Smart Home

Report By – Shefali Bishnoi(2301CS87), Juhi Sahni(2301CS88), Saniya Prakash(2301CS49), Manvitha Reddy(2301CS29)

Introduction

This project integrates three critical smart home systems into a unified Arduino-based controller:

1. **Home Security and Fire Detection System**: Monitors for unauthorized entry and fire hazards, with controlled door locking mechanisms and alarm capabilities.
2. **Rainwater Harvesting System**: Automatically collects rainwater when rain is detected and manages tank capacity.
3. **Motion-Activated Smart Lighting**: Intelligently controls lighting based on motion detection.

The integration allows these systems to work together seamlessly while sharing hardware resources and providing centralized monitoring through a serial interface.

Motivation

The motivation for this project stems from several factors:

- **Resource Efficiency**: Combining multiple home management systems on a single microcontroller reduces hardware costs and power consumption.
- **Safety and Security**: Creating an integrated approach to home safety that addresses multiple threats simultaneously (intrusion, fire) while providing emergency responses.
- **Sustainability**: Implementing rainwater harvesting to promote environmental conservation and reduce water consumption.
- **Energy Conservation**: Using motion detection for lighting control to minimize electricity usage.

Innovation/Uniqueness

The project stands out in several ways:

1. **Integrated Approach**: Unlike most DIY solutions that focus on a single aspect of home automation, this system combines security, safety and resource management in one circuit.
2. **Emergency Intelligence**: The system features cross-functional emergency protocols, such as automatically unlocking doors when a fire is detected.

3. **Debounced Sensing**: Implements sophisticated sensor debouncing techniques to prevent false alarms from ultrasonic and IR sensors.
4. **Password Protection**: Incorporates a code verification system with multiple authorized users.
5. **Graceful Motion**: Servo motors move gradually rather than abruptly, extending component life and providing visual feedback.
6. **Non-Blocking Design**: The entire system operates without blocking delays, allowing all systems to run concurrently.
7. **Adaptive Calibration**: The rainwater sensor uses adaptive thresholds to accommodate different environmental conditions.

Hardware Requirements

Security and Fire Detection System

- Arduino board (e.g., Arduino Uno or Mega)
- Ultrasonic distance sensor (HC-SR04)
- IR motion sensor
- Flame sensor
- Piezo buzzer
- Servo motor (for door lock)
- Red and green LEDs
- Resistors for LEDs (220Ω)

Rainwater Harvesting System

- Rain sensor (analog)
- Water level sensor
- Servo motor (for valve control)
- Resistors for pull-up (if needed)

Smart Lighting System

- Additional ultrasonic sensor (HC-SR04)
- Blue LED
- Power supply for lights

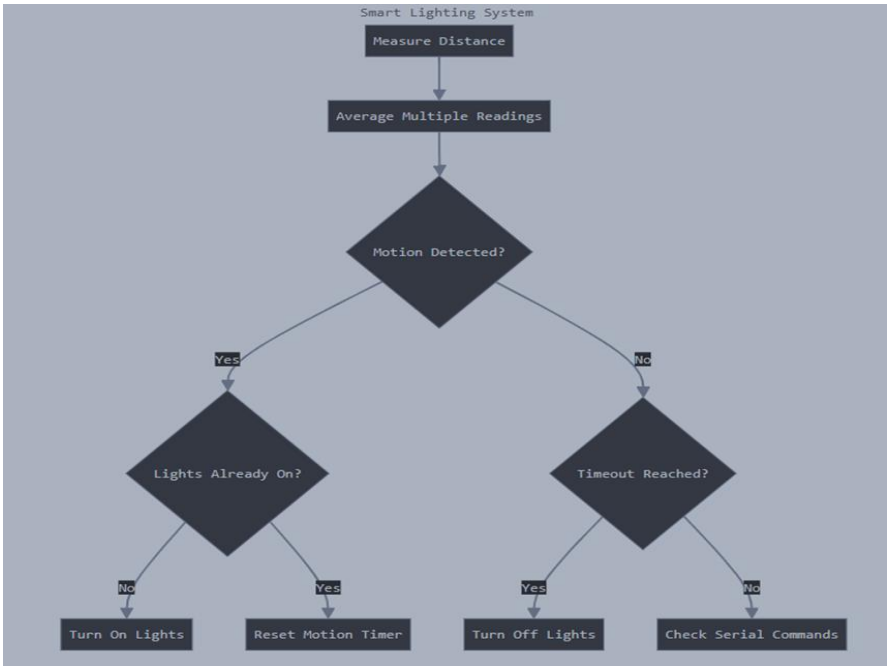
Miscellaneous

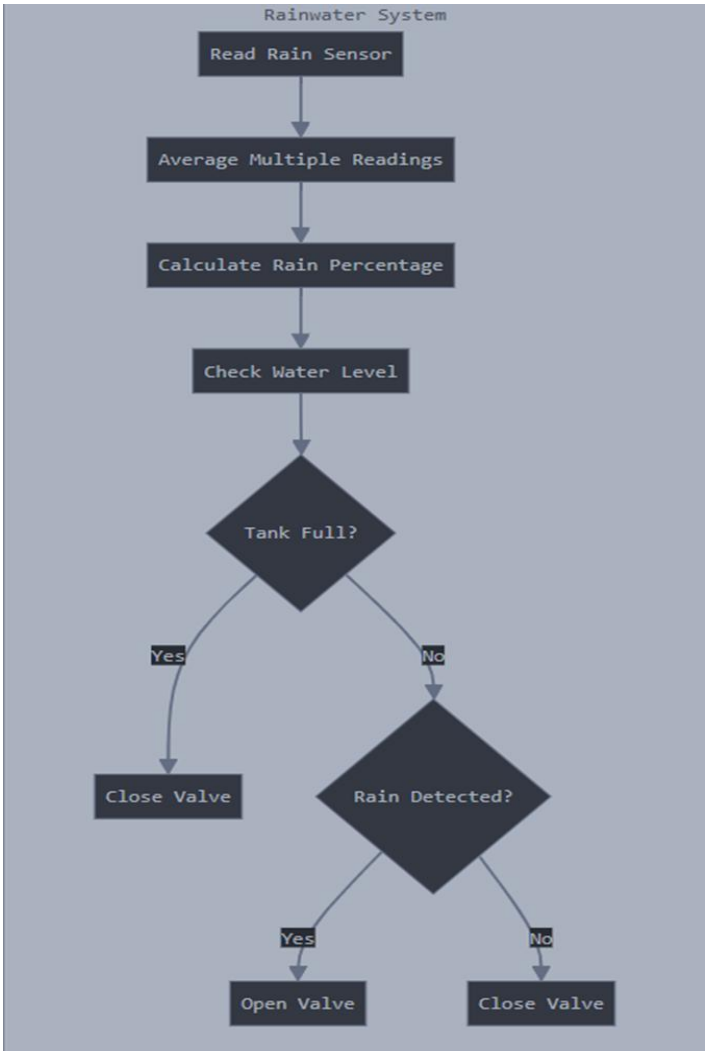
- Breadboard and jumper wires
- Power supply for Arduino
- Serial communication interface (USB or wireless)

Component Name	Quantity
Arduino Board (e.g., Uno)	**1**
Ultrasonic Sensor	**2**
IR Sensor	**1**
Buzzer	**1**
Servo Motor	**2**
LED (Red)	**1**
LED (Green)	**1**
LED (Blue)	**1**
Flame Sensor (Digital)	**1**
Rain Sensor (Analog)	**1**
Water Level Sensor	**1**
Breadboard	**1**

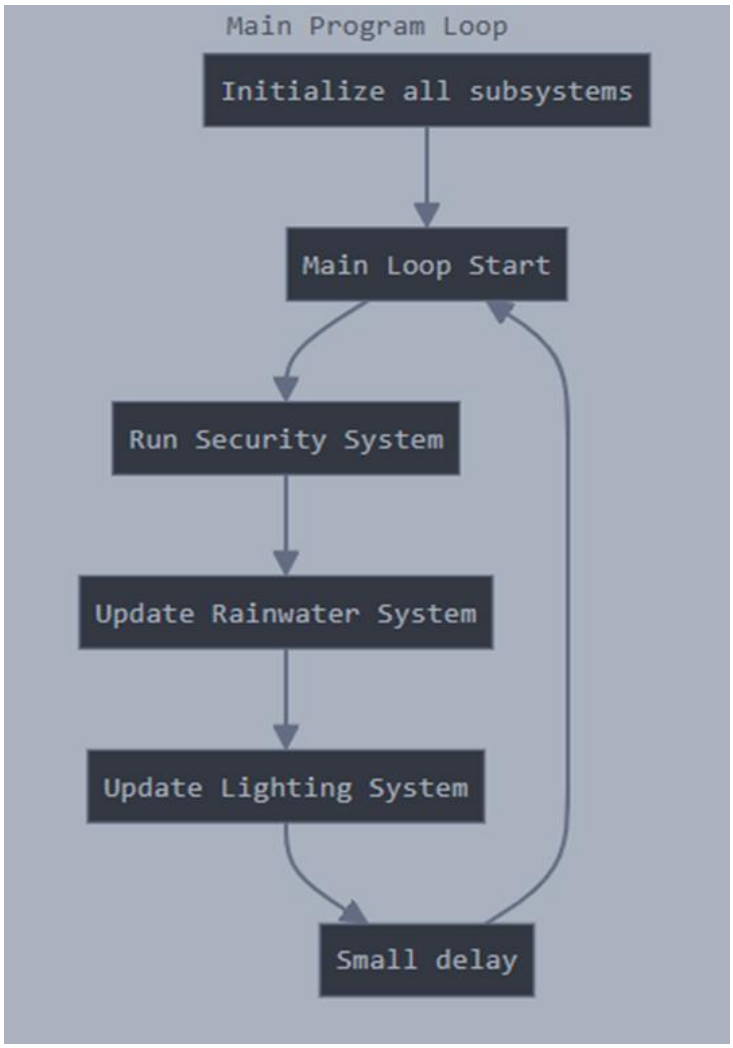
Flowcharts

Smart Lighting System

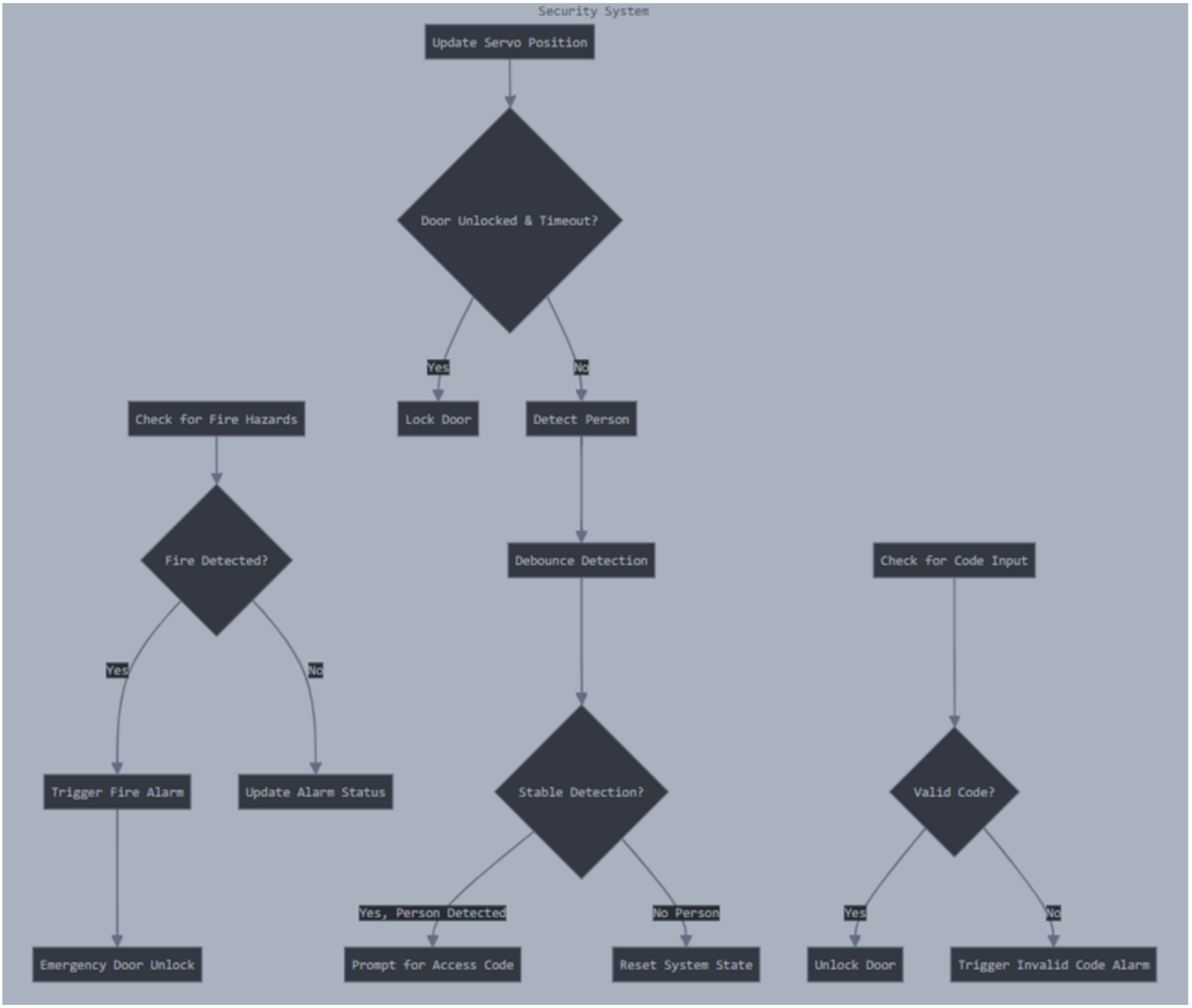




Rainwater Harvesting System

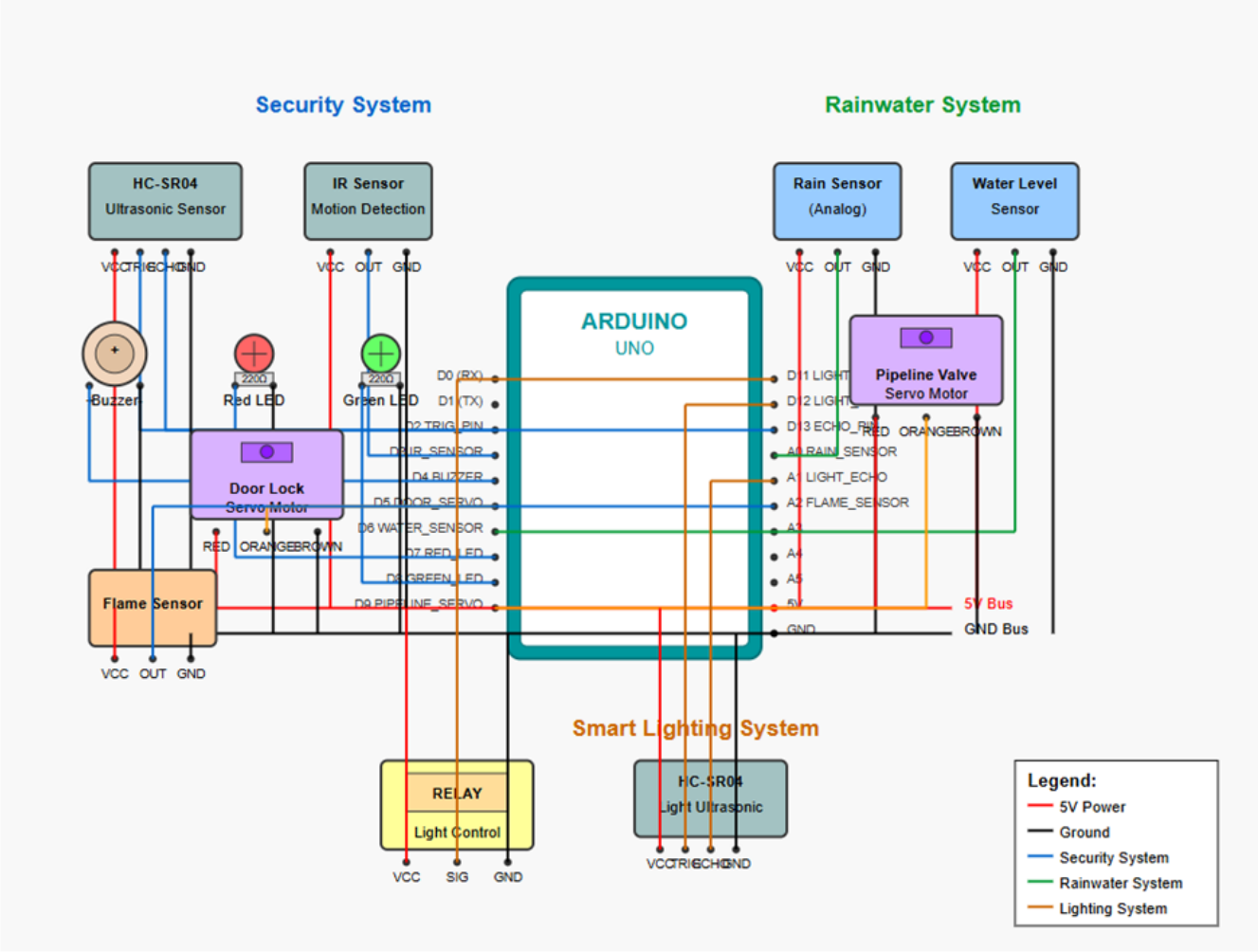


Main Program Loop



Security System

Circuit Diagram



Code Details

The code is organized into three main subsystems that share the Arduino's resources:

1. Security System: Handles intrusion detection, fire detection, and door locking mechanisms.
2. Rainwater System: Manages rain detection and water valve control.
3. Smart Lighting System: Controls lighting based on motion detection.

Each system has its own initialization function, update function, and dedicated set of pins. The systems operate independently but can influence each other, such as when the fire detection system triggers the emergency door unlock mechanism. Code has following functions:

1. Security System Functions

- initializeSecuritySystem(): Sets up pins, servo, and initial states for the security system
- detectIntrusion(): Uses ultrasonic and IR sensors to detect if someone is near the door
- promptForCode(): Requests an access code when a person is detected
- verifyAccessCode(): Checks if entered code matches authorized codes

- unlockDoor(): Opens the door when access is granted
- lockDoor(): Re-locks the door after timeout or on command
- updateServoPosition(): Gradually moves the door lock servo to target position
- resetAwaitingCode(): Cancels code request if no person is detected anymore
- triggerInvalidCodeAlarm(): Activates alarm when incorrect code is entered
- doubleBeep(): Produces notification sound for various security events
- runSecuritySystem(): Main function that coordinates security operations

2. Fire Detection System Functions

- detectFireHazard(): Checks flame sensor for fire detection
- triggerFireAlarm(): Activates alarm and emergency procedures when fire is detected
- updateFireAlarm(): Manages ongoing fire alarm state and responses
- resetFireAlarm(): Turns off alarm when fire is no longer detected
- emergencyUnlock(): Automatically unlocks door during fire emergency

3. Rainwater Harvesting System Functions

- initializeRainwaterSystem(): Sets up pins and servo for rainwater system
- updateRainwaterSystem(): Checks rain sensor and water tank level to control valve

4. Smart Lighting System Functions

- initializeLightingSystem(): Sets up pins for motion-activated lighting
- updateLighting(): Detects motion and controls lights based on presence

Each subsystem works independently but is integrated into the main program flow, with setup() initializing all systems and loop() running them continuously.

```
#include <Servo.h>

// Pin Definitions - Security System
const int ULTRASONIC_TRIG_PIN = 2;
const int ULTRASONIC_ECHO_PIN = 13;
const int IR_SENSOR_PIN = 3;
const int BUZZER_PIN = 4;
const int DOOR_LOCK_SERVO_PIN = 5;
const int LED_RED_PIN = 7;
const int LED_GREEN_PIN = 8;
const int TEMP_SENSOR_PIN = A2;  // Used for 3-pin NTC thermistor

//NTC Thermistor Parameters
const float THERMISTOR_NOMINAL = 12400;    // Nominal resistance at 25°C (12.4k
ohm)
const float TEMPERATURE_NOMINAL = 25;      // Temperature for nominal resistance
(25°C)
const float B_COEFFICIENT = 4590;          // Beta coefficient

// Distance threshold for ultrasonic sensors (in cm)
```

```
const int DISTANCE_THRESHOLD = 100;

// Pin Definitions - Rainwater System
#define RAIN_SENSOR A0
#define WATER_SENSOR 6
#define PIPELINE_SERVO_PIN 9

// Pin Definitions - Smart Lighting System
const int LIGHT_CONTROL_PIN = 11;
const int LIGHT_ULTRASONIC_TRIG_PIN = 12;
const int LIGHT_ULTRASONIC_ECHO_PIN = A1;

// Constants
const int RAIN_THRESHOLD = 400;
const int SERVO_OPEN = 90;
const int SERVO_CLOSED = 0;
const int RAIN_CHECK_INTERVAL = 2000;
const int LIGHT_CHECK_INTERVAL = 2000;
const unsigned long DOOR_OPEN_DURATION = 5000;
const unsigned long FIRE_CHECK_INTERVAL = 5000;
const unsigned long ALARM_TOGGLE_INTERVAL = 300;
const unsigned long SERVO_MOVE_INTERVAL = 20;
const unsigned long MOTION_CHECK_INTERVAL = 1000;
const unsigned long DEBOUNCE_DELAY = 500;     // Debounce delay for ultrasonic
detection

// Servo objects
Servo doorLockServo;
Servo pipelineValve;

// Timing and State Variables
unsigned long doorUnlockTime = 0;
unsigned long lastFireCheckTime = 0;
unsigned long lastAlarmToggleTime = 0;
unsigned long lastServoMoveTime = 0;
unsigned long lastRainCheckTime = 0;
unsigned long lastLightCheckTime = 0;
unsigned long lastMotionCheckTime = 0;
unsigned long lastDetectionChange = 0;      // For debouncing
bool isDoorUnlocked = false;
bool isFireAlarmActive = false;
bool isAwaitingCode = false;
bool isInvalidCodeAlarm = false;
bool alarmState = false;
int currentServoPosition = 0;
int targetServoPosition = 0;
bool lastIntrusionState = false;
bool stableIntrusionState = false;          // Debounced intrusion state
```

```
// Rain System Variables
String valveStatus = "CLOSED";

// Smart Lighting Variables
bool isLightOn = false;

// Valid access codes
const String validCodes[3] = {
  "1234",  // Family Member 1
  "5678",  // Family Member 2
  "9012"   // Family Member 3
};


// Constants
#define TEMP_SENSOR_PIN A0
#define THERMISTOR_NOMINAL 10000.0   // Resistance at 25°C
#define TEMPERATURE_NOMINAL 25.0     // 25°C
#define B_COEFFICIENT 3950.0         // Beta value of thermistor
#define NUM_TEMP_SAMPLES 10          // For averaging

// Security System Initialization Function
void initializeSecuritySystem() {
  pinMode(ULTRASONIC_TRIG_PIN, OUTPUT);
  pinMode(ULTRASONIC_ECHO_PIN, INPUT);
  pinMode(IR_SENSOR_PIN, INPUT);
  pinMode(BUZZER_PIN, OUTPUT);
  pinMode(LED_RED_PIN, OUTPUT);
  pinMode(LED_GREEN_PIN, OUTPUT);

  digitalWrite(LED_RED_PIN, LOW);
  digitalWrite(LED_GREEN_PIN, LOW);
  digitalWrite(BUZZER_PIN, LOW);

  doorLockServo.attach(DOOR_LOCK_SERVO_PIN);
  doorLockServo.write(0);  // Initial locked position
  currentServoPosition = 0;
  targetServoPosition = 0;

  Serial.println("Security and Fire Detection System Initialized");
}

// Read Temperature from 3-pin NTC Thermistor Module
float readThermistorTemperature() {
  long sumADC = 0;

  // Take multiple readings for noise reduction
  for (int i = 0; i < NUM_TEMP_SAMPLES; i++) {
    sumADC += analogRead(TEMP_SENSOR_PIN);
```

```
    delay(5); // Small delay between samples
  }
  int rawADC = sumADC / NUM_TEMP_SAMPLES;

  // Check for extreme values
  if (rawADC <= 5 || rawADC >= 1020) {
    Serial.println("WARNING: Temperature sensor reading out of range!");
    return 25.0; // Return fallback room temperature
  }

  // Convert ADC to voltage
  float voltage = rawADC * (5.0 / 1023.0);

  // Convert voltage to resistance using voltage divider formula
  float resistance = (10000.0 * voltage) / (5.0 - voltage); // Assuming 10k pull-
up

  // Steinhart-Hart Equation to calculate temperature
  float steinhart = resistance / THERMISTOR_NOMINAL;  // (R/R0)
  steinhart = log(steinhart);                         // ln(R/R0)
  steinhart /= B_COEFFICIENT;                         // 1/B * ln(R/R0)
  steinhart += 1.0 / (TEMPERATURE_NOMINAL + 273.15);  // + 1/T0
  steinhart = 1.0 / steinhart;                        // Invert
  steinhart -= 273.15;                                // Convert to Celsius

  // Sanity check on result
  if (steinhart < -20 || steinhart > 125) {
    Serial.println("WARNING: Temperature out of reasonable range!");
    return 25.0;
  }

  // Debug print every 5 seconds
  static unsigned long lastTempDebug = 0;
  if (millis() - lastTempDebug > 5000) {
    Serial.print("Temp Debug | ADC: ");
    Serial.print(rawADC);
    Serial.print(" | Voltage: ");
    Serial.print(voltage, 3);
    Serial.print(" V | Resistance: ");
    Serial.print(resistance, 1);
    Serial.print(" Ω | Temp: ");
    Serial.print(steinhart, 2);
    Serial.println(" °C");
    lastTempDebug = millis();
  }

  return steinhart;
}
```

```
// Function to produce a double beep notification
void doubleBeep() {
  digitalWrite(BUZZER_PIN, HIGH);
  delay(200);
  digitalWrite(BUZZER_PIN, LOW);
  delay(200);
  digitalWrite(BUZZER_PIN, HIGH);
  delay(200);
  digitalWrite(BUZZER_PIN, LOW);
}

bool detectIntrusion() {
  // Ultrasonic distance measurement with averaging
  long duration, distance = 0;

  // Take multiple readings and average them for stability
  const int NUM_READINGS = 3;
  for (int i = 0; i < NUM_READINGS; i++) {
    // Clear the trigger pin
    digitalWrite(ULTRASONIC_TRIG_PIN, LOW);
    delayMicroseconds(2);

    // Send 10µs pulse to trigger
    digitalWrite(ULTRASONIC_TRIG_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(ULTRASONIC_TRIG_PIN, LOW);

    // Measure the echo time and calculate distance
    duration = pulseIn(ULTRASONIC_ECHO_PIN, HIGH, 30000); // Timeout after 30ms

    // Check if we got a valid reading
    if (duration > 0) {
      distance += (duration / 2) / 29.1; // Convert to cm
    }

    // Small delay between readings
    delay(10);
  }

  // Calculate average distance
  distance = distance / NUM_READINGS;

  // Intrusion detected if object is closer than threshold and IR is triggered
  int irState = digitalRead(IR_SENSOR_PIN);

  // Print debug information periodically
  static unsigned long lastDebugPrint = 0;
  if (millis() - lastDebugPrint > 1000) {
    Serial.print("Distance: ");
```

```
    Serial.print(distance);
    Serial.print(" cm, IR State: ");
    Serial.println(irState == HIGH ? "ACTIVE" : "INACTIVE");
    lastDebugPrint = millis();
  }

  // Return intrusion state based on both sensors
  return (distance < DISTANCE_THRESHOLD && distance > 0 && irState == HIGH);
}

bool detectFireHazard() {
  // Read current temperature using the NTC thermistor
  float currentTemperature = readThermistorTemperature();

  // Print the current temperature (for debugging)
  static unsigned long lastTempPrint = 0;
  if (millis() - lastTempPrint > 2000) { // Print every 2 seconds
    Serial.print("Current Temperature: ");
    Serial.print(currentTemperature);
    Serial.println(" °C");
    lastTempPrint = millis();
  }

  // Implement averaging for more stable readings
  static float tempReadings[5] = {0, 0, 0, 0, 0};
  static int readIndex = 0;
  static float tempSum = 0;

  // Remove the old reading from the sum
  tempSum = tempSum - tempReadings[readIndex];
  // Add the new reading to the array and the sum
  tempReadings[readIndex] = currentTemperature;
  tempSum = tempSum + currentTemperature;
  // Move to the next position in the array
  readIndex = (readIndex + 1) % 5;

  // Calculate the average after we have some readings
  float avgTemperature = tempSum / 5;

  // Check for absolute temperature threshold with averaged value
  if (avgTemperature > MAX_TEMPERATURE) {
    Serial.println("FIRE HAZARD: Extremely High Temperature Detected!");
    return true;
  }

  // Check for rapid temperature increase with better noise rejection
  if (previousTemperature > 0) {
    float temperatureChange = avgTemperature - previousTemperature;
    if (temperatureChange > RAPID_TEMP_INCREASE) {
```

```
      Serial.println("FIRE HAZARD: Rapid Temperature Increase Detected!");
      Serial.print("Change: ");
      Serial.print(temperatureChange);
      Serial.println(" °C");
      return true;
    }
  }

  // Update previous temperature for next comparison
  previousTemperature = avgTemperature;

  return false;
}

void triggerFireAlarm() {
  isFireAlarmActive = true;
  Serial.println("FIRE EMERGENCY! EVACUATE IMMEDIATELY!");

  // Start emergency actions
  emergencyUnlock();
}

void updateFireAlarm() {
  // Non-blocking fire alarm update
  if (isFireAlarmActive) {
    // Check if temperature has returned to safe levels
    float currentTemp = readThermistorTemperature();

    // If temperature is back to normal, automatically reset the alarm after a
delay
    static unsigned long tempNormalTime = 0;
    if (currentTemp < MAX_TEMPERATURE - 5.0) { // 5 degrees below threshold
      if (tempNormalTime == 0) {
        tempNormalTime = millis();
      } else if (millis() - tempNormalTime > 10000) { // 10 seconds of normal
temp
        resetFireAlarm();
        tempNormalTime = 0;
        return;
      }
    } else {
      tempNormalTime = 0; // Reset the timer if temperature rises again
    }

    // Continue with alarm if not reset
    if (millis() - lastAlarmToggleTime >= ALARM_TOGGLE_INTERVAL) {
      alarmState = !alarmState;  // Toggle alarm state

      digitalWrite(BUZZER_PIN, alarmState);
```

```
      digitalWrite(LED_RED_PIN, alarmState);

      lastAlarmToggleTime = millis();

      // Print continuous fire alert periodically
      if (alarmState) {
        Serial.println("FIRE EMERGENCY! EVACUATE IMMEDIATELY!");
        Serial.print("Current temperature: ");
        Serial.print(currentTemp);
        Serial.println(" °C");
      }
    }
  }
}

// function to reset the fire alarm
void resetFireAlarm() {
  isFireAlarmActive = false;
  alarmState = false;
  digitalWrite(BUZZER_PIN, LOW);
  digitalWrite(LED_RED_PIN, LOW);
  Serial.println("Fire alarm reset.");
}

void emergencyUnlock() {
  // Automatically unlock door during fire emergency
  targetServoPosition = 90;  // Set unlock position as target
  Serial.println("Emergency Door Unlock Activated");
  isDoorUnlocked = true;
  doorUnlockTime = millis();
}

bool verifyAccessCode() {
  // Code Verification Process
  if (Serial.available() > 0) {
    String inputCode = Serial.readStringUntil('\n');
    inputCode.trim();  // Remove whitespace

    // For debugging
    Serial.print("Code entered: ");
    Serial.println(inputCode);

    // Check against valid codes
    for (int i = 0; i < 3; i++) {
      if (inputCode == validCodes[i]) {
        unlockDoor();
        isAwaitingCode = false;
        isInvalidCodeAlarm = false;
        digitalWrite(BUZZER_PIN, LOW);
```

```
        digitalWrite(LED_RED_PIN, LOW);
        return true;
      }
    }

    // Invalid code attempt
    Serial.println("Invalid Access Code!");
    if (!isInvalidCodeAlarm) {
      triggerInvalidCodeAlarm();
    }
  }
  return false;
}

void unlockDoor() {
  // Servo-based Door Unlocking Mechanism
  Serial.println("Access Granted! Door Unlocking...");

  // Visual Indication
  digitalWrite(LED_GREEN_PIN, HIGH);

  // Set target for servo to gradually move to unlock position
  targetServoPosition = 90;
  isDoorUnlocked = true;
  doorUnlockTime = millis();
}

void lockDoor() {
  // Re-lock Door gradually
  targetServoPosition = 0;  // Set target position to locked
  digitalWrite(LED_GREEN_PIN, LOW);
  isDoorUnlocked = false;

  Serial.println("Door Locking...");
}

void updateServoPosition() {
  // Gradually move servo toward target position
  if (currentServoPosition != targetServoPosition) {
    if (millis() - lastServoMoveTime >= SERVO_MOVE_INTERVAL) {
      // Move servo one degree at a time toward target
      if (currentServoPosition < targetServoPosition) {
        currentServoPosition++;
      } else {
        currentServoPosition--;
      }

      doorLockServo.write(currentServoPosition);
      lastServoMoveTime = millis();
```

```
      // Print when door is fully locked/unlocked
      if (currentServoPosition == 90) {
        Serial.println("Door Fully Unlocked");
      } else if (currentServoPosition == 0) {
        Serial.println("Door Fully Locked");
      }
    }
  }
}

void promptForCode() {
  isAwaitingCode = true;
  Serial.println("Person detected. Please enter access code:");

  // Play double beep notification
  doubleBeep();
}

void resetAwaitingCode() {
  // Reset the awaiting code state if no person detected
  if (isAwaitingCode && !stableIntrusionState && !isInvalidCodeAlarm &&
!isDoorUnlocked) {
    isAwaitingCode = false;
    Serial.println("No person detected, system reset");
  }
}

void triggerInvalidCodeAlarm() {
  // Turn on alarm for invalid code
  digitalWrite(BUZZER_PIN, HIGH);
  digitalWrite(LED_RED_PIN, HIGH);
  isInvalidCodeAlarm = true;

  Serial.println("INVALID CODE ALARM: Enter correct code to disable");
}

void runSecuritySystem() {
  // Check for Fire Hazards at Intervals
  if (millis() - lastFireCheckTime >= FIRE_CHECK_INTERVAL) {
    if (detectFireHazard()) {
      triggerFireAlarm();
    }
    lastFireCheckTime = millis();
  }

  // Update fire alarm (non-blocking)
  updateFireAlarm();
```

```
  // Update servo position (gradual movement)
  updateServoPosition();

  // Auto-lock door after timeout
  if (isDoorUnlocked && currentServoPosition == 90 && (millis() - doorUnlockTime
> DOOR_OPEN_DURATION)) {
    lockDoor();
  }

  // Main Security System Logic - Detect person and perform debouncing
  bool currentIntrusionState = detectIntrusion();

  // Debounce the intrusion detection to prevent oscillation
  if (currentIntrusionState != lastIntrusionState) {
    lastDetectionChange = millis();
  }

  // After the debounce period, consider the state stable
  if ((millis() - lastDetectionChange) > DEBOUNCE_DELAY) {
    if (currentIntrusionState != stableIntrusionState) {
      stableIntrusionState = currentIntrusionState;

      // Only prompt for code when a new stable intrusion is detected
      if (stableIntrusionState && !isAwaitingCode && !isDoorUnlocked) {
        promptForCode();
      }
    }
  }

  // Update last intrusion state for debounce comparison
  lastIntrusionState = currentIntrusionState;

  // Reset awaiting code if no person detected for a period
  if (millis() - lastMotionCheckTime >= MOTION_CHECK_INTERVAL) {
    resetAwaitingCode();
    lastMotionCheckTime = millis();
  }

  // Check for Authorized Access Code
  verifyAccessCode();
}

// Rainwater System Functions
void initializeRainwaterSystem() {
  // Set pin modes
  pinMode(RAIN_SENSOR, INPUT);
  pinMode(WATER_SENSOR, INPUT_PULLUP);  // Assuming active-high sensor

  // Attach and initialize servo
```

```
  pipelineValve.attach(PIPELINE_SERVO_PIN);
  pipelineValve.write(SERVO_CLOSED);    // Start with valve closed

  Serial.println("Rainwater Harvesting System Initialized");
  Serial.println("Rain (%) | Tank Full | Valve Status");
}

void updateRainwaterSystem() {
  // Read sensor values
  int rainValue = analogRead(RAIN_SENSOR);
  int waterLevel = digitalRead(WATER_SENSOR);

  // Convert to meaningful values
  int rainPercentage = map(rainValue, 0, 1023, 0, 100);  // Convert to percentage
  bool isTankFull = (waterLevel == HIGH);                // HIGH means full

  // Control logic
  if (rainValue < RAIN_THRESHOLD) {  // Rain detected
    if (!isTankFull) {
      // Tank not full
      pipelineValve.write(SERVO_OPEN);
      valveStatus = "OPEN";
    } else {
      // Tank full
      pipelineValve.write(SERVO_CLOSED);
      valveStatus = "CLOSED - FULL";
    }
  } else {
    // No rain
    pipelineValve.write(SERVO_CLOSED);
    valveStatus = "CLOSED - NO RAIN";
  }

  // Print status periodically
  static unsigned long lastStatusPrint = 0;
  if (millis() - lastStatusPrint >= 5000) {
    Serial.print("RAIN: ");
    Serial.print(rainPercentage);
    Serial.print("% | TANK: ");
    Serial.print(isTankFull ? "Full" : "Not Full");
    Serial.print(" | VALVE: ");
    Serial.println(valveStatus);

    lastStatusPrint = millis();
  }
}

// Smart Lighting Functions
void initializeLightingSystem() {
```

```
  // Configure pin modes
  pinMode(LIGHT_CONTROL_PIN, OUTPUT);
  pinMode(LIGHT_ULTRASONIC_TRIG_PIN, OUTPUT);
  pinMode(LIGHT_ULTRASONIC_ECHO_PIN, INPUT);

  // Initialize light as off
  digitalWrite(LIGHT_CONTROL_PIN, LOW);
  isLightOn = false;

  Serial.println("Smart Lighting System Initialized");
}

void updateLighting() {
  // Ultrasonic distance measurement with averaging
  long duration, distance = 0;

  // Take a few readings for stability
  const int NUM_READINGS = 3;
  for (int i = 0; i < NUM_READINGS; i++) {
    // Clear the trigger pin
    digitalWrite(LIGHT_ULTRASONIC_TRIG_PIN, LOW);
    delayMicroseconds(2);

    // Send 10µs pulse to trigger
    digitalWrite(LIGHT_ULTRASONIC_TRIG_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(LIGHT_ULTRASONIC_TRIG_PIN, LOW);

    // Measure the echo time and calculate distance
    duration = pulseIn(LIGHT_ULTRASONIC_ECHO_PIN, HIGH, 30000); // Timeout after
30ms

    // Check if we got a valid reading
    if (duration > 0) {
      distance += (duration / 2) / 29.1; // Convert to cm
    }

    // Small delay between readings
    delay(10);
  }

  // Calculate average distance
  distance = distance / NUM_READINGS;

  // Motion detection logic with debouncing
  static unsigned long lastMotionTime = 0;
  static unsigned long lastMotionChange = 0;
  static bool lastMotionState = false;
  bool currentMotionState = (distance < DISTANCE_THRESHOLD && distance > 0);
```

```
  const unsigned long LIGHT_TIMEOUT = 30000; // 30 seconds

  // Debounce motion detection
  if (currentMotionState != lastMotionState) {
    lastMotionChange = millis();
  }

  if ((millis() - lastMotionChange) > DEBOUNCE_DELAY) {
    // Motion state is stable
    if (currentMotionState && !isLightOn) {
      // Motion detected, turn on lights
      digitalWrite(LIGHT_CONTROL_PIN, HIGH);
      isLightOn = true;
      Serial.println("LIGHTING: Motion detected - Lights ON");

      // Reset the motion timer
      lastMotionTime = millis();
    }
    else if (currentMotionState) {
      // Still detecting motion, update timer
      lastMotionTime = millis();
    }
    else if (!currentMotionState && isLightOn && (millis() - lastMotionTime >
LIGHT_TIMEOUT)) {
      // No motion and timeout elapsed, turn off lights
      digitalWrite(LIGHT_CONTROL_PIN, LOW);
      isLightOn = false;
      Serial.println("LIGHTING: No motion for timeout period - Lights OFF");
    }
  }

  // Update last motion state
  lastMotionState = currentMotionState;
}
```

Project Outcome

The integrated smart home system successfully achieved the following results:

1. **Unified Home Management**: Created a single system that handles security, safety, resource management, and convenience features.
2. **Reliable Detection**: The security system can detect intruders using multiple sensors with debouncing to minimize false alarms.
3. **Fire Safety**: The system can detect fire hazards and automatically unlock doors for emergency evacuation while activating alarms.

4. **Access Control**: Implemented a multi-user code verification system for authorized entry.
5. **Water Conservation**: Created an automated rainwater harvesting system that collects water when it rains and prevents overflow.
6. **Energy Efficiency**: Implemented motion-activated lighting that only operates when needed.
7. **Hardware Economy**: Achieved multiple home automation functions with minimal hardware through efficient integration.
8. **Non-Blocking Operation**: All systems operate concurrently without interfering with each other through proper timing management.

Individual Contributions

Security and Fire Detection Systems

- **Shefali Bishnoi (2301CS87):** Developed the security system with ultrasonic and IR sensor integration for intrusion detection. Implemented the access code verification system and door locking mechanism with servo control for authorized entry.
- **Juhi Sahni (2301CS88):** Created the fire detection system using flame sensors to monitor for fire hazards. Implemented emergency protocols including alarm triggering, automatic door unlocking during fire emergencies, and reset functionality when fire is no longer detected.

Environmental Control Systems

- **Saniya Prakash (2301CS49):** Designed the rainwater harvesting system with rain sensors and water level detection. Implemented automated valve control to direct rainwater collection based on rainfall intensity and storage tank capacity.
- **Manvitha Reddy (2301CS29):** Developed the smart lighting system using ultrasonic sensors for motion detection. Created energy-efficient lighting control with automatic timeout features for testing and operation.

Each subsystem functions independently while being integrated into a cohesive smart home automation solution. The code demonstrates effective teamwork through standardized timing mechanisms, shared hardware resources, and consistent debugging outputs.