**CSE Department – Faculty of Engineering - MSA Fall 2024**

**CSE4531 IOT**

**Course Project**

**Course Instructor: Dr. Ehab Awad**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name** | **Youssef Amr** | **Student ID** | **220863** |
| **Student Name** | **Youssef Abdelfattah** | **Student ID** | **222843** |
| **TA**  **Name** | **Eng. Hussein Mostafa** | **Grade: /** | |

Smart Home

**Table of Contents**

[Project Overview 3](#_gjdgxs)

[**Objectives** 3](#_tyjcwt)

[**Roles and Responsibilities** 4](#_30j0zll)

[**Algorithm and external libraries** 5](#_1fob9te)

[Code explaining 7](#_3dy6vkm)

[Output and results 11](#_3znysh7)

[References 13](#_2et92p0)

**Project Overview**

This project focuses on developing a \*Smart Home Automation System\* that integrates multiple sensors and actuators to enhance home safety, security, and convenience. The system incorporates real-time monitoring and control mechanisms, allowing it to detect critical events such as temperature anomalies, motion detection, and smoke presence. Upon detecting such events, the system activates appropriate responses, such as turning on alarms, lights, or ventilation systems. The system architecture employs a centralized control hub and an event manager to ensure efficient coordination between sensors and actuators. The modular design facilitates scalability and adaptability, enabling future enhancements like IoT integration for remote monitoring and control.

**Objectives**

1. Real-Time Monitoring: Continuously monitor environmental parameters such as temperature, motion, and smoke levels using advanced sensors.

2. Threshold-Based Responses: Trigger specific actuators, including alarms, lights, and motors, based on threshold violations detected by the sensors.

3. Centralized Coordination: Utilize a central hub to manage communication between sensors and actuators, ensuring seamless and efficient operation.

4. Critical Event Detection: Employ an event manager to analyze data from multiple sensors collaboratively, detecting critical events with high accuracy.

5. Scalable and Adaptable Design: Build a modular system that can be expanded or integrated with IoT platforms for remote access and advanced automation.

6. User Notifications: Alert users through alarms, LEDs, and other actuators in response to detected safety concerns or environmental changes.

This smart home automation system is designed to prioritize safety and convenience while demonstrating a robust and scalable approach to modern home management solutions.

## Roles and Responsibilities

* + **Code: Youssef Abdelfattah & Youssef Amr**
  + **Hardware setup: Youssef Amr**
  + **Report: Youssef Amr**

Algorithm and External Libraries

External Libraries

1. SPI.h:

- Facilitates communication between the microcontroller and other peripheral devices, such as sensors and actuators, using the SPI protocol.

2. Ethernet.h:

- Manages Ethernet connectivity for enabling IoT integration, allowing real-time data transmission and remote control of the smart home system.

Algorithm

1. System Initialization:

- Configure sensors for temperature, motion, and smoke detection and assign appropriate analog/digital pins.

- Initialize actuators such as alarms, lights, and ventilation systems.

- Set up the central hub to manage communication between sensors, actuators, and the event manager.

2. Data Acquisition:

- Continuously poll sensors for data:

- Temperature Sensor: Read analog values and convert them to degrees Celsius.

- Motion Sensor: Detect motion using digital signals.

- Smoke Sensor: Read analog smoke levels.

3. Event Detection and Analysis:

- Process sensor data using the central hub:

- Compare readings to predefined safety thresholds.

- Determine critical events (e.g., high temperature, smoke presence, or motion detection).

4. Trigger Responses:

- If a critical event is detected, activate relevant actuators:

- Sound an alarm if smoke is detected.

- Turn on lights for motion detection during the night.

- Start ventilation for elevated temperatures.

5. Data Logging and Visualization (Optional IoT Integration):

- Send sensor data to an IoT platform like ThingSpeak using the Ethernet library.

- Log events and actuator statuses for trend analysis and monitoring.

6. Retry Mechanism:

- Implement a retry mechanism for failed network connections or data uploads.

- Automatically restart network services after multiple failures.

7. User Alerts:

- Use LEDs, buzzers, or notification systems to inform users of critical events.

**Sensors**

1. Temperature Sensor

- Reads ambient temperature for detecting overheating or fire hazards.

2. Motion Sensor

- Detects unauthorized entry or movement in the house.

3. Smoke Sensor

- Monitors air quality to identify potential fire incidents.

**Actuators**

1. Alarm System

- Alerts occupants to potential threats like fire or unauthorized entry.

2. Lights

- Enhances safety and convenience by lighting up dark areas when motion is detected.

3. Ventilation System

- Activates to reduce temperature or smoke concentration in critical areas.

This algorithm ensures seamless integration of sensors, actuators, and a centralized hub to create a reliable and efficient smart home system.

# Code explaining

# #include "Sensor.h"

# #include <Arduino.h>

# #include <DHT.h>

# // Sensor Base Class Implementation

# Sensor::Sensor(int id, const String& type, float threshold, float weight)

# : id(id), type(type), threshold(threshold), weight(weight), value(0), observerCount(0) {}

# Sensor::~Sensor() {}

# void Sensor::notifyObservers() {

# for (int i = 0; i < observerCount; i++) {

# if (observers[i] != nullptr) {

# observers[i]->readData();

# }

# }

# }

# void Sensor::attach(Sensor\* observer) {

# if (observerCount < MAX\_OBSERVERS) {

# observers[observerCount++] = observer;

# }

# }

# void Sensor::detach(Sensor\* observer) {

# for (int i = 0; i < observerCount; i++) {

# if (observers[i] == observer) {

# for (int j = i; j < observerCount - 1; j++) {

# observers[j] = observers[j + 1];

# }

# observerCount--;

# break;

# }

# }

# }

# float Sensor::getValue() const { return value; }

# float Sensor::getThreshold() const { return threshold; }

# int Sensor::getId() const { return id; }

# float Sensor::getWeight() const { return weight; }

# // TemperatureSensor Implementation

# TemperatureSensor::TemperatureSensor(int id, int pin, float threshold, float weight)

# : Sensor(id, "Temperature", threshold, weight), dht(pin, DHT22) {}

# void TemperatureSensor::readData() {

# dht.begin();

# float temp = dht.readTemperature();

# if (!isnan(temp)) {

# value = temp;

# if (value > threshold) {

# notifyObservers();

# }

# } else {

# Serial.println("TemperatureSensor: Failed to read temperature!");

# }

# }

# // MotionSensor Implementation

# MotionSensor::MotionSensor(int id, int pin, float threshold, float weight)

# : Sensor(id, "Motion", threshold, weight), pin(pin) {}

# void MotionSensor::setup() {

# pinMode(pin, INPUT);

# }

# void MotionSensor::readData() {

# int motionDetected = digitalRead(pin);

# value = motionDetected; // 1 for motion, 0 for no motion

# if (value > threshold) {

# notifyObservers();

# }

# }

# // SmokeSensor Implementation

# SmokeSensor::SmokeSensor(int id, int analogPin, float threshold, float weight)

# : Sensor(id, "Smoke", threshold, weight), analogPin(analogPin), smokeLevel(0) {}

# void SmokeSensor::setup() {

# pinMode(analogPin, INPUT);

# }

# void SmokeSensor::readData() {

# int rawValue = analogRead(analogPin);

# smokeLevel = convertToPPM(rawValue);

# value = smokeLevel;

# if (smokeLevel > threshold) {

# Serial.print("SmokeSensor: Smoke detected! Level: ");

# Serial.println(smokeLevel);

# notifyObservers();

# } else {

# Serial.print("SmokeSensor: Smoke level normal: ");

# Serial.println(smokeLevel);

# }

# }

# bool SmokeSensor::isSmokeDetected() const {

# return smokeLevel > threshold;

# }

# float SmokeSensor::convertToPPM(int rawValue) {

# float voltage = (rawValue / 1023.0) \* 5.0;

# float ppm = (voltage / 5.0) \* 1000;

# return ppm;

# }

Explaination:

### *Abstract Class Sensor and Its Role*

The Sensor abstract class serves as a foundational blueprint for various sensor types in a smart home system. It defines the common attributes and behaviors shared by all sensors, including sensor ID, type, value, threshold, and observer management, enabling event-driven architecture. Specific sensors, such as temperature, motion, or smoke sensors, extend this base class to implement specialized functionality while maintaining a unified interface through inheritance and polymorphism.

#### *Features of the Abstract Class*

1. **Attributes**:  
   * **id**: Uniquely identifies the sensor.
   * **type**: Describes the type of sensor, such as temperature, motion, or smoke.
   * **value**: Stores the current reading of the sensor.
   * **threshold**: Defines the boundary condition that triggers actions, such as alarms.
   * **weight**: Represents the sensor's priority or significance in decision-making processes.
   * **observers**: Maintains a list of other sensors observing this sensor to enable event propagation.
   * **observerCount**: Tracks the number of attached observers.
2. **Core Methods**:  
   * **readData()**: A pure virtual function that must be implemented by derived classes to read sensor-specific data.
   * **notifyObservers()**: Propagates changes to all attached observers.
   * **attach() / detach()**: Manages observers, allowing sensors to subscribe or unsubscribe for updates.
3. **Encapsulation**:  
   * Provides getter methods, such as getValue() and getThreshold(), for secure access to private attributes.

#### *Specialized Sensor Classes*

1. **Temperature Sensor**:  
   * **Role**: Monitors ambient temperature to detect overheating or fire hazards and optimizes HVAC systems.
   * **Implementation**: Utilizes the DHT library to interface with DHT temperature sensors. Implements readData() to fetch and update the temperature value.
   * **Usage**: Automatically adjusts heating or cooling systems and triggers alarms for high-temperature conditions.
2. **Motion Sensor**:  
   * **Role**: Detects movement for purposes such as security and automation. It can turn on lights or activate alarms.
   * **Implementation**: Monitors a digital pin to detect motion using PIR or similar sensors. Implements readData() to update the motion state.
   * **Usage**: Enhances security by identifying unauthorized access and enables energy-efficient lighting systems.
3. **Smoke Sensor**:  
   * **Role**: Identifies smoke or fire hazards and initiates safety measures, such as alarms or sprinklers.
   * **Implementation**: Reads analog data from a smoke sensor and converts raw readings into PPM (Parts Per Million) for better interpretation. Includes an additional method, isSmokeDetected(), for logical checks.
   * **Usage**: Prevents fire damage and ensures occupant safety.

#### *Additional Sensor Types for Smart Homes*

* **Gas Leak Sensor**: Detects hazardous gases like methane or carbon monoxide and triggers alarms or shuts off supply valves to prevent accidents.
* **Water Leak Sensor**: Identifies water leaks in pipes or appliances, activating alerts and water shutoff valves to avoid flooding.
* **Light Sensor**: Automates lighting by adjusting brightness based on ambient conditions to conserve energy.
* **Humidity Sensor**: Works alongside temperature sensors to optimize HVAC systems and maintain comfort by regulating humidity levels.
* **Energy Usage Sensor**: Monitors energy consumption, identifies inefficient devices, and suggests optimization strategies.
* **Door/Window Sensor**: Detects unauthorized entry or ensures doors and windows are closed, enhancing security and energy efficiency.
* **Vibration Sensor**: Monitors structural health, such as during earthquakes, and alerts users about potential impacts or failures.

#### *Advantages of this Architecture*

1. **Reusability**:  
   * The abstract Sensor class ensures code reuse by defining a consistent structure for all sensors.
2. **Extensibility**:  
   * New sensor types can be easily added by inheriting from the Sensor class and implementing their specific behavior.
3. **Scalability**:  
   * The observer pattern enables efficient communication between sensors, supporting larger system deployments.
4. **Modularity**:  
   * Each sensor type operates independently, simplifying maintenance and debugging.

By leveraging the Sensor class and its specialized subclasses, the smart home system achieves a robust and versatile framework, capable of integrating various sensors to create an efficient and responsive environment.

***#include "Actuators.h"***

***// Base Actuator Class***

***Actuator::Actuator(int id, String type) : id(id), type(type), status(false) {}***

***Actuator::~Actuator() {}***

***int Actuator::getId() const {***

***return id;***

***}***

***bool Actuator::getStatus() const {***

***return status;***

***}***

***void Actuator::turnOn() {***

***status = true;***

***}***

***void Actuator::turnOff() {***

***status = false;***

***}***

***// Relay Module Implementation***

***RelayModule::RelayModule(int id, int pin) : Actuator(id, "Relay"), pin(pin) {}***

***void RelayModule::setup() {***

***pinMode(pin, OUTPUT);***

***digitalWrite(pin, LOW); // Default OFF***

***}***

***void RelayModule::executeAction(float data) {***

***if (data > 0) {***

***digitalWrite(pin, HIGH); // Turn ON***

***turnOn();***

***} else {***

***digitalWrite(pin, LOW); // Turn OFF***

***turnOff();***

***}***

***}***

***// Buzzer Implementation***

***Buzzer::Buzzer(int id, int pin) : Actuator(id, "Buzzer"), pin(pin) {}***

***void Buzzer::setup() {***

***pinMode(pin, OUTPUT);***

***digitalWrite(pin, LOW); // Default OFF***

***}***

***void Buzzer::executeAction(float data) {***

***if (data > 0) {***

***digitalWrite(pin, HIGH); // Activate buzzer***

***turnOn();***

***} else {***

***digitalWrite(pin, LOW); // Deactivate buzzer***

***turnOff();***

***}***

***}***

***// Motorized Valve Implementation***

***MotorizedValve::MotorizedValve(int id, int relayPin)***

***: Actuator(id, "MotorizedValve"), relayPin(relayPin) {}***

***void MotorizedValve::setup() {***

***pinMode(relayPin, OUTPUT);***

***digitalWrite(relayPin, LOW); // Default OFF***

***}***

***void MotorizedValve::executeAction(float data) {***

***if (data > 0) {***

***digitalWrite(relayPin, HIGH); // Open valve***

***turnOn();***

***} else {***

***digitalWrite(relayPin, LOW); // Close valve***

***turnOff();***

***}***

***}***

***// Smart Light Implementation***

***SmartLight::SmartLight(int id, int pwmPin)***

***: Actuator(id, "SmartLight"), pwmPin(pwmPin) {}***

***void SmartLight::setup() {***

***pinMode(pwmPin, OUTPUT);***

***analogWrite(pwmPin, 0); // Default OFF***

***}***

***void SmartLight::executeAction(float data) {***

***if (data >= 0 && data <= 255) {***

***analogWrite(pwmPin, static\_cast<int>(data)); // Set brightness***

***if (data > 0) {***

***turnOn();***

***} else {***

***turnOff();***

***}***

***}***

***}***

***// Servo Motor Implementation***

***ServoMotor::ServoMotor(int id, int pwmPin)***

***: Actuator(id, "ServoMotor"), pwmPin(pwmPin), currentAngle(0) {}***

***void ServoMotor::setup() {***

***servo.attach(pwmPin);***

***servo.write(0); // Default to 0 degrees***

***}***

***void ServoMotor::executeAction(float data) {***

***if (data >= 0 && data <= 180) { // Servo angle range***

***servo.write(static\_cast<int>(data));***

***currentAngle = static\_cast<int>(data);***

***turnOn(); // Assuming moving servo indicates activity***

***}***

***}***

***int ServoMotor::getCurrentAngle() const {***

***return currentAngle;***

***}***

### *Abstract Class Actuator and Its Role*

***The Actuator abstract class provides a standardized interface for all types of actuators in a smart home system. It defines core attributes and functionality common to all actuators, such as managing state (on/off) and executing specific actions. Specialized actuators inherit from this base class, enabling a unified approach to control various devices, such as lights, motors, and alarms.***

#### *Features of the Abstract Class*

1. ***Attributes:***
   * ***id: Uniquely identifies each actuator.***
   * ***type: Describes the type of actuator (e.g., relay, buzzer, servo motor).***
   * ***status: Indicates the current state of the actuator (on or off).***
2. ***Core Methods:***
   * ***Pure Virtual Function executeAction(float data): Must be implemented by derived classes to perform specific actions.***
   * ***Control Methods:***
     + ***turnOn() and turnOff(): Toggle the actuator's state.***
   * ***Getters:***
     + ***getId() and getStatus(): Provide safe access to private attributes.***
3. ***Extensibility:***
   * ***The abstract nature of the class allows developers to add new actuator types with minimal changes to the overall system architecture.***

#### *Specialized Actuator Classes*

1. ***Relay Module:***
   * ***Role: Acts as a general-purpose actuator to control devices such as fans, lights, and water pumps.***
   * ***Implementation:***
     + ***Toggles an electrical circuit by controlling a relay through a digital pin.***
     + ***Executes actions such as turning devices on or off based on input data.***
   * ***Usage: Essential for managing power to high-voltage appliances in smart homes.***
2. ***Buzzer (Alarm):***
   * ***Role: Provides audible alerts for critical events, such as smoke detection or security breaches.***
   * ***Implementation:***
     + ***Activates a buzzer when triggered by external inputs.***
   * ***Usage: Alerts occupants of emergencies to ensure a swift response.***
3. ***Motorized Valve:***
   * ***Role: Controls the flow of water or gas, typically used in safety systems for leak prevention.***
   * ***Implementation:***
     + ***Operates via a relay to open or close the valve.***
   * ***Usage: Automatically shuts off water or gas during leaks to prevent damage or hazardous situations.***
4. ***Smart Light:***
   * ***Role: Manages lighting systems, including dimming and scheduling, for energy efficiency and ambiance.***
   * ***Implementation:***
     + ***Uses PWM (Pulse Width Modulation) to adjust brightness levels.***
   * ***Usage: Automatically adjusts lighting based on user preferences or sensor data.***
5. ***Servo Motor:***
   * ***Role: Automates physical movement, such as opening windows or adjusting blinds.***
   * ***Implementation:***
     + ***Uses a servo motor to precisely control angles for specific movements.***
   * ***Usage: Enhances convenience by automating blinds, locks, or other moving components.***

#### *Additional Actuators for Smart Homes*

* ***Sprinkler System: Activates water sprinklers for fire suppression, ensuring rapid response to detected fires.***
* ***Smart Lock: Provides remote or automated door locking/unlocking, enhancing security and convenience.***
* ***CCTV Camera: Offers real-time video monitoring for security and surveillance.***
* ***HVAC Controller: Manages heating, ventilation, and air conditioning systems for optimal comfort and energy efficiency.***
* ***Exhaust Fan: Removes smoke or hazardous gases, improving air quality during critical situations.***
* ***Vibration Alert Device: Activates alerts for structural warnings, such as during earthquakes.***
* ***Humidifier/Dehumidifier: Adjusts indoor air humidity to maintain comfort and health.***
* ***Energy Optimizer: Dynamically manages energy flow to appliances, reducing costs and optimizing consumption.***
* ***Curtain Controller: Opens or closes curtains based on light sensors or user schedules.***
* ***Doorbell Camera: Captures video and audio at the door for visitor identification and security.***
* ***Emergency Siren: Emits loud alarms during security breaches or emergencies, alerting occupants and neighbors.***
* ***Appliance Switch: Remotely controls appliances, such as TVs or washing machines.***
* ***Charging Controller: Manages electric vehicle or battery charging to optimize energy usage.***
* ***Floor Heater: Adjusts heating levels for floors, enhancing comfort during colder seasons.***
* ***Backup Generator Start: Automatically starts backup power during outages, ensuring uninterrupted electricity.***

#### *Advantages of This Architecture*

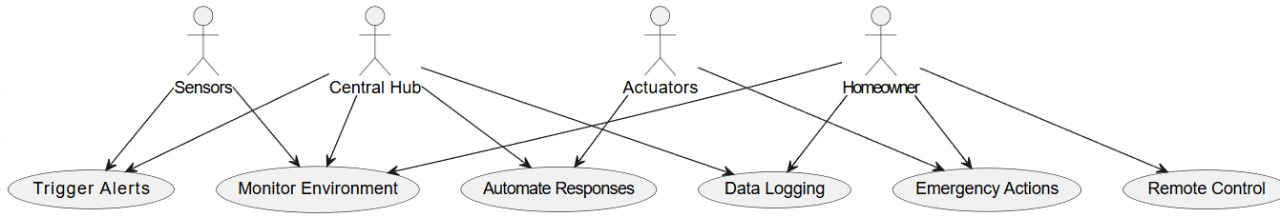
1. ***Flexibility:***
   * ***New actuators can be seamlessly added by extending the Actuator class.***
2. ***Modularity:***
   * ***Each actuator is self-contained, simplifying debugging and maintenance.***
3. ***Scalability:***
   * ***The system supports a wide variety of actuators, accommodating expanding smart home features.***
4. ***Interoperability:***
   * ***A unified control interface ensures smooth integration with other system components, such as sensors and the central hub.***

***By adopting this architecture, smart home systems can efficiently manage diverse actuators, improving automation, security, and user convenience.***

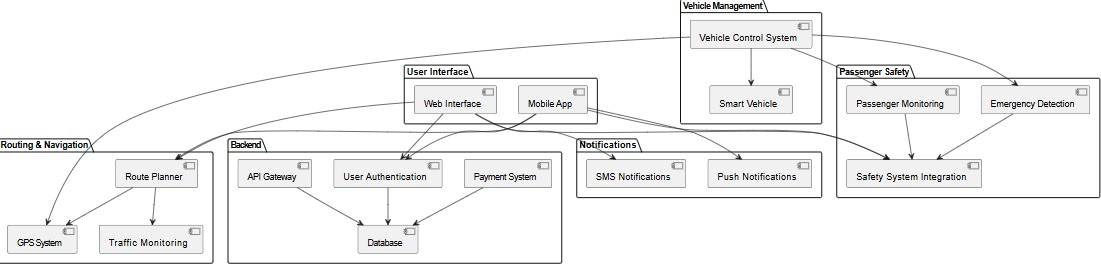
# Output and results

Diagrams:

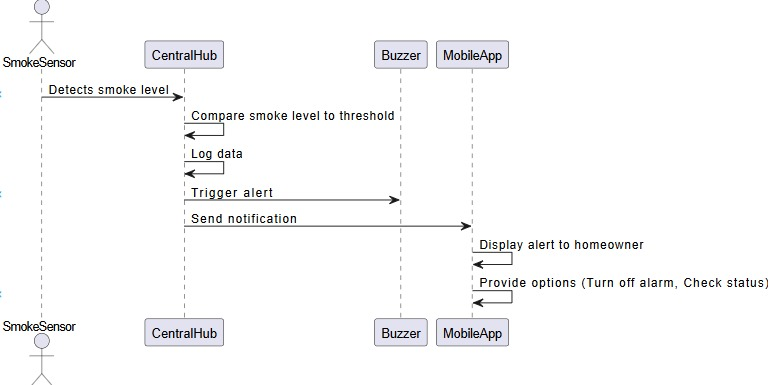
Use case:



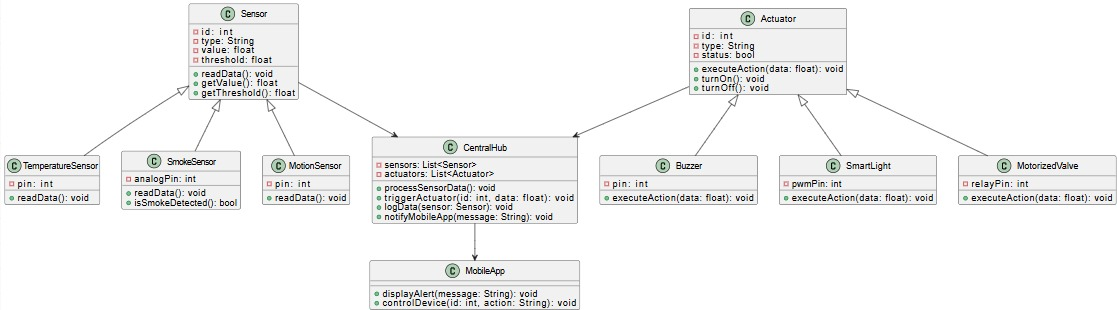
Blockdiagram:



Sequencediagram:



class diagram:



# References

**Books and Documentation:**

Arduino Documentation: https://www.arduino.cc/en/Reference/HomePage

"Programming Arduino: Getting Started with Sketches" by Simon Monk.

Sensor and Actuator Datasheets:

pH Sensor Datasheet (e.g., Gravity Analog pH Sensor).

DHT Sensor Documentation (DHT11/DHT22).

Smoke Sensor (MQ-2 or MQ-135) Datasheet.

Motion Sensor (PIR Sensor) Datasheet.

External Libraries:

SPI and Ethernet Library (for Arduino): https://www.arduino.cc/reference/en/libraries/ethernet/

Servo Library (for servo motors): https://www.arduino.cc/en/Reference/Servo