VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

# PROJECT REPORT

## Logic Design Project

**TOPIC: SMART GARDENING PROJECT**

**HK251**
**INSTRUCTOR: M.Eng. NGUYEN THIEN AN**

| student's name | Student's ID |
|---|---|
| Nguyen Viet Trung | 2353251 |
| Nguyen Hieu Trung | 2353244 |
| Nguyen Tam Long | 2352694 |
| Pham Minh Nhat | 2352863 |
| Tran Quoc Thang | 2353125 |

**Ho Chi Minh City, 2025**

**GROUP WORK RESULTS REPORT**

| Number | Full name | Student ID | Contribution |
|---|---|---|---|
| 1 | Nguyen Viet Trung | 2353251 | 100% |
| 2 | Nguyen Hieu Trung | 2353244 | 100% |
| 3 | Nguyen Tam Long | 2352694 | 100% |
| 4 | Pham Minh Nhat | 2352863 | 100% |
| 5 | Tran Quoc Thang | 2353125 | 100% |

# Contents

## List of Tables

## List of Figures

# 1 Executive Summary

## 1.1 Goal

Automate tree washing and irrigation based on real conditions and schedules; reduce human effort; save water versus manual methods; keep foliage clean for better photosynthesis and urban aesthetics.

## 1.2 KPIs

According to the project goals:

- High system uptime speed.
- Water saving efficiency.
- Launching flexibility.

# 2 Context and Scope

## 2.1 Context and Need

- **Dusty urban areas:** Require periodic washing of leaves and trunks to improve photosynthesis and appearance.
- **Campuses/parks/farms:** Require irrigation optimized by soil moisture and weather.

## 2.2 Use Cases

The system is designed for the following scenarios:

1. **Home Balcony:** 1-4 pots; mini pump + mist nozzles; app on/off; morning/evening schedule; rain skip.
2. **Campus/Park:** 10-50 watering points; zoning; per-zone water statistics; alert for low tank/pump jam.
3. **Urban washing:** Light-pressure wash cycles at night/off-peak; prioritize cleaning leaves/trunks without wasting water.

## 2.3 In Scope & Out Scope

- **In scope:** Washing/irrigation automation, remote monitoring, scheduling, alerts, OTA updates, dashboard.
- **Out scope:** Computer vision, autonomous tanker vehicles, automated fertilization (placed on roadmap).

# 3 System Requirements

## 3.1 Functional Requirements

The system must fulfill the following functions:

- Read sensors: soil moisture, rain, light, flow, ambient temperature/humidity.
- Control pump/valves by thresholds, schedule, and manual command from app/web.
- Skip washing/irrigation when it rains; stop on detecting leaked water in the dry environment.
- Scheduling (cron-like), multi-zone control, manual override with safe timeout.
- Log data & send telemetry (MQTT/HTTPS); buffer when offline.
- OTA firmware; remote configuration (Wi-Fi, thresholds, schedules, zones, etc.).
- Alert: tank low/dry-run/unusual pressure-flow/disconnect.

### 3.2 Non-Functional Requirements

The design adheres to these standards:

- Outdoor durability; UV-resistant; waterproof cabling and rust protection.
- Security: TLS, key-management, secure boot.
- Control latency (delay) $< 2s$ via LAN and $< 5s$ via cloud.
- Power: Battery/solar on 12V adapter; deep-sleep, watchdog, auto recovery.
- Maintainable: Replace filters/nozzles; check valves/pump; hot-swap node.

## 4 System Architecture

### 4.1 Logic Flow

The operational logic is defined as follows:

1. **Sensor reading:** Soil moisture, rain detection, light, temperature & humidity.
2. **Condition check:**
   - IF soil moisture < threshold (e.g., 40%).
   - AND rain sensor indicates "dry".
3. **Decision:** ESP32 triggers MOSFET $\rightarrow$ pump ON.
4. **Water amount calculation:**
$$T_{on} = \frac{V_{water,need}}{Q_{pump}} \tag{1}$$
   Example: $V_{water,need} = 0.6L$, $Q_{pump} = 1.2L/min \Rightarrow T_{on} = 30s$.
5. **Update:** After watering, pump OFF, ESP32 logs data and returns to deep-sleep.
6. **Repeat cycle:** Re-check every $T_{check}$ (e.g., 10-15 minutes).

### 4.2 Flow Chart

The overall schematic of the proposed irrigation system includes:

- **Laptop:** Connected via Wi-Fi for monitoring.
- **ESP32:** Central processing unit.
- **LM393 boards:** Interface for sensors.
- **Sensors:** Provide environmental data.
- **Pumping Motor:** Actuator controlled by the system.

## 5 Hardware Design

### 5.1 Material

The following table lists the required components:

Table 1: Materials list

| **Type** | **Device** | **Note** |
|---|---|---|
| MCU | ESP32 | Wi-Fi, OTA, TLS |
| Soil sensor | Soil Moisture Sensor | Active when the soil moisture is over 60 percent. |
| Rain sensor | Rain Water Sensor | Detect current weather |
| Temp/Humid | Temperature DHT22/11 | Detect temperature and humidity |
| Driver | MOSFET (F5305S) | Control the pump |
| Water Pump | Water Pump 12VDC | Water pumping when matching right conditions |
| Power | 12V-2A adapter | Maintain the pump anytime |

## 5.2 Schematic

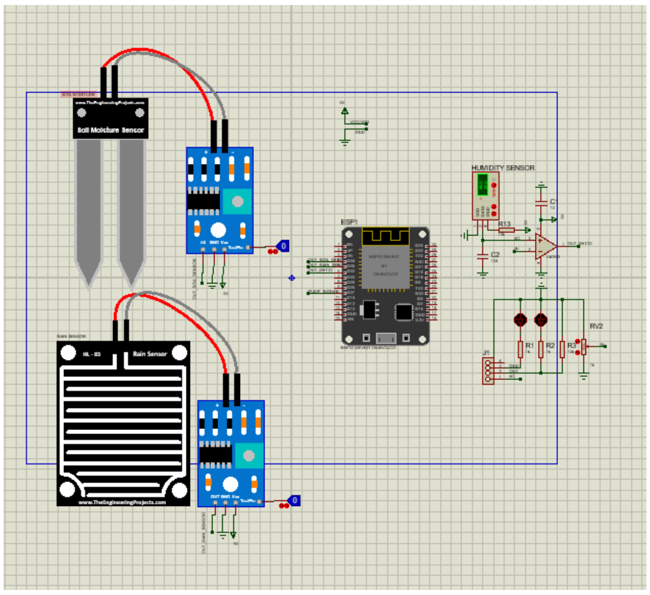

Figure 1: Schematic of the project

## 5.3 Calculation

### 5.3.1 Assumption

Parameters for calculation:

- Sources:
  - Pump source: Adapter AC - DC 12 - 2A. So, $P_{max} = 12V \times 2A = 24W$.
  - MCU source: USB port 5V - 500mA(Wi-Fi activated).
- Pump Load (R385):
  - Operating voltage: 6 - 12V.
  - Operating current: 0.5 - 0.7A.
  - Flow rate: 1-2L/min.
- Control load: ESP32 + Sensors:
  - Power consumption: 120mA(Wi-Fi activated).

### 5.3.2 Power margin check

The power margin of the water pump is calculated to ensure the adapter is not overloaded:

$$SF = \frac{I_{adapter-max}}{I_{pump}} = \frac{2.0A}{0.7A} \approx 2.85 \tag{2}$$

Conclusion: The 12V - 2A adapter is sufficient in this case, operating at approximately 35 percent of its maximum capacity.

Otherwise, the ESP32 is isolated from the pump's power rail, powered directly via USB:

$$I_{mcu} \approx 120mA < I_{USB-limit} = 500mA \tag{3}$$

Conclusion: The USB interface provides ample current for the MCU and sensors, and the physical isolation from the 12V pump circuit prevents potential brownout resets caused by inductive voltage drops.

### 5.3.3 Energy consumption estimate

The daily energy consumption is estimated for operational cost analysis (assuming a standard 10-second watering cycle per day).

$$E_{total} = E_{control} + E_{pump} \tag{4}$$

- Control unit:

$$E_{control} = 5V \times 0.12A \times 24h = 14.4Wh/day \tag{5}$$

- Pump unit:

$$E_{pump} = 12V \times 0.7A \times \frac{10}{3600} \approx 0.0233Wh/day \tag{6}$$

- Total energy:

$$E_{total} = 14.4 + 0.0233 = 14.4233Wh/day \tag{7}$$

## 5.4 Source code

### 5.4.1 Configuration pin - globalize valuation

Table 2: ESP32 Pin Configuration

| No. | ESP32 Pin | Device / Note |
|---|---|---|
| 1 | GPIO 34 | Soil Moisture Sensor (Analog Input) |
| 2 | GPIO 39 | Rain Sensor (Digital Input Only) |
| 3 | GPIO 32 | DHT22 Sensor (Digital I/O) |
| 4 | GPIO 33 | MOSFET (Active High) |

For updating the value for the server, all *glob* variables are responsible for this job.
In this system, we apply the FreeRTOS, which separates the system into many tasks and manages them with the scheduler.
Using a semaphore in the FreeRTOS structure ensures the system executes only one task at a time, which prevents to the wrong input signal for the MCU.

```
float glob_temp = 0;
float glob_humid = 0;
float glob_soil= 0;
float glob_total_ml = 0;
float glob_rain = 0;
```

```
bool glob_pump_running = false;

SemaphoreHandle_t xSensorMutex = NULL;
SemaphoreHandle_t xSerialMutex = NULL;
SemaphoreHandle_t xMqttMutex = NULL;
```

### 5.4.2 Soil sensor

```
void soil_sensor(void *pvParameter)
{
    int soil_val = 0;
    int curr_soil_per = 0;

    while (1)
    {
        soil_val = analogRead(SOIL_PIN);

        curr_soil_per = map(soil_val, 4095, 0, 0, 100);

        if (xSensorMutex != NULL &&
            xSemaphoreTake(xSensorMutex, portMAX_DELAY) == pdTRUE)
        {
            glob_soil = curr_soil_per;
            xSemaphoreGive(xSensorMutex);
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

### 5.4.3 Humidity and temperature sensor

```
void temp_humid_sensor(void *pvParameter)
{
    dht.begin();

    float current_temp = 0;
    float current_humid = 0;
    while (1)
    {
        current_temp = dht.readTemperature();
        current_humid = dht.readHumidity();
        if (xSensorMutex != NULL &&
            xSemaphoreTake(xSensorMutex, portMAX_DELAY) == pdTRUE)
        {
            glob_temp = current_temp;
            glob_humid = current_humid;
            xSemaphoreGive(xSensorMutex);
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

### 5.4.4 Rain sensor

```
void rain_sensor(void *pvParameter)
{
    pinMode(RAIN_PIN, INPUT);

    while (1)
    {
        int rain_value = analogRead(RAIN_PIN);
        if (xSensorMutex != NULL &&
            xSemaphoreTake(xSensorMutex, portMAX_DELAY) == pdPASS)
        {
            glob_rain = rain_value;
            xSemaphoreGive(xSensorMutex);
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

### 5.4.5 WI-FI connection

To use the system wirelessly, we implement the task WiFi to control the MCU and check them every 5 second.

```
void task_WiFi(void *pvParameter)
{

    bool logged = false;

    while (1)
    {
        if (WiFi.status() != WL_CONNECTED)
        {
            logged = false;

            if (xSerialMutex != NULL &&
                xSemaphoreTake(xSerialMutex, portMAX_DELAY) == pdPASS)
            {
                Serial.print("WiFi not connected, start connecting");
                WiFi_Connect();
                xSemaphoreGive(xSerialMutex);
            }
        }
        else if (!logged)
        {
            if (WiFi.status() == WL_CONNECTED)
            {
                if (xSerialMutex != NULL &&
                    xSemaphoreTake(xSerialMutex, portMAX_DELAY) == pdPASS)
                {
                    Serial.print("WiFi connected, IP: ");
                    Serial.println(WiFi.localIP());
                    xSemaphoreGive(xSerialMutex);
                }
            }

            logged = true;
        }
        else if (logged)
        {
            if (xSerialMutex != NULL &&
                xSemaphoreTake(xSerialMutex, portMAX_DELAY) == pdPASS)
            {
                Serial.println("WiFi still connected");
                xSemaphoreGive(xSerialMutex);
            }
        }

        vTaskDelay(pdMS_TO_TICKS(5000));
    }
}
```

# 6 Detailed Implementation and Security

This section expands on the general design with specific technical details regarding hardware connections, communication protocols, and security measures.

## 6.1 SHA-256 Data Encoding Module

### 6.1.1 Overview

The following module is responsible for collecting five floating-point sensor values, serializing them into a 20-byte buffer, hashing that buffer using the SHA-256 algorithm (from the mbedTLS library), and finally producing a 52-byte output. The output consists of the raw input data followed by the computed hash digest:

$$\text{output52} = \text{input}[20] \parallel \text{digest}[32]$$

This design makes it possible to transmit the original sensor data together with a hash-based integrity check.

### 6.1.2 Source Code

```
uint8_t output52[52];

void sha2(float temp, float hum, float soid, float total_ml, float rain){
    uint8_t input[20];

    memcpy(input + 0,  &temp,     4);
    memcpy(input + 4,  &hum,      4);
    memcpy(input + 8,  &soid,     4);
    memcpy(input + 12, &total_ml, 4);
    memcpy(input + 16, &rain,     4);

    uint8_t out32[32];

    mbedtls_sha256_context ctx;
    mbedtls_sha256_init(&ctx);
    mbedtls_sha256_starts_ret(&ctx, 0);
    mbedtls_sha256_update_ret(&ctx, input, 20);
    mbedtls_sha256_finish_ret(&ctx, out32);

    memcpy(output52,      input, 20);
    memcpy(output52 + 20, out32, 32);
}
```

### 6.1.3 Function Explanation

- Five floating-point values are packed into a 20-byte array using `memcpy`, each occupying 4 bytes (IEEE-754 binary representation).
- The SHA-256 hashing context is initialized, updated with the 20 bytes of input, and finalized to produce a 32-byte digest.
- The final output array is formed by concatenating:
  1. the original 20 bytes of raw input data,
  2. the 32-byte SHA-256 hash digest.
- The resulting 52-byte buffer can be transmitted or stored for later verification.

### 6.1.4 Data Flow Diagram

$$\underbrace{4B_{temp} + 4B_{hum} + 4B_{soil} + 4B_{total\_ml} + 4B_{rain}}_{\text{20 bytes input}} \xrightarrow{\text{SHA-256}} \text{digest (32 bytes)}$$

$$\text{output52} = \boxed{\text{input (20B)}} \;||\; \boxed{\text{SHA-256 digest (32B)}}$$

### 6.1.5 Notes and Considerations

- The floats are hashed in their raw binary form, meaning the output may vary across platforms with different endianness. For cross-platform reproducibility, data should be normalized (e.g., convert to big-endian or text representation).
- SHA-256 provides integrity but **not encryption**. The original data is still readable.
- For authentication (preventing forgery), the system should use HMAC-SHA256 with a secret key.
- Error checking and `mbedtls_sha256_free` are recommended for robustness in production code.

## 6.2 Task Synchronization using Semaphore & Mutex

In a multi-tasking FreeRTOS environment, sensor acquisition, control logic, and communication processes run concurrently on the ESP32. Without proper synchronization, shared variables such as temperature, humidity or soil moisture could be read at the same time they are written, leading to corrupted values, unstable control behavior or wrong telemetry packets.

To guarantee safe data sharing between tasks, the system applies **mutex** / **binary semaphore** mechanisms:

- **xSensorMutex** – protects access to global sensor data `glob_temp, glob_humid, glob_soil, glob_rain`.
- **xSerialMutex** – mutex for serial console printing to avoid overlapped or interleaved logs.
- **xMqttMutex** – ensures only one MQTT publish routine executes at a time, preventing memory corruption during JSON serialization.
- **xPumpMutex** – serializes access to the pump so that only one watering session (automatic or manual) can run at a time.

**Reasons for using mutexes:**

1. Prevent race conditions between sensor tasks updating values simultaneously.
2. Ensure atomic reads when the MQTT task accesses shared data for publishing.
3. Avoid serial-output corruption caused by concurrent print calls.
4. Guarantee exclusive control over pump start/stop logic.

### 6.2.1 Data Flow Model with Mutex Protection

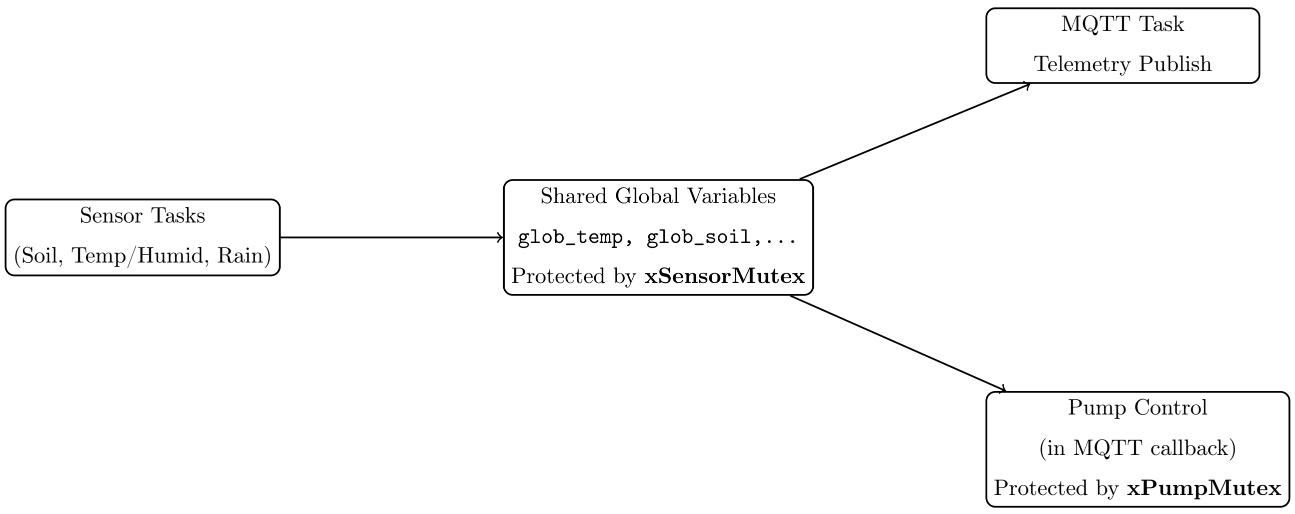

Figure 2: Mutex-based data flow between sensor tasks, shared state, telemetry and pump control

Shared global data is only accessed when the mutex is locked:

```
// Write (Sensor Task)
if (xSemaphoreTake(xSensorMutex, portMAX_DELAY) == pdTRUE) {
    glob_temp  = current_temp;
    glob_humid = current_humid;
    glob_soil  = current_soil;
    glob_rain  = current_rain;
    xSemaphoreGive(xSensorMutex);
}

// Read (MQTT Task or Pump Control Path)
if (xSemaphoreTake(xSensorMutex, portMAX_DELAY) == pdTRUE) {
    float t = glob_temp;
    float h = glob_humid;
    float s = glob_soil;
    float r = glob_rain;
    xSemaphoreGive(xSensorMutex);
}
```

### 6.2.2 Pump Control and Mutual Exclusion

Pump operation is exposed through a small hardware abstraction layer: `pump_init()`, `pump_start(durationMs, mode)`, `pump_stop(runTimeMs)` and `pump_set_mode(mode)`. These functions drive `PUMP_PIN` and update global state (`glob_pump_running`); all concurrency guarantees are enforced at the task level.

In the current implementation, all pump commands originate from the MQTT callback. The downlink payload is a fixed-length binary frame; after SHA-256 verification, three fields are decoded:

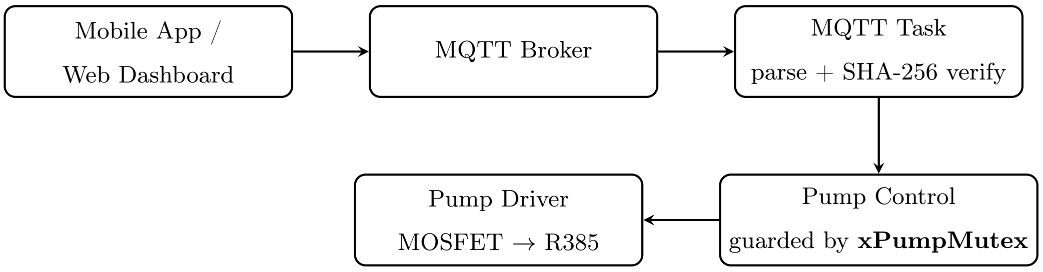

Figure 3: Command execution path for manual/remote pump control

Before calling `pump_start` or `pump_stop`, the callback must take `xPumpMutex`. Only when the mutex is acquired will the pump state and mode be changed:

```
// Inside mqttCallback after hash verification
if (xPumpMutex != NULL &&
    xSemaphoreTake(xPumpMutex, portMAX_DELAY) == pdPASS)
{
    if (pump_control == 1) {
        pump_start(duration * 1000, mode);
    } else if (pump_control == 0) {
        pump_stop(duration);
    }

    pump_set_mode(mode);
    xSemaphoreGive(xPumpMutex);
}
else
{
    Serial.println("[MQTT] Cannot take xPumpMutex, command ignored");
}
```

This design ensures that:

- Only one pump command sequence is active at a time, even if future automatic tasks are added.
- Manual commands cannot overlap and generate conflicting ON/OFF transitions on the motor.
- The actuation path (MQTT callback → pump driver) is deterministic and free of race conditions.

### 6.2.3 Secure Command Path Flow

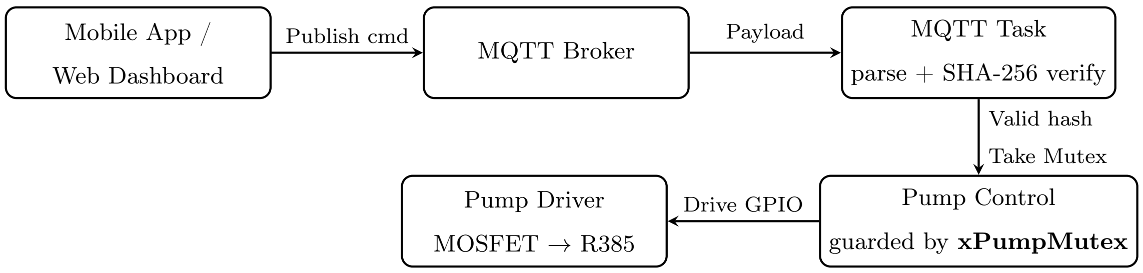

Figure 4: Command execution path for manual/remote pump control

The downlink command must pass several safety layers:

1. The mobile/web client sends a binary control frame through the MQTT broker.
2. The ESP32 MQTT task receives the payload and recomputes the SHA-256 hash to verify integrity.
3. If the hash is valid, the task attempts to acquire `xPumpMutex` and, if successful, executes `pump_start` or `pump_stop` and updates the mode.
4. If the hash check fails or the mutex cannot be acquired, the command is rejected safely and the current watering session continues unaffected.

**Conclusion** By implementing semaphore–mutex protection, communication between sensing tasks, communication tasks and the pump actuator is safe, atomic and deadlock-free. Data integrity during MQTT uplink and exclusive access to the pump during downlink commands are preserved, ensuring reliable cloud monitoring and stable irrigation execution.

## 6.3 MQTT Module Overview

The MQTT Task (`taskMqtt`) is a critical component responsible for Internet of Things (IoT) connectivity. It abstracts the complexities of the TCP/IP stack and the MQTT protocol, providing a seamless interface for control and monitoring.

**Key Responsibilities:**

- **Connectivity Management:** Automatically establishes and maintains Wi-Fi and MQTT connections.
- **Telemetry Uplink:** Periodically aggregates sensor data (Temperature, Humidity, Soil Moisture, Rain) and publishes it to the cloud.
- **Command Downlink:** Listens for remote control commands to operate the water pump.
- **Security:** Implements SHA-256 hashing to verify the integrity and authenticity of received commands.

### 6.3.1 System Configuration & Dependencies

**Library Dependencies** The module relies on the following standard and custom libraries:

| **Library** | **Purpose** |
|---|---|
| `WiFi.h` | Manages the physical layer and IP stack connection. |
| `PubSubClient.h` | A lightweight client for simple MQTT communication. |
| `ArduinoJson.h` | Efficient serialization/deserialization of JSON data. |
| `mbedtls/sha256.h` | Hardware-accelerated cryptographic hashing. |
| `pump_control.h` | Hardware abstraction layer for the water pump. |

Table 3: Library Dependencies

**Network Configuration** The system connects to a public broker for demonstration purposes but is designed to be broker-agnostic.

- **Broker Address:** `broker.hivemq.com`
- **Port:** 1883 (Non-SSL)
- **Topics:**
  - **Sensor Data (Pub):** `device/sensor/data`
  - **Command (Sub):** `device/command`

### 6.3.2 Secure Command Protocol (Downlink)

To prevent unauthorized or corrupted commands from triggering the actuators, the system employs a **Binary Protocol with SHA-256 Verification**. This is defined under the `NON_ENCRYPTION` macro in the source code.

**Packet Structure** The incoming payload must be at least **38 bytes** long. The structure is defined as follows:

| **Offset** | **Size** | **Type** | **Description** |
|---|---|---|---|
| 0 | 1 Byte | `uint8_t` | **Pump Control**: 1 (Start), 0 (Stop). |
| 1 | 4 Bytes | `uint32_t` | **Duration**: Runtime in seconds (Little Endian). |
| 5 | 1 Byte | `uint8_t` | **Mode**: 0 (Manual), 1 (Automatic). |
| 6 | 32 Bytes | `uint8_t[]` | **SHA-256 Hash**: Integrity signature. |

Table 4: Binary Command Packet Structure

**Integrity Verification Logic** Upon receiving a message, the `mqttCallback` function performs the following validation steps:

1. **Length Check:** Discards packet if `len < 38`.
2. **Extraction:** Separates the data payload (Bytes 0-5) from the received hash (Bytes 6-37).
3. **Hashing:** Uses `mbedtls_sha256` to calculate the hash of the first 6 bytes.
4. **Comparison:** Compares the calculated hash with the received hash using `memcmp`.
5. **Execution:**

- If hashes match: The pump state is updated via `pump_start()` or `pump_stop()`.
- If mismatch: Prints "HASH FAILED" and ignores the command.

### 6.3.3 Telemetry Protocol (Uplink)

The system publishes sensor data using **JSON format** for high compatibility with web dashboards and databases.

**Data Aggregation & Hashing** Before serialization, the system aggregates 5 sensor values (20 bytes total) and calculates a SHA-256 hash. This hash is stored in a global buffer `output52` for potential future verification features.

**Input Data Map:**

- Bytes 0-3: Temperature (float)
- Bytes 4-7: Humidity (float)
- Bytes 8-11: Soil Moisture (float)
- Bytes 12-15: Total Water Volume (float)
- Bytes 16-19: Rain Status (float)

**JSON Serialization** The `ArduinoJson` library is used to format the payload. The `serialized()` function is employed to format floating-point numbers to exactly 2 decimal places, optimizing the string length.

**Example Payload:**

```
{
  "temp": 28.50,
  "hum": 65.20,
  "soil": 45.00,
  "rain": 0.00,
  "water_ml": 1200.50
}
```

### 6.3.4 Task Implementation Details

The module runs as a FreeRTOS task, ensuring non-blocking operation alongside other system components.

**Initialization Phase**

- **Pump Init:** Configures GPIO pins for the relay/motor driver.
- **MQTT Setup:** Configures the Broker URL, Port, and Callback function.

**Main Loop Logic** The task executes the following operations in a continuous loop:

[Image of MQTT architecture diagram]

1. **Connection Maintenance:**
   - Checks Wi-Fi status. If disconnected, waits 2 seconds.
   - Checks MQTT status. If disconnected, attempts to reconnect using a random Client ID to prevent session conflicts.
2. **Keep-alive:** Calls `s_mqttClient.loop()` to handle incoming packets (PINGRESP, PUBLISH).
3. **Sensor Data Acquisition:**
   - Uses `xSemaphoreTake(xSensorMutex)` to safely read global variables updated by the Sensor Task.
   - This prevents race conditions where data might be read while being written.
4. **Data Sanitization:** Checks for `NaN` (Not a Number) values and resets them to 0.0 to prevent JSON errors.
5. **Publishing:** Serializes the data and publishes it to `TOPIC_SENSOR`.
6. **Timing:** Uses `vTaskDelayUntil` for a precise 5-second sampling interval.

# 7 Software System

## 7.1 Backend Logic

The backend is built using **Node.js** with the **Express** framework, offering non-blocking I/O suitable for real-time IoT applications. It serves as the bridge between the frontend dashboard, the AI service, and the MQTT broker.

The server maintains the system state in memory and uses `socket.io` to push real-time updates to connected clients.

```
// System state
let systemState = {
  pumpOn: false,
  mode: 'automatic', // 'automatic' or 'manual'
  pumpStartTime: null,
  pumpDuration: 0,
  aiEnabled: process.env.AI_ENABLED !== 'false',
  lastAIDecision: null
};

// Listen for sensor updates via MQTT
mqttClient.on('message', (topic, message) => {
    // ... parse data ...
    io.emit('sensor_update', sensorData);
});
```

Listing 1: Server State Management (server.js)

## 7.2 AI Decision Service

In the architecture of the smart watering system, the **AI Decision Service** subsystem plays the role of a central processing unit. It is responsible for analyzing real-time environmental parameters and making automated pump control decisions (ON/OFF), replacing traditional rigid control rules.

### 7.2.1 Theoretical Background

To address the binary classification problem for irrigation decisions, the team selected the **Random Forest** algorithm.

**Algorithm Theory:** Random Forest is a machine learning method belonging to the *Ensemble Learning* group. Instead of building a single Decision Tree, Random Forest constructs a collection (forest) consisting of multiple decision trees during the training process.

- **Operation Mechanism:** Each tree in the forest outputs an individual prediction (e.g., "Water" or "Do not water"). The final result of the model is determined based on the *Voting* mechanism (majority vote) from all individual trees.
- **Bagging & Random Subspace:** The algorithm utilizes Bootstrap Aggregating techniques to create random subsets of data for each tree, which enhances diversity and the stability of the model.

**Justification for Selection:** Compared to other algorithms such as Neural Networks (RNN/LSTM) or a Single Decision Tree, Random Forest was chosen because:

1. **High Accuracy:** It overcomes the *Overfitting* drawback commonly found in single decision trees.
2. **Suitability for Tabular Data:** The system's input data consists of discrete fields (Temperature, Humidity, Soil Moisture) and does not require sequential time-series processing like RNN/LSTM.
3. **Low Computational Cost:** It offers fast inference speeds, making it suitable for embedded integration or running on servers with limited resources.

### 7.2.2 Data Processing & Training

**1. Data Source:** Due to time constraints in the project implementation, collecting real-world data on crops over a long period was not feasible. The team utilized the standard **Tomato Irrigation Dataset** from the Mendeley research data repository[1]. This dataset records experimental environmental parameters affecting the growth of tomato plants.

[1]`https://data.mendeley.com/datasets/33cngpcrmx/2`

**2. Data Preprocessing:** The raw initial data contained many parameters incompatible with the current hardware (such as Nitrogen, Potassium levels, pH...). The processing was performed via the `pre_train.py` script with the following steps:

- **Cleaning and Renaming:** Removed trailing whitespace and normalized column names for easier manipulation.
- **Feature Selection:** Extracted only the 3 parameters corresponding to the system's sensors:

```python
# pre_train.py - Lines 8-12
selected_columns = ['Temperature [_ C]', 'Humidity [%]', 'Soil moisture']
df_clean = df[selected_columns].copy()
df_clean.columns = ['temp', 'hum', 'soil']
```

- **Data Labeling:** Since the original dataset lacked a pump decision column (Label), the team applied agricultural logic to automatically generate the `Pump_Action` label. This logic includes 3 priority condition branches to optimize watering:

```python
# pre_train.py - Lines 15-24: Label generation logic
def generate_label(row):
    # 1. Soil too dry (<350) -> Pump immediately
    if row['soil'] < 350:
        return 1
    # 2. Hot weather (>30 C) and soil starting to dry -> Pump to prevent heat shock
    elif row['temp'] > 30 and row['soil'] < 450:
        return 1
    # 3. Dry air (<50%) -> Pump to compensate for transpiration
    elif row['hum'] < 50 and row['soil'] < 550:
        return 1
    else:
        return 0
```

The processed result was saved as `dataset_chuan.csv` to serve the training process.

**3. Model Training:** The training process was executed in the `training.py` file using the `scikit-learn` library.

- **Data Splitting:** The data was split into 80% for Training and 20% for Testing.
- **Hyperparameters:** Configured `RandomForestClassifier` with 100 decision trees (`n_estimators=100`) and a maximum depth of 10 (`max_depth=10`) to balance complexity and performance.

```python
# training.py - Lines 17-20
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

print("training Random Forest...")
clf = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
clf.fit(X_train, y_train)
```

- **Evaluation:** The model achieved an accuracy of approximately **99.83%** on the test set, with Precision and Recall metrics reaching maximum levels for both label classes (0 and 1).
- **Serialization:** The trained model was serialized into a `model_tuoi_cay.pkl` file using the `joblib` library for reuse.

### 7.2.3 Model Integration into Backend

To integrate AI capabilities into the Node.js Web system, the team utilized a **Child Process** mechanism to create a communication bridge between the Node.js and Python environments.

**1. Python AI Service:** An independent Python script (`AI_service.py`) was built to load the `.pkl` model and perform predictions upon invocation. This script accepts input parameters via command-line arguments and returns the result in JSON format.

```python
model = joblib.load('../backend/model_tuoi_cay.pkl')

# Receive arguments from Node.js (sys.argv)
temp = float(sys.argv[1])
humid = float(sys.argv[2])
soil = float(sys.argv[3])

# Create DataFrame and Predict
iinput_data = pd.DataFrame([[temp, humid, (soil + 1)*50]],
                                        columns=['temp', 'hum', 'soil'])
prediction = model.predict(input_data)

# Return JSON result
result = {
    "action": int(prediction[0]),
    "reason": "AI Model Decision"
}
print(json.dumps(result))
```

**2. Invoking AI from Node.js Server:** Whenever the server receives new sensor data via the MQTT protocol, it triggers the prediction process. The `askPythonAI` function executes the Python script, passing the `temp, hum, soil` parameters, and awaits the returned result to control the device.

This workflow ensures the decoupling of the Server's business logic and the AI's processing logic, facilitating easy maintenance and upgrades.

## 7.3 Frontend Web Dashboard

The user interface is developed using **ReactJS** and styled with Tailwind CSS. It provides a comprehensive view of the garden's status and allows for manual intervention. Key components include:

1. **SmartWateringDashboard.jsx:** The main controller that connects to the WebSocket to display real-time sensor gauges (Temperature, Humidity, Soil) and pump status.
2. **History.jsx:** A log viewer that stores watering events (timestamp, duration, water volume) in local storage and allows CSV export.
3. **ConfirmStopPage.jsx:** A safety mechanism that requires user confirmation before interrupting an active watering cycle.

```
useEffect(() => {
    // Connect to WebSocket
    websocketService.connect();

    // Handle sensor updates from WebSocket
    const handleSensorUpdate = (data) => {
      setSensors({
        temp: data.temp || 0,
        hum: data.hum || 0,
        soil: data.soil || 0,
        level: data.level || 0,
        // ...
      });
    };

    websocketService.on('sensor_update', handleSensorUpdate);
    websocketService.on('status_update', handleStatusUpdate);

    return () => { /* cleanup */ };
}, []);
```

Listing 2: Real-time Dashboard Logic (SmartWateringDashboard.jsx)

## 8 Estimated Cost

The following table provides a cost estimation for building a single hardware node:

Table 5: Component Cost Estimation (1 Node)

| No. | Component Name | Qty | Total (VND) |
|---|---|---|---|
| 1 | ESP32-NodeMCU-32S CH340 Ai-Thinker | 1 | 190.000 |
| 2 | Soil Moisture Sensor | 1 | 12.000 |
| 3 | Rain Water Sensor | 1 | 11.000 |
| 4 | DHT22 Temperature Humidity Sensor | 1 | 103,000 |
| 5 | MOSFET F5305S | 1 | 21.000 |
| 6 | Water Pump R385(12V) | 1 | 47.000 |
| 7 | Power Adapter 12V-2A | 1 | 65.000 |
| 8 | Waterproof Enclosure | 1 | 50.000 |
| 9 | Accessories (Wires, Tubing, Nozzles) | 1 | 50.000 |
| | | **TOTAL** | **549,000** |

## 9 Conclusion

The Smart Gardening Project has been successfully designed from high-level logic down to specific hardware and software components. By combining robust hardware (ESP32, weatherproof sensors) with intelligent software (Fuzzy Logic, MQTT with Encryption), the system addresses the critical needs of urban and agricultural irrigation, ensuring water efficiency and operational reliability.

## 10 Source Code