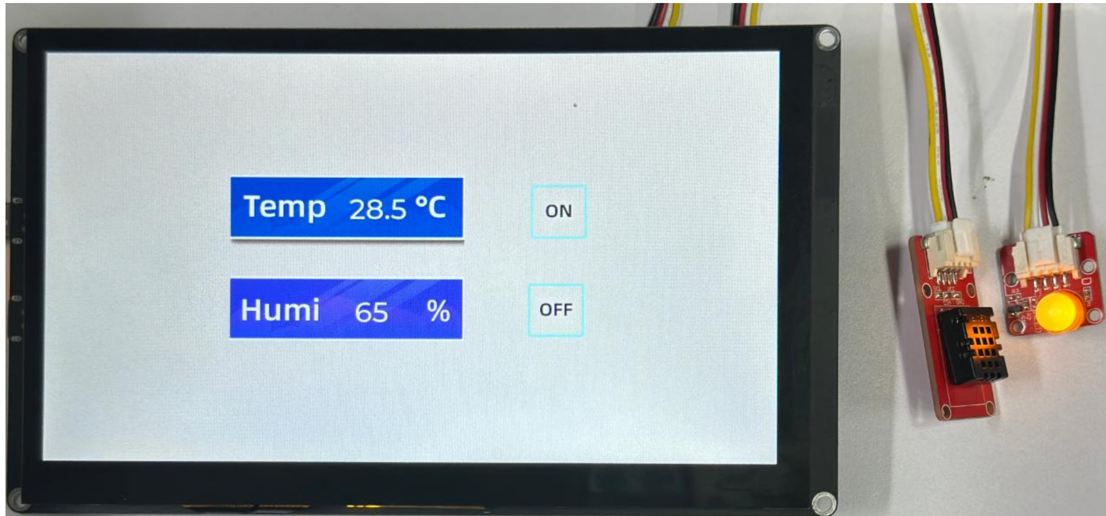
In this tutorial, we will show you how to design a user interface using SquareLine Studio and demonstrate how to upload code through Thonny IDE.
This case study demonstrates how sensors acquire ambient temperature and humidity values and display them on a screen, as well as how buttons on the screen control the on/off state of LED lights.

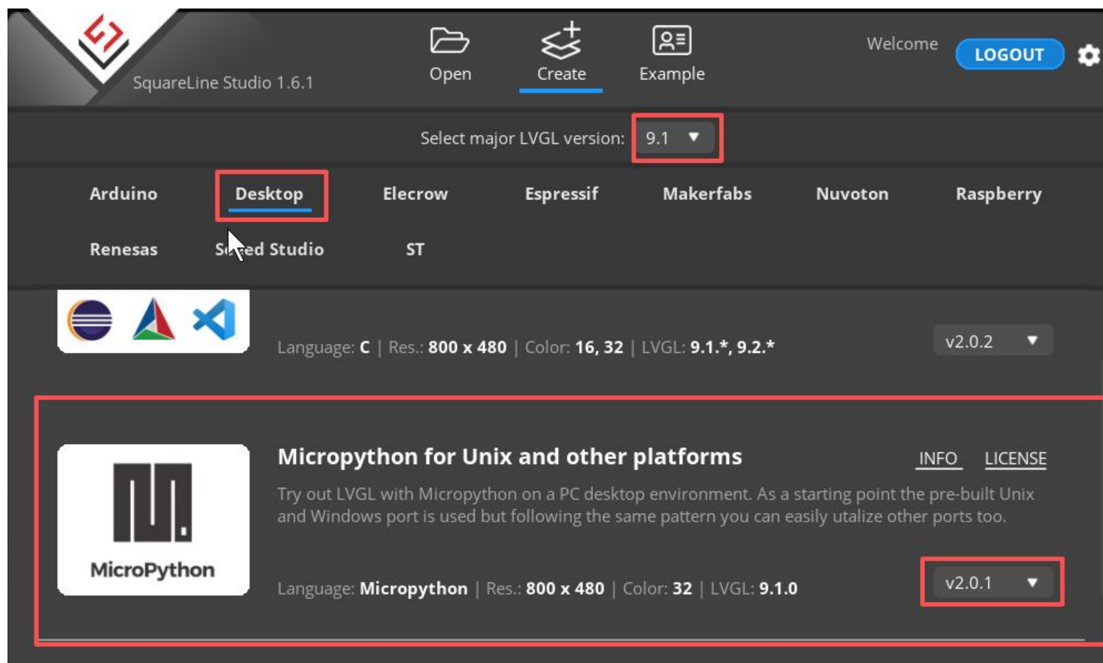


Design UI file with SquareLine Studio

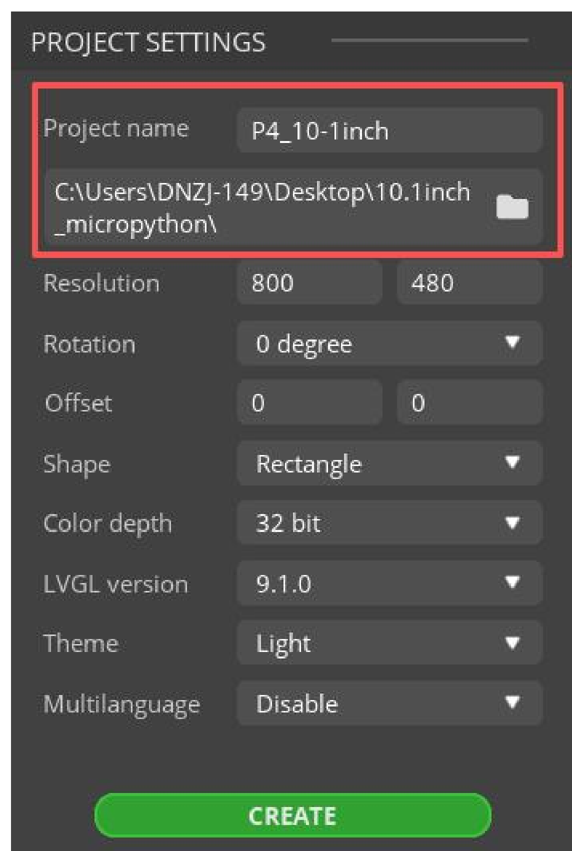
Click the link below to learn how to install SquareLine Studio software and how to export the UI files you need.
https://www.elecrow.com/wiki/Get_Started_with_SquareLine_Studio.html

After gaining a basic understanding of SquareLine Studio, let's learn how to create the interface we'll need for this lesson and how to export the UI file.

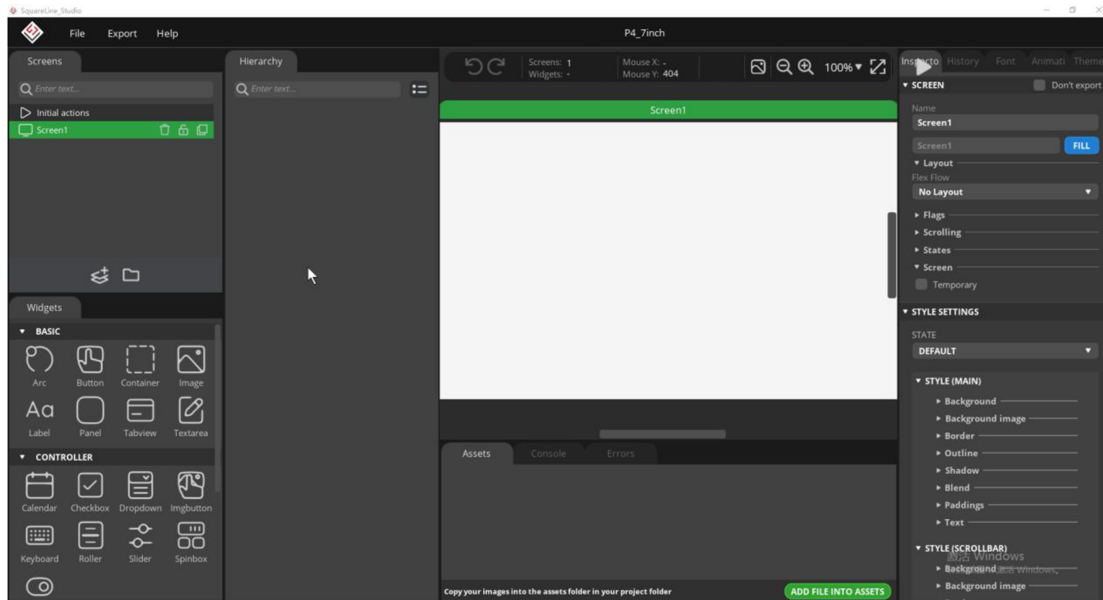
1. Open SquareLine Studio and create a project. Note that we choose version 9.1 and find the corresponding project that supports Micropython.



2. Set the project name and save path; keep other settings as default. After setting, click the "CREATE" button to create the project.

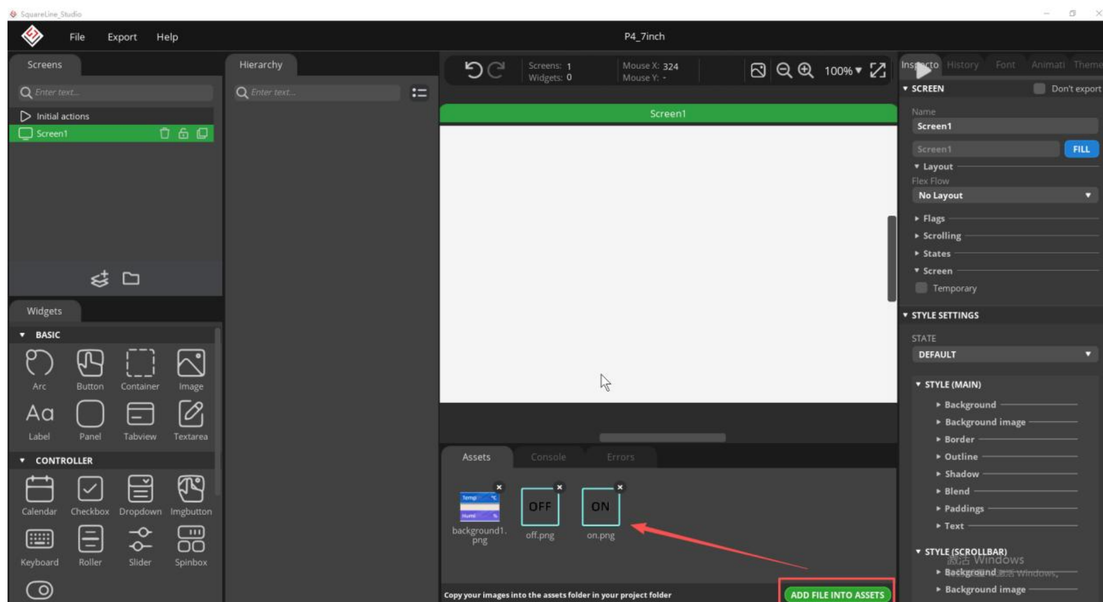


3. After creation, you will enter a blank interface.



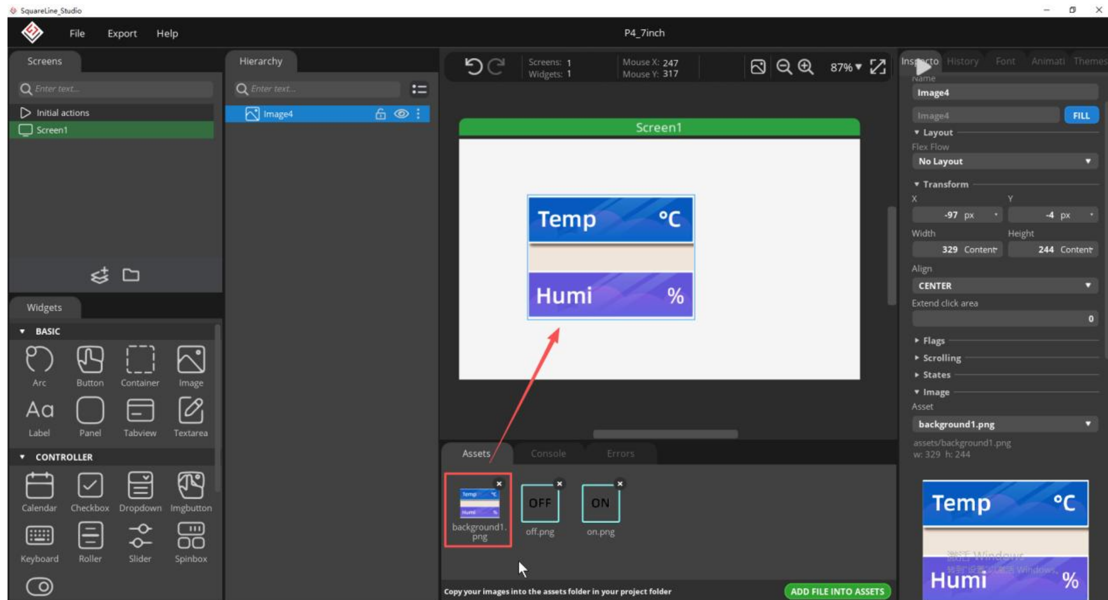
4. In the "Assets" area, click the "ADD FILE INTO ASSETS" button to add a custom image or icon. The materials used in this example can be downloaded from the following link:

https://www.elecrow.com/download/product/CrowPanel/ESP32-HMI/5.0-DIS07050H/MicroPython/assets.zip



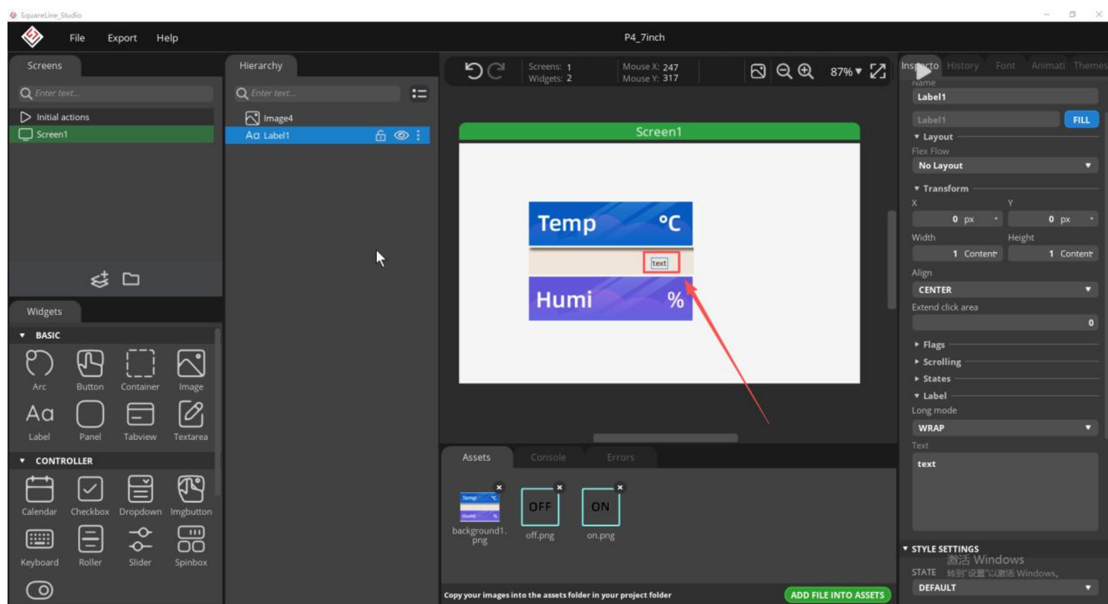
5. Add materials

Drag the material to the window and adjust its position.

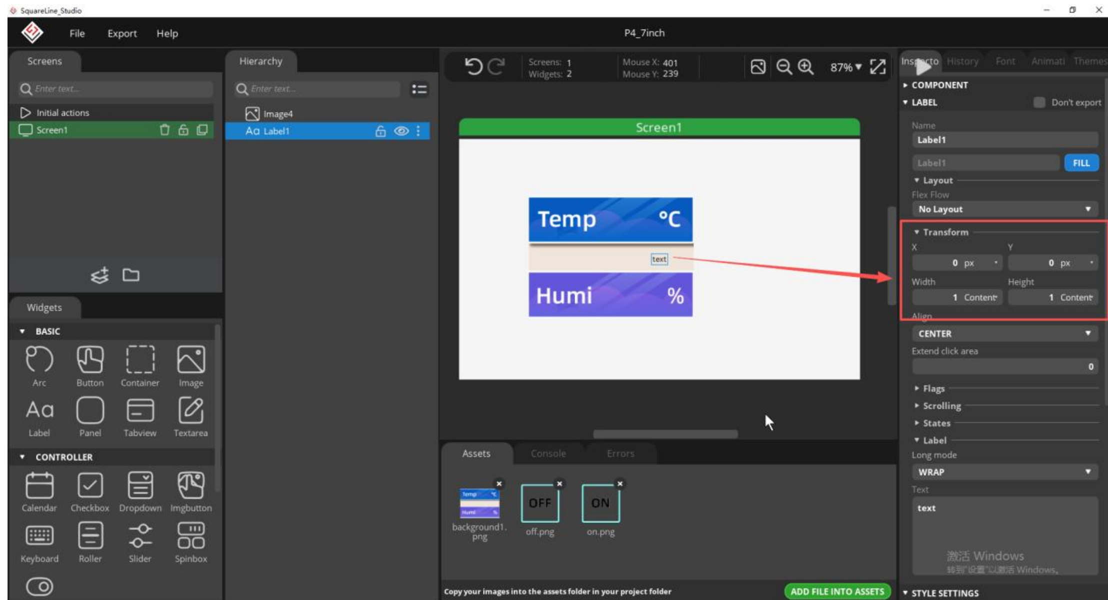


6. Add a label widget and display temperature and humidity values.

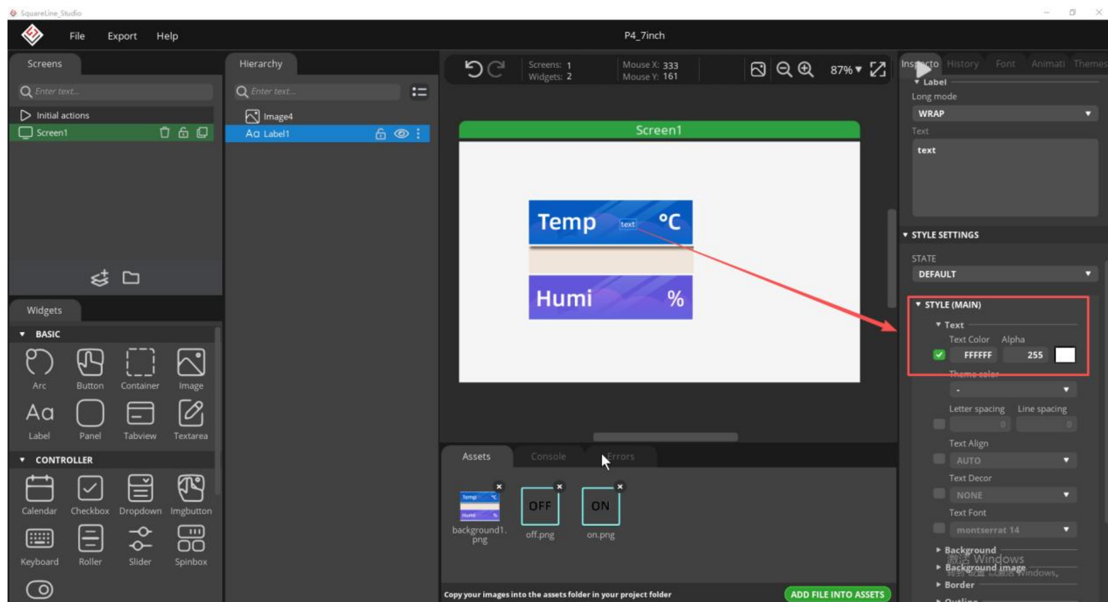
Click "Label" in the "Widgets" area to add a label to the current screen.



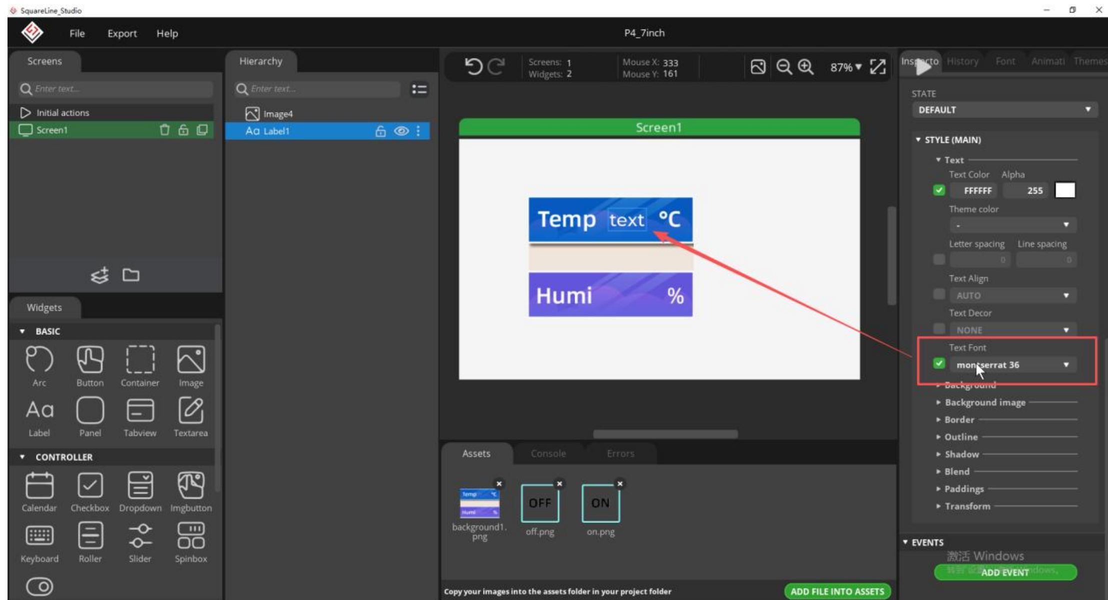
You can adjust the label position by dragging the mouse. You can also directly enter values in "Transform" to adjust it.



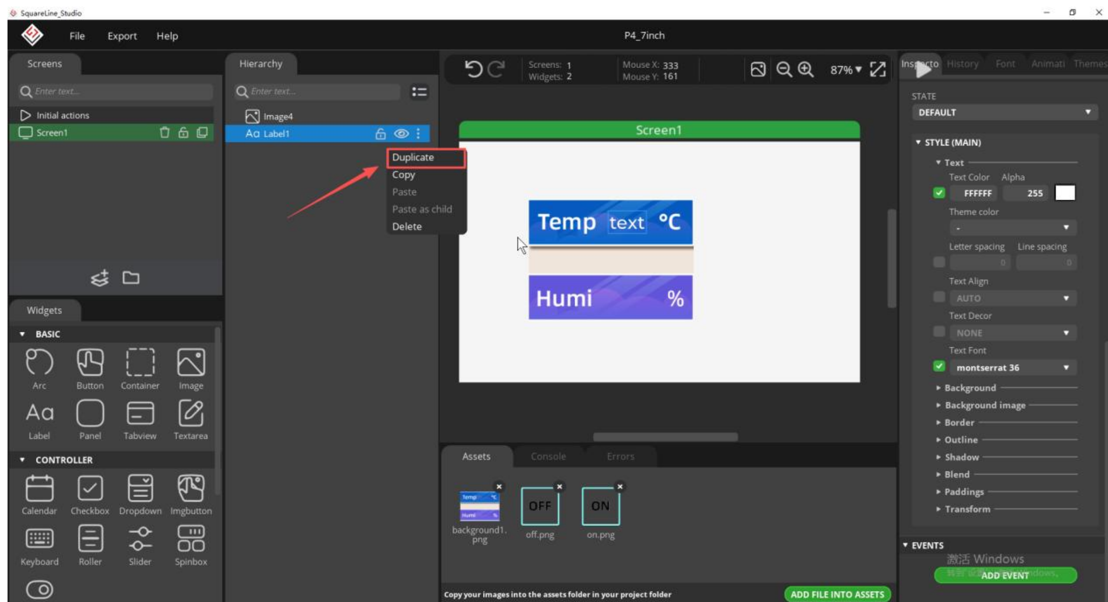
You can find "Text Color" in "STYLE (MAIN)" to adjust the font color.



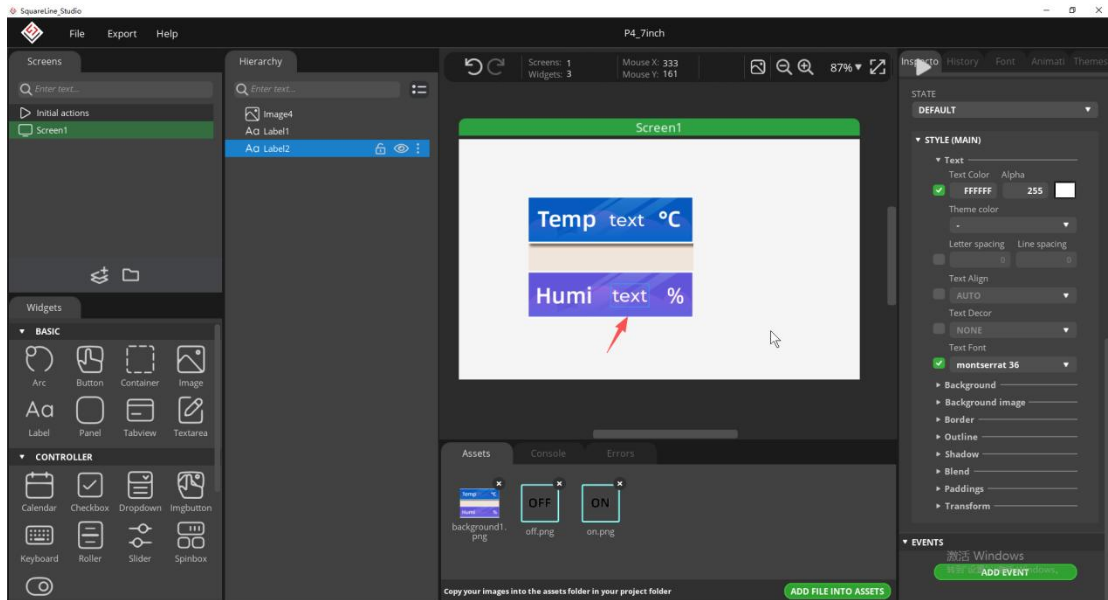
You can find "Text Font" in "STYLE (MAIN)" to adjust the font size.



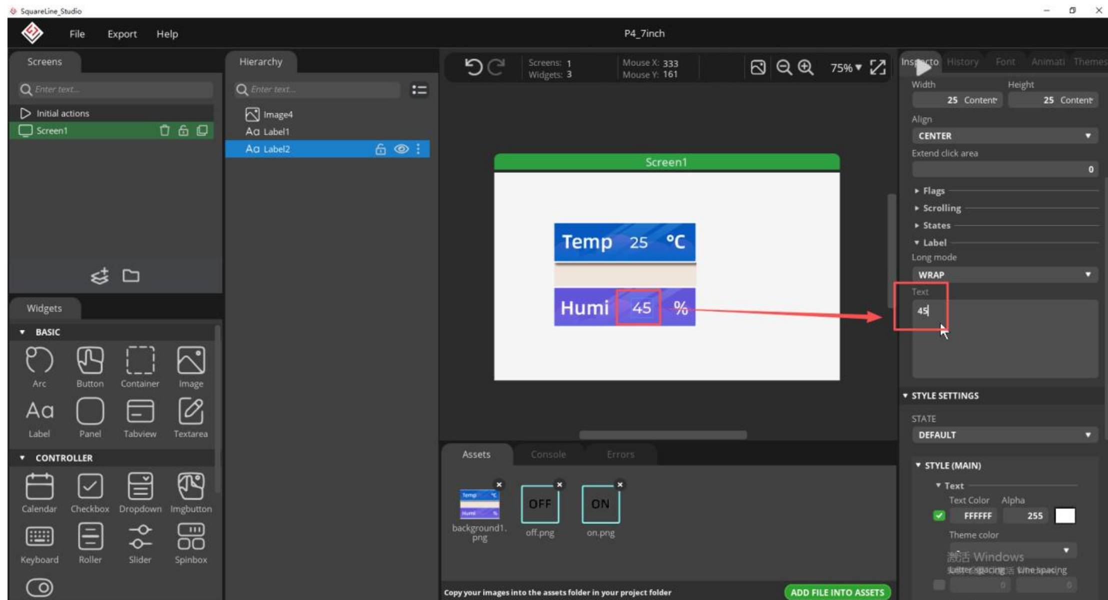
Copy Label2 to display humidity value



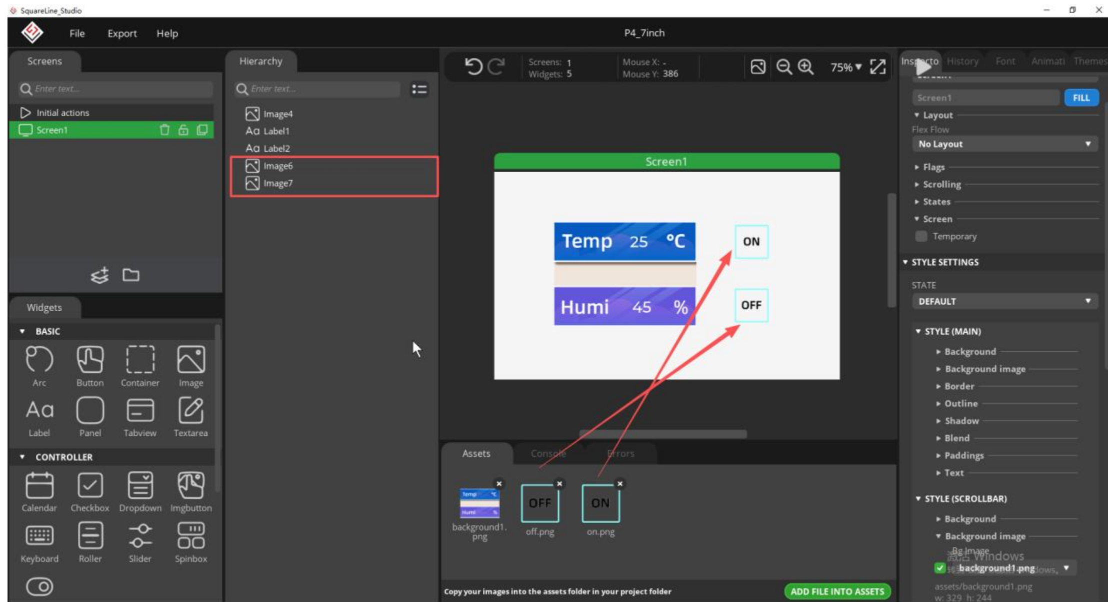
Adjust position

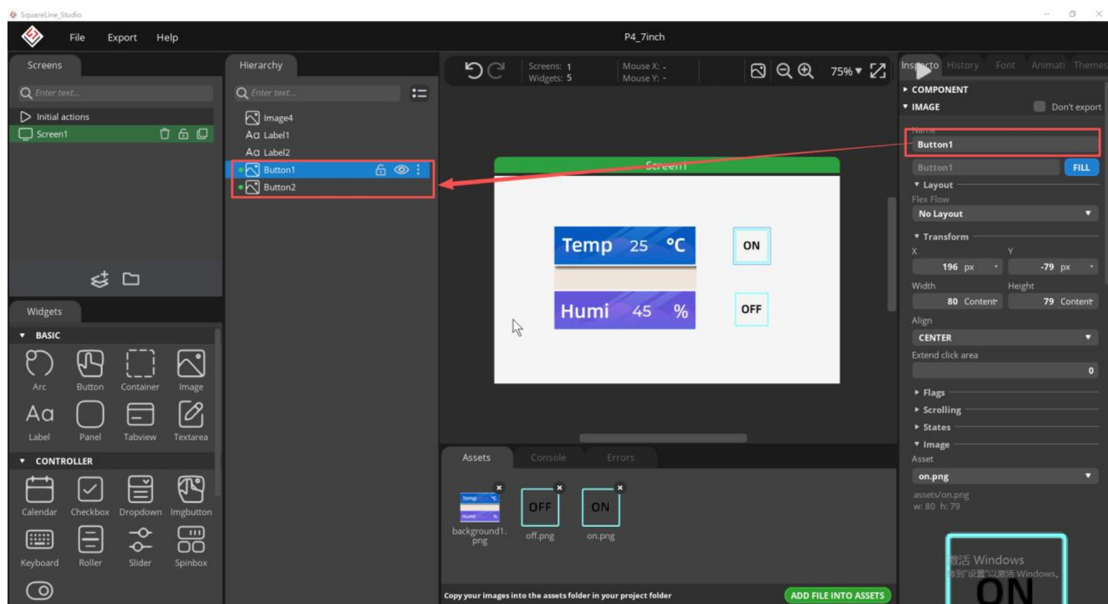


You can change the label text to set the default value.



7. Add buttons to control the LEDs; remember to rename the buttons " Button1 " and " Button2 ".

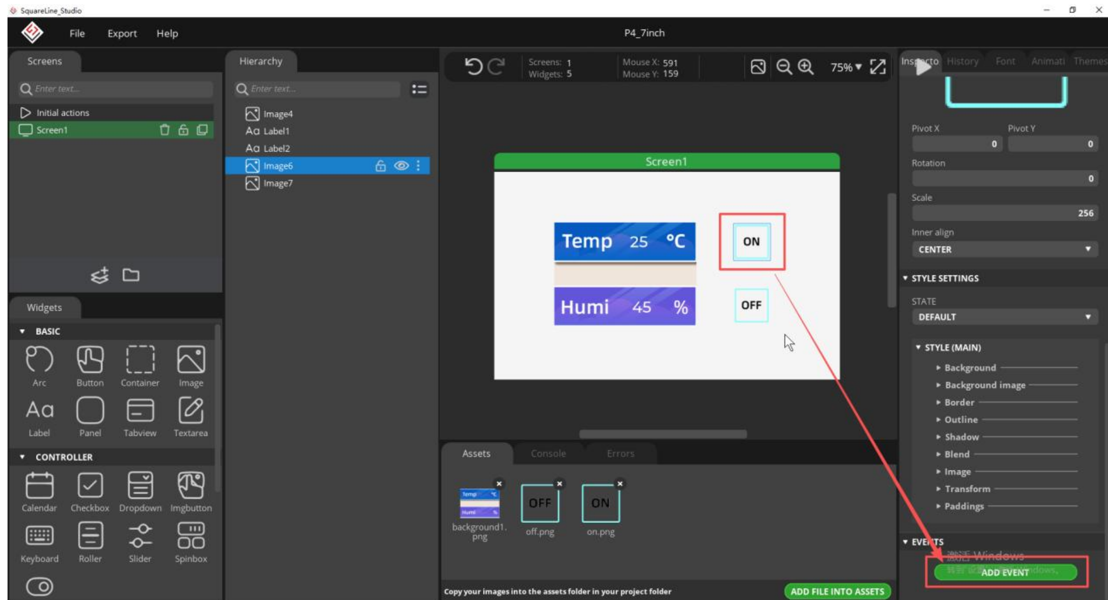




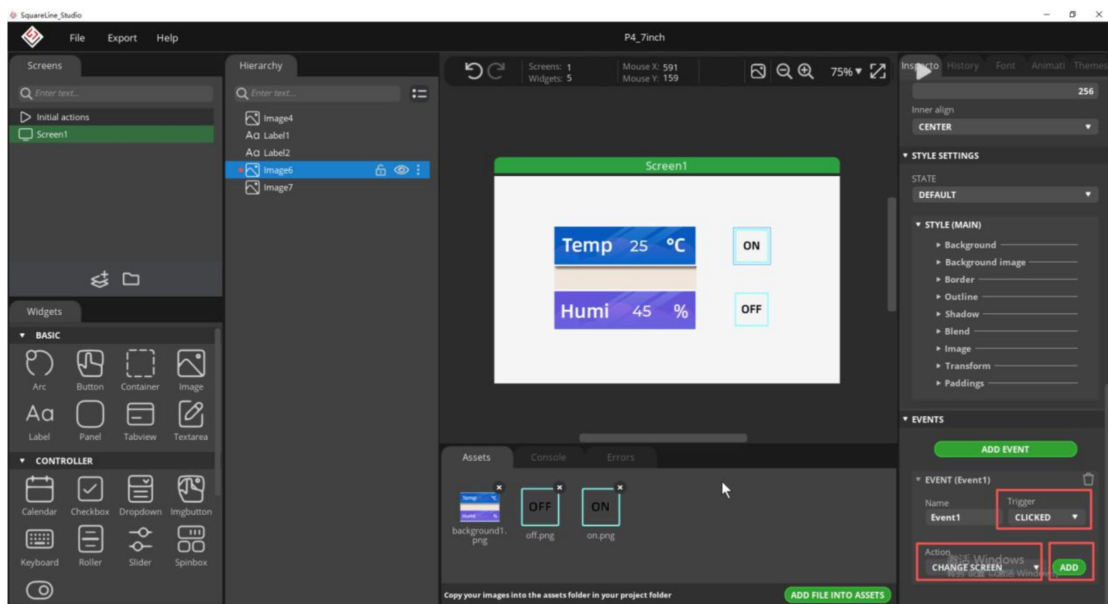
8. Add events to the button

Note: Since the button controls the LED's on/off state, we can add any event here to generate the button event code framework when exporting the UI file. We will modify the button event code later to control the LED.

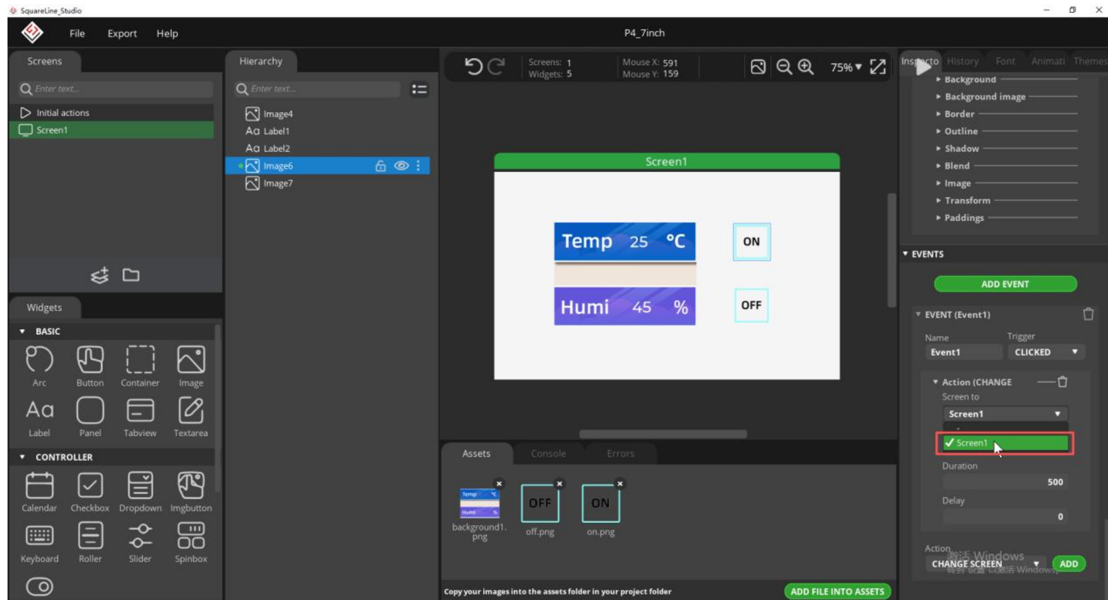
Select the button, and then click " ADD EVENT ".



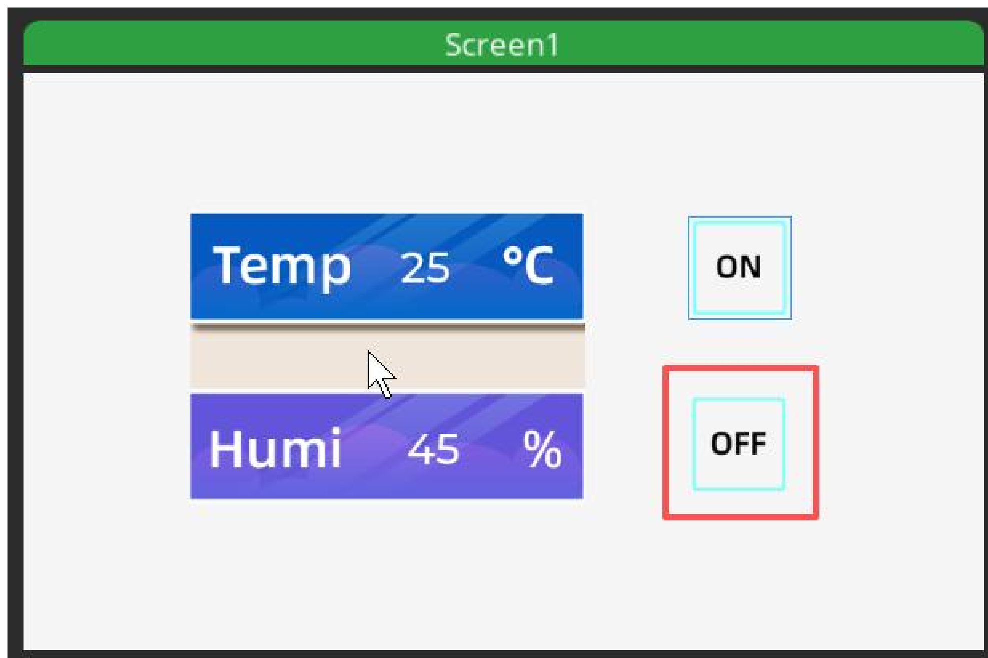
Select " clicked " as the trigger condition and choose a trigger event in " Action " . The generated program will be modified to implement the LED control function.



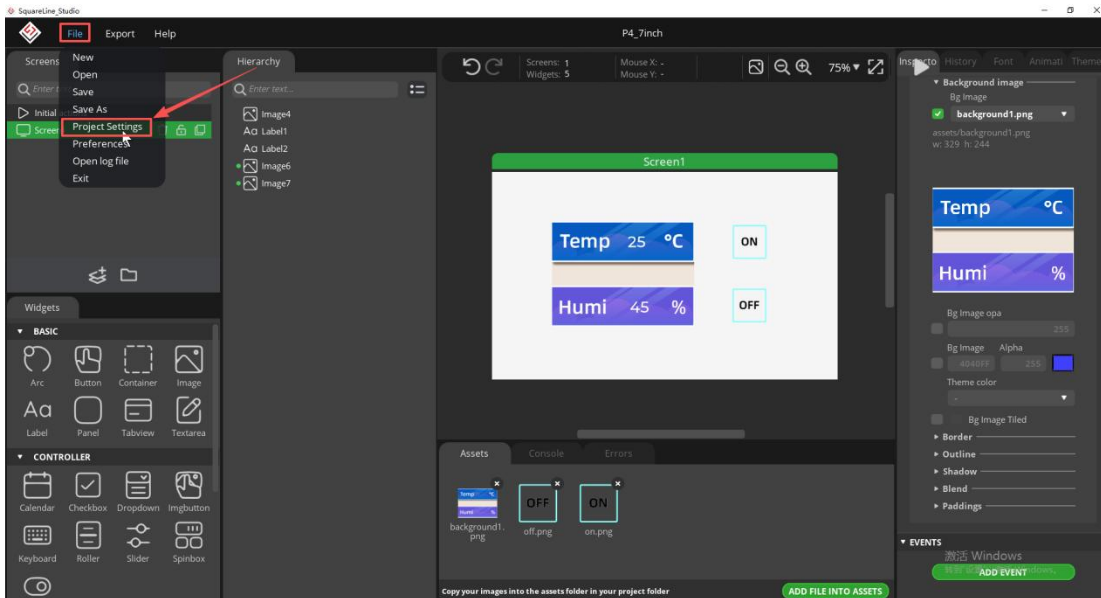
Here I choose to switch screens; the screen I want to switch to is " Screen1 ".



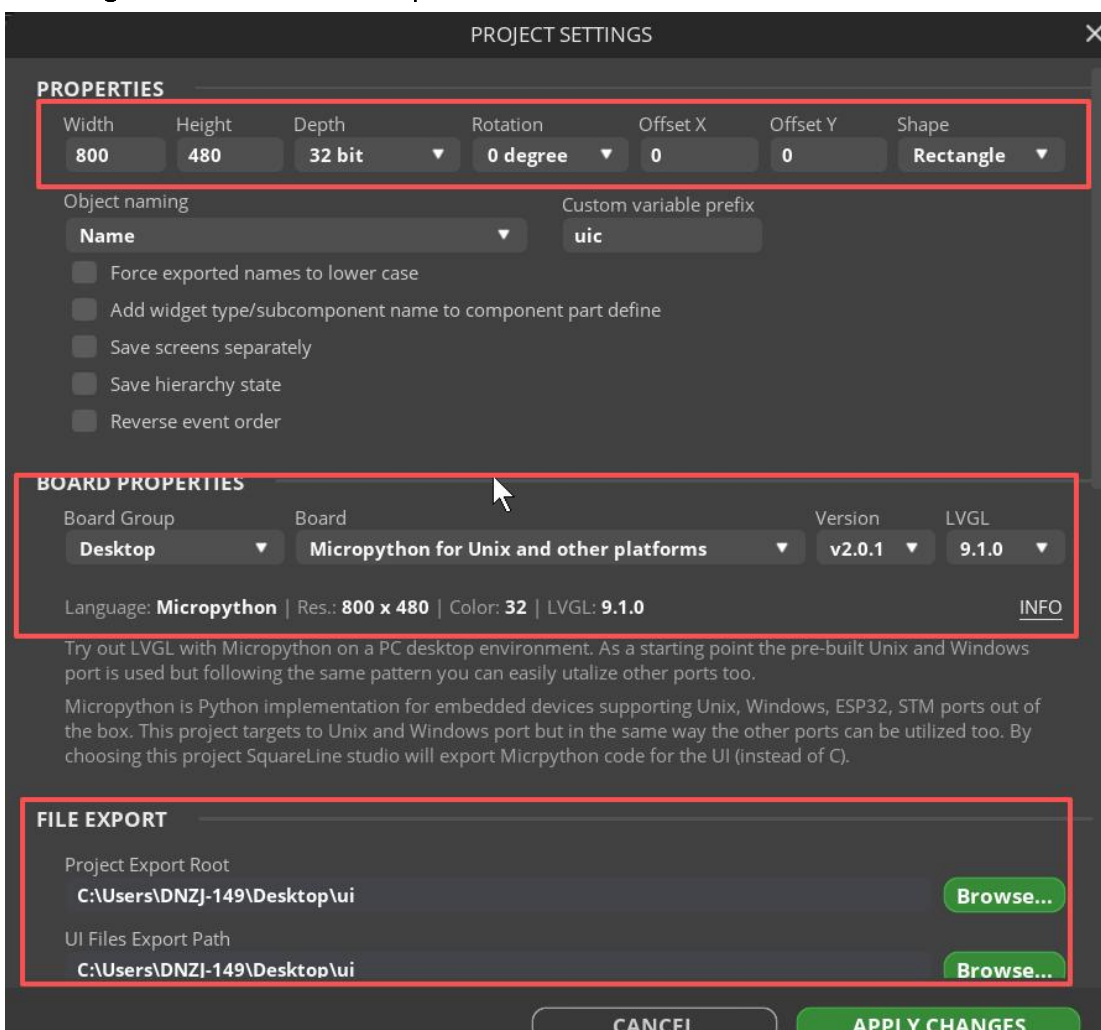
Perform the same settings on the " OFF " button.

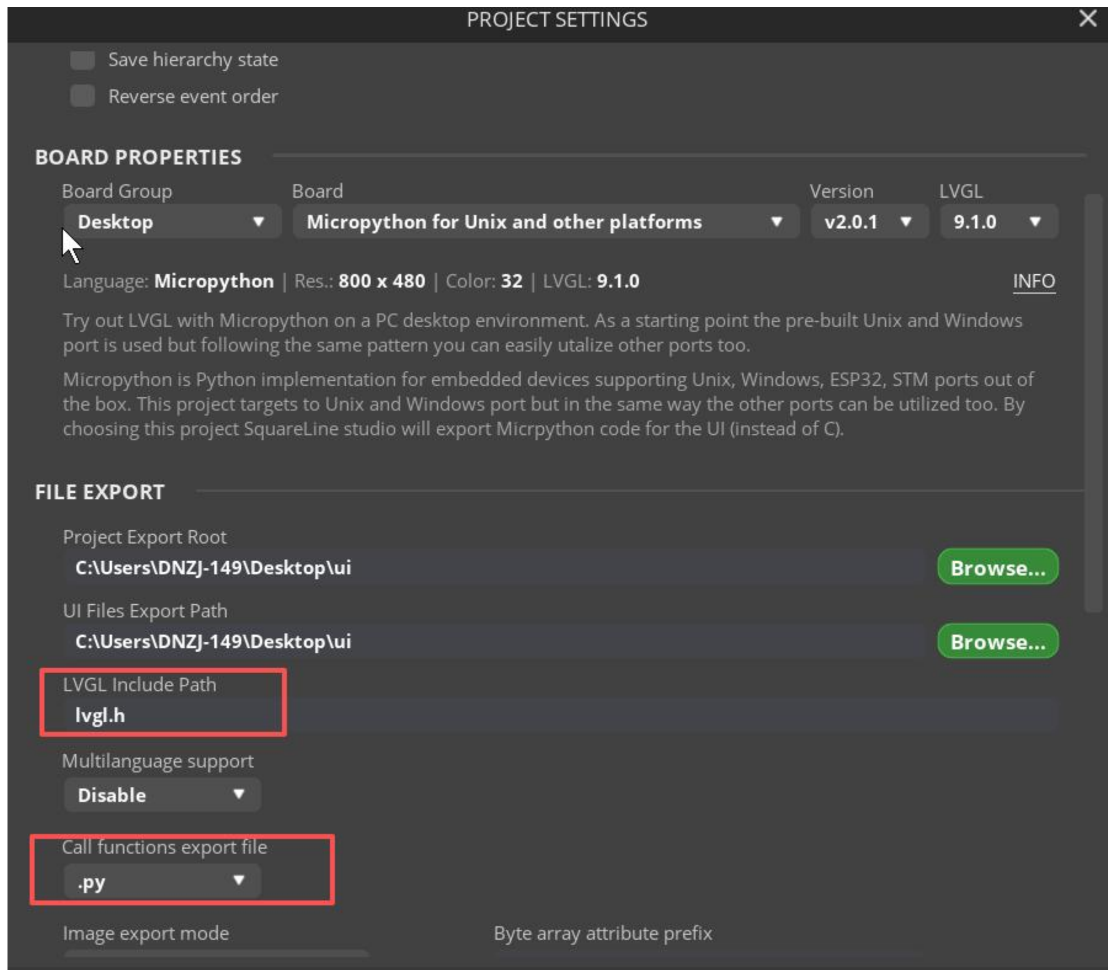


9. Export the user interface file

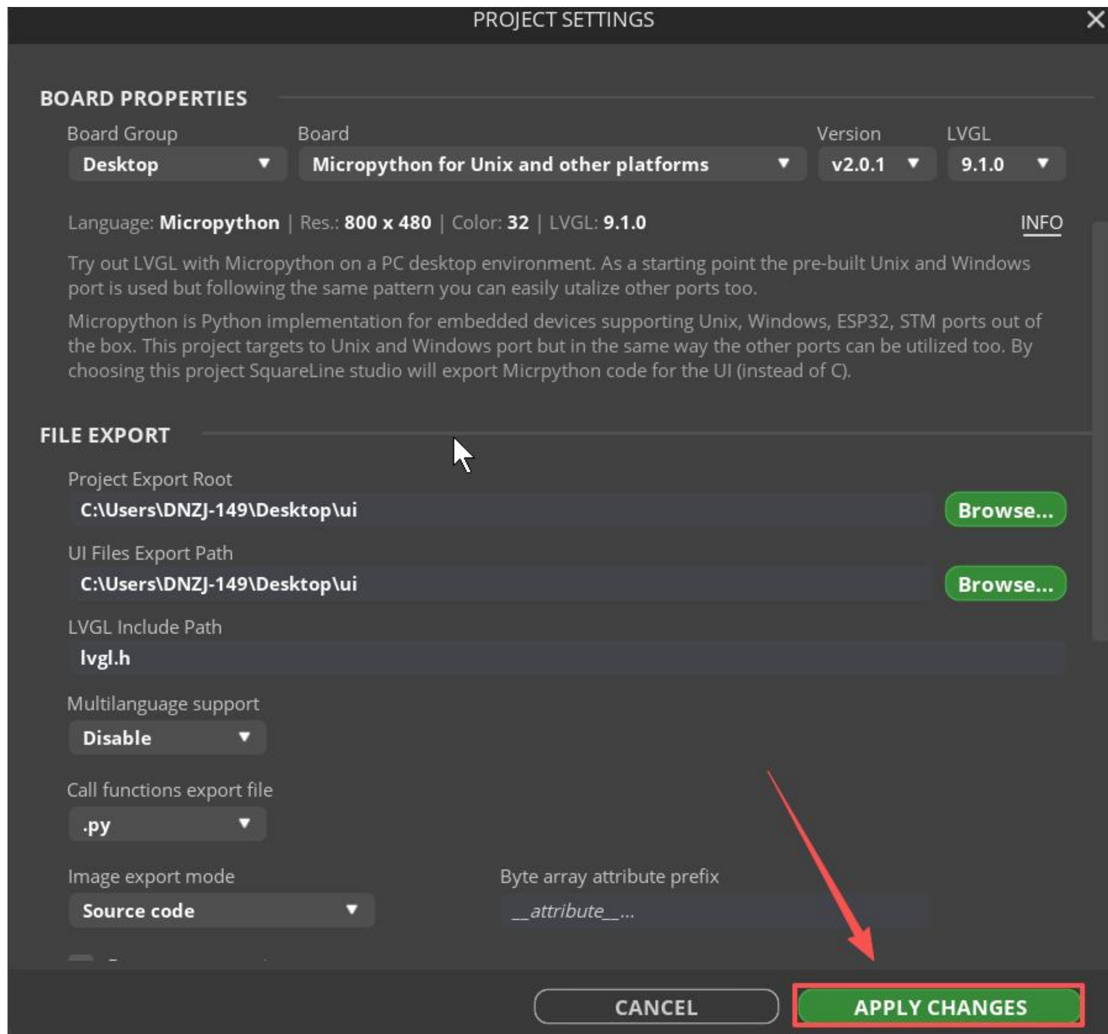


Set the export path for the files (set the path according to your own files).
Enter lvgl.h in the LVGL include path.

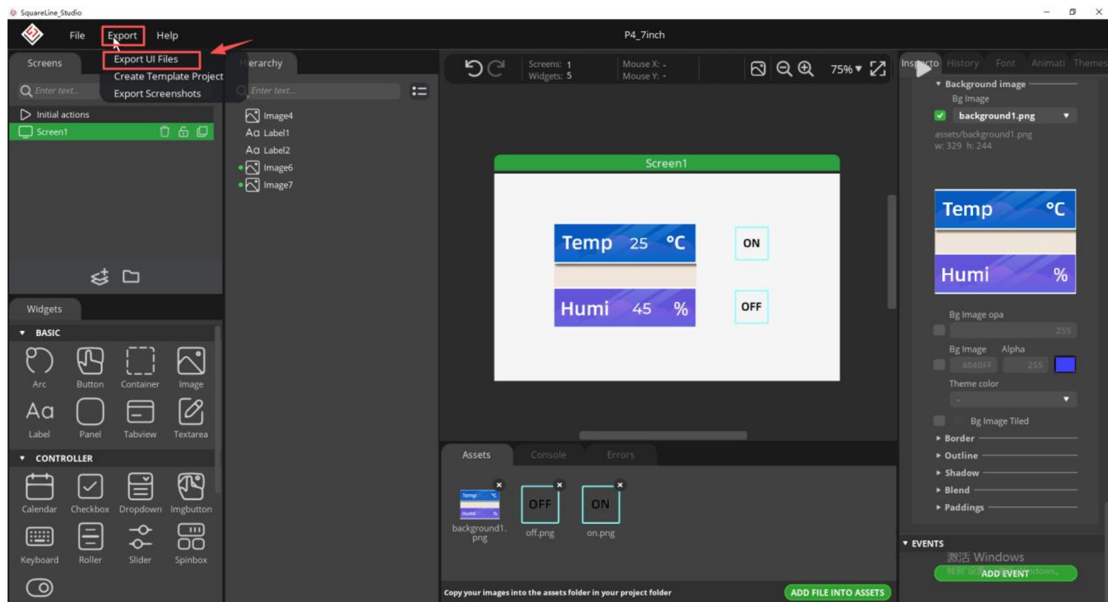




After setting it up, click " APPLY CHANGES ".



Export the user UI files to the folder we just set up.



project.info	2026/5/21 18:47	INFO 文件	1 KB
ui.py	2026/5/21 18:47	Python file	8 KB
ui_images.py	2026/5/21 18:47	Python file	1,524 KB

The UI file export is complete; we will only need the " ui_images.py " file. Next, we will learn the main program and how to use the Thonny IDE to upload the code to the development board.

Build projects using Thonny IDE

1. Please visit https://thonny.org/ and download the corresponding software version (the Windows version is used as an example here).

Note: If the latest version of the editor fails to upload firmware or compile successfully, please try the stable version 4.1.7. Download link:
https://github.com/thonny/thonny/releases/tag/v4.1.7

github.com/thonny/thonny/releases/tag/v4.1.7

thonny-4.1.7.pkg contains Universal2 build of Python 3.10 -- this means it is suitable for both Arm and Intel Macs.

Linux

thonny-4.1.7.bash is a script, which downloads and installs thonny-4.1.7-x86_64.tar.gz (with Python 3.10) when run on x86_64 machines. On other platforms it tries to use system python3 (creates a virtual environment for Thonny and installs `thonny` and its dependencies there)

Changes since 4.1.6

- Fix PyPI package search. Thonny now bases search results on the list of 5000 most popular PyPI packages. **If you need to install a less popular package, you need to enter the exact name**, #3401
- Allow selecting ESP32-C6 family in esptool dialog, #3363
- Update org.thonny.Thonny.appdata.xml
- Update bundled esptool
- Fix missing dbus-next dependency in Linux

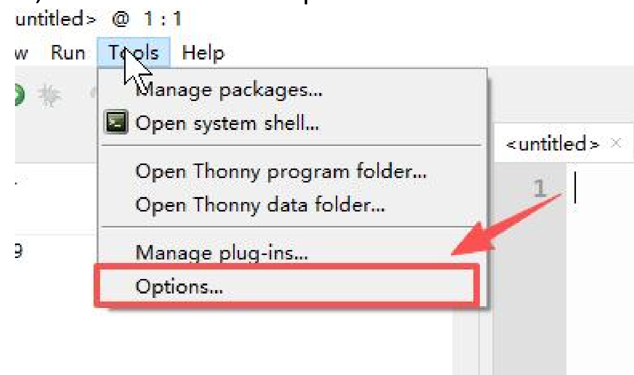
Assets 10

File	Size	Date
thonny-4.1.7-windows-portable.zip	33.1 MB	Dec 17, 2024
thonny-4.1.7-x86_64.tar.gz	41.4 MB	Dec 17, 2024
thonny-4.1.7.bash	4.28 KB	Dec 17, 2024
thonny-4.1.7.exe	22.4 MB	Dec 17, 2024
thonny-4.1.7.pkg	43 MB	Dec 17, 2024
thonny-py38-4.1.7-windows-portable.zip	26.2 MB	Dec 17, 2024
thonny-py38-4.1.7.exe	17.6 MB	Dec 17, 2024
thonny-xxl-4.1.7.exe	76.8 MB	Dec 17, 2024
Source code (zip)		Dec 17, 2024
Source code (tar.gz)		Dec 17, 2024

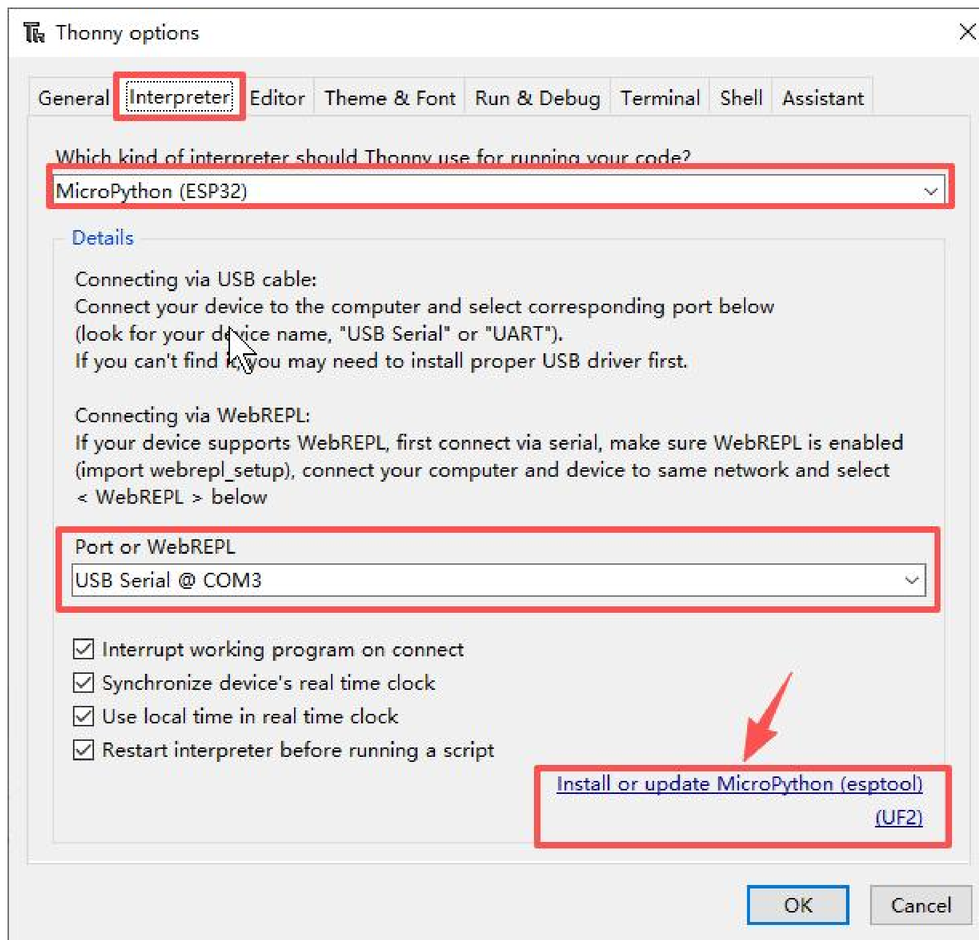
2. Double-click the downloaded .exe file to install the software.

Upload firmware

1. Connect the CrowPanel Advanced 10.1-inch ESP32-P4 HMI AI Display to your computer.
2. Open Thonny IDE, click " Tools " -> " Options "



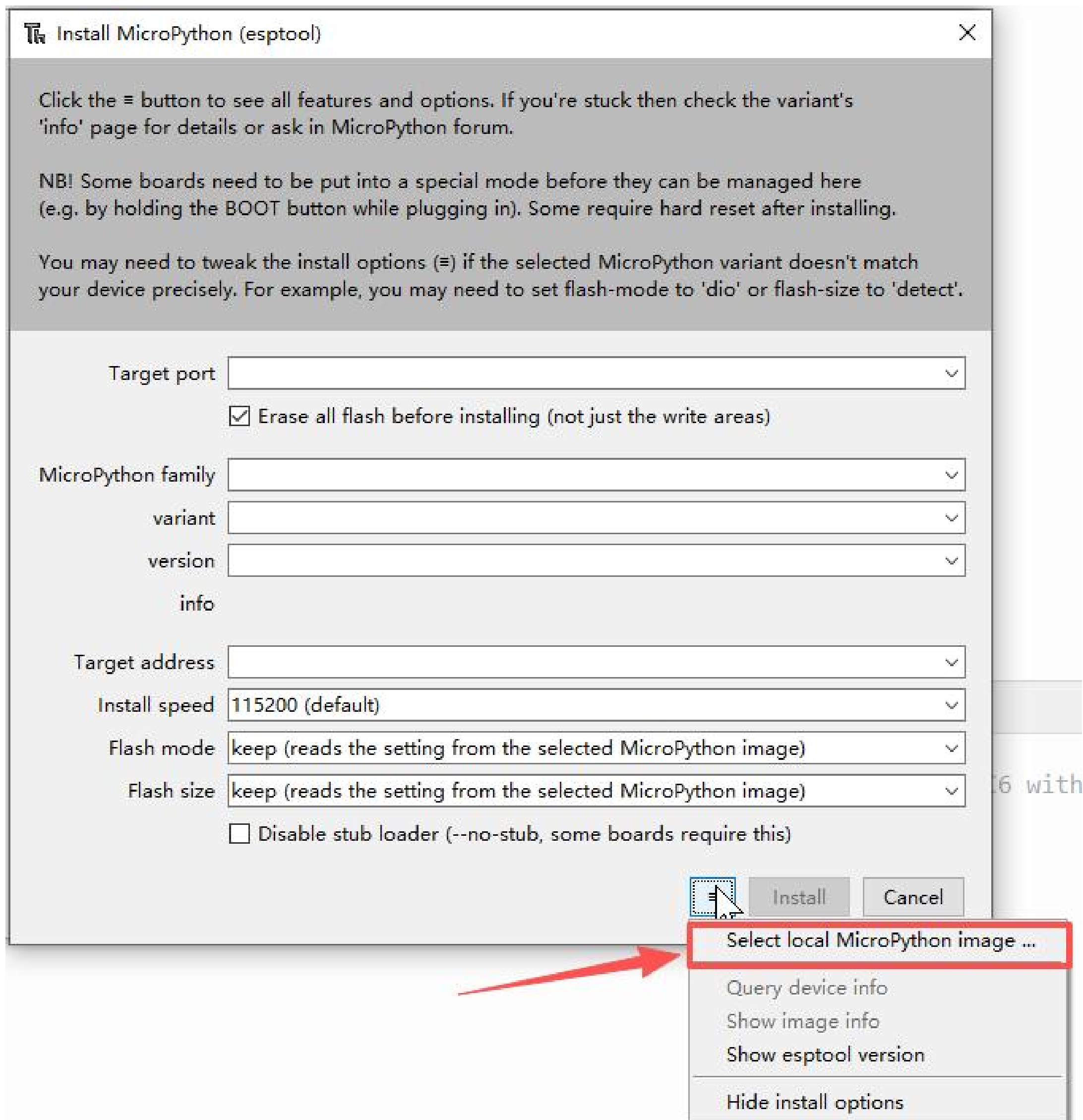
3. Select " MicroPython (ESP32) " as the interpreter, select the corresponding serial port, and then click " Install or update MicroPython (esptool) " .



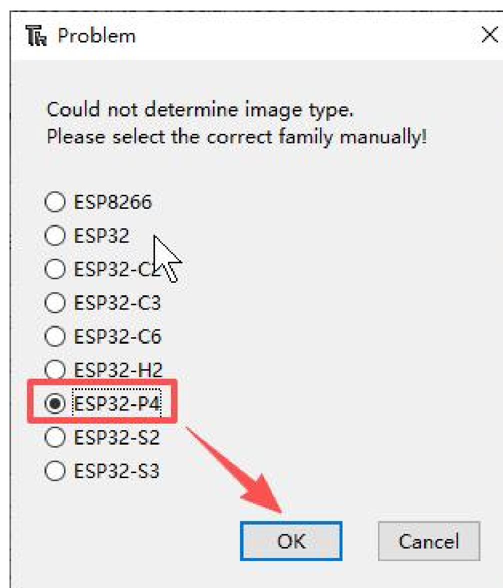
4. Click the icon with three horizontal lines, then click " Select local MicroPython image " , select "lvgl_micropy_ESP32_GENERIC_P4-C6_WIFI-16_ELECROW_INCH7-9-10_1_V1_0.bin" and install it.

Please click the following link to download the bin file:

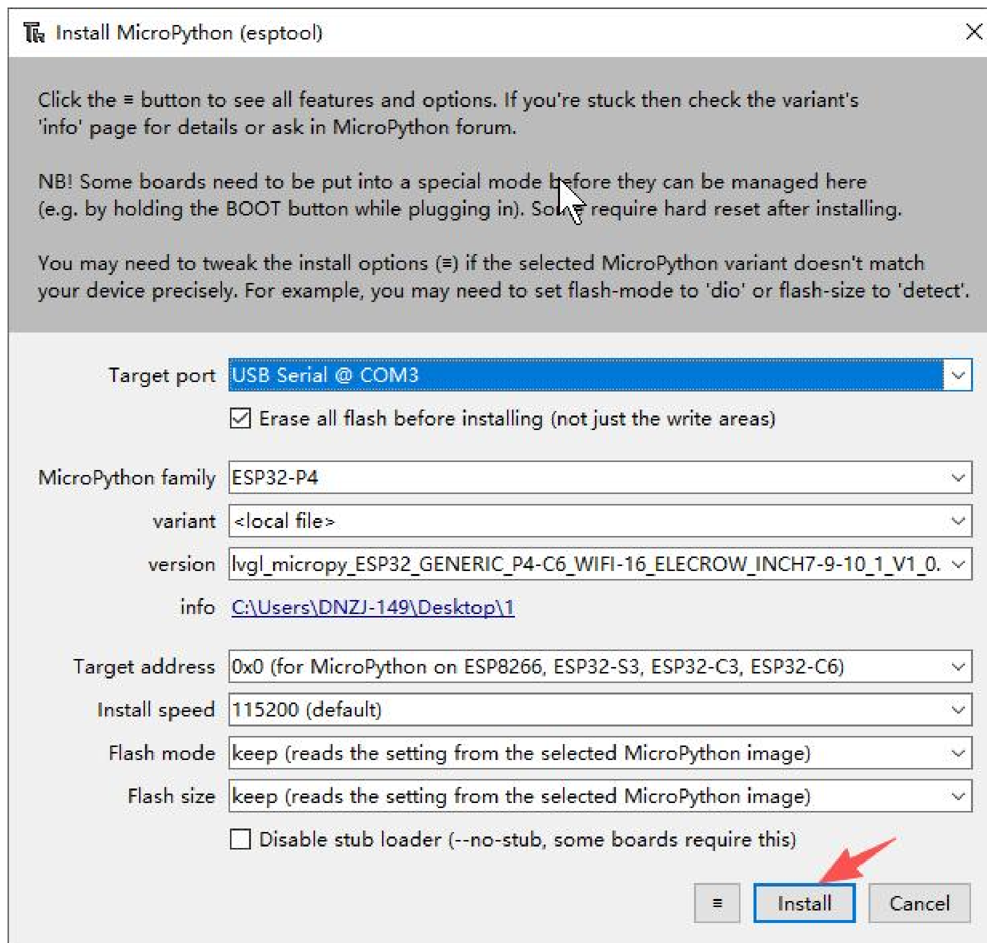
https://github.com/Elecrow-RD/CrowPanel-Advanced-10.1inch-ESP32-P4-HMI-AI-Display-1024x600-IPS-Touch-Screen/tree/master/example/V1.0/Micropython



5. Select ESP32-P4



6. After selecting the serial port and other installation parameters, click Install.



7. Waiting for installation and the installation success indicator.

Install MicroPython (esptool)

Click the ≡ button to see all features and options. If you're stuck then check the variant's 'info' page for details or ask in MicroPython forum.

NB! Some boards need to be put into a special mode before they can be managed here (e.g. by holding the BOOT button while plugging in). Some require hard reset after installing.

You may need to tweak the install options (≡) if the selected MicroPython variant doesn't match your device precisely. For example, you may need to set flash-mode to 'dio' or flash-size to 'detect'.

Target port: USB Serial @ COM3

☑ Erase all flash before installing (not just the write areas)

MicroPython family: ESP32-P4

variant: <local file>

version: lvgl_micropy_ESP32_GENERIC_P4-C6_WIFI-16_ELECROW_INCH7-9-10_1_V1_0.

info: C:\Users\DNZJ-149\Desktop\1

Target address: 0x0 (for MicroPython on ESP8266, ESP32-S3, ESP32-C3, ESP32-C6)

Install speed: 115200 (default)

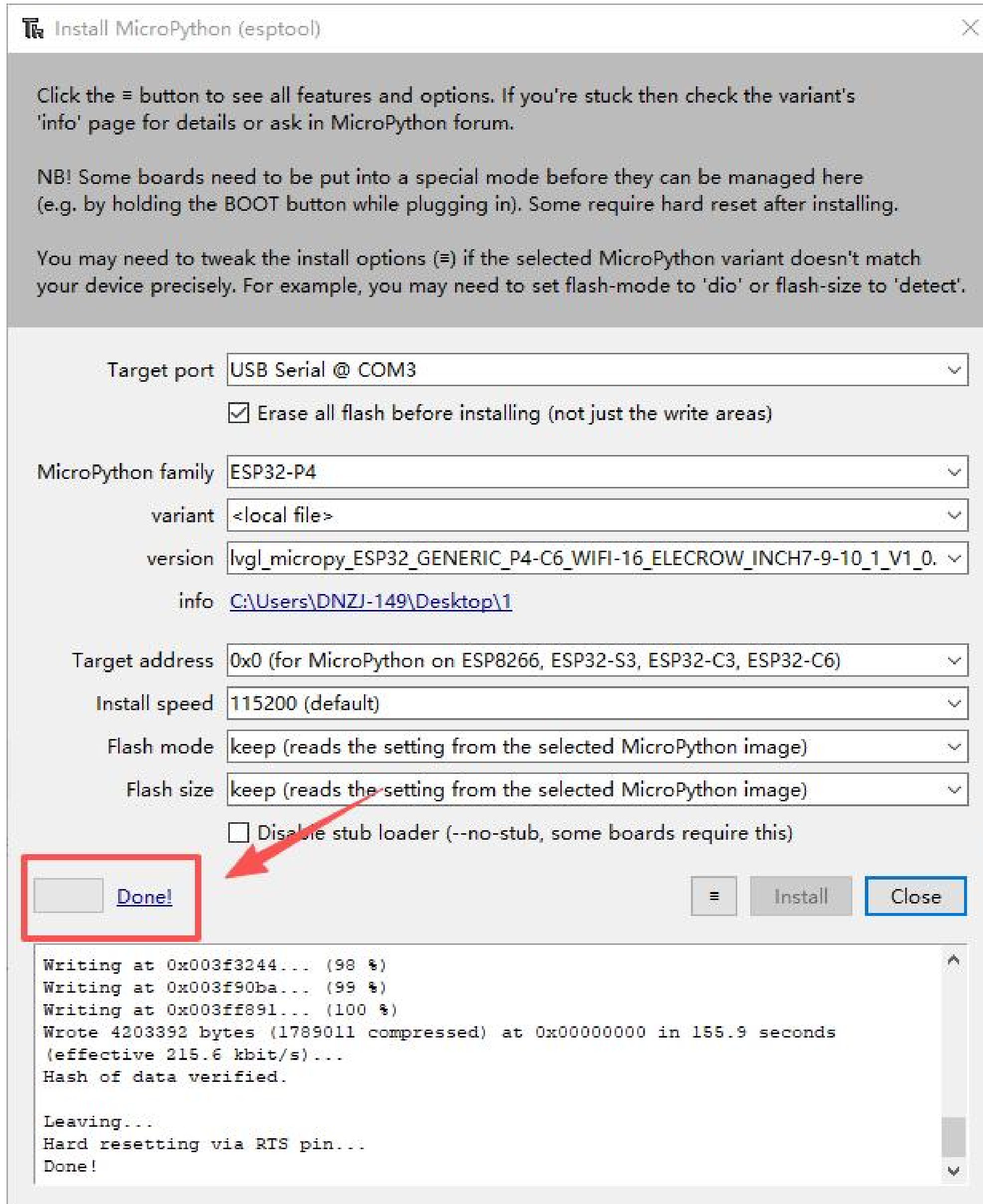
Flash mode: keep (reads the setting from the selected MicroPython image)

Flash size: keep (reads the setting from the selected MicroPython image)

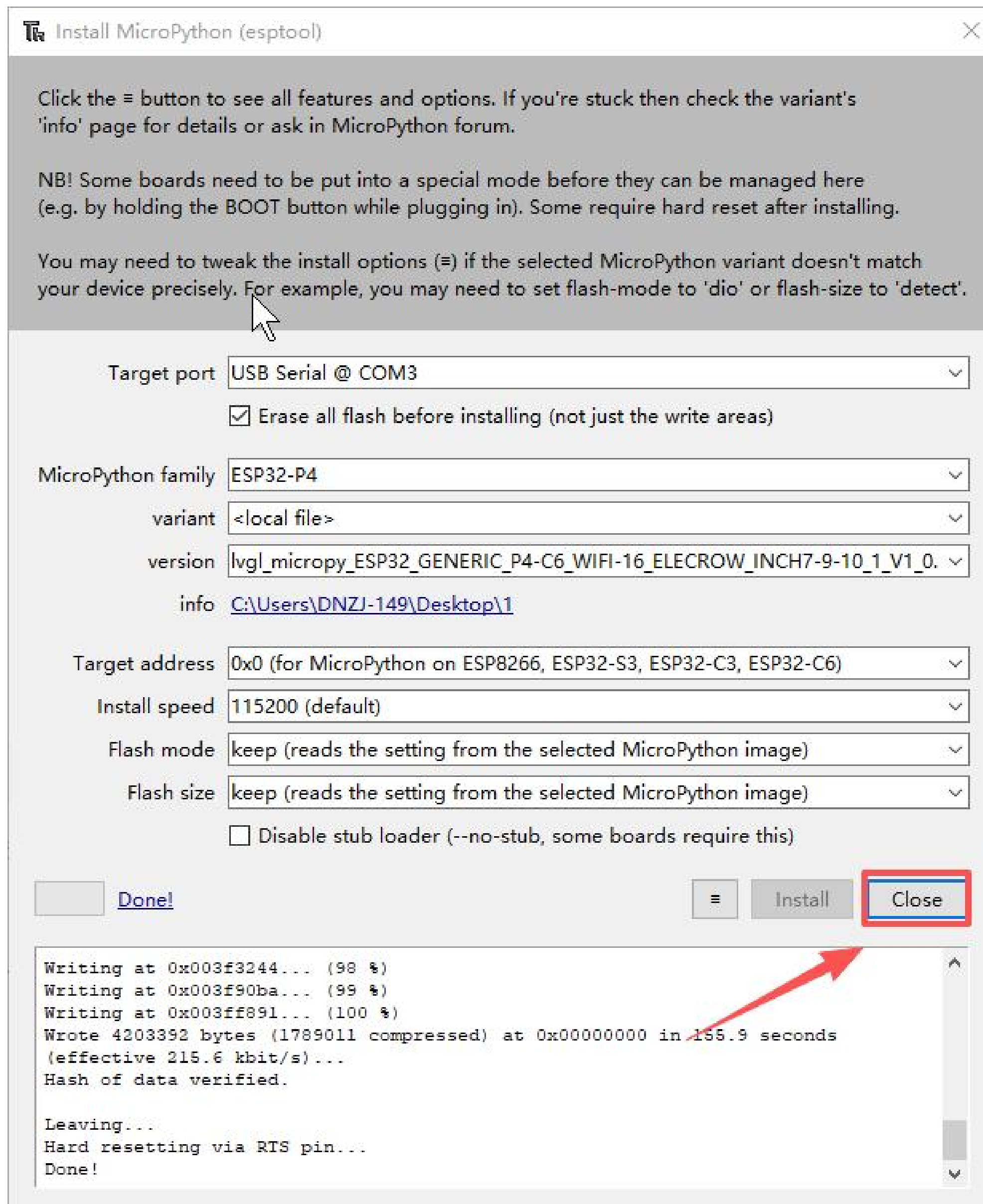
☐ Disable stub loader (--no-stub, some boards require this)

Writing at 0x00074baf... (10 %)

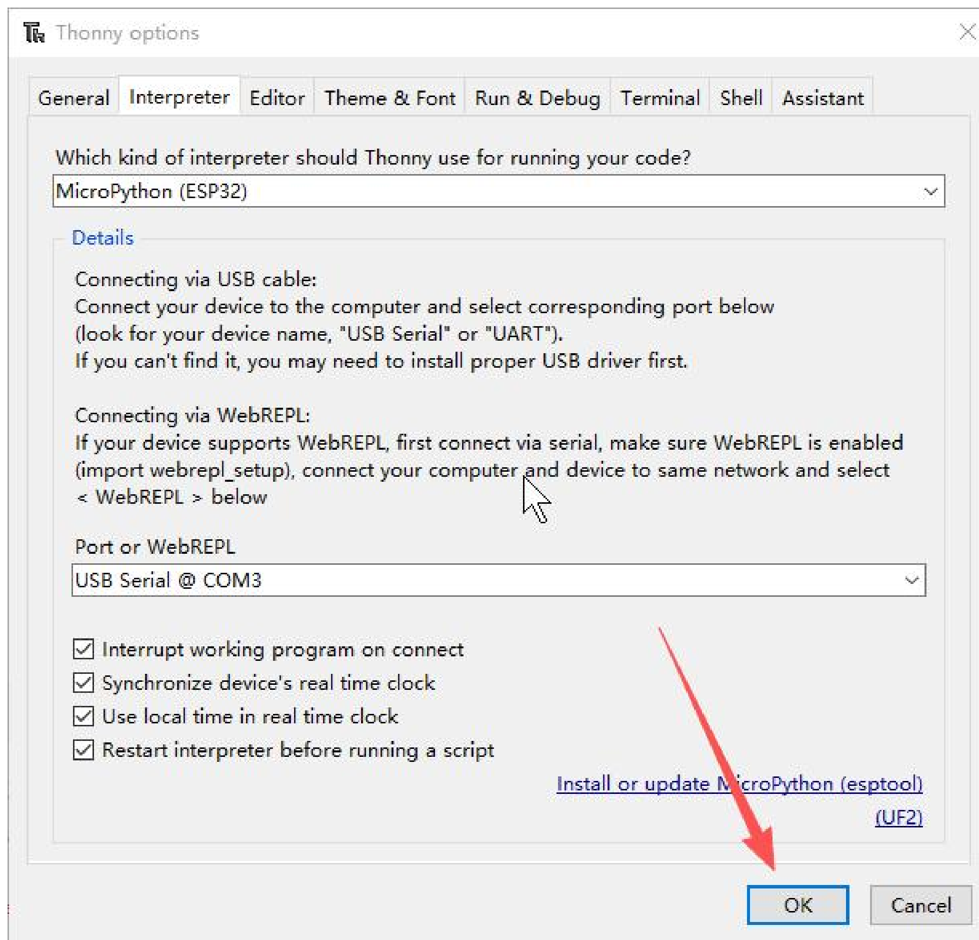
≡ Install Cancel



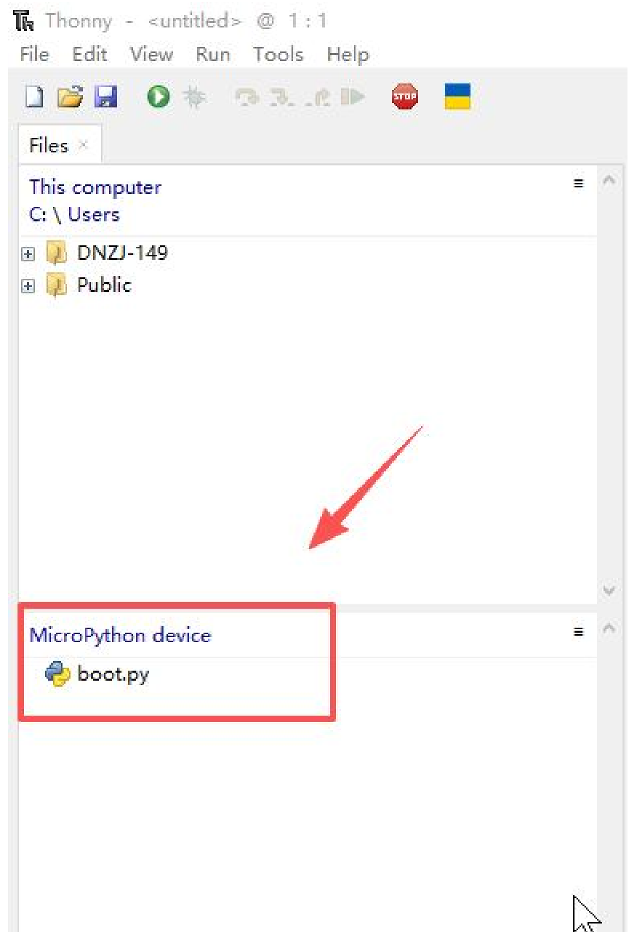
8. Click " Close "



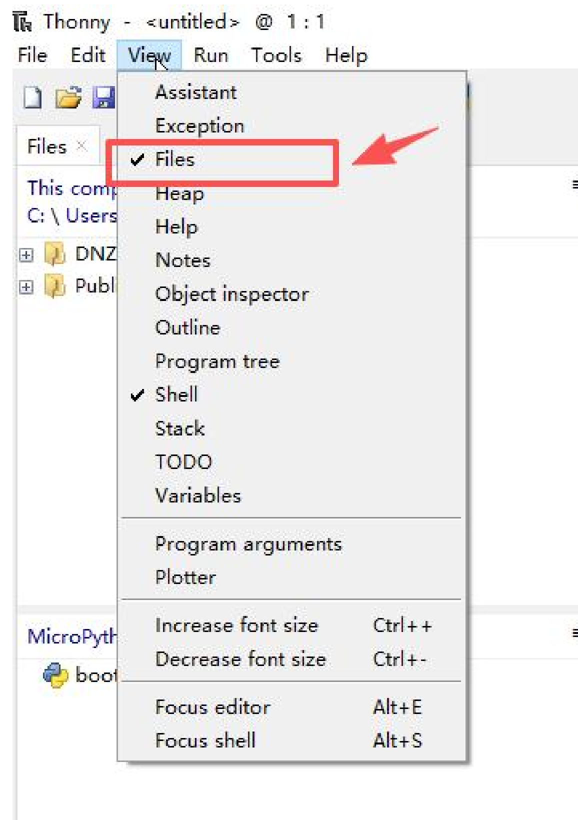
9. Click OK



10. If a " MicroPython device " window and a " boot.py " file appear in the lower left corner, the installation is successful.

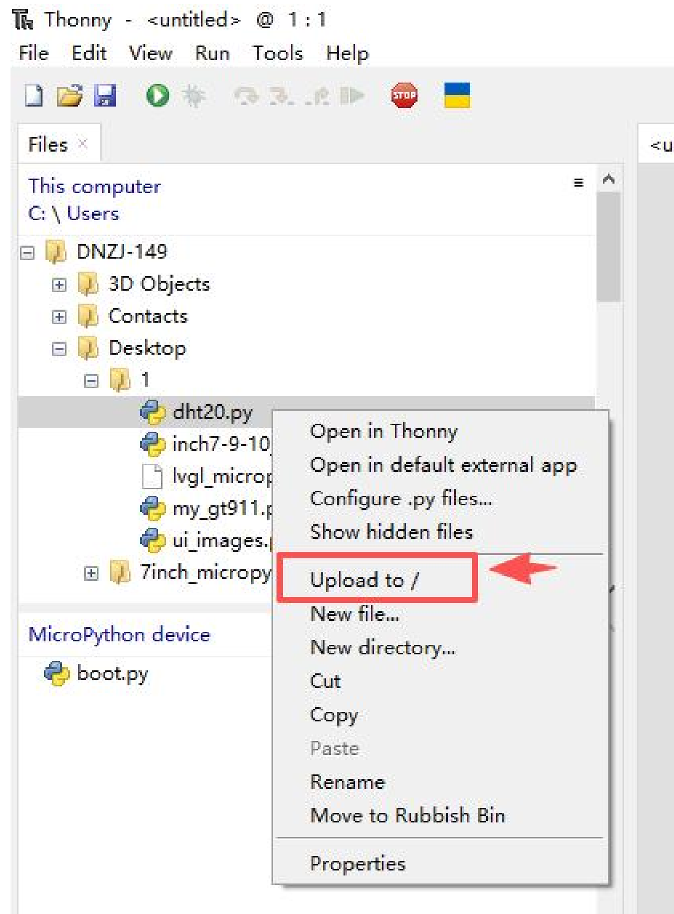


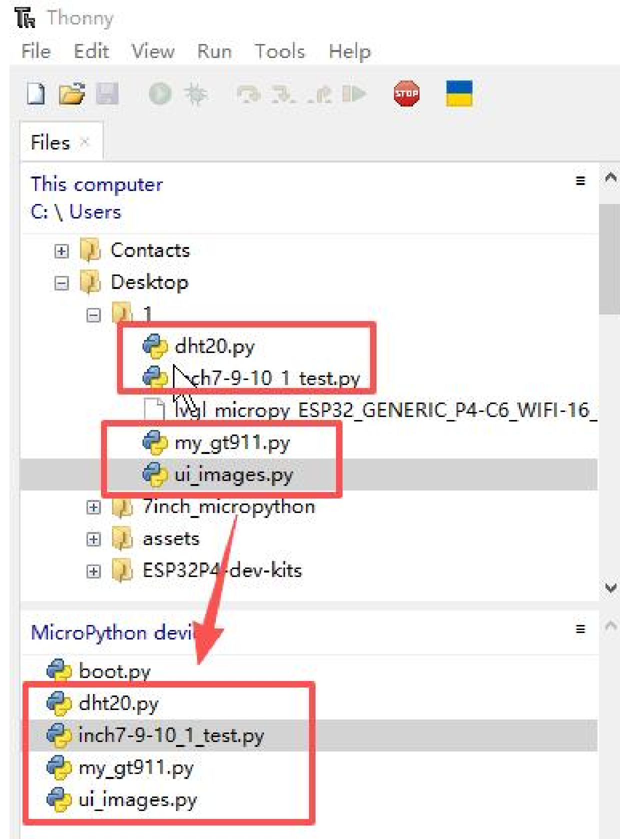
Note: If this window does not appear, you can open it by clicking " View " -> " File " at the top.



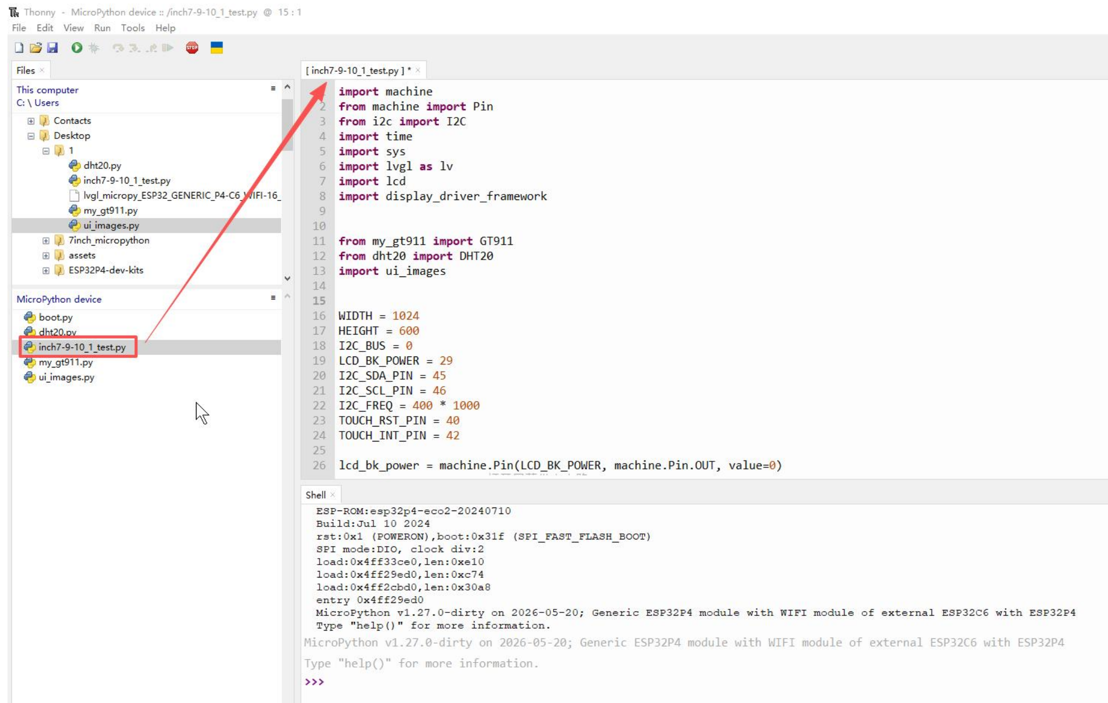
Upload code

1. In the " This computer " window, locate the downloaded code files and upload all four files to the " MicroPython device " in sequence.

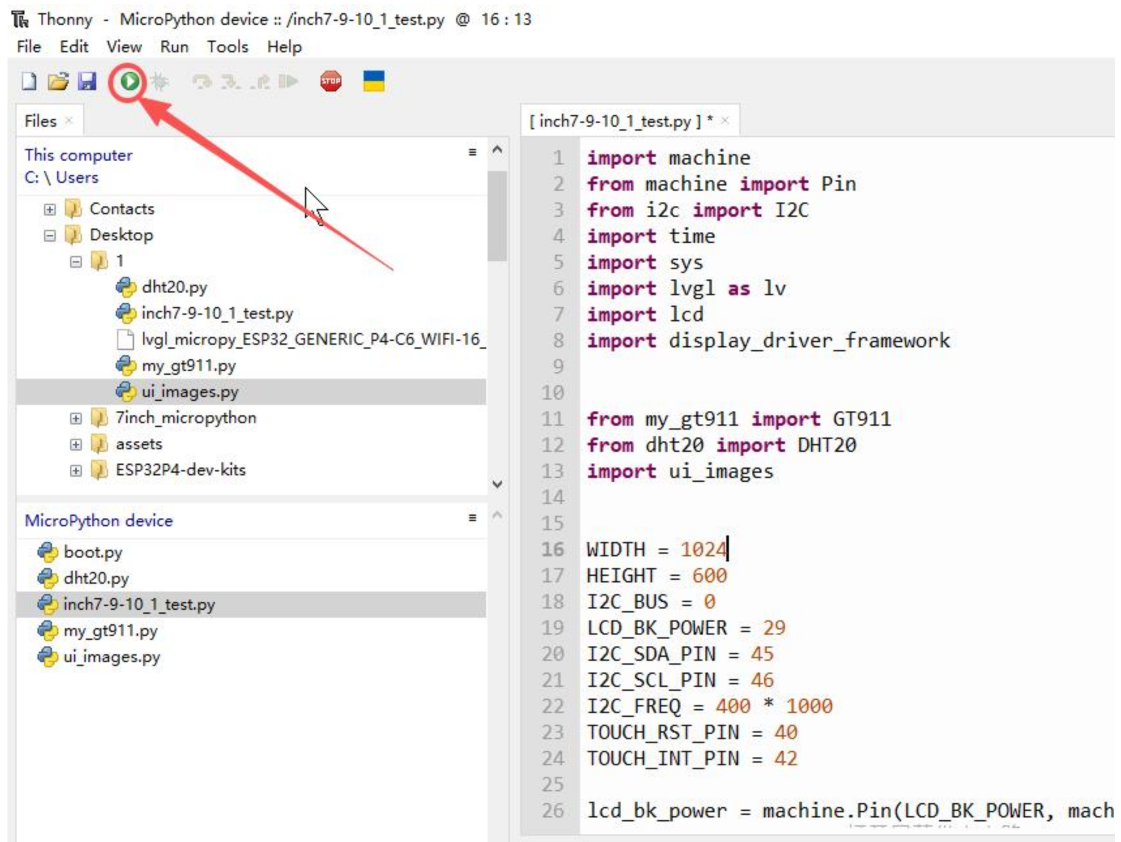


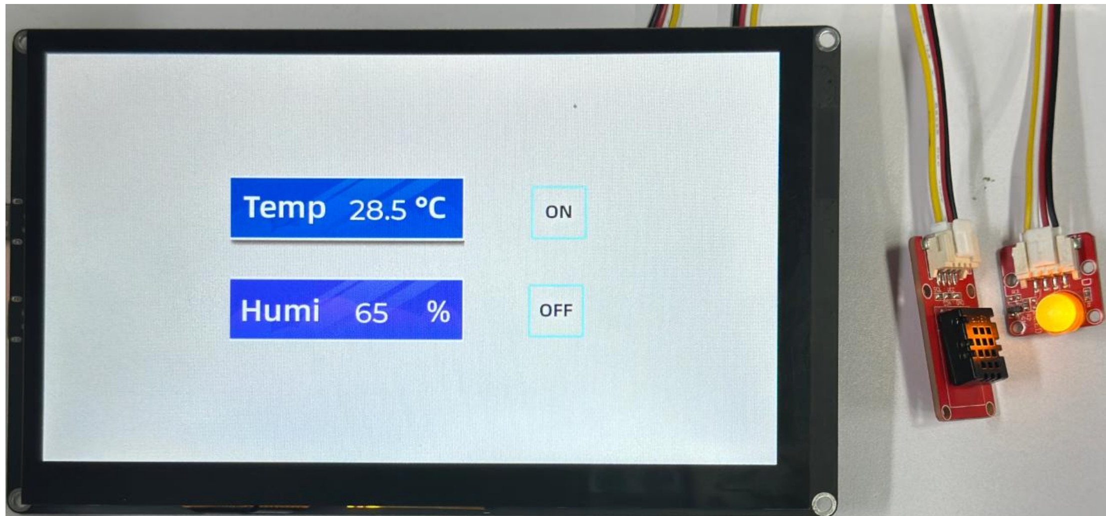


2. Double-click to open the main program " inch7-9-10_1test.py ".



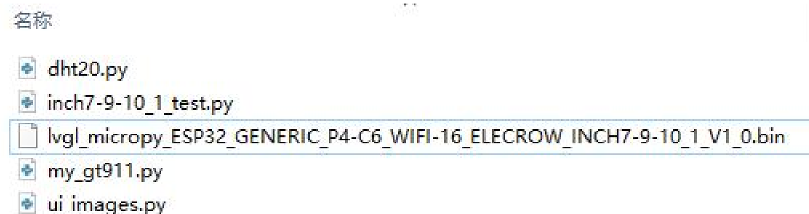
3. Click the Run button, and you will see the UI interface appear on the screen.





Code Explanation

The code files we downloaded contained four "python" files and one "bin" file.



dht20.py : The driver file for the temperature and humidity sensor.
my_gt911.py : Driver file for the GT911 touchscreen.
ui_images.py : Image resource files required for the interface exported from SquareLine Studio.
lvgl_micropy_ESP32_GENERIC_P4-C6_WIFI-16_ELECROW_INCH7-9-10_1_V1_0.bin : Elecrow packaged firmware that allows your Python scripts to display the LVGL UI, handle touch events, and interact with the DHT20 sensor on the LCD of the "CrowPanel Advanced 10.1inch ESP32-P4 HMI" .
inch7-9-10_1_test.py : This is the main program that runs the entire case study, and we will mainly explain this code file.

Import libraries and modules

```
import machine
from machine import Pin
from i2c import I2C
import time
import sys
import lvgl as lv
import lcd
```

```
import display_driver_framework

from my_gt911 import GT911
from dht20 import DHT20
import ui_images
```

machine : MicroPython's hardware interface module, used to control GPIO, I2C, SPI, etc.
Pin : Used to control a single pin (GPIO).
i2c.I2C : I2C bus module, used for communication with peripherals (DHT20, GT911).
time : Provides a delay function.
sys : System module used for exception printing.
LVGL : LVGL is an open-source, lightweight GUI library **.**
lcd, display_driver_framework : LCD display driver.
my_gt911 : GT911 touchscreen driver.
dht20 : Driver for DHT20 temperature and humidity sensor.
ui_images : Image resources used in the interface.

Hardware parameter configuration

```
WIDTH = 1024
HEIGHT = 600
I2C_BUS = 0
LCD_BK_POWER = 29
I2C_SDA_PIN = 45
I2C_SCL_PIN = 46
I2C_FREQ = 400 * 1000
TOUCH_RST_PIN = 40
TOUCH_INT_PIN = 42

lcd_bk_power = machine.Pin(LCD_BK_POWER, machine.Pin.OUT, value=0)
lcd_bk_power(0)      # Enable screen power circuit
LED = Pin(48, Pin.OUT)   # Set GPIO pin 48 to output mode
```

The hardware pinouts, I2C bus numbers, and frequencies for the LCD and touchscreen are defined. 'lcd_bk_power' controls the LCD backlight power supply. 'LED' is a GPIO output pin used to indicate the LED's status (on/off).

Initialize device

```
def device_init():
    lv.init()
    lcd.init()
    lcd.clear(0xFFFFFF) # Fill the entire screen with white
    lcd.backlight_set(100) # Enable backlight (0~100)
    print("✅ MIPI LCD hardware initialization completed")
```

lv.init() initializes LVGL.
The lcd.init() function initializes the LCD display.

lcd.clear(0xFFFFFF) clears the screen with white.
lcd.backlight_set(100) sets the backlight brightness to the maximum.

Create an LVGL display instance

```
# Create LVGL display instance
display = lv.display_create(WIDTH, HEIGHT)
display.set_default()

# Configure display buffers and flush callback
BUFFER_SIZE = (WIDTH * HEIGHT * 3)
draw_buf1 = bytearray(BUFFER_SIZE)
draw_buf2 = bytearray(BUFFER_SIZE)
```

The 'display_create' function creates an LVGL display object. LVGL requires a framebuf (draw_buf) to render the interface; here, double-buffered draw_buf1 and draw_buf2 are created, with each pixel occupying 3 bytes (RGB888). Double buffering can avoid screen flicker and improve UI smoothness.

Set refresh callback

```
def flush_cb(display, area, color_p):
    w = area.x2 - area.x1 + 1
    h = area.y2 - area.y1 + 1
    size_in_bytes = w * h * 3
    try:
        data_view = color_p.__dereference__(size_in_bytes)
        lcd.flush(area.x1, area.y1, area.x2, area.y2, data_view)
    except Exception as e:
        print(f"❌ LCD refresh exception: {e}")
    display.flush_ready()

display.set_flush_cb(flush_cb)
print("✅ LVGL display initialization completed")
```

flush_cb is a callback function called when LVGL renders the screen.
The area represents the rectangular region that needs to be refreshed.
color_p is the data rendered by LVGL.
lcd.flush(...) transfers pixel data to the LCD.
display.flush_ready() tells LVGL that the flush is complete.

Initialize I2C device

```
i2c_bus = I2C.Bus(
    host=I2C_BUS,
    scl=I2C_SCL_PIN,
    sda=I2C_SDA_PIN,
    freq=I2C_FREQ,
)
```

```
print(f"\n Devices on I2C bus: {[hex(d) for d in i2c_bus.scan()]}")
```

Create an I2C bus object. **The 'scan()'** function scans the device addresses on the bus to confirm if the peripheral is connected correctly. If no I2C device is connected, the program will freeze here.

Initialize the DHT20 temperature and humidity sensor

```
i2c_dht20 = I2C.Device(
    bus = i2c_bus,
    dev_id = DHT20.I2C_ADDR,
    reg_bits=8,
)

# Initialize DHT20 sensor
try :
    dht20 = DHT20(i2c_dht20)
except Exception as e:
    sys.print_exception(e)
    print('Failed to initialize DHT20 sensor!')
```

Create a DHT20 device object via I2C and then initialize it.

Initialize the GT911 touchscreen

```
# Initialize GT911 touch driver
device_gt911 = I2C.Device(
    bus=i2c_bus,
    dev_id=GT911.I2C_ADDR,
    reg_bits=16,
)

touch = GT911(
    device=device_gt911,
    reset_pin=TOUCH_RST_PIN,
    interrupt_pin=TOUCH_INT_PIN,
)
```

The touchscreen also connects via I2C. **The reset_pin** and **interrupt_pin** are used to control and respond to touch events.

Create an LVGL input device

```
indev = lv.indev_create()
indev.set_type(lv.INDEV_TYPE.POINTER)
indev.set_read_cb(touch._get_coords)
```

Create an LVGL input device object of type POINTER (touch, mouse). **'set_read_cb'** binds the GT911 touch coordinate function to LVGL.

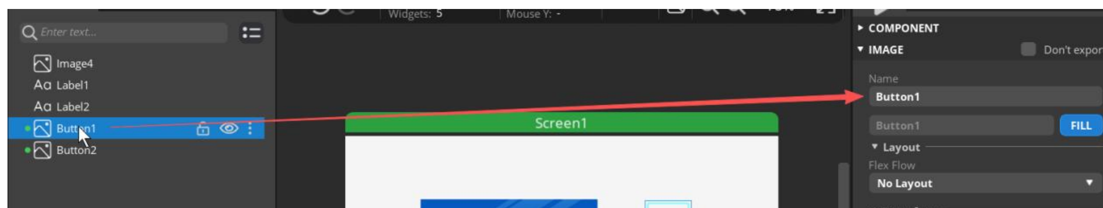
LVGL UI Build Create Button Event

```
def Button1_eventhandler(event_struct):
    event = event_struct.get_code()
    if event == lv.EVENT.CLICKED:
        LED.value(1)

def Button2_eventhandler(event_struct):
    event = event_struct.get_code()
    if event == lv.EVENT.CLICKED:
        LED.value(0)
```

Clicking the button changes the LED state: Button1 turns the light on, and Button2 turns the light off.

Note: If you did not name the UI elements Button1 and Button2 when creating them, you need to change the function names to the button names used when creating the UI.



Creating UI

```
def create_lvgl_ui():
    ui_Screen1 = lv.obj()
    SetFlag(ui_Screen1, lv.obj.FLAG.SCROLLABLE, False)
    ui_Screen1.set_style_bg_color(lv.color_hex(0xF0E5DA), lv.PART.MAIN | lv.STATE.DEFAULT)
```

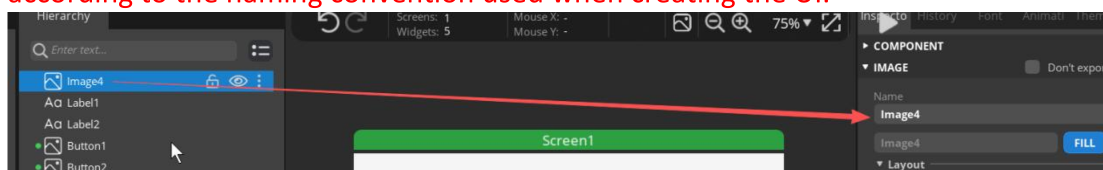
'lv.obj()' creates a screen object. The background color and style are set. Buttons, images, and labels are then created, and their positions and events are set.

Buttons, images, and labels

```
ui_Button1 = lv.image(ui_Screen1)
ui_Button1.set_src(ui_images.ui_img_on_png)
ui_Button1.add_event_cb(Button1_eventhandler, lv.EVENT.ALL, None)
```

Replace buttons with images. Set their size, position, and clickability. Add click event callbacks. Similar functionality applies to the close button **(Button2)** , the background image **(ui_Image4)** , and the temperature and humidity labels **(ui_Label1** and **ui_Label2**).

Note: Within the **'create_lvgl_ui'** function, 'ui_Image4' is also strictly modified according to the naming convention used when creating the UI.



Main loop

```
while True:
    current_time = time.ticks_ms()

    if time.ticks_diff(current_time, last_dht_time) >= 1000:
        dht20.measure()
        temp = dht20.temperature()
        hum = dht20.humidity()
        ui_Label1.set_text(f"{round(temp, 1)}")
        ui_Label2.set_text(f"{round(hum)}")
        last_dht_time = current_time

    lv.tick_inc(5)
    lv.timer_handler()
    time.sleep_ms(5)
```

1. Read the temperature and humidity once per second and update the UI accordingly.
2. lv.tick_inc(5) tells LVGL that 5ms has passed.
3. lv.timer_handler() handles touch and redraw.
4. sleep_ms(5) prevents excessive CPU usage and ensures smooth UI operation.