

Sistemas Embebidos

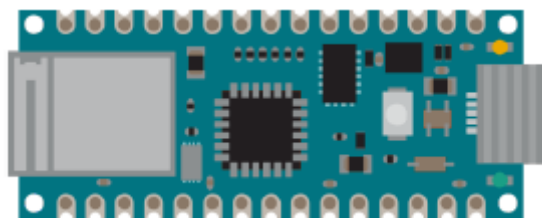
CONECTIVIDAD, TIEMPO REAL E INTERRUPCIONES

Practica 7

Introducción

En esta práctica se centra en el estudio y puesta en práctica de los microcontroladores utilizando dos dispositivos distintos: el Arduino Nano IoT 33 y el ESP32 en su versión Lolin32. El Arduino Nano IoT 33 es un microcontrolador diseñado especialmente para aplicaciones de Internet de las Cosas, mientras que el ESP32 ofrece potentes capacidades de procesamiento y conectividad Wi-Fi y Bluetooth. Hasta ahora, se ha utilizado el Arduino Nano IoT para construir el primer dispositivo embebido, obteniendo datos de un sensor. Sin embargo, en esta ocasión, ampliará las capacidades del dispositivo al aprovechar su capacidad de conectividad. Por otro lado, explorará el ESP32 para entender su potencial en términos de programación en tiempo real, haciendo hincapié en el uso de interrupciones y los modos de bajo consumo disponibles en este microprocesador. Este enfoque dual permitirá adentrarnos en diferentes aspectos del desarrollo de sistemas embebidos y comprender mejor las posibilidades que ofrecen estos microcontroladores en diversas aplicaciones.

El dispositivo Arduino Nano 33 IoT que además las características habituales del Arduino Nano, incorpora conectividad WiFi y conectividad Bluetooth mediante un módulo Nina W102 uBlox, por lo que es ideal para proyectos IoT.

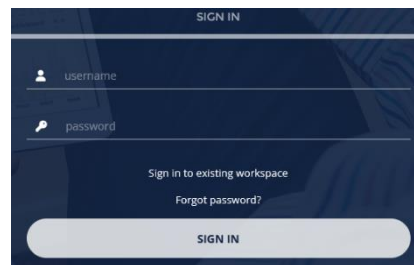


Para poder programar esta placa con el Arduino IDE, tendremos que instalar mediante el Boards manager la familia de placas Arduino SAMD Boards (32-bits ARM Cortex-M0+). Como siempre también tendrá que seleccionar la placa Arduino NANO 33 IoT y el puerto del ordenador al que este conectada.

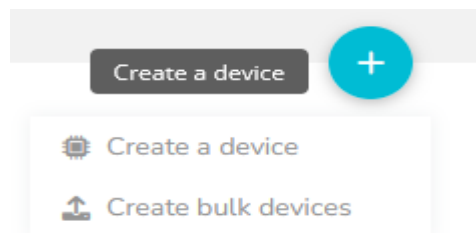
Parte 1

Antes de empezar a resolver la practica cabe indicar que no pondré pregunta a pregunta, si no que iré contestando a ellas de manera ordenada tal y como se indica en el pdf de la práctica.

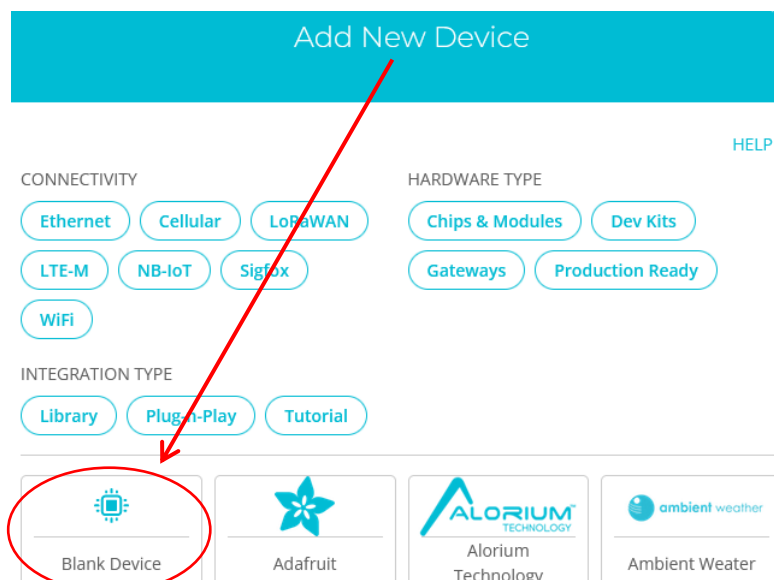
En primer lugar tendrá que **configurar Ubidots Steam** para ello tendrá que crear una cuenta en la plataforma, accediendo al portal de esta, <https://stem.ubidots.com>.



Una vez terminado el registro, tendrá que crear un dispositivo llamado "Nano-Iot-Test", para realizar esto tendrá que, desde el menú superior hacer clic en Dispositivos y en su página crear un dispositivo seleccionando en "**Create Device**".

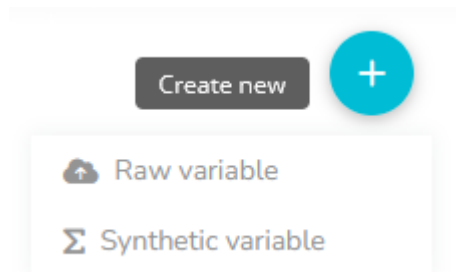


Se abrirá un desplegable, selecciona dispositivo en blanco tal y como se indica en el enunciado. Y por último, completa cualquier otra información relevante que se requiera y haz clic para finalizar la creación del dispositivo.



Una vez dentro del dispositivo "Nano-Iot-Test", siga estos pasos para **crear la variable pulsaciones**, que se encargará de registrar las pulsaciones obtenidos por nuestro sensor.

Para crearla correctamente, tal y como se indica, tendrá que ser una 'Raw Variable'



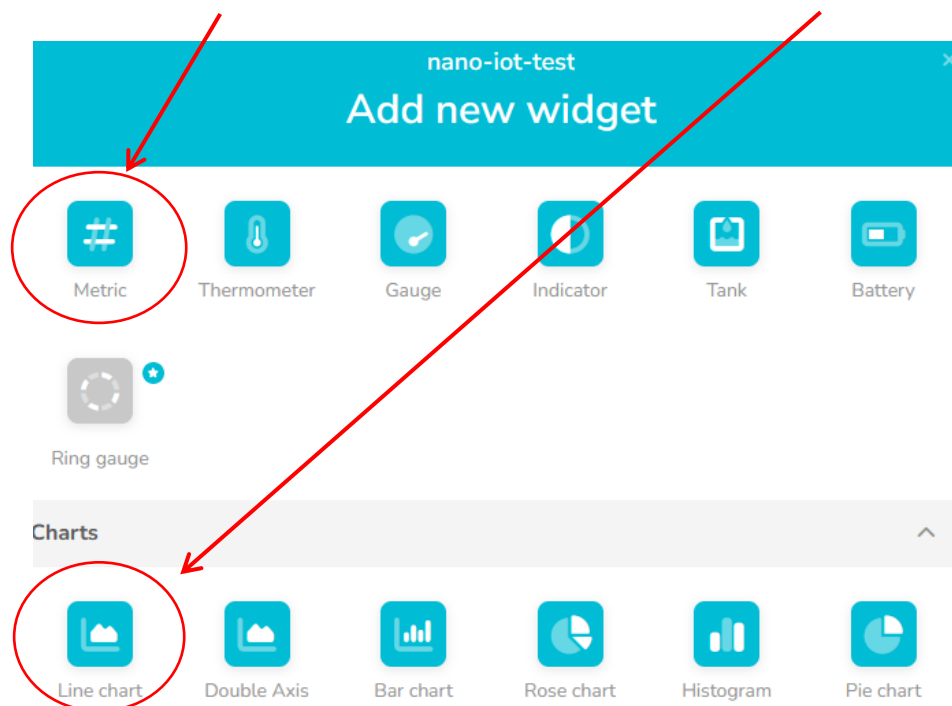
Tendrá que asignarle el nombre 'pulsaciones' y posteriormente establecer un rango permitido delimitado entre 0 y 220.

Una vez tenga la variable insertada en el device, podrá pasar a **modificar el Dashboard**, para ello utilice el predefinido y simplemente lo modifíquelo para añadir las siguientes widgets.

Tal y como se indica, una **widget de métrica** para mostrar el último valor de la variable 'pulsaciones', un **widget de grafico de línea** para representar los últimos valores de esta y por último uno **para mostrar una imagen de nuestro dispositivo**.

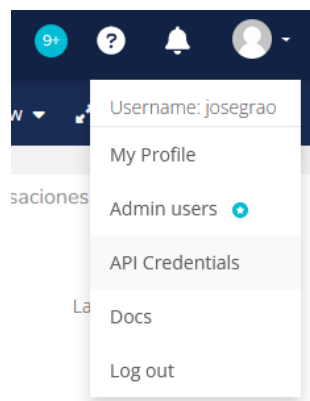
Widget para ultimo valor pulsaciones

Widget para la gráfica de pulsaciones



Una vez ya ha terminado de configurar Ubidots, ya podrá realizar la implementación tanto del código pertinente para conectar con Ubidots, como de la implementación del Arduino en nuestra placa.

Primer a obtendrá los credenciales de Ubidots que utilizará más tarde. Para ello desde el menú de Ubidots seleccione la opción **API Credentials** y copie el token de Ubidots, ya que este nos permitirá establecer conexión con la plataforma posteriormente.



Para la **conectividad WiFi**, haremos uso de la librería de Arduino Wi-FiNINA que deberá instalar previamente en el gestor de librerías del IDE.

Para los credenciales de la red WiFi, es una buena practica mantener los secretos y contraseñas separadas del código de producción. Para ello tendrá que crear otro archivo Arduino llamado secrets.h , cuyo contenido podría ser:

```
1 // Use this file to store all of the private credentials
2 // and connection details
3
4 #define SECRET_SSID "POCOX3NFC" // replace MySSID with your WiFi network name
5 #define SECRET_PASS "honorpass1234" // replace MyPassword with your WiFi password
```

Comenzando con el programa principal, importaremos la libreria WiFi y también el archivo de secretos creado anteriormente.

```
26 #include <WiFiNINA.h>
27 #include "secrets.h"
```

Podrá pasar las variables del compilador en variables char[].

```
47 char ssid[] = SECRET_SSID;
48 char pass[] = SECRET_PASS;
```

Aunque solo sea para conectar a la red WiFi no es necesario, es habitual crear una variable global de programa para el cliente WiFi que permitirá usar los protocolos TCP, UDP, DNS.

```
52  WiFiClient wifiClient;
```

Esta misma variable luego podrá ser pasada a librerías que implementen otros protocolos como HTTP y MQTT. Esto se verá en los próximos apartados.

Antes de conectar a la red, tendrá que hacer algunas comprobaciones para asegurar que puede comunicarse correctamente con el chip WiFi y que la versión firmware es correcta. Para ello implemente el siguiente código.

```
171  void connectWiFi() {
172      int status = WL_IDLE_STATUS;
173
174      if (status == WL_NO_MODULE) {
175          Serial.println("Communication with WiFi module failed!");
176          while (true);
177      }
178
179      String fv = WiFi.firmwareVersion();
180      if (fv != "1.0.0") {
181          Serial.println("Please upgrade the firmware");
182      }
183
184      while (status != WL_CONNECTED) {
185          Serial.println("Conectando a la red WiFi de: ");
186          Serial.println(ssid);
187          status = WiFi.begin(ssid, pass);
188          delay(3000);
189      }
190      printWifiStatus();
191  }
```

Se realizará la conexión mediante la función `WiFi.begin()` que recibe como parámetros el nombre y contraseña de la red WiFi. Además esta se encuentra dentro de un bucle por su falla su intento de conexión para que lo vuelva a intentar.

Por último hay que comentar que la función `printWifiStatus()` imprime en el Serial algunas características de nuestra red WiFi si la conexión se ha efectuado exitosamente.

```
193 void printWifiStatus() {  
194     Serial.print("SSID: ");  
195     Serial.println(WiFi.SSID());  
196  
197     IPAddress ip = WiFi.localIP();  
198     Serial.print("IP Address: ");  
199     Serial.println(ip);  
200  
201     long rssi = WiFi.RSSI();  
202     Serial.print("signal strength (RSSI):");  
203     Serial.print(rssi);  
204     Serial.println(" dBm");  
205 }
```

Ahora una vez implementado lo anterior debería de devolvernos lo siguiente:

```
Conectando a la red WiFi de:  
POCOX3NFC  
SSID: POCOX3NFC  
IP Address: 192.168.43.217  
signal strength (RSSI):-55 dBm
```

Tal y como se pide en la practica a **enviar las pulsaciones a Ubidots mediante el protocolo HTTP**. Para ello añade a continuación todo lo relacionado con HTTP.

Para facilitar el uso del protocolo HTTP en el programa use la libreria **ArduinoHttpClient** que habrá instalar mediante el gestor de paquetes. A continuación importe la libreria y cree el cliente HTTP:

```
29 #include <ArduinoHttpClient.h> // Incluye la librería ArduinoHttpClient
```

```
63 HttpClient httpClient(wifiClient, "industrial.api.ubidots.com", 80);
```

Con esta última linea, el cliente HTTP, obtendrá la red WiFi, para tener acceso a los protocolos TCP, UDP..., dispondrá del endpoint que posteriormente se indicará como establecerlo y por último, el puerto de conexión.

Ahora bien, para enviar la variable a la plataforma, según la documentación de Ubidots, para actualizar el valor de una variable habrá que hacer una petición POST al endpoint:

`https://industrial.api.ubidots.com/api/v1.6/variables/<variable id>/values`

Una vez conectados a una red WiFi, se podrá comenzar a hacer peticiones, para ello utilice las siguientes funciones manteniendo el orden:

- **httpClient.beginRequest()** Comenzamos una petición HTTP.
- **httpClient.get(url), httpClient.post()** Especificamos el endpoint, pasando un path absoluto como parámetro /api/v1.6/variables/pulsaciones/values.
- **httpClient.sendHeader(key, value)** Especifica headers. Este método es opcional y se puede repetir varias veces para especificar múltiples headers.
- **httpClient.beginBody()** Iniciamos la especificación del body de la petición.
- **httpClient.print(postData)** Añadimos el body. Esta función también puede usarse tantas veces como haga falta.
- **httpClient.endRequest()** Finalizamos y enviamos la petición.

Una vez hecho esto, el servidor debería recibir y responder a la petición. Una vez mandada la petición, usaremos las siguientes funciones para obtener la respuesta:

- **httpClient.responseStatusCode()** Devuelve un int con el status de la petición HTTP.
- **httpClient.responseBody()** Devuelve un String con la respuesta del servidor

Una vez entendidas todas las funciones que se pueden utilizar solo falta implementar la función que se encargará de enviar la información.

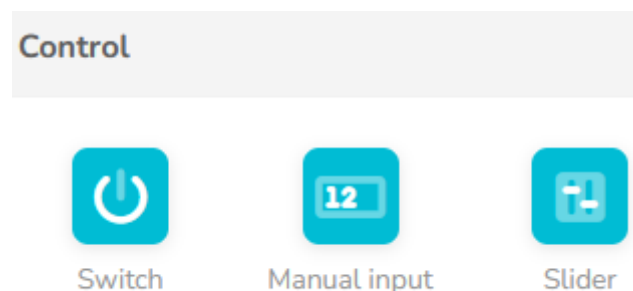
```
250 void sendDataToUbidots(int pulsaciones) {
251     String variableId = "65fb079e7c349d000d4827d6";
252     String postData = "{\"value\": " + String(pulsaciones) + "}";
253     httpClient.beginRequest();
254     httpClient.post("/api/v1.6/variables/" + variableId + "/values");
255     httpClient.sendHeader("Content-Type", "application/json");
256     httpClient.sendHeader("X-Auth-Token", ubidotsToken);
257     httpClient.sendHeader("Content-Length", postData.length());
258     httpClient.beginBody();
259     httpClient.print(postData);
260     httpClient.endRequest();
261     int statusCode = httpClient.responseStatusCode();
262     String response = httpClient.responseBody();
263     if (statusCode != 201) {
264         Serial.println("Posible error al enviar las pulsaciones:");
265         Serial.print("Status code: ");
266         Serial.println(statusCode);
267         Serial.print("Response: ");
268         Serial.println(response);
269     }
```

Como se pide en la práctica, **observa los cambios obtenidos en el dashboard** una vez se envía la información desde nuestro Arduino a la plataforma de Ubidots a lo largo del tiempo.



Observa como el Line Chart muestra la evolución de las pulsaciones obtenidas por el sensor a lo largo de las mediciones, como en el centro está la imagen del Arduino y por último el Last Value de la variable pulsaciones.

Una vez finalizado con el protocolo HTTP, se pide implementar el **protocolo MQTT**, para enviar y recibir datos. Para ello cree un nuevo botón en el dashboard como se ha realizado anteriormente.



Seleccione la opción **Manual input**, añada el comportamiento para cambiar el valor numérico cuando se pulsa el botón, esto hará que se envíe un mensaje al topic de MQTT correspondiente.

Para facilitar el uso del protocolo MQTT en el programa use la librería **PubSubClient** que habrá instalar mediante el gestor de paquetes. A continuación importe la librería y cree el cliente MQTT y le pasamos el cliente WiFi.

```
53 PubSubClient mqttClient(wifiClient);
```

Una vez creado el cliente, habrá que pasarle la configuración, que será dos variables una con el endpoint y la otra el puerto del bróker MQTT que en este caso usaremos Ubidots.

```
161 mqttClient.setServer("industrial.api.ubidots.com", 1883);
```


Para realizarlo de una manera más sencilla, aquí está el código utilizado para realizar la conexión mediante MQTT a Ubidots.

```
158 void connectMQTT() {
159     Serial.println("Conectando al broker MQTT...");
160
161     mqttClient.setServer("industrial.api.ubidots.com", 1883);
162     // set the message receive callback
163     mqttClient.setCallback(onMqttMessage);
164
165     while (!mqttClient.connected()) {
166         Serial.println("Intentando conexión MQTT...");
167         if (mqttClient.connect("nano-iot-test", ubidotsToken, "")) {
168             Serial.println("Conectado al broker MQTT!");
169             // subscribe to a topic
170             mqttClient.subscribe(mqttTopic);
171             // topics can be unsubscribed
172         } else {
173             Serial.print("Error al conectar al broker MQTT, rc=");
174             Serial.print(mqttClient.state());
175             Serial.println(" Intentando de nuevo en 5 segundos...");
176             delay(5000);
177         }
178     }
179 }
```

Como se puede observar, necesitará tanto el token al igual que en el protocolo HTTP y necesitará la información sobre el topic al que se va a conectar, el cual será:

```
50 const char *mqttTopic = "/v1.6/devices/nano-iot-test/ubidots-input/lv";
```

Podrá ver como se realiza con éxito la conexión tanto a MQTT como la **suscripción al topic**, ejecuta el código y obtendrá lo siguiente.

```
Intentando conexión MQTT...
Conectado al broker MQTT!
Coloque su dedo índice en el sensor con presión constante.
Mensaje recibido: /v1.6/devices/nano-iot-test/ubidots-input/lv
```

Una vez finalizado lo anterior, ya solo quedaría **modificar el envío** de los datos anteriormente implementado por HTTP a una **nueva versión con MQTT** e implementar lógica en el código para que cuando llegue un mensaje por MQTT el dispositivo quede **suspendido**, esto quiere decir que no realice ninguna medición y que muestre por pantalla la palabra suspendido. Hasta que se reinicie o llegue otro mensaje.

Para este ultimo apartado de la parte 1 de la practica tendremos que eliminar todo lo anteriormente creado por HTTP y adaptarlo a MQTT.

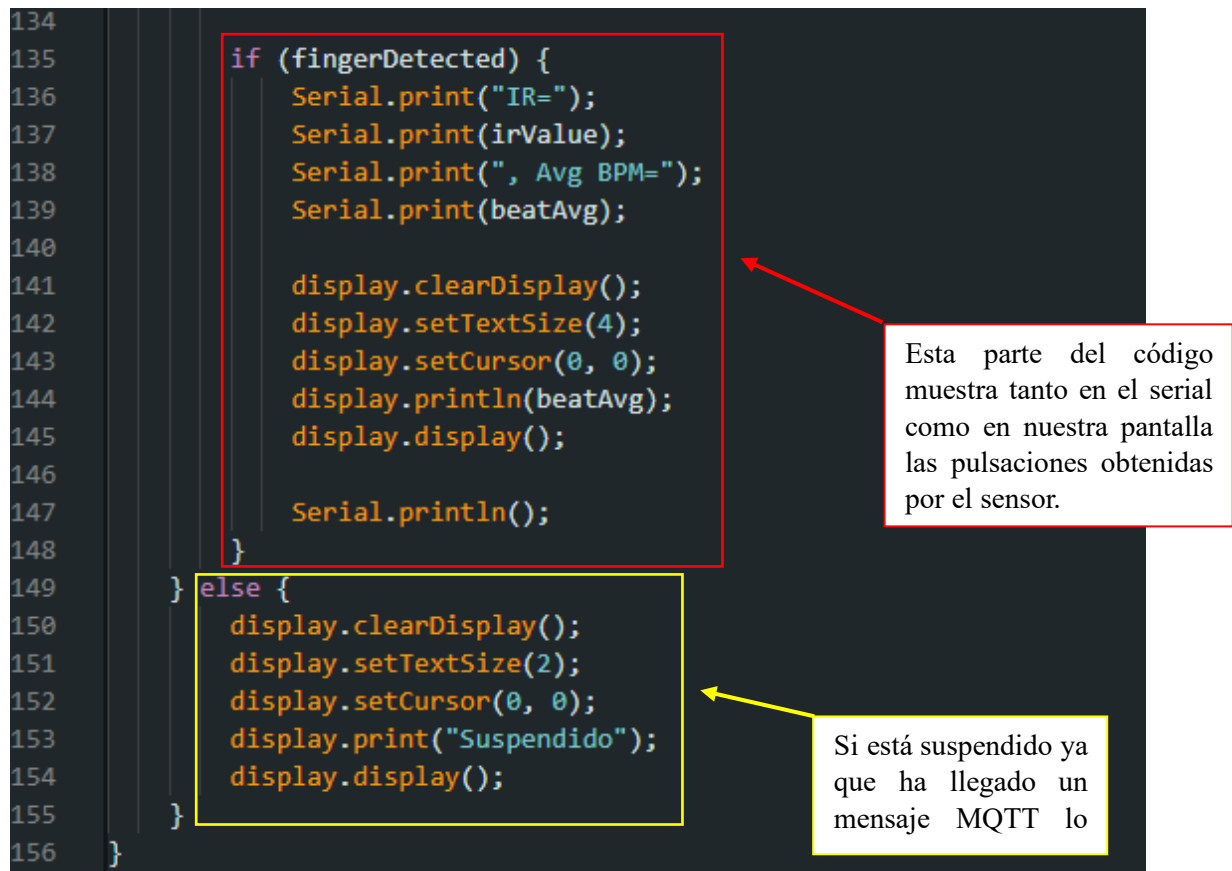
Para adaptar la lógica de nuestro código tendrá que modificar la función `loop()` la cual contiene desde el primer momento todo el código asociado al sensor de ritmo cardiaco y oxígeno de la practica 2. Implementa el siguiente código, posteriormente habrá una explicación de su funcionamiento.

```

91 void loop() {
92   mqttClient.loop();
93   unsigned long currentMillis = millis(); // Obtener el tiempo actual
94
95   // Verificar si ha pasado el intervalo de tiempo establecido en la
96   if (suspendido == 0) {
97     if (currentMillis - previousMillis >= interval) {
98       previousMillis = currentMillis; // Actualizar el tiempo ant
99       // Enviar datos a Ubidots
100      sendDataToUbidots(beatAvg);
101     }
102     long irValue = particleSensor.getIR();
103
104     if (irValue < 50000) {
105       fingerDetected = false;
106       display.clearDisplay();
107       display.setTextSize(2);
108       display.setCursor(0, 0);
109       display.print("Situe dedo");
110       display.display();
111     } else {
112       if (!fingerDetected) {
113         fingerDetected = true;
114         Serial.println("Dedo detectado. Analizando ritmo...");
115       }
116
117       if (checkForBeat(irValue)) {
118         long delta = millis() - lastBeat;
119         lastBeat = millis();
120
121         beatsPerMinute = 60 / (delta / 1000.0);
122
123         if (beatsPerMinute < 255 && beatsPerMinute > 20) {
124           rates[rateSpot++] = (byte)beatsPerMinute;
125           rateSpot %= RATE_SIZE;
126
127           beatAvg = 0;
128           for (byte x = 0; x < RATE_SIZE; x++)
129             beatAvg += rates[x];
130           beatAvg /= RATE_SIZE;
131         }
132       }
133     }
  
```

Condición para saber si el dedo esta situado en el sensor, gracias a la función `getIR()` que recoge los valores del led infrarrojo del sensor

Función que obtiene las pulsaciones detectadas por el sensor.



Todo este código como se puede observar estará contenido en un gran bloque if, else; el cual controlará los mensajes recibidos por MQTT, ya que si recibe un mensaje habrá que dejar el sensor suspendido para que no analice hasta nuevo aviso (otro mensaje MQTT). En este caso para hacer la implementación mas sencilla de entender implementa una función auxiliar para controlar esta variable suspended.

Antes de entrar en detalle en la parte de suspender al recibir un mensaje MQTT, vamos a ver como se **envía la información a Ubidots mediante MQTT**. Para ello haz uso de una función auxiliar que se encargue exclusivamente de ello.

```
220 void sendDataToUbidots(int pulsaciones) {
221     String topic = "/v1.6/devices/nano-iot-test/pulsaciones";
222     String payload = String(pulsaciones);
223     mqttClient.publish(topic.c_str(), payload.c_str());
224 }
```

Una vez entendido este apartado, implementa una nueva función auxiliar que será la llamada en el bucle principal al hacer mqttclient.loop()

Implementa la lógica pedida en la practica para que cuando llegue un mensaje por MQTT el dispositivo quede **suspendido**, esto quiere decir que no realice ninguna medición y que muestre por pantalla la palabra suspendido. Hasta que se reinicie o llegue otro mensaje.

```
226 void onMqttMessage(char* topic, byte* payload, unsigned int length) {  
227  
228     int receivedValue = atoi((char *)payload);  
229     if(lastReceivedValue != receivedValue){  
230         Serial.print("Mensaje recibido: ");  
231         Serial.println(topic);  
232         Serial.print("Contenido:");  
233         Serial.println(receivedValue);  
234         lastReceivedValue = receivedValue;  
235         suspendido = 1;  
236     }else{  
237         suspendido = 0;  
238     }  
239 }
```

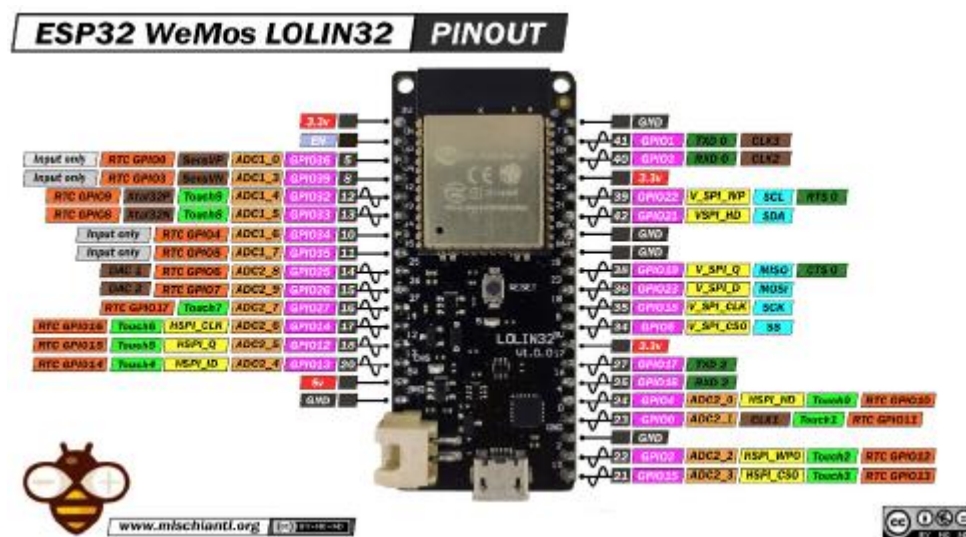
Como puede observar, parsea el mensaje recibido del topic suscrito y si es distinto el numero recibido al que se había recibido anteriormente entonces la variable suspendido valdrá 1 y el loop() principal no hará nada, no tomará mas mediciones hasta que el mismo numero vuelva a ser recibido por MQTT.

Con esto se finaliza el bloque 1 de la practica que estaba dirigido a la comunicación mediante distintos protocolos.

Parte 2

En este apartado de la práctica se realizarán pruebas con las funcionalidades de tareas disponibles en el sistema operativo incluido en los MCU ESP32. Introducir el dispositivo en primera instancia hará que sea más sencillo su utilización y programación.

Creado y desarrollado por Espressif Systems, ESP32, una serie de microcontroladores de bajo costo y de bajo consumo con sistema en chip con WiFi y Bluetooth de modo dual integrados. El procesador tiene dos núcleos de procesamiento cuyas frecuencias operativas pueden controlarse independientemente entre 80 megahercios (MHz) y 240 MHz.




Una vez tenga una idea del microcontrolador que utilizará, proceda a vincularlo con Arduino IDE. Para ello en primer lugar tendrá que instalar ciertas librerías para su utilización. Estas serán:

- Arduino_ESP32_OTA

Una vez instalada, si se realiza la practica con Windows puede que el IDE no le detecte el **port**, por lo que para ello tendrá que instalar los drivers necesarios de la placa. Para ello, siga los siguientes pasos:

En primer lugar para confirmar que el driver no esta instalado y que el problema es este, vaya a **Administrador de dispositivos** en su sistema Windows.

Cuando no se encuentra instalado el driver en la PC, por lo regular, la tarjeta ESP32 aparece en Otros dispositivos como CP2102 USB to UART Bridge Controller, con un símbolo de warning .



Si ya comprobamos que realmente no está instalado el driver, descargamos el instalador desde la página oficial de SILICON LABS, para Windows mediante el siguiente enlace:

<https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers?tab=downloads>

Como recomendación, instale el que se indica seguidamente, ya que es el que viene con instalador.

Software Downloads

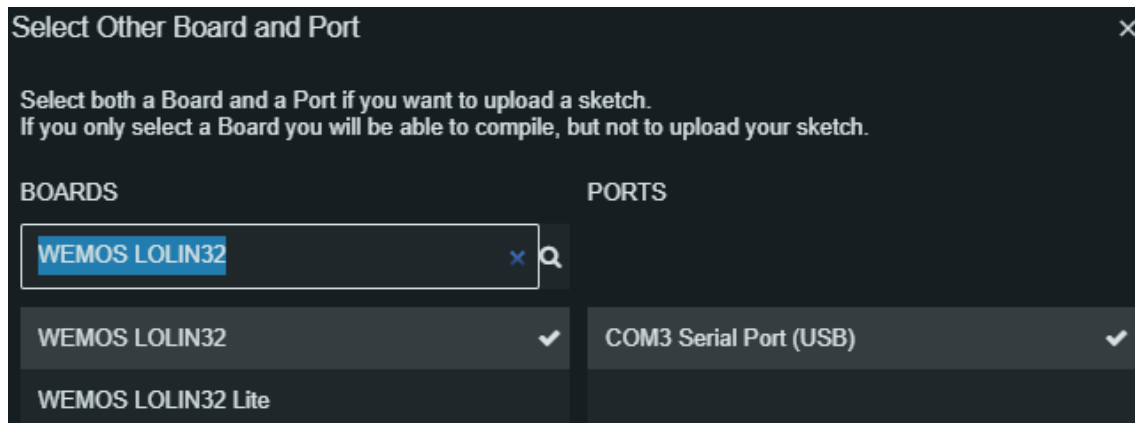
Software (10)

Software · 10

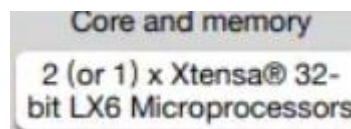
CP210x Universal Windows Driver	v11.3.0 5/25/2023
CP210x VCP Mac OSX Driver	v6.0.2 10/26/2021
CP210x Windows Drivers	v6.7.6 9/3/2020
CP210x Windows Drivers with Serial Enumerator	v6.7.6 9/3/2020
CP210x_5x_AppNote_Archive	9/3/2020

Posteriormente, descomprima y ejecute el archivo que corresponda a su PC. Si es de 64 bits, ejecute el archivo CP210xVCPInstaller_x64.exe

Cuando finalice la instalación, reinicie el PC para que se apliquen los cambios, una vez realizado vuelva a Arduino IDE, y ya verá como le detecta el microcontrolador.



Una de las principales ventajas del ESP32 sobre las placas Arduino tradicionales es su capacidad para realizar tareas en paralelo gracias a sus dos núcleos de procesamiento Xtensa LX6 de 32 bits: core 0 y core 1. Mientras que la mayoría de las placas Arduino están basadas en microcontroladores de un solo núcleo, el ESP32 tiene dos núcleos de procesamiento que pueden funcionar de forma independiente uno del otro. Una vez terminada esta breve introducción y configuración del dispositivo proceda con la práctica.



En esta primera parte se creará un **script con dos tareas una que imprime un mensaje y la otra que enciende y apaga el led integrado en la placa.**

Para realizar esto de manera sencilla vamos a ver unos concetos sobre la programación de la placa.

- `xPortGetCoreID()` sirve para identificar que núcleo ejecuta la tarea

Arduino IDE es compatible con FreeRTOS para ESP32, que es un sistema operativo en tiempo real. Esto permite manejar varias tareas en paralelo que se ejecutan de forma independiente. Las tareas son fragmentos de código que ejecutan algo. Por ejemplo, puede hacer parpadear un LED, realizar una solicitud de red, medir lecturas de sensores, publicar lecturas de sensores, etc.

- `TaskHandle_t Task1` sirve para crear un identificador de una tarea

Posteriormente para configurar la tarea creada, tendrá que utilizar la siguiente función:

```

10 // Crear tarea 1 que escribe en el puerto serial cada 100 ms
11 xTaskCreatePinnedToCore(
12   taskFunction1,    // Función de la tarea
13   "Task1",         // Nombre de la tarea
14   10000,           // Tamaño de la pila de la tarea
15   NULL,            // Parámetro de la tarea
16   1,               // Prioridad de la tarea
17   &Task1,          // Manejador de la tarea
18   0                // Asignar tarea al núcleo 0
19 );
  
```

Una vez ya creada la tarea, podrá implementar la función que necesita ser ejecutada por esa tarea.

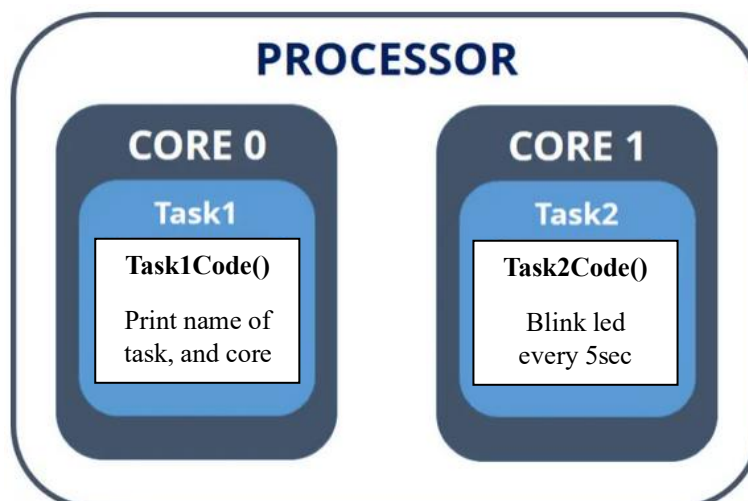
```

33 // Tarea 1: escribe en el puerto serial cada 100 ms
34 void taskFunction1(void * pvParameters) {
35   for (;;) {
36     Serial.printf("Hola soy la tarea %s, me estoy ejecutando en el core %d\n", pcTaskGetName(NULL), xPortGetCoreID());
37     vTaskDelay(pdMS_TO_TICKS(100)); // Esperar 100 ms
38   }
39 }
  
```

Como se puede intuir, si es capaz de crear una tarea, del mismo modo será capaz de eliminarla, para ello dispone de la siguiente función:

- `vTaskDelete(Task1);` elimina la tarea pasa por parámetro

Para implementar el primer apartado de la segunda parte simplemente tendrá que hacer esto dos veces y realizar en un núcleo la impresión del mensaje y en el otro llevar el control de la led integrada. Podemos verlo reflejado de la siguiente manera:



Para realizar lo anterior, implemente dos funciones y dos creaciones de tasks en la función `setup()`, de la siguiente manera.

```
7 void setup() {
8     Serial.begin(115200);
9
10    // Crear tarea 1 que escribe en el puerto serial cada 100 ms
11    xTaskCreatePinnedToCore(
12        taskFunction1,    // Función de la tarea
13        "Task1",          // Nombre de la tarea
14        10000,            // Tamaño de la pila de la tarea
15        NULL,             // Parámetro de la tarea
16        1,                // Prioridad de la tarea
17        &Task1,           // Manejador de la tarea
18        0                 // Asignar tarea al núcleo 0
19    );
20
21    // Crear tarea 2 que enciende y apaga el LED integrado cada 5 segundos
22    xTaskCreatePinnedToCore(
23        taskFunction2,    // Función de la tarea
24        "Task2",          // Nombre de la tarea
25        10000,            // Tamaño de la pila de la tarea
26        NULL,             // Parámetro de la tarea
27        1,                // Prioridad de la tarea
28        &Task2,           // Manejador de la tarea
29        1                 // Asignar tarea al núcleo 1
30    );
31 }
```

Y simplemente necesitará las funciones de control de cada tarea.

```
33 // Tarea 1: escribe en el puerto serial cada 100 ms
34 void taskFunction1(void *pvParameters) {
35     for (;;) {
36         Serial.printf("Hola soy la tarea %s, me estoy ejecutando en el core %d\n", pcTaskGetName(NULL), xPortGetCoreID());
37         vTaskDelay(pdMS_TO_TICKS(100)); // Esperar 100 ms
38     }
39 }
40
41 // Tarea 2: enciende y apaga el LED integrado cada 5 segundos
42 void taskFunction2(void *pvParameters) {
43     pinMode(LED_BUILTIN, OUTPUT);
44     for (;;) {
45         digitalWrite(LED_BUILTIN, HIGH);
46         vTaskDelay(pdMS_TO_TICKS(5000));
47         digitalWrite(LED_BUILTIN, LOW);
48         vTaskDelay(pdMS_TO_TICKS(5000));
49     }
50 }
```

Como puede observar una vez ejecutado una led azul se enchufará cada 5 segundos en el board y cada segundo se imprimirá en el Serial el mensaje siguiente:

```
Hola soy la tarea Task1, me estoy ejecutando en el core 0
```

Con esto se da por concluida la implementación del primer apartado, pero faltaría ver lo siguiente, **¿Pese al DELAY, las tareas se ejecutan en paralelo?**

El sistema de temporización de FreeRTOS se basa en "ticks", que son unidades de tiempo básicas que son controladas por un "tick interrupt". Este "tick interrupt" ocurre a intervalos regulares y es responsable de aumentar un contador de interrupciones interno, mantener la temporización de tareas y administrar la planificación del sistema.

La función `vTaskDelay()` suspende temporalmente y se pone en estado de bloqueo durante un número determinado de ticks cuando una tarea la llama. La tarea no se ejecutará durante este tiempo y estará en espera hasta que pase el número de ticks especificado.

Por lo tanto, otras tareas del sistema que están listas para hacerlo seguirán ejecutándose incluso si una tarea esté bloqueada debido a un `vTaskDelay()`. Esto permite que En sistemas que utilizan FreeRTOS, como ESP32, las tareas se pueden ejecutar en paralelo.

Siguiendo con lo mencionado, ahora tendrá que adaptar el código a la siguiente funcionalidad: Conecta el emisor led de la práctica 2 al esp32 de forma similar a como hiciste con Arduino. Programa ahora **tres tareas, una por cada color del RGB**, estas tareas deben encender y apagar el color correspondiente siguiendo los siguientes tiempos: 1000 ms para el rojo. 2000ms para el azul. 5000 ms para el verde.

Como se mencionó con anterioridad, para esta pequeña modificación simplemente habrá que añadir una nueva tarea y en las funciones correspondientes a cada una de ellas plantear la lógica de enchufado y apagado de los leds.

```
Enchufando LED rojo... cada 1 segundo
Enchufando LED rojo... cada 1 segundo
Enchufando LED azul...cada 2 segundos
Enchufando LED rojo... cada 1 segundo
Enchufando LED rojo... cada 1 segundo
Enchufando LED azul...cada 2 segundos
Enchufando LED rojo... cada 1 segundo
Enchufando LED verde... cada 5 segundos
```

Como se puede observar el pin rojo se enchufa cada segundo, el azul cada 2 y el verde cada 5. Esto ocurre al igual que en el anterior apartado gracias a la función:

```
vTaskDelay(pdMS_TO_TICKS(X000)); //Siendo X el valor que necesitamos
```

Para implementar la lógica del enchufado y apagado de los leds simplemente implementa las siguientes funciones.

Para el led rojo:

```
49 // Tarea para el LED rojo
50 void taskRedLED(void * pvParameters) {
51     pinMode(RED_LED_PIN, OUTPUT);
52
53     for (;;) {
54         digitalWrite(RED_LED_PIN, HIGH);
55         vTaskDelay(pdMS_TO_TICKS(1000));
56         digitalWrite(RED_LED_PIN, LOW);
57         Serial.println("Enchufando LED rojo... cada 1 segundo");
58         vTaskDelay(pdMS_TO_TICKS(1000));
59     }
60 }
```

Para el led azul:

```
62 // Tarea para el LED azul
63 void taskBlueLED(void * pvParameters) {
64     pinMode(BLUE_LED_PIN, OUTPUT);
65
66     for (;;) {
67         digitalWrite(BLUE_LED_PIN, HIGH);
68         vTaskDelay(pdMS_TO_TICKS(2000));
69         digitalWrite(BLUE_LED_PIN, LOW);
70         Serial.println("Enchufando LED azul...cada 2 segundos");
71         vTaskDelay(pdMS_TO_TICKS(2000));
72     }
73 }
```

Para el led verde:

```
75 // Tarea para el LED verde
76 void taskGreenLED(void * pvParameters) {
77     pinMode(GREEN_LED_PIN, OUTPUT);
78
79     for (;;) {
80         digitalWrite(GREEN_LED_PIN, HIGH);
81         vTaskDelay(pdMS_TO_TICKS(5000));
82         digitalWrite(GREEN_LED_PIN, LOW);
83         Serial.println("Enchufando LED verde... cada 5 segundos");
84         vTaskDelay(pdMS_TO_TICKS(5000));
85     }
86 }
```

Con esto puedes observar como el modulo led RGB va cambiando de color, a veces y como consecuencia de que estén trabajando las tareas en paralelo, podrán solaparse los colores, entonces obtendrás colores como el blanco, el morado o el naranja.

Para finalizar esta segunda parte de la práctica, realiza otra modificación para que estas funciones de **control sobre led estén implementadas en una sola tarea**. Para implementar esto realiza una única función a la cual llamarás tres veces mediante la misma tarea con distintos parámetros.

Primero y antes que nada nos crearemos la estructura del tipo de dato y además crea la tarea, que se encargue de llamar tres veces a la función con los distintos tipos de led.

A continuación, inicializa los leds, de la siguiente manera.

```
17 // Crear tres instancias de la estructura GenericData con diferentes parámetros
18 static GenericData paramsRed = {27, 1000}; // Pin 2 para el LED rojo, delay de 1000ms
19 static GenericData paramsBlue = {25, 2000}; // Pin 4 para el LED azul, delay de 2000ms
20 static GenericData paramsGreen = {26, 5000}; // Pin 5 para el LED verde, delay de 5000ms
```

Una vez realizo esto, simplemente crea las tareas (llamadas a la función que se encargará de enchufar o apagar los led).

```
22 // Crear las tareas para controlar los LEDs con diferentes parámetros
23 xTaskCreate(tareaLed, "Controlador Red", 1024, (void*)&paramsRed, 1, NULL);
24 xTaskCreate(tareaLed, "Controlador Green", 1024, (void*)&paramsGreen, 1, NULL);
25 xTaskCreate(tareaLed, "Controlador Blue", 1024, (void*)&paramsBlue, 1, NULL);
```

Y por último para la gestión de leds, crea esta función para su procesamiento:

Con esto como en la anterior parte, veras distintos colores ya que hay solapes en la ejecución de las tareas.

```
28 // Tarea que controla los LEDs según los parámetros recibidos
29 void tareaLed(void *params) {
30     GenericData* taskParams = (GenericData *)params;
31
32     for (;;) {
33         // Configurar el pin como salida
34         pinMode(taskParams->pin, OUTPUT);
35
36         // Encender el LED correspondiente
37         digitalWrite(taskParams->pin, HIGH);
38         Serial.print("Encendiendo LED en pin ");
39         Serial.print(taskParams->pin);
40         Serial.println("...");
41         // Esperar el tiempo especificado
42         vTaskDelay(pdMS_TO_TICKS(taskParams->delayTime));
43
44         // Apagar el LED
45         digitalWrite(taskParams->pin, LOW);
46         Serial.print("Apagando LED en pin ");
47         Serial.print(taskParams->pin);
48         Serial.println("...");
49         // Esperar el tiempo especificado
50         vTaskDelay(pdMS_TO_TICKS(taskParams->delayTime));
51     }
52 }
```

Parte 3

En esta última parte simplemente habrá que familiarizarse con el modo bajo consumo del ESP32, que reduce considerablemente su consumo y sus prestaciones, esto es utilizado para aumentar la duración de la batería entre medidas

En esta primera parte se pide sobre el código de la parte 2, añadir la librería de sueño y crear un script que realice diferentes funcionalidades.

En primer lugar tendrá que realizar un script que implemente lo siguiente: Encender un led verde cada 100ms por 10 veces y una vez realizado se encienda un led rojo y se duerma por 15 segundos.

```
19 void loop(){
20     digitalWrite(greenPin, HIGH);
21     for (int i = 0; i < 10; i++){
22         Serial.println("Estoy haciendo trabajo ");
23         Serial.println(contador_reinicios);
24         delay(1000);
25     }
26
27     digitalWrite(greenPin, LOW);
28     digitalWrite(redPin, HIGH);
29     esp_sleep_enable_timer_wakeup(15000000);
30     gpio_hold_en(redPin);
31     esp_deep_sleep_start();
32     //Cuando se duerme el led se apaga.
33 }
```

Como se puede apreciar en el código, implementar primero las diez ejecuciones de la led Verde para posteriormente poner el pin de la led rojo en HIGH y ponerlo a dormir por 15 segundos.

En este caso, la función **gpio_hold_en()** se encargará de mantener encendido el led rojo cumplimentando así la pregunta 3, configura una alarma de temporizador para despertar después de 15 segundos, mantiene el estado del pin rojo durante el sueño profundo y entra en el modo de sueño profundo, donde el sistema se detiene hasta que se activa la alarma del temporizador.

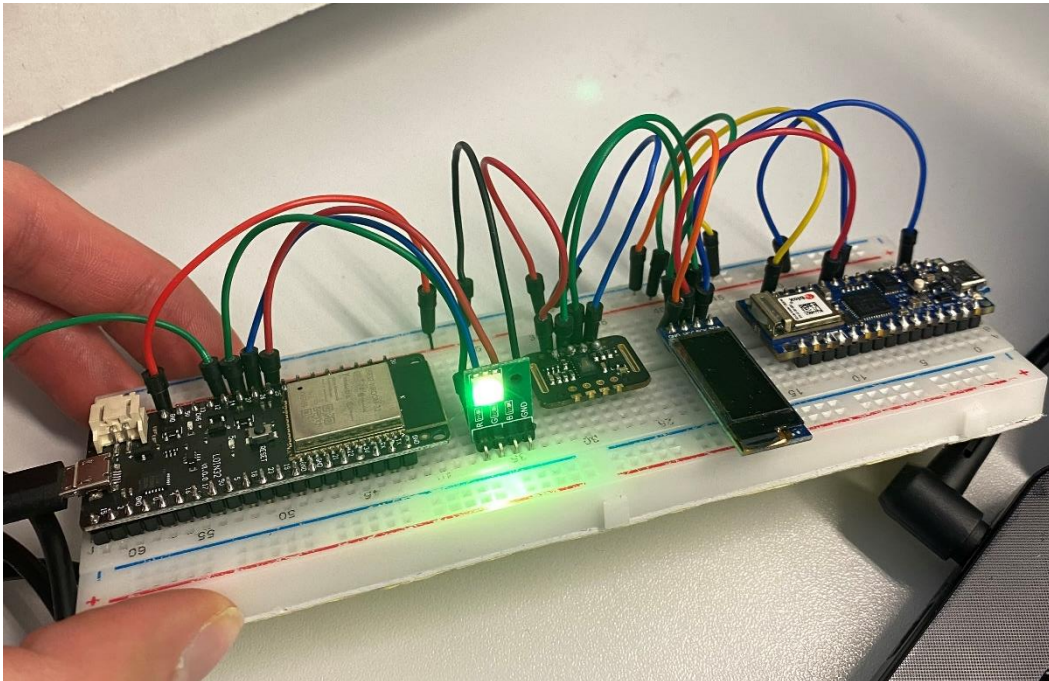
Se pide **añadir un contador de reinicios** para cada vez que se ponga a dormir, para ello añade la siguiente línea:

```
13     contador_reinicios++;
```

Para **implementar la pregunta 4 y 5**, simplemente habrá que añadir dos nuevos pines de nuestro microcontrolador, los cuales se encargaran de cortocircuitar la ejecución, mediante la conexión de un jumper cuando se este ejecutando el código. En este caso, habrá que añadir, lo siguiente al código de la anterior pregunta de la parte 3:

```
10  gpio_num_t wakeUpPin = (gpio_num_t) 4;  
11  gpio_num_t unusedPin = (gpio_num_t) 14;
```

Utilizaremos el pin numero 4 como cortocircuito y el pin numero 14, como pin que en realidad no hace nada.



Como se puede apreciar uno de los jumpers conectados al Lolin 32, solo se encuentra conectado por uno de sus extremos, este permitirá interrumpir el modo sueño y poner a trabajar otra vez al microcontrolador, con esto se puede entender mejor los modos de eficiencia y bajo consumo de los mismos, ya que estos pueden estar enchufados pero esperando alguna causa que active al mismo y desenboque en un efecto. Para implementar esto último en el código simplemente modifica lo anterior de la siguiente manera.


```
41 void loop() {
42
43     digitalWrite(greenPin, HIGH);
44     for (int i = 0; i < 10; i++){
45         Serial.println("Estoy haciendo trabajo ");
46         Serial.println(contador_reinicios);
47         delay(1000);
48     }
49     digitalWrite(greenPin, LOW);
50     digitalWrite(unusedPin, HIGH); // Mantener el pin en HIGH
51     digitalWrite(redPin, HIGH);
52
53     gpio_deep_sleep_hold_en();
54     // Mantengo el pin en alto incluso en modo de sueño
55     gpio_hold_en(unusedPin);
56     gpio_hold_en(redPin);
57
58     esp_sleep_enable_ext0_wakeup(wakeUpPin, HIGH);
59
60     esp_deep_sleep(15000000);
61 }
```

Como se puede apreciar el pin que no se utiliza simplemente esta en HIGH todo el rato, pondrá el microcontrolador a dormir pero siempre dejando los pines rojos y el unused en HIGH para que sigan activos mediante la función nombrada anteriormente, por último, la función `esp_sleep_enable_ext0_wakeup()`, permitirá despertar el microcontrolador en caso de que algo externo suceda, en este caso y como se especifica en los parámetros, si el pin `wakeUpPin` se pone a HIGH.

Como último apartado de la practica se intenta **implementar un código que especifique cuanto tiempo ha pasado en el modo deepsleep el MCU** y que especifique porque ha sido despertado mediante la función `esp_sleep_get_wakeup_cause()` y `ESP32Time.h`

Para ello implementa lo siguiente:

```
63 void obtenerTiempoDespierto() {
64     uint64_t tiempo_dormido = esp_timer_get_time() - tiempo_despierto;
65     Serial.print("El microcontrolador estuvo dormido durante: ");
66     Serial.print(tiempo_dormido);
67     Serial.println(" microsegundos");
68
69     // Obtén la causa del despertar
70     esp_sleep_wakeup_cause_t causa = esp_sleep_get_wakeup_cause();
71     switch(causa) {
72         case ESP_SLEEP_WAKEUP_EXT1:
73             Serial.println("El microcontrolador fue despertado por un evento externo en el pin");
74             break;
75         default:
76             Serial.println("El microcontrolador fue despertado por una causa desconocida");
77             break;
78     }
```



Con lo anterior implementado, obtendrá como salida algo parecido a lo siguiente:

```
El microcontrolador estuvo dormido durante: 18446744073709428813 microsegundos
El microcontrolador fue despertado por una causa desconocida
Estoy haciendo trabajo
20
Estoy haciendo trabajo
20
```

Indicando el tiempo como se pedía en el enunciado de la practica.