EFY TESTED Ashwini K. Sinha

IoT DEVICE HEALTH MONITORING Over MQTT

PRADIPTA GHOSH AND CHALLA NAGA SAI RAVALI

IoT devices are widely used to monitor and control physical systems such as lights, fans, motors, and industrial equipment. This system extends that capability by monitoring the health and operational status of the IoT device itself using the ESP32-C3 Super Mini. The ESP32-C3 continuously monitors key parameters, including CPU usage, RAM utilisation, Wi-Fi signal strength (RSSI), chip temperature, flash memory usage, reset reason, device uptime, and Wi-Fi channel. These telemetry values are periodically published to a public MQTT broker for real-time visualisation on a smartphone dashboard. Bidirectional MQTT communication also enables remote control of the onboard LED and software resets. The system provides a practical, low-cost solution for remote diagnostics, preventive maintenance, and performance monitoring of embedded IoT devices.

As IoT deployments continue to grow, remote health monitoring has become increasingly important. Continuous telemetry helps identify issues such as memory leaks, poor network connectivity, overheating, and unexpected resets before they result in device failure. This improves system reliability while reducing maintenance effort.

With built-in Wi-Fi, low power consumption, and adequate processing capability, the ESP32-C3 Super Mini is well-suited for a compact health monitoring node. Combined with the lightweight MQTT protocol, it provides an efficient and scalable solution for real-time IoT device monitoring and remote management. Fig. 1 shows the ESP32-C3 IoT device health monitoring system, while the components required to build the system are listed in Table 1.

Fig. 1: Authors' prototype setup for IoT device health monitoring

TABLE 1
BILL OF MATERIALS

Component	Quantity
ESP32-C3 super mini	1
USB Type-C cable	1
Breadboard (optional)	1
Jumper wires	As required
LED (onboard LED may be used)	1
Wi-Fi-enabled smartphone	1
Laptop/PC for programming	1

```
const char* ssid = "Galaxy_5GMW_4325";
const char* password = "xghm1846";

const char* telemetry_topic = "esp32c3/telemetry";
const char* led_control_topic = "esp32c3/led/control";
const char* led_status_topic = "esp32c3/led/status";
const char* esp32c3_reset_topic = "esp32c3/reset";
const char* uptime_topic = "esp32c3/uptime";
```

Fig. 2: Wi-Fi credentials and MQTT topic configuration in the ESP32-C3 source code

TABLE 2
MQTT TOPIC STRUCTURE

Message Description	MQTT Topic	Published By	Subscribed By
Telemetry	esp32c3/telemetry	ESP32-C3 super mini	IoT MQTT panel app
LED control	Esp32c3/led/control	IoT MQTT panel app	ESP32-C3 super mini
LED status	Esp32c3/led/status	ESP32-C3 super mini	IoT MQTT panel app
Remote reset	Esp32c3/reset	IoT MQTT panel app	ESP32-C3 super mini
Formatted uptime	Esp32c3/uptime	ESP32-C3 super mini	IoT MQTT panel app

EFY note. *Since this system uses a public MQTT broker, MQTT topic name conflicts may occur. To avoid collisions, append initials, a system identifier, or another unique suffix to each MQTT topic.*

MQTT broker configuration

This system uses the Switchsys public MQTT broker, a free MQTT

broker suitable for learning, testing, and IoT prototyping. It supports standard MQTT clients over TCP on port 1883 and does not require user authentication.

Switchsys MQTT broker details:
Broker URL: mqtt.switchsys.in
Port: 1883 (TCP, non-TLS)

Table 2 shows the MQTT topic structure used in this system for communication between the ESP32-C3 Super Mini and the IoT MQTT Panel application.

Example of a telemetry payload.

```
{
 "cpu_usage": 21.4,
 "ram_usage": 34.7,
 "rssi": -56,
 "chip_temperature": 45.8,
 "last_reset_reason": "POWER_ON",
 "wifi_channel": 1,
 "flash_usage": 23.56
}
```

Working with the Switchsys MQTT broker

The ESP32-C3 Super Mini connects to the local Wi-Fi network and establishes an MQTT session with the Switchsys public MQTT broker. Once connected, the device periodically publishes health and system telemetry while subscribing to MQTT topics for remote control commands.

Telemetry published by the ESP32-C3 super mini.

1. CPU usage (%)
2. RAM usage (%)
3. Wi-Fi signal strength (RSSI in dBm)
4. Chip temperature (°C)
5. Flash memory usage (%)
6. Reset reason
7. Wi-Fi channel (CHx)
8. Device uptime (Days:Hours:Minutes:Seconds)
9. Remote LED status

Control commands subscribed by the ESP32-C3 super mini.

1. Remote LED control (on/off)
2. Remote ESP32 reset

Software configuration

Before uploading the program to the ESP32-C3 Super Mini, update the following parameters in the source code:

1. Wi-Fi SSID and password. Replace these with the credentials of the local Wi-Fi network.

2. MQTT topic names (optional). Modify the topic names if required. When using a public MQTT broker, unique topic names are recommended to prevent conflicts with topics used by other users. Appending initials, a system identifier, or another unique suffix to each topic helps avoid naming collisions.

In this system, the code is configured to publish telemetry data, LED status, and uptime information to the MQTT broker while subscribing to topics for remote LED control and device reset commands. Fig. 2 shows the Wi-Fi credentials and MQTT topic configuration defined in the ESP32-C3 source code.

TABLE 3
TYPICAL WI-FI SIGNAL STRENGTH RANGES

RSSI Value (dBm)	Signal Quality
-30	Excellent
-60	Good
-89	Weak

TABLE 4
CHIP TEMPERATURE

Temperature	Condition
20 to 50°C	Normal
50 to 70°C	Warm but safe
>80°C	Risk of instability

```
float getCpuUsage() {
 static char stats[512];
 vTaskGetRunTimeStats(stats);

 char* idlePtr = strstr(stats, "IDLE");
 if (!idlePtr) return -1;

 int idlePercent = 0;
 sscanf(idlePtr, "%*s %*d %d%%", &idlePercent);
 return 100.0 - idlePercent;
}
```

Fig. 3: Function to calculate CPU usage by measuring the ESP32-C3 idle task runtime

```
float getRamUsagePercent() {
 uint32_t totalHeap = ESP.getHeapSize();
 uint32_t freeHeap = ESP.getFreeHeap();

 float usedPercent = ((float)(totalHeap - freeHeap) / totalHeap) * 100.0;

 if (usedPercent < 0) usedPercent = 0;
 if (usedPercent > 100) usedPercent = 100;

 return usedPercent;
}
```

Fig. 4: Function to calculate RAM usage percentage using ESP32-C3 heap memory

```
int rssi = WiFi.RSSI();
```

Fig. 5: The code used to read the Wi-Fi signal strength (RSSI)

```
float getChipTempAvg(uint8_t samples = 5) {
 float sum = 0;
 for (uint8_t i = 0; i < samples; i++) {
  sum += temperatureRead();
  delay(10);
 }
 return sum / samples;
}
```

Fig. 6: Function to calculate the average on-chip temperature of the ESP32-C3

CPU usage (%)

CPU usage indicates the percentage of the ESP32-C3 Super Mini's processing capacity currently being used by running tasks. The ESP32 operates on the FreeRTOS real-time operating system (RTOS), which manages multiple tasks simultaneously.

CPU utilisation is determined by measuring the processor's idle time. Whenever no application tasks are running, the FreeRTOS idle task executes. By monitoring the runtime of the idle task, the system estimates the percentage of time the processor is idle. The getCpuUsage() function retrieves the FreeRTOS runtime statistics, identifies the idle task, determines its idle percentage, and calculates CPU usage using the following equation:

```
CPU usage (%)=100-Idle Time (%)
```

Fig. 3 shows the getCpuUsage() function used to calculate ESP32-C3

TABLE 5
COMMON ESP32-C3 RESET CAUSES

Reset Type	Description
POWER_ON	Device powered on normally
SOFTWARE_RESET	Reset initiated by the firmware or software
WATCHDOG	Reset triggered by the watchdog timer due to system unresponsiveness
DEEP_SLEEP_WAKE	The device woke up from deep sleep mode
BROWNOUT	Reset caused by a drop in the supply voltage below the safe operating level
PANIC	Reset caused by a fatal runtime exception or system crash
Other	Any other reset reason not covered by the above categories

```
float getFlashUsedPercent() {
  uint32_t flashSize = ESP.getFlashChipSize();
  uint32_t sketchSize = ESP.getSketchSize();

  if (flashSize == 0) return 0;

  return ((float)sketchSize / flashSize) * 100.0;
}
```

Fig. 7: Function used to calculate the percentage of flash memory

```
const char* getResetReasonText() {
 switch (esp_reset_reason()) {
  case ESP_RST_POWERON: return "POWER_ON";
  case ESP_RST_SW: return "SOFTWARE_RESET";
  case ESP_RST_WDT: return "WATCHDOG";
  case ESP_RST_DEEPSLEEP: return "DEEP_SLEEP_WAKE";
  case ESP_RST_BROWNOUT: return "BROWNOUT";
  case ESP_RST_PANIC: return "PANIC";
  default: return "OTHER";
 }
}
```

Fig. 8: Function to identify and return the ESP32-C3 reset reason

```
String wifiChText = "CH" + String(WiFi.channel());
```

Fig. 9: Function to retrieve the Wi-Fi channel

CPU usage from the FreeRTOS idle task runtime statistics.

RAM usage (%)

RAM usage indicates the percentage of the ESP32-C3 Super Mini's dynamic memory (heap) currently occupied by the firmware during operation. The heap is used to allocate memory for variables, buffers, network communication, and other runtime processes. RAM utilisation is calculated using the following equation:

```
RAM usage (%)=((total heap - free heap)
/total heap)×100
```

where total heap is the total heap memory available to the application, and free heap is the heap memory currently available for allocation.

The getRamUsagePercent() function reads the total and available heap memory, calculates the percentage currently in use, constrains the result to the valid range of 0-100%, and returns the RAM usage percentage. Fig. 4 shows the getRamUsagePercent() function used to calculate ESP32-C3 heap memory utilisation.

RSSI dBm

RSSI (received signal strength indicator) represents the strength of the Wi-Fi signal received by the ESP32-C3. It is measured in dBm (decibels referenced to one milliwatt), where values closer to 0dBm indicate a stronger signal and more negative values indicate a weaker signal. Table 3 shows the typical Wi-Fi signal strength ranges.

Monitoring RSSI remotely helps determine whether communication problems are caused by poor Wi-Fi signal strength. The WiFi.RSSI() function retrieves the current RSSI value of the connected wireless network, enabling the ESP32-C3 to continuously monitor the quality of its Wi-Fi connection. Fig. 5 shows the code used to read the Wi-Fi signal strength (RSSI) of the connected network.

```
uint32_t uptime = millis() / 1000;
String uptimeDHMS = getUptimeDHMS(uptime);
.............
.............

String getUptimeDHMS(uint32_t seconds) {
  uint32_t days = seconds / 86400;
  seconds %= 86400;

  uint8_t hours = seconds / 3600;
  seconds %= 3600;

  uint8_t minutes = seconds / 60;
  uint8_t secs = seconds % 60;

  char buffer[20];
  sprintf(buffer, "%03lu:%02u:%02u:%02u",
        days, hours, minutes, secs);

  return String(buffer);
}
```

Fig. 10: Function to calculate and format the ESP32-C3 uptime

```
void publishLedStatus() {
  if (ledState) {
    client.publish(led_status_topic, "ON", true);
  } else {
    client.publish(led_status_topic, "OFF", true);
  }
}
```

Fig. 11: Function to publish the LED on/off status to the MQTT broker

```
  if (String(topic) == led_control_topic) {

    if (message == "ON") {
      digitalWrite(LED_PIN, LOW);
      ledState = true;
      publishLedStatus();
    } else if (message == "OFF") {
      digitalWrite(LED_PIN, HIGH);
      ledState = false;
      publishLedStatus();
    }
  }
```

Fig. 12: Callback function to receive MQTT commands and control the onboard LED

```
if (String(topic) == esp32c3_reset_topic) {
  if (message == "RESET") {
    delay(1000);
    ESP.restart();
  }
}
```

Fig. 13: Callback function to receive the MQTT reset command

Chip temperature (°C)

The ESP32-C3 includes an internal temperature sensor that provides an

approximate measurement of the chip temperature. Although it is not intended for precise ambient temperature measurement, it is useful for monitoring temperature trends and detecting overheating during operation. Table 4 lists the measured chip temperature ranges.

The getChipTempAvg() function samples the internal temperature sensor multiple times, averages the readings to minimise fluctuations, and returns a more stable estimate of the chip temperature. Fig. 6 shows the getChipTempAvg() function used to calculate the average on-chip temperature of the ESP32-C3.

Flash usage (%)

Flash usage represents the size of the compiled firmware relative to the total available flash memory of the ESP32-C3. The flash usage percentage is calculated as:

```
Flash usage (%)=(Used flash/total flash)
×100
```

The getFlashUsedPercent() function reads the total flash chip size and the compiled sketch size before calculating the percentage of flash memory occupied by the firmware. Monitoring flash usage helps ensure that sufficient flash memory remains available for future firmware updates and additional application features. Fig. 7 shows the function used to calculate ESP32-C3 firmware flash usage.

Reset reason

The reset reason indicates why the ESP32-C3 restarted. Monitoring this information helps identify whether the reset was caused by normal operation or an unexpected event. Displaying the reset reason on the IoT dashboard enables remote diagnosis of system failures and improves troubleshooting efficiency. Table 5 lists the common ESP32-C3 reset causes, while Fig. 8 shows the function used to identify and return the reset reason.

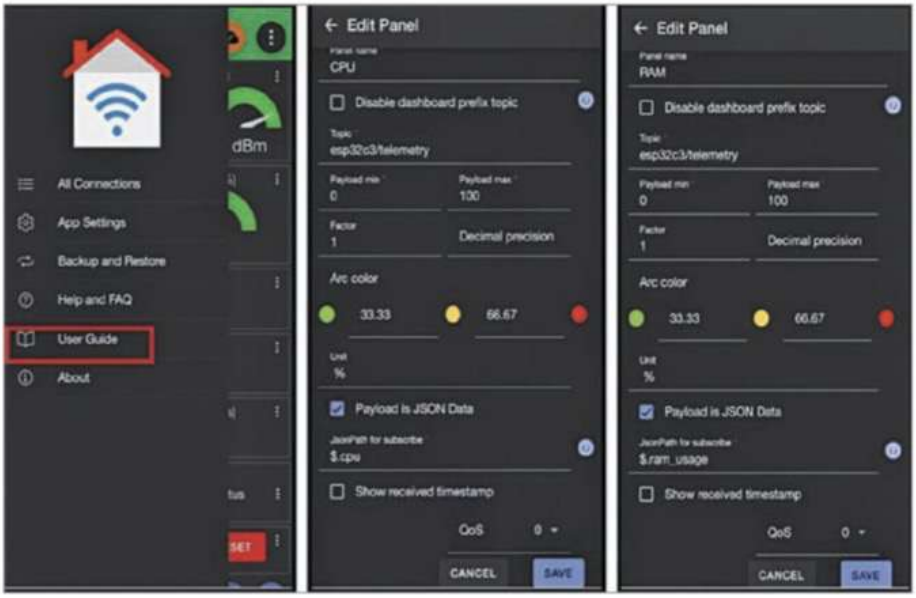

Fig. 14: Accessing the IoT MQTT panel user guide and configuring the CPU and RAM telemetry panels

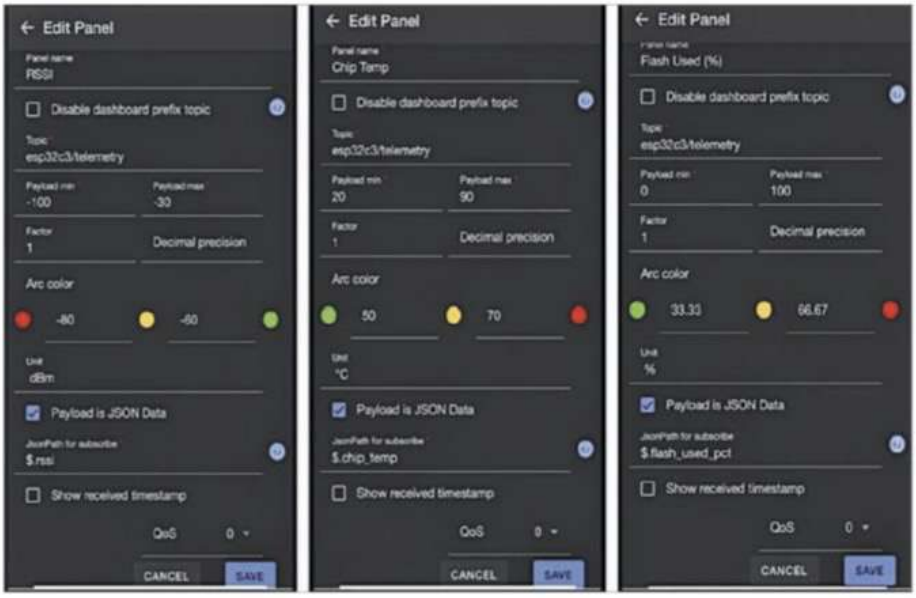

Fig. 15: Configuration of the RSSI, chip temperature, and flash usage telemetry panels in the IoT MQTT panel app

Wi-Fi channel (CHx)

The Wi-Fi channel indicates the frequency channel used by the wireless access point to which the ESP32-C3 Super Mini is connected. For 2.4GHz Wi-Fi networks, the channel number typically ranges from 1 to 13 (or 1 to 14, depending on the region). Monitoring the Wi-Fi channel helps identify network congestion, interference from nearby Wi-Fi networks, and channel overlap that may affect communication performance. Fig. 9 shows the function used to retrieve the connected Wi-Fi channel.

Uptime (Days: Hours: Minutes: Seconds)

Uptime indicates how long the ESP32-C3 has been running continuously since its last reset. It is calculated using the internal system timer (millis()), which measures the elapsed time since the microcontroller was powered on or restarted. The uptime is formatted as DDD:HH:MM:SS (Days:Hours:Minutes:Seconds).

Example: 002:05:14:22 (2 days, 5 hours, 14 minutes, and 22 seconds)

Monitoring uptime helps verify

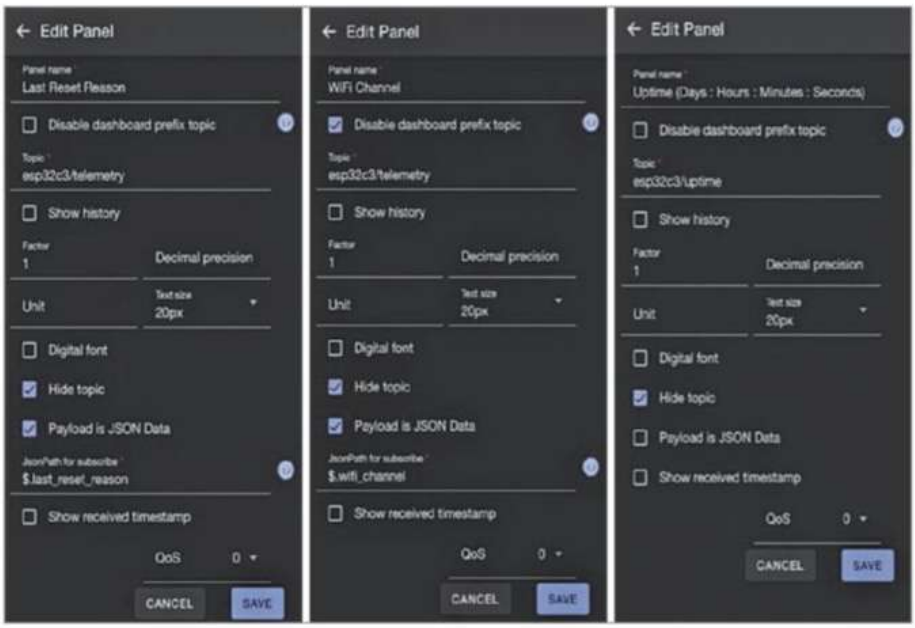

Fig. 16: Configuration of the last reset reason, Wi-Fi channel, and uptime telemetry panels in the IoT MQTT Panel app

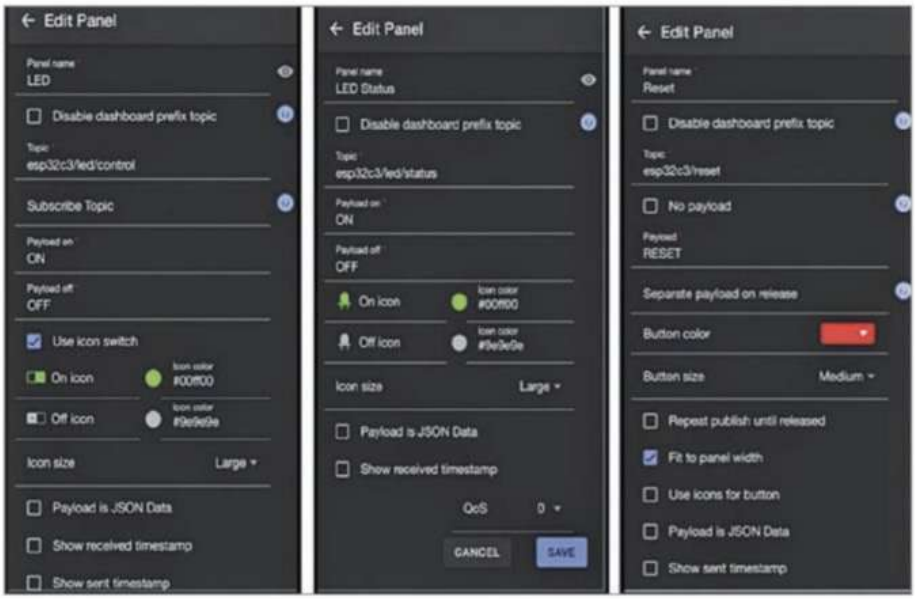

Fig. 17: Configuration of the LED control, LED status, and reset control panels in the IoT MQTT panel app

system stability and reliability while detecting unexpected resets or power interruptions. Fig. 10 shows the function used to calculate and format the ESP32-C3 uptime in DDD:HH:MM:SS format.

Remote LED status

The device publishes the LED status to the MQTT broker, ensuring that the dashboard always reflects the actual hardware state, even after a restart or a remote command. Fig. 11 shows the function used to publish the LED on/off status to the MQTT broker.

Remote LED control (on/off)

The dashboard enables remote control of the ESP32-C3 Super Mini's onboard LED through MQTT messages. The ESP32-C3 subscribes to the LED control topic and updates the LED state whenever an on or off command is received. After changing the LED state, the device publishes the updated status to the dashboard, ensuring that

Fig. 18: IoT MQTT Panel dashboard displaying real-time ESP32-C3 Super Mini telemetry, onboard LED control, LED status, and remote reset functionality

the displayed state always matches the actual hardware state. Fig. 12 shows the MQTT callback function used to receive LED control commands and operate the onboard LED.

Note. The ESP32-C3 Super Mini's onboard LED is connected to GPIO 8 and uses active-low logic. Setting GPIO 8 low turns the LED on, while setting GPIO 8 high turns it off.

Remote ESP reset

The dashboard enables the ESP32-C3 Super Mini to be restarted remotely by sending an MQTT reset command. When the device receives RESET message on the subscribed reset topic, the firmware executes the ESP.restart() function, causing the microcontroller to reboot. Fig. 13 shows the callback function used to receive the MQTT reset command and restart the ESP32-C3 Super Mini. This feature is useful for remotely recovering devices that become unresponsive without requiring physical access.

IoT MQTT Panel app

The IoT MQTT Panel is a highly customisable MQTT-based dashboard application for monitoring real-time telemetry data and remotely controlling IoT devices. The application is available for both Android and iOS platforms. For general setup and operation, refer to the built-in user guide. The dashboard configuration for this system is described in the following figures.

Figures 14 through 17 illustrate the step-by-step configuration of the system dashboard in the IoT MQTT Panel app. Fig. 14 shows how to access the app's user guide and configure the CPU and RAM telemetry panels. Fig. 15 shows the configuration of the RSSI, chip temperature, and flash usage telemetry panels for monitoring the ESP32-C3's operating status. Fig. 16 illustrates the configuration of the last reset reason, Wi-Fi channel, and uptime telemetry panels. Finally, Fig. 17 shows the configuration of the LED control, LED status, and device reset panels, enabling remote control of the onboard LED and remote restarting of the ESP32-C3 directly from the dashboard.

Uploading the firmware and configuring the dashboard

The firmware configuration is now complete. Compile the sketch and upload it to the ESP32-C3 Super Mini by selecting the appropriate board and COM port in the Arduino IDE.

Next, install the IoT MQTT panel app and configure it to connect to the selected MQTT broker. Then, configure the dashboard to display the telemetry data published by the ESP32-C3 Super Mini.

The dashboard displays real-time values for CPU usage, RAM utilisation, Wi-Fi signal strength (RSSI), chip temperature, flash usage, reset reason, device uptime, and Wi-Fi channel. It also provides remote control of the onboard LED and supports restarting the ESP32-C3 Super Mini through MQTT commands, as shown in Fig. 18.

TO DOWNLOAD SOURCE CODE AND RESOURCES

Scan this QR code or go to https://sourcecode.electronicsforu.com to access and download source code and other resources for this project

Testing

After power-up, the ESP32-C3 Super Mini connects to the configured Wi-Fi network and establishes a connection with the MQTT broker. Once connected, it continuously maintains the MQTT connection, collects system health parameters, publishes telemetry data at one-second intervals, and listens for incoming MQTT commands.

When an ON command is received from the dashboard, the onboard LED turns on. When an OFF command is received, the LED turns off. Similarly, when a RESET command is received, the ESP32-C3 Super Mini restarts.

Throughout operation, the device continuously monitors its health while maintaining low CPU, memory, and network resource usage, making the system suitable for reliable real-time IoT monitoring and control applications. EFY

Pradipta Ghosh holds an M.E. in Electronics and Instrumentation. He is passionate about electronics, microcontroller-based embedded systems, and IoT devices.

Challa Naga Sai Ravali holds an M.Tech in Communications and Signal Processing. She is an Embedded Hardware Design Engineer with experience in embedded systems, FPGA-based designs, IoT solutions, and telecom product development. Her interests include hardware design, wireless communication, and connected technologies.