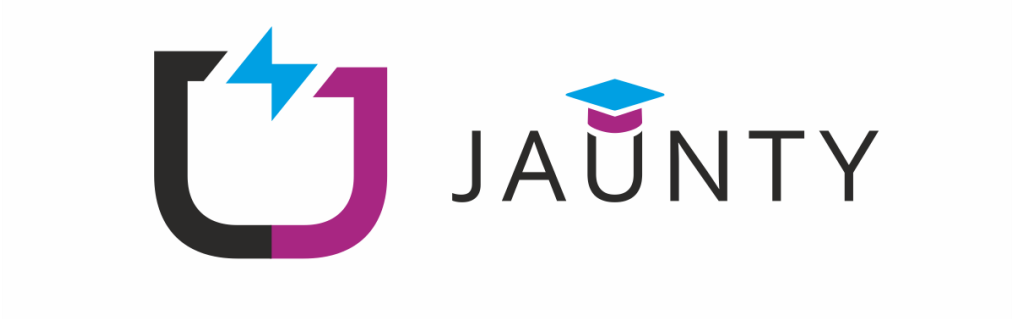
MQTT programming with ESP32 and FreeRTOS



Author: Public Power Corporation S.A

Co-funded by the
Erasmus+ Programme
of the European Union

Copyright

Consisting of

Coordinator:	Technical University of Sofia	Bulgaria
Partners:	University of Western Macedonia	Greece
	International Hellenic University	Greece
	Public Power Corporation S.A.	Greece
	University of Cyprus	Cyprus
	K3Y Ltd	Bulgaria
	Software Company EOOD	Bulgaria

Co-funded by the Erasmus+ Programme of the European Union

Table of Contents

1. Abbreviations

IoT	Internet of Things
MQTT	MQ Telemetry Transport
IDE	Integrated Development Environment
ROS	Real-time Operating System
OS	Operating System
GPIO	General Purpose Input Output
CPU	Central Processing Unit
RAM	Random Access Memory
ROM	Read Only Memory
SCL	Serial Clock Line
SDA	Serial Data Line
API	Application Programming Interface
TCP	Transmission Control Protocol
DHCP	Dynamic Host Configuration Protocol
IP	Internet Protocol

2. Scope

The scope of this laboratory module is to develop and deploy a real-time application on an ESP32 IoT device, that retrieves measurements via I2C and publishes them to an MQTT broker. In particular, this module will provide a comprehensive introduction to the ESP32 platform as well as the relevant Integrated Development Environment (IDE) utilised to develop and deploy applications on ESP32, written in C. Moreover, the students will be introduced to basic concepts of real-time operating systems since the application will be written utilising the FreeRTOS kernel and scheduler. Finally, the students will be introduced to basic concepts of the MQTT protocol and the publisher/subscriber (pub/sub) communication model.

2.1 Specific outcomes

Upon completion of this exercise, individuals will be able to:

- Understand and program the ESP32 IoT device.
- Use the Arduino IDE to develop and deploy applications on ESP32 written in C.
- Develop multi-tasking applications on ESP32 utilising the FreeRTOS kernel.
- Basic concepts and rationale behind asynchronous programming.
- Deploy an MQTT broker by utilising docker technologies.
- Understand the MQTT protocol and the pub/sub communication model.

2.2 General description

ESP32 is a modern, low-power system on a chip (SoC) microcontroller that integrates Wi-Fi and Bluetooth connectivity. It features a dual-core CPU at 160 MHz or 240 MHz, 320 KiB RAM, and 448 KiB ROM. Like Raspberry Pi 4B, ESP32 also integrates multiple programmable General-Purpose Input/Output (GPIO) as well as two I2C interfaces.

Figure 1 illustrates the pinout scheme of ESP32 [1]. Note that pin 3V3 provides 3.3 V to external devices (e.g., sensors), pin D22 is for Serial Clock Line (SCL), pin D21 for Serial Data Line (SDA). The microUSB connector is utilised for connecting and programming ESP32 via computer, while it can also supply ESP32 with the required voltage.

Usually, the supplied power via USB is adequate for ESP32 to supply external sensors via the 3V3 pin. However, if a low-quality or long USB cable is used instead, the supplied voltage may be not enough for ESP32 to supply external devices. In this case, the “Brownout detector was triggered” error will appear, and ESP32 will reboot endlessly [2]. This is a security feature that is triggered if the system voltage is below a threshold, also known as the "brownout voltage". This is to preserve memory contents and avoid system corruption.

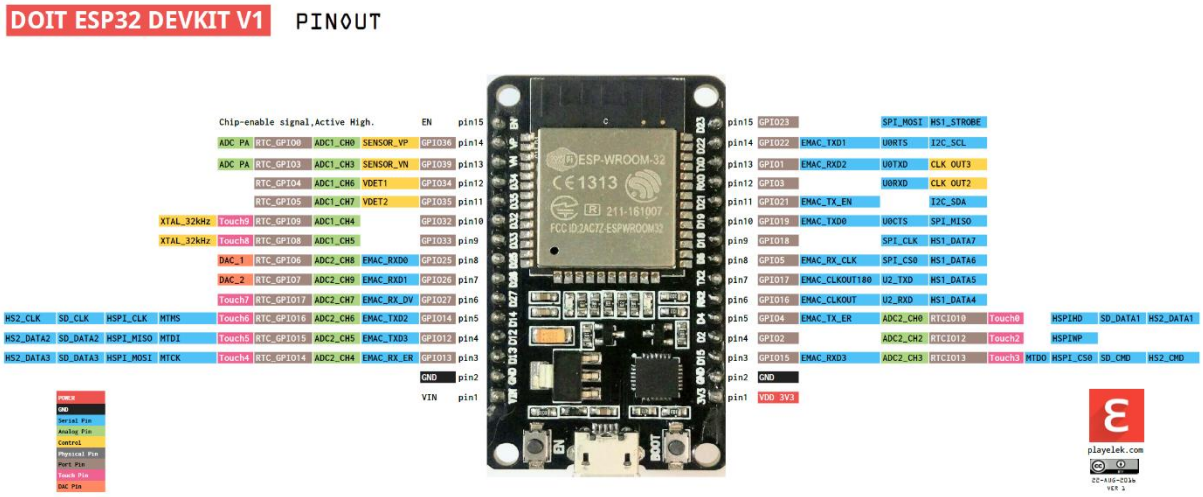


Figure 1: Pinout scheme of ESP32 [1]

Whereas Raspberry Pi 4B behaves as a general-purpose computing device, like a desktop computer, ESP32 is programmed only to implement specific pre-defined tasks, e.g., implement automation procedures, collect, and translate signals between different formats etc. ESP32 can be programmed by employing various OS kernels.

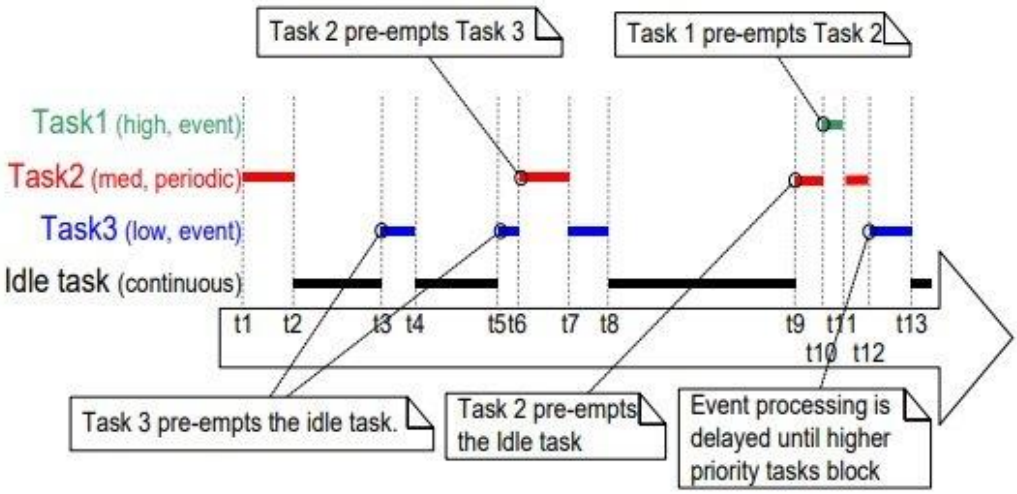


Figure 2: FreeRTOS scheduling example [3]

FreeRTOS is a popular open-source real-time operating system (RTOS) kernel for embedded devices, that has also been ported for ESP32. Kernel is an essential computer program that runs on top of the device hardware and undertakes to manage the hardware resources (memory management, CPU priority) as well as to schedule and execute the tasks created by higher-level applications. Moreover, compared to a conventional OS, an RTOS is intended to serve real-time applications that are bound to critical and defined time constraints, while they are characterised as event-driven and pre-emptive. FreeRTOS supports two scheduling policies, namely the round-robin (or time slicing) algorithm (all equal priority tasks get equal portions of CPU time), and fixed priority pre-emptive

scheduling (a high priority task gets the CPU before a low priority task; the low priority task executes only when there is no high priority task in the ready state) [3].

In this lab module, basic concepts of an RTOS will be examined, like tasks and queues for exchanging information between tasks.

Finally, this lab module studies the MQ Telemetry Transport (MQTT) standard, which is a popular protocol for lightweight Machine to Machine (M2M) communication. MQTT follows the pub/sub model, which is a form of asynchronous communication. According to this paradigm, clients are characterised as publishers and subscribers. Publishers are posting messages to specific topics on an intermediate server (broker), while subscribers are subscribed to specific topics and receive the published messages asynchronously [4]. Pub/Sub is a fundamental concept for cloud architectures and microservices [5].

Purpose of this laboratory is to develop an ESP32 application that:

1. Retrieves measurements from an SHT40 sensor via I2C.
2. Converts the measurements to JSON format.
3. Publishes those measurements to an MQTT broker.

2.3 Lab configuration

The following parameters are provided by the lab instructor:

Property	Value
Workstation - IP address	
Workstation - Username	
Workstation - Password	
Access method to the Workstation	VNC

3. Exercise 1: Infrastructure Preparation

Step 1: Topology implementation

This prototype topology has already been implemented and is remotely available in the physical laboratory premises. It is recommended to follow the steps of this exercise to replicate the prototype with your own equipment. In case that this is not possible, then you could study the implementation steps and proceed to the next step by accessing the pre-configured VM already provided by the lab instructor.

The breadboard topology required for the laboratory exercises is depicted in Figure 3. To implement the topology, the following equipment is needed:

- x1 wide breadboard or x2 smaller breadboards.
- x1 ESP32.
- x1 Adafruit SHT40 temperature and humidity sensor.
- x6 wires (depending on the implementation).
- x1 micro-USB to USB type A cable.

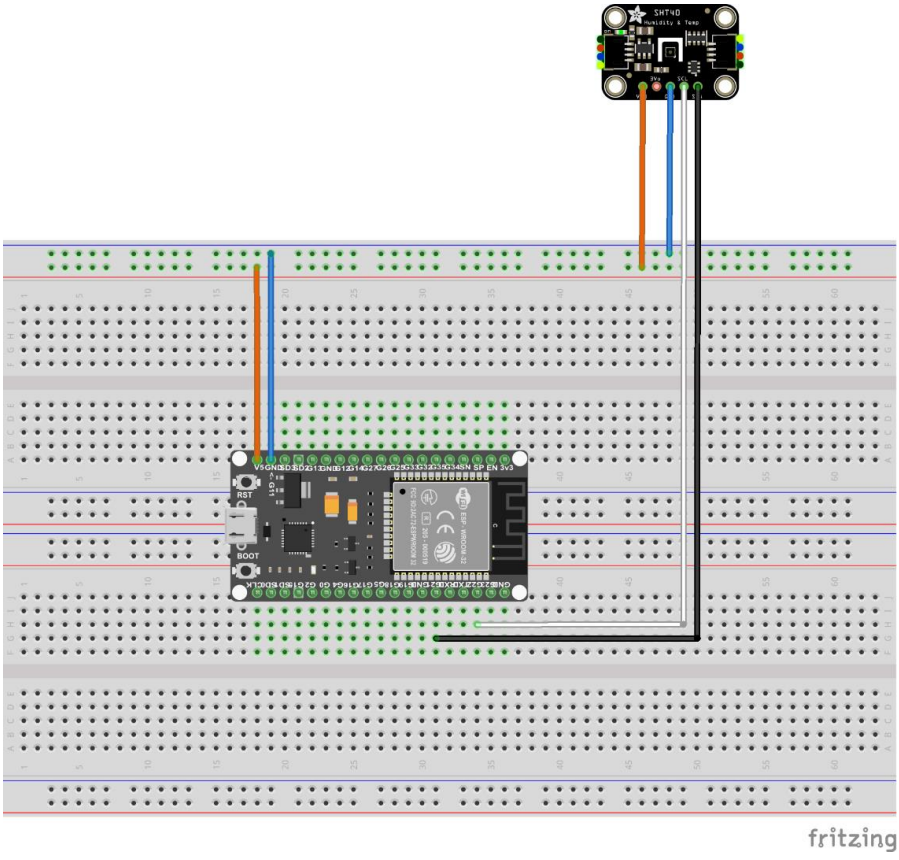


Figure 3: Topology of ESP32 with SHT40

ESP32 does not need any external power supply since it is being supplied by the USB cable. Utilising this source, ESP32 can supply external sensors, similarly to Raspberry Pi4B. To this aim, pins 19 (5V) and 18 (GND) are wired to the breadboard, to create a DC circuit that supplies the SHT40 sensor. Then, pin D21 (SDA) and pin D22 (SCL) are connected to the breadboard to provide the I2C bus for the SHT40 sensor.

The actual pins on your ESP32 may vary between the different vendors. Please check the labels above each pin, to select the right one.

Step 2: Install and Prepare the Arduino IDE for ESP32

Start by accessing the VM already provided for this lab, according to the information provided in section 2.3.

Various Integrated Development Environments (IDE) can be used for developing and deploying applications on ESP32:

- ESP-IDF (Espressif IoT Development Framework)
- Arduino IDE
- Visual Studio Code with Espressif IDF extensions

For the purposes of this lab, the Arduino IDE will be selected [6]. However, the student is free to select any IDE suitable to their preferences.

After installing Arduino IDE, the necessary ESP32 addons should be installed. For this purpose, navigate to “File > Preferences” as depicted in Figure 4.

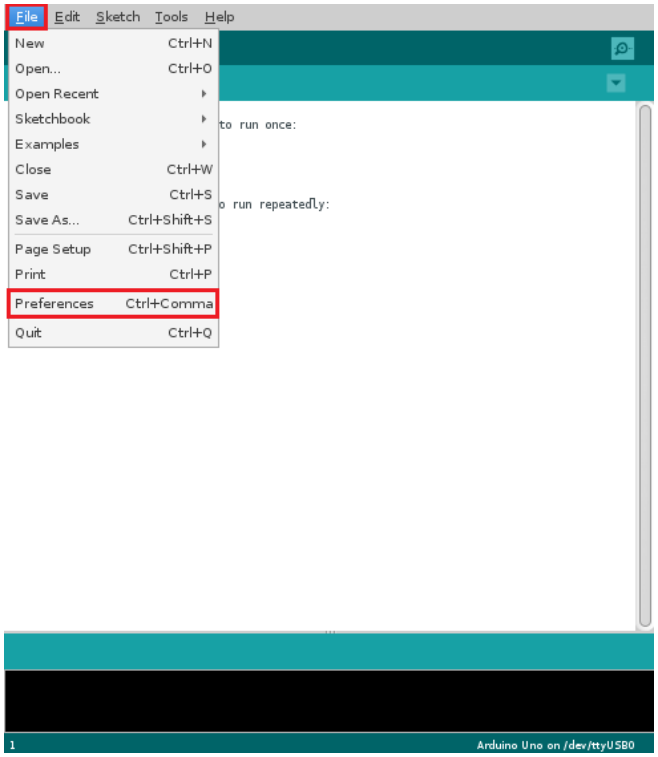


Figure 4: The "Preferences" menu in Arduino IDE

Then, add the following ESP32 repository provided by Espressif in the field indicated in Figure 5. Click "OK" after inserting the following URL.

https://dl.espressif.com/dl/package_esp32_index.json

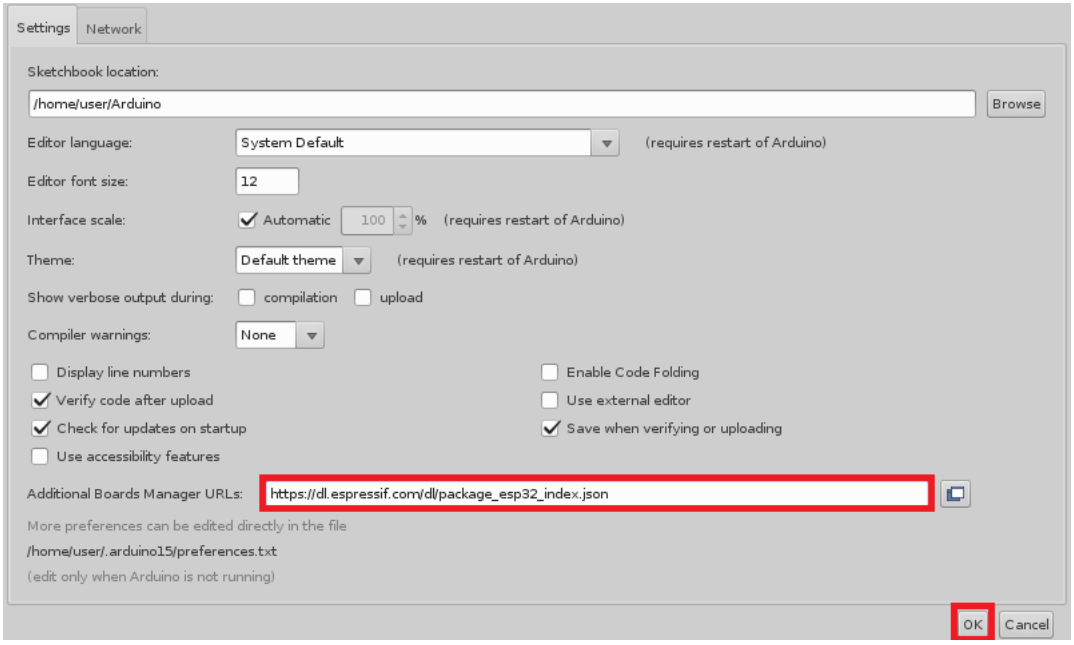


Figure 5: Provide the ESP32 repository URL in Arduino IDE

Next, navigate to “Tools > Board: > Boards Manager…” as depicted in Figure 6. This will open the Board Manager menu, which allows installing the ESP32 plugin.

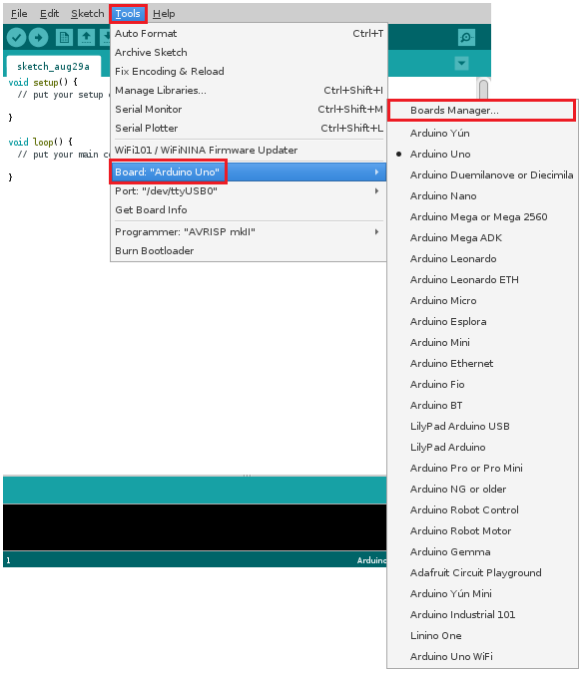


Figure 6: Navigate to Boards Manager in Arduino IDE

Search for the ESP32 plugin and install, as depicted in Figure 7.

Figure 7: Install the ESP32 plugin in Arduino IDE

After installation is completed, you need to configure Arduino IDE in order to deploy the code, based on the ESP32 characteristics. For this purpose, navigate to “Tools > Board > ESP32 Arduino” and select “ESP32 Dev Module”, as depicted in Figure 8.

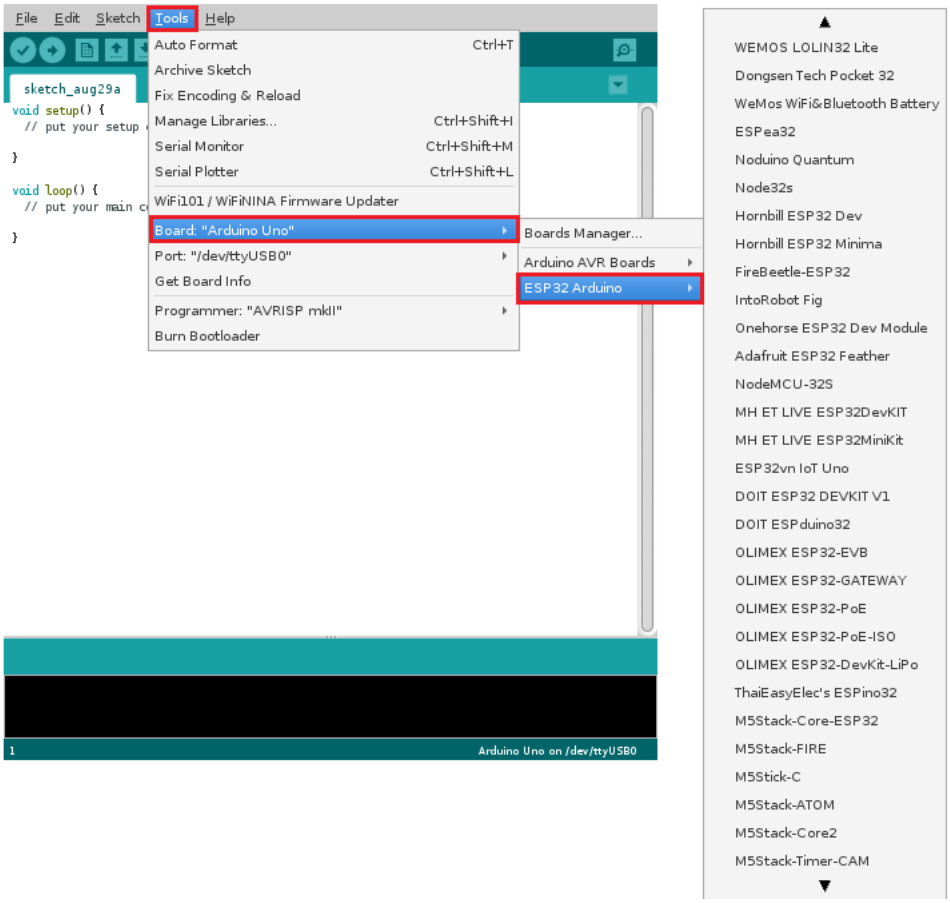


Figure 8: Select the ESP32 Dev Module profile

Step 3: Hello World - Test serial connection with ESP32

In this step, a simple application will be deployed on the ESP32 platform that retrieves and prints various hardware information of the device.

A typical Arduino/ESP32 app consists of two main functions, the setup() and loop(). The first function is executed only one right after the application is deployed, and usually undertakes to initialise network or serial interfaces. The loop() function runs the main code of the application. It should be highlighted that the code under loop() runs repeatedly.

Start by inserting the source code of this exercise as depicted in Figure 9. The source code is also provided bellow.

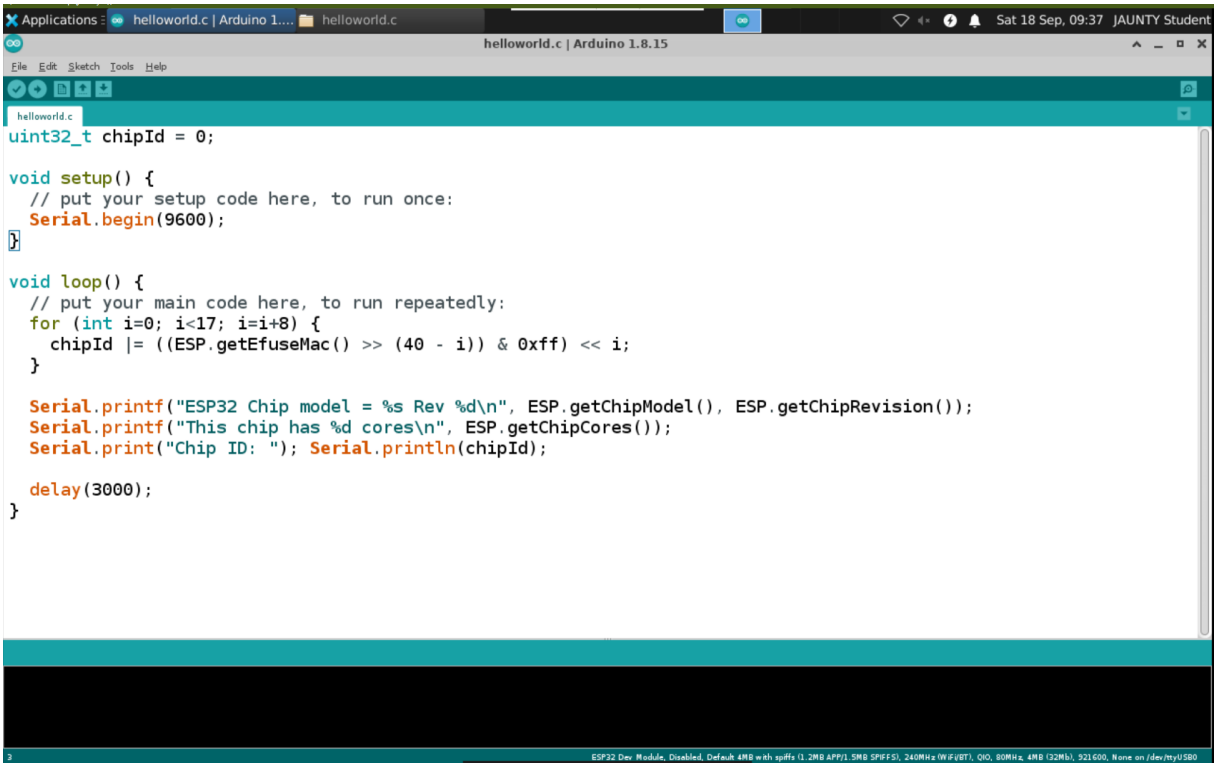


Figure 9: Source code of Exercise 2

```
1  uint32_t chipId = 0;
2
3  void setup() {
4    // put your setup code here, to run once:
5    Serial.begin(9600);
6  }
7
8  void loop() {
9    // put your main code here, to run repeatedly:
10   for (int i=0; i<17; i=i+8) {
11     chipId |= ((ESP.getEfuseMac() >> (40 - i)) & 0xff) << i;
12   }
13
14   Serial.printf("ESP32 Chip model = %s Rev %d\n", ESP.getChipModel(),
     ESP.getChipRevision());
15   Serial.printf("This chip has %d cores\n", ESP.getChipCores());
16   Serial.print("Chip ID: "); Serial.println(chipId);
17
18   delay(3000);
19 }
```

Before deploying the application, press on the “Verify” button, to make an initial check that there are no syntax errors. This button is depicted in Figure 10.

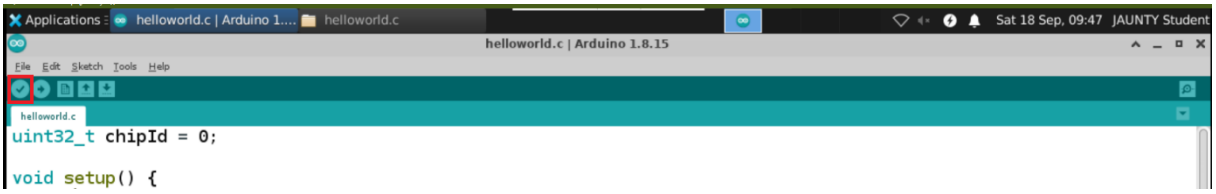


Figure 10: The "Verify" button on Arduino IDE

If no errors are detected, press the “Upload” button to deploy the sketch.

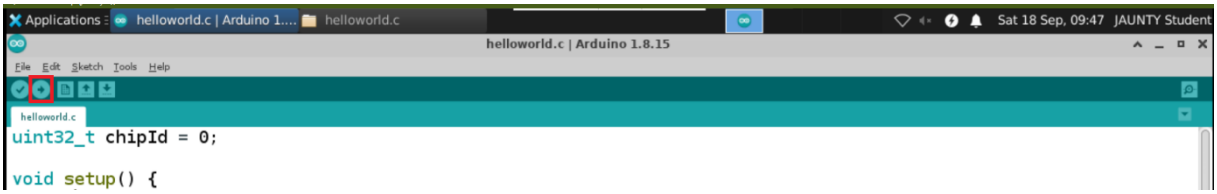


Figure 11: The "Upload" button on Arduino IDE

Figure 12 shows that the uploading procedure was successful. Right after, navigate through “Tools > Serial Monitor” to open the serial terminal of the ESP32 device. You should notice that the hardware information is printed repeatedly.

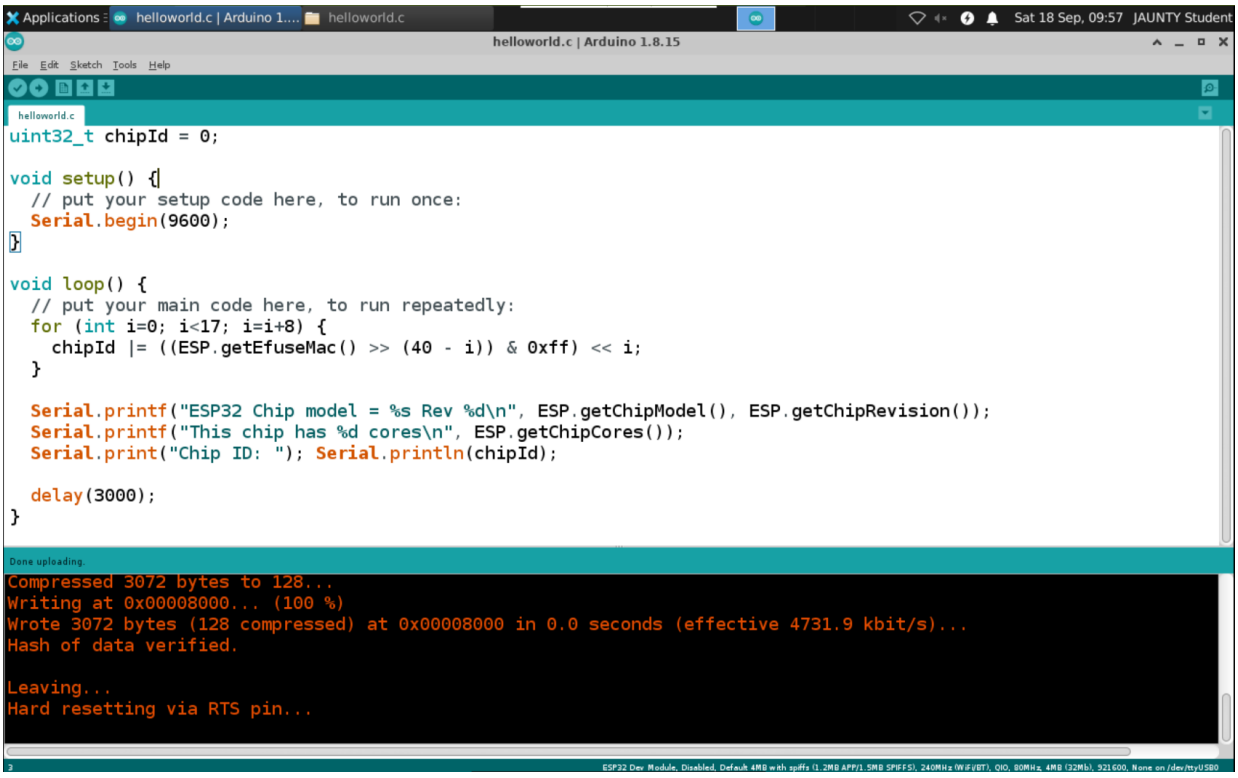


Figure 12: Successful uploading of ESP32 sketch

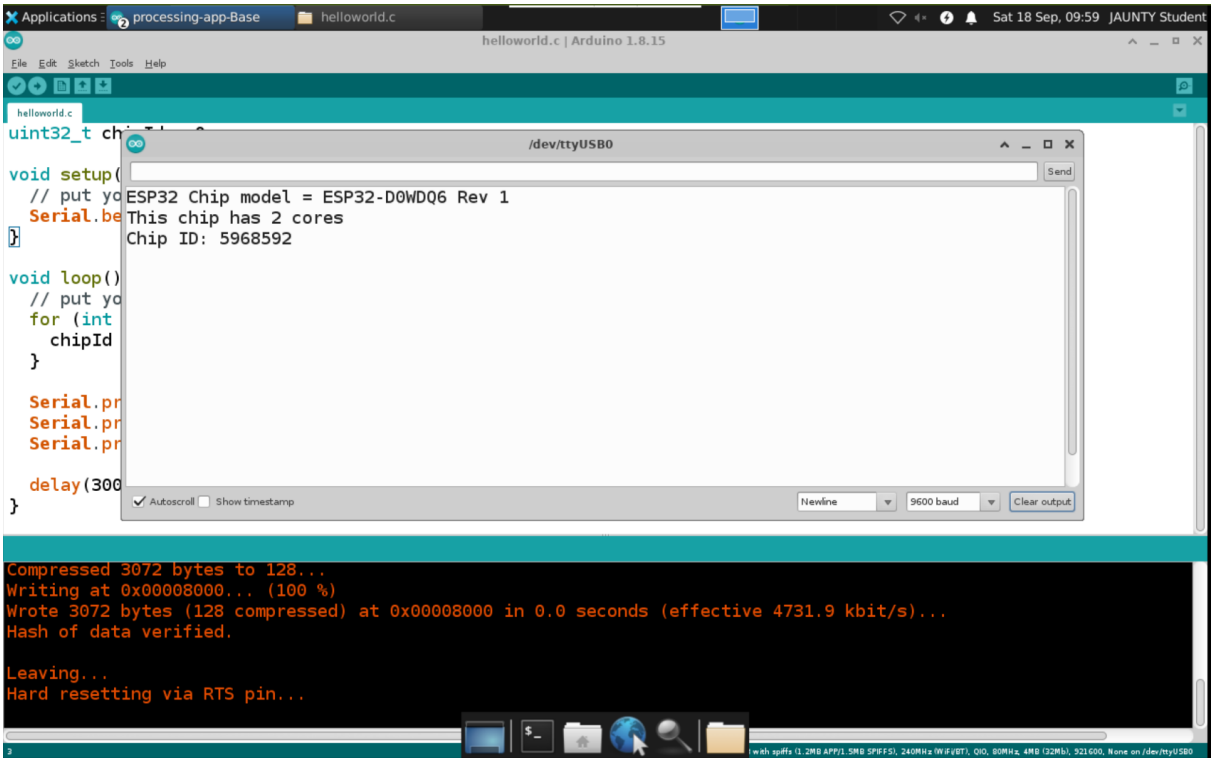


Figure 13: Serial output of the ESP32 sketch

Step 4: Deploy the MQTT broker

The MQTT broker is a service used for receiving messages from MQTT publishers and routing them to MQTT subscribers. In this step, a simpler MQTT broker will be deployed that receives unencrypted messages and uses TCP port 1883.

While MQTT broker can be installed as system service in most Linux distributions, the MQTT broker is going to be deployed as Docker container, using the docker-compose tool. The Docker technology enables OS-level virtualization, allowing the deployment of applications (docker containers) in an OS-agnostic manner, meaning that the underlying OS remains untouched, and the containerized services are isolated from the rest system.

First, create a new folder and inside that folder create a new file named “docker-compose.yml” with the following content:

```
1  version: "3"
2
3  services:
4    mqtt:
5      restart: always
6      hostname: mqtt-broker
7      container_name: mqtt- broker
8      image: amd64/eclipse-mosquitto
9      volumes:
10       - "./mosquitto.conf:/mosquitto/config/mosquitto.conf"
```

```
    ports:
      - "1883:1883"
```

In the same folder, create a new file named mosquitto.conf with the following content:

```
persistence false
listener 1883
allow_anonymous true
connection_messages true
log_type all
persistence false
```

You can deploy the MQTT broker container by issuing the command: `docker-compose up -d`

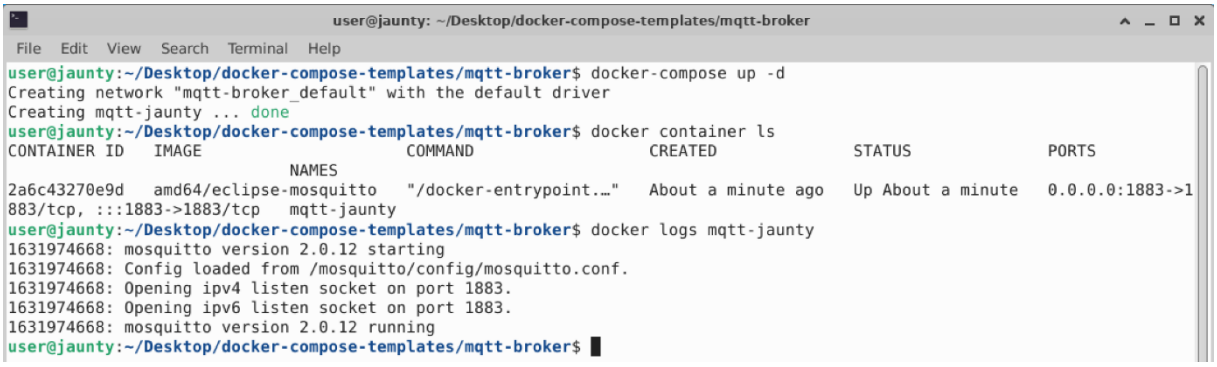


Figure 14: Deploying the MQTT broker

4. Exercise 2: Preparation of the I2C Task

As an RTOS, FreeRTOS provides an API for creating and managing tasks that are executed concurrently in a multi-tasking context. In this lab, you are going to implement two tasks that run in parallel, aiming to retrieve measurements via I2C and publish them to an MQTT broker. This exercise deals with the first part that concerns the retrieval of I2C measurements.

Step 1: The basic FreerRTOS code structure

The structure of the source code, while using the FreeRTOS kernel, is provided bellow:

```
// Includes
#include <freertos/FreeRTOS.h>

// Global variables

//Definition of FreeRTOS Tasks

void setup() {
  // Initialise Serial console
  // Initialise I2C bus
  // Create tasks
```

```
}

void loop() {
  // Any code that should run repeatedly
}
```

Initially, the include directives should include at least the FreeRTOS kernel, and the following libraries:

- `ArduinoJson.h`: A convenient library for creating and serialising JSON structures.
- `Adafruit_SHT4x.h`: A library by Adafruit to easily communicate with the SHT40 sensor. While in previous lab exercises, we used manual methods to retrieve I2C measurements, in this lab topic we will use the library from Adafruit that greatly facilitates the I2C communication.
- More libraries are going to be included in the upcoming exercises.

Right after, global variables are defined, which are accessed by the various tasks, as well the loop() and setup() functions.

Then, for each task, a respective void function is defined.

The setup() function is defined next, which undertakes the following:

- Initialises the serial console with a supported baud rate.
- Initialise the I2C bus, by verifying the availability of the I2C device and configuring additional parameters (e.g., precision).
- Create the FreeRTOS tasks.

Finally, the loop() function can execute any necessary code repeatedly. In the context of this lab, this function will not be utilised since all procedures will be undertaken by the FreeRTOS tasks.

Step 2: Global variables and initialisations

At this point, create a new project for this lab by navigating to “File > New”. Then, add the following source code.

```
1  #include <freertos/FreeRTOS.h>
2  #include <ArduinoJson.h>
3  #include "Adafruit_SHT4x.h"
4
5  #define JSON_SIZE 50
6  QueueHandle_t xQueueMeasurements = xQueueCreate(1,
   sizeof(StaticJsonDocument<JSON_SIZE>));
7  Adafruit_SHT4x sht4 = Adafruit_SHT4x();
```

Except FreeRTOS, the following libraries must be included:

- (Line 2): `ArduinoJson.h`: A convenient library for creating and serialising JSON structures. According to this library, each JSON structure is defined as a JsonDocument object or variants.
- (Line 3): `Adafruit_SHT4x.h`: A library by Adafruit to easily communicate with the SHT40 sensor. While in previous lab exercises, we used manual methods to retrieve I2C measurements, in this lab topic we will use the library from Adafruit that greatly facilitates the I2C communication.

Next, in line 5, the size of the JSON structure that will hold a single measurement instance should be defined. This information is required to define the structures that will temporarily hold the current

measurement. The following JSON structure is assumed to represent each measurement record (values are random):

```
{
  "temperature": 48.75608,
  "timestamp": 1351824120,
  "humidity": 23.30203
}
```

By providing this structure to the ArduinoJSON assistant [7], we conclude that the required size is 48 bytes. For flexibility, we choose to reserve 50 bytes instead.

Then, in line 6, the xQueue structure is defined, which is a queue that stores the latest measurement retrieved by the I2C sensor. Since the size and structure of the JSON document is known beforehand, we choose the StaticJsonDocument type. The `xQueueCreate()` function returns the queue handler, and accepts the following parameters:

- Queue length, i.e., how many items the queue can store. In our case, we need to store only the latest measurement, thus, queue length is 1.
- Length of each item in bytes.

Finally, line 7 initialises the C++ object that interacts with the I2C bus. Please note that the `sht4` object will be used later by the I2C task in order to interact with the I2C bus.

Step 3: Developing the I2C task

The I2C task undertakes to retrieve a measurement set from the sensor, construct the JSON document, and insert that object in the xQueue.
Create a new function before setup() and after the initialisation, with the following content:

```
1  void I2CTask(void* parameters) {
2    while (true) {
3      StaticJsonDocument<JSON_SIZE> newMeasurement;
4      sensors_event_t humidity, temp;
5      sht4.getEvent(&humidity, &temp);
6      uint32_t timestamp = millis();
7      newMeasurement["temp"] = temp.temperature;
8      newMeasurement["humidity"] = humidity.relative_humidity;
9      newMeasurement["timestamp"] = timestamp;
10     xQueueOverwriteFromISR(xQueueMeasurements, &newMeasurement, NULL);
11     Serial.printf("[I2CTask] Pushed i2C measurement to queue: temp: %f, Hum:
   %f, Time: %d\n", temp.temperature, humidity.relative_humidity, timestamp);
12     vTaskDelay(1000 / portTICK_PERIOD_MS);
13   }
14 }
```

The I2CTask consists of a single endless loop, which continuously retrieves the latest measurements from the SHT40 sensor, creates the corresponding StaticJsonDocument and writes that to the xQueue. In particular:

- A temporary variable is defined in line 3 (newMeasurement), which stores the StaticJsonDocument that is going to be saved to the xQueue.
- Temporary variables are defined for the retrieved humidity and temperature in line 4.
- The temporary variables are populated with the humidity and temperature measurements retrieved by the I2C bus, in line 5.
- In lines 7-9, the measurements are stored into the temporary StaticJsonDocument, while a timestamp is used in order to distinguish between older and newer measurements. The millis() function returns the milliseconds passed after boot.
- The `xQueueOverwriteFromISR()` function is used to write the StaticJsonDocument into the xQueue. "Overwrite" means that the function will write to the queue even if the queue is full, overwriting data that is already held in the queue. This function receives three parameters:
    - The queue handle to which the data is to be stored.
    - A pointer of the variable that will be written into the queue.
- The results are printed in line 11.
- The task is delayed for 1 second (1000 ticks divided by the duration of a tick in milliseconds (portTICK_PERIOD_MS) results to the ticks expressed in milliseconds).

5. Exercise 3: Preparation of the MQTT Publisher Task and WiFi configuration

Step 1: Global variables and initialisations

Revise the first part of the source code (initially developed in step 2 of exercise 2) as follows:

```
1  #include <ArduinoJson.h>
2  #include <AsyncMqttClient.h>
3  #include <AsyncTCP.h>
4  #include <freertos/FreeRTOS.h>
5  #include "Adafruit_SHT4x.h"
6  #include <WiFi.h>
7
8  // Global variables and definitions
9  #define JSON_SIZE 50
10 #define WIFI_SSID "*****"
11 #define WIFI_PASSWORD "*****"
12 #define MQTT_HOST IPAddress(192, 168, 21, 20)
13 #define MQTT_PORT 1883
14 #define MQTT_PUB_TOPIC "esp32/sht40"
15
16 QueueHandle_t xQueueMeasurements = xQueueCreate(1,
   sizeof(StaticJsonDocument<JSON_SIZE>));
```

```
17 Adafruit_SHT4x sht4 = Adafruit_SHT4x();
18 AsyncMqttClient mqttClient;
19 TimerHandle_t mqttReconnectTimer;
20 TimerHandle_t wifiReconnectTimer;
```

In addition to the previous exercise, the following additions have been incorporated:

- The AsyncMqttClient and AsyncTCP libraries are included, which provide the appropriate methods for asynchronous MQTT communication.
- The Wi-Fi library is included in line 6, to establish connectivity with a Wi-Fi access point.
- The Wi-Fi details are defined in lines 10-11 as constants (i.e., Wi-Fi SSID and password). Consult your lab instructor for the valid credentials.
- The MQTT details are defined in lines 12-14, including the IP address of the MQTT broker, the MQTT TCP port, and the topic where the measurements will be published. Consult your lab instructor for the valid information.
- The object representing the asynchronous MQTT client is created in line 18. While the object will be populated later in the setup() function, this initialisation makes the object accessible from the MQTT task.
- Two software timers are initialised, which are used to execute a function at a specific time in the future. The timers will be utilised in order to asynchronously connect to the Wi-Fi and the MQTT broker.

Step 2: Define the asynchronous MQTT and Wi-Fi call-backs

According to the asynchronous programming model, distinguishable units of work (defined as functions) run separately from the main application thread. Upon completion, they notify the main thread for their status, success, or failure. Asynchronous programming is often employed in networking to deal with the asynchronous nature of communications as well as to increase the efficiency, since blocking is minimised.

In this step, several callbacks will be implemented that enable ESP32 to asynchronously react to network events, e.g., disconnection from Wi-Fi.

Starting with the Wi-Fi connectivity, the following functions are defined:

connectToWiFi(): Establishes Wi-Fi connection with the given access point. Insert the following code right after the definitions of step 1:

```
23 void connectToWiFi() {
24   Serial.println("Connecting to Wi-Fi...");
25   WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
26 }
```

WiFiEvent(): This function is called asynchronously, upon various events related to Wi-Fi. Insert the following code right after connectToWiFi():

```
28 void WiFiEvent(WiFiEvent_t event) {
29   Serial.printf("[WiFi]: %d\n", event);
30   switch(event) {
31     case SYSTEM_EVENT_STA_GOT_IP:
32       Serial.println("WiFi connected");
33       Serial.println("IP address: ");
34       Serial.println(WiFi.localIP());
35       connectToMqtt();
36       break;
37     case SYSTEM_EVENT_STA_DISCONNECTED:
38       Serial.println("WiFi lost connection");
39       xTimerStop(mqttReconnectTimer, 0); // ensure we don't reconnect
   to MQTT while reconnecting to Wi-Fi
40       xTimerStart(wifiReconnectTimer, 0);
41       break;
42   }
43 }
```

Depending on the Wi-Fi event (retrieval of IP via Dynamic Host Configuration Protocol – DHCP or disconnection from access point), the following actions are taken:

- In case of receiving IP from DHCP (line 31), ESP32 tries to connect to MQTT via connectToMqtt() (this is a custom function that is defined later).
- In case of Wi-Fi disconnection (line 37), the MQTT reconnection timer is stopped (to avoid trying MQTT connection attempt, while Wi-Fi connection does not exist), and a new timer for Wi-Fi reconnection starts.

Proceeding with the MQTT functions, the following are defined:

connectToMqtt(): Attempts connection with an MQTT broker. Please note that the connection parameters (broker IP and port) will be configured in the setup() function. Insert the following code right after:

```
46 void connectToMqtt() {
47   Serial.println("Connecting to MQTT...");
48   mqttClient.connect();
49 }
```

onMqttConnect(): This is a callback function that is called when an MQTT session is established. Insert the following code right after connectToMqtt():

```
51 void onMqttConnect(bool sessionPresent) {
52   Serial.println("Connected to MQTT.");
53   Serial.print("Session present: ");
54   Serial.println(sessionPresent);
```

```
55  }
```

onMqttDisconnect() : This is a callback function that is called upon the client disconnects from the MQTT broker. In this case, if Wi-Fi is connected, then the MQTT reconnection timer is started immediately. Insert the following code right after onMqttConnect():

```
57  void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
58    Serial.println("Disconnected from MQTT.");
59    if (WiFi.isConnected()) {
60      xTimerStart(mqttReconnectTimer, 0);
61    }
62  }
```

onMqttPublish(): This is a callback function that is called upon a new message is published to the MQTT broker. Insert the following code right after onMqttDisconnect():

```
64  void onMqttPublish(uint16_t packetId) {
65    Serial.print("Publish acknowledged.");
66    Serial.print("  packetId: ");
67    Serial.println(packetId);
68  }
```

Step 3: Define the MQTT Publisher Task

The MQTT publisher task undertakes to retrieve the StaticJsonDocument measurement from the xQueue, serialise it and publish the character sequence to the MQTT broker. Insert the following code right after onMqttPublish():

```
91  void MQTTPublisherTask(void* parameters) {
92    StaticJsonDocument<JSON_SIZE> measurementToSend;
93    char serialisedJSON[JSON_SIZE];
94    while (true) {
95      if (xQueueReceiveFromISR(xQueueMeasurements, &measurementToSend, NULL) == pdPASS) {
96        serializeJson(measurementToSend, serialisedJSON);
97        uint16_t packetIdPub1 = mqttClient.publish(MQTT_PUB_TOPIC, 0, true, serialisedJSON);
98        Serial.printf("[MQTTTask] Publishing packetId: %i, on topic %s. Content: %s\n", packetIdPub1, MQTT_PUB_TOPIC, serialisedJSON);
99      } else {
100       Serial.println("[MQTTTask] Failed to receive from Queue. Try again after 1 second");
101     }
102     vTaskDelay(1000 / portTICK_PERIOD_MS);
103   }
104 }
```

First, line 92 initialises the variable that caches the current content of the xQueue. Line 93 initialises the character array which temporarily stores the serialialised content of the retrieved StaticJsonDocument.

After initialising the local variables, MQTTPublisherTask() tries repeatedly to retrieve the contents of the xQueue (line 95). If the operation is successful, the content is serialised (line 96) and then is published to the MQTT broker (line 96).

Finally, the results are printed to the console (line 98) and the task is blocked for 1 second (line 102).

6. Exercise 4: Multitasking with FreeRTOS and Verification

After finalising the I2C and MQTT tasks as well as the callback and asynchronous functions, this exercise develops the setup() function that configures the multiple tasks. Moreover, the MQTT application on ESP32 will be validated by developing an MQTT subscriber and verifying that the published messages are successfully received.

Step 1: Develop the setup() function

Add the following code at the end of your source code:

```
106 // Setup function - Executed only once
107 void setup() {
108   Serial.begin(57600);
109   WiFi.disconnect(true);
110
111   // Initialise I2C bus
112   while (true) {
113     if (! sht4.begin()) {
114       Serial.println("Couldn't find SHT4x. Trying again in 100");
115       delay(10000);
116     } else {
117       Serial.println("Found SHT4x sensor");
118       sht4.setPrecision(SHT4X_LOW_PRECISION);
119       sht4.setHeater(SHT4X_NO_HEATER);
120       Serial.print("Serial number 0x");
121       Serial.println(sht4.readSerial(), HEX);
122       break;
123     }
124   }
125
126   mqttReconnectTimer = xTimerCreate("mqttTimer", pdMS_TO_TICKS(2000),
pdFALSE, (void*)0,
reinterpret_cast<TimerCallbackFunction_t>(connectToMqtt));
127   wifiReconnectTimer = xTimerCreate("wifiTimer", pdMS_TO_TICKS(2000),
pdFALSE, (void*)0,
reinterpret_cast<TimerCallbackFunction_t>(connectToWiFi));
128
129   WiFi.onEvent(WiFiEvent);
130
131   //Initialise MQTT callbacks
132   mqttClient.onConnect(onMqttConnect);
133   mqttClient.onDisconnect(onMqttDisconnect);
134   mqttClient.onPublish(onMqttPublish);
135   mqttClient.setServer(MQTT_HOST, MQTT_PORT);
```

```
136
137   xTaskCreate(
138     I2CTask, // function name
139     "I2CTask", //task name
140     10000, //stack size
141     NULL, // task parameters
142     1, //task priority
143     NULL // task handle
144   );
145
146   xTaskCreate(
147     MQTTPublisherTask, // function name
148     "MQTTPublisherTask", //task name
149     10000, //stack size
150     NULL, // task parameters
151     1, //task priority
152     NULL // task handle
153   );
154
155   connectToWiFi();
156 }
157
158 void loop() {
159   // put your main code here, to run repeatedly:
160 }
```

The setup() function begins by initialising the baud rate of the serial console output at 57600 bps (line 108). Then, it is ensured that Wi-Fi is disconnected by calling the disconnect() function of the WiFi instance (line 109). This is a recommended practice to ensure that the client is not associated with the access point when trying to establish connection for the first time.

The I2C bus is initialised right after, in lines 112-24. In particular, ESP32 repeatedly searches for an Adafruit SHT4x sensor by probing the I2C bus (line 113). When a sensor is detected, the precision and heater configuration parameters are set (lines 118-119), while the serial number is printed to the console.

The MQTT and Wi-Fi reconnection timers are initialised in lines 126 and 127 respectively. A timer schedules a function to be executed at a specific time in the future. The xTimerCreate() API function requires the following parameters:

- A descriptive name of the timer, useful only for debugging purposes.
- The period of the timer in ticks. When this period is expired, a callback function will be executed (passed also as an argument). The pdMS_TO_TICKS() macro is used to convert milliseconds to ticks.
- Whether the timer should restart each time it expires. pdFALSE denotes that the timer is executed only once and expires forever.
- A unique identifier for the timer.
- The last input is the callback function that is executed when the timer expires.

In lines 129-134, the asynchronous callbacks are configured for all Wi-Fi and MQTT events. In particular:

- For any Wi-Fi event, the WiFiEvent() function will be called.
- When an MQTT connection is established, the onMqttConnect() function will run.
- When the client/publisher disconnects from the MQTT broker, the onMqttDisconnect() function will run.
- When the client publishes a new message, the onMqttPublish() function will run.

The MQTT broker details are configured on the mqttClient instance (line 135).

Finally, xTaskCreate() is utilised to create the I2CTask and MQTTPublisherTask.

Step 2: Run the snippet

Follow the guidelines of Exercise 1 / Step 3 to run the snippet. Verify that the results are printed to the console output.

Step 3: Verifying with MQTT subscriber

In this step, an MQTT subscriber will be developed in Python, to verify that the messages are successfully published.

Prepare a new Python environment on any Windows or Linux-based system and install the paho-mqtt package via pip.

After preparing the python environment, create a new file and insert the following source code, which implements an MQTT client that subscribes to the MQTT broker and receives the published messages.

```
1  import paho.mqtt.client as mqtt_client
2
3
4  mqttClient = mqtt_client.Client("jaunty-client")
5
6  def on_connect(client, userdata, flags, rc):
7      if rc == 0:
8          print("Connected to broker")
9          mqttClient.subscribe("esp32/sht40")
10     else:
11         print("Connection failed")
12
13
14 def on_message(client, userdata, message):
15     payload = message.payload.decode()
16     print("Message received: " + payload)
```

```
17
18
19  if __name__ == '__main__':
20      mqttClient.on_connect = on_connect
21      mqttClient.on_message = on_message
22      mqttClient.connect("***", 1883)
23      mqttClient.loop_forever()
```

After importing the necessary Python libraries (paho-mqtt), the MQTT client object is initialised in line 4. The constructor requires a string as input, which denotes the unique ID of the client.

By following the asynchronous programming paradigm, the on_connect() and on_message() callback functions are defined in lines 6 and 14 respectively, which are called upon a connection is established or a new message is received.

The main function implements the following:

- The callback functions (previously declared) are assigned to the corresponding asynchronous events, in lines 20 and 21 respectively.
- The IP address and TCP port of the MQTT broker are configured in line 22. Please replace the wildcard characters with the IP address provided by your lab instructor.
- The loop_forever() function (line 23) blocks the program and runs the callbacks depending on the incoming events.

Run the Python script and verify that the MQTT messages of ESP32 are successfully received.

7. Appendix: Source code of the ESP32 application

```
#include <ArduinoJson.h>
#include <AsyncMqttClient.h>
#include <AsyncTCP.h>
#include <freertos/FreeRTOS.h>
#include "Adafruit_SHT4x.h"
#include <WiFi.h>

// Global variables and definitions
#define JSON_SIZE 50  //https://arduinojson.org/v6/assistant/
#define WIFI_SSID "***"
#define WIFI_PASSWORD "***"
#define MQTT_HOST IPAddress(192, 168, 10, 20)
#define MQTT_PORT 1883
#define MQTT_PUB_TOPIC "esp32/sht40"

QueueHandle_t xQueueMeasurements = xQueueCreate(1,
sizeof(StaticJsonDocument<JSON_SIZE>));
```

```
Adafruit_SHT4x sht4 = Adafruit_SHT4x();
AsyncMqttClient mqttClient;
TimerHandle_t mqttReconnectTimer;
TimerHandle_t wifiReconnectTimer;

// Asynchronous WiFi functions
void connectToWiFi() {
  Serial.println("Connecting to Wi-Fi...");
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}

void WiFiEvent(WiFiEvent_t event) {
  Serial.printf("[WiFi]: %d\n", event);
  switch(event) {
    case SYSTEM_EVENT_STA_GOT_IP:
      Serial.println("WiFi connected");
      Serial.println("IP address: ");
      Serial.println(WiFi.localIP());
      connectToMqtt();
      break;
    case SYSTEM_EVENT_STA_DISCONNECTED:
      Serial.println("WiFi lost connection");
      xTimerStop(mqttReconnectTimer, 0); // ensure we don't
reconnect to MQTT while reconnecting to Wi-Fi
      xTimerStart(wifiReconnectTimer, 0);
      break;
  }
}

// Asynchronous MQTT functions
void connectToMqtt() {
  Serial.println("Connecting to MQTT...");
  mqttClient.connect();
}

void onMqttConnect(bool sessionPresent) {
  Serial.println("Connected to MQTT.");
  Serial.print("Session present: ");
  Serial.println(sessionPresent);
}

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
  Serial.println("Disconnected from MQTT.");
  if (WiFi.isConnected()) {
    xTimerStart(mqttReconnectTimer, 0);
  }
}

void onMqttPublish(uint16_t packetId) {
  Serial.print("Publish acknowledged.");
```

```
  Serial.print("  packetId: ");
  Serial.println(packetId);
}

void I2CTask(void* parameters) {
  while (true) {
      StaticJsonDocument<JSON_SIZE> newMeasurement;

      sensors_event_t humidity, temp;
      sht4.getEvent(&humidity, &temp);

      uint32_t timestamp = millis();

      newMeasurement["temp"] = temp.temperature;
      newMeasurement["humidity"] = humidity.relative_humidity;
      newMeasurement["timestamp"] = timestamp;

      xQueueOverwriteFromISR(xQueueMeasurements, &newMeasurement,
NULL);

      Serial.printf("[I2CTask] Pushed i2C measurement to queue:
temp: %f, Hum: %f, Time: %d\n", temp.temperature,
humidity.relative_humidity, timestamp);

      vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}

void MQTTPublisherTask(void* parameters) {
  StaticJsonDocument<JSON_SIZE> measurementToSend;
  char serialisedJSON[JSON_SIZE];
  for (;;) {
    if (xQueueReceiveFromISR(xQueueMeasurements, &measurementToSend,
NULL) == pdPASS) {
      serializeJson(measurementToSend, serialisedJSON);
      uint16_t packetIdPub1 = mqttClient.publish(MQTT_PUB_TOPIC, 0,
true, serialisedJSON);
      Serial.printf("[MQTTTask] Publishing packetId: %i, on topic
%s. Content: %s\n", packetIdPub1, MQTT_PUB_TOPIC, serialisedJSON);
    } else {
      Serial.println("[MQTTTask] Failed to receive from Queue. Try
again after 10 ticks");
    }
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}

// Setup function - Executed only once
void setup() {
  Serial.begin(57600);
  WiFi.disconnect(true);
```

```
  // Initialise I2C bus
  while (true) {
    if (! sht4.begin()) {
      Serial.println("Couldn't find SHT4x. Trying again in 100");
      delay(10000);
    } else {
      Serial.println("Found SHT4x sensor");
      sht4.setPrecision(SHT4X_LOW_PRECISION);
      sht4.setHeater(SHT4X_NO_HEATER);
      Serial.print("Serial number 0x");
      Serial.println(sht4.readSerial(), HEX);
      break;
    }
  }

  mqttReconnectTimer = xTimerCreate("mqttTimer",
pdMS_TO_TICKS(2000), pdFALSE, (void*)0,
reinterpret_cast<TimerCallbackFunction_t>(connectToMqtt));
  wifiReconnectTimer = xTimerCreate("wifiTimer",
pdMS_TO_TICKS(2000), pdFALSE, (void*)0,
reinterpret_cast<TimerCallbackFunction_t>(connectToWiFi));

  WiFi.onEvent(WiFiEvent);

  //Initialise MQTT callbacks
  mqttClient.onConnect(onMqttConnect);
  mqttClient.onDisconnect(onMqttDisconnect);
  mqttClient.onPublish(onMqttPublish);
  mqttClient.setServer(MQTT_HOST, MQTT_PORT);

  xTaskCreate(
    I2CTask, // function name
    "I2CTask", //task name
    10000, //stack size
    NULL, // task parameters
    1, //task priority
    NULL // task handle
  );

  xTaskCreate(
    MQTTPublisherTask, // function name
    "MQTTPublisherTask", //task name
    10000, //stack size
    NULL, // task parameters
    1, //task priority
    NULL // task handle
  );

  connectToWiFi();
```

```
}

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

8. Exam Questions

1. What is the difference between a Raspberry Pi 4B and an ESP32?
2. What is the difference between the synchronous and the asynchronous communication model?
3. What information would you need to receive the data stream generated by an MQTT publisher?
4. What are the main components of the MQTT architecture?

References

[1] microcontrollerslab.com, “ESP32 Pinout – How to use GPIO pins?,” [Online]. Available: https://microcontrollerslab.com/esp32-pinout-use-gpio-pins/. [Accessed 28 April 2022].

[2] S. Santos, “ESP32 Troubleshooting Guide,” RandomNerdTutorials, [Online]. Available: https://randomnerdtutorials.com/esp32-troubleshooting-guide/. [Accessed 28 April 2022].

[3] Microcontrollerslab.com, “FreeRTOS Scheduler: Learn to Configure Scheduling Algorithm,” [Online]. Available: https://microcontrollerslab.com/freertos-scheduler-learn-to-configure-scheduling-algorithm/. [Accessed 28 April 2022].

[4] ISO/IEC JTC 1 Information technology, “ISO/IEC 20922:2016 (Information technology — Message Queuing Telemetry Transport (MQTT) v3.1.1),” June 2016. [Online]. Available: https://www.iso.org/standard/69466.html. [Accessed 28 April 2022].

[5] Amazon Web Services (AWS), “Pub/Sub Messaging - Asynchronous event notifications,” [Online]. Available: https://aws.amazon.com/pub-sub-messaging/. [Accessed 28 April 2022].

[6] Arduino, “Arduino IDE 1 Installation (Linux),” 28 April 2022. [Online]. Available: https://docs.arduino.cc/software/ide-v1/tutorials/Linux. [Accessed 28 April 2022].

[7] ArduinoJson, “ArduinoJson Assistant,” [Online]. Available: https://arduinojson.org/v6/assistant/. [Accessed 28 April 2022].

9. Contacts

Project Coordinator:

- Name: Technical University of Sofia
- Address:
    - Technical University of Sofia,
      Kliment Ohridsky Bd 8
      1000, Sofia, Bulgaria
- Phone: +3592623073

Output 2 Leader:

- Name: FOSS Research Centre for Sustainable Energy, University of Cyprus
- Address:
    - University of Cyprus,
      Panepistimiou 1 Avenue
      P.O. Box 20537
      1678, Nicosia, Cyprus
- Email: foss@ucy.ac.cy
- Phone: +357 22 894288