



.....

BRUFACE
BRUSSELS FACULTY
OF ENGINEERING

.....



OPERATING SYSTEMS AND SECURITY

Final Report

Data Encryption and Transmission Based on ESP32 Boards

Authors:

HAN BAO
JIALUN KOU
ANDI SULA

Professor:

BRUNO DA SILVA

Monday 28th November 2022

Contents

1	Introduction	1
1.1	Introduction of FreeRTOS	1
1.2	Introduction of ESP32	1
1.3	Introduction of MQTT	1
1.4	Introduction of Node-Red	2
2	Benchmark of Single Core ESP32 and Dual Core ESP32	3
2.1	Benchmark of Single Core ESP32	3
2.2	Benchmark of Dual Core ESP32	4
2.3	Why Dual Core ESP32?	5
2.4	Semaphore and Multitask scheduling	6
3	MQTT Configuration	8
3.1	Function of MQTT	8
3.2	Configuring MQTT	8
4	Node-Red Configuration	9
4.1	Function of Node-Red	9
4.2	Configuring Node-Red	9
5	Data Transmission and Receiving	11
5.1	Un-encrypted data transmission and receiving	11
5.2	Encrypted data transmission and receiving	12
6	Data Encryption and Decryption	14
6.1	Encryption	14
6.2	Decryption	16
7	Result	17

1 Introduction

In this project, the performance of the single core and dual core will be benchmarked. And a secure protocol for data transfer will also be developed on a real-time operating system (RTOS). The proposed solutions will be rigorously tested and validated using actual measurements taken from microcontrollers. To accomplish this, two ESP32 boards are utilized as our test platform.

1.1 Introduction of FreeRTOS

FreeRTOS is a real-time operating system (RTOS) for microcontrollers and small microprocessors. It is designed to be small and simple, making it suitable for use in embedded systems and IoT devices. The RTOS provides basic features such as task management, inter-task communication, and resource management, allowing developers to create real-time applications with predictable behavior. FreeRTOS is open-source and can be freely used, modified, and distributed.

1.2 Introduction of ESP32

The ESP32 is a low-cost, low-power microcontroller with integrated WiFi and Bluetooth capabilities. It is developed by Espressif Systems and is based on the popular ESP8266 microcontroller. The ESP32 is designed for Internet of Things (IoT) applications and is capable of handling a wide range of tasks, from low-power sensor networks to more demanding applications such as voice encoding, music streaming, and MP3 decoding. The ESP32 also features a dual-core processor, high-speed communication peripherals, and a wide range of memory options. It's a powerful microcontroller with on-board Wi-Fi and Bluetooth capabilities, making it a great choice for IoT projects.

1.3 Introduction of MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe network protocol that is designed for communication in machine-to-machine (M2M) and Internet of Things (IoT) contexts. It is a publish-subscribe based messaging protocol that is designed for use on top of the TCP/IP protocol. The protocol is designed to be lightweight, making it ideal for use in resource-constrained devices and low-bandwidth, high-latency or unreliable networks.

In MQTT, there are two main components: the MQTT broker and the MQTT clients. The broker acts as a message hub, receiving messages from the clients and forwarding them on to the appropriate recipients. The clients can be either publishers, which send messages to the broker, or subscribers, which receive messages from the broker.

MQTT is widely used in IoT applications due to its small footprint, low power consumption and the ability to run over unreliable networks. It is also commonly used in

industrial automation, and M2M communication. Due to its simple design, it's easy to implement, and can be used to connect a wide range of devices, from sensors and actuators to smartphones and servers.

1.4 Introduction of Node-Red

Node-RED is a visual programming tool for wiring together hardware devices, APIs, and online services in new and interesting ways. It is built on Node.js and provides a browser-based flow editor that makes it easy to wire together flows using a variety of nodes.

Node-RED allows you to create and manage connections between different devices, services, and APIs by connecting "nodes" together in a flow. These flows can be deployed to the runtime in a single-click. The flows can be triggered by events, and can also be scheduled to run at specified intervals.

Node-RED is commonly used in IoT projects to connect different devices and services together, without needing to write any code. It can also be used to create simple automation flows, such as turning on lights when motion is detected, or sending a text message when a sensor goes out of range.

2 Benchmark of Single Core ESP32 and Dual Core ESP32

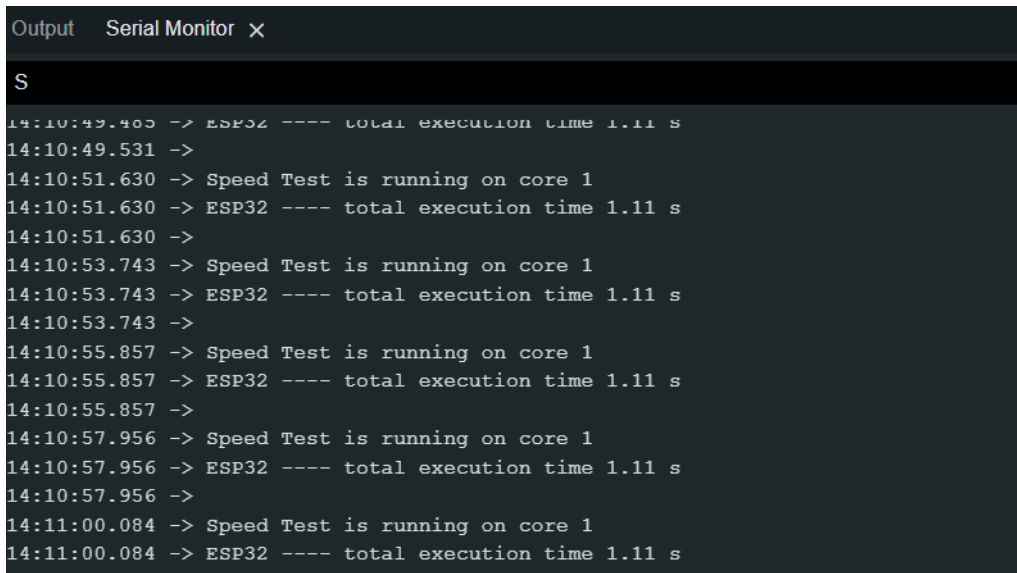
2.1 Benchmark of Single Core ESP32

During this section, only 1 of its core will be used. In the code it is clear that mainly two tasks are created, which are `uint32_t double_pow(uint32_t count)` and `uint32_t loop_gpio(uint32_t count)`, respectively.

The first function is a pow-calculating function, which aims at simulating a software loaded on the CPU. And the second function is a LED light control function, which controls a LED light on and off very fast, by doing so, the hardware loaded on the CPU is simulated. The parameter **count** inside the functions stands for the number of loops.

And to compare the performance of single core and dual core, a 'clock' is put inside each function, which are `start_millis=millis()` to start timing and `stop_millis=millis()` to stop, respectively. In the `void do_messure()` functions, one can see that the pow function loops for 1000000 times and gpio control function loops for 100000 times.

Moreover, a useful function which is `xPortGetCoreID()` is used inside the `void do_messure()` function. By introducing this function, one can easily know that on which core the tasks run. The result of this section is shown in figure below. One can easily tell that the tasks only run on Core 1 and the execution time is 1.11s.



```
Output  Serial Monitor x
S
14:10:49.403 -> ESP32 ---- total execution time 1.11 s
14:10:49.531 ->
14:10:51.630 -> Speed Test is running on core 1
14:10:51.630 -> ESP32 ---- total execution time 1.11 s
14:10:51.630 ->
14:10:53.743 -> Speed Test is running on core 1
14:10:53.743 -> ESP32 ---- total execution time 1.11 s
14:10:53.743 ->
14:10:55.857 -> Speed Test is running on core 1
14:10:55.857 -> ESP32 ---- total execution time 1.11 s
14:10:55.857 ->
14:10:57.956 -> Speed Test is running on core 1
14:10:57.956 -> ESP32 ---- total execution time 1.11 s
14:10:57.956 ->
14:11:00.084 -> Speed Test is running on core 1
14:11:00.084 -> ESP32 ---- total execution time 1.11 s
14:11:00.084 ->
```

Figure 2.1: Single Core

2.2 Benchmark of Dual Core ESP32

During this section, the same scheme is implemented as mentioned above, which are **double_pow** function and **loop_gpio** function. And this section aims at benchmarking the dual core ESP32, there fore three tasks running on the both cores are created, which are **void myCore0Task(void pvParameters)**, **void myCore1Task(void pvParameters)**, **void myResultTask(void pvParameters)** (actually only first two functions work to benchmark the dual core ESP32 while the last one is used for get the result). The first task is pinned to the core 0 and the second task is pinned to the core 1, and the third task is pinned to the core 1 whose function is to collect the results from the queue and print it on the serial monitor.

One can see that in the each of the **myCoreTask**, a **do_measure** function is inserted. However, since two tasks will run in parallel on each core, so these two tasks may send the result to the serial monitor at at the same time which will cause the collision. Seen as below.

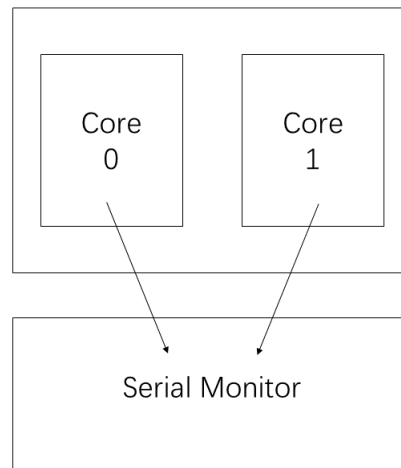


Figure 2.2: Collision

So in the **do_measure** function a queue is created to solve this problem. By introducing the queue, the result of the task can be stored into the queue temporarily when the result task is stuck or busy so that the result won't lost. Then the **void myResultTask(void pvParameters)** function is able to get the information of core and total execution time and prints them to the serial monitor. Seen as below.

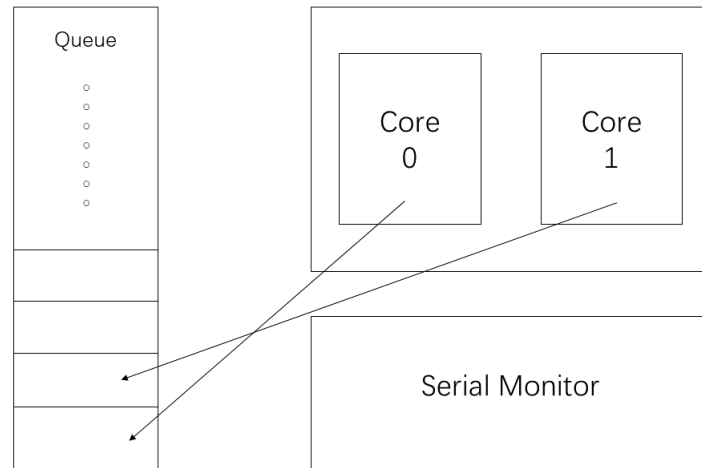


Figure 2.3: Queue

The result of dual core ESP32 test can be seen as below. One can know the tasks execute both on Core 0 and Core 1 and the total execution time is 0.56s which is much less than 1.11s of single core.

```

Output  Serial Monitor ✕
Message (Enter to send message to 'ESP32-WROOM-DA Module' on 'COM7')
14:21:48.738 -> core 0
14:21:48.738 -> ESP32 ---- total execution time: 0.56 s
14:21:48.738 ->
14:21:50.297 -> core 1
14:21:50.297 -> ESP32 ---- total execution time: 0.56 s

```

Figure 2.4: Dual Core

2.3 Why Dual Core ESP32?

The ESP32 has built-in Wi-Fi functionality, which allows it to connect to wireless networks and communicate with other devices. However, running the Wi-Fi stacks can consume a significant amount of processing power, which can impact the performance of other tasks running on the ESP32.

By using a dual-core design, the ESP32 can dedicate one core to handle the Wi-Fi and MQTT connection, while the other core can be processing sensor data. This allows the ESP32 to handle network connectivity and application processing simultaneously, without compromising performance. In addition, from the previous section, one can notice that when multi-tasks run on dual-core ESP32 can be much faster than running on a single-core. Therefore, in this project, the dual-core scheme has been chosen. By transferring the single core to the dual-core, the 'structure' can become as shown in Fig 2.5.

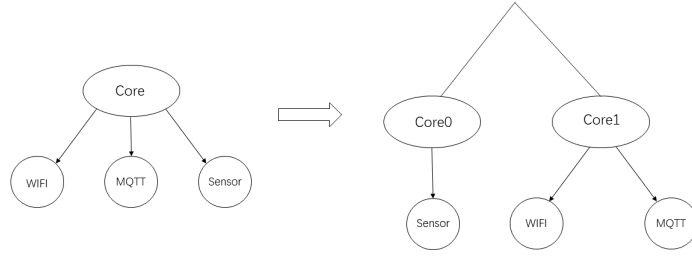


Figure 2.5: Transfer from Single Core to Dual Core

This means that Sensor Task and MQTT-WiFi Task are running in parallel, this will surely increase the efficiency. However, it is worth noting that, even though the ESP32 uses a dual-core design, the Wi-Fi and MQTT tasks are still running under the same core, which may lead to some priority issues. During the project, the priorities of those two tasks are set as follow: (The scheduling of the MQTT and Wi-Fi tasks will be discussed in the next section.)

- WiFi Task: Core: 1 Priority: 15
- MQTT Task: Core: 1 Priority: 13

2.4 Semaphore and Multitask scheduling

In a real-time operating system (RTOS) that uses semaphores and priority-based scheduling, it is possible to use semaphores and task priorities to coordinate the execution of multiple tasks that handle WiFi and MQTT connections in a specific order, with the WiFi task having a higher priority than the MQTT task and both tasks are processed in the same core.

The MQTT connection is dependent on the WiFi connection, so it is important that the WiFi connection is established first before the MQTT connection can be established. To ensure this, we can use semaphores and task priorities in the following way:

- A semaphore, let's say "wifi_sem", can be used to control access to the WiFi connection. The WiFi task must have a higher priority than the MQTT task, and when the WiFi task wants to establish a connection, it must first acquire the semaphore "wifi_sem". If the semaphore is available, the WiFi task will be granted access to the WiFi connection, and the semaphore's value will be decremented. If the semaphore is not available, the WiFi task will be blocked until the semaphore is released by another task.
- A semaphore, let's say "mqtt_sem", can be used to control access to the MQTT connection. The MQTT task must have a lower priority than the WiFi task. When

the MQTT task wants to establish a connection, it must first check the value of the semaphore "wifi_sem". If the semaphore is zero, it means the WiFi connection is not yet established and the task will be blocked until the semaphore is released. If the semaphore is non-zero, it means the WiFi connection is established and the task will acquire the semaphore "mqtt_sem" and established the MQTT connection.

- To check if both mqtt and wifi connection disconnected, both task will run in background, using a multitasking scheduler, and will check periodically if the connection is still up and running. If the connection is disconnected, it will try to re-establish the connection. If it is unable to re-establish the connection, it will trigger a restart of the ESP32.

By using semaphores and task priorities in this way, the RTOS can ensure that the WiFi connection is established before the MQTT connection, the WiFi task has higher priority than the MQTT task, and both tasks are processed in the same core. This allows the RTOS to handle the shared resources safely and efficiently and to ensure that the most important task is executed first. Additionally, if any of the WiFi or MQTT connection is disconnected, it will restart the ESP32 to ensure that the device is always connected.

Moreover, in this project, a hardware Interrupt Service Routine, namely hardware ISR, is added. It is a physical button in this project and is automatically set to the highest priority. When pushing the button, all other tasks stop and the *Mode* parameter switches from 0 to 1 and when it reaches 2 it will become 0 again, which means that it will loop between 0 and 1, standing for encryption and non-encryption modes. The sketch map of multi-task scheduling is shown as follow:

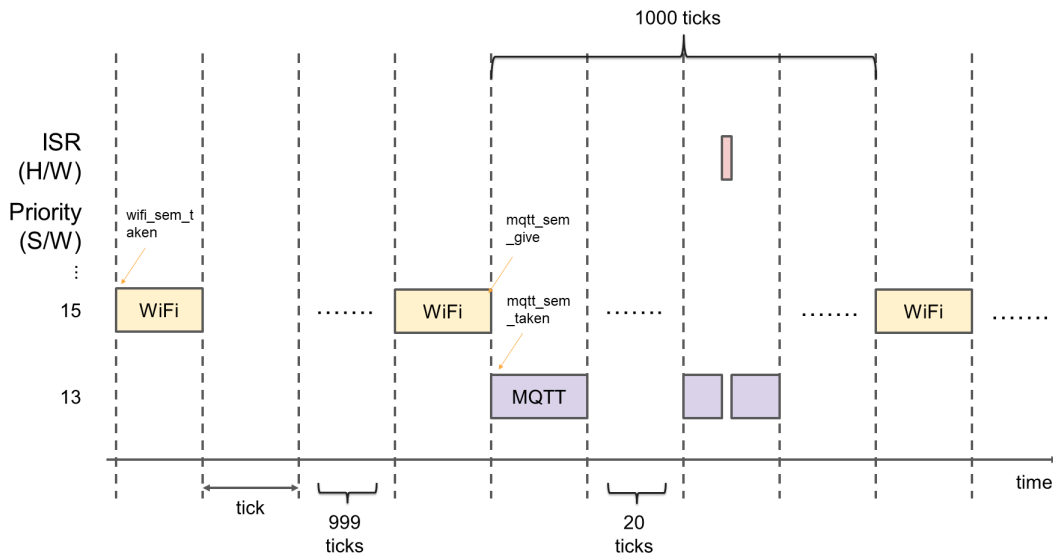


Figure 2.6: Multi-task scheduling

Finally, when both MQTT and wifi are successfully connected, it will release a semaphore to the sensor task, which can then proceed to run on another core.

3 MQTT Configuration

3.1 Function of MQTT

During this chapter, the MQTT will be configured. In the project all the data collected by the sensor will be first read by the ESP32(board A) and then transmitted to the MQTT server and finally create a topic on the MQTT . By doing so, the another ESP32 board(board B) can access the topic and obtain the data read by board A from the MQTT server, vice versa. In this way, two boards can realize the function of communicate with each other.

However, since MQTT is, to some extent, a 'transparent' server which means that everyone can access the topic that one just created. Therefore, in order to protect the data, several methods are implemented in the project (which will be detailedly stated in the Chapter 6).

3.2 Configuring MQTT

There are several steps to configure MQTT:

- Install a broker: An MQTT broker is the server that handles all the message transfers between the clients. There are many open-source MQTT brokers available, such as Mosquitto and RabbitMQ. One can install one of these on the server or device, or use a cloud-based service like AWS IoT or Azure IoT Hub.
- Configure the broker: Once the broker is installed, one will need to configure it by setting the appropriate options, such as the listening IP and port, authentication and security settings, and other parameters. The exact configuration steps will depend on the specific broker that are using.
- Create a client: An MQTT client is a program or device that connects to the broker and sends or receives messages. One can use one of the many MQTT client libraries available for various programming languages, or use a pre-built client application.
- Connect the client to the broker: Once the client is created, one will need to configure it to connect to the broker by specifying the broker's IP or hostname and port, and any authentication or security settings that are required.
- Publish and subscribe to topics: With the client connected to the broker, one can now publish messages to specific topics and subscribe to receive messages from specific topics.
- Test the configuration: Once the client and broker are configured, one can test the connection and messaging functionality to ensure everything is working correctly.

4 Node-Red Configuration

4.1 Function of Node-Red

Node-RED provides a visual, drag-and-drop interface for creating IoT flows, which makes it easy to deal with all kinds of data and it can run on a wide range of devices and platforms, from small microcontrollers to powerful servers. Therefore it is able to perfectly visualize the data.

4.2 Configuring Node-Red

There are several steps to configure Node-Red:¹

- **Install Node.js:** Node-RED is built on **Node.js**, so **Node.js** must be installed on the system before running Node-RED. Download the installer for the operating system.
- **Install Node-RED:** Once **Node.js** is installed, open a terminal or command prompt and run the command `"npm install -g node-red"` to install Node-RED globally on your system.
- **Start Node-RED:** To start Node-RED, run the command `"node-red"` in the terminal or command prompt. By default, Node-RED will listen on port 1880, so you can access the editor by opening a web browser and navigating to `"http://localhost:1880"`.
- **Configure the Node-RED server:** Once the server is running, one can access the **settings.js** file in the Node-RED directory, which contains various settings that can be configured, such as the HTTP port, the credentials for the admin interface, and the storage settings.
- **Install additional nodes:** Node-RED comes with a set of built-in nodes, but one can also install additional nodes to add extra functionality. New nodes can be installed by using the "Manage Palette" option in the menu or by using the command `"npm install <node-name>"` in the terminal.
- **Create a flow:** With the Node-RED server running, one can now create a flow by dragging and dropping nodes from the palette onto the canvas and connecting them together.
- **Deploy the flow:** Once the flow has been created, one can deploy it by clicking the "Deploy" button in the top right corner of the editor. This will activate the flow and start processing the data.

¹This is not difficult to configure, but do cost lots of time on it. So these parts(MQTT and Node-Red Configuration) are finally decided to be written here.

The topology of the Node-Red can be seen from the figure below: So far, the con-

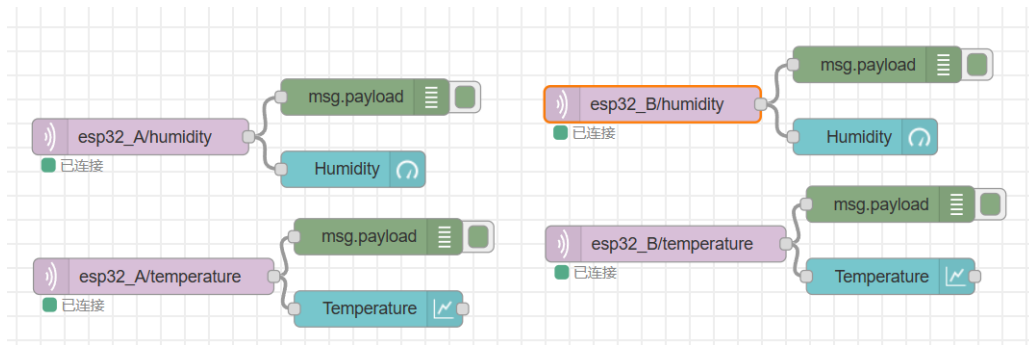


Figure 4.1: Caption

figuration of Node-Red has succeed. The following chapters will begin to demonstrate how to realize the data transmission and receiving and also the data encryption and data decryption.

5 Data Transmission and Receiving

In this section, the data transmission and receiving between two ESP32 boards via MQTT server will be demonstrated.

5.1 Un-encrypted data transmission and receiving

Process demonstrated in this section allows the two boards to share and display each other's data on their serial monitors, and also provides a way to visualize the data in real-time using Node-Red dashboard.

1. DH11 sensor on Board A collects 2 bytes of temperature data and 2 bytes of humidity data, and then sends this data to the MQTT server.
2. When the MQTT server receives the data from Board A, it creates a new topic (Topic A) and stores the data sent from Board A in that topic. Node-Red, a visual programming tool, can then access the data stored in MQTT Topic A and display it in a dashboard.
3. Board B can then read and obtain the data stored in Topic A from the MQTT server, and display that data in the serial monitor.
4. Additionally, Board B can also collect temperature and humidity data using its DH11 sensor, and send that data to the MQTT server. The MQTT server will then create a new topic (Topic B) for this data.
5. Node-Red can also access the data stored in Topic B, and display it in its dashboard.
6. Board A can also obtain the data sent by Board B from Topic B in the MQTT server, and display that data in the serial monitor.

The sketch map can be seen as shown in 5.1.

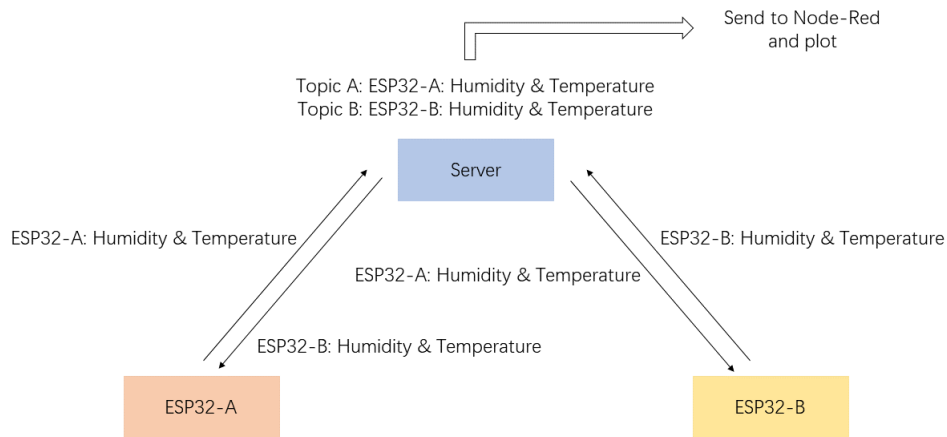


Figure 5.1: Data Transmission and Receiving

5.2 Encrypted data transmission and receiving

This process has almost the same step as the un-encrypted data transmission and receiving:

1. DH11 sensor on Board A collects 2 bytes of temperature data and 2 bytes of humidity data, and use encryption method introduced in Chapter 6 to encrypt them. Then the encrypted data will be sent to the MQTT server.
2. When the MQTT server receives the encrypted data from Board A, it creates a new topic (Topic A) and stores the data sent from Board A in that topic.
3. Board B can then read and obtain the data stored in Topic A from the MQTT server. After passing the integrity check and authentication check it can be decrypted via a AES key and display the decrypted version of data in the serial monitor.
4. Additionally, Board B can also collect temperature and humidity data using its DH11 sensor and encrypt them. After encryption they will be sent to the MQTT server. The MQTT server will then create a new topic (Topic B) for this encrypted data.
5. Board A can also obtain the data sent by Board B from Topic B in the MQTT server and display that data in the serial monitor after decryption(also integrity check, authentication check and AES decryption).

The detailed steps are show in the 5.2.

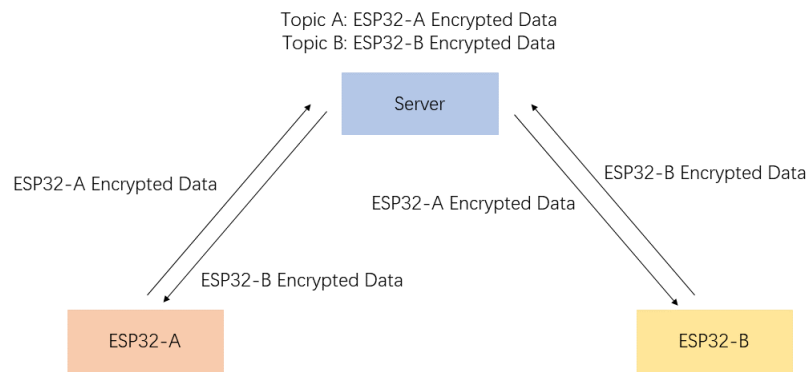


Figure 5.2: Caption

6 Data Encryption and Decryption

6.1 Encryption

One possible way to protect temperature and humidity data collected from the DH11 sensor is to use AES encryption, Hash, and HMAC for confidentiality and integrity, respectively.

- AES (Advanced Encryption Standard) is a symmetric encryption algorithm that is used to encrypt and decrypt data. It uses a secret key to encrypt plaintext and decrypt ciphertext. AES is considered to be very secure and is widely used in various applications such as disk encryption, VPNs, and wireless communication.
- Hash is a one-way function that takes an input (or ‘message’) and returns a fixed-size string of characters, which is usually a ‘digest’ that is unique to the original input. The common use of hash is to ensure data integrity and to index data in a database or in a data structure such as a hash table.
- HMAC, or Hash-based Message Authentication Code, is a specific type of keyed hash function that combines a hash function with a secret key. The key is used to produce a message authentication code (MAC) that is sent along with the message to ensure its authenticity. The recipient can then use the same key to verify the authenticity of the message by recalculating the MAC and comparing it to the one received.

The AES and HMAC keys would be stored locally on the ESP32 device. The process of protecting the data would involve the following steps:

1. Collect temperature and humidity data from the DH11 sensor.
2. Combine the data into a 4-byte plaintext.
3. Pad the plaintext to 16 bytes using PKCS5 padding method. Shown as below:

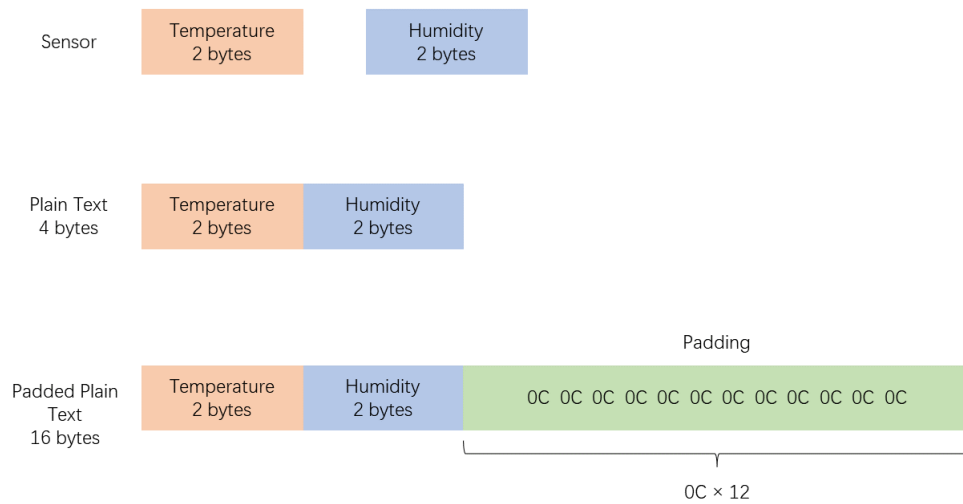


Figure 6.1: Encryption

4. Use the AES key to encrypt the padded plaintext and obtain a 16-byte ciphertext.
5. Apply HMAC on the 16-byte ciphertext to get a 32-byte authentication check data.
6. Hash the ciphertext to get 32-byte integrity check data.
7. Store the length of ciphertext in the last byte.
8. Combine the 16-byte ciphertext, 32-byte authentication check data, 32-byte integrity check data and the last byte indicating the length of cipher-text together. Shown as below:

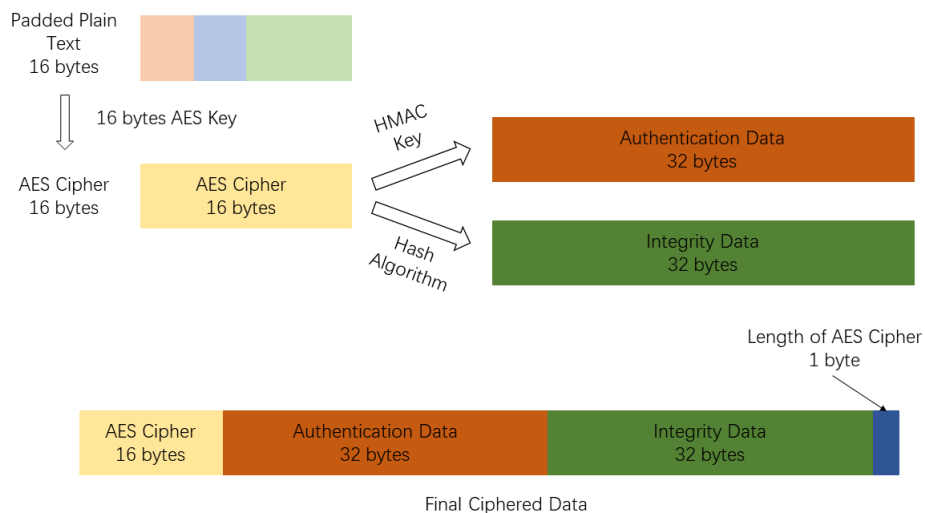


Figure 6.2: Encryption

This method provides both confidentiality and integrity for the temperature and humidity data, ensuring that the data is both protected from unauthorized access and tamper-proof.

6.2 Decryption

One way to process the received data on the receiver ESP32 device is as follows:

1. First, the 81 byte data is divided into three parts based on the value of the last byte, which indicates the length of the ciphertext: the 16 byte AES cipher, the 32 byte authentication data, and the 32 byte integrity data.
2. Next, the HMAC is applied on the received AES cipher to recalculate the authentication check data. The hash is also applied on the received AES cipher to recalculate the integrity check data.
3. The recalculated data is then compared with the received authentication and integrity data. If the recalculated data matches the received data, it is assumed that the data is authentic and has not been tampered with.
4. Finally, the AES decryption is applied on the 16 byte ciphertext to get the padded plaintext. The padding data is then removed, and the original plaintext is successfully obtained.

Shown as below:

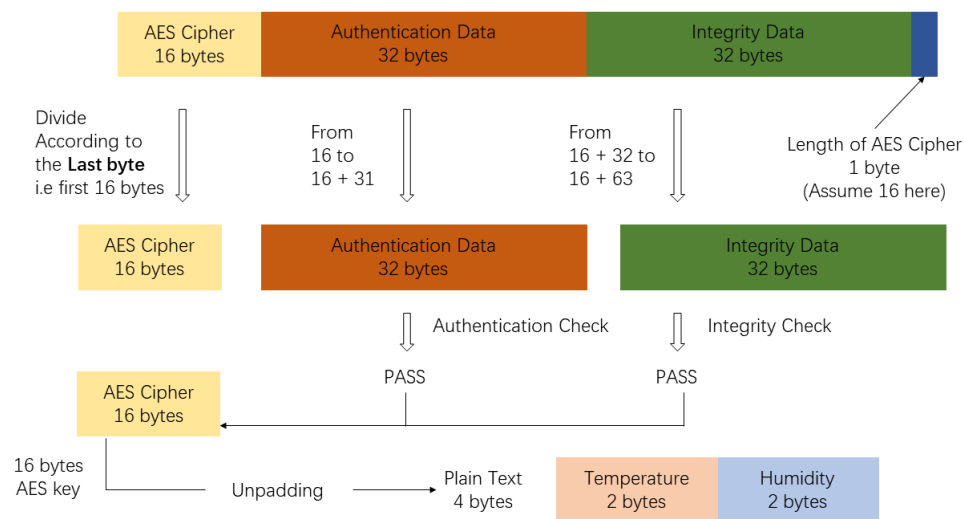


Figure 6.3: Decryption

7 Result

During this project, a Data Encryption and Transmission Based on ESP32 Boards System has been successfully implemented. The final result of the un-encryption data transmission and receiving are shown as below:

```
1674848112: Sending PUBLISH to ESP32_A (d0, q0, r0, m0, 'dataESP2', ... (4 bytes))
1674848114: Received PUBLISH from ESP32_A (d0, q0, r0, m0, 'esp32_A/temperature', ... (5 bytes))
1674848114: Sending PUBLISH to nodered_c980038239158254 (d0, q0, r0, m0, 'esp32_A/temperature', ... (5 bytes))
1674848114: Received PUBLISH from ESP32_A (d0, q0, r0, m0, 'esp32_A/humidity', ... (5 bytes))
1674848114: Sending PUBLISH to nodered_c980038239158254 (d0, q0, r0, m0, 'esp32_A/humidity', ... (5 bytes))
1674848114: Received PUBLISH from ESP32_A (d0, q0, r1, m0, 'dataESP1', ... (4 bytes))
1674848114: Sending PUBLISH to ESP32_B (d0, q0, r0, m0, 'dataESP1', ... (4 bytes))
1674848116: Received PUBLISH from ESP32_B (d0, q0, r0, m0, 'esp32_B/temperature', ... (5 bytes))
1674848116: Sending PUBLISH to nodered_c980038239158254 (d0, q0, r0, m0, 'esp32_B/temperature', ... (5 bytes))
1674848117: Received PUBLISH from ESP32_B (d0, q0, r0, m0, 'esp32_B/humidity', ... (5 bytes))
1674848117: Sending PUBLISH to nodered_c980038239158254 (d0, q0, r0, m0, 'esp32_B/humidity', ... (5 bytes))
1674848117: Received PUBLISH from ESP32_B (d0, q0, r1, m0, 'dataESP2', ... (4 bytes))
```

Figure 7.1: Un-encryption Data Transmission and Receiving

One can see from the 7.1 that:

1. MQTT received the data sent from the ESP32_A
2. MQTT received the data sent from the ESP32_B.
3. MQTT sends the data of ESP32_A to the Node-Red.
4. MQTT sends the data of ESP32_B to the Node-Red.
5. MQTT sends the data of ESP32_A to the ESP32_B.
6. MQTT sends the data of ESP32_B to the ESP32_A.

And the final result of the Node-Red is shown as follow:

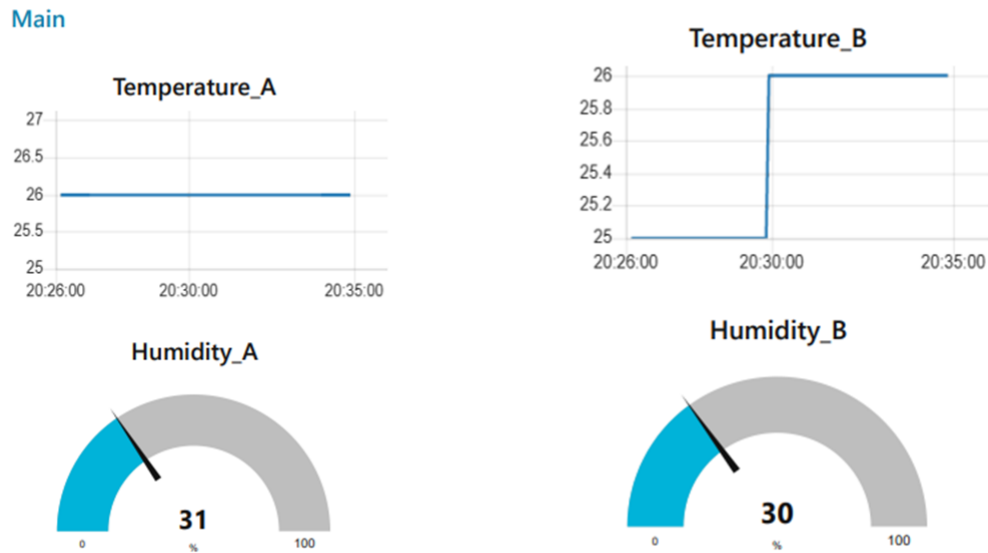


Figure 7.2: Node-Red result

After encryption, one can see the encrypted version of data from MQTT as shown below:

```
1674848500: Sending PUBLISH to ESP32_A (d0, q0, r0, m0, 'encryptESP2', ... (81 bytes))
1674848503: Received PUBLISH from ESP32_A (d0, q0, r1, m0, 'encryptESP1', ... (81 bytes))
1674848503: Sending PUBLISH to ESP32_B (d0, q0, r0, m0, 'encryptESP1', ... (81 bytes))
1674848503: Received PUBLISH from ESP32_B (d0, q0, r1, m0, 'encryptESP2', ... (81 bytes))
```

Figure 7.3: Encryption version of data

One can see that the data has all been successfully encrypted, and the length of each set of data is obviously

$$\begin{aligned}
 length &= length_of_AES + length_of_Authentication + length_of_integrity + length \\
 &= 16bytes + 32bytes + 32bytes + 1bytes = 81bytes
 \end{aligned}$$

And the decryption of data is shown as below:

```
20:40:13.986 -> Message arrived on topic:
20:40:13.986 -> encryptESP1
20:40:13.986 ->
20:40:13.986 -> Received data length :81
20:40:13.986 -> Received Encryption data length :16
20:40:13.986 -> Received Authentication data length :32
20:40:13.986 -> Received integrity data length :32
20:40:13.986 ->
20:40:13.986 -> Authentication check success
20:40:13.986 ->
20:40:13.986 -> integrity check success
20:40:13.986 ->
20:40:13.986 -> Deciphered Hex:
20:40:13.986 -> 32 36 33 31 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c
20:40:14.032 ->
20:40:14.032 -> unpadding_decrypted HEX :
20:40:14.032 -> 32 36 33 31
20:40:14.032 ->
20:40:14.032 -> unpadding_decrypted TEXT :
20:40:14.032 -> Received temperature:
20:40:14.032 -> 26
20:40:14.032 -> Received Humidity:
20:40:14.032 -> 31
20:40:15.504 -> how many stack meamory left for sensor task (byte)6436
```

Figure 7.4: Decryption of data

One can see from the 7.4 that only after the Authentication check and integrity check have all successfully passed, the AES decryption part will start to work. After decryption and un-pad, the decrypted version of data can be easily obtained, i.e. temperature of 26 and humidity of 31.

For complete code and reference, one can obtain them from the link below:

<https://github.com/KJialun/Operating-system-security-project>