Is the A/C on? Final Project

Team Cualquiera

Microprocessors I Fall Semester 2022

Draft 0.1

Edmarie Santana Camacho

Perla N. Colón Marrero

Jonhuel A. De León Hernandez

Contents

The Logical View Architecture

From a logical architecture perspective, the system must use sensors, transmissions, and data processing. To achieve this, you must work with cloud and communication server components.The use of temperature sensors will be implemented with an ESP32 device that will have it included. They will be placed on campus to send measurements to the cloud server. After the sensors measure the temperature values, the data transmission will take place, sending them to a central computer in the cloud, where they can be stored in a database, using wireless communication technologies such as WiFi or Bluetooth. Users should be able to access the system through a website or through a voice assistant like Siri, which allows them to interact with the system to learn temperature measurements.

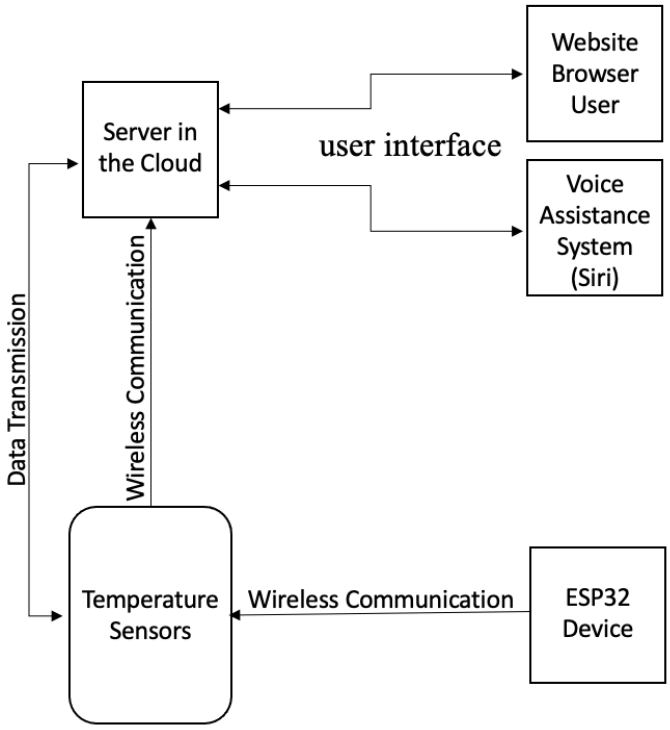


Figure 1 - Diagram for The Logical View Architecture.

In Figure 1, the ESP32 device equipped with temperature sensors is represented. Data transmitted wirelessly from the device is represented by arrows pointing towards the central cloud server. The user interface communicates with the cloud server component to retrieve temperature measurements and display them to users.

The Process View Architecture

The system process architecture view will be split to create a better structure and facilitate the project process. As a first step, all the necessary tools to carry out the project must be installed; Softwares, Hardwares, VS Code, Other editors, Compilers, Frameworks, etc. An execution plan must be created, along with an Architecture and Design Document. API or Pub/Sub data formats must be documented for all communications between system components. All required functionality from ESP32 code will be implemented and a prototype will be made that implements distance sensing on a set of reference points. The Backend of the cloud server will be implemented to achieve the required functionality.

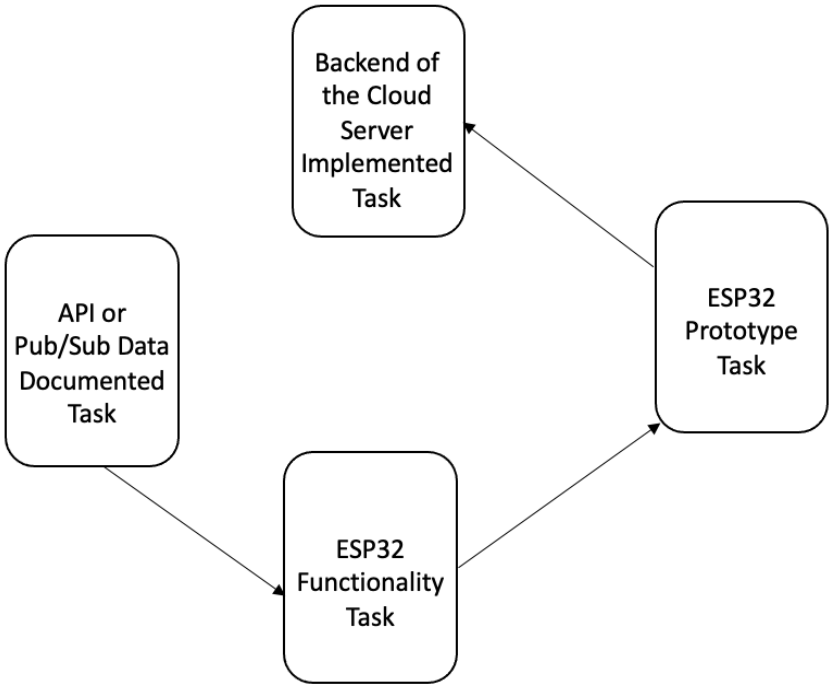


Figure 2 - Diagram for The Process View Architecture.

In Figure 2, the ESP32 device equipped with temperature sensors sends the data to the cloud server via the communication layer, showing the different layers and their interactions to enable continuous data collection, processing and presentation to the users.

The Development View Architecture

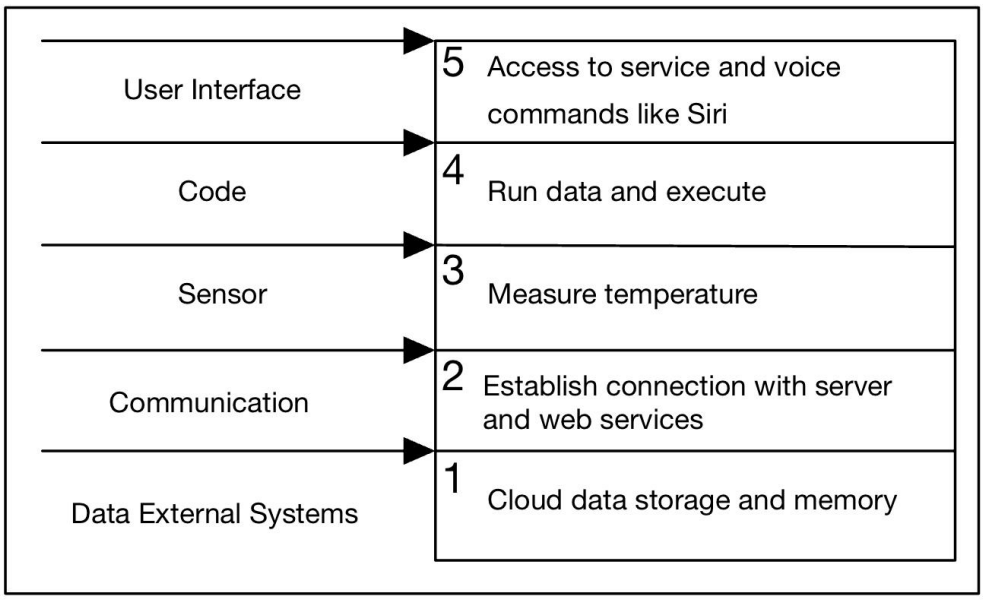


Figure 3 - Diagram for The Development View Architecture

In Figure 3, the user interface layer and communication layer are responsible for providing a way for users to interact with the system and transmitting data to other systems, respectively. The code layer is responsible for the high-level logic of the system, such as how often to read the temperature sensor and how to respond to temperature readings. The data layer processes and stores the temperature data received from the sensor layer. Finally, the sensor layer interfaces with the temperature sensor and reads its data, while the data layer also provides a way to transmit data to cloud-based data storage services or other systems.

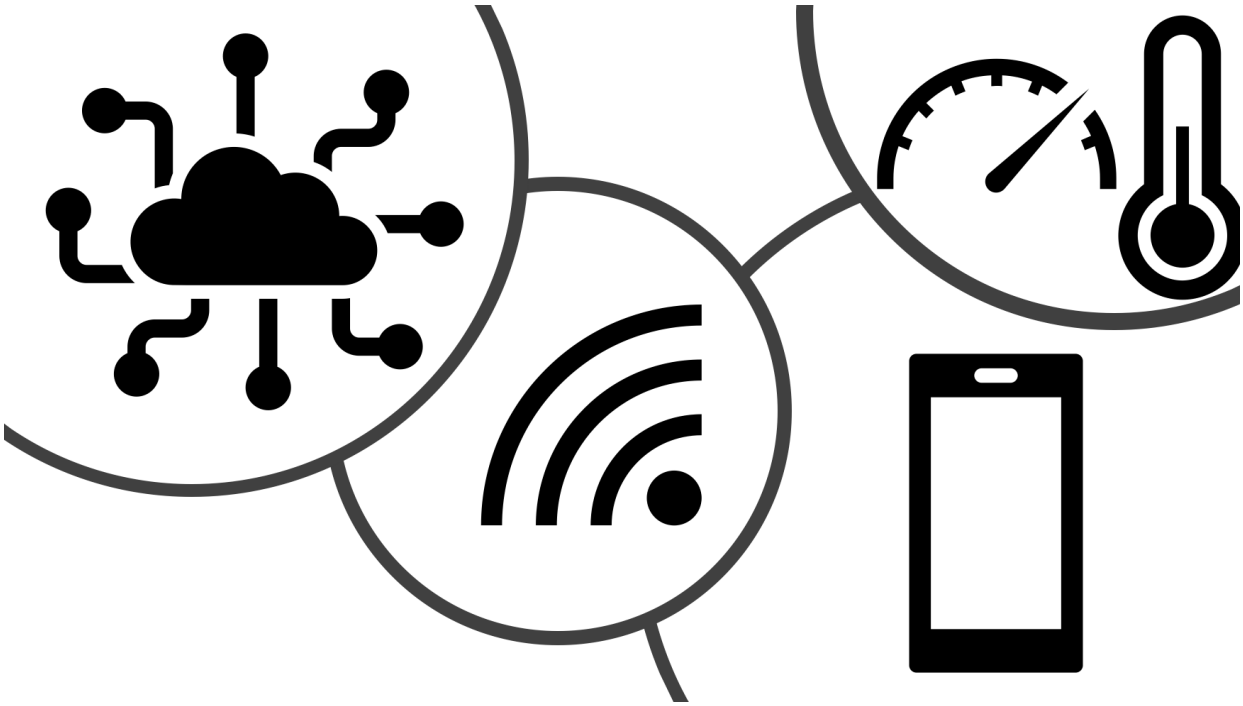
The Physical View Architecture

In this diagram, the temperature sensor is connected to the ESP32 board, which processes the temperature data and communicates with the cloud server over Wi-Fi. The LED display shows the current temperature of the room in real-time. The cloud server stores the temperature data and provides access to it for remote monitoring and analysis.

Figure 4 - Diagram for The Physical View Architecture

Scenarios

1. The ESP32 device along with the temperature sensors will be located on campus.
2. These devices will send constant temperature measurements to a central server in the cloud to share the information.
3. The data is transmitted to the cloud server using a wireless communication protocol, such as Wi-Fi or Bluetooth, where the data is stored to provide useful information to the user.
4. The central server in the cloud stores and processes the data and provides it to the users through the application layer.
5. The interface communicates with the cloud server through API endpoints, which provide access to data stored on the backend server.
6. Users can access a website to review current and historical data on area temperature measurements. They can also query the system through a "Siri" (by saying, for example, "Hey Siri, give me the status of room 205").



Documentation of State Machine

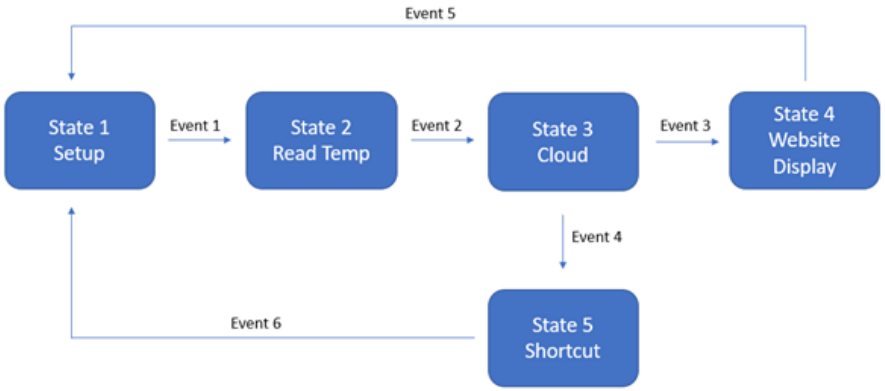


Figure 4 - State Diagram

Present State	Event	Next State
State 1 - SetUp	Event 1	State 2 - Read Temp
State 2 - Read Temp	Event 2	State 3 - Cloud
State 3 - Cloud	Event 3	State 4 - Website Display
State 3 - Cloud	Event 4	State 5 - Shortcut
State 4 - Website Display	Event 5	State 1 - SetUp
State 5 - Shortcut	Event 6	State 1 - SetUp

Figure 5 - State Table

- Setup: The ESP32 device and the temperature sensor are initialized, and serial communication is configured.
- Read Temperature: The temperature is read and stored.
- Cloud: The ESP32 device sends and receives the data to and from the cloud server.
- Website Display: Displays the temperature data on a screen.
- Shortcut: The cloud server receives queries from end users via voice assistants or the web interface, retrieves the appropriate data from the database and delivers the data.

C Code Implemented in Wokwi (Deliverable 3)

```
#include <OneWire.h>
#include <DallasTemperature.h>

const int oneWireBus = 4;

OneWire oneWire(oneWireBus);

DallasTemperature sensors(&oneWire);

void setup() {
  // Start the Serial Monitor
  Serial.begin(115200);
  // Start the DS18B20 sensor
  sensors.begin();
}

void loop() {
  sensors.requestTemperatures();
  float temperatureC = sensors.getTempCByIndex(0);
  float temperatureF = sensors.getTempFByIndex(0);
  Serial.print(temperatureC);
  Serial.println("ºC");
  Serial.print(temperatureF);
  Serial.println("ºF");
  delay(5000);
}
```

```
    }
    else if (strcmp(state, "cloud") == 0) {
        if (strcmp(event, "true") == 0){
            printf("choose: %c or %c\n", 's', 'w');
            char input[10];
            scanf("%9s", input);
            printf("%s", input);
            strcpy(current_state, input);
        }
        if (strcmp(event, "false") == 0) current_state = "cloud";
    }
    else if (strcmp(state, "website display") == 0) {
        if (strcmp(event, "true") == 0){
            current_state = "setup";
        }
        if (strcmp(event, "false") == 0){
            current_state = "website display";
        }
    }
    else if (strcmp(state, "shortcut") == 0) {
        if (strcmp(event, "true") == 0){
            current_state = "setup";
        }
        if (strcmp(event, "false") == 0){
            current_state = "shortcut";
        }
    }
    else{
        printf("invalid\n");
    }
}
```

```ino
#include <OneWire.h>
#include <DallasTemperature.h>

const int oneWireBus = 4;

OneWire oneWire(oneWireBus);

DallasTemperature sensors(&oneWire);

void setup() {
  // Start the Serial Monitor
  Serial.begin(115200);
  // Start the DS18B20 sensor
  sensors.begin();
}

void loop() {
  sensors.requestTemperatures();
  float temperatureC = sensors.getTempCByIndex(0);
  float temperatureF = sensors.getTempFByIndex(0);
  Serial.print(temperatureC);
  Serial.println("ºC");
  Serial.print(temperatureF);
  Serial.println("ºF");
  delay(5000);
}
```

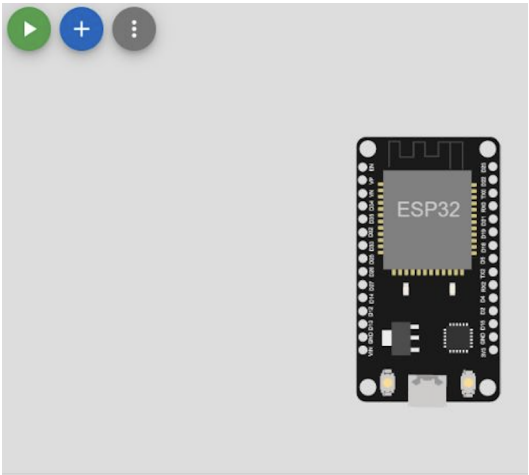


Figure 6 - Output

C Code Implemented in Wokwi (Deliverable 4)

sketch.ino ● diagram.json ● libraries.txt Library Manager

```
#include <OneWire.h>
#include <DallasTemperature.h>

const int oneWireBus = 4;

OneWire oneWire(oneWireBus);

DallasTemperature sensors(&oneWire);

void setup() {
  // Start the Serial Monitor
  Serial.begin(115200);
  // Start the DS18B20 sensor
  sensors.begin();
}

void loop() {
  sensors.requestTemperatures();
  float temperatureC = sensors.getTempCByIndex(0);
  float temperatureF = sensors.getTempFByIndex(0);
  Serial.print(temperatureC);
  Serial.println("ºC");
  Serial.print(temperatureF);
  Serial.println("ºF");
  delay(5000);
}
```

Figure 7 - C Code Implemented in Wokwi

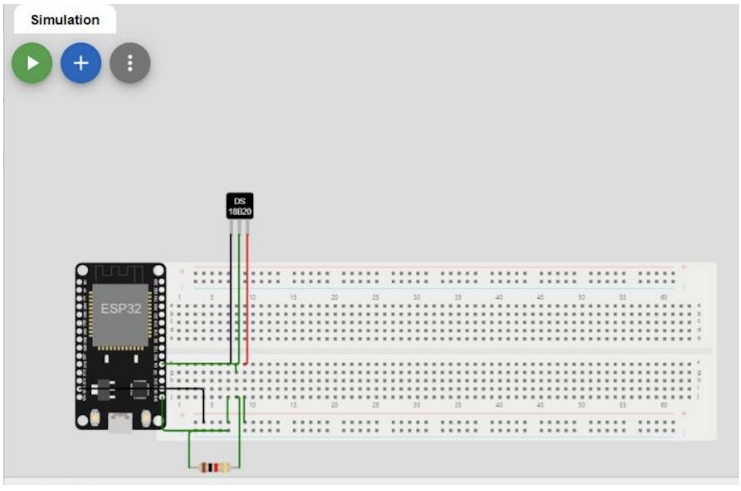


Figure 8 - ESP32 with Temperature Gauge Implemented in Wokwi

Provisioning and Configuration

Provisioning:

• We choose an AWS cloud provider that supports device connectivity.

• An account is created, and a virtual machine is configured on the cloud provider's platform.

• The necessary software components (MQTT, Node Red) have been installed.

• An MQTT broker will be configured in the virtual machine that will act as the messaging service between ESP32 and the Node Red application.

• The virtual machine will be configured to communicate with the ESP32 device.

Setting:

• We will write the firmware for the ESP32 device that reads the temperature from a sensor and publishes it to an MQTT topic.

• An MQTT client will be configured in the virtual machine to subscribe to the MQTT topic and receive temperature data.

• We configure the Node red application to subscribe to the MQTT topic and receive temperature data.

• The virtual machine's firewall is configured to allow incoming Siri requests.

• Configure a Siri Shortcut that sends a request to the Node red app to get the current temperature data.

• The project will be tested to ensure that the ESP32 device publishes temperature data to the MQTT broker, the Node red application receives temperature data from the MQTT broker and responds to Siri requests with accurate temperature data.

In general, this setup allows the ESP32 to communicate with the virtual machine via MQTT, allowing the Node red app to receive temperature data from the ESP32 and respond to Siri requests with the current temperature data.