Universidad Autónoma de Baja California

Facultad de Ciencias Químicas e Ingeniería

INTERNET DE LAS COSAS

TALLER 4: Prototipo IoT

Docente: Aguilar Noriega, Leocundo

Alumno: Gómez Cárdenas, Emmanuel Alberto

Matricula: 01261509

Objetivo

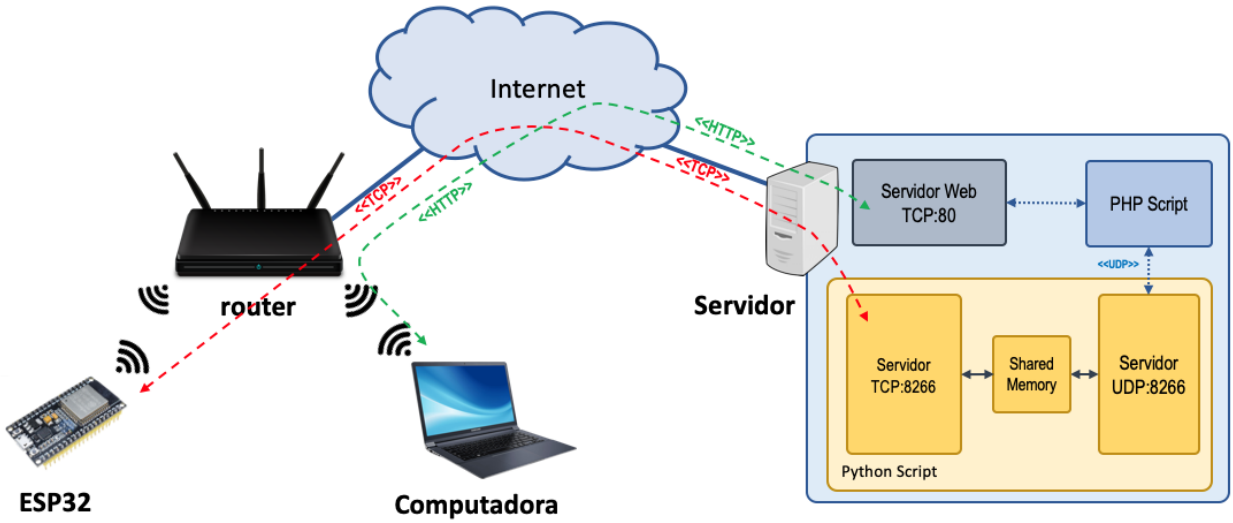
El objetivo del tallller-lab es hacer que un prototipo de dispositivo IoT basado en un ESP32 que opera junto con una infraestructura IoT en internet. La idea es que el ESP32 se conecte a la infraestructura IoT y entonces pueda reportar el estado de sus recursos, además de poder modificar su estado de forma remota mediante una aplicación web simple ya existente.

Equipo

Computadora personal con conexión a internet.

ESP32 Devkit1

Desarrollo



Usando la plataforma ESP-IDF realizar los cambios necesarios al prototipo del taller anterior (Prototipo en Red Local) para lograr la nueva funcionalidad.

1. El dispositivo ESP32 una vez que se enciende deberá conectarse al servidor de la infraestructura IoT en la dirección IP correspondiente a **iot-uabc.site,** usando el protocolo TCP con puerto 8266. El comando a enviar para la solicitud de acceso tiene el siguiente formato:

UABC:<usuario><operación>:<recurso>:<comentario>

donde:
<usuario>: Es un nuevo campo en la trama del protocolo que funciona para identificar el ESP32 del usuario y está conformado por solo 3 letras (Uds. ya tiene su key de usuario)

<operación>: En este caso de solicitud de acceso la operación estada dada por la letra **L** (login).

<recurso>: El recurso se refiere al elemento a operar y para este caso será el recurso asignado a la letra **S** que es el servidor.

<comentario>: Se refiere a un pequeño texto relacionado a la operación y es opcional.

Ejemplo del comando:

UABC:LAN:L:S: Login el server
El servidor responderá con un **ACK** si se logró el acceso, de lo contrario se responde con **NACK** indicando que no se reconoce la acción solicitada.

2. Una vez que el ESP32 ha logrado el acceso al servidor se deberá estar enviando cada 10 segundo el siguiente comando con el fin de estar informando al servidor que el ESP32 está activo y conectado. Importante, esto se hace sobre la misma conexión ya establecida.

UABC:<usuario>**:K:S: Keep-Alive al server**

El comando usa la letra **K** como operación Keep-Alive sobre el recurso **S** que es el servidor.

El servidor responderá con un **ACK** si se logró la operación, de lo contrario se responde con **NACK** indicando que no se reconoce la acción solicitada. Si por algún motivo el ESP32 deja de enviar este comando el servidor asumirá que el ESP32 ha quedado fuera de linea, pero si el ESP32 retoma a la actividad se deberá solicitar acceso (login) para luego enviar cada 10 segundo este comando.

3. Una vez que el ESP32 logra el punto 1 y 2 deberá estar listo para recibir algún comando de operación sobre los recursos los cuales son el LED y el ADC tal como se manejó en el taller anterior. Sin embargo, ahora la trama del comando debe incluir el usuario como se describió en el punto 1.

   Ahora, por ejemplo el comando para encender el LED es:

UABC:<usuario>**:W:L:1: Encender LED**

y si se logró encender el LED la contestación es : **ACK:1**

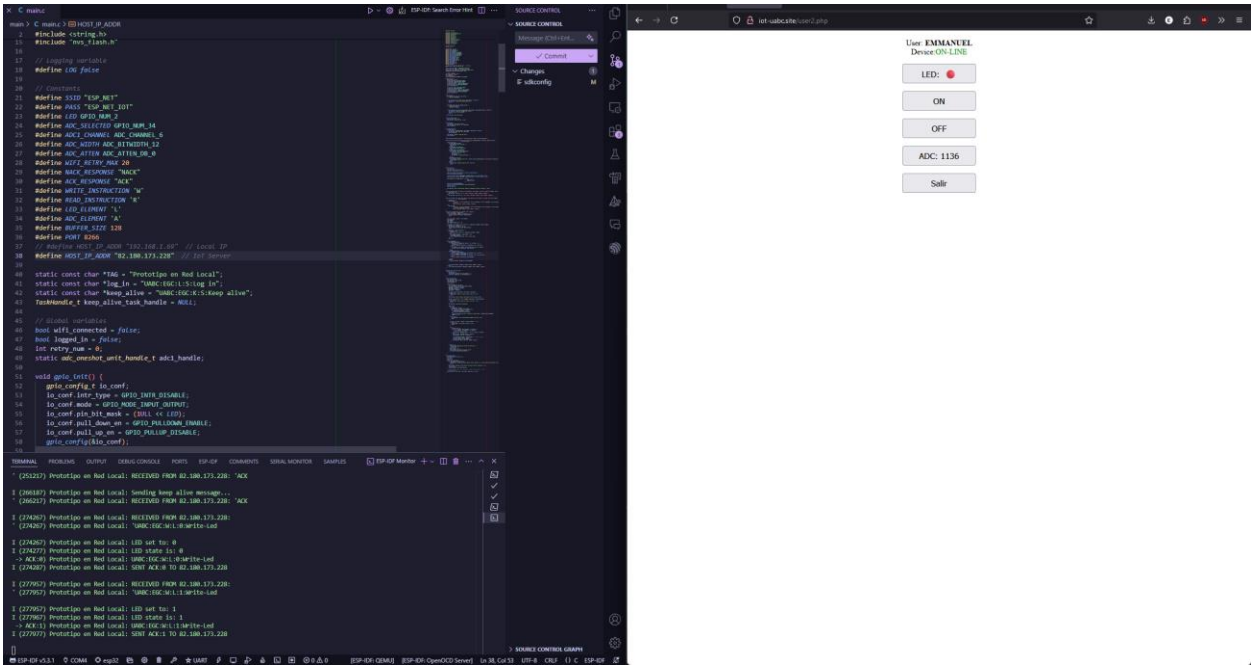
Para leer el ADC el comando es:

UABC:<usuario>**:R:A: Lee ADC**

y la contestación es: **ACK:<valor del ADC>**

4. Los comandos serán enviados mediante una simple página web en el sito http://iot-uabc.site al que se debe acceder con su usuario y contraseña (estos ya se les proporcionó a cada uno de Uds.)

5. Realice las pruebas necesarias para verificar que los cambios del ESP32 operan correctamente según la funcionalidad correspondiente.



Video del programa en funcionamiento puede encontrarse en DRIVE

Conclusiones y Comentarios

El correcto entendimiento de los protocolos TCP y UDP es fundamental al trabajar en IoT, ya que permiten garantizar comunicaciones eficientes, confiables y adaptables a las diferentes necesidades del sistema, optimizando el uso de redes para asegurar el intercambio de datos de manera efectiva.

Dificultades en el Desarrollo

La configuración e implementación del protocolo TCP fue una de las principales dificultades, principalmente porque no había trabajado directamente con dicho protocolo. No obstante, siguiendo el ejemplo proporcionado en la documentación, se logró una implementación funcional, aunque no del todo ideal, ya que carece de ciertos manejos.

Código

El código puede ser encontrado en el repositorio de Github

```
#include <string.h>
#include "driver/gpio.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"

// Logging variable
#define LOG false

// 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 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

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";
TaskHandle_t keep_alive_task_handle = NULL;

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

```c
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);
}

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 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;
}
```

```
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 * 1000 / 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:
         ESP_LOGI(TAG, "Connected with IP %s",
                  ip4addr_ntoa(&((ip_event_got_ip_t *)event_data)->ip_info.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_wifi_connect();

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

void print_command(const char *prefix, char operation, char element,
                   int value, const char *comment, char *response) {
   if (operation == 'R')
      ESP_LOGI(TAG, "%s%c:%c:%s -> %s", prefix, operation, element,
                                        comment, response);
   else
      ESP_LOGI(TAG, "%s%c:%c:%c:%s -> %s", prefix, operation, element,
                                           value, comment, response);
}
```

```
void print_command_parsed(const char *prefix, char operation, char element, int
value, const char *comment,
                          char *response) {
   if (operation == 'R')
      ESP_LOGI(TAG,
               "\n\tPrefix: \t\"%s\"
                \n\tOperation: \t\"%c\"
                \n\tElement: \t\"%c\"
                \n\tComment: \t\"%s\"\n
                \n\tResponse: "
               "\t\"%s\"",
               prefix, operation, element, comment, response);
   else
      ESP_LOGI(TAG,
               "\n\tPrefix: \t\"%s\"
                \n\tOperation: \t\"%c\"
                \n\t\tElement: \t\"%c\"
                \n\tValue: \t\"%c\"
                \n\tComment: \t\"%s\"\n
                \n\tResponse: \t\"%s\"",
               prefix, operation, element, value, comment, response);
}

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;
   char comment[BUFFER_SIZE] = {0};

   int parsed = sscanf(cmd, "%c:%c:%c:%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, "%c%s", value, temp);
      }
      value = -1;
   }
```

```
  switch (operation) {
      case WRITE_INSTRUCTION:
         if (element == LED_ELEMENT && (value == '0' || value == '1')) {
            set_led(value - '0');
            snprintf(response, BUFFER_SIZE, ACK_RESPONSE ":%d", read_led());
         } else {
            if (element == ADC_ELEMENT) ESP_LOGI(TAG, "ADC value is readonly");
            snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
         }
         break;
      case READ_INSTRUCTION:
         if (element == LED_ELEMENT) {
            snprintf(response, BUFFER_SIZE, ACK_RESPONSE ":%d", read_led());
         } else if (element == ADC_ELEMENT) {
            snprintf(response, BUFFER_SIZE, ACK_RESPONSE ":%d",
read_adc_value());
         } else {
            snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
         }
         break;
      default: {
         snprintf(response, BUFFER_SIZE, NACK_RESPONSE);
      }
   }

      print_command(prefix, operation, element, value, comment, response);
   if (LOG) {
      print_command_parsed(prefix, operation, element, value, comment, response);
   }
}

void keep_alive_task(int *sock) {
   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;

      int 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, &sock, 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 app_main(void) {
   ESP_ERROR_CHECK(nvs_flash_init());
   wifi_init();
   gpio_init();
   adc_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(udp_server_task, "udp_server", 4096, (void *)AF_INET, 5, NULL);
   // xEventGroupSync
   xTaskCreate(tcp_client_task, "tcp_client", 4096, NULL, 5, NULL);
}
```