STM32F407 Discovery Kit with ESP32 and MQTT

Contents

ESP32 microcontroller

This project follows from the `tft_lcd` project, completed previously. The `tft_lcd` project printed temperatures from a thermistor to an TFT LCD touchscreen, used buttons on the touchscreen to toggle LEDs on the Discovery board, and used the blue user button on the Discovery board to turn the image of a light bulb on the LCD screen on and off.

In the current project, the temperature from the thermistor will be posted to the cloud over Wi-Fi, using an ESP32-C3 microcontroller development board. In addition, the state of a remote button, recorded in the cloud, will be used to turn the image of the light bulb on the LCD screen on and off.

The ESP32 is a single core RISC microcontroller. It has fewer and lower quality peripherals than the STM32 family, but is also less expensive. The ESP32 is of interest in this project because, unlike the STM32 family, it offers Wi-Fi and Bluetooth connectivity.

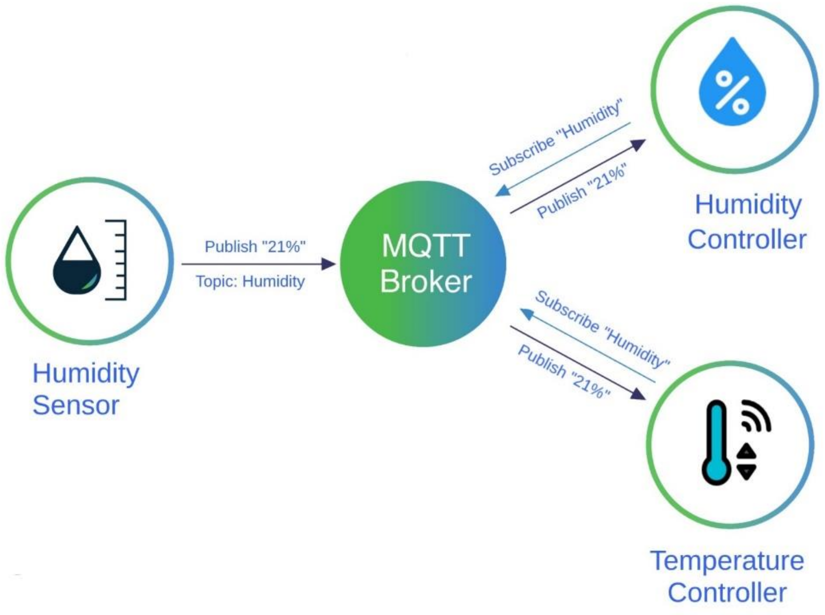
Since this project will extend the capabilities of the `tft_lcd` project, follow the directions in Appendix 1 to make a copy of the `tft_lcd` project and call it `tft_lcd_mqtt`.

MQTT communication protocol

MQTT originally stood for Message Queuing Telemetry Transport, but these days MQTT is just MQTT. It is a simple machine-to-machine communication protocol that is "lightweight" in the sense that it requires little bandwidth and is suited to low power and limited processing scenarios.

Under MQTT, a client connects to an MQTT broker. The client can subscribe to messages on a topic, or publish messages on a topic, or both. When the MQTT broker receives a published message, it forwards the message on to clients that are subscribed to the topic of the message. The publishers and subscribers are not directly connected and, since the broker exists in the cloud, can be physically far apart.

In the figure a sensor publishes a humidity sensor reading. Two other devices subscribe to, and can react to, this topic. Multiple clients can publish to an MQTT broker; the broker is implemented as a cluster so that the failure of one point in the broker will not disturb communications between the publishers and subscribers. Topics are expressed in the form myhome/bathroom/humidity, or factory1/temperature.



There are many MQTT brokers and most make a free public version available. In this project you will use HiveMQ.

Note: From Bhutan I had trouble connecting to the HiveMQ public broker. I provide alternatives below.

JSON data exchange format

JavaScript Object Notation (JSON) is a lightweight data exchange format expressed in human-readable text form. Here is an example of a JSON object. Objects are enclosed by { }, arrays are enclosed by [ ], and each data item has a name and a value written in the form “name”: “value”.

```
{
"employees": [
{"firstName": "John", "lastName": "Doe"},
{"firstName": "Anna", "lastName": "Smith"},
{"firstName": "Peter", "lastName": "Jones"}
]
}
```

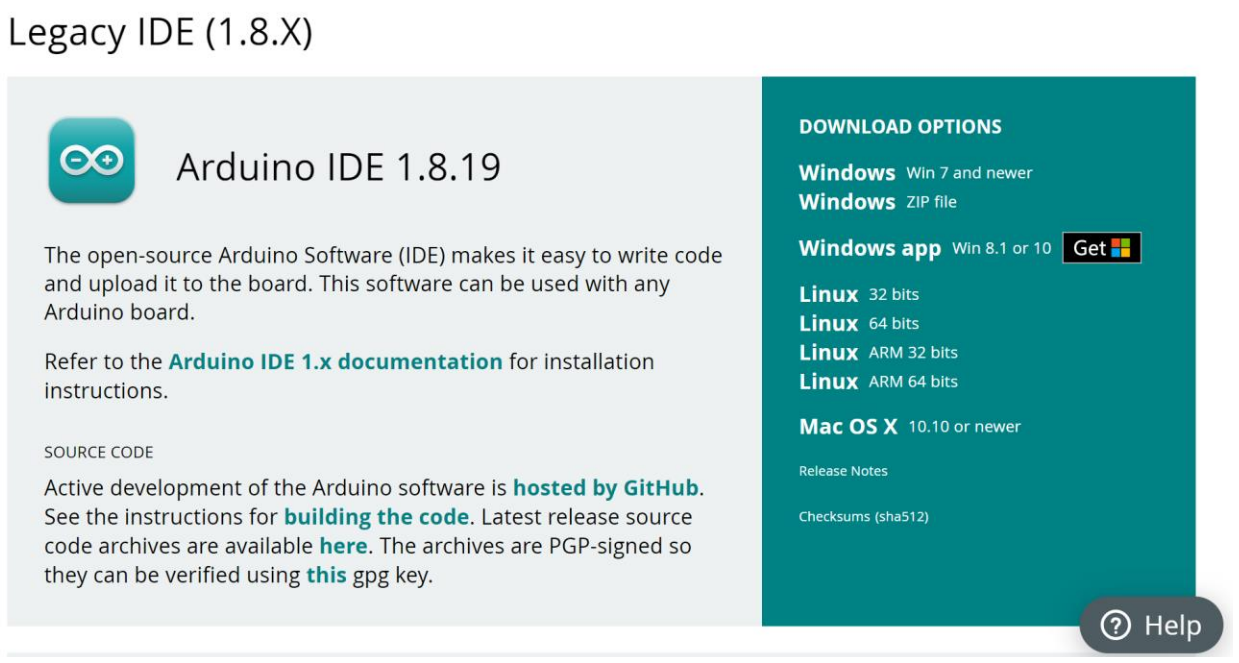
The JSON format is suitable to many platforms and computer languages, and is widely used. Its simple structure means it is efficient for the device-to-device communication for the Internet of Things (IoT).

Integrated development environment for ESP32

Multiple development environments are available for the ESP32, indeed ESP-IDF is the official integrated development framework for ESP. In this project, you will use the Arduino integrated development environment (IDE) to interact with the ESP32-C3.

Arduino IDE 1 offers portable installation, which keeps all the relevant files together in one place so that you can copy your projects with all dependencies intact. Arduino IDE 2 does not (currently) offer this option.

From Software | Arduino, download the Windows ZIP file.



Legacy IDE (1.8.X)

Arduino IDE 1.8.19

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. This software can be used with any Arduino board.

Refer to the **Arduino IDE 1.x documentation** for installation instructions.

SOURCE CODE

Active development of the Arduino software is **hosted by GitHub**. See the instructions for **building the code**. Latest release source code archives are available **here**. The archives are PGP-signed so they can be verified using **this** gpg key.

DOWNLOAD OPTIONS

Windows Win 7 and newer
Windows ZIP file
Windows app Win 8.1 or 10 Get
Linux 32 bits
Linux 64 bits
Linux ARM 32 bits
Linux ARM 64 bits
Mac OS X 10.10 or newer

Release Notes

Checksums (sha512)

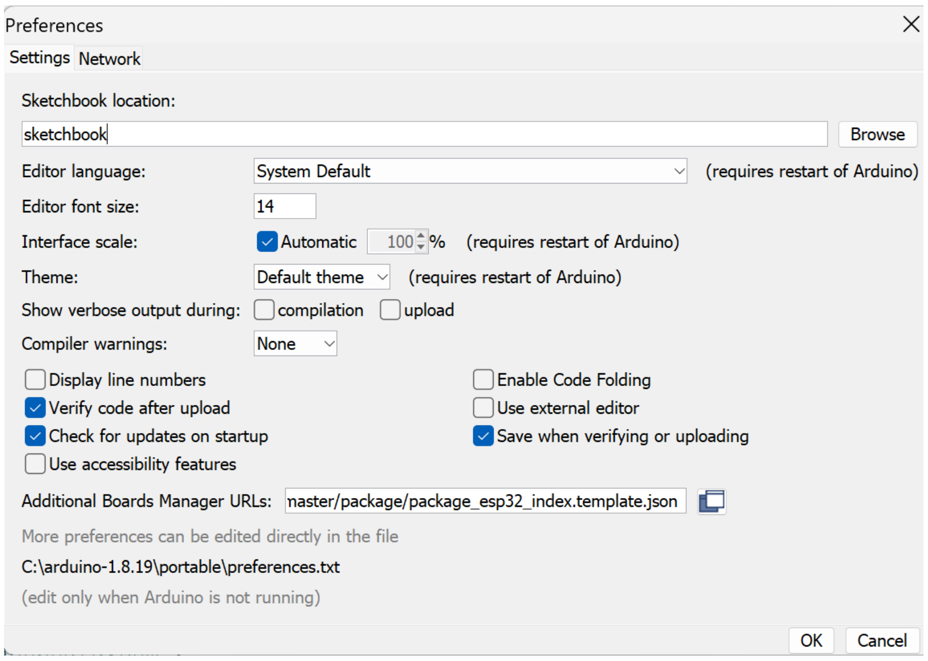
As described at Arduino IDE 1 Portable Installation | Arduino Documentation, extract the ZIP file to the C:/ root directory. It is an advantage to keep your Arduino close to the root to keep pathnames relatively short (long pathnames lead to a configuration error).

Before running `arduino.exe`, create a `/portable` directory inside your Arduino folder. From this point on, when you create and use Arduino sketches, they will be stored in `/portable/sketchbook`.

Run `arduino.exe`.

Under File – Preferences, enter the following ESP32 json link in the Additional Boards Manager URLs, which

https://raw.githubusercontent.com/espressif/arduino-esp32/refs/heads/master/package/package_esp32_index.template.json



Sketchbook location: sketchbook

Editor language: System Default (requires restart of Arduino)

Editor font size: 14

Interface scale: [✓] Automatic 100 % (requires restart of Arduino)

Theme: Default theme (requires restart of Arduino)

Show verbose output during: [ ] compilation [ ] upload

Compiler warnings: None

[ ] Display line numbers
[✓] Verify code after upload
[✓] Check for updates on startup
[ ] Use accessibility features
[ ] Enable Code Folding
[ ] Use external editor
[✓] Save when verifying or uploading

Additional Boards Manager URLs: naster/package/package_esp32_index.template.json

More preferences can be edited directly in the file

C:\arduino-1.8.19\portable\preferences.txt

(edit only when Arduino is not running)

The ESP32 json file is provided at https://github.com/espressif/arduino-esp32/tree/master/package and provides links to ESP32 libraries.

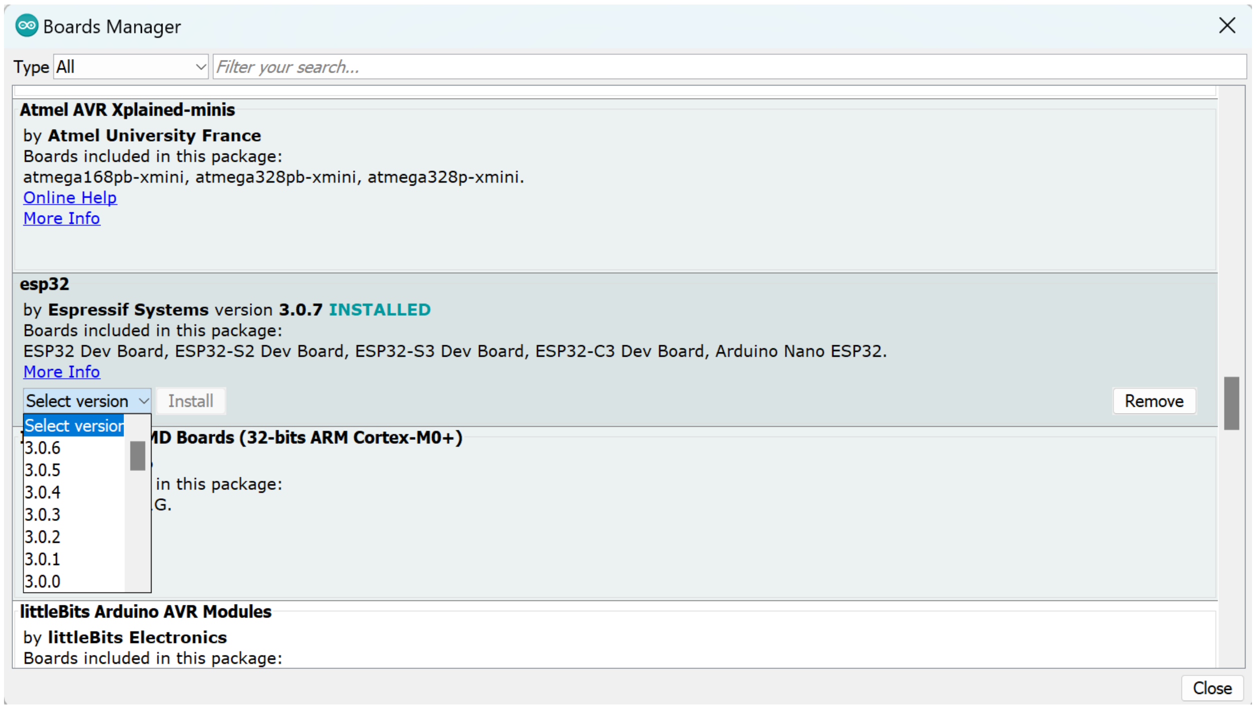
Click on Tools – Port. A COM port should have a checkmark beside it.

If Port is greyed out, your PC may need an additional driver to communicate with the ESP32. When you try to run a sketch, you may get the error “serial port not connected.” If these situations apply to you, consult Appendix 2.

The specific ESP32 device you will be using is the ESP32-C3, with details available at this link:

ESP32-C3-DevKitC-02 - ESP32-C3 - — esp-dev-kits latest documentation

To prepare the IDE with the architecture details of the ESP32-C3, go to Tools – Board – Boards Manager, and install the latest version of esp32.


Boards Manager

Type All | Filter your search...

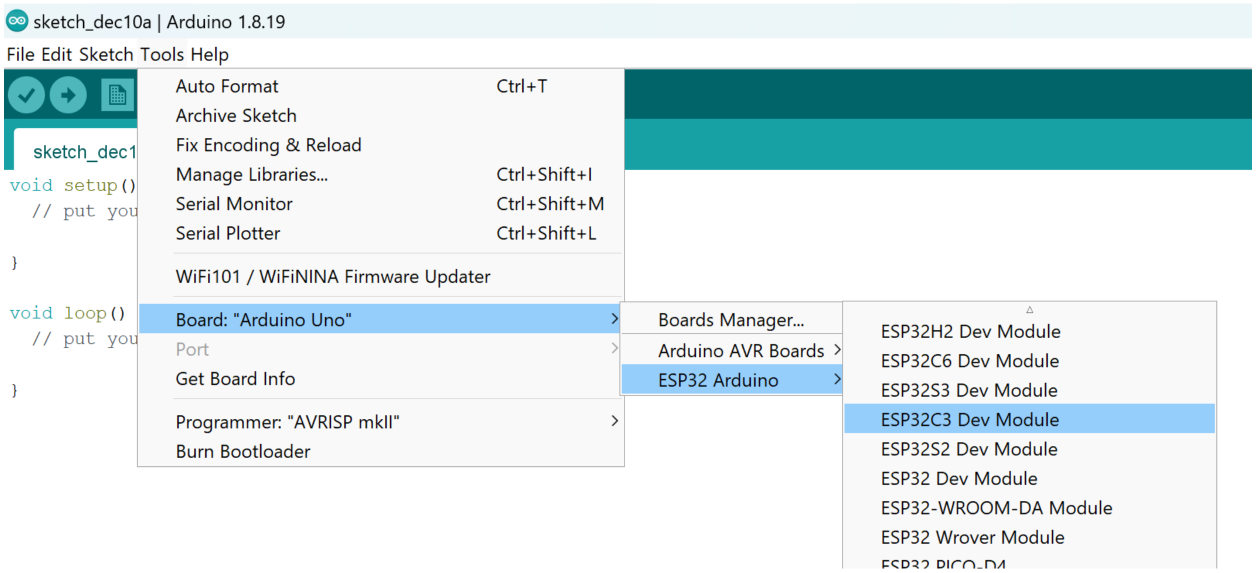
Atmel AVR Xplained-minis
by **Atmel University France**
Boards included in this package:
atmega168pb-xmini, atmega328pb-xmini, atmega328p-xmini.
Online Help
More Info

esp32
by **Espressif Systems** version **3.0.7** **INSTALLED**
Boards included in this package:
ESP32 Dev Board, ESP32-S2 Dev Board, ESP32-S3 Dev Board, ESP32-C3 Dev Board, Arduino Nano ESP32.
More Info

D Boards (32-bits ARM Cortex-M0+)
in this package:
G.

littleBits Arduino AVR Modules
by **littleBits Electronics**
Boards included in this package:

Next, go to Tools – Board – ESP32 Arduino – ESP32C3 Dev Module.


sketch_dec10a | Arduino 1.8.19

```
void setup() {
  // put you

}

void loop() {
  // put you

}
```

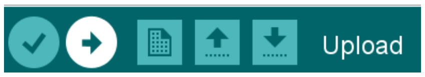
RGB LED on ESP32

The ESP32-C3 has an RGB LED onboard. This kind of LED can be any colour and any brightness. RGB values must lie between 0 and 255.

Go to File – New to create a new sketch.

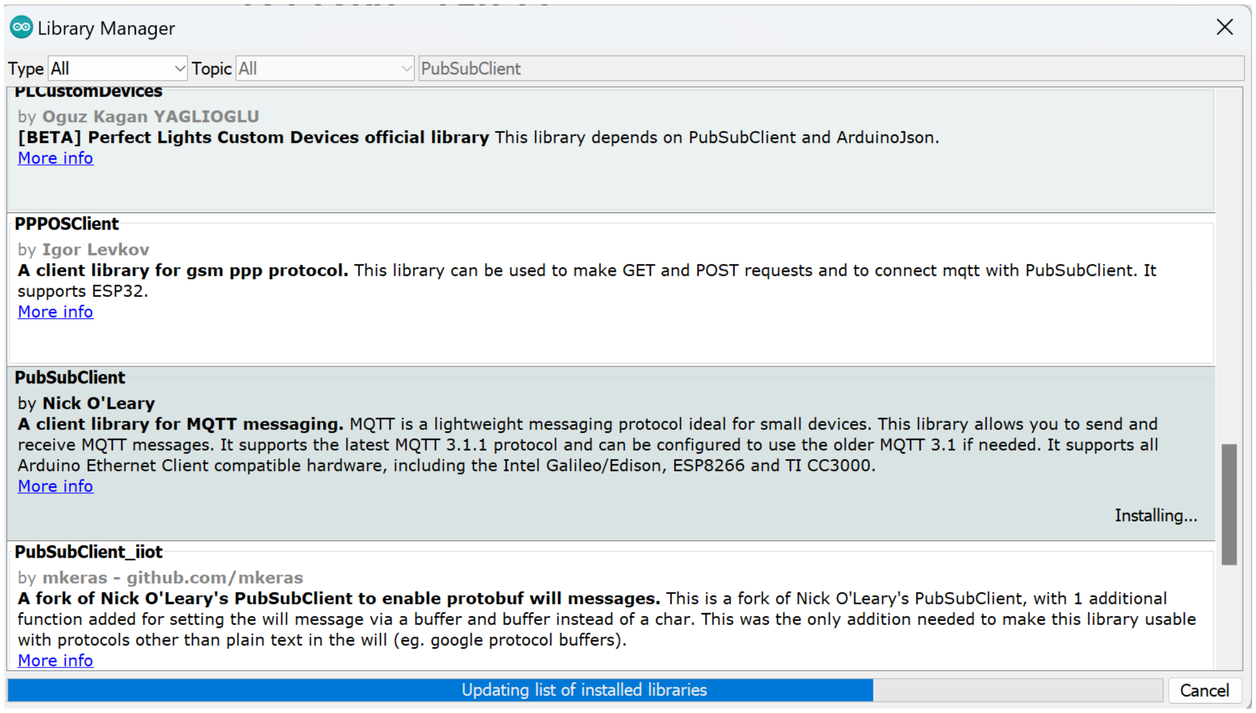
```
uint8_t bright, red, green, blue;

void setup() {
  // put your setup code here, to run once:
    bright = 40;
    red = 0;
    green = 50;
    blue = 200;
    rgbLedWrite(RGB_BUILTIN, (red*bright/255)&255,
(green*bright/255)&255, (blue*bright/255)&255);
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Click on the arrow to compile and upload. Verify that the LED on your ESP32-C3 has the expected colour and brightness. Experiment with other values.



ESP32 MQTT Subscribe

Two libraries are needed for MQTT and JSON functionality. Go to Tools – Manage Libraries. Install PubSubClient by Nick O'Leary and also Arduino Json by Benoit Blanchon.

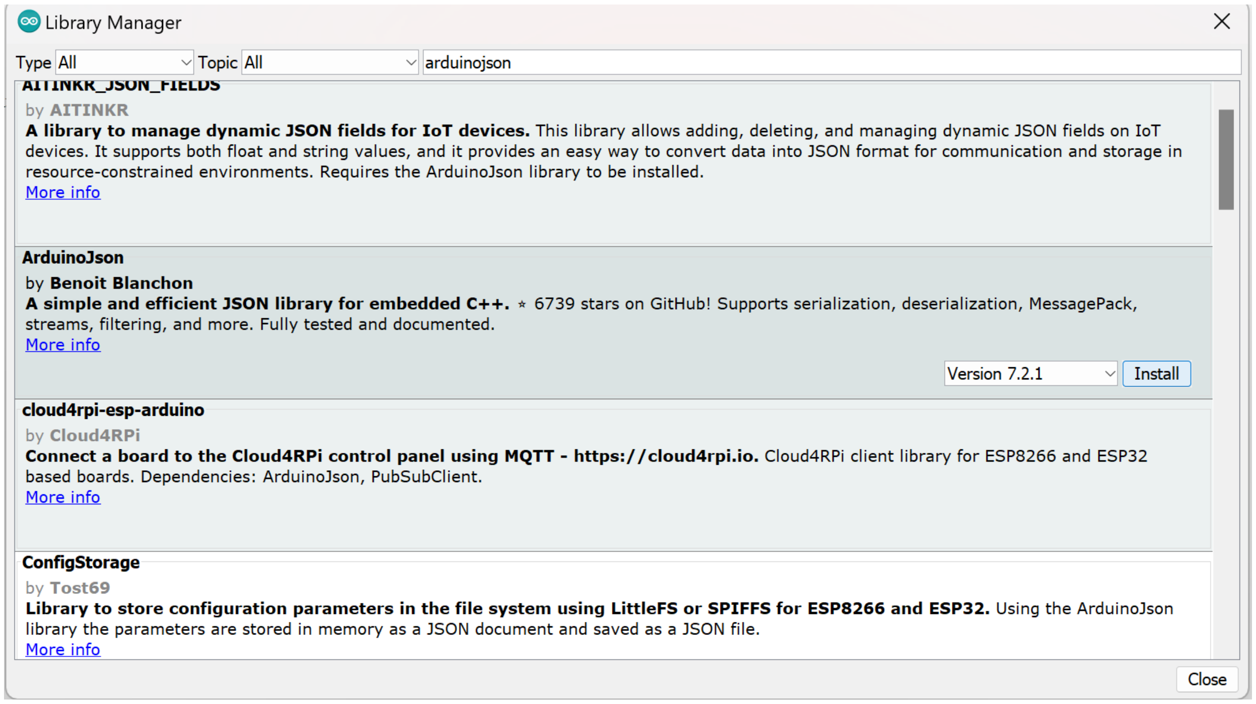


PLCustomDevices
by Oguz Kagan YAGLIOGLU
[BETA] Perfect Lights Custom Devices official library This library depends on PubSubClient and ArduinoJson.
More info

PPPOSClient
by Igor Levkov
A client library for gsm ppp protocol. This library can be used to make GET and POST requests and to connect mqtt with PubSubClient. It supports ESP32.
More info

PubSubClient
by Nick O'Leary
A client library for MQTT messaging. MQTT is a lightweight messaging protocol ideal for small devices. This library allows you to send and receive MQTT messages. It supports the latest MQTT 3.1.1 protocol and can be configured to use the older MQTT 3.1 if needed. It supports all Arduino Ethernet Client compatible hardware, including the Intel Galileo/Edison, ESP8266 and TI CC3000.
More info
Installing...

PubSubClient_iiot
by mkeras - github.com/mkeras
A fork of Nick O'Leary's PubSubClient to enable protobuf will messages. This is a fork of Nick O'Leary's PubSubClient, with 1 additional function added for setting the will message via a buffer and buffer instead of a char. This was the only addition needed to make this library usable with protocols other than plain text in the will (eg. google protocol buffers).
More info



AITINKR_JSON_FIELDS
by AITINKR
A library to manage dynamic JSON fields for IoT devices. This library allows adding, deleting, and managing dynamic JSON fields on IoT devices. It supports both float and string values, and it provides an easy way to convert data into JSON format for communication and storage in resource-constrained environments. Requires the ArduinoJson library to be installed.
More info

ArduinoJson
by Benoit Blanchon
A simple and efficient JSON library for embedded C++. ⭑ 6739 stars on GitHub! Supports serialization, deserialization, MessagePack, streams, filtering, and more. Fully tested and documented.
More info
Version 7.2.1 Install

cloud4rpi-esp-arduino
by Cloud4RPi
Connect a board to the Cloud4RPi control panel using MQTT - https://cloud4rpi.io. Cloud4RPi client library for ESP8266 and ESP32 based boards. Dependencies: ArduinoJson, PubSubClient.
More info

ConfigStorage
by Tost69
Library to store configuration parameters in the file system using LittleFS or SPIFFS for ESP8266 and ESP32. Using the ArduinoJson library the parameters are stored in memory as a JSON document and saved as a JSON file.
More info

The program used to subscribe to JSON messages for control of the RGB LED on the ESP32-C3 development board is listed below.

The `setup()` function connects to WiFi, identifies the MQTT server that will be used, and announces the name of the callback function that will be called when a message is received from the MQTT server, in this case `callback()`.

The `client` referred in `loop()` is the MQTT client connection. The `reconnect()` function is called to make a connection to the MQTT server, where the MQTT broker resides. Within the `reconnect()` function, the line

```
client.subscribe("esp32c3/yourname/led");
```

subscribes to all messages that have the topic `esp32c3/yourname/led`. As suggested by the comment, the legal format of a message for the LED is:

```
{"bright": 202, "red": 34, "green": 68, "blue": 102}
```

Note that this string is a JSON object.

When the MQTT broker receives a message on the topic `esp32c3/yourname/led`, the message sent to all devices that subscribe to this topic, including the ESP32-C3 running your project.

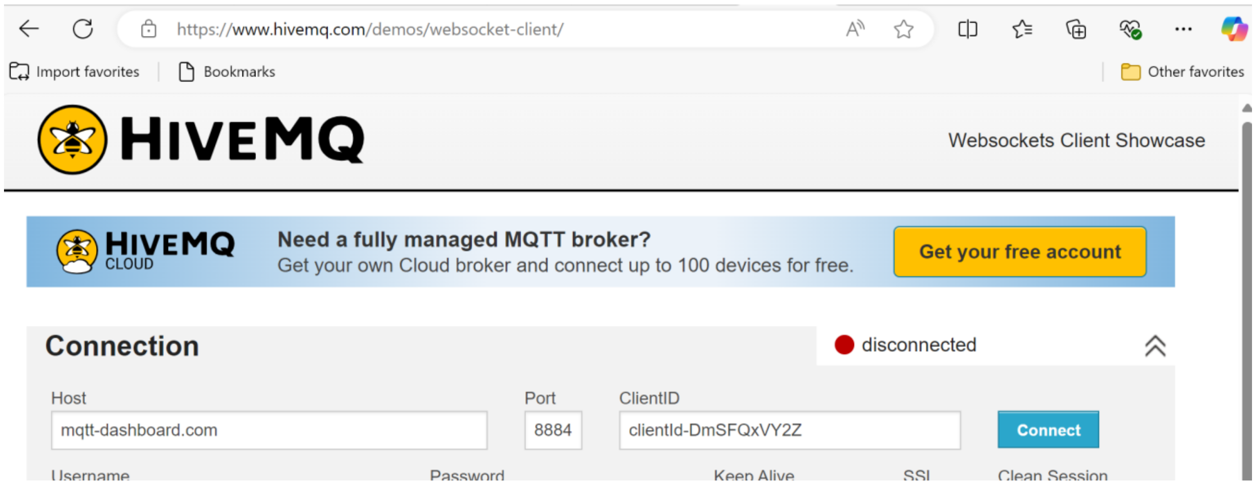
When a message on a subscribed topic is received, the `callback()` function deserializes the JSON object. The values for `bright`, `red`, `green`, and `blue` are extracted from the JSON string, and `ArduinoJson` implicitly converts the value to the expected type. The values of these variables control the colour and brightness of the onboard RGB LED.

```
/*
MQTT subscribe to RGB LED JSON
*/

#include <WiFi.h>
#include <PubSubClient.h>
#include <ArduinoJson.h>

// update ssid and password for your network
const char* ssid = "insert_your_network_name";
const char* password = "insert_your_network_password";
const char* mqtt_server = "broker.mqtt-dashboard.com";
// if problems connecting to hivemq, replace with:
// const char* mqtt_server = "broker.emqx.io";

int bright, red, green, blue;

// allocate the JSON document
JsonDocument doc;
const char* json;
```

```
WiFiClient espClient;
PubSubClient client(espClient);
unsigned long lastMsg = 0;
#define MSG_BUFFER_SIZE    (25)
char msg[MSG_BUFFER_SIZE];

void setup_wifi() {

  delay(10);
  // connect to a WiFi network
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  randomSeed(micros());

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();

  // deserialize the JSON document
  DeserializationError error = deserializeJson(doc, (char *) payload);
  // test if parsing succeeds
  if (error) {
    Serial.print(F("deserializeJson() failed: "));
    Serial.println(error.f_str());
    return;
  }
  bright = doc["bright"];
  red = doc["red"];
  green = doc["green"];
  blue = doc["blue"];
  rgbLedWrite(RGB_BUILTIN, (red*bright/255)&255,
(green*bright/255)&255, (blue*bright/255)&255);
}

void reconnect() {
  // loop until reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection ...");
    // create a random client ID
    String clientId = "ESP32Client-";
    clientId += String(random(0xffff), HEX);
    // attempt to connect
    if (client.connect(clientId.c_str())) {
      Serial.println(" connected");
      // resubscribe
      // sample message: {"bright": 202, "red": 34, "green": 68,
"blue": 102 }
      client.subscribe("esp32c3/yourname/led");
    } else {
      Serial.print("failed: client state =");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}

void loop() {

  if (!client.connected()) {
    reconnect();
  }
  client.loop();

}
```

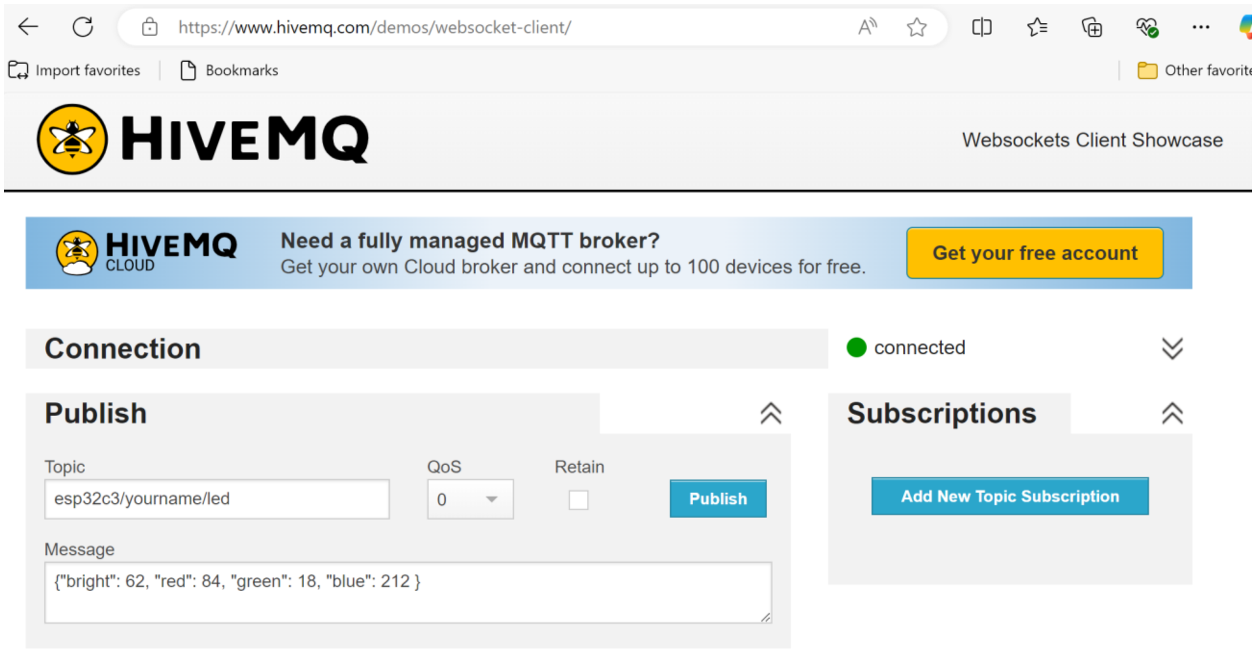
Save your code, e.g. mqtt_json, and click on the arrow in the Arduino IDE to compile and upload your program. Leave the program running.

For this project, you will use the HiveMQ MQTT broker to publish messages on the topic `esp32c3/yourname/led`. Go to [MQTT Websocket Client]() and click Connect. Note that the ClientID shown on the HiveMQ site need not match the (random) clientID generated by your ESP32 Arduino sketch. Any client who publishes on the topic `esp32c3/yourname/led` (with the right variable names) will be able to control the RGB LED on your board, and any client who subscribes to the topic `esp32c3/yourname/led` will receive messages you publish from HiveMQ.


https://www.hivemq.com/demos/websocket-client/
Import favorites
Bookmarks
Other favorites
HIVEMQ
Websockets Client Showcase
HIVEMQ CLOUD
Need a fully managed MQTT broker?
Get your own Cloud broker and connect up to 100 devices for free.
Get your free account
Connection
disconnected
Connect
Username
Password
Keep Alive
SSL
Clean Session

Host: mqtt-dashboard.com
Port: 8884
ClientID: clientId-DmSFQxVY2Z

Once you are connected, type in the name of the topic on which you wish to publish messages. Then, type in a message of the correct form.


https://www.hivemq.com/demos/websocket-client/
Import favorites
Bookmarks
Other favorite
HIVEMQ
Websockets Client Showcase
HIVEMQ CLOUD
Need a fully managed MQTT broker?
Get your own Cloud broker and connect up to 100 devices for free.
Get your free account
Connection
connected
Publish
Subscriptions
Publish
Add New Topic Subscription

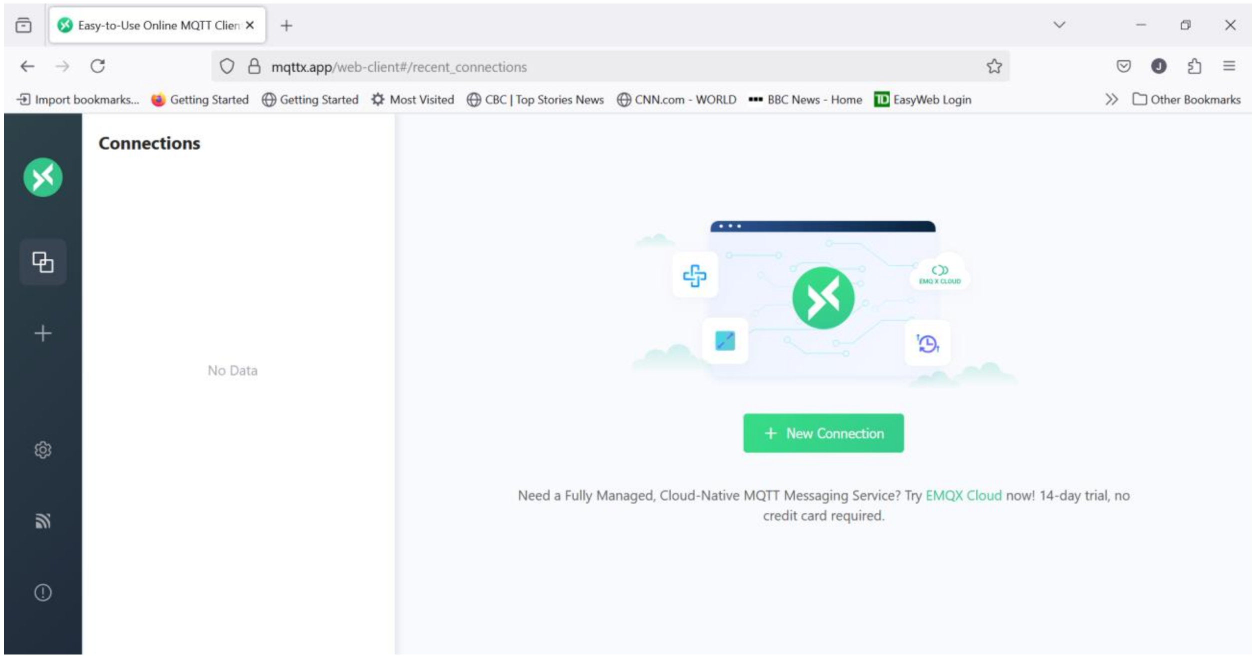
Topic: esp32c3/yourname/led
QoS: 0
Retain: ☐
Message: {"bright": 62, "red": 84, "green": 18, "blue": 212 }

Click Publish. Does your LED show the expected colour? Change the brightness and/or RGB values and Publish again. If you change the publishing topic, or send a message in an unexpected format, the LED will not respond.

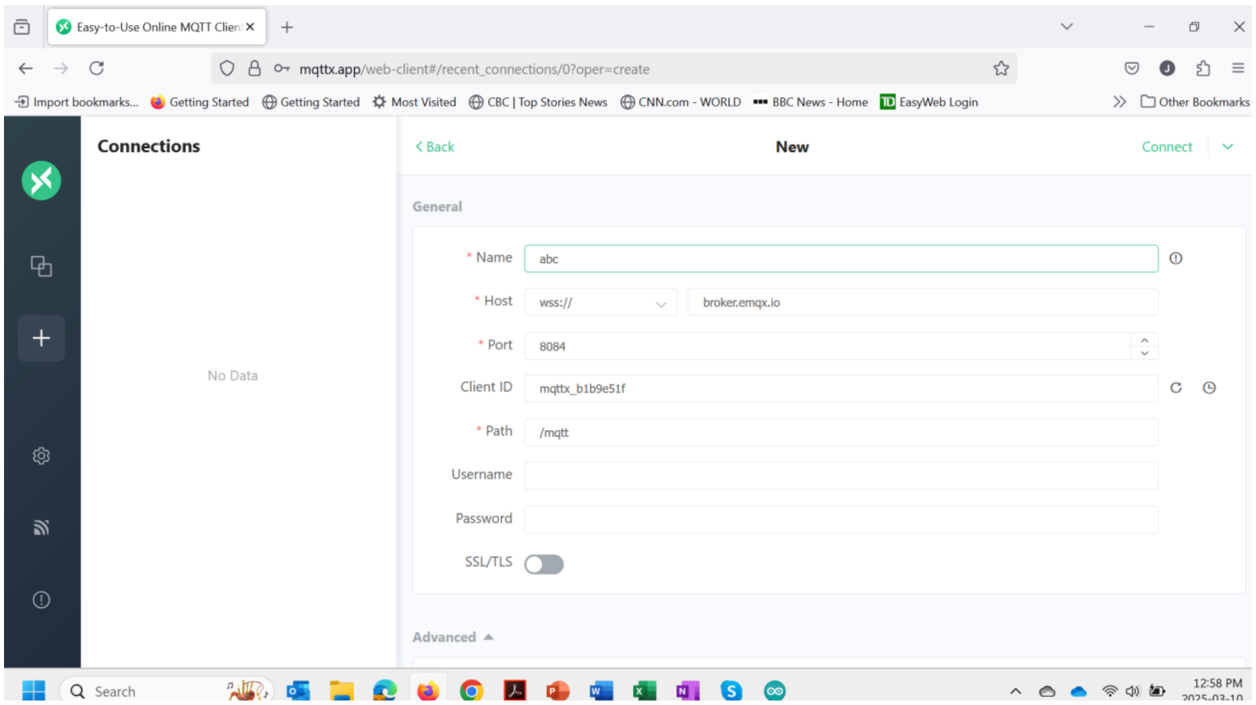
In the Arduino IDE, you can click on Tools – Serial Monitor to view the messages as they arrive.

Alternative MQTT broker

If you had trouble with HiveMQ and used `const char* mqtt_server = "broker.emqx.io";` in your code above, you can connect using https://mqttx.app/web-client#/. Click New Connection.


Easy-to-Use Online MQTT Clien
mqttx.app/web-client#/recent_connections
Import bookmarks... Getting Started Getting Started Most Visited CBC | Top Stories News CNN.com - WORLD BBC News - Home EasyWeb Login
Other Bookmarks
Connections
No Data
New Connection
Need a Fully Managed, Cloud-Native MQTT Messaging Service? Try EMQX Cloud now! 14-day trial, no credit card required.

Type any Name you like, and click Connect:


Easy-to-Use Online MQTT Client

mqttx.app/web-client#/recent_connections/0?oper=create

Connections

No Data

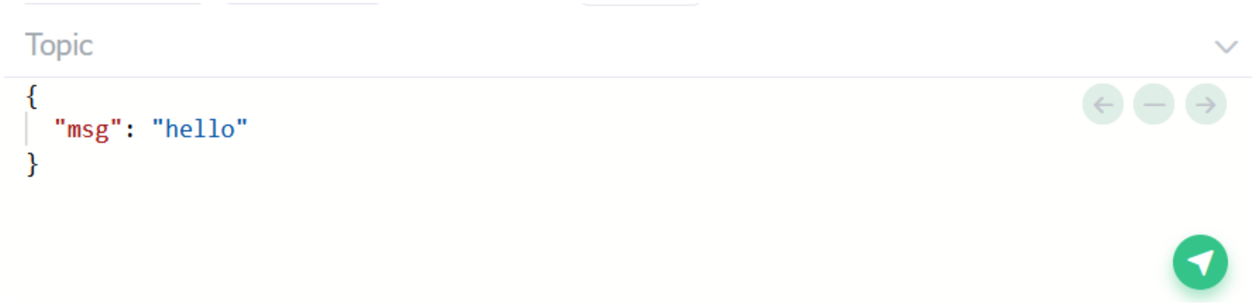
Back | New | Connect

General

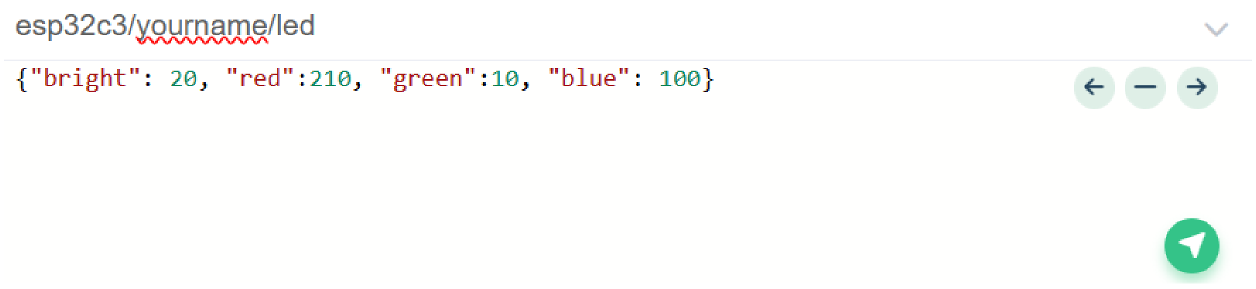
- * Name: abc
- * Host: wss:// broker.emqx.io
- * Port: 8084
- Client ID: mqttx_b1b9e51f
- * Path: /mqtt
- Username:
- Password:
- SSL/TLS: off

Advanced

At the bottom right, you can type in a topic to publish to, e.g. `esp32c3/yourname/led`. The message you wish to publish goes below the topic, e.g. `{"bright": 202, "red": 34, "green": 68, "blue": 102}`. Click send to publish.


Topic

```
{
  "msg": "hello"
}
```


esp32c3/yourname/led

```
{"bright": 20, "red":210, "green":10, "blue": 100}
```

ESP32 MQTT Publish

In this part of the project, you will connect a pushbutton to a GPIO pin of the ESP32-C3 and publish its state to the MQTT broker.

A number of additions are required for your sketch.

Add a `#define` statement for the length of the message buffer:

```
#define MSG_BUFFER_SIZE  40
```

Declare global variables, including designating the pin of the ESP32-C3 that will be connected to the pushbutton:

```
const int buttonPin = 3;  // the number of the pushbutton pin

// variable for storing the pushbutton status
int buttonState = 0; // by default not pressed
int lastButtonState = 0;
```

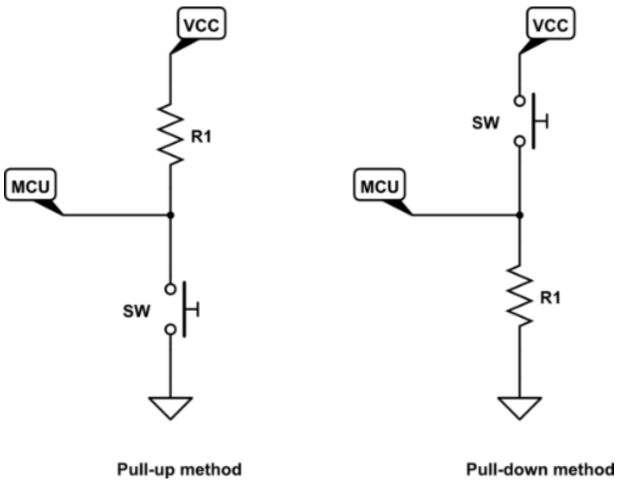
In `setup()`, add:

```
  pinMode(BUILTIN_LED, OUTPUT); // initialize BUILTIN_LED as output
  pinMode(buttonPin, INPUT);    // initialize pushbutton as input
```

In `loop()`, add the lines below. Note especially how the state of the button is published to the topic `esp32c3/yourname/button`, but only when the state changes:

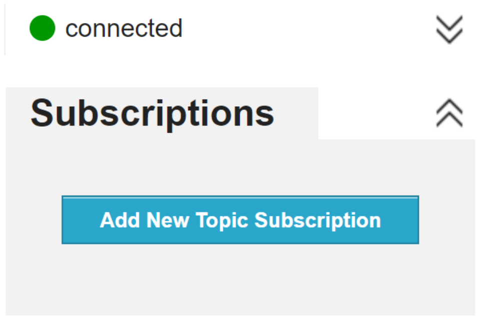
```
  // read the state of the pushbutton value
  buttonState = digitalRead(buttonPin);
  // if the pushbutton is pressed, buttonState is HIGH (pull-down)
  if (buttonState == HIGH && buttonState != lastButtonState) {
      snprintf (msg, MSG_BUFFER_SIZE, "{\"buttonStatus\":true}");
      Serial.print("Publish message: ");
      Serial.println(msg);
      client.publish("esp32c3/yourname/button", msg);
      lastButtonState = buttonState;
  } else if (buttonState == LOW && buttonState != lastButtonState){
      snprintf (msg, MSG_BUFFER_SIZE, "{\"buttonStatus\":false}");
      Serial.print("Publish message: ");
      Serial.println(msg);
      client.publish("esp32c3/yourname/button", msg);
      lastButtonState = buttonState;
  }
```

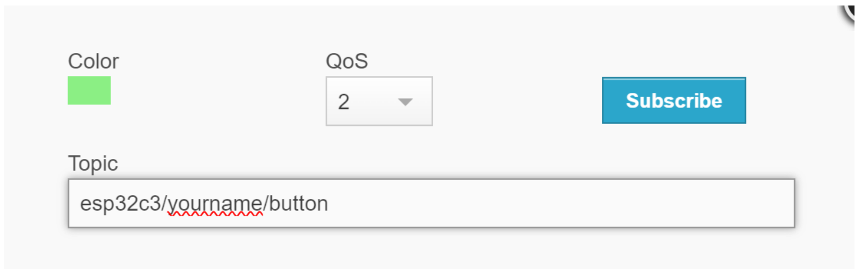
The program code indicates that the button pin should be `HIGH` when the button is pushed, which requires a pull-down circuit, such as the circuit on the right. Build a pull-down switch circuit on a breadboard and use a jumper to connect the point marked MCU with pin 3 of your ESP32-C3, as designated in your sketch.



Save your code, and click on the arrow in the Arduino IDE to compile and upload your program. Leave the program running.

At MQTT Websocket Client, make sure you are still connected. Click Add New Topic Subscription and enter the topic `esp32c3/yourname/button`.





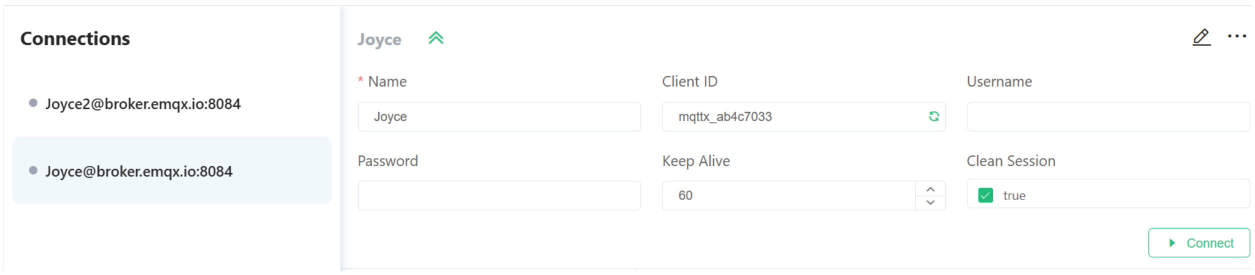
QoS: 2

Topic: esp32c3/yourname/button

Press the button. You should see a message pop up at the HiveMQ MQTT broker. The Serial Monitor of your Arduino IDE should show the message that was published by your ESP32-C3. The message is received by the HiveMQ MQTT broker because it subscribes to the topic.

Alternative Broker

If you are using `broker.emqx.io`, go to https://mqttx.app/web-client#/, choose one of your connections and click Connect.



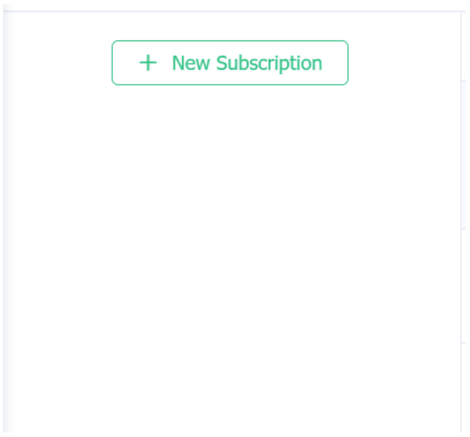
Connections

- Joyce2@broker.emqx.io:8084
- Joyce@broker.emqx.io:8084

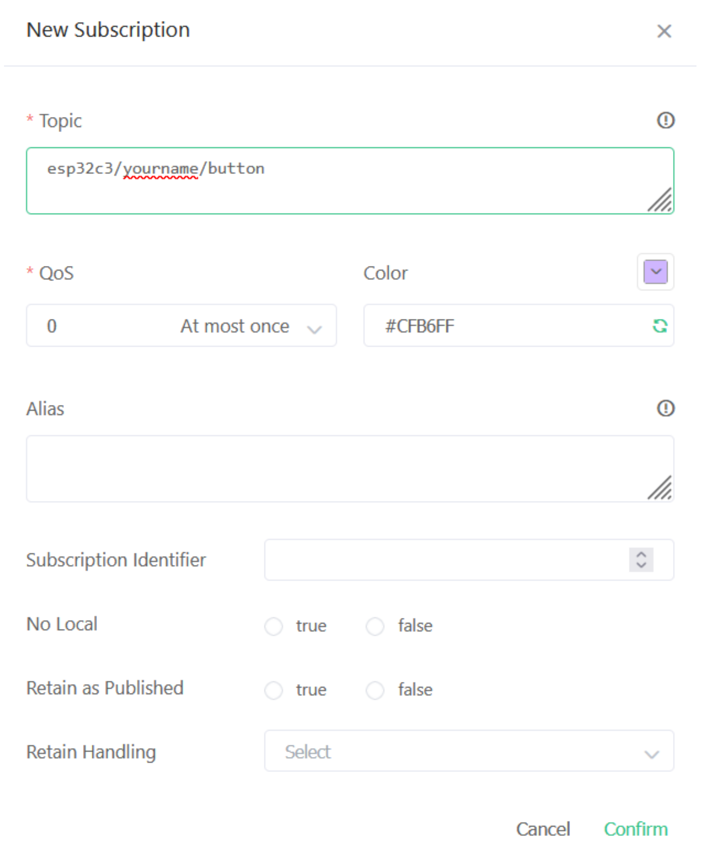
Joyce

* Name	Client ID	Username
Joyce	mqttx_ab4c7033	
Password	Keep Alive	Clean Session
	60	✓ true

To subscribe to a topic, click New Subscription:



Enter the topic name click Confirm:



* Topic: esp32c3/yourname/button

* QoS: 0 At most once

Color: #CFB6FF

Alias:

Subscription Identifier:

No Local: true false

Retain as Published: true false

Retain Handling: Select

Adding WiFi capability to STM32F407 with ESP32

The goal of the next parts of the project will be to allow your STM32F407 to use WiFi courtesy of your ESP32.

In your `tft_lcd` project, you displayed the temperature from a thermistor on an LCD screen. The same temperature data will be sent as a string by serial UART communication to the ESP32, which will publish it to the HiveMQ MQTT broker, where it can monitored from the cloud.

Your ESP32 code will subscribe to the press of a pushbutton, which you can imagine as being located in some remote place. Upon reception of a button press, your ESP32 will send the button state to the STM32, where it will turn on and off the image of the light bulb that was previously controlled by the blue user button on the STM32F407 Discovery board.

For convenience, you will pretend that the pushbutton you set up in the last section is the remote pushbutton. In other words, not only will you publish the state of your pushbutton, you will also subscribe to it. In this way, you can test the intended functionality of the project.

ESP32 receives button state as JSON via MQTT and sends button state to STM32 via UART

Add the `#include`:

```
#include "driver/uart.h"
```

Add the following `#define` statements. Note that the transmit (and receive) pins must be different from the button pin. Also, the baud rate of the ESP32 must match that used by the STM32:

```
#define RXPIN      4      // GPIO 4 => RX for Serial1
#define TXPIN      5      // GPIO 5 => TX for Serial1
#define BAUD       115200
```

In the `setup()` function, initialize the serial port and send the initial button state:

```
  // GPIO4 is Rx and GPIO5 is Tx
  Serial1.begin(BAUD, SERIAL_8N1, RXPIN, TXPIN);  // Rx = 4, Tx = 5
will work for ESP32, S2, S3 and C3

  // send initial button state
  Serial1.write(buttonState);
```

In the `reconnect()` function, right after the subscribe to `esp32c3/yourname/led`, add the line:

```
client.subscribe("esp32c3/yourname/button");
```

An example of the correct format for a button message is `{"buttonState":true}`.

Change the `callback()` function to handle two cases one for messages for the topic `esp32c3/yourname/led`, and one for messages for the topic `esp32c3/yourname/button`. All received messages are deserialized. The LED is then handled as before. For the button, after the state is extracted from the JSON-formatted message received from the MQTT broker, it is written to the serial port of the ESP32.

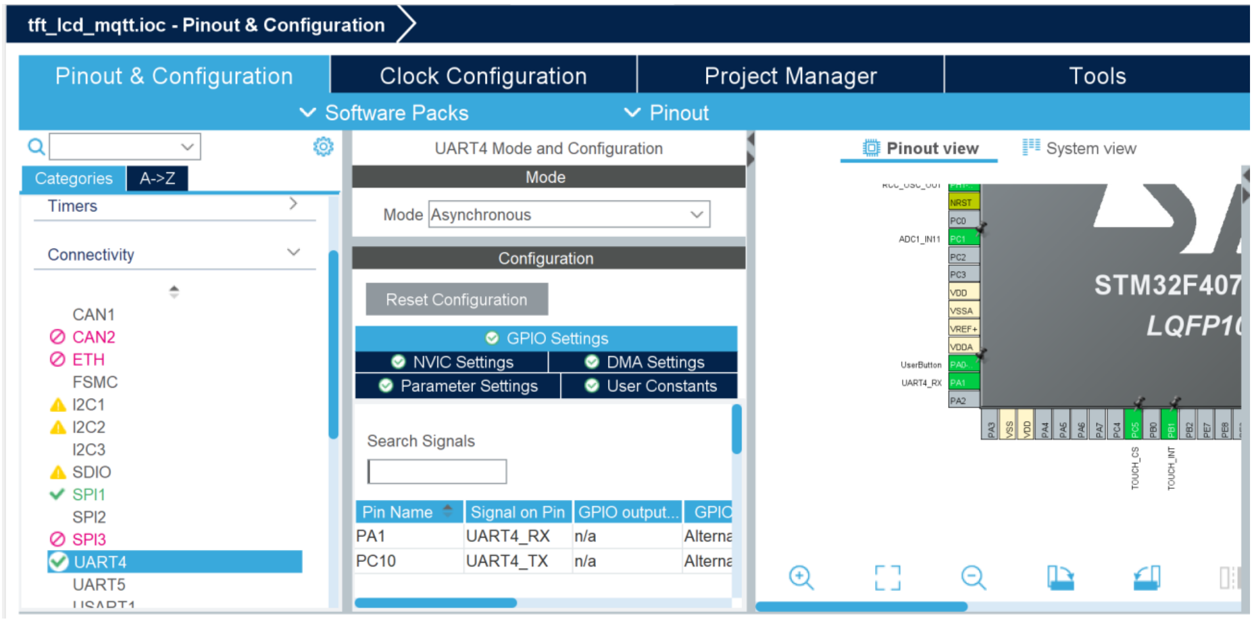
```
  if(strcmp(topic,"esp32c3/yourname/led") == 0) {
    bright = doc["bright"];
    red = doc["red"];
    green = doc["green"];
    blue = doc["blue"];
    rgbLedWrite(RGB_BUILTIN, (red*bright/255)&255,
(green*bright/255)&255, (blue*bright/255)&255);
  }
  else if(strcmp(topic,"esp32c3/yourname/button") == 0) {
    buttonState = doc["buttonState"];
    // write data to UART
    Serial1.write(buttonState);
  }
```

STM32 receives button state to control LCD image

Go to STM32CubeIDE. Choose File – Open Projects from File System… and find your `tft_lcd_mqtt` project folder.

Double click on `tft_lcd_mqtt.ioc` to open the configuration graphic.

Select Connectivity – UART4 and assign it Asynchronous mode. Make note of the transmit and receive pins. Here, pin PA1 is the UART Rx pin and pin PC10 is the UART Tx pin.



UART4 Mode and Configuration

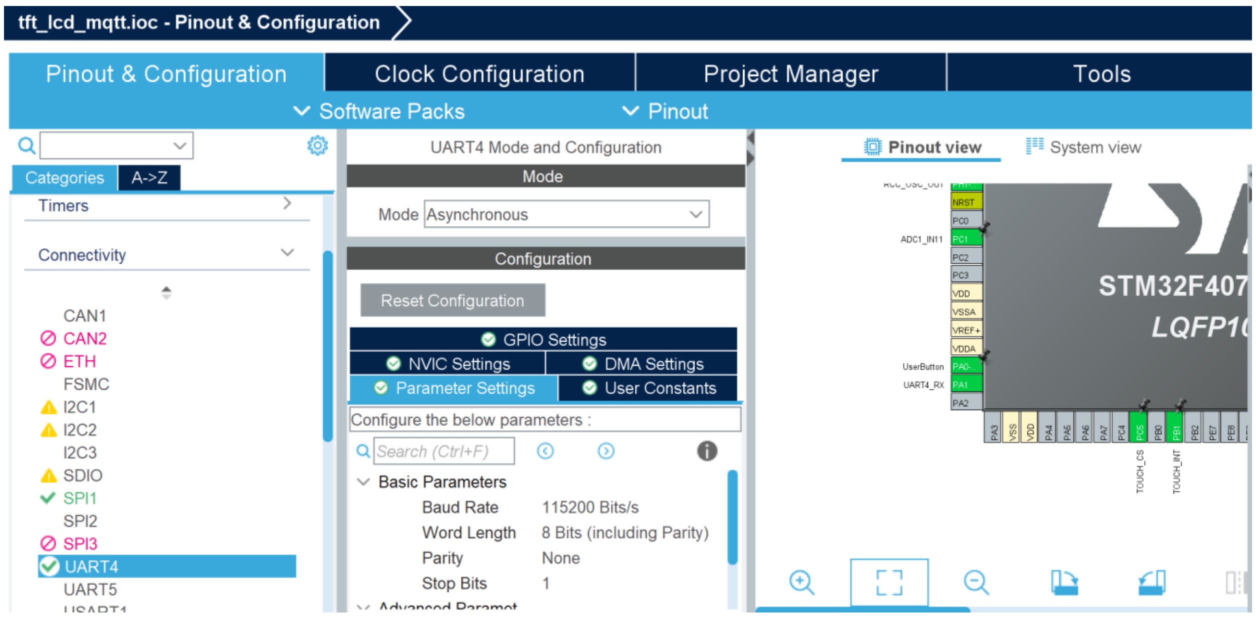
Mode

Mode Asynchronous

Configuration

Pin Name	Signal on Pin	GPIO output...	GPIO
PA1	UART4_RX	n/a	Alterna
PC10	UART4_TX	n/a	Alterna

In the parameter settings for the UART, ensure the baud rate is 115200 bits per second, as for the ESP32 serial port.



UART4 Mode and Configuration

Mode

Mode Asynchronous

Configuration

Configure the below parameters :

Basic Parameters

Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1

Generate code.

Some modifications are needed in `main.c`.

In the `USER CODE` private defines area, add two constants. One should be `0` and the other `1`.

```
#define USER_BUTTON 0
#define MQTT_BUTTON 1
```

In the `USER CODE` private variables area, declare the serial buffer that will receive the button state:

```
uint8_t button_buffer[1] = {0};
```

Modify your `while(1)` loop to permit control of the LCD bulb image by the blue user button on the Discovery board or the pushbutton on your breadboard, whichever is selected by `#define`. New code is shown in bold:

```
  while (1)
  {
        // this equation converts ADC value to thermistor resistance
        // 10k is resistor between thermistor and 3.3 V
        thermRes = 10000.0*adc_input/(ADCMAX-adc_input);
        // solving equation 1/T = 1/To + 1/B ln(R/Ro)
        // thermistor has resistance Ro at To, temperatures in Kelvin
        temperature = thermRes/THERMRESNOM;
        temperature = log(temperature);
        temperature /= BCOEFF;
        temperature += 1.0/(TEMPNOM + 273.15);
        temperature = 1.0/temperature;
        temperature -= 273.15;

        #if USER_BUTTON
        // use the following code with User Button control
      if(HAL_GPIO_ReadPin(UserButton_GPIO_Port, UserButton_Pin) ==
GPIO_PIN_RESET)
      {
            buttonPressed = false;
        }
        else if (HAL_GPIO_ReadPin(UserButton_GPIO_Port, UserButton_Pin)
== GPIO_PIN_SET)
        {
            buttonPressed = true;
        }
        #elif MQTT_BUTTON
        // use the following code with MQTT button subscription control
        HAL_UART_Receive(&huart4,(uint8_t *) button_buffer, 1, 1); //
with 1 msec timeout since messages are rare
        if(button_buffer[0] == 0 || button_buffer[0] == 1)
             buttonPressed = (bool)button_buffer[0];
      #endif
    /* USER CODE END WHILE */

      MX_TouchGFX_Process();
    /* USER CODE BEGIN 3 */
  }
```

Connect a jumper wire from the transmit pin of the ESP32-C3 (pin 5) to the receive pin of the STM32F407 (pin PA1).

Run – Debug and play/resume in STM32CubeIDE.

Compile and upload in ESP32 Arduino IDE.

When you press the pushbutton on your breadboard, the button's state is published by the ESP32 to the HiveMQ MQTT broker in JSON format. Since your ESP32 also subscribes to the button topic, the button's state is received as a JSON message by the ESP32. The button's state is extracted from the message, and passed to the STM32 via UART. On the STM32, the button's state controls the image of the light bulb on the LCD screen. Verify that when you press the button on your breadboard, the on light bulb is displayed, and when you release the button, the off light bulb is displayed on the LCD screen. If you change the #define statements to select `USER_BUTTON`, control of the light bulb image should revert to the blue user button on the Discovery board.

STM32 sends temperature to ESP32 via UART

You will modify your `tft_lcd_mqtt` project to send temperatures from the STM32 to the ESP32.

```
/* USER CODE BEGIN Includes */
#include <stdbool.h>
#include <math.h>
#include <stdio.h>
```

In the USER CODE private variables area, add . Note that the buffer containing the temperature string is initialized to a null string.

```
char temp_data_buffer[6] ="\0"; // buffer of temperature data
```

The UART for the STM32 was already set up in the previous step. In `USER CODE` area just before `while(1)`, add a line to send the initial null string to the transmit buffer. The final argument is a timeout in msec.

```
/* USER CODE BEGIN WHILE */
// transmit initial temperature null string
HAL_UART_Transmit(&huart4,(uint8_t *)temp_data_buffer,strlen(temp_data_buffer),50);

 while (1)
{
```

In the `USER CODE` area inside the `while(1)`, add a UART transmit for temperatures within a reasonable range. The temperature is printed to a buffer as a four-character string, which is sent to the serial port for transmit.

```
// solving equation 1/T = 1/To + 1/B ln(R/Ro)
// thermistor has resistance Ro at To, temperatures in Kelvin
if(avg_count == AVG_OVER) {
    thermRes = avg_total/(float)AVG_OVER;
    temperature = thermRes/THERMRESNOM;
    temperature = log(temperature);
    temperature /= BCOEFF;
    temperature += 1.0/(TEMPNOM + 273.15);
    temperature = 1.0/temperature;
    temperature = temperature - 273.15;
    avg_total = 0.0;
    avg_count = 0;
    if(temperature > -120.0 && temperature < 120.0) {
        sprintf(temp_data_buffer,"%4.1f",temperature);
        HAL_UART_Transmit(&huart4,(uint8_t *)temp_data_buffer,strlen(temp_data_buffer),50);
    }
}
```

Save your code.

ESP32 receives temperature from STM32 via UART and sends temperature as JSON via MQTT

In your ESP32 sketch, add the `#define` statement for the length of the temperature string:

```
#define NUM_TEMPERATURE_BYTES 4
```

Add declarations that will be needed:

```
bool timer_flag = false;
bool rx_flag = false;
size_t available;
hw_timer_t *timer = NULL;
```

Modify your `setup()` function. A timer is set to have frequency 1 MHz, and a timer alarm is set to fire every one million ticks of the timer, resulting in one alarm per second. The callback function for the timer is identified as `onTimer`. In addition, the receive buffer for the serial port is set up and its callback function, `onReceiveFunction`, is identified.

```
void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);

  pinMode(BUILTIN_LED, OUTPUT);  // initialize the BUILTIN_LED pin as an output
  pinMode(buttonPin, INPUT);     // initialize pin connected to pushbutton as an input

  timer = timerBegin(1000000); //timer frequency
  timerAttachInterrupt(timer, &onTimer);
  timerAlarm(timer, 1000000, true, 0); // number of ticks; timer interrupts once per second (1000000/1000000 = 1)

  // GPIO4 is Rx and GPIO5 is Tx
  Serial1.begin(BAUD, SERIAL_8N1, RXPIN, TXPIN);  // Rx = 4, Tx = 5 will work for ESP32, S2, S3 and C3
  Serial1.setRxBufferSize(NUM_TEMPERATURE_BYTES);
  Serial1.onReceive(onReceiveFunction, false);    // sets a RX callback function for Serial 1

  // send initial button state
  Serial1.write(buttonState);
}
```

The callback functions must be added to the sketch. Each sets a flag to indicate an interrupt for the timer (an alarm has been raised) or the receive buffer (something has been received) has occurred.

```
void onTimer() {
  if(timer_flag == false) {
    timer_flag = true;
  }
}

void onReceiveFunction() {
  if(rx_flag == false) {
    rx_flag = true;
  }
}
```

Add code to your `loop()`. This code works as follows: If the receive flag is true, but the timer alarm has not yet fired, then the receive buffer is emptied. If the receive flag is true and the timer alarm is also true, then the characters describing the temperature string are collected into an array `newTemp` (excess characters are removed). A null terminator is added to the array, and the temperature string is published in JSON format to the HiveMQ MQTT Broker. Receive flags and timer flags are reset.

```
if(rx_flag == true) {
    if(timer_flag == true) {
      available = Serial1.available();

      Serial.printf("onTimer callback:: %d bytes available: ",
available); // serial receive buffer is 256 bytes max
      int i=0;
      while (available--) {
        if(i < NUM_TEMPERATURE_BYTES) {
          newTemp[i] = (char)Serial1.read();
          Serial.print(newTemp[i++]);
        }
        else
          Serial1.read();
      }
      newTemp[i]='\0';

      Serial.println();
      snprintf (msg, MSG_BUFFER_SIZE, "{\"temperature\":%s}",
newTemp);
      Serial.print("Publish message: ");
      Serial.println(msg);
      Serial.println();
      client.publish("esp32c3/yourname/temperature", msg);

      timer_flag = false;
    }
    else {
      available = Serial1.available();
      while (available--) {
        Serial1.read();
      }
    }
    rx_flag = false;
  }
```

How to test your STM32 plus ESP32 project

To test everything:

1. Connect a jumper wire from the receive pin of the ESP32-C3 (pin 4) to the transmit pin of the STM32F407 (pin PC10). There should be two jumper wires between the boards now, one in each direction.

2. Run – Debug your `tft_lcd_mqtt` project in STM32CubeIDE. Play – Resume.

3. Compile and upload your ESP32 Arduino sketch `mqtt_json_uart`. Open the Serial Monitor for your Arduino IDE.

4. Connect at MQTT Websocket Client.

    (a) Publish to `esp32c3/yourname/led`.
    (b) Subscribe to `esp32c3/yourname/button`.

Things that should work:

1. You should be able to press the Red/Blue/Green buttons on your LCD with the stylus and see LEDs on the Discovery board toggle.

2. If `MQTT_BUTTON` control is selected, the blue user button on the Discovery board should not do anything.

3. If you press the pushbutton on your breadboard, the image of the light bulb on your LCD should turn on. When you release the pushbutton, the image of the light bulb should turn off.

4. When you press the pushbutton on your breadboard, you should see messages posted at the HiveMQ MQTT broker.

5. When you press the pushbutton on your breadboard, you should see pairs of messages on the Arduino IDE Serial Monitor, one to publish to the button topic and one because the ESP32 subscribes to the button topic.

6. When you publish an LED message from the HiveMQ MQTT broker, your LED should change colour and brightness.

At this point, your STM32 is connected to the HiveMQ MQTT broker in the cloud via the ESP32. Imagine thermistors in many different locations each publishing temperature information, and a central control facility monitoring them all by subscribing to them. From the central facility, an operator can press a button to cause an alert on the LCD screen of any of the local microcontrollers.

Simple web page receives and sends MQTT messages

Another way you can interact with MQTT clients is via a website. There are many ways to do this, but this project will use a simple HTML webpage with straightforward JavaScript (and CSS for styling, though this is technically optional).

The complete code for the web page is provided in Appendix 3.

MQTT.js is a popular JavaScript library for connecting and interacting with MQTT. This library is imported using a script tag in the document's head section with the source `https://unpkg.com/mqtt/dist/mqtt.min.js`:

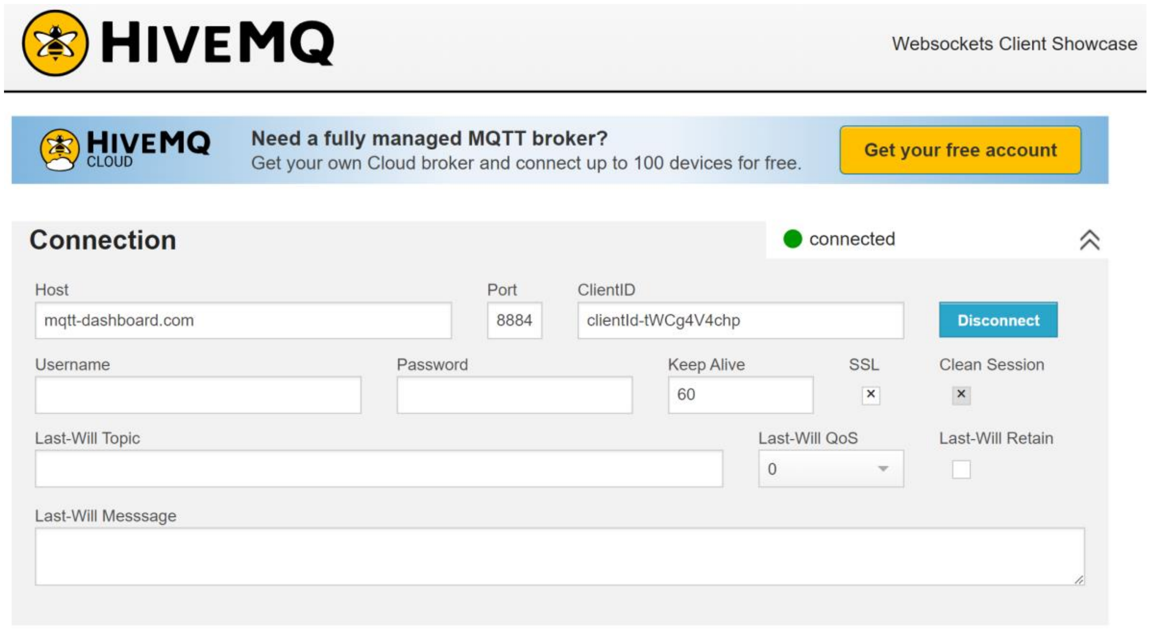
```
<script src="https://unpkg.com/mqtt/dist/mqtt.min.js"></script>
```

This script initializes a variable called `mqtt` and provides all the functions needed to interact with the MQTT broker and its clients.

The rest of the functions are enclosed within a second script tag (or alternatively saved into a separate JavaScript file and imported to your HTML document using `<script src="yourfilename.js"></script>`). The JavaScript code first creates a client that will interact with the MQTT broker:

```
const client = mqtt.connect('wss://yourHiveMQurl:yourHiveMQport/mqtt');
```

where `yourHiveMQurl` is the URL provided as Host on your HiveMQ Websocket dashboard (https://www.hivemq.com/demos/websocket-client/), and `yourHiveMQport` is the number provided as Port.


HIVEMQ
Websockets Client Showcase
HIVEMQ CLOUD
Need a fully managed MQTT broker?
Get your own Cloud broker and connect up to 100 devices for free.
Get your free account
Connection
connected
Disconnect

Host: mqtt-dashboard.com
Port: 8884
ClientID: clientId-tWCg4V4chp
Username:
Password:
Keep Alive: 60
SSL: ✓
Clean Session: ✓
Last-Will Topic:
Last-Will QoS: 0
Last-Will Retain:
Last-Will Messsage:

For example, for the HiveMQ broker shown, the client definition would be:

```
const client = mqtt.connect('wss://mqtt-dashboard.com:8884/mqtt');
```

Once your client is connected, you can start interacting with MQTT messages. The MQTT.js library has a number of built-in events you can listen for, such as “connect”, which is triggered when the client’s connection is successful, and “message”, which is triggered when a message arrives from a topic this client is subscribed to.

You can react to the “connect” event as follows:

```
client.on('connect', function () {
        // Your code
        // Runs on successful connection
});
```

You can subscribe to a topic to listen continuously. For example:

```
client.on('connect', function () {
        console.log('Connected');
        // subscribe to test topic
        client.subscribe('esp32c3/yourname/test', function (err) {
                if (!err) {
                        // if no error, publish a message to test topic
                        client.publish('esp32c3/yourname/test', 'Connected!');
                }
        })
        // subscribe to desired topics
        client.subscribe('esp32c3/yourname/temperature');
        client.subscribe('esp32c3/yourname/button');
});
```

After logging a successful connection, the client (your web page) subscribes to a test topic to which a message is published if the subscription is successful. Subscribing to a test topic step is optional but confirms basic functionality. The code also subscribes to topics of interest, in this case `esp32c3/yourname/temperature` and `esp32c3/yourname/button`.

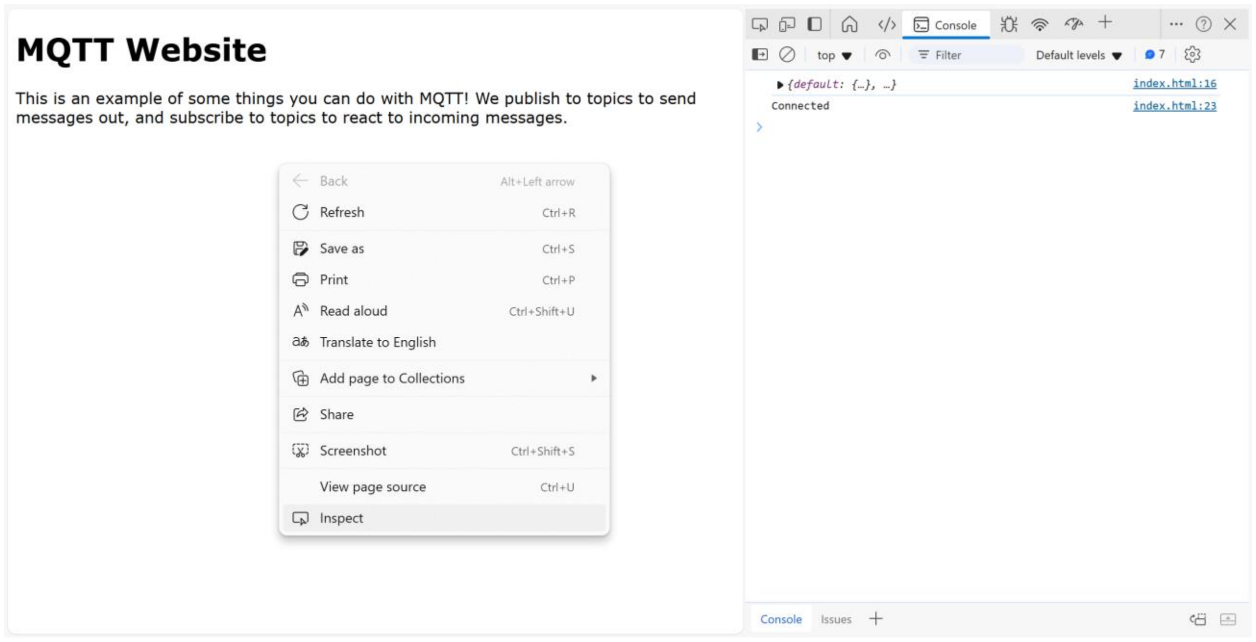
Now the code reacts to incoming messages in the form:

```
client.on('message', function (topic, message) {
        // Your code
        // Runs when message arrives from topic to which you are subscribed
});
```

For example, you can log the message to the console:

```
client.on('message', function (topic, message) {
        // Log newest message
        console.log(message.toString());
});
```

To view the messages logged to the console, open your webpage, then right click and choose Inspect, which will open the development tools panel. One of the tabs in this panel is Console, which is where `console.log` messages are printed.



MQTT Website

This is an example of some things you can do with MQTT! We publish to topics to send messages out, and subscribe to topics to react to incoming messages.

```
▶{default: {…}, …}                                  index.html:16
Connected                                           index.html:23
```

By defining elements within the `<body>` section of your HTML page, you can create reactions and controls for MQTT messages visible on your webpage. Headings and text can be added using `<h1>` and `<p>` tags, for example. In the code, four sliders (red, green, blue, brightness) are created that will be used to control the RGB LED on the ESP32:

```
	<h2>Set ESP32 LED</h2>
	<p>
	<label for="red">Red</label>
	0<input id="red" type="range" min="0" max="255" value="50"
onchange="setLED()">255
	</p>

	<p>
	<label for="green">Green</label>
	0<input id="green" type="range" min="0" max="255" value="50"
onchange="setLED()">255
	</p>

	<p>
	<label for="blue">Blue</label>
	0<input id="blue" type="range" min="0" max="255" value="50"
onchange="setLED()">255
	</p>
```

```
        <p>
        <label for="bright">Brightness</label>
        0<input id="bright" type="range" min="0" max="255" value="50"
onchange="setLED()">255
        </p>
```

The onchange property allows you to trigger a function when the user changes any of the sliders. The Javascript function setLED() creates a message in JSON format containing the four colour and brightness values, and publishes it to the LED topic. The values of the sliders are obtained using their IDs. You can simply use client.publish without first subscribing to the topic, but wrapping it in the subscription callback function allows you to verify it was published successfully because you will also receive the message you just published.

```
        function setLED() {
                console.log("Publishing LED values...");
                const msg = "{\"bright\": " +
        document.getElementById("bright").value + ", \"red\": " +
        document.getElementById("red").value + ", \"green\": " +
        document.getElementById("green").value + ", \"blue\": " +
        document.getElementById("blue").value + "}";
                client.subscribe('esp32c3/yourname/led', function (err) {
                        if (!err) {
                                // publish a message to a topic
                                client.publish('esp32c3/yourname/led', msg);
                        }
                });
        }
```

A slider with only two settings is created to control is the lightbulb image on the STM32 LCD display. For the `toggle` class, you can add some styling to make the slider short to appear more like a switch. This, and any other styling you may choose to add (such as colouring the colour sliders), can be put in a `style` tag in the head section of your HTML document or in a separate CSS file which you can then import to your HTML document. Some example styling is included in the full code in Appendix 3:

```
        <h2>LCD Lightbulb Switch</h2>
        <p>
        <label for="lightbulb">Lightbulb</label>
        Off<input class="toggle" id="lightbulb" type="range" min="0" max="1"
value="0" onchange="setLightbulb()">On
        </p>
```

For this switch slider, we trigger the `setLightbulb()` Javascript function. As for the `setLED()` function, a JSON message is constructed containing the switch status and the message is published to the button topic:

```
function setLightbulb() {
      const msg = "{\"buttonState\": " +
document.getElementById("lightbulb").value + "}";
      client.subscribe('esp32c3/yourname/button', function (err) {
            if (!err) {
                  // Publish a message to a topic
                  client.publish('esp32c3/yourname/button', msg);
            }
      });
}
```

The web page will also monitor the temperature reported by a thermistor, so a placeholder for the temperature value is added in the HTML:

```
<h2>Current temperature</h2>
<p>It is <span id="temperature"></span> °C.</p>
```

To update this value, we need to react to messages coming in from the temperature topic. The function reacting to the “message” event includes code for messages from specific topics. Here, if the incoming message is from the temperature topic, the placeholder (with ID “temperature”) is set to the incoming temperature value. Because these MQTT messages are in JSON format, `JSON.parse()` is used to create a JavaScript array from the JSON, which can then be indexed easily by name. Messages from the button topic are also checked here, since in earlier steps of this project, physical buttons were used to control the LCD lightbulb. If the button state is set elsewhere, the website’s button switch is altered to match the current state of the lightbulb. The + operator before the incoming button state ensures it is read as a number (0 for false, 1 for true):

```
client.on('message', function (topic, message) {
      // Log newest message
      console.log(message.toString());
      if (topic == 'esp32c3/yourname/temperature') {
            const message_arr = JSON.parse(message);
            // Display current temperature
            document.getElementById("temperature").innerHTML =
      message_arr["temperature"];
      } else if (topic == 'esp32c3/yourname/button') {
            const message_arr = JSON.parse(message);
            // Set Lightbulb slider value to current lightbulb status
            document.getElementById("lightbulb").value =
      +message_arr["buttonState"];
      }
      // Update webpage console
      updateConsole(message);
});
```

The final (optional) function is `updateConsole()`. For ease of viewing, the three most recent MQTT messages are displayed on the webpage, so the user does not have to open the console. First HTML placeholders are created for these messages:

```
<h2>Incoming Messages</h2>
<p id="newest"></p>
<p id="older"></p>
<p id="oldest"></p>
```

In Javascript, the `updateConsole()` function fills these values as follows:

```
function updateConsole(message) {
      document.getElementById("oldest").innerHTML =
document.getElementById("older").innerHTML;
      document.getElementById("older").innerHTML =
document.getElementById("newest").innerHTML;
      document.getElementById("newest").innerHTML = message.toString();
}
```

If desired, `client.end()` can be called at any point to disconnect your websocket client. In this case, `mqtt.connect()` must be used again to reconnect, and all topics of interest must be resubscribed.

Appendix 1 Making a copy of a project

To begin, go to your STM32 workspace folder and make a copy of your `tft_lcd` project folder. Rename the copied folder `tft_lcd_mqtt`.

In the File Explorer, ensure hidden file are showing. If there is a `.git` folder in your project folder, delete it.

Open each of the following files in the Notepad and change every occurrence of `tft_lcd` to `tft_lcd_mqtt` (including in the filenames). The last three of the files are in the `TouchGFX` directory of your project.

```
.project
.cproject
tft_lcd.ioc
tft_lcd.launch
ApplicationTemplate.touchgfx.part
target.config
tft_lcd.touchgfx
```

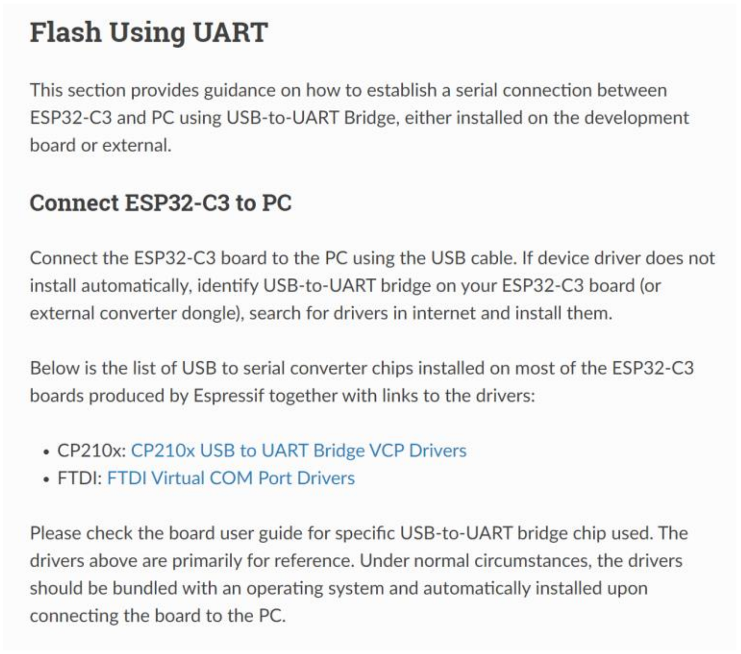
In STM32CubeIDE, choose File – Open Projects from File System..., locate your new `tft_lcd_mqtt` project folder and click Select Folder. Compile your project.

Right-click on your project's name to open its Properties. At the left, under C/C++ Build, choose Settings. Click on Libraries and verify that the library search path contains the `tft_lcd_mqtt` project name. If all is well, click Cancel.

Appendix 2 Installing Windows driver for ESP32

One possible reason for a “serial port not connected” error is using a charge-only rather than a data-capable USB cable. If your USB cable is not the problem, you most likely need to install a driver.

The steps for connecting the ESP32 to your PC are described at Establish Serial Connection with ESP32-C3 - ESP32-C3 - — ESP-IDF Programming Guide v5.3.2 documentation, in the Flash Using UART section.



Flash Using UART

This section provides guidance on how to establish a serial connection between ESP32-C3 and PC using USB-to-UART Bridge, either installed on the development board or external.

Connect ESP32-C3 to PC

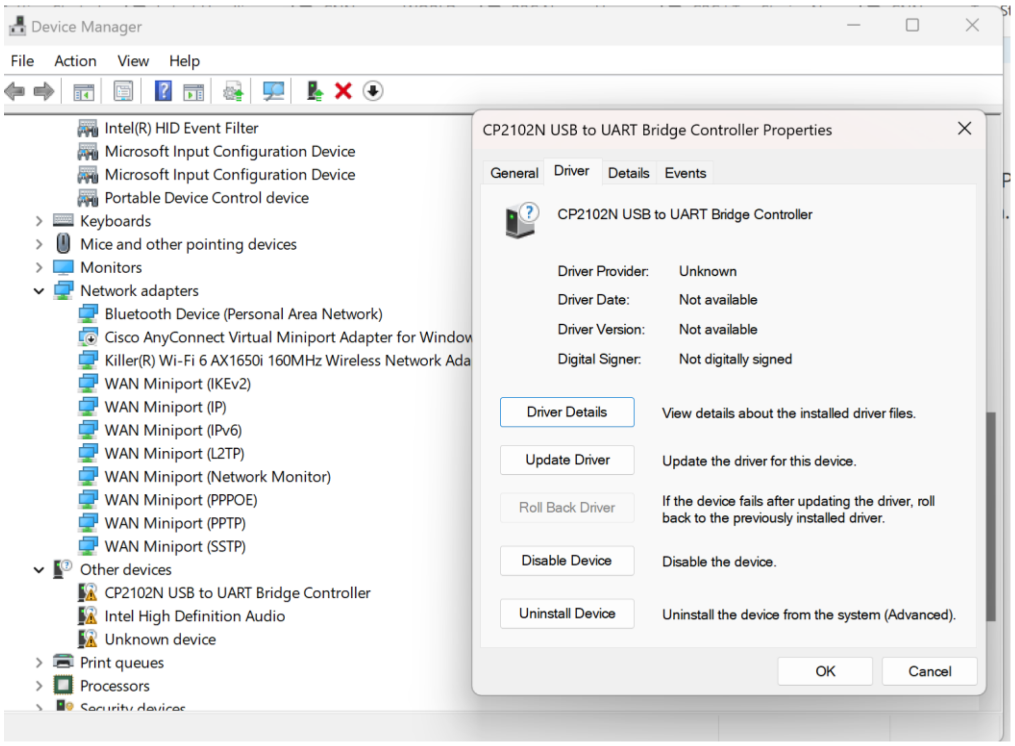
Connect the ESP32-C3 board to the PC using the USB cable. If device driver does not install automatically, identify USB-to-UART bridge on your ESP32-C3 board (or external converter dongle), search for drivers in internet and install them.

Below is the list of USB to serial converter chips installed on most of the ESP32-C3 boards produced by Espressif together with links to the drivers:

- CP210x: CP210x USB to UART Bridge VCP Drivers
- FTDI: FTDI Virtual COM Port Drivers

Please check the board user guide for specific USB-to-UART bridge chip used. The drivers above are primarily for reference. Under normal circumstances, the drivers should be bundled with an operating system and automatically installed upon connecting the board to the PC.

If your PC does not install a driver automatically, go to your PC’s Device Manager to determine which serial converter chip is present on your ESP32-C3. In this example, it is CP2102N USB to UART Bridge Controller, appearing under Other Devices.


Device Manager

File Action View Help

Intel(R) HID Event Filter
Microsoft Input Configuration Device
Microsoft Input Configuration Device
Portable Device Control device
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Bluetooth Device (Personal Area Network)
Cisco AnyConnect Virtual Miniport Adapter for Window
Killer(R) Wi-Fi 6 AX1650i 160MHz Wireless Network Ada
WAN Miniport (IKEv2)
WAN Miniport (IP)
WAN Miniport (IPv6)
WAN Miniport (L2TP)
WAN Miniport (Network Monitor)
WAN Miniport (PPPOE)
WAN Miniport (PPTP)
WAN Miniport (SSTP)
Other devices
CP2102N USB to UART Bridge Controller
Intel High Definition Audio
Unknown device
Print queues
Processors
Security devices

CP2102N USB to UART Bridge Controller Properties

General Driver Details Events

CP2102N USB to UART Bridge Controller

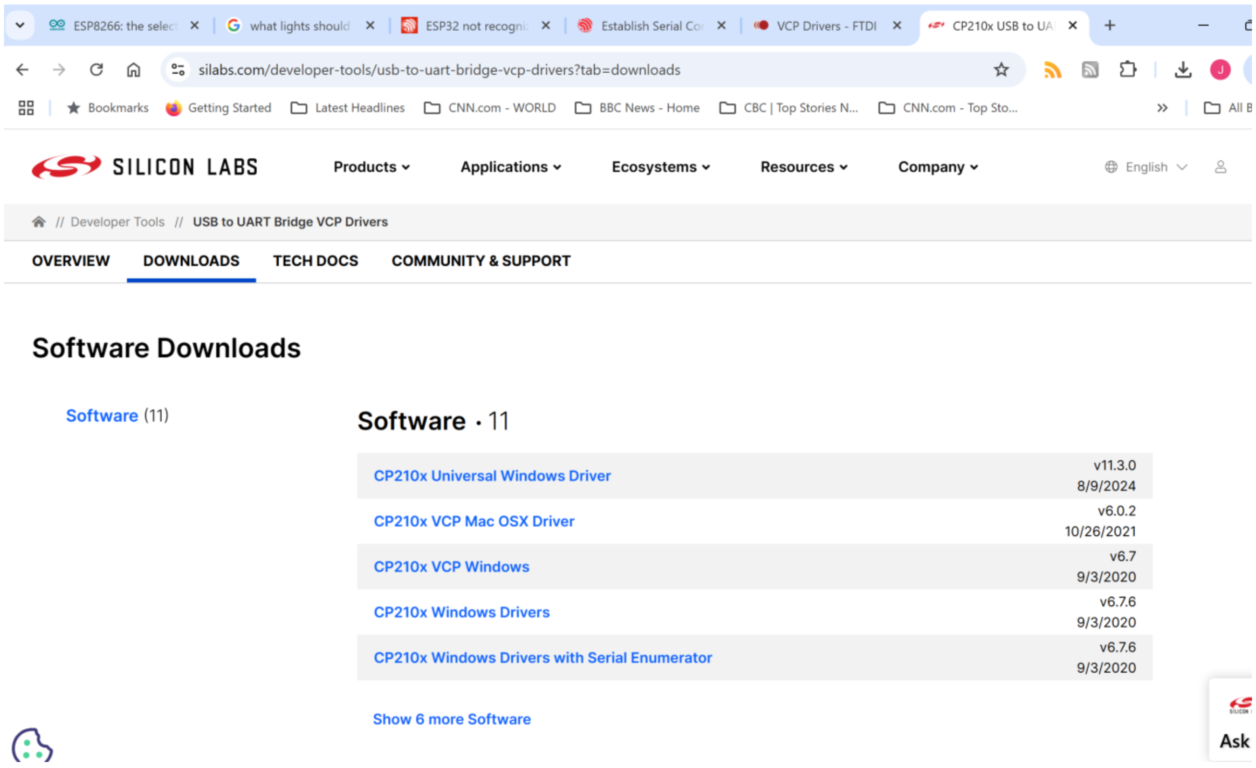
Driver Provider:	Unknown
Driver Date:	Not available
Driver Version:	Not available
Digital Signer:	Not digitally signed

Driver Details	View details about the installed driver files.
Update Driver	Update the driver for this device.
Roll Back Driver	If the device fails after updating the driver, roll back to the previously installed driver.
Disable Device	Disable the device.
Uninstall Device	Uninstall the device from the system (Advanced).

OK Cancel

In this example, the CP210x Universal Windows Driver may be downloaded from:

https://www.silabs.com/developer-tools/usb-to-uart-bridge-vcp-drivers?tab=downloads


silabs.com/developer-tools/usb-to-uart-bridge-vcp-drivers?tab=downloads

SILICON LABS Products Applications Ecosystems Resources Company English

// Developer Tools // USB to UART Bridge VCP Drivers

OVERVIEW DOWNLOADS TECH DOCS COMMUNITY & SUPPORT

Software Downloads

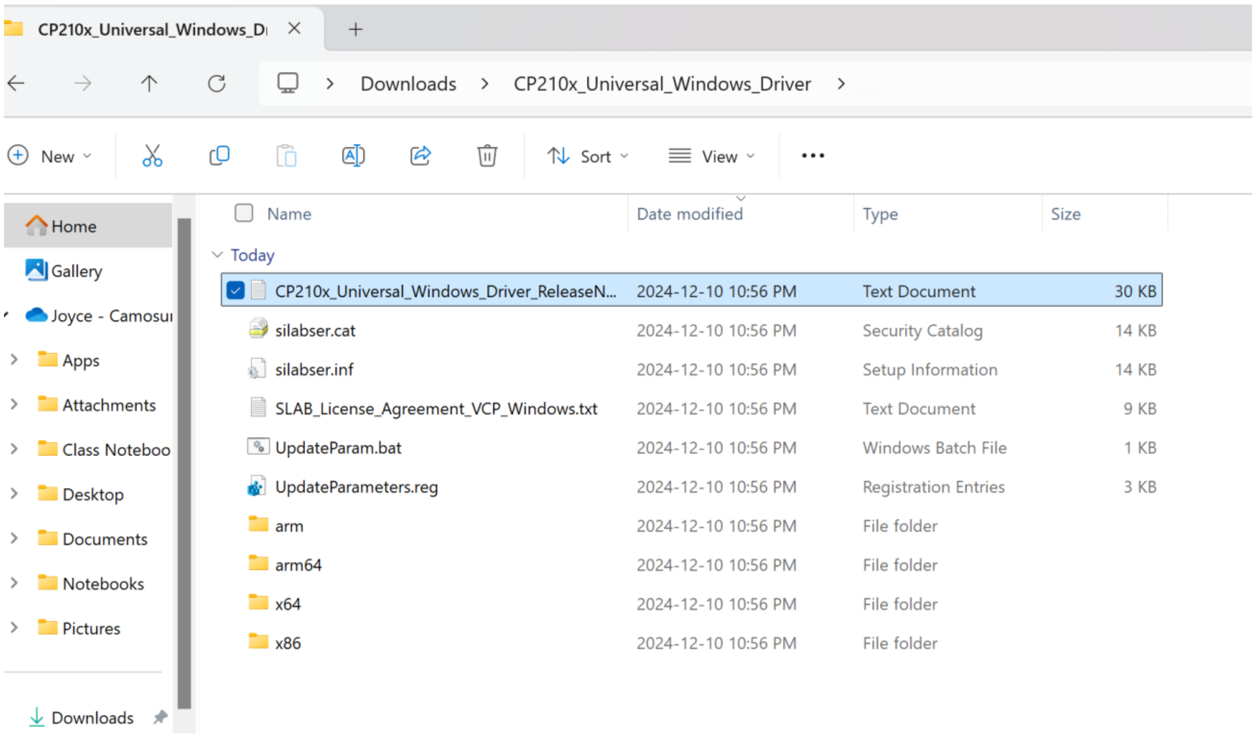
Software (11)

Software · 11

CP210x Universal Windows Driver	v11.3.0 8/9/2024
CP210x VCP Mac OSX Driver	v6.0.2 10/26/2021
CP210x VCP Windows	v6.7 9/3/2020
CP210x Windows Drivers	v6.7.6 9/3/2020
CP210x Windows Drivers with Serial Enumerator	v6.7.6 9/3/2020

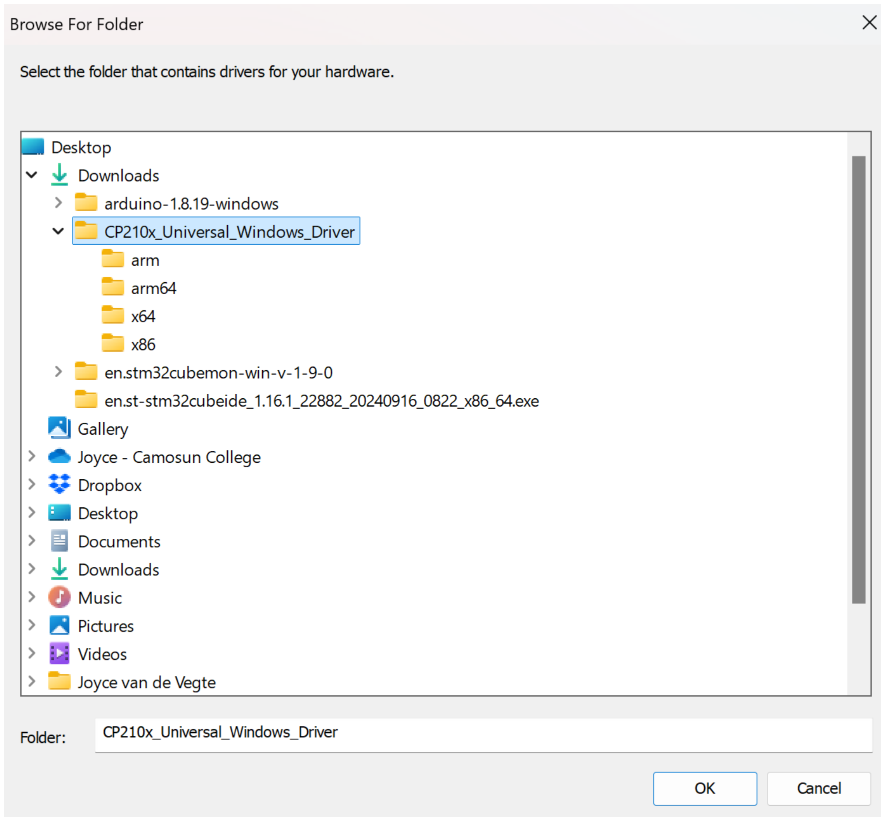
Show 6 more Software

Extract all.



Name	Date modified	Type	Size
CP210x_Universal_Windows_Driver_ReleaseN...	2024-12-10 10:56 PM	Text Document	30 KB
silabser.cat	2024-12-10 10:56 PM	Security Catalog	14 KB
silabser.inf	2024-12-10 10:56 PM	Setup Information	14 KB
SLAB_License_Agreement_VCP_Windows.txt	2024-12-10 10:56 PM	Text Document	9 KB
UpdateParam.bat	2024-12-10 10:56 PM	Windows Batch File	1 KB
UpdateParameters.reg	2024-12-10 10:56 PM	Registration Entries	3 KB
arm	2024-12-10 10:56 PM	File folder	
arm64	2024-12-10 10:56 PM	File folder	
x64	2024-12-10 10:56 PM	File folder	
x86	2024-12-10 10:56 PM	File folder	

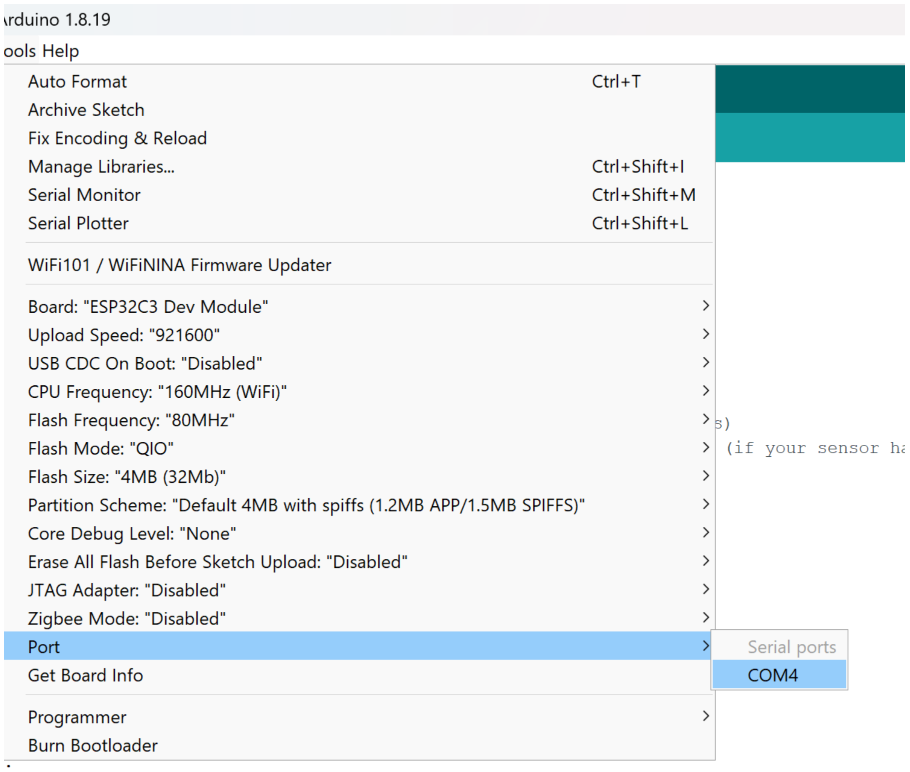
In your Device Manager, double-click on the CP2102N USB to UART Bridge Controller (it may appear under Other Devices or Ports, choose Update Driver, and navigate to the unzipped folder. Click OK.


Browse For Folder

Select the folder that contains drivers for your hardware.

Folder: CP210x_Universal_Windows_Driver

After the driver has been successfully installed, a COM port should be available.


rduino 1.8.19
ools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater

Board: "ESP32C3 Dev Module"
Upload Speed: "921600"
USB CDC On Boot: "Disabled"
CPU Frequency: "160MHz (WiFi)"
Flash Frequency: "80MHz"
Flash Mode: "QIO"
Flash Size: "4MB (32Mb)"
Partition Scheme: "Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS)"
Core Debug Level: "None"
Erase All Flash Before Sketch Upload: "Disabled"
JTAG Adapter: "Disabled"
Zigbee Mode: "Disabled"
Port
Serial ports
COM4

Appendix 3 Website Code

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta charset="UTF-8">

  <title>MQTT Website</title>

  <!-- Import MQTT.js library -->
  <script src="https://unpkg.com/mqtt/dist/mqtt.min.js"></script>

  <script>
    // The MQTT.js library script will initialise an mqtt variable globally
    // We can log it to see
    console.log(mqtt);

    // Connect to MQTT websocket
    const client = mqtt.connect('wss://mqtt-dashboard.com:8884/mqtt');

    // Once connected, use a test topic to verify functionality
    client.on('connect', function () {
      console.log('Connected');
      // Subscribe to test topic
      client.subscribe('esp32c3/Joyce/test', function (err) {
        if (!err) {
          // If no error, publish a message to test topic
          client.publish('esp32c3/Joyce/test', 'Connected!');
        }
      })
      // Then subscribe to desired topics
      client.subscribe('esp32c3/Joyce/temperature');
      client.subscribe('esp32c3/Joyce/button');
    });

    // Receive messages
    client.on('message', function (topic, message) {
      // Log newest message
      console.log(message.toString());
      if (topic == 'esp32c3/Joyce/temperature') {
        const message_arr = JSON.parse(message);
        // Display current temperature
        document.getElementById("temperature").innerHTML =
message_arr["temperature"];
      } else if (topic == 'esp32c3/Joyce/button') {
        const message_arr = JSON.parse(message);
        // Set Lightbulb slider value to current lightbulb status
        document.getElementById("lightbulb").value =
+message_arr["buttonState"];
      }
      // Update webpage console
      updateConsole(message);
    });

    // sample publish message: {"bright": 202, "red": 34, "green": 68,
"blue": 102}
    function setLED() {
      console.log("Publishing LED values...");
      const msg = "{\"bright\": " + document.getElementById("bright").value +
", \"red\": " + document.getElementById("red").value + ", \"green\": " +
document.getElementById("green").value + ", \"blue\": " +
document.getElementById("blue").value + "}";
      client.subscribe('esp32c3/Joyce/led', function (err) {
        if (!err) {
          // Publish a message to a topic
          client.publish('esp32c3/Joyce/led', msg);
        }
      });
    }

    // sample publish message: {"buttonState": true}
    function setLightbulb() {
      const msg = "{\"buttonState\": " +
document.getElementById("lightbulb").value + "}";
      client.subscribe('esp32c3/Joyce/button', function (err) {
        if (!err) {
          // Publish a message to a topic
          client.publish('esp32c3/Joyce/button', msg);
        }
      });
    }

    function updateConsole(message) {
      document.getElementById("oldest").innerHTML =
document.getElementById("older").innerHTML;
      document.getElementById("older").innerHTML =
document.getElementById("newest").innerHTML;
```

```
      document.getElementById("newest").innerHTML = message.toString();
    }

    // Use client.end() to disconnect if desired (need to use mqtt.connect
again to reconnect)
  </script>

  <style>
    /* Styling */
    h1, h2, p, label {
      font-family: Verdana, sans-serif;
    }
    .toggle {
      width: 25pt;
    }
    #red {
      accent-color: red;
    }
    #green {
      accent-color: green;
    }
    #blue {
      accent-color: blue;
    }
    #bright {
      accent-color: yellow;
    }
    .content {
      float: left;
    }
    .console {
      float: right;
    }
  </style>
</head>

<body>
  <h1>MQTT Website</h1>
  <p>This is an example of some things you can do with MQTT! We publish to
topics to send messages out, and subscribe to topics to react to incoming
messages.</p>

  <div class="content">
    <h2>Set ESP32 LED</h2>
    <p>
```

```
      <label for="red">Red</label>
      0<input id="red" type="range" min="0" max="255" value="50"
onchange="setLED()">255
    </p>

    <p>
      <label for="green">Green</label>
      0<input id="green" type="range" min="0" max="255" value="50"
onchange="setLED()">255
    </p>

    <p>
      <label for="blue">Blue</label>
      0<input id="blue" type="range" min="0" max="255" value="50"
onchange="setLED()">255
    </p>

    <p>
      <label for="bright">Brightness</label>
      0<input id="bright" type="range" min="0" max="255" value="50"
onchange="setLED()">255
    </p>

    <h2>LCD Lightbulb Switch</h2>
    <p>
      <label for="lightbulb">Lightbulb</label>
      Off<input class="toggle" id="lightbulb" type="range" min="0" max="1"
value="0" onchange="setLightbulb()">On
    </p>

    <h2>Current temperature</h2>
    <p>It is <span id="temperature"></span> °C.</p>
  </div>

  <div class="console">
    <h2>Incoming Messages</h2>
    <p id="newest"></p>
    <p id="older"></p>
    <p id="oldest"></p>
  </div>
</body>

</html>
```