ESP32 Attendance System

Real-time VGA output for seamless monitoring on standard displays.

January 7, 2025

Amir Hossein Raeghi
Ali Shokoohi

موضوع: گزارش روند پیشرفت پروژه پایانی

درس: سیستم‌های نهفته و بی‌درنگ

گروه Embedded Developers

علی شکوهی

امیرحسین راعقی

شماره دانشجویی:

۴۰۰۵۲۱۴۷۷

۴۰۰۵۲۲۳۷۳

مقدمه و هدف پروژه

پروژه‌ای که در دست داریم، ترکیبی از دو بخش عمدهٔ نرم‌افزار و سخت‌افزار است. هدف اصلی، راه‌اندازی یک سیستم حضور و غیاب است که قادر باشد با دریافت پیام‌های MQTT از طریق اینترنت (روی میکروکنترلر ESP32)، وضعیت ورود و خروج افراد را ثبت کرده و سپس نتیجهٔ این حضور و غیاب را در قالب یک خروجی گرافیکی (VGA) نمایش دهد. برای این منظور چند مرحله اصلی طی شد:

راه‌اندازی منتشرکننده (Publisher) و سابسکرایبر (Subscriber) در پایتون برای تست ارسال و دریافت پیام‌های MQTT.

انتخاب بروکر (ابتدا تلاش با لوکال‌هاست و سپس مهاجرت به یک بروکر آنلاین نظیر emqx).

انتقال کد Subscriber به ESP32 و اطمینان از کارکرد درست آن.

پیداکردن کتابخانهٔ مناسب برای خروجی VGA. در ابتدا کتابخانهٔ LVGL (نسخه ۹) آزموده شد؛ ولی کامپایل موفق نبود. سپس fabgl آزمایش شد که موفقیت‌آمیز نبود، بعد Bitluni (ESP32Video.h) تست و موفق شدیم. در نهایت برای بهبود دیزاین، تلاش شد از کتابخانهٔ TFT_eSPI استفاده شود که مخصوص نمایشگر TFT بود و نه VGA. دوباره به LVGL بازگشتیم و این بار با نسخهٔ ۸.۳.۰ موفق شدیم نمونه کدها را کامپایل کنیم.

پیاده‌سازی دو بخش مهم در خروجی تصویر: یکی صفحهٔ IDLE (جدول حضوروغیاب) و دیگری صفحاتی برای موفقیت و عدم موفقیت که پس از تشخیص تگ یا اطلاعات، نمایش داده شود.

رفع مشکل عدم تغییر صفحه از طریق نوشتن تابعی جداگانه برای عوض‌کردن صفحات در LVGL و هماهنگی آن با کتابخانهٔ Bitluni.

در این گزارش، با نگاهی موشکافانه روند پیشرفت پروژه را شرح می‌دهیم؛ از اولین قدم‌های برنامه‌نویسی MQTT در پایتون تا پیاده‌سازی در محیط آردوینو IDE، و در نهایت مشکلات و راهکارهای پیاده‌سازی خروجی VGA. در این گزارش تلاش کرده‌ایم تمام جزئیات لازم را به‌صورت مفصل بیان کنیم.

بخش نخست: بررسی مقدماتی MQTT و پیاده‌سازی اولیه در پایتون

۱.۱) راه‌اندازی Publisher و Subscriber در پایتون

در ابتدای کار، به‌منظور اطمینان از صحت عملکرد بخش شبکه و پروتکل MQTT، تصمیم گرفتیم که ابتدا از پایتون استفاده کنیم. در این مرحله، دو فایل مجزا ایجاد شد:

publisher.py: برای ارسال پیام به بروکر MQTT

subscriber.py: برای دریافت پیام از بروکر

کتابخانه‌ای که مورد استفاده قرار گرفت، paho-mqtt بود. این کتابخانه در پایتون ابزاری قدرتمند و ساده برای ساخت کلاینت MQTT محسوب می‌شود. ما با استفاده از pip آن را نصب کردیم:

```
pip install paho-mqtt
```

سپس در کدها، با استفاده از توابع mqtt.Client() و on_connect() و on_message() به‌راحتی ساختار مورد نیاز برای ارسال و دریافت پیام‌ها فراهم شد.

۱.۲) چالش در استفاده از Localhost و مهاجرت به بروکر آنلاین

ابتدا تلاش کردیم با localhost (به‌عنوان یک بروکر محلی) کار کنیم؛ یعنی هم Publisher و هم Subscriber روی همان سیستم محلی اجرا شوند. اما به دلایلی چون پیکربندی فایروال یا ناسازگاری پورت‌ها، پیام‌ها به‌درستی تبادل نمی‌شد. سپس به سراغ یک بروکر آنلاین به نام emqx رفتیم. با استفاده از آدرس broker.emqx.io و پورت ۱۸۸۳ و همچنین تنظیم موضوع پیام (Topic)، موفق شدیم پیام‌ها را در Publisher ارسال کرده و در Subscriber به‌درستی دریافت کنیم.

بخش دوم: پیاده‌سازی Subscriber در Arduino IDE برای ESP32

۲.۱) انتقال کد به محیط آردوینو و استفاده از کتابخانه PubSubClient

پس از اطمینان از عملکرد صحیح در پایتون، نوبت به پیاده‌سازی بر روی میکروکنترلر ESP32 رسید. این کار در محیط Arduino IDE انجام شد و کتابخانهٔ مهمی که برای MQTT بر روی میکروکنترلر استفاده گردید، PubSubClient بود.

کد زیر، بخشی از راه‌اندازی اولیه برای MQTT در فایل main.ino است:

```
WiFiClient espClient;
PubSubClient client(espClient);

const char *mqtt_broker = "broker.emqx.io";
const char *topic = "Employees Es4031/topic";
const char *mqtt_username = "emqx";
const char *mqtt_password = "public";
const int mqtt_port = 1883;

// ...
void connectToMQTT() {
    while (!client.connected()) {
        String client_id = "esp32-client-";
        client_id += String(WiFi.macAddress());
        if (client.connect(client_id.c_str(), mqtt_username, mqtt_password)) {
            Serial.println("Connected to MQTT broker");
            client.subscribe(topic);
        } else {
            delay(2000);
        }
    }
}
```

۲.۲) اتصال به Wi-Fi

پیش از اتصال به MQTT، باید ابتدا ESP32 به شبکه Wi-Fi متصل شود. این امر با کتابخانه‌های استاندارد آردوینو قابل انجام است. در کد زیر، روند اتصال به Wi-Fi نمایش‌داده‌شده است:

```
const char *ssid = "AmirHossein";
const char *password = "********";

void connectToWiFi() {
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
    }
    // ...
}
```

با این روش، ما اول اتصال به شبکه را برقرار کرده و سپس سراغ اتصال به بروکر MQTT می‌رویم.

بخش سوم: پایگاه داده SQLite در سمت پایتون و همگام‌سازی با ESP32

۳.۱) پایگاه داده و جداول مورد نیاز

برای ثبت سوابق کارمندان و وضعیت حضوروغیاب، از SQLite در پایتون استفاده شد. در کد publisher.py، مشاهده می‌کنیم که پایگاه داده با نام Employees.db ساخته شده و جدولی به نام employees به شکل زیر ایجاد می‌شود:

```
cur.execute('''
CREATE TABLE IF NOT EXISTS employees (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    enter_time TEXT NOT NULL,
    exit_time TEXT,
    status TEXT NOT NULL,
    elapsed_time INTEGER DEFAULT 0
)
''')
conn.commit()
```

این جدول، اطلاعات لازم برای شناسایی افراد را شامل می‌شود:

id = شناسه کارمند

name = نام کارمند

enter_time = زمان ورود

exit_time = زمان خروج

status = وضعیت حضور

elapsed_time = مدت زمانی که از ورود گذشته است

۳.۲) تولید پیام‌های نمونه و اضافه‌کردن به دیتابیس

در تابع simulate_attendance() در فایل publisher.py، ما یک فهرست نمونه از کارکنان را ایجاد کرده و اطلاعات را به صورت پیام‌های JSON برای MQTT می‌فرستیم. سپس در دیتابیس نیز به‌روزرسانی انجام می‌گیرد:

```
employees_data = [
    {
        "ID": 0,
        "name": "Unknown",
        "status": "Failed"
    },
    {
        "ID": 1,
        "name": "Amirhosein",
        "status": "Success"
    },
    {
        "ID": 2,
        "name": "Ali",
        "status": "Success"
    }
]
```

با هر بار اجرای این تابع، برای تست سیستم حضور و غیاب، این داده‌ها به بروکر ارسال می‌شود تا بتوانیم پیغام دریافت‌شده توسط ESP32 را ببینیم.

بخش چهارم: بررسی کتابخانه‌های مختلف برای خروجی VGA

یکی از اصلی‌ترین بخش‌های پروژه، گرفتن خروجی VGA از روی برد ESP32 است. به طور کلی چند کتابخانهٔ مهم برای این منظور وجود دارد که هرکدام مزایا و معایب خود را دارند. ما قدم به قدم آن‌ها را بررسی کردیم:

۴.۱) کتابخانه LVGL (نسخه ۹)

در ابتدا LVGL که یک کتابخانهٔ گرافیکی جامع و پیشرفته است، مورد استفاده قرار گرفت. متأسفانه با نسخهٔ ۹ این کتابخانه در کامپایل و تنظیمات برای VGA به مشکل خوردیم و نمی‌توانستیم اجرای موفقی داشته باشیم. کتابخانهٔ LVGL به صورت عمومی برای کنترل صفحات نمایشگرهای LCD/TFT به کار می‌رود اما با VGA هم با تنظیمات اضافی قابل استفاده است.

۴.۲) کتابخانه fabgl

سپس به سمت کتابخانهٔ fabgl رفتیم که به‌طور خاص برای VGA روی برخی بردهای خاص ESP32 طراحی شده است. با این حال، در تست‌های اولیه و با توجه به سخت‌افزار موجود، موفق به گرفتن خروجی پایدار نشدیم. مشکلاتی در تنظیم پین‌ها و وضوح تصویر وجود داشت که مانع از ادامهٔ کار با fabgl شد.

۴.۳) کتابخانه ESP32Video.h (Bitluni)

مرحله بعد، استفاده از کتابخانهٔ Bitluni یا همان فایل هدر ESP32Video.h بود. این کتابخانه توسط فردی به نام "Bitluni" توسعه داده شده و راهکار آسان‌تری برای خروجی VGA فراهم می‌کند. ما با استفاده از مثال Hello World توانستیم به‌سرعت یک متن ساده را روی صفحهٔ مانیتور VGA ببینیم که نشانگر موفقیت اولیه در پیکربندی پین‌ها و نمایش تصویر بود.

علی‌رغم این‌که خروجی اولیه به درستی کار کرد، کیفیت ظاهری (یا به عبارت دیگر، دیزاین گرافیکی) چندان چشمگیر نبود؛ چرا که Bitluni بیشتر روی پایهٔ تولید سیگنال VGA در سطح خیلی پایه و مقدماتی تمرکز دارد. با این وجود، برای نمایش پیام‌های ساده (مانند پیام موفقیت یا عدم موفقیت ورود) کاملاً کاربردی است.

۴.۴) کتابخانه TFT_eSPI

به صورت جداگانه، برای بهبود ظاهر خروجی، تلاش کردیم از کتابخانهٔ TFT_eSPI استفاده کنیم. اما پس از مدتی متوجه شدیم که این کتابخانه برای اتصال به نمایشگرهای TFT طراحی شده و ارتباطی با پین‌های سیگنال VGA ندارد. در نتیجه، این کتابخانه ناکارآمد بود و از چرخهٔ انتخاب خارج شد.

۴.۵) بازگشت مجدد به LVGL (نسخه ۸.۳.۰)

در نهایت برای استفادهٔ هم‌زمان از امکانات گرافیکی LVGL و سیگنال VGA، نسخهٔ ۸.۳.۰ کتابخانهٔ LVGL تست شد. در این نسخه، ما توانستیم مثال‌های اولیه را کامپایل و اجرا کنیم. سپس با اتصال آن به Bitluni، یک تست Hello World ترکیبی انجام دادیم که نشان داد قابلیت همزمانی آن‌ها فراهم است. این یک گام مهم در پیشبرد پروژه محسوب می‌شد، زیرا ما امکانات پیشرفتهٔ LVGL (نظیر ساخت صفحات مختلف، دکمه‌ها، جداول، و ...) را در کنار سیگنال‌دهی VGA کتابخانهٔ Bitluni یک‌جا به دست می‌آوردیم.

بخش پنجم: طراحی صفحه IDLE، صفحه موفقیت و صفحه عدم موفقیت در LVGL

۵.۱) صفحه IDLE (نمایش جدول)

پس از آن که متوجه شدیم می‌توان از LVGL (نسخهٔ ۸.۳.۰) در کنار Bitluni استفاده کرد، تصمیم گرفتیم محتوای اصلی را در قالب سه صفحه نمایش دهیم:

صفحه IDLE: در حالت عادی که کسی اطلاعات ارسال نمی‌کند یا پس از گذشت مدتی که پیام جدیدی دریافت نشده، سیستم یک جدول (Table) شامل اطلاعات آخرین حاضران (فیلدهای ID، نام، زمان ورود، زمان خروج) را نشان دهد.

صفحه موفقیت (Success): زمانی که کاربری وارد سیستم شده و وضعیتش در پیام MQTT برابر "Success" است، این صفحه نشان داده شود.

صفحه ناموفق (Failed): زمانی که کارت یا مشخصات فردی در سیستم ثبت نشده یا هرگونه خطای شناسایی رخ داده و پیام MQTT برابر "Failed" است، سیستم این صفحه مشکی رنگ را نشان دهد.

در کد نمونه، یک شیء جدول با دستور lv_table_create() ساخته شده و با lv_table_set_cell_value() برای هر خانه، اطلاعات مربوطه درج می‌شود. این روند در تابع update_table() انجام می‌گیرد. نمونهٔ کد:

```
void update_table() {
    lv_table_set_col_cnt(table, 4); // ID, Name, Enter, Exit
    lv_table_set_row_cnt(table, max_entries + 1); // Extra row for header

    lv_table_set_cell_value(table, 0, 0, "ID");
    lv_table_set_cell_value(table, 0, 1, "Name");
    lv_table_set_cell_value(table, 0, 2, "Enter");
    lv_table_set_cell_value(table, 0, 3, "Exit");

    for (int i = 0; i < max_entries; i++) {
        String id_str = String(attendees[i].id);
        lv_table_set_cell_value(table, i + 1, 0, id_str.c_str());
        lv_table_set_cell_value(table, i + 1, 1, attendees[i].name.c_str());
        lv_table_set_cell_value(table, i + 1, 2, attendees[i].enter_time.c_str());
        lv_table_set_cell_value(table, i + 1, 3, attendees[i].exit_time.c_str());
    }
}
```

۵.۲) صفحات موفقیت و ناموفق

دو صفحهٔ دیگر، با کد سادهٔ زیر ساخته شده‌اند:

```
success_screen = lv_obj_create(NULL);
lv_obj_t *success_label = lv_label_create(success_screen);
lv_label_set_text(success_label, "Login Successful!");
lv_obj_align(success_label, LV_ALIGN_CENTER, 0, 0);

failure_screen = lv_obj_create(NULL);
lv_obj_set_style_bg_color(failure_screen, lv_color_hex(0xFF0000), LV_PART_MAIN);
lv_obj_t *failure_label = lv_label_create(failure_screen);
lv_label_set_text(failure_label, "Login Failed!");
lv_obj_align(failure_label, LV_ALIGN_CENTER, 0, 0);
```

در صفحهٔ ناموفق، پس‌زمینه را مشکی تنظیم کردیم تا کاربر سریعاً متوجه خطا شود.

بخش ششم: مکانیزم تغییر صفحات در حین دریافت پیام MQTT

۶.۱) تابع switch_screen

مشکل اصلی که ابتدا داشتیم این بود که پس از دریافت پیام‌های MQTT، صفحه تغییر نمی‌کرد و سیستم در همان صفحهٔ IDLE می‌ماند. برای حل این چالش، تابع switch_screen() نوشته شد تا با فراخوانی آن بتوانیم هر زمان که خواستیم صفحهٔ جدید را بارگذاری کنیم. در کد زیر نمونه‌ای از آن آمده است:

```
void switch_screen(lv_obj_t *screen, const String &name = "") {
    if (screen == success_screen) {
        lv_obj_t *success_label = lv_obj_get_child(screen, 0);
        String success_msg = "Login Successful " + name;
        lv_label_set_text(success_label, success_msg.c_str());
    }
    lv_scr_load(screen);
    lv_refr_now(NULL);
    last_screen_change = millis();
}
```

هنگامی که پیام MQTT با وضعیت "Success" دریافت می‌شود، ما از این تابع استفاده می‌کنیم تا به صفحهٔ موفقیت برویم و در عین حال نام فرد را نیز به متن نمایش داده‌شده اضافه کنیم. در صورت "Failed"، ما وارد صفحهٔ مشکی رنگ می‌شویم.

۶.۲) تایمر بازگشت به صفحه IDLE

در کد اصلی، متغیری برای نگهداری زمان آخرین تغییر صفحه در نظر گرفته شده است (last_screen_change). یک فاصله زمانی مشخص (مانند ۳ ثانیه) را تعریف کرده‌ایم تا بعد از گذشت این مدت، دوباره به صفحهٔ اصلی (IDLE) بازگردیم:

```
const unsigned long screen_timeout = 3000;

void loop() {
    // ...
    if (millis() - last_screen_change > screen_timeout) {
        lv_scr_load(idle_screen);
        lv_refr_now(NULL);
    }
    lv_timer_handler();
    delay(5);
}
```

به این ترتیب، سیستم بعد از چند ثانیه به‌طور خودکار از صفحات موفقیت یا ناموفق دوباره به صفحهٔ جدول بازمی‌گردد.

بخش هفتم: مکانیزم تغییر وضعیت به Idle در سمت پایتون

۷.۱) تاخیر ۵ ثانیه‌ای و محاسبه مدت زمان حضور

در فایل publisher.py تابعی با نام update_status_to_idle() وجود دارد که کاری جالب انجام می‌دهد. پس از گذشت ۵ ثانیه از ورود فرد (که در دیتابیس ثبت شده است)، به‌صورت خودکار وضعیت را از "Success" به "Idle"

تغییر می‌دهد. این تابع علاوه بر تغییر status، مقدار exit_time را نیز به زمان فعلی تنظیم می‌کند و مقدار زمانی که فرد حضور داشته را محاسبه کرده و در فیلد elapsed_time ذخیره می‌کند.

```
def update_status_to_idle(employee_id):
    time.sleep(5)
    cur.execute('SELECT enter_time FROM employees WHERE id = ?', (employee_id,))
    row = cur.fetchone()
    if row and row[0]:
        enter_time_str = row[0]
        enter_time = datetime.datetime.strptime(enter_time_str, "%H:%M:%S")
        current_time = datetime.datetime.now()
        elapsed_time = int((current_time - enter_time).total_seconds())
        exit_time_str = current_time.strftime("%Y-%m-%d %H:%M:%S")

        cur.execute('''
        UPDATE employees
        SET status = ?, exit_time = ?, elapsed_time = ?
        WHERE id = ?
        ''', ("Idle", exit_time_str, elapsed_time, employee_id))
        conn.commit()
        # سپس پیام جدید را منتشر می‌کند
        # ...
```

به این شکل، سمت پایتون نیز فعالانه پایگاه داده را به‌روز می‌کند و در صورت نیاز پیام‌های جدید MQTT را برای اطلاع‌رسانی به ESP32 ارسال می‌کند.

۷.۲) هماهنگی اطلاعات نمایش داده‌شده

هر بار که پیام جدیدی از سمت پایتون ارسال می‌شود (چه پیام ورود موفق، چه پیام تغییر وضعیت به Idle)، ESP32 آن را دریافت کرده، در آرایهٔ attendees ذخیره می‌کند و تابع update_table() را صدا می‌زند تا جدول صفحهٔ IDLE به‌روز شود. این همگام‌سازی با ساختار حلقهٔ MQTT و تابع callback() مدیریت می‌شود:

```
void callback(char *topic, byte *payload, unsigned int length) {
    // ... دریافت JSON و تجزیه ...
    if (status == "Success") {
        switch_screen(success_screen, name);
    } else if (status == "Failed") {
        switch_screen(failure_screen);
    }
    // درج رکورد در آرایه
    for (int i = max_entries - 1; i > 0; i--) {
        attendees[i] = attendees[i - 1];
    }
    attendees[0] = {id, name, enter_time, exit_time};
    update_table();
}
```

بخش هشتم: ساختار کلی فایل main.ino

۸.۱) معرفی کتابخانه‌ها

در ابتدای فایل main.ino، کتابخانه‌های کلیدی که مورد استفاده قرار می‌گیرند را می‌بینیم:

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <ArduinoJson.h>
#include <ESP32Video.h> // Bitluni
#include <lvgl.h>       // LVGL
```

این کتابخانه‌ها هریک وظایف مجزایی دارند:

WiFi.h: مدیریت شبکه وای‌فای

PubSubClient.h: مدیریت پروتکل MQTT

ArduinoJson.h: تجزیه و تولید داده‌های JSON

ESP32Video.h: کتابخانه Bitluni برای سیگنال VGA

lvgl.h: کتابخانه LVGL برای ایجاد محیط گرافیکی و ویجت‌های متنوع

۸.۲) پیکربندی پین‌های VGA در Bitluni

در ادامه، یک سری پین‌ها برای خروجی VGA تعریف شده‌اند. به عنوان مثال:

```
const int VGA_RED_PIN = 18;
const int VGA_GREEN_PIN = 22;
const int VGA_BLUE_PIN = 21;
const int VGA_HSYNC_PIN = 16;
const int VGA_VSYNC_PIN = 17;
```

سپس با دستور vga.init(VGAMode::MODE320x240, VGA_RED_PIN, ...)، حالت 320x240 برای رزولوشن در نظر گرفته شده است.

۸.۳) پیکربندی LVGL

برای استفاده از LVGL در کنار Bitluni، باید یک Display Driver برای LVGL تعریف شود. در اینجا تابع lvgl_flush() برای رسم پیکسل‌ها بر روی صفحه فراخوانی می‌شود:

```
void lvgl_flush(lv_disp_drv_t *disp, const lv_area_t *area, lv_color_t *color_p) {
    for (int y = area->y1; y <= area->y2; y++) {
        for (int x = area->x1; x <= area->x2; x++) {
            vga.dot(x, y, color_p->full);
            color_p++;
        }
    }
    lv_disp_flush_ready(disp);
}
```

به این شکل، هر بار که LVGL نیاز به رندر کردن بخشی از صفحه دارد، از این تابع کمک می‌گیرد تا با توجه به مختصات پیکسل‌ها، نقاط مورد نظر را رنگ‌آمیزی کند.

بخش نهم: تفسیر دقیق‌تر از روند عملکرد برنامه

برای فهم بهتر، می‌توان روند کلی را این‌گونه جمع‌بندی کرد:

راه‌اندازی اولیه: ESP32 روشن شده، به Wi-Fi وصل می‌شود، به بروکر متصل می‌شود. همزمان با استفاده از کتابخانهٔ Bitluni، پورت‌های VGA را پیکربندی می‌کند. سپس با LVGL، سه صفحه (IDLE، Success، Failure) ساخته می‌شوند.

در حلقهٔ اصلی (loop):

بررسی می‌شود آیا به MQTT متصل هست یا نه. اگر نبود مجدداً تلاش می‌شود.

دریافت پیام‌های MQTT در پس‌زمینه (callback) انجام می‌شود. هر زمان پیام جدید بیاید، پردازش شده و بسته به پارامتر status، سیستم به صفحهٔ مرتبط رفته یا جدول را به‌روز می‌کند.

اگر مدت زمان مشخصی (مثلاً ۳ ثانیه) از آخرین تغییر صفحه گذشته باشد، صفحه دوباره به IDLE بازمی‌گردد تا جدول نمایش یابد.

دستور lv_timer_handler() نیز مرتبا فراخوانی می‌شود تا LVGL بتواند روال داخلی خود را مدیریت کند (انیمیشن‌ها، رویدادها، ...).

برای جلوگیری از استفاده بیش از حد از CPU، در انتهای حلقه کمی تاخیر delay(5) قرار داده شده است.

بخش دهم: بررسی کد publisher.py به طور مفصل

۱۰.۱) تابع simulate_attendance()

این تابع نقش شبیه‌ساز ورود کارکنان را ایفا می‌کند. به این معنی که مجموعه‌ای از رکوردها (ID، name، status) را یکی پس از دیگری به بروکر MQTT ارسال می‌کند. هر رکورد، دارای زمان ورود (enter_time) است که در لحظهٔ ساخت پیام از datetime.datetime.now() گرفته می‌شود:

```python
def simulate_attendance():
    employees_data = [
        {"ID": 0, "name": "Unknown", "status": "Failed"},
        {"ID": 1, "name": "Amirhosein", "status": "Success"},
        {"ID": 2, "name": "Ali", "status": "Success"}
    ]

    for emp in employees_data:
        current_time = datetime.datetime.now().strftime("%H:%M:%S")
        payload = {
            "ID": emp["ID"],
            "name": emp["name"],
            "enter_time": current_time,
            "exit_time": None,
            "status": emp["status"],
            "elapsed_time": 0
        }
        add_to_db(payload)
        publish_employee_data(payload)
        threading.Thread(target=update_status_to_idle, args=(emp["ID"],)).start()
        time.sleep(2)
```

در این بخش:

add_to_db(payload): رکورد جدید را در دیتابیس وارد یا جایگزین می‌کند.

publish_employee_data(payload): پیام را به بروکر ارسال می‌کند تا Subscriber آن را دریافت کند.

threading.Thread(...): یک نخ جدید برای تابع update_status_to_idle را استارت می‌زند تا بعد از ۵ ثانیه وضعیت فرد را به Idle تغییر دهد.

۱۰.۲) ساختار جیسون پیام‌ها

پیامی که ارسال می‌شود این شکل را دارد:

```json
{
    "ID": 1,
    "name": "Amirhosein",
    "enter_time": "12:34:56",
    "exit_time": null,
    "status": "Success",
    "elapsed_time": 0
}
```

و Subscriber در ESP32 از طریق تابع کال‌بک callback()، آن را deserialize می‌کند و محتوایش را استخراج می‌نماید.

بخش یازدهم: عیب‌یابی و تجربه‌های کلیدی

طی این پروژه، مشکلات و تجربه‌های مختلفی به دست آمد که در ادامه به برخی اشاره می‌کنیم:

خطاهای پورت و شبکه در Localhost: گاهی فایروال سیستم اجازه نمی‌دهد پورت ۱۸۸۳ به درستی باز شود. استفاده از یک بروکر آنلاین مثل emqx که رایگان و تست‌شده است، کار را ساده‌تر می‌کند.

مشکلات ناسازگاری نسخه‌های کتابخانه: یکی از دلایل مهم در عدم موفقیت اجرای کدها، به‌روز نبودن یا ناسازگاری نسخهٔ کتابخانهٔ LVGL یا نیاز به تنظیمات خاص برای تلفیق با Bitluni بود.

اهمیت ترتیب بارگذاری صفحه: در LVGL ضروری است که حتماً مرحلهٔ ایجاد و لود شدن صفحات کامل شود تا بتوان اشیای گرافیکی (ویجت‌ها) را روی آنها مدیریت کرد.

کاسته شدن از کیفیت گرافیکی در Bitluni: نسبت به کتابخانه‌های دیگر، Bitluni گرافیک ساده‌تری دارد. اما چون هدف اصلی این پروژه بیشتر نمایش متن و جدول است، این موضوع مشکل اساسی محسوب نمی‌شود.

جمع‌بندی نهایی و نتیجه‌گیری

در این گزارش، گام به گام پروژهٔ حاضر را بررسی کردیم. در ابتدا با ارسال و دریافت پیام‌های MQTT توسط Publisher و Subscriber به زبان پایتون آشنا شدیم. سپس مشکلاتی که با Localhost وجود داشت را شناسایی و برای ساده‌تر شدن کار، به سراغ یک بروکر آنلاین (emqx) رفتیم. کد پایتون ما ضمن ارسال پیام‌های نمونه، وظیفهٔ ذخیره در SQLite را نیز بر عهده داشت. همچنین تابعی برای تغییر وضعیت افراد به Idle پس از مدتی عدم فعالیت در نظر گرفته شد.

در سمت میکروکنترلر ESP32، ابتدا با استفاده از کتابخانهٔ PubSubClient در Arduino IDE، کد Subscriber را پیاده‌سازی کردیم. به محض دریافت پیام جدید، وضعیت را بررسی کرده و بسته به موفقیت‌آمیز بودن یا نبودن حضور، صفحهٔ گرافیکی مربوطه را نمایش می‌دهیم. برای ایجاد سیگنال VGA از کتابخانهٔ Bitluni استفاده شد و سپس برای طراحی رابط گرافیکی زیباتر به سراغ LVGL (نسخهٔ ۹) رفتیم که ابتدا ناکام ماند. پس از تلاش با کتابخانه‌های دیگر مانند fabgl و TFT_eSPI و مواجهه با مشکلات مختلف، در نهایت LVGL نسخهٔ ۸.۳.۰ و Bitluni پاسخگوی نیاز ما شد.

صفحهٔ IDLE، یک جدول از اطلاعات ده فرد آخر را نشان می‌دهد. درصورت ورود فردی که در دیتابیس ثبت باشد (و پیام آن "Success" باشد)، صفحهٔ موفقیت به‌مدت چند ثانیه نمایش داده می‌شود و سپس خودکار به IDLE بازمی‌گردد. در صورت دریافت پیام "Failed"، صفحهٔ مشکی رنگ خطا را برای اطلاع سریع نمایش می‌دهیم.

تمامی این زنجیره بر بستر پروتکل MQTT کار می‌کند و می‌تواند برای سامانه‌های حضور و غیاب یا هر پروژهٔ دیگری با نیاز به تبادل پیام سبک و کم‌حجم به کار رود و آنچه به‌دست آمده، یک نمونهٔ کامل و کاربردی است که تمامی بخش‌ها از دریافت پیام MQTT گرفته تا نمایش نتیجه روی مانیتور VGA را پوشش می‌دهد.

این کد داخل ریپازیتوری گیت‌هاب نیز موجود می‌باشد.