BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

IMP project

Gesture-controlled Display with MQTT Communication

December 8, 2023

Oleksandr Turytsia (xturyt00)

Contents

1 Introduction

Welcome to the documentation for the Gesture-Controlled Weather Display project. This project aims to create an interactive weather display system controlled by gestures. The display showcases real-time weather information obtained from a weather station, and user interaction is facilitated through an external gesture sensor connected to an ESP32 board.

1.1 Project Overview

The primary goal of this project is to design and implement a weather display system where users can navigate through weather information using gestures. The project utilizes an ESP32 board, connects to a weather station, and incorporates a gesture sensor for intuitive interaction.

2 Installation

This section will walk you through the necessary steps to set up and deploy the project on your ESP32 microcontroller. Before you begin, ensure that you have all the required hardware components and a development environment with the ESP-IDF framework installed.

2.1 Hardware Setup

Before proceeding with the installation, make sure you have the following:

- **ESP32 Board:** The core of the system, responsible for processing data, controlling the display, and managing user interactions.
- **Gesture Sensor (APDS9960):** An external component for detecting user gestures, enabling a hands-free and interactive experience.
- **Display (SSD1306):** An external component for displaying UI.

2.1.1 OLED Display Setup (SSD1306)

To properly interface the SSD1306 OLED display with the ESP32 microcontroller, you need to connect the display to specific GPIO pins. Follow the pin configuration below for a successful setup:

- **Chip Select GPIO** Pin Number: 5
- **Data/Command GPIO** Pin Number: 27
- **Reset GPIO** Pin Number: 17
- **Master Out Slave In GPIO** Pin Number: 23
- **Serial Clock GPIO** Pin Number: 18

Ensure that these connections are made securely, and double-check the pin mapping on your ESP32 development board to confirm the accurate placement.

2.1.2 Gesture Sensor Setup (APDS9960)

For the proper integration of the APDS9960 Gesture Sensor with the ESP32, make the following GPIO pin connections:

- **Serial Data Line GPIO** Pin Number: 25
- **Serial Clock Line GPIO** Pin Number: 26

Refer to the APDS9960 sensor and ESP32 hardware documentation for additional details on pin configuration and any specific considerations for your setup.

Also do not forget to setup VCC and GND on both devices. Ensure that the hardware components are connected properly to the ESP32.

2.1.3 Custom Pins Setup

If you are interested in more custom pins setup, you can change it. All the settings related to pins are located in **main.c** file in root folder of a project:

```
// Pin configurations for SSD1306 OLED display
#define CONFIG_CS_GPIO 5
#define CONFIG_DC_GPIO 27
#define CONFIG_RESET_GPIO 17
#define CONFIG_MOSI_GPIO 23
#define CONFIG_SCLK_GPIO 18

// Pin configurations for I2C communication
#define CONFIG_SDA_GPIO 25
#define CONFIG_SCL_GPIO 26

// APDS9960 sensor configurations
#define APDS9960_ADDR 0x39
```

2.2 Software Installation

Follow these steps to set up the software environment for your project:

- If not already installed, download and install Visual Studio Code and open it.
- Install PlatformIO IDE:
  - Open Visual Studio Code.
  - Navigate to the extensions tab.
  - Install and configure PlatformIO IDE to run ESP-IDF project.
- Copy files to Your project folder.
- Update Wi-Fi credentials:
  - Open source file main.c

  - Locate and update the **SSID** and **PASSWORD** constants in the code with your Wi-Fi credentials

- PlatformIO build:
  - Click on the **Build** button to compile the project
  - After successful compilation, click on **Upload and Run** in the PlatformIO extension

Ensure a stable internet connection and verify that your ESP32 is properly connected to your development machine during the software installation process.

3 Implementation

The Implementation section provides a detailed insight into the structure and execution flow of the project's software components

3.1 Application structure

- **main.c:** The main application file containing initialization logic, event handlers, and the main program loop.
- **ssd1306.c:** Manages interactions with the SSD1306 OLED display, including functions for displaying text and clearing the screen.
- **apds9960.c:** Interfaces with the APDS9960 gesture sensor, providing functions for gesture detection and initialization.

3.2 Project Execution Flow

This section provides a comprehensive overview of how the software and hardware components interact to achieve the intended functionality of the project.

This section breaks down the sequence of operations performed by the system, detailing the major steps and interactions between different modules. Whether you are a new contributor or an experienced developer, gaining insights into the project's execution flow will enhance your ability to comprehend and contribute effectively.

Before diving into the code implementation details, take a moment to familiarize yourself with the overall sequence of events outlined in this section. This knowledge will serve as a roadmap as you navigate through the codebase and make informed decisions when customizing or extending the project functionality.

3.2.1 Entry point

The entry point of a program is **app_main**, which serves as the starting point of the application's execution. This function is crucial for initializing essential components, establishing connections, and launching key processes.

```
void app_main(void)
{
    // Initialize NVS
    init_nvs_flash();

    // Initialize WIFI
    init_wifi();

    // Create a process that handles mqtt events
    xTaskCreate(mqtt_task, "mqtt_task", 8192, NULL, 5, NULL);

    // Start app
    app_run();

    // Cleanup
    cleanup();

    // Restart app if error occured
    esp_restart();
}
```

3.2.2 NVS Flash

The NVS Flash section encompasses the **init_nvs_flash** function, responsible for initializing the Non-Volatile Storage (NVS) flash on the ESP32. This flash memory serves as a persistent key-value storage solution for crucial configuration data [1].

```
void init_nvs_flash() {

    // Initialize the NVS flash
    esp_err_t ret = nvs_flash_init();

    // Check for specific NVS initialization errors
    if (
        ret == ESP_ERR_NVS_NO_FREE_PAGES ||
        ret == ESP_ERR_NVS_NEW_VERSION_FOUND
    ) {
        // Erase NVS flash and attempt initialization again
        ESP_ERROR_CHECK(nvs_flash_erase());

        ret = nvs_flash_init();
    }

    // Log any errors encountered during NVS flash initialization
    ESP_ERROR_CHECK(ret);
}
```

3.2.3 Wi-Fi

The **init_wifi** function, responsible for setting up and initializing the Wi-Fi connectivity on the ESP32.

```
void init_wifi() {

    // Initialize the network interface
    ESP_ERROR_CHECK(esp_netif_init());

    // Create the default event loop
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    // Create the default WiFi station interface
    esp_netif_create_default_wifi_sta();

    // Initialize the WiFi driver with default configuration
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    ...

    // Configure WiFi connection parameters
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = SSID,
            .password = PASSWORD,
        },
    };

    // Set WiFi storage mode to RAM
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));

    // Set WiFi mode to station (STA)
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));

    // Set WiFi configuration
   ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config));

    // Start the WiFi driver
    ESP_ERROR_CHECK(esp_wifi_start());

    // Wait to connect to wifi
    vTaskDelay(1000 / portTICK_PERIOD_MS);

}
```

3.2.4 Wi-Fi Configurations

Wi-Fi related configurations are located in **main.c** in the root folder of a project:

```
#define SSID "<SSID>"
#define PASSWORD "<PASSWORD>"
```

3.2.5 MQTT Integration

The MQTT integration in the project involves handling MQTT (Message Queuing Telemetry Transport) events and communication [2].

The **mqtt_task** function, created as a separate task, manages the MQTT communication. It handles events related to MQTT, ensuring that the application can send and receive messages through the MQTT protocol.

3.2.6 MQTT Communication

Within the **mqtt_task** function, specific logic is implemented to process MQTT messages. This includes handling incoming messages, extracting relevant information, and executing actions based on the received data.

```
void app_main(void)
{
    ...

    // Create a process that handles mqtt events
    xTaskCreate(mqtt_task, "mqtt_task", 8192, NULL, 5, NULL);

    ...
}
```

The project incorporates several communication components, including a Python-based MQTT client responsible for managing weather data. This client subscribes to a specific subject and transmits the data to an MQTT broker. Additionally, there is a cloud server equipped with an MQTT broker and an ESP32 unit, which serves a dual role as both a client and a publisher. The ESP32 client subscribes to the same subject as the weather data client and relays the program's state to the MQTT broker (Figure 1).

The MQTT client handling weather data interprets this state information and, based on the received data, transmits relevant information to the broker. The ESP32 MQTT client then retrieves this data, utilizing it to enable user interaction through sensors and a display interface.

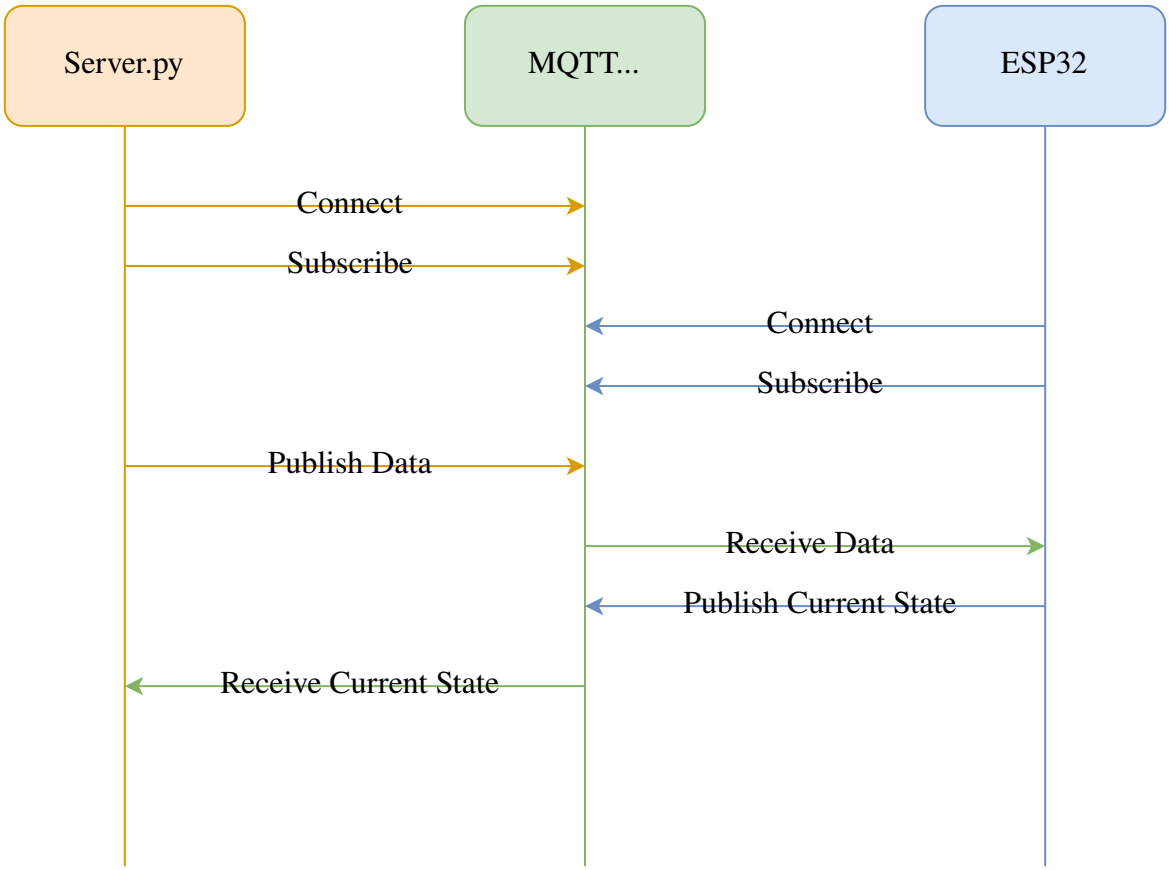

Figure 1: Data sharing between the server, MQTT Brocker and ESP32 using MQTT messages.

3.2.7 MQTT Configurations

MQTT related configurations are located in a file **main.c** in the project root folder. By default values are set for testing purposes:

```
// MQTT message prefixes
#define PREFIX_CITY "[CITY]"
#define PREFIX_DATA "[DATA]"

// MQTT broker configuration
#define CONFIG_BROKER_URL "mqtt://broker.hivemq.com"
#define CONFIG_BROKER_PORT 1883
#define CONFIG_MQTT_TOPIC "test"
```

For the whole setup, user can launch **server.py** to inject testing data into the program:

```
python server.py
```

3.3 Menu structure

The menu system is an abstract component of the project, providing users with a navigable interface to interact with various features and settings. This section offers a comprehensive guide to understand and modify the menu structure.

3.3.1 Menu Hierarchy

As a first page there is a welcome page. After proceeding further, using gestures, user will be met with the following menu hierarchy:

- **Temperature:** This option allows the user to access information related to the current temperature of a chosen city. As a unit program uses Celsius only.
- **Humidity:** This option provides details about humidity levels. Users can check the current humidity in percentages. (0% - very dry, 100% - very wet)
- **Visibility:** This option offers information about visibility conditions. For visualization purposes it shows how clear is the weather in percentages. (0% - visibility is very low, 100% - very clear visibility)
- **Cities:** This is a sub-menu that allows users to explore weather information for specific cities. It contains the following sub-options:
  - Brno
  - London
  - Paris

Users can navigate through these options to gather specific details about temperature, humidity, and visibility, as well as obtain location-specific weather data for the mentioned cities.

Each page in the program is called view. Views are functions, that can be called recursively to imitate inner views inside of others (For example **Menu** $\rightarrow$ **Temperature**)

3.3.2 Menu Navigation

The menu system provides intuitive navigation controls to move between options and sub-menus. Users can interact with the menu using gestures, that are being read from APDS9960 module.

There are following gestures registered to control menu:

- **Up/Down:** Scroll through menu options Up or Down.
- **Select/Enter:** In order to select certain option in menu, user would have to make gesture to the right.
- **Back/Exit:** For exiting nested pages a user can make gesture to the left.

3.3.3 Other View Types

Apart from such views as menu list and informational ones, there is also a view, that requires user's confirmation, which is shown when a user tries to change the city data in **Cities** menu option.

4 Conclusion

This project successfully implements a gesture-controlled weather display, providing an interactive and informative user experience. The combination of the ESP32 microcontroller, OLED display, and gesture sensor creates a versatile and engaging weather station.

5 References

[1] docs.espressif.com. Non-volatile storage library. [Online] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/nvs_flash.html, 2016.

[2] EMQX Team. What is the mqtt protocol and how does it work? [Online] https://www.emqx.com/en/blog/the-easiest-guide-to-getting-started-with-mqtt, 2023.