Design Document

Part 1: System Design

Overview

- LED used to simulate heater element or load
- For actuation, relay used to drive heater element, COM would be connected to external power source.
- DS18B20 temp sensor used in pull up configuration, supports one wire mode.
- In real world high-voltage wires on COM/NO never share ground with the ESP32.
- Why ESP32 used?
- Future Road map: how the system could evolve to support overheating protection, multiple heating profiles.

Key Requirements

Requirement	Component Choice
Temperature sensing	**1 × Digital sensor (DS18B20)**
Actuation	**1 × Relay driving the heater load**
MCU	**ESP32** (has built-in Wi-Fi/BLE, dual-core for future RTOS tasks)
Resistor	For pull up configuration

Why DS18B20 is chosen:

1. Provides direct digital temp. No analog to digital conversion
2. ±0.5 °C error range
3. 1-Wire support
4. Can support 100 C temps. Can reach in cooking.

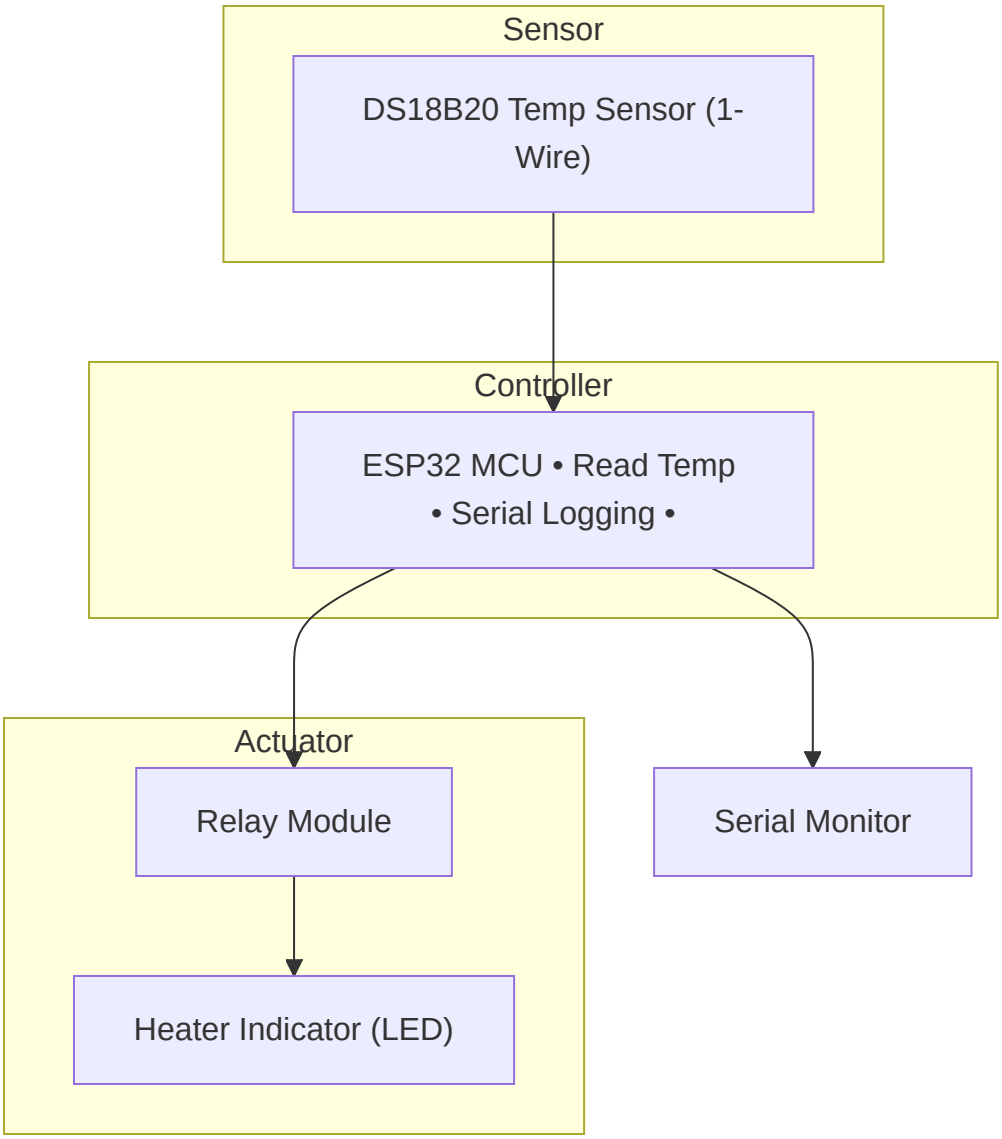
ESP32 is chosen because it has built-in WiFi + BLE for remote networking. PWM. Has low power modes.

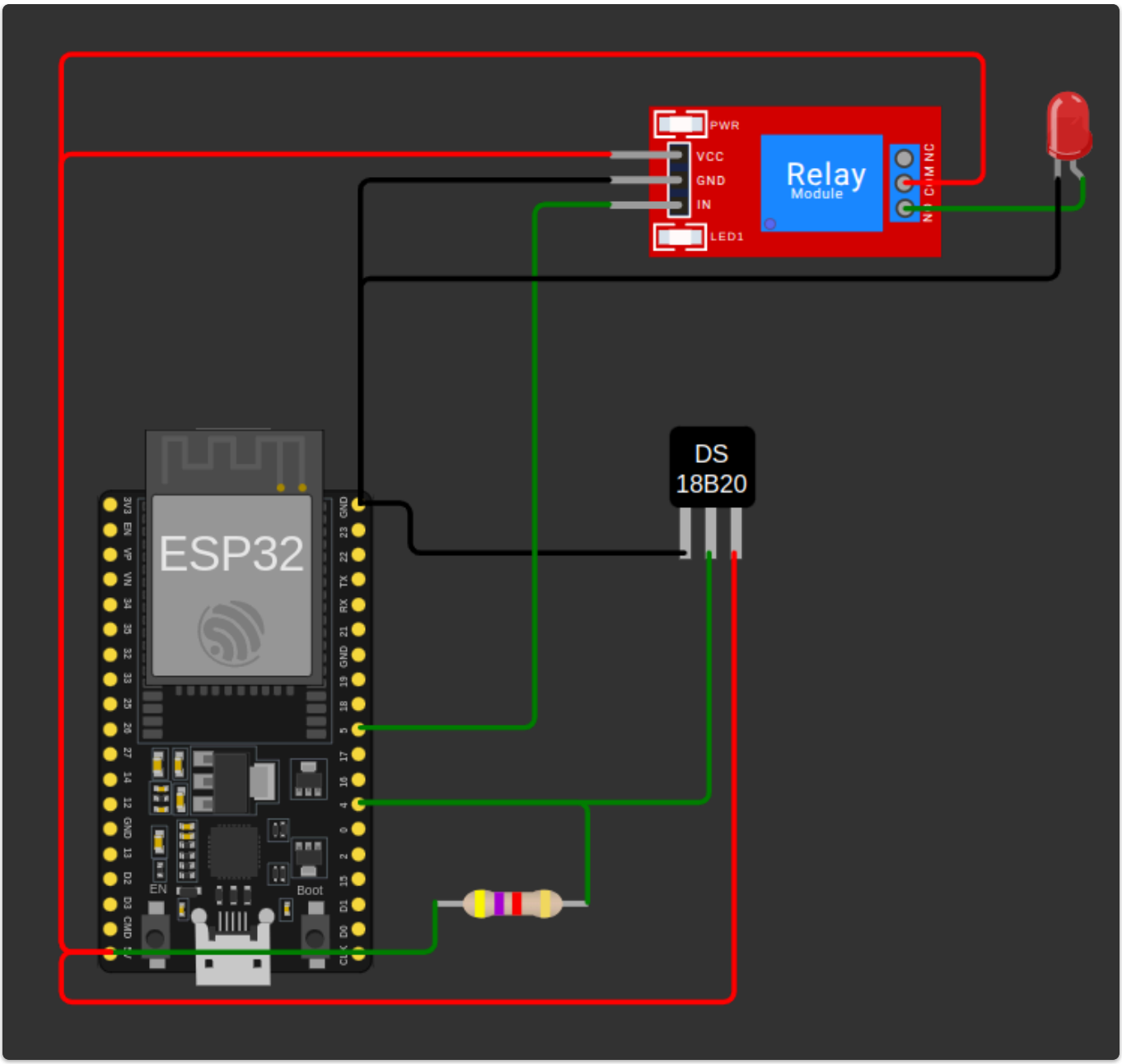
Communication protocol choice

Protocol	Pros	Cons
1-Wire (DS18B20)	Single GPIO, parasitic-power option, CRC in data, Wokwi ready	64-bit ROM search adds a little code
I²C (TMP102)	Multi-sensor bus, wide ecosystem	Needs pull-ups; two wires instead of one
SPI	Very fast	More pins; overkill

Chosen: **1-Wire** for wiring simplicity. 1-Wire allows multi-slave, which means many other 1-one wire sensors can attach to the dq line and based on ROM-id the controller can talk to them, this reduces wiring simplicity at the cost of managing communication complexity in code.

Block Diagram


Sensor
DS18B20 Temp Sensor (1-Wire)
Controller
ESP32 MCU • Read Temp • Serial Logging •
Actuator
Relay Module
Heater Indicator (LED)
Serial Monitor



Future roadmap

- For overheat protection, there could be a thermal fuse/ cutoff which would stop the heater if threshold is exceeded.
- A redundant sensor could also be used to cross-check
- For supporting Multiple heating profiles (eg. Warm food, Quick thaw etc.) The target temps & threshold can be stored in Flash Memory or NVS.
- These temps will be loaded in code for different heating modes.
- A web dashboard can be developed with intuiitve UI to view current heating modes and how much time left.

- Data can also be logged to the cloud using protocols such as MQTT pub-sub etc.
- We can also monitor power consumption for the heater load and optimize energy efficiency.

Part 2: Embedded Implementation

Platform: ESP32
Wokwi simulation link: https://wokwi.com/projects/430314335195796481
Github repo link: https://github.com/h4mmad/upliance

```
#include <OneWire.h>

#include <DallasTemperature.h>



#define ONE_WIRE_BUS 4 // GPIO4

#define HEATER_PIN 5 // GPIO5



const float TARGET_TEMP = 50.0;

const float LOWER_THRESHOLD = 45.0;

const float OVERHEAT_THRESHOLD = 60.0;



OneWire oneWire(ONE_WIRE_BUS);

DallasTemperature sensors(&oneWire);



enum State {IDLE, HEATING, STABILIZING, TARGET_REACHED, OVERHEAT};

State state = IDLE;
```

```
void printState(State s) {

static const char* names[] =

{"IDLE","HEATING","STABILIZING","TARGET_REACHED","OVERHEAT"};

Serial.println(names[s]);

}



void setup() {

Serial.begin(115200);

pinMode(HEATER_PIN, OUTPUT);

sensors.begin();

}



void loop() {

sensors.requestTemperatures();

float tempC = sensors.getTempCByIndex(0);



// State transitions

if (tempC > OVERHEAT_THRESHOLD) state = OVERHEAT;

else if (tempC >= TARGET_TEMP) state = TARGET_REACHED;

else if (tempC > LOWER_THRESHOLD) state = STABILIZING;

else state = HEATING;
```

```
//Actuation

switch(state){

case HEATING:

digitalWrite(HEATER_PIN, HIGH);

break;

case STABILIZING:

digitalWrite(HEATER_PIN, HIGH);

break;

case TARGET_REACHED:

digitalWrite(HEATER_PIN, LOW);

break;

case OVERHEAT:

digitalWrite(HEATER_PIN, LOW);

break;

case IDLE:

digitalWrite(HEATER_PIN, LOW);

break;

}



Serial.print("Temp: ");

Serial.print(tempC);

Serial.print(" °C | State: ");

printState(state);
```

```
delay(100);

}
```

Note: I wanted to add BLE feature, however on Wokwi the following libraries were not supported.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
```

https://github.com/wokwi/wokwi-features/issues/234
Therefore I could not test the functionality of BLE, but understood the working of it.

- When ESP32 boots up, the BLE radio is called, and a human-readable name is given (ex. "Upliance Heater") that other scanners can see.
- Then we need to create a GATT (Generic Attribute Profile) server. It is a framework that defines how two Bluetooth devices exchange data. The GATT server is the device the stores the data attributes you want to expose and responds to requests from a GATT client (ex. smartphone).
- You then add a 128-bit UUID and one or more services. Then add characteristics to each service such as the heater state and start advertising.
- Once the service is started, clients can subscribe to updates.
- Once state changes it pushes the update over the air to any subscribed client.
- So, when the loop runs, it periodicallly sends heater state to the client, such as "IDLE", "HEATING" etc.