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.

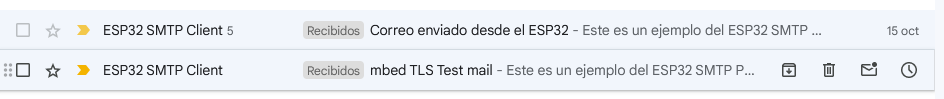
* **Como actividad adicional investigue otras alternativas de envío de correo electrónico de forma directo (sin uso de un tercero).**

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  


A screenshot of a computer

Description automatically generated

**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.

1. **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.

1. El comando debe ser enviado al servidor mediante la URL [iot-uabc.site](http://iot-uabc.site) usando el protocolo TCP y puerto 8266.
2. 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

1. 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.

1. 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.

A computer screen shot of a program

Description automatically generated

A screen shot of a computer

Description automatically generated

A black rectangular sign with white text

Description automatically generated

## 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](https://github.com/AlbGmx/IoT_UABC/tree/main/T7-Notifications)

#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*);

}