Universidad Autónoma de Baja California

Facultad de Ciencias Químicas e Ingeniería

INTERNET DE LAS COSAS

TALLER 7: Notificaciones

Docente: Aguilar Noriega, Leocundo

Alumno: Gómez Cárdenas, Emmanuel Alberto

Matricula: 01261509

Objetivo

- Una vez que el ESP32 detecte algún evento en específico, se debe enviar una notificación a algún usuario, ya sea por correo electrónico o por **SMS**.

Desarrollo

1ra Parte:

- **Incluya al evento detectado el envió de una notificación a un correo electrónico, para ello investigue la forma GRATIS mediante la compañía IFTTT (If This Then That).**

Al IFTTT ya no ofrecer servicios gratis, nos hemos saltado este paso.

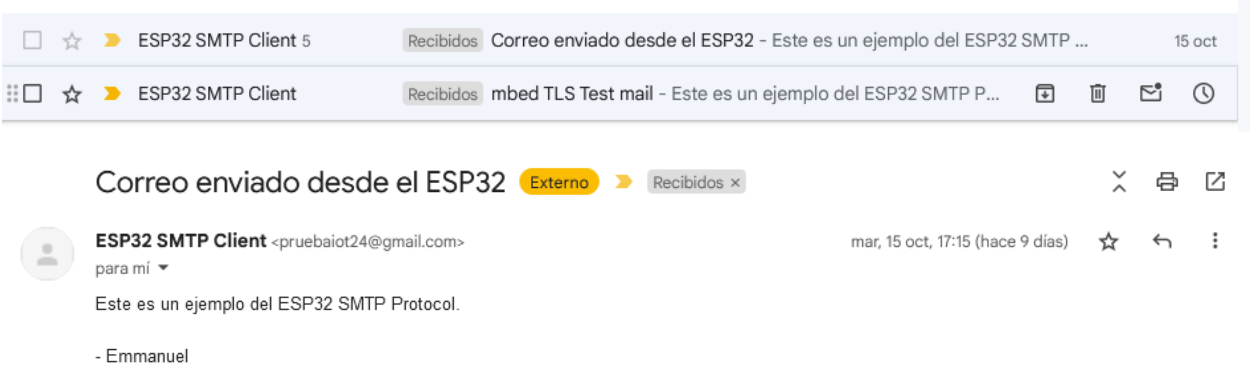
- **Como actividad adicional investigue otras alternativas de envío de correo electrónico de forma directo (sin uso de un tercero).**

Utilizamos como base la plantilla de ejemplo del uso de un cliente SMTP (Simple Mail Transfer Protocol o Protocolo de Transferencia de Correo Simple) y lo modificamos para poder transmitir nuestros propios mensajes.

Creamos un correo para poder probar la funcionalidad, creamos y agregamos las credenciales al código para poder mandar el correo

Enviados desde el correo creado

Recibido en el correo escolar



2da Parte:

1. **Integre a su aplicación del ESP32 la detección de un evento, en este caso puede ser un interruptor, un valor del umbral del ADC.**

- He optado por utilizar un interruptor en el GPIO 23.
- El botón es configurado con pull-up y NEGEDGE para el tipo de interrupción a detectar, por lo que solo será detectado al presionar el botón (flanco de positivo a negativo).
- Para lidiar con el "debounce" que puede ocurrir, simplemente se toma el tiempo en el que fue presionado el botón y se ignoran las siguientes interrupciones por 10 segundos.

2. **Una vez que el evento puede ser detectado, incluya el envío de un comando con la información necesaria para entregar un mensaje SMS según el formato siguiente:**

UABC:<usuario>:<operación>:<recurso>:<no_tel>:<mensaje>

- **<usuario>:** Campo en la trama del protocolo que funciona para identificar el ESP32 del usuario y está conformado por solo 3 letras (En mi caso EGC).
- **<operación>:** En este caso de solicitud, la escritura (**M**) de un mensaje a enviar.
- **<recurso>:** El recurso se refiere al elemento a operar y para este caso será el servidor (**S**) envía le mensaje SMS (Short Message Service).
- **<no_tel>:** Número telefónico al que se le enviará el mensaje SMS (debe ser un teléfono móvil con el servicio SMS activado.
- **<mensaje>:** Se refiere a un pequeño texto correspondiente a la notificación a enviar.

3. El comando debe ser enviado al servidor mediante la URL iot-uabc.site usando el protocolo TCP y puerto 8266.
4. Realice las pruebas necesarias para verificar que opera correctamente.

Los mensajes llegan correctamente después de detectar la interrupción, ignorando cualquier detección en los próximos 10 segundos.

Correción de clase

Importante: Para ejemplo de detección de evento

a) Deberá incluir la detección de flanco (subida o bajada) para el envío de mensajes SMS.
La detección del flanco es mediante muestreo -- no usar el mecanismo de interrupciones del ESP32.
b) Deberá manejar un margen mínimo de 1 minuto para enviar la detección del mismo evento. Es decir, nunca deberá existir menos de un minuto entre mensajes enviados.

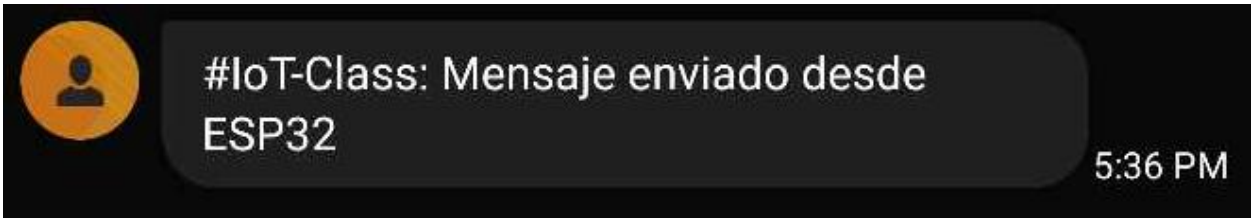
```
I (6350) esp_netif_handlers: sta ip: 192.168.1.70, mask: 255.255.255.0, gw: 192.168.1.254
I (6890) main_task: Returned from app_main()
I (6890) Prototipo en Red Local: Socket created, connecting to 82.180.173.228:8266
I (7040) Prototipo en Red Local: Successfully connected
I (7040) Prototipo en Red Local: Sending login message...
I (8060) wifi:<ba-add>idx:0 (ifx:0, f4:e4:51:be:af:6d), tid:0, ssn:5, winSize:64
' (8480) Prototipo en Red Local: RECEIVED FROM 82.180.173.228: 'ACK

I (10740) Prototipo en Red Local: Boton presionado, enviando mensaje

' (11960) Prototipo en Red Local: RECEIVED FROM 82.180.173.228: 'ACK

I (20360) wifi:<ba-add>idx:1 (ifx:0, f4:e4:51:be:af:6d), tid:1, ssn:5, winSize:64
I (22050) Prototipo en Red Local: Sending keep alive message...
' (22810) Prototipo en Red Local: RECEIVED FROM 82.180.173.228: 'ACK
```

```
[Thu Oct 24 17:36:29 2024] Rx: UABC:EGC:L:S:Log in
[Thu Oct 24 17:36:29 2024] Logged:EGC --> ACK to client: ('148.231.168.195', 56644)
[Thu Oct 24 17:36:33 2024] Rx: UABC:EGC:M:S:6656560351:Mensaje enviado desde ESP32
[Thu Oct 24 17:36:43 2024] Rx: UABC:EGC:K:S:Keep alive
[Thu Oct 24 17:36:44 2024] KeepAlive --> ACK to client: ('148.231.168.195', 56644)
[Thu Oct 24 17:36:58 2024] Rx: UABC:EGC:K:S:Keep alive
[Thu Oct 24 17:36:59 2024] KeepAlive --> ACK to client: ('148.231.168.195', 56644)
```



Conclusiones y Comentarios

La notificación es esencial para no pasar por alto cuando un evento en específico es detectado, ya que, dependiendo de la importancia, puede afectar negativamente al dispositivo o su ejecución.

Código

El código puede ser encontrado en el [repositorio de GitHub]

```
#include <string.h>

#include "driver/gpio.h"
#include "driver/ledc.h"
#include "esp_adc/adc_oneshot.h"
#include "esp_event.h"
#include "esp_log.h"
#include "esp_netif.h"
#include "esp_system.h"
#include "esp_wifi.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "lwip/err.h"
#include "lwip/sockets.h"
#include "nvs_flash.h"

// Constants
#define SSID "ESP_NET"
#define PASS "ESP_NET_IOT"
#define LED GPIO_NUM_2
#define ADC_SELECTED GPIO_NUM_34
#define ADC1_CHANNEL ADC_CHANNEL_6
#define ADC_WIDTH ADC_BITWIDTH_12
#define ADC_ATTEN ADC_ATTEN_DB_0
#define WIFI_RETRY_MAX 20
#define NACK_RESPONSE "NACK"
#define ACK_RESPONSE "ACK"
#define WRITE_INSTRUCTION 'W'
#define READ_INSTRUCTION 'R'
#define LED_ELEMENT 'L'
#define ADC_ELEMENT 'A'
#define PWM_ELEMENT 'P'
#define BUFFER_SIZE 128
#define PORT 8266
// #define HOST_IP_ADDR "192.168.1.69"  // Local IP
#define HOST_IP_ADDR "82.180.173.228"  // IoT Server
#define LED_PWM GPIO_NUM_21
#define LEDC_TIMER LEDC_TIMER_0
#define LEDC_CHANNEL 0
#define LEDC_MODE 0
#define LEDC_OUTPUT_IO GPIO_NUM_15
#define LEDC_DUTY_RESOLUTION LEDC_TIMER_10_BIT
```

```
#define TWO_TO_THE_POWER_OF_10 1024  // Manually calculated to avoid math.h
#define LEDC_FREQUENCY 400

#define BUTTON_SEND_MESSAGE GPIO_NUM_23
#define BUTTON_BOUNCE_TIME 150
#define SECOND_IN_MILLIS 1000
#define SEND_MESSAGE_DELAY_TIME 60 * SECOND_IN_MILLIS
#define RELEASED 0
#define PRESSED 1
#define MINIMUM_DELAY_MS 10
#define FREED 1
#define PUSHED 0

static const char *TAG = "Prototipo en Red Local";
static const char *log_in = "UABC:EGC:L:S:Log in";
static const char *keep_alive = "UABC:EGC:K:S:Keep alive";
static const char *message = "UABC:EGC:M:S:6656560351:Mensaje enviado desde
ESP32";
int32_t lastStateChange = 0;
TaskHandle_t keep_alive_task_handle = NULL;

// Global variables
bool wifi_connected = false;
bool logged_in = false;
int retry_num = 0;
int sock = 0;
bool messageSent = 0;
static adc_oneshot_unit_handle_t adc1_handle;
```

```
void gpio_init() {
   gpio_config_t io_conf;
   io_conf.intr_type = GPIO_INTR_DISABLE;
   io_conf.mode = GPIO_MODE_INPUT_OUTPUT;
   io_conf.pin_bit_mask = (1ULL << LED);
   io_conf.pull_down_en = GPIO_PULLDOWN_ENABLE;
   io_conf.pull_up_en = GPIO_PULLUP_DISABLE;
   gpio_config(&io_conf);

   io_conf.intr_type = GPIO_INTR_DISABLE;
   io_conf.mode = GPIO_MODE_INPUT;
   io_conf.pin_bit_mask = (1ULL << ADC_SELECTED);
   io_conf.pull_down_en = GPIO_PULLDOWN_DISABLE;
   io_conf.pull_up_en = GPIO_PULLUP_DISABLE;
   gpio_config(&io_conf);

   io_conf.intr_type = GPIO_INTR_NEGEDGE;
   io_conf.mode = GPIO_MODE_INPUT;
   io_conf.pin_bit_mask = (1 << BUTTON_SEND_MESSAGE);
   io_conf.pull_down_en = 0;
   io_conf.pull_up_en = 1;
   gpio_config(&io_conf);
}

void adc_init() {
   adc_oneshot_unit_init_cfg_t adc_config = {
       .unit_id = ADC_UNIT_1,
   };

   if (adc_oneshot_new_unit(&adc_config, &adc1_handle) == ESP_FAIL) {
      ESP_LOGE(TAG, "Failed to initialize ADC unit");
      return;
   }

   adc_oneshot_chan_cfg_t adc_channel_config = {
       .atten = ADC_ATTEN,
       .bitwidth = ADC_WIDTH,
   };
```

```
  if (adc_oneshot_config_channel(adc1_handle, ADC1_CHANNEL,
                                 &adc_channel_config) == ESP_FAIL) {
      ESP_LOGE(TAG, "Failed to configure ADC channel");
      adc_oneshot_del_unit(adc1_handle);
      return;
   }
   ESP_LOGI(TAG, "ADC initialized");
}

void ledc_init() {
   ledc_timer_config_t ledc_timer = {
       .duty_resolution = LEDC_TIMER_10_BIT,
       .freq_hz = LEDC_FREQUENCY,
       .speed_mode = LEDC_HIGH_SPEED_MODE,
       .timer_num = LEDC_TIMER_0,
       .clk_cfg = LEDC_AUTO_CLK,
   };
   ESP_ERROR_CHECK(ledc_timer_config(&ledc_timer));

   ledc_channel_config_t ledc_channel = {
       .speed_mode = LEDC_MODE,
       .channel = LEDC_CHANNEL,
       .timer_sel = LEDC_TIMER,
       .intr_type = LEDC_INTR_DISABLE,
       .gpio_num = LEDC_OUTPUT_IO,
       .duty = 0,  // Set duty to 0%
       .hpoint = 0,
   };
   ESP_ERROR_CHECK(ledc_channel_config(&ledc_channel));
}

void set_led(int value) {
   gpio_set_level(LED, value);
   ESP_LOGI(TAG, "LED set to: %d", value);
}

int read_led() {
   int led_state = gpio_get_level(LED);
   ESP_LOGI(TAG, "LED state is: %d", led_state);
   return led_state;
}
```

```
void set_pwm(uint16_t percentage) {
   // Formula for value = (2 ^ LEDC_DUTY_RESOLUTION) * percentage / 100
   int32_t value = (TWO_TO_THE_POWER_OF_10 * percentage) / 100;
   ledc_set_duty(LEDC_MODE, LEDC_CHANNEL, value);
   ledc_update_duty(LEDC_MODE, LEDC_CHANNEL);
   ESP_LOGI(TAG, "PWM LED set to: %%%d, %ld", percentage, value);
}

uint16_t read_pwm() {
   uint16_t pwm_value = ledc_get_duty(LEDC_MODE, LEDC_CHANNEL);
   pwm_value = (pwm_value * 100) / TWO_TO_THE_POWER_OF_10;
   ESP_LOGI(TAG, "PWM LED is: %d", pwm_value);
   return pwm_value;
}

int read_adc_value() {
   int adc_value = 0;
   if (adc_oneshot_read(adc1_handle, ADC1_CHANNEL, &adc_value) == ESP_OK) {
      ESP_LOGI(TAG, "ADC value: %d", adc_value);
      return adc_value;
   }
   ESP_LOGE(TAG, "Failed to read ADC value");
   return ESP_FAIL;
}

void delaySeconds(uint8_t seconds) { vTaskDelay(seconds
                                    * SECOND_IN_MILLIS / portTICK_PERIOD_MS); }

static void wifi_event_handler(void *event_handler_arg,
                               esp_event_base_t  event_base,
                               int32_t event_id, void *event_data) {
```

```
 switch (event_id) {
      case WIFI_EVENT_STA_START:
         ESP_LOGI(TAG, "Wi-Fi starting...");
         retry_num = 0;
         break;
      case WIFI_EVENT_STA_CONNECTED:
         ESP_LOGI(TAG, "Wi-Fi connected");
         break;
      case WIFI_EVENT_STA_DISCONNECTED:
         ESP_LOGE(TAG, "Wi-Fi lost connection");
         if (retry_num < WIFI_RETRY_MAX) {
            esp_wifi_connect();
            retry_num++;
            ESP_LOGE(TAG, "Retrying connection...");
         }
         break;
      case IP_EVENT_STA_GOT_IP:
         wifi_connected = true;
         break;
      default:
         ESP_LOGW(TAG, "Unhandled event ID: %ld", event_id);
         break;
   }
}

void wifi_init() {
   esp_netif_init();
   esp_event_loop_create_default();
   esp_netif_create_default_wifi_sta();

   wifi_init_config_t wifi_initiation = WIFI_INIT_CONFIG_DEFAULT();
   esp_wifi_init(&wifi_initiation);

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

   wifi_config_t wifi_configuration = {.sta = {.ssid = SSID, .password = PASS}};
```

```
    esp_wifi_set_mode(WIFI_MODE_STA);
    esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_configuration);
    esp_wifi_start();

    ESP_LOGI(TAG, "Wi-Fi initialization complete.
                   Attempting to connect to SSID: %s", SSID);
    esp_wifi_connect();
}

int read_element(int element) {
    switch (element) {
        case LED_ELEMENT:
            return read_led();
        case ADC_ELEMENT:
            return read_adc_value();
        case PWM_ELEMENT:
            return read_pwm();
        default:
            return ESP_FAIL;
    }
}

void process_command(const char *command, char *response) {
    const char *prefix = "UABC:EGC:";
    if (strncmp(command, prefix, strlen(prefix)) != 0) {
        snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
        return;
    }

    const char *cmd = command + strlen(prefix);
    char operation;
    char element;
    char value[3] = {0};
    char comment[BUFFER_SIZE] = {0};

    int parsed = sscanf(cmd, "%c:%c:%3[^:]s:%127[^:]s",
                             &operation, &element, value, comment);
    if (parsed <= 2 || parsed > 4) {
        ESP_LOGE(TAG, "Parsed: %d", parsed);
        snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
        return;
    }
```

```
   if (operation == READ_INSTRUCTION) {
      if (parsed == 3) {
         sscanf(cmd, "%c:%c:%127[^:]s", &operation, &element, comment);
      } else {
         char temp[BUFFER_SIZE - sizeof(value)] = {0};
         strcpy(temp, comment);
         snprintf(comment, BUFFER_SIZE, "%s%s", value, temp);
      }
      value[0] = 0;
   }

   switch (operation) {
      case WRITE_INSTRUCTION:
         if (element == LED_ELEMENT && (value[0] == '0' || value[0] == '1')) {
            set_led(value[0] - '0');
            snprintf(response, BUFFER_SIZE, ACK_RESPONSE ":%d", read_led());
         } else if (element == PWM_ELEMENT) {
            set_pwm(atoi(value));
            snprintf(response, BUFFER_SIZE, ACK_RESPONSE ":%d", read_pwm());
         } else {
            if (element == ADC_ELEMENT) ESP_LOGI(TAG, "ADC value is readonly");
            snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
         }
         break;
      case READ_INSTRUCTION:
         int readed_value = read_element(element);
         if (readed_value != ESP_FAIL) {
            snprintf(response, BUFFER_SIZE, ACK_RESPONSE ":%d", readed_value);
         } else {
            snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
         }
         break;

      default:
         snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
   }
}

void keep_alive_task() {
   while (true) {
      delaySeconds(15);
      ESP_LOGI(TAG, "Sending keep alive message...");
      send(sock, keep_alive, strlen(keep_alive), 0);
   }
}
```

```
void tcp_client_task() {
   char rx_buffer[128];
   char host_ip[] = HOST_IP_ADDR;
   int addr_family = 0;
   int ip_protocol = 0;

   while (true) {
      struct sockaddr_in dest_addr;
      inet_pton(AF_INET, host_ip, &dest_addr.sin_addr);
      dest_addr.sin_family = AF_INET;
      dest_addr.sin_port = htons(PORT);
      addr_family = AF_INET;
      ip_protocol = IPPROTO_IP;

      sock = socket(addr_family, SOCK_STREAM, ip_protocol);
      if (sock < 0) {
         ESP_LOGE(TAG, "Unable to create socket: errno %d", errno);
         break;
      }
      ESP_LOGI(TAG, "Socket created, connecting to %s:%d", host_ip, PORT);

      int err = connect(sock, (struct sockaddr *)&dest_addr, sizeof(dest_addr));
      if (err != 0) {
         ESP_LOGE(TAG, "Socket unable to connect: errno %d", errno);
         break;
      }
      ESP_LOGI(TAG, "Successfully connected");

      while (true) {
         err = 0;
         if (logged_in == false) {
            ESP_LOGI(TAG, "Sending login message...");
            err = send(sock, log_in, strlen(log_in), 0);
            if (keep_alive_task_handle != NULL)
               vTaskResume(keep_alive_task_handle);
            else
               xTaskCreate(&keep_alive_task, "keep_alive",
                           4096, NULL, 5, &keep_alive_task_handle);
            logged_in = true;
         }

         if (err < 0) {
            ESP_LOGE(TAG, "Error occurred during sending: errno %d", errno);
            break;
         }
```

```
        int len = recv(sock, rx_buffer, sizeof(rx_buffer) - 1, 0);
        if (len < 0) {
           ESP_LOGE(TAG, "recv failed: errno %d", errno);
           break;
        }

        else {
           rx_buffer[len] = 0;
           if (strstr(rx_buffer, NACK_RESPONSE) == rx_buffer
               || strstr(rx_buffer, ACK_RESPONSE) == rx_buffer) {
              // TODO: Add logic for nack
              ESP_LOGI(TAG, "RECEIVED FROM %s: \'%s\'\n", host_ip, rx_buffer);
           } else {
              ESP_LOGI(TAG, "RECEIVED FROM %s:", host_ip);
              ESP_LOGI(TAG, "\'%s\'\n", rx_buffer);

              char answer[BUFFER_SIZE] = NACK_RESPONSE;  // Default response
              process_command(rx_buffer, answer);
              send(sock, answer, strlen(answer), 0);
              ESP_LOGI(TAG, "SENT %s TO %s\n", answer, host_ip);
           }
        }
     }

     if (sock != -1) {
        ESP_LOGE(TAG, "Shutting down socket and restarting...");
        shutdown(sock, 0);
        close(sock);
     } else if (sock == 0) {
        ESP_LOGE(TAG, "Connection closed by server");
        vTaskSuspend(keep_alive_task_handle);
     }
  }
}
```

```
void send_email_task() {
   bool buttonState = 0;
   lastStateChange = -SEND_MESSAGE_DELAY_TIME;

   while (true) {
      buttonState = gpio_get_level(BUTTON_SEND_MESSAGE);
      int64_t now = xTaskGetTickCount() * portTICK_PERIOD_MS;
      // ESP_LOGI(TAG, "buttonState %s, now %lld", buttonState == FREED ?
"Liberado" : "P", now);

      // ESP_LOGI(TAG, "buttonState %d, now %lld", buttonState, now);
      if (buttonState == FREED) {
         while (gpio_get_level(BUTTON_SEND_MESSAGE) == FREED) {
            // ESP_LOGI(TAG, "Esperando a que el boton sea presionado");
            vTaskDelay(10 / portTICK_PERIOD_MS);
         }
         if (now - lastStateChange > SEND_MESSAGE_DELAY_TIME) {
            ESP_LOGI(TAG, "Boton presionado, enviando mensaje\n");
            send(sock, message, strlen(message), 0);
            lastStateChange = xTaskGetTickCount() * portTICK_PERIOD_MS;
         }
         while (gpio_get_level(BUTTON_SEND_MESSAGE) == PUSHED) {
            // ESP_LOGI(TAG, "Esperando a que el boton sea liberado");
            vTaskDelay(10 / portTICK_PERIOD_MS);
         }
         // ESP_LOGI(TAG, "Se ha soltado el button\n");
      }
      vTaskDelay(10 / portTICK_PERIOD_MS);
   }
}
```

```
void app_main(void) {
   ESP_ERROR_CHECK(nvs_flash_init());
   wifi_init();
   gpio_init();
   adc_init();
   ledc_init();
   while (!wifi_connected) {
      if (retry_num == WIFI_RETRY_MAX) {
         ESP_LOGE(TAG,
                  "Connection failed. Maximum retries reached, it is likely "
                  "that the SSID cannot be found.");
         return;
      }
      ESP_LOGI(TAG, "Waiting for WIFI
                     before starting TCP server connection...\n");
      fflush(stdout);
      vTaskDelay(1000 / portTICK_PERIOD_MS);
   }
   xTaskCreate(tcp_client_task, "tcp_client", 4096, NULL, 5, NULL);
   xTaskCreate(send_email_task, "sendEmail", 2048, NULL, 1, NULL);
}
```