Comprehensive Documentation of the GYM Management System

Manus AI

July 25, 2025

Abstract

This document provides a comprehensive and in-depth technical overview of the GYM Management System. It details the hardware components, the embedded C code running on the ESP32 microcontroller for RFID scanning, and the Flask-based server application responsible for user management, attendance tracking, and administrative functionalities. The aim is to present a well-structured and professional documentation for the entire system, highlighting its architecture, implementation details, and key features.

Contents

1 Hardware Components

This section provides a detailed examination of the hardware components that form the backbone of the GYM Management System. The system's physical layer is primarily composed of an ESP32 development module, which serves as the central processing unit for the RFID scanning operations, and an RC522 RFID reader module, responsible for interacting with RFID cards. Understanding these components is crucial for comprehending the system's capabilities and limitations.

1.1 ESP32 Development Module

The ESP32 is a series of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth. Developed by Espressif Systems, it is widely used in IoT applications due to its versatility, robust feature set, and strong community support. In the context of the GYM Management System, the ESP32 acts as the intelligent interface between the physical RFID scanning process and the Flask server. Its integrated Wi-Fi capabilities are essential for transmitting RFID data to the central server, enabling real-time attendance tracking.

Key features relevant to this project include:

- **Integrated Wi-Fi:** Facilitates seamless communication with the Flask server over a local network, allowing the ESP32 to send RFID UIDs and receive responses.
- **Dual-Core Processor:** Provides sufficient processing power to handle RFID data acquisition, basic data processing, and network communication concurrently.
- **GPIO Pins:** Offers a multitude of General Purpose Input/Output pins necessary for interfacing with the RC522 RFID reader module via SPI (Serial Peripheral Interface) communication.
- **Low Power Consumption:** Important for potential battery-powered deployments, although in this setup, it is likely continuously powered.
- **ESP-IDF Support:** The Espressif IoT Development Framework (ESP-IDF) provides a robust programming environment in C/C++, offering comprehensive APIs and tools for developing applications on the ESP32.

The specific ESP32 development module used typically integrates the ESP32 chip with necessary peripheral components such as a USB-to-UART converter for programming and serial communication, voltage regulators, and reset/boot buttons. This makes it a self-contained unit ready for development and deployment [1].

1.2 RC522 RFID Reader Module

The RC522 is a highly integrated reader/writer IC for contactless communication at 13.56 MHz. It is designed for low-power, low-cost, and compact applications, making it an ideal choice for embedded systems like the GYM Management System. The module communicates with the ESP32 via the Serial Peripheral Interface (SPI) protocol, which is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems.

Key aspects of the RC522 module in this system:

- **Operating Frequency:** Operates at 13.56 MHz, which is a standard frequency for RFID applications, particularly for ISO/IEC 14443 Type A cards.
- **Communication Protocol:** Utilizes SPI for data exchange with the ESP32. This requires specific pin connections for MISO (Master In, Slave Out), MOSI (Master Out, Slave In), SCLK (Serial Clock), and SDA (Slave Data/Chip Select).
- **PICC (Proximity Integrated Circuit Card) Support:** The RC522 is capable of reading and writing to various types of passive RFID tags and cards, including MIFARE Classic 1K/4K, MIFARE Ultralight, and others that comply with the ISO/IEC 14443 A standard.
- **Antenna Integration:** The module typically comes with an integrated antenna, simplifying hardware design and reducing the need for external antenna tuning.

The RC522 module's ability to quickly and reliably read unique identifiers (UIDs) from RFID cards is fundamental to the system's attendance tracking functionality. When an RFID card is presented to the reader, the RC522 detects its presence, reads its UID, and transmits this information to the ESP32 for further processing and communication with the server.

1.3 RFID Card (13.56MHz Frequency)

RFID cards, also known as proximity cards or smart cards, are passive devices that do not contain a battery. They are powered by the electromagnetic field emitted by the RFID reader (in this case, the RC522 module). The 13.56 MHz frequency is part of the High Frequency (HF) RFID band, which is commonly used for applications requiring short to medium read ranges and higher data transfer rates, such as access control, payment systems, and public transport ticketing.

Each RFID card contains a unique identifier (UID) that the RC522 reader can detect and transmit. In the GYM Management System, these UIDs serve as the primary means of identifying individual gym members. When a member scans their card, the system uses this UID to look up their profile, log their attendance, and update their session count. The use of RFID cards provides a convenient, fast, and secure method for members to check in and out of the gym.

2 ESP32 C Code Implementation

This section delves into the C code running on the ESP32, which is responsible for RFID card detection, Wi-Fi connectivity, and communication with the Flask server. The code is developed using the Espressif IoT Development Framework (ESP-IDF), providing a robust environment for embedded application development. The primary goal of this firmware is to act as a bridge, reading RFID UIDs and relaying them to the central server for processing.

2.1 Code Overview and Structure

The ESP32 firmware is structured to handle concurrent operations: maintaining a Wi-Fi connection, continuously scanning for RFID cards, and asynchronously communicating with the Flask server. The core logic resides within the 'main.c' file, utilizing various ESP-IDF components and FreeRTOS for task management and inter-task communication.

2.1.1 Header Includes and Configuration

The code begins by including essential header files for standard I/O, string manipulation, ESP-IDF logging, time functions, and specific ESP-IDF components like Wi-Fi, event loop, NVS flash (for non-volatile storage), network interface, HTTP client, and HTTP server. Crucially, it includes headers for the 'rc522' library, which simplifies interaction with the RFID module, and FreeRTOS for real-time operating system functionalities.

Configuration macros define critical parameters such as Wi-Fi SSID and password, the Flask server's URL, and the GPIO pins connected to the RC522 module. These parameters are vital for the ESP32 to connect to the network and communicate with the server.

```c
#include <stdio.h>
#include <string.h>
#include <esp_log.h>
#include <time.h>
#include <stdlib.h>
#include "esp_wifi.h"
#include "esp_event.h"
#include "nvs_flash.h"
#include "esp_netif.h"
#include "esp_http_server.h"
#include "esp_http_client.h"
#include "rc522.h"
#include "driver/rc522_spi.h"
#include "picc/rc522_mifare.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"

// Configuration
#define WIFI_SSID "krat"
#define WIFI_PASS "123456789"
#define SERVER_URL "http://192.168.137.1:5000"

// RC522 Pins
#define MISO_PIN 25
#define MOSI_PIN 23
```

```
#define SCLK_PIN 19
#define SDA_PIN  22
#define RST_PIN  -1

static const char *TAG = "RFID_SERVER";
```

Listing 1: Essential Includes and Configuration

2.1.2 Global Variables and Data Structures

Several global variables and a custom data structure are defined to manage the state and facilitate communication between different parts of the firmware. The ‘http$_r esponse_b uffer‘isastaticcha$
$taskcommunication, specificallytopassRFIDUIDdatafromthecarddetectioneventhandlertotheHTT$

```
static char http_response_buffer[512] = {0};
static rc522_driver_handle_t driver;
static rc522_handle_t scanner;
static QueueHandle_t rfid_queue;

typedef struct {
    char uid[32];
    bool scanned;
} rfid_data_t;

static rfid_data_t current_rfid = {0};
```

Listing 2: Global Variables and Data Structure

2.1.3 UID to String Conversion (‘uid_to_string‘)

This utility function converts the raw RFID UID, which is typically an array of bytes, into a human-readable hexadecimal string. This conversion is necessary for transmitting the UID over HTTP to the Flask server, as JSON payloads typically use string representations.

```
void uid_to_string(rc522_picc_t *picc, char *str, size_t len) {
    memset(str, 0, len);
    for (int i = 0; i < picc->uid.length && i < (len-1)/2; i++) {
        sprintf(str + (i*2), "%02X", picc->uid.value[i]);
    }
}
```

Listing 3: UID to String Conversion

2.1.4 RFID Card Detection Handler (‘on_card_detected‘)

This function serves as an event handler triggered when the RC522 module detects a change in the state of a PICC (Proximity Integrated Circuit Card), specifically when a card becomes active. It extracts the RFID UID, converts it to a string using ‘uid$_t o_s tring‘, andthensendst$

```
void on_card_detected(void *arg, esp_event_base_t base, int32_t
   event_id, void *data) {
    rc522_picc_state_changed_event_t *event = data;
    rc522_picc_t *picc = event->picc;

    if (picc->state != RC522_PICC_STATE_ACTIVE) return;

    ESP_LOGI(TAG, "Card detected");
    uid_to_string(picc, current_rfid.uid, sizeof(current_rfid.uid));
    current_rfid.scanned = true;

    ESP_LOGI(TAG, "Scanned UID: %s", current_rfid.uid);

    rfid_data_t data_to_send = {0};
    strcpy(data_to_send.uid, current_rfid.uid);
    data_to_send.scanned = true;

    if (xQueueSend(rfid_queue, &data_to_send, 0) != pdTRUE) {
        ESP_LOGE(TAG, "Failed to send UID to queue");
    }
}
```

Listing 4: RFID Card Detection Handler

2.1.5 HTTP Event Handler (‘http_event_handler‘)

This callback function is registered with the ESP-IDF HTTP client and is invoked when various HTTP events occur during a client request. Its primary role here is to append received data chunks from the HTTP response to the ‘http$_r esponse_b uffer‘.Thisiscrucialforcollectingthe$

```
esp_err_t _http_event_handler(esp_http_client_event_t *evt)
{
    switch(evt->event_id) {
        case HTTP_EVENT_ON_DATA:
            // Append received data to our global buffer
            if (evt->data_len > 0) {
                int current_len = strlen(http_response_buffer);
                int remaining_space = sizeof(http_response_buffer) - current_len - 1;
                int copy_len = (evt->data_len < remaining_space) ? evt->data_len : remaining_space;

                if (copy_len > 0) {
                    strncat(http_response_buffer, (char*)evt->data, copy_len);
                }
            }
            break;
        default:
            break;
    }
    return ESP_OK;
}
```

Listing 5: HTTP Event Handler

2.1.6 RFID Status Handler (‘status_handler‘)

This function acts as an HTTP GET endpoint (‘/rfid-status‘) on the ESP32’s internal web server. When accessed (e.g., by the Flask server to check for new scans), it attempts to retrieve an RFID UID from the ‘rfid$_q ueue‘.IfaUIDisavailable,itconstructsanHTTPPOSTrequesttothe$ $status‘endpoint.ThismechanismallowstheFlaskservertopolltheESP32fornewRFIDscans.$

```c
esp_err_t status_handler(httpd_req_t *req) {
    rfid_data_t data = {0};

    if (xQueueReceive(rfid_queue, &data, 0) == pdTRUE) {
        strcpy(current_rfid.uid, data.uid);
        current_rfid.scanned = true;
        ESP_LOGI(TAG, "Processing UID from queue: %s", current_rfid.uid);
    }

    char response[512];
    if (current_rfid.scanned) {
        // Clear the response buffer before each request
        memset(http_response_buffer, 0, sizeof(http_response_buffer));

        char post_data[100];
        snprintf(post_data, sizeof(post_data), "{\"rfid_uid\":\"%s\"}", current_rfid.uid);

        esp_http_client_config_t config = {
            .url = SERVER_URL "/check_user",
            .method = HTTP_METHOD_POST,
            .timeout_ms = 5000,
            .event_handler = _http_event_handler, // Use our custom event handler
        };

        esp_http_client_handle_t client = esp_http_client_init(&config);
        esp_http_client_set_header(client, "Content-Type", "application/json");
        esp_http_client_set_post_field(client, post_data, strlen(post_data));

        esp_err_t err = esp_http_client_perform(client);
        int status = esp_http_client_get_status_code(client);

        if (err == ESP_OK && status == 200) {
            ESP_LOGI(TAG, "Raw Server Response: %s", http_response_buffer);

            // Check if our buffer actually received data
            if (strlen(http_response_buffer) > 0) {
                // Directly forward the JSON response from the server
                strcpy(response, http_response_buffer);
            } else {
                ESP_LOGE(TAG, "Response body was empty despite 200 OK status.");
                snprintf(response, sizeof(response), "{\"scanned\":true,\"uid\":\"%s\",\"registered\":false}", current_rfid.uid);
            }
        } else {
            ESP_LOGE(TAG, "HTTP Error: %s, Status: %d", esp_err_to_name(err), status);
            snprintf(response, sizeof(response), "{\"scanned\":true,\"uid\":\"%s\",\"registered\":false}", current_rfid.uid);
        }
        esp_http_client_cleanup(client);
    } else {
        strcpy(response, "{\"scanned\":false,\"uid\":\"\",\"registered
    \":false}");
    }

    httpd_resp_set_type(req, "application/json");
    httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*"); // Add
     CORS header
    httpd_resp_send(req, response, HTTPD_RESP_USE_STRLEN);
    return ESP_OK;
}
```

Listing 6: RFID Status Handler

2.1.7 User Registration Handler (‘register_handler‘)

This function implements an HTTP POST endpoint (‘/register‘) on the ESP32. It is designed to receive user registration data (e.g., from a web form submitted to the ESP32 directly, though in this system, the Flask server handles direct user registration). It forwards this received data as a JSON payload to the Flask server’s ‘/save‘ endpoint. This allows the ESP32 to act as an intermediary for registration, though its primary role is RFID scanning.

```
esp_err_t register_handler(httpd_req_t *req) {
    char buf[512];
    int ret = httpd_req_recv(req, buf, sizeof(buf)-1);
    if (ret <= 0) {
        httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST, "No data
    received");
        return ESP_FAIL;
    }
    buf[ret] = '\0';

    ESP_LOGI(TAG, "Registration data: %s", buf);

    // Send to server
    esp_http_client_config_t config = {
        .url = SERVER_URL "/save",
        .method = HTTP_METHOD_POST,
        .timeout_ms = 5000,
    };

    esp_http_client_handle_t client = esp_http_client_init(&config);
    esp_http_client_set_header(client, "Content-Type", "application/
    json");
    esp_http_client_set_post_field(client, buf, strlen(buf));

    esp_err_t err = esp_http_client_perform(client);
    int status = esp_http_client_get_status_code(client);

    if (err == ESP_OK && status == 200) {
        ESP_LOGI(TAG, "Registration successful");
        httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*"); //
     Add CORS header
        httpd_resp_sendstr(req, "Registration successful!");
        memset(&current_rfid, 0, sizeof(current_rfid));
    } else {
        ESP_LOGE(TAG, "Server error: %s, status: %d", esp_err_to_name(err), status);
        httpd_resp_set_hdr(req, "Access-Control-Allow-Origin", "*"); // Add CORS header
        httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR, "Server sync failed");
    }

    esp_http_client_cleanup(client);
    return ESP_OK;
}
```

Listing 7: User Registration Handler

2.1.8 Web Server Initialization (‘start_server‘)

This function initializes and starts a lightweight HTTP server on the ESP32. It registers the ‘/rfid-status‘ (GET) and ‘/register‘ (POST) endpoints. The ESP32’s web server is configured to handle API requests, allowing the Flask server or other clients to interact with it. It’s important to note that this server only exposes API endpoints and does not serve HTML pages.

```
httpd_handle_t start_server(void) {
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();
    httpd_handle_t server = NULL;

    if (httpd_start(&server, &config) == ESP_OK) {
        httpd_uri_t routes[] = {
            {.uri = "/rfid-status", .method = HTTP_GET, .handler = status_handler},
            {.uri = "/register", .method = HTTP_POST, .handler = register_handler}
        };

        for (int i = 0; i < 2; i++) { // Only 2 routes now (removed root handler)
            httpd_register_uri_handler(server, &routes[i]);
        }
        ESP_LOGI(TAG, "Web server started (API endpoints only)");
    }
    return server;
}
```

Listing 8: Web Server Initialization

2.1.9 Wi-Fi Initialization (‘init_wifi‘ and ‘wifi_event_handler‘)

These functions manage the ESP32’s Wi-Fi connectivity. ‘init$_{w}ifi$‘$configurestheWi-$
$Fiinstationmode(STA)toconnecttoanaccesspointdefinedby$‘$WIFI_SSID$‘$and$‘$WIFI_PASS$‘$.The$‘$wifi_e$‘
$Fievents, suchasthedevicestartinguporsuccessfullyobtaininganIPaddress.OnceanIPisacquired, the$‘

```
void wifi_event_handler(void* arg, esp_event_base_t base, int32_t event_id, void* data) {
    if (base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
    } else if (base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t* event = data;
```

```c
        ESP_LOGI(TAG, "WiFi connected, IP: " IPSTR, IP2STR(&event->ip_info.ip));
        start_server();
    }
}

void init_wifi(void) {
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &wifi_event_handler, NULL));
    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &wifi_event_handler, NULL));

    wifi_config_t wifi_config = {
        .sta = {.ssid = WIFI_SSID, .password = WIFI_PASS},
    };

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config));
    ESP_ERROR_CHECK(esp_wifi_start());
}
```

Listing 9: Wi-Fi Initialization

2.1.10 RFID Initialization (‘init_rfid‘)

This function sets up the RC522 RFID reader. It initializes the FreeRTOS queue (‘rfid$_{q}ueue$‘)$forinter-taskcommunication, configurestheSPIinterfacefortheRC522moduleusing$‘$rc522_{s}pi_{c}onfig_{t}$‘$, andcrea$

```c
void init_rfid(void) {
    srand(time(NULL));

    rfid_queue = xQueueCreate(10, sizeof(rfid_data_t));
    if (!rfid_queue) {
        ESP_LOGE(TAG, "Queue creation failed");
        return;
    }

    rc522_spi_config_t driver_config = {
        .host_id = SPI3_HOST,
        .bus_config = &(spi_bus_config_t){
            .miso_io_num = MISO_PIN,
            .mosi_io_num = MOSI_PIN,
            .sclk_io_num = SCLK_PIN,
        },
        .dev_config = {.spics_io_num = SDA_PIN},
        .rst_io_num = RST_PIN,
    };

    rc522_spi_create(&driver_config, &driver);
    rc522_driver_install(driver);

    rc522_config_t scanner_config = {.driver = driver};
    rc522_create(&scanner_config, &scanner);
    rc522_register_events(scanner, RC522_EVENT_PICC_STATE_CHANGED,
   on_card_detected, NULL);
    rc522_start(scanner);

    ESP_LOGI(TAG, "RFID initialized");
}
```

Listing 10: RFID Initialization

2.1.11 Main Application Entry (‘app_main‘)

This is the entry point for the ESP32 application. It initializes the NVS flash (used by ESP-IDF for storing configuration data), then calls ‘init$_rfid‘and‘init_wifi‘tosetuptheRFIDscannerandn$

```
void app_main(void) {
    ESP_LOGI(TAG, "Starting RFID Registration System");

    ESP_ERROR_CHECK(nvs_flash_init());
    init_rfid();
    init_wifi();

    ESP_LOGI(TAG, "System ready - ESP32 provides API endpoints only");
    ESP_LOGI(TAG, "Web interface is served from PC server at http
   ://192.168.137.1:5000");
}
```

Listing 11: Main Application Entry

3 Flask Server Capabilities

This section details the Flask-based web server, which serves as the central hub for the GYM Management System. It handles user registration, stores member data, manages attendance records, and provides an administrative interface. The server is built using Python and the Flask micro-framework, known for its flexibility and simplicity, making it ideal for developing web applications and APIs.

3.1 Server Architecture and Role

The Flask server acts as the central brain of the GYM Management System. It is responsible for data persistence, business logic, and providing both a web interface for user interaction (registration, admin dashboard) and an API for communication with the ESP32 RFID scanner. The server is designed to be lightweight and scalable, capable of handling multiple user requests and RFID scan events concurrently.

Its primary roles include:

- **Data Management:** Stores and retrieves user profiles, attendance logs, and administrative credentials, currently using JSON files for simplicity, but designed with potential for database integration.
- **Business Logic:** Implements rules for user registration, session calculation (e.g., for 16-session subscriptions), and attendance validation.
- **Web Interface:** Renders HTML templates for the user registration page, admin login, and the admin dashboard, providing a user-friendly experience.
- **API Endpoints:** Exposes RESTful API endpoints that the ESP32 scanner and the web frontend interact with to perform operations like checking user status, saving new users, and fetching statistics.
- **Image Processing:** Handles user image uploads, including optimization and secure storage.

3.2 Core Functionalities

3.2.1 User Management

The server provides robust functionalities for managing gym members. New users can register through the web interface, providing their personal details, including a unique RFID UID. The system validates this data to ensure completeness and correctness. User data, including their chosen subscription type and remaining sessions, is stored in ‘users.json‘. The server also handles the secure storage of user profile images, optimizing them for web display.

Administrators have capabilities to view all registered users, delete user accounts, and reset user sessions, particularly for subscription types that have a limited number of sessions per month. This ensures that membership rules are enforced and managed effectively.

3.2.2 Attendance Tracking

Attendance tracking is a core feature, leveraging the RFID technology. When an ESP32 scanner reads an RFID card, it sends the UID to the Flask server. The server then:

- Identifies the user associated with the RFID UID.
- Checks if the user is eligible to use a session for the current day (e.g., preventing multiple check-ins on the same day for session-based plans).
- Updates the user's session count if applicable (e.g., decrementing for 16-session plans).
- Logs the attendance event, including timestamp, user details, and action (check-in/out), into ‘attendance.json‘.

This automated process ensures accurate and real-time attendance records, reducing manual overhead.

3.2.3 Admin Dashboard

The administrative dashboard provides a centralized view for gym staff to monitor and manage the system. Accessible via a secure login, it displays key statistics such as total registered users, active users for the current day, and subscription type distribution. It also allows administrators to view detailed lists of users and attendance records, offering tools for user deletion and session resets. This dashboard is crucial for operational oversight and decision-making.

3.3 Communication Protocols

Effective communication between the different components of the GYM Management System is paramount for its functionality. This system primarily relies on two key communication paradigms: the Serial Peripheral Interface (SPI) for local hardware interaction and the Hypertext Transfer Protocol (HTTP) coupled with JSON for network-based data exchange.

3.3.1 Serial Peripheral Interface (SPI)

SPI is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems. Think of it as a dedicated, high-speed conversation channel between two devices on the same circuit board. In our system, SPI is the language spoken between the ESP32 microcontroller and the RC522 RFID reader module.

- **Intuition:** Imagine you have two friends, Alice (ESP32) and Bob (RC522). They want to exchange information very quickly and precisely. Instead of shouting across a room, they sit next to each other and use a set of dedicated wires. One wire is for Alice to send data to Bob, another for Bob to send data to Alice, a third for a shared clock (to keep their conversation in sync), and a fourth for Alice to tell Bob, "Hey, I'm talking to you now!" (Chip Select).

- **Mini-Example:** When the ESP32 wants to read an RFID card, it sends a command to the RC522 via SPI. The RC522 then performs the scan and sends the card's unique identifier (UID) back to the ESP32, all synchronized by the shared clock signal.

SPI is chosen for this interaction because it's fast, efficient, and well-suited for direct communication between microcontrollers and peripheral devices like the RC522, where low latency and high data integrity are critical.

3.3.2 Hypertext Transfer Protocol (HTTP) and JSON

HTTP is the foundation of data communication for the World Wide Web. It's a request-response protocol, meaning a client sends a request to a server, and the server sends a response back. In our system, HTTP is used for two main types of communication:
1. **ESP32 to Flask Server:** The ESP32 acts as an HTTP client, sending RFID scan data to the Flask server. 2. **Web Browser to Flask Server:** The web browser (running the gym's frontend) acts as an HTTP client, sending user registration data and requesting admin dashboard information from the Flask server.

- **Intuition (HTTP):** Think of HTTP like sending a letter. You (the client) write a letter (HTTP request) to a specific address (server URL) asking for something or telling them something. The post office (internet infrastructure) delivers it. The recipient (server) reads your letter and sends a reply letter (HTTP response) back to you. Each letter has a clear purpose (GET for asking, POST for sending data).

- **Mini-Example (ESP32 to Flask):** When an RFID card is scanned, the ESP32 constructs an HTTP POST request. The 'letter' contains the RFID UID. It sends this 'letter' to the Flask server's '/check$_u ser`address.TheFlaskserverreceivesit,processestheUID,andsendsbacka$

JSON (JavaScript Object Notation) JSON is a lightweight data-interchange format. It's human-readable and easy for machines to parse and generate. It's essentially a way to structure information so that both the sender and receiver understand it. In our system, JSON is the format used for the 'content' of the HTTP 'letters'.

- **Intuition (JSON):** Imagine you're ordering food. Instead of just saying

  "pizza," you use a structured order form (JSON) that clearly states "item: pizza, size: large, toppings: pepperoni, mushrooms." This makes it unambiguous for the kitchen.

- **Mini-Example (JSON Payload):** When the ESP32 sends an RFID UID to the Flask server, the HTTP request body might look like this:

```
{
    "rfid_uid": "A1B2C3D4E5F6"
}

```

Listing 12: Example JSON Payload from ESP32

The Flask server then sends back a JSON response, perhaps like this:

```json
{
    "scanned": true,
    "uid": "A1B2C3D4E5F6",
    "registered": true,
    "username": "John Doe",
    "sessions_left": 15
}

```

Listing 13: Example JSON Response from Flask Server

JSON is preferred for web APIs due to its simplicity, readability, and widespread support across different programming languages and platforms. It allows for efficient and structured exchange of data between the ESP32, the Flask server, and the web browser.

3.4 Flask Server API Endpoints

The Flask server exposes several API endpoints that facilitate communication with the ESP32 scanner and the web-based frontend. These endpoints are designed to handle specific requests, process data, and return appropriate responses, typically in JSON format.

3.4.1 ‘/‘ (GET) - Main Registration Page

This is the root endpoint of the web application. When a user navigates to the server's base URL, this endpoint serves the ‘index.html‘ template, which contains the user registration form. It also passes the configured ESP32 IP address to the template, which might be used by the frontend JavaScript for direct communication or informational purposes.

```python
@app.route("/")
def index():
    """Serve the main registration page"""
    return render_template("index.html", esp32_ip=ESP32_IP)
```

Listing 14: Flask Route: / (GET)

3.4.2 ‘/admin‘ (GET) - Admin Login Page

This endpoint serves the ‘admin$_{l}ogin.html‘template, presentingthelogininterfaceforadministrators.I$

```python
@app.route("/admin")
def admin_login():
    """Serve admin login page"""
    if 'admin_logged_in' in session:
        return redirect(url_for('admin_dashboard'))
    return render_template('admin_login.html')
```

Listing 15: Flask Route: /admin (GET)

3.4.3 ‘/admin/login‘ (POST) - Admin Login Handler

This API endpoint handles the submission of admin login credentials. It expects a JSON payload containing ‘username‘ and ‘password‘. The provided password is hashed and compared against the stored hashed password in ‘admin.json‘. Upon successful authentication, a session variable ‘admin$_{l}ogged_{i}n‘isset, grantingaccesstoprotectedadminroutes.$

```python
@app.route("/admin/login", methods=['POST'])
def admin_login_post():
    """Handle admin login"""
    try:
        data = request.get_json()
        username = data.get('username')
        password = data.get('password')

        if not username or not password:
            return jsonify({"error": "Username and password required"}), 400

        # Hash the provided password
        hashed_password = hashlib.sha256(password.encode()).hexdigest()

        # Load admin credentials
        with open(ADMIN_FILE, 'r') as f:
            admin_data = json.load(f)

        if (username == admin_data['username'] and
            hashed_password == admin_data['password']):
            session['admin_logged_in'] = True
            session['admin_username'] = username
            return jsonify({"success": True}), 200
        else:
            return jsonify({"error": "Invalid credentials"}), 401

    except Exception as e:
        logger.error(f"Error during admin login: {str(e)}")
        return jsonify({"error": "Login failed"}), 500
```

Listing 16: Flask Route: /admin/login (POST)

3.4.4 ‘/admin/logout‘ (POST) - Admin Logout Handler

This endpoint handles administrator logout. It clears the ‘admin$_{l}ogged_{i}n‘sessionvariable, effectivelyer$

```
@app.route("/admin/logout", methods=['POST'])
def admin_logout():
    """Handle admin logout"""
    session.pop('admin_logged_in', None)
    session.pop('admin_username', None)
    return jsonify({"success": True}), 200
```

Listing 17: Flask Route: /admin/logout (POST)

3.4.5 ‘/admin/dashboard‘ (GET) - Admin Dashboard Page

This route serves the ‘admin$_{d}ashboard.html‘template.Accesstothispageisprotectedbythe‘@admin_{r}equir$

```
@app.route("/admin/dashboard")
@admin_required
def admin_dashboard():
    """Serve admin dashboard"""
    return render_template('admin_dashboard.html')
```

Listing 18: Flask Route: /admin/dashboard (GET)

3.4.6 ‘/admin/api/stats‘ (GET) - Get Admin Statistics

This API endpoint provides statistical data for the admin dashboard, including total users, active users today, subscription type distribution, and daily attendance for the last 7 days. This data is aggregated from the ‘users.json‘ and ‘attendance.json‘ files.

```
@app.route("/admin/api/stats")
@admin_required
def admin_stats():
    """Get admin statistics"""
    try:
        stats = get_user_stats()
        return jsonify(stats), 200
    except Exception as e:
        logger.error(f"Error getting admin stats: {str(e)}")
```

```python
        return jsonify({"error": "Failed to get statistics"}), 500
```

Listing 19: Flask Route: /admin/api/stats (GET)

3.4.7 '/admin/api/users' (GET) - Get All Users

This endpoint returns a list of all registered users. For security, sensitive data like passwords are removed before sending the response. This is used by the admin dashboard to display a list of members.

```python
@app.route("/admin/api/users")
@admin_required
def admin_get_users():
    """Get all users for admin"""
    try:
        users = load_users()
        # Remove sensitive data
        safe_users = []
        for user in users:
            safe_user = {k: v for k, v in user.items() if k not in ['
    password']}
            safe_users.append(safe_user)
        return jsonify(safe_users), 200
    except Exception as e:
        logger.error(f"Error getting users: {str(e)}")
        return jsonify({"error": "Failed to get users"}), 500
```

Listing 20: Flask Route: /admin/api/users (GET)

3.4.8 '/admin/api/users/<rfid_uid>' (DELETE) - Delete a User

This endpoint allows administrators to delete a user account based on their RFID UID. It removes the user from the 'users.json' file.

```python
@app.route("/admin/api/users/<rfid_uid>", methods=['DELETE'])
@admin_required
def admin_delete_user(rfid_uid):
    """Delete a user"""
    try:
        users = load_users()
        users = [user for user in users if user.get('rfid_uid') !=
    rfid_uid]

        if save_users(users):
            logger.info(f"User deleted by admin: {rfid_uid}")
            return jsonify({"success": True}), 200
        else:
            return jsonify({"error": "Failed to save changes"}), 500

    except Exception as e:
        logger.error(f"Error deleting user: {str(e)}")
        return jsonify({"error": "Failed to delete user"}), 500
```

Listing 21: Flask Route: /admin/api/users/<rfid$_u id>$ $(DELETE)$

3.4.9 ‘/admin/api/users/<rfid_uid>/reset_sessions‘ (POST) - Reset User Sessions

This endpoint enables administrators to reset a user’s session count. This is particularly useful for subscription types with a limited number of sessions, allowing staff to re-enable a user’s access for a new period. It updates the ‘sessions$_{l}eft‘and‘last_{v}isit_{d}ate‘fieldsforthespecifieduser.$

```python
@app.route("/admin/api/users/<rfid_uid>/reset_sessions", methods=['POST'])
@admin_required
def admin_reset_user_sessions(rfid_uid):
    """Reset user sessions"""
    try:
        users = load_users()

        for i, user in enumerate(users):
            if user.get('rfid_uid') == rfid_uid:
                subscription_type = user.get('subscription_type')
                users[i]['sessions_left'] = calculate_initial_sessions(subscription_type)
                users[i]['last_visit_date'] = None
                users[i]['reset_timestamp'] = datetime.now().isoformat()

                if save_users(users):
                    logger.info(f"Sessions reset by admin for user: {rfid_uid}")
                    return jsonify({"success": True}), 200
                else:
                    return jsonify({"error": "Failed to save changes"}), 500

        return jsonify({"error": "User not found"}), 404

    except Exception as e:
        logger.error(f"Error resetting sessions: {str(e)}")
        return jsonify({"error": "Failed to reset sessions"}), 500
```

Listing 22: Flask Route: /admin/api/users/<rfid$_{u}id > /reset_{s}essions(POST)$

3.4.10 ‘/admin/api/attendance‘ (GET) - Get Attendance Records

This endpoint retrieves recent attendance records, sorted by timestamp. It provides the data for the attendance log display in the admin dashboard.

```python
@app.route("/admin/api/attendance")
@admin_required
def admin_get_attendance():
    """Get attendance records"""
    try:
        attendance_records = load_attendance()
        # Get recent records (last 100)
        recent_records = sorted(attendance_records, key=lambda x: x['timestamp'], reverse=True)[:100]
        return jsonify(recent_records), 200
    except Exception as e:
        logger.error(f"Error getting attendance: {str(e)}")
        return jsonify({"error": "Failed to get attendance records"}),
    500
```

Listing 23: Flask Route: /admin/api/attendance (GET)

3.4.11 ‘/user$_{i}mage/ < filename >$‘$(GET) - ServeUserImages$

This endpoint serves user profile images from the ‘user$_{i}mages$‘$directory.Itincludessecuritymeasuressuc$

```
@app.route('/user_image/<filename>')
def user_image(filename):
    """Serve user images with improved security and error handling"""
    try:
        filename = secure_filename(filename)
        if not filename or not allowed_file(filename):
            logger.warning(f"Invalid filename requested: {filename}")
            abort(400, "Invalid filename")

        filepath = os.path.join(UPLOAD_FOLDER, filename)

        if not os.path.exists(filepath):
            logger.warning(f"Image file not found: {filepath}")
            abort(404, "Image not found")

        file_size = os.path.getsize(filepath)
        if file_size > 10 * 1024 * 1024:  # 10MB limit
            logger.warning(f"Image file too large: {filename} ({
    file_size} bytes)")
            abort(413, "File too large")

        return send_file(
            filepath,
            mimetype='image/jpeg',
            as_attachment=False,
            conditional=True,
            max_age=3600
        )

    except Exception as e:
        logger.error(f"Error serving image {filename}: {str(e)}")
        abort(500, f"Server error: {str(e)}")
```

Listing 24: Flask Route: /user$_{i}mage/ < filename > (GET)$

3.4.12 ‘/save‘ (POST) - Save User Data

This crucial endpoint handles the registration of new users. It receives user data, including the RFID UID and an optional base64-encoded image, validates the data, and saves it to ‘users.json‘. It also handles image optimization and storage. This endpoint is typically called by the web frontend when a new user registers.

```
@app.route('/save', methods=['POST'])
def save_user():
    """Save user with improved validation and error handling"""
    try:
        data = request.get_json()
        if not data:
            return 'Bad Request: No data provided', 400

        is_valid, validation_message = validate_user_data(data)
        if not is_valid:
            logger.warning(f"Invalid user data: {validation_message}")
            return f'Validation Error: {validation_message}', 400

        users = load_users()

        rfid_uid = data.get('rfid_uid')
        for existing_user in users:
            if existing_user.get('rfid_uid') == rfid_uid:
                logger.warning(f"Duplicate RFID UID registration attempt: {rfid_uid}")
                return 'Error: RFID card already registered', 409 # Conflict

        # Hash the password before saving
        password = data.get('password')
        if password:
            data['password'] = hashlib.sha256(password.encode()).hexdigest()

        # Handle image upload
        image_data = data.pop('image_data', None)
        image_filename = save_user_image(image_data, rfid_uid)
        if image_filename:
            data['image_filename'] = image_filename

        # Calculate initial sessions based on subscription type
        data['sessions_left'] = calculate_initial_sessions(data.get('subscription_type'))
        data['last_visit_date'] = None # No visit yet

        users.append(data)
        if save_users(users):
            logger.info(f"User registered successfully: {data.get('username')}")
            return jsonify({"success": True, "message": "User registered successfully"}), 201
        else:
            return jsonify({"error": "Failed to save user"}), 500

    except Exception as e:
        logger.error(f"Error saving user: {str(e)}")
        return jsonify({"error": "Internal server error"}), 500
```

Listing 25: Flask Route: /save (POST)

3.4.13 ‘/check$_u ser‘(POST) - CheckUserStatus$

This endpoint is primarily called by the ESP32 scanner. It receives an RFID UID and checks the user’s status, including whether they are registered, their subscription type, and their remaining sessions. It also handles logging attendance and updating session counts. This is the core endpoint for the RFID-based check-in system.

```
@app.route('/check_user', methods=['POST'])
```

```python
def check_user():
    """Check user status and log attendance"""
    try:
        data = request.get_json()
        rfid_uid = data.get('rfid_uid')

        if not rfid_uid:
            return jsonify({"error": "RFID UID is required"}), 400

        users = load_users()
        user_found = None
        for user in users:
            if user.get('rfid_uid') == rfid_uid:
                user_found = user
                break

        if user_found:
            # Check if user can use a session today
            if can_use_session_today(user_found):
                # Update user's session count and last visit date
                updated_user = update_user_session(user_found)
                save_users(users) # Save the updated users list
                log_attendance(updated_user, "check_in")

                return jsonify({
                    "scanned": True,
                    "uid": rfid_uid,
                    "registered": True,
                    "username": updated_user.get('username'),
                    "subscription_type": updated_user.get('
subscription_type'),
                    "sessions_left": updated_user.get('sessions_left'),
                    "message": f"Welcome, {updated_user.get('username')
}!"
                }), 200
            else:
                return jsonify({
                    "scanned": True,
                    "uid": rfid_uid,
                    "registered": True,
                    "username": user_found.get('username'),
                    "subscription_type": user_found.get('
subscription_type'),
                    "sessions_left": user_found.get('sessions_left'),
                    "message": f"{user_found.get('username')}, you have
 already checked in today."
                }), 200
        else:
            return jsonify({"scanned": True, "uid": rfid_uid, "
registered": False, "message": "User not found. Please register."}),
 404

    except Exception as e:
        logger.error(f"Error checking user: {str(e)}")
        return jsonify({"error": "Internal server error"}), 500
```

Listing 26: Flask Route: /check$_user(POST)$

4 Conclusion

This document has provided a comprehensive overview of the GYM Management System, detailing its hardware components, the embedded C code on the ESP32, and the Flask server capabilities. The system effectively integrates RFID technology for automated attendance tracking with a robust web-based management interface, offering a modern solution for gym operations. The modular design, separating hardware interaction from server-side logic, ensures scalability and maintainability.

Future enhancements could include migrating data storage from JSON files to a more robust database system (e.g., PostgreSQL, MySQL) for improved data integrity, scalability, and security. Implementing a more sophisticated user interface for the admin dashboard, incorporating real-time data visualization, and exploring secure over-the-air (OTA) updates for the ESP32 firmware would further enhance the system's capabilities and user experience.

References

[1] Espressif ESP-IDF Programming Guide: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html