SMART WATER MANAGEMENT SYSTEM

A PROJECT REPORT FOR EMBEDDED SYSTEM DESIGN LABORATORY

Submitted by

VISHWA R	**2022505019**
NITHISHWARAN S	**2022505306**
KALAISELVAN L	**2022505040**
VENKATAJHALAM S	**2022505005**
SRIRAM M	**2022505305**
BHARATH R	**2022505017**
JAYA SURYA B	**2022505308**

Submitted to

Dr. S. MEYYAPAN

ASSISTANT PROFESSOR

MADRAS INSTITUTE OF TECHNOLOGY

ANNA UNIVERSITY :: CHENNAI 600 044

November 2024

TABLE OF CONTENTS:

INTRODUCTION:

Gist of the experimental setup

The Smart Water Management System introduces an automated water tap system driven by motion detection and real-time scheduling. It uses an ESP32 microcontroller integrated with a PIR sensor to detect motion and open the tap for a specified duration.
The duration for which the tap remains open is determined by the system's built-in Real-Time Clock (RTC), ensuring water flow is controlled according to pre-defined time intervals. This combination of motion-triggered activation and time-based flow control provides efficient water usage while minimizing wastage.

System Workflow:

- The *PIR sensor* detects motion near the tap and signals the ESP32.
- The ESP32 consults its real-time clock to determine the activation period and controls the tap accordingly.
- After the specified duration, the tap closes automatically.

COMPONENTS USED:

- ESP-32 MCU
- PIR sensor
- A4988 driver
- Nema 17 stepper motor
- Adapter 12V
- DC barrel jack connector
- Screw driver
- voltage regulator
- Bread Board
- Jumper wires

SOFTWARES USED:

- Arduino IDE.
- ESP32 RTC Library

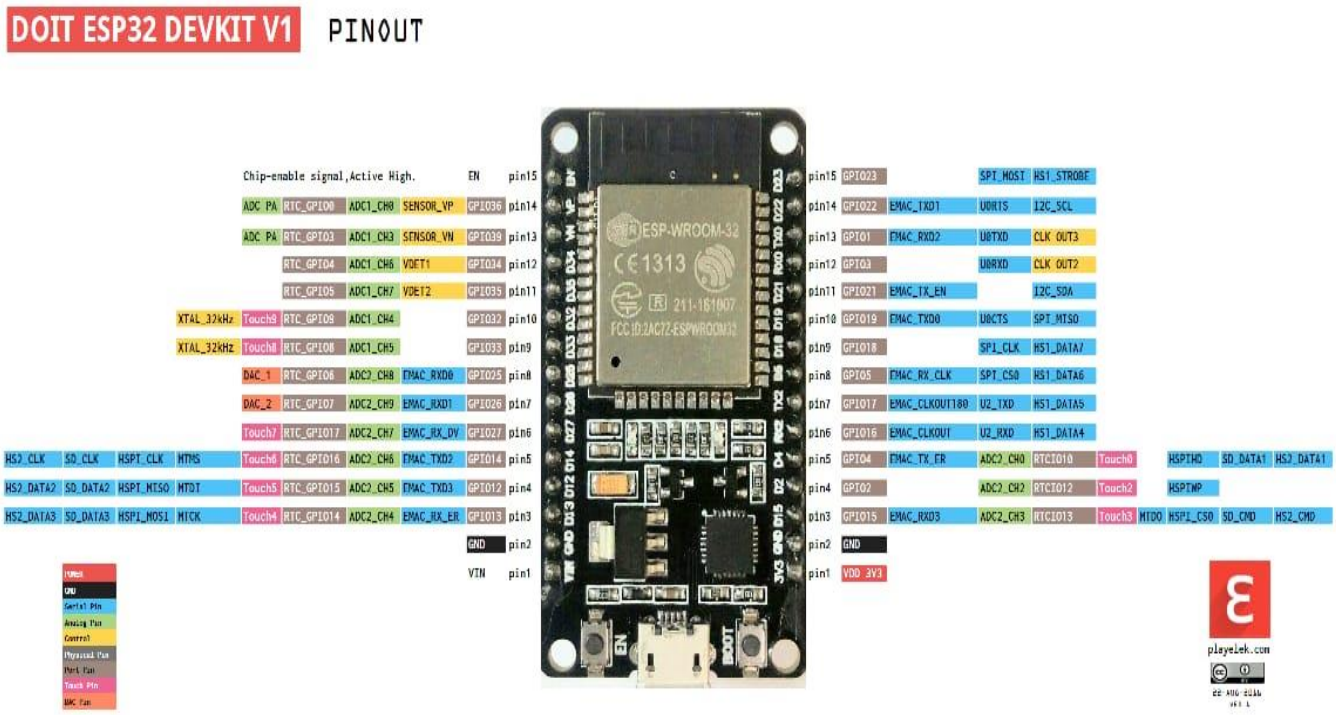
ESP32 MCU:

ESP32 comes with an on-chip 32-bit microcontroller with integrated Wi-Fi + Bluetooth + BLE features that targets a wide range of applications. It is a series of low-power and low-cost developed by Espressif Systems.

Core features:

- **Processors** – The ESP32 uses a Tensilica Xtensa 32-bit LX6 microprocessor. This typically relies on a dual core architecture, with the exception of one module, the ESP32-S0WD, which uses a single-core system.
- **Wireless connectivity** – The ESP32 enables connectivity to integrated Wi-Fi through the 802.11 b/g/n/e/i/.
- **Memory** – Internal memory for the ESP32 is as follows. ROM: 448 KB (for booting/core functions), SRAM: 520 KB (for data/instructions), RTC fast

SRAM: 8 KB (for data storage/main CPU during boot from sleep mode), RTC slow SRAM: 8 KB (for co-processor access during sleep mode)



PIR Passive Infrared Sensor:

A *Passive Infrared (PIR) Sensor* detects infrared radiation from moving objects (like humans). This sensor is used to trigger the water tap based on motion near the tap .

The PIR sensor detects changes in infrared radiation in its environment caused by movement. When motion is detected, the sensor outputs a HIGH signal to the ESP32, initiating water flow.

The PIR sensor works by detecting changes in infrared radiation. Infrared radiation is a type of electromagnetic energy emitted by objects with heat (e.g., humans, animals). The PIR sensor is equipped with two key components:

1. *Pyroelectric Sensor*: This is the heart of the PIR sensor. It detects changes in the levels of infrared radiation in its surroundings.
2. *Fresnel Lens*: This lens is placed over the pyroelectric sensor to focus infrared signals and expand the field of detection.

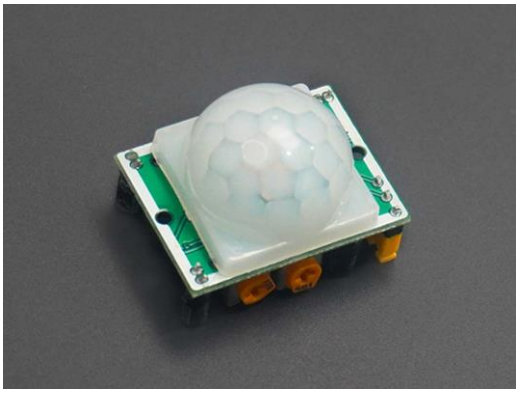
Detection Process

- The PIR sensor has two slots (part of the pyroelectric element), and both detect IR radiation.
- When the environment is stable (no motion), both slots detect similar IR levels.
- When motion occurs (e.g., a person moving), one slot detects a higher IR level than the other due to the moving heat source. This difference is processed by the sensor to generate a digital HIGH output signal.

Key Features

- *Range*: Typically detects motion up to 5–7 meters.
- *Field of View:* Usually covers a 120° angle, depending on the lens design.
- *Power Consumption*: Extremely low, making it ideal for IoT applications.
- *Output*: Digital output (HIGH when motion is detected, LOW when no motion is detected).

PIR Sensor Connection with ESP32



The PIR sensor is connected to the ESP32 microcontroller as follows:

- *VCC*: 3.3V or 5V from the ESP32 .
- *GND:* Ground pin of the ESP32.
- *OUT*: Connected to one of the GPIO pins of the ESP32. This pin sends a HIGH signal to the microcontroller when motion is detected**.**

Advantages of Using a PIR Sensor

- Low Power Consumption: It requires minimal energy, making it efficient for battery-powered or low-energy systems.
- High Sensitivity: It can detect even subtle movements within its range.
- Non-Intrusive: As it passively detects IR radiation, it does not emit any harmful waves or signals.
- Compact and Cost-Effective: Small in size and affordable, it is suitable for large-scale implementations.

Real-Time Clock (RTC):

The Real-Time Clock (RTC) functionality in the ESP32 is crucial for controlling water flow based on specific time intervals. The ESP32's built-in RTC, along with the time.h library, is used to synchronize the system with network time via an NTP (Network Time Protocol) server.

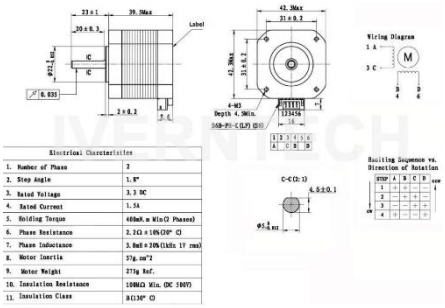
Features of RTC in ESP32

- *NTP Synchronization*: Syncs with servers like pool.ntp.org to fetch real-time clock data.
- *Time Zones*: Can be configured for local time zones (e.g., IST: UTC+5:30).
- *Low Power*: RTC operates in low-power modes, maintaining time even during deep sleep

NEMA 17 Stepper Motor

The NEMA 17 Stepper Motor is a widely used motor that provides precise control of rotation. The motor operates by moving in discrete steps, making it ideal for controlling rotational movements with high accuracy.

In this case, the NEMA 17 motor is used to control water flow by adjusting a valve when triggered by the PIR sensor.



Working Principle

A stepper motor operates by energizing coils (windings) in a specific sequence to create magnetic fields. These fields interact with the permanent magnets on the rotor to produce rotational motion. The motor moves in discrete steps, with the step size determined by the motor's design (commonly 1.8° per step for NEMA 17).

Control Process

- **Step Pulse**: Each pulse sent to the driver corresponds to one step of the motor.
- **Direction Control**: A direction pin on the driver determines the direction of rotation.
- **Speed Control**: The frequency of step pulses controls the speed of rotation

Key Features

- **Step Angle**: 1.8° per step (200 steps/revolution).
- **Torque**: Typically between 30–50 N·cm, depending on the model.
- **Voltage/Current**: Operates at 12V or higher, with a current rating between 1A and 2A.
- **Precision**: High accuracy due to discrete stepping.
- **Holding Torque**: Maintains position even without movement.

Library Modules Used

- **<time.h>**: Used for managing and configuring time in the system.
    - *configTime()*: Sets the time zone and connects to an NTP server.
    - *getLocalTime()*: Fetches the current local time in tm structure format.
    - *struct tm:* Provides access to date and time elements (e.g., hours, minutes).

Implementation in the Project

- The RTC determines the tap's active duration.
- For example, if motion is detected, the tap opens for 10 seconds, which is tracked using RTC timestamps.
- The clock resets and restarts from the beginning when the motion is detected within this 10 second.
- The RTC also ensures time-based restrictions (e.g., deactivating the system during specified hours).

A4988 Motor Driver:

The *A4988 Motor Driver* is a micro stepping driver for controlling bipolar stepper motors. It is widely used in automation and robotics due to its compact size, ease of control, and ability to support precise motor movement through micro stepping.

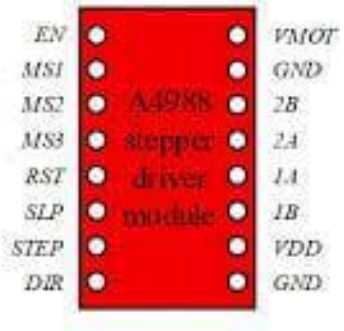
In this project, the *A4988 motor driver* controls the stepper motor that adjusts the position of the tap or other mechanisms based on real-time motion detection and RTC timing. The A4988 is connected to the ESP32 for precise movement of the stepper motor, enabling smooth and reliable operations.

Advantages of Using the A4988 Motor Driver:

1. *Precision Control*: Micro stepping provides smooth and precise motor movement, which is essential for applications requiring fine adjustments.
2. *Current Regulation*: Ensures optimal motor operation and prevents overheating.
3. *Simple Control Interface*: Only two control pins (STEP and DIR) are required for operation, simplifying ESP32 programming.
4. *Built-in Protections*: Prevents damage due to overheating or overcurrent conditions.

Pin description:

- VMOT
- GND
- VDD
- DIR
- STEP
- MS1, MS2, MS3
- ENABLE



VMOT - Power supply for the motor

GND - Ground pin for motor and logic power.

VDD - Logic voltage input (3.3V or 5V, typically from ESP32).

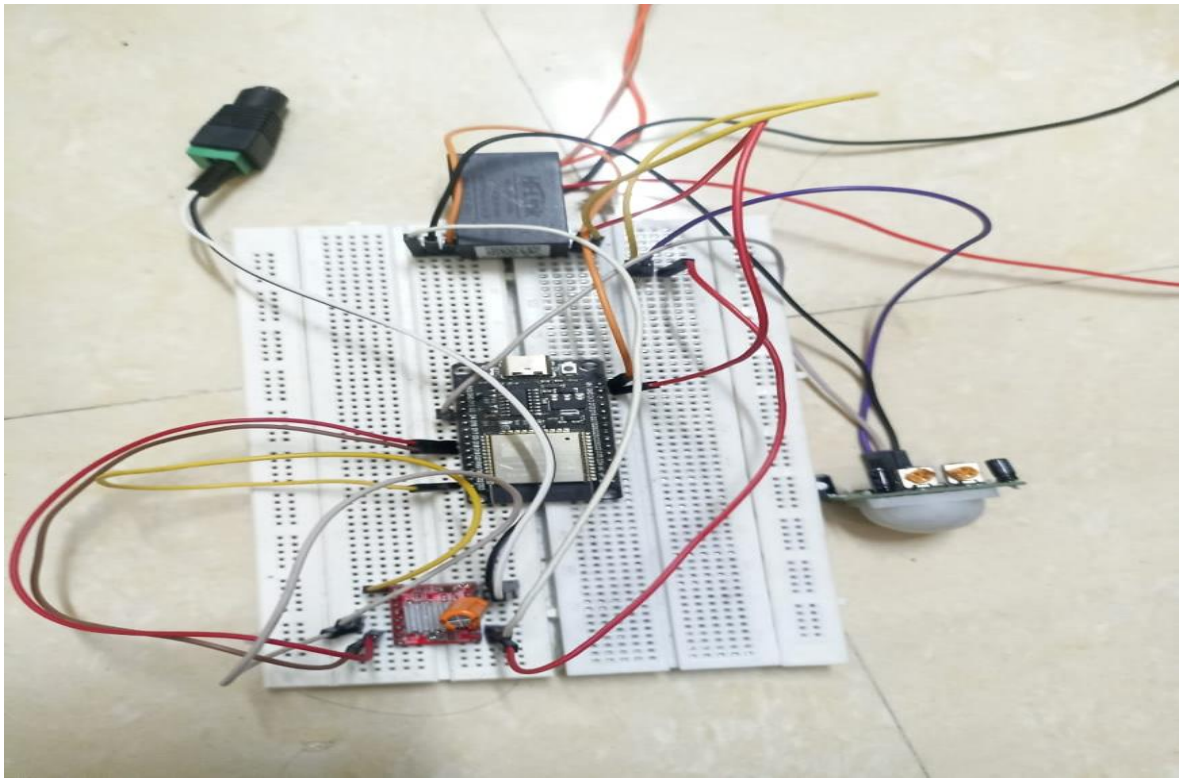
STEP - Step signal input; each pulse moves the motor one step.

DIR -Direction signal input; sets the direction of rotation

MS1,MS2 - Micro stepping mode selection.

ENABLE - Enables or disables the motor driver (active LOW).

HARDWARE SETUP:



ARDUINO IDE CODE :

In the *Smart Water Management System*, the control logic is implemented using Arduino C/C++ programming. The Arduino code manages the following system functionalities:

1. *Motion Detection*: The PIR sensor detects motion and sends a signal to the ESP32 microcontroller to initiate tap control.
2. *Real-Time Clock (RTC) Management*: The built-in RTC of the ESP32, along with the *time.h library*, ensures time-based scheduling for tap operation.
3. *Tap Control*: The A4988 motor driver is controlled via the ESP32 to adjust the stepper motor for precise tap operation.

***Code**:*

```
#include <WiFi.h>
#include <time.h>

// Wi-Fi credentials
const char* ssid  = "Batman";       // Wi-Fi SSID
const char* password = "12345678"; // Wi-Fi password

// Define A4988 driver pins
#define STEP_PIN 18
#define DIR_PIN 19
#define ENABLE_PIN 23

// Define PIR sensor pin
#define PIR_PIN 4 // Connect PIR sensor output to GPIO4

// Define built-in LED pin (GPIO2 for ESP32)
#define LED_PIN 2

// Motor parameters
const int stepsPerRevolution = 200;
const int microstepping = 1;
const int motorSpeed = 700;

// PIR sensor timing
const unsigned long motionTimeout = 10000; // 10 seconds timeout if no motion is detected
```

```
unsigned long lastMotionTime = 0;         // Stores the last time motion was detected
bool motorInOriginalPosition = true;      // Tracks if the motor is in its original position
bool motionHandled = false;               // Tracks if the motion event has been handled

// Define minute intervals for motor activation
struct MinuteInterval {
  int startMinute; // Start minute of the interval
  int endMinute;   // End minute of the interval
};

// Motor conditions a,b,c,d,e,f are the time intervals which we need to calibrate for our requirements here I create 3 intervals for my purpose
MinuteInterval condition1 = {a,b};
MinuteInterval condition2 = {c, d};
MinuteInterval condition3 = {e, f};

void setup() {
  // Set up pins
  pinMode(STEP_PIN, OUTPUT);
  pinMode(DIR_PIN, OUTPUT);
  pinMode(ENABLE_PIN, OUTPUT);
  pinMode(PIR_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT); // Built-in LED

  // Turn off LED initially
  digitalWrite(LED_PIN, LOW);
```

```
  // Enable motor driver
  digitalWrite(ENABLE_PIN, LOW);

  // Initialize Serial
  Serial.begin(115200);

  // Connect to Wi-Fi
  connectToWiFi();

  // Set RTC to IST (UTC+5:30)
  configTime(19800, 0, "pool.ntp.org", "time.nist.gov"); // 19800 seconds = 5 hours 30 minutes
  Serial.println("Waiting for time synchronization...");
  delay(2000);
  printLocalTime();
}

void loop() {
  // Get current time
  struct tm timeinfo;
  if (!getLocalTime(&timeinfo)) {
    Serial.println("Failed to obtain time");
    return;
  }

  int currentMinute = timeinfo.tm_min;
```

```
// Check if the time is within any active interval
bool isActive = isWithinMinuteInterval(currentMinute, condition1) ||
                isWithinMinuteInterval(currentMinute, condition2) ||
                isWithinMinuteInterval(currentMinute, condition3);

int rotationSteps = 0;

if (isWithinMinuteInterval(currentMinute, condition1)) {
  rotationSteps = stepsPerRevolution / 4; // 90° rotation
} else if (isWithinMinuteInterval(currentMinute, condition2)) {
  rotationSteps = stepsPerRevolution / 2; // 180° rotation
} else if (isWithinMinuteInterval(currentMinute, condition3)) {
  rotationSteps = stepsPerRevolution;    // 360° rotation
}

if (isActive) {
  // Check PIR sensor for motion
  bool motionDetected = digitalRead(PIR_PIN);

  if (motionDetected) {
    lastMotionTime = millis(); // Reset timeout timer

    // Turn on the built-in LED
    digitalWrite(LED_PIN, HIGH);

    if (motorInOriginalPosition && !motionHandled) {
      // Handle motion only once per interval
      Serial.println("Motion detected! Rotating motor.");
```

```
      rotateMotor(rotationSteps, HIGH); // Rotate as per the active interval
      motionHandled = true;           // Mark the motion as handled
      motorInOriginalPosition = false;
    }
  } else if (!motorInOriginalPosition && millis() - lastMotionTime >= motionTimeout) {
    Serial.println("No motion detected for 10 seconds. Returning to original position.");
    rotateMotor(rotationSteps, LOW); // Rotate back to original position
    motorInOriginalPosition = true;  // Motor returned to original position
    motionHandled = false;          // Reset motion flag for the next interval

    // Turn off the built-in LED
    digitalWrite(LED_PIN, LOW);
  }
 } else if (!motorInOriginalPosition) {
  // If outside active interval, ensure motor is in the original position
  Serial.println("Outside active interval. Returning motor to original position.");
  rotateMotor(rotationSteps, LOW);
  motorInOriginalPosition = true;
  motionHandled = false;

  // Turn off the built-in LED
  digitalWrite(LED_PIN, LOW);
 }

 delay(1000);
}
```

```
// Function to check if current minute is within a given interval
bool isWithinMinuteInterval(int currentMinute, MinuteInterval interval) {
  return (currentMinute >= interval.startMinute && currentMinute < interval.endMinute);
}

// Function to rotate the motor
void rotateMotor(int steps, bool direction) {
  if (direction == HIGH) {
    Serial.println("Motor rotating clockwise.");
  } else {
    Serial.println("Motor rotating counterclockwise.");
  }

  digitalWrite(DIR_PIN, direction); // Set motor direction

  for (int i = 0; i < steps * microstepping; i++) {
    digitalWrite(STEP_PIN, HIGH);
    delayMicroseconds(1000000 / motorSpeed / 2);
    digitalWrite(STEP_PIN, LOW);
    delayMicroseconds(1000000 / motorSpeed / 2);
  }
}

// Connect to Wi-Fi
void connectToWiFi() {
  WiFi.begin(ssid, password);
```

```
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nWi-Fi connected");
}

// Print the current local time
void printLocalTime() {
  struct tm timeinfo;
  if (!getLocalTime(&timeinfo)) {
    Serial.println("Failed to obtain time");
    return;
  }
  Serial.println(&timeinfo, "%A, %B %d %Y %H:%M:%S");
}
```

Code Summary:

1. The ESP32 connects to a Wi-Fi network using provided credentials for NTP synchronization.
2. A PIR sensor detects motion and sends a signal to the ESP32.
3. Built-in LED on GPIO2 lights up when motion is detected.
4. The A4988 motor driver controls a stepper motor using STEP, DIR, and ENABLE pins.
5. The motor rotates 90°, 180°, or 360° based on predefined time intervals.
6. The ESP32 synchronizes time with an NTP server to maintain accurate real-time control.
7. The MinuteInterval structure defines specific time intervals for motor activation.

8. Motion triggers motor rotation, and after a timeout of 10 seconds, it returns to its original position.
9. The motor rotation direction is controlled by the DIR pin.
10. Step pulses for motor control are generated on the STEP pin with specified speed.
11. The system checks the current minute to decide if the motor should rotate.
12. If no motion is detected, the system ensures the motor returns to its default state.
13. The connectToWiFi() function establishes a network connection for time synchronization.
14. Serial logs are used for debugging, displaying events like motion detection and motor rotation.
15. The program continuously monitors motion and time to control motor and LED behavior.Continue this loop infinitely.

INFERENCES:

The provided code demonstrates an automated system using an ESP32 microcontroller to control a stepper motor based on motion detection and real-time intervals. A PIR sensor triggers motor operation upon detecting motion, while an LED provides a visual indication of activity. The ESP32 connects to Wi-Fi and synchronizes with an NTP server to maintain accurate timing, allowing the motor to rotate in predefined intervals (90°, 180°, or 360°) and return to its original position after a 10-second timeout if no motion is detected. The A4988 motor driver ensures precise stepper motor control, while the system logic ensures efficient operation by combining motion sensing, real-time clock functionality, and interval-based activation. Serial monitoring provides debugging support, making the code robust for time and motion-dependent automation tasks.

CONCLUSION:

The *Smart Water Management System* leverages PIR motion detection and RTC scheduling to provide an intelligent, automated solution for water flow control. By combining motion sensing with precise time management, the system ensures efficient water usage, offering significant benefits in domestic, public, and industrial applications.