CrowPanel 1.46inch-HMI ESP32 Rotary Display

--- Arduino Tutorial

Function Overview

In this class, we will use SquareLine Studio to design the LVGL graphical interface, and further modify and improve the code based on the generated UI project file to realize the human-computer interaction function of the knob screen.

During the course, we will learn how to build multi-page LVGL interfaces, and combine touch screens and rotary encoders to achieve page switching, menu selection, and parameter adjustment operations; at the same time, we will also learn the usage methods of common interaction components such as PWM backlight control, RGB lighting effects control, and Arc circular slider.

In addition, this course will introduce the FreeRTOS multi-tasking mechanism to help you understand how ESP32 can simultaneously complete interface refreshing, input detection, and light animation control.

Download Software

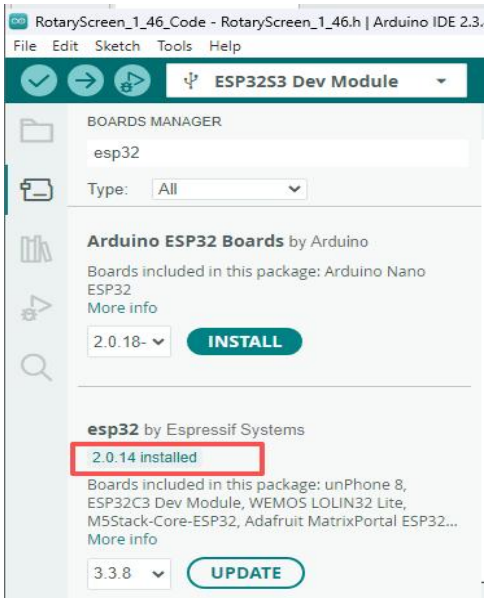
Get Started with Arduino IDE

Please click the link below to learn how to install the Arduino IDE and install the ESP32 development board in the Arduino IDE.

You can simply download the latest version of the Arduino IDE.

https://www.elecrow.com/wiki/Get_Started_with_Arduino_IDE.html

Please install the ESP32 version as 2.0.14.



Get Started with SquareLine Studio

Refer to the instructions in the link below on how to install the SquareLine Studio platform.

Here, all you need to do is install the SquareLine Studio platform.

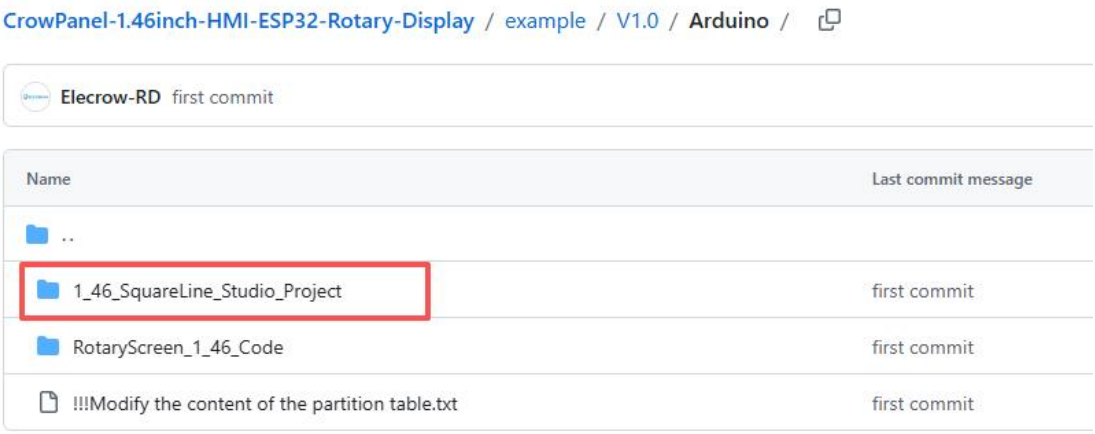
https://www.elecrow.com/wiki/7.0_3_Use_LVGL_library_to_create_UI_interface_and_light_up_lights.html

Here, please download the version 1.5.1 of the SquareLine Studio platform.

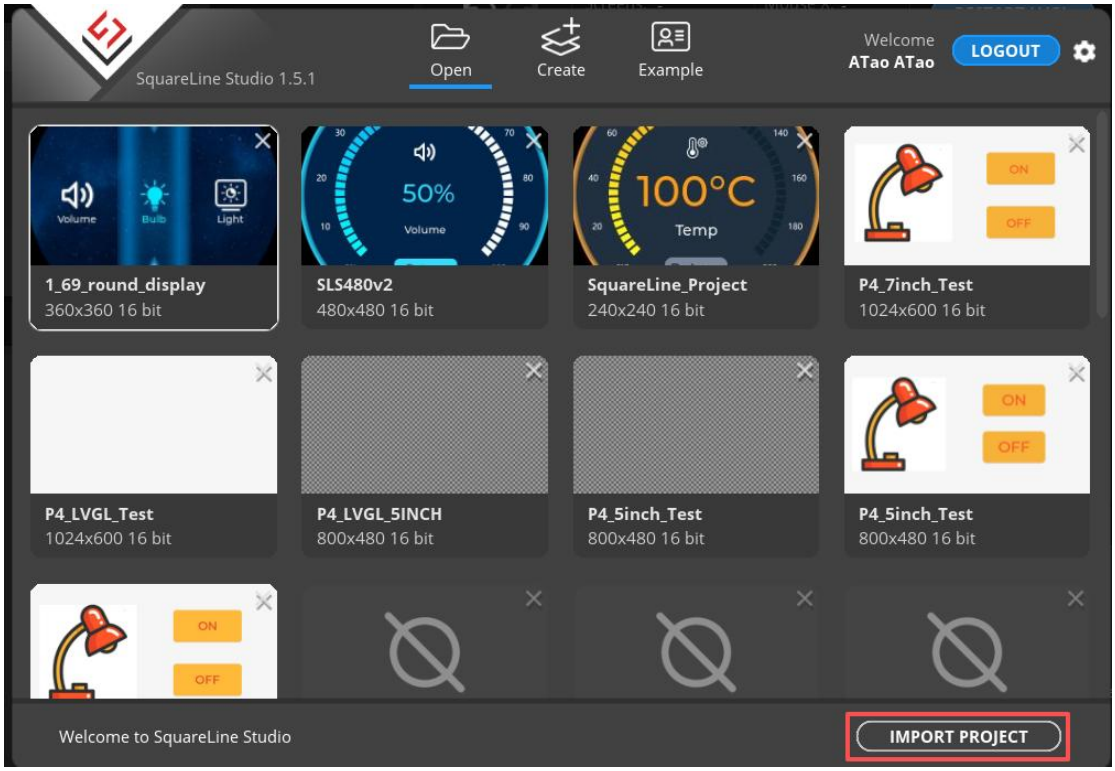
After downloading SquareLine Studio, we use it to open this UI project that we have already completed.

First, download this SquareLine Studio project that we have provided.

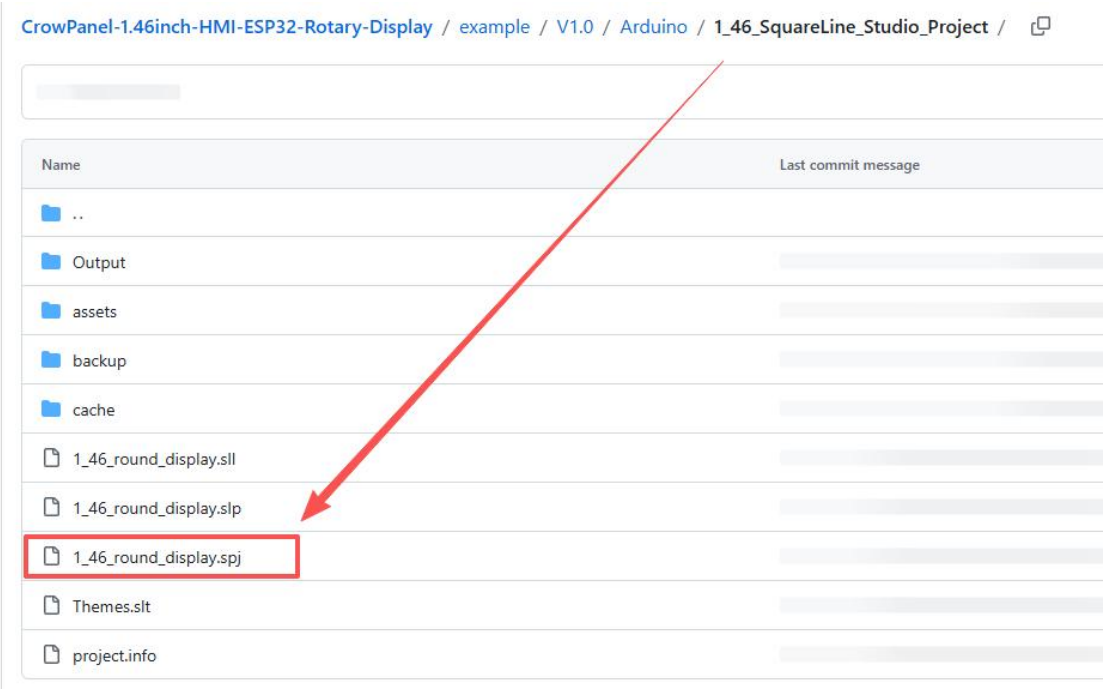
https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/tree/master/example/V1.0/Arduino



After downloading it, use the SquareLine Studio software to open this project.



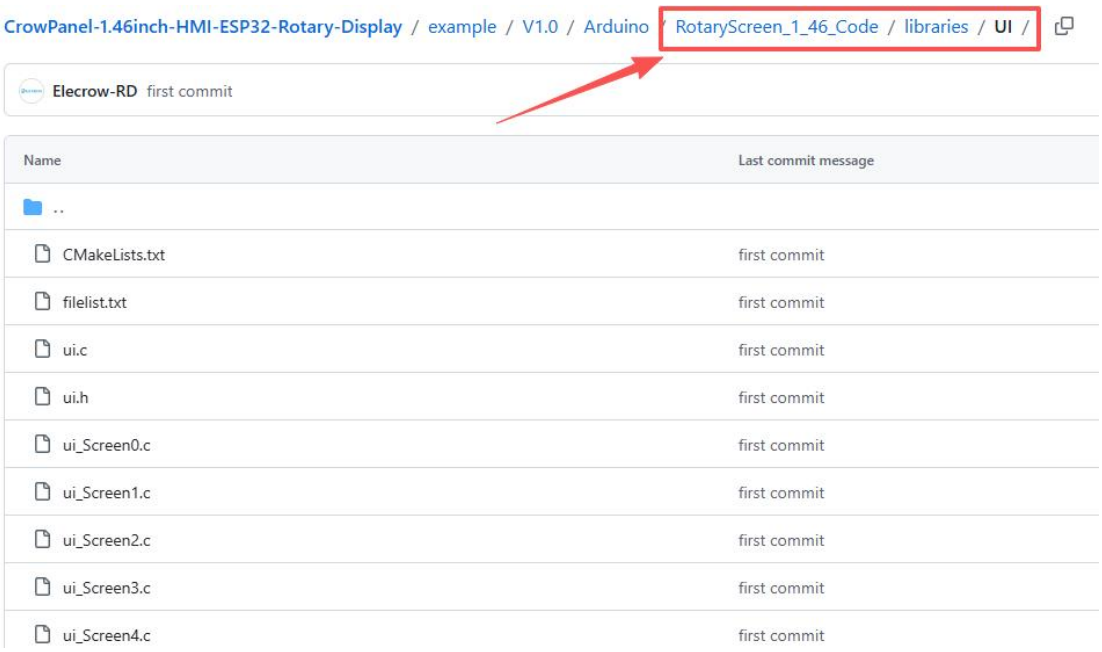
Then open this ".spj" file, and you will be able to see our complete knob screen UI interface project.



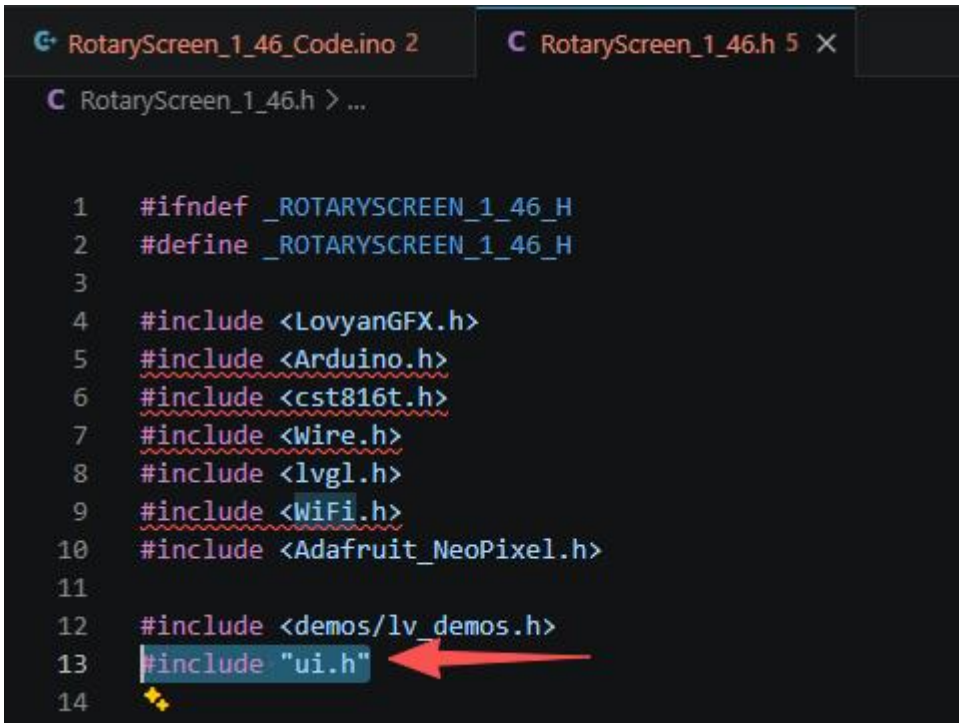
Complete UI Project:

You can make modifications based on our design. Please pay attention to every variable name of every interface in our project.

Then, place the exported UI file in the UI folder within the libraries folder.



In this way, we only need to add "#include "ui.h"" in the main program to be able to reference and access the UI interface code generated by SquareLine Studio; and these source code files related to the interfaces will all be saved in the "ui" folder within the project directory.

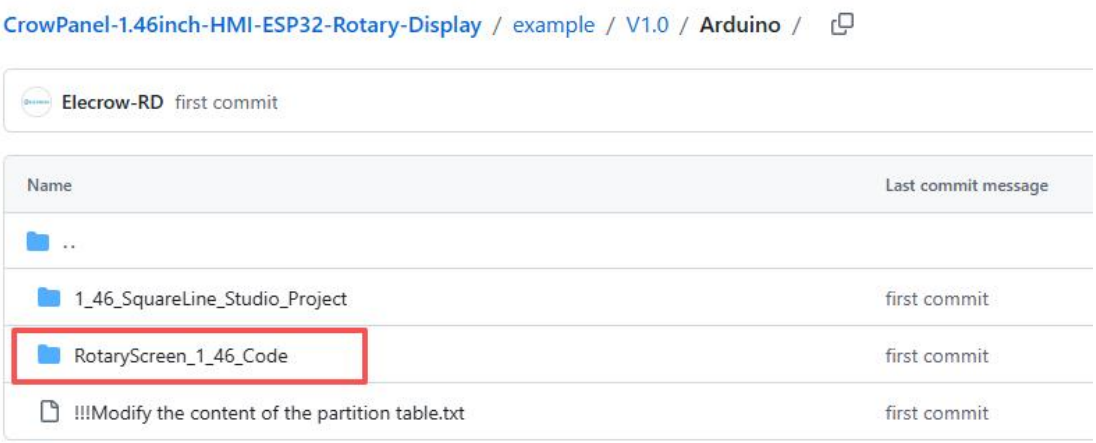


Open the code

Click the link below to download the function code for the knob screen of this lesson.

Code link:

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

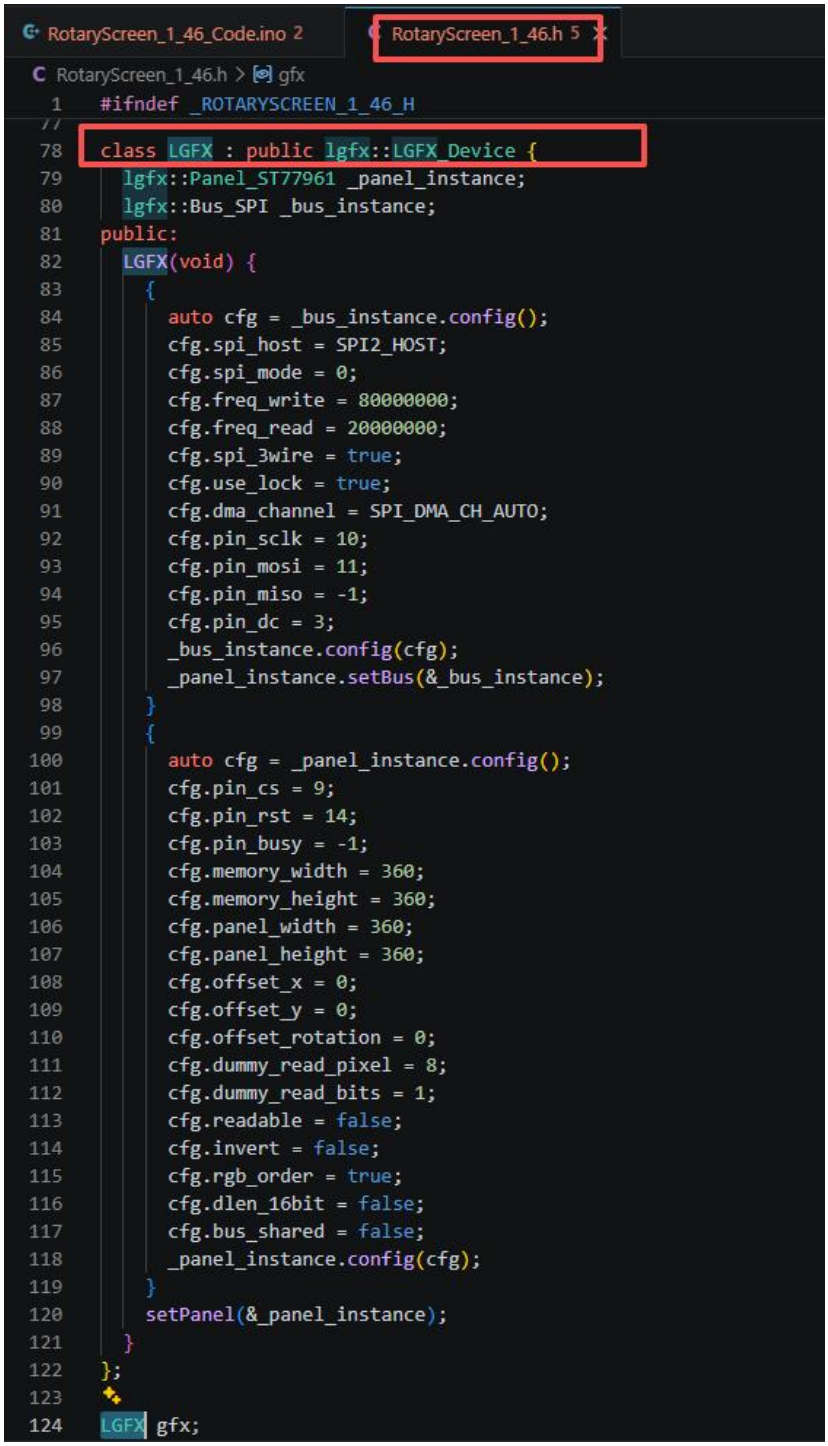


Code explanation

Next, let's take a detailed look at the function codes of the knob screen.

Screen display

Let's first take a look at the **LGFX** content in the **RotaryScreen_1_46.h** file.



The main function of this code is to configure and initialize a 360×360 circular TFT screen based on the ST77961 driver chip for the ESP32 using the LovyanGFX graphics library, thereby enabling the ESP32 to have graphic display capabilities.

The code first creates a custom display device class through "class LGFX : public lgfx::LGFX_Device", and defines the SPI bus object "_bus_instance" and the screen

driver object "_panel_instance" inside the class. The SPI bus is responsible for data communication between the ESP32 and the screen, while the ST77961 driver chip is responsible for actually controlling the screen pixel refresh.

In the constructor, the SPI communication parameters are first configured, including SPI controller, communication mode, 80 MHz high-speed write frequency, DMA automatic transmission, three-wire SPI mode, clock line, data line, and command control pin, etc. The activation of DMA allows the ESP32 to not need the CPU to transfer data pixel by pixel during screen refreshing, thereby significantly improving the refresh efficiency and smoothness of the LVGL interface;

Then, the screen itself is configured, including screen resolution, memory size, chip select pin, reset pin, color order, and whether to support screen reading, etc., to ensure that the graphic data can be correctly mapped to the actual display area;

Finally, "setPanel(&_panel_instance)" is used to formally bind the SPI bus with the screen driver, and ultimately a complete display driver system is constructed. Therefore, this class essentially tells the ESP32: which pins should be used, what communication method, what resolution and display parameters to drive this TFT circular screen, providing underlying hardware support for the subsequent LVGL graphic interface display.

All you need to do is to directly use this single display driver. We have already set it up for everyone.

All you need to do is to create a global screen driver instance object.

```
118
119    LGFX gfx;
```

As the sole operation interface between ESP32 and the physical display screen, all subsequent screen initialization, drawing, and refreshing operations are accomplished through this "gfx" object.

Next, let's take a look at **RotaryScreen_1_46_Code.ino** to see how the gfx library is used for screen display-related operations.

This code is the core initialization process for transforming the screen from a state without power or signal to a display-ready state:

First, the gfx.init() function is used to complete the hardware reset, register configuration, and communication establishment of the ST77961 screen.

Then, gfx.initDMA() is used to enable the high-speed transmission capability of the hardware DMA to improve the efficiency of screen refreshing.

Next, gfx.startWrite() takes over the SPI bus to prepare for drawing, and at the same time, the default drawing color and text size parameters are configured.

Finally, gfx.fillScreen(TFT_BLACK) is executed to send full-screen black data to the screen, completing the screen lighting, clearing, and putting the display into a ready-for-drawing state.

```
39  void my_disp_flush(lv_disp_drv_t *disp, const lv_area_t *area, lv_color_t *color_p) {
40    if (gfx.getStartCount() > 0) {
41      gfx.endWrite();
42    }
43    gfx.pushImageDMA(area->x1, area->y1, area->x2 - area->x1 + 1, area->y2 - area->y1 + 1, (lgfx::rgb565_t *)&color_p->full);
44
45    lv_disp_flush_ready(disp);
46  }
```

This function serves as the core bridge between the LVGL graphics library and the physical screen. Its role is to transfer the image data drawn by LVGL in memory to the screen for display at high speed through the hardware driver: Firstly, the function checks and terminates any unfinished SPI drawing operations that might have occurred previously to ensure communication security. Then, it calls the gfx.pushImageDMA hardware DMA function to precisely and rapidly send the image data of the specified area to the corresponding coordinate position on the screen, completing the rendering of the image. Immediately after this, it notifies LVGL

through lv_disp_flush_ready that the data transmission is complete, allowing LVGL to continue drawing the next frame of the image, thus enabling the screen to display UI, text, and patterns. This is the final execution entry point for enabling the screen to display UI, text, and patterns.

```
size_t buffer_size = sizeof(lv_color_t) * screenWidth * screenHeight;
buf  = (lv_color_t *)heap_caps_malloc(buffer_size, MALLOC_CAP_SPIRAM);
buf1 = (lv_color_t *)heap_caps_malloc(buffer_size, MALLOC_CAP_SPIRAM);
if (!buf)
  Serial.println("Failed to allocate for LVGL buf!");
if (!buf1)
  Serial.println("Failed to allocate for LVGL buf1!");
lv_disp_draw_buf_init(&draw_buf, buf, buf1, screenWidth * screenHeight);
```

This code is for creating the memory canvas required for screen drawing in LVGL: Firstly, calculate the memory size needed for a frame image that is suitable for a 360 × 360 screen. Then, allocate two separate display buffers, buf and buf1, from the PSRAM (external high-speed memory) of the ESP32. At the same time, perform anomaly detection on the memory allocation result. If the allocation fails, print a prompt message. Finally, register these two buffers as the double-buffer drawing space of LVGL, allowing LVGL to draw in one buffer while sending the other buffer to the screen for display, achieving smooth and efficient double-buffered screen refreshing, which is the core memory foundation for smooth screen display.

The complete process displayed on the entire screen can be summarized concisely in a few sentences as follows:

First, in the header file, define the LGFX screen driver class, configure the SPI communication pins, frequency, and parameters of the ST77961 driver chip, and create a global gfx object as the sole entry point for screen operations.

During the setup initialization stage, complete the screen hardware initialization through gfx.init(), enable high-speed DMA transmission through gfx.initDMA(), and allocate two PSRAM memory blocks as the double-buffer canvases for LVGL, and complete the initialization registration;

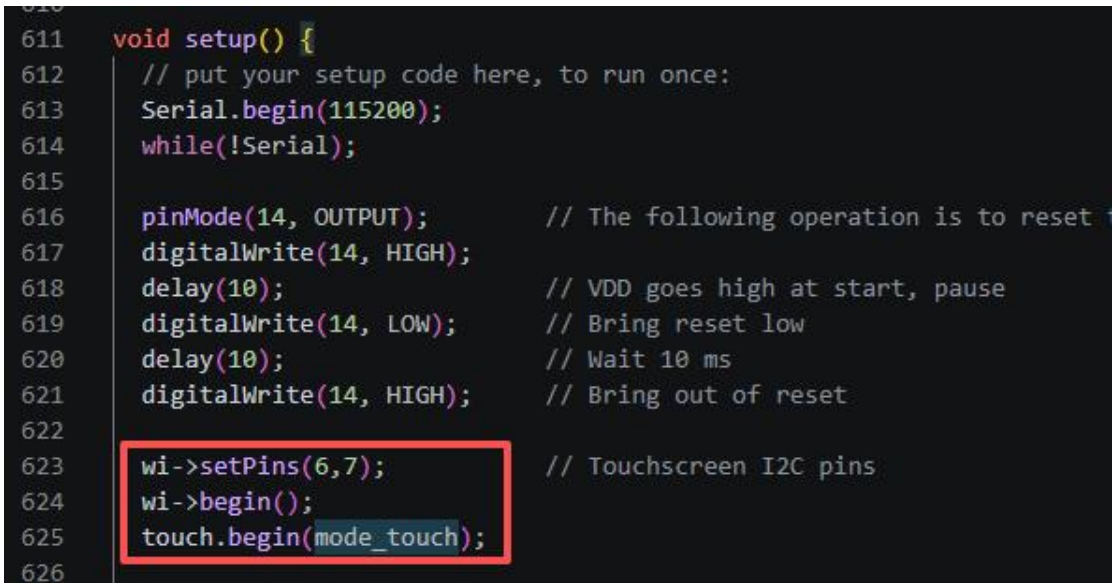
then, the LVGL graphics library draws all display contents such as UI interfaces, text, and sliders on the memory canvas, and after the drawing is completed, automatically calls the core refresh callback function my_disp_flush, which transmits the image data of the specified area on the canvas to the physical screen through hardware DMA at high speed.

Finally, after the screen receives the data, it completes the rendering of the picture, and the process repeats, thereby enabling the screen to continuously and smoothly display the complete dynamic interactive interface.

Screen touch

Now that we have figured out how to make the screen display normally, let's take a look at how to achieve screen touch functionality next.

This line of code creates a global CST816T capacitive touch object named "touch", explicitly specifying the use of the ESP32's hardware I2C bus "Wire" for communication with the touch chip. It also binds the interrupt pin 13 and reset pin 5 of the touch chip, allowing the program to subsequently read the coordinates of the finger and detect the touch state through this "touch" object. This is the hardware operation entry point for implementing touchscreen interaction.



This code is used to start and initialize the touch system: First, configure the SDA=6 and SCL=7 pins of the I2C bus and enable I2C communication. Then, call touch.begin() to start the CST816T touch chip, allowing the chip to enter the working state and start real-time detection of finger presses.

```
48  void my_touchpad_read(lv_indev_drv_t *indev_driver, lv_indev_data_t *data) {
49
50    //if ( gfx.getTouch( &touchX, &touchY ) ) {
51    if(touch.available())
52    {
53      data->state = LV_INDEV_STATE_PR;
54      if(touch.x == 0 && touch.y == 0)
55        return ;
56      data->point.x = touch.x;
57
58      data->point.y = touch.y;
59
60  #ifdef DEBUG_PRINT
61      Serial.print( "Data x " );
62      Serial.println( data->point.x );
63      Serial.print( "Data y " );
64      Serial.println( data->point.y );
65  #endif
66    }
67    else
68    {
69      data->state = LV_INDEV_STATE_REL;
70    }
71  }
```

This function serves as the core bridge between LVGL and the touch hardware. LVGL continuously calls this function to obtain the touch status: The function first checks if the CST816T detects a valid touch. If there is a press, it sets the status to "pressed" and passes the X and Y coordinates obtained from the chip to LVGL; if there is no press, it sets the status to "released", allowing the LVGL interface to know whether the user is currently touching and where the touch is, thereby enabling interactions such as button clicks and slider dragging.

```
675    static lv_indev_drv_t indev_drv;
676    lv_indev_drv_init(&indev_drv);
677    indev_drv.type = LV_INDEV_TYPE_POINTER;
678    indev_drv.read_cb = my_touchpad_read;
679    lv_indev_drv_register(&indev_drv);
680    delay(100);
```

This code officially integrates the hardware touch functionality into the LVGL system: First, create an input device driver object, set the type to "touch / mouse", and bind the above my_touchpad_read read callback function. Finally, register it with the LVGL kernel so that LVGL can obtain the touch coordinates in real time and achieve complete screen interaction.

The complete process of the touch function:

The system first creates a CST816T touch object and configures the I2C pins and interrupt pins. During the initialization stage, it starts the I2C communication and the touch chip, making the chip enter the real-time detection of pressing state. LVGL uses the registered input device driver to continuously call the my_touchpad_read

callback function. It reads whether there is a press and the corresponding X and Y coordinates from the CST816T chip, and then passes these touch data to the LVGL graphics library, enabling the interface to respond to click, sliding and other operations. Finally, it realizes the complete touch screen interaction function.

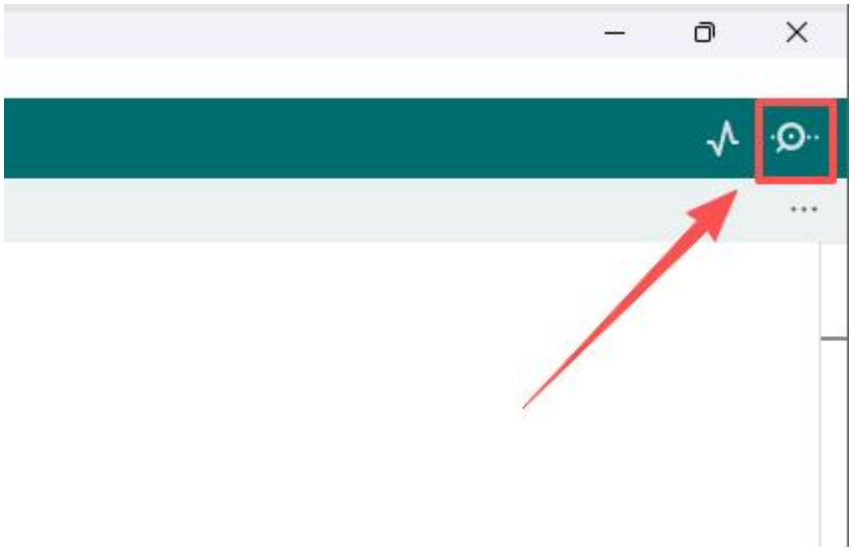
Initialization of the system in the Setup process

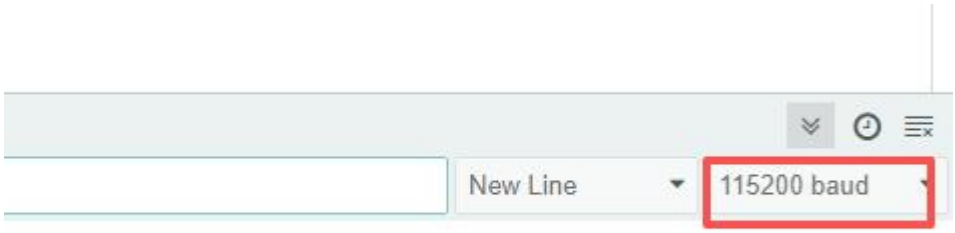
Previously, we discussed the drivers for screen display and screen touch. Now, let's take a look at how to initialize the system so that it can fully enter the operational state.

```
611  void setup() {
612    // put your setup code here, to run once:
613    Serial.begin(115200);
614    while(!Serial);
```

Enable the hardware serial communication of the ESP32 and set the baud rate to 115200. At the same time, wait for the successful connection of the serial port to ensure that the subsequent debugging information can be normally output to the serial port monitor.

Subsequently, by setting the same baud rate in the built-in serial monitor of the Arduino IDE, you will be able to see the relevant information printed.





Then you will be able to see the relevant printed information.

```
616   pinMode(14, OUTPUT);
617   digitalWrite(14, HIGH);
618   delay(10);
619   digitalWrite(14, LOW);
620   delay(10);
621   digitalWrite(14, HIGH);
```

The reset pin 14 of the screen driver chip ST77961 is controlled by high and low voltage levels to complete the standard hardware reset process, allowing the screen to transition from the power-off state to a ready state that is capable of initialization and communication.

Only in this way can the screen work properly.

```
623   wi->setPins(6,7);           // Touchscreen I2C pins
624   wi->begin();
625   touch.begin(mode_touch);
626
627   IIC.begin(I2C_SDA_PIN, I2C_SCL_PIN);     // IIC interface
628
```

Initialize the two I2C buses separately, configure the I2C pins 6 and 7 used by the touch chip CST816T, and start the communication.

At the same time, initialize the other I2C device to provide a stable communication foundation for the touch function and other I2C peripherals.

This interface corresponds to:

```
15  /*      I2C Define          */
16  //I2C SDA Pin
17  #define I2C_SDA_PIN       38
18  //I2C SCL Pin
19  #define I2C_SCL_PIN       39
```

Let's then proceed to look at the initialization of the relevant power supply pins.

```
629  pinMode(POWER_LIGHT_PIN, OUTPUT);
630  digitalWrite(POWER_LIGHT_PIN, LOW);      // The power indicator light is on
631
632  // pinMode(4 , OUTPUT);
633  // pinMode(12, OUTPUT);
634
635  pinMode(1, OUTPUT);
636  digitalWrite(1, HIGH);      // Pin 1 and pin 2 need to be turned on simultaneously in order to turn on the screen's power
637  pinMode(2, OUTPUT);
638  digitalWrite(2, HIGH);
639
640  pinMode(17, OUTPUT);        // Power pins for RGB lights
641  digitalWrite(17, HIGH);
```

Sequentially configure the power indicator light, the screen power supply pin, and the RGB light power supply pin. Turn on all the hardware power supplies to enable

the screen, touch, RGB lights and other modules to receive power and enter the working state.

```
653    lv_init();
654
```

lv_init(); It is the overall entry point of the LVGL graphics library, after execution, it will complete the initialization of core systems such as internal memory management, timers, display drivers, and input device drivers within LVGL, providing a basic operating environment for all GUI functions such as creating UI interfaces, refreshing the screen, and responding to touch buttons. It is the prerequisite for ensuring that the entire screen UI can function properly.

```
680    ui_init();
681
```

ui_init(); It is an interface initialization function automatically generated by UI design tools such as SquareLine Studio. It will create and load all the preset screens, arc sliders, text labels, icons and other UI objects in the project at once, complete the layout, style and default state configuration, and directly display the main interface, allowing the screen to immediately present the fully visualized interactive page designed by the user.

It comes from #include "ui.h".

```
707    delay(200);
708    initBacklight();
```

```
73  void initBacklight() {
74    ledcSetup(pwmChannel, pwmFreq, pwmResolution);
75    ledcAttachPin(SCREEN_BACKLIGHT_PIN, pwmChannel);
76    ledcWrite(pwmChannel, (50 * 255) / 100);
77  }
```

The "initBacklight()" function is used to initialize the PWM control for the screen backlight. First, it configures the frequency and resolution of the specified LEDC channel. Then, it binds this channel to the screen backlight pin 46. Finally, it outputs a 50% duty cycle PWM signal to set the screen backlight to a medium brightness and enable it to light up normally.

loop

```
714  void loop() {
715    // put your main code here, to run repeatedly:
716    lv_timer_handler(); /* let the GUI do its work */
717
718    vTaskDelay(pdMS_TO_TICKS(5));
719  }
```

"loop()" is the main loop function of ESP32. It continuously calls "lv_timer_handler()" to let the LVGL graphics library handle all GUI tasks such as interface refresh, animations, touch response, etc. Then, through a 5-millisecond RTOS delay, it releases the CPU resources to ensure the system is stable, smooth, and does not consume all the processor performance.

Built-in keys (single press / double press) integrated with the rotary encoder

This project uses the built-in key switch integrated with the rotary encoder as the input hardware. It detects the key press action through pin 41 and adopts an external interrupt mechanism to collect the key signal in real time along with software logic judgment. This enables precise and stable interaction control for single-click confirmation to enter and double-click to return to the home page. It is the core operation key of the entire screen interface.

This code binds the pin 41 to the external interrupt service routine, setting it to trigger on the falling edge (responding immediately upon the key press). This enables the CPU to avoid continuous polling and automatically jump to the interrupt function to handle the pressing event as soon as the key is pressed, achieving efficient and real-time key detection.

```
23  void IRAM_ATTR buttonISR() {
24    static unsigned long lastInterruptTime = 0;
25    unsigned long interruptTime = millis();
26    if (interruptTime - lastInterruptTime > debounceTime) {
27      debounceTime = 40;
28      if (digitalRead(SWITCH_PIN)) {
29        pressFlag = false;
30      } else {
31        pressFlag = true;
32        lastPressTime = interruptTime;
33        clickCount++;
34      }
35    }
36    lastInterruptTime = interruptTime;
37  }
```

This is the underlying processing interrupt function for key press. It activates the software anti-shake mechanism to prevent accidental triggering due to physical shaking of the keys. When an effective press is detected, it records the press timestamp, increments the clickCount by 1, sets the pressFlag to the pressed state, and provides the original data for subsequent judgment of single-click and double-click.

```
325     if (clickCount == 1 && millis() - lastPressTime > doubleClickTime) {
326       Serial.println("click ");
327       Serial.printf("clickCount Value:");
328       Serial.println(clickCount);
329       performClickAction();
330       debounceTime = 40;
331       clickCount = 0;
332     }
333     else if (clickCount == 2) {
334       Serial.println("double click");
335       Serial.printf("clickCount Value:");
336       Serial.println(clickCount);
337       performDoubleClickAction();
338       debounceTime = 160;
339       clickCount = 0;
340     }
```

This code is the core logic for detecting single-click and double-click events: By comparing the count value of clickCount with the pressing time interval, if only one press is made within a short period of time and it exceeds the double-click waiting time, it is determined as a single click; if two consecutive quick presses are made, it is determined as a double click, and the corresponding interface switching functions will be executed respectively.

```
void performClickAction()
{
  current_screen = lv_scr_act();
  if(current_screen == ui_Screen1)
  {
    if(current_screen == ui_Screen1)
    {
      if(screen1_index == 0)
      {
        _ui_screen_change(&ui_Screen2, LV_SCR_LOAD_ANIM_FADE_ON, 200 ,0, &ui_Screen2_screen_init);
      }
      else if(screen1_index == 1)
      {
        _ui_screen_change(&ui_Screen3, LV_SCR_LOAD_ANIM_FADE_ON, 200 ,0, &ui_Screen3_screen_init);
      }
      else if(screen1_index == 2)
      {
        _ui_screen_change(&ui_Screen4, LV_SCR_LOAD_ANIM_FADE_ON, 200 ,0, &ui_Screen4_screen_init);
      }
    }
  }
}
```

The click function "performClickAction" first obtains the current LVGL display screen. If it is the main interface "ui_Screen1", it will, based on the selected "screen1_index" value of the encoder, call the interface switching function. Using a 200-millisecond fade-in animation, it will respectively navigate to the adjustment sub-interfaces corresponding to volume, light bulb, and brightness, achieving the interactive effect of confirming the click and entering the corresponding function page.

At the beginning, we will be on the main interface.

At this point, you can turn the knob to switch to the interface you want to enter. After confirming, simply click and press the screen to achieve the effect of switching interfaces.

(Here, it's a screen press for switching. You can also switch by touching the icons.)

After clicking and pressing, you can enter the interface you have selected.

Next, let's look at the effect when you double-click on the screen to return.

```
102  void performDoubleClickAction()
103  {
104    current_screen = lv_scr_act();
105    if(current_screen == ui_Screen2 || current_screen == ui_Screen3 || current_screen == ui_Screen4)
106    {
107
108      _ui_screen_change(&ui_Screen1, LV_SCR_LOAD_ANIM_FADE_ON, 200 ,0, &ui_Screen1_screen_init);
109
110    }
111  }
```

The double-click function "performDoubleClickAction" first obtains the currently displayed screen. When it detects that the user is on any of the volume, light bulb, or brightness adjustment sub-interface, it immediately switches back to the main interface "ui_Screen1" through a 200-millisecond fade-in animation, achieving the interactive function of quickly returning to the home page upon double-click.

Sliding bar / Arc control

The working principles of the volume, bulb, and screen brightness circular sliders in the project are exactly the same. They all use the LVGL circular control to achieve 0-100% numerical adjustment. They support touch dragging and encoder rotation control.

When the slider value changes, the screen percentage text will be refreshed synchronously, and it will be converted into PWM signals to control the corresponding hardware separately, achieving a unified control logic for the synchronous linkage of UI display and physical output.

```
682    // Configure arc ranges (0-100 for all)
683    lv_arc_set_range(ui_VolumeArc,  0, 100);
684    lv_arc_set_range(ui_BulbArc   ,  0, 100);
685    lv_arc_set_range(ui_LightArc ,  0, 100);
686
687    // Set initial values
688    lv_arc_set_value(ui_VolumeArc, 50);
689    lv_arc_set_value(ui_BulbArc   , 50);
690    lv_arc_set_value(ui_LightArc , 50);
```

This code uniformly initializes three arc sliders when the system starts up. It sets the value range of each slider to 0-100 (representing a percentage), and sets the initial value of each slider to 50, so that the sliders are preset to be in the middle position upon startup and are ready for subsequent adjustments.

```
692    // Bind arc event callbacks for slider value changes
693    lv_obj_add_event_cb(ui_VolumeArc, volume_arc_event_cb , LV_EVENT_VALUE_CHANGED, NULL);
694    lv_obj_add_event_cb(ui_BulbArc   , bulb_arc_event_cb   , LV_EVENT_VALUE_CHANGED, NULL);
695    lv_obj_add_event_cb(ui_LightArc , light_arc_event_cb  , LV_EVENT_VALUE_CHANGED, NULL);
696
```

This code binds a unique callback function to each arc slider, and sets the trigger condition as "when the value changes". As long as the slider value changes (whether through touch dragging or encoder rotation), LVGL will automatically call the

corresponding function to complete operations such as value update, brightness synchronization, and text refresh.

Slider callback function (taking the light bulb as an example, applicable to all sliders)

```
129  static void bulb_arc_event_cb(lv_event_t *e)
130  {
131      lv_obj_t *arc = lv_event_get_target(e);
132      int value = lv_arc_get_value(arc);
133
134      // Update text label
135      char bulbText[8];
136      if (value == 100) {
137          snprintf(bulbText, sizeof(bulbText), "%d%%", value);
138      } else {
139          snprintf(bulbText, sizeof(bulbText), " %d%%", value);
140      }
141      lv_label_set_text(ui_BulbNum, bulbText);
142
143      // Update LED brightness on pin 43
144      int pwm_value = (value * 255) / 100;
145      ledcWrite(breathPwmChannel, pwm_value);
146  }
```

This is the core processing function of the slider. When the slider value changes, it will automatically obtain the current percentage value. First, it will synchronously refresh the value to the percentage text display on the screen, then convert the percentage into the duty cycle of the hardware PWM, and output it to the corresponding pin to achieve real-time synchronization of the slider value, the screen text, and the hardware brightness.

```
186          if (current_screen == ui_Screen2) {
187              int currentVol = lv_arc_get_value(ui_VolumeArc);
188              Serial.printf(" ++ currentVol = %d\n", currentVol);
189              int newVol = (currentVol + 5) > 100 ? 100 : currentVol + 5;
190              Serial.printf(" ++ END currentVol = %d\n", newVol);
191              lv_arc_set_value(ui_VolumeArc, newVol);
192
```

In the encoder task, the current slider value is read based on the rotation direction. Each rotation increases or decreases by a fixed 5%. The slider position is directly modified using lv_arc_set_value(), and at the same time, the callback function is triggered to synchronously update the text and the hardware brightness, thus achieving touchless adjustment of the slider by the rotating encoder.

The slider controls the entire workflow (ultimate summary):

When the system is powered on, it first initializes three circular sliders to have a range of 0-100 and a default value of 50%. Then, it binds a value change callback

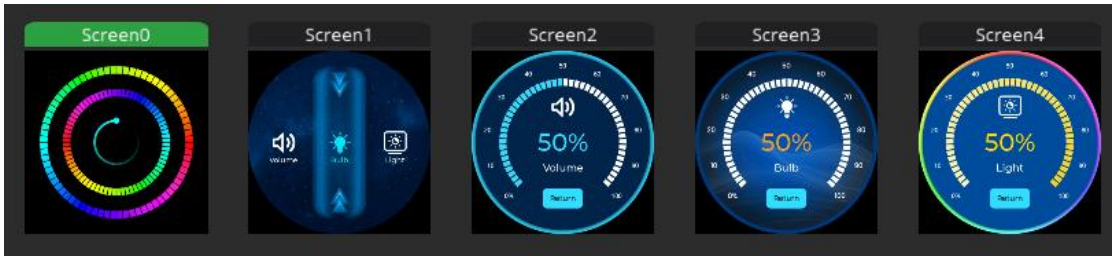
function to each slider. Users can either drag the sliders with their fingers or rotate the encoders to adjust the values.

Whenever the slider value changes, the callback function will automatically synchronize the percentage to the screen label and convert it into a PWM signal to control the brightness of the bulbs, backlight, etc. hardware in real time, forming a complete closed-loop control of UI slider → value display → hardware output.

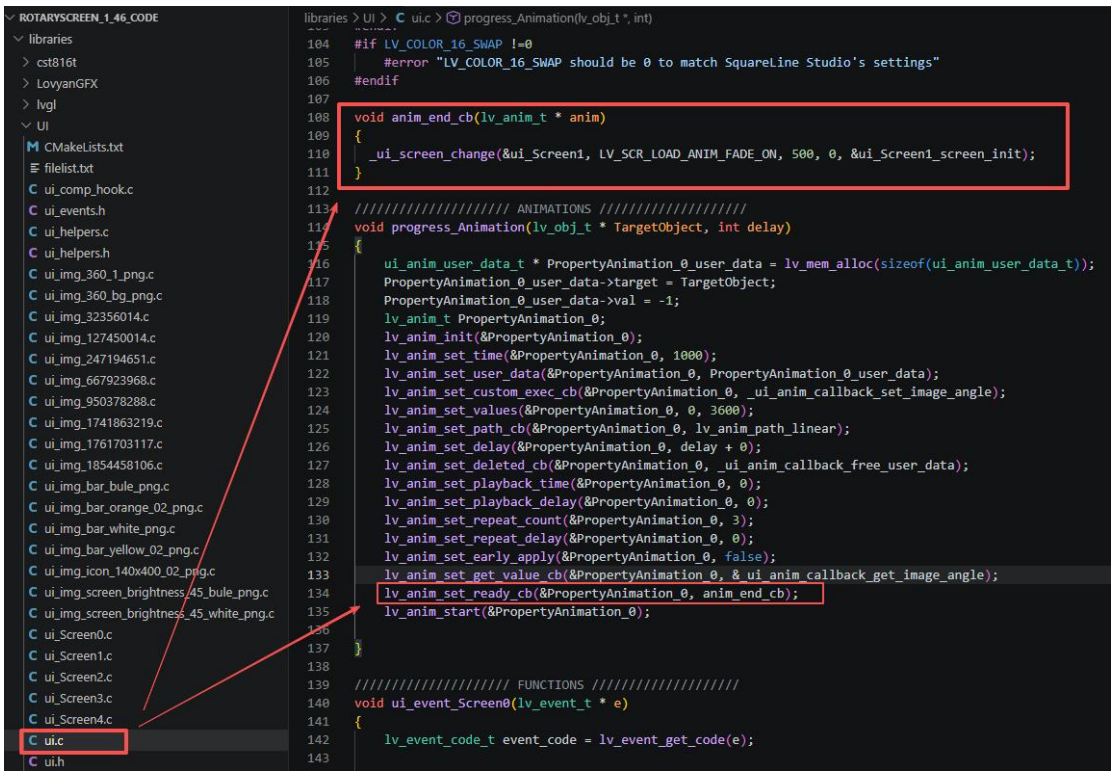
UI interface (0-4)

Next, let's take a look at how each UI interface is connected and see what processing has been done in the code.

First, after we reset the system, we saw Screen0.



Here, we have added these codes in the ui.c file within the UI folder of the code. This is done so that after Screen0 has been running for a period of time, we can transition to Screen1, which is more convenient for us to operate the system functions.



This selected code is an animation end callback mechanism: Firstly, the "anim_end_cb" function was defined, which is automatically called when the animation is completed. It switches back to the main interface "ui_Screen1" using a fade-in animation through the "_ui_screen_change" function; subsequently, in the "progress_Animation" function, this callback is bound to the progress bar animation using "lv_anim_set_ready_cb", thereby achieving the effect of automatically jumping back to the main interface after the progress bar animation is completed.

After entering Screen1, let's take a look at the display logic here. It's very ingenious.

Then in the code, we handle it in this way.

```
void processEncoder()
{
  current_screen = lv_scr_act();

  if(current_screen == ui_Screen1)
  {
    if(position_tmp == 1)
    {
      if(screen1_index < 2)
      {
        screen1_index++;
      }

      Serial.printf("cur_index : %d\n", screen1_index);
    }
    else if(position_tmp == 0)
    {
      if(screen1_index > 0)
      {
        screen1_index--;
      }

      Serial.printf("cur_index : %d\n", screen1_index);
    }
    updataScreen(screen1_index);
    position_tmp = -1;
  }
}
```

This function is the core function for rotating processing of the main interface encoder. It first obtains the current displayed screen, which is only effective for Screen1 of the main interface. Based on the rotation direction of the encoder (clockwise / counterclockwise), it performs addition or subtraction control within the range of 0 to 2 on the menu selection number screen1_index. After completing the number update, it calls updataScreen to refresh the interface and reset the rotation status, thereby realizing the basic logic for switching menu options through encoder rotation.

Then for "updataScreen",

```
349  void processEncoder()
353    if(current_screen == ui_Screen1)
364      else if(position_tmp == 0)
373      updataScreen(screen1_index);
374      position_tmp = -1;
375    }
376  }
377
378  void updataScreen(int index)
379  {
380    if(index < 0)
381    {
382      index = 0;
383    }
384    else if (index > 2)
385    {
386      index = 2;
387    }
388
389    Serial.printf("cur_index :%d\n", screen1_index);
390
391    lv_obj_add_flag(ui_volumeBlue,  LV_OBJ_FLAG_HIDDEN);
392    lv_obj_add_flag(ui_volumeWhite, LV_OBJ_FLAG_HIDDEN);
393    lv_obj_add_flag(ui_BulbBlue,    LV_OBJ_FLAG_HIDDEN);
394    lv_obj_add_flag(ui_BulbWhite,   LV_OBJ_FLAG_HIDDEN);
395    lv_obj_add_flag(ui_LightBlue,   LV_OBJ_FLAG_HIDDEN);
396    lv_obj_add_flag(ui_LightWhite,  LV_OBJ_FLAG_HIDDEN);
397
398    switch(index)
399    {
400      case 0:
401        //volume
402        lv_obj_clear_flag(ui_volumeBlue, LV_OBJ_FLAG_HIDDEN);
403        lv_obj_clear_flag(ui_volumeTextBlue, LV_OBJ_FLAG_HIDDEN);
404        lv_obj_add_flag(ui_volumeWhite, LV_OBJ_FLAG_HIDDEN);
```

This function is the core UI refresh function for displaying the menu on the main interface Screen1. Firstly, it limits the boundary of the input menu number to ensure it always falls within the range of 0-2 (corresponding to volume, bulb, and brightness). Then, it hides all icons and text to clear the previous interface state, and subsequently executes the branch logic based on the current selected menu number:

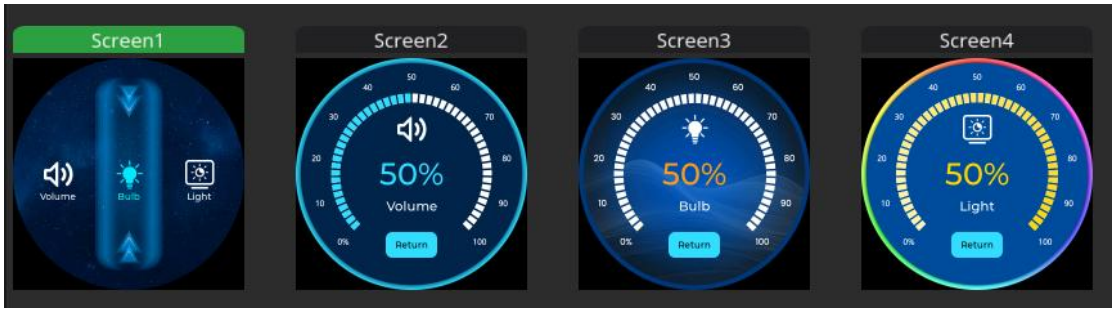
when 0 (volume) is selected, the volume icon and text display blue highlight and center alignment, the bulb displays white and moves to the right, and the brightness is hidden;

when 1 (bulb) is selected, the bulb icon and text display blue highlight and center alignment, the volume displays white and moves to the left, and the brightness displays white and moves to the right;

when 2 (brightness) is selected, the brightness icon and text display blue highlight and center alignment, the bulb displays white and moves to the left, and the volume is hidden.

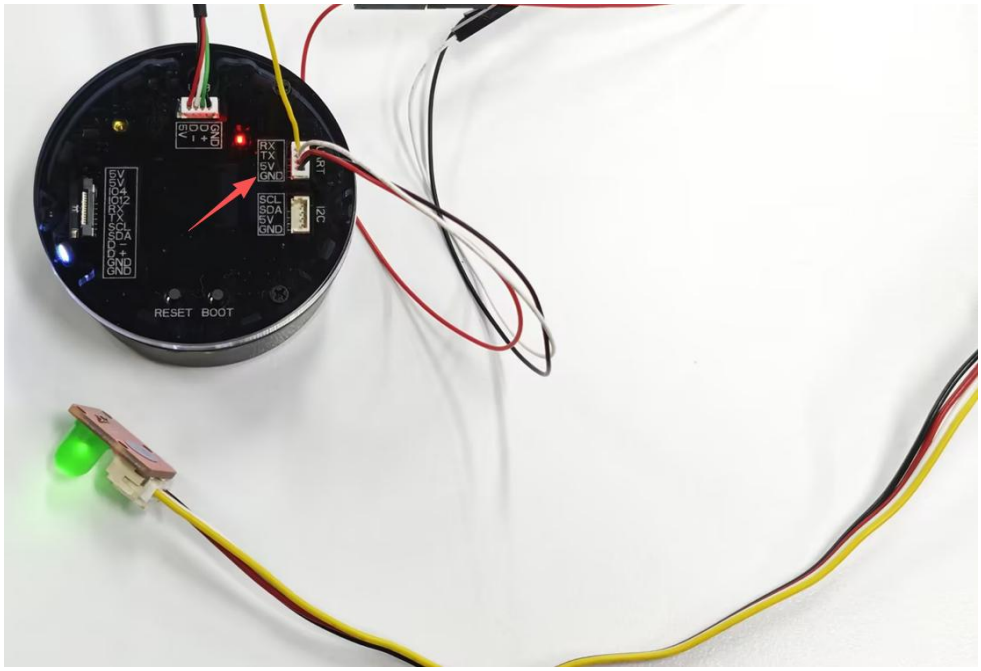
Through controlling the visibility, position movement, and color switching of the icons, a complete visual effect of smooth sliding switch of the main menu, highlighting of the current option, and moving of the unselected option to the side is achieved.

Screens 2, 3, and 4 are all functional sub-interfaces of the project, corresponding to volume adjustment, bulb brightness adjustment, and screen backlight adjustment respectively. The working principles of these three are exactly the same: After entering the interface, you can control the circular slider by rotating the encoder to increase or decrease the values. When the slider changes, the percentage text is refreshed synchronously, and the values from 0 to 100% are converted into PWM signals for output, respectively controlling the corresponding hardware to achieve real-time adjustment of volume, light, and screen brightness. Double-clicking the button allows you to return from any sub-interface to the main interface Screen 1.



For Screen 3, by turning the knob or sliding the slider, you can control the brightness of the LED bulb connected to the UART interface. During the operation, you need to connect the SIG pin of the bulb to the TX pin of the UART interface, 5V corresponds to the 5V pin, and GND corresponds to the GND pin.

As shown in the figure



The Crowtail LED in the picture is a product of our company. If you need it, you can visit the following link to have a look.

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

The implementation of the rotating encoder

The statement xTaskCreatePinnedToCore(encTask, "ENC", 2048, NULL, 1, &encTaskHandle, 0) executed in the setup() function is a crucial operation in the entire embedded system for starting the real-time encoder service. It is based on the FreeRTOS real-time operating system kernel and encapsulates all the functions of the encoder, such as rotation detection, direction determination, screen control, single-click and double-click processing, into a separate, preemptive, and fixed to CPU core 0 running real-time task.

```
711     xTaskCreatePinnedToCore(encTask, "ENC", 2048, NULL, 1, &encTaskHandle, 0);
712   }
```

The first parameter "encTask" specifies the actual executor of the task, which is the core task of the encoder that we will elaborate on in detail later.

The second parameter "ENC" is the name given to the task, mainly used for identifying the task during debugging.

The third parameter 2048 allocates 2KB of stack space for the task, which is sufficient to store local variables, function calls, and status information during the task execution process.

The fourth parameter NULL indicates that no additional parameters need to be passed when the task starts.

The fifth parameter 1 sets the priority of the task to 1, ensuring that the encoder task can obtain a stable running time slice and will not be preempted by lower-priority tasks.

The sixth parameter &encTaskHandle is used to save the task handle, facilitating subsequent management such as suspending, restoring, or deleting the task.

```
9   /*RTOS Task*/
10  TaskHandle_t ledTestTaskHandle = NULL;
11  TaskHandle_t encTaskHandle = NULL;
12
```

The last parameter 0 forces this task to run on CPU core 0 of the ESP32, avoiding resource contention with high-frequency tasks such as LVGL interface refresh and touchscreen processing running on core 1, thereby ensuring the real-time, stability, and response speed of the encoder signal acquisition, making the entire knob operation smooth and without delay.

Next, let's take a look at the implementation of "encTask".

```
167  void encTask(void *pvParameters) {
168    while (1) {
169      // Read the current state of CLK
170      currentStateCLK = digitalRead(ENCODER_A_PIN);
```

This code serves as the "entry point and framework" for the entire encoder task. It first defines the task function encTask, which follows the standard format of void type and takes void* as parameters. Here, pvParameters is used as the entry parameter for the task but is not utilized. This is the standard way of writing embedded RTOS tasks.

The while (1) infinite loop inside the function is the core design of the embedded peripheral task. Since the encoder needs to continuously and real-time collect signals, the infinite loop ensures that the task never exits once it is started and remains in the collection and processing state, avoiding the situation of missed signal collection.

The following currentStateCLK = digitalRead(ENCODER_A_PIN); is the first core operation of the task - real-time reading of the level state of the encoder A phase pin (high level 1 or low level 0). The A phase serves as the clock phase of the encoder, and its level change is the key to determining whether the encoder is rotating. Each cycle re-reads this level to ensure that the level jump during the encoder's rotation can be captured in time.

```
174      if (currentStateCLK != lastStateCLK) {
175
176        current_screen = lv_scr_act();
```

The core function of this code is to determine whether the encoder has rotated. It employs the most commonly used edge detection method in embedded systems.

The underlying principle is as follows: When the rotating encoder rotates, the level of phase A will change (from 0 to 1 or from 1 to 0), while if the encoder does not rotate, the level of phase A will remain stable. Therefore, by comparing whether the current read level of phase A (currentStateCLK) is different from the previously saved level of phase A (lastStateCLK), it is possible to determine whether the encoder has rotated.

When a level difference is detected (indicating rotation), the next step is to execute the line "current_screen = lv_scr_act ();", which is to obtain the current screen that LVGL is displaying.

Because the functions of the encoder are completely different on different screens - when rotating on screen 1 (main menu), it is used to switch menu options; when rotating on screens 2, 3, and 4 (sub-pages), it is used to adjust the values of

corresponding controls. Only by clearly identifying the current screen can the subsequent corresponding control logic be executed.

```
180         if (digitalRead(ENCODER_B_PIN) != currentStateCLK) {
181           if (abs(last_counter - counter) > 200) {
182             continue;
183           }
184           position_tmp = 1;
```

```
307           position_tmp = 0;
308           counter--;
309           currentDir = "CW";
310         }
```

This code is the core for determining the rotation direction of the encoder. Essentially, it is the software decoding of the orthogonal signals of the incremental encoder.

The core principle is as follows: The A phase and B phase of the encoder are orthogonal signals (with a phase difference of 90°). When there is a level change in the A phase, reading the level state of B phase at that moment can determine the rotation direction - if the level of B phase is different from the current level of A phase, it indicates that the encoder is rotating clockwise. At this time, set position_tmp = 1 to mark the direction as "increase", and increment the rotation step counter counter.

Also, record the direction string in currentDir (for serial port debugging); if the level of B phase is the same as the current level of A phase, it indicates that the encoder is rotating counterclockwise. At this time, set position_tmp = 0 to mark the direction as "decrease", decrement the counter counter, and also record the direction string.

```
186         if (current_screen == ui_Screen2) {
187           int currentVol = lv_arc_get_value(ui_VolumeArc);
188           Serial.printf(" ++ currentVol = %d\n", currentVol);
189           int newVol = (currentVol + 5) > 100 ? 100 : currentVol + 5;
190           Serial.printf(" ++ END currentVol = %d\n", newVol);
191           lv_arc_set_value(ui_VolumeArc, newVol);
192
193           char volumeText[8];
194           if (newVol == 100) {
195             snprintf(volumeText, sizeof(volumeText), "%d%%", newVol);
196             lv_label_set_text(ui_VolNum, volumeText);
197           } else {
198             snprintf(volumeText, sizeof(volumeText), " %d%%", newVol);
199             lv_label_set_text(ui_VolNum, volumeText);
200           }
201
202         } else if (current_screen == ui_Screen3) {
203           int currentBulb = lv_arc_get_value(ui_BulbArc);
204           Serial.printf(" ++ currentBulb = %d\n", currentBulb);
205           int newBulb = (currentBulb + 5) > 100 ? 100 : currentBulb + 5;
206           Serial.printf(" ++ END currentBulb = %d\n", newBulb);
207           lv_arc_set_value(ui_BulbArc, newBulb);
208
209           char BulbText[8];
210           if (newBulb == 100) {
211             snprintf(BulbText, sizeof(BulbText), "%d%%", newBulb);
212             lv_label_set_text(ui_BulbNum, BulbText);
213           } else {
214             snprintf(BulbText, sizeof(BulbText), " %d%%", newBulb);
215             lv_label_set_text(ui_BulbNum, BulbText);
216           }
217           // Update LED brightness on pin 43
218           int pwm_value = (newBulb * 255) / 100;
219           ledcWrite(breathPwmChannel, pwm_value);
220
221         } else if (current_screen == ui_Screen4) {
```

This code implements the specific control logic for clockwise rotation. The core is "perform the operation of increasing the value of the corresponding control based on the current screen".

When it is determined that the encoder is rotating counterclockwise (position_tmp = 1), the following steps are taken: first, through the if-else if branches, determine which sub-screen is currently active: if it is screen 2 (volume adjustment page), then first obtain the current value of the volume arc control using lv_arc_get_value (ui_VolumeArc), then calculate the new volume value - increase by 5 each time, while setting the upper limit to 100 (to avoid values exceeding a reasonable range), and finally update the display of the arc control using lv_arc_set_value (ui_VolumeArc, newVol); if it is screen 3 (light bulb brightness page) or screen 4 (screen brightness page), the logic is exactly the same, incrementing the value of the corresponding control by 5 each time, and simultaneously updating the hardware PWM signal (light bulb brightness is controlled by breathPwmChannel, screen

brightness is controlled by pwmChannel), achieving "synchronization of software display and hardware control".

```
244         } else {
245           if (one_test == false)
246           {
247             one_test = true;
248             continue;
249           }
250
251           if (current_screen == ui_Screen2) {
252             int currentVol = lv_arc_get_value(ui_VolumeArc);
253             Serial.printf(" -- currentVol = %d\n", currentVol);
254             int newVol = (currentVol - 5) < 0 ? 0 : currentVol - 5;
255             Serial.printf(" -- END currentVol = %d\n", newVol);
256             lv_arc_set_value(ui_VolumeArc, newVol);
257
258             char volumeText[8];
259             if (newVol == 100) {
260               snprintf(volumeText, sizeof(volumeText), "%d%%", newVol);
261               lv_label_set_text(ui_VolNum, volumeText);
262             } else {
263               snprintf(volumeText, sizeof(volumeText), " %d%%", newVol);
264               lv_label_set_text(ui_VolNum, volumeText);
265             }
266
267           } else if (current_screen == ui_Screen3) {
268             int currentBulb = lv_arc_get_value(ui_BulbArc);
269             Serial.printf(" -- currentBulb = %d\n", currentBulb);
270             int newBulb = (currentBulb - 5) < 0 ? 0 : currentBulb - 5;
271             Serial.printf(" -- END currentBulb = %d\n", newBulb);
272             lv_arc_set_value(ui_BulbArc, newBulb);
273
274             char BulbText[8];
275             if (newBulb == 100) {
276               snprintf(BulbText, sizeof(BulbText), "%d%%", newBulb);
277               lv_label_set_text(ui_BulbNum, BulbText);
278             } else {
279               snprintf(BulbText, sizeof(BulbText), " %d%%", newBulb);
280               lv_label_set_text(ui_BulbNum, BulbText);
281             }
282             // Update LED brightness on pin 43
283             int pwm_value = (newBulb * 255) / 100;
284             ledcWrite(breathPwmChannel, pwm_value);
285
286           } else if (current_screen == ui_Screen4) {
```

This code is logically symmetrical to the previous section and implements the control logic for counterclockwise rotation. The core is "perform the operation of reducing the value of the corresponding control element based on the current screen".

When it is determined that the encoder is rotating counterclockwise (position_tmp = 0), a branch judgment is made based on the current screen: when it is screen 2, the volume value is reduced by 5 each time, with the lower limit set to 0 (to avoid negative values); when it is screen 3, the bulb brightness value is reduced by 5, and

the PWM output of the bulb is updated simultaneously; when it is screen 4, the screen brightness value is reduced by 5, and the PWM output of the screen backlight is updated simultaneously.

```
312         Serial.print("Direction: ");
313         Serial.print(currentDir);
314         Serial.print(" | Counter: ");
315         Serial.println(counter);
316         last_counter = counter;
317
318         processEncoder();
319
320       }
321
322       // Remember last CLK state
323       lastStateCLK = currentStateCLK;
```

This code is the "cleanup operation" after the rotation action processing is completed, and it is also the key to ensuring the continuous and stable operation of the encoder.

Firstly, the rotation direction (currentDir) and the rotation steps (counter) are printed through the serial port, which is a common debugging method in embedded development, allowing developers to observe the working status of the encoder in real time and troubleshoot issues such as incorrect rotation direction and numerical jumps;

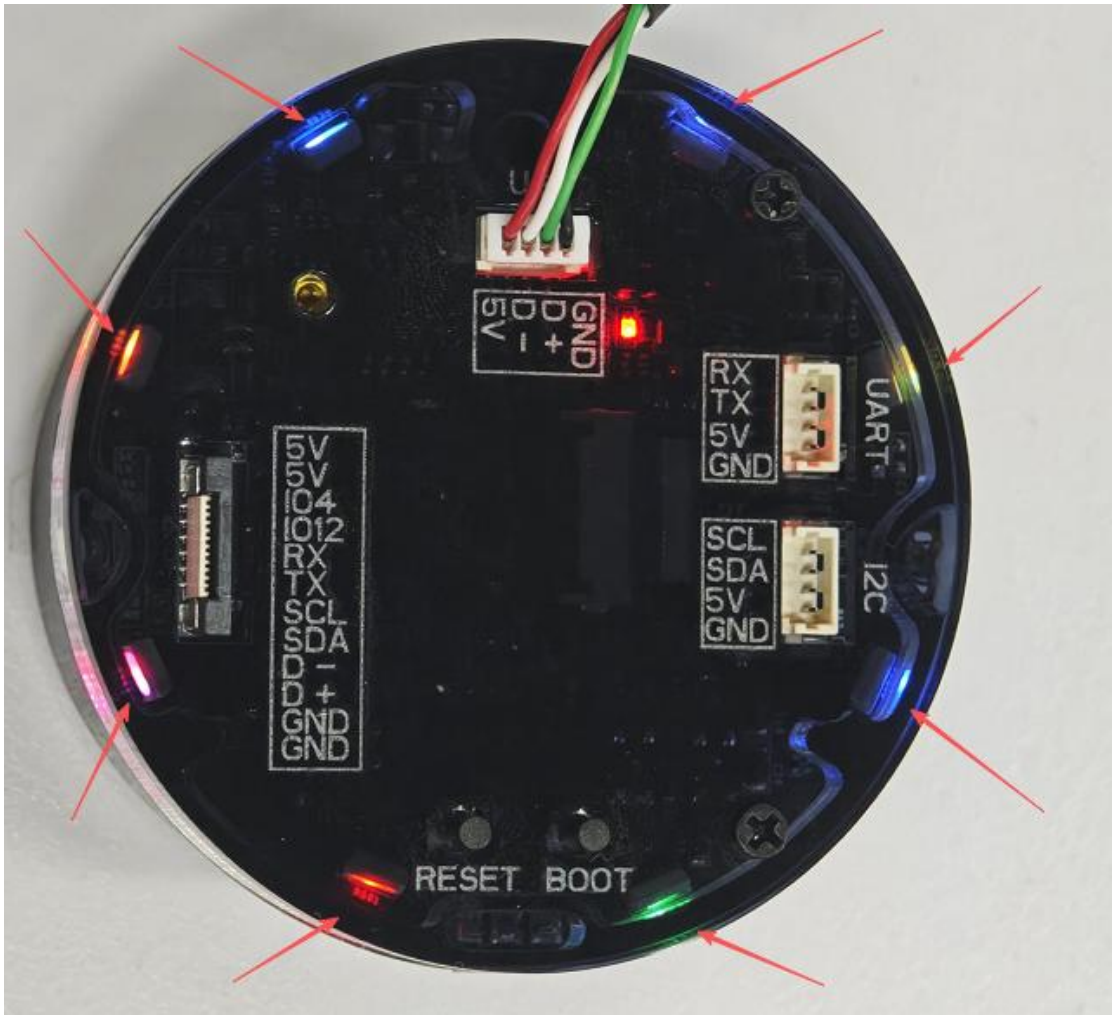
then, the current counter value is saved in last_counter to be used for subsequent anti-misoperation judgment (to avoid abnormal numerical values caused by rapid rotation);

then, the processEncoder() function is called, which is the core function of this function to handle the menu switching logic of screen 1 (main menu) - only when the current screen is screen 1, will the rotation encoder trigger the menu switching (switching volume, bulb, and brightness options), this step realizes the "distinguishing of different screen rotation functions", reflecting the "layered logic" in embedded development;

finally, execute lastStateCLK = currentStateCLK, this is the most crucial step, saving the current read A-phase level as the previous state, preparing for the edge detection in the next loop, if this line is not written, in the subsequent steps, it will be impossible to determine whether A-phase has jumped, and the encoder will not be able to work normally.

8 LED flashing tasks

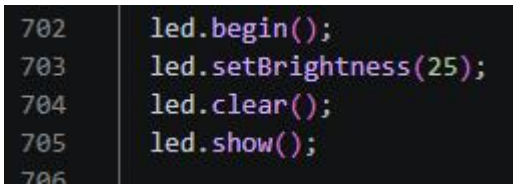
```
RotaryScreen_1_46_Code.ino 2    RotaryScreen_1_46.h 5
RotaryScreen_1_46.h > ENCODER_A_PIN
 1   #ifndef _ROTARYSCREEN_1_46_H
21   /*   LED Light Define   */
22   #define POWER_LIGHT_PIN 40
23   #define LED_PIN         48
24   #define LED_NUM         8
25
```



This section of code serves as the fundamental configuration for the entire LED hardware system. It is located in the header file and is used to centrally manage the hardware pins and parameters, enhancing the maintainability and portability of the code. Among them, POWER_LIGHT_PIN 40 defines the pin for the system power indicator light, which is used to indicate whether the device is properly powered on and working. LED_PIN 48 defines the data control pin for the external WS2812 series programmable color lights. All 8 colors, brightness, and switch states of the lights are all controlled through this single pin via serial communication. LED_NUM 8 clearly

states that the number of LEDs in the light string is 8, providing accurate hardware parameters for the subsequent initialization of the Adafruit_NeoPixel library.

This part of the code declares all the global variables and driver objects required for controlling the LED color lights. Firstly, an instance of the Adafruit_NeoPixel class named led is created, and the number of LEDs, data pin, and communication timing format (NEO_GRB + NEO_KHZ800) are passed in to complete the low-level binding of the color light driver library. The global boolean variable isLed serves as the overall switch for the LED lighting effect, with an initial value of true indicating that the lighting effect is enabled by default upon power-on. It controls the running and suspension of the entire LED task. ledCount is an unsigned character counter used to record the number of cycles for various lighting effects, ensuring that each animation is executed according to the set number of times. ledBrightness is used in the breathing light effect to store the current brightness level, enabling smooth gradient control of the brightness. These global variables are declared in the header file and can be accessed by the ledTestTask task, the main function, and other functional functions, achieving global sharing of LED status throughout the system. This is a standard variable design method for peripheral control in multi-task environments.



This part of the code is located in the "setup()" function and serves as the initialization entry point for the LED hardware and the driver library after the system powers on. The execution sequence strictly follows the embedded hardware startup logic of "power supply first, then initialization, and finally clearing the status".

Firstly, pin 17 is configured as an output mode and a high level is output. This pin is the power enable pin for the LED light strip, used to provide the working power for the entire light strip module, ensuring that the hardware is ready upon power-on.

Then, the "led.begin()" function is called to initialize the Adafruit_NeoPixel driver library, completing the handshake with the hardware light strip, timing configuration,

and pin binding, enabling the MCU to control the light strip. Next, "led.setBrightness(25)" is used to set the global default brightness to 25 (maximum value 255), avoiding overly bright initial lighting that could cause glare or excessive power consumption.

Then, "led.clear()" is used to clear the color cache of all light beads, setting all light beads to an off state. Finally, "led.show()" is called to refresh the cleared state to the actual hardware light beads, ensuring that the initial power-on state has no stray light or garbled bright lights. The entire initialization process is rigorous and standardized, ensuring that the LED system is in a stable, clean, and controllable initial state before entering the animation task, laying a hardware foundation for the normal operation of the subsequent "ledTestTask" task.

```
710    xTaskCreatePinnedToCore(ledTestTask, "LED Test", 2048, NULL, 1, &ledTestTaskHandle, 0);
```

This sentence is in the FreeRTOS system of ESP32, where a dedicated real-time task is created specifically for the color light effect. The entry function of the task is ledTestTask, and the task is named "LED Test" for easy debugging and viewing. A 2048-byte task stack is allocated, which is sufficient to accommodate the color light logic, local variables, and function call overhead. There are no entry parameters, the task priority is set to 1, and the task handle is stored in ledTestTaskHandle for later suspension and restoration. It is also fixedly bound to CPU core 0 for operation. The color light logic is made into a separate RTOS task. The main advantage is that it does not block the LVGL interface, does not block the encoder detection, and does not occupy the loop main thread. The color light animation can run independently in a loop, and it does not interfere with screen operations or knob operations.

```
487  void ledTestTask(void *pvParameters) {
488    while (1) {
489      led.clear();
490      led.show();
```

```
596      //****************led stop***************
597      if (!isLed) {
598        ledCount = 0;
599        ledBrightness = 0;
600        led.setBrightness(25);
601        led.clear();
602        led.show();
603        vTaskSuspend(NULL);
604      }
605
606      vTaskDelay(pdMS_TO_TICKS(5));
607    }
608  }
```

The ledTestTask follows the standard format of FreeRTOS tasks. Internally, it continuously runs the color light animation logic in an infinite loop to ensure that the preset lighting effects are automatically played in a loop after power-on. At the beginning of each cycle, all the light bulbs are cleared and the hardware display is refreshed to ensure that the residual lights from the previous cycle do not affect the rendering of the next animation.

The entire task encapsulates five sets of fixed lighting modes: white sequential waterfall light, full-color synchronous flash, color sequential waterfall, multi-color synchronous slow flash, and global color breathing gradient. These modes are executed in sequence to form a complete lighting effect process.

Throughout the process, a global variable is used as the overall switch: as long as isLed is false, the light count and brightness parameters are immediately reset, the light bulb display is cleared, and vTaskSuspend(NULL) is called to suspend the current task, thus not occupying the CPU until it is awakened by an external signal.

This self-suspension writing method is very classic and can achieve task sleep without external handles; finally, a 5ms task delay is added to give the CPU to other tasks, ensuring that the lighting animation is smooth without lag, and does not monopolize system resources, balancing real-time performance and system stability.

```
491     //Five circles of white flowing water lights
492     while ((isLed) && (ledCount++ < 5)) {
493       for (int i = 0; isLed && i < 8; i++) {
494         led.setPixelColor(i, led.Color(255, 255, 255));
495         led.show();
496         vTaskDelay(pdMS_TO_TICKS(250));
497         led.clear();
498         led.show();
499       }
500     }
501     ledCount = 0;
```

This section achieves a five-round pure white sequential flashing effect for each individual light bulb. The outer while loop controls the overall loop to run only 5 times. The inner for loop iterates through 8 light bulbs, setting the maximum RGB value of pure white for each bulb. The led.show() function is called immediately to refresh the hardware and light up the current bulb. A delay of 250ms is added to maintain visual persistence.

Then, the light bulbs are cleared and refreshed to form a visual effect where each bulb flows sequentially. The loop condition includes an isLed check throughout to ensure that if the light switch is turned off halfway through, the logic for the light effect can immediately exit, and no invalid execution will continue. After completing 5 rounds, the count variable ledCount is automatically reset to prepare for the next set of light effects.

```
503     //All the lights flash in rapid succession in various colors simultaneously.
504     for (int i = 0; isLed && i < 5; i++) {
505       led.setPixelColor(0, led.Color(255, 0, 0));
506       led.setPixelColor(1, led.Color(0, 255, 0));
507       led.setPixelColor(2, led.Color(0, 0, 255));
508       led.setPixelColor(3, led.Color(255, 255, 0));
509       led.setPixelColor(4, led.Color(130, 0, 255));
510       led.setPixelColor(5, led.Color(0, 130, 255));
511       led.setPixelColor(6, led.Color(255, 130, 0));
512       led.setPixelColor(7, led.Color(255, 0, 130));
513
514       led.show();
515       vTaskDelay(pdMS_TO_TICKS(100));
516       led.clear();
517       led.show();
518       vTaskDelay(pdMS_TO_TICKS(100));
519     }
```

This code implements a five-round full-lamp color synchronization flash, repeating the complete flashing process 5 times. In each round, 8 light beads are respectively assigned different fixed colors such as red, green, blue, yellow, and purple. After all the values are assigned once, the led.show() function is called simultaneously to light up all the light beads, staying for 100ms to create a bright visual effect.

Then, the light beads are cleared and refreshed, and a delay of 100ms is added to form a dimming interval, creating a rapid flashing effect with one bright flash and one dimming interval. The colors of all the light beads remain fixed and unchanged, presenting a colorful and simultaneous flashing visual effect. The isLed global switch judgment is also nested to support the interruption of the light effect at any time. The structure is simple and the timing control is regular, making it the most commonly used synchronous flashing code for Neopixel color lights.

```
521     //Colorful flowing lights in 5 circles
522     while (isLed && ledCount < 5) {
523       for (int i = 0; isLed && i < 8; i++) {
524         led.clear();
525         switch (i) {
526           case 0: led.setPixelColor(i, led.Color(255, 0, 0));   break;
527           case 1: led.setPixelColor(i, led.Color(0, 255, 0));   break;
528           case 2: led.setPixelColor(i, led.Color(0, 0, 255));   break;
529           case 3: led.setPixelColor(i, led.Color(255, 255, 0)); break;
530           case 4: led.setPixelColor(i, led.Color(130, 0, 255)); break;
531           case 5: led.setPixelColor(i, led.Color(0, 130, 255)); break;
532           case 6: led.setPixelColor(i, led.Color(255, 130, 0)); break;
533           case 7: led.setPixelColor(i, led.Color(255, 0, 130)); break;
534         }
535         led.show();
536         vTaskDelay(pdMS_TO_TICKS(250));
537       }
538       ledCount++;
539     }
540     ledCount = 0;
```

This section is a five-round color sequential sequential light display. The outer layer is limited to running 5 rounds of animation. When the inner layer traverses 8 light bulbs, it first clears the entire screen of lights, then through the switch branch, assigns a unique fixed single color to each numbered light bulb, sequentially lights them up, delays for a period of time, and forms a color flowing effect that moves sequentially;

it is consistent with the logic structure of the white flowing effect, but instead of using a fixed white color, each light is independently assigned a color. The visual hierarchy is more rich. After one round, it automatically counts and accumulates, after five rounds, the counting variable is reset, ensuring that the next section of the light effect state is clean and without conflicts. The modular writing method allows each set of light effects to be independent and closed-loop, and not interfere with each other.

```
542     //All the lights flash in a slow, colored pattern simultaneously.
543     for (int i = 0; isLed && i < 5; i++) {
544       led.setPixelColor(0, led.Color(255, 0, 0));
545       led.setPixelColor(1, led.Color(0, 255, 0));
546       led.setPixelColor(2, led.Color(0, 0, 255));
547       led.setPixelColor(3, led.Color(255, 255, 0));
548       led.setPixelColor(4, led.Color(130, 0, 255));
549       led.setPixelColor(5, led.Color(0, 130, 255));
550       led.setPixelColor(6, led.Color(255, 130, 0));
551       led.setPixelColor(7, led.Color(255, 0, 130));
552       led.show();
553       vTaskDelay(pdMS_TO_TICKS(250));
554       led.clear();
555       led.show();
556       vTaskDelay(pdMS_TO_TICKS(250));
557     }
```

This is exactly the same as the previous colorful flash logic, except that the delay between on-off states has been extended to 250ms, achieving slow and synchronized colorful flashing. By increasing the delay time, the animation rhythm can be changed.

Without modifying the color configuration and lighting logic, simply adjusting the timing can switch between fast and slow effects. This reflects the design concept of decoupling code timing from business logic; also, it is limited to looping 5 times only, maintaining the regularity of the entire lighting effect duration, and adapting to the rhythm design of the entire machine's UI lighting.

```
559     //Colorful breathing light, breathing 5 times
560     led.setPixelColor(0, led.Color(255, 0, 0));
561     led.setPixelColor(1, led.Color(0, 255, 0));
562     led.setPixelColor(2, led.Color(0, 0, 255));
563     led.setPixelColor(3, led.Color(255, 255, 0));
564     led.setPixelColor(4, led.Color(130, 0, 255));
565     led.setPixelColor(5, led.Color(0, 130, 255));
566     led.setPixelColor(6, led.Color(255, 130, 0));
567     led.setPixelColor(7, led.Color(255, 0, 130));
568     while ((isLed) && (ledCount++ < 10)) {
569       for (ledBrightness = 0; isLed && ledBrightness <= 25; ledBrightness++) {
570         led.setBrightness(ledBrightness);
571         led.setPixelColor(0, led.Color(255, 0, 0));
572         led.setPixelColor(1, led.Color(0, 255, 0));
573         led.setPixelColor(2, led.Color(0, 0, 255));
574         led.setPixelColor(3, led.Color(255, 255, 0));
575         led.setPixelColor(4, led.Color(130, 0, 255));
576         led.setPixelColor(5, led.Color(0, 130, 255));
577         led.setPixelColor(6, led.Color(255, 130, 0));
578         led.setPixelColor(7, led.Color(255, 0, 130));
579         led.show();
580         vTaskDelay(pdMS_TO_TICKS(50));
581       }
582       for (; isLed && ledBrightness >= 0; ledBrightness--) {
583         led.setBrightness(ledBrightness);
584         led.show();
585         vTaskDelay(pdMS_TO_TICKS(50));
586       }
587       ledCount++;
588     }
```

This is the most complex global color breathing light in the entire lighting effect set.

First, the colors of the 8 light beads are fixed, without changing the hue, only adjusting the overall brightness; through two layers of for loops, the brightness gradually increases from 0 to 25, then decreases from 25 to 0. Each level of brightness is re-set and refreshed for display.

With a 50ms small delay, the gradual transition is very smooth, creating a natural breathing effect of light and darkness; the outer layer limits the breathing cycle to run only 10 times.

After that, the count and brightness parameters are reset, the default brightness is restored, and the light bead images are cleared, returning to the initial state.

The entire process relies on the global brightness control interface of the Neopixel library. Without modifying the RGB values one by one, the overall breathing effect can be achieved. The code is concise and the animation effect is exquisite.

At the same time, the isLed switch judgment is retained, allowing the lighting effect to be turned off at any time.

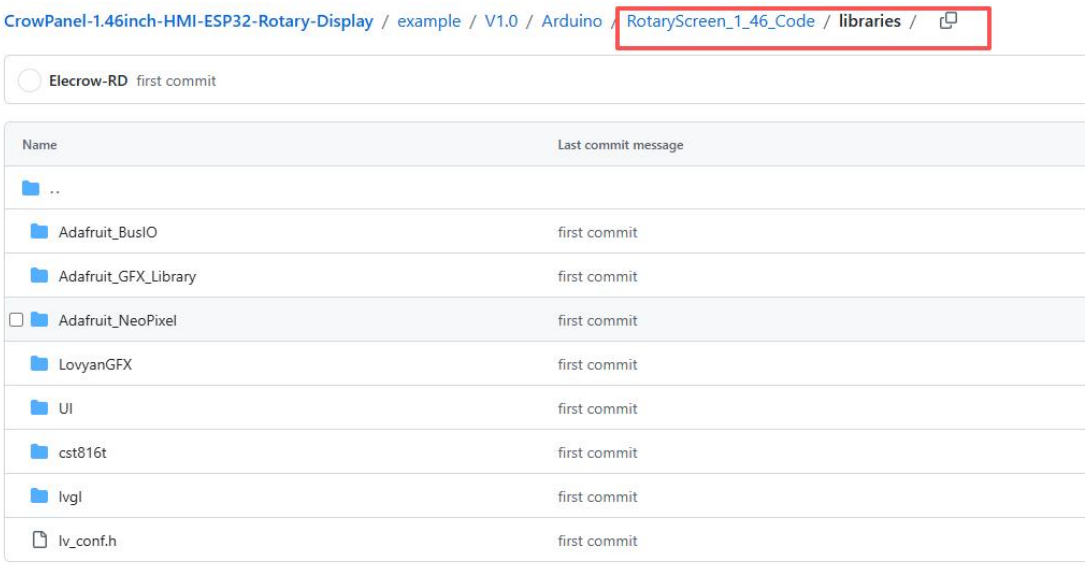
At this point, all the code functions covered in this course have been explained.

Library file path

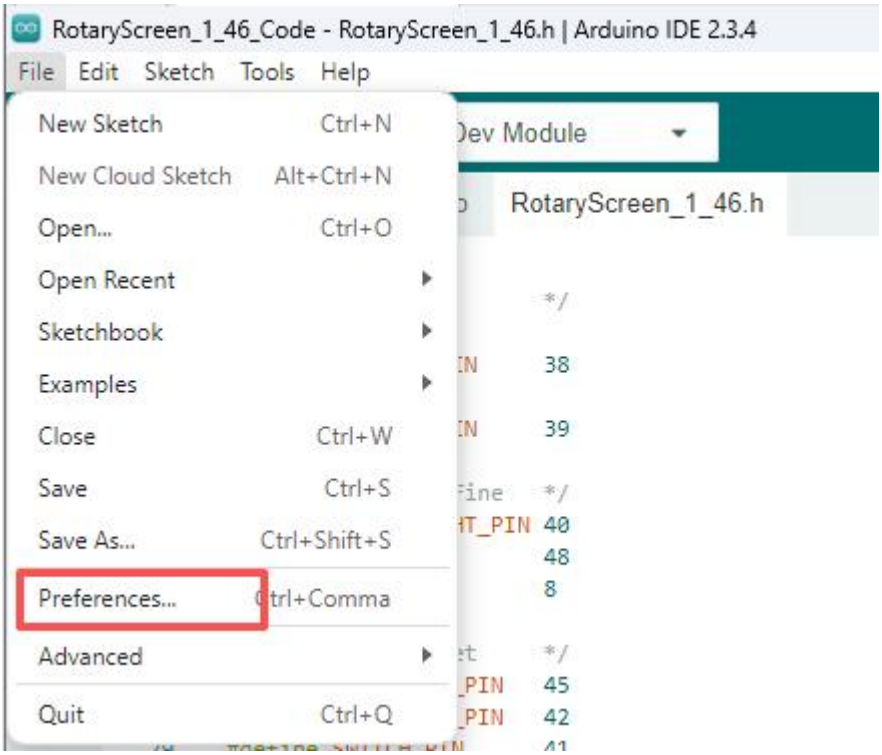
In order for the code to run properly, we also need to download the library files we provided.

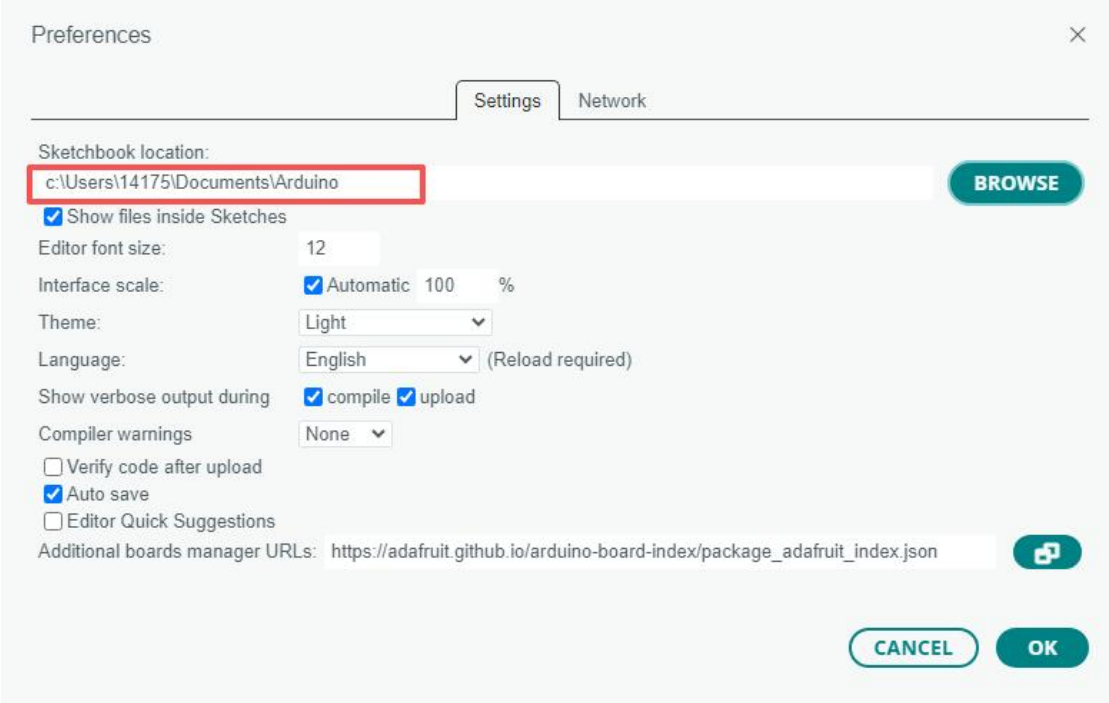
Library file download link:

https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/tree/master/example/V1.0/Arduino/RotaryScreen_1_46_Code

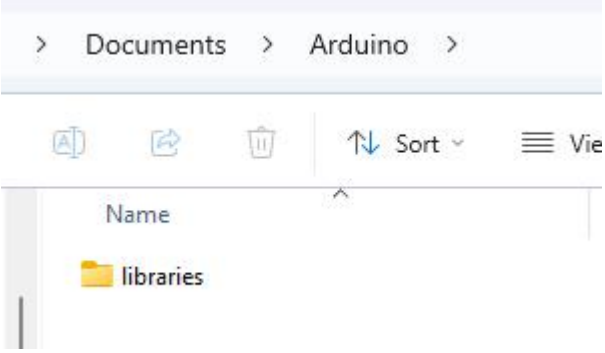


After downloading, we will place these library files in the default library file path of the Arduino IDE.

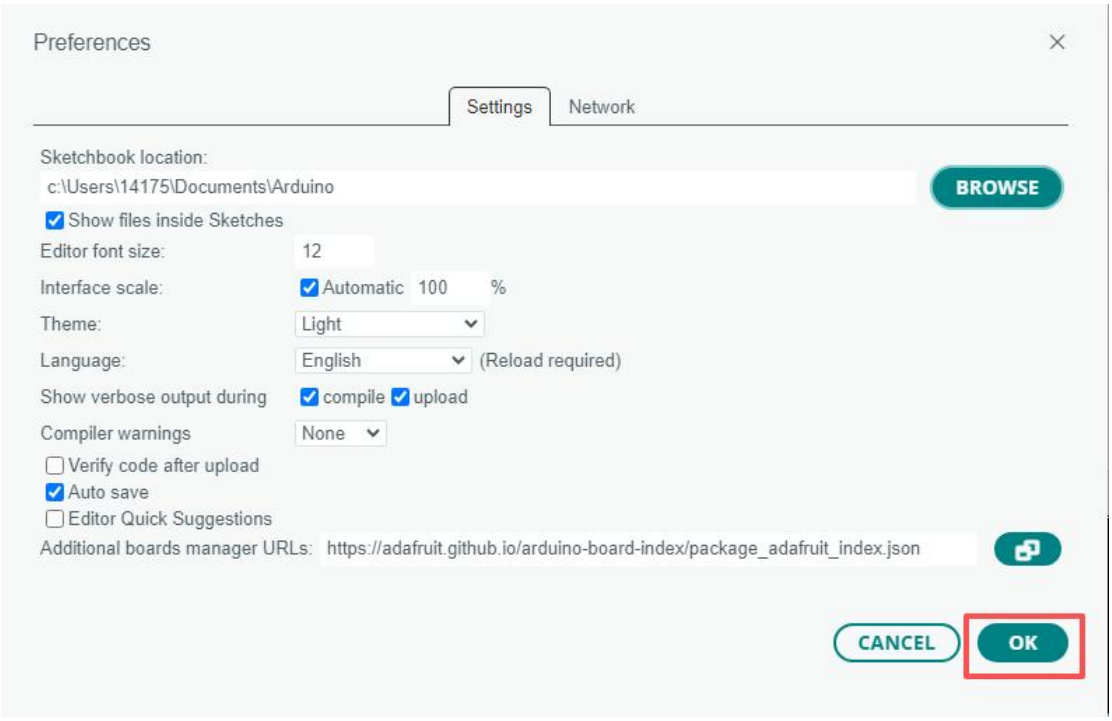




Then, place the downloaded library files into it.



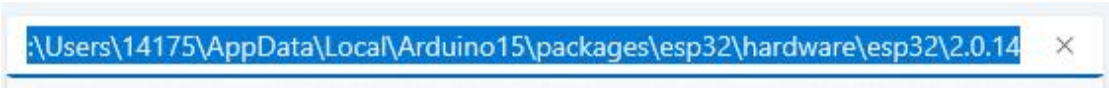
After placing it properly, remember to save it.



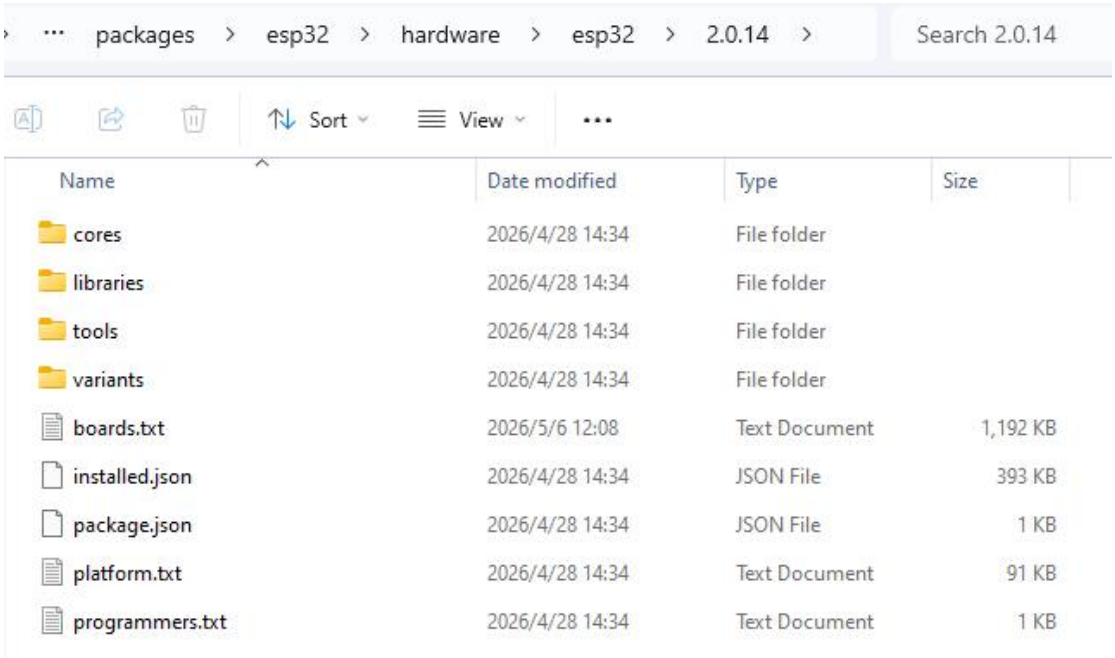
Modification of partition table

Due to the large scale of our code project, the required firmware size exceeds the limit of the default partition table. Direct compilation will result in an error and no executable file can be generated. Therefore, next we need to modify the partition table configuration to allocate sufficient storage space for the program, ensuring that the project can be compiled and run normally.

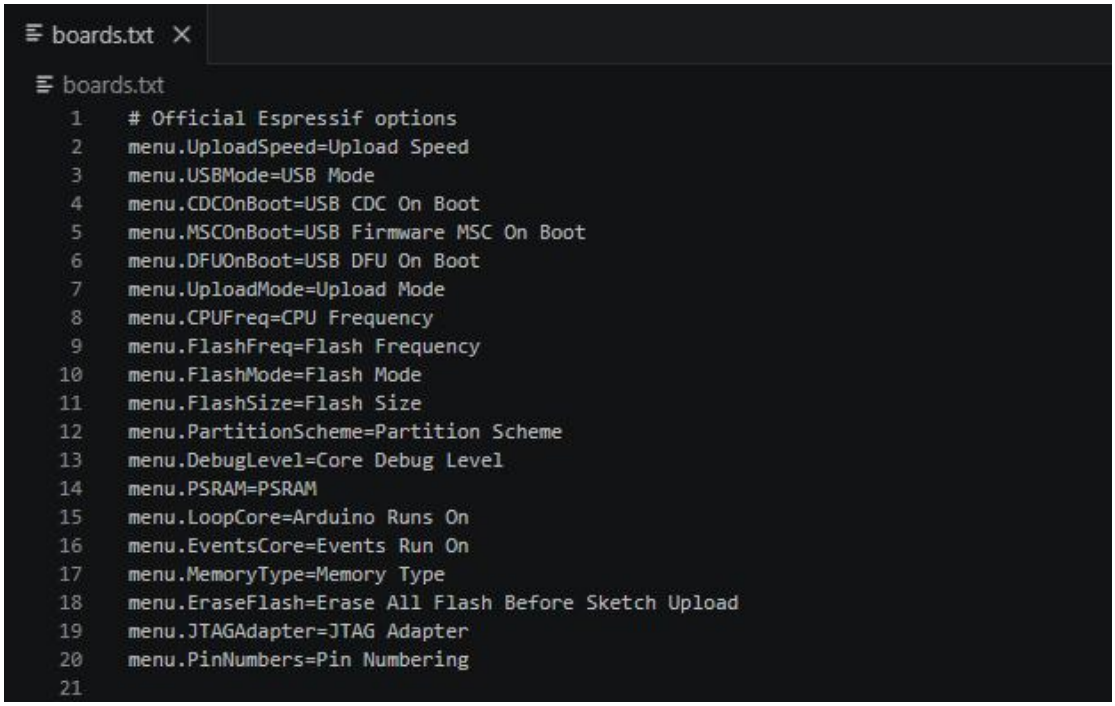
Open the file system, and we arrive at the file path of the previously installed ESP32 version 2.0.14.



Come to this interface



Open the "boards.txt" file



Press Ctrl + F on the keyboard, and search for

"esp32s3.menu.PartitionScheme.huge_app"

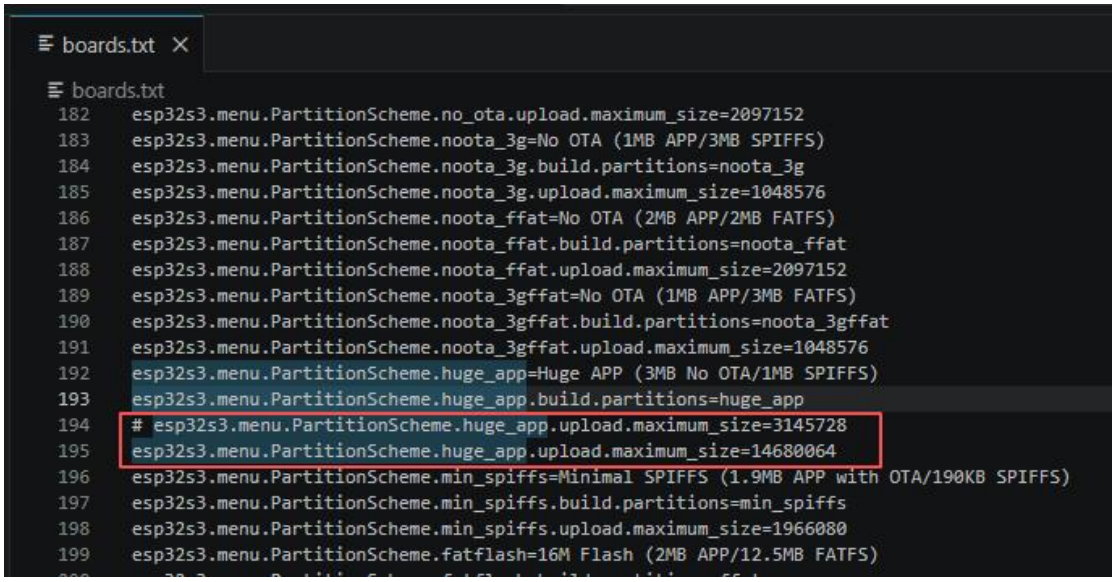
boards.txt

```
182  esp32s3.menu.PartitionScheme.no_ota.upload.maximum_size=2097152
183  esp32s3.menu.PartitionScheme.noota_3g=No OTA (1MB APP/3MB SPIFFS)
184  esp32s3.menu.PartitionScheme.noota_3g.build.partitions=noota_3g
185  esp32s3.menu.PartitionScheme.noota_3g.upload.maximum_size=1048576
186  esp32s3.menu.PartitionScheme.noota_ffat=No OTA (2MB APP/2MB FATFS)
187  esp32s3.menu.PartitionScheme.noota_ffat.build.partitions=noota_ffat
188  esp32s3.menu.PartitionScheme.noota_ffat.upload.maximum_size=2097152
189  esp32s3.menu.PartitionScheme.noota_3gffat=No OTA (1MB APP/3MB FATFS)
190  esp32s3.menu.PartitionScheme.noota_3gffat.build.partitions=noota_3gffat
191  esp32s3.menu.PartitionScheme.noota_3gffat.upload.maximum_size=1048576
192  esp32s3.menu.PartitionScheme.huge_app=Huge APP (3MB No OTA/1MB SPIFFS)
193  esp32s3.menu.PartitionScheme.huge_app.build.partitions=huge_app
194  # esp32s3.menu.PartitionScheme.huge_app.upload.maximum_size=3145728
195  esp32s3.menu.PartitionScheme.huge_app.upload.maximum_size=14680064
196  esp32s3.menu.PartitionScheme.min_spiffs=Minimal SPIFFS (1.9MB APP with OTA/190KB SPIFFS)
197  esp32s3.menu.PartitionScheme.min_spiffs.build.partitions=min_spiffs
198  esp32s3.menu.PartitionScheme.min_spiffs.upload.maximum_size=1966080
199  esp32s3.menu.PartitionScheme.fatflash=16M Flash (2MB APP/12.5MB FATFS)
```

Then open the modification file we provided.

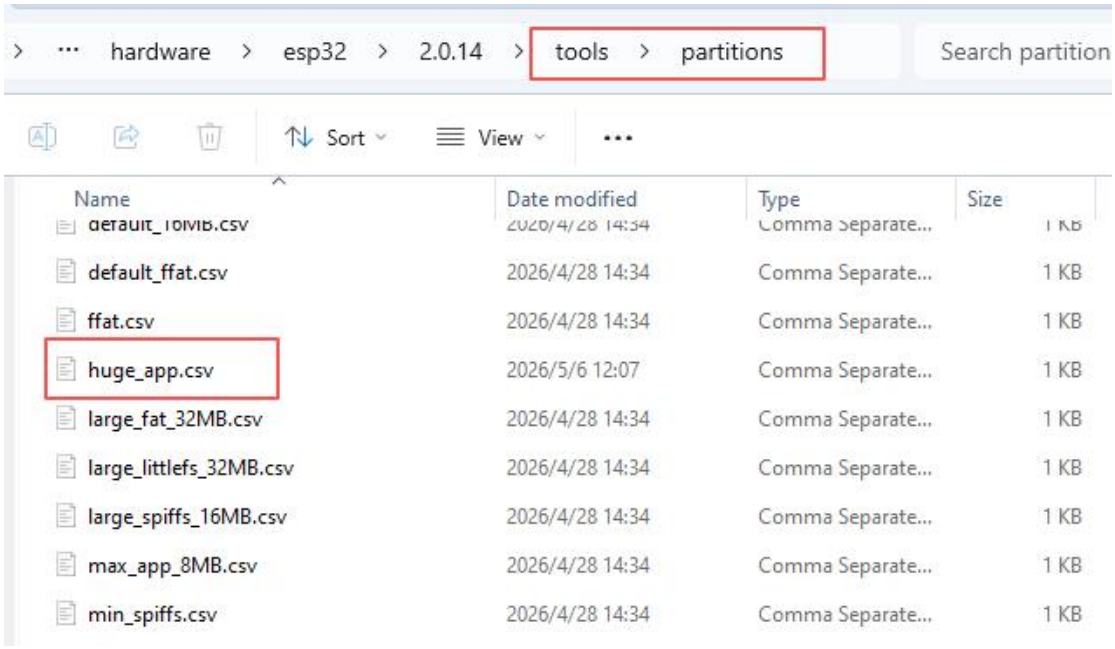
https://github.com/Elecrow-RD/CrowPanel-1.46inch-HMI-ESP32-Rotary-Display/tree/master/example/V1.0/Arduino

Copy this content here and remove the original comments marked with "#".



Save and exit.

Then proceed to the "partitions" folder and locate the "hugh_app.csv" file.



After opening, replace it with the partition table size provided by us.

CrowPanel-1.46inch-HMI-ESP32-Rotary-Display / example / V1.0 / Arduino / !!!Modify the content of the partition table.txt

Elecrow-RD first commit

Code Blame 12 lines (9 loc) · 374 Bytes

```
1   ----1: huge_app.csv
2   # Name,   Type, SubType, Offset,  Size,     Flags
3   nvs,      data, nvs,     0x9000,  0x5000,
4   otadata,  data, ota,     0xe000,  0x2000,
5   app0,     app,  ota_0,   0x10000, 0xE00000,
6   spiffs,   data, spiffs,  0xE10000,0x180000,
7   coredump, data, coredump,0xF90000,0x10000,
8
9
10
11  ----2: boards.txt
12  esp32s3.menu.PartitionScheme.huge_app.upload.maximum_size=14680064
```

huge_app.csv

C: > Users > 14175 > AppData > Local > Arduino15 > packages > esp32 > hardware > esp32 > 2.0.14 > tools > partitions > huge_app.csv

```
1   # Name,   Type, SubType, Offset,  Size,     Flags
2   nvs,      data, nvs,     0x9000,  0x5000,
3   otadata,  data, ota,     0xe000,  0x2000,
4   app0,     app,  ota_0,   0x10000, 0xE00000,
5   spiffs,   data, spiffs,  0xE10000,0x180000,
6   coredump, data, coredump,0xF90000,0x10000,
```

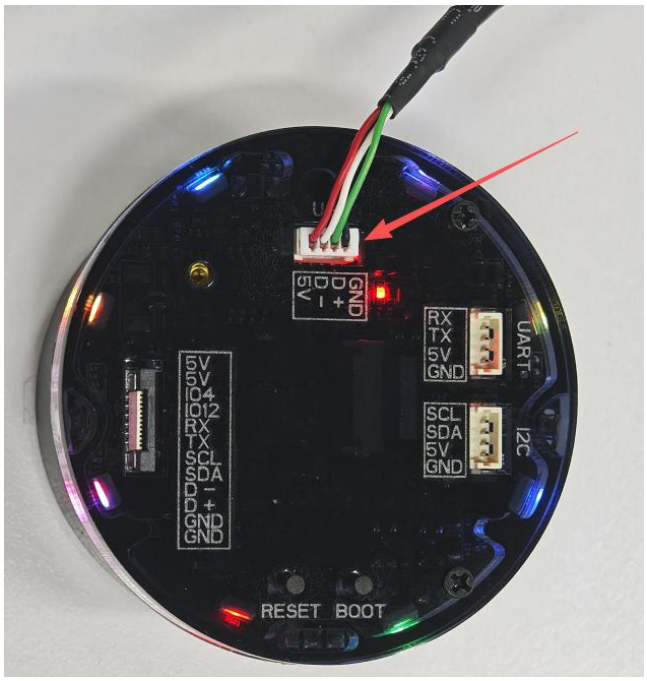
After the replacement is completed, save and exit.

Then close all the Arduino IDE programs you have opened and reopen them.

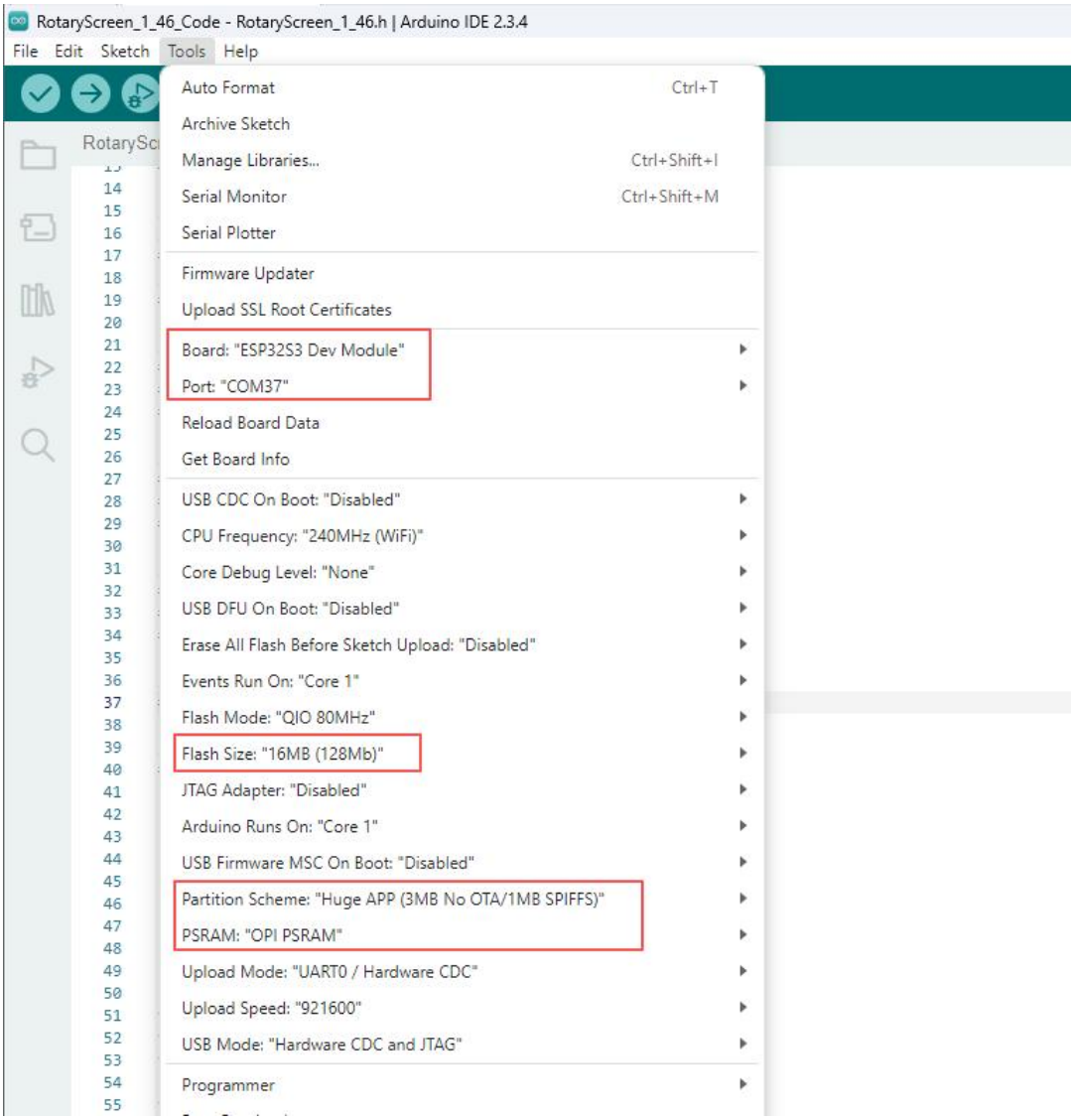
Upload the code

Now we can proceed with the code upload.

First, connect the USB interface using the data cable provided by us.



Then, based on our upload environment, make the selection.



Here, we select the partition table as the one we just modified, namely Huge_APP.

This way, your code can be compiled and burned successfully.

Finally, click on "Upload" and just wait for the code upload to complete.