Software Technology for Internet of Things

Portfolio #2: Temperature Publication

Ignatios Mantanis ¡igman23@student.sdu.dk¿

May 13, 2024

1 Introduction

This project outlines the implementation of a temperature monitoring system using an ESP32-DevKitC device and MQTT broker for data publication. The project uses an LMT86LPM analog temperature sensor, and the ESP32's ADC capabilities to measure and convert analog temperature data to digital format. This data is then sent through Wi-Fi using MQTT protocol for real-time temperature monitoring.

2 System Initialization

The initialization of the ESP32 system begins with the setup of Non-Volatile Storage (NVS) which securely stores essential data such as network credentials and configuration settings. Following NVS setup, the system connects to a Wi-Fi network using SSID and password retrieved dynamically from the Kconfig.projbuild file. The MQTT client is configured to connect to a specified broker, with all necessary parameters, including URI and communication topics, also managed dynamically through the same configuration file.

3 Analog to Temperature Conversion

The process of converting the analog output from the LMT86 temperature sensor into a digital temperature reading begins with the ADC (Analog-to-Digital Converter) sampling the voltage output from the sensor. This voltage is proportional to the ambient temperature, with the relationship defined by the sensor's specifications.

Once the voltage is sampled, it is converted into a temperature using the formula:

$$T = \frac{(V_0 - V)}{TC} + T_{\text{ref}}$$

where:

- V_0 is the baseline voltage at zero degrees Celsius, determined during sensor calibration.
- V is the sampled voltage from the sensor.
- TC is the temperature coefficient, indicating the change in voltage per degree Celsius.
- T_{ref} is the reference temperature at which the calibration was done.

4 Event Handling

The system utilizes event-driven programming to manage Wi-Fi and MQTT events effectively, ensuring robust connectivity and responsive interaction with the MQTT broker. This includes:

- **Wi-Fi Events:** These events include connection establishment and disconnection events that dynamically adjust the system's network connectivity status.
- **MQTT Events:** The system manages a range of MQTT events such as:
  - *Subscription:* Confirming successful subscription to MQTT topics.

- *Command Reception:* Handling incoming MQTT commands that specify the number of temperature measurements and their frequency.
- *Response Handling:* Handling temperature measurements and sending the results back to the MQTT broker as per the received commands.

5 Publication Period

The publication period balances real-time updates with network efficiency. The chosen interval minimizes drift by using the ESP32's precise timers and FreeRTOS's mechanisms. These elements ensure consistent timing between updates, maintaining stable and reliable temperature monitoring without the risk of timing errors over extended periods.

6 Conclusion

This system demonstrates effective use of the ESP32's capabilities and the MQTT protocol for real-time environmental monitoring. The event-driven architecture ensures that the system remains responsive and efficient, making it suitable for various IoT applications.

A Appendix

Kconfig.projbuild

```
menu "Project Configuration"

config WIFI_SSID
    string "WiFi SSID"
    default "Igko"

config WIFI_PASSWORD
    string "WiFi Password"
    default "11111111"

config MQTT_BROKER_URI
    string "MQTT Broker URI"
    default "mqtt://broker.hivemq.com"

config MQTT_COMMAND_TOPIC
    string "MQTT Command Topic"
    default "org/sdu/2024/iot/ignatiosmantanis/command"

config MQTT_RESPONSE_TOPIC
    string "MQTT Response Topic"
    default "org/sdu/2024/iot/ignatiosmantanis/response"

endmenu
```

Setup for Non-Volatile Storage

```
esp_err_t ret = nvs_flash_init();
if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
    nvs_flash_erase();
    ret = nvs_flash_init();
}
ESP_ERROR_CHECK(ret);
```

ADC Initialization and Configuration

```
void configure_adc() {
    adc_oneshot_unit_init_cfg_t init_cfg = {
        .unit_id = ADC_UNIT_1,
    };
    ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_cfg, &adc_handle));
    adc_oneshot_chan_cfg_t chan_cfg = {
        .bitwidth = ADC_BITWIDTH_12,
        .atten = ADC_ATTEN_DB_12,
    };
```

```
    ESP_ERROR_CHECK(adc_oneshot_config_channel(adc_handle, ADC_CHANNEL_6, &chan_cfg));
    adc_cali_line_fitting_config_t cali_cfg = {
        .unit_id = ADC_UNIT_1,
        .atten = ADC_ATTEN_DB_12,
        .bitwidth = ADC_BITWIDTH_12,
    };
    ESP_ERROR_CHECK(adc_cali_create_scheme_line_fitting(&cali_cfg, &cali_handle));
}
```

Wi-Fi Initialization and Configuration

```
static void wifi_init_sta(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));

    wifi_config_t wifi_config = {
        .sta = {
            .ssid = CONFIG_WIFI_SSID,
            .password = CONFIG_WIFI_PASSWORD,
            .threshold.authmode = WIFI_AUTH_WPA2_PSK,
        },
    };

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

Temperature Reading and Conversion

```
float read_temperature() {
    int raw = 0;
    ESP_ERROR_CHECK(adc_oneshot_read(adc_handle, ADC_CHANNEL_6, &raw));

    int voltage = 0;
    ESP_ERROR_CHECK(adc_cali_raw_to_voltage(cali_handle, raw, &voltage));

    float temperature_c = (LMT86_V0_MV - voltage) / LMT86_TC_MV_PER_C + LMT86_REF_TEMP_C;
    return temperature_c;
}
```

Temperature Measurements

```
static void start_temperature_measurements(int count, int interval) {
    if (count == 3) {
```

```
        start_time = (unsigned long)(esp_timer_get_time() / 1000);
    }
    ESP_LOGI(TAG, "Measurement count: %d", count);
    float temperature = read_temperature();
    unsigned long expected_uptime = start_time + (3 - count) * interval;
    char response[100];
    snprintf(response, sizeof(response), "%d,%.2f,%lu", count - 1, temperature, expected_uptime);
    ESP_LOGI(TAG, "Publishing temperature: %s", response);
    esp_mqtt_client_publish(mqtt_client, CONFIG_MQTT_RESPONSE_TOPIC, response, 0, 1, 0);
    vTaskDelay(pdMS_TO_TICKS(interval));
    if (count == 1) {
        ESP_LOGI(TAG, "No more measurements needed.");
        vTaskDelay(pdMS_TO_TICKS(5000));
        return;
    }
    start_temperature_measurements(count - 1, interval);
}
```

Wi-Fi Event Handling

```
static void wifi_event_handler(void* arg, esp_event_base_t event_base,
                               int32_t event_id, void* event_data) {
    ESP_LOGI("wifi_event_handler", "Event received: Base=%s, ID=%" PRIi32,
             event_base == WIFI_EVENT ? "WIFI_EVENT" : "IP_EVENT", event_id);
    if (strcmp(event_base, WIFI_EVENT) == 0) {
        if (event_id == WIFI_EVENT_STA_START) {
            esp_wifi_connect();
            ESP_LOGI("wifi_event_handler", "WiFi started, trying to connect...");
        } else if (event_id == WIFI_EVENT_STA_DISCONNECTED) {
            ESP_LOGI("wifi_event_handler", "Disconnected from WiFi, trying to reconnect...");
            esp_wifi_connect();
            xEventGroupClearBits(s_wifi_event_group, CONNECTED_BIT);
        }
    } else if (strcmp(event_base, IP_EVENT) == 0) {
        if (event_id == IP_EVENT_STA_GOT_IP) {
            ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
            esp_ip4addr_ntoa(&event->ip_info.ip, addr_str, IP4_ADDR_STRLEN);
            ESP_LOGI("wifi_event_handler", "Got IP: %s", addr_str);
            xEventGroupSetBits(s_wifi_event_group, CONNECTED_BIT);
            retry_count = 0;
        }
    }
}
```

MQTT Event Handling

```
static void mqtt_event_handler(void *handler_args, esp_event_base_t base,
                               int32_t event_id, void *event_data) {
    esp_mqtt_event_handle_t event = event_data;
```

```
    switch (event_id) {
        case MQTT_EVENT_CONNECTED:
            ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
            esp_mqtt_client_subscribe(mqtt_client, CONFIG_MQTT_COMMAND_TOPIC, 0);
            break;
        case MQTT_EVENT_DISCONNECTED:
            ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
            break;
        case MQTT_EVENT_SUBSCRIBED:
            ESP_LOGI(TAG, "MQTT_EVENT_SUBSCRIBED");
            break;
        case MQTT_EVENT_DATA:
            ESP_LOGI(TAG, "MQTT_EVENT_DATA: Topic=%.*s, Data=%.*s",
                     event->topic_len, event->topic, event->data_len, event->data);
            char* payload = event->data;
            int measurements, interval;
            if (sscanf(payload, "measure:%d,%d", &measurements, &interval) == 2) {
                ESP_LOGI(TAG, "Received measurement command: %d measurements, %d ms interval",
                         measurements, interval);
                start_temperature_measurements(measurements, interval);
            } else {
                ESP_LOGE(TAG, "Received malformed command");
            }
            break;
    }
}
```