UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MULTIPROCESSING SYSTEMS IN INTEGRATED CIRCUITS

DESIGN AND IMPLEMENTATION OF AN IoT EMBEDDED SYSTEM FOR TEMPERATURE AND HUMIDITY MONITORING BASED ON ESP32 AND MQTT

Ronaldo Tsela
Fall 2024-2025

Contents

List of Figures

List of Equations

List of Algorithms

Purpose

The purpose of this report is the design and implementation of an embedded application for monitoring air temperature and relative humidity using the IoT platform ESP32 and the MQTT protocol. Specifically the project involves designing and deploying a program on an ESP32 device to read data from a DHT22 sensor and communicate wirelessly via Wi-Fi to a remote server using the MQTT protocol.

This project is part of the final assignment for the course "*Multiprocessing Systems in Integrated Circuits*" in the Master's Program in "*Computer Engineering*" at the National and Kapodistrian University of Athens, Department of Informatics and Telecommunications.

Overview

This project involves the design and deployment of an embedded IoT application for monitoring air temperature and relative humidity. The primary objective is to gain familiarity with efficient algorithms designed for embedded processing systems capable of performing edge data processing. The IoT device is controlled remotely via MQTT with appropriate messages the familiarity of which is also considered an important goal in this project.

The following sections present the hardware components used in this project such as the development board based on the ESP32-WROOM-32 SoC module and the DHT22 sensor module, provides fundamental background on the MQTT protocol used in IoT applications, describes the specific network used in this application and the operations performed by the device, describes the implementation of a CRC 8-bit scheme employed, and explains the task of measurement acquisition and on-board data processing. The last section describes the testing and evaluation procedures conducted to verify the system’s operability.

Methodology

1. Hardware Components and Wiring

The hardware components used in this project include the ESP32 platform and a DHT22 humidity and temperature sensor module. The connection between the board and the sensor is illustrated in **Fig. 1**.

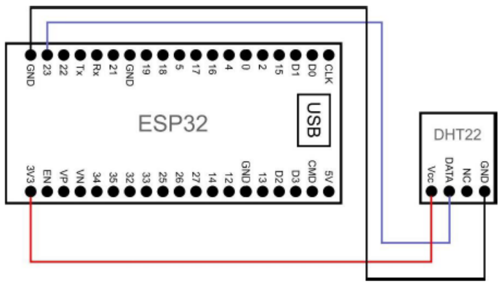


Fig. 1 - Wiring diagram of the ESP32 board and the DHT22 sensor.

The IoT platform utilized is the ESP32-DevKitC, a development board provided by Espressif. This compact board features the ESP32-WROOM-32 module, making it ideal for development and prototyping. It exposes multiple I/O pins to external headers–38 headers grouped into two blocks (J2, J3)--allowing developers to interface with various external devices and peripherals. The board is equipped with a USB-to-UART converter supporting communication speeds up to 3 Mbps. This micro-USB port serves multiple purposes such as powering the board (via a 5V regulator), programming the ESP32 flash memory, and functioning as a serial terminal for development and debugging. For further technical details about the board and its ESP32 core, refer to [1].

This platform is an excellent choice for IoT applications due to its compact design, low power consumption, and support for a wide range of embedded applications, from smart home devices to precision machinery control systems and automotive industry. The reason is due to its fully featured SoC and its relatively low price per unit. Specifically compared to other microcontrollers, the ESP32 for its price per unit offers the best solution for embedded smart applications with demand in processing power and wireless connectivity. Along with the dual-core Xtensa 32-bit processor capable of operating at up to 240 MHz and the 520 KB of on-chip SRAM which is sufficient for most embedded real-time applications, it also features integrated Wi-Fi (802.11n, up to 150 Mbps @ 2.4 GHz).

In this project, the ESP32-based platform is primarily used for its Wi-Fi capabilities and for controlling the DHT22 sensor module. The DHT22 (or AM2302) is a capacitive-type humidity and temperature sensor module capable of measuring temperature in range from -40°C to +80°C with $\pm$ 0.5 °C accuracy and relative humidity in range from 0% to 100% with $\pm$ 2% accuracy. The sensor outputs calibrated digital signals via an embedded 8-bit microcontroller through a bidirectional single wire communication channel. For most applications such a sensor system is enough. The only thing that can be considered as a downside is that the reading period should not be less than 2 seconds on average. For more details refer to the device's datasheet [2].

2. MQTT Basics

In this project the MQTT protocol is used for controlling the device from a remote server. MQTT (MQ Telemetry Transport–not to be confused with Message Queue–) is a widely used messaging protocol for IoT applications. It defines the rules for how IoT devices send and receive data messages over the Internet (It is an application layer protocol such as HTTP). MQTT is a common solution for messaging and data exchange between embedded devices, sensors, and other low-power and resource constrained edge devices

MQTT is an event-driven protocol (as opposed to HTTP) that employs the publish/subscribe (*Pub*/*Sub*) communication model. In the *Pub*/*Sub* model the publisher generates messages while the subscriber consumes them using *Topics*. Topics are a fundamental concept in MQTT. They are similar to a URL path in structure but without the protocol and domain components. MQTT topics are basically a string that acts as an address for messages. They are used to label different messages and provide a way for clients to subscribe to specific messages as requested. It is important to keep in mind that topics are hierarchical and are divided by a forward slash (such as URLs).

The concept of topics is what makes the Pub/Sub model that powerful, they decouple publishers and subscribers, allowing devices to communicate without needing direct knowledge of each other (such as IP addresses). Also topics provide a way to efficiently filter and organize messages (hierarchical structure) and each client can access only what it needs at the time. Finally topics help scale IoT applications since they logically group and route messages.

In addition to the logical entities of topics, two physical primary components are also required to establish an MQTT connection: The MQTT client and the MQTT server (which is known as a broker). An MQTT network consists of interconnected MQTT clients with an MQTT broker. Publisher MQTT clients are devices that publish messages to the broker, while other subscriber MQTT clients –also devices– by subscribing to specific topics within the broker receive the messages they are interested in. The MQTT broker has the ability to dispatch the received messages to the respective topics where clients are subscribed and keeps track of what messages are coming and where they are destined to. This is important since that is the reason why MQTT implementations are lightweight, and therefore making them suitable for deployment on resource-constrained devices on the edge such as the ESP32 device used in this project. Most of the complexity relates to the server side implementation.

One notable feature of MQTT is its ability to maintain persistent sessions between the client and the broker. This allows communication sessions to persist even during network interruptions. When the client reconnects to the network it can resume communication without re-establishing a new session with the broker. This makes MQTT particularly efficient over unreliable cellular networks compared to HTTP.

Any MQTT message contains a payload. The payload is the actual content of the message and can contain any kind of data. MQTT is data-agnostic, meaning it can handle different data types, including images, text in any encoding, encrypted data, and binary data. However, it’s important to note that the payload size can impact network performance and memory usage on the client and broker. Therefore, keeping payloads as small as possible is recommended, especially when publishing messages with a high frequency. For further and more detailed information on the topic please refer to [3].

3. Device Operation

The goal of this project, as previously mentioned, is to retrieve temperature and relative humidity readings from the DHT22 sensor in a controlled manner via the Internet through an MQTT network and perform data processing operations locally using an ESP32 device. The structure of the MQTT network is illustrated in **Fig. 2** where the blue arrows indicate the direction of data transfer from a topic to the respective subscriber.

Specifically the MQTT network consists of three topics to exchange data between the server and the ESP32, all associated with the unique ID “user10” of the current device. Specifically these topics are as follows:

1. ***user10/control***: The ESP32 subscribes and publishes to this topic messages that represent commands. These command messages are:

- “sendConfig”,
- “startMeasurements”,
- “updateConfig”.

2. ***user10/data***: The ESP32 subscribes to this topic waiting to receive configuration messages. The messages received here consist of a 6-digit string of the form $D_0D_1T_0T_1H_0H_1$.

    - D_0D_1 is the sampling period in seconds.
    - T_0T_1 is the temperature threshold above which the ESP32 will indicate a warning message.
    - H_0H_1 is the humidity threshold above which the ESP32 will indicate a warning message.

3. ***user10/dataCrc***: The ESP32 publishes to this topic the CRC generated from the configuration data received. In case that the CRC checksum send is not correct, then the MQTT server will publish in the same topic the message “crcError”.

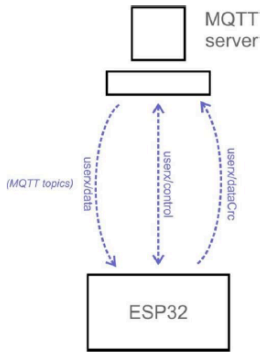


Fig. 2 - MQTT topics for message exchange between the ESP32 MQTT client and the MQTT broker.

Once powered on, the board initializes the system and connects to the local network via Wi-Fi to access the Internet. Upon successfully connecting, the device establishes a connection with the MQTT broker and subscribes to the topics *user10/dataCrc, user10/control, user10/dataCrc*.

Next, the device publishes the message “sendConfig” to the *user10/control* topic and waits for a response on the *user10/data* topic. The response is expected to be a 6-digit value which contains the configuration parameters for the device. From this data is then computed the checksum using a CRC 8-bit algorithm in pairs and then published to the *user10/dataCrc* topic. This message containing the CRC will be then checked by the server. If the CRC is correct, the server publishes the message “startMeasurements” to the *user10/control* topic. If the CRC fails, the server publishes the “crcError” message instead on the same topic *user10/dataCrc*.

Upon receiving the “startMeasurements” message, the device begins reading temperature and humidity data from the DHT22 sensor and proceeds with subsequent processing steps locally. The system will continue reading measurements and processing them until the MQTT broker publishes the message “updateConfig” on the *user10/control* topic. This is issued every minute on average. Upon receiving this message, the device re-enters the configuration mode, updates its parameters, and repeats the setup process as described above.

4. Cyclic Redundancy Check

The first task for the system, upon receiving configuration data from the *user10/data* topic in the format $D_0D_1T_0T_1H_0H_1$, is to compute a checksum using an 8-bit Cyclic Redundancy Check (CRC) algorithm applied in pairs. CRC is a widely used error-detection algorithm in digital communication and storage systems, capable of identifying bit errors in data. This is accomplished by appending a checksum to the actual data. The checksum is derived from the data itself, enabling the receiver to verify its integrity by recalculating the CRC value and comparing it to the transmitted one [4][5]. In this system, the ESP32 computes the CRC for the transmitted configuration data, sends it back to the server, and the server evaluates whether the data received by the client is valid.

The CRC algorithm treats data as a binary polynomial and performs polynomial division. The input data is interpreted as a long binary sequence (the dividend), which is divided by a fixed binary polynomial (the divisor). The remainder from this division serves as the CRC value. When the receiver performs the same operation on the transmitted data, a remainder of zero indicates that the data is error-free; otherwise, a bit error occurred during transmission [4][5].

Although polynomial division might seem complex, binary arithmetic significantly simplifies the process in CRC implementation. The division is performed using the XOR (exclusive OR) operation with the generator polynomial and a series of single left logical shifts. In the following pseudocode the 8-bit CRC calculation algorithm is provided (**Algorithm 1**). Specifically we initialize the CRC value with the input byte. Then, we iterate over its 8 bits, shifting the CRC value one position to the left on each iteration. If the most significant bit is 1, a XOR operation is performed between the CRC value and the generator polynomial.

In this project, the generator polynomial used is represented by the value 10 (or b00001010 in binary), corresponding to the polynomial $x^3 + x$. This generator polynomial is used because the device ID is *user10*. Once the checksum is computed, it is sent to the server for validation. If the server confirms the checksum as correct, the system proceeds to further operations ("startMeasurements"); otherwise, an error is flagged ("crcError").

Algorithm 1 - The 8-bit CRC algorithm

```
input: byte, generator
initialize: crc ← byte

for i in 0 to 7 loop
    if crc & b10000000 then
        crc ← (crc << 1) ^ generator
    else
        crc ← crc << 1
    end if
end loop

output: crc
```

5. Reading and Processing Measurements

Upon verifying the CRC successfully, the server responds on the *user10/control* topic with the message "startMeasurements", which triggers the ESP32 to begin reading temperature and humidity data from the sensor module and processing them locally. The *DHTesp [7]* library is used to interact with the sensor, providing an intuitive and easy-to-use API for obtaining readings.

Each time a new temperature and humidity measurement is obtained, it is compared against the configured threshold values T_0T_1 and H_0H_1, respectively. If the measured values exceed these thresholds, a warning message is displayed in the serial monitor to indicate the condition. The messages displayed here are:

- "[WARNING] Temperature High"
- "[WARNING] Humidity High"

Next, the temperature readings are appended to a shift buffer of constant length (5 elements), which is utilized by a linear least squares approximation algorithm [6]. This algorithm computes the linear best-fit function (given in **Eq. 1**) to estimate future temperature values:

$$T_{i+1} = a \cdot T_i + b \quad (1)$$

where T_{i+1} is the predicted temperature, T_i the current temperature, a is the slope of the linear function which represents the temperature rate of change through time and b is an intercept of the linear model. The slope and intercept values are calculated using the following formulas:

$$a = \frac{N \cdot \sum_{i=0}^{N-1} x_i \cdot y_i - \sum_{i=0}^{N-1} x_i \cdot \sum_{i=0}^{N-1} y_i}{\sum_{i=0}^{N-1} x_i \cdot \sum_{i=0}^{N-1} y_i - \sum_{i=0}^{N-1} x_i^2}, \quad b = \frac{1}{n} \left(\sum_{i=0}^{N-1} y_i - a \cdot \sum_{i=0}^{N-1} x_i \right) \quad (2)$$

where x_i, here represents the sample number/ index with $x = i = [0, 1, 2, 3, 4]$ while y_i represents the reading of temperature in the buffer at position/ index i.

The system remains in this reading and processing mode until the MQTT broker publishes the message “updateConfig” to the *user10/control* topic. When this occurs, the device re-enters the configuration mode, updating its parameters as described earlier.

Note here that time management in the system operates in milliseconds. Therefore, upon receiving configuration parameters D_0D_1, the values are multiplied by 1000 to convert them into milliseconds.

6. The setup() and loop() Functions

The program for the device is developed using the Arduino framework which means properly configuring the *setup*() and *loop*() functions. Below is a description of these functions and their respective roles in the system's operation.

The *setup*() function initializes the system by performing the following tasks:

1. Starts the serial communication by enabling the serial port with a baud rate of 115200 for debugging and monitoring.
2. Initializes the DHT22 sensor for temperature and humidity readings.
3. Establishes a WiFi connection to the local network for internet access.
4. Sets the callback function to handle incoming messages from the MQTT broker.
5. Establishes a connection with the MQTT broker to enable communication and subscribes to the defined topics.

After the system is initialized in *setup*(), the program enters the *loop*() function, where the main operational tasks are executed iteratively. The *loop*() contains four core functionalities, which are controlled or control various boolean flags.

1. ***initialized***: Controls whether the ESP32 should publish the message “sendConfig” to the *user10/control* topic. It is set to true when the ESP32 receives the configuration string from the MQTT broker in *user10/data*. It is reset to false if the MQTT broker publishes the messages “crcError” or “updateConfig.”

2. ***startMeasurements***: It is set to true when the ESP32 receives the “startMeasurements” command from the MQTT broker in the *user10/control* topic. It is reset to false when the MQTT broker publishes the message “updateConfig” in the *user10/control* topic.

3. ***lsqInit***: Set to true when the *startMeasurements* flag is true and the system has collected at least five samples in the shift buffer. Once set to true, it remains so as long as the system operates or something unexpected happens.

Below the implemented functionalities within this loop are described:

1. **Network Connection Check**: It continuously verifies the WiFi connection. If the connection is lost, it attempts to reconnect in a blocking manner. If reconnection fails for more than 1 minute, all control flags (*initialized*, *startMeasurements*, *lsqInit*) are reset to false. In that way the system is obligated to reconfigure itself.

2. **MQTT Transactions Management**: Handles interactions with the MQTT broker based on the following logic:

    - If both *initialized* and *startMeasurements* are false, publish “sendConfig” to the *user10/control* topic.
    - If *initialized* is true and *startMeasurements* is false, publish the CRC to the *user10/dataCrc* topic.

    If the connection with the MQTT broker is lost then it attempts to reconnect. If reconnection fails for more than 1 minute, all control flags are reset to false.

3. **Linear Least Squares Approximation Initialization**: Determines when to start the linear least squares approximation. Specifically if the shift buffer is full, *lsqInit* is false, and *startMeasurements* is true then set *lsqInit* to true.

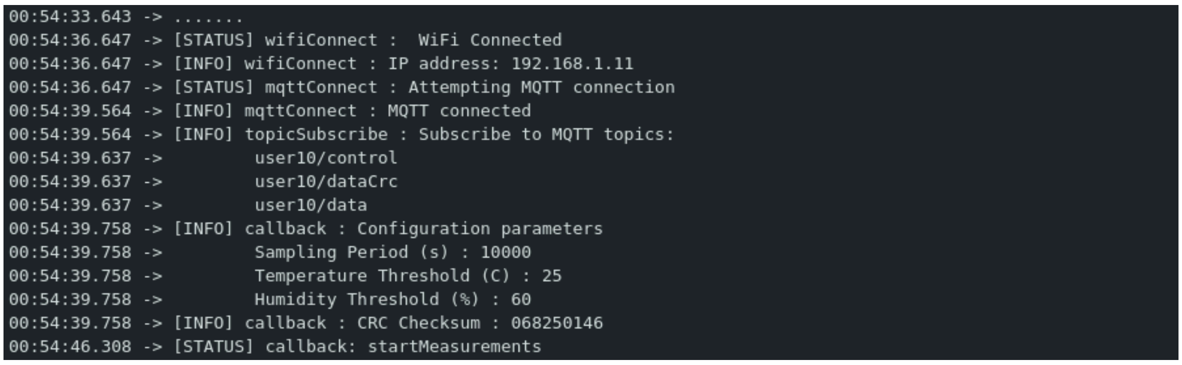
4. **Measurement Reading and Processing**: Here it performs reading and processing of temperature and humidity data if both *initialized* and *startMeasurements* flags are true, operating according to the sampling frequency specified in the configuration. If the sensor stops working, the system displays an error message in the terminal.

Results

To assess the system's functionality, a debug mode is available in the program (activated by uncommenting the *#define DEBUG_MODE* directive on line 5). In this mode, the output to the serial monitor is more verbose than under normal operating conditions. The system's behavior is evaluated by running it in debug mode and analyzing the messages printed to the terminal.

In **Fig. 3**, it is observed the initial phase of system operation, from boot-up to receiving the “startMeasurements” command. The process begins with the board powering on and connecting to the local network via WiFi, which takes approximately 3 seconds. Subsequently, the board successfully connects to the MQTT server and subscribes to the three topics: *user10/control, user10/dataCrc*, and *user10/data*, as specified.

Next, the board receives the configuration parameters in the string “102560” and calculates the CRC checksum (068250146). The checksum is verified as correct, as also proved by the MQTT server responding with the message “startMeasurements” after the checksum is published to *user10/dataCrc*. This configuration stage typically lasts about 10 to 20 seconds.



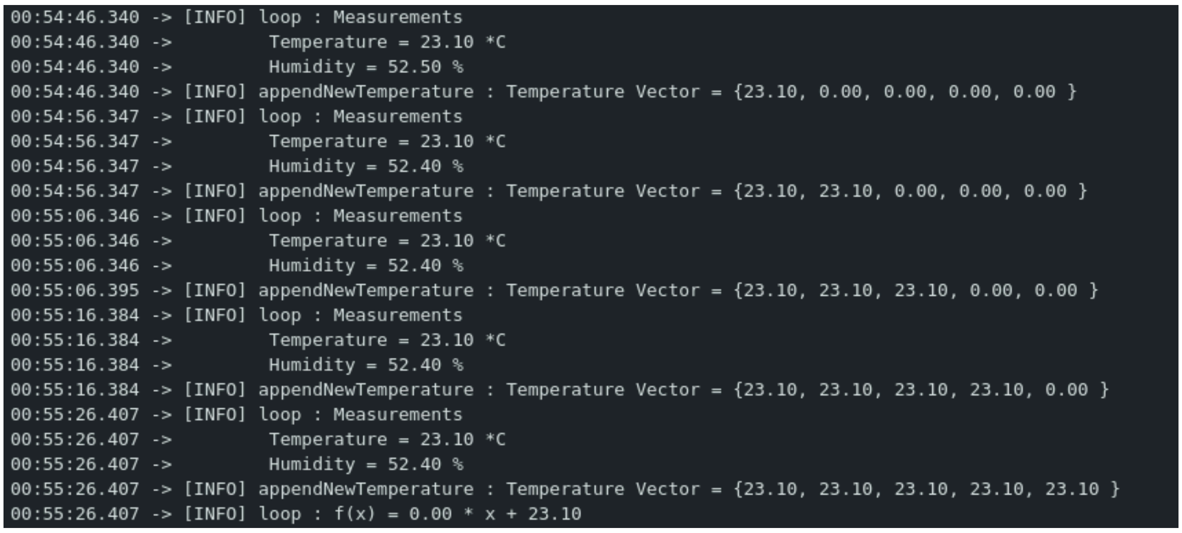
```
00:54:33.643 -> .......
00:54:36.647 -> [STATUS] wifiConnect :  WiFi Connected
00:54:36.647 -> [INFO] wifiConnect : IP address: 192.168.1.11
00:54:36.647 -> [STATUS] mqttConnect : Attempting MQTT connection
00:54:39.564 -> [INFO] mqttConnect : MQTT connected
00:54:39.564 -> [INFO] topicSubscribe : Subscribe to MQTT topics:
00:54:39.637 ->         user10/control
00:54:39.637 ->         user10/dataCrc
00:54:39.637 ->         user10/data
00:54:39.758 -> [INFO] callback : Configuration parameters
00:54:39.758 ->         Sampling Period (s) : 10000
00:54:39.758 ->         Temperature Threshold (C) : 25
00:54:39.758 ->         Humidity Threshold (%) : 60
00:54:39.758 -> [INFO] callback : CRC Checksum : 068250146
00:54:46.308 -> [STATUS] callback: startMeasurements
```

Fig. 3 - The initial phase of the system operation.

Next, the system begins sampling and processing data. For the first five samples, it simply displays the measured temperature and relative humidity values, comparing them with the set threshold values. During the experiment, the temperature remained constant at 23.1°C and the humidity at 52.4%, both of which were below the thresholds. As a result, no warning messages were displayed.

By measuring the time interval between samples, it can be evaluated whether the system performed the sampling operations on schedule. As shown in **Fig. 4**, the first sample is recorded at 00:54:46:34, the second at 00:54:56:34, the third at 00:55:06:34, and so on. The measurements are consistently taken at 10-second intervals, aligning with the current configuration.

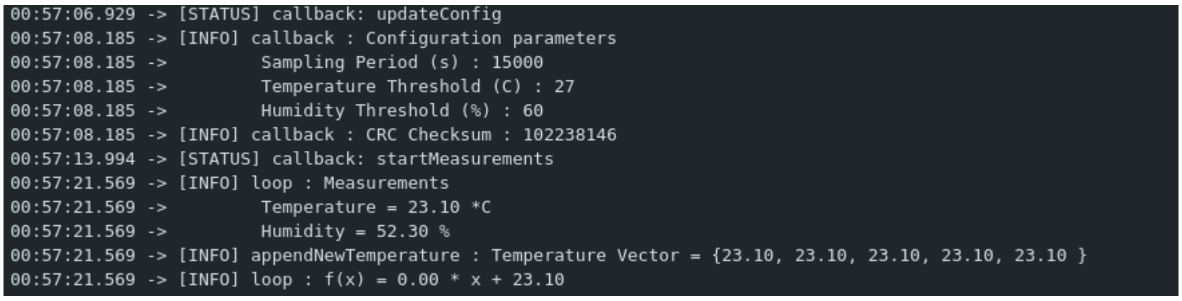
Finally is evaluated the operability of the LSQ algorithm which starts producing results once the “*Temperature Vector”* is fully populated. Since all measured temperature values were identical, the rate of change was determined to be 0ºC. This is reflected in the fit function f(x) = 0.00 · x + 23.10 as observed.



```
00:54:46.340 -> [INFO] loop : Measurements
00:54:46.340 ->         Temperature = 23.10 *C
00:54:46.340 ->         Humidity = 52.50 %
00:54:46.340 -> [INFO] appendNewTemperature : Temperature Vector = {23.10, 0.00, 0.00, 0.00, 0.00 }
00:54:56.347 -> [INFO] loop : Measurements
00:54:56.347 ->         Temperature = 23.10 *C
00:54:56.347 ->         Humidity = 52.40 %
00:54:56.347 -> [INFO] appendNewTemperature : Temperature Vector = {23.10, 23.10, 0.00, 0.00, 0.00 }
00:55:06.346 -> [INFO] loop : Measurements
00:55:06.346 ->         Temperature = 23.10 *C
00:55:06.346 ->         Humidity = 52.40 %
00:55:06.395 -> [INFO] appendNewTemperature : Temperature Vector = {23.10, 23.10, 23.10, 0.00, 0.00 }
00:55:16.384 -> [INFO] loop : Measurements
00:55:16.384 ->         Temperature = 23.10 *C
00:55:16.384 ->         Humidity = 52.40 %
00:55:16.384 -> [INFO] appendNewTemperature : Temperature Vector = {23.10, 23.10, 23.10, 23.10, 0.00 }
00:55:26.407 -> [INFO] loop : Measurements
00:55:26.407 ->         Temperature = 23.10 *C
00:55:26.407 ->         Humidity = 52.40 %
00:55:26.407 -> [INFO] appendNewTemperature : Temperature Vector = {23.10, 23.10, 23.10, 23.10, 23.10 }
00:55:26.407 -> [INFO] loop : f(x) = 0.00 * x + 23.10
```

Fig. 4 - Measurement retrieval and data processing results.

When the server publishes the “updateConfig” command to *user10/control,* the device reconfigures as expected. As shown in **Fig. 5** below, the device receives a new string of configuration parameters and recalculates the CRC checksum for these parameters. Upon receiving the “startMeasurements” command again, it resumes data collection using the updated configuration. It is also important to note that the approximator continues operating seamlessly throughout this process.

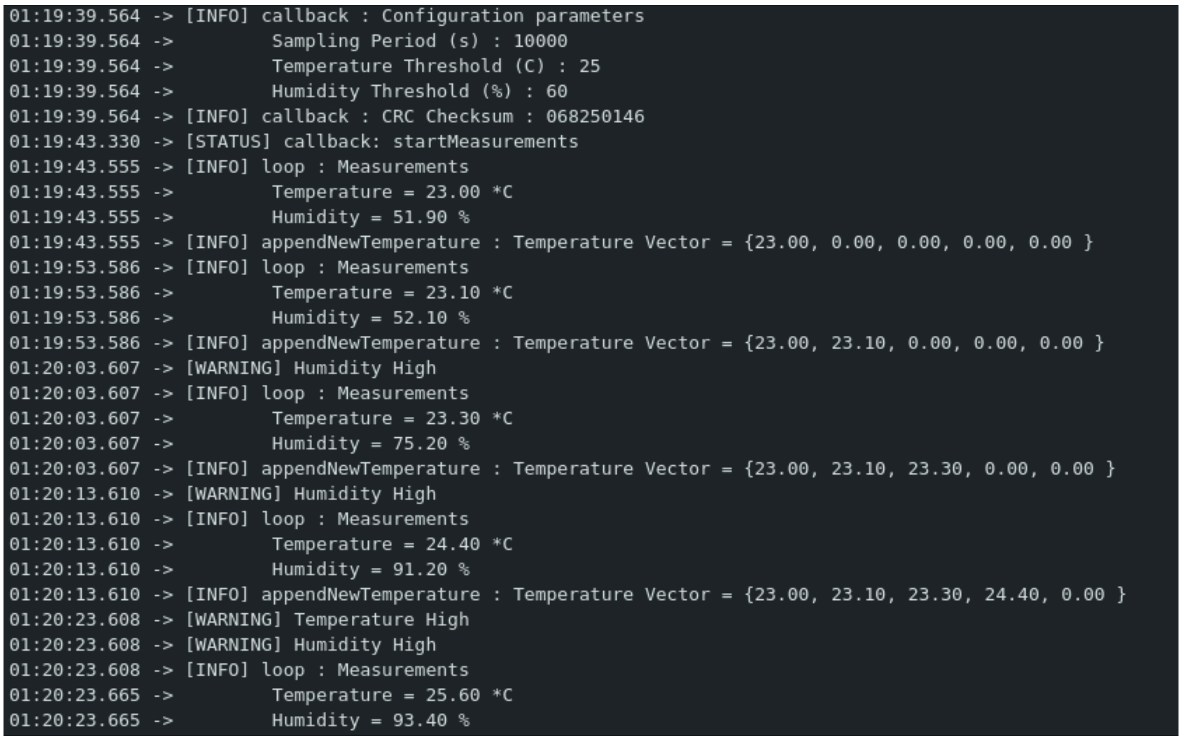


```
00:57:06.929 -> [STATUS] callback: updateConfig
00:57:08.185 -> [INFO] callback : Configuration parameters
00:57:08.185 ->         Sampling Period (s) : 15000
00:57:08.185 ->         Temperature Threshold (C) : 27
00:57:08.185 ->         Humidity Threshold (%) : 60
00:57:08.185 -> [INFO] callback : CRC Checksum : 102238146
00:57:13.994 -> [STATUS] callback: startMeasurements
00:57:21.569 -> [INFO] loop : Measurements
00:57:21.569 ->         Temperature = 23.10 *C
00:57:21.569 ->         Humidity = 52.30 %
00:57:21.569 -> [INFO] appendNewTemperature : Temperature Vector = {23.10, 23.10, 23.10, 23.10, 23.10 }
00:57:21.569 -> [INFO] loop : f(x) = 0.00 * x + 23.10
```

Fig. 5 - The operations after updating the configuration.

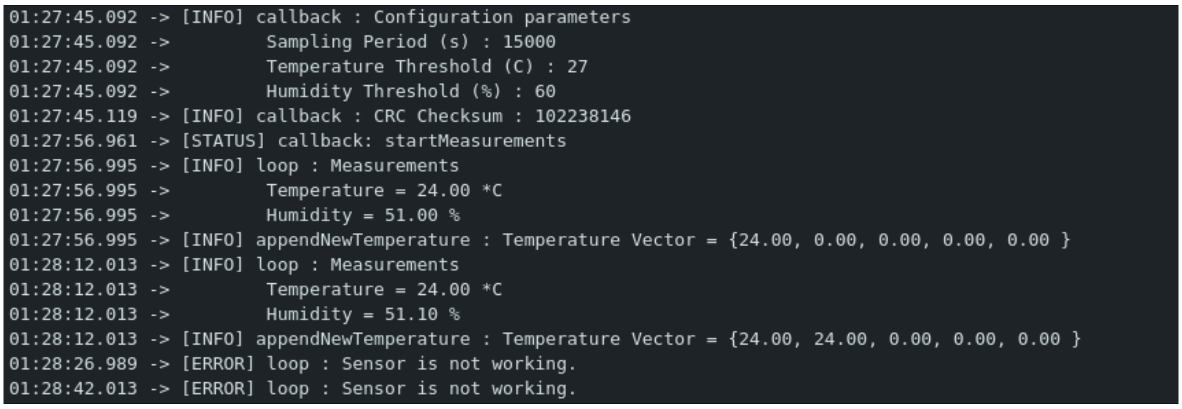
In this next evaluation, the system's response to values that exceed the defined thresholds is tested. To do that the sensor is placed in a warm and humid environment in order to affect its readings. As the humidity rises above the threshold, the predefined warning message is displayed. The same behavior is observed for temperature, as well as when both parameters surpass their thresholds simultaneously. The thresholds are set at 25°C and 60% humidity, respectively. Refer to the measurements recorded at 01:20:03 and 01:20:23 in **Fig. 6**.

Additionally, **Fig. 7** illustrates the error message displayed when the sensor is not operational, as seen at time 01:28:26. To simulate a non-functional sensor, it is simply disconnected with caution from the board. Once the sensor was reconnected, the board resumed displaying data as expected.



```
01:19:39.564 -> [INFO] callback : Configuration parameters
01:19:39.564 ->         Sampling Period (s) : 10000
01:19:39.564 ->         Temperature Threshold (C) : 25
01:19:39.564 ->         Humidity Threshold (%) : 60
01:19:39.564 -> [INFO] callback : CRC Checksum : 068250146
01:19:43.330 -> [STATUS] callback: startMeasurements
01:19:43.555 -> [INFO] loop : Measurements
01:19:43.555 ->         Temperature = 23.00 *C
01:19:43.555 ->         Humidity = 51.90 %
01:19:43.555 -> [INFO] appendNewTemperature : Temperature Vector = {23.00, 0.00, 0.00, 0.00, 0.00 }
01:19:53.586 -> [INFO] loop : Measurements
01:19:53.586 ->         Temperature = 23.10 *C
01:19:53.586 ->         Humidity = 52.10 %
01:19:53.586 -> [INFO] appendNewTemperature : Temperature Vector = {23.00, 23.10, 0.00, 0.00, 0.00 }
01:20:03.607 -> [WARNING] Humidity High
01:20:03.607 -> [INFO] loop : Measurements
01:20:03.607 ->         Temperature = 23.30 *C
01:20:03.607 ->         Humidity = 75.20 %
01:20:03.607 -> [INFO] appendNewTemperature : Temperature Vector = {23.00, 23.10, 23.30, 0.00, 0.00 }
01:20:13.610 -> [WARNING] Humidity High
01:20:13.610 -> [INFO] loop : Measurements
01:20:13.610 ->         Temperature = 24.40 *C
01:20:13.610 ->         Humidity = 91.20 %
01:20:13.610 -> [INFO] appendNewTemperature : Temperature Vector = {23.00, 23.10, 23.30, 24.40, 0.00 }
01:20:23.608 -> [WARNING] Temperature High
01:20:23.608 -> [WARNING] Humidity High
01:20:23.608 -> [INFO] loop : Measurements
01:20:23.665 ->         Temperature = 25.60 *C
01:20:23.665 ->         Humidity = 93.40 %
```

Fig. 6 - Warning messages appearing when measurements surpass threshold values.



```
01:27:45.092 -> [INFO] callback : Configuration parameters
01:27:45.092 ->         Sampling Period (s) : 15000
01:27:45.092 ->         Temperature Threshold (C) : 27
01:27:45.092 ->         Humidity Threshold (%) : 60
01:27:45.119 -> [INFO] callback : CRC Checksum : 102238146
01:27:56.961 -> [STATUS] callback: startMeasurements
01:27:56.995 -> [INFO] loop : Measurements
01:27:56.995 ->         Temperature = 24.00 *C
01:27:56.995 ->         Humidity = 51.00 %
01:27:56.995 -> [INFO] appendNewTemperature : Temperature Vector = {24.00, 0.00, 0.00, 0.00, 0.00 }
01:28:12.013 -> [INFO] loop : Measurements
01:28:12.013 ->         Temperature = 24.00 *C
01:28:12.013 ->         Humidity = 51.10 %
01:28:12.013 -> [INFO] appendNewTemperature : Temperature Vector = {24.00, 24.00, 0.00, 0.00, 0.00 }
01:28:26.989 -> [ERROR] loop : Sensor is not working.
01:28:42.013 -> [ERROR] loop : Sensor is not working.
```

Fig. 7 - Error messages appearing when the sensor is not working properly.

References

[1] ESP32 Series Datasheet Version 4.7. URL.

[2] DHT22 Sensor Module Datasheet. URL.

[3] MQTT Hive. MQTT Tutorials. URL.

[4] Cyclic Redundancy Check - Wikipedia. URL.

[5] Understanding and Implementing CRC. URL.

[6] Linear Least Squares - Wikipedia. URL.

[7] DHTesp Library. URL.