you created earlier show as **ONLINE** in the top right corner.



And if you also add the functional code we provided, you will be able to see “Hello World!” displayed on the screen.

After it shows ONLINE here, whenever you make modifications to the code later and want to upload the code again, you can use the Wireless upload method to upload the code, which will be much more convenient.



Lesson03---Rotate the Knob to Adjust Screen Brightness

1. Course Introduction

In the previous lesson, we learned how to light up the rotary display and show text on the screen. In this lesson, we will build on that foundation to learn how to rotate the knob to adjust the screen brightness.

2. Learning Objectives

Learn how to drive the rotary function on the rotary display

Use the rotary function to control screen brightness

3. Project Demonstration

You can adjust the screen brightness by rotating the knob, and the current brightness value will be displayed on the screen in real time.

(Rotate clockwise to increase screen brightness, rotate counterclockwise to decrease screen brightness)

4. Example Code Download Link and Key Code Explanation

Click the GitHub link below to download the complete code:

https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/blob/master/example/V1.0/ESPHome/Lesson_Code/146-adjust-screen-brightness.yaml

Next, we will explain the key parts of this code and how to write code on the ESPHome platform to rotate the knob and control the screen brightness.

Some related code configurations below were already explained in detail in the previous lesson, so they will not be repeated here. We will only explain the new code knowledge related to this lesson's functionality.

(1) Initialization Process (esphome block)

This on_boot code implements the complete initialization process after the device powers on. Its main purpose is to ensure that the ESP32-S3 correctly initializes the power supply, display, backlight, rotary encoder, and LVGL interface in the proper order during startup.

After the system starts, it first outputs the "POWER ON" log for debugging confirmation, then enables the 5V power supply, 3.3V power switch, and red indicator LED to provide power for peripheral hardware (UART, IIC, and the screen), followed by a 150ms delay to stabilize the voltage.

Next, it sets the screen backlight to 100% to ensure the display starts correctly, then waits an additional 500ms to provide enough initialization time for the LCD and LVGL.

Afterward, the program enters the core initialization stage. It first disables encoder response functionality to prevent accidental knob operations during startup, while setting the brightness variable to the default 50%, and uses publish_state(0) to forcibly reset the internal count of the rotary encoder to zero, ensuring the software state remains synchronized with the actual hardware state and preventing sudden brightness changes after power-on.

Then it sets the actual screen backlight brightness to 50%, uses snprintf() to generate the "50%" string to update the LVGL label, and finally calls lv_refr_now(NULL) to force an immediate screen refresh so the current brightness value is displayed. After waiting 200ms to ensure the encoder state is fully synchronized, the encoder processing function is re-enabled, and the device officially enters normal operating mode.

```
on_boot:
  priority: 900
  then:
    - logger.log: "POWER ON"
    - output.turn_on: VCC_5
    - switch.turn_on: Vcc_3_Switch
    - output.turn_on: Red_LED
    - delay: 150ms
    - light.turn_on:
        id: display_backlight
        brightness: 100%
    # Brightness initialization logic
    - delay: 500ms

    # Initialize brightness and encoder (frozen state)
    - lambda: |-
        id(encoder_ready) = false;
        id(brightness_value) = 50;

        // Force rotary_encoder internal state alignment
        id(knob).publish_state(0);

    # Initial brightness (50%)
    - light.turn_on:
        id: display_backlight
        brightness: 0.5

    # Initialize LVGL text on startup (display 50%)
    - lambda: |-
        char buf[10];
        snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));
        lv_label_set_text(id(lbl_brightness), buf);
        lv_refr_now(NULL);
    - delay: 200ms
    # Enable encoder after initialization
    - lambda: |-
        id(encoder_ready) = true;
```

Key points of this code section:

```
8   on_boot:
9     priority: 900
10    then:
11      - logger.log: "POWER ON"
12      - output.turn_on: VCC_5
13      - switch.turn_on: Vcc_3_Switch
14      - output.turn_on: Red_LED
15      - delay: 150ms
16      - light.turn_on:
17          id: display_backlight
18          brightness: 100%
19      # Brightness initialization logic
20      - delay: 500ms
```

on_boot is the startup hook in ESPHome, and priority indicates the execution order of this startup task. The larger the value, the earlier it executes.

```
8   on_boot:
9     priority: 900
10    then:
11      - logger.log: "POWER ON"
12      - output.turn_on: VCC_5
13      - switch.turn_on: Vcc_3_Switch
14      - output.turn_on: Red_LED
15      - delay: 150ms
16      - light.turn_on:
17          id: display_backlight
18          brightness: 100%
19      # Brightness initialization logic
20      - delay: 500ms
```

Turn on the screen backlight display_backlight, turn on the power indicator Red_LED, and enable the screen power pin and interface power pins Vcc_3_Switch and VCC_5. (Explained in detail in the previous lesson)

Delay 500ms to wait for hardware initialization to complete successfully.

```
27      # === Initialize brightness and encoder (frozen state) ===
28      - lambda: |-
29          id(encoder_ready) = false;  // Disable encoder processing
30          id(brightness_value) = 50;  // Set internal brightness value to 50%
31
32          // Force rotary_encoder internal state to align with 50
33          id(knob).publish_state(0);
```

encoder_ready = false: Temporarily disable brightness modification triggered by rotary events.

brightness_value = 50: Set the internal software brightness variable to 50%, matching the PWM output.

publish_state(0): Reset the internal count state of rotary_encoder to 0, ensuring the rotary logic is synchronized with the software variable.

The purpose of this design is to prevent sudden "jumping" during rotary encoder initialization, such as the brightness suddenly jumping to 0 or 100. It ensures that the "rotary position, internal brightness value, and actual PWM brightness" are completely aligned. This is a commonly used safe initialization strategy in many rotary screen brightness systems.

```
40      # Initialize LVGL brightness label at boot (show 50%)
41      - lambda: |-
42          char buf[10];               // Temporary buffer for text
43          snprintf(buf, sizeof(buf), "%d%%", id(brightness_value)); // Format percentage
44          lv_label_set_text(id(lbl_brightness), buf); // Update LVGL label
45
46      # Wait to ensure publish_state takes effect
47      - delay: 200ms
48
49      # Enable encoder after everything is aligned
50      - lambda: |-
51          id(encoder_ready) = true;   // Allow encoder events to be processed
```

The purpose of this code is to initialize both the brightness variable and the percentage text displayed on the screen to 50% during startup, display it through the LVGL label "lbl_brightness", then wait 200ms for the display and internal state to stabilize, and finally set "encoder_ready" to "true" to unlock the rotary encoder so it can respond to user operations. This design ensures that the screen display and internal brightness value remain synchronized during startup while also preventing accidental rotary operations during initialization from causing brightness jumps, achieving smooth and safe alignment between interface display and operation logic.

(2) Global Variables

Please note again: many sections of the code in the middle were already explained in detail in the previous lesson. This lesson only explains the parts related to this lesson's functionality.

This "globals" code defines two global variables: "brightness_value" is used to store the current screen brightness integer value (0–100), with an initial value of 50, meaning the default startup brightness is 50%; "encoder_ready" is a boolean flag used to control whether the rotary encoder can respond to user operations. Its initial value is "false", meaning the rotary encoder is temporarily disabled during startup or initialization to prevent accidental operations. This design allows the system to first complete brightness and display initialization during startup, then safely allow the user to adjust the rotary encoder.

```
# Global variables (defined only, assigned in code)
globals:
  - id: brightness_value
    type: int                       # Integer brightness value (0-100)
    restore_value: false            # Do not restore from flash
    initial_value: "50"             # Default brightness value

  - id: encoder_ready
    type: bool                      # Flag to enable encoder processing
    restore_value: false            # Do not restore from flash
    initial_value: "false"          # Disabled by default
```

(3) Configuring the Rotary Encoder

This code implements the entire rotary encoder input detection and screen brightness control functionality. It reads the A and B phase signals of the rotary encoder connected to GPIO45 and GPIO42 through the rotary_encoder platform, and enables internal pull-up resistors to ensure stable input. resolution: 1 means the value updates once for every encoder step detected. debounce: 20ms is used for debounce processing to reduce false triggering caused by mechanical vibration, while round(x) converts the encoder reading into an integer to improve counting stability. When the encoder value changes, the on_value logic is triggered. It first checks whether the system initialization is complete to prevent accidental operations during startup; then it records the encoder's initial position and calculates the difference between the current value and the previous recorded value. The difference is used to determine clockwise or counterclockwise rotation direction, thereby increasing or decreasing screen brightness in 5% steps, while restricting the brightness within the 0%–100% range to prevent exceeding valid limits. Then the percentage brightness is converted into a PWM duty cycle, and the backlight output is directly modified through set_level(), enabling real-time brightness adjustment. At the same time, LVGL updates the percentage display on the interface and forces the display buffer to refresh, allowing users to see both the screen brightness and numerical value change synchronously while rotating the knob.

```
sensor:
  - platform: rotary_encoder
    id: knob
    name: "Encoder"
    pin_a:
      number: 45
      mode:
        input: true
        pullup: true
```

```
  pin_b:
    number: 42
    mode:
      input: true
      pullup: true
  resolution: 1

  filters:
    - debounce: 20ms
    - lambda: return round(x);

  on_value:
    then:
      - lambda: |-
            if (!id(encoder_ready)) return;

            static bool first_run = true;
            static int last_encoder = 0;

            int current = (int) id(knob).state;

            if (first_run) {
              last_encoder = current;
              first_run = false;
              return;
            }

            int delta = current - last_encoder;
            last_encoder = current;
            if (delta == 0) return;

            if (delta < 0) {
              id(brightness_value) -= 5;
            } else {
              id(brightness_value) += 5;
            }
```

```
if (id(brightness_value) < 0) id(brightness_value) = 0;
if (id(brightness_value) > 100) id(brightness_value) = 100;

float level = id(brightness_value) / 100.0f;
id(backlight).set_level(level);

char buf[10];
snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));
lv_label_set_text(id(lbl_brightness), buf);
lv_refr_now(NULL);
```

Key explanations in this code section:

```
# Sensor components
sensor:
  # Rotary encoder for brightness adjustment
  - platform: rotary_encoder                    # Rotary encoder sensor platform
    id: knob                                    # Unique ID for encoder sensor
    name: "Encoder"                             # Name for Home Assistant integration
    pin_a:                                      # Encoder A pin configuration
      number: 45                                # GPIO pin number for encoder A
      mode:
        input: true                             # Set pin as input
        pullup: true                            # Enable internal pullup resistor
    pin_b:                                      # Encoder B pin configuration
      number: 42                                # GPIO pin number for encoder B
      mode:
        input: true                             # Set pin as input
        pullup: true                            # Enable internal pullup resistor
    resolution: 1                               # Encoder resolution (steps per detent)

    filters:
      - debounce: 20ms                          # Debounce filter to reduce noise (20ms)
      - lambda: return round(x);                # Round encoder value to integer

    # Action to execute when encoder value changes
    on_value:
      then:
        - lambda: |-
            if (!id(encoder_ready)) return;         // Ignore input if encoder not ready (init phase)

            static bool first_run = true;           // Static flag for first encoder update
            static int last_encoder = 0;            // Static storage for previous encoder value
```

This code defines a rotary encoder sensor used to read the rotation state of the physical knob, assigns it a unique ID "knob", names it "Encoder", and marks it as an internal sensor (only called locally and not exposed to external platforms);

Then it configures the encoder's core hardware pins by binding the A phase (CLK clock line) to GPIO45 and the B phase (DT data line) to GPIO42. Both are configured as input mode to read encoder pulse signals, while internal pull-up resistors are enabled to prevent floating pins from causing voltage fluctuations and false signal triggering;

Then two levels of signal filtering are added: the 20ms debounce filter eliminates mechanical vibration noise generated during encoder rotation, and the value rounding filter converts the encoder's floating-point output value into an integer, simplifying subsequent calculations;

Finally, the encoder resolution is set to 1, meaning the value changes by only 1 step for each mechanical detent rotation, ensuring accurate signal reading and simplified calculation. Overall, this establishes the underlying hardware and signal processing foundation for the encoder to stably and accurately output rotation signals.

```
# Action to execute when encoder value changes
on_value:
  then:
    - lambda: |-
        if (!id(encoder_ready)) return;          // Ignore input if encoder not ready (init phase)

        static bool first_run = true;            // Static flag for first encoder update
        static int last_encoder = 0;             // Static storage for previous encoder value

        int current = (int) id(knob).state;      // Get current encoder state as integer

        if (first_run) {
          last_encoder = current;                // Set initial baseline for encoder value
          first_run = false;                     // Clear first run flag
          return;                                // Skip processing on first run
        }

        int delta = current - last_encoder;      // Calculate change in encoder position
        last_encoder = current;                  // Update last encoder value
        if (delta == 0) return;                  // Skip if no actual movement detected

        // Step adjustment (5% per encoder step)
        if (delta < 0) {
          id(brightness_value) -= 5;             // Decrease brightness by 5% for counter-clockwise
        } else {
          id(brightness_value) += 5;             // Increase brightness by 5% for clockwisea
        }

        // Clamp brightness range between 0-100%
        if (id(brightness_value) < 0) id(brightness_value) = 0;
        if (id(brightness_value) > 100) id(brightness_value) = 100;

        // Write PWM directly (no queue, no delay)
        float level = id(brightness_value) / 100.0f;    // Convert percentage to 0.0-1.0 range
        id(backlight).set_level(level);                 // Set PWM output level immediately

        // Update LVGL display (slower but non-blocking for brightness)
        char buf[10];                                                 // Character buffer for display text
        snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));     // Format as percentage string
        lv_label_set_text(id(lbl_brightness), buf);                   // Update brightness label text
        lv_refr_now(NULL);                                            // Force immediate display refresh

```

This code is the core on_value callback logic triggered when the rotary encoder value changes in the ESPHome framework. It fully implements the entire process from encoder signal parsing to backlight brightness adjustment and LVGL interface synchronization updates, while also including multiple fault-tolerance and initialization protection mechanisms. The logic can be broken down as follows:

First, initialization protection is implemented through the global boolean variable encoder_ready. If the system has not completed startup initialization (this variable is false), all encoder processing logic is skipped directly to avoid accidental operations during startup;

Next, two static variables are defined (first_run, which marks the first execution, and last_encoder, which stores the previous encoder value). These variables retain their values across multiple callback executions. The current real-time encoder value is then read and converted into an integer;

If this is the first execution of the callback function, it only synchronizes last_encoder with the current encoder value, disables the first_run flag, and exits without making any brightness adjustments, ensuring encoder state alignment during initialization. Then it calculates the difference delta between the current value and the historical value (used mainly to determine rotation direction and step count), updates last_encoder with the current value for the next calculation, and exits directly if delta equals 0 (no valid rotation), thereby reducing unnecessary computations;

The rotation direction is determined based on whether delta is positive or negative. When rotating left (delta < 0), the global brightness variable brightness_value is reduced by 5%; when rotating right (delta > 0), it is increased by 5%. At the same time, conditional checks strictly clamp the brightness value within the valid range of 0–100% to prevent exceeding the brightness range supported by the hardware. Then the percentage brightness value is converted into a PWM level value from 0.0 to 1.0 and directly written to the backlight PWM output pin, enabling immediate hardware-level brightness control without queues or delays;

Finally, a temporary character buffer is created, the brightness value is formatted into a percentage string (such as "50%"), the text content of the brightness display label in the LVGL interface is updated, and lv_refr_now(NULL) is called to force a screen refresh, ensuring the UI display remains synchronized with the actual backlight brightness in real time.

Summary: You rotate the encoder → The encoder outputs A/B phase pulse signals → The ESP32 reads the signals and converts them into digital values → The code calculates the rotation direction / step count and adjusts the brightness value → The brightness value is converted into a PWM signal to control the backlight current → The screen brightness changes in real time, and at the same time, the LVGL interface synchronously displays the percentage of brightness.

The key point of this process is: The encoder only "tells the MCU the direction and step count of rotation", while the actual control of brightness is done by the PWM signal output by the ESP32 - the encoder is an "input device", the PWM output is a "control method", and they are connected through software logic, achieving the intuitive interaction of "rotating knob → brightness change".

The "lambda" usage here is an inline C++ code block syntax provided by ESPHome, which allows you to directly write C++ logic in YAML. It enables access to ESPHome-defined global variables (such as "id(brightness_value)") and sensor objects (such as "id(knob)"), allowing flexible handling of sensor events, calculations,

conditional judgments, and interface updates without needing to write separate external C++ files.

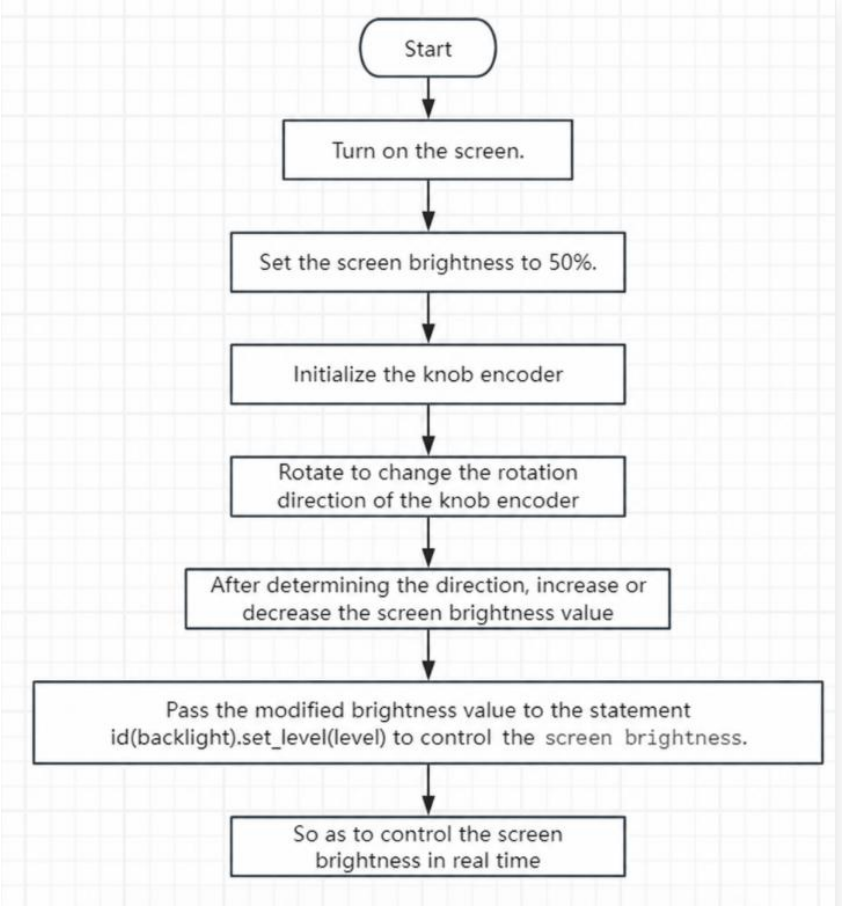
(4) LVGL Configuration

This code configures the use of the LVGL graphical interface library in ESPHome: it binds the "my_display" display to LVGL and allocates 50% of the buffer for interface rendering. Then it creates two label controls: the first label "lbl_title" is centered at the top of the screen and uses the "montserrat_28" font to display the title "Screen Brightness";

The second label "lbl_brightness" is centered below and uses the larger "montserrat_48" font to display the current brightness percentage, with the initial text set to "50%". This allows the screen to simultaneously display both the interface title and the brightness value controlled by the rotary knob, providing a user-friendly real-time feedback interface.

```
214  # LVGL (Light and Versatile Graphics Library) configuration
215  lvgl:
216    displays:
217      - round_display                          # Link LVGL to our display component
218    buffer_size: 50%                           # Use 50% of available RAM for LVGL buffer
219
220    # LVGL UI layout - brightness title and percentage
221    widgets:
222      # Title label widget
223      - label:
224          align: CENTER                        # Align text to center of display
225          id: lbl_title                        # Unique ID for title label
226          y: -30                               # Y position offset (30px up from center)
227          text_font: montserrat_22             # Font type and size (Montserrat 22pt)
228          text: "Screen Brightness"            # Default text for title
229
230      # Brightness percentage label widget
231      - label:
232          align: CENTER                        # Align text to center of display
233          id: lbl_brightness                   # Unique ID for brightness label
234          y: 20                                # Y position offset (20px down from center)
235          text_font: montserrat_48             # Font type and size (Montserrat 48pt)
236          text: "50%"                          # Default brightness text
```

5. Code Logic Flowchart



6. Flashing Steps

Next, we will teach you how to use ESPHome to write code, including the complete operation process. Please follow us step by step.

Here we emphasize again:

The following devices need to be on the same LAN:

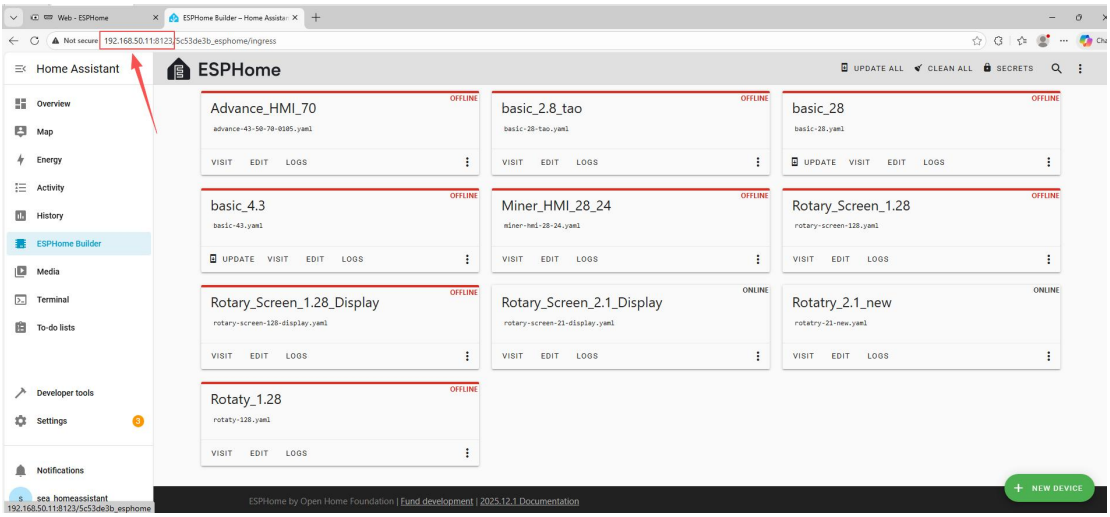
① Your computer

② Crowpanel HMI ESP32 Rotary Display

③ Raspberry Pi with the Home Assistant system

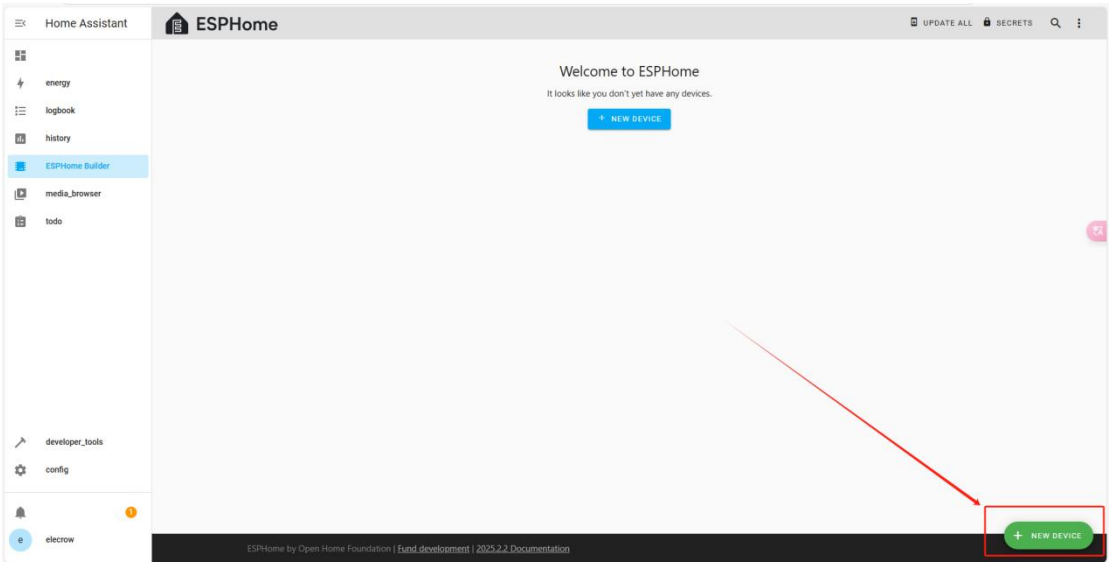
Here, our Raspberry Pi with the Home Assistant system acts as the server in the project, so every time we enter the IP address of this Raspberry Pi with the Home Assistant system, we are entering Home Assistant.

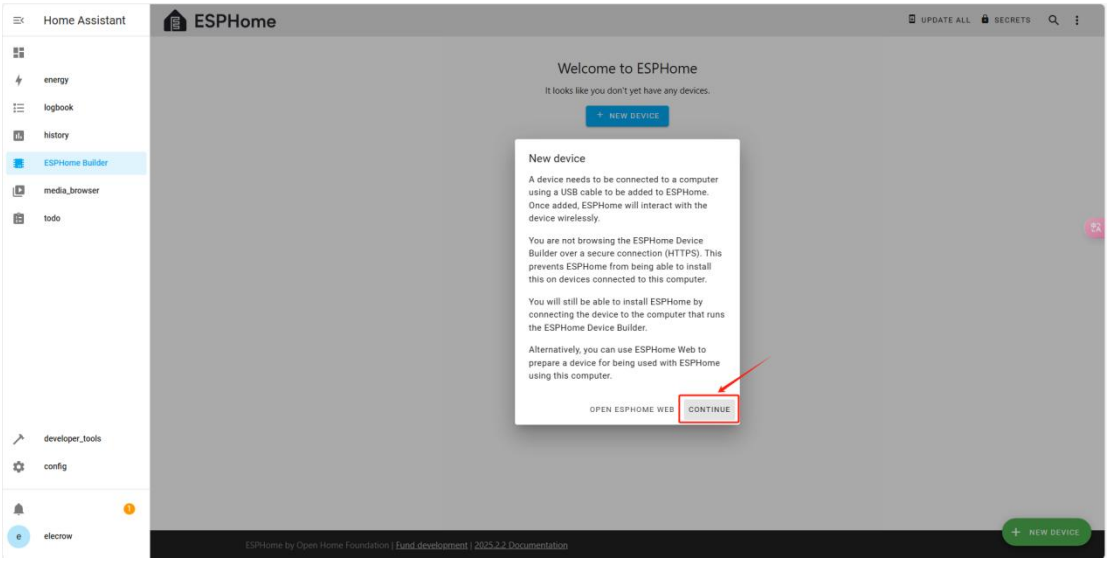
That means you must first enter this page before you can perform the following operations.



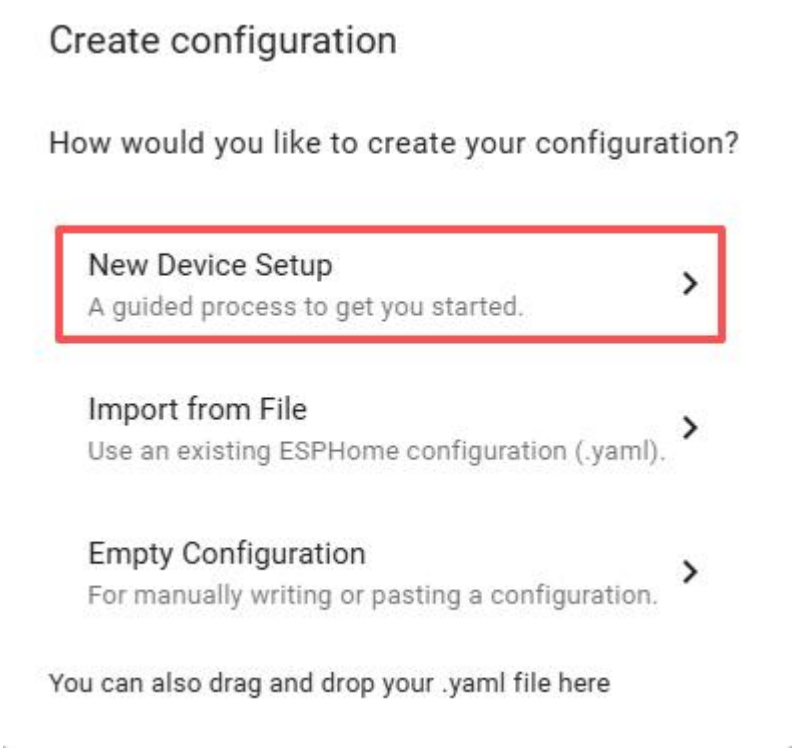
The following operations assume that you have already completed the installation steps in Lesson 1 and have arrived at the ESPHome main page.

Once the installation is complete, we can start adding devices. Click on **+ New Device** -> **Continue**.



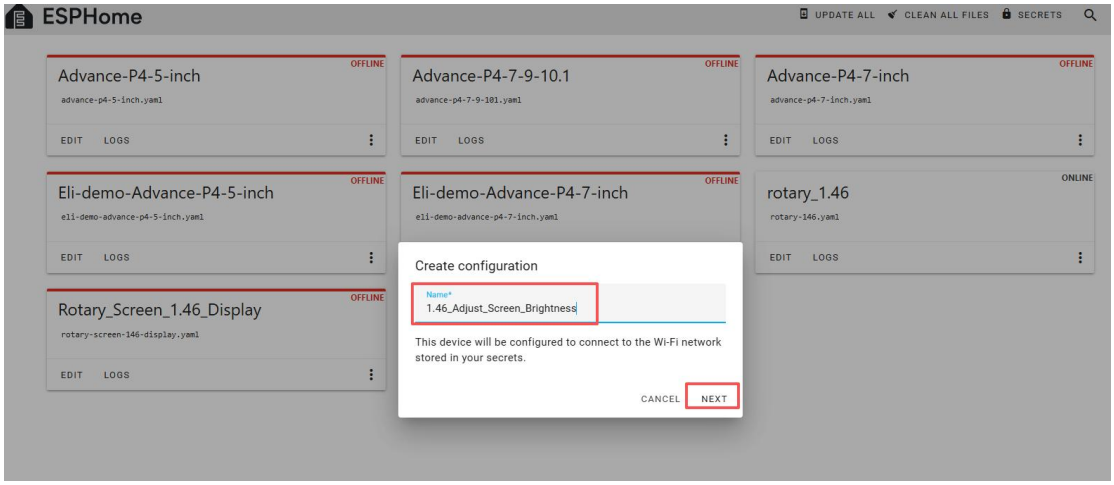


Click "New Device Setup"

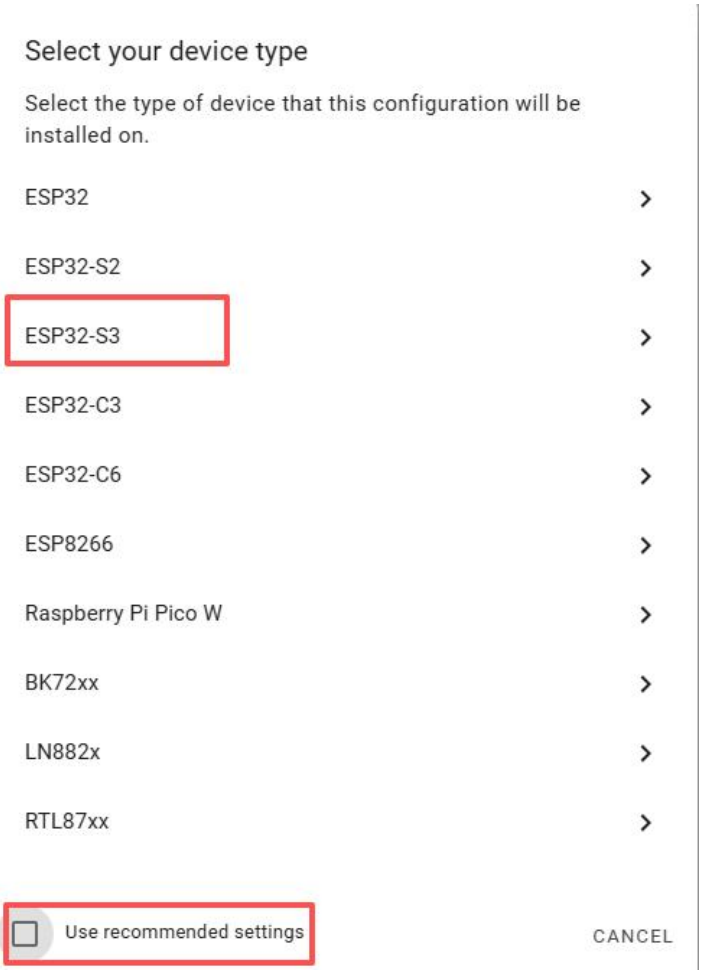


Enter a name and click **Next.**

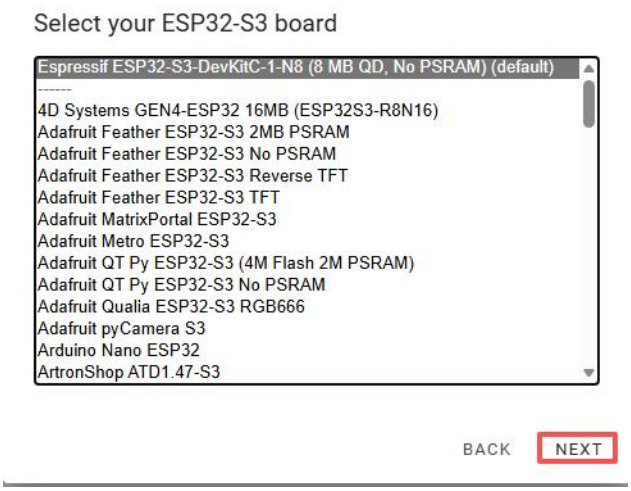
(You can use any custom name. Do not include any strange symbols such as @, #, etc.)



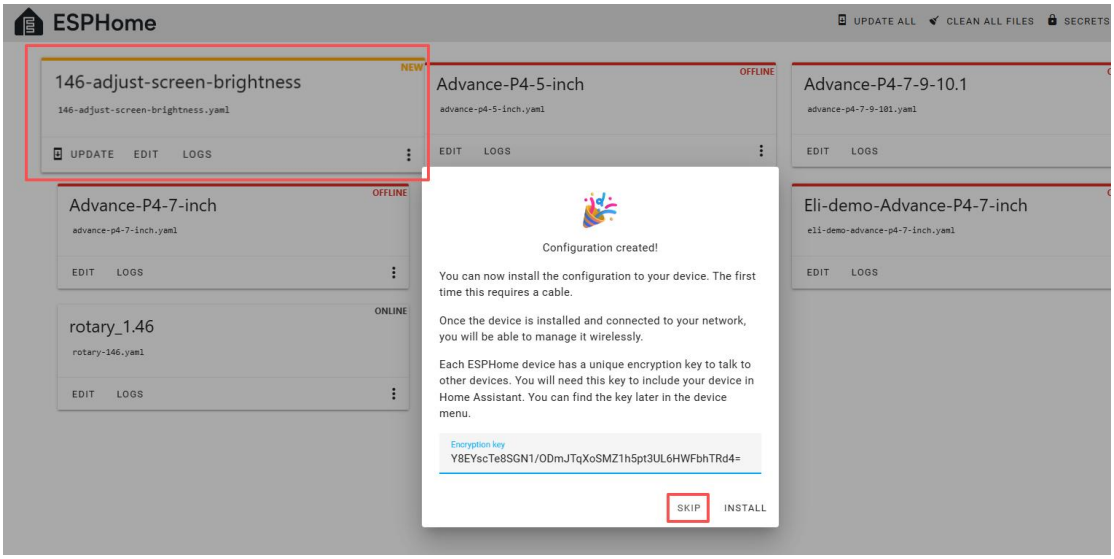
Here, do not check "Use recommended settings."
Select the main chip of the CrowPanel 1.46inch-HMI ESP32 Rotary Display , **ESP32-S3**.



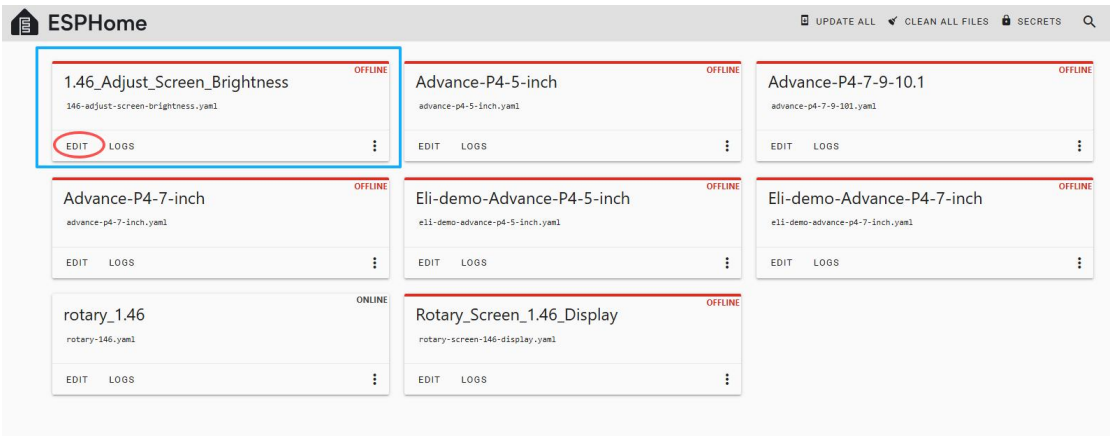
Next, choose any option (since we will replace it in the code later).



Here, click **"SKIP"**.



Then, return to the main interface, find the **1.46_Adjust_Screen_Brightness** you just created, click **"EDIT"**, and enter the code editor.



This code is automatically generated based on the previous steps. Next, we will make replacements in it, which will help optimize the code for more efficient operation.

Automatically generated code:

× 146-adjust-screen-brightness.yaml

```yaml
esphome:
  name: 146-adjust-screen-brightness
  friendly_name: 1.46_Adjust_Screen_Brightness

esp32:
  board: esp32-s3-devkitc-1
  framework:
    type: esp-idf

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "Y8EYscTe8SGN1/ODmJTqXoSMZ1h5pt3UL6HWFbhTRd4="

ota:
  - platform: esphome
    password: "e0d040a9f5eb7281aee59745d420d668"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "146-Adjust-Screen-Brightness"
    password: "Bf3LZhmWeoOt"

captive_portal:
```

This is the complete code. You can click on the code link below to obtain it.

https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/blob/master/example/V1.0/ESPHome/Lesson_Code/146-adjust-screen-brightness.yaml

Next, you can replace the relevant content in **esphome** and **ESP32** as needed.And add the item **"PSRAM".**

× 146-adjust-screen-brightness.yaml

```yaml
esphome:
  name: 146-adjust-screen-brightness
  friendly_name: 1.46_Adjust_Screen_Brightness
  platformio_options:
    build_flags: "-DBOARD_HAS_PSRAM"
    board_build.esp-idf.memory_type: qio_opi
    board_build.flash_mode: dio

esp32:
  board: esp32-s3-devkitc-1
  flash_size: 16MB
  framework:
    type: esp-idf
    sdkconfig_options:
      CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240: y
      CONFIG_ESP32S3_DATA_CACHE_64KB: y
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y
      CONFIG_SPIRAM_RODATA: y

psram:
  mode: octal
  speed: 80MHz

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "Y8EYscTe8SGN1/ODmJTqXoSMZ1h5pt3UL6HWFbhTRd4="

ota:
  - platform: esphome
    password: "e0d040a9f5eb7281aee59745d420d668"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password
```

Then, based on your existing code, follow the format shown in the figure to add or replace these codes.
(Other configurations remain unchanged)

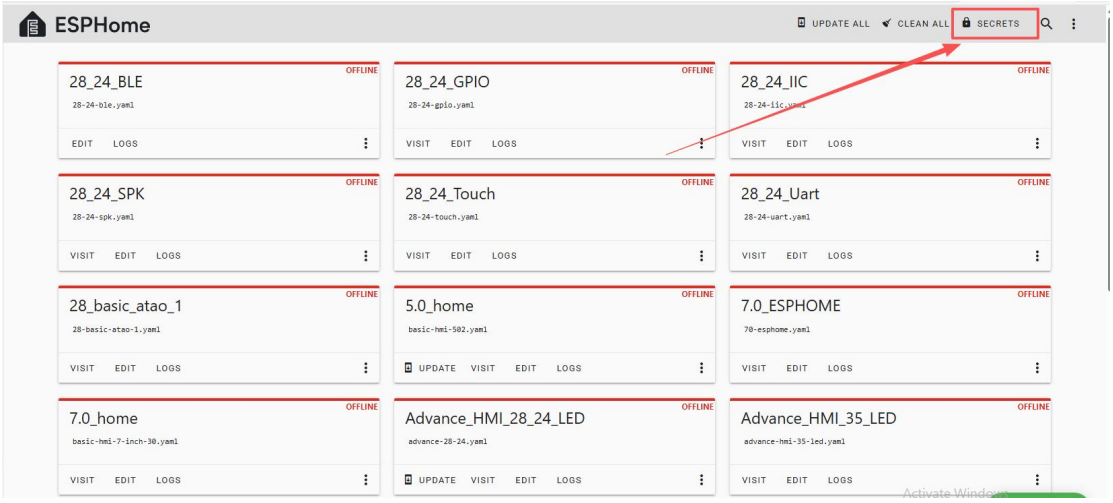
Remember to replace your own Wi-Fi name and password.

Note: This Wi-Fi connection must be in the same local network as your computer and Raspberry Pi!

146-adjust-screen-brightness.yaml

```yaml
esp32:
  framework:
    sdkconfig_options:
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y
      CONFIG_SPIRAM_RODATA: y

psram:
  mode: octal
  speed: 80MHz

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "Y8EYscTe8SGN1/ODmJTqXoSMZ1h5pt3UL6HWFbhTRd4="

ota:
  - platform: esphome
    password: "e0d040a9f5eb7281aee59745d420d668"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "146-Adjust-Screen-Brightness"
    password: "Bf3LZhmWeoOt"

captive_portal:
```

Here, our Wi-Fi account and password are configured on the ESPHome main page.



So we can write it like this in the code, meaning we are using the "elecrow888" Wi-Fi.

Of course, you can also write the Wi-Fi account and password explicitly in the code.

146-adjust-screen-brightness.yaml

```yaml
esp32:
  framework:
    sdkconfig_options:
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y
      CONFIG_SPIRAM_RODATA: y

psram:
  mode: octal
  speed: 80MHz

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "Y8EYscTe8SGN1/ODmJTqXoSMZ1h5pt3UL6HWFbhTRd4="

ota:
  - platform: esphome
    password: "e0d040a9f5eb7281aee59745d420d668"

wifi:
  ssid: "elecrow888"
  password: "elecrow2014"

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "146-Adjust-Screen-Brightness"
    password: "Bf3LZhmWeoOt"

captive_portal:
```

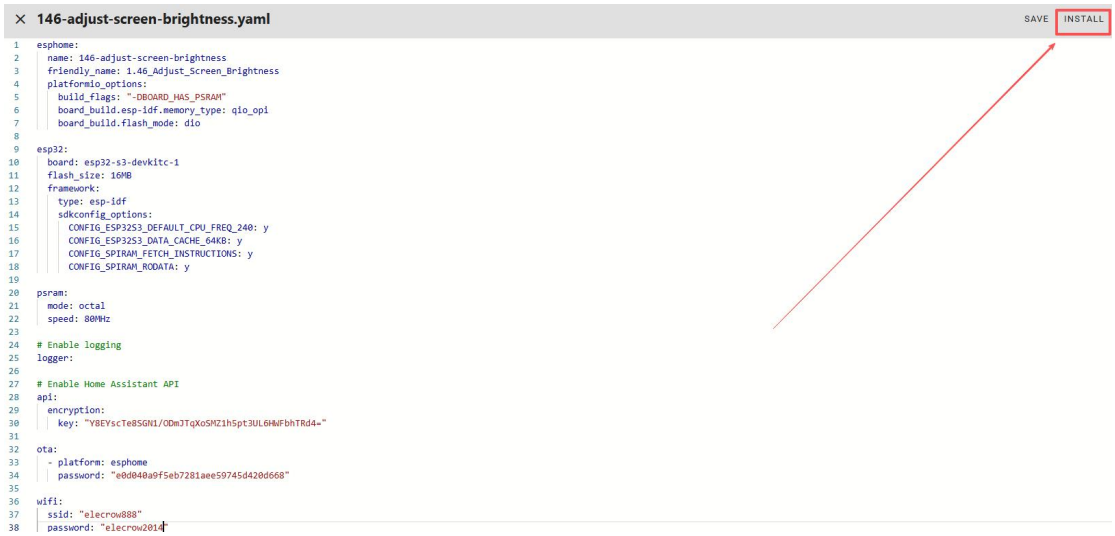
Note: These pieces of content are unique to the device you created. They cannot be the same as mine. Keep the original ones generated for your device.

You can copy my other functional code.

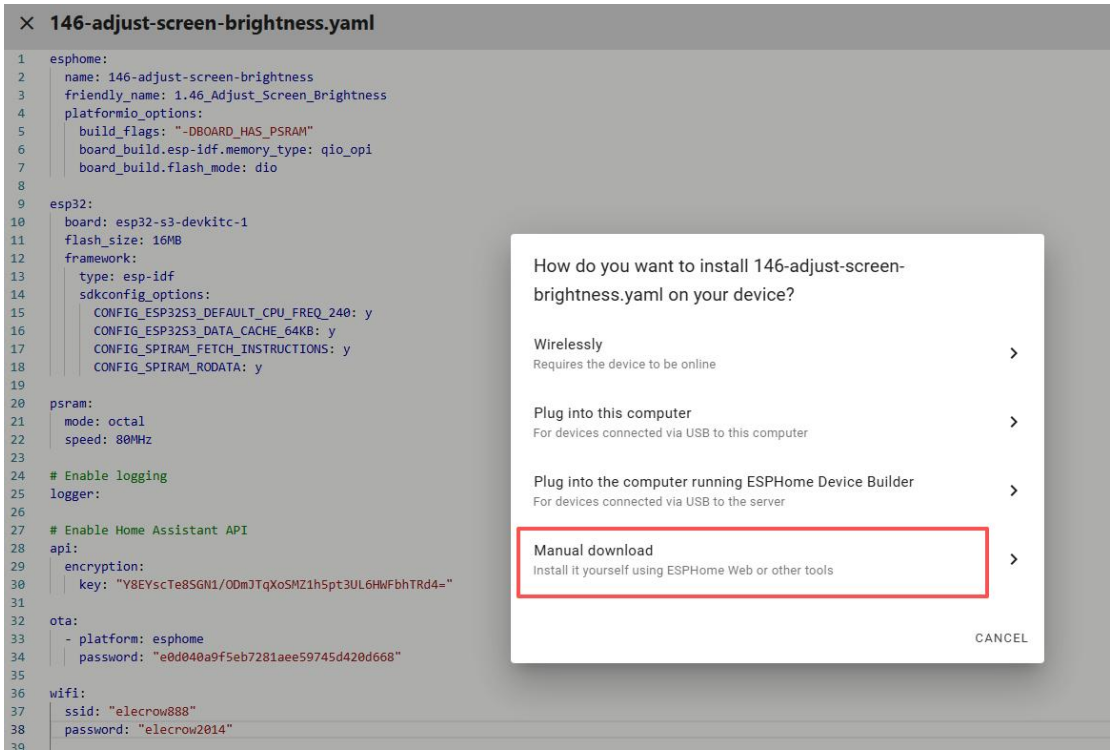
146-adjust-screen-brightness.yaml

```yaml
esphome:
  name: 146-adjust-screen-brightness
  friendly_name: 1.46_Adjust_Screen_Brightness
  platformio_options:
    build_flags: "-DBOARD_HAS_PSRAM"
    board_build.esp-idf.memory_type: qio_opi
    board_build.flash_mode: dio

esp32:
  board: esp32-s3-devkitc-1
  flash_size: 16MB
  framework:
    type: esp-idf
    sdkconfig_options:
      CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240: y
      CONFIG_ESP32S3_DATA_CACHE_64KB: y
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y
      CONFIG_SPIRAM_RODATA: y

psram:
  mode: octal
  speed: 80MHz

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "Y8EYscTe8SGN1/ODmJTqXoSMZ1h5pt3UL6HWFbhTRd4="

ota:
  - platform: esphome
    password: "e0d040a9f5eb7281aee59745d420d668"

wifi:
  ssid: "elecrow888"
  password: "elecrow2014"

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "146-Adjust-Screen-Brightness"
    password: "Bf3LZhmWeoOt"

captive_portal:
```

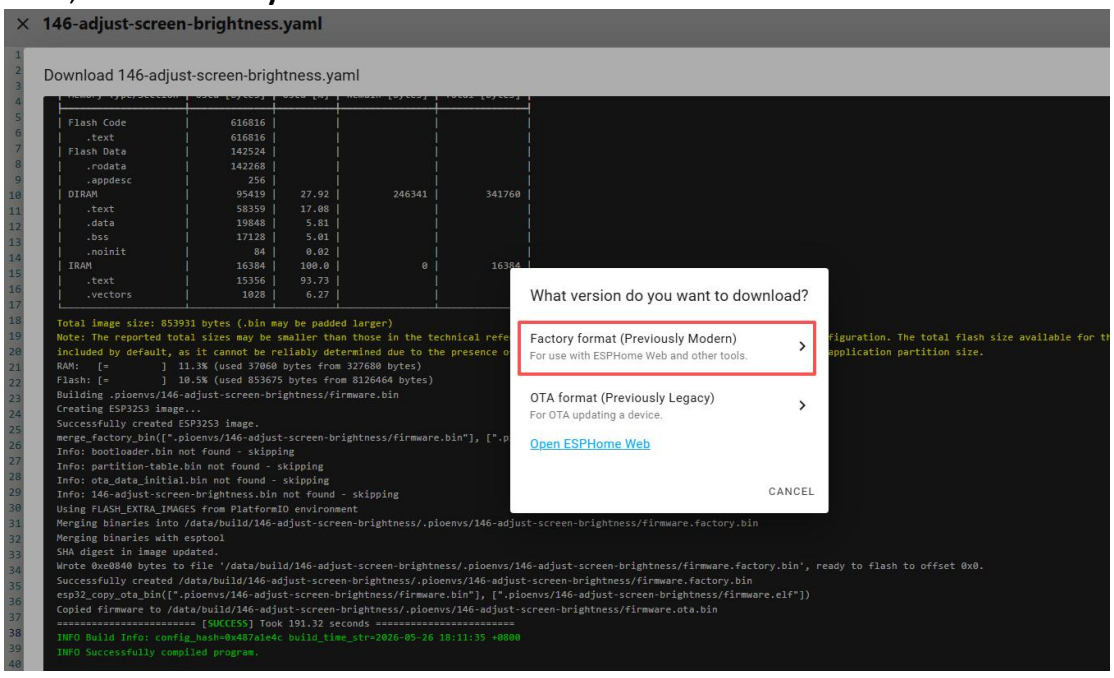
Once the code replacement is complete, click **"INSTALL"** in the top right corner.



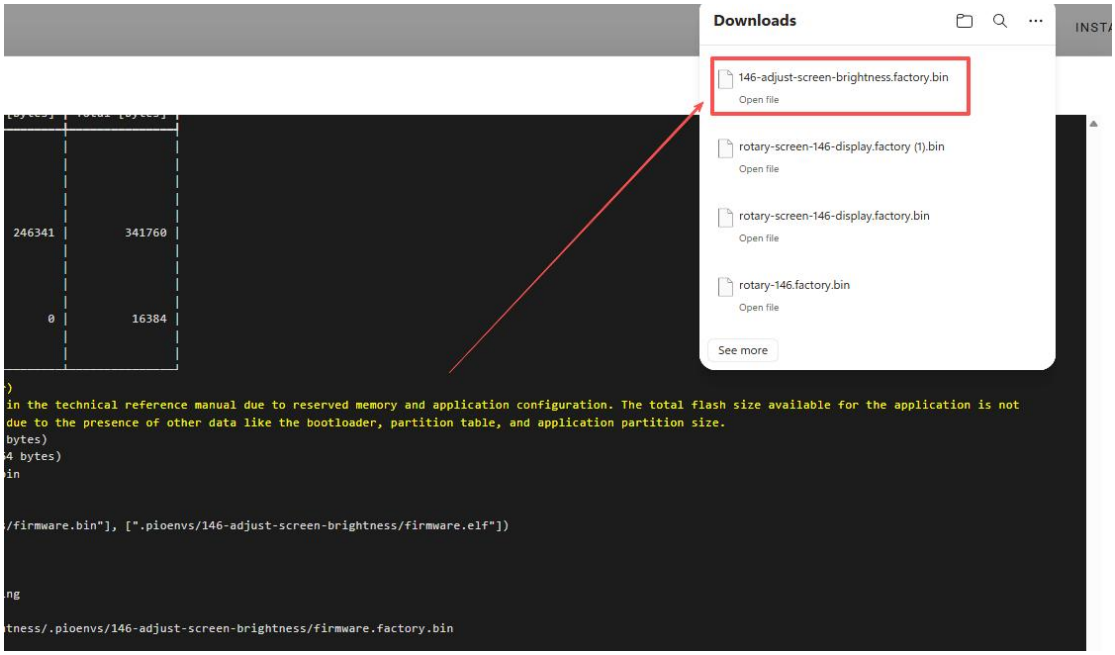
Select **"Manual download"**.



Wait for a few minutes until the installation is complete.
Then, select **"Factory format"**.

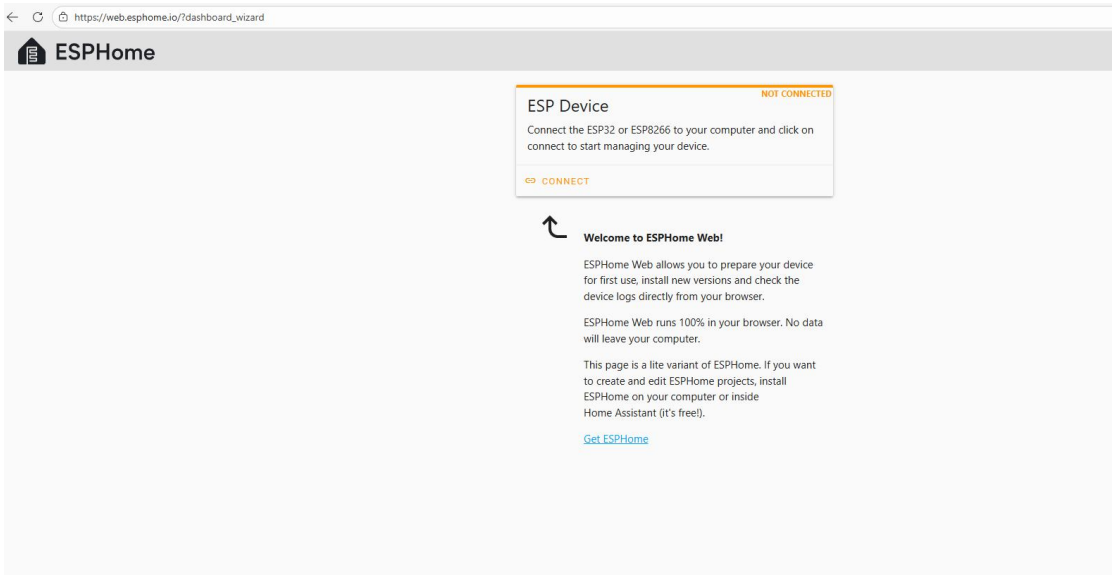


Once the download is complete, you will see the **.bin** file.



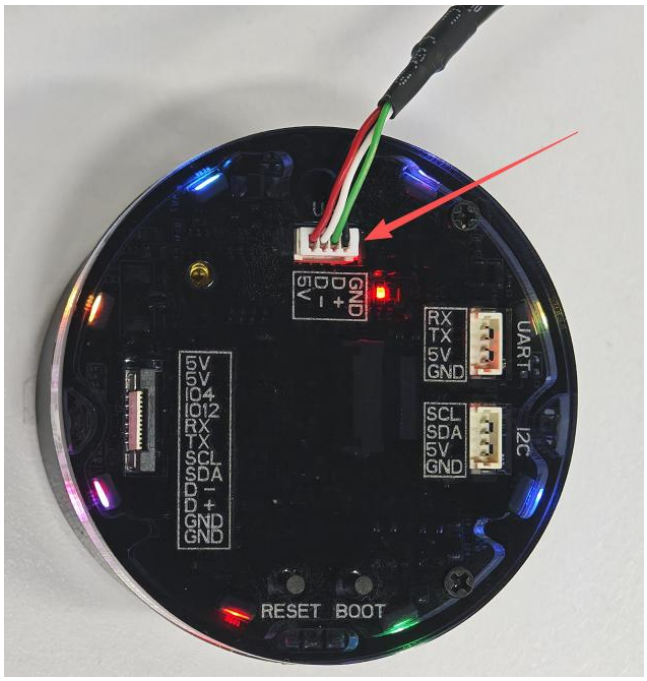
Remember the path of this **.bin** file.
Open the following website: https://web.esphome.io/?dashboard_wizard

After opening this website, you will arrive at this interface:

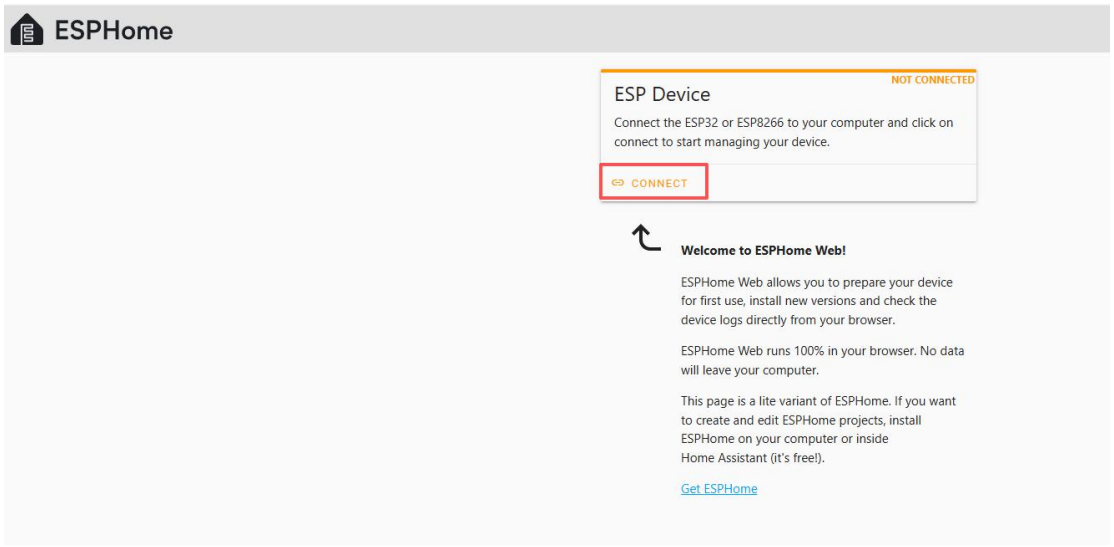


Next, we will flash this **.bin** file into the CrowPanel 1.46inch-HMI ESP32 Rotary Display .

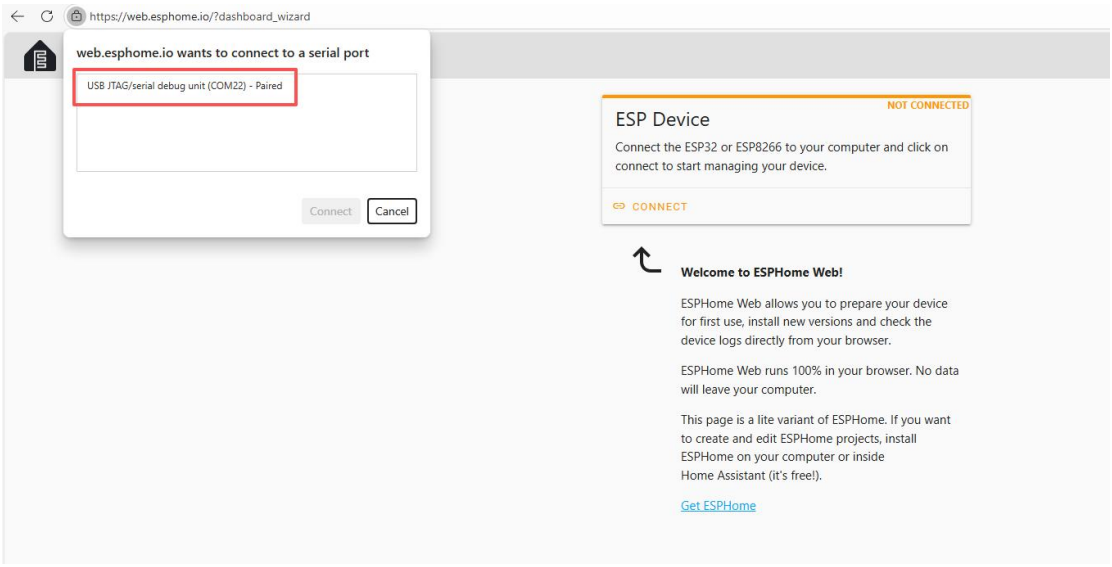
Connect the CrowPanel 1.46inch-HMI ESP32 Rotary Display to your computer.



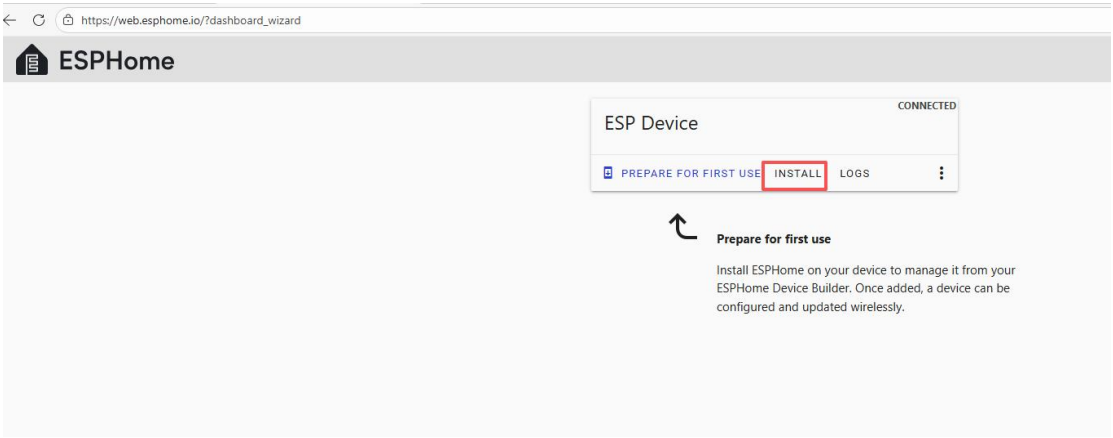
Click **"Connect"**



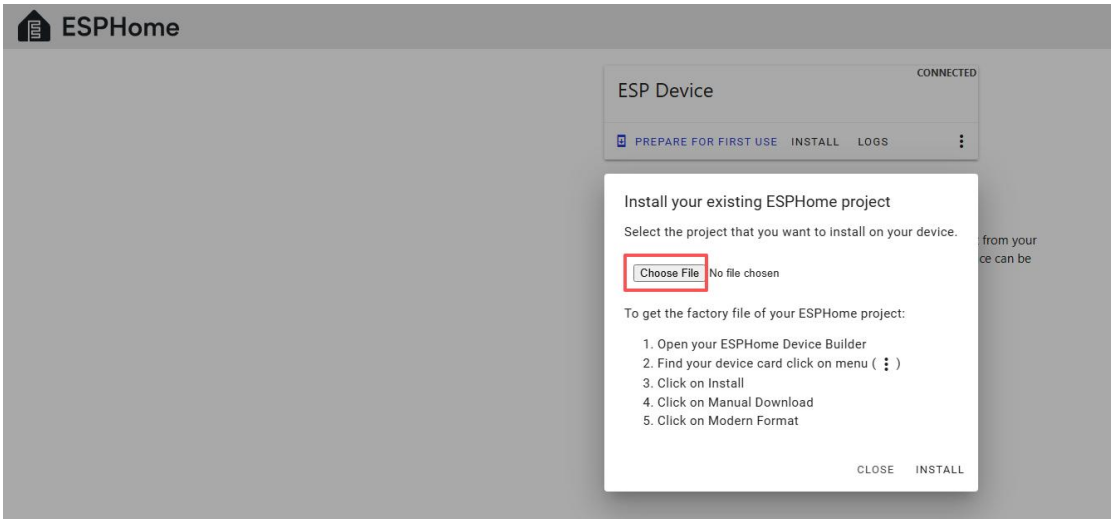
Select the COM port, and connect it.



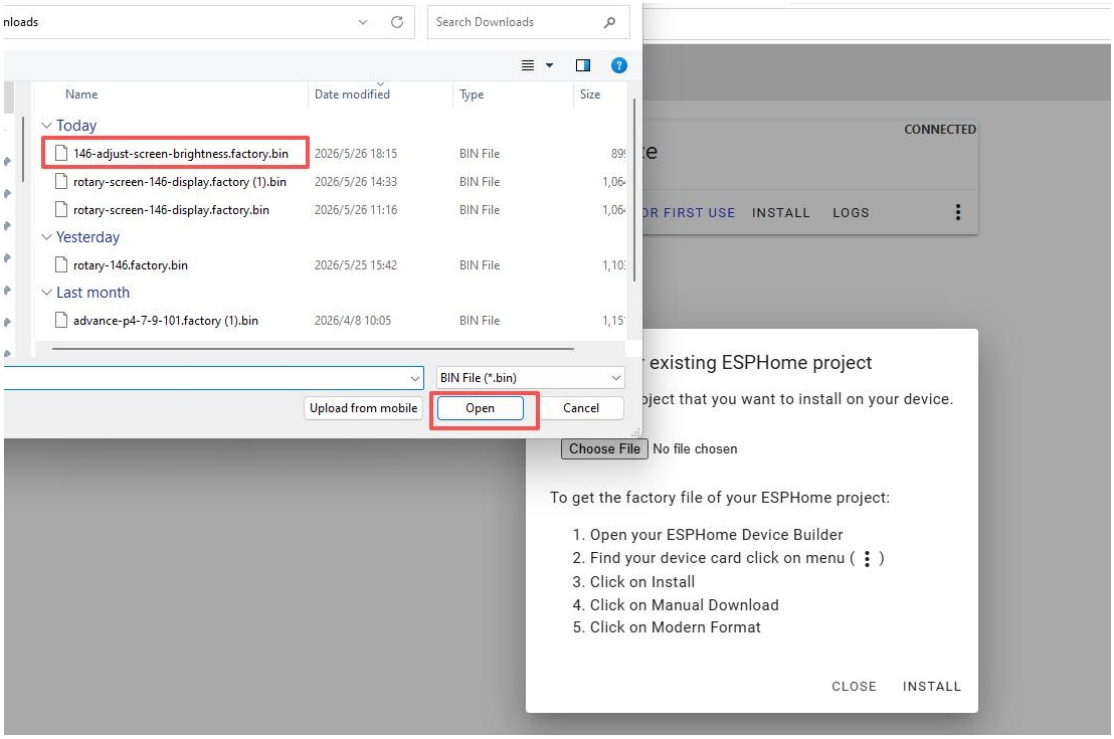
After connecting the CrowPanel 1.46inch-HMI ESP32 Rotary Display , click **"Install"**.



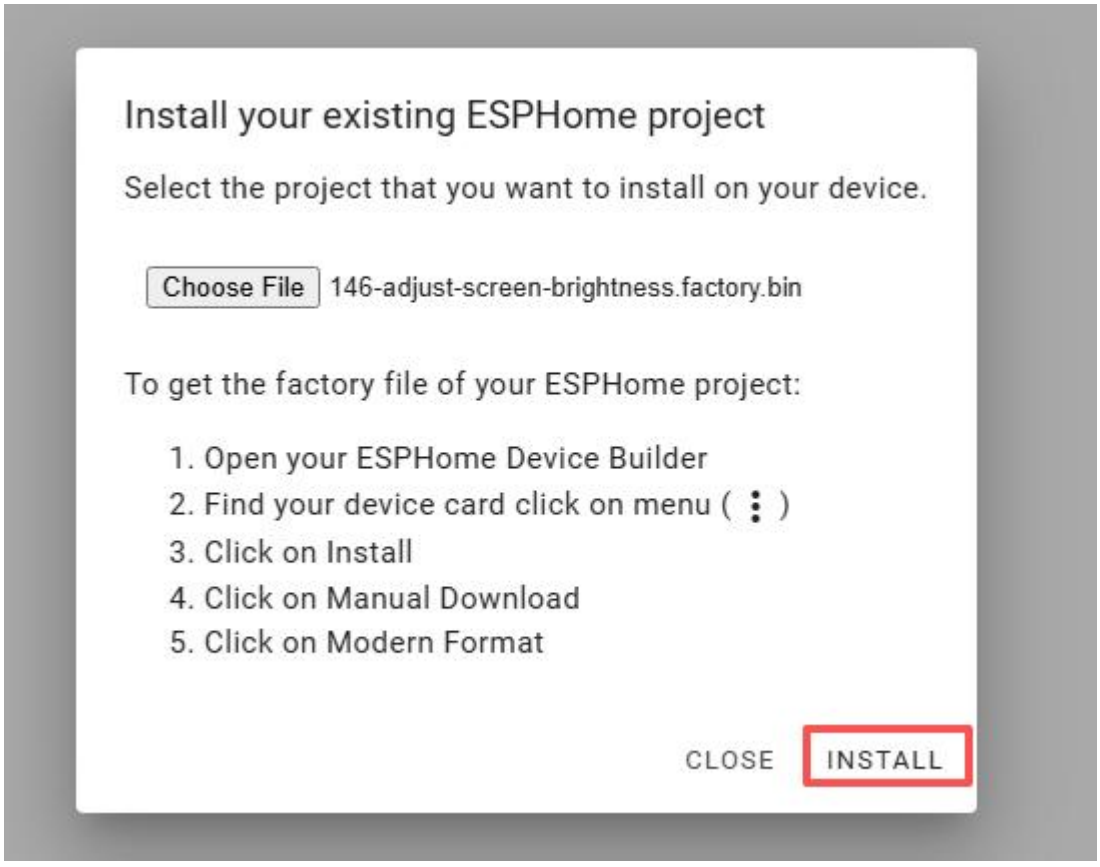
Add the **.bin** file you just downloaded, then click **"Install"**.



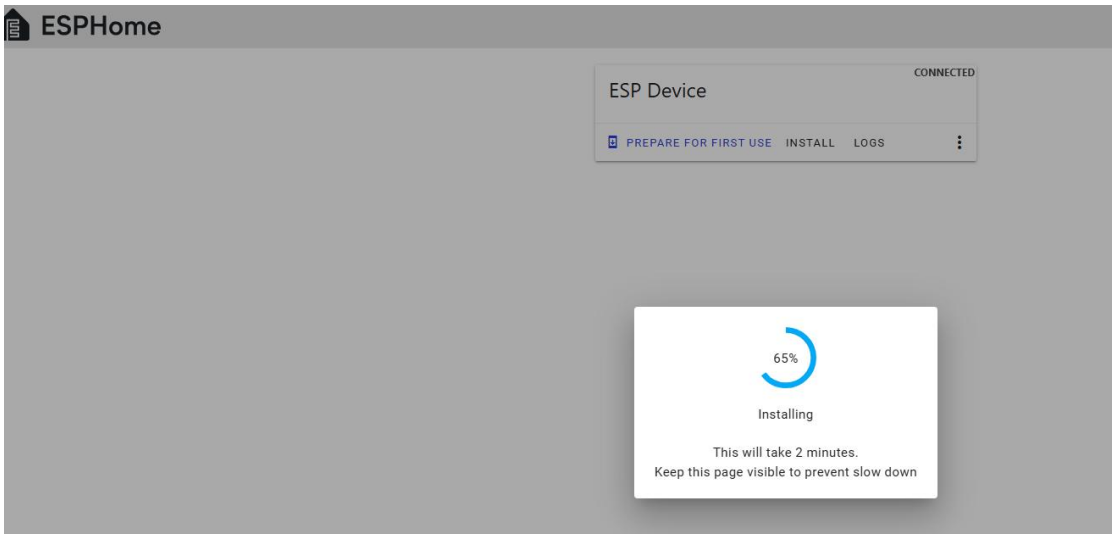
Then select the .bin file that you just downloaded.



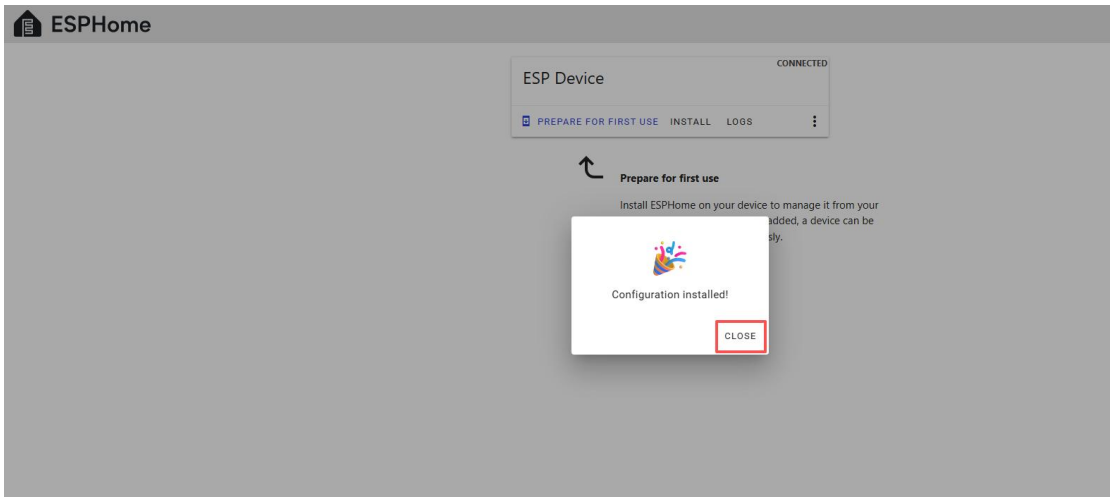
Click "INSTALL"



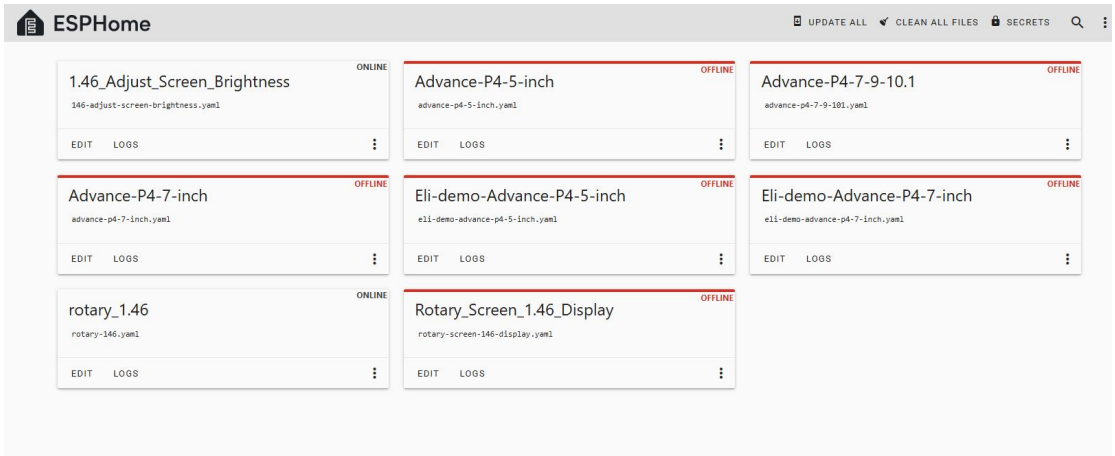
Wait for a few minutes.



After the installation is complete, click **"Close"**.

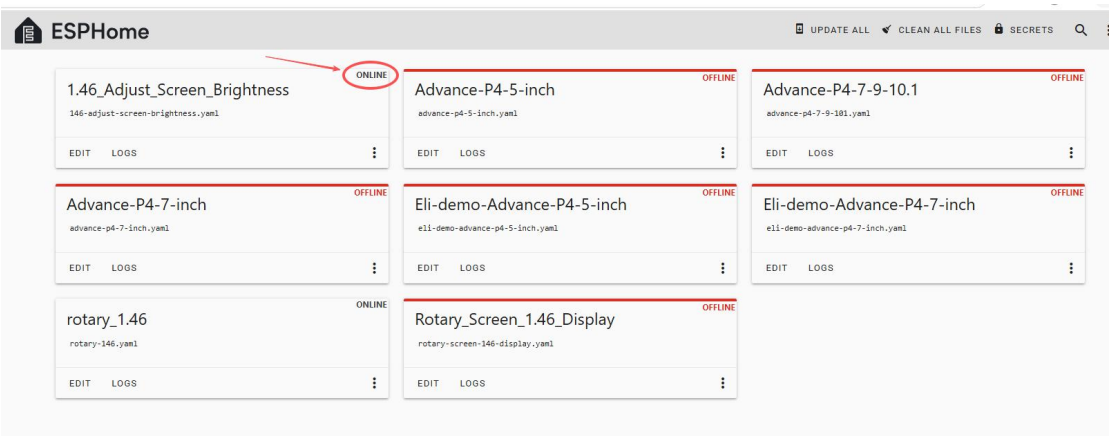


After successfully flashing the **.bin** file, return to the ESPHome page in Home Assistant.



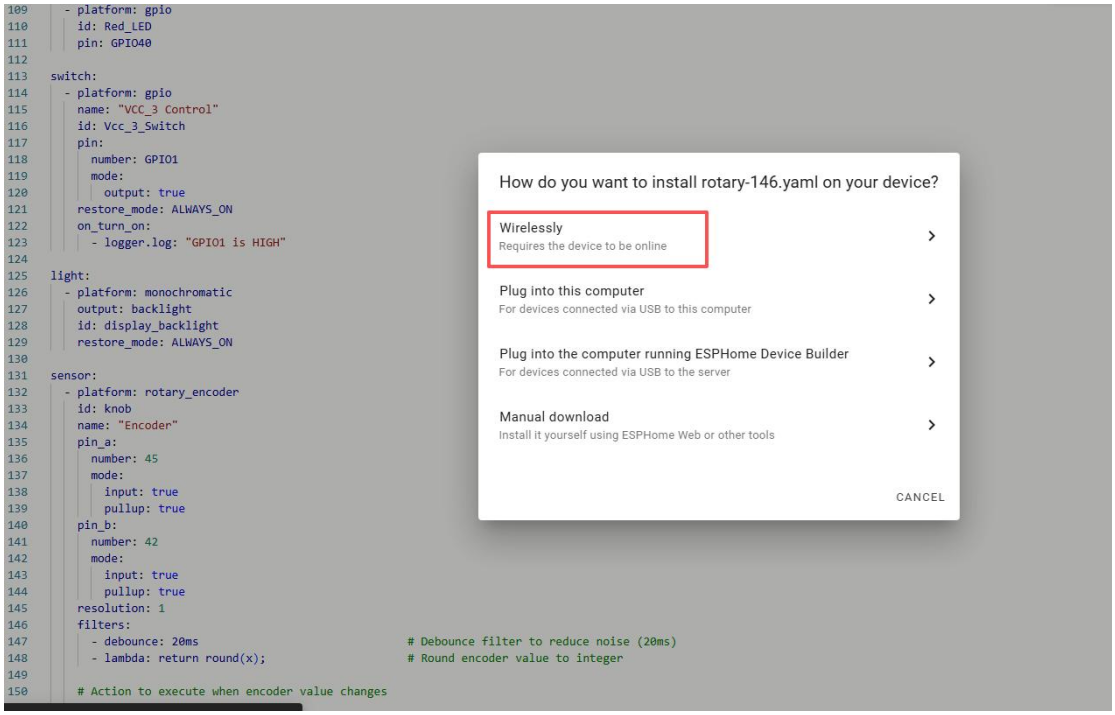
Press the RESET button on the CrowPanel 1.46inch-HMI ESP32 Rotary Display.

Restart the ESP32 display, and you should see the device you created earlier show as **ONLINE** in the top right corner.



If you have also added the functional code we provided, you will be able to see the current screen brightness value displayed on the screen. You can also try rotating the knob and observe the screen brightness changing in real time.

Once it shows ONLINE here, any future code modifications can be uploaded through the Wireless upload method, making subsequent uploads much more convenient.



Lesson04---LVGL Interface for Adjusting Screen Brightness

1. Course Introduction

In the previous lesson, we learned how to write code on the ESPHome platform to adjust screen brightness. In this lesson, based on the previous lesson, we will add a beautiful LVGL interface to display and control the screen brightness, providing stronger interactivity. This is also a tutorial for helping us deeply understand LVGL.

2. Learning Objectives

Learn how to use LVGL to create screen interfaces

Learn how to combine LVGL widgets with products

3. Project Demonstration

You can rotate the knob to adjust the screen brightness. The current brightness value will be displayed on the screen in real time, and the slider on the screen will also move and update in real time as you rotate the knob.

(Rotate clockwise to increase the screen brightness, rotate counterclockwise to decrease the screen brightness)

4. Example Code Download Link and Key Code Explanation

Click the GitHub link below to download the complete code:

https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/blob/master/example/V1.0/ESPHome/Lesson_Code/146-lvgl-interface.yaml

Next, we will explain the key parts of this code and how to write code on the ESPHome platform to connect the LVGL interface with the product's rotary interaction effect.

In the following code explanation, some related configuration content has already been explained in detail in the previous lesson, so it will not be repeated here. This lesson will mainly focus on the newly added core knowledge: how to link the LVGL interface with the rotary knob function, allowing the slider on the screen to synchronize with the knob rotation and enabling the interface interaction to respond to hardware operations in real time.

(Since the functionality of this lesson is based on the screen brightness adjustment function from ESPHome Lesson 03, if you encounter any code you do not understand, you can review the explanation in ESPHome Lesson 03.)

(1) Lambda Section (esphome block)

This Lambda code is the key synchronization logic for the LVGL visual interface during device startup initialization. Its core function is to accurately synchronize the initial global brightness value of 50 to the LVGL arc brightness widget and the center percentage text label, and then force LVGL to refresh the screen immediately, allowing the device to instantly and accurately display the 50% brightness state after startup, achieving synchronization between the hardware backlight initial brightness and the visual interface.

```
1   esphome:
9     on_boot:
11      then:
20        - lambda: |-
24
25        - light.turn_on:
26            id: back_light
27            brightness: 0.5
28
29        - lambda: |-
30            lv_arc_set_value(id(arc_brightness), id(brightness_value));
31            char buf[10];
32            snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));
33            lv_label_set_text(id(lbl_brightness), buf);
34            lv_refr_now(NULL);
35
36        - delay: 200ms
37        - lambda: |-
38            id(encoder_ready) = true;
```

In the code, the native LVGL API lv_arc_set_value is first used to pass the arc widget ID and the initial brightness value, setting the display progress of the arc progress bar.

```
29      - lambda: |-
30          lv_arc_set_value(id(arc_brightness), id(brightness_value));
31          char buf[10];
32          snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));
33          lv_label_set_text(id(lbl_brightness), buf);
34          lv_refr_now(NULL);
35
```

lv_arc_set_value is the core native API in the LVGL graphics library specifically used to set the current progress value of an arc widget. It is also the fundamental function for controlling arc widgets. For beginners, it is the key instruction that allows the arc to change according to business data (such as the brightness value in this case).

In our code, its function is to pass the initial value 50 of the global brightness variable to the predefined arc widget arc_brightness, allowing this 270° arc progress bar to directly display a 50% progress state, achieving precise mapping from data to the visual arc.

```
340     - arc:  # Brightness arc indicator
341         align: CENTER                    # Align to center
342         id: arc_brightness               # Widget ID
343         value: 50                        # Default value
344         min_value: 0                      # Minimum value
345         max_value: 100                    # Maximum value
346         start_angle: 135                  # Arc start angle
347         end_angle: 45                    # Arc end angle
348         width: 260                       # Widget width
349         height: 260                      # Widget height
350
351         arc_color: 0x1A1A1A              # Background arc color
352         arc_width: 18                    # Background arc thickness
353         arc_rounded: true                # Rounded ends
354
355         indicator:                        # Progress indicator
356           arc_color: 0x00D2FF            # Indicator color
357           arc_width: 18                  # Indicator thickness
358           arc_rounded: true              # Rounded ends
```

The calling format of this function is fixed as lv_arc_set_value(arc widget ID, value to set). The first parameter must be the unique ID of the arc widget defined in the LVGL widgets section (in the code it is id(arc_brightness), which in ESPHome must be accessed through id()), and the second parameter is the target value of the arc, which must match the min_value and max_value range configured for the arc widget (in your code, the arc range is 0-100, so passing a brightness value between 0 and 100 matches perfectly. Values outside the range will automatically be clamped to 0

or 100 by LVGL). This parameter is an integer type, so no manual type conversion is required. You can directly pass an integer or an integer global variable.

After calling this function, LVGL will automatically adjust the display length of the colored indicator segment in the arc (the sky-blue indicator in your code) according to the passed value. The smaller the value, the shorter the indicator segment; the larger the value, the longer the indicator segment. At the same time, the knob at the end of the arc will synchronously move to the corresponding position. This entire process is automatically rendered internally by LVGL. You only need to provide the correct widget ID and a value within the matching range to achieve synchronization between the arc progress and business data, without needing to care about the underlying drawing details. This is very suitable for beginners using LVGL arc widgets.

Then, a 10-character buffer buf is defined. The safe string formatting function snprintf is used to convert the integer brightness value into a string with a percentage sign (such as 50 → “50%”) and store it in the buffer. Next, the LVGL lv_label_set_text API is called to set the formatted string to the center text label widget, completing the text display configuration. Finally, lv_refr_now(NULL) is used to force LVGL to immediately refresh all associated screens, avoiding display delays caused by the default refresh cycle.

```
- lambda: |-
    lv_arc_set_value(id(arc_brightness), id(brightness_value));
    char buf[10];
    snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));
    lv_label_set_text(id(lbl_brightness), buf);
    lv_refr_now(NULL);
```

Lambda Usage Notes:

1. When writing this code in an ESPHome Lambda, you must follow the core syntax rules: Lambda is a C++ code block embedded in YAML, and all components and global variables defined in YAML must be accessed using id(xxx). IDs cannot be written directly.

2. Every executable statement must end with an English semicolon ;

3. At the end, lambda: must be immediately followed by |-, and the code indentation must remain consistent (YAML identifies the code block scope through indentation). There is no need to wrap multiline code with {}, and there is no need to manually include C standard library header files. Functions such as snprintf can be used directly.

4. When defining character buffers, sufficient extra space should be reserved to prevent overflow. At the same time, avoid performing time-consuming operations inside Lambda to prevent blocking the ESP32 main loop. Various native LVGL APIs can also be called directly without additional initialization.

(2) Adjusting Brightness with the Knob and Synchronizing the LVGL Interface

First, the hardware-level configuration of the encoder is completed, which was already explained in detail in the previous lesson.

Next, we will explain the important parts related to this lesson and see how the brightness adjustment made by the rotary knob is synchronized to LVGL:

```
sensor:
  - platform: rotary_encoder                    # Rotary encoder sensor
    filters:  # Anti-noise signal filters

    on_value:  # Trigger on encoder rotation
      then:
        - lambda: |-
            if (!id(encoder_ready)) return;         // Skip if not initialized
            static bool first_run = true;           // First run flag
            static int last_encoder = 0;            // Last encoder state
            int current = (int) id(knob).state;     // Current encoder value

            if (first_run) {                        // Initialize baseline
              last_encoder = current;
              first_run = false;
              return;
            }

            int delta = current - last_encoder;     // Calculate rotation change
            last_encoder = current;                 // Update last value
            if (delta == 0) return;                 // Ignore no movement

            if (delta < 0) {                        // CCW: decrease brightness
              id(brightness_value) -= 5;
            } else {                                // CW: increase brightness
              id(brightness_value) += 5;
            }

            if (id(brightness_value) < 0) id(brightness_value) = 0;       // Min limit
            if (id(brightness_value) > 100) id(brightness_value) = 100;   // Max limit

            float level = id(brightness_value) / 100.0f;                  // Convert to 0.0-1.0
            id(backlight).set_level(level);                               // Set PWM output

            lv_arc_set_value(id(arc_brightness), id(brightness_value));   // Update arc
            char buf[10];                                                 // String buffer
            snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));     // Format text
            lv_label_set_text(id(lbl_brightness), buf);                   // Update label
            lv_refr_now(NULL);                                            // Refresh display
```

This Lambda code inside on_value is the core logic for adjusting the backlight brightness using the rotary encoder and synchronizing the LVGL interface. It is triggered every time the encoder rotates one step. The overall process follows the closed-loop workflow of “first validate to prevent accidental triggers → calculate rotation increment → adjust brightness and apply range limits → update hardware backlight → synchronize the visual interface”:

First, the global variable encoder_ready is used to filter accidental operations before the device startup is completed. Then two static variables, first_run and last_encoder, are defined (used to initialize the reference value on the first execution and preserve the previous encoder value afterward). The current floating-point accumulated encoder value is obtained and converted to an integer. If this is the first trigger, the reference value is initialized and the function exits, preventing abnormal increment calculations during the first rotation.

Next, the rotation direction is determined by the difference delta between the current value and the previous value (positive for clockwise, negative for counterclockwise). The reference value is synchronously updated, and invalid triggers with delta equal to 0 are filtered out. Then the global brightness variable

brightness_value is increased or decreased in steps of 5, while strictly limiting the brightness value within the valid range of 0-100 to prevent it from exceeding the reasonable range for PWM duty cycle calculations. After that, the integer brightness value from 0-100 is converted into a floating-point duty cycle value from 0.0-1.0 and assigned to the backlight PWM output component through the set_level method, truly achieving hardware backlight brightness adjustment.

```
133  sensor:
134    - platform: rotary_encoder                    # Rotary encoder sensor
153      on_value:  # Trigger on encoder rotation
154        then:
155          - lambda: |-
                }

                if (id(brightness_value) < 0) id(brightness_value) = 0;       // Min limit
                if (id(brightness_value) > 100) id(brightness_value) = 100;   // Max limit

                float level = id(brightness_value) / 100.0f;                  // Convert to 0.0-1.0
                id(backlight).set_level(level);                               // Set PWM output

                lv_arc_set_value(id(arc_brightness), id(brightness_value));   // Update arc
                char buf[10];                                                 // String buffer
                snprintf(buf, sizeof(buf), "%d%%", id(brightness_value));     // Format text
                lv_label_set_text(id(lbl_brightness), buf);                   // Update label
                lv_refr_now(NULL);                                            // Refresh display
```

First, the global integer brightness value brightness_value (0-100) is divided by the floating-point value 100.0f to convert it into the 0.0-1.0 floating-point duty cycle required by the ESP32 LEDC PWM output, and the result is assigned to level. Then the set_level method of the backlight PWM output component backlight is called, passing in the duty cycle value and applying it immediately, thereby truly completing the actual hardware backlight brightness adjustment. The larger the duty cycle, the brighter the backlight.

Immediately afterward, the core native API lv_arc_set_value for the LVGL arc widget is called. This function is specifically used to set the current display value of the arc progress bar. The first parameter passes the arc widget ID arc_brightness defined in YAML, and the second parameter directly passes the current integer brightness value. LVGL will automatically adjust the length of the colored indicator segment in the arc and the position of the knob at the end according to the passed brightness value. Whatever the brightness value is, the arc will accurately display the corresponding progress proportion, achieving direct mapping from value to arc visualization.

To make the brightness value more intuitive, a 10-character buffer buf is then defined. The safe C-language string formatting function snprintf is used to format the integer brightness value into a string with a percentage sign (such as converting 50 into “50%”), and the formatted string is stored in the buffer. Then the LVGL label widget API lv_label_set_text is called to pass the string in the buffer to the center brightness text label lbl_brightness, updating the label display content. Finally, lv_refr_now(NULL) is called to force LVGL to immediately refresh all associated screens, skipping the default screen update cycle so that the changes in the arc

progress and percentage text are displayed on the screen in real time without delay, ensuring instant feedback for operations. As the key connection between brightness values and arc display, lv_arc_set_value does not require you to care about the underlying drawing details. You only need to ensure that the brightness value passed in is within the 0-100 range preset by the arc widget to achieve precise and automatic synchronization between values and arc progress. It is the fundamental and core instruction for beginners using LVGL arc widgets to implement data visualization.

(3) Importing Materials

As you can see in our demonstration image, this beautiful picture is uploaded using the method we will explain next, and then referenced in the code.

146-adjust-screen-brightness.yaml

```
font:  # Font definitions for LVGL UI
  - file: "gfonts://Montserrat"                # Montserrat font source
  - file: "gfonts://Montserrat"                # Montserrat font source
    id: montserrat_14                          # Font ID (14px)
    size: 14                                   # Font size in pixels
  - file: "gfonts://Montserrat"                # Montserrat font source
    id: montserrat_12                          # Font ID (12px)
    size: 12                                   # Font size in pixels

image:  # Image assets for LVGL UI
  - file: "bj_light_1_360x360.png"             # Background image file
    id: img_bg_light                           # Image resource ID
    resize: 360x360                            # Resize to screen size
    type: RGB565                               # Display color format
```

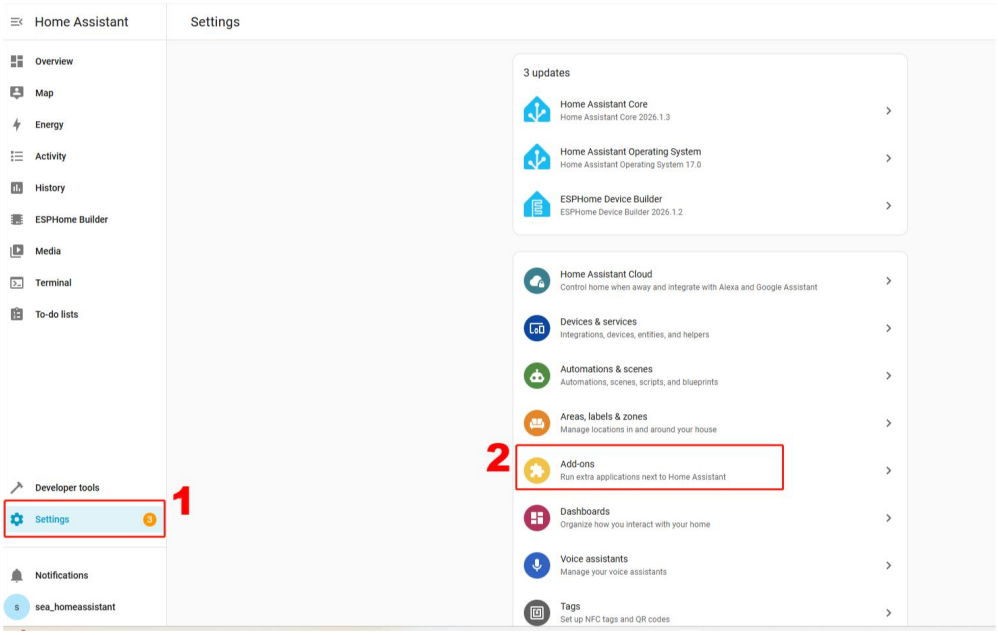
Before we begin, you can click the link below to download the materials we provide.

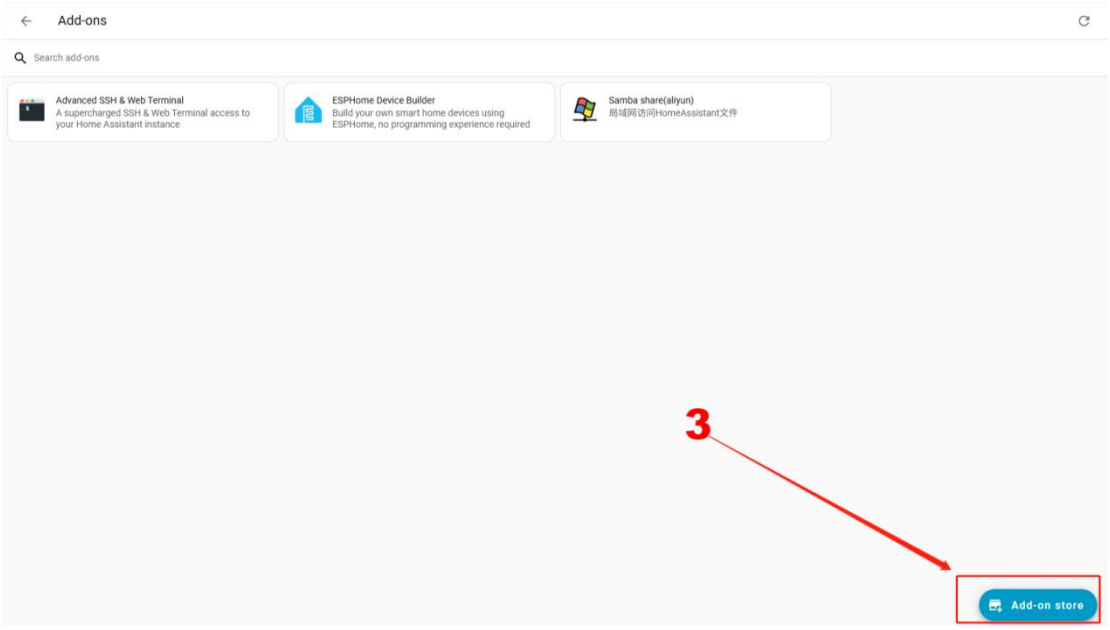
https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/tree/master/example/V1.0/ESPHome/Material

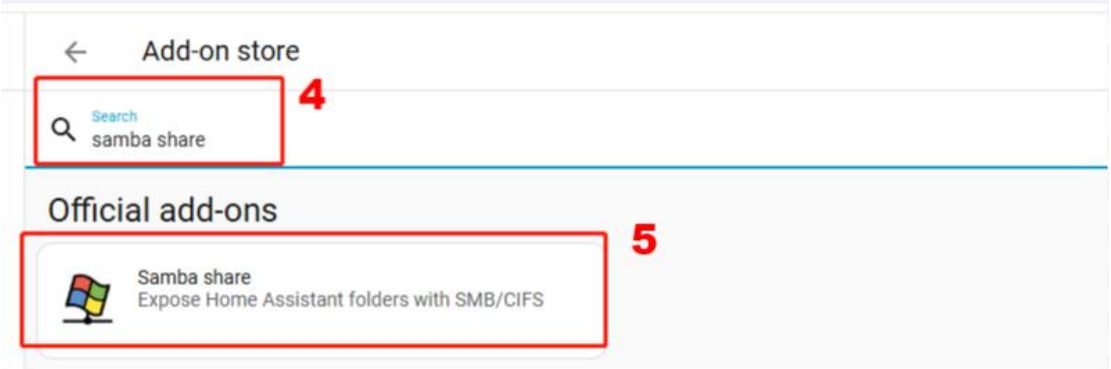
Next, we will teach you how to upload the materials you want to use so they can be used on the ESPHome platform.

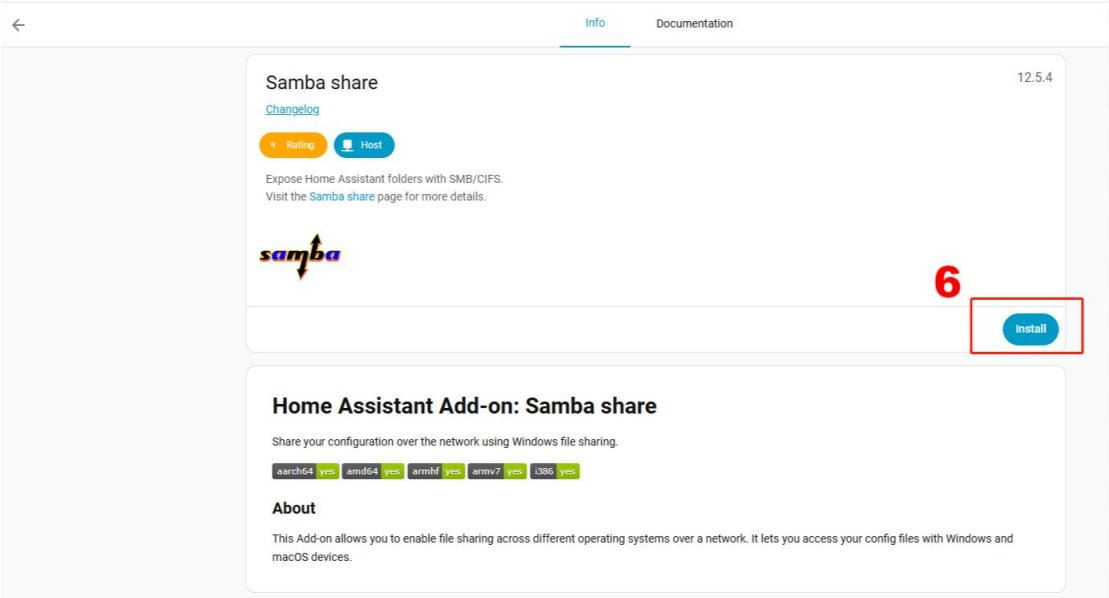
First, download the Samba Share tool. We need this tool to conveniently upload the image materials we want to use.

Follow the steps below one by one to complete the download.

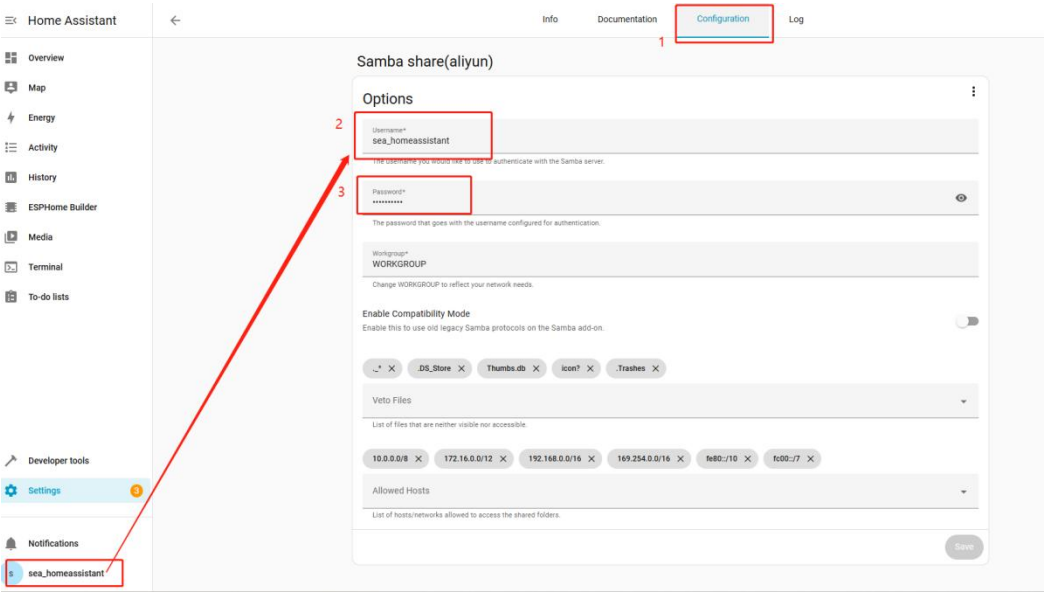






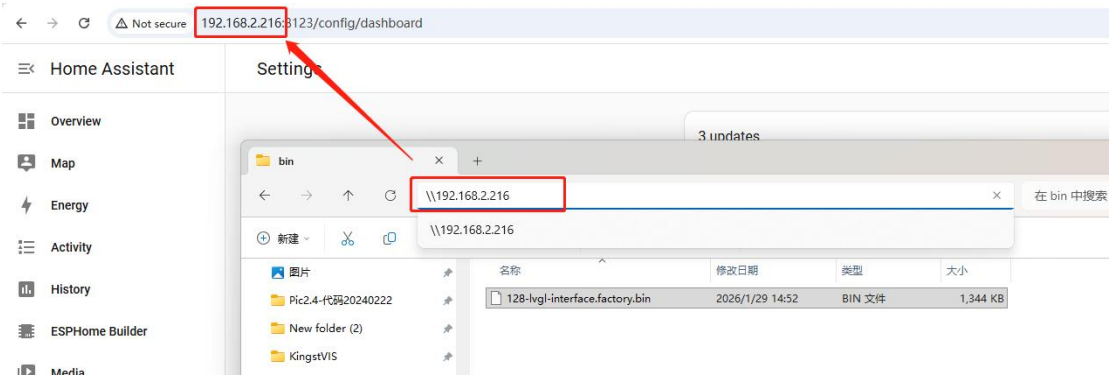


After the download is complete, configure your account on the page of this tool. Enter the account name and password used for your Home Assistant.

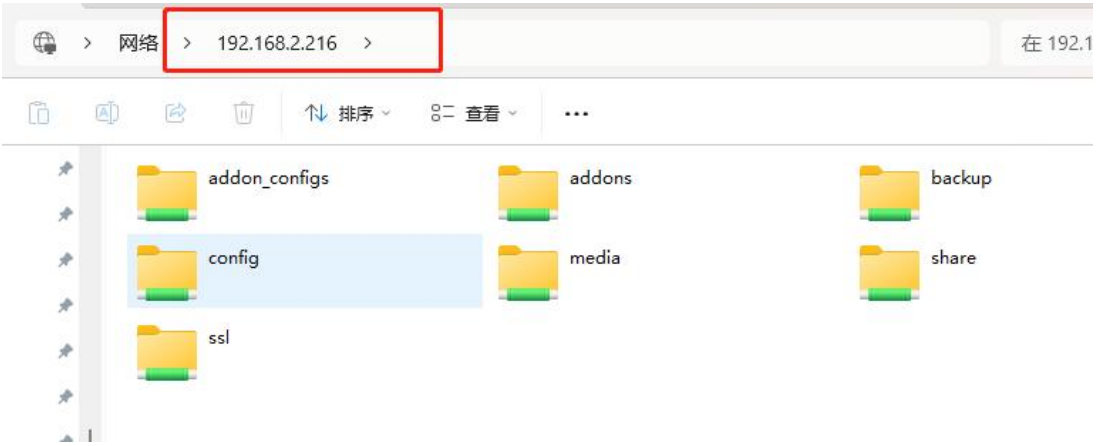


After the configuration is completed, remember to click Save in the lower-right corner.

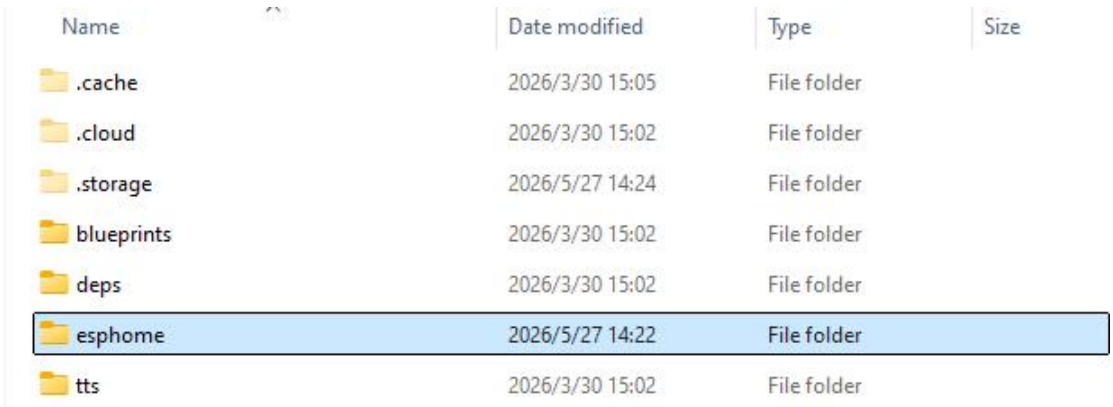
Then, on your computer, open a file explorer window and enter \\ + your Home Assistant IP address.



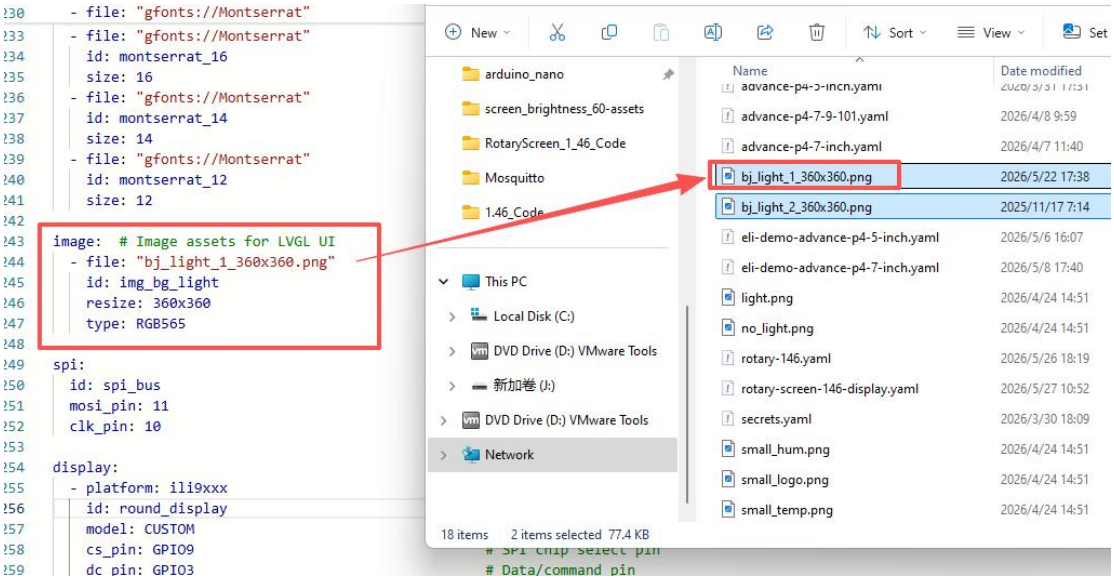
After entering, you will arrive at the current ESPHome file management interface.



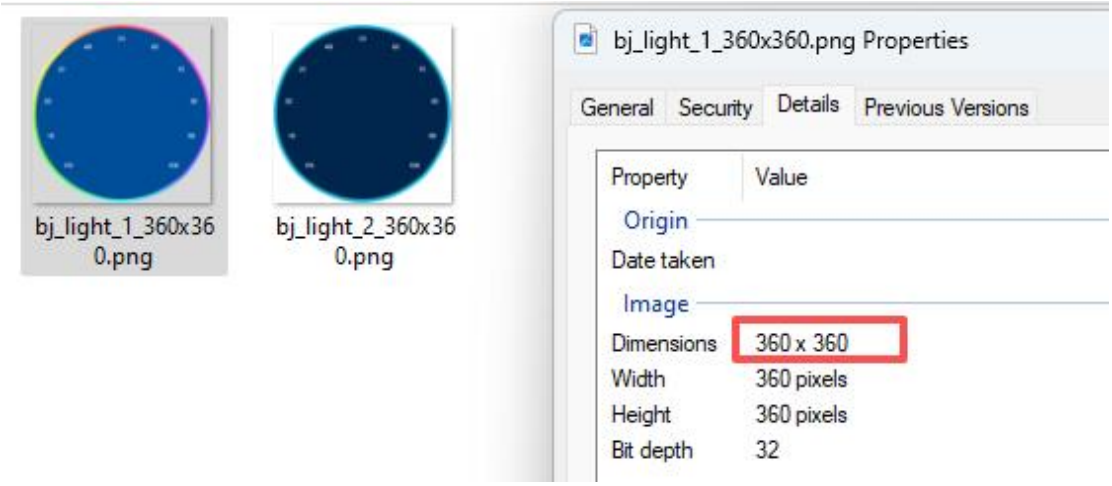
First enter the config folder, and then enter the esphome folder.



You can place the materials you need to use here. The material names used in your code must also remain consistent with the names of the materials you place here.



Note: Here we are using the CrowPanel 1.46inch-HMI ESP32 Rotary Display, and the resolution of the material images we use is 360x360.



Therefore, please keep the settings in the code consistent as well.

In this way, when your code is compiled, it will be able to find the materials you placed and display them on the screen.

(4) LVGL Configuration

This code is the complete configuration of the LVGL graphics library in ESPHome. The core purpose is to build a visual brightness adjustment interface for the 360x360 1.46-inch circular GC9A01A screen. It first completes the basic association configuration between LVGL and the screen, and then defines four overlaid interface widgets according to the drawing order “from bottom layer to top layer”, forming a brightness display interface with clear layering and adaptation to the circular screen.

First, the configured circular screen round_display is associated, and the LVGL buffer is set to 50% of the screen memory to ensure smooth interface rendering without lag.

```
206  lvgl:
207    displays:
208      - round_display
209    buffer_size: 50%
210
```

Then, the widgets are defined in sequence:

The bottom layer is the centered 360x360 background image img_bg, with anti-aliasing enabled for smoother display, serving as the background of the entire interface.

```
widgets:  # UI widget definitions (bottom to top)
  - image:  # Background image widget
      align: CENTER
      id: img_bg
      src: img_bg_light
      width: 360
      height: 360
      antialias: true
```

The second layer is the centered title label lbl_title, offset upward by 40 pixels, displaying “Brightness” in white 24-point Montserrat font to clearly indicate the interface function.

```
- label:  # Brightness title label
    align: CENTER
    id: lbl_title
    y: -70
    text_font: montserrat_24
    text: "Brightness"
    text_color: 0xFFFFFF
    text_opa: "100%"
```

The core third layer is the centered brightness arc widget arc_brightness, which is also the visual focus of the interface. Its initial value is 50, with a value range of 0-100, fully matching the brightness variable. Through a 135° start angle and a 45° end angle, it forms a 270° arc suitable for a circular screen. The dark gray arc background line has a width of 12 with rounded ends, while the sky-blue brightness indicator segment uses the same specification. A 20x20 white circular knob is placed at the end, intuitively indicating the current brightness progress.

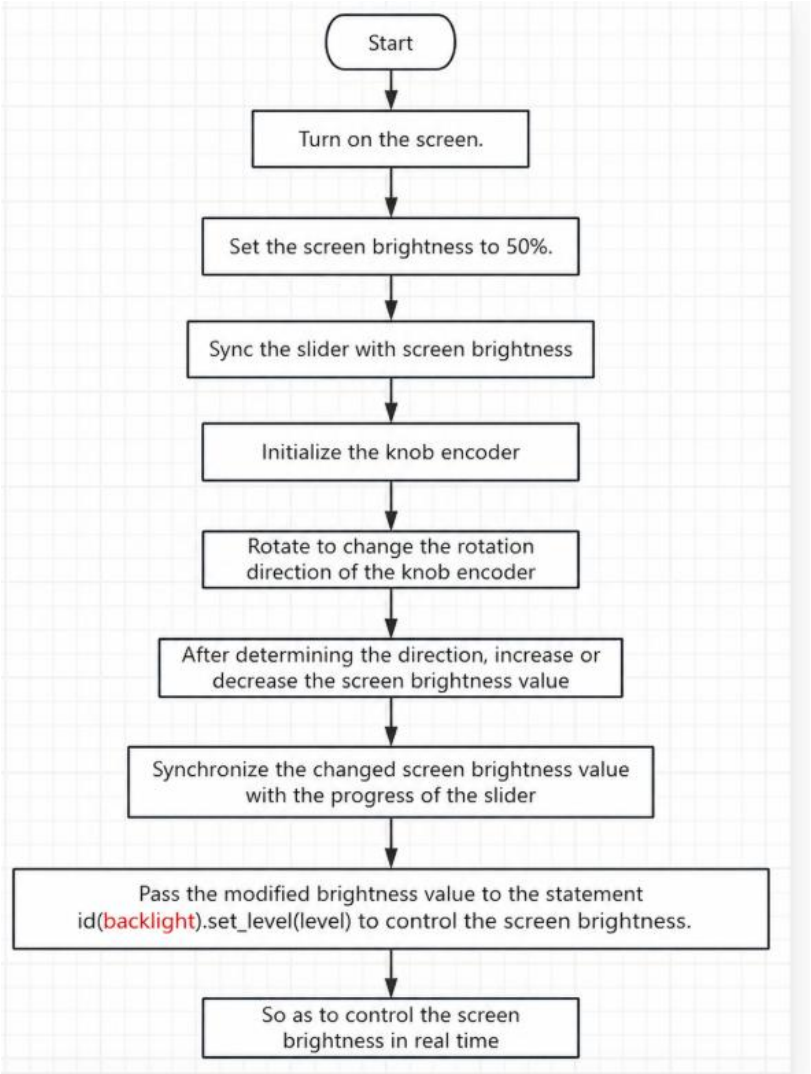
```
340     - arc:  # Brightness arc indicator
341         align: CENTER                          # Align to center
342         id: arc_brightness                     # Widget ID
343         value: 50                              # Default value
344         min_value: 0                            # Minimum value
345         max_value: 100                          # Maximum value
346         start_angle: 135                        # Arc start angle
347         end_angle: 45                          # Arc end angle
348         width: 260                             # Widget width
349         height: 260                            # Widget height
350
351         arc_color: 0x1A1A1A                    # Background arc color
352         arc_width: 18                          # Background arc thickness
353         arc_rounded: true                      # Rounded ends
354
355         indicator:                              # Progress indicator
356           arc_color: 0x00D2FF                  # Indicator color
357           arc_width: 18                        # Indicator thickness
358           arc_rounded: true                    # Rounded ends
359
360         knob:                                   # Arc knob
361           bg_color: 0xFFFFFF                   # Knob color
362           bg_opa: "100%"                       # Full opacity
363           width: 32                            # Knob width
364           height: 32                           # Knob height
365           radius: 10                           # Circular knob
```

The top layer is the centered brightness percentage label lbl_brightness, displaying the initial value “50%” in large 56-point white Montserrat font to prominently show the current brightness value. All widgets are laid out based on center alignment and stacked according to the drawing order to achieve the layered effect. The

parameters of the arc widget are also fully compatible with the subsequent encoder adjustment logic and lv_arc_set_value configuration, laying the foundation for real-time visual synchronization of brightness values later.

```
- label:  # Brightness percentage label
    align: CENTER
    id: lbl_brightness
    text_font: montserrat_56
    text: "50%"
    text_color: 0xFFFFFF
    text_opa: "100%"
```

5. Code Logic Flowchart



6. Flashing Steps

Next, we will teach everyone how to use ESPHome to write code, including the complete operation process. Please follow us step by step.

Here we emphasize again:

The following devices need to be on the same LAN:

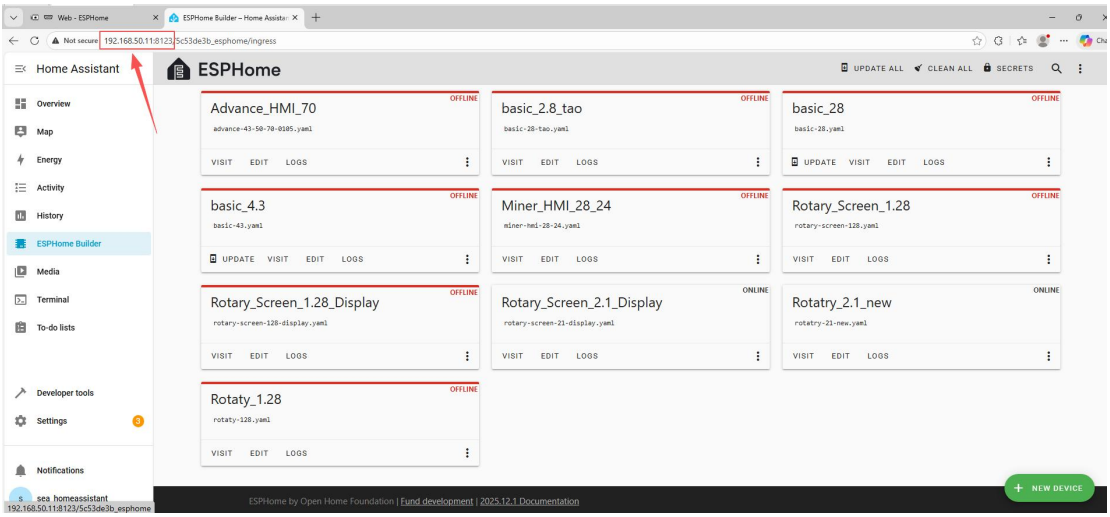
① Your computer

② Crowpanel HMI ESP32 Rotary Display

③ Raspberry Pi with the Home Assistant system

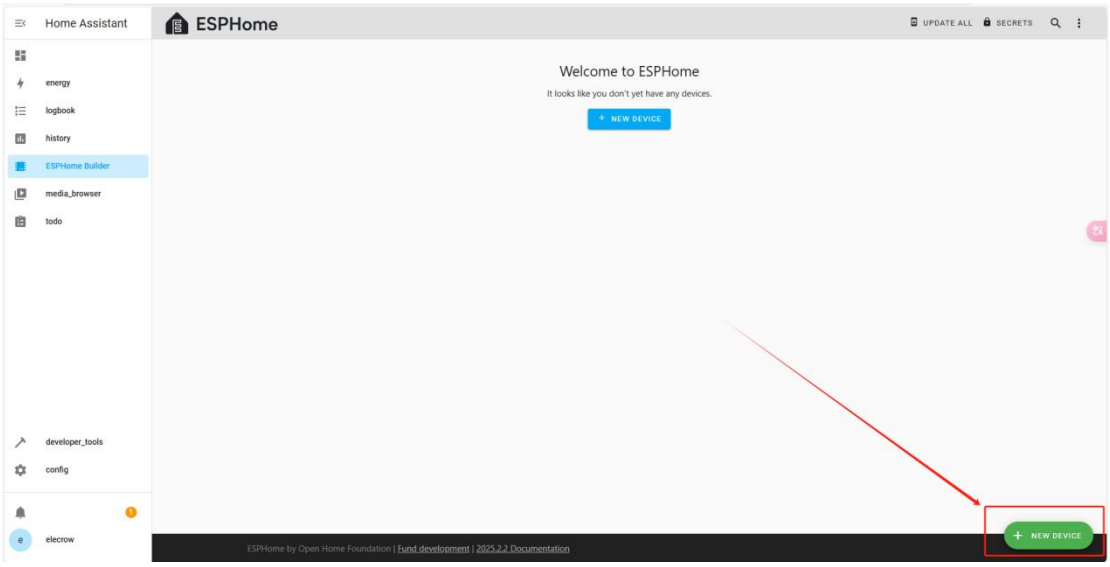
Here, our Raspberry Pi with the Home Assistant system acts as the server in this project, so every time we access the IP address of the Raspberry Pi with the Home Assistant system, we are entering Home Assistant.

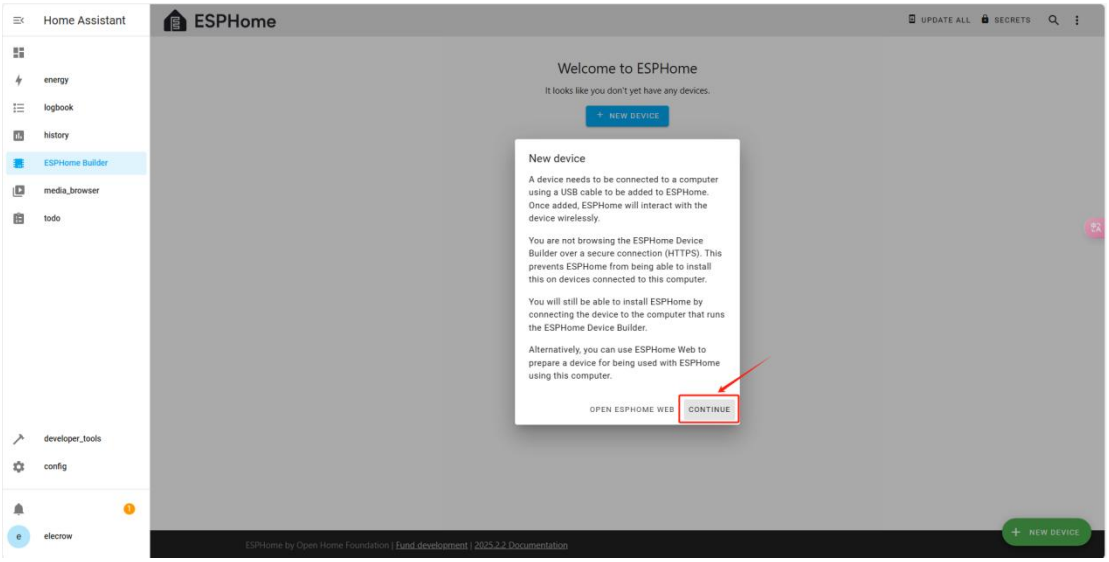
In other words, only after entering this page can you perform the following operations.



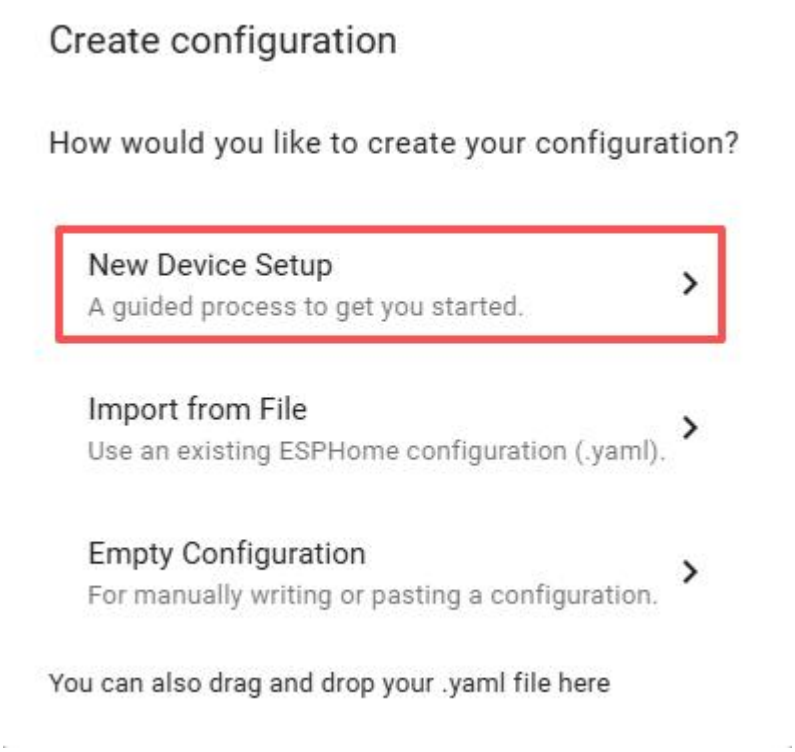
The following operations assume that you have already completed the installation steps in Lesson 01 and arrived at the ESPHome main page.

Once the installation is complete, we can start adding devices. Click on **+ New Device** -> **Continue**.



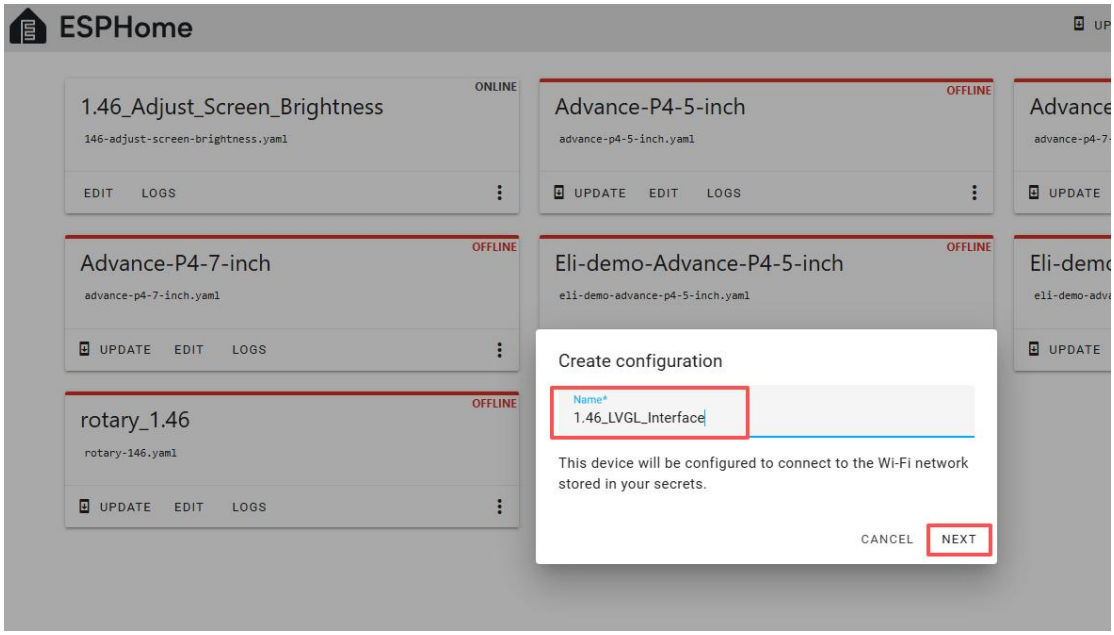


Click “New Device Setup”



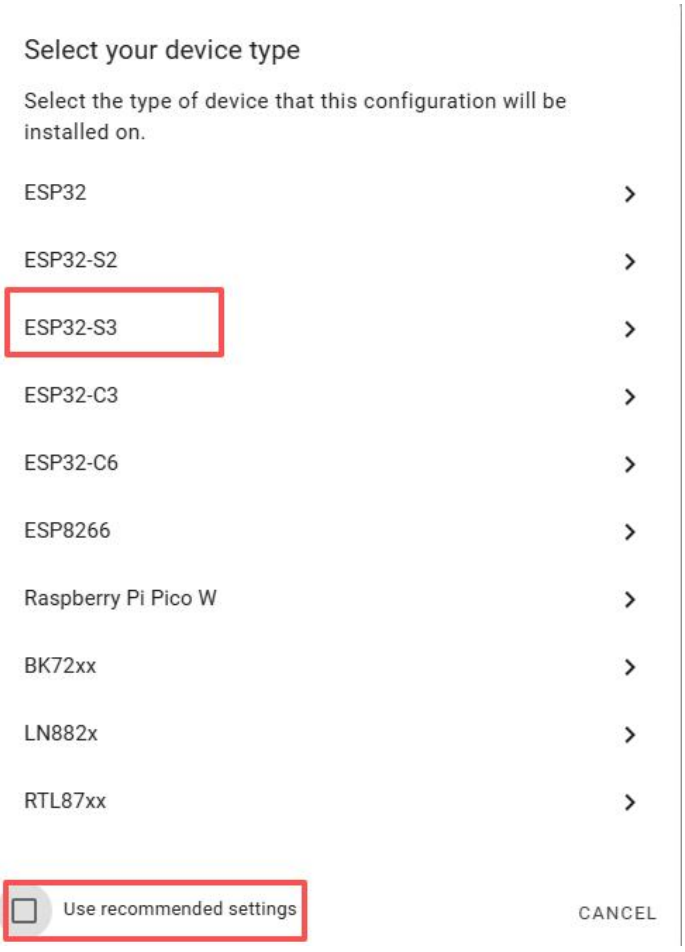
Enter a name and click **Next**.

(You can use any custom name. Do not include any strange symbols such as @, #, etc.)

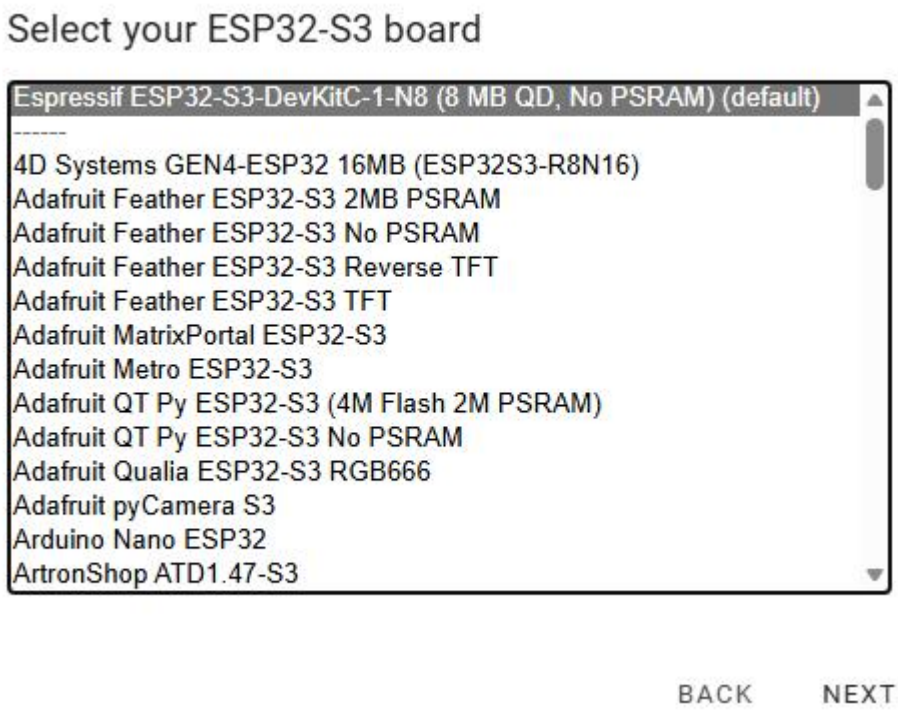


Here, do not check "Use recommended settings."

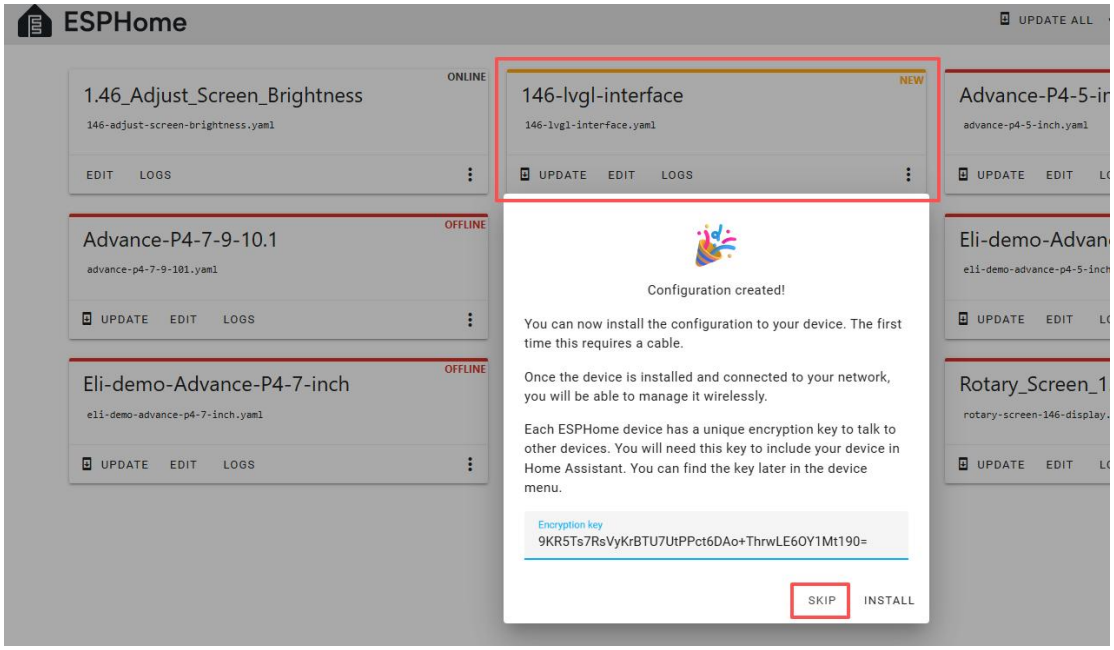
Select the main chip of the CrowPanel 1.46inch-HMI ESP32 Rotary Display , **ESP32-S3**.



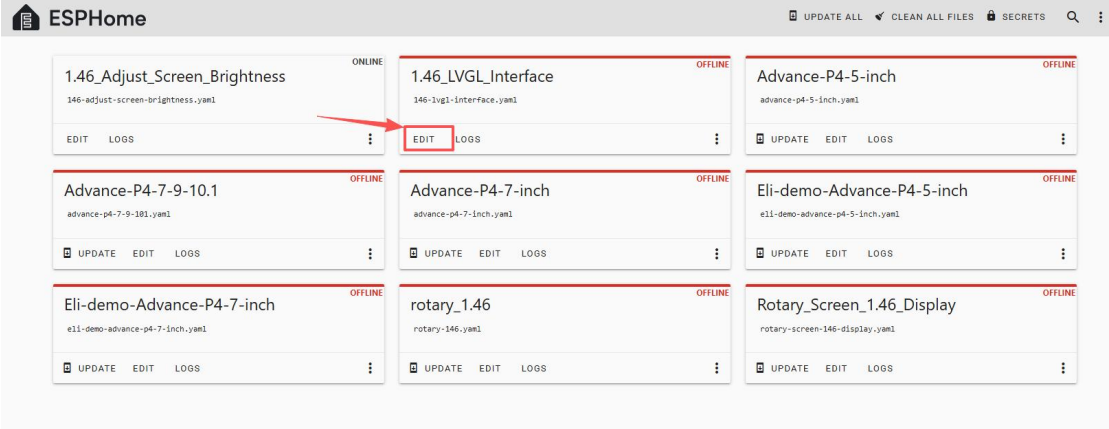
Next, choose any option (since we will replace it in the code later).



Here, click **"SKIP"**.



Then, return to the main interface, find the **1.46_LVGL_Interface** you just created, click **"EDIT"**, and enter the code editor.



This code is automatically generated based on the previous steps. Next, we will make replacements in it, which will help optimize the code for more efficient operation.

Automatically generated code:

146-lvgl-interface.yaml

```yaml
esphome:
  name: 146-lvgl-interface
  friendly_name: 1.46_LVGL_Interface

esp32:
  board: esp32-s3-devkitc-1
  framework:
    type: esp-idf

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "9KR5Ts7RsVyKrBTU7UtPPct6DAo+ThrwLE6OY1Mt190="

ota:
  - platform: esphome
    password: "8efe0c7e7ec5b46ff2835a07a62675a6"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "146-Lvgl-Interface"
    password: "UMWQH1lXcNAT"

captive_portal:
```

This is the complete code. You can click on the code link below to obtain it.

https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/blob/master/example/V1.0/ESPHome/Lesson_Code/146-lvgl-interface.yaml

Next, you can replace the relevant content in **esphome** and **ESP32** as needed.And add the item **"PSRAM".**

× **146-lvgl-interface.yaml**

```yaml
esphome:
  name: 146-lvgl-interface
  friendly_name: 1.46_LVGL_Interface
  platformio_options:
    build_flags: "-DBOARD_HAS_PSRAM"                # Enable PSRAM support
    board_build.esp-idf.memory_type: qio_opi        # Set PSRAM memory type
    board_build.flash_mode: dio                     # Set flash operating mode

esp32:
  board: esp32-s3-devkitc-1                         # Target ESP32-S3 board model
  flash_size: 16MB                                  # Flash memory size
  framework:
    type: esp-idf                                   # Use ESP-IDF framework
    sdkconfig_options:
      CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240: y        # Set CPU clock to 240MHz
      CONFIG_ESP32S3_DATA_CACHE_64KB: y             # Enable 64KB data cache
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y           # Allow instructions from PSRAM
      CONFIG_SPIRAM_RODATA: y                       # Allow read-only data from PSRAM

psram:
  mode: octal                                       # PSRAM operating mode
  speed: 80MHz                                      # PSRAM clock speed

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "9KR5Ts7RsVyKrBTU7UtPPct6DAo+ThrwLE6OY1Mt190="

ota:
  - platform: esphome
    password: "8efe0c7e7ec5b46ff2835a07a62675a6"
```

Then, based on your existing code, follow the format shown in the figure to add or replace these codes.
(Other configurations remain unchanged)

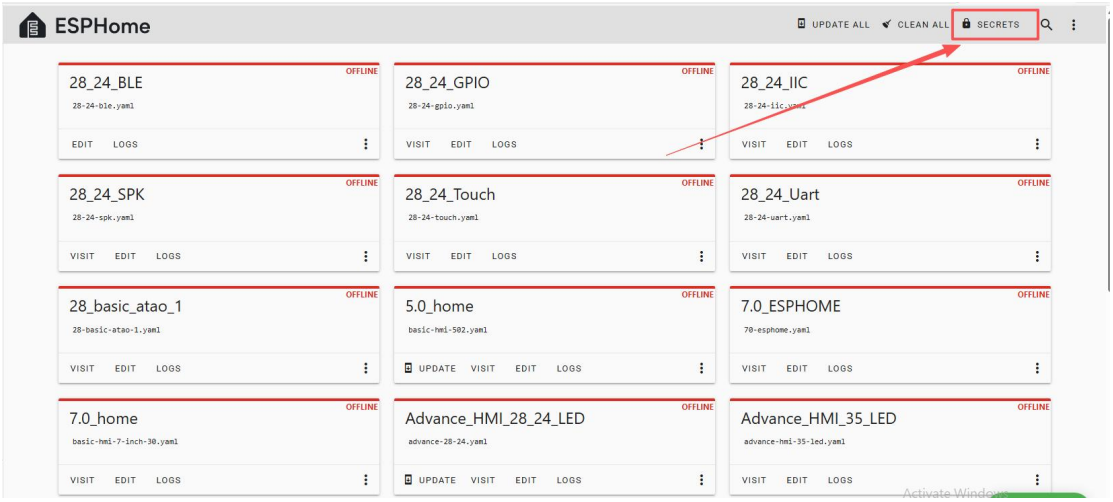
Remember to replace your own Wi-Fi name and password.

Note: This Wi-Fi connection must be in the same local network as your computer and Raspberry Pi!

146-lvgl-interface.yaml

```
25  logger:
26
27  # Enable Home Assistant API
28  api:
29    encryption:
30      key: "9KR5Ts7RsVyKrBTU7UtPPct6DAo+ThrwLE6OY1Mt190="
31
32  ota:
33    - platform: esphome
34      password: "8efe0c7e7ec5b46ff2835a07a62675a6"
35
36  wifi:
37    ssid: !secret wifi_ssid
38    password: !secret wifi_password
39
40    # Enable fallback hotspot (captive portal) in case wifi connection fails
41    ap:
42      ssid: "146-Lvgl-Interface"
43      password: "UMWQH1lXcNAT"
44
45  captive_portal:
46
```

Here, our Wi-Fi account and password are configured on the ESPHome main page.



Therefore, we can write it like this in the code, which means we are using the "elecrow888" Wi-Fi.

Of course, you can also directly write the detailed Wi-Fi account and password in the code.

146-lvgl-interface.yaml

```yaml
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "9KR5Ts7RsVyKrBTU7UtPPct6DAo+ThrwLE6OY1Mt190="

ota:
  - platform: esphome
    password: "8efe0c7e7ec5b46ff2835a07a62675a6"

wifi:
  ssid: "elecrow888"
  password: "elecrow2014"

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "146-Lvgl-Interface"
    password: "UMWQH1lXcNAT"

captive_portal:
```

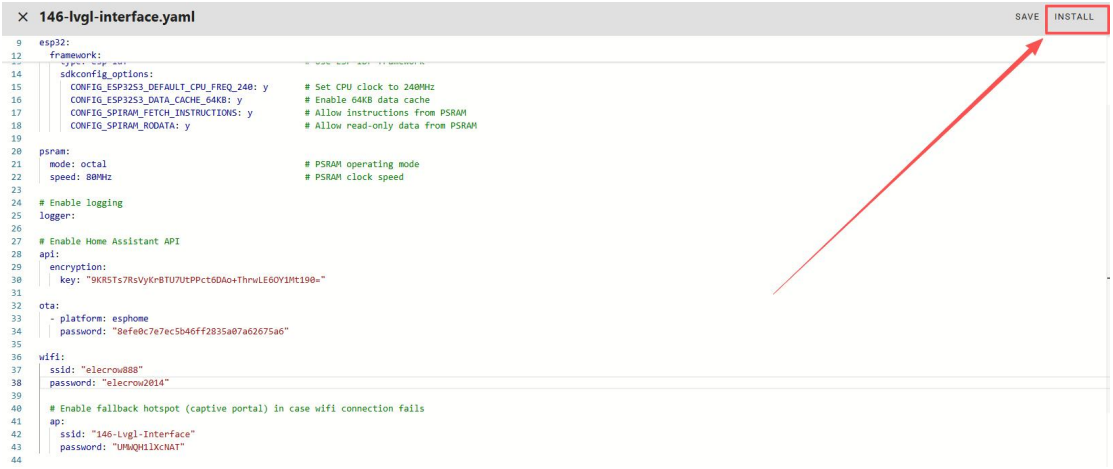
Note: These pieces of content are unique to the device you created. They cannot be the same as mine. Just keep the original values generated for your own device.

You can copy my other functional code.

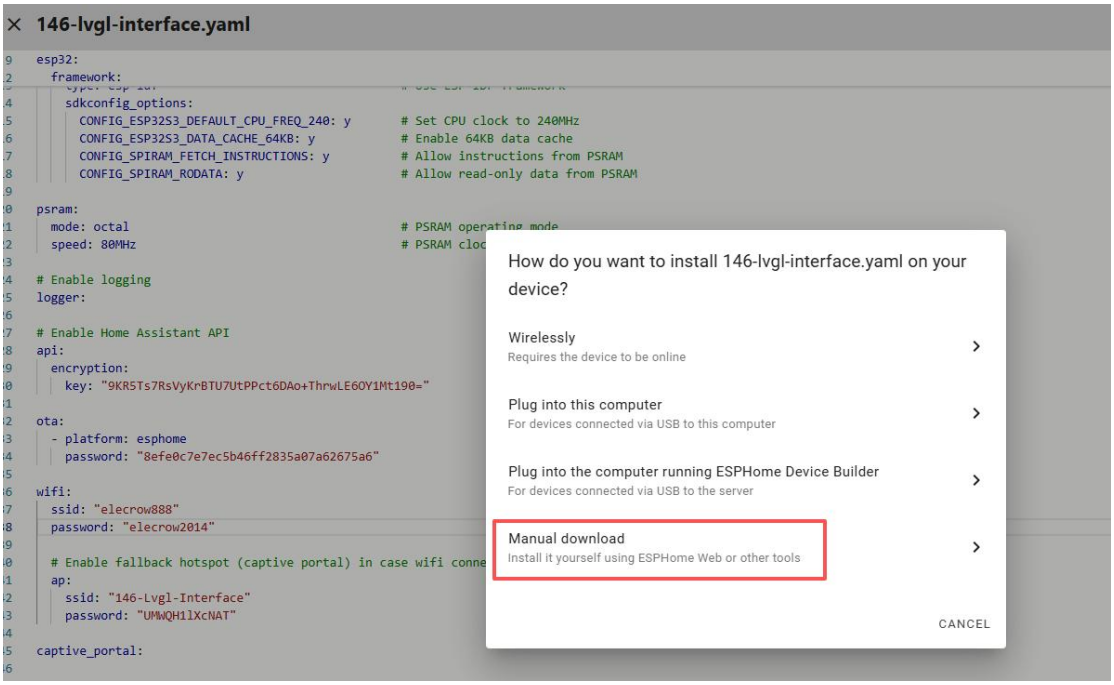
146-lvgl-interface.yaml

```yaml
esp32:
  framework:
    sdkconfig_options:
      CONFIG_ESP32S3_DEFAULT_CPU_FREQ_240: y       # Set CPU clock to 240MHz
      CONFIG_ESP32S3_DATA_CACHE_64KB: y            # Enable 64KB data cache
      CONFIG_SPIRAM_FETCH_INSTRUCTIONS: y          # Allow instructions from PSRAM
      CONFIG_SPIRAM_RODATA: y                      # Allow read-only data from PSRAM

psram:
  mode: octal                                      # PSRAM operating mode
  speed: 80MHz                                     # PSRAM clock speed

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "9KR5Ts7RsVyKrBTU7UtPPct6DAo+ThrwLE6OY1Mt190="

ota:
  - platform: esphome
    password: "8efe0c7e7ec5b46ff2835a07a62675a6"

wifi:
  ssid: "elecrow888"
  password: "elecrow2014"

  # Enable fallback hotspot (captive portal) in case wifi connection fails
  ap:
    ssid: "146-Lvgl-Interface"
    password: "UMWQH1lXcNAT"

captive_portal:
```

Once the code replacement is complete, click **"INSTALL"** in the top right corner.

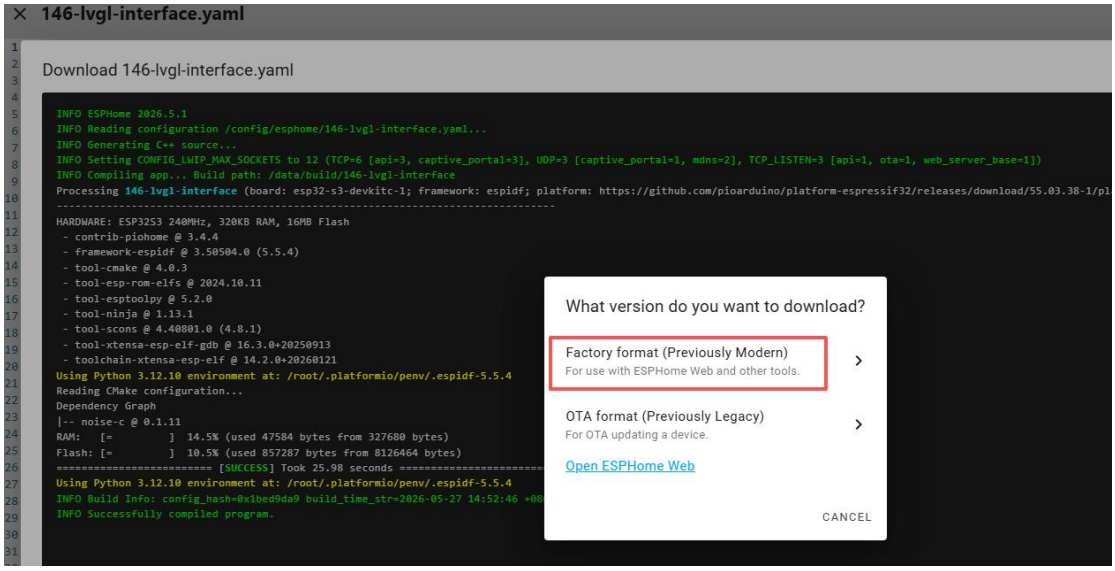


Select **"Manual download"**.

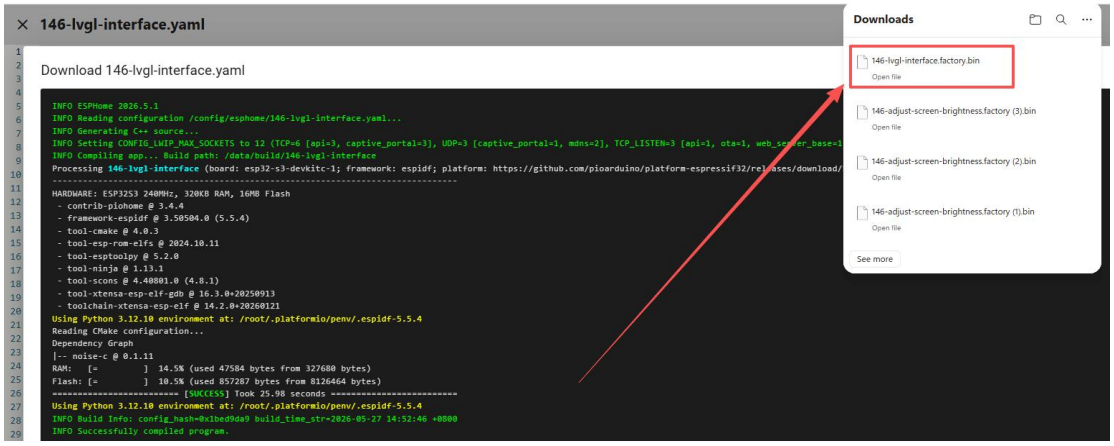


Wait for a few minutes until the installation is complete.

Then, select **"Factory format"**.



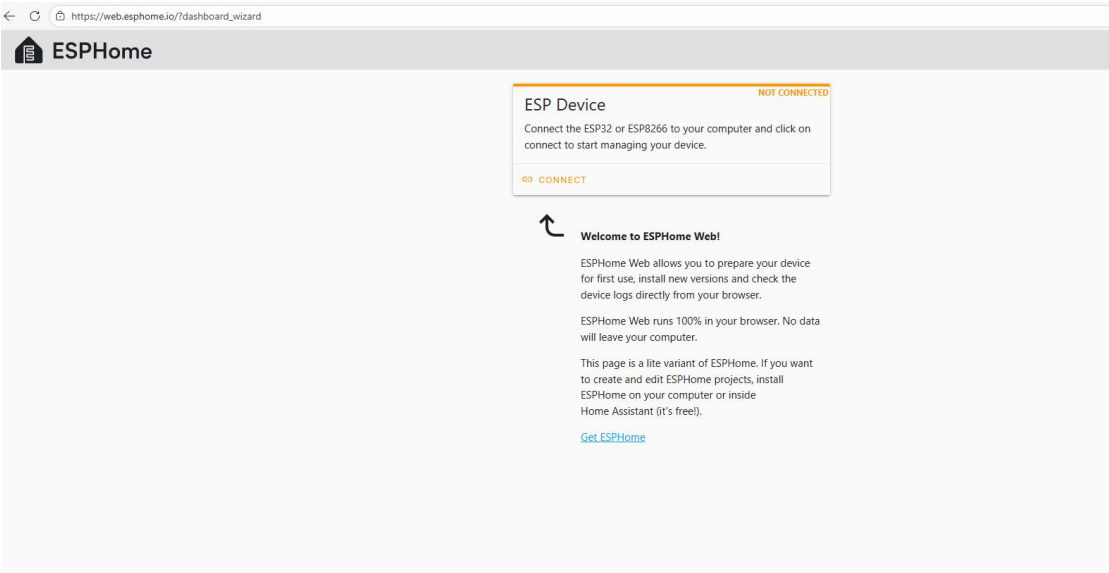
Once the download is complete, you will see the **.bin** file.



Remember the path of this **.bin** file.

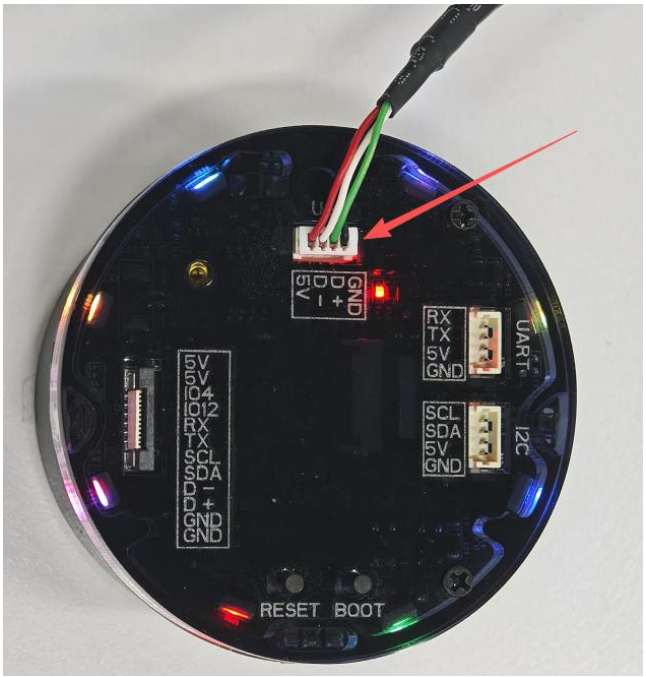
Open the following website: https://web.esphome.io/?dashboard_wizard

After opening this website, you will arrive at this interface:

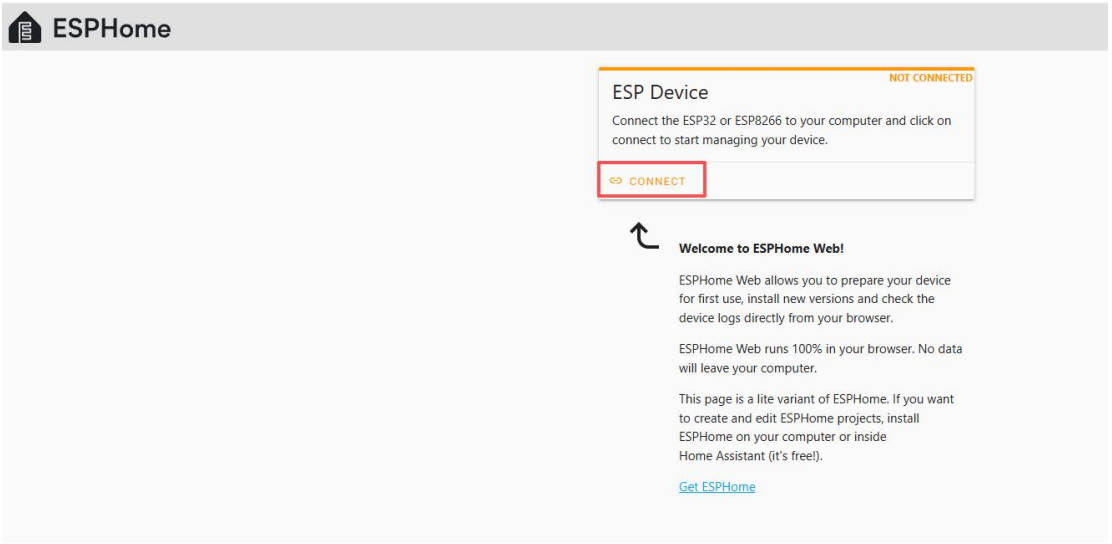


Next, we will flash this **.bin** file into the CrowPanel 1.46inch-HMI ESP32 Rotary Display .

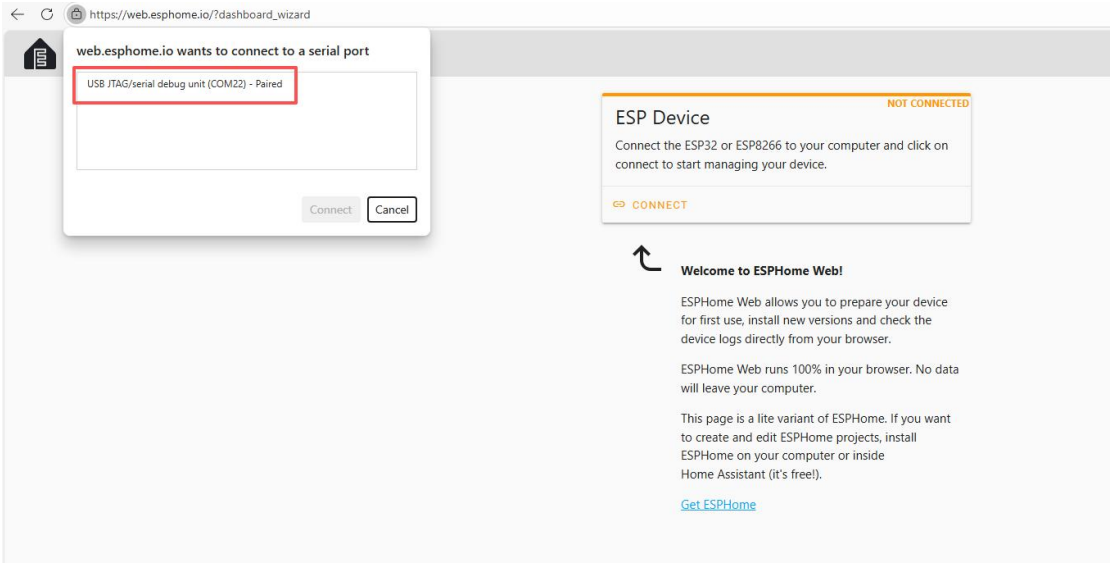
Connect the CrowPanel 1.46inch-HMI ESP32 Rotary Display to your computer.



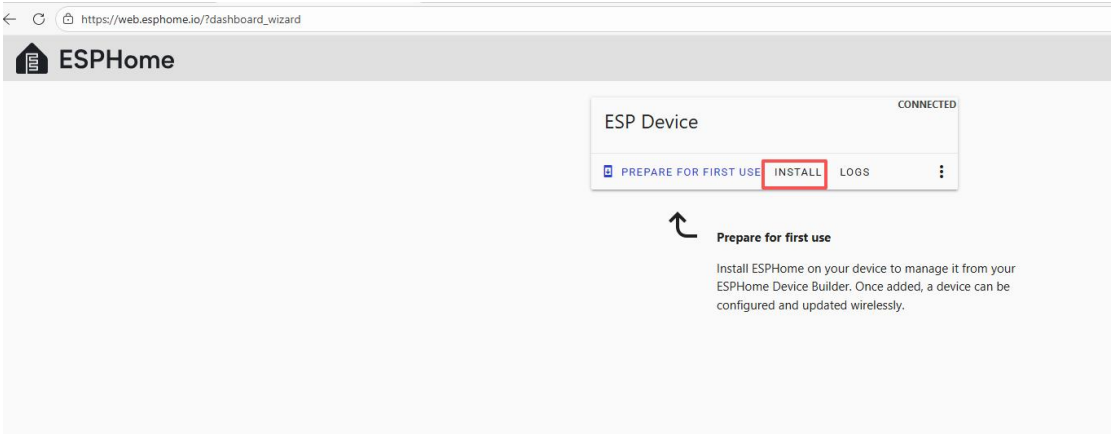
Click **"Connect"**



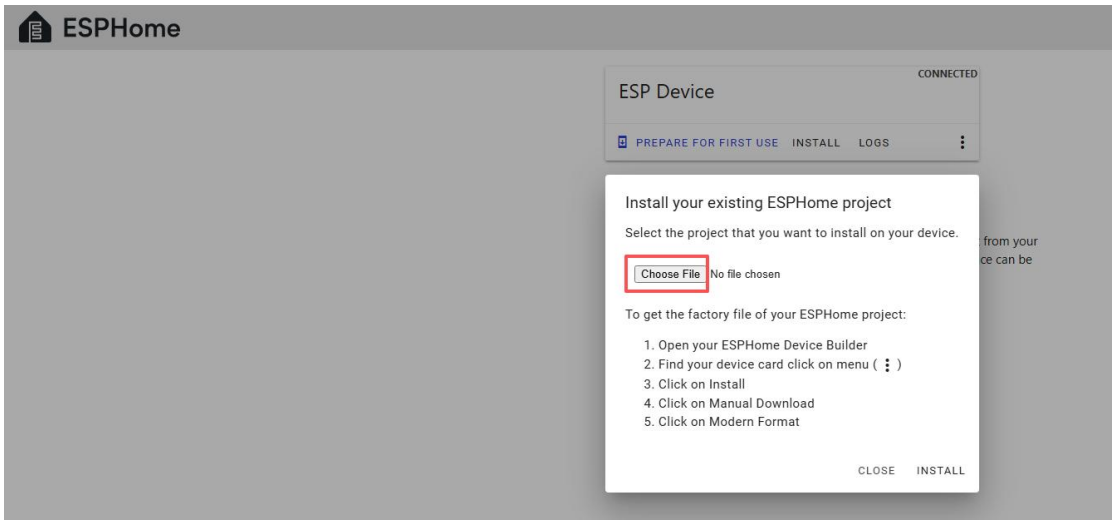
select the COM port, and connect it.



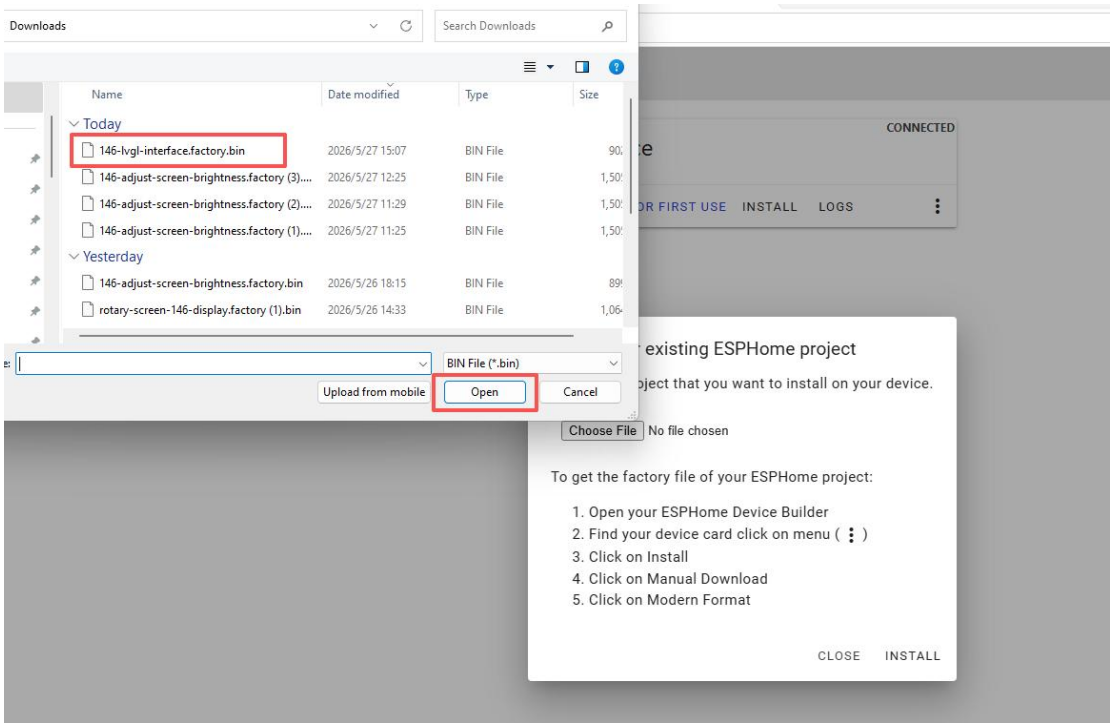
After connecting the CrowPanel 1.46inch-HMI ESP32 Rotary Display , click **"Install"**.



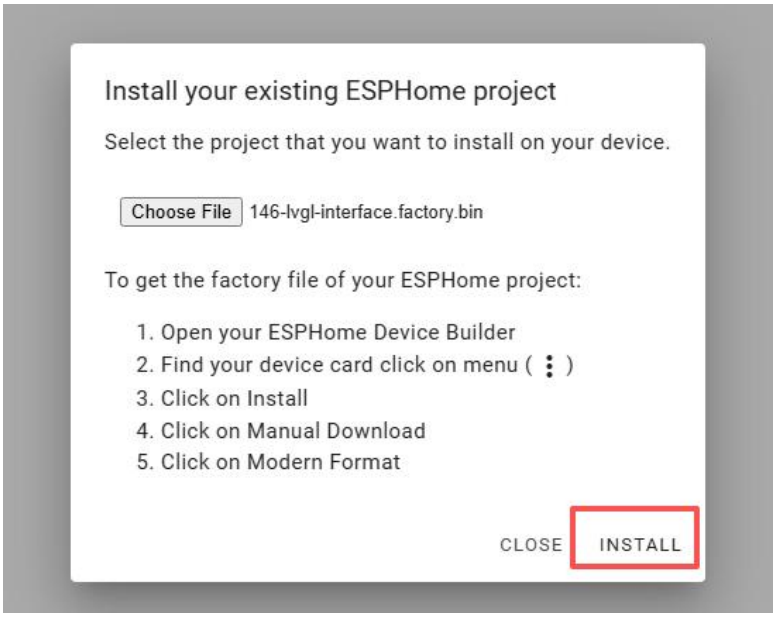
Add the **.bin** file you just downloaded, then click **"Install"**.



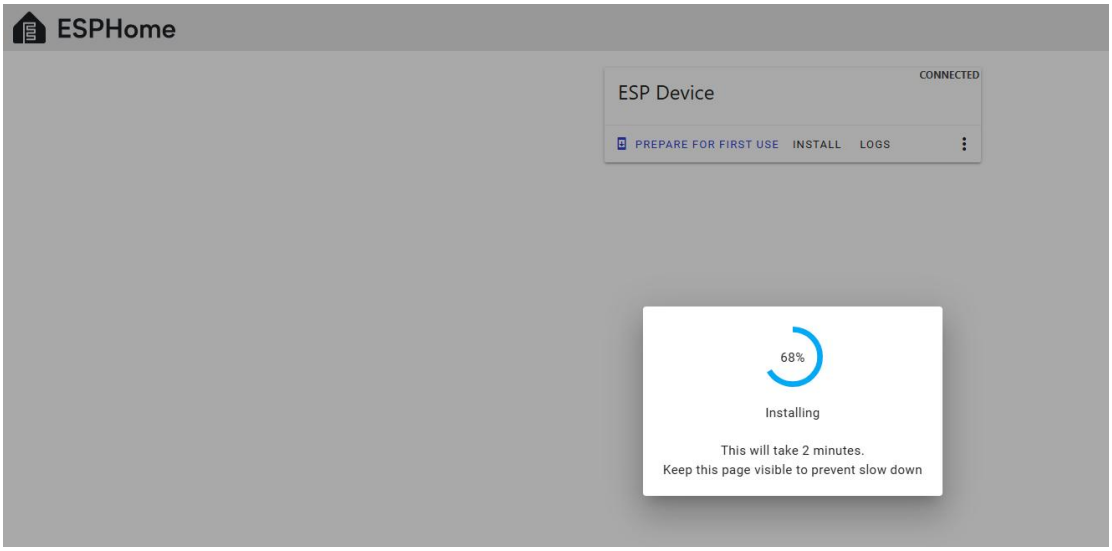
Then select the bin file that you just downloaded.



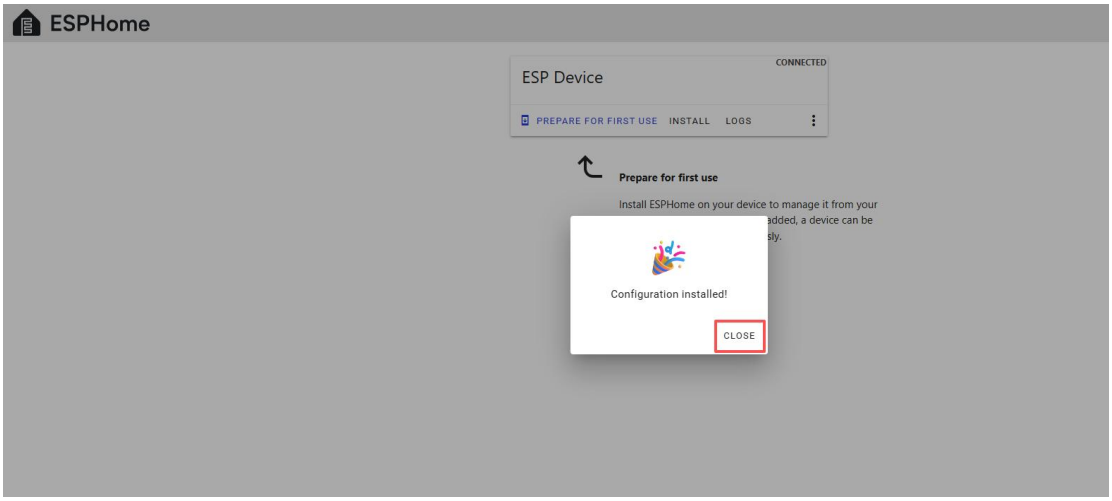
Click "INSTALL"



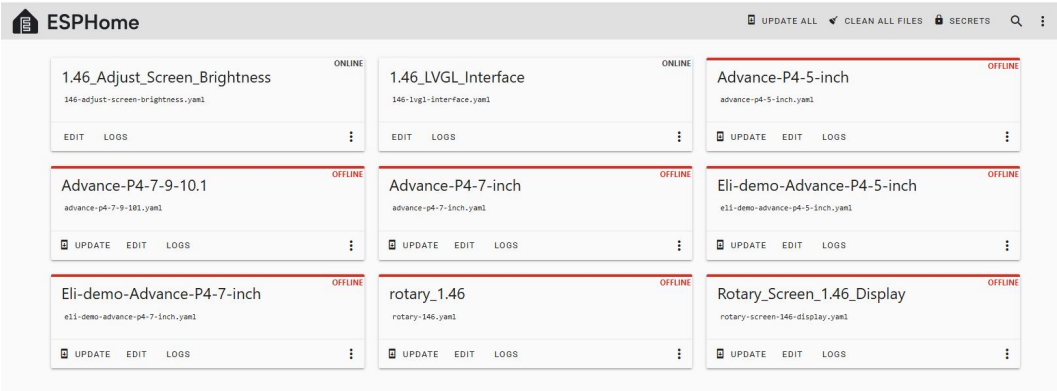
Wait for a few minutes.



After the installation is complete, click **"Close"**.

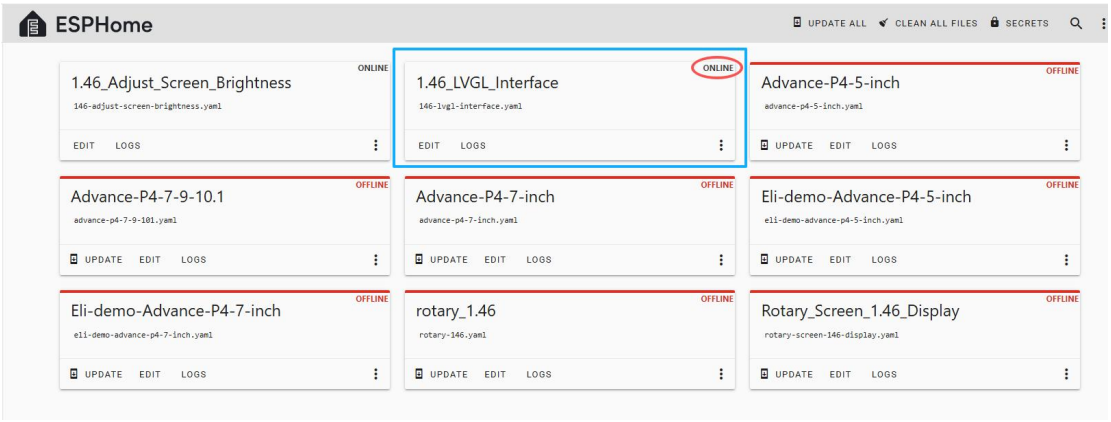


After successfully flashing the **.bin** file, return to the ESPHome page in Home Assistant.



Press the RESET button on the CrowPanel 1.46inch-HMI ESP32 Rotary Display.

Restart the ESP32 display, and you should see the device you created earlier show as **ONLINE** in the top right corner.



In addition, if you have also added the functional code we provided, you will be able to see the current screen brightness value displayed on the screen. You can also try rotating the knob. The current brightness value will be displayed on the screen in real time, and the slider on the screen will also move and update in real time as you rotate the knob.

(Rotate clockwise to increase the screen brightness, rotate counterclockwise to decrease the screen brightness)

After it shows ONLINE here, whenever you make modifications to the code later and want to upload the code again, you can use the Wireless upload method to upload the code, which will be much more convenient.