Embedded Systems
Intern Assignment - upliance.ai

By
Shiva Goud
Email: Shivasaigoud99@gmail.com
Ph: 91-7032463940

Embedded Systems Intern Assignment - upliance.ai
Part 1: System Design

Q1. Identify the minimum sensors required for heating detection and control

Minimum Sensors Required:

Temperature Sensor (TMP36 or LM75)
To measure the current temperature so the Arduino can decide whether to turn the heater on or off.

- **Required**: Analog: TMP36, LM75 — simple, low cost, connect to an analog input pin.
- **Not Required but recommended:** Digital: DS18B20 (1-Wire), DHT22 (Not recommended because digital sensors can't support analog)

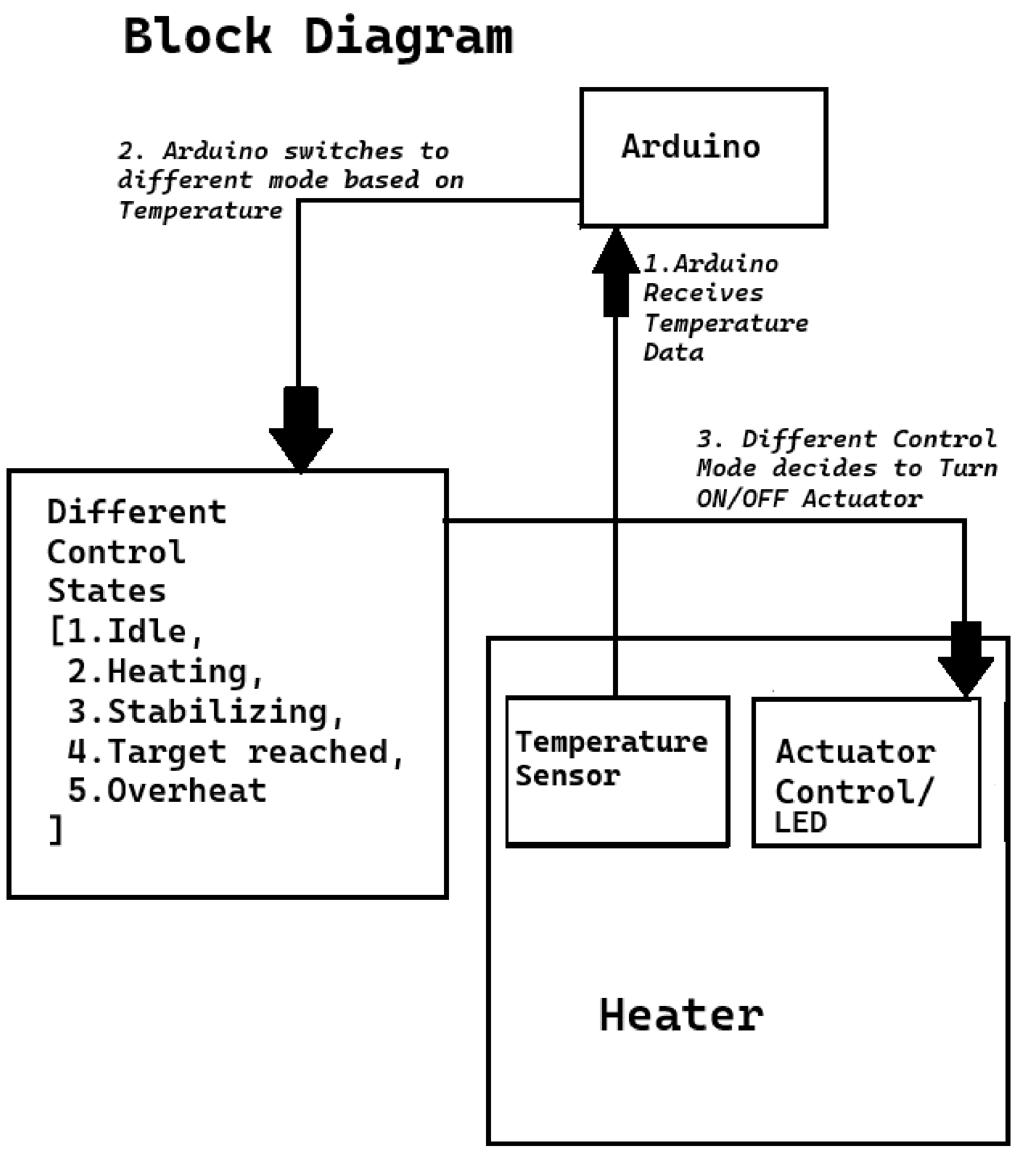
TMP36: Used when no protocol is used & TMP36 uses just in-built ADC
LM75: Can be used with communication protocols such as I2C.

Q2. Recommend a communication protocol and justify the choice

Recommendation: I²C (Inter Integrated Circuit)

Reasons:
1. Scalable: Can add multiple sensors/devices. I2C supports multiple masters & multiple slaves on just two wires (SDA/SCL) .
2. Low pin count/Lines: just 2 signal lines regardless of devices you add . On Arduino Uno that's A4 = SDA, A5 = SCL.
3. No High frequency requirement: Temperature changes slowly; I²C's speed is slow compared to SPI but it is enough.
4.Many Tools & Support: Many temperature sensors support (TMP102, LM75, SHT31), with mature Arduino libraries.
5.Simple wiring: Not many wires like SPI, So no unnecessary complexity.
6.Why no UART or SPI: UART supports only two devices, SPI is complex.

Q3. Provide a block diagram showing key modules



Q4. Outline a future roadmap: How the system could evolve to support overheating protection, multiple heating profiles.

Future roadmap

1. **Overheating protection (safety first)**

- Fault detection: detect sensor open/short (implausible ADC codes, stuck values), relay/MOSFET stuck-on (temperature rising while command=OFF) → latch fault.
- Safe recovery logic: once Overheat is triggered, require cooldown below a safe temp and a manual reset input (button) before re-enabling heater.
- Watchdog + brown-out: enable MCU watchdog; handle brown-out reset to avoid uncontrolled states.

2. **Multiple heating profiles**

- Profiles model: define profiles as arrays of stages (e.g., *Preheat → Soak → Peak → Cool*).
- UI/Storage:
  I2C OLED to display current temp, state, and active profile
  Buttons/encoder to select profile and start/stop.
  Safety per profile: per-stage max temp/rate limits; abort if exceeded.

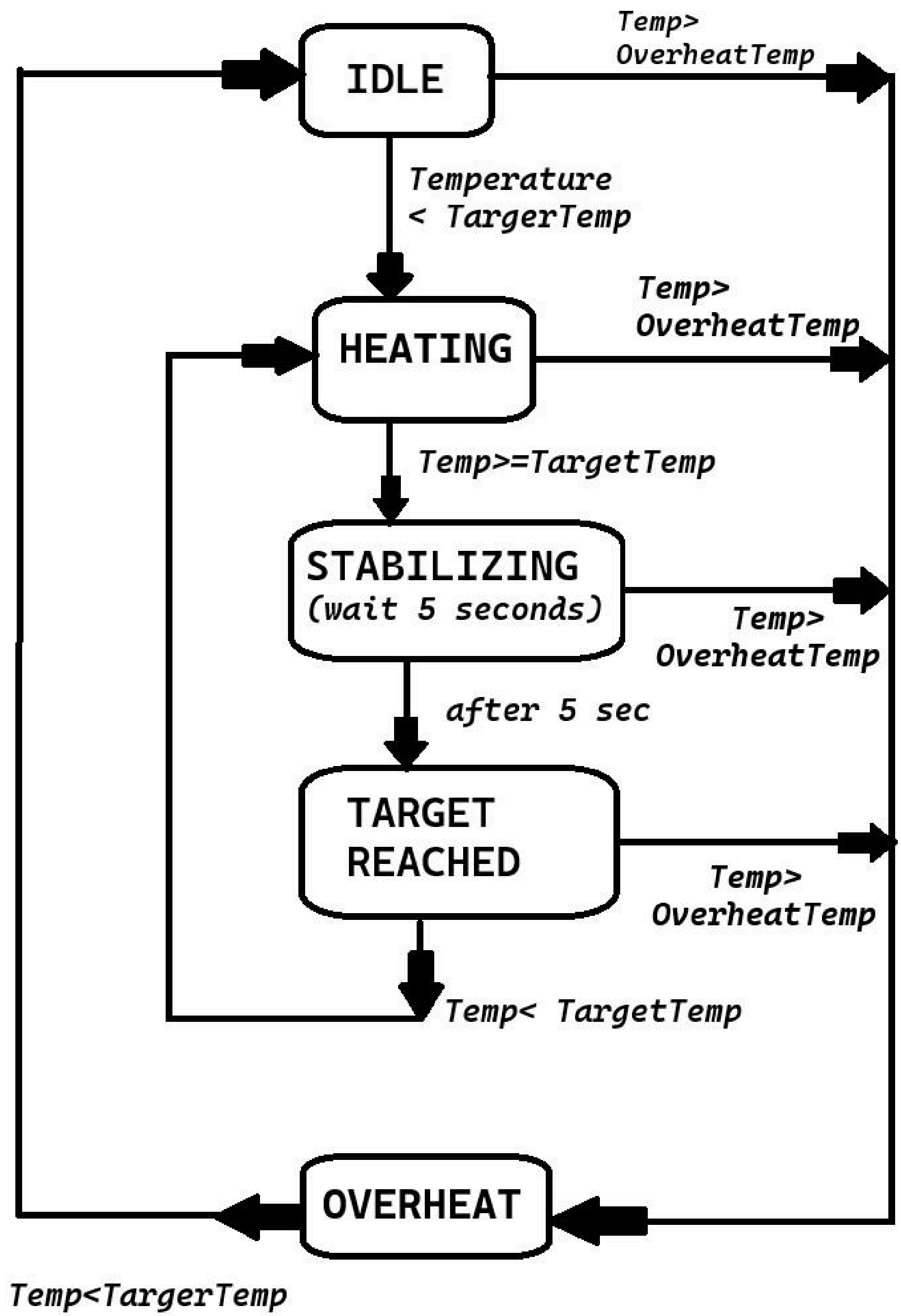
3. **System improvements**

- Scheduling: use timer interrupts for fixed-interval sampling (e.g., 100–250 ms) to keep control timing stable.
- Data logging: log to Serial now data and retain cache
  .

4. **Test & validation**

- Threshold sweep tests: simulate temps across boundaries to verify each state transition.
- Overheat drills: force Overheat path and confirm latch-off & manual reset.

Finite state machine(FSM) diagram of states



Part 2: Embedded Implementation

NOTE: I implemented this project in two ways

IMPLEMENTATION 1: Without protocol (No SPI), using arduino in built ADC function.

IMPLEMENTATION 2: Using SPI protocol.

PROJECT_1

Implementation 1: Project_1 Link: LINK

Implementation 1: Code_1 Link: LINK

PROJECT_2

Implementation 2: Project_2 Link: LINK

Implementation 2: Code_2 Link: LINK

Whole Project Link: LINK

Demo in TinkerCAD: LINK

Demo in Cirkit Designer: LINK

PLEASE NOTE:

I could not find the all the necessary sensors in "WOKWI", So, I used other Platforms such as "Tinkercad" & "CirKit Designer"

CODE(This is the same code that is provided above using Github link)

```
#include <Arduino.h>

#include <Wire.h>  // Library for I²C communication

// ====== I²C SENSOR  ======

// The I²C address for the LM75 temperature sensor (default is 0x48)

const byte LM75_ADDRESS = 0x48;

// ====== PIN DEFINITIONS ======

// Pin used to control the heater (relay or LED)

const int heaterPin = 8;

// Pin used for the warning LED that lights up in overheat conditions

const int ledPin = 13;

// ====== TEMPERATURE THRESHOLDS ======

// Desired target temperature in degrees Celsius

const float targetTemp = 40.0;

// Hysteresis helps avoid frequent on/off switching around the target temperature

const float hysteresis = 2.0;

// Safety limit: If temperature exceeds this, enter OVERHEAT state

const float overheatTemp = 50.0;

// ====== FSM STATES ======

// Enum to represent the current state of the system (finite state machine)

enum State {

   IDLE,           // Waiting for temperature to drop
```

```
    HEATING,        // Heater is on, trying to reach target temperature
    STABILIZING,    // Temperature reached; waiting to stabilize
    TARGET_REACHED, // Temperature stable; heater off
    OVERHEAT        // Emergency state; heater off, warning LED on
};
// Holds the current state of the system
State currentState = IDLE;
// Used to track when a state started (for timing purposes)
unsigned long stateStartTime = 0;
// Time to wait in STABILIZING state before moving to TARGET_REACHED
(milliseconds)
const unsigned long stabilizingTime = 5000;
// ====== FUNCTION PROTOTYPES ======
// Reads temperature from LM75 sensor
float readTemperature();
// Changes the system to a new state
void changeState(State newState);
// Turns the heater on or off
void controlHeater(bool turnOn);
// ====== SETUP ======
void setup() {
    Serial.begin(9600);   // Start serial communication for debugging
    Wire.begin();         // Initialize I²C communication
    pinMode(heaterPin, OUTPUT);   // Set heater pin as output
```

```
  pinMode(ledPin, OUTPUT);     // Set LED pin as output
  // Start in IDLE state
  changeState(IDLE);
  }
// ====== MAIN LOOP ======
void loop() {
  // Read the current temperature
  float temp = readTemperature();

  // Print temperature and current state to Serial Monitor
  Serial.print("Temperature: ");
  Serial.print(temp);
  Serial.print(" °C | State: ");

  // ====== HANDLE EACH STATE ======
  switch (currentState){
    case IDLE:
      Serial.println("IDLE");
      // If temperature is below lower threshold, start heating
      if (temp < targetTemp - hysteresis){
        changeState(HEATING);
      }
      break;
```

```
case HEATING:
    Serial.println("HEATING");
    controlHeater(true);  // Turn heater on
    // If target temperature is reached, start stabilizing
    if (temp >= targetTemp) {
        changeState(STABILIZING);
    }
    break;
case STABILIZING:
    Serial.println("STABILIZING");
    controlHeater(false);  // Turn heater off
    // Wait for a fixed time before assuming temperature is stable
    if (millis() - stateStartTime >= stabilizingTime) {
        changeState(TARGET_REACHED);
    }
    break;

case TARGET_REACHED:
    Serial.println("TARGET_REACHED");
    controlHeater(false);  // Keep heater off
    // If temperature drops, start heating again
    if (temp < targetTemp - hysteresis) {
        changeState(HEATING);
```

```
        }
        break;
      case OVERHEAT:
        Serial.println("OVERHEAT!!!");
        controlHeater(false);          // Turn heater off immediately
        digitalWrite(ledPin, HIGH);    // Turn on warning LED
        // If temperature drops below target, return to IDLE
        if (temp < targetTemp) {
          digitalWrite(ledPin, LOW);   // Turn off warning LED
          changeState(IDLE);
        }
        break;
  }

 // ====== GLOBAL OVERHEAT CHECK ======
 // If temperature exceeds overheat limit from any state, go to OVERHEAT
 if (temp >= overheatTemp && currentState != OVERHEAT) {
      changeState(OVERHEAT);
  }

  delay(500);  // Wait 0.5 seconds before next loop
}
```

```
// ====== READ TEMPERATURE FUNCTION ======

float readTemperature() {

    // Start communication with the LM75 sensor by addressing it over I²C.

    Wire.beginTransmission(LM75_ADDRESS);

    // Tell the LM75 that we want to read from register 0x00, which holds the
temperature.

     Wire.write(0x00);  // 0x00 is the temperature register in LM75

    // End the transmission, but pass 'false' to keep the connection open for the next
read.

    Wire.endTransmission(false);  // This allows a repeated start without releasing the
bus

  // Request 2 bytes from the LM75 sensor. These contain the temperature data.

    Wire.requestFrom(LM75_ADDRESS, (uint8_t)2);

  // Check if we actually received 2 bytes from the sensor

    if (Wire.available() >= 2) {

   // Read the first byte (MSB - most significant byte)

      uint8_t msb = Wire.read();


   // Read the second byte (LSB - least significant byte)

      uint8_t lsb = Wire.read();

   /*

    The LM75 sends temperature as a 9-bit two's complement number:

    - The MSB contains the integer part.

    - The LSB's most significant bit (bit 7) represents 0.5°C.

    - The remaining 7 bits in LSB are unused and should be ignored.
```

```
      So to combine both bytes:

      - First, shift MSB 8 bits to the left.

      - OR with LSB to combine them into a 16-bit number.

      - Then shift the result to 7 bits to the right to drop unused bits.

    */

        int16_t tempRaw = ((msb << 8) | lsb) >> 7;


    // Multiply the raw value by 0.5 to convert it to degrees Celsius

        float temperature = tempRaw * 0.5;


    // Return the final temperature value

        return temperature;

        }

  // If 2 bytes weren't received properly, return an error value

        return -100.0;  // Arbitrary low number indicating failure to read temperature

    }


// ====== READ TEMPERATURE FUNCTION FINISH======

// ====== CHANGE STATE FUNCTION ======

void changeState(State newState){

        currentState = newState;         // Update state

        stateStartTime = millis();       // Record time of change (for timing logic)
```

```
  }

// ====== CONTROL HEATER FUNCTION ======
void controlHeater(bool turnOn) {
    if (turnOn == true) {
      digitalWrite(heaterPin, HIGH);  // Turn heater ON
    }
    else {
      digitalWrite(heaterPin, LOW);   // Turn heater OFF
    }
  }
```