# SYSTEM DESIGN DOCUMENT
# BASIC HEATER CONTROL SYSTEM

**INTERNSHIP ASSIGNMENT – [UPLIANCE.AI](UPLIANCE.AI)**

**Assignment: Part 1 – System Design Deliverable**

**Wokwi** : https://wokwi.com/projects/436648901922350081

**Github** : https://github.com/Sajidk2002/heater-control-upliance

# 1. MINIMUM SENSORS AND ACTUATORS REQUIRED

A heater control system relies on a combination of input sensors and output actuators to regulate and simulate real-world thermal behavior. Selecting the right components is essential to ensure accurate sensing, reliable actuation, and responsive feedback.

## Required Sensors and Actuators:

➢ **Temperature Sensor (LM35 / DHT22):**

- Measures ambient temperature in real-time.
- Analog (LM35) or digital (DHT22) options are available.
- Enables conditional logic for heating control based on thresholds.
- Highly accurate, easy to interface with Arduino.

➢ **Heater Simulation Output (LED / Digital Pin):**

- Represents the heating element.
- Simple ON/OFF control logic via digital GPIO pin.
- LED can visually simulate heating being active or inactive.

➢ **Feedback Device (Optional LED or Buzzer):**

- Indicates system status:
  - Blinking LED for Heating
  - Solid LED for Target Reached
  - Buzzer alert for Overheat
- Improves usability and debugging, especially in simulation environments.
- Minimum Components Table:

| Components | Quantity | Purpose |
|---|---|---|
| Temperature Sensor | 1 | Reads ambient temperature |
| LED (Heater Sim) | 1 | Simulates heater ON/OFF state |
| LED/Buzzer (Optional) | 1 | Status feedback |

# 2. RECOMMENDED COMMUNICATION PROTOCOL

A key requirement in embedded systems is efficient and clear communication between modules, especially for debugging, monitoring, or external interfacing.

## Selected Protocol: UART (Serial Communication)

**➢ What is UART?**
Universal Asynchronous Receiver-Transmitter (UART) is a serial communication protocol that transmits data asynchronously between two devices using two wires: TX (transmit) and RX (receive).

## Why UART is Ideal for This Project:

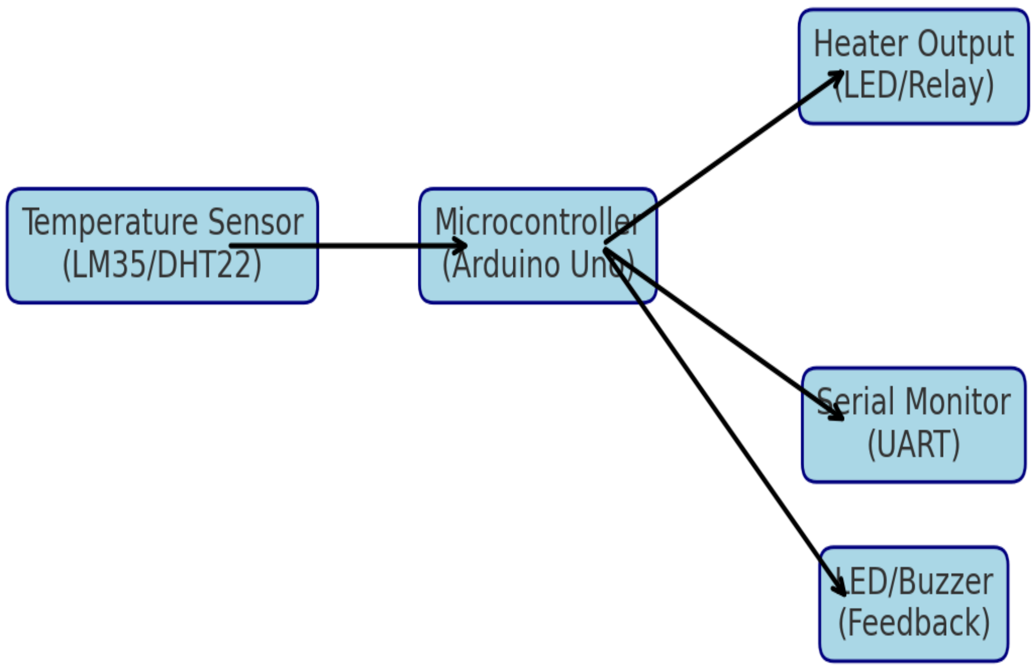
- **Native Support:** Arduino Uno comes with built-in UART over USB.
- **Easy Debugging:** Messages can be viewed using the Serial Monitor.
- **Minimal Hardware:** No external modules needed — just the onboard USB/Serial.
- **Wokwi Compatibility:** Perfectly works with virtual console output in Wokwi simulations.

## Use Cases in This Project:

- Log real-time temperature values
- Show current system state (Heating, Stabilizing, Target Reached, Overheat)
- Debug sensor readings and system response

# 3. BLOCK DIAGRAM OF KEY MODULES

Visual representation of the system architecture enhances understanding of how individual components interact. This block diagram illustrates all functional units and their communication flow.



## Modules Explained:

➢ **Temperature Sensor Module:**

- Continuously monitors the room temperature.
- Sends data to the microcontroller.

➢ **Microcontroller (Arduino Uno):**

- The brain of the system.
- Implements logic using FreeRTOS tasks.
- Controls heater ON/OFF via digital output.
- Sends logs to the serial monitor.
- Triggers visual or audio alerts.

➢ **Heater Output (LED):**

- Simulates real heating element.
- Turns ON when temperature is below threshold (e.g., 35°C).
- Turns OFF when target temperature is reached.

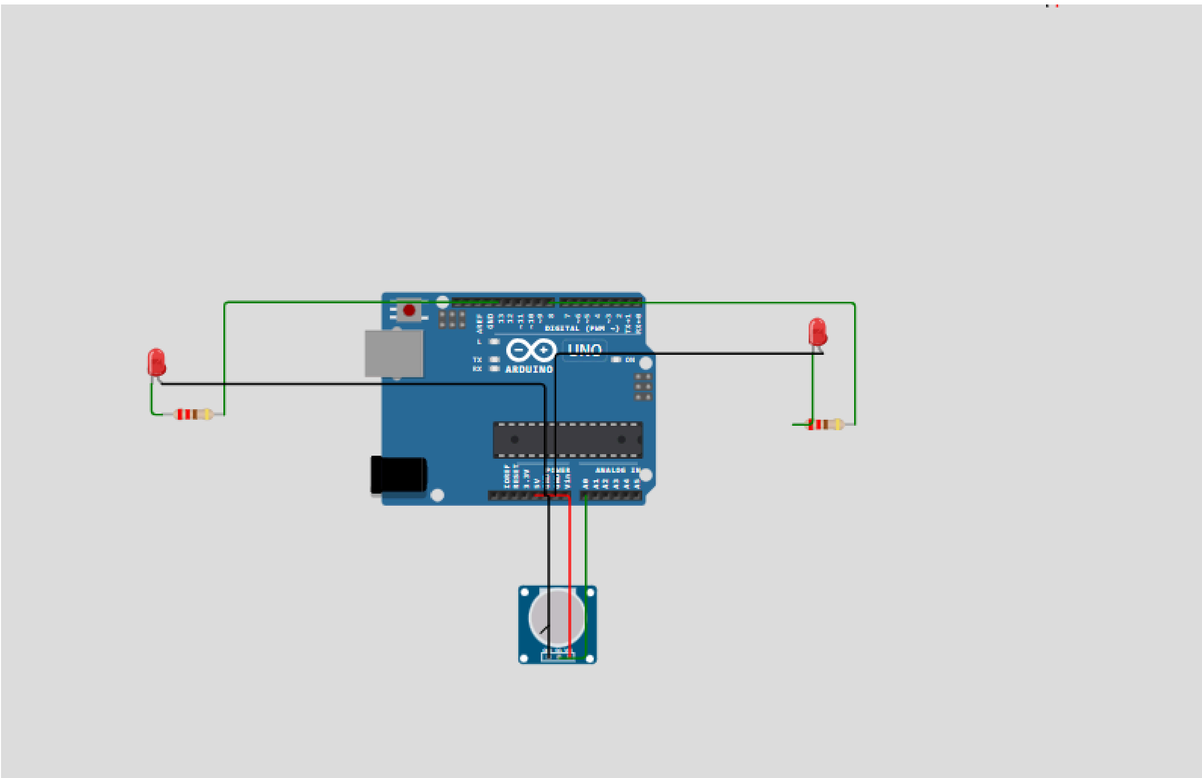
➢ **LED/Buzzer (Optional):**

- Provides clear feedback to the user or tester.

➢ **Serial Monitor:**

- Continuously logs system data for visibility and debugging.

# 4. CIRCUT DIAGRAM



# 5. CODE

```
#include <Arduino_FreeRTOS.h>

#define TEMP_SENSOR_PIN A0
#define HEATER_PIN 8
#define STATUS_LED_PIN 13

#define HEAT_START_TEMP 30
#define TARGET_TEMP 35
#define OVERHEAT_TEMP 45

enum HeaterState { IDLE, HEATING, STABILIZING, TARGET_REACHED, OVERHEAT
};
HeaterState currentState = IDLE;
float temperature = 0;

// Read analog voltage and convert to °C
float readTemperature() {
  int v = analogRead(TEMP_SENSOR_PIN);
  float voltage = v * (5.0f / 1023.0f);
```

```
    return voltage * 100.0f; // 10 mV/°C
  }


  void TempTask(void *pvParameters) {
    while (1) {
      temperature = readTemperature();


      switch (currentState) {
        case IDLE:
          digitalWrite(HEATER_PIN, LOW);
          digitalWrite(STATUS_LED_PIN, LOW);
          if (temperature < HEAT_START_TEMP)
            currentState = HEATING;
          break;
        case HEATING:
          digitalWrite(HEATER_PIN, HIGH);
          if (temperature >= TARGET_TEMP)
            currentState = TARGET_REACHED;
          else
            currentState = STABILIZING;
          break;
```

```
      case STABILIZING:
        if (temperature >= TARGET_TEMP)
          currentState = TARGET_REACHED;
        break;
      case TARGET_REACHED:
        digitalWrite(HEATER_PIN, LOW);
        if (temperature < HEAT_START_TEMP)
          currentState = HEATING;
        else if (temperature >= OVERHEAT_TEMP)
          currentState = OVERHEAT;
        break;
      case OVERHEAT:
        digitalWrite(HEATER_PIN, LOW);
        digitalWrite(STATUS_LED_PIN, HIGH);
        break;
    }

    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}
```

```
void LoggerTask(void *pvParameters) {
  while (1) {
    Serial.print("Temp: ");
    Serial.print(temperature, 1);
    Serial.print(" °C | State: ");
    switch (currentState) {
      case IDLE: Serial.println("IDLE"); break;
      case HEATING: Serial.println("HEATING"); break;
      case STABILIZING: Serial.println("STABILIZING"); break;
      case TARGET_REACHED: Serial.println("TARGET_REACHED"); break;
      case OVERHEAT: Serial.println("OVERHEAT"); break;
    }

    vTaskDelay(pdMS_TO_TICKS(2000));
  }
}

void setup() {
  Serial.begin(9600);
  pinMode(TEMP_SENSOR_PIN, INPUT);
  pinMode(HEATER_PIN, OUTPUT);
```

```
  pinMode(STATUS_LED_PIN, OUTPUT);
  digitalWrite(HEATER_PIN, LOW);
  digitalWrite(STATUS_LED_PIN, LOW);

  xTaskCreate(TempTask, "TempTask", 128, NULL, 2, NULL);
  xTaskCreate(LoggerTask, "LoggerTask", 128, NULL, 1, NULL);
}

void loop() {
  // Handled by FreeRTOS tasks
}
```

# 6. FUTURE ROADMAP

As embedded systems evolve, it is essential to plan for scalability and safety. Below are enhancements that can transform this basic heater controller into a smart, production-grade system.

| Feature | Description |
| --- | --- |
| **Overheat Protection** | System can auto-disable heater if temperature > 70°C to avoid hardware damage. |
| **Multiple Heating Profiles** | Support user-configurable modes like Eco, Normal, Turbo. |
| **BLE Support** | ESP32 can replace Arduino Uno to broadcast heating state wirelessly. |
| **OLED/LCD Display** | Real-time display of temperature and system state. |
| **Mobile App Integration** | Remote control and alerts via smartphone using BLE/Wi-Fi. |
| **Touch/Physical Buttons** | Add manual control options for mode selection or override. |
| **PID Control** | Implement Proportional-Integral-Derivative logic for smoother heating transitions. |

# 7. CONCLUSION

The Basic Heater Control System is a robust and scalable embedded application that demonstrates fundamental control logic, sensor-actuator integration, and task scheduling with FreeRTOS. Designed using an Arduino-based architecture, the system:

- Efficiently reads and processes real-time temperature data
- Simulates heating via output control
- Logs system states via UART for easy debugging
- Supports visual or audio status indicators
- Lays the foundation for smart appliance upgrades