ESP32-S3 STICK POE A CAM

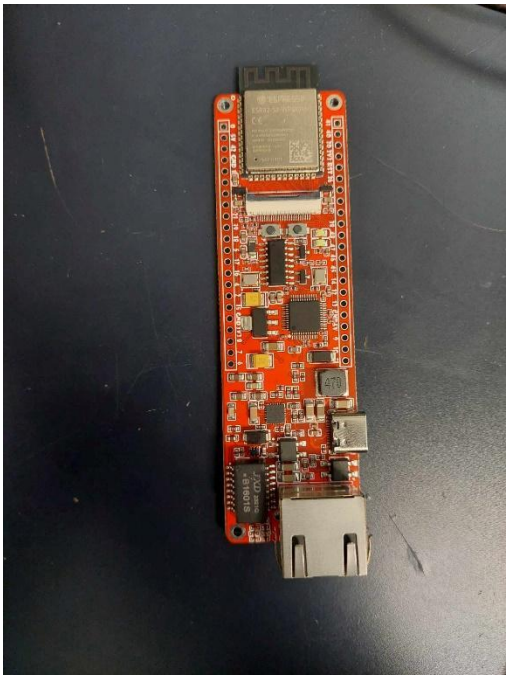
INTRODUCTION:

Hello everyone, I'm introduce you to the ESP32-S3 stick. In this demonstration, I'll show you the process of connecting a camera (Ov2640 or Ov5640) to the board and capturing an image to send to a web server. For this project, we're going to use Visual Studio Code for both the Python script (camera screen) and C++ programming (ESP32).

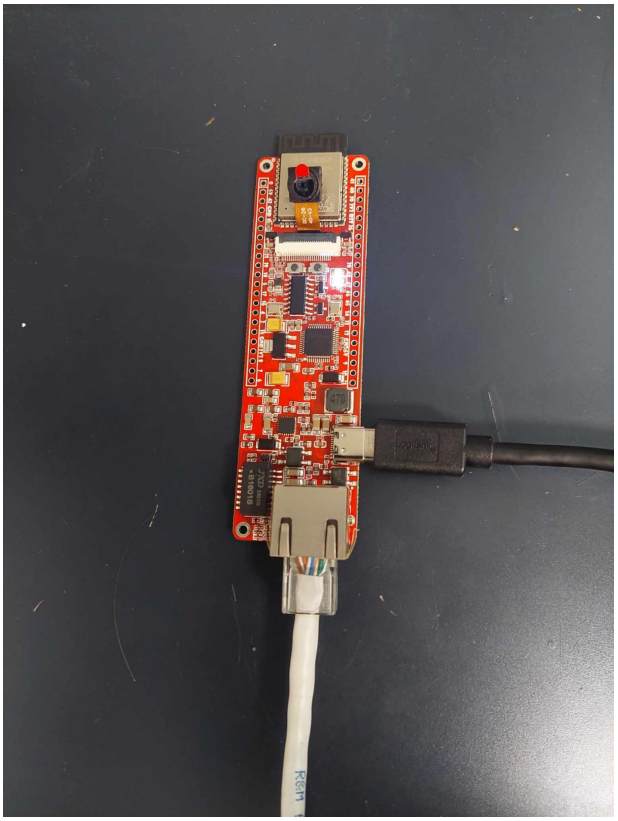
List of hardware you need:

- 1x USB-C cable
- 1x Ethernet cable
- ESP32-S3 stick
- Camera Ov2640 or Ov5640

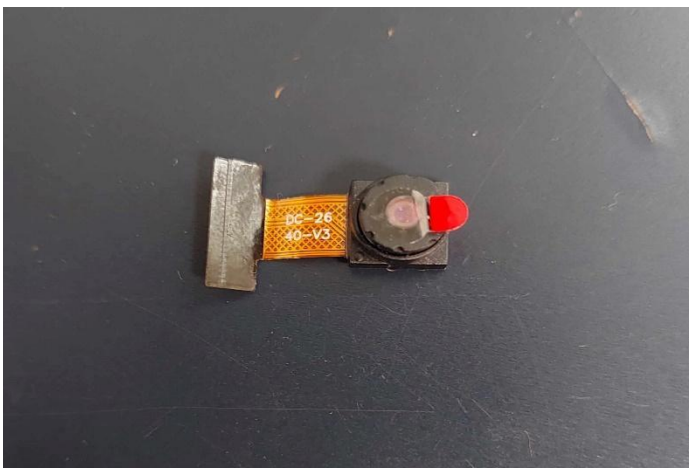
ESP32 board



Picture of how it should be connected

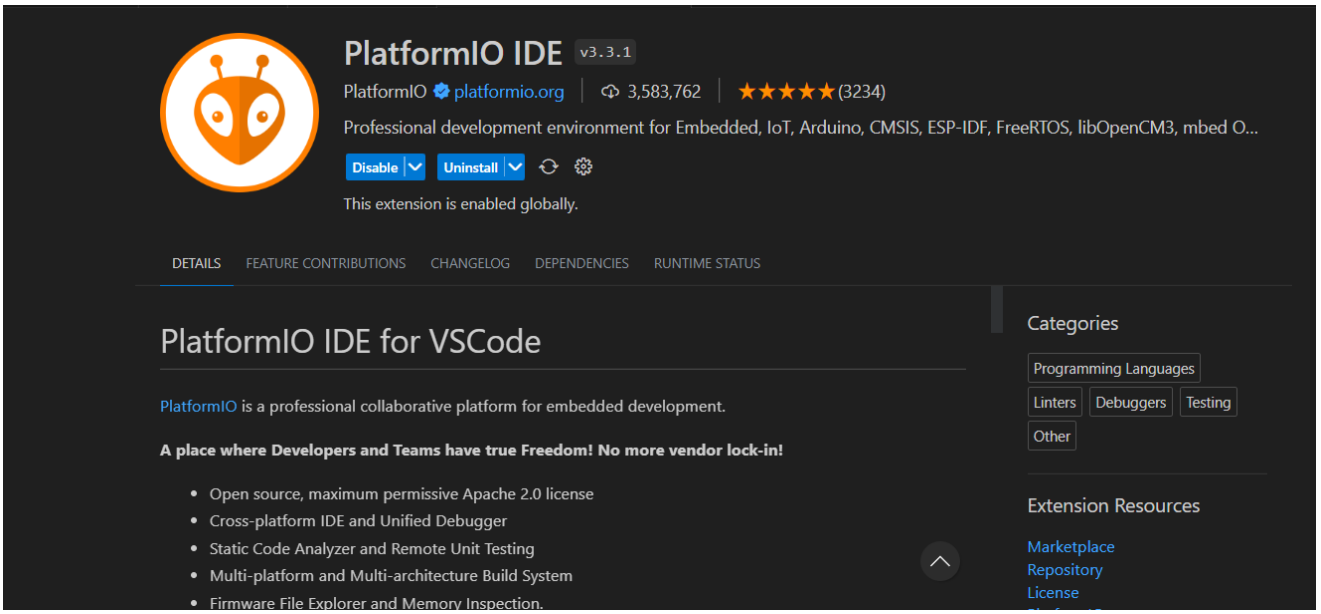


Camera we are going to use



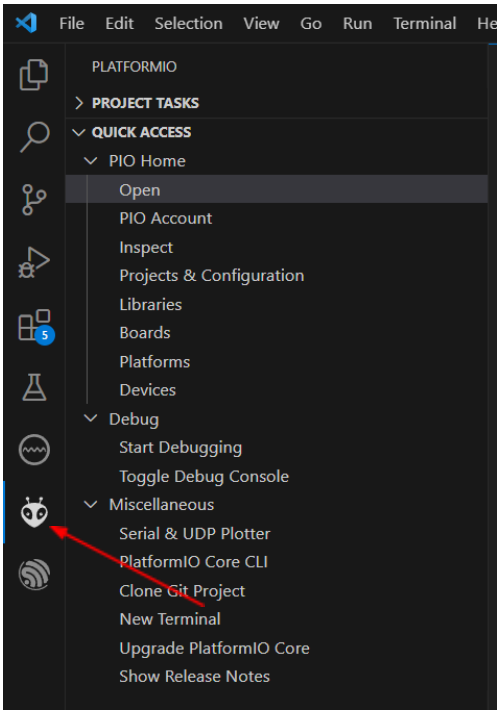
1. Software

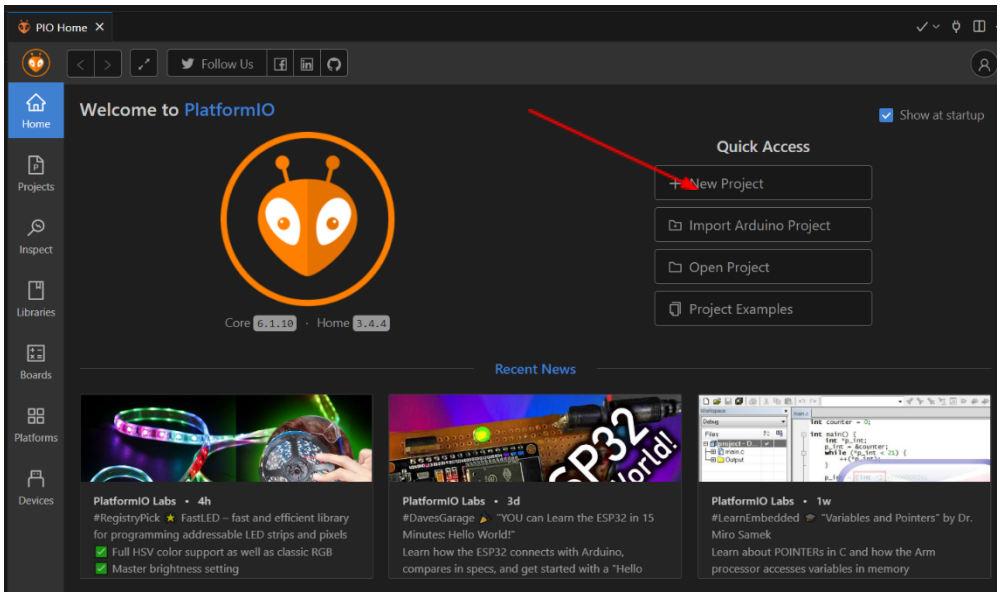
If we're planning to work with the ESP32, it's important to install the necessary extension **PlatformIO IDE** in Visual Studio Code.



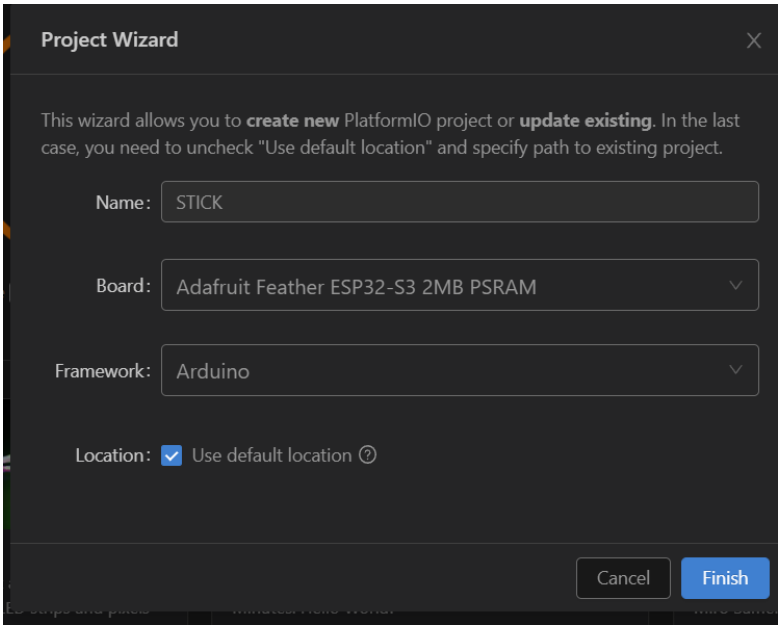
2. Create a project

After installing and restarting VScode you should see in the bar this icon in a picture and click on ‘Open’ create a project.



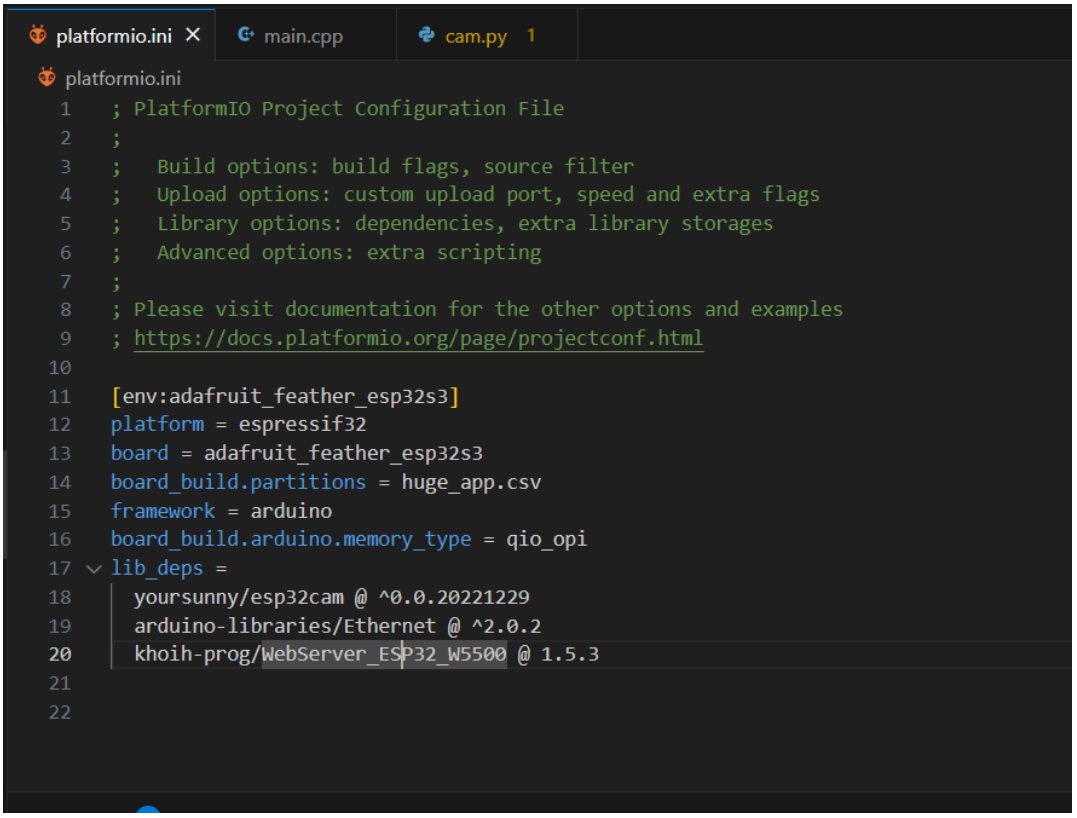


Choose this board below in picture and framework Arduino



3. PlatformIO file

Firstly, you'll need to define these lines in the platformio.ini file, it should look like this.



4. C++ file

After creating the C++ file, you need to include these libraries and define the Ethernet log level:

```
#define _ETHERNET_WEBSERVER_LOGLEVEL_       0
#include <Arduino.h>
#include "SPI.h"
#include <esp32cam.h>
```

Define ethernet port:

```
//If ESP32 is not defined it will show error
#if !( defined(ESP32) )
  #error This code is designed for (ESP32 + W5500) to run on ESP32 platform! Please check your Tools->Board setting.
#endif

#define DEBUG_ETHERNET_WEBSERVER_PORT       Serial0
```

Here are the predefined values for configuring communication with SPI peripherals.

```
// Optional values to override default settings
// Don't change unless you know what you're doing
#define ETH_SPI_HOST        SPI2_HOST
#define SPI_CLOCK_MHZ       25

// Must connect INT to GPIOxx or not working
#define INT_GPIO            38

#define MISO_GPIO           37
#define MOSI_GPIO           35
#define SCK_GPIO            36
#define CS_GPIO             39
```

You need to include the web server library for the ESP32 if you intend to send captured pictures to the web.

```
#include <WebServer_ESP32_W5500.h>

WebServer server(80);

// Enter a MAC address and IP address for your controller below.
#define NUMBER_OF_MAC      20
```

Here, you have defined the local IP (myIP), gateway (myGW), and subnet mask(mySN).

```
// Select the IP address according to your local network
IPAddress myIP(192, 168, 2, 232);
IPAddress myGW(192, 168, 2, 1);
IPAddress mySN(255, 255, 255, 0);

// Google DNS Server IP
IPAddress myDNS(8, 8, 8, 8);
```

In this code, these MAC addresses are used to configure the network device for communication with the W5500 module on the ESP32 platform.

```
byte mac[][NUMBER_OF_MAC] =
{
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x01 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x02 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x03 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x04 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x05 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x06 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x07 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x08 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x09 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x0A },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x0B },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x0C },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x0D },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x0E },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x0F },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x10 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x11 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x12 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0x13 },
  { 0xDE, 0xAD, 0xBE, 0xEF, 0xBE, 0x14 },
};
```

The 'serverJpg' function is to handle requests for sending an image (JPEG) from the camera through the web server. The 'esp32cam::capture()' function is responsible for capturing the current frame from the camera, storing it in the 'frame' variable. If the snapshot is successfully captured, the HTTP response header is configured.

The content length is set to match the size of the image, and an HTTP response with the status code 200 (OK) and 'image/jpeg' is transmitted. The actual frame content is sent to the client using the 'frame->writeTo(client)' call. This action transmits the binary data of the frame to the connected client.

Finally, the measurements of time are output through the serial port (Serial0), allowing the monitoring of durations for different parts of the process

```
void serveJpg()
{
  uint32_t t0=millis();
  auto frame = esp32cam::capture();
  uint32_t t1=millis();
  if (frame == nullptr) {
    Serial0.println("CAPTURE FAIL");
    server.send(503, "", "");
    return;
  }
  server.setContentLength(frame->size());
  server.send(200, "image/jpeg");
  WiFiClient client = server.client();
  uint32_t t2=millis();
  frame->writeTo(client);
  uint32_t t3=millis();
  Serial0.printf("capt %lu,send %lu,write %lu\n\r",t1-t0,t2-t1,t3-t2);

}
```

The function initiates by printing a message to the serial port (via Serial0.printf), indicating the reception of a request for an image from the camera.

Next, the 'esp32cam::Camera.changeResolution()' function is requested to modify the camera's resolution to a setting (320x240 pixels). In the event that this attempt to change the resolution is unsuccessful (resulting in a 'false' return value), the 'serveJpg()' function is called upon to manage the process of sending the image.

```
void handleJpgLo()
{
  Serial0.printf("cam request\n\r");
  if (!esp32cam::Camera.changeResolution(esp32cam::Resolution::find(320, 240))) {
    Serial0.println("SET-LO-RES FAIL");
  }

  serveJpg();
}
```

Here are defined pins:

```
namespace my_pins {
    constexpr esp32cam::Pins MyESP32CAM{
      D0: 11,
      D1: 9,
      D2: 8,
      D3: 10,
      D4: 12,
      D5: 18,
      D6: 17,
      D7: 16,
      XCLK: 15,
      PCLK: 13,
      VSYNC: 6,
      HREF: 7,
      SDA: 4,
      SCL: 5,
      RESET: -1,
      PWDN: -1,
    };
} // namespace my_pins
```

Descriptions of individual lines of code in void setup:

```
void setup()
{
  // Initializes serial communication at 115200 baud rate.
  Serial0.begin(115200);
  // Configuration object for the ESP32-CAM camera module
  esp32cam::Config cfg;
  // Sets the pins (pinout) for the camera module according to predefined settings
  cfg.setPins(my_pins::MyESP32CAM);
  // Sets the camera resolution to 320x240 pixels.
  cfg.setResolution(esp32cam::Resolution::find(320, 240));
  // Sets the number of buffers for the camera to 2. This means that there will be 2 memory blocks ready for storing image data.
  cfg.setBufferCount(2);
  // Sets JPEG compression quality to 80%.
  cfg.setJpeg(80);

  /* Starts the camera operation with the settings defined in the cfg configuration object.
  The result of the operation (success or failure) is stored in the ok variable.*/
  bool ok = esp32cam::Camera.begin(cfg);
  delay(2000);
  // Print the message whether the initialization of the camera was successful or failed.
  Serial0.println(ok ? "CAMERA OK" : "CAMERA FAIL");

  /* This is the part of the code that waits until the Serial0 port is initialized
  or until 5000 milliseconds (5 seconds) have passed.
  This waits for the serial communication to become operational before continuing to process the code further.*/
  while (!Serial0 && (millis() < 5000));

  /* This is about starting the web server and various parameters such as the board type (Arduino board),
  the shield type and the web server library version.*/
  Serial0.print(F("\nStart WebServer on "));
  Serial0.print(ARDUINO_BOARD);
  Serial0.print(F(" with "));
  Serial0.println(SHIELD_TYPE);
  Serial0.println(WEBSERVER_ESP32_W5500_VERSION);


  /* Initializes Ethernet communication on the W5500 module.
  The function ETH.begin() sets the communication parameters via SPI and other necessary parameters.
  The function returns a number indicating the success of the initialization,
  which is printed over the serial port using Serial0.printf().*/
  ET_LOGWARN(F("Default SPI pinout:"));
  ET_LOGWARN1(F("SPI_HOST:"), ETH_SPI_HOST);
  ET_LOGWARN1(F("MOSI:"), MOSI_GPIO);
  ET_LOGWARN1(F("MISO:"), MISO_GPIO);
  ET_LOGWARN1(F("SCK:"),  SCK_GPIO);
  ET_LOGWARN1(F("CS:"),   CS_GPIO);
  ET_LOGWARN1(F("INT:"),  INT_GPIO);
  ET_LOGWARN1(F("SPI Clock (MHz):"), SPI_CLOCK_MHZ);
  ET_LOGWARN(F("========================"));
  // To be called before ETH.begin(), configures the connection
  ESP32_W5500_onEvent();

  // start the ethernet connection and the server:
  // Use DHCP dynamic IP and random mac
  Serial0.printf("%d\n\r",ETH.begin( MISO_GPIO, MOSI_GPIO, SCK_GPIO, CS_GPIO, INT_GPIO, SPI_CLOCK_MHZ, ETH_SPI_HOST ));
  pinMode(INT_GPIO,INPUT_PULLUP);

  ESP32_W5500_waitForConnect();

  // start the web server on port 80
  server.on("/cam-lo.jpg", handleJpgLo);
  server.begin();
}
```

Here in loop is called continuously after the setup() function ends. In this case, this code is used to handle clients that connect to the web server and send HTTP requests.

```
void loop()
{
  server.handleClient();
  delay(5);
}
```

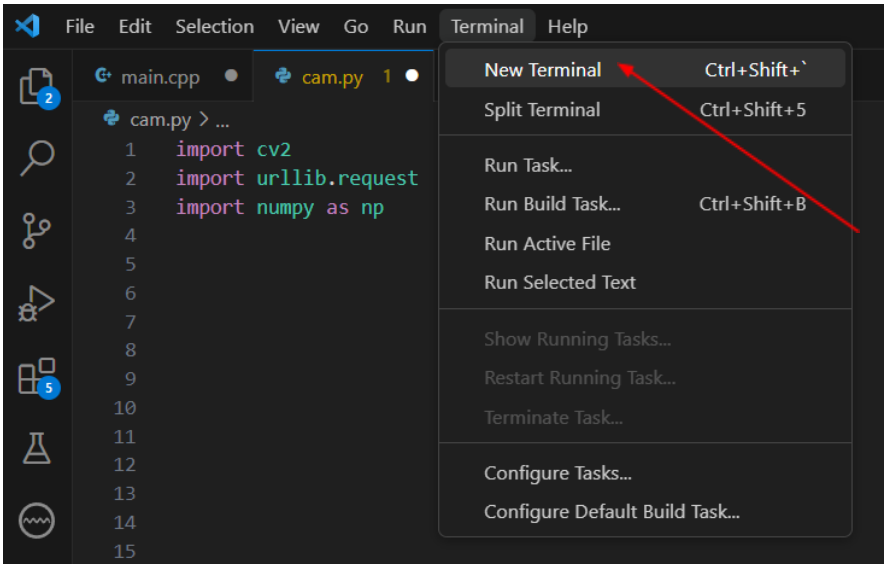
That's all and to upload the C++ code to the board, simply press Ctrl+Alt+U.

5. Python script for camera

To run the web server with the Python script and enable face detection, ensure you include the following libraries: OpenCV, urllib, and NumPy.

```
import cv2
import urllib.request
import numpy as np
```

If you haven't installed OpenCV, run new terminal



and write into terminal this command:

"pip install OpenCV-python"

urllib and NumPy should be downloaded by default

You need to define IP, for IP of board use Arduino IDE

```
#change the IP address below according to the
#IP shown in the Serial monitor of Arduino code
url='http://172.19.11.22/cam-lo.jpg'
```

Now create a window for the live transmission. You're free to choose any name you prefer for the window, but in this code, the window is named "live transmission" and it will automatically adjust its size.

```
cv2.namedWindow("live transmission", cv2.WINDOW_AUTOSIZE)
```

This window is used to display the graphical user interface (GUI), which contains trackbars for setting values. Trackbars are controls that allow the user to change the values of certain parameters in real time.

```
cv2.namedWindow("Tracking")
cv2.createTrackbar("LH", "Tracking", 0, 255, nothing)
cv2.createTrackbar("LS", "Tracking", 0, 255, nothing)
cv2.createTrackbar("LV", "Tracking", 0, 255, nothing)
cv2.createTrackbar("UH", "Tracking", 255, 255, nothing)
cv2.createTrackbar("US", "Tracking", 255, 255, nothing)
cv2.createTrackbar("UV", "Tracking", 255, 255, nothing)
```

Six trackbars, each with a name and initial value:

"LH" (Lower Hue)

"LS" (Lower Saturation)

"LV" (Lower Value)

"UH" (Upper Hue)

"US" (Upper Saturation)

"UV" (Upper Value)

This line of code is responsible for face detection and use an XML file for recognizing faces(Keep in mind that face detection is not accurate and errors might arise during the detection process).

The link for haarcascade_frontalface_default.xml file:

https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
```

Main loop and descriptions in code:

```
while True:
    # for opening URL link
    img_resp=urllib.request.urlopen(url)

    # Using an 8-bit number and the NumPy library, the result is an array of numbers that represents the image data.
    imgnp=np.array(bytearray(img_resp.read()),dtype=np.uint8)

    # Used to decode image data from imgnp array of numbers.
    # The -1 parameter means that OpenCV should automatically determine the appropriate decoding format based on the data.
    # The result is a frame that can be displayed or processed using OpenCV functions.
    frame=cv2.imdecode(imgnp,-1)

    # This line converts the frame from the BGR (Blue-Green-Red) color space to the HSV (Hue-Saturation-Value) color space.
    # The HSV color space separates information about hue, saturation and value of colors.
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # This line converts a frame from the BGR color space to grayscale.
    # Grayscale is a simple way of representing color where each pixel has only one value that indicates its intensity of gray.
    # This conversion is often used primarily for simple image processing operations such as edge detection.
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # This line uses an instance of the face_cascade classifier to detect objects in the image.
    # Face detection is used here (face_cascade was created based on the haarcascade_frontalface_default).
    # The detectMultiScale method looks for areas in the image that might contain objects (in this case faces)
    # and returns a list of detected areas with their position and dimensions.
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

    #It goes through the list of detected faces and if a face is detected, a rectangle is made around it
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)

    # get the set values of the trackbars and use them to define the range of colors that will filter the image.
    l_h = cv2.getTrackbarPos("LH", "Tracking")
    l_s = cv2.getTrackbarPos("LS", "Tracking")
    l_v = cv2.getTrackbarPos("LV", "Tracking")
    u_h = cv2.getTrackbarPos("UH", "Tracking")
    u_s = cv2.getTrackbarPos("US", "Tracking")
    u_v = cv2.getTrackbarPos("UV", "Tracking")

    l_b = np.array([l_h, l_s, l_v])
    u_b = np.array([u_h, u_s, u_v])

    # This line creates a binary mask that indicates which pixels in the image fall within the specified color range
    mask = cv2.inRange(hsv, l_b, u_b)

    # This line creates a new image that is the result of applying the logical operator "AND" between the original frame and the mask mask.
    # This means that only pixels that are white in the mask (pixels in the color range) will be included in the new image. Other pixels will be black.
    # The result is an image that displays only the parts of the original image that match the given color range.
    res = cv2.bitwise_and(frame, frame, mask=mask)

    # Launch a 3 windows
    cv2.imshow("live transmission", frame)
    cv2.imshow("mask", mask)
    cv2.imshow("res", res)
    #It waits for click on the keyboard letter 'q', if it's clicked it's going to break while loop
    key=cv2.waitKey(5)
    if cv2.waitKey(10) & 0xFF == ord('q'):
        break

cv2.destroyAllWindows()
```