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

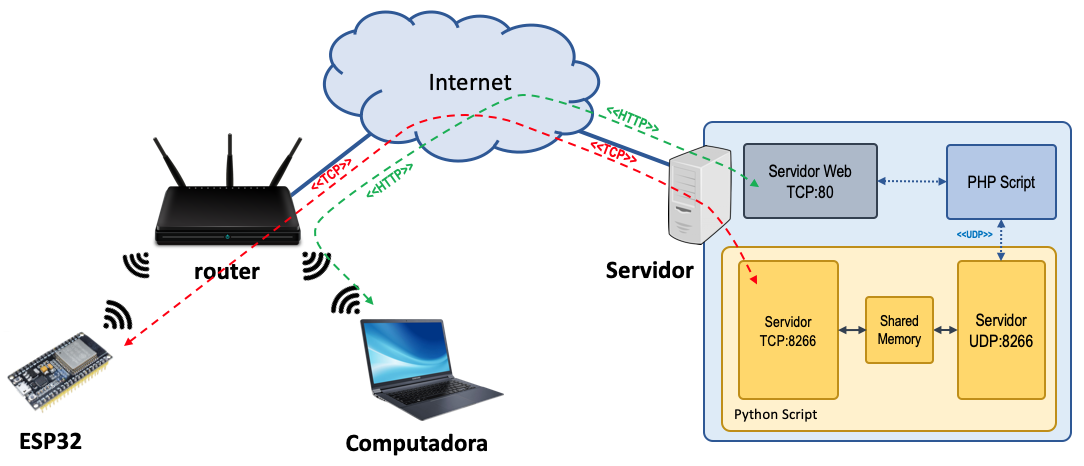
El objetivo del talller-lab es hacer que un prototipo de dispositivo IoT basado en un ESP32 que opera junto con una infraestructura IoT en internet. La idea es que el ESP32 se conecte a la infraestructura IoT y entonces pueda reportar el estado de sus recursos, además de poder modificar su estado de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

1. 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.

1. 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>**

1. 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.)
2. Realice las pruebas necesarias para verificar que los cambios del ESP32 operan correctamente según la funcionalidad correspondiente.

A screenshot of a computer

Description automatically generated

Video del programa en funcionamiento puede encontrarse en [DRIVE](https://drive.google.com/file/d/1sA3NU824CWfVU5HIyYL2S4YwoGg5nUPp/view?usp=drive_link)

## 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](https://github.com/AlbGmx/IoT_UABC/tree/c60d8328fa7c454afa19bd7fa69ef8e323592ccb/T4_Prototipo-IoT)

#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;

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*);

}