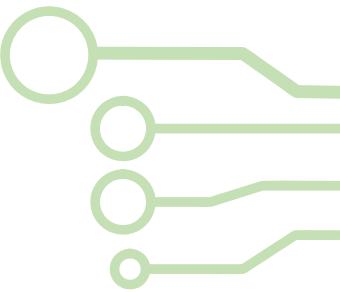
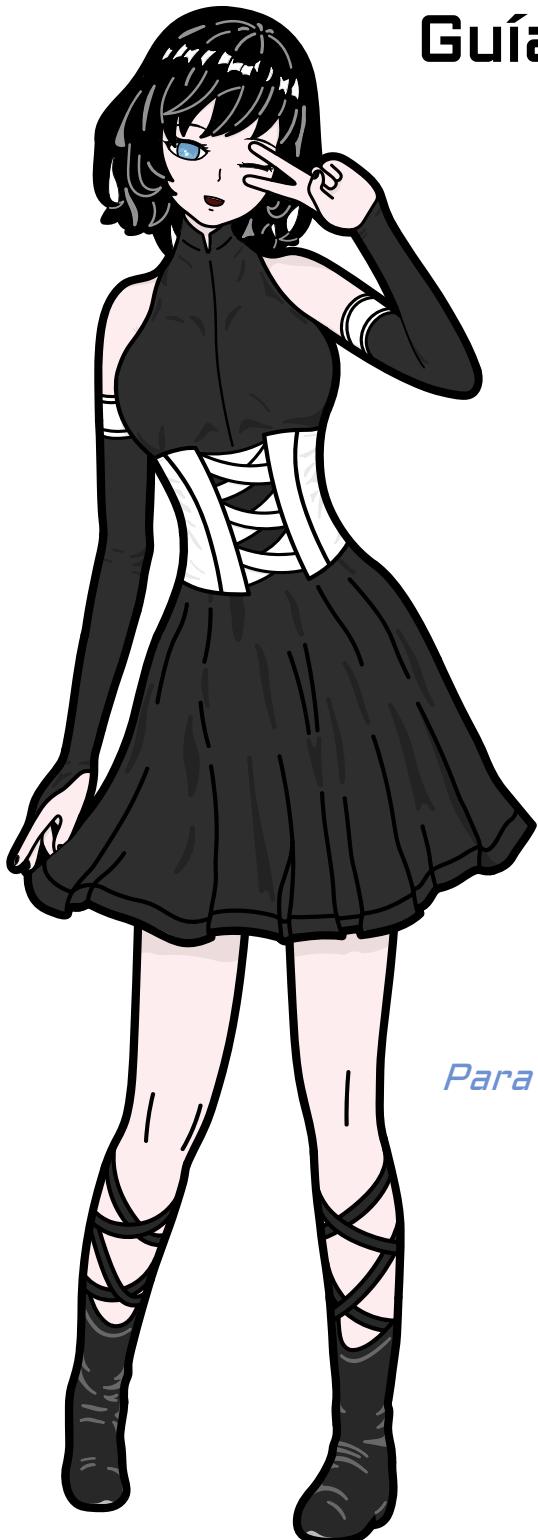
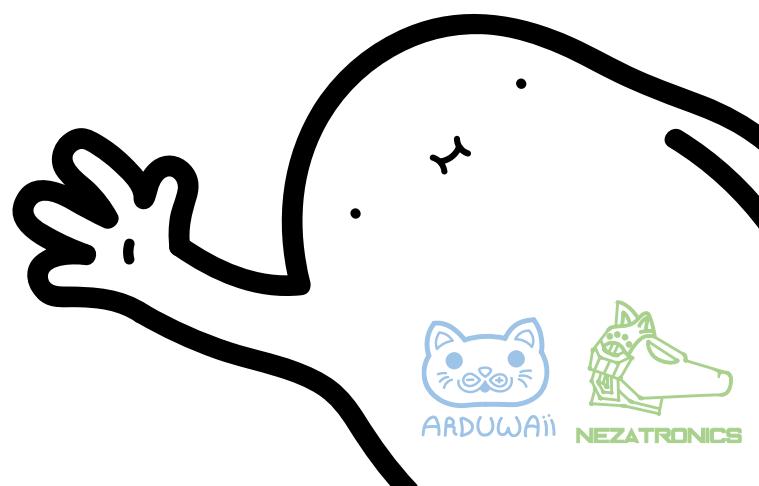


# CONOCE AL ESP32:

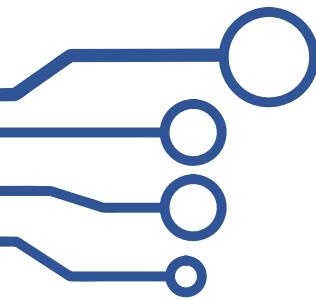
## Guía de funciones básicas y ejercicios prácticos



*Para tarjetas KYOMI Dev Board y compatibles*

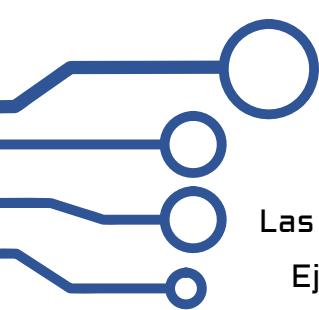


*Esta página ha sido dejada en blanco intencionalmente*

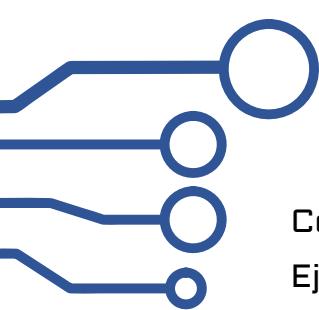


# CONTENIDO

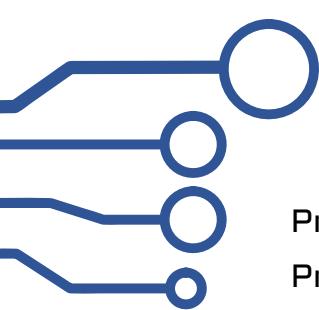
<b>Capítulo 1: ¿Qué es el ESP32?</b> .....	7
Primero, una breve introducción .....	8
Las características del micro .....	9
Especificaciones de hardware del ESP32 WROOM .....	10
Características eléctricas del ESP32 WROOM .....	12
PINOUT de la tarjeta ESP32 KYOMI .....	14
Instala todo lo necesario .....	17
Instalación nueva.....	17
Limpiar una instalación anterior .....	21
Primera prueba.....	24
Comprueba el controlador USB .....	30
<b>Capítulo 2: Primeros pasos con los GPIO's</b> .....	31
Los GPIO's como entradas/salidas digitales.....	32
Básicos: Las resistencias PULL-UP y PULL-DOWN .....	32
Básicos: Los tipos de variables básicas en Arduino .....	35
Básicos: Estructura condicional simple ('if').....	35
Básicos: Estructura cíclica 'for'.....	36
Básicos: Mostrar datos en el Monitor Serie .....	36
Ejercicio 1 .....	37
Ejercicio 2 .....	40
Básicos: Estructura condicional compuesta ('switch') .....	40
Básicos: Cómo usar variables de tipo arreglo ('arrays').....	41
Los GPIO's táctiles (touch) .....	49
Básicos: Graficador Serial (Serial Plotter) .....	50
Calibración de un GPIO touch .....	52
Ejercicio 3 .....	55
Ejercicio 4 .....	59



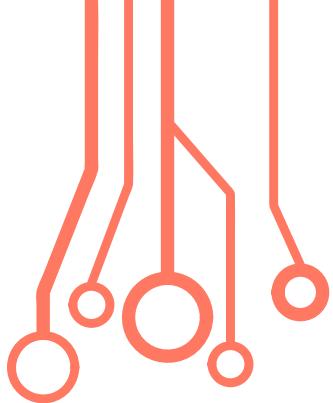
Las Señales PWM (Modulación de Ancho de Pulso) .....	66
Ejercicio 5 .....	69
Básicos: Las subrutinas en Arduino.....	72
Básicos: Las variables globales y locales .....	74
Ejercicio 6 .....	75
Los GPIO's como entradas analógicas (ADC).....	81
Básicos: El divisor de voltaje.....	83
Básicos: Estructura cíclica 'while'.....	84
Ejercicio 7 .....	85
Ejercicio 8.....	90
Sensor de efecto hall integrado .....	94
Ejercicio 9.....	96
Interrupciones y temporizadores en el ESP32.....	101
Básicos: Interrupciones en Arduino .....	101
Básicos: Temporizadores .....	102
Ejercicio 10.....	104
Guardar datos en la memoria Flash .....	108
Ejercicio 11.....	110
<b>Capítulo 3: Modo DeepSleep y sus funciones .....</b>	<b>116</b>
Los modos de consumo energético del ESP32 .....	117
DeepSleep con temporizadores .....	118
Ejercicio 12.....	120
DeepSleep con GPIO táctil (touch) .....	126
Ejercicio 13.....	126
DeepSleep con señales externas .....	131
Modo ext(0).....	131
Modo ext(1) .....	132
Ejercicio 14 .....	133
Básicos: Uso de definiciones (#define) .....	137
Ejercicio 15.....	137
Combinando métodos para despertar del modo DeepSleep .....	141



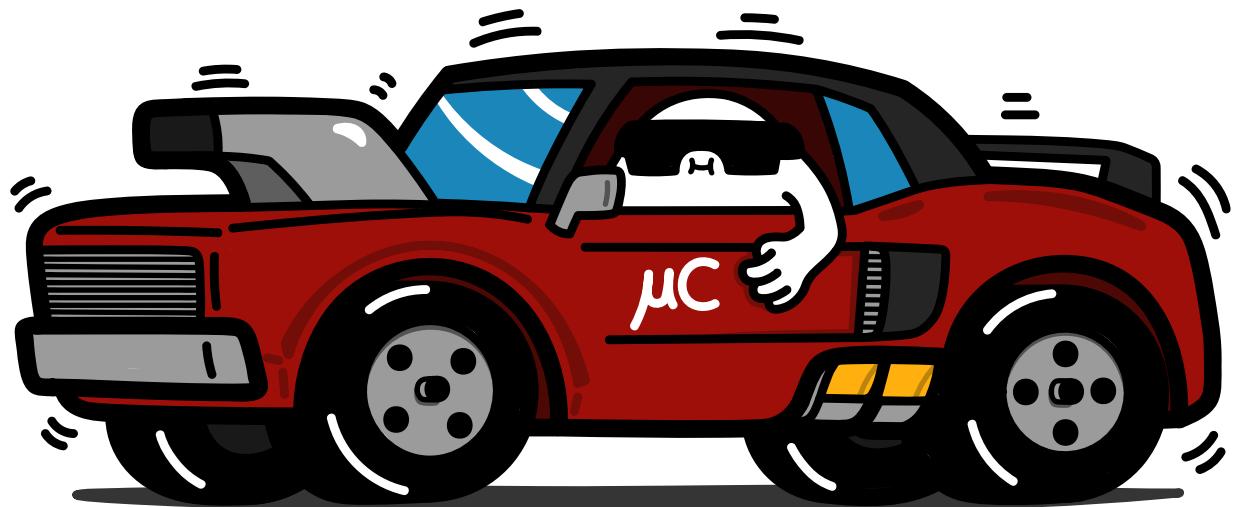
Cómo detectar el método que despertó al ESP32.....	142
Ejercicio 16.....	143
<b>Cómo usar los otros modos de consumo energético .....</b>	<b>148</b>
ModemSleep .....	148
LightSleep.....	149
Hibernación.....	149
Disminuir la velocidad del procesador principal .....	151
<b>Capítulo 4: Wifi y servidores Web .....</b>	<b>153</b>
El servidor web .....	154
HTML y CSS básicos para crear páginas web .....	155
El documento HTML.....	155
Mejoras con CSS .....	162
Servidor Web básico con ESP32.....	166
Ejercicio 17 .....	166
Manejando HTML desde el IDE de Arduino .....	179
Ejercicio 18.....	179
Aplicación práctica: control de relevadores.....	191
Ejercicio 19.....	192
Servidor protegido por contraseña.....	203
Ejercicio 20.....	203
Accede a tu servidor desde cualquier lugar .....	213
Set de caracteres para HTML y más sobre CSS .....	215
Ejercicio 21.....	217
Aplicación práctica: servidor asíncrono con sensor de temperatura y humedad (DHT11) .....	235
Otras funciones para HTML y JavaScript básico .....	235
Básicos: añadir librerías nuevas al IDE de Arduino .....	240
Ejercicio 22.....	242
<b>Capítulo 5: Bluetooth clásico y BLE .....</b>	<b>256</b>
¿Qué es Bluetooth? .....	257
Bluetooth de baja energía (BLE) .....	257



Protocolo GAP .....	257
Protocolo GATT .....	258
BLE en modo Servidor .....	259
BLE en modo Notificación .....	264
Aplicación práctica: servidor Bluetooth BLE con sensor de temperatura y humedad (DHT11) .....	271
Ejercicio 23.....	272
BLE en modo Cliente.....	281
Ejercicio 24 .....	282
Bluetooth clásico.....	293
Ejercicio 25.....	294
<b>Capítulo 6: La última y nos vamos .....</b>	<b>301</b>
Cómo usar los dos núcleos del ESP32 .....	302
Ejercicio 26.....	304
El convertidor digital a analógico (DAC) .....	311
Ejercicio 27 .....	314
Guarda datos en la Flash mediante SPIFFS.....	317
Circuito de potencia sencillo.....	329
Ejercicio 28.....	332
Fuentes de alimentación para tu tarjeta .....	334
Baterías de litio .....	334
Baterías comunes.....	337
Baterías de plomo/ácido y gel .....	338
Fuentes conmutadas.....	339
Cómo usar fuentes de más de 12V.....	340
¡No dañes a tu ESP32!, toma en cuenta estos consejos.....	342
¿Qué más aprender sobre el ESP32?.....	343
<b>Referencias.....</b>	<b>344</b>
<b>Anexos.....</b>	<b>345</b>
Programa DEMO .....	345



# Capítulo 1: ¿Qué es el ESP32?





## Primero, una breve introducción

El ESP32 es un microcontrolador ('micro' o representado como 'μC') de 32bits, desarrollado por el fabricante ESPRESSIF y es sucesor del ESP8266. El ESP32 se ha vuelto popular gracias a sus altas prestaciones de hardware, los sensores que integra y a sus radios de comunicación Wifi y Bluetooth, todo en un mismo módulo de bajo costo.



Módulo microcontrolador ESP32, modelo WROOM 32D.

Desde su antecesor, el ESP8266 (que sólo incluía comunicación Wifi), los micros de ESPRESSIF han estado recibiendo cierto soporte por parte de comunidades de programadores e ingenieros, para ser programados con el famoso IDE de Arduino, la "navaja suiza" de los microcontroladores.

La plataforma Arduino tiene un lenguaje de programación sencillo y fácil de aprender, muy similar al lenguaje C y C++. El IDE de Arduino permite programar no sólo a los ya populares Arduino UNO, Nano, Leonardo, Mini, Mega, etc. (pertenecientes a la familia de micros ATMEL-Microchip), también a microcontroladores de otros fabricantes y familias, incluyendo al ESP32. Esto facilita el uso y aprovechamiento de las capacidades del ESP32 sin recurrir a complicados entornos de programación, ahorrando mucho tiempo a la hora de desarrollar proyectos.

Las principales ventajas de utilizar el ESP32 (para tus proyectos basados en Arduino) son sus prestaciones superiores (comparado con los 'Arduinos' clásicos) y los periféricos e interfaces que incorpora, dando la posibilidad de actualizar tus proyectos existentes o desarrollar otros nuevos, para aprovechar la comunicación Wifi y Bluetooth e integrarlos al 'Internet de las cosas' (IoT), servicios en línea, 'Home Automation' o incluso a 'Inteligencias Artificiales en la Nube'.



## Las características del micro

Existen muchos modelos de tarjetas ESP32, ya que ESPRESSIF proporciona libremente los circuitos básicos para que uno mismo pueda diseñar su tarjeta, pero casi todos los modelos comerciales tienen las mismas funciones y capacidades, ya que la mayoría se basan en los módulos ESP32 WR0OM.



*Algunos modelos de tarjetas ESP-32 con el módulo WR0OM.*

Esta guía se centra en la tarjeta ESP32 KYOMI Dev Board, que tiene algunos añadidos útiles y está basada en el módulo WR0OM 32D, por lo que ésta guía también te será de utilidad si tienes otro modelo de tarjeta con módulo WR0OM.



*Tarjeta ESP32 KYOMI Dev Board.*



## Especificaciones de hardware del ESP32 WROOM

En la siguiente tabla se muestran las especificaciones principales de los módulos ESP32 WROOM:

<b>Procesador</b>	Xtensa Dual-Core de 32-bits con una velocidad de hasta 240MHz
<b>Comunicación Wifi</b>	Wifi 802.11b/g/n de 2.4GHz
<b>Comunicación Bluetooth</b>	Bluetooth 4.2 compatible con BLE
<b>Memoria Flash integrada</b>	4MB, 8MB o 16MB según el módulo (WROOM de 4MB para la tarjeta KYOMI)
<b>Memoria ROM</b>	448KB para funciones de arranque y del sistema
<b>Memoria RAM</b>	<ul style="list-style-type: none"><li>• 520KB de SRAM para datos y ejecución de instrucciones</li><li>• 8KB de SRAM para el bloque RTC</li></ul>
<b>Periféricos integrados y otras comunicaciones</b>	<ul style="list-style-type: none"><li>• Controlador para tarjeta SD y periféricos SDIO</li><li>• Comunicación UART</li><li>• Comunicación SPI</li><li>• Comunicación I<sup>2</sup>C,</li><li>• Generador de PWM</li><li>• Comunicación I<sup>2</sup>S</li><li>• Comunicación CAN</li><li>• Convertidores analógico-digital y digital-analógico</li><li>• Comunicación TWAI®</li></ul>
<b>Sensores integrados</b>	<ul style="list-style-type: none"><li>• Sensores touch capacitivos</li><li>• Sensor de efecto Hall</li></ul>
<b>Temperatura de operación</b>	-40°C hasta 85°C



**Sobre el procesador:** los módulos WROOM tienen un procesador de dos núcleos que permite ejecutar dos o más tareas al mismo tiempo (similar a como lo hace una computadora). Otros módulos (como el ESP32 WROVER) también integran un procesador con las mismas características que los módulos WROOM.

**NOTA:** la principal diferencia entre los módulos WROOM y WROVER, es el tamaño mayor del WROVER y que integra un módulo de memoria PSRAM adicional, además de tener radios de comunicación un poco más estables. Fuera de esto, ambos comparten las mismas características básicas.

Además de los dos núcleos principales, el ESP32 cuenta con un tercer núcleo llamado bloque RTC, el cual funciona con muy poca energía y sólo puede ejecutar instrucciones sencillas. El bloque RTC está formado por un Procesador de Ultra Bajo Consumo Energético (ULP), un bloque pequeño de memoria RAM (8KB) y una unidad de medición de fasores (PMU), que se utiliza para detectar señales de entrada.

El bloque RTC se puede utilizar cuando se programa al ESP32 para activar modos de trabajo como Hibernación y ‘DeepSleep’ (esto lo veremos en otro capítulo).

**NOTA:** los módulos WROOM 32S son ‘Single-Core’, por lo que sólo tienen un núcleo principal y el bloque RTC. El resto de módulos WROOM y WROVER cuentan con dos núcleos.

**Sobre la memoria:** los módulos WROOM tienen 512KB de memoria RAM estática (SRAM), que puede ser utilizada por los programas para mover datos y ejecutar instrucciones (un poco menos teniendo en cuenta que se reserva algo de RAM para las funciones básicas). Los módulos WROVER tienen un módulo adicional de 4MB de RAM pseudo estática (PSRAM), que sirve para ejecutar programas que requieran procesar y mover muchos datos. El bloque RTC cuenta con su propio módulo de 8KB de SRAM.

El bloque de memoria Flash de 4MB (8MB o 16MB cuando lo indica el modelo del módulo) puede ser utilizado para guardar datos de nuestros programas de manera permanente, similar a como lo hacen las memorias Flash USB. Los datos guardados permanecen intactos, aún cuando se reinicia la tarjeta o se desconecta de la alimentación.



## Características eléctricas del ESP32 WROOM

En la siguiente tabla se muestran las características eléctricas de los módulos ESP32 WROOM:

Parámetro	Mínimo	Máximo
Voltaje de alimentación del módulo	3V	3.6V
Corriente consumida por el módulo	20mA	1100mA
Voltaje de detección LOW en los pines GPIO (modo digital)	-0.3V	0.8V
Voltaje de detección HIGH en los pines GPIO (modo digital)	2.4V	3.3V
Corriente de entrada en los pines GPIO	-	50nA
Voltaje de salida HIGH en los pines GPIO	2.64V	3.3V
Voltaje de salida LOW en los pines GPIO	-	0.3V
Corriente de salida entregada por cada GPIO	40mA	
Resistencia PULL-UP interna de cada GPIO	45KΩ	
Resistencia PULL-DOWN interna de cada GPIO	45KΩ	

**Sobre el voltaje y corriente de alimentación:** el voltaje de alimentación es proporcionado por el regulador de 3.3V de la tarjeta.

La corriente mínima que puede consumir el módulo WROOM es de 15µA (microamperios), cuando funciona en el modo ‘DeepSleep’ mínimo (incluso 10µA en modo Hibernación). Sin embargo, dependiendo del diseño de la tarjeta, el consumo mínimo real generalmente ronda los 15mA (miliamperios).

El consumo de corriente máxima del módulo, considerando que todos los pines trabajen como salidas, varía entre 1100mA hasta 1500mA, dependiendo de la carga de trabajo de los radios Wifi o Bluetooth. Según el diseño de la tarjeta, puede consumir un máximo de 2000mA (2A) al alimentarla por el puerto micro-USB.



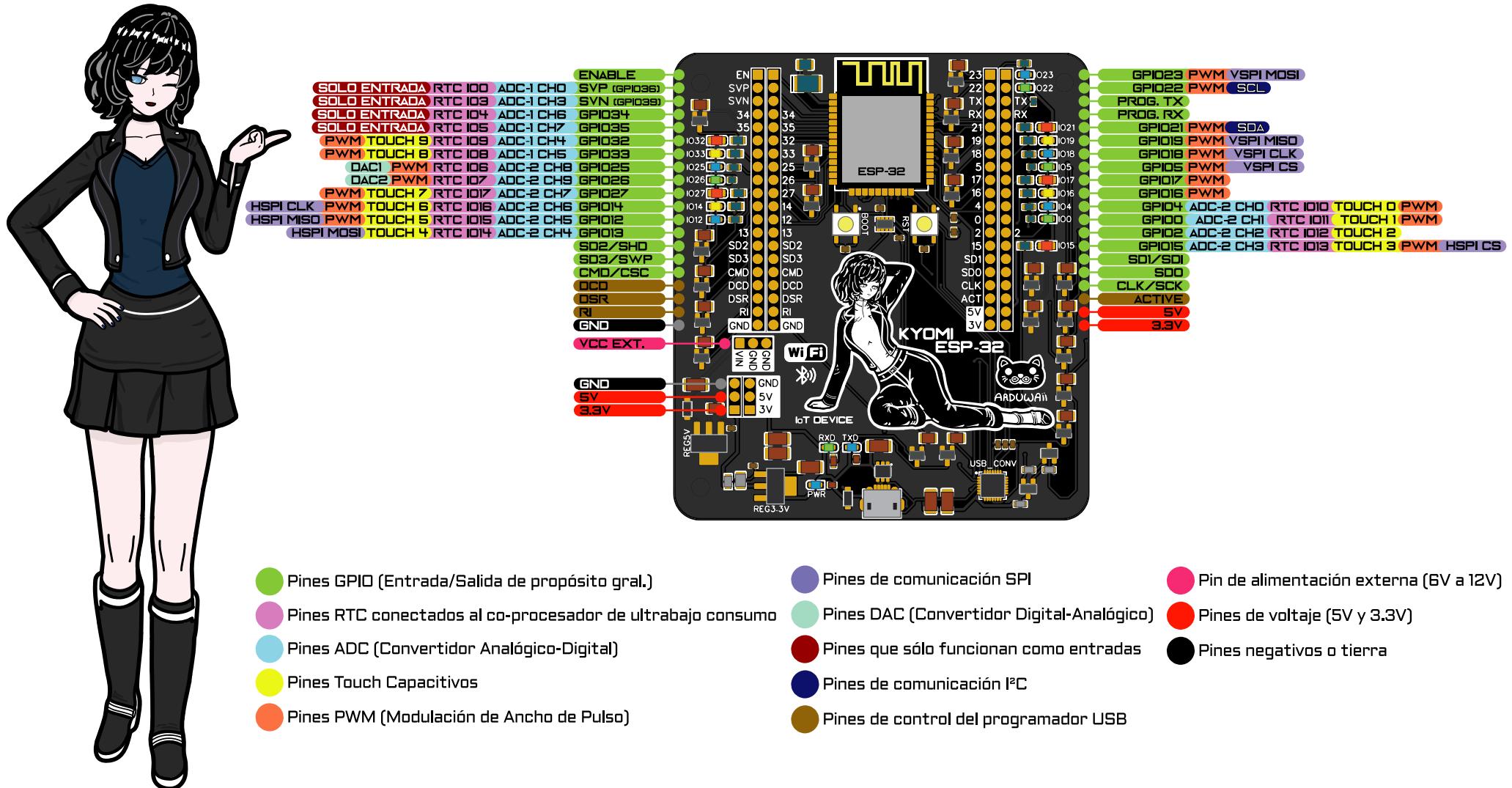
**Sobre el voltaje y corriente de los pines GPIO:** los pines de entrada/salida de propósito general (GPIO) pueden detectar señales analógicas y digitales entre 0V y 3.3V, por lo que sus niveles están entre estos voltajes, mientras que otras tarjetas (como el Arduino UNO) tienen niveles lógicos de 0V a 5V.

La mayoría de sensores y periféricos que trabajan con comunicaciones digitales (como SPI o I<sup>2</sup>C) funcionan sin problemas con el ESP32. Pero los sensores que generan señales analógicas de 0V a 5V, diseñados para las tarjetas Arduino comunes, pueden dañar los GPIO del ESP32. Estas señales se pueden adaptar a los niveles de 0V a 3.3V con un divisor de voltaje de dos resistencias (esto lo veremos en el siguiente capítulo), y con esto podremos usar la mayoría de sensores comunes diseñados para las tarjetas Arduino clásicas.

En modo de entrada, la corriente máxima que puede circular hacia un GPIO es de 50nA (nanoamperios), por lo que prácticamente no se verán afectadas las señales generadas por sensores de pequeña señal, con voltaje y corriente débiles.

La corriente máxima que puede entregar cada GPIO, cuando trabaja como salida, es de 40mA. Por lo tanto, no se podrán alimentar cargas grandes o dispositivos que consuman más de 40mA. Al igual que con casi todos los microcontroladores comerciales, si debemos controlar cargas de gran consumo energético, necesitaremos de algún ‘circuito de potencia’ externo, que será controlado por el propio micro (veremos un ejemplo en el último capítulo).

## PINOUT de la tarjeta ESP32 KYOMI





La tarjeta ESP32 KYOMI tiene el mismo orden de pines de las tarjetas clónicas comerciales. En este modelo, cada pin tiene la opción para utilizar cables tipo ‘Dupont’ o jumpers de conexión macho y hebra, además de contar con algunos pines de alimentación extras y otros que son útiles para realizar funciones avanzadas. A continuación, se describen más a detalle las funciones de los pines:

- **Pines GPIO:** son todos los pines de entrada/salida de propósito general conectados al módulo ESP32.
- **Pines RTC:** son los pines que están conectados al coprocesador del bloque RTC para recibir/enviar señales cuando el ESP32 trabaja en los modos de bajo consumo energético. Generalmente, estos pines se usan para despertar al micro de los modos ‘DeepSleep’ o Hibernación y así reactivar los núcleos principales y los radios de comunicación.
- **Pines ADC:** son pines conectados a los convertidores analógico-digital (Analog to Digital Converter) ADC-1 y ADC-2 de 12 bits. Los convertidores ADC-1 y ADC-2 a su vez se dividen en canales independientes para medir varias señales analógicas.
- **Pines DAC:** son pines conectados al convertidor digital-analógico (Digital to Analog Converter) de 8 bits, para generar señales analógicas.
- **Pines Touch:** son pines táctiles (touch) capacitivos que pueden detectar las pequeñas cargas eléctricas de los dedos o la mano al tocarlos o al tocar una superficie metálica conectada a uno de estos pines.
- **Pines PWM:** son aquellos que pueden generar señales de modulación de ancho de pulso (Pulse Width Modulation), con frecuencia y resolución de bits configurables (hasta 16 bits de resolución y frecuencia de hasta 40MHz). Sólo los GPIO34 al GPIO39 no pueden generar señales PWM.
- **Pines de sólo entrada:** son los únicos pines GPIO que no pueden generar señales de ningún tipo, únicamente funcionan como entradas, y tampoco cuentan con resistencias PULL-UP o PULL-DOWN internas.



- **Pines de comunicación SPI:** son los pines utilizados para controlar periféricos que trabajan con bus SPI (Serial Peripheral Interface).
- **Pines de comunicación I<sup>2</sup>C:** son los pines utilizados para controlar periféricos que trabajan con bus I<sup>2</sup>C (Inter-Integrated Circuit).
- **Pines de control del programador USB:** son pines adicionales conectados al programador USB a TTL CP2102 (independiente al ESP32), reservados para ciertas funciones avanzadas de este chip. Prácticamente no se usan, a menos que requieras de alguna función específica del CP2102.
- **Pines de voltaje:** son pines que se pueden utilizar para alimentar sensores y periféricos externos que trabajen con 5V o 3.3V. El grupo de pines de 5V puede entregar hasta 800mA de corriente total y el grupo de 3.3V hasta 500mA.
- **Pin VIN:** la tarjeta KYOMI se puede alimentar con una fuente de voltaje externa de 6V hasta 12V a través de este pin, sin depender de la alimentación del puerto micro-USB. Puede soportar hasta 20V de entrada, pero no se recomienda. Esto es debido a que, al superar los 12V, el regulador de 5V comenzará a calentarse y las salidas de 5V y 3.3V se volverán inestables. Al utilizar baterías, se recomienda agregar algún circuito de protección para no reducir la vida útil de las mismas (veremos unos ejemplos en el último capítulo).
- **Pines negativos o GND:** son los pines de tierra común o negativo (Ground). Cuando utilices alimentación externa, realiza una buena conexión entre GND y la conexión negativa de la fuente, para evitar generar ruido eléctrico que afecte a las señales de los GPIO.

NOTA: los pines SD2/SHD, SD3/SWP, CMD/CSC, SD1/SDI, SDO/SDO y CLK/SCK (GPIO9, GPIO10, GPIO11, GPIO8, GPIO7 y GPIO6 respectivamente) están conectados a la memoria Flash interna (que se comunica con el procesador usando bus SPI). Sólo se pueden utilizar para propósitos que tengan que ver con el manejo de memorias Flash externas y no pueden ser usados como GPIO's comunes.



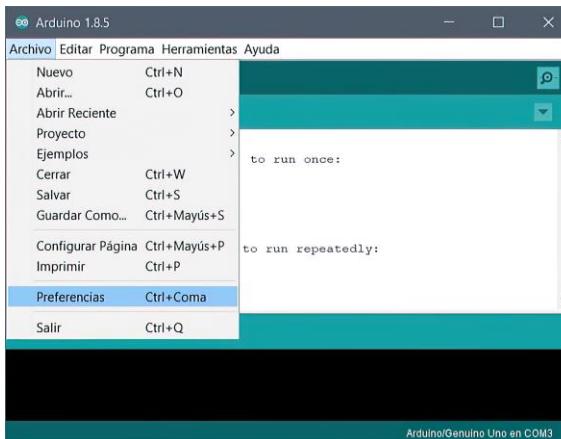
## Instala todo lo necesario

Antes de comenzar a programar tu tarjeta, debes descargar los complementos necesarios para el IDE de Arduino. Si aún no tienes instalado el IDE, puedes descargarlo de la página oficial de Arduino, es completamente gratis. También es recomendable mantener actualizado el IDE a la última versión.

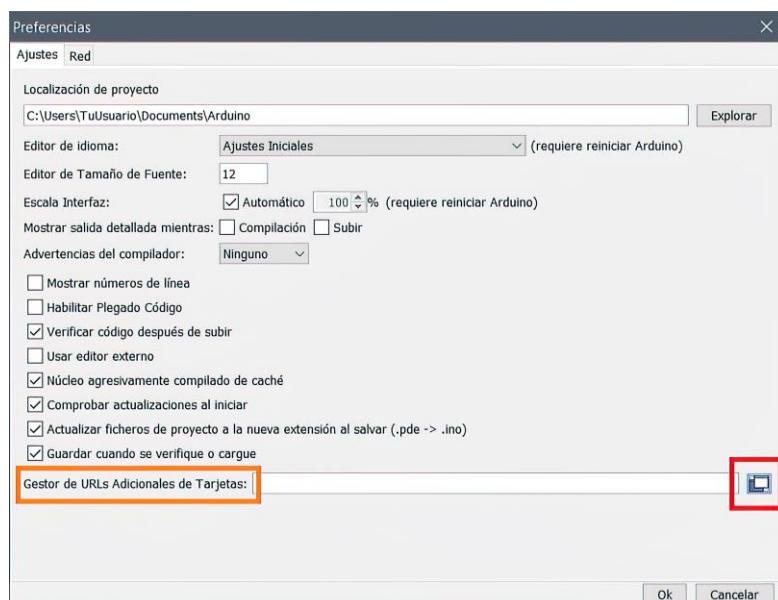
Si vas a instalar los complementos por primera vez, continúa con la guía de ‘Instalación nueva’. Si ya tenías instalados los complementos con el método manual, primero revisa la guía ‘Limpiar una instalación anterior’. Las guías son para Windows, pero los pasos son similares en MacOS y Linux.

### Instalación nueva

1. Abre el IDE de Arduino, ve a la pestaña ‘Archivo’ y entra en la opción ‘Preferencias’:

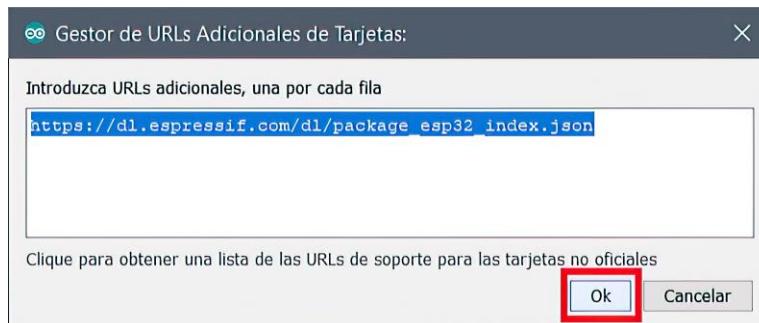


2. Dentro de Preferencias, abre la ventana del ‘Gestor de URL’s’:

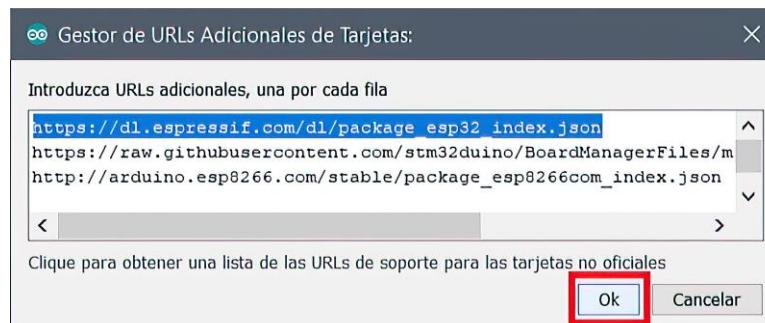




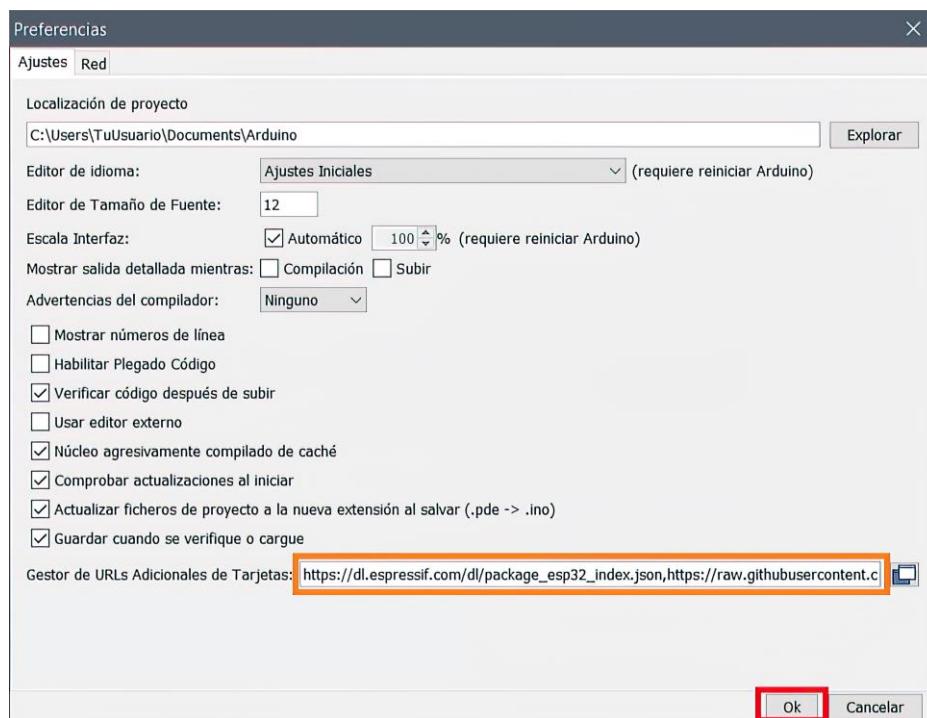
3. En la ventana del Gestor de URL's, agrega la siguiente dirección web: [https://dl.espressif.com/dl/package\\_esp32\\_index.json](https://dl.espressif.com/dl/package_esp32_index.json)



Guarda los cambios. Si ya tienes URL's adicionales para otras tarjetas, simplemente separa por renglones las direcciones nuevas:

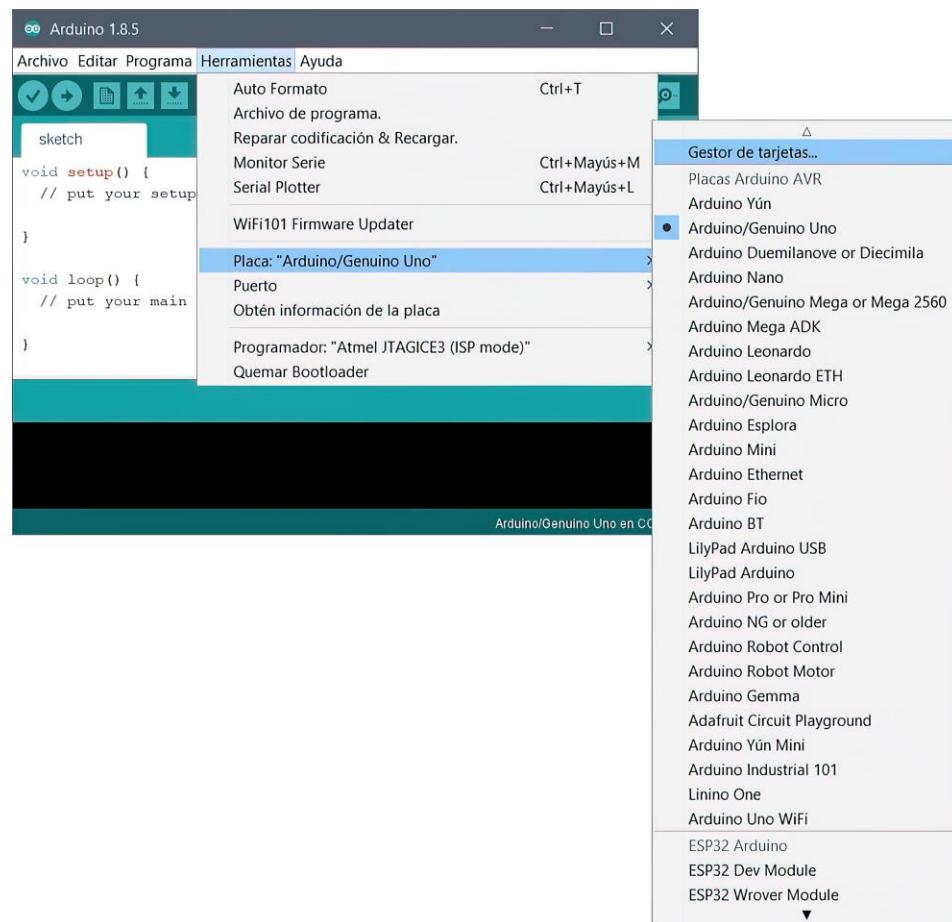


Deberá aparecer la URL nueva, guarda los cambios:

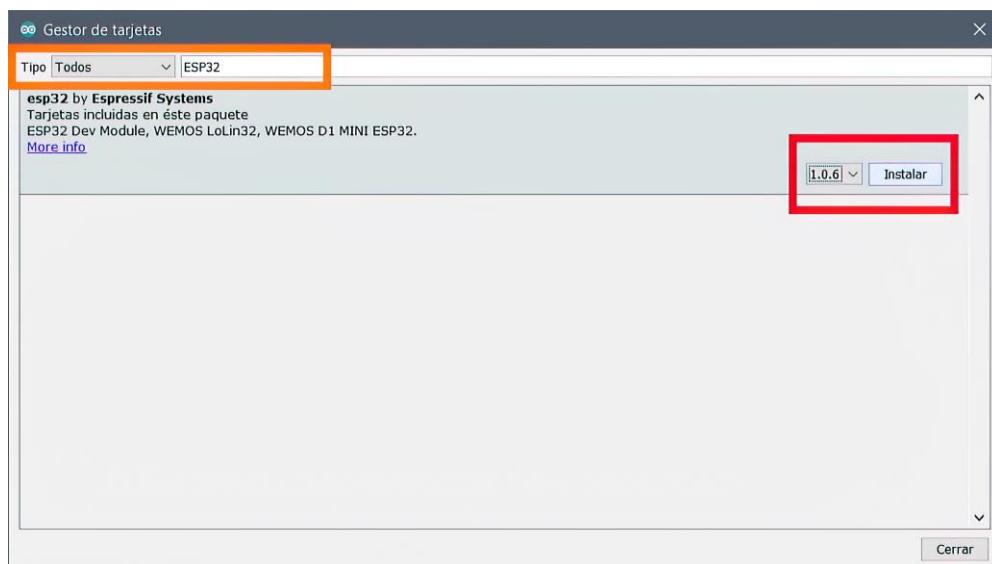




4. De nuevo en la ventana principal, ve a la pestaña ‘Herramientas’ > ‘Placa’ y abre el ‘Gestor de Tarjetas’:

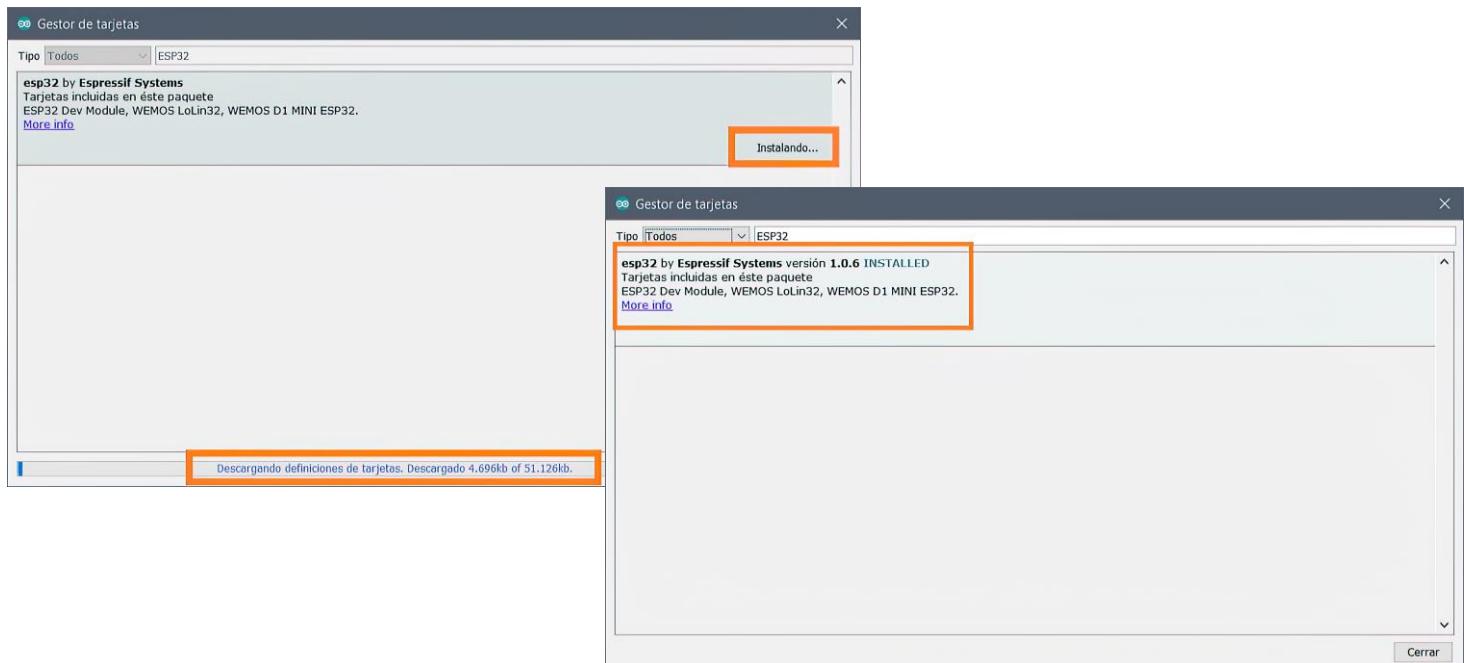


5. En la ventana del Gestor, en el buscador escribe **ESP32**. Deberá aparecer el paquete de tarjetas ‘esp32 de Espressif Systems’. Instala la versión más reciente:

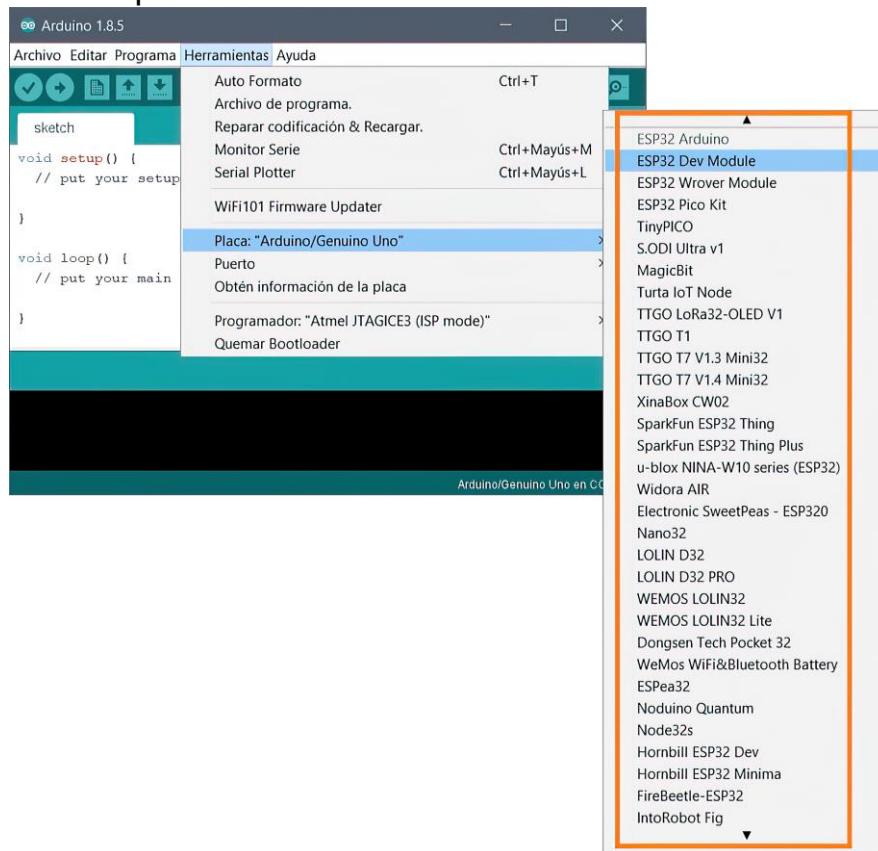




Se descargará el paquete y al terminar aparecerá instalado:



## 6. Si todo se instaló correctamente, los modelos de tarjetas ESP32 compatibles aparecerán en el selector de ‘Placa’:

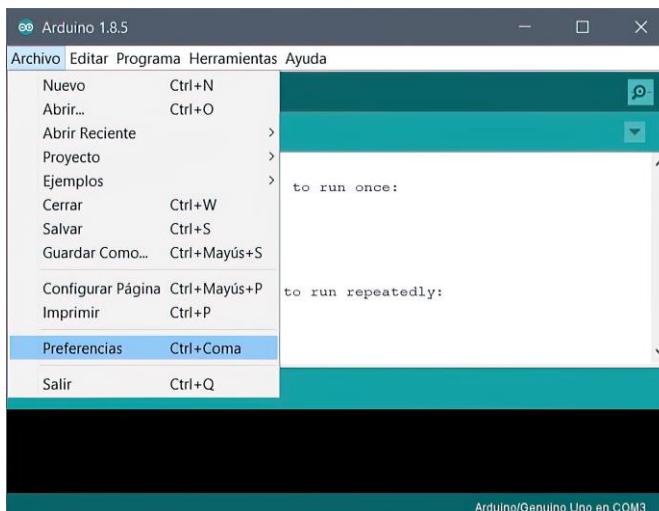




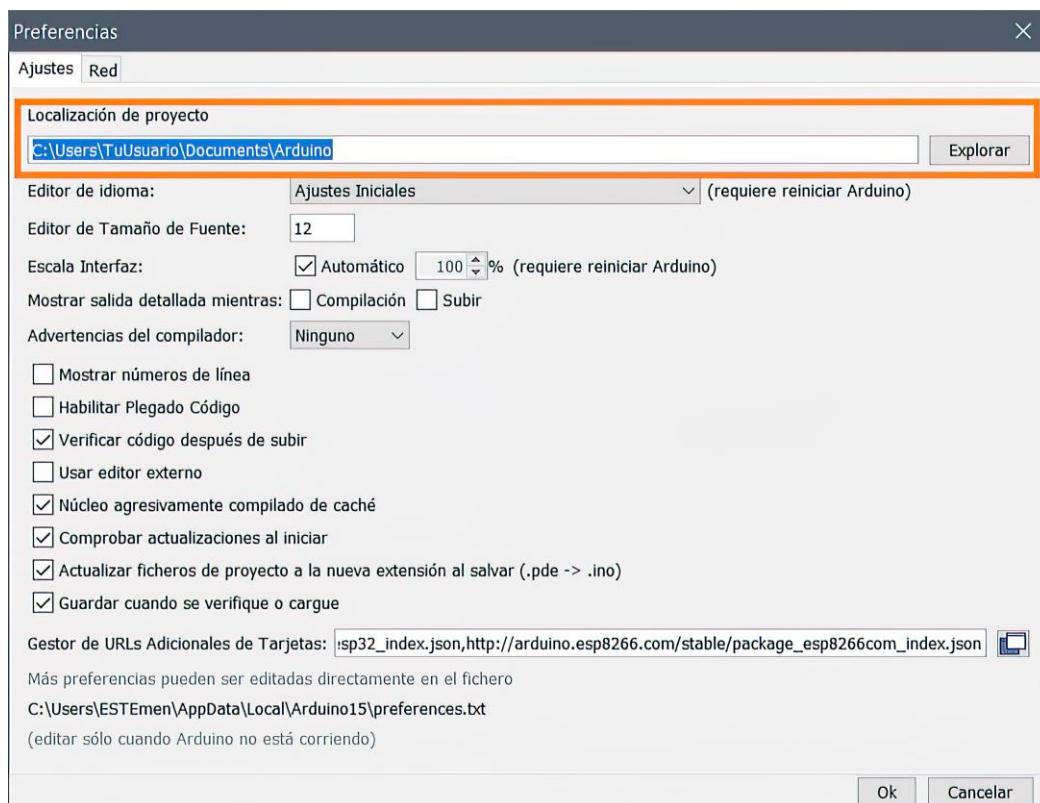
## Limpiar una instalación anterior

Si instalaste los complementos del ESP32 manualmente con Git GUI, deberás borrar la carpeta 'espressif' para eliminarlos del IDE de Arduino:

1. Para encontrar la carpeta espressif, abre el IDE de Arduino y ve a la pestaña Archivo, entra en la opción Preferencias:

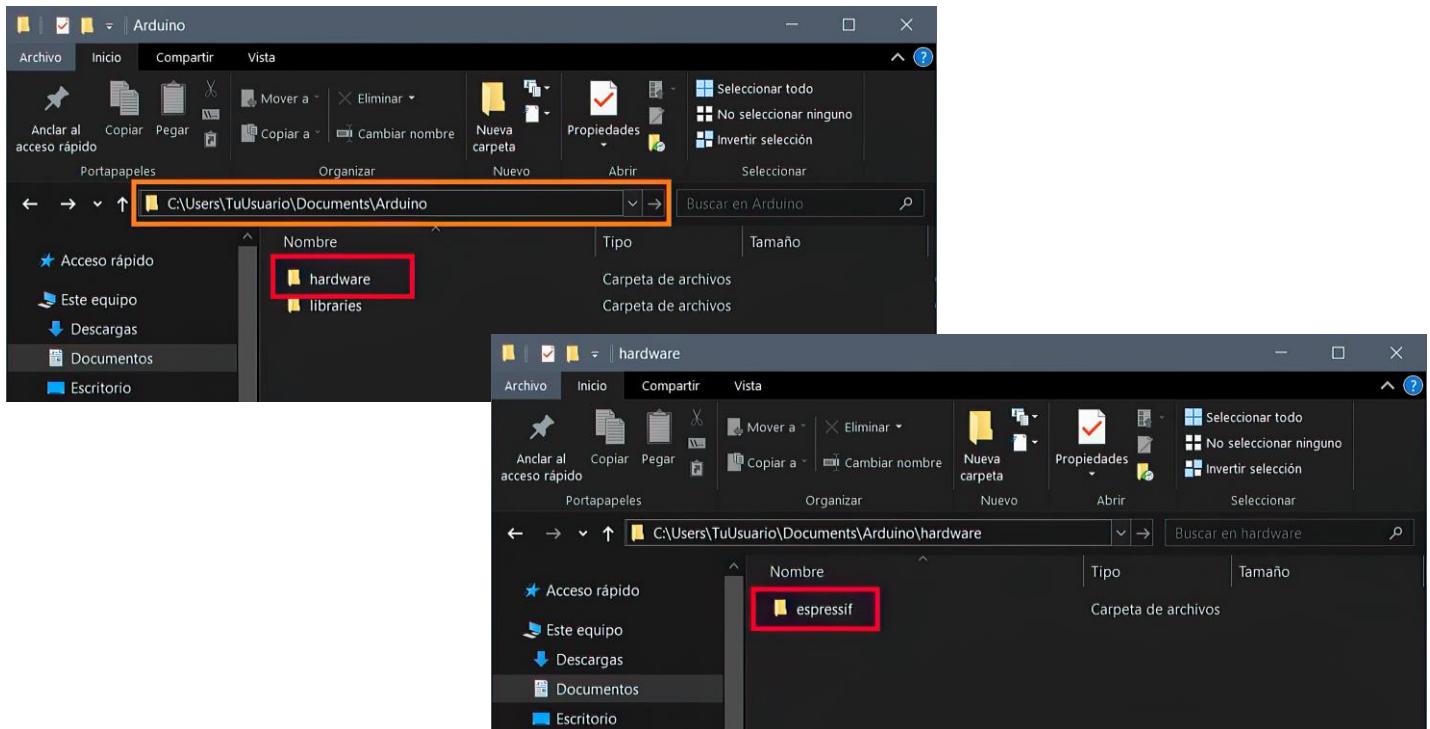


2. Dentro de Preferencias, copia la ruta de 'Localización de Proyecto' y ábrela en el explorador de Windows:





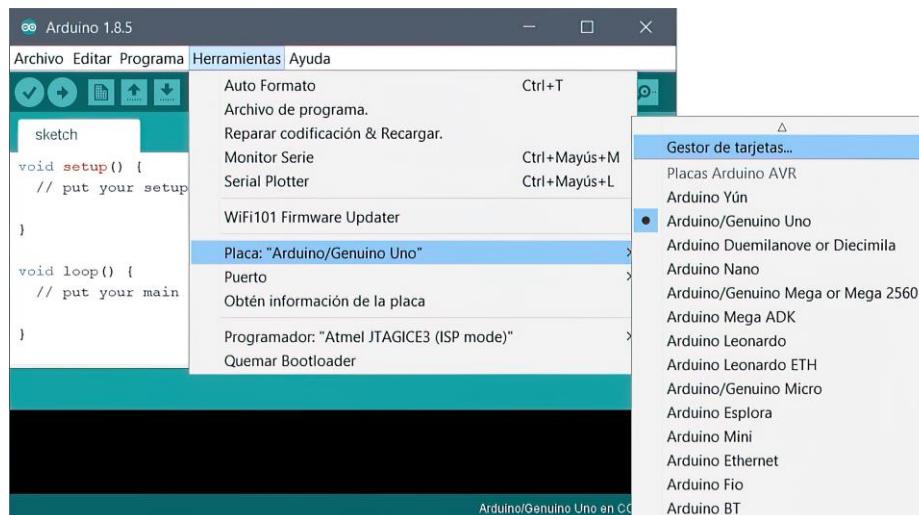
3. En la carpeta ‘Arduino’, abre la carpeta ‘hardware’ y borra la carpeta espressif:



Hecho esto, continúa con la guía de ‘Instalación nueva’ para descargar los complementos actualizados.

Si tienes complementos desactualizados que fueron instalados con el método de ‘Instalación nueva’, actualízalos de la siguiente manera:

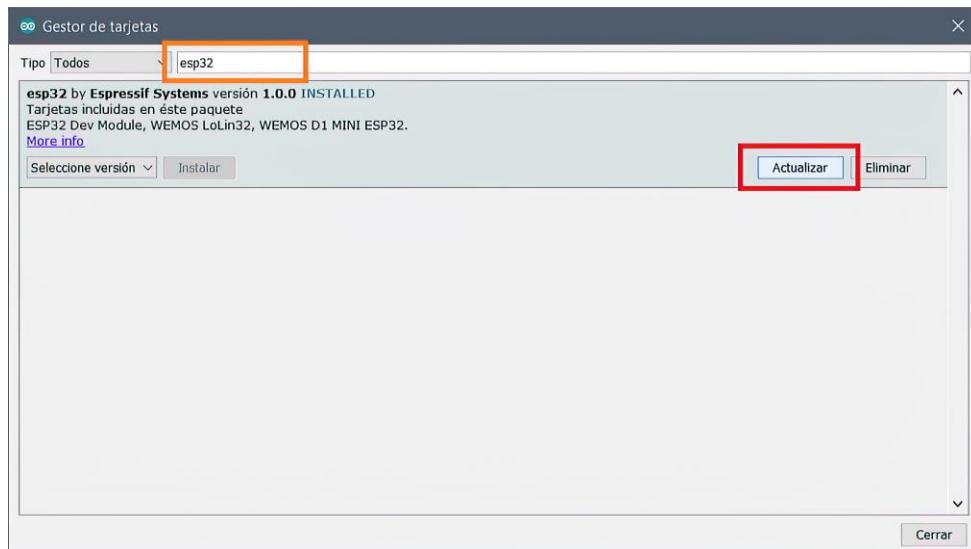
1. Ve a la pestaña Herramientas > Placa y abre el Gestor de Tarjetas:



2. En la ventana del Gestor, en el buscador escribe ESP32. Deberá aparecer el paquete de tarjetas ‘esp32’ que ya tienes instalado.



Simplemente da clic en actualizar y automáticamente se descargará los complementos nuevos, reemplazando los anteriores:



Si requieres instalar una versión distinta, selecciónala en el menú de versiones y da clic en instalar, esto también reemplazará la versión previamente instalada:



Una vez instalados los complementos, ya puedes empezar a programar tu tarjeta. Recuerda mantenerlos actualizados para obtener funciones y librerías nuevas para el ESP32.

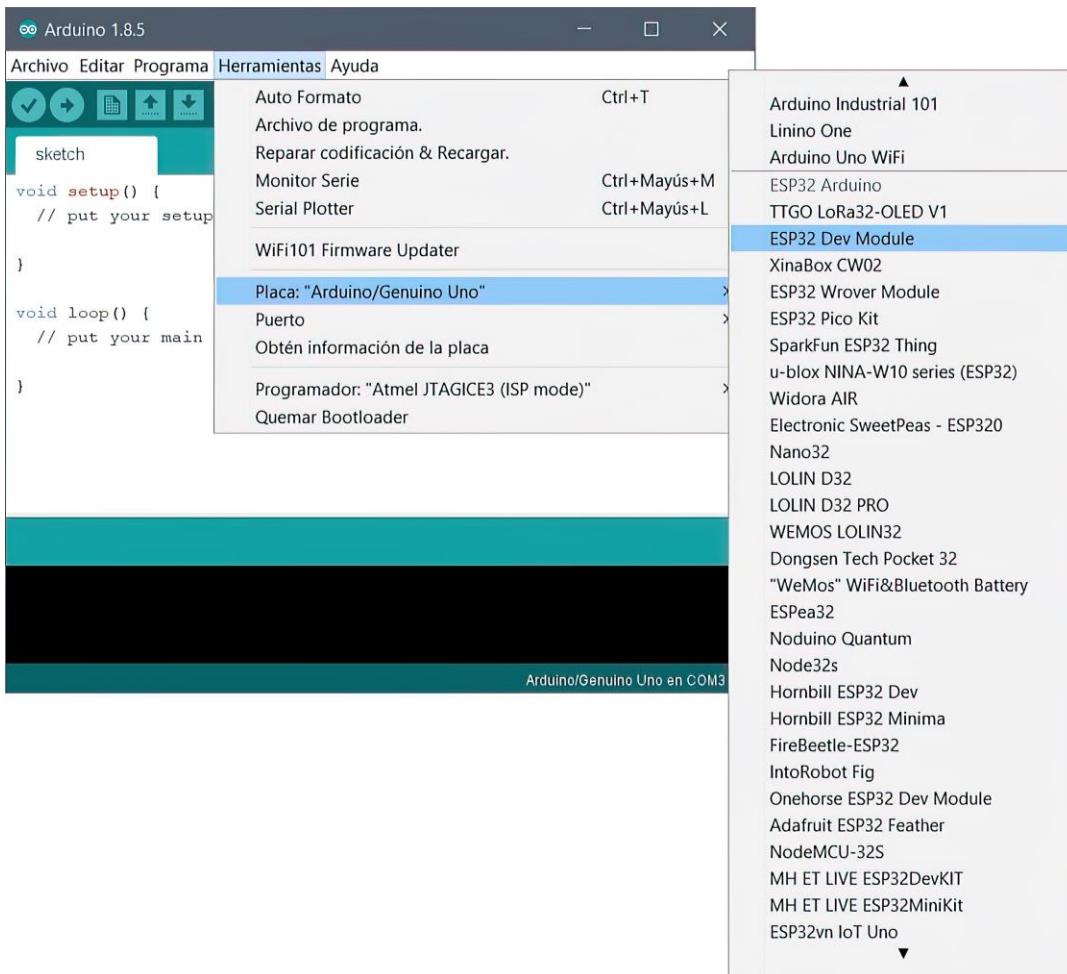


## Primera prueba

Vamos a realizar una prueba sencilla de nuestro ESP32. Si es la primera vez que usas tu tarjeta KYOMI, entonces debería estar corriendo el programa DEMO. El código (sketch) del programa DEMO lo encontrarás en los Anexos.

Usaremos uno de los ejemplos que vienen con el paquete de complementos para el ESP32 (usa las configuraciones de los pasos 1 y 2 para subir tus programas al ESP32):

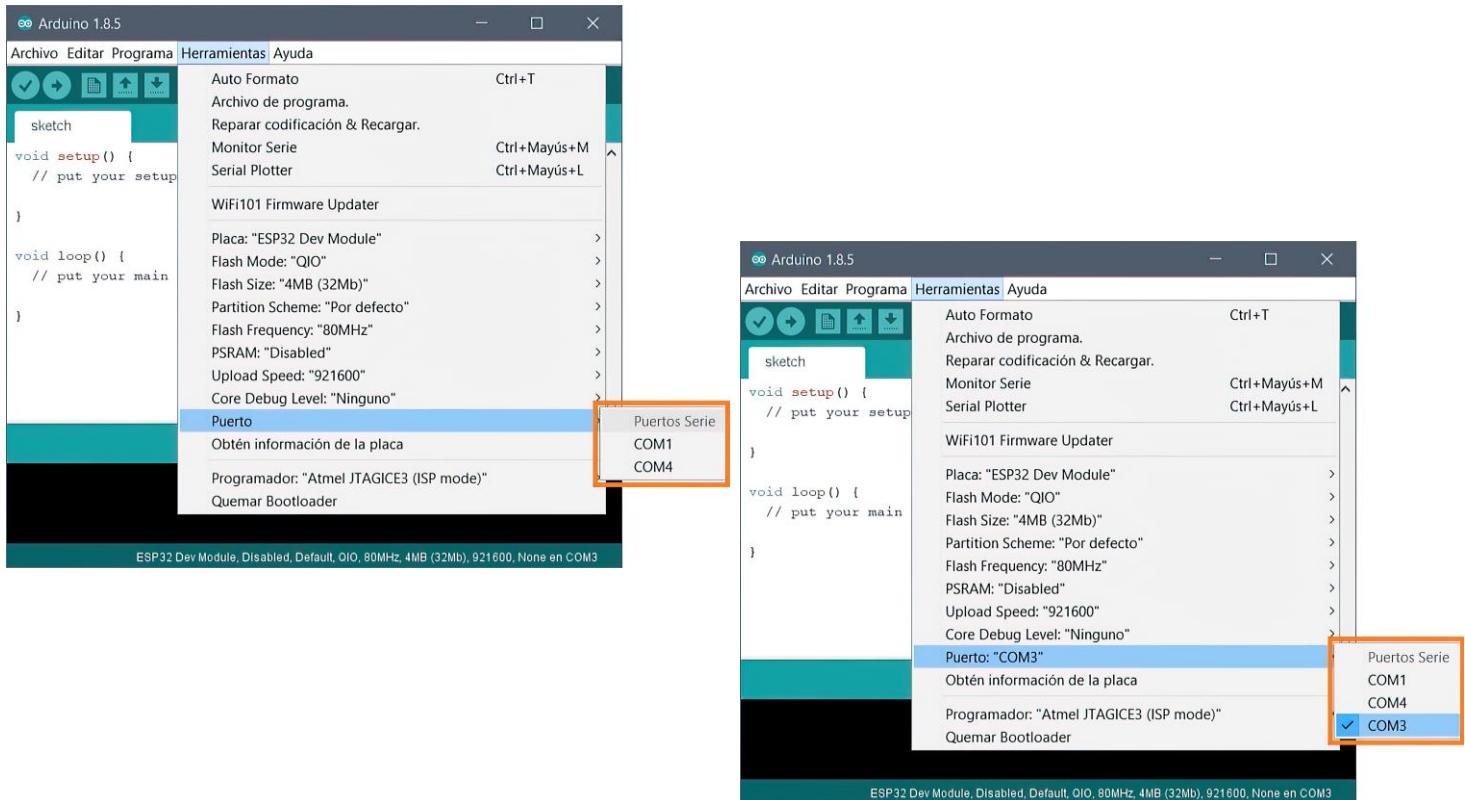
1. En la pestaña Herramientas, elige el modelo de tarjeta que tengas. Si tienes la tarjeta KYOMI, elige ‘ESP32 Dev Module’. Esta opción sirve para la mayoría de tarjetas, úsala si no conoces el modelo exacto de tu tarjeta:



2. Ahora toca elegir el puerto comunicación. Para identificar el puerto COM que es asignado a tu tarjeta (al momento de conectarla a la computadora por USB), revisa primero cuáles dispositivos que aparecen en la sección ‘Puerto’ (sin conectar la



tarjeta). Ahora conecta tu tarjeta y vuelve a revisar la sección Puerto, el nuevo dispositivo COM que aparezca será el de tu tarjeta (se asignará otro cuando cambies de puerto USB). Asegúrate de usar un cable USB de datos y no de sólo carga.

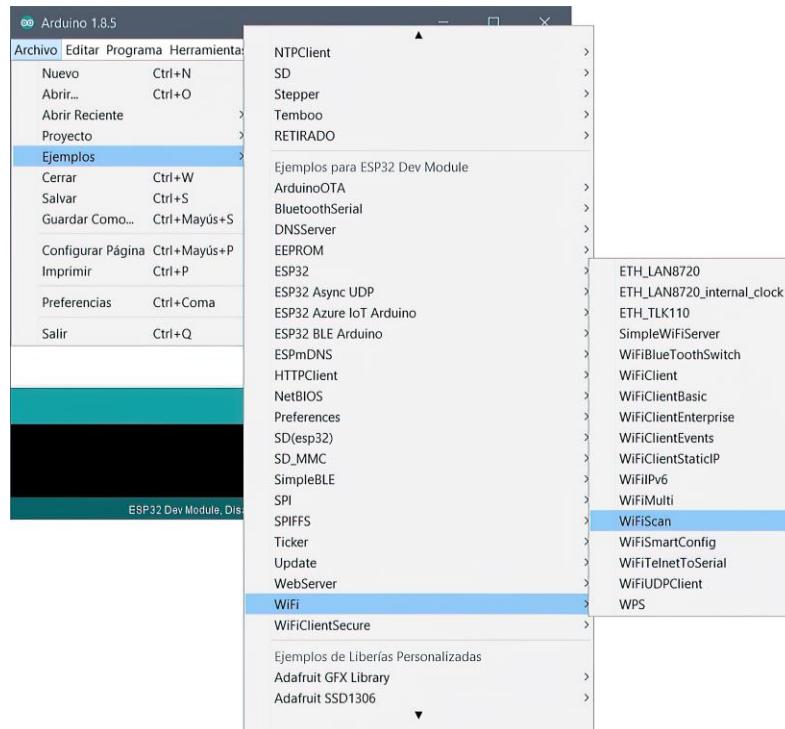


### Dejaremos la configuración por defecto de la tarjeta:

- Flash Mode “QIO”.
- Flash Size “4MB”, lo puedes cambiar a 8MB o 16MB si el modelo de módulo indica que tiene 8MB o 16MB.
- Partition Scheme “Por defecto”.
- Flash Frequency “80MHz”.
- PSRAM “Disabled”, puedes habilitarla en tarjetas con módulo WROVER.
- Upload Speed “921600”, puedes elegir otra velocidad de subida si lo requieres.
- Core Debug Level “Ninguno”.

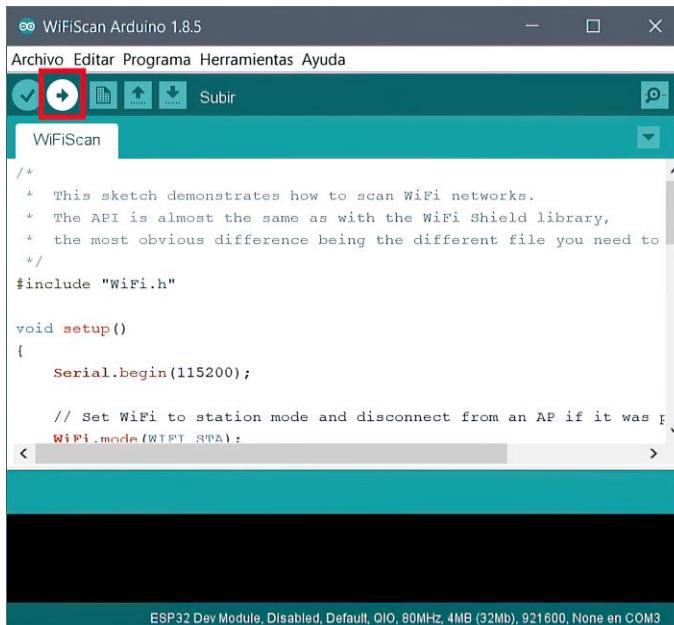


3. Busca el programa de ejemplo en la pestaña Archivo > Ejemplos > Ejemplos para ESP32 Dev Module > WiFi, da clic en el ejemplo llamado ‘WiFiScan’, se abrirá una ventana nueva con el sketch del ejemplo:



‘WiFiScan’ busca redes Wifi cercanas a la tarjeta.

4. Simplemente da clic en el botón ‘Subir’ para cargarlo a tu tarjeta (esto borra cualquier programa cargado anteriormente):





**IMPORTANTE:** para cargar un programa al ESP32, hay que cambiarlo al modo ‘Flasheo’. Primero hay que esperar a que compile el programa:

The screenshot shows the WiFiScan Arduino 1.8.5 IDE interface. The code editor contains the WiFiScan sketch. A progress bar at the bottom indicates the compilation status, with the text "Compilando programa..." (Compiling program...) displayed. The status bar at the bottom right shows "1 ESP32 Dev Module, Disabled, Default, QIO, 80MHz, 4MB (32Mb), 921600, None en COM3".

```
/*
 * This sketch demonstrates how to scan WiFi networks.
 * The API is almost the same as with the WiFi Shield library,
 * the most obvious difference being the different file you need to
 */
#include "WiFi.h"

void setup()
{
    Serial.begin(115200);

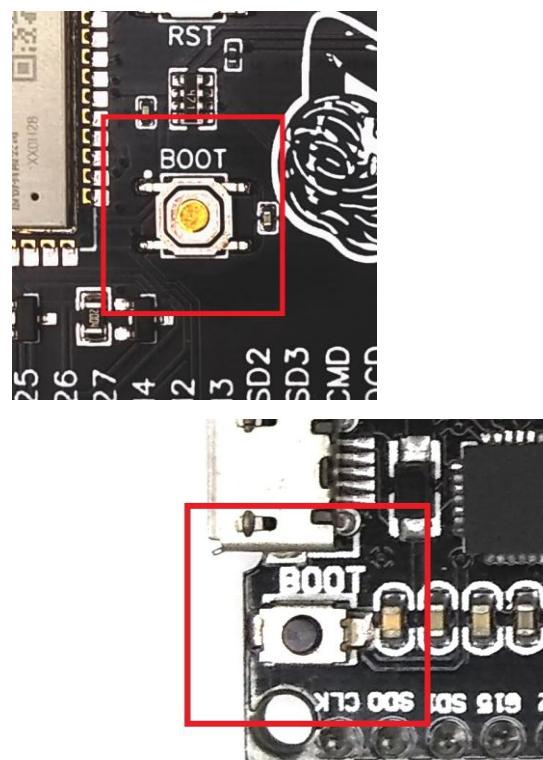
    // Set WiFi to station mode and disconnect from an AP if it was p
    WiFi.mode(WIFI_STA);
}


```

Cuando termine la compilación y trate de subir el programa, tienes que mantener presionado el botón ‘BOOT’ de la tarjeta para habilitar el modo de Flasheo:

The screenshot shows the WiFiScan Arduino 1.8.5 IDE interface. The code editor contains the WiFiScan sketch. A progress bar at the bottom indicates the upload status, with the text "Subiendo..." (Uploading...) displayed. The status bar at the bottom right shows "1 ESP32 Dev Module, Disabled, Default, QIO, 80MHz, 4MB (32Mb), 921600, None en COM3".

```
El Sketch usa 599086 bytes (45%) del espacio de almacenamiento de pro
Las variables Globales usan 41404 bytes (12%) de la memoria dinámica,
  global puntero 2 1
Connecting.....
```





Después de mantener presionado el botón un par de segundos, el programa se cargará rápidamente en la tarjeta. Una vez cargado el programa, aparecerá el siguiente mensaje:

```
WiFiScan Arduino 1.8.5
Archivo Editar Programa Herramientas Ayuda
WIFIscan
/*
 * This sketch demonstrates how to scan WiFi networks.
 * The API is almost the same as with the WiFi Shield library,
 * the most obvious difference being the different file you need to
 */
#include "WiFi.h"

void setup()
{
    Serial.begin(115200);

    // Set WiFi to station mode and disconnect from an AP if it was previously connected
    WiFi.mode(WIFI_STA);
}

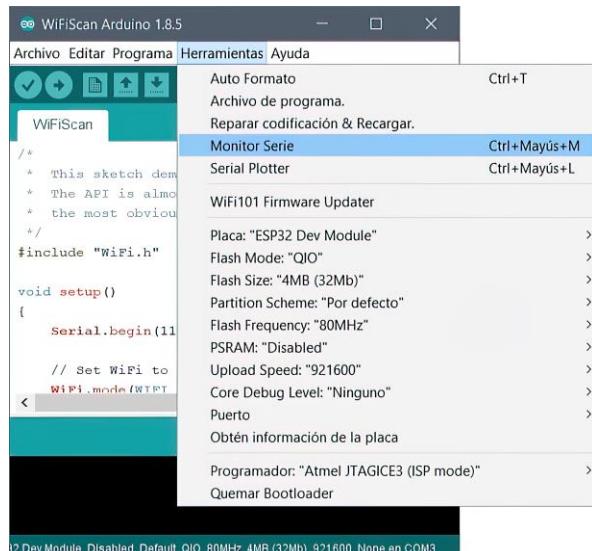
Subido
Writing at 0x00001000... (50 %)
Writing at 0x00008000... (100 %)
Wrote 599232 bytes (360801 compressed) at 0x00010000 in 7.0 seconds (Hash of data verified.
Compressed 3072 bytes to 144...

Writing at 0x00008000... (100 %)
Wrote 3072 bytes (144 compressed) at 0x00008000 in 0.0 seconds (Effectively 0.0 seconds (Hash of data verified.

Leaving...
Hard resetting via RTS pin...
1
ESP32 Dev Module, Disabled, Default, QIO, 80MHz, 4MB (32Mb), 921600, None en COM3
```

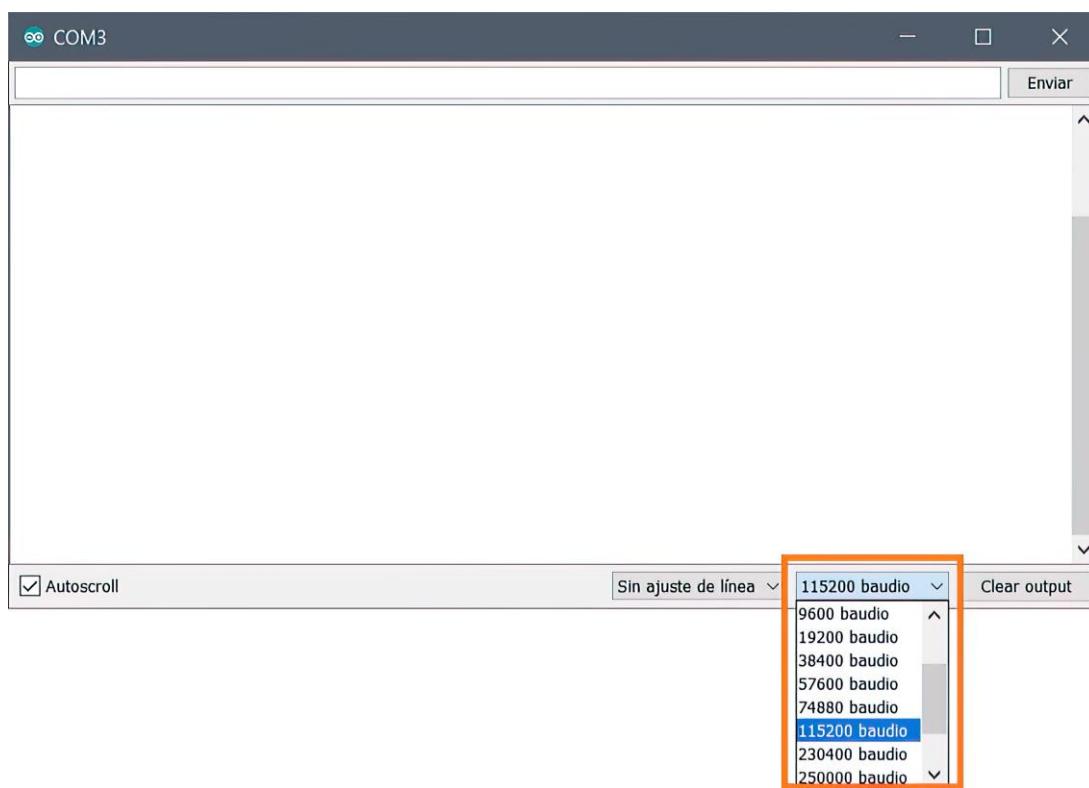
la tarjeta se reiniciará y comenzará a correr el programa.

## 5. Para ver las redes Wifi que ha detectado la tarjeta, hay que abrir el ‘Monitor Serie’, se encuentra en la pestaña Herramientas:

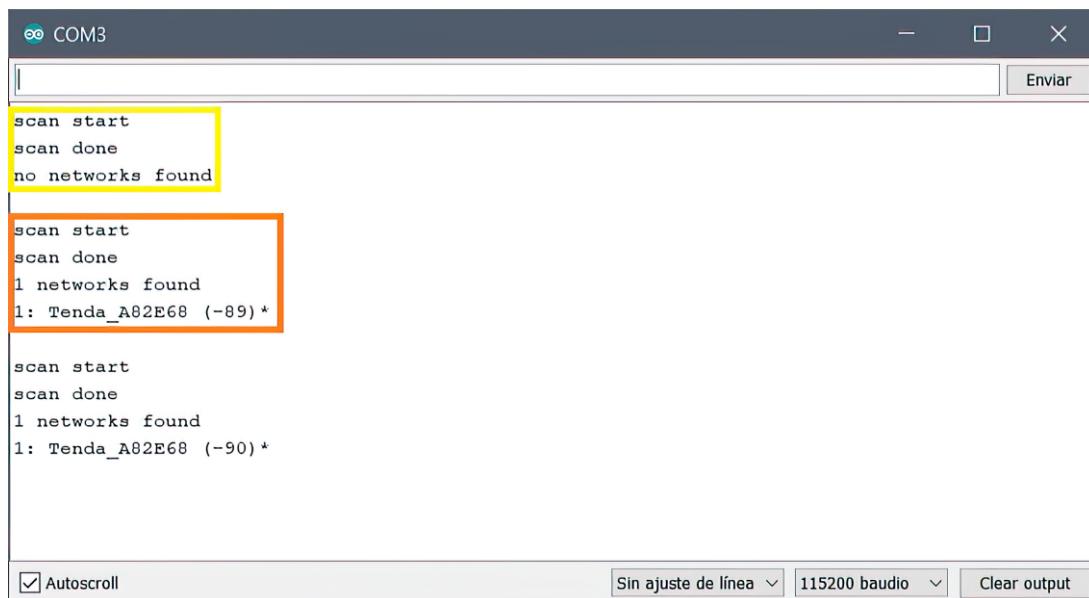




## 6. Configura la velocidad de comunicación a 115200 baudios:



Empezarán a mostrarse las redes Wifi que alcanza a detectar la tarjeta durante cada escaneo:



Con este programa de ejemplo, habrás comprobado que tu tarjeta funciona correctamente con el IDE de Arduino.

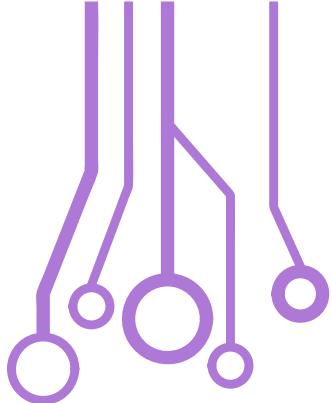


## Comprueba el controlador USB

Si tu tarjeta es reconocida por la computadora, pero no aparece un nuevo puerto COM en el IDE de Arduino, probablemente necesites descargar el controlador para el CP2102. Simplemente descarga los controladores del fabricante Silicon Labs e instálalos:

The screenshot shows a Google search results page. The search query is "CP2102 drivers". Below the search bar are two buttons: "Buscar con Google" and "Me siento con suerte". The page is in English, as indicated by the "Ofrecido por Google en: English" message. The main result is a link to the Silicon Labs website for the CP2102 Classic USB Bridges. The page features an image of the chip, the product name "CP2102 Classic USB Bridges", a "Buy Now" button, a "Data Sheet" link, and a red box highlighting the "Download USB to UART Bridge VCP Drivers >" link. Navigation links at the bottom include "SPECs", "TECH DOCS", "SOFTWARE & TOOLS" (which is underlined), "QUALITY & PACKAGING", and "COMMUNITY & SUPPORT".

Una vez instalados, vuelve a abrir el IDE y deberá aparecer un nuevo puerto COM correspondiente a tu tarjeta. También verifica que el cable USB de datos no esté dañado.



## Capítulo 2: Primeros pasos con los GPIO's





## Los GPIO's como entradas/salidas digitales

En esta sección veremos cómo utilizar los GPIO's para recibir y generar señales digitales en distintas situaciones, por ejemplo, recibir una señal que provenga de un botón o generar una señal para encender un LED. Si has trabajado antes con un Arduino clásico o un ESP8266, esto no será nuevo para ti.

Recuerda la estructura básica de un sketch en Arduino:

```
void setup()
{
    // AQUÍ VAN LOS PARÁMETROS Y CONFIGURACIONES INICIALES
}

void loop()
{
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN CONTINUAMENTE
}
```

Para usar los GPIO's como entradas o salidas, primero debemos configurarlos en `setup()` usando la siguiente instrucción:

```
pinMode(PIN,MODO);
```

donde en `PIN` hay que colocar el número de pin o GPIO a utilizar y en `MODO` se configura al pin como `INPUT` u `OUTPUT`, entrada o salida respectivamente.

Dentro de `loop()` colocaremos las instrucciones para el GPIO. Para leer una señal digital, se utiliza:

```
Variable=digitalRead(PIN);
```

donde `Variable` puede ser de tipo entero o booleana.

Para generar una señal digital, se utiliza:

```
digitalWrite(PIN,MODO);
```

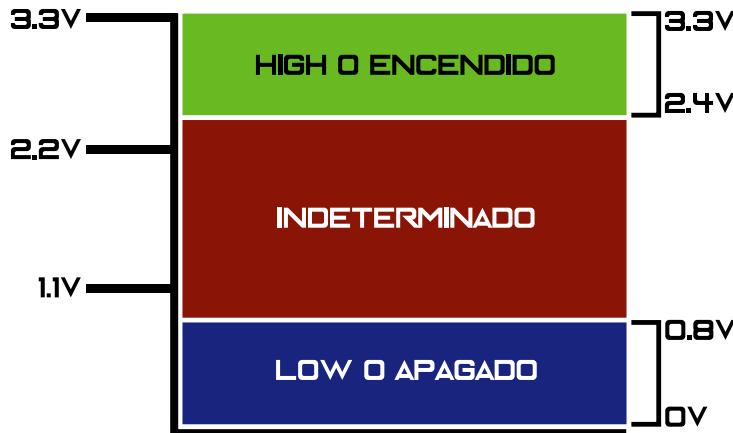
donde `MODO` puede ser `HIGH` o `LOW` (encendido o apagado) e incluso un entero, `1` o `0` respectivamente.

### Básicos: Las resistencias PULL-UP y PULL-DOWN

Como se vio en las especificaciones, el ESP32 tiene resistencias PULL-UP y PULL-DOWN internas de 45KΩ, que se pueden activar con un par de instrucciones.



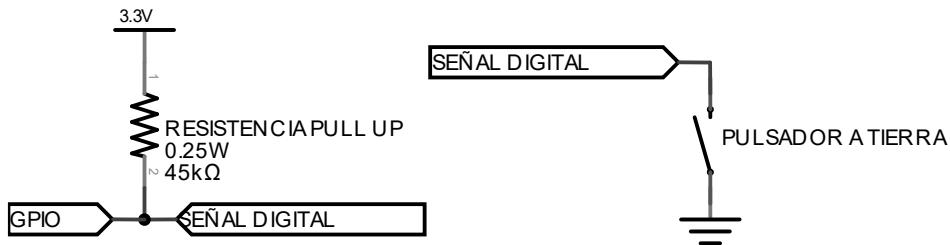
Las resistencias PULL-UP y PULL-DOWN se utilizan para establecer un valor de encendido o apagado (1 o 0) estable. El ESP32 determina el estado **HIGH** o **LOW** de una señal digital de acuerdo al siguiente rango:



si recibe una señal digital dentro de los rangos de voltaje **HIGH** y **LOW**, puede determinar cualquiera de estos estados sin problemas. Pero, cuando no está recibiendo una señal que se encuentre entre estos rangos, se dice que el GPIO o la entrada está “flotando”, por lo que el micro no puede determinar si existe un estado **HIGH** o **LOW**.

Las resistencias PULL-UP o PULL-DOWN estabilizan al GPIO en un valor inicial **HIGH** o **LOW**, para que pueda detectar los cambios de estado.

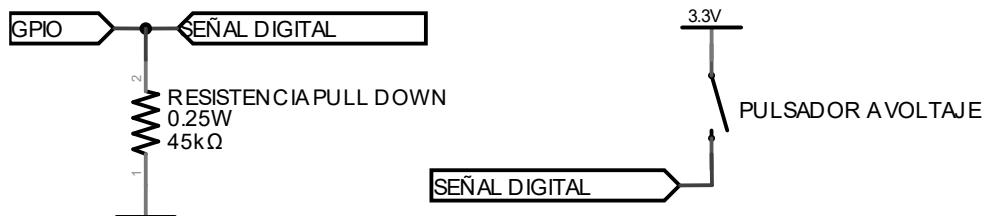
La resistencia PULL-UP estabiliza la entrada en un estado 1 o **HIGH**. Por ejemplo, se tiene el siguiente circuito:



la resistencia PULL-UP de  $45\text{k}\Omega$  conecta el GPIO a 3.3V (también limitando la corriente), por lo que se mantiene en el estado **HIGH**. Si ‘Señal Digital’ también es **HIGH**, no cambiará nada, pero si es una señal **LOW**, que en este caso es un botón pulsador que hace una conexión a tierra, entonces la resistencia PULL-UP y el GPIO quedarán conectados a tierra o 0V, haciendo el cambio a **LOW**. También es por esto que se utiliza una resistencia de un valor alto, ya que la fuente de 3.3V queda cortocircuitada a tierra y la resistencia se debe encargar de limitar la corriente.



La resistencia PULL-DOWN estabiliza la entrada en un estado 0 o *LOW*. Por ejemplo, se tiene el siguiente circuito:



la resistencia PULL-DOWN de  $45\text{k}\Omega$  conecta el GPIO a tierra o 0V, por lo que se mantiene en el estado *LOW*. De forma similar, si ‘Señal Digital’ también es *LOW*, no cambiará nada, pero si es una señal *HIGH*, que en este caso es un botón pulsador que hace una conexión a 3.3V, entonces la resistencia PULL-UP y el GPIO quedarán conectados a 3.3V, haciendo el cambio a *HIGH*. La resistencia limita la corriente que circula de la fuente de 3.3V (conectada al pulsador) hacia tierra, generando una diferencia de potencial de 3.3V entre ambos polos de la resistencia.

Las siguientes instrucciones activan las resistencias PULL-UP y PULL-DOWN internas del ESP32 cuando usemos los GPIO's como entradas digitales:

```
pinMode(PIN, INPUT_PULLUP);  
o  
pinMode(PIN, INPUT_PULLDOWN);
```

simplemente se le especifica a *INPUT* si se va a utilizar una resistencia PULL-UP o PULL-DOWN. Si vas a implementar un circuito externo PULL-UP o PULL-DOWN como los anteriores, entonces puedes usar el *INPUT* simple.

Ten en cuenta que con las señales de entrada analógicas no necesitas las resistencias PULL-UP o PULL-DOWN, sólo con señales de entrada digitales.



## Básicos: Los tipos de variables básicas en Arduino

Veamos las variables más comúnmente usadas:

- `char`: se usa para almacenar un carácter, ocupa un byte (8 bits) de espacio.
- `String`: se usa para almacenar cadenas de caracteres (como textos cortos).
- `bool`: solo almacena uno de dos estados lógicos: `true` o `false`.
- `byte`: puede almacenar un número entero entre 0 y 255 (1 byte).
- `int`: almacena enteros entre -32,768 y 32,767, ocupa 2 bytes (16 bits).
- `unsigned int`: similar a `int`, también almacena enteros, pero solamente positivos o sin signo. También ocupa 2 bytes, por lo que puede tomar valores entre 0 y 65,535.
- `Long`: almacena enteros de hasta 32 bits (4 bytes), es decir, desde -2,147,483,648 a 2,147,483,647.
- `unsigned Long`: similar a `unsigned int`, almacena enteros `Long` sin signo o desde 0 a 4,294,967,295.
- `Long64` y `unsigned Long64`: similares a `Long` y `unsigned Long`, almacenan enteros de hasta 64 bits (8 bytes)
- `float`: almacena enteros y decimales de hasta 32 bits (4 bytes), es decir, guarda valores entre -3.4028235^38 y 3.4028235^38.
- `double`: similar a `float`, almacena enteros y decimales de hasta 64 bits (8 bytes).

## Básicos: Estructura condicional simple ('if')

El condicional `if` se utiliza para ejecutar una instrucción o un grupo de instrucciones sólo si se cumple una condición dada. Tiene la siguiente estructura:

```
if(CONDICIÓN)
{
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN
}
```

donde `CONDICIÓN` puede ser de tipo matemática/lógica o booleana. De manera opcional, si se agrega una estructura `else`, proveemos una ruta alternativa al `if` en caso de no cumplir su condición dada:

```
if(CONDICIÓN)
{
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN
}
```



```
else
{
    // AQUÍ VAN LAS INSTRUCCIONES ALTERNATIVAS QUE SE EJECUTARÁN
}
```

## Básicos: Estructura cíclica 'for'

El ciclo **for** es la primera estructura cíclica básica. Se utiliza para ejecutar una instrucción o grupo de instrucciones un número determinado de veces, hasta cumplir con una condición establecida. Tiene la siguiente estructura:

```
for(VALOR_INICIAL;CONDICIÓN;ACUMULADOR)
{
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN
}
```

donde **VALOR\_INICIAL** es una variable con el valor del que partirá el conteo de ciclos, **CONDICIÓN** es una condición lógica que deberá cumplirse para detener el ciclo, por ejemplo, que continúe el ciclo hasta alcanzar un valor mayor a (**>**), menor a (**<**), igual a (**=**), o alguna combinación de estas condiciones. El **ACUMULADOR** es una operación matemática que aumenta o disminuye el valor de la variable **VALOR\_INICIAL** cada vez que acaba un ciclo, para poder cumplir con la **CONDICIÓN** establecida en algún momento.

## Básicos: Mostrar datos en el Monitor Serie

Para enviar datos por puerto serial y poder verlos en el Monitor Serie, primero hay que configurar la comunicación serial en **setup()**, usando la siguiente instrucción:

```
Serial.begin(VELOCIDAD);
```

donde **VELOCIDAD** indica la velocidad de comunicación en baudios. Puedes usar alguna de las velocidades seleccionables del Monitor Serie, pero las más recomendadas son **9600** o **115200**. Las instrucciones para enviar datos por puerto serial son:

```
Serial.print(DATO);
o
Serial.println(DATO);
```

donde **DATO** puede ser una variable o un texto (este último colocado entre comillas como "Mensaje"). La diferencia entre **Serial.println** y **Serial.print**, es que **Serial.println** hará un salto de línea o renglón cada vez que envíe el contenido de **DATO**.

Ahora hagamos algunos ejercicios.



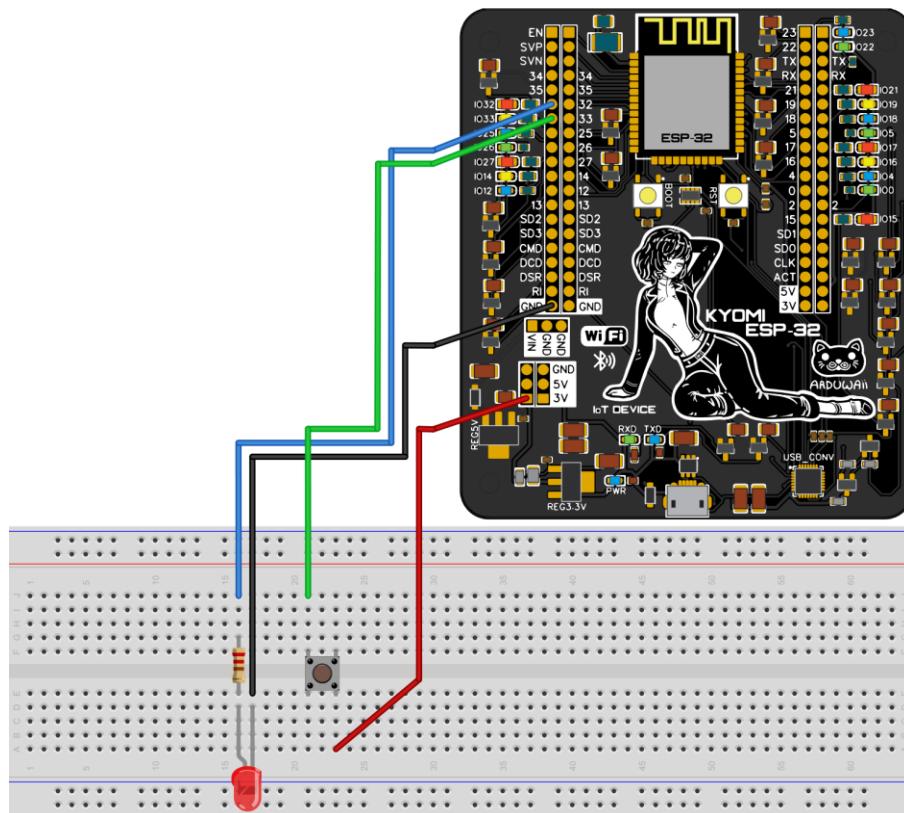
# Ejercicio 1

Realizaremos un ejercicio sencillo donde encenderemos y apagaremos un LED con un botón, a través de la lectura y escritura digital y una estructura condicional.

## Materiales:

- 1 botón pulsador
  - 1 LED (opcional)
  - 1 resistencia de  $220\Omega$  (opcional)
  - 4 jumpers o cables tipo Dupont (2 para el LED)
  - plantilla de experimentos

**El circuito es el siguiente:**



Si tienes la tarjeta KYOMI, la resistencia y el LED los puedes omitir, ya que cada GPIO tiene un LED integrado.

Empecemos con el código, primero hay que declarar unas variables para los GPIO del botón y del LED:

```
int BOTON=33, LED=32, ESTADO_BOTON=0;
```

para el ESP32, sólo hay que poner el número de GPIO en las instrucciones o las variables.



**CONSEJO:** es recomendable utilizar variables para referenciar a los GPIO's, en lugar de colocar el número de GPIO en cada instrucción donde lo vayas a utilizar, esto te ahorrará tiempo cuando necesites hacer cambios.

En `setup()`, configuraremos los pines y la comunicación serial para el Monitor Serie, usaremos el modo PULL-DOWN en el GPIO para el botón:

```
void setup()
{
    pinMode(BOTON, INPUT_PULLDOWN);
    pinMode(LED, OUTPUT);
    Serial.begin(115200);
}
```

Ahora en `loop()`, vamos a colocar las siguientes instrucciones:

```
void loop()
{
    ESTADO_BOTON=digitalRead(BOTON);
    if(ESTADO_BOTON==1)
    {
        digitalWrite(LED,HIGH);
        Serial.println("El led está encendido");
    }
    else
    {
        digitalWrite(LED,LOW);
        Serial.println("El led está apagado");
    }
}
```

primero se determina el estado del GPIO del botón. Si es igual a `HIGH` o 1, entonces el GPIO del LED también cambia a `HIGH`, además de mostrar un mensaje en el Monitor Serie. Si es igual a `LOW` o 0, el GPIO del LED también se mantiene en `LOW`, y muestra otro mensaje en el Monitor Serie.

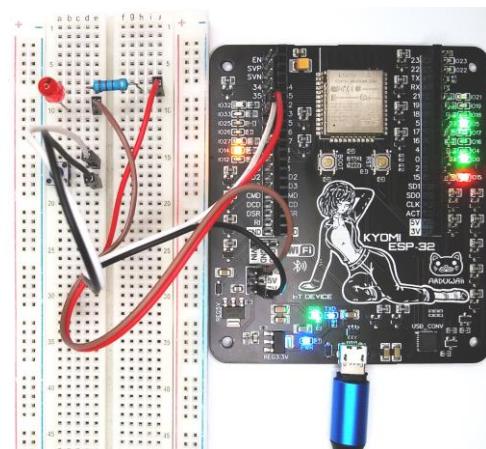
Ahora sube el programa a tu tarjeta y pruébalo. En la tarjeta KYOMI también se encenderá el LED integrado del GPIO del botón cada vez que lo pulses.

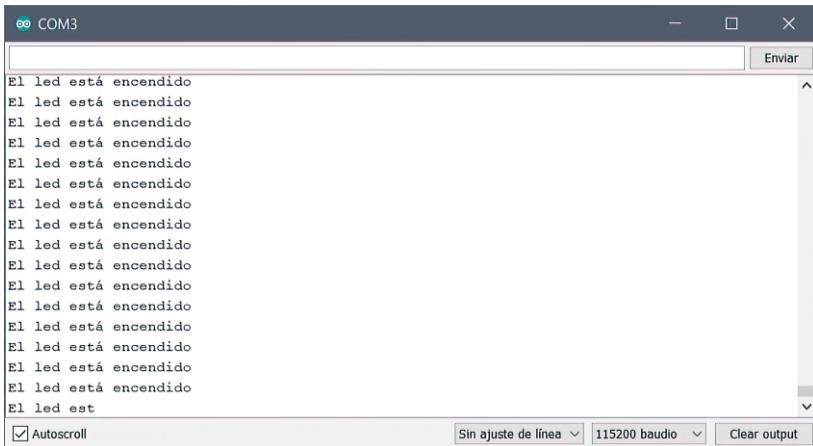


El código completo queda de la siguiente forma:

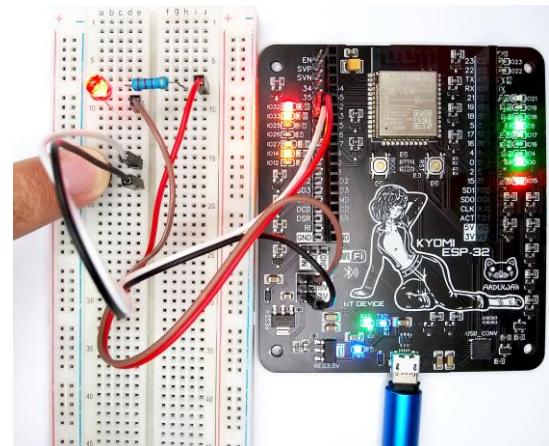
```
int BOTON=33,LED=32,ESTADO_BOTON=0;  
  
void setup()  
{  
    pinMode(BOTON, INPUT_PULLDOWN);  
    pinMode(LED,OUTPUT);  
    Serial.begin(115200);  
}  
  
void loop()  
{  
    ESTADO_BOTON=digitalRead(BOTON);  
    if(ESTADO_BOTON==1)  
    {  
        digitalWrite(LED,HIGH);  
        Serial.println("El led está encendido");  
    }  
    else  
    {  
        digitalWrite(LED,LOW);  
        Serial.println("El led está apagado");  
    }  
}
```

Una vez subido al programa a la tarjeta, podrás ver el mensaje de encendido o apagado en el Monitor Serie (no olvides configurarlo con la velocidad usada en el código):





The screenshot shows the Arduino IDE's Serial Monitor window titled "COM3". It displays the text "El led está encendido" (The LED is on) repeated multiple times. At the bottom of the window, there are buttons for "Enviar" (Send), "Autoscroll", "Sin ajuste de línea" (No line break), "115200 baudio", and "Clear output".



En el caso de la tarjeta KYOMI, habrás notado que los GPIO 0, 5, 14 y 15 se mantienen encendidos, aunque no los uses en el sketch. Esto es normal, ya que son pines que generan señales durante el arranque y mantienen su último estado cuando el programa empieza a correr, pero podrás configurarlos para que regresen al estado LOW cuando arranque tu programa, y también usarlos para lo que necesites. Veamos el siguiente ejercicio, donde configuraremos todos los pines.

## Ejercicio 2

Ahora realizaremos un ejercicio donde utilizaremos una estructura cíclica y una estructura condicional para activar un par de secuencias de LEDs. Antes de pasar al ejercicio, veamos la estructura condicional **switch** y cómo usar variables de tipo arreglo o array.

### Básicos: Estructura condicional compuesta ('switch')

El condicional **switch** se utiliza para ejecutar dos o más instrucciones o grupos de instrucciones que son seleccionados de acuerdo a una condición dada. Es el equivalente a utilizar varios condicionales **if**, cuyas condiciones dependen de la misma variable. Tiene la siguiente estructura:

```
switch(CONDICIÓN)
{
    case CASO 1:
        // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN EN EL CASO 1
        break;

    case CASO 2:
        // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN EN EL CASO 2
        break;
}
```



`default:`

```
// AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN EN EL CASO DEFAULT
```

```
break;
```

```
}
```

donde `CONDICIÓN` es una variable que puede tomar un valor correspondiente a `CASO 1`, `CASO 2`, `CASO 3`, etc., para ejecutar las instrucciones de dicho caso. La instrucción `break` simplemente cierra cada caso. El caso `default` es opcional, y en él podemos colocar instrucciones que se ejecuten cuando el valor de `CONDICIÓN` nunca habilite alguno de los casos anteriores.

## Básicos: Cómo usar variables de tipo arreglo ('arrays')

Una variable array es aquella que puede guardar varios valores independientes, los cuales son accesibles mediante un índice. Puede ser una variable `int`, `float`, `double`, `char`, etc. Veamos cómo declarar un arreglo de tipo entero:

```
int VARIABLE[CANTIDAD DE VALORES];
```

donde `CANTIDAD DE VALORES` es el número de valores que guardará el array. No siempre es necesario especificar una cantidad, ya que el array puede tomar el tamaño correspondiente según la cantidad de valores que vaya guardando, entonces también puede declararse de la siguiente manera:

```
int VARIABLE[];
```

Para inicializar el array con varios valores, hay que declarar la cantidad de valores guardados y luego agregarlos, por ejemplo:

```
int NUMEROS[10]={6,2,3,7,9,8,1,4,10,5};
```

donde los valores se colocan entre corchetes y van separados por comas.

**Importante:** los elementos que guardemos en el array van ordenados por un índice, empezando desde el 0. Entonces, para el ejemplo anterior, tenemos lo siguiente:





Para acceder a un valor del array, por ejemplo, al número 9 de `NUMEROS[10]`, podemos usar una variable que esté igualada a dicho array con la posición donde se encuentra el 9:

*N=NUMEROS[4];*

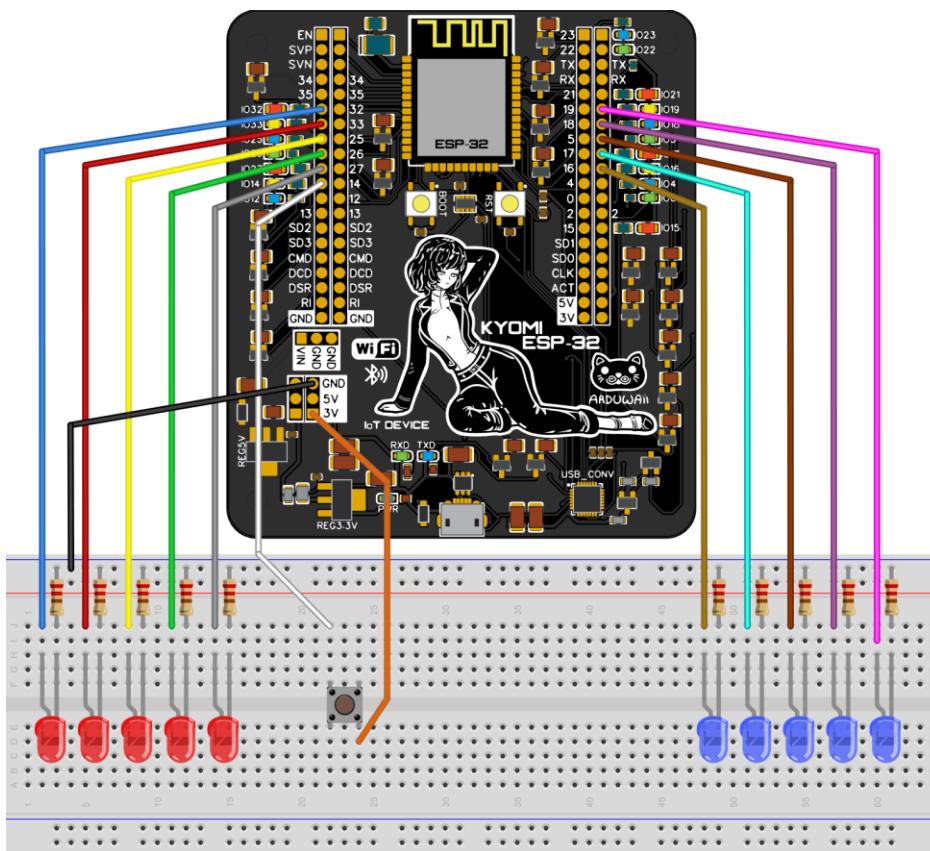
entonces *N* será igual al valor de la posición 4 del array (en este caso 9).

Ahora vayamos al ejercicio.

## Materiales:

- 1 botón pulsador
  - 10 LEDs (opcional)
  - 10 resistencias de  $220\Omega$  (opcional)
  - 13 jumpers o cables tipo Dupont (11 para los LEDs)
  - plantilla de experimentos

**El circuito es el siguiente:**



Empecemos con el código. Ahora declararemos todos los GPIO dentro de un par de arrays, uno guardará los GPIO de lado izquierdo de la tarjeta y el otro los del lado derecho. También agregaremos unas variables auxiliares para un *for* y para un *switch*.



```
const int GPIOIZQ[6]={32, 33, 25, 26, 27, 12}, GPIO_DER[11]={23, 22, 21, 19, 18, 5, 17, 16, 4, 0, 15}, BOTON=14;  
int contador_FOR=0, SECUENCIA=0;
```

al usar el parámetro `const` cuando colocamos el tipo de variable, convertiremos a la variable declarada en constante, por lo que no modificará su valor.

**CONSEJO:** usar variables tipo `const` para referenciar a los GPIO's, mantiene sus valores fijos y evita que sean modificados en alguna otra instrucción del programa.

En `setup()` configuramos los pines y la comunicación serial. En lugar de usar `pinMode` para cada GPIO, podemos hacerlo en un solo paso: en un ciclo `for` colocamos como **VALOR INICIAL** un 0, que sería el primer lugar del índice del array de GPIO's, como **CONDICIÓN**, indicaremos que haga el ciclo siempre y cuando **VALOR INICIAL** sea menor o igual a la última posición del índice del array de GPIO's, finalmente, en **ACUMULADOR** indicamos que se sume un 1 a la variable de **VALOR INICIAL** cada vez que termine un ciclo. La variable para **VALOR INICIAL** y **ACUMULADOR** también la usamos para el índice de posición del array, al configurar los GPIO como **OUTPUT** en `pinMode`. También inicializamos los GPIO's para que arranquen con el estado **LOW**:

```
void setup()  
{  
    for(contador_FOR=0; contador_FOR<=5; contador_FOR++)  
    {  
        pinMode(GPIOIZQ[contador_FOR], OUTPUT);  
        digitalWrite(GPIOIZQ[contador_FOR], LOW);  
    }  
    for(contador_FOR=0; contador_FOR<=10; contador_FOR++)  
    {  
        pinMode(GPIO_DER[contador_FOR], OUTPUT);  
        digitalWrite(GPIOIZQ[contador_FOR], LOW);  
    }  
    pinMode(BOTON, INPUT_PULLDOWN);  
    Serial.begin(115200);  
}
```

Un ciclo `for` se encarga de configurar los GPIO del lado izquierdo y otro los del lado derecho. Solo el GPIO 14 (el del botón) se configura como `INPUT_PULLDOWN`.



Ahora en `Loop()`, vamos a colocar las siguientes instrucciones:

```
void Loop()
{
    SECUENCIA=SECUENCIA+digitalRead(BOTON);
    if(SECUENCIA>1){SECUENCIA=0;}

    switch(SECUENCIA)
    {
        case 0:
            Serial.println("Secuencia Izquierda");
            for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)
            {
                digitalWrite(GPIOIZQ[contador_FOR],HIGH);
            }
            for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)
            {
                digitalWrite(GPIOIZQ[contador_FOR],LOW);
            }
            delay(80);
            for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)
            {
                digitalWrite(GPIOIZQ[contador_FOR],HIGH);
            }
            for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)
            {
                digitalWrite(GPIOIZQ[contador_FOR],LOW);
            }
            delay(80);
            break;

        case 1:
            Serial.println("Secuencia Derecha");
            for(contador_FOR=3; contador_FOR<=7; contador_FOR++)
            {
                digitalWrite(GPIODER[contador_FOR],HIGH);
                delay(20);
            }
    }
}
```



```
        }
        for(contador_FOR=3; contador_FOR<=7; contador_FOR++)
        {
            digitalWrite(GPIO_DER[contador_FOR],LOW);
            deLay(20);
        }
        break;
    }

    for(contador_FOR=0; contador_FOR<=5; contador_FOR++)
    {
        digitalWrite(GPIO_IQZ[contador_FOR],LOW);
    }
    for(contador_FOR=0; contador_FOR<=10; contador_FOR++)
    {
        digitalWrite(GPIO_IQZ[contador_FOR],LOW);
    }
}
```

En *SECUENCIA* se coloca un acumulador, que tiene un valor inicial 0, y cuando *digitalRead(BOTON)* es *HIGH* entonces se acumularía un 0+1. El siguiente *if* regresa a 0 al acumulador de *SECUENCIA* si el valor acumulado es mayor a 1, que ocurriría cuando el botón se presiona dos veces.

De acuerdo al valor de *SECUENCIA*, se ejecutará uno de los dos casos del *switch*. En *case 0*, la secuencia solo utiliza los GPIO 32, 33, 25, 26 y 27, que corresponden a los índices 0,1,2,3 y 4 del array *GPIO\_IQZ[]*. El *case 0* funciona de la siguiente manera:

1. Se envía un mensaje al Monitor Serie.
2. En el primer ciclo *for* se encienden los GPIO 32, 25 y 27, por lo que en el *ACUMULADOR* se suma un 2 para omitir los otros GPIO's.
3. En el segundo ciclo *for* se apagan los GPIO 33 y 26, entonces en el *ACUMULADOR* se suma un 2 para omitir los otros GPIO's, pero *VALOR INICIAL* ahora comienza en 1.
4. Se hace una pausa de 80 milisegundos con *deLay(80)*, para poder observar el encendido de los LEDs.
5. En el tercer ciclo *for* se encienden los GPIO 33 y 26, e igualmente en el *ACUMULADOR* se suma un 2 para omitir los otros GPIO's y *VALOR INICIAL* comienza en 1.
6. En el cuarto ciclo *for* se apagan los GPIO 32, 25 y 27, así que en *ACUMULADOR* se suma un 2 para omitir los otros GPIO's.



## 7. Nuevamente se hace una pausa de 80 milisegundos con `delay(80)`.

En `case 1`, la secuencia solo utiliza los GPIO 19, 18, 5, 17 y 16, que corresponden a los índices 3,4,5,6 y 7 del array `GPIO_DER[]`. El `case 1` es un poco más sencillo:

1. Se envía un mensaje al Monitor Serie.
2. En el primer ciclo `for` se van encendiendo todos los GPIO's, cada uno haciendo una pausa de 20 milisegundos con `delay(20)`.
3. En el segundo ciclo `for` ahora se van apagando todos los GPIO's, igualmente con pausas de 20 milisegundos.

Finalmente, al terminar el `switch`, se usa otro par de `for` para colocar todos los GPIO's en `LOW`, ya que al cambiar de secuencia (cuando se presiona el botón), quedarían encendidos los LEDs de la secuencia anterior.

El código completo queda de la siguiente forma:

```
const int GPIO_IQZ[6]={32, 33, 25, 26, 27, 12}, GPIO_DER[11]={23, 22, 21, 19, 18,  
5, 17, 16, 4, 0, 15}, BOTON=14;  
  
int contador_FOR=0, SECUENCIA=0;  
  
void setup()  
{  
    for(contador_FOR=0; contador_FOR<=5; contador_FOR++)  
    {  
        pinMode(GPIO_IQZ[contador_FOR],OUTPUT);  
        digitalWrite(GPIO_IQZ[contador_FOR],LOW);  
    }  
    for(contador_FOR=0; contador_FOR<=10; contador_FOR++)  
    {  
        pinMode(GPIO_DER[contador_FOR],OUTPUT);  
        digitalWrite(GPIO_IQZ[contador_FOR],LOW);  
    }  
    pinMode(BOTON, INPUT_PULLDOWN);  
    Serial.begin(115200);  
}  
  
void Loop()  
{  
    SECUENCIA=SECUENCIA+digitalRead(BOTON);  
    if(SECUENCIA>1){SECUENCIA=0;}  
  
    switch(SECUENCIA)
```



```
{  
    case 0:  
        Serial.println("Secuencia Izquierda");  
        for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],HIGH);  
        }  
        for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],LOW);  
        }  
        delay(80);  
        for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],HIGH);  
        }  
        for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],LOW);  
        }  
        delay(80);  
    break;  
  
    case 1:  
        Serial.println("Secuencia Derecha");  
        for(contador_FOR=3; contador_FOR<=7; contador_FOR++)  
        {  
            digitalWrite(GPIODER[contador_FOR],HIGH);  
            delay(20);  
        }  
        for(contador_FOR=3; contador_FOR<=7; contador_FOR++)  
        {  
            digitalWrite(GPIODER[contador_FOR],LOW);  
            delay(20);  
        }  
    break;
```



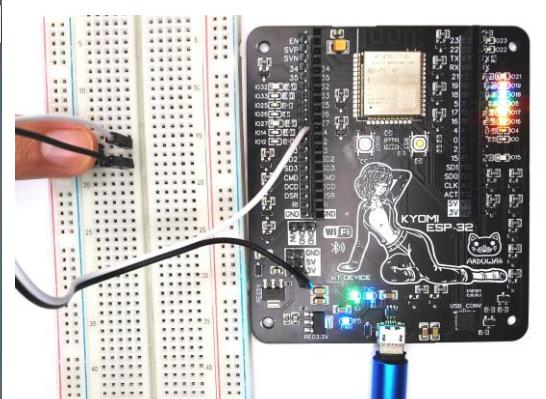
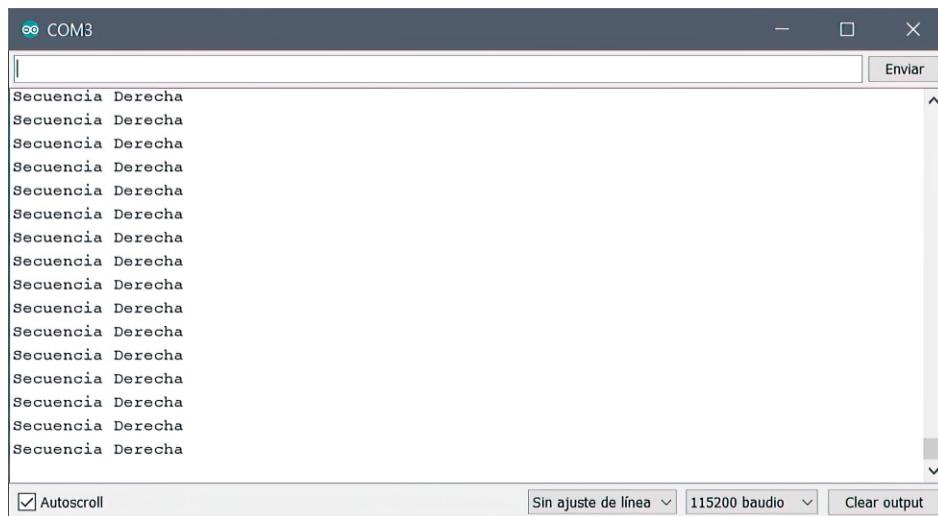
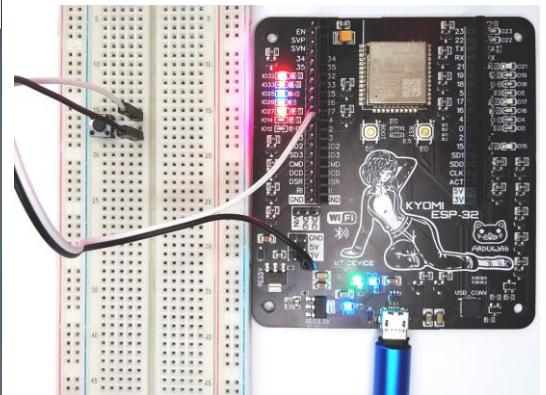
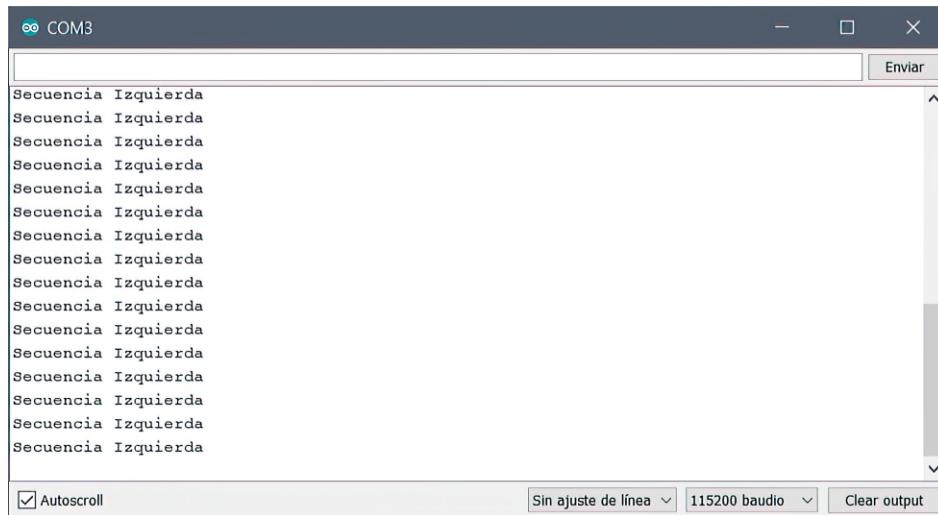
```
}

for(contador_FOR=0; contador_FOR<=5; contador_FOR++)
{
    digitalWrite(GPIO0_IZQ[contador_FOR], LOW);
}

for(contador_FOR=0; contador_FOR<=10; contador_FOR++)
{
    digitalWrite(GPIO0_IZQ[contador_FOR], LOW);
}

}
```

Ahora prueba el programa. Podrás ver el mensaje de activación cada secuencia en el Monitor Serie:





## Los GPIO's táctiles (touch)

Ahora que conoces las funciones digitales de los GPIO's, veamos una de las funciones especiales que integra el ESP32 en algunos de estos, el modo táctil o touch.

El ESP32 tiene 10 GPIO's con funcionalidad touch (puedes verlos en el pinout). Cuentan con touch de tipo capacitivo, el cual detecta pequeñas cargas eléctricas que acumulan ciertos objetos, como la piel (particularmente en los dedos o la mano).

Los GPIO's touch se pueden conectar a superficies capacitivas (objetos metálicos o ciertos plásticos) para reemplazar a los pulsadores mecánicos, requiriendo solo cable de conexión (a diferencia de los botones, que requieren dos).

La señal detectada por los GPIO en modo touch es analógica y hay que establecer valores de umbral que correspondan a un estado que consideremos como *HIGH* o *LOW*, similar a los rangos de voltaje que utiliza el ESP32 para detectar un *HIGH* o *LOW* digital.

Para obtener la lectura de un GPIO touch, usaremos la siguiente instrucción:

`touchRead(GPIO TOUCH)`

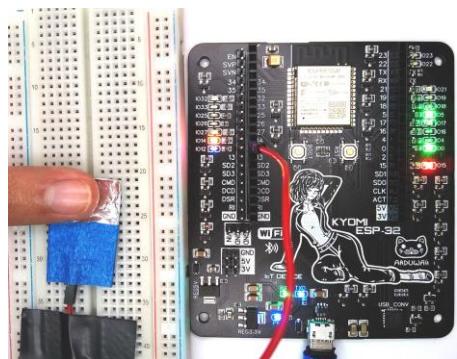
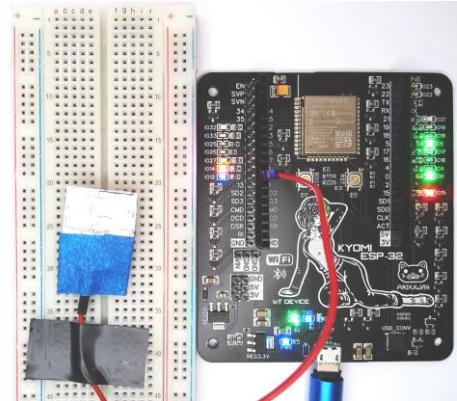
podemos usar `touchRead` en una variable o directamente en un `Serial.print`, en `GPIO TOUCH` se coloca el número de pin touch, no el número de GPIO (puedes revisarlos en el pinout). Por ejemplo, para usar el **Touch 5** (que corresponde al GPIO12), debes colocarlo como `touchRead(T5)`.

Usa el siguiente código para hacer una primera prueba, puedes pegar un trozo de papel aluminio a un jumper o cable Dupont para usarlo como pad touch:

```
const int GPIO_12=12;
void setup()
{
    pinMode(GPIO_12, INPUT);
    Serial.begin(115200);
}
void Loop()
{
    Serial.println(touchRead(T5));
}
```



En el Monitor Serie, verás que se muestra un valor casi constante si no tocas el trozo de papel aluminio, pero al tocarlo con el dedo, verás que este valor se reduce:



Para calibrar el valor umbral, usaremos la herramienta ‘Graficador Serial’.

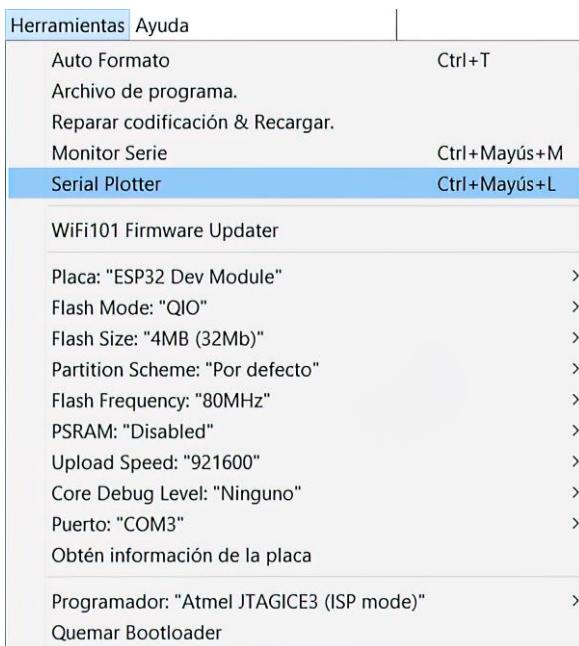
## Básicos: Graficador Serial (Serial Plotter)

Esta herramienta se utiliza para mostrar los cambios del valor de una variable a lo largo del tiempo, mostrándolos como formas de onda.

El Graficador Serial también utiliza las instrucciones `Serial.print`, siempre y cuando los datos enviados sean numéricos y no de texto.



El Graficador Serial se encuentra debajo del Monitor Serie:



Retomando la prueba del GPIO touch, en el Graficador Serial podrás ver la gráfica de valores de `touchRead()` y cómo varía al tocar el trozo de papel aluminio (la escala vertical se ajustará automáticamente):



Recuerda configurar la velocidad del Graficador Serial con la que usas en `Serial.begin()`.

También puedes tener varias graficas correspondientes a distintas variables, separando cada una en un `Serial.print`.



Por ejemplo, para graficar tres variables se utilizaría la siguiente estructura:

```
Serial.print(VARIABLE_1);
Serial.print("\t");
Serial.print(VARIABLE_2);
Serial.print("\t");
Serial.println(VARIABLE_3);
```

donde `\t` es una tabulación que separa a las variables, ya que todas se encuentran en un mismo renglón, y el último `Serial.println` se encarga de hacer un salto de línea. Por ejemplo, para dos variables que correspondan a las lecturas de dos GPIO's touch, el resultado se vería así:



Automáticamente se asignará un color de línea a cada variable y los colores irán ordenados según fueron colocados los `Serial.print`, por lo que `VARIABLE_1` tendría el color azul y `VARIABLE_2` el rojo.

**NOTA:** no se puede abrir el Monitor Serie y el Graficador Serial al mismo tiempo, ya que abrirían el mismo puerto serie y no es posible tenerlo abierto dos veces, así que solo puedes estar usando una herramienta a la vez.

## Calibración de un GPIO touch

Como habrás visto con el código de prueba, los valores máximos de las lecturas rondaban de 10 a 20 unidades, y al tocar el pad de aluminio, las lecturas caían por debajo de las 5 unidades. Estos valores varían de



tarjeta a tarjeta, ya que en algunas se pueden registrar valores máximos de 50 u 80 unidades y en otras se obtienen valores más bajos, como es el caso de la tarjeta KYOMI. Es muy recomendable aumentar las escalas obtenidas para ampliar el rango de valores y así tener más espacio para la calibración.

La calibración para los estados que consideraremos como *HIGH* y *LOW* la haremos en tres etapas:

1. Aumentar escala (Opcional)
2. Estabilizar valores de lectura
3. Comparación de valores para el estado *HIGH* y *LOW*

En la primera etapa simplemente hay que multiplicar el valor de *touchRead()* por una constante. Para la tarjeta KYOMI se recomienda hacerlo por 100.

En la segunda etapa se realiza un tratamiento matemático para eliminar variaciones rápidas, y así obtener valores más estables. Este tratamiento lo podemos hacer con un promedio simple, acumulando unas cuantas lecturas y dividiendo el resultado entre la cantidad de lecturas acumuladas. El promedio se puede implementar con un *for*, 50 valores serán suficientes:

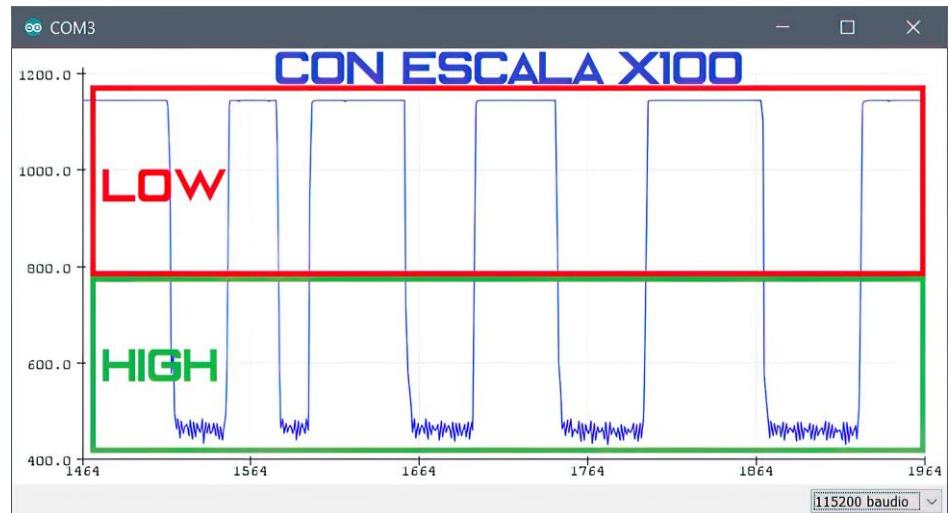
```
for(contador_FOR=0; contador_FOR<=50; contador_FOR++)  
{  
    Valor_touch=Valor_touch+touchRead(Touch)  
}  
Valor_touch=Valor_touch/50;
```

El resultado en el Graficador Serial se verá más limpio y estable:





En la tercera etapa, construiremos el comparador de valores para el estado **HIGH** y **LOW**, usando la gráfica de las lecturas como referencia:



Mediante condicionales **if**, podremos establecer los rangos para **HIGH** y **LOW**, por ejemplo:

```
//PARA LA ESCALA NORMAL
if(Valor_touch>=8)
{
    ESTADO=0;
}
else
{
    ESTADO=1;
}
```



```
//PARA LA ESCALA x100  
if(Valor_touch>=750)  
{  
ESTADO=0;  
}  
else  
{  
ESTADO=1;  
}
```

Ahora veamos un par de ejercicios con GPIO touch.

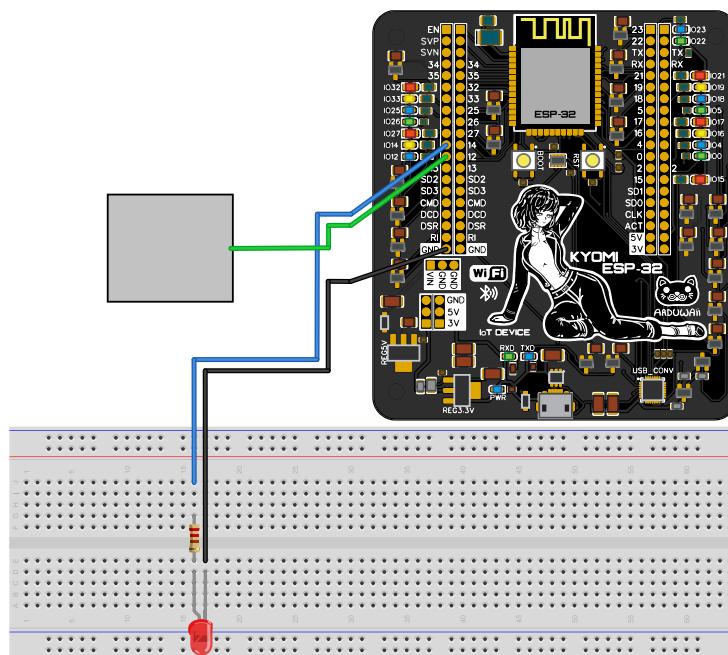
## Ejercicio 3

Realizaremos un ejercicio donde encenderemos y apagaremos un LED con un pad touch de papel aluminio, aplicando el método de calibración anterior.

### Materiales:

- 1 trozo de papel aluminio
- 1 LED (opcional)
- 1 resistencia de 220Ω (opcional)
- 3 jumpers o cables tipo Dupont (2 para el LED)
- plantilla de experimentos

El circuito es el siguiente:





Veamos el código. Primero declaramos las variables para los GPIO's del pad touch y del LED. También usaremos otras variables para el valor de las lecturas touch, el valor del estado 'digital' de dichas lecturas y el contador auxiliar para el *for*:

```
const int TOUCH=12,LED=14;  
int Valor_touch=0,ESTADO=0,contador_FOR=0;  
  
void setup()  
{  
    pinMode(TOUCH, INPUT);  
    pinMode(LED,OUTPUT);  
    Serial.begin(115200);  
}
```

En *setup()*, el GPIO que se usará como pin touch puede ser configurado como *INPUT* en *pinMode*, aunque no es completamente necesario hacer esto.

Ahora en *Loop()*, primero haremos unas cuantas lecturas con *touchRead()* (escalándolas x100) y después obtendremos su promedio:

```
for(contador_FOR=0; contador_FOR<=50; contador_FOR++)  
{  
    Valor_touch=Valor_touch+(100*touchRead(T5));  
}  
Valor_touch=Valor_touch/50;
```

Con un *if* construimos el comparador para establecer los rangos de *HIGH* y *LOW* que tomará la variable *ESTADO*. Usaremos un rango similar al que vimos anteriormente, pero es mejor que hagas una prueba para verificar que los valores de umbral sean aceptables.

```
if(Valor_touch>=750)  
{  
    ESTADO=0;  
}  
else  
{  
    ESTADO=1;  
}
```



Finalmente, usamos otro **if** para encender o apagar el LED de acuerdo al valor de *ESTADO*:

```
if(ESTADO==1)
{
    digitalWrite(LED,HIGH);
    Serial.println("El led está encendido");
}
else
{
    digitalWrite(LED,LOW);
    Serial.println("El led está apagado");
}
```

El código completo queda de la siguiente forma:

```
const int TOUCH=12,LED=14;
int Valor_touch=0,ESTADO=0,contador_FOR=0;

void setup()
{
    pinMode(TOUCH, INPUT);
    pinMode(LED,OUTPUT);
    Serial.begin(115200);
}

void Loop()
{

for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
{
    Valor_touch=Valor_touch+(100*touchRead(T5));
}
Valor_touch=Valor_touch/50;

if(Valor_touch>=750)
{
    ESTADO=0;
}
```



```
else
{
    ESTADO=1;
}
if(ESTADO==1)
{
    digitalWrite(LED,HIGH);
    Serial.println("El led está encendido");
}
else
{
    digitalWrite(LED,LOW);
    Serial.println("El led está apagado");
}
```

Ahora prueba el programa, verás que el LED enciende cada vez que toques el pad de aluminio.



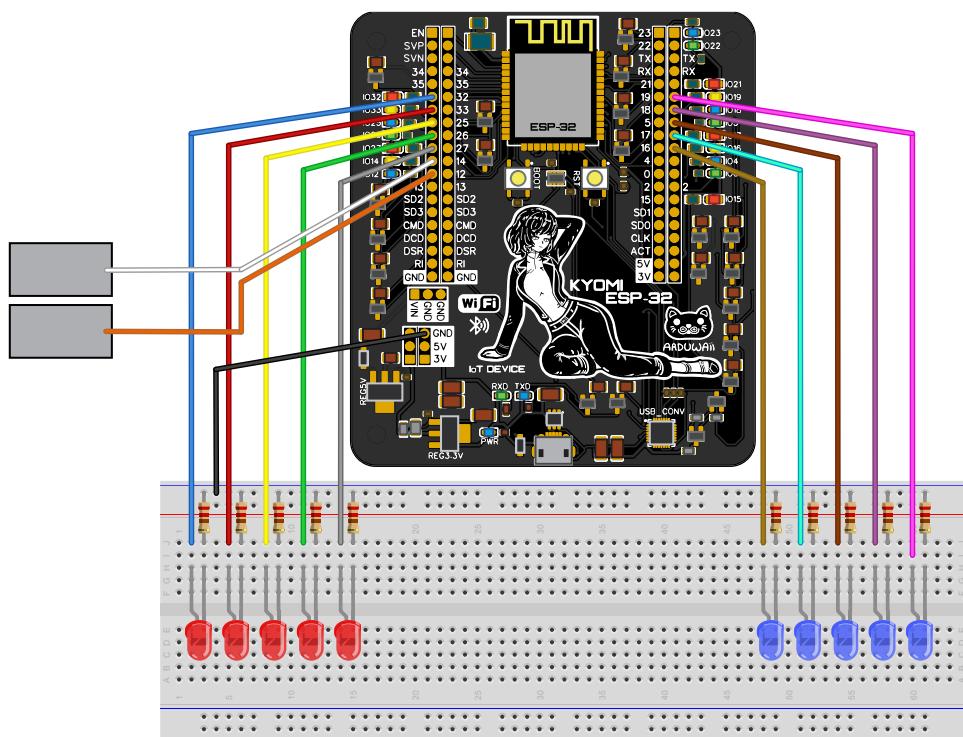
## Ejercicio 4

Retomaremos el ejercicio 2 para reemplazar el botón pulsador por un par de pads touch para activar las secuencias de LEDs.

Materiales:

- 2 trozos de papel aluminio
- 10 LEDs (opcional)
- 10 resistencias de 220Ω (opcional)
- 13 jumpers o cables tipo Dupont (11 para los LEDs)
- plantilla de experimentos

El circuito es el siguiente:



El código se mantiene igual, excepto la parte de la selección de secuencia con el pulsador. En la declaración de variables, se van a agregar un par de estas para los valores de las lecturas touch, otro par para los valores de estado touch y otro par para declarar los GPIO touch:

```
const int GPIO_I2Q[6]={32, 33, 25, 26, 27, 12}, GPIO_DER[11]={23, 22, 21, 19, 18, 5, 17, 16, 4, 0, 15}, TOUCH_1=14, TOUCH_2=12;  
int contador_FOR=0, SECUENCIA=0, Valor_touch_1=0, ESTADO_1=0, Valor_touch_2=0, ESTADO_2=0;
```



En `setup()` solo hay que agregar la configuración de los GPIO touch como `INPUT` (esto lo puedes omitir):

```
for(contador_FOR=0; contador_FOR<=5; contador_FOR++)
{
    pinMode(GPIO_I2Q[contador_FOR],OUTPUT);
    digitalWrite(GPIO_I2Q[contador_FOR],LOW);
}
for(contador_FOR=0; contador_FOR<=10; contador_FOR++)
{
    pinMode(GPIO_DER[contador_FOR],OUTPUT);
    digitalWrite(GPIO_I2Q[contador_FOR],LOW);
}
pinMode(TOUCH_1,INPUT);
pinMode(TOUCH_2,INPUT);
Serial.begin(115200);
```

Al principio de `Loop()`, se hace el promedio de las lecturas touch para ambos pads y se determina si están en estado `HIGH` o `LOW`, en este caso se están usando los pines touch `T6` y `T5`, que corresponden al GPIO14 y 12 respectivamente:

```
for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
{
    Valor_touch_1=Valor_touch_1+(100*touchRead(T6));
}
Valor_touch_1=Valor_touch_1/50;
if(Valor_touch_1>=750)
{
    ESTADO_1=0;
}
else
{
    ESTADO_1=1;
}
for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
{
    Valor_touch_2=Valor_touch_2+(100*touchRead(T5));
```



```
Valor_touch_2=Valor_touch_2/50;  
if(Valor_touch_2>=750)  
{  
    ESTADO_2=0;  
}  
else  
{  
    ESTADO_2=1;  
}
```

En la siguiente parte se asigna un valor a *SECUENCIA* de acuerdo a los valores de *ESTADO\_1* y *ESTADO\_2*:

```
if(ESTADO_1==1 && ESTADO_2==0){SECUENCIA=0;}  
if(ESTADO_1==0 && ESTADO_2==1){SECUENCIA=1;}  
SECUENCIA=SECUENCIA;
```

Si *ESTADO\_1=1* y *ESTADO\_2=0*, entonces *SECUENCIA=0*, si *ESTADO\_1=0* y *ESTADO\_2=1*, entonces *SECUENCIA=1*. Al colocar *SECUENCIA=SECUENCIA*, estamos construyendo un acumulador que conserva su valor asignado en los *if* anteriores, esto para los momentos en los que no se presiona ningún pad touch, guardando el último valor asignado.

El resto de *Loop()* se mantiene sin cambios:

```
switch(SECUENCIA)  
{  
    case 0:  
        Serial.println("Secuencia Izquierda");  
        for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],HIGH);  
        }  
        for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],LOW);  
        }  
        delay(80);  
        for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],HIGH);  
        }
```



```
for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)
{
    digitalWrite(GPIOIZQ[contador_FOR],LOW);
}
delay(80);
break;

case 1:
    Serial.println("Secuencia Derecha");
    for(contador_FOR=3; contador_FOR<=7; contador_FOR++)
    {
        digitalWrite(GPIO_DER[contador_FOR],HIGH);
        delay(20);
    }
    for(contador_FOR=3; contador_FOR<=7; contador_FOR++)
    {
        digitalWrite(GPIO_DER[contador_FOR],LOW);
        delay(20);
    }
    break;
}

for(contador_FOR=0; contador_FOR<=5; contador_FOR++)
{
    digitalWrite(GPIOIZQ[contador_FOR],LOW);
}
for(contador_FOR=0; contador_FOR<=10; contador_FOR++)
{
    digitalWrite(GPIOIZQ[contador_FOR],LOW);
}
```

El código completo queda de la siguiente forma:

```
const int GPIOIZQ[6]={32, 33, 25, 26, 27, 12}, GPIO_DER[11]={23, 22, 21, 19, 18,
5, 17, 16, 4, 0, 15}, TOUCH_1=14, TOUCH_2=12;
int contador_FOR=0, SECUENCIA=0, Valor_touch_1=0, ESTADO_1=0, Valor_touch_2=0,
ESTADO_2=0;
```



```
void setup()
{
    for(contador_FOR=0; contador_FOR<=5; contador_FOR++)
    {
        pinMode(GPIO_I2Q[contador_FOR],OUTPUT);
        digitalWrite(GPIO_I2Q[contador_FOR],LOW);
    }
    for(contador_FOR=0; contador_FOR<=10; contador_FOR++)
    {
        pinMode(GPIO_DER[contador_FOR],OUTPUT);
        digitalWrite(GPIO_I2Q[contador_FOR],LOW);
    }
    pinMode(TOUCH_1,INPUT);
    pinMode(TOUCH_2,INPUT);
    Serial.begin(115200);
}

void Loop()
{
    for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
    {
        Valor_touch_1=Valor_touch_1+(100*touchRead(T6));
    }
    Valor_touch_1=Valor_touch_1/50;
    if(Valor_touch_1>=750)
    {
        ESTADO_1=0;
    }
    else
    {
        ESTADO_1=1;
    }

    for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
    {
        Valor_touch_2=Valor_touch_2+(100*touchRead(T5));
    }
}
```



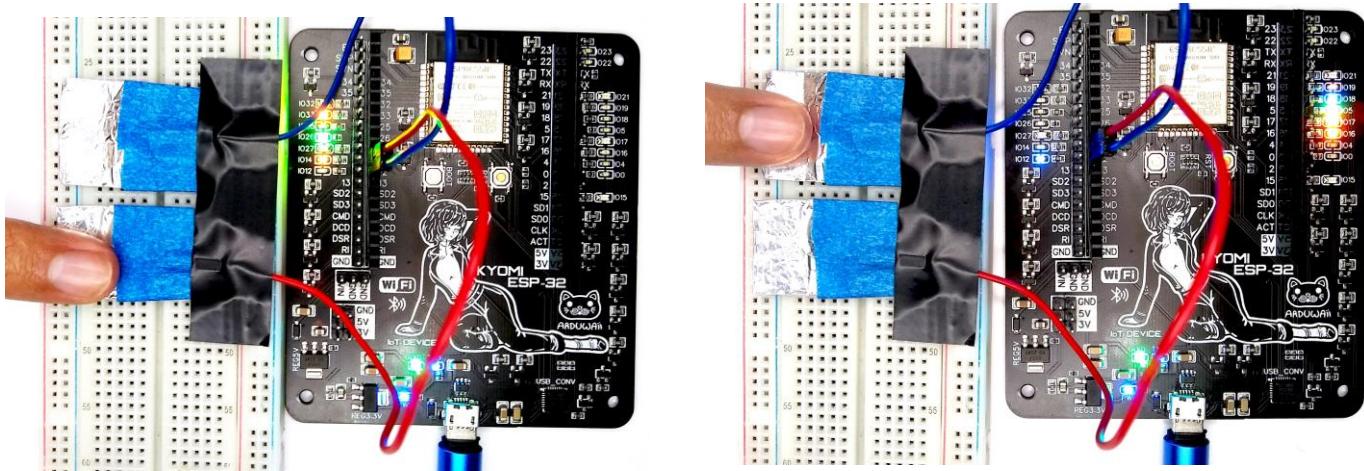
```
Valor_touch_2=Valor_touch_2/50;  
if(Valor_touch_2>=750)  
{  
    ESTADO_2=0;  
}  
else  
{  
    ESTADO_2=1;  
}  
  
if(ESTADO_1==1 && ESTADO_2==0){SECUENCIA=0;}  
if(ESTADO_1==0 && ESTADO_2==1){SECUENCIA=1;}  
SECUENCIA=SECUENCIA;  
  
switch(SECUENCIA)  
{  
    case 0:  
        Serial.println("Secuencia Izquierda");  
        for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],HIGH);  
        }  
        for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],LOW);  
        }  
        delay(80);  
        for(contador_FOR=1; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],HIGH);  
        }  
  
        for(contador_FOR=0; contador_FOR<=4; contador_FOR=contador_FOR+2)  
        {  
            digitalWrite(GPIOIZQ[contador_FOR],LOW);  
        }  
}
```



```
delay(80);  
  
break;  
  
case 1:  
    Serial.println("Secuencia Derecha");  
    for(contador_FOR=3; contador_FOR<=7; contador_FOR++)  
    {  
        digitalWrite(GPIO_DER[contador_FOR],HIGH);  
        delay(20);  
    }  
    for(contador_FOR=3; contador_FOR<=7; contador_FOR++)  
    {  
        digitalWrite(GPIO_DER[contador_FOR],LOW);  
        delay(20);  
    }  
    break;  
}  
  
for(contador_FOR=0; contador_FOR<=5; contador_FOR++)  
{  
    digitalWrite(GPIO_IQZ[contador_FOR],LOW);  
}  
for(contador_FOR=0; contador_FOR<=10; contador_FOR++)  
{  
    digitalWrite(GPIO_IQZ[contador_FOR],LOW);  
}  
}
```



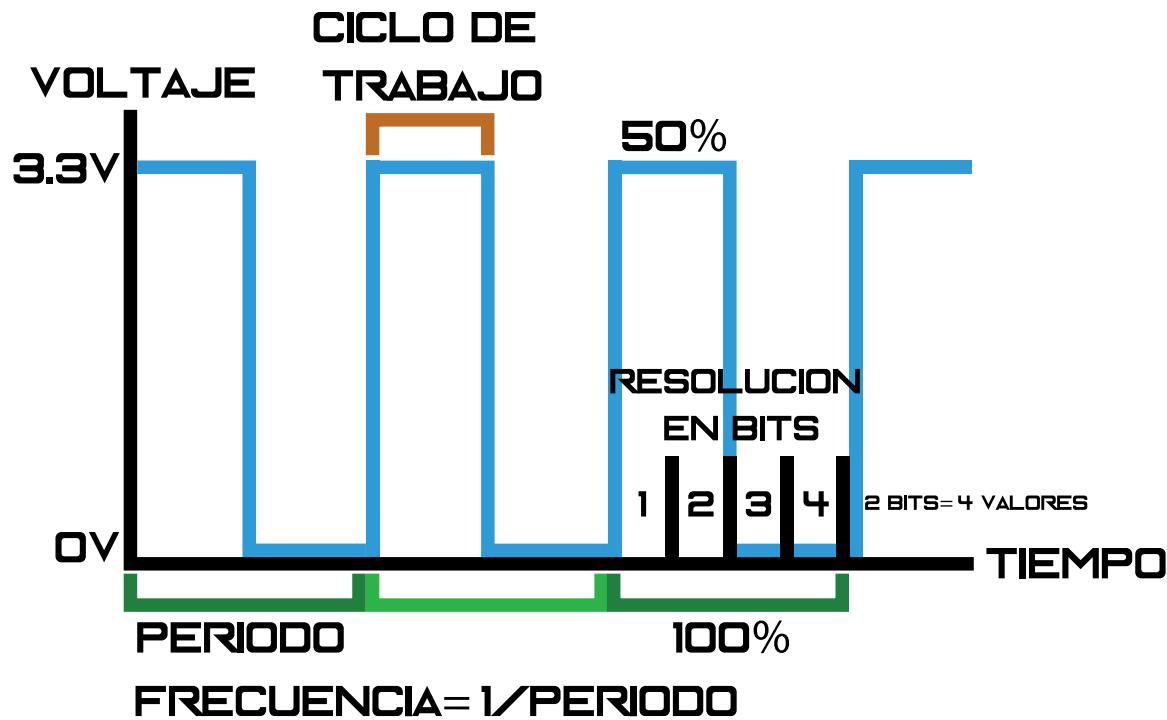
Prueba el programa. Al tocar brevemente un pad u otro, activarás la secuencia de LEDs correspondiente:



## Las Señales PWM (Modulación de Ancho de Pulso)

Las señales de ‘Modulación de Ancho de Pulso’ o PWM (Pulse Width Modulation) son aquellas que están formadas por pulsos de corriente directa y que permiten variar su magnitud de voltaje al modular el ancho de dichos pulsos.

La señal PWM tiene la siguiente estructura y parámetros:

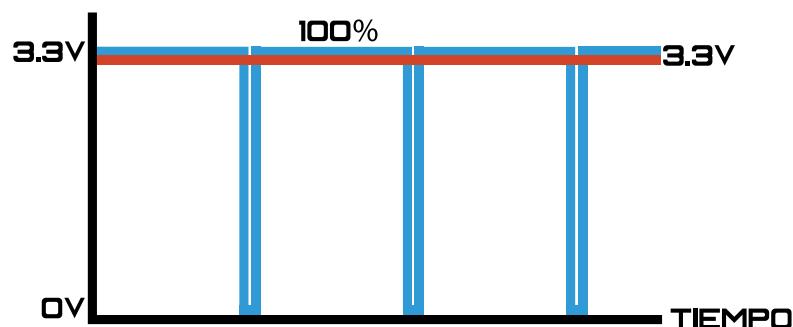
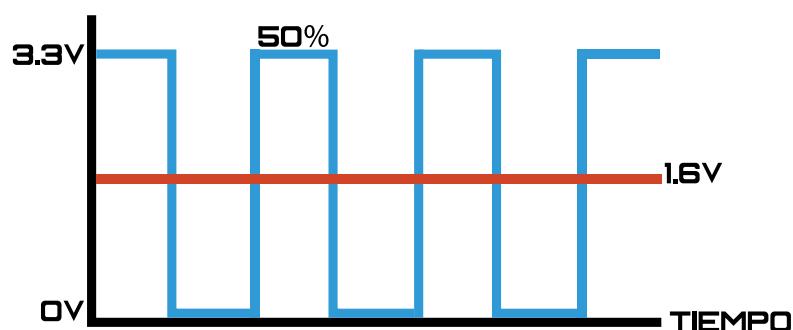
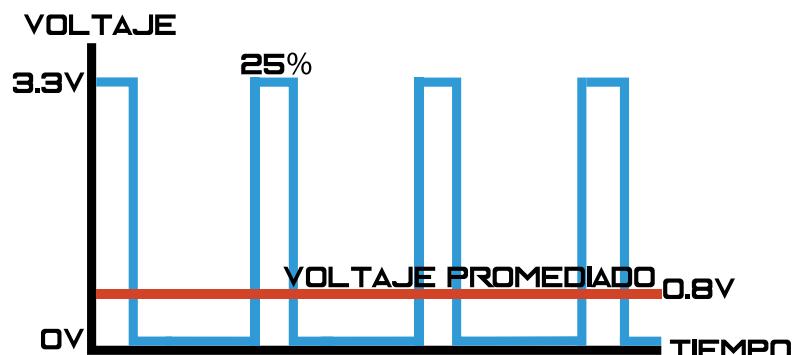




Puedes ver que el PWM es una señal periódica con una frecuencia determinada. Podría decirse que es una señal digital que, durante un periodo, permanece cierta cantidad de tiempo en **HIGH** y el resto en **LOW**. El tiempo que permanece en **HIGH** se le llama ciclo de trabajo ('*Duty Cycle*') y al variar dicho tiempo, estaremos modulando este ciclo.

Para modular el ciclo de trabajo, primero hay que dividir el periodo de la señal en bits (no en tiempo). Por ejemplo, si se tiene una señal PWM de 250Hz (hertz), su periodo será de 1/250Hz o 4 milisegundos (ms). Estos 4ms deben dividirse en cierta cantidad de trozos del mismo tamaño o cierta cantidad de valores. La cantidad de valores en los que se divide el periodo se le llama resolución, que se mide en bits. Por ejemplo, si el PWM tiene una resolución de 2 bits, esto equivale a  $2^2$  valores o 4 valores decimales, otro ejemplo, si la resolución es de 8 bits, entonces tendríamos  $2^8$  o 256 valores para usar.

Al modular el ciclo de trabajo del PWM, podremos variar el voltaje de la propia señal:





Realmente la señal solo es capaz de cambiar entre el voltaje de **HIGH** y **LOW**, pero al alimentar alguna carga (por ejemplo, un LED) y medir la señal con un voltímetro, verás que el voltaje de la señal puede tomar valores entre los 3.3V de **HIGH** y los 0V de **LOW**. Esto ocurre porque el voltaje 'visto' es el resultado de una especie de 'promedio'.

Digamos que la magnitud de voltaje la medimos con las áreas de la parte en **HIGH** y en **LOW** (esto durante un periodo) y hacemos un promedio de dichas áreas con respecto al área total que ocuparía el periodo completo. Si aumenta el ciclo de trabajo, aumenta el área en **HIGH** y el área en **LOW** disminuye, entonces el promedio del área durante un periodo aumenta y, por lo tanto, el voltaje 'visto' también aumenta. Caso contrario, si se reduce el ciclo de trabajo, el área en **HIGH** también se reduce, mientras que el área en **LOW** aumenta, entonces el promedio da como resultado un valor menor, lo que significa menor voltaje. Este tratamiento lo hacen los aparatos de medición como los voltímetros, aunque en las cargas como resistencias, capacitores, inductores, LEDs, etc., no ocurre exactamente así, depende de más factores.

La ventaja del PWM del ESP32, comparado con los Arduinos clásicos, es que podemos configurar la frecuencia y resolución de bits de la señal, mientras que en el Arduino UNO y el Nano, por ejemplo, la frecuencia de sus señales PWM es fija (500Hz o 1KHz aprox. según el pin) y tiene una resolución estática de 8 bits (ambos parámetros se pueden modificar con programación avanzada).

En el ESP32, la frecuencia del PWM se puede configurar a un máximo de 40MHz, pero a una resolución de 1bit, ya que las altas frecuencias tienen un periodo muy pequeño y el procesador no es capaz de dividirlo en porciones aún más pequeñas. A menor frecuencia, puedes usar una resolución de bits mayor, pero a mayor frecuencia, debes reducir la resolución.

Para generar una señal PWM en el ESP32, necesitaremos unas cuantas variables: una con el número de GPIO a usar, dos con valor fijo para la frecuencia y resolución y una más para el valor del ciclo de trabajo:

```
const int GPIO_PWM=PIN, FRECUENCIA=VALOR EN Hz, RESOLUCION=VALOR EN BITS;  
int VALOR_PWM=0;
```

En el ESP32 podemos organizar los GPIO's PWM en 16 canales (los canales van del 0 al 15). Realmente vamos a controlar el canal y no el GPIO. Por ejemplo, si asignamos dos GPIO al canal 0, esos dos GPIO trabajarán igual cuando estemos controlando el ciclo de trabajo. La ventaja de esto, es que distintos canales pueden trabajar a frecuencias y resoluciones diferentes, y podemos asignar uno o más GPIO's a un mismo canal. Entonces podemos generar hasta 16 señales PWM diferentes.



En `setup()` vamos a configurar los canales PWM de la siguiente manera:

```
LedcSetup(NUMERO DE CANAL, FRECUENCIA, RESOLUCION);  
LedcAttachPin(GPIO_PWM, NUMERO DE CANAL);  
pinMode(GPIO_PWM, OUTPUT);
```

donde *NUMERO DE CANAL* es el canal que vamos a utilizar. En `LedcAttachPin()` asignamos los GPIO's al canal previamente configurado con `LedcSetup()`. No es del todo necesario usar `pinMode` para configurar los GPIO's PWM como *OUTPUT*.

En `loop()` generamos la señal PWM y controlamos su ciclo de trabajo, asignando un valor a *VALOR\_PWM* dentro de la siguiente instrucción:

```
LedcWrite(NUMERO DE CANAL, VALOR_PWM);
```

Los valores para el ciclo de trabajo empiezan en 0. Ahora pasemos a los ejercicios.

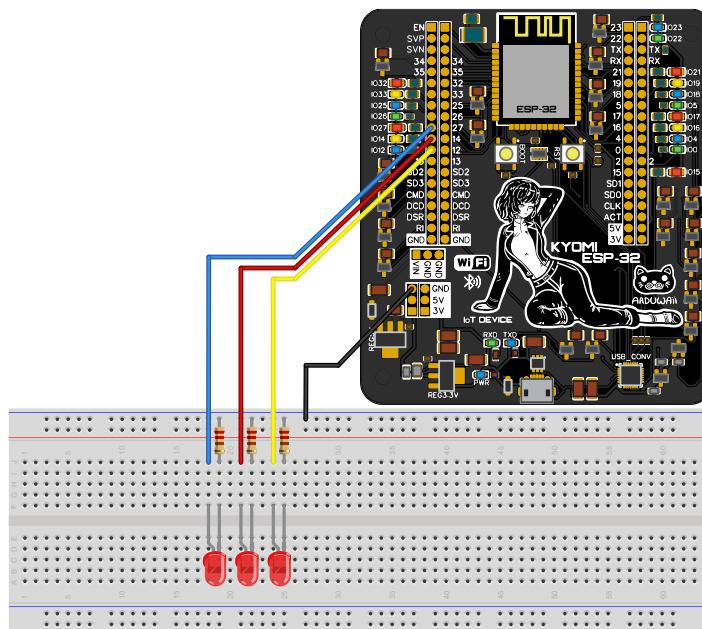
## Ejercicio 5

En este ejercicio variaremos el brillo de unos LEDs automáticamente, alimentándolos con señales PWM.

Materiales:

- 3 LEDs (opcional)
- 3 resistencias de 220Ω (opcional)
- 4 jumpers o cables tipo Dupont (para los LEDs)
- plantilla de experimentos

El circuito es el siguiente:





Veamos el código. Usaremos dos canales PWM: uno configurado con una frecuencia de 100KHz (100000 Hz) y resolución de 8 bits, mientras que el otro trabajará a 20KHz (20000Hz) y con una resolución de 10 bits. También vamos a declarar un par de variables para el ciclo de trabajo de los dos canales:

```
const int GPIO_27=27, GPIO_14=14, GPIO_12=12, FRECUENCIA_1=100000,  
RESOLUCION_1=8, FRECUENCIA_2=20000, RESOLUCION_2=10;  
  
int VALOR_PWM_1=0, VALOR_PWM_2=0;
```

En `setup()` asignaremos el GPIO27 y 14 al canal 0, mientras que el GPIO12 estará en el canal 1:

```
LedcSetup(0, FRECUENCIA_1, RESOLUCION_1);  
LedcAttachPin(GPIO_27, 0);  
LedcAttachPin(GPIO_14, 0);  
  
LedcSetup(1, FRECUENCIA_2, RESOLUCION_2);  
LedcAttachPin(GPIO_12, 1);  
  
pinMode(GPIO_27,OUTPUT);  
pinMode(GPIO_14,OUTPUT);  
pinMode(GPIO_12,OUTPUT);
```

En `Loop()` usaremos ciclos `for` para incrementar y reducir los valores de `VALOR_PWM_1` y `VALOR_PWM_2` (así variaremos el voltaje de los LEDs y su brillo):

```
for(VALOR_PWM_1=0; VALOR_PWM_1<=255; VALOR_PWM_1++)  
{  
    LedcWrite(0, VALOR_PWM_1);  
    delay(4);  
}  
  
for(VALOR_PWM_1=255; VALOR_PWM_1>=0; VALOR_PWM_1--)  
{  
    LedcWrite(0, VALOR_PWM_1);  
    delay(4);  
}
```

Para el incremento de `VALOR_PWM_1` (en el canal 0), el ciclo `for` empezará en 0 y acabará cuando su valor sea mayor o igual a 255 (8bits=256 valores). Para el decremento de `VALOR_PWM_1`, ahora el ciclo empieza con 255 y terminará cuando el valor sea menor que 0. Para este caso, en `ACUMULADOR` indicamos que se reste un 1 a `VALOR_INICIAL` en cada ciclo. La pausa de 4ms



[*delay*(4)] nos permitirá ver la variación del brillo del LED con cada cambio de *VALOR\_PWM\_1*.

Para el canal 1, *VALOR\_PWM\_2* podrá tomar valores de 0 a 1023 (10bits=1024 valores) y se hará el incremento y decremento dentro de este rango. También tendrá una pausa de 4ms en cada cambio de *VALOR\_PWM\_2*:

```
for(VALOR_PWM_2=0; VALOR_PWM_2<=1023; VALOR_PWM_2++)
{
    LedcWrite(1, VALOR_PWM_2);
    delay(4);
}

for(VALOR_PWM_1=1023; VALOR_PWM_1>=0; VALOR_PWM_1--)
{
    LedcWrite(1, VALOR_PWM_2);
    delay(4);
}
```

El código completo queda de la siguiente forma:

```
const int GPIO_27=27, GPIO_14=14, GPIO_12=12, FRECUENCIA_1=100000,
RESOLUCION_1=8, FRECUENCIA_2=20000, RESOLUCION_2=10;

int VALOR_PWM_1=0, VALOR_PWM_2=0;

void setup()
{
    LedcSetup(0, FRECUENCIA_1, RESOLUCION_1);
    LedcAttachPin(GPIO_27, 0);
    LedcAttachPin(GPIO_14, 0);

    LedcSetup(1, FRECUENCIA_2, RESOLUCION_2);
    LedcAttachPin(GPIO_12, 1);

    pinMode(GPIO_27,OUTPUT);
    pinMode(GPIO_14,OUTPUT);
    pinMode(GPIO_12,OUTPUT);
}

void loop()
{
    for(VALOR_PWM_1=0; VALOR_PWM_1<=255; VALOR_PWM_1++)
    {
        LedcWrite(0, VALOR_PWM_1);
        delay(4);
    }
}
```



```
delay(4);  
}  
  
for(VALOR_PWM_1=255; VALOR_PWM_1>=0; VALOR_PWM_1--)  
{  
    LedcWrite(0, VALOR_PWM_1);  
    delay(4);  
}  
  
for(VALOR_PWM_2=0; VALOR_PWM_2<=1023; VALOR_PWM_2++)  
{  
    LedcWrite(1, VALOR_PWM_2);  
    delay(4);  
}  
  
for(VALOR_PWM_1=1023; VALOR_PWM_1>=0; VALOR_PWM_1--)  
{  
    LedcWrite(1, VALOR_PWM_2);  
    delay(4);  
}  
}
```

Prueba el programa. Notarás que el LED del GPIO12 tarda más tiempo en aumentar y disminuir el brillo, en comparación a los LEDs del GPIO27 y 14. Esto ocurre porque el GPIO12 hace más pausas de 4ms, ya que su señal PWM tiene que hacer 1024 cambios de valor de ciclo de trabajo y en los GPIO27 y 14 ocurren solo 256 cambios.

Antes de pasar el siguiente ejercicio, primero veamos qué son las subrutinas y las variables globales y locales.

## Básicos: Las subrutinas en Arduino

Las subrutinas (o funciones) son una herramienta de programación que nos permite crear rutinas de instrucciones en grupos separados. Las rutinas `setup()` y `loop()` se consideran como principales (siendo `loop()` la que se ejecuta repetidamente), pero podemos construir otras subrutinas que puedan ser ‘llamadas’ desde las principales. Tienen la siguiente estructura:

```
PARAMETRO SUB_RUTINA(VALORES COMPARTIDOS)  
{  
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN  
    return(VALOR A DEVOLVER);  
}
```



donde *PARAMETRO* es el tipo de valor que va a devolver la subrutina, ya sea *int*, *float*, *double*, etc., pero, si la subrutina no va a devolver ningún valor, entonces *PARAMETRO* se puede configurar como *void*. En *VALORES COMPARTIDOS* se colocan las variables cuyo valor será traído de la rutina principal hacia la subrutina, o incluso desde otra subrutina. Si no es necesario traer algún valor desde otra rutina, *VALORES COMPARTIDOS* se puede dejar vacío. La instrucción *return* se usa para especificar a la variable cuyo valor será devuelto a la rutina principal.

Para construir una subrutina sencilla, que no devolverá ni llamará valores, podemos usar la siguiente estructura:

```
void Loop()
{
    SUB_RUTINA();
}

void SUB_RUTINA()
{
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN
}
```

Para construir una subrutina más completa, que devuelva y llame valores, podemos usar la estructura del siguiente ejemplo:

```
void Loop()
{
    A=3;
    B=7;
    C=FUNCION_DE_SUMA(A,B);
}

int FUNCION_DE_SUMA(int X, int Y)
{
    Z=X+Y;
    return(Z);
}
```

donde la variable *c* da como resultado el valor que es devuelto por *FUNCION\_DE\_SUMA*, y donde los valores de *A* y *B* son enviados a *FUNCION\_DE\_SUMA* y asignados a *x* y *y* respectivamente. La variable *z* da como resultado la suma de *x* y *y*. Finalmente, el resultado de *z* es devuelto a la variable *c*.



Las subrutinas nos permitirán agrupar instrucciones o crear funciones para organizar mejor nuestro programa, e incluso nos ayudarán al momento de modificar nuestro programa.

## Básicos: Las variables globales y locales

Cuando trabajamos con subrutinas o funciones secundarias, tenemos la posibilidad de declarar variables que funcionen en todo el programa o crear variables que sean de uso exclusivo para ciertas subrutinas.

**Las variables globales** son aquellas que podemos usar (o modificar) en cualquier rutina principal o subrutina del programa. Son las variables que declaramos fuera de `setup()` y `Loop()`.

**Las variables locales** son aquellas que se declaran dentro de una rutina principal o subrutina, por lo que sólo pueden ser usadas o modificadas dentro de la misma. Estas variables no son vistas por otras rutinas y tampoco las pueden usar, pero dos rutinas sí pueden tener variables declaradas con los mismos nombres, ya que una subrutina no puede ver las variables de la otra ni saber que tienen los mismos nombres. Veamos algunos ejemplos:

- Variable local declarada dentro de una subrutina:

```
void SUB_RUTINA()
{
    int VARIABLE;
}
```

- Variable local declarada dentro de una instrucción:

```
for(int CONTADOR=0; CONTADOR>=10; CONTADOR++)
{
    float X=5;
    if(X==5)
    {
        float Y=20;
    }
}
```

- Variables locales declaradas en la cabecera de la subrutina y dentro de la misma:

```
int SUBRUTINA(int X, int Y, int Z)
{
    int S;
    S=X+Y-Z;
    return(S);
}
```

Veamos otro ejercicio con PWM, aplicando subrutinas.



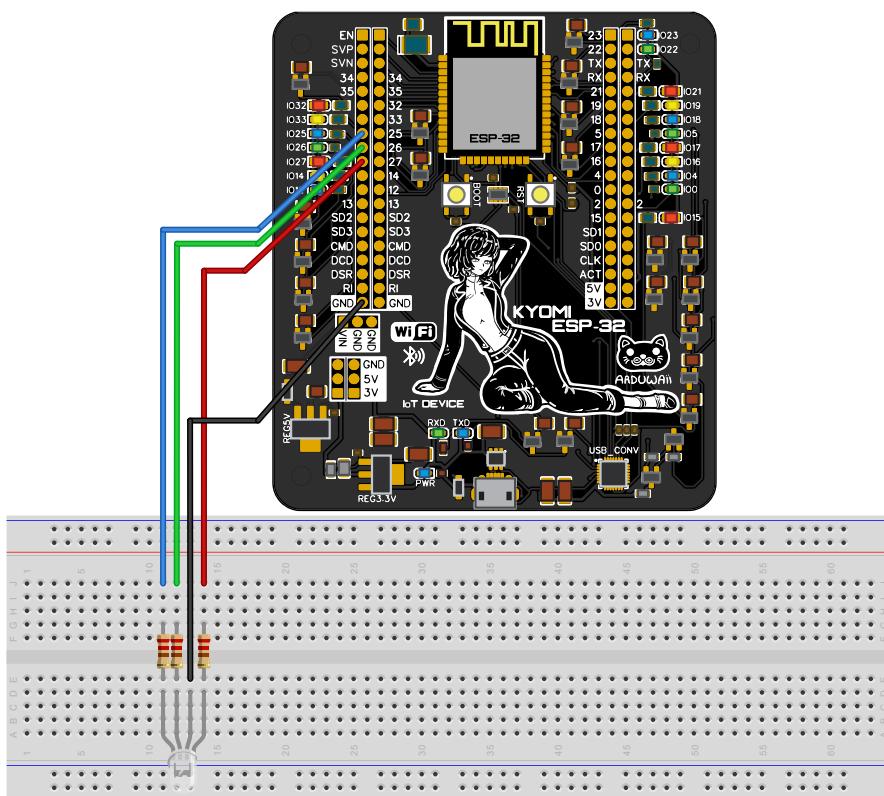
## Ejercicio 6

En este ejercicio controlaremos un LED RGB (rojo, verde, azul) con señales PWM para generar luz de distintos colores, variando el brillo de sus tres colores principales.

Materiales:

- 1 LED RGB de cátodo común o un módulo de LED RGB (como el KY-016 o KY-009)
- 3 resistencias de 220Ω (no son necesarias si usas el módulo de LED RGB)
- 4 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:



Veamos el código. En la declaración de variables sólo necesitaremos constantes para los GPIO's y los parámetros de PWM. En este caso usaremos el GPIO25, 26 y 27, mientras que el PWM estará configurado a 5KHz y con una resolución de 6 bits (64 valores):

```
const int GPIO_ROJO=27, GPIO_VERDE=26, GPIO_AZUL=25, FRECUENCIA=5000,  
RESOLUCION=6;
```



En `setup()` se asignan los canales PWM 0, 1 y 2 al GPIO27, 26 y 25 respectivamente, además de configurarlos como `OUTPUT` y todos trabajando con la misma frecuencia y resolución de bits:

```
void setup()
{
    LedcSetup(0, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_ROJO, 0);

    LedcSetup(1, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_VERDE, 1);

    LedcSetup(2, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_AZUL, 2);

    pinMode(GPIO_ROJO, OUTPUT);
    pinMode(GPIO_VERDE, OUTPUT);
    pinMode(GPIO_AZUL, OUTPUT);
}
```

En `Loop()`, llamaremos a las funciones o subrutinas que generarán las señales PWM, para regular el brillo de los colores principales del LED RGB, obteniendo algunas combinaciones de color:

```
void Loop()
{
    COLOR_ROJO();
    COLOR_VERDE();
    COLOR_AZUL();
    COLOR_AMARILLO(63,16,0);
    COLOR_MORADO(63,0,32);
    COLOR_CIAN(0,63,63);
}
```

En `COLOR_ROJO()`, `COLOR_VERDE()` y `COLOR_AZUL()` no enviaremos ningún valor a las subrutinas, pero en `COLOR_AMARILLO()`, `COLOR_MORADO()` y `COLOR_CIAN()` enviaremos algunos valores PWM para realizar las combinaciones de color. Todas las subrutinas se configuran como `void`, ya que no devolverán ningún valor. En `COLOR_ROJO()`, `COLOR_VERDE()` y `COLOR_AZUL()` se configura un valor PWM máximo (63) para el color respectivo y los otros dos colores que no se usarán se configuran con el valor mínimo (0). Por ejemplo, para `COLOR_ROJO()`, el LED RGB se configura con el PWM rojo en 64 y el verde y



azul en 0. En estas subrutinas, después de generar las señales PWM, se hace una pausa de 1 segundo con `delay(1000)`, para que podamos observar el color del led RGB:

```
void COLOR_ROJO()
{
    LedcWrite(0,63);
    LedcWrite(1,0);
    LedcWrite(2,0);
    delay(1000);
}

void COLOR_VERDE()
{
    LedcWrite(0,0);
    LedcWrite(1,63);
    LedcWrite(2,0);
    delay(1000);
}

void COLOR_AZUL()
{
    LedcWrite(0,0);
    LedcWrite(1,0);
    LedcWrite(2,63);
    delay(1000);
}
```

En `COLOR_AMARILLO()`, `COLOR_MORADO()` y `COLOR_CIAN()` se traerán los valores que se les asignaron en `loop()` y se guardarán en las variables locales `ROJO`, `VERDE` y `AZUL`, las cuales se usarán para los valores PWM. Igualmente se hace una pausa de 1 segundo:

```
void COLOR_AMARILLO(int ROJO, int VERDE, int AZUL)
{
    LedcWrite(0,ROJO);
    LedcWrite(1,VERDE);
    LedcWrite(2,AZUL);
    delay(1000);
}
```



```
void COLOR_MORADO(int ROJO, int VERDE, int AZUL)
{
    LedcWrite(0,ROJO);
    LedcWrite(1,VERDE);
    LedcWrite(2,AZUL);
    delay(1000);
}

void COLOR_CIAN(int ROJO, int VERDE, int AZUL)
{
    LedcWrite(0,ROJO);
    LedcWrite(1,VERDE);
    LedcWrite(2,AZUL);
    delay(1000);
}
```

El código completo queda de la siguiente forma:

```
const int GPIO_ROJO=27,     GPIO_VERDE=26,     GPIO_AZUL=25,      FRECUENCIA=5000,
RESOLUCION=6;

void setup()
{
    LedcSetup(0, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_ROJO, 0);

    LedcSetup(1, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_VERDE, 1);

    LedcSetup(2, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_AZUL, 2);

    pinMode(GPIO_ROJO,OUTPUT);
    pinMode(GPIO_VERDE,OUTPUT);
    pinMode(GPIO_AZUL,OUTPUT);
}
```



```
void Loop()
{
    COLOR_ROJO();
    COLOR_VERDE();
    COLOR_AZUL();
    COLOR_AMARILLO(63,16,0);
    COLOR_MORADO(63,0,32);
    COLOR_CIAN(0,63,63);
}

void COLOR_ROJO()
{
    LedcWrite(0,63);
    LedcWrite(1,0);
    LedcWrite(2,0);
    delay(1000);
}

void COLOR_VERDE()
{
    LedcWrite(0,0);
    LedcWrite(1,63);
    LedcWrite(2,0);
    delay(1000);
}

void COLOR_AZUL()
{
    LedcWrite(0,0);
    LedcWrite(1,0);
    LedcWrite(2,63);
    delay(1000);
}
```

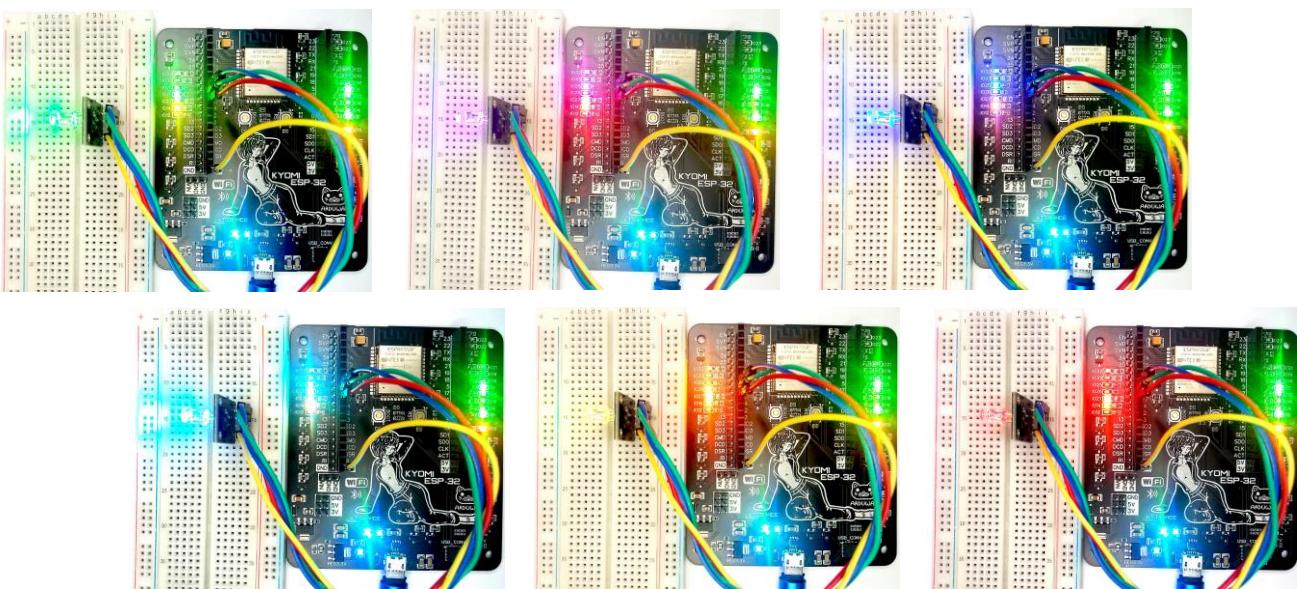


```
void COLOR_AMARILLO(int ROJO, int VERDE, int AZUL)
{
    LedcWrite(0,ROJO);
    LedcWrite(1,VERDE);
    LedcWrite(2,AZUL);
    delay(1000);
}

void COLOR_MORADO(int ROJO, int VERDE, int AZUL)
{
    LedcWrite(0,ROJO);
    LedcWrite(1,VERDE);
    LedcWrite(2,AZUL);
    delay(1000);
}

void COLOR_CIAN(int ROJO, int VERDE, int AZUL)
{
    LedcWrite(0,ROJO);
    LedcWrite(1,VERDE);
    LedcWrite(2,AZUL);
    delay(1000);
}
```

Prueba el programa. Los LEDs integrados de la placa KYOMI te ayudarán a ver cuáles colores del LED RGB están activos y con cuánto brillo:



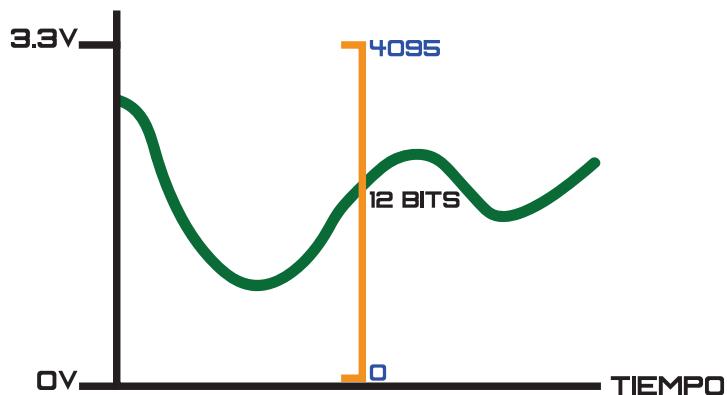


## Los GPIO's como entradas analógicas (ADC)

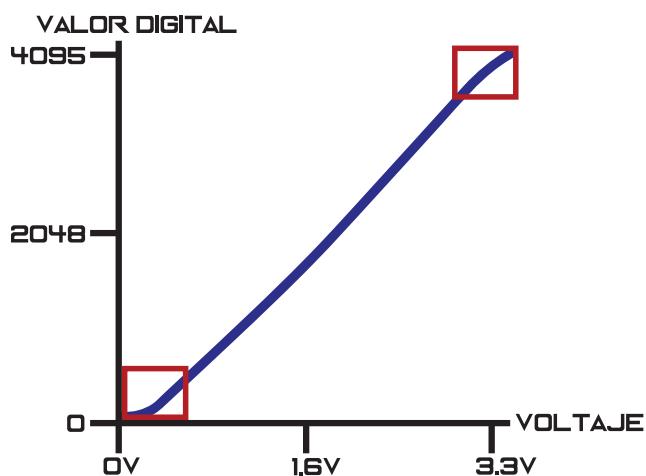
Las señales analógicas son todas aquellas que presentan variaciones a lo largo del tiempo y no pueden ser definidas dentro de un rango de valores bien establecidos, como en las señales digitales.

Los GPIO's pueden medir señales analógicas convirtiéndolas en valores digitales, usando el convertidor analógico-digital (ADC), que parte de un principio similar al que vimos con los sensores touch y el PWM.

El rango de voltaje del ESP32 para medir una señal analógica va de 0V hasta 3.3V y es capaz de digitalizarla con una resolución de 12bits (4096 valores). A diferencia de la señal PWM, la resolución de bits aplicada por el ADC se hace sobre la magnitud de la señal y no sobre un periodo determinado:



Entonces, la señal analógica puede tomar valores entre 0 y 4095 dependiendo de la magnitud de su voltaje. Cabe aclarar que el ADC no hace una conversión totalmente lineal y tiene un comportamiento similar a este:



esto quiere decir que el ADC no es capaz de diferenciar entre valores muy pequeños o muy grandes. Por ejemplo, 0V y 0.1V los ve como 0 en valor digital, mientras que 3.2V y 3.3V los ve como 4095. Este es un



problema típico de muchos microcontroladores, y no es muy recomendado usar el ADC para medir voltajes muy pequeños o precisos.

Para realizar las lecturas analógicas, usaremos la siguiente instrucción:

`analogRead(PIN)`

donde en `PIN` simplemente colocamos el número de GPIO (revisa en el pinout cuales GPIO están conectados a los ADC's).

Es muy recomendable aplicar un tratamiento matemático a las lecturas analógicas para reducir los ruidos eléctricos e interferencias, así obtendremos mediciones digitales más estables. Podemos aplicar el tratamiento de promedio que usamos con los GPIO's touch:

```
for(contador_FOR=0; contador_FOR<=50; contador_FOR++)  
{  
    LECTURA_ANALOGICA=LECTURA_ANALOGICA+analogRead(PIN);  
}  
LECTURA_ANALOGICA=LECTURA_ANALOGICA/50;
```

**NOTA:** No se recomienda medir señales analógicas con frecuencias superiores a los 3KHz, ya que `analogRead()` toma muestras de la señal a una frecuencia relativamente baja (6KHz aprox.) y esto puede distorsionar las lecturas obtenidas. Usando programación avanzada se pueden medir señales de más de 3Khz, pero en estos casos es más recomendable utilizar un módulo ADC externo.

Los GPIO's con ADC están organizados en dos grupos o asignados a dos convertidores: ADC-1 y ADC-2. Cada GPIO de cada ADC también está asignado a un canal independiente (similar a los canales PWM), así que todos estos GPIO's pueden realizar mediciones diferentes.

**NOTA:** Los GPIO's del ADC-2 no pueden ser usados al mismo tiempo con la comunicación Wifi, ya que el radio Wifi también usa el ADC-2 para ciertas funciones. En este caso, usa los GPIO's del ADC-1.

En el programa, podemos reconvertir el valor digital de `analogRead()` al valor de voltaje real, usando las siguientes instrucciones:

```
LECTURA_ANALOGICA=analogRead(PIN);  
VOLTAJE=(3.3/4096)*LECTURA_ANALOGICA;
```

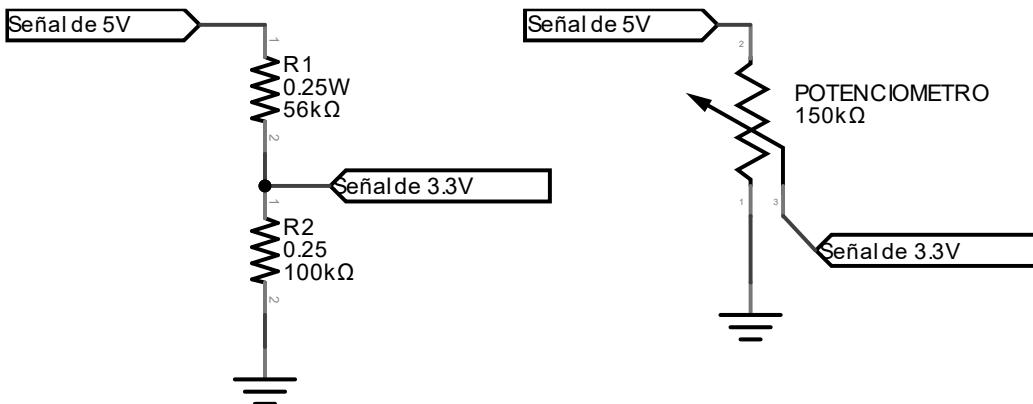
Antes de pasar a los ejercicios, veamos cómo construir un divisor de voltaje para los GPIO's del ADC y la estructura cíclica 'while'.



## Básicos: El divisor de voltaje

Debido a que el voltaje máximo que podemos aplicar a un GPIO es 3.3V, cualquier señal con un voltaje máximo que sea mayor a 3.3V puede dañar a los GPIO's. Para poder medir señales con voltajes superiores, hay que utilizar un divisor de voltaje.

Un divisor de voltaje básicamente es un circuito con dos resistencias en serie, como el del siguiente ejemplo:



entre las dos resistencias obtenemos el voltaje de salida deseado. De hecho, los potenciómetros o resistencias variables están construidos como divisores de voltaje ajustables. Si aumenta o disminuye el voltaje de la señal de entrada, también lo hará el voltaje de salida, esto de manera proporcional. Es como si redujéramos la escala del voltaje.

Para calcular las resistencias del divisor, usaremos la siguiente ecuación:

$$\text{Voltaje salida} = \frac{R2}{R1 + R2} * \text{Voltaje entrada}$$

donde el voltaje de entrada y de salida máximos ya los conoces, así que solo hay que proponer el valor de una resistencia y despejar el valor de la resistencia restante. Usa valores de resistencias comerciales. Para el circuito del ejemplo anterior, tenemos:

$$3.3V = \frac{100K\Omega}{100K\Omega + R1} * 5V \rightarrow R1 = \frac{100K\Omega(5V)}{3.3V} - 100K\Omega \quad R1 = 51.5K\Omega$$

entonces, para R2 usaremos una resistencia de 100KΩ y para R1 una resistencia de 56KΩ, que es el valor comercial más cercano a 51.5 KΩ. Es recomendable usar valores de resistencia altos como los del ejemplo, especialmente cuando midas señales analógicas débiles, ya que así el circuito del divisor consumirá poca corriente y no afectará a la señal medida.



Para obtener el voltaje real de la medición, podemos despejar nuevamente la ecuación del divisor de voltaje de la siguiente manera:

$$\frac{(R1 + R2) * \text{Voltaje salida}}{R2} = \text{Voltaje entrada}$$

esta ecuación la podemos aplicar en el programa usando la siguiente instrucción (con una lectura de `analogRead` previamente convertida a voltaje):

```
VOLTAJE_ENTRADA_REAL=(VOLTAJE*(R1+R2))/R2;
```

## Básicos: Estructura cíclica 'while'

El ciclo `while` es otra estructura cíclica que necesitarás. Su estructura es más simple que la del ciclo `for`, ya que no depende de un `VALOR INICIAL` ni un `ACUMULADOR` para contar los ciclos. El ciclo `while` se ejecuta siempre y cuando se cumpla una condición establecida, así que la cantidad de ciclos que realizará no necesariamente es finita, a diferencia del ciclo `for` (que sí realiza cierta cantidad de ciclos).

El ciclo `while` tiene la siguiente estructura:

```
while(CONDICIÓN)
{
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN
}
```

donde `CONDICIÓN` es una condición lógica que deberá cumplirse para que el ciclo continúe ejecutándose o se detenga. El ciclo `do...while` es una configuración especial del `while` y tiene la siguiente estructura:

```
do
{
    // AQUÍ VAN LAS INSTRUCCIONES QUE SE EJECUTARÁN
}
while(CONDICIÓN)
```

a diferencia del ciclo `while`, en `do...while` primero se ejecutan las instrucciones del ciclo y luego se evalúa la `CONDICIÓN`, mientras que en `while` primero se evalúa la `CONDICIÓN` y luego se ejecutan las instrucciones.



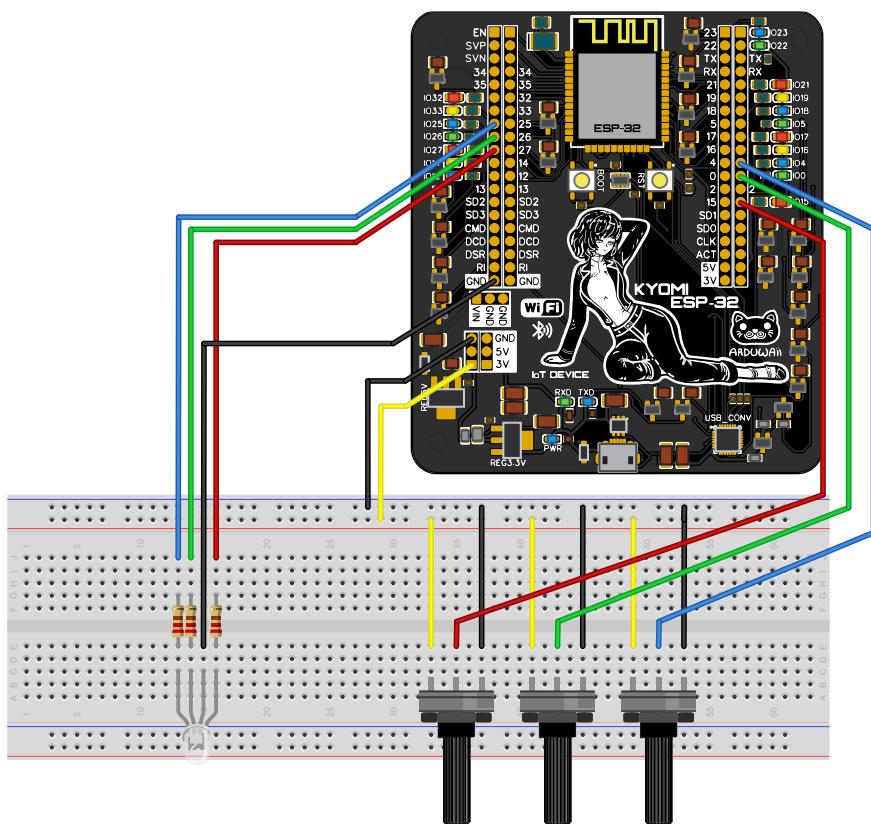
## Ejercicio 7

En este ejercicio controlaremos un LED RGB a través de la lectura de señales de voltaje analógico, que serán controladas con potenciómetros, y usaremos los valores de las lecturas analógicas como valores de ciclo de trabajo PWM.

Materiales:

- 1 LED RGB de cátodo común o un módulo de LED RGB (como el KY-016 o KY-009)
- 3 resistencias de 220Ω (no son necesarias si usas el módulo de LED RGB)
- 3 potenciómetros de 1KΩ
- 15 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:



Veamos el código. En la declaración de variables tendremos tanto a los GPIO's del LED RGB como a los GPIO's conectados a los potenciómetros. Usaremos una frecuencia de 5KHz para el PWM, con una resolución de 12 bits, esto para hacer coincidir los valores de 12 bits del ADC y no tener



que hacer alguna conversión. También tendremos unas cuantas variables para guardar las lecturas analógicas y para el *for*:

```
const int GPIO_ROJO=27, GPIO_VERDE=26, GPIO_AZUL=25;  
const int GPIO_ROJO_ANALOGICO=15, GPIO_VERDE_ANALOGICO=0, GPIO_AZUL_ANALOGICO=4;  
const int FRECUENCIA=5000, RESOLUCION=12;  
int VALOR_ANALOGICO_ROJO=0, VALOR_ANALOGICO_VERDE=0, VALOR_ANALOGICO_AZUL=0,  
contador_FOR;
```

En *setup()* asignamos los GPIO's del LED RGB a tres canales PWM independientes y los configuramos como *OUTPUT*, de igual manera, los GPIO's analógicos los configuramos como *INPUT*, aunque esto último no es estrictamente necesario. También configuraremos la comunicación serial:

```
LedcSetup(0, FRECUENCIA, RESOLUCION);  
LedcAttachPin(GPIO_ROJO, 0);  
  
LedcSetup(1, FRECUENCIA, RESOLUCION);  
LedcAttachPin(GPIO_VERDE, 1);  
  
LedcSetup(2, FRECUENCIA, RESOLUCION);  
LedcAttachPin(GPIO_AZUL, 2);  
  
pinMode(GPIO_ROJO, OUTPUT);  
pinMode(GPIO_VERDE, OUTPUT);  
pinMode(GPIO_AZUL, OUTPUT);  
  
pinMode(GPIO_ROJO_ANALOGICO, INPUT);  
pinMode(GPIO_VERDE_ANALOGICO, INPUT);  
pinMode(GPIO_AZUL_ANALOGICO, INPUT);  
  
Serial.begin(115200);
```

Dentro de *loop()*, las lecturas analógicas se promediarán en tres ciclos *for* (correspondientes a las lecturas de cada potenciómetro):

```
for(contador_FOR=0; contador_FOR<=50; contador_FOR++)  
{  
    VALOR_ANALOGICO_ROJO=VALOR_ANALOGICO_ROJO+analogRead(GPIO_ROJO_ANALOGICO);  
}  
VALOR_ANALOGICO_ROJO=VALOR_ANALOGICO_ROJO/50;
```



```
for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
{
    VALOR_ANALOGICO_VERDE=VALOR_ANALOGICO_VERDE+analogRead(GPIO_ROJO_ANALOGICO);
}
VALOR_ANALOGICO_VERDE=VALOR_ANALOGICO_VERDE/50;

for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
{
    VALOR_ANALOGICO_AZUL=VALOR_ANALOGICO_AZUL+analogRead(GPIO_AZUL_ANALOGICO);
}
VALOR_ANALOGICO_AZUL=VALOR_ANALOGICO_AZUL/50;
```

Los resultados de los promedios se enviarán directamente a los `LedcWrite()` para regular el brillo de los colores del LED RGB:

```
LedcWrite(0,VALOR_ANALOGICO_ROJO);
LedcWrite(1,VALOR_ANALOGICO_VERDE);
LedcWrite(2,VALOR_ANALOGICO_AZUL);
```

Finalmente, los resultados de los promedios son enviados por puerto serie al Graficador Serial, esto para ver el comportamiento de los ajustes de los potenciómetros. Ordenaremos las variables de tal manera que las señales de los colores RGB coincidan con los colores rojo, verde y azul que asigna el graficador serial:

```
Serial.print(VALOR_ANALOGICO_AZUL);
Serial.print("\t");
Serial.print(VALOR_ANALOGICO_ROJO);
Serial.print("\t");
Serial.println(VALOR_ANALOGICO_VERDE);
```

El código completo queda de la siguiente forma:

```
const int GPIO_ROJO=27, GPIO_VERDE=26, GPIO_AZUL=25;
const int GPIO_ROJO_ANALOGICO=15, GPIO_VERDE_ANALOGICO=0, GPIO_AZUL_ANALOGICO=4;
const int FRECUENCIA=5000, RESOLUCION=12;

int VALOR_ANALOGICO_ROJO=0, VALOR_ANALOGICO_VERDE=0, VALOR_ANALOGICO_AZUL=0,
contador_FOR;

void setup()
{
    LedcSetup(0, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_ROJO, 0);
```



```
LedcSetup(1, FRECUENCIA, RESOLUCION);
LedcAttachPin(GPIO_VERDE, 1);

LedcSetup(2, FRECUENCIA, RESOLUCION);
LedcAttachPin(GPIO_AZUL, 2);

pinMode(GPIO_ROJO,OUTPUT);
pinMode(GPIO_VERDE,OUTPUT);
pinMode(GPIO_AZUL,OUTPUT);

pinMode(GPIO_ROJO_ANALOGICO,INPUT);
pinMode(GPIO_VERDE_ANALOGICO,INPUT);
pinMode(GPIO_AZUL_ANALOGICO,INPUT);

Serial.begin(115200);

}

void Loop()
{
    for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
    {
        VALOR_ANALOGICO_ROJO=VALOR_ANALOGICO_ROJO+analogRead(GPIO_ROJO_ANALOGICO);
    }
    VALOR_ANALOGICO_ROJO=VALOR_ANALOGICO_ROJO/50;

    for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
    {
        VALOR_ANALOGICO_VERDE=VALOR_ANALOGICO_VERDE+analogRead(GPIO_VERDE_ANALOGICO);
    }
    VALOR_ANALOGICO_VERDE=VALOR_ANALOGICO_VERDE/50;

    for(contador_FOR=0; contador_FOR<=50; contador_FOR++)
    {
        VALOR_ANALOGICO_AZUL=VALOR_ANALOGICO_AZUL+analogRead(GPIO_AZUL_ANALOGICO);
    }
    VALOR_ANALOGICO_AZUL=VALOR_ANALOGICO_AZUL/50;
```

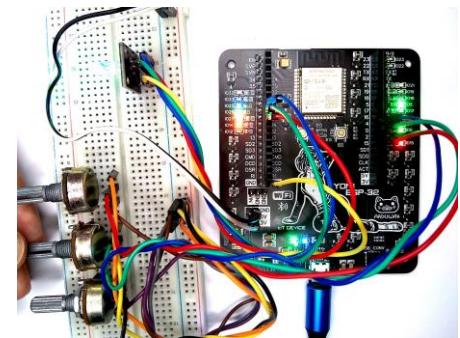
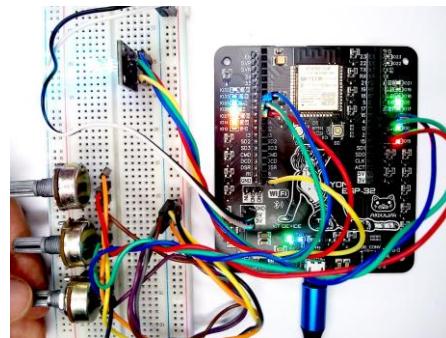
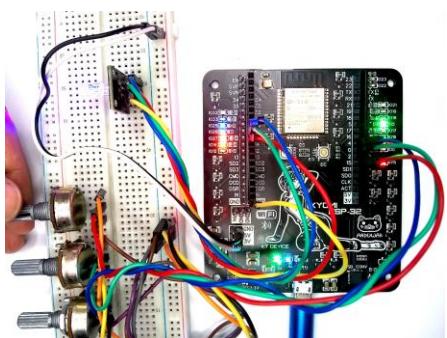
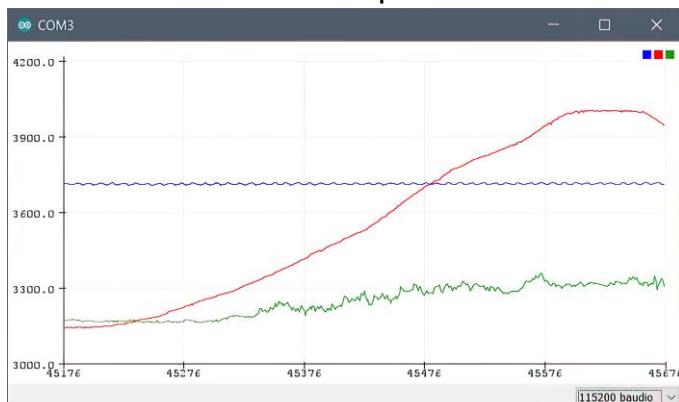


```
LedcWrite(0,VALOR_ANALOGICO_ROJO);
LedcWrite(1,VALOR_ANALOGICO_VERDE);
LedcWrite(2,VALOR_ANALOGICO_AZUL);

Serial.print(VALOR_ANALOGICO_AZUL);
Serial.print("\t");
Serial.print(VALOR_ANALOGICO_ROJO);
Serial.print("\t");
Serial.println(VALOR_ANALOGICO_VERDE);
}
```

**NOTA:** Recuerda que los GPIO's 0,5,14 y 15 son requeridos para arrancar la tarjeta y para cambiar al modo de Flasheo. En este ejercicio en particular, debes desconectar el GPIO0 del potenciómetro, tanto al subir el programa como al reiniciar la tarjeta, de lo contrario, no podrás subir el programa (aunque presiones el botón BOOT) y no arrancará cuando se reinicie la tarjeta. Vuelve a conectar el GPIO0 hasta que esté corriendo el programa.

Ahora prueba el programa. Podrás usar el graficador serial para ver la regulación de los valores PWM con los potenciómetros:





## Ejercicio 8

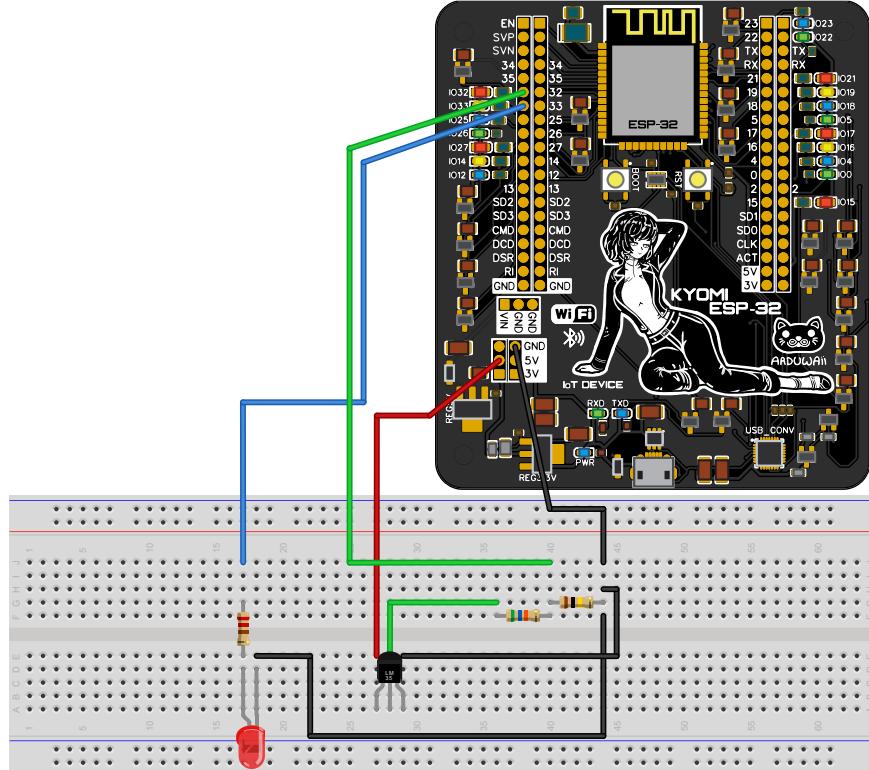
En este ejercicio usaremos el sensor de temperatura LM35 para activar un LED cuando registre 40°C o más.

El sensor LM35 puede ser alimentado con 4V hasta 30V, y su señal de temperatura (en teoría) puede llegar hasta 6V (alimentándolo con más de 6V). Su señal de temperatura tiene una equivalencia de 10mV (milivoltios) por grado Celsius, entonces 25°C equivalen a 250mV o 0.25V. El LM35 soporta hasta 150°C de temperatura. Alimentaremos el LM35 con 5V, por lo que necesitaremos un divisor de voltaje para la señal de temperatura (suponiendo que puede alcanzar 5V).

Materiales:

- 1 LED (opcional)
- 1 resistencia de 220Ω (opcional)
- 1 resistencia de 100kΩ
- 1 resistencia de 56kΩ
- 1 sensor LM35
- 7 jumpers o cables tipo Dupont (2 para el LED)
- plantilla de experimentos

El circuito es el siguiente:





Ahora veamos el código. Esta vez usaremos unas cuantas variables *float* para los cálculos de voltaje y temperatura:

```
const int GPIO_ANALOGICO=32, GPIO_LED=33;  
int contador_FOR;  
float VALOR_ANALOGICO=0, VOLTAJE_MEDIDO=0, VOLTAJE_REAL=0, TEMPERATURA=0;
```

En *setup()*, solo configuraremos los GPIO del LED y el LM35, además de la comunicación serial:

```
pinMode(GPIO_ANALOGICO, INPUT);  
pinMode(GPIO_LED, OUTPUT);  
Serial.begin(115200);
```

Dentro de *loop()*, el resultado de *TEMPERATURA* vendrá de la subrutina *CALCULAR\_TEMPERATURA()*. Un ciclo *while* primero evaluará si *TEMPERATURA* tiene un valor igual o mayor a 40°C. Si se cumple la condición, entonces se ejecutará el ciclo, encendiéndo el LED. Al final del ciclo hay que obtener nuevamente el valor de *TEMPERATURA*, para que *while* pueda ver los cambios y no se ejecute infinitamente evaluando siempre el mismo valor:

```
TEMPERATURA=CALCULAR_TEMPERATURA();  
while(TEMPERATURA>=40)  
{  
    digitalWrite(GPIO_LED, HIGH);  
    TEMPERATURA=CALCULAR_TEMPERATURA();  
}  
digitalWrite(GPIO_LED, LOW);
```

cuando el ciclo no se ejecute (si *TEMPERATURA* es menor a 40°C) el LED siempre se mantendrá en *LOW*.

La subrutina *CALCULAR\_TEMPERATURA()* la configuraremos para devolver un valor *float*. Primero realizamos un promedio de 100 lecturas analógicas. **IMPORTANTE:** mediremos voltajes muy pequeños que entran en la sección no lineal de la conversión del ADC, así que los compensaremos multiplicando las lecturas de *analogRead* por el factor 4095/1134 (dentro del promedio), para obtener las lecturas correctas:

```
float CALCULAR_TEMPERATURA()  
{  
    for(contador_FOR=0; contador_FOR<=100; contador_FOR++)  
    {  
        VALOR_ANALOGICO=VALOR_ANALOGICO+(analogRead(GPIO_ANALOGICO)*4095/1134);  
    }  
    VALOR_ANALOGICO=VALOR_ANALOGICO/100;
```



```
VOLTAJE_MEDIDO=(3.3/4096)*VALOR_ANALOGICO;  
VOLTAJE_REAL=(VOLTAJE_MEDIDO*(100000+56000))/100000;  
TEMPERATURA=VOLTAJE_REAL/0.01;  
  
Serial.print(TEMPERATURA);  
Serial.println("°C");  
  
return(TEMPERATURA);  
}
```

después convertiremos las lecturas analógicas a voltaje, luego usaremos el despeje de la ecuación del divisor de voltaje, para obtener el voltaje real que envía el LM35. La temperatura simplemente se calcula dividiendo el voltaje real entre el factor de 10mV por °C. Finalmente, enviamos el valor de *TEMPERATURA* al Monitor Serie y también lo regresamos a la variable que usamos en *Loop()*.

El código completo queda de la siguiente forma:

```
const int GPIO_ANALOGICO=32, GPIO_LED=33;  
int contador_FOR;  
float VALOR_ANALOGICO=0, VOLTAJE_MEDIDO=0, VOLTAJE_REAL=0, TEMPERATURA=0;  
  
void setup()  
{  
    pinMode(GPIO_ANALOGICO, INPUT);  
    pinMode(GPIO_LED, OUTPUT);  
    Serial.begin(115200);  
}  
  
Void Loop()  
{  
    TEMPERATURA=CALCULAR_TEMPERATURA();  
    while(TEMPERATURA>=40)  
    {  
        digitalWrite(GPIO_LED, HIGH);  
        TEMPERATURA=CALCULAR_TEMPERATURA();  
    }  
    digitalWrite(GPIO_LED, LOW);  
}
```



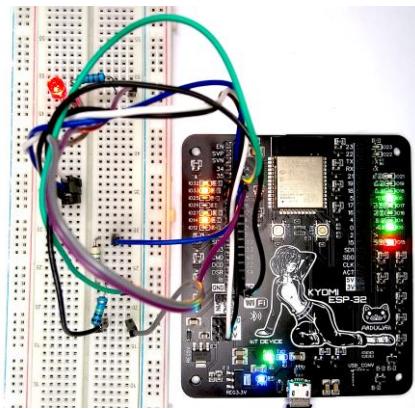
```
float CALCULAR_TEMPERATURA()
{
    for(contador_FOR=0; contador_FOR<=100; contador_FOR++)
    {
        VALOR_ANALOGICO=VALOR_ANALOGICO+(analogRead(GPIO_ANALOGICO)*4095/1134);
    }
    VALOR_ANALOGICO=VALOR_ANALOGICO/100;
    VOLTAJE_MEDIDO=(3.3/4096)*VALOR_ANALOGICO;
    VOLTAJE_REAL=(VOLTAJE_MEDIDO*(100000+56000))/100000;
    TEMPERATURA=VOLTAJE_REAL/0.01;

    Serial.print(TEMPERATURA);
    Serial.println("°C");

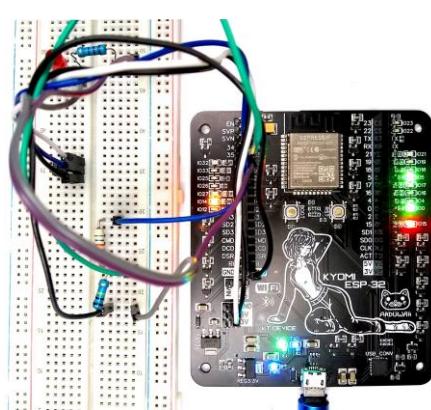
    return(TEMPERATURA);
}
```

Prueba el programa. Puedes aumentar la temperatura del LM35 hasta 40°C si lo presionas con los dedos:

```
42.09°C  
42.45°C  
42.28°C  
42.21°C  
42.36°C  
42.16°C  
41.92°C  
41.83°C  
41.67°C  
41.96°C  
41.72°C  
42.05°C  
41.90°C  
41.84°C  
42.32°C  
42.32°C
```



```
30.87°C  
30.82°C  
30.36°C  
30.52°C  
30.32°C  
30.32°C  
30.39°C  
30.45°C  
30.25°C  
30.55°C  
30.80°C  
31.09°C  
30.75°C  
30.03°C  
31.06°C  
30.76°C
```

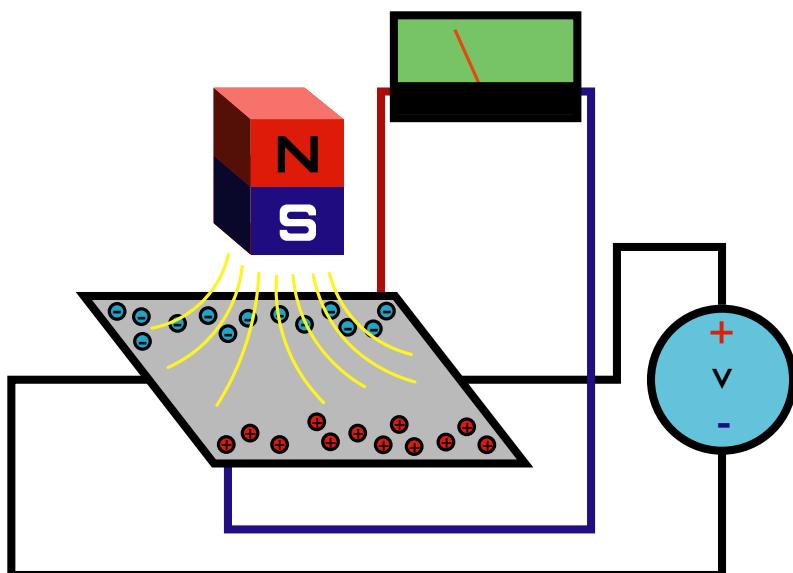




## Sensor de efecto hall integrado

El ESP32 cuenta con un sensor de efecto hall integrado, que se usa para detectar campos magnéticos cercanos.

El efecto hall funciona de la siguiente manera: al conectar una placa metálica a una fuente de voltaje, comenzará a circular corriente por la misma, donde las cargas negativas (electrones) irán del polo negativo al positivo y las cargas positivas (huecos de electrones) irán del positivo al negativo; al acercar un imán a la placa de manera perpendicular, las cargas positivas y negativas se alejarán hacia los extremos de la placa (pero seguirán circulando), esto provocará una diferencia de potencial eléctrico en sus extremos, que se le conoce como voltaje hall:



Cuanto más cercano o más fuerte sea el campo magnético, mayor será el voltaje hall, y si el campo magnético se invierte, el voltaje hall también se invertirá.

El sensor de efecto hall se encuentra por la zona de la cubierta metálica del módulo WROOM:





Para usar el sensor hall integrado, tenemos la siguiente instrucción:

```
hallRead()
```

que podemos usar en una variable o enviarla directamente al Monitor Serie.

Con el siguiente ejemplo, podrás ver las lecturas del sensor hall en el Monitor Serie, con un comportamiento similar a las lecturas analógicas:

```
void setup()
{
    Serial.begin(115200);
}

void loop()
{
    Serial.println(hallRead());
}
```

Verás que al acercar un imán, los valores incrementan. Con uno de los polos de imán obtendrás lecturas positivas y con el otro polo obtendrás lecturas negativas:

```
-53  
-52  
-57  
-59  
-53  
-56  
-59  
-52  
-53  
-55  
-53  
-54  
-54  
-54  
-59  
-54  
-54  
-59  
-54  
-5  
Autoscroll Sin ajuste de línea 115200 baudio Clear output
```



```
57  
61  
60  
61  
60  
58  
60  
64  
59  
58  
64  
61  
63  
64  
61  
59  
Autoscroll Sin ajuste de línea 115200 baudio Clear output
```





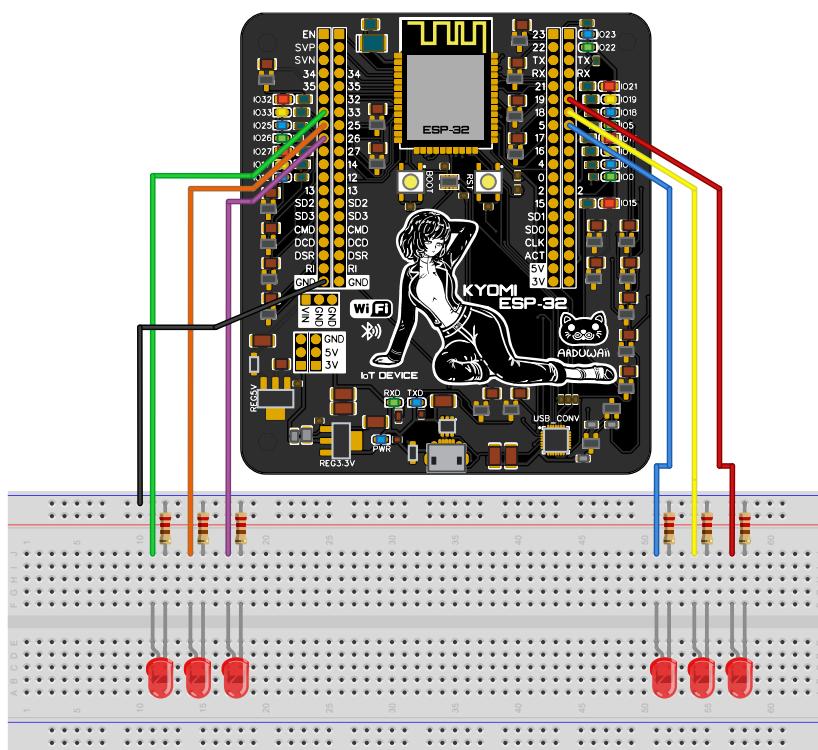
## Ejercicio 9

En este ejercicio usaremos el sensor hall para encender unos cuantos LEDs, que nos indicarán la cercanía del campo magnético de un imán y su polaridad.

Materiales:

- 6 LEDs (opcional)
- 6 resistencias de  $22\Omega$  (opcional)
- 7 jumpers o cables tipo Dupont (para los LEDs)
- 1 imán pequeño
- plantilla de experimentos

El circuito es el siguiente:



Veamos el código. Crearemos dos arrays con todos los GPIO's del lado izquierdo y derecho de la tarjeta. Las otras variables servirán para los valores de las lecturas hall, una variable auxiliar para un *for* y otra para el nivel de campo magnético:

```
const int GPIO_IZO[7]={32, 33, 25, 26, 27, 12, 14}, GPIO_DER[11]={23, 22, 21, 19, 18, 5, 17, 16, 4, 0, 15};  
int LECTURA_HALL=0, contador_FOR=0, NIVEL=0;
```



En `setup()` simplemente configuramos todos los GPIO's como `OUTPUT` con un par de ciclos `for` y también iniciamos la comunicación serial:

```
for(contador_FOR=0; contador_FOR<=6; contador_FOR++)
{
    pinMode(GPIOIZQ[contador_FOR],OUTPUT);
    digitalWrite(GPIOIZQ[contador_FOR],LOW);
}

for(contador_FOR=0; contador_FOR<=10; contador_FOR++)
{
    pinMode(GPIO_DER[contador_FOR],OUTPUT);
    digitalWrite(GPIOIZQ[contador_FOR],LOW);
}
Serial.begin(115200);
```

Dentro de `Loop()`, haremos un promedio de 100 lecturas hall para eliminar las variaciones rápidas del campo magnético detectado, el resultado también lo enviaremos al Monitor Serie:

```
for(contador_FOR=0;contador_FOR<=100;contador_FOR++)
{
    LECTURA_HALL=LECTURA_HALL+hallRead();
}
LECTURA_HALL=LECTURA_HALL/100;
Serial.println(LECTURA_HALL);
```

Con condicionales `if`, asignaremos un valor a `NIVEL` dependiendo del rango de valores en el que se encuentre el resultado de `LECTURA_HALL`. Observa que las condiciones lógicas se invierten para las lecturas hall negativas:

```
if(LECTURA_HALL>=20 && LECTURA_HALL<40){NIVEL=1;}
if(LECTURA_HALL>=40 && LECTURA_HALL<60){NIVEL=2;}
if(LECTURA_HALL>=60){NIVEL=3;}

if(LECTURA_HALL<=-20 && LECTURA_HALL>-40){NIVEL=-1;}
if(LECTURA_HALL<=-40 && LECTURA_HALL>-60){NIVEL=-2;}
if(LECTURA_HALL<=-60){NIVEL=-3;}
```

Usaremos el GPIO33, 25 y 26 para las lecturas positivas y el GPIO19, 18 y 5 para las negativas. Los LEDs se irán encendiendo según el nivel de campo magnético detectado. La `CONDICION` del siguiente ciclo `for` dependerá del valor de `NIVEL` y los GPIO's que cambien a `HIGH` también



dependerán de `contador_FOR`. Por ejemplo, si `NIVEL` vale 3, el ciclo de detendrá en 3, y los GPIO del array irán cambiando a `HIGH` conforme aumente el valor de `contador_FOR`:

```
for(contador_FOR=1;contador_FOR<=NIVEL;contador_FOR++)  
{  
    digitalWrite(GPIOIZQ[contador_FOR],HIGH);  
    delay(1);  
  
}  
  
for(contador_FOR=3;contador_FOR<=((NIVEL*-1)+2);contador_FOR++)  
{  
    digitalWrite(GPIO_DER[contador_FOR],HIGH);  
    delay(1);  
}
```

Para los GPIO's de valores hall negativos, `NIVEL` se multiplica por un -1 para volverlo positivo y se le suma un 2 para desplazar los GPIO's del array (para que correspondan con los que vamos a utilizar).

Finalmente, cuando se haya ejecutado alguno de los `for`, `NIVEL` lo reiniciamos a 0 para las nuevas lecturas hall y todos los GPIO's también los devolvemos a `LOW`:

```
NIVEL=0;  
for(contador_FOR=0;contador_FOR<=6;contador_FOR++)  
{  
    digitalWrite(GPIOIZQ[contador_FOR],LOW);  
}  
for(contador_FOR=0;contador_FOR<=11;contador_FOR++)  
{  
    digitalWrite(GPIO_DER[contador_FOR],LOW);  
}
```

El código completo queda de la siguiente forma:

```
const int GPIOIZQ[7]={32, 33, 25, 26, 27, 12, 14}, GPIO_DER[11]={23, 22, 21, 19,  
18, 5, 17, 16, 4, 0, 15};  
int LECTURA_HALL=0, contador_FOR=0, NIVEL=0;  
  
void setup()  
{  
    for(contador_FOR=0; contador_FOR<=6; contador_FOR++)
```

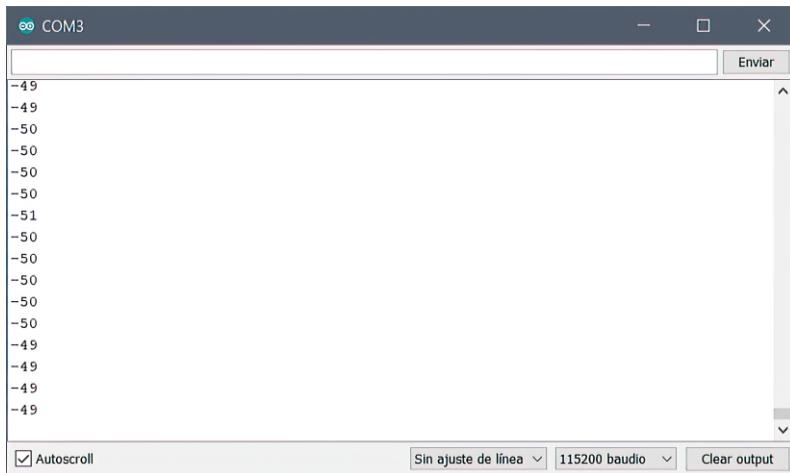


```
{  
    pinMode(GPIO_I2Q[contador_FOR],OUTPUT);  
    digitalWrite(GPIO_I2Q[contador_FOR],LOW);  
  
}  
  
  
for(contador_FOR=0; contador_FOR<=10; contador_FOR++)  
{  
    pinMode(GPIO_DER[contador_FOR],OUTPUT);  
    digitalWrite(GPIO_I2Q[contador_FOR],LOW);  
}  
Serial.begin(115200);  
}  
  
  
void Loop()  
{  
    for(contador_FOR=0;contador_FOR<=100;contador_FOR++)  
    {  
        LECTURA_HALL=LECTURA_HALL+hallRead();  
    }  
    LECTURA_HALL=LECTURA_HALL/100;  
    Serial.println(LECTURA_HALL);  
  
if(LECTURA_HALL>=20 && LECTURA_HALL<40){NIVEL=1;}  
if(LECTURA_HALL>=40 && LECTURA_HALL<60){NIVEL=2;}  
if(LECTURA_HALL>=60){NIVEL=3;}  
  
if(LECTURA_HALL<=-20 && LECTURA_HALL>-40){NIVEL=-1;}  
if(LECTURA_HALL<=-40 && LECTURA_HALL>-60){NIVEL=-2;}  
if(LECTURA_HALL<=-60){NIVEL=-3;}  
  
for(contador_FOR=1;contador_FOR<=NIVEL;contador_FOR++)  
{  
    digitalWrite(GPIO_I2Q[contador_FOR],HIGH);  
    delay(1);  
}  
for(contador_FOR=3;contador_FOR<=((NIVEL*-1)+2);contador_FOR++)
```



```
{  
    digitalWrite(GPIO_DER[contador_FOR], HIGH);  
    delay(1);  
}  
  
NIVEL=0;  
  
for(contador_FOR=0;contador_FOR<=6;contador_FOR++)  
{  
    digitalWrite(GPIO_IZQ[contador_FOR], LOW);  
}  
  
for(contador_FOR=0;contador_FOR<=11;contador_FOR++)  
{  
    digitalWrite(GPIO_DER[contador_FOR], LOW);  
}  
}
```

Ahora prueba el programa. Verás cómo se encienden los LEDs conforme acerques el imán o lo cambies de polo:



Arduino Serial Monitor window titled "COM3". The text area displays a continuous stream of the character "-49". At the bottom, there are checkboxes for "Autoscroll", "Sin ajuste de línea" (unselected), "115200 baudio" (selected), and "Clear output".



Arduino Serial Monitor window titled "COM3". The text area displays a continuous stream of the character "90". At the bottom, there are checkboxes for "Autoscroll", "Sin ajuste de línea" (unselected), "115200 baudio" (selected), and "Clear output".





# Interrupciones y temporizadores en el ESP32

Podemos implementar interrupciones y temporizadores en el ESP32 al igual que en un Arduino clásico, aunque con el ESP32 podremos aprovecharlas mejor para programas multitarea o para ejecutar múltiples funciones, gracias al procesador de doble núcleo de 240MHz.

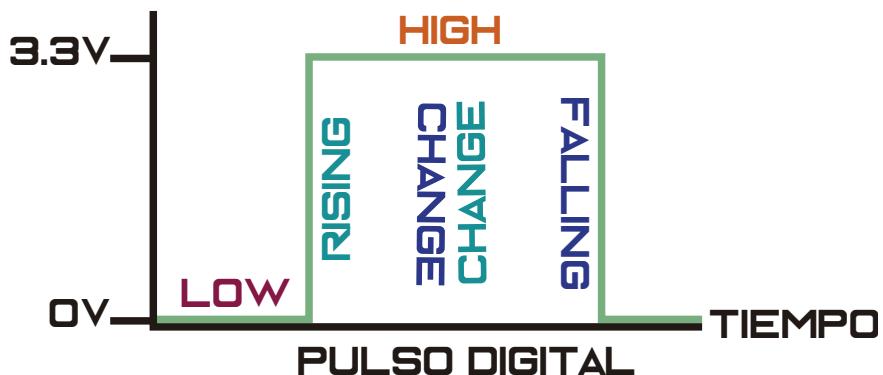
## Básicos: Interrupciones en Arduino

Las interrupciones son utilizadas para ejecutar una subrutina cuando ocurre algún cambio en la señal que recibe algún pin (una señal digital). Cuando la señal es detectada, la rutina principal interrumpe su flujo de ejecución, sin importar la instrucción que esté ejecutándose. Es entonces que se ejecuta la subrutina asociada a la interrupción. Cuando termina la subrutina, el programa vuelve a la última instrucción que se estaba ejecutando.

Para crear una interrupción, se usa la siguiente instrucción:

```
attachInterrupt(digitalPinToInterrupt(PIN), SUBRUTINA, MODO);
```

donde *PIN* es el número de GPIO que recibirá la señal digital de interrupción, *SUBRUTINA* es el nombre de la subrutina que asociaremos a la interrupción y *MODO* es el tipo de interrupción que se detectará de la señal del GPIO. El parámetro *MODO* soporta los siguientes tipos de interrupción:



- **LOW:** la interrupción ocurre siempre que se detecte una señal digital *LOW*.
- **HIGH:** la interrupción ocurre siempre que se detecte una señal digital *HIGH*.
- **RISING:** la interrupción ocurre sólo cuando se detecta el cambio de *LOW* a *HIGH* de la señal digital.
- **FALLING:** la interrupción ocurre sólo cuando se detecta el cambio de *HIGH* a *LOW* de la señal digital.
- **CHANGE:** la interrupción ocurre cuando se detecta el cambio de *LOW* a *HIGH* o *HIGH* a *LOW* de la señal digital.



La interrupción que configuremos en `attachInterrupt()` debe ir en `setup()`, para que el programa monitoree la señal de interrupción desde el inicio.

## Básicos: Temporizadores

La instrucción `delay()` ya la hemos usado para hacer pausas entre instrucciones. Esta es una instrucción de ‘bloqueo’ y mantiene a todo el programa detenido hasta que termine el tiempo especificado. La instrucción `delayMicroseconds()` funciona de la misma manera, sólo que el tiempo especificado se toma en microsegundos.

Las instrucciones de bloqueo no son muy útiles cuando debemos trabajar con múltiples tareas dentro del mismo programa, ya que pausar toda la ejecución provoca retrasos de tiempo, que pueden afectar a funciones más críticas.

Podemos construir instrucciones dependientes del tiempo sin recurrir a la función `delay()`, para no bloquear todo el programa. Esto lo haremos con temporizadores.

Los temporizadores cuentan el tiempo transcurrido desde que arrancó un programa. Este tiempo se puede usar para calcular la distribución de otros tiempos de ejecución que requieran las instrucciones del programa. Puedes usar alguno de los siguientes temporizadores:

```
millis()  
o  
micros()
```

donde `millis()` entrega el tiempo transcurrido desde el arranque en milisegundos y `micros()` lo entrega en microsegundos.

Puedes guardar el valor de `millis()` en una variable numérica, pero debe ser de tipo `unsigned long`, ya que, al pasar el tiempo, la variable tendrá un valor muy grande, y si usas otro tipo de variable que soporte números más pequeños (`int` por ejemplo) desbordarás a la variable muy rápido.

Si usas `unsigned long` como tipo de variable para `millis()`, esta se desbordará en aproximadamente 50 días, momento en el que se reiniciará a 0. En el caso de `micros()` ocurre algo similar, pero la variable se desbordará en 70 minutos, también reiniciándose a 0. Cualquier otra variable que utilice los valores de tiempo dependientes de `millis()` también debe ser de tipo `unsigned long` o superior.

Veamos un ejemplo para hacer parpadear un LED, aplicando el temporizador `millis()`. Usaremos una variable que irá guardando ciertos intervalos de tiempo (comenzará en 0) y otra para el estado del LED (de tipo `bool`), en `GPIO_LED` coloca cualquier pin:

```
unsigned Long TEMPORIZADOR=0; bool ESTADO_LED=false; int GPIO_LED=PIN;
```



En `setup()` sólo configuramos el LED y le indicamos que comience en `LOW` o `false`:

```
pinMode(GPIO_LED,OUTPUT);
digitalWrite(GPIO_LED,ESTADO_LED);
```

Dentro de `Loop()` construimos el siguiente temporizador de ejecución:

```
if((millis()-TEMPORIZADOR)>=1000)
{
    ESTADO_LED=!ESTADO_LED;
    digitalWrite(GPIO_LED,ESTADO_LED);
    TEMPORIZADOR= millis();
}
```

donde en `if` establecemos que, si la diferencia entre el tiempo que lleva contando `millis()` y el tiempo guardado en `TEMPORIZADOR` es de 1000ms, entonces se ejecutarán las instrucciones del `if`. Por ejemplo, al inicio `millis()` vale 0ms y `TEMPORIZADOR` también vale 0ms, entonces no se cumple la condición y el `if` no se ejecuta; cuando pase el tiempo y `millis()` alcance los 1000ms, entonces la diferencia entre `millis()` y `TEMPORIZADOR` será de 1000ms, momento en el que se cumple la condición y el `if` se ejecuta, cambiando `ESTADO_LED` a su valor negado, es decir, de `false` a `true` (`LOW` a `HIGH`), encendiéndo el LED. También `TEMPORIZADOR` tomará el valor actual de `millis()` (1000ms).

Seguirá transcurriendo el tiempo, pero ahora `TEMPORIZADOR` vale 1000ms y la condición del `if` no se cumplirá hasta que `millis()` alcance al menos 2000ms, momento en el que la diferencia entre `millis()` y `TEMPORIZADOR` será de 1000ms nuevamente, negando a `ESTADO_LED` (ahora de `true` a `false`) y actualizando el valor de `TEMPORIZADOR` al valor actual de `millis()` (2000ms). Así continuará cambiando el valor de `ESTADO_LED` cada vez que se cumpla la condición del `if`, haciendo parpadear al LED cada 1000ms.

Con el ejemplo anterior vemos algo importante: no es necesario detener la ejecución de todo el programa con `delay()` para encender y apagar el LED con pausas. Además, cuando no se cumple la condición del `if`, el programa puede continuar su ejecución para hacer otras tareas, ahorrando todo el tiempo que perderíamos con `delay()`.



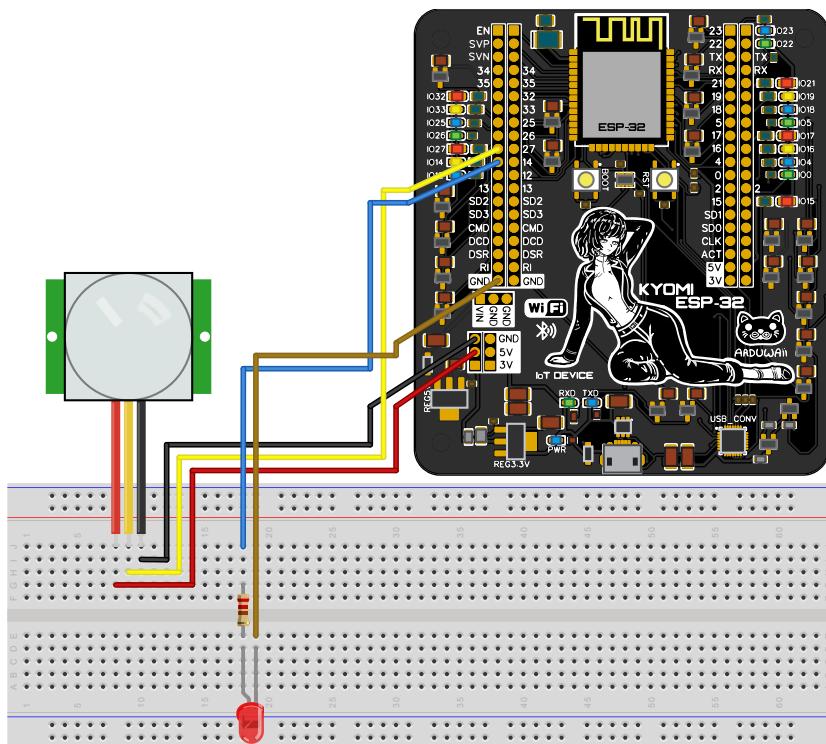
## Ejercicio 10

En este ejercicio usaremos un sensor PIR para detectar movimiento, aplicando interrupciones y temporizadores. El sensor PIR detecta el movimiento de los objetos que emiten luz infrarroja (básicamente los que emiten calor, como el cuerpo humano). Cuando se detecta movimiento, genera un pulso digital que dura unos cuantos segundos.

Materiales:

- 1 LED (opcional)
- 1 resistencia de 220Ω (opcional)
- 5 jumpers o cables tipo Dupont (2 para el LED)
- 1 sensor PIR modelo MQ3, AM312 o similar
- plantilla de experimentos

El circuito es el siguiente:



Algunos sensores PIR tienen el pin de voltaje y GND invertidos, revisalo antes de armar el circuito.

Veamos el código. Tenemos las variables básicas de los GPIO's para la señal del sensor PIR y el LED:

```
const int GPIO_LED = 14, SENSOR_PIR = 27;  
int TIEMPO_LED_SEGUNDOS=10;
```



```
unsigned Long TIEMPO_ACTUAL = millis(), ULTIMA_ACTIVACION = 0;  
bool ARRANCAR_TEMPORIZADOR = false;
```

la variable `TIEMPO_LED_SEGUNDOS` definirá los segundos que permanecerá encendido el LED una vez que se detecte movimiento, también usaremos un par de variables `unsigned Long` para el temporizador y una variable `bool` para guardar el estado que determina si el temporizador está activo.

En `setup()` configuraremos los GPIO's del LED y del sensor PIR como `INPUT_PULLUP` y `OUTPUT`, también iniciaremos el LED en `LOW` y configuraremos la comunicación serial:

```
pinMode(SENSOR_PIR, INPUT_PULLUP);  
pinMode(GPIO_LED, OUTPUT);  
digitalWrite(GPIO_LED, LOW);  
Serial.begin(115200);  
attachInterrupt(digitalPinToInterrupt(SENSOR_PIR), DETECTA_MOVIMIENTO, RISING);
```

creamos una interrupción que se active con la señal del sensor PIR, que tendrá asociada la subrutina `DETECTA_MOVIMIENTO` y la interrupción será de tipo `RISING`.

Cuando se activa la subrutina `DETECTA_MOVIMIENTO`, se envía un mensaje al Monitor Serie y se activa el LED. La variable `ARRANCAR_TEMPORIZADOR` cambia a `true` y `ULTIMA_ACTIVACION` toma el valor actual de `millis()`, que sería el tiempo en el que ocurrió la interrupción:

```
void DETECTA_MOVIMIENTO()  
{  
    Serial.println("Hay movimiento!");  
    digitalWrite(GPIO_LED, HIGH);  
    ARRANCAR_TEMPORIZADOR = true;  
    ULTIMA_ACTIVACION = millis();  
}
```

En `Loop()`, `TIEMPO_ACTUAL` simplemente es el valor actual de `millis()`. La condición `if` se activa sólo cuando se cumplen las siguientes condiciones: `ARRANCAR_TEMPORIZADOR` es `true` (se activó la interrupción) y que la diferencia de tiempo entre `TIEMPO_ACTUAL` y `ULTIMA_ACTIVACION` sea mayor a `TIEMPO_LED_SEGUNDOS*1000` (10 segundos o 10000ms):

```
TIEMPO_ACTUAL = millis();  
if(ARRANCAR_TEMPORIZADOR==true && (TIEMPO_ACTUAL-ULTIMA_ACTIVACION>(TIEMPO_LED_SEGUNDOS*1000))  
{  
    Serial.println("No hay movimiento...");
```



```
    digitalWrite(GPIO_LED, LOW);
    ARRANCAR_TEMPORIZADOR = false;
}
```

si se cumplen las condiciones, en el `if` se enviará un mensaje al Monitor Serie y el LED se apagará, además de cambiar `ARRANCAR_TEMPORIZADOR` a `false` para indicar que el temporizador ya no está activo. Básicamente hemos construido un temporizador similar al del ejemplo anterior.

El código completo queda de la siguiente forma:

```
const int GPIO_LED = 14, SENSOR_PIR = 27;
int TIEMPO_LED_SEGUNDOS=10;
unsigned Long TIEMPO_ACTUAL = millis(), ULTIMA_ACTIVACION = 0;
bool ARRANCAR_TEMPORIZADOR = false;

void setup()
{
    pinMode(SENSOR_PIR, INPUT_PULLUP);
    pinMode(GPIO_LED, OUTPUT);
    digitalWrite(GPIO_LED, LOW);
    Serial.begin(115200);
    attachInterrupt(digitalPinToInterrupt(SENSOR_PIR),DETECTA_MOVIMIENTO,RISING);
}

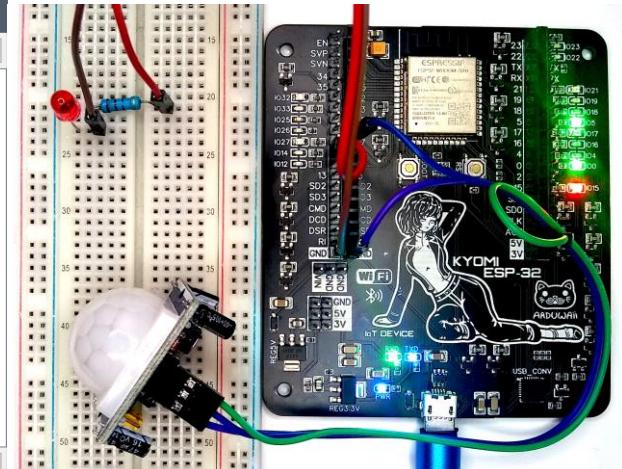
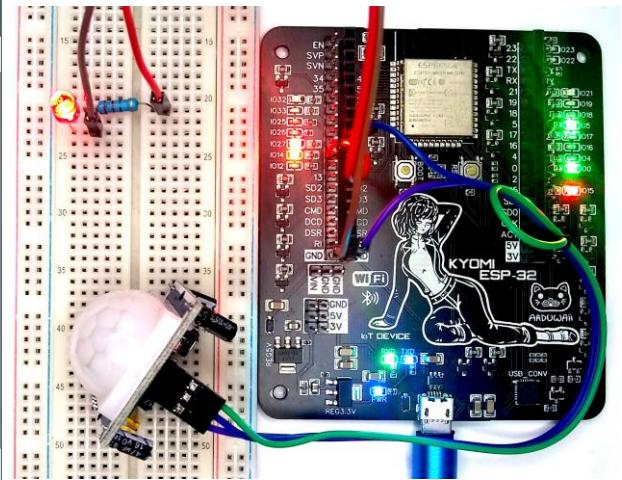
void loop()
{
    if(ARRANCAR_TEMPORIZADOR==true && (TIEMPO_ACTUAL-ULTIMA_ACTIVACION>(TIEMPO_LED_SEGUNDOS*1000)))
    {
        Serial.println("No hay movimiento...");
        digitalWrite(GPIO_LED, LOW);
        ARRANCAR_TEMPORIZADOR = false;
    }
}

void DETECTA_MOVIMIENTO()
{
    Serial.println("Hay movimiento!");
    digitalWrite(GPIO_LED, HIGH);
```



```
ARRANCAR_TEMPORIZADOR = true;  
ULTIMA_ACTIVACION = millis();  
}
```

Prueba el programa. Para ajustar la detección de movimiento y el tiempo que permanece activa la señal del sensor PIR, puedes mover el par de potenciómetros de la parte posterior, la función de cada potenciómetro depende del modelo de sensor que uses:



No es muy recomendable usar instrucciones lentas para las subrutinas de las interrupciones (`Serial.print` por ejemplo) y tampoco es muy conveniente que devuelvan valores a otras rutinas. Estas subrutinas deben ser rápidas para interrumpir al programa por poco tiempo.



## Guardar datos en la memoria Flash

Podemos guardar datos en la memoria Flash integrada de dos maneras: usando las instrucciones de la librería `Preferences.h` o con el sistema de archivos SPIFFS.

El método de `Preferences.h` es recomendable para guardar valores de variables, tiempos, configuraciones, estados de activación de un pin o cualquier otro dato pequeño. El método con SPIFFS es recomendable para guardar archivos grandes, como archivos de texto o páginas web. El sistema SPIFFS funciona de manera similar al sistema de archivos de una computadora.

Aquí solo detallaremos el método con `Preferences.h`, el sistema SPIFFS lo veremos en otro capítulo.

Los datos que guardemos en la Flash permanecerán intactos aún después de presionar el botón RESET e incluso cuando la tarjeta no reciba energía.

Para guardar datos, debemos usar la siguiente estructura de instrucciones:

1. Primero debemos incluir la librería `Preferences.h` y luego crear un objeto dependiente de `Preferences.h` (`MEMORIA1`). Este objeto lo consideraremos como una ‘memoria’:

```
#include <Preferences.h>
Preferences MEMORIA1;
int VALOR1, VALOR2, VALOR3;
```

2. En `setup()` configuramos a `MEMORIA1` de la siguiente manera:

```
MEMORIA1.begin("ESPACIO1", false);
MEMORIA1.putTIPO_DE_VARIABLE("CELDA1", VALOR1);
MEMORIA1.putTIPO_DE_VARIABLE ("CELDA2", VALOR2);
MEMORIA1.end();
```

donde `MEMORIA1.begin` configura un espacio de memoria llamado `ESPACIO1` y el parámetro `false` indica que este espacio de memoria permitirá la lectura y escritura de datos, pero si lo cambias a `true`, sólo permitirá la lectura de datos. El espacio de memoria se divide en celdas, que van a guardar el valor de una variable que le asignemos: con `MEMORIA1.put` configuramos la celda para asignarle una variable, entonces `CELDA1` tendrá el valor de la variable `VALOR1`. En `TIPO_DE_VARIABLE` colocamos el tipo de variable que le asignamos a la celda (en este caso `Int`). Puedes usar alguno de los siguientes tipos de variables:

Tipo de variable	Parámetro	Tipo de variable	Parámetro
<code>char</code>	<code>Char</code>	<code>Long64</code>	<code>Long64</code>
<code>unsigned char</code>	<code>UChar</code>	<code>unsigned Long64</code>	<code>ULong64</code>



<i>short</i>	<i>Short</i>	<i>float</i>	<i>Float</i>
<i>unsigned short</i>	<i>UShort</i>	<i>double</i>	<i>Double</i>
<i>int</i>	<i>Int</i>	<i>bool</i>	<i>Bool</i>
<i>unsigned int</i>	<i>UInt</i>	<i>String</i>	<i>String</i>
<i>long</i>	<i>Long</i>	<i>bytes</i>	<i>Bytes</i>
<i>unsigned long</i>	<i>ULong</i>		

Puedes dividir el espacio de memoria en varias celdas, en este caso solo tenemos a *CELDA1* y *CELDA2*, ambas con una variable asignada. Finalmente usamos *MEMORIA1.end()* para cerrar la configuración de *MEMORIA1*.

3. Para guardar datos en las celdas de un espacio de memoria, ya sea dentro de *Loop()* o cualquier otra subrutina, tenemos que abrir y cerrar los espacios de memoria de la siguiente manera:

```
VALOR1= OTROVALOR;  
MEMORIA1.begin("ESPACIO1", false);  
MEMORIA1.putTIPO_DE_VARIABLE("CELDA1", VALOR1);  
MEMORIA1.end();
```

Si *VALOR1* ya cambió, su nuevo valor lo podremos guardar abriendo *ESPACIO1* con la instrucción *MEMORIA1.begin*. Con *MEMORIA1.put* guardamos el valor actual de *VALOR1* (en este caso *OTROVALOR*) en *CELDA1*. Nuevamente cerramos el espacio de memoria con *MEMORIA1.end()*.

4. Para recuperar valores de la memoria, nuevamente hay que abrir el espacio de memoria requerido:

```
MEMORIA1.begin("ESPACIO1", false);  
VALOR_RECUPERADO= MEMORIA1.getTIPO_DE_VARIABLE("CELDA1",0);  
MEMORIA1.end();
```

La instrucción *MEMORIA1.get* nos permite recuperar el valor de *CELDA1*. En *TIPO\_DE\_VARIABLE* nuevamente colocamos el tipo de variable que guarda *CELDA1*. En *("CELDA1",0)*, 0 es el valor por defecto que recuperaremos de *CELDA1* en caso de que no haya valores previamente guardados. Finalmente cerramos el espacio de memoria con *MEMORIA1.end()*.

NOTA: Para recuperar valores desde el arranque (si quieres conservarlos después de un reinicio), tienes que hacerlo en *setup()* antes de configurar las celdas del espacio de memoria, por ejemplo:

```
MEMORIA1.begin("ESPACIO1", false);  
VALOR_RECUPERADO= MEMORIA1.getTIPO_DE_VARIABLE("CELDA1",0);  
MEMORIA1.end();  
MEMORIA1.begin("ESPACIO1", false);  
MEMORIA1.putTIPO_DE_VARIABLE("CELDA1", VALOR1);  
MEMORIA1.end();
```



Los nombres que asignemos a las celdas pueden repetirse, siempre y cuando sean asignados en espacios de memoria distintos:

```
MEMORIA1.begin("ESPACIO1", false);
MEMORIA1.putTIPO_DE_VARIABLE("CELDA1", VALOR1);
MEMORIA1.end();
MEMORIA1.begin("ESPACIO2", false);
MEMORIA1.putTIPO_DE_VARIABLE("CELDA1", VALOR2);
MEMORIA1.end();
```

**NOTA:** El nombre que asignemos a las celdas no debe exceder los 15 caracteres.

Hagamos un ejercicio práctico para guardar datos en la memoria Flash.

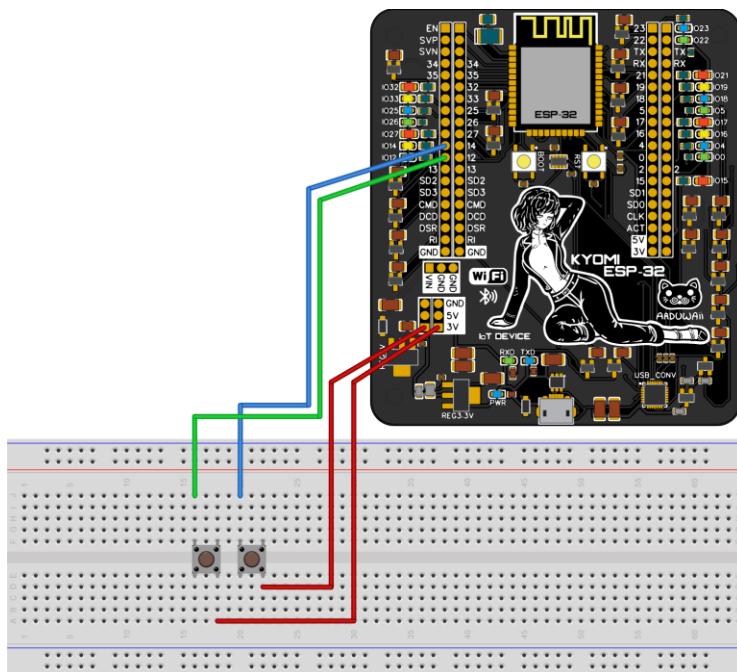
## Ejercicio 11

En este ejercicio guardaremos el valor de una variable en la Flash, el cual podremos modificar sumando o restando un 1 con un par de botones. Después de reiniciar la tarjeta, recuperaremos el último valor que guardamos.

Materiales:

- 2 botones pulsadores
- 4 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:





Veamos el código. Incluiremos la librería `Preferences.h` y crearemos el objeto `MEMORIA`, además de declarar los pines GPIO's y unas variables para manejar los datos:

```
#include <Preferences.h>
Preferences MEMORIA;
const int GPIO_SUMA=14,GPIO_RESTA=12;
int VALOR_SUMA_RESTA=0, VALOR_GUARDADO;
```

En `setup()` primero recuperaremos cualquier valor que esté guardado en la celda `VALOR1` de `ESPACIO1`. Luego vamos a configurar estos espacios de memoria, además de configurar los GPIO's y la comunicación serial:

```
MEMORIA.begin("ESPACIO1", false);
VALOR_GUARDADO=MEMORIA.getInt("VALOR1",0);
MEMORIA.end();

MEMORIA.begin("ESPACIO1", false);
MEMORIA.putInt("VALOR1", VALOR_SUMA_RESTA);
MEMORIA.end();

pinMode(GPIO_SUMA, INPUT_PULLDOWN);
pinMode(GPIO_RESTA, INPUT_PULLDOWN);
Serial.begin(115200);
```

La primera vez que funcione el programa, la recuperación de valores de la Flash se ignorará, ya que aún no estarán configurados los espacios de memoria. Cuando reinicies la tarjeta o le quites la alimentación, los espacios de memoria ya estarán configurados y con datos guardados. Al arrancar nuevamente el programa, se podrán recuperar los valores guardados, por que las celdas con datos ya se encontrarán en la memoria Flash. La recuperación se hace antes debido a que se van a configurar los espacios de memoria en el arranque y se van a reiniciar, así que hay que recuperar todos los datos de interés antes de que esto ocurra.

En `Loop()`, cuando el botón de 'suma' sea presionado, a `VALOR_GUARDADO` se le va a sumar un 1 y se ejecutará la subrutina `GUARDADO()`, para guardar este cambio en la memoria (también se enviará un mensaje al Monitor Serie):

```
if(digitalRead(GPIO_SUMA)==HIGH)
{
    VALOR_GUARDADO++;
    Serial.print("SUMA+1=");
    GUARDADO();
```



```
}

if(digitalRead(GPIO_RESTA)==HIGH)
{
    VALOR_GUARDADO--;
    Serial.print("RESTA-1=");
    GUARDADO();
}
delay(200);
```

si se presiona el botón de ‘resta’, a `VALOR_GUARDADO` se le va a restar un 1 y se ejecutará la subrutina `GUARDADO()` (igualmente se enviará un mensaje al Monitor Serie). En la subrutina `GUARDADO()`, `VALOR_SUMA_RESTA` toma el valor de `VALOR_GUARDADO` y luego se guarda en memoria, además de mostrarlo en el Monitor Serie:

```
void GUARDADO()
{
    VALOR_SUMA_RESTA=VALOR_GUARDADO;
    MEMORIA.begin("ESPACIO1", false);
    MEMORIA.putInt("VALOR1", VALOR_SUMA_RESTA);
    MEMORIA.end();
    Serial.println(VALOR_SUMA_RESTA);
}
```

Los cambios de `VALOR_GUARDADO` se reflejarán en `VALOR_SUMA_RESTA` y se podrá recuperar el último cambio de `VALOR_GUARDADO` cuando arranque nuevamente el programa.

El código completo queda de la siguiente forma:

```
#include <Preferences.h>
Preferences MEMORIA;
const int GPIO_SUMA=14,GPIO_RESTA=12;
int VALOR_SUMA_RESTA=0, VALOR_GUARDADO;

void setup()
{
    MEMORIA.begin("ESPACIO1", false);
    VALOR_GUARDADO=MEMORIA.getInt("VALOR1",0);
    MEMORIA.end();

    MEMORIA.begin("ESPACIO1", false);
```



```
MEMORIA.putInt("VALOR1", VALOR_SUMA_RESTA);
MEMORIA.end();

pinMode(GPIO_SUMA, INPUT_PULLDOWN);
pinMode(GPIO_RESTA, INPUT_PULLDOWN);
Serial.begin(115200);
}

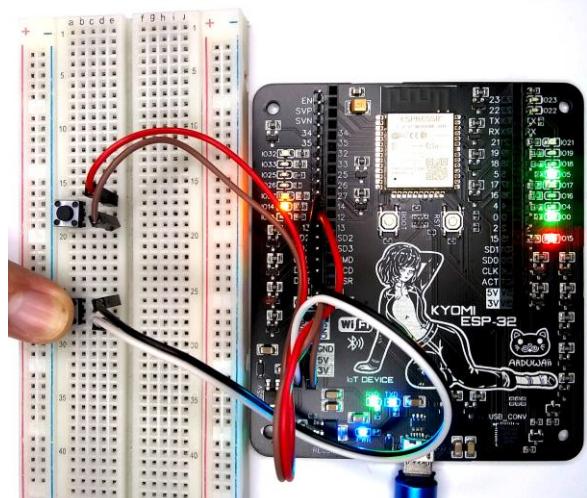
void Loop()
{
    if(digitalRead(GPIO_SUMA)==HIGH)
    {
        VALOR_GUARDADO++;
        Serial.print("SUMA+1=");
        GUARDADO();
    }
    if(digitalRead(GPIO_RESTA)==HIGH)
    {
        VALOR_GUARDADO--;
        Serial.print("RESTA-1=");
        GUARDADO();
    }
    delay(200);
}

void GUARDADO()
{
    VALOR_SUMA_RESTA=VALOR_GUARDADO;
    MEMORIA.begin("ESPACIO1", false);
    MEMORIA.putInt("VALOR1", VALOR_SUMA_RESTA);
    MEMORIA.end();
    Serial.println(VALOR_SUMA_RESTA);
}
```



## Capítulo 2: Primeros pasos con los GPIO's

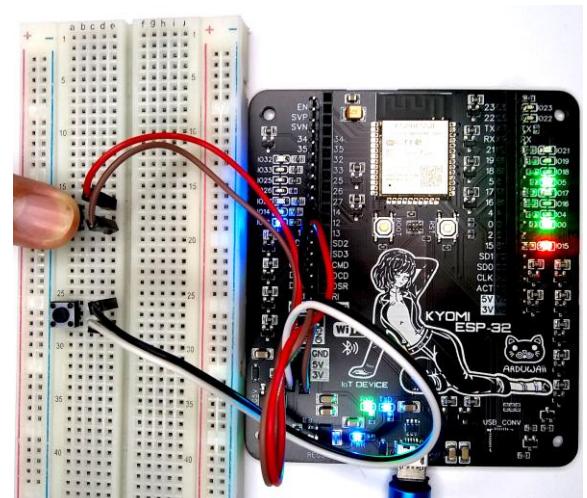
Prueba el código. Verás que el resultado de la última suma o resta se conservará después de que reinicies la tarjeta:



COM3

```
SUMA+1=0
SUMA+1=1
SUMA+1=2
SUMA+1=3
SUMA+1=4
SUMA+1=5
```

Autoscroll    Sin ajuste de línea    115200 baudio    Clear output



COM3

```
SUMA+1=0
SUMA+1=1
SUMA+1=2
SUMA+1=3
SUMA+1=4
SUMA+1=5
RESTA-1=4
RESTA-1=3
RESTA-1=2
RESTA-1=1
RESTA-1=0
RESTA-1=-1
RESTA-1=-2
RESTA-1=-3
RESTA-1=-4
```

Autoscroll    Sin ajuste de línea    115200 baudio    Clear output



COM3

```
RESTA-1=-2
RESTA-1=-3
RESTA-1=-4
ets Jun 8 2016 00:22:57

rst:0x1 (POWERON_RESET) boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
RESTA-1=-5
RESTA-1=-6
```

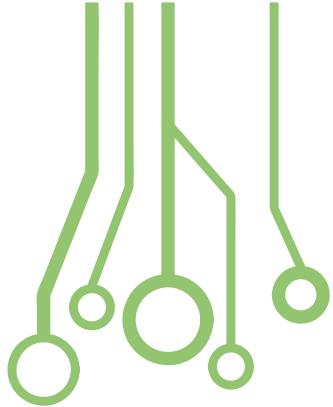
Autoscroll    Sin ajuste de línea    115200 baudio    Clear output



Para borrar por completo todos los espacios de memoria que hayan quedado en la Flash, sube el siguiente programa a tu tarjeta para formatearla:

```
#include <nvs_flash.h>
void setup()
{
    nvs_flash_erase();
    nvs_flash_init();
    while(true);
}
void loop()
{}
```

Los espacios de memoria en la Flash pueden permanecer guardados aún después de subir un programa distinto, así que es importante que formatee la Flash si vas subir otro programa que dependa de datos guardados en memoria.



## Capítulo 3: Modo DeepSleep y sus funciones





## Los modos de consumo energético del ESP32

El ESP32 puede trabajar en distintos modos de consumo energético para aumentar su eficiencia, desactivando funciones que no sean necesarias o incluso apagando los núcleos del procesador principal.

Los modos de consumo energético son:

- **Activo:** este es el modo energético normal. El procesador principal, los radios Wifi y Bluetooth y el bloque RTC (con su procesador ULP y la unidad PMU) están activos. En este modo, el módulo ESP32 puede tener picos de consumo de hasta 1.5A (amperios).
- **ModemSleep:** en este modo, los radios Wifi y Bluetooth están desactivados, pero el resto de componentes están activos. De esta manera se pueden reducir hasta 200mA del consumo normal del ESP32.
- **LightSleep:** en este modo, el procesador principal, parte de la memoria RAM y los periféricos digitales funcionan de manera intermitente, mientras que el bloque RTC con el procesador ULP y la unidad PMU están activos. Con este modo, el ESP32 puede consumir menos de 10mA.
- **DeepSleep:** este modo sólo mantiene activo el bloque RTC con el ULP y la unidad PMU. El consumo del ESP32 va de los 10 $\mu$ A hasta los 0.15mA en este modo.
- **Hibernación:** es el modo de menor consumo. Sólo se mantiene activa la unidad PMU del bloque RTC, para monitorear algunos GPIO's conectados al propio RTC. En este caso, el ESP32 puede reducir su consumo hasta los 2.5 $\mu$ A.

Los modos de bajo consumo energético son muy importantes para mantener funcionando al ESP32 cuando opera con baterías o alguna otra fuente limitada de energía.



La tarjeta KYOMI puede funcionar con distintos tipos de baterías y fuentes de energía.



Debes tener en cuenta que el consumo energético real de los modos LightSleep, DeepSleep e Hibernación son considerablemente más altos, ya que hay que sumar el consumo de los otros chips y componentes de la tarjeta, además de su diseño de hardware. En la siguiente tabla se muestran los consumos de corriente de la tarjeta KYOMI, en los distintos modos energéticos:

Modo	Consumo de corriente aprox. (alimentando la tarjeta con 5V por el puerto micro-USB)
ModemSleep	88mA
LightSleep	62mA
DeepSleep	46mA
Hibernación	46mA

El modo de bajo consumo energético más utilizado es DeepSleep, ya que no existe mucha diferencia de consumo entre ModemSleep y LightSleep, comparados con el modo Activo, mientras que el modo Hibernación no se puede aprovechar mucho, debido al consumo energético real de cada tarjeta y a su poca practicidad. Por ahora nos centraremos en el modo DeepSleep.

Estos modos de consumo energético se pueden activar y desactivar de varias maneras. Las tres formas más comunes de desactivar el modo DeepSleep son: con GPIO's touch, con una señal externa en los GPIO's conectados al RTC (revísalos en el pinout) y mediante temporizadores.

## DeepSleep con temporizadores

Podemos activar el modo DeepSleep desde cualquier parte del código utilizando la siguiente instrucción:

```
esp_deep_sleep_start();
```

Lo importante al activar el modo DeepSleep, es guardar los datos que vayamos a necesitar y tener un método para despertar o reactivar al ESP32.

El temporizador del bloque RTC nos permite despertar al ESP32 después de un tiempo que especifiquemos, funciona de manera similar a `millis()`. La siguiente instrucción la usaremos para despertar al ESP32:

```
esp_sleep_enable_timer_wakeup(TIEMPO)
```

donde `TIEMPO` va especificado en microsegundos. Cuando se active el modo DeepSleep, el temporizador del RTC empezará a contar, y cuando alcance



el valor especificado en `TIEMPO`, se despertará el ESP32. El temporizador `esp_sleep_enable_timer_wakeup()` debe estar configurado previamente antes de activar el modo DeepSleep, generalmente va dentro de `setup()`.

Cuando se despierta el ESP32, el programa arranca desde el principio (igual que cuando se reinicia la tarjeta), así que las variables se reinician y cualquier dato no guardado se pierde.

Podemos guardar los datos en la Flash o bien podemos usar la memoria SRAM del bloque RTC. Para guardar valores de variables en el RTC, usaremos el siguiente atributo al declarar la variable que queramos conservar durante el DeepSleep:

```
RTC_DATA_ATTR int VARIABLE;
```

también podemos agregar más variables antes de cerrar la instrucción:

```
RTC_DATA_ATTR int VARIABLE1, VARIABLE2, VARIABLE3;
```

**CONSEJO:** recuerda que el bloque RTC sólo tiene 8KB de SRAM, así que sólo podrás guardar unas cuantas variables con valores no muy grandes. Mejor reserva la memoria del RTC para valores importantes y usa la Flash para los datos más grandes.

No es muy recomendable guardar variables grandes como `unsigned Long64` o `double` en el RTC, ya que podrías desbordar la memoria.

**IMPORTANTE:** los datos guardados en el RTC permanecerán intactos al recuperarlos después de un DeepSleep, pero no al reiniciar la tarjeta o al apagarla. Para estos casos usa la Flash.

La ventaja de guardar datos en el RTC, es que su recuperación es inmediata, por lo que no es necesario abrir espacios de memoria para recuperarlos, como sí pasa con los datos en Flash. Esto quiere decir que las variables guardadas en el RTC conservan el último valor que tenían antes de activar el DeepSleep, y al despertar al ESP32, las variables asociadas al RTC recuperarán sus valores automáticamente y podremos usarlos de nuevo en el programa.

Ahora hagamos un ejercicio para activar el DeepSleep y usar un temporizador como despertador.



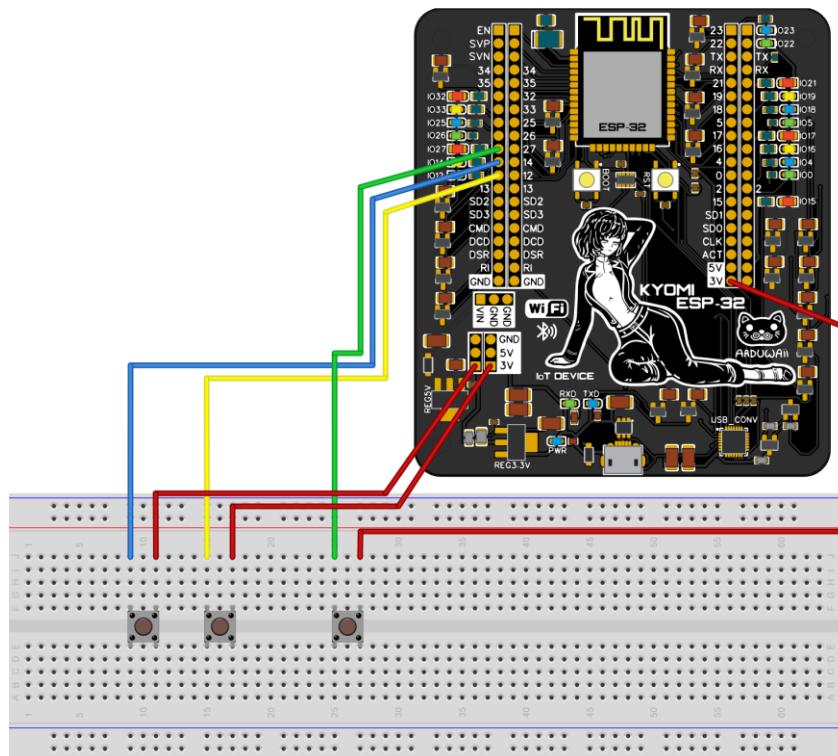
## Ejercicio 12

Este ejercicio será parecido al ejercicio 11, pero ahora usaremos un botón para sumar un 1 a una variable que será guardada en la Flash y con otro botón también sumaremos un 1 a una variable que guardaremos en el RTC. Los valores de estas sumas los recuperaremos al inicio, después de despertar de un DeepSleep. Un tercer botón lo usaremos para activar el modo DeepSleep, que durará 10 segundos.

Materiales:

- 3 botones pulsadores
- 6 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:



Veamos el código. Incluiremos la librería `Preferences.h` y crearemos el objeto `FLASH` para guardar datos en la Flash. Las variables `FACTOR_us` y `TIEMPO_SLEEP` las usaremos para calcular el tiempo en el que se despertará el ESP32 del DeepSleep, en este caso 10 segundos o  $10000000\mu s$ :

```
#include <Preferences.h>
Preferences FLASH;
const unsigned int FACTOR_us=1000000;
const unsigned int TIEMPO_SLEEP=10;
```



```
const int GPIO_SUMA_FLASH=14, GPIO_SUMA_RTC=12, GPIO_DEEPSLEEP=27;  
int VALOR_SUMA_FLASH=0, VALOR_GUARDADO;  
RTC_DATA_ATTR int VALOR_SUMA_RTC=0;
```

También tendremos unas variables para guardar los valores en la Flash y para la declaración de los GPIO's. La variable `VALOR_SUMA_RTC` la guardaremos en el RTC.

Dentro de `setup()`, primero colocamos las instrucciones para recuperar a `VALOR_SUMA_FLASH` del espacio `VALOR` y después configuramos el espacio de memoria:

```
FLASH.begin("SUMA_FLASH", false);  
VALOR_GUARDADO=FLASH.getInt("VALOR",0);  
FLASH.end();  
  
FLASH.begin("SUMA_FLASH", false);  
FLASH.putInt("VALOR", VALOR_SUMA_FLASH);  
FLASH.end();
```

Luego configuraremos el temporizador del RTC para despertar al ESP32 10 segundos después de activar el modo DeepSleep:

```
esp_sleep_enable_timer_wakeup(TIEMPO_SLEEP*FACTOR_uS);
```

Finalmente configuraremos los GPIO's de los botones en modo `INPUT_PULLDOWN` y mostramos los valores recuperados del RTC y la Flash en el Monitor Serie (durante el arranque del programa):

```
pinMode(GPIO_SUMA_FLASH, INPUT_PULLDOWN);  
pinMode(GPIO_SUMA_RTC, INPUT_PULLDOWN);  
pinMode(GPIO_DEEPSLEEP, INPUT_PULLDOWN);  
Serial.begin(115200);  
Serial.print("Valor recuperado de flash=");  
Serial.println(VALOR_GUARDADO);  
Serial.print("Valor recuperado de RTC=");  
Serial.println(VALOR_SUMA_RTC);
```

Dentro de `Loop()` colocaremos tres condicionales `if` para agregar las acciones de cada botón:

```
if(digitalRead(GPIO_SUMA_FLASH)==HIGH)  
{  
    VALOR_GUARDADO++;  
    Serial.print("Suma en FLASH +1=");  
    GUARDADO();
```



```
}

if(digitalRead(GPIO_SUMA_RTC)==HIGH)
{
    VALOR_SUMA_RTC++;
    Serial.print("Suma en RTC +1=");
    Serial.println(VALOR_SUMA_RTC);
}

if(digitalRead(GPIO_DEEPSLEEP)==HIGH)
{
    Serial.println("Modo DeepSleep 10 segundos");
    delay(200);
    esp_deep_sleep_start();
}
delay(200);
```

cuando `GPIO_SUMA_FLASH` está en `HIGH`, a `VALOR_GUARDADO` se le sumará un 1 y su valor se guardará con la subrutina `GUARDADO()`. Cuando el `GPIO_SUMA_RTC` está en `HIGH`, a `VALOR_SUMA_RTC` se le sumará un 1 y su valor se mantendrá guardado en la SRAM del bloque RTC. Finalmente, cuando el `GPIO_DEEPSLEEP` está en `HIGH`, el modo DeepSleep se activará y el temporizador empezará a contar para despertar al ESP32 después de 10 segundos.

La subrutina `GUARDADO()` simplemente contiene las instrucciones para hacer un guardado sencillo en la Flash:

```
void GUARDADO()
{
    VALOR_SUMA_FLASH=VALOR_GUARDADO;
    FLASH.begin("SUMA_FLASH", false);
    FLASH.putInt("VALOR", VALOR_SUMA_FLASH);
    FLASH.end();
    Serial.println(VALOR_SUMA_FLASH);
}
```

El código completo queda de la siguiente forma:

```
#include <Preferences.h>
Preferences FLASH;
const unsigned int FACTOR_uS=1000000;
const unsigned int TIEMPO_SLEEP=10;
```



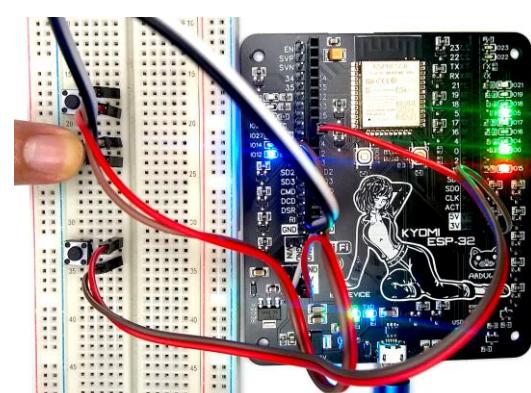
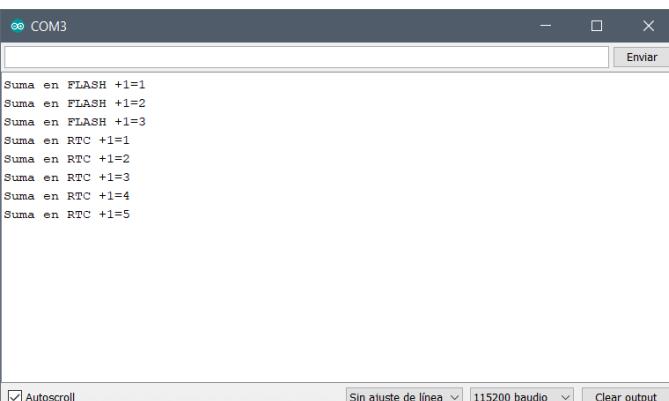
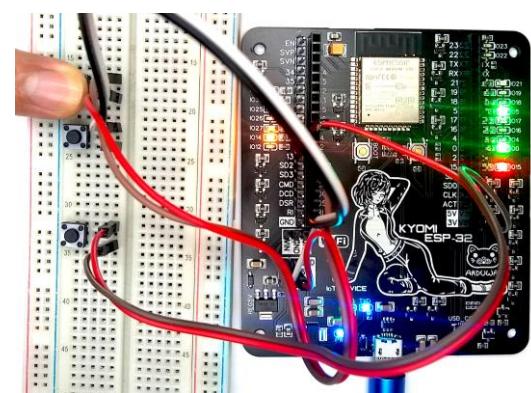
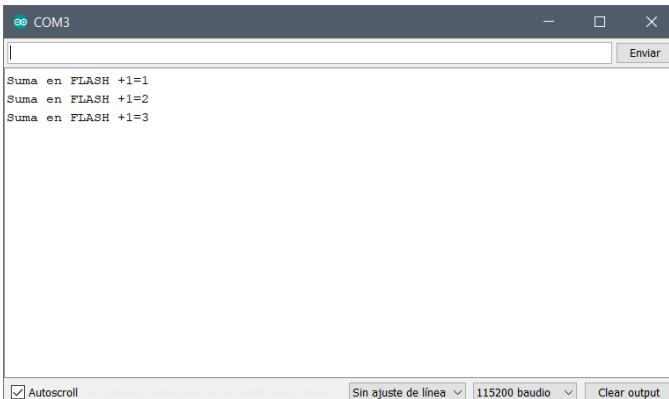
```
const int GPIO_SUMA_FLASH=14, GPIO_SUMA_RTC=12, GPIO_DEEPSLEEP=27;  
int VALOR_SUMA_FLASH=0, VALOR_GUARDADO;  
RTC_DATA_ATTR int VALOR_SUMA_RTC=0;  
void setup()  
{  
    FLASH.begin("SUMA_FLASH", false);  
    VALOR_GUARDADO=FLASH.getInt("VALOR",0);  
    FLASH.end();  
    FLASH.begin("SUMA_FLASH", false);  
    FLASH.putInt("VALOR", VALOR_SUMA_FLASH);  
    FLASH.end();  
    esp_sleep_enable_timer_wakeup(TIEMPO_SLEEP*FACTOR_uS);  
    pinMode(GPIO_SUMA_FLASH, INPUT_PULLDOWN);  
    pinMode(GPIO_SUMA_RTC, INPUT_PULLDOWN);  
    pinMode(GPIO_DEEPSLEEP, INPUT_PULLDOWN);  
    Serial.begin(115200);  
    Serial.print("Valor recuperado de flash=");  
    Serial.println(VALOR_GUARDADO);  
    Serial.print("Valor recuperado de RTC=");  
    Serial.println(VALOR_SUMA_RTC);  
}  
void loop()  
{  
    if(digitalRead(GPIO_SUMA_FLASH)==HIGH)  
    {  
        VALOR_GUARDADO++;  
        Serial.print("Suma en FLASH +1=");  
        GUARDADO();  
    }  
    if(digitalRead(GPIO_SUMA_RTC)==HIGH)  
    {  
        VALOR_SUMA_RTC++;  
        Serial.print("Suma en RTC +1=");  
        Serial.println(VALOR_SUMA_RTC);  
    }  
    if(digitalRead(GPIO_DEEPSLEEP)==HIGH)
```



## Capítulo 3: Modo DeepSleep y sus funciones

```
{  
    Serial.println("Modo DeepSleep 10 segundos");  
    delay(200);  
    esp_deep_sleep_start();  
}  
delay(200);  
}  
  
void GUARDADO()  
{  
    VALOR_SUMA_FLASH=VALOR_GUARDADO;  
    FLASH.begin("SUMA_FLASH", false);  
    FLASH.putInt("VALOR", VALOR_SUMA_FLASH);  
    FLASH.end();  
    Serial.println(VALOR_SUMA_FLASH);  
}
```

Ahora prueba el programa. Suma unos cuantos valores a las variables guardadas en la Flash y en el RTC:



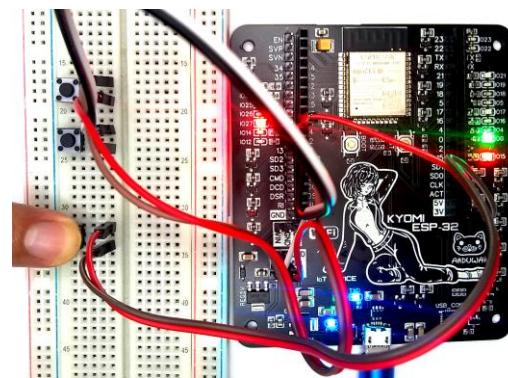


## Capítulo 3: Modo DeepSleep y sus funciones

Cuando presiones el tercer botón, el ESP32 entrará en DeepSleep por 10 segundos:

```
COM3
Suma en FLASH +1=1
Suma en FLASH +1=2
Suma en FLASH +1=3
Suma en RTC +1=1
Suma en RTC +1=2
Suma en RTC +1=3
Suma en RTC +1=4
Suma en RTC +1=5
Modo DeepSleep 10 segundos

 Autoscroll Sin ajuste de línea 115200 baudio Clear output
```



Verás que se van a recuperar las últimas sumas después de despertar al ESP32. Si sumas unos cuantos valores más y presionas el botón RESET de la tarjeta, la última suma guardada en la Flash se recuperará, pero el valor guardado en el RTC volverá a 0.

```
COM3
Suma en RTC +1=4
Suma en RTC +1=5
Modo DeepSleep 10 segundos
ets Jun 8 2016 00:22:57

rst:0x5 DEEPSLEEP_RESET ,boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
Valor recuperado de flash=3
Valor recuperado de RTC=5

 Autoscroll
```

```
COM3
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
Valor recuperado de flash=3
Valor recuperado de RTC=5

 Autoscroll
```

```
COM3
Suma en RTC +1=8
Suma en RTC +1=9
Suma en RTC +1=10
ets Jun 8 2016 00:22:57

rst:0x1 POWERON_RESET ,boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
Valor recuperado de flash=8
Valor recuperado de RTC=0

 Autoscroll Sin ajuste de línea 115200 baudio Clear output
```



## DeepSleep con GPIO táctil (touch)

Otra manera de despertar al ESP32 de un DeepSleep, es utilizando los GPIO's touch. Cualquiera de los GPIO's con funcionalidad touch puede ser usado.

Para utilizar un GPIO touch en DeepSleep, primero debemos configurar una función de interrupción como la siguiente:

```
touchAttachInterrupt(PIN TOUCH, callback, VALOR UMBRAL);  
void callback()  
{  
}  
}
```

donde en *PIN TOUCH* colocamos el pin touch de la misma manera a como lo hacíamos en las funciones táctiles (no el número de GPIO), *callback* es una subrutina auxiliar que se ejecutará al detectar la interrupción (podemos dejarla vacía) y *VALOR UMBRAL* es el valor mínimo en el cual se detectará la lectura touch. En este caso, *VALOR UMBRAL* se encuentra en la escala de 0 a 20, es decir, una escala sin tratamiento matemático, así que debes tener en cuenta los valores de umbral que vimos en la primera prueba touch. Un valor mínimo de 5 puede ser suficiente.

Una vez configurado el GPIO, simplemente colocamos la instrucción para despertar al ESP32 con una interrupción touch:

```
esp_sleep_enable_touchpad_wakeup();
```

Puedes usar más de un GPIO touch configurando más interrupciones, y pueden compartir la misma subrutina *callback* o puedes asignarles una propia. Recuerda que estas configuraciones van en *setup()*.

### Ejercicio 13

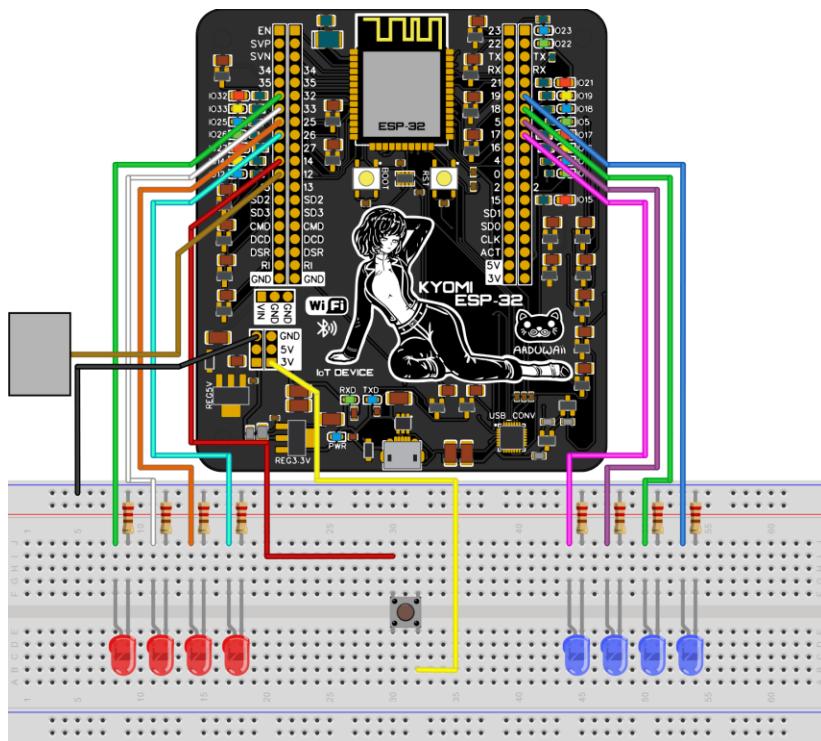
En este ejercicio usaremos un botón para interrumpir una secuencia de LEDs y activar el DeepSleep. Despertaremos al ESP32 con un GPIO touch. También contaremos el número de secuencias que han transcurrido y lo guardaremos en el RTC.

Materiales:

- 1 trozo de papel aluminio
- 1 botón pulsador
- 8 LEDs (opcional)
- 8 resistencias de 220Ω (opcional)
- 12 jumpers o cables tipo Dupont (9 para los LEDs)
- plantilla de experimentos



El circuito es el siguiente:



Ahora veamos el código. Tendremos un par de arrays con todos los GPIO, una variable para los ciclos *for* y otra variable para el conteo de las secuencias, que será guardada en el RTC:

```
const int PINS_IQZ[5]={32, 33, 25, 26, 27}, PINS_DER[11]={23, 22, 21, 19, 18, 5, 17, 16, 4, 0, 15}, GPIO_BOTON=14;  
RTC_DATA_ATTR int CONTADOR_DE_SECUENCIAS = 0;  
int CONTADOR_FOR=0;
```

Dentro de *setup()*, configuraremos la comunicación serial, los GPIO's para los LEDs y el GPIO para el botón:

```
Serial.begin(115200);  
Serial.println("ESP32 Activo");  
for(CONTADOR_FOR=0;CONTADOR_FOR<=4;CONTADOR_FOR++)  
{  
    pinMode(PINS_IQZ[CONTADOR_FOR],OUTPUT);  
    digitalWrite(PINS_IQZ[CONTADOR_FOR],LOW);  
}  
  
for(CONTADOR_FOR=0;CONTADOR_FOR<=10;CONTADOR_FOR++)  
{  
    pinMode(PINS_DER[CONTADOR_FOR],OUTPUT);  
}
```



```
    digitalWrite(PINS_DER[CONTADOR_FOR], LOW);  
}  
pinMode(GPIO_BOTON, INPUT_PULLDOWN);
```

después vamos a configurar una interrupción para el botón y así poder activar el modo DeepSleep en cualquier momento, usaremos la subrutina `MODO_DEEPSLEEP()` para activarlo, la interrupción será de tipo `RISING`:

```
attachInterrupt(digitalPinToInterrupt(GPIO_BOTON), MODO_DEEPSLEEP, RISING);
```

Finalmente, configuraremos la interrupción touch para despertar al ESP32 usando el pin touch `T5` (GPIO12), con un valor umbral de 5:

```
touchAttachInterrupt(T5, callback, 5);  
esp_sleep_enable_touchpad_wakeup();
```

Dentro de `loop()`, primero ejecutamos la subrutina `SECUENCIA_LEDS()` para iniciar la secuencia de LEDs. Al terminar, `CONTADOR_DE_SECUENCIAS` incrementa en 1 y su valor queda guardado en el RTC, también mostraremos el conteo en el Monitor Serie:

```
SECUENCIA_LEDS();  
CONTADOR_DE_SECUENCIAS++;  
Serial.print("Secuencia número:");  
Serial.println(CONTADOR_DE_SECUENCIAS);  
delay(50);
```

La subrutina `SECUENCIA_LEDS` simplemente enciende y apaga los LEDs de los GPIO's 32, 33, 25, 26, 19, 18, 5 y 17:

```
void SECUENCIA_LEDS()  
{  
    for(CONTADOR_FOR=0; CONTADOR_FOR<=3; CONTADOR_FOR++){digitalWrite(PINS_I2Q[CONTADOR_FOR], HIGH); delay(100);}  
  
    for(CONTADOR_FOR=3; CONTADOR_FOR<=6; CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR], HIGH); delay(100);}  
  
    for(CONTADOR_FOR=0; CONTADOR_FOR<=3; CONTADOR_FOR++){digitalWrite(PINS_I2Q[CONTADOR_FOR], LOW); delay(100);}  
  
    for(CONTADOR_FOR=3; CONTADOR_FOR<=6; CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR], LOW); delay(100);}  
}
```



La subrutina `MODO_DEEPSLEEP()` se ejecuta cuando ocurre la interrupción del botón del GPIO14, deteniendo la secuencia de LEDs y activando el modo DeepSleep:

```
void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}
```

La subrutina `callback` la dejaremos vacía:

```
void callback()
{
}
```

El código completo queda de la siguiente forma:

```
const int PINS_IQZ[5]={32, 33, 25, 26, 27}, PINS_DER[11]={23, 22, 21, 19, 18, 5,
17, 16, 4, 0, 15}, GPIO_BOTON=14;

RTC_DATA_ATTR int CONTADOR_DE_SECUENCIAS = 0;
int CONTADOR_FOR=0;
void setup()
{
    Serial.begin(115200);
    Serial.println("ESP32 Activo");

    for(CONTADOR_FOR=0;CONTADOR_FOR<=4;CONTADOR_FOR++)
    {
        pinMode(PINS_IQZ[CONTADOR_FOR],OUTPUT);
        digitalWrite(PINS_IQZ[CONTADOR_FOR],LOW);
    }
    for(CONTADOR_FOR=0;CONTADOR_FOR<=10;CONTADOR_FOR++)
    {
        pinMode(PINS_DER[CONTADOR_FOR],OUTPUT);
        digitalWrite(PINS_DER[CONTADOR_FOR],LOW);
    }
    pinMode(GPIO_BOTON,INPUT_PULLDOWN);

    attachInterrupt(digitalPinToInterruption(GPIO_BOTON),MODO_DEEPSLEEP,RISING);
    touchAttachInterrupt(T5,callback,5);
    esp_sleep_enable_touchpad_wakeup();
}
```



```
void Loop()
{
    SECUENCIA_LEDS();
    CONTADOR_DE_SECUENCIAS++;
    Serial.print("Secuencia número:");
    Serial.println(CONTADOR_DE_SECUENCIAS);
    delay(50);
}

void SECUENCIA_LEDS()
{
    for(CONTADOR_FOR=0;CONTADOR_FOR<=3;CONTADOR_FOR++){digitalWrite(PINS_IQZ[CONTADOR_FOR],HIGH); delay(100);}

    for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],HIGH); delay(100);}

    for(CONTADOR_FOR=0;CONTADOR_FOR<=3;CONTADOR_FOR++){digitalWrite(PINS_IQZ[CONTADOR_FOR],LOW); delay(100);}

    for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],LOW); delay(100);}
}

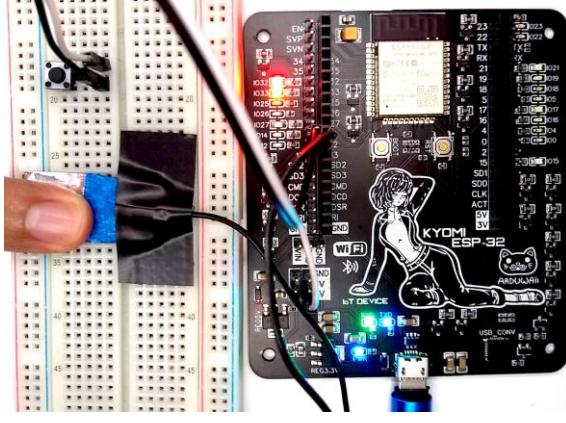
void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}

void callback()
{}
```



## Capítulo 3: Modo DeepSleep y sus funciones

Prueba el programa. Podrás interrumpir la secuencia de LEDs en cualquier momento con el botón de DeepSleep. Cuando despiertes al ESP32 (con el pad touch de papel aluminio), el conteo enviado al Monitor Serie continuará con el último valor guardado:

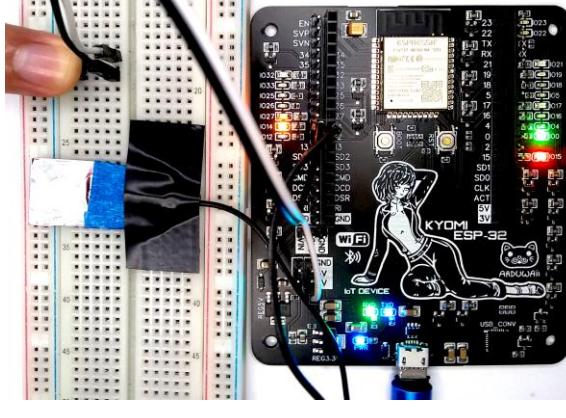


```
∞ COM3
ESP32 Activo
Secuencia número:1
Secuencia número:2
Secuencia número:3
Secuencia número:4
Secuencia número:5
Secuencia número:6
Secuencia número:7
Secuencia número:8
Secuencia número:9
Secuencia número:10
Secuencia número:11
Secuencia número:12
Secuencia número:13
Secuencia número:14
Secuencia número:15

 Autoscroll




```



```
∞ COM3
Secuencia número:14
Secuencia número:15
Modets Jun 8 2016 00:22:57

rst:0x5 [DEEPSLEEP_RESET],boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
ESP32 Activo
Secuencia número:16
Secuencia número:17

 Autoscroll




```

## DeepSleep con señales externas

Además del temporizador del RTC y los GPIO's touch, podemos usar señales digitales externas para despertar al ESP32. Estas señales pueden ser detectadas por todos los GPIO's conectados al bloque RTC (revisa el pinout). Existen dos modos de uso para estas señales: `ext(0)` y `ext(1)`.

El modo `ext(0)` sólo permite usar uno de los GPIO's conectados al RTC, mientras que el modo `ext(1)` permite usar varios GPIO's a la vez.

### modo `ext(0)`

En este modo sólo se puede configurar un GPIO a la vez para detectar una señal digital. Para configurar el GPIO, se usa la siguiente instrucción:

```
esp_sleep_enable_ext0_wakeup(GPIO_NUM_PIN, TIPO DE SEÑAL)
```



donde en *PIN* se coloca el número del GPIO que vayamos a utilizar, y en *TIPO DE SEÑAL* colocamos un 1 o *HIGH* o un 0 o *LOW* para especificar el estado digital que despertará al ESP32.

## Modo ext(1)

En este modo se puede configurar un grupo de GPIO's para despertar al ESP32. También puedes configurar sólo un GPIO, pero es poco práctico y la mejor opción sería utilizar el modo ext(0). Para configurar el grupo de GPIO's, se usa la siguiente instrucción:

```
esp_sleep_enable_ext1_wakeup(MASCARA DE PINES, MODO)
```

donde *MASCARA DE PINES* es un número hexadecimal que agrupa a todos los GPIO's que vayas a utilizar, mientras que *MODO* puede ser uno de los siguientes modos de detección de señal:

- *ESP\_EXT1\_WAKEUP\_ALL\_LOW* : en este modo, todos los GPIO's elegidos deben estar en *LOW* para despertar al ESP32.
- *ESP\_EXT1\_WAKEUP\_ANY\_HIGH* : en este modo, sólo uno los GPIO's elegidos debe estar en *HIGH* para despertar al ESP32.

El número que se coloca en *MASCARA DE PINES* lo calcularemos de la siguiente manera:

1. Elegimos los GPIO's que necesitemos y sus números de pin los convertimos a decimal binario, por ejemplo:

para usar el GPIO12, el resultado sería:  $2^{12}=4096$

para usar el GPIO4 y el GPIO25, el resultado sería:

$$(2^4)+(2^{25})=16+33554432=33554448$$

2. El resultado obtenido lo convertimos a su equivalente hexadecimal, puedes usar cualquier convertidor decimal a hexadecimal:

A screenshot of a hex converter tool. It has two input fields: 'Decimal Value (max: 9223372036854775807)' containing '33554448' and a 'Convert' button. To its right is another field labeled 'Hexadecimal Value' containing '2000010'. Below these fields is a link 'swap conversion: Hex to Decimal'.

para  $2^{12}=4096$ , el equivalente hexadecimal sería:1000

para  $(2^4)+(2^{25})=33554448$ , el equivalente hexadecimal sería:  
2000010

3. El resultado en hexadecimal lo colocaremos en *MASCARA DE PINES*. Recuerda agregar '0x' para indicar la base numérica, por ejemplo: **0x1000** o **0x2000010**.



**CONSEJO:** es recomendable que uses los GPIO's con números de pin pequeños para calcular la **MASCARA DE PINES** más rápido. Además, no es muy práctico usar más de tres GPIO para despertar al ESP32, particularmente con el modo **ESP\_EXT1\_WAKEUP\_ALL\_LOW**, donde tendrías generar varias señales **LOW** al mismo tiempo.

Ahora hagamos un ejercicio sencillo con `ext(0)`.

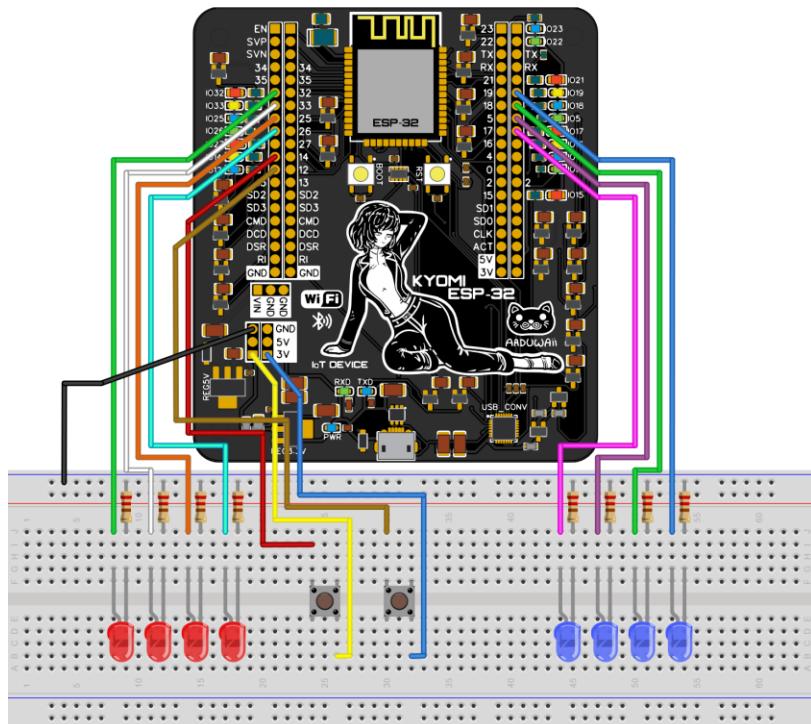
## Ejercicio 14

Vamos a retomar el ejercicio 13 y cambiaremos el pad touch por un botón para despertar al ESP32 con `ext(0)`. De igual manera, contaremos el número de secuencias de LEDs que han transcurrido y lo guardaremos en la memoria del RTC.

**Materiales:**

- 2 botones pulsadores
- 8 LEDs (opcional)
- 8 resistencias de 220Ω (opcional)
- 13 jumpers o cables tipo Dupont (9 para los LEDs)
- plantilla de experimentos

El circuito es el siguiente:





Ahora veamos el código. Los arrays con todos los GPIO's, la variable para los ciclos `for` y la variable para el conteo de las secuencias se mantienen iguales:

```
const int PINS_IQZ[5]={32 ,33 ,25 ,26 ,27 }, PINS_DER[11]={23 ,22 ,21 ,19 ,18 ,5  
,17 ,16 ,4 ,0 , 15}, BOTON_DEEP_SLEEP=14;  
  
RTC_DATA_ATTR int CONTADOR_DE_SECUENCIAS = 0;  
  
int CONTADOR_FOR=0;
```

Dentro de `setup()`, la comunicación serial, los GPIO's para los LEDs y el GPIO del botón (así como su interrupción) tendrán la misma configuración:

```
Serial.begin(115200);  
  
Serial.println("ESP32 Activo");  
  
for(CONTADOR_FOR=0;CONTADOR_FOR<=4;CONTADOR_FOR++)  
{  
    pinMode(PINS_IQZ[CONTADOR_FOR],OUTPUT);  
    digitalWrite(PINS_IQZ[CONTADOR_FOR],LOW);  
  
}  
  
for(CONTADOR_FOR=0;CONTADOR_FOR<=10;CONTADOR_FOR++)  
{  
    pinMode(PINS_DER[CONTADOR_FOR],OUTPUT);  
    digitalWrite(PINS_DER[CONTADOR_FOR],LOW);  
  
}  
pinMode(BOTON_DEEP_SLEEP,INPUT_PULLDOWN);  
attachInterrupt(digitalPinToInterrupt(BOTON_DEEP_SLEEP),MODO_DEEPSLEEP,RISING);
```

Finalmente, usaremos el GPIO12 en modo `HIGH` para el `ext0`, pero primero lo configuraremos como `INPUT_PULLDOWN`:

```
pinMode(12,INPUT_PULLDOWN);  
esp_sleep_enable_ext0_wakeup(GPIO_NUM_12, HIGH);
```

Dentro de `loop()` conservaremos las mismas instrucciones y la subrutina `SECUENCIA_LEDS()`:

```
SECUENCIA_LEDS();  
CONTADOR_DE_SECUENCIAS++;  
Serial.print("Secuencia número:");  
Serial.println(CONTADOR_DE_SECUENCIAS);  
delay(50);
```



La subrutina `SECUENCIA_LEDS()` y `MODO_DEEPSLEEP()` también permanecen sin cambios:

```
void SECUENCIA_LEDS()
{
for(CONTADOR_FOR=0;CONTADOR_FOR<=3;CONTADOR_FOR++){digitalWrite(PINS_IQZ[CONTADOR_FOR],HIGH); delay(100);}

for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],HIGH); delay(100);}

for(CONTADOR_FOR=0;CONTADOR_FOR<=3;CONTADOR_FOR++){digitalWrite(PINS_IQZ[CONTADOR_FOR],LOW); delay(100);}

for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],LOW); delay(100);}

}

void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}
```

El código completo queda de la siguiente forma:

```
const int PINS_IQZ[5]={32 ,33 ,25 ,26 ,27 }, PINS_DER[11]={23 ,22 ,21 ,19 ,18 ,5 ,17 ,16 ,4 ,0 ,15}, BOTON_DEEP_SLEEP=14;
RTC_DATA_ATTR int CONTADOR_DE_SECUENCIAS = 0;
int CONTADOR_FOR=0;
void setup()
{
    Serial.begin(115200);
    Serial.println("ESP32 Activo");
    for(CONTADOR_FOR=0;CONTADOR_FOR<=4;CONTADOR_FOR++)
    {
        pinMode(PINS_IQZ[CONTADOR_FOR],OUTPUT);
        digitalWrite(PINS_IQZ[CONTADOR_FOR],LOW);
    }
    for(CONTADOR_FOR=0;CONTADOR_FOR<=10;CONTADOR_FOR++)
    {
        pinMode(PINS_DER[CONTADOR_FOR],OUTPUT);
        digitalWrite(PINS_DER[CONTADOR_FOR],LOW);
    }
}
```



```
pinMode(BOTON_DEEP_SLEEP, INPUT_PULLDOWN);
attachInterrupt(digitalPinToInterrupt(BOTON_DEEP_SLEEP), MODO_DEEPSLEEP, RISING);
pinMode(12, INPUT_PULLDOWN);
esp_sleep_enable_ext0_wakeup(GPIO_NUM_12, HIGH);
}

void Loop()
{
    SECUENCIA_LEDS();
    CONTADOR_DE_SECUENCIAS++;
    Serial.print("Secuencia número:");
    Serial.println(CONTADOR_DE_SECUENCIAS);
    delay(50);
}

void SECUENCIA_LEDS()
{
    for(CONTADOR_FOR=0;CONTADOR_FOR<=3;CONTADOR_FOR++){digitalWrite(PINS_IZO[CONTADOR_FOR],HIGH); delay(100);}

    for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],HIGH); delay(100);}

    for(CONTADOR_FOR=0;CONTADOR_FOR<=3;CONTADOR_FOR++){digitalWrite(PINS_IZO[CONTADOR_FOR],LOW); delay(100);}

    for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],LOW); delay(100);}
}

void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}
```

Ahora prueba el programa. Su funcionamiento es el mismo del ejercicio 13, con la diferencia que ahora el pad touch es reemplazado por el botón pulsador. La ventaja de usar el pulsador es que la señal detectada no necesita calibrarse con un valor umbral, simplemente es un valor `HIGH` o



**LOW.** Ahora hagamos un ejercicio con `ext(1)`, pero primero, veamos cómo usar ‘definiciones’ en Arduino.

### Básicos: Uso de definiciones (`#define`)

Las definiciones permiten crear valores constantes asociados a un nombre particular, similar a declarar variables tipo `const`. Tienen la siguiente estructura:

```
#define NOMBRE VALOR
```

donde en `NOMBRE` colocamos el nombre que usaremos como referencia y en `VALOR` colocamos el valor constante para la definición. La ventaja de utilizar definiciones es que `VALOR` puede ser un número, cadena de caracteres, valor booleano, etc., y la instrucción donde vayamos a utilizar la definición detectará el tipo de valor dado, por ejemplo:

```
#define MASCARA_HEX 0x0123  
esp_sleep_enable_ext1_wakeup(MASCARA_HEX, ESP_EXT1_WAKEUP_ALL_LOW);  
o también  
#define GPIO_LED 32  
pinMode(GPIO_LED,OUTPUT);
```

**NOTA:** las estructuras `#define` no deben cerrarse con ; y tampoco se pueden asignar sus valores con un =, como `#define NOMBRE = VALOR`. Los nombres que asignes a las definiciones no deben ser iguales a los de otras variables que vayas a declarar, porque se va a generar un conflicto al momento de compilar el código.

## Ejercicio 15

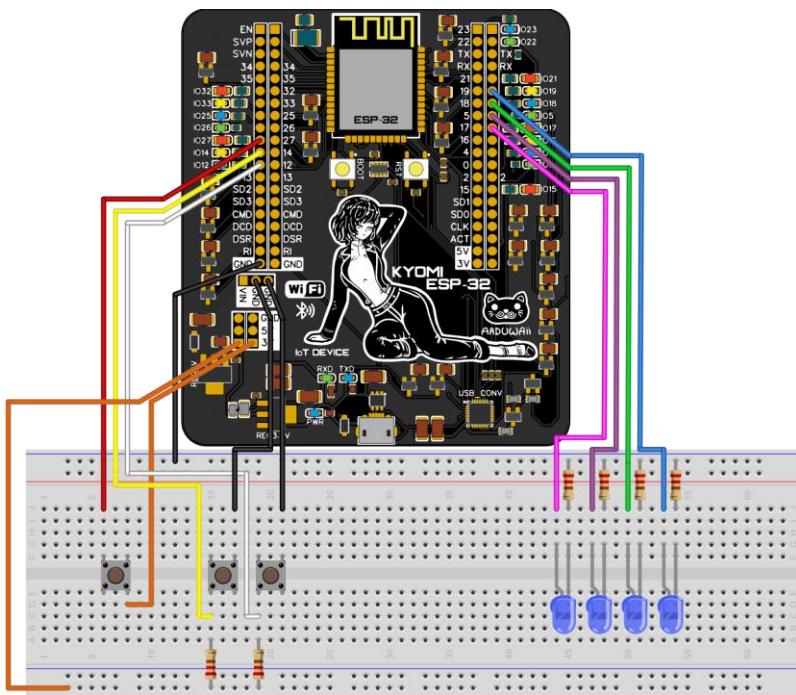
Este ejercicio será similar al ejercicio 13, pero ahora tendremos una secuencia de LEDs pequeña y despertaremos al ESP32 con `ext(1)`, usando dos botones en modo `ESP_EXT1_WAKEUP_ALL_LOW`. También tendremos un botón para activar el DeepSleep en cualquier momento.

Materiales:

- 3 botones pulsadores
- 4 LEDs (opcional)
- 6 resistencias de 220Ω (4 opcionales para los LEDs)
- 11 jumpers o cables tipo Dupont (5 para los LEDs)
- plantilla de experimentos



El circuito es el siguiente:



Usaremos un par de circuitos PULL-UP para los botones del ext(1), ya que la configuración `INPUT_PULLUP` de `pinMode` no funciona cuando está trabajando el bloque RTC en el modo DeepSleep.

Veamos el código. Crearemos una definición para la `MASCARA DE PINES` (`MASCARA_GPIO`), que tendrá el valor hexadecimal que incluye al GPIO14 y 12 ( $4096+16384=0x5000$ ). También incluiremos un array para los GPIO's de los LEDs, el GPIO del botón para DeepSleep y una variable auxiliar para unos ciclos `for`:

```
#define MASCARA_GPIO 0x5000
const int PINS_DER[11]={23,22,21,19,18,5,17,16,4,0,15}, BOTON_DEEP_SLEEP=27;
int CONTADOR_FOR=0;
```

En `setup()` vamos a configurar la comunicación serial, los GPIO's para los LEDs y la interrupción para el botón de DeepSleep:

```
Serial.begin(115200);
Serial.println("ESP32 Activo");

for(CONTADOR_FOR=0;CONTADOR_FOR<=10;CONTADOR_FOR++)
{
    pinMode(PINS_DER[CONTADOR_FOR],OUTPUT);
    digitalWrite(PINS_DER[CONTADOR_FOR],LOW);
}
```



```
pinMode(BOTON_DEEP_SLEEP, INPUT_PULLDOWN);
attachInterrupt(digitalPinToInterrupt(BOTON_DEEP_SLEEP), MODO_DEEPSLEEP, RISING);
```

también configuraremos el ext(1) con el valor de `MASCARA_GPIO` y el modo `ESP_EXT1_WAKEUP_ALL_LOW`:

```
pinMode(12, INPUT);
pinMode(14, INPUT);
esp_sleep_enable_ext1_wakeup(MASCARA_GPIO, ESP_EXT1_WAKEUP_ALL_LOW);
```

Con el modo `ESP_EXT1_WAKEUP_ALL_LOW` debemos presionar los botones del GPIO12 y 14 al mismo tiempo para despertar al ESP32.

Dentro de `Loop()` sólo vamos a ejecutar una subrutina:

```
SECUENCIA_LEDS();
```

La subrutina `SECUENCIA_LEDS()` simplemente enciende y apaga los LEDs de los GPIO's 19 ,18 ,5 y 17:

```
void SECUENCIA_LEDS()
{
for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],HIGH); delay(100);}

for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],LOW); delay(100);}
}
```

Finalmente, la subrutina `MODO_DEEPSLEEP()` se ejecuta cuando se detecta la interrupción del botón de DeepSleep:

```
void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}
```

El código completo queda de la siguiente forma:

```
#define MASCARA_GPIO 0x5000
const int PINS_DER[11]={23,22,21,19,18,5,17,16,4,0,15}, BOTON_DEEP_SLEEP=27;
int CONTADOR_FOR=0;

void setup()
{
    Serial.begin(115200);
    Serial.println("ESP32 Activo");
```



```
for(CONTADOR_FOR=0;CONTADOR_FOR<=10;CONTADOR_FOR++)
{
    pinMode(PINS_DER[CONTADOR_FOR],OUTPUT);
    digitalWrite(PINS_DER[CONTADOR_FOR],LOW);
}

pinMode(BOTON_DEEP_SLEEP,INPUT_PULLDOWN);
attachInterrupt(digitalPinToInterrupt(BOTON_DEEP_SLEEP),MODO_DEEPSLEEP,RISING);

pinMode(12,INPUT);
pinMode(14,INPUT);
esp_sleep_enable_ext1_wakeup(MASCARA_GPIO, ESP_EXT1_WAKEUP_ALL_LOW);
}

void Loop()
{
    SECUENCIA_LEDS();
}

void SECUENCIA_LEDS()
{
    for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],HIGH); delay(100);}

    for(CONTADOR_FOR=3;CONTADOR_FOR<=6;CONTADOR_FOR++){digitalWrite(PINS_DER[CONTADOR_FOR],LOW); delay(100);}
}

void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}
```



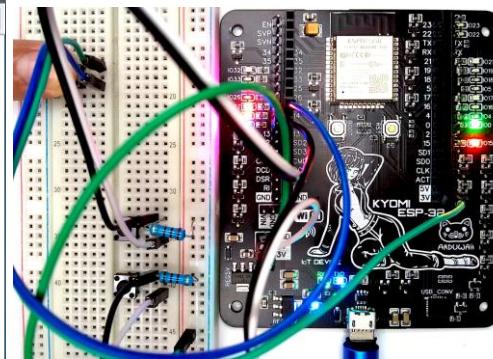
## Capítulo 3: Modo DeepSleep y sus funciones

Ahora prueba el programa. Verás que el ESP32 no despertará de DeepSleep a menos que presiones los dos botones al mismo tiempo. Este modo es útil para no despertar al ESP32 por accidente, requiriendo que ambos botones sean presionados, en lugar de usar un solo pad touch o un solo botón:

```
∞ COM3
ets Jun 8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x33 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
ESP32 Activo

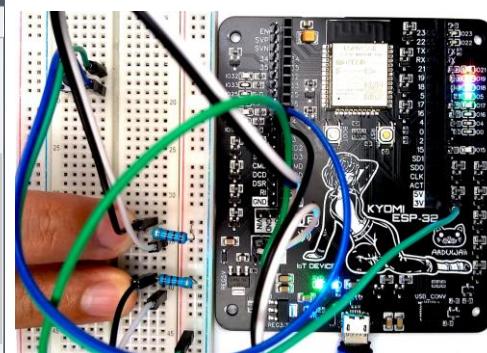
 Autoscroll      Sin ajuste de línea      115200 baudio      Clear output
```



```
∞ COM3
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
ESP32 Activo
ets Jun 8 2016 00:22:57

rst:0x5 [DEEPSLEEP RESET],boot:0x33 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
ESP32 Activo

 Autoscroll      Sin ajuste de línea      115200 baudio      Clear output
```



## Combinando métodos para despertar del modo DeepSleep

Podemos usar varios de los métodos para despertar al ESP32 en el mismo programa, configurando cada método con los GPIO's que necesitemos. También podemos incluir el temporizador del RTC.

Al combinar múltiples formas para despertar al ESP32, es importante saber cuál de todas reactivó al micro en un momento dado, como una especie de auditoría. Para esto tenemos algunas instrucciones útiles que podemos agregar a nuestro programa.



## Cómo detectar el método que despertó al ESP32

La siguiente instrucción devuelve un valor entero que determina el método con el que se despertó al ESP32:

```
esp_sleep_get_wakeup_cause()
```

esta instrucción la podemos usar con una variable, por ejemplo:

```
int DESPERTADO_POR=esp_sleep_get_wakeup_cause();
```

Dependiendo del número devuelto por `esp_sleep_get_wakeup_cause()`, corresponderá con alguno de los siguientes métodos:

- **1:** fue despertado por ext(0)
- **2:** fue despertado por ext(1)
- **3:** fue despertado por el temporizador del RTC
- **4:** fue despertado por un pin touch

Cuando `esp_sleep_get_wakeup_cause()` entrega un **5**, el ESP32 fue despertado por una instrucción programada en el procesador del RTC, pero esto tiene que ver con programación más avanzada.

Al configurar múltiples GPIO's touch, podemos usar la siguiente instrucción para saber cuál de todos despertó al ESP32:

```
esp_sleep_get_touchpad_wakeup_status()
```

que igualmente la podemos usar con una variable, por ejemplo:

```
int PIN_TOUCH = esp_sleep_get_touchpad_wakeup_status();
```

El valor entero que entrega `esp_sleep_get_touchpad_wakeup_status()` va de 0 a 9 y corresponderá con alguno de los pines touch T0 a T9, respectivamente.

En el modo ext(1) con múltiples GPIO's, también podemos saber cuál de todos despertó al micro (solo cuando está configurado como `ESP_EXT1_WAKEUP_ANY_HIGH`), para esto usaremos:

```
esp_sleep_get_ext1_wakeup_status();
```

también puede ir en una variable, por ejemplo:

```
int GPIO_EXT_1 = esp_sleep_get_ext1_wakeup_status();
```

el valor entregado por `esp_sleep_get_ext1_wakeup_status()` es hexadecimal, pero lo podemos convertirlo a entero con la siguiente fórmula:

```
int GPIO_DECIMAL = Log(GPIO_EXT_1)/Log(2);
```

donde la variable `GPIO_DECIMAL` da como resultado el número entero del GPIO que activó al micro (puedes usar otros nombres para las variables), por ejemplo:

si `GPIO_EXT_1=4096`, `GPIO_DECIMAL=Log(GPIO_EXT_1)/Log(2)=Log(4096)/Log(2)=12`



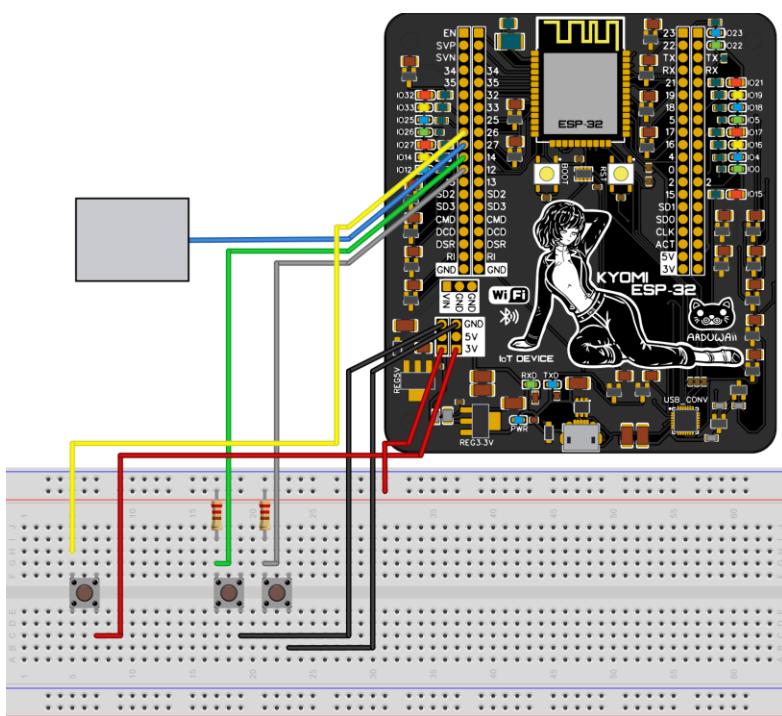
## Ejercicio 16

En este ejercicio usaremos el temporizador, un par de botones configurados para ext(1) y un pin touch para despertar al ESP32 del modo DeepSleep. También mostraremos un mensaje en el Monitor Serie, que nos indicará cuál método despertó al ESP32.

Materiales:

- 3 botones pulsadores
- 2 resistencias de 220Ω
- 1 trozo de papel aluminio
- 8 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:



En este ejercicio también tendremos que usar un par de circuitos PULL-UP para los botones del ext(1).

Ahora veamos el código. Crearemos una definición para la [MASCARA DE PINES](#) ([MASCARA\\_GPIO](#)) con el GPIO14 y 12 (0x5000). También tendremos las variables para el temporizador, el valor umbral del pin touch y otras variables auxiliares:

```
#define MASCARA_GPIO 0x5000  
const int UMBRAL_TOUCH=5, TIEMPO_DEEP_SLEEP=10, FACTOR_uS=1000000,  
BOTON_DEEP_SLEEP= 26;
```



```
int MODO_DESPERTADO, PIN_TOUCH;
```

En `setup()` vamos a configurar la comunicación serial, el GPIO12 y 14 y la interrupción para el botón que activará el DeepSleep:

```
Serial.begin(115200);
Serial.println("ESP32 Activo");
pinMode(BOTON_DEEP_SLEEP, INPUT_PULLDOWN);
attachInterrupt(digitalPinToInterrupt(BOTON_DEEP_SLEEP), MODO_DEEPSLEEP, RISING);

pinMode(12, INPUT);
pinMode(14, INPUT);
```

durante el arranque del programa (en `setup()`), las subrutinas `RAZON_DESPERTADO()` y `VER_PIN_TOUCH_DETECTADO()` se ejecutarán para saber cuál método despertó al ESP32:

```
RAZON_DESPERTADO();
VER_PIN_TOUCH_DETECTADO();
```

configuraremos la interrupción touch para despertar al ESP32 con `T7` y con el valor de `UMBRAL_TOUCH`. El temporizador para el DeepSleep estará configurado con el tiempo dado por `TIEMPO_DEEP_SLEEP*FACTOR_uS`, y `ext(1)` va estar configurado con el valor de `MASCARA_GPIO` y con el modo `ESP_EXT1_WAKEUP_ALL_LOW`. Todos los métodos para despertar al ESP32 estarán activos (en `setup()`):

```
touchAttachInterrupt(T7, callback, UMBRAL_TOUCH);
esp_sleep_enable_touchpad_wakeup();
esp_sleep_enable_timer_wakeup(TIEMPO_DEEP_SLEEP*FACTOR_uS);
esp_sleep_enable_ext1_wakeup(MASCARA_GPIO, ESP_EXT1_WAKEUP_ALL_LOW);
```

La rutina `Loop()` estará vacía.

La subrutina `RAZON_DESPERTADO()` enviará un mensaje al Monitor Serie según el valor entregado por `esp_sleep_get_wakeup_cause()`, que dependerá del método usado para despertar al ESP32. Usaremos una estructura `switch` para enviar los mensajes al Monitor Serie:

```
void RAZON_DESPERTADO()
{
    MODO_DESPERTADO = esp_sleep_get_wakeup_cause();

    switch(MODO_DESPERTADO)
    {
        case 1: Serial.println("Despertado por ext(0)"); break;
        case 2: Serial.println("Despertado por ext(1)"); break;
        case 3: Serial.println("Despertado por temporizador"); break;
    }
}
```



```
    case 4: Serial.println("Despertado por pin touch"); break;
}
}
```

La subrutina `VER_PIN_TOUCH_DETECTADO()` funciona de manera similar a `RAZON_DESPERTADO()`, sólo que ahora detecta el pin touch usado de acuerdo al valor entregado por `esp_sleep_get_touchpad_wakeup_status()`:

```
void VER_PIN_TOUCH_DETECTADO()
{
    PIN_TOUCH = esp_sleep_get_touchpad_wakeup_status();
    switch(PIN_TOUCH)
    {
        case 0: Serial.println("Touch T0 detectado"); break;
        case 1: Serial.println("Touch T1 detectado"); break;
        case 2: Serial.println("Touch T2 detectado"); break;
        case 3: Serial.println("Touch T3 detectado"); break;
        case 4: Serial.println("Touch T4 detectado"); break;
        case 5: Serial.println("Touch T5 detectado"); break;
        case 6: Serial.println("Touch T6 detectado"); break;
        case 7: Serial.println("Touch T7 detectado"); break;
        case 8: Serial.println("Touch T8 detectado"); break;
        case 9: Serial.println("Touch T9 detectado"); break;
    }
}
```

La subrutina `MODO_DEEPSLEEP()` se ejecuta cuando se detecta la interrupción del botón para DeepSleep:

```
void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}
```

Finalmente, la subrutina `callback` de la interrupción touch la dejaremos vacía:

```
void callback()
{
}
```



El código completo queda de la siguiente forma:

```
#define MASCARA_GPIO 0x5000
const int UMBRAL_TOUCH=5,      TIEMPO_DEEP_SLEEP=10,      FACTOR_uS=1000000,
BOTON_DEEP_SLEEP= 26;
int MODO_DESPERTADO, PIN_TOUCH;
void setup()
{
    Serial.begin(115200);
    Serial.println("ESP32 Activo");
    pinMode(BOTON_DEEP_SLEEP, INPUT_PULLDOWN);
    attachInterrupt(digitalPinToInterrupt(BOTON_DEEP_SLEEP),MODO_DEEPSLEEP,RISING);
    pinMode(12,INPUT);
    pinMode(14,INPUT);
    RAZON_DESPERTADO();
    VER_PIN_TOUCH_DETECTADO();
    touchAttachInterrupt(T7, callback, UMBRAL_TOUCH);
    esp_sleep_enable_touchpad_wakeup();
    esp_sleep_enable_timer_wakeup(TIEMPO_DEEP_SLEEP*FACTOR_uS);
    esp_sleep_enable_ext1_wakeup(MASCARA_GPIO, ESP_EXT1_WAKEUP_ALL_LOW);
}
void Loop()
{
}
void RAZON_DESPERTADO()
{
    MODO_DESPERTADO = esp_sleep_get_wakeup_cause();

    switch(MODO_DESPERTADO)
    {
        case 1: Serial.println("Despertado por ext(0)"); break;
        case 2: Serial.println("Despertado por ext(1)"); break;
        case 3: Serial.println("Despertado por temporizador"); break;
        case 4: Serial.println("Despertado por pin touch"); break;
    }
}
```



```
void VER_PIN_TOUCH_DETECTADO()
{
    PIN_TOUCH = esp_sleep_get_touchpad_wakeup_status();
    switch(PIN_TOUCH)
    {
        case 0: Serial.println("Touch T0 detectado"); break;
        case 1: Serial.println("Touch T1 detectado"); break;
        case 2: Serial.println("Touch T2 detectado"); break;
        case 3: Serial.println("Touch T3 detectado"); break;
        case 4: Serial.println("Touch T4 detectado"); break;
        case 5: Serial.println("Touch T5 detectado"); break;
        case 6: Serial.println("Touch T6 detectado"); break;
        case 7: Serial.println("Touch T7 detectado"); break;
        case 8: Serial.println("Touch T8 detectado"); break;
        case 9: Serial.println("Touch T9 detectado"); break;
    }
}

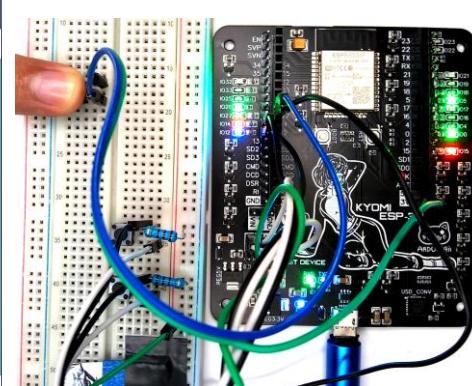
void MODO_DEEPSLEEP()
{
    esp_deep_sleep_start();
}

void callback()
{}
```

**Prueba el programa.** Al activar el modo DeepSleep, si no presionamos los botones del ext(1) o el pad touch, el ESP32 simplemente se despertará por el temporizador del RTC (10 segundos después):

```
00 COM
[redacted] Envío
ets Jun  8 2016 00:22:57
rst:0x5 [DEEPSLEEP_RESET], boot:0x33 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0010,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
ESP32 ACTIVO
Despertado por temporizador

[redacted]
```



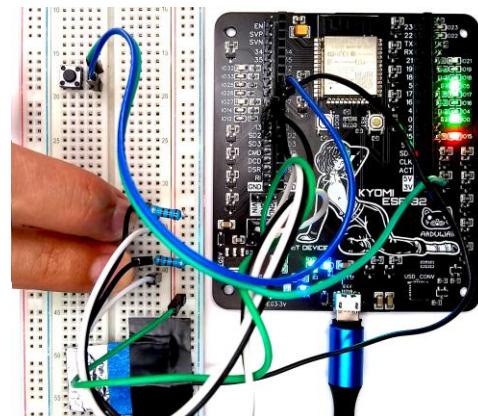


Al presionar el pad touch o los dos botones del ext(1) durante DeepSleep, estos métodos llevarán prioridad sobre el temporizador y el ESP32 se despertará:

```
00 COM3
entry 0x40080310
ESP32 ACTIVO
Despertado por temporizador
ets Jun 8 2016 00:22:57

rst:0x5 [DEEPSLEEP_RESET], boot:0x33 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
ESP32 ACTIVO
Despertado por ext(1)

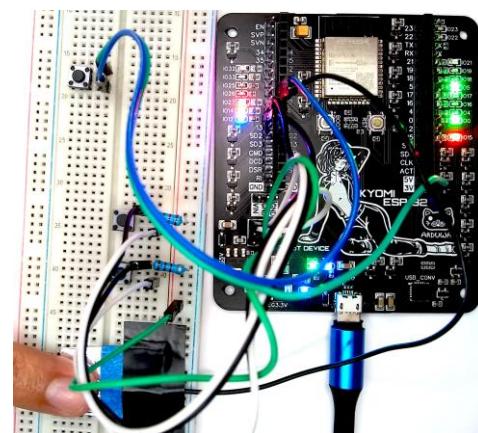
 Autoscroll      Sin ajuste de línea ▾ 115200 baudio ▾ Clear output
```



```
00 COM3
||| 
ESP32 ACTIVO
Despertado por ext(1)
ets Jun 8 2016 00:22:57

rst:0x5 [DEEPSLEEP_RESET], boot:0x33 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:952
load:0x40078000,len:6084
load:0x40080000,len:7936
entry 0x40080310
ESP32 ACTIVO
Despertado por pin touch
Touch T7 detectado

 Autoscroll      Sin ajuste de línea ▾ 115200 baudio ▾ Clear output
```



## Cómo usar los otros modos de consumo energético

El modo DeepSleep es el más utilizado para obtener una mayor eficiencia energética en ciertas aplicaciones, pero también podemos usar los otros modos energéticos si necesitamos deshabilitar ciertas funciones que no vayamos a requerir.

### ModemSleep

Ya hemos visto que en este modo la comunicación Wifi y Bluetooth no están activas. En nuestros programas de Arduino, generalmente se mantienen desactivados estos radios de comunicación cuando no se usan las funciones Wifi o Bluetooth. Pero incluso cuando si utilizamos alguno de los dos sistemas de comunicación, el propio ESP32 los desactiva durante los períodos en los que no van a recibir o transmitir información.



A esto se le conoce como ModemSleep. Sin embargo, con unas cuantas instrucciones podemos desactivar los radios de comunicación Wifi y Bluetooth manualmente, para asegurarnos de que no consuman energía si no los vamos a utilizar.

Para desactivar la comunicación Wifi, tenemos las siguientes instrucciones:

```
WiFi.disconnect(true);  
WiFi.mode(WIFI_OFF);
```

donde `WiFi.disconnect` termina cualquier conexión Wifi activa y `WiFi.mode` deshabilita la comunicación.

Para desactivar la comunicación Bluetooth, tenemos la siguiente instrucción:

```
btStop();
```

Podemos usar estas instrucciones en cualquier parte del código (en una subrutina, por ejemplo). Pero, si queremos habilitar alguno de los dos sistemas de comunicación, tendremos que crear otra subrutina que los reconfigure. No es muy práctico apagar y encender el Wifi o el Bluetooth manualmente si el mismo ESP32 puede administrar su consumo energético automáticamente.

Es recomendable mantener apagados los radios de comunicación sólo cuando no los vayas a usar en tu programa.

## LightSleep

Se mencionó que en este modo la comunicación Wifi y Bluetooth están inactivas, mientras que el procesador principal funciona de manera intermitente y en períodos cortos de tiempo. Es parecido a DeepSleep, pero el procesador principal aún puede realizar ciertas funciones.

Con LightSleep también podemos usar los mismos métodos de DeepSleep para despertar al ESP32 (con las mismas instrucciones).

Para activar el modo LightSleep, tenemos la siguiente instrucción:

```
esp_light_sleep_start();
```

El modo LightSleep es útil para conservar ciertas funciones del procesador, pero con un consumo energético muy bajo.

## Hibernación

En el modo Hibernación, prácticamente todo el módulo ESP32 está apagado, excepto el reloj lento del RTC y la unidad PMU, para monitorear unos pocos GPIO's. Por lo tanto, al activar este modo, no podremos guardar datos en la memoria del RTC (también estará apagada) o ejecutar alguna instrucción en el coprocesador ULP.



Para activar el modo hibernación, prácticamente se igual que con el modo DeepSleep, pero hay que configurar los siguientes parámetros antes de usar la instrucción `esp_deep_sleep_start()`:

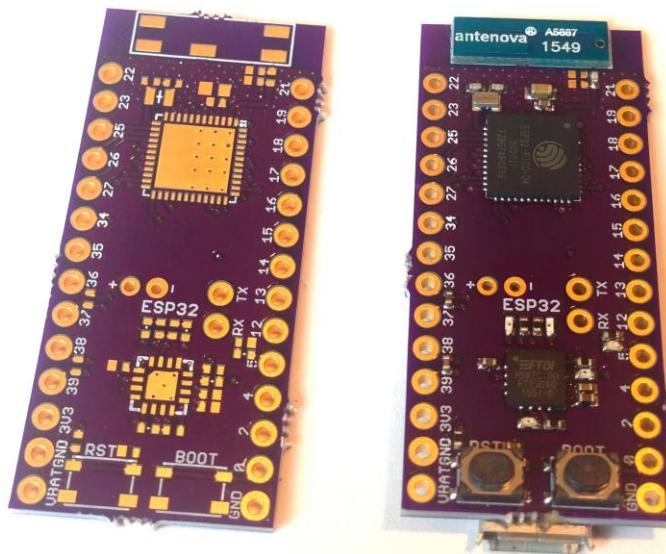
```
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH,    ESP_PD_OPTION_OFF);
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_SLOW_MEM,  ESP_PD_OPTION_OFF);
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_FAST_MEM,   ESP_PD_OPTION_OFF);
esp_sleep_pd_config(ESP_PD_DOMAIN_XTAL,           ESP_PD_OPTION_OFF);
esp_deep_sleep_start();
```

Los parámetros de configuración controlan las siguientes funciones del bloque RTC:

- `ESP_PD_DOMAIN_RTC_PERIPH` controla la mayoría de los periféricos del RTC.
- `ESP_PD_DOMAIN_RTC_FAST_MEM` controla el bloque de memoria rápida del RTC.
- `ESP_PD_DOMAIN_RTC_SLOW_MEM` controla el bloque de memoria lenta del RTC.
- `ESP_PD_DOMAIN_XTAL` controla el oscilador o reloj principal del RTC.

No es necesario desactivar todas las funciones del RTC, aunque mantener algunas activadas no se consideraría como modo Hibernación.

Realmente el modo hibernación sólo es útil con tarjetas de diseño optimizado, que no tienen el ESP32 en forma de módulo, sino que el propio chip del procesador va soldado directamente a la tarjeta, además de usar otros periféricos y reguladores de voltaje más eficientes.



Tarjeta ESP32 de diseño personalizado, optimizada para aplicaciones de baja energía.

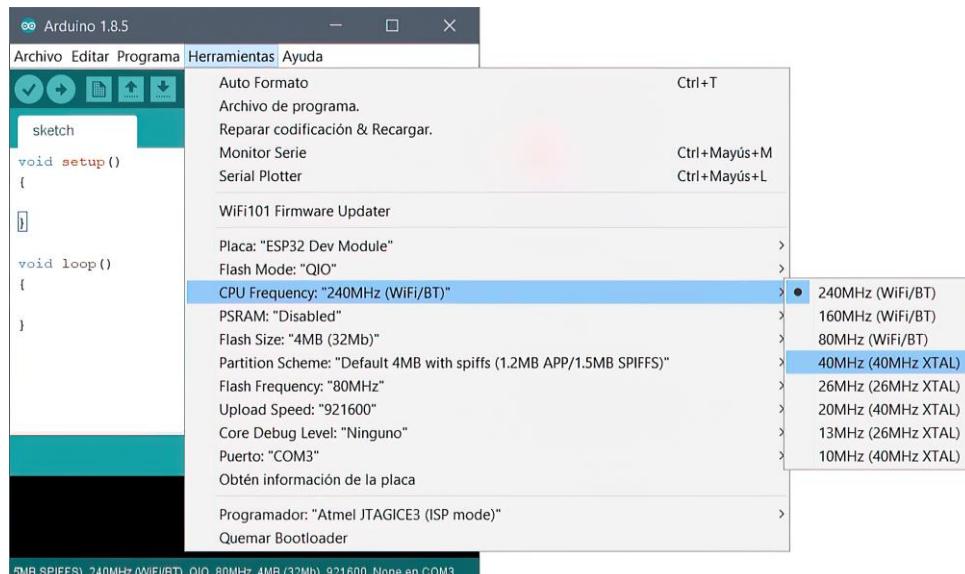


En las tarjetas de desarrollo comunes, prácticamente no hay diferencia de consumo energético entre Hibernación y DeepSleep.

## Disminuir la velocidad del procesador principal

Otra manera de reducir el consumo energético del ESP32, es bajando la velocidad del procesador principal, cambiando su frecuencia.

Normalmente el ESP32 trabaja a 240MHz (máximo), pero se puede reducir su frecuencia hasta 10MHz. Esto lo puedes configurar directamente en el IDE de Arduino, en la pestaña de Herramientas:



o bien, puedes configurar la frecuencia con la siguiente librería e instrucción:

```
#include "esp32-hal-cpu.h"  
setCpuFrequencyMhz(FRECUENCIA);
```

donde **FRECUENCIA** está dada en MHz y `setCpuFrequencyMhz()` debe ir dentro de `setup()`, preferiblemente al principio.

Con la siguiente instrucción podremos saber a qué frecuencia está trabajando el ESP32 (también necesita la librería `esp32-hal-cpu.h`):

```
getCpuFrequencyMhz();
```

Por ejemplo, para configurar la velocidad del procesador a 80MHz, tendríamos:

```
#include "esp32-hal-cpu.h"  
  
void setup()  
{  
    setCpuFrequencyMhz(80);
```

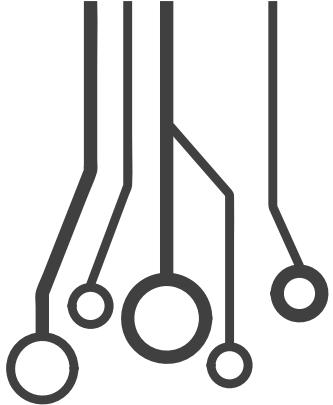


```
Serial.begin(115200)  
int FRECUENCIA = getCPUFrequencyMHz();  
Serial.print("Frecuencia en MHz:")  
Serial.println(FRECUENCIA);  
}
```

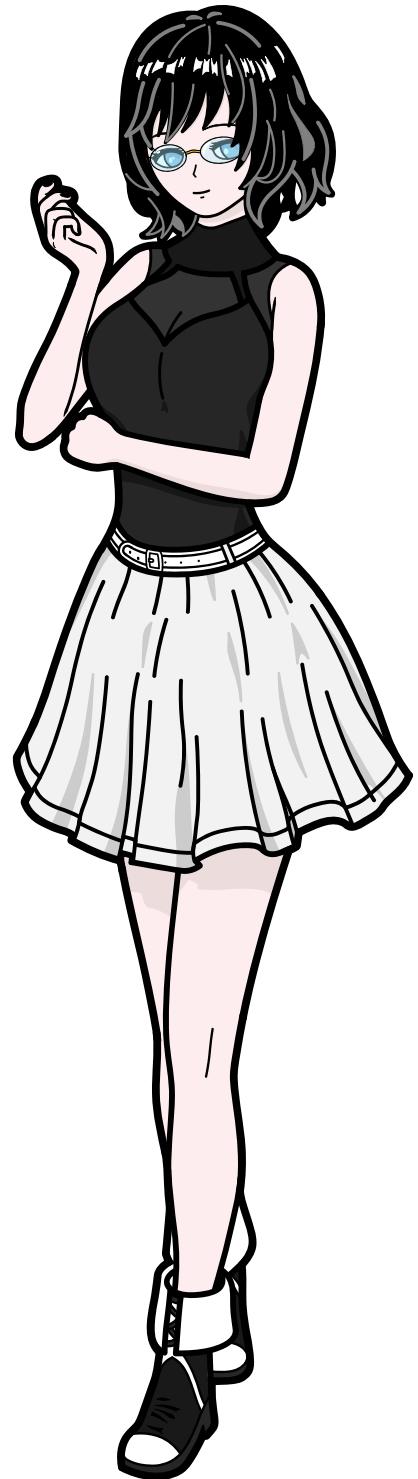
La ventaja de bajar la frecuencia del procesador (además de reducir el consumo energético), es que aún conservaremos la mayoría de las funciones del ESP32, aunque la ejecución de nuestro programa será más lenta. Con frecuencias menores a 80MHz, la comunicación Wifi o Bluetooth se volverá inestable o directamente no funcionará.

Cuando el procesador trabaja a frecuencias bajas, su temperatura también se verá reducida, evitando sobrecalentamientos y alargando su vida útil.

**CONSEJO:** configurar al ESP32 a su máximo de 240MHz se recomienda para programas multitarea o que procesen muchos datos, como en un servidor pequeño. En programas sencillos, la velocidad máxima del procesador es poco aprovechada.



# Capítulo 4: Wifi y servidores Web





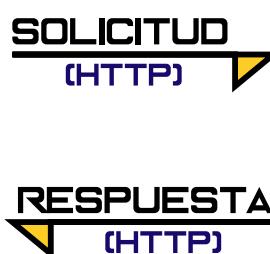
## El servidor web

Dentro de las redes de computadoras y dispositivos electrónicos, existen los servidores web, que se encargan de administrar la comunicación entre dichos dispositivos y proveerles de datos e información que necesiten. Un servidor web es un sistema con la capacidad de recibir y procesar peticiones de un cliente, para luego devolverle una respuesta. A este patrón de intercambio de información se denomina ‘solicitud-respuesta’ y es el método principal para compartir información en estas estructuras de cliente-servidor.

Cuando colocas una dirección web (URL) en el navegador, estás solicitando acceder a cierta información alojada en un servidor (en este caso una página web), mediante un protocolo de transferencia de hipertexto, el famoso HTTP. El servidor recibe la solicitud y regresa una respuesta, que es la página web solicitada:

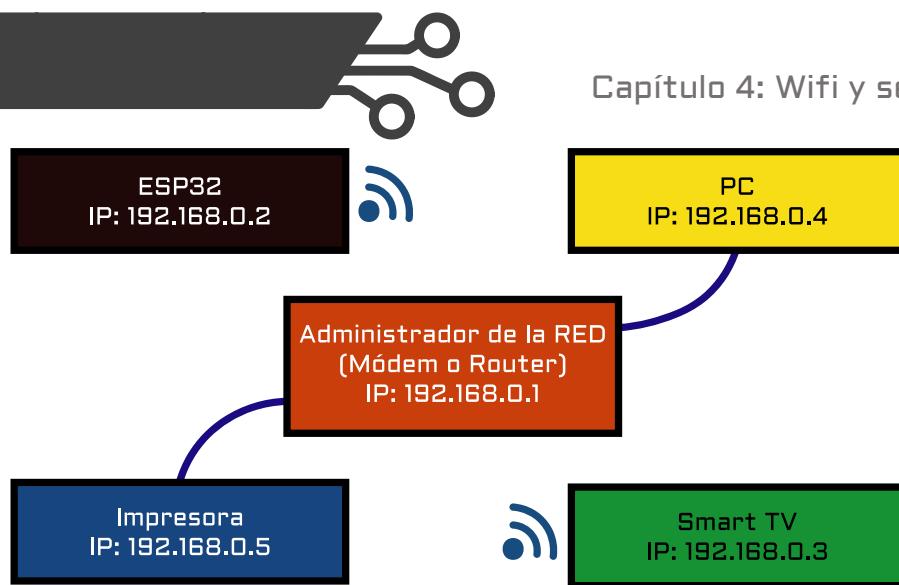


CLIENTE



El servidor anfitrión (Server host o Host) se encarga de ejecutar uno o más programas que permiten compartir recursos con los clientes (páginas web, bases de datos, etc.). El ESP32 puede actuar como un servidor anfitrión para recibir peticiones y enviar respuestas por HTTP.

Cuando se trabaja con una red de computadoras u otros dispositivos (que comparten recursos de forma local o por internet), cada uno de estos es identificado con una dirección IP. Las direcciones IP son identificadores numéricos que se asignan a cada dispositivo conectado a una red, ya sea de forma cableada o inalámbrica (Wifi). Generalmente, la dirección IP es asignada automáticamente por el dispositivo que administra la red (un módem o un router), que también cuenta con su propia dirección IP.



Cuando el ESP32 funciona como servidor Host, éste se conecta al router o al modem de tu red y el resto de dispositivos conectados le pueden hacer peticiones (a través de HTTP).

Antes de montar un servidor web con el ESP32, primero veamos algunos conceptos básicos de HTML y CSS para crear una página web.

## HTML y CSS básicos para crear páginas web

Las páginas web básicamente son documentos de texto que pueden contener elementos interactivos, como audio, video, programas embebidos, etc. El lenguaje marcado de hipertexto (HTML) es el lenguaje básico que permite crear páginas web. Las versiones más modernas de HTML permiten incrustar elementos interactivos más complejos y diseños de interfaz más atractivos, pero en su forma más simple, HTML puede ser un documento de texto sencillo. Un documento HTML contiene todo el código de la página web diseñada, que es interpretado por el navegador para poder mostrarla.

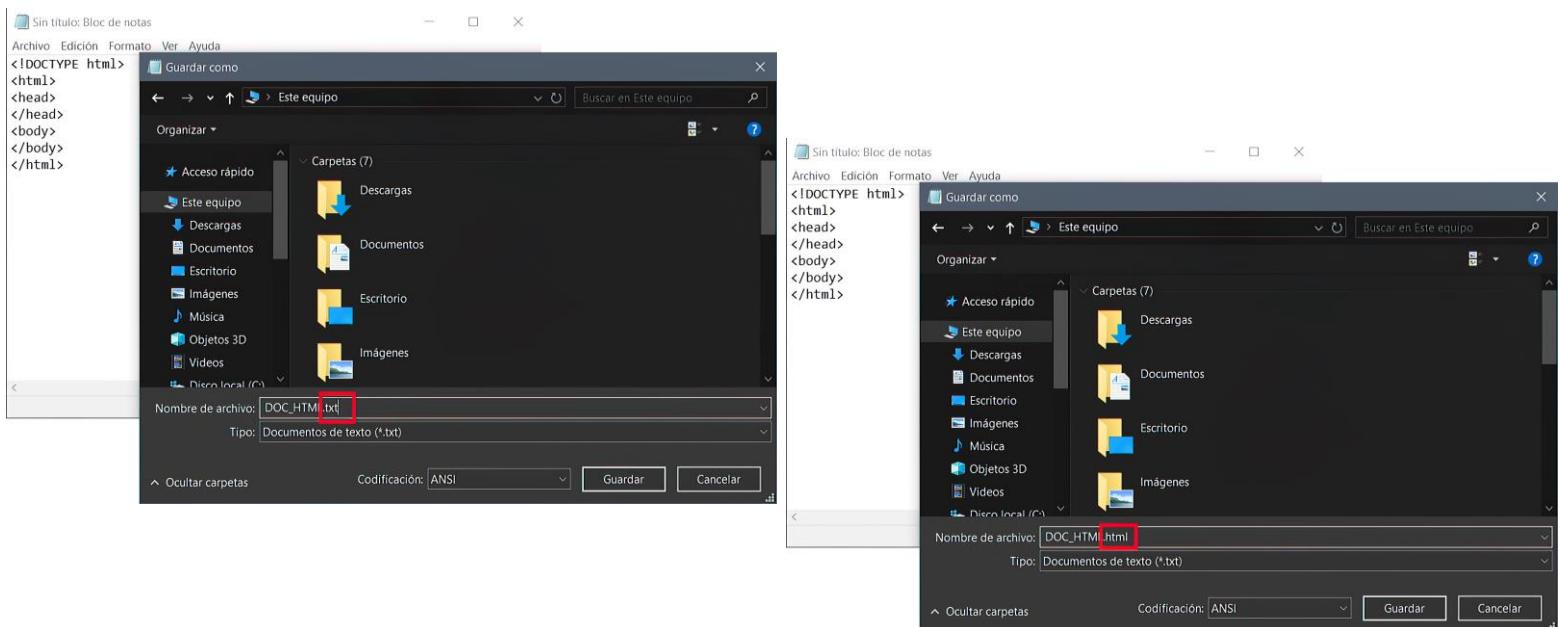
En los inicios de HTML, aparecieron las hojas de estilo en cascada (CSS), que podían integrarse a las páginas web para mejorar su aspecto visual y hacerlas más atractivas. Al igual que HTML, CSS tiene lenguaje de programación propio, pero se puede integrar dentro del mismo documento HTML. A pesar de que el HTML moderno permite diseñar páginas web más atractivas y complejas, CSS aún es muy utilizado para diseñar páginas web.

### El documento HTML

Podemos crear un documento HTML con cualquier procesador de texto, como el bloc de notas de Windows, e incluso existen sitios web que te permiten crear documentos HTML y ver el resultado de tu diseño en tiempo real. Por ahora usaremos la opción más sencilla: el bloc de notas.



Al guardar tu documento HTML, deberás cambiar la extensión .txt del archivo a .html, así podrás abrirlo con cualquier navegador web:



La estructura básica de un documento HTML es la siguiente:

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
</body>
</html>
<!DOCTYPE html>
```

donde la línea `<!DOCTYPE html>` le indica al navegador que se trata de un documento HTML. La página web se desarrolla entre las etiquetas `<html>` y `</html>`.

El documento HTML se divide en dos secciones principales: el encabezado y el cuerpo. El encabezado se coloca entre las etiquetas `<head>` y `</head>`, mientras que el cuerpo va entre las etiquetas `<body>` y `</body>`.

En el encabezado van las instrucciones que sirven para agregar ciertas características a la página web, como el título mostrado en el navegador o algunos estilos y fuentes de texto. En el cuerpo va el contenido general de la página web (texto, botones, tablas, etc.), cuyos elementos estarán organizados de acuerdo a la secuencia que les des, similar a un programa en Arduino.



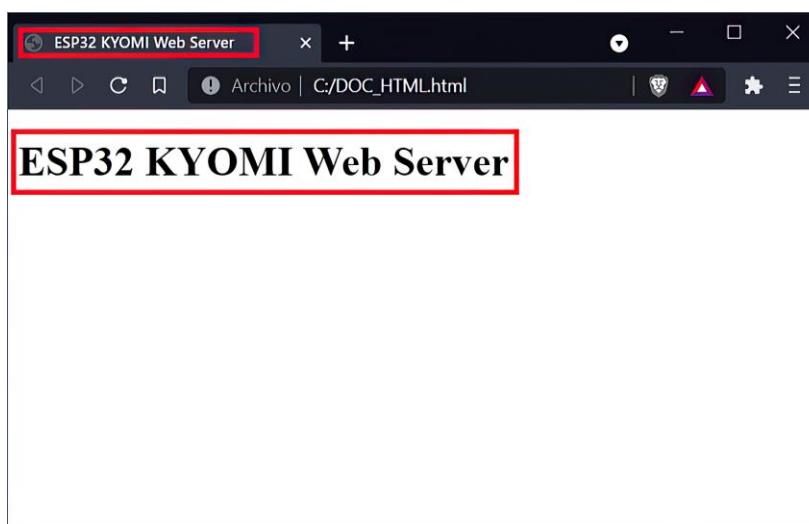
Hagamos un diseño sencillo. Para agregar el título de la página web (el que se muestra en la pestaña del navegador) lo haremos dentro del encabezado, entre las etiquetas `<title>` y `</title>`:

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 KYOMI Web Server</title>
</head>
<body>
</body>
</html>
```

Para añadir un título al cuerpo de la página web, usaremos las etiquetas de título grande `<h1>` y `</h1>`. Puedes agregar más títulos con distintos tamaños, por ejemplo, con `<h2>` y `</h2>` obtendrás un texto de título más pequeño que con `<h1>` y `</h1>`, mientras que con `<h3>` y `</h3>` el texto será más pequeño que con `<h2>` y `</h2>` y así sucesivamente:

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 KYOMI Web Server</title>
</head>
<body>
    <h1>ESP32 KYOMI Web Server</h1>
</body>
</html>
```

Guarda tu documento y ábrelo con el navegador web para ver el diseño de tu página (podrás seguir modificando el documento HTML si lo abres con el bloc de notas):



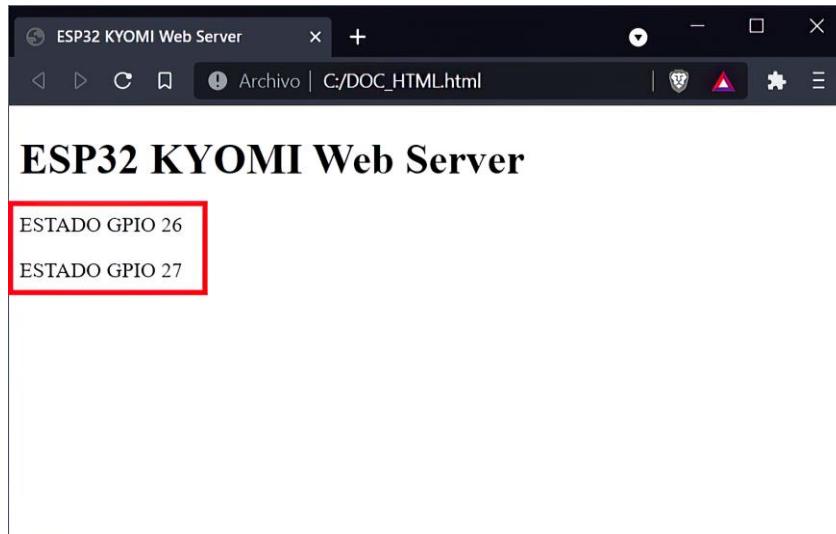


Seguiremos modificando este documento HTML para poder usarlo en los próximos ejercicios.

Podemos organizar mejor nuestra página web agregando párrafos al documento HTML, utilizando las etiquetas `<p>` y `</p>`. En este caso, agregaremos dos párrafos con un texto que indicará el estado de dos GPIO's del ESP32 (que luego controlaremos):

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 KYOMI Web Server</title>
</head>
<body>
    <h1>ESP32 KYOMI Web Server</h1>
    <p>ESTADO GPIO 26</p>
    <p>ESTADO GPIO 27</p>
</body>
</html>
```

Podrás ver los cambios en el navegador:



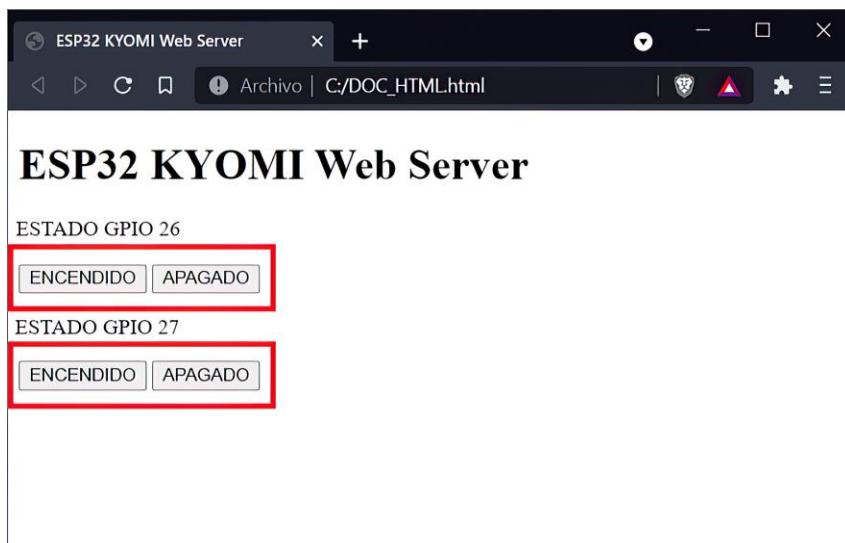
Pasemos a la parte interactiva de la página web. Para agregar botones a nuestra página, utilizamos las etiquetas `<button>` y `</button>`. Entre estas etiquetas colocamos el texto incrustado del botón. Usaremos un par de botones para cada GPIO a controlar y los organizaremos en párrafos, donde cada uno tendrá dos botones (uno de encendido y otro de apagado):

```
<!DOCTYPE html>
<html>
```



```
<head>
    <title>ESP32 KYOMI Web Server</title>
</head>
<body>
    <h1>ESP32 KYOMI Web Server</h1>
    <p>ESTADO GPIO 26</p>
    <p><button>ENCENDIDO</button> <button>APAGADO</button></p>
    <p>ESTADO GPIO 27</p>
    <p><button>ENCENDIDO</button> <button>APAGADO</button></p>
</body>
</html>
```

El resultado quedaría así:



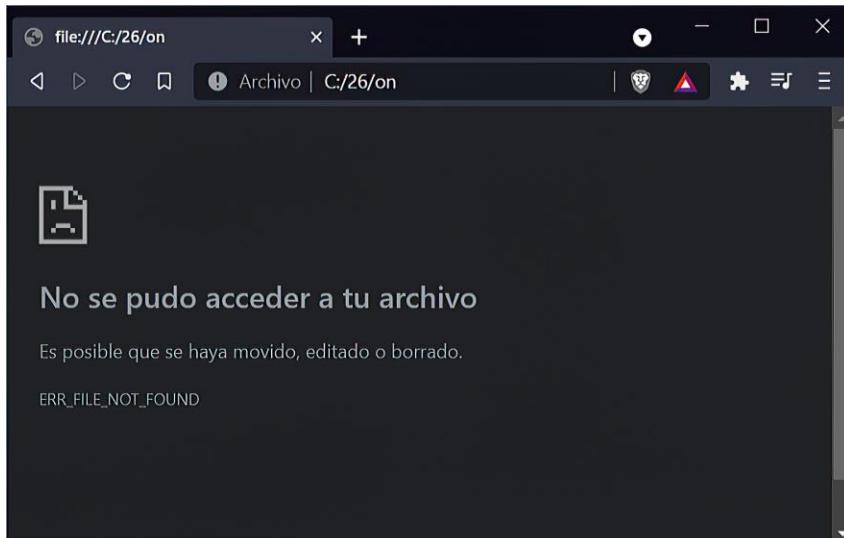
Por ahora los botones no harán nada. Para agregarles una función, necesitaremos hipervínculos de referencia. Estos hipervínculos o enlaces se agregan con las etiquetas `<a>` y `</a>`, entre estas etiquetas debemos colocar el elemento interactivo (los botones). Con el atributo `href` (dentro de `<a>`) se especifica el enlace web al que será redirigida la página al hacer clic en el botón, que técnicamente sería otra página web que contenga un subprograma o una función con una acción a ejecutar:

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 KYOMI Web Server</title>
</head>
<body>
```



```
<h1>ESP32 KYOMI Web Server</h1>
<p>ESTADO GPIO 26</p>
<p><a href='/26/on'><button>ENCENDIDO</button></a></p>
    <a href='/26/off'><button>APAGADO</button></a></p>
<p>ESTADO GPIO 27</p>
<p><a href='/27/on'><button>ENCENDIDO</button></a></p>
    <a href='/27/off'><button>APAGADO</button></a></p>
</body>
</html>
```

Los botones de encendido redirigirán a las subpáginas `/26/on` y `/27/on` (para los GPIO's 26 y 27), mientras que los botones de apagado redirigirán a las páginas `/26/off` y `/27/off`. Por el momento, los botones generarán el error “no se encontró el archivo” al hacerles clic, porque no hay un archivo HTML para esas subdirecciones web:



esto lo resolveremos más adelante en el código de Arduino, al crear las funciones para dichos botones cuando montemos el servidor web con el ESP32.

Es recomendable hacer uso de ‘clases’ para especificar uno o más nombres para ciertos elementos, lo cual permite identificarlos según su ‘clase’ (esto también será útil para los estilos CSS):

```
<elemento class='nombre de clase'>
```

Especificaremos la clase “BOTON1” para los botones de encendido y la clase “BOTON2” para los botones de apagado:

```
<!DOCTYPE html>
<html>
<head>
```



```
<title>ESP32 KYOMI Web Server</title>
</head>
<body>
    <h1>ESP32 KYOMI Web Server</h1>
    <p>ESTADO GPIO 26</p>
    <p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a></p>
        <a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p>
    <p>ESTADO GPIO 27</p>
    <p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a>
        <a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p>
</body>
</html>
```

La siguiente instrucción (sección de metadatos) la colocaremos dentro de la sección del encabezado, la cual mejorará la compatibilidad con ciertos navegadores web, configurando el escalado de la página:

```
<meta name='viewport' content='width=device-width, initial-scale=1'>
```

el atributo `viewport` indica que la página solo puede ocupar el área visible del navegador y `width=device-width` e `initial-scale=1` configuran la página al ancho del área visible con una escala de '1' o 100% (básicamente es el zoom a la página).

La siguiente instrucción también la colocaremos dentro de la sección del encabezado. Servirá para evitar que el navegador envíe solicitudes sobre el 'favicon', que es el pequeño ícono que se muestra en la pestaña del navegador (a un lado del nombre de la página web):

```
<link rel='icon' href='data:, '>
```

El favicon se puede personalizar, pero por ahora lo dejaremos vacío. Hasta este punto habremos terminado con la parte HTML de nuestra página web:

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 KYOMI Web Server</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='icon' href='data:, '>
</head>
<body>
    <h1>ESP32 KYOMI Web Server</h1>
    <p>ESTADO GPIO 26</p>
```



```
<p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a>
    <a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p>
<p>ESTADO GPIO 27</p>
<p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a>
    <a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p>
</body>
</html>
```

Ahora vamos a mejorar su estilo visual mediante CSS.

## Mejoras con CSS

Las hojas de estilo en cascada o CSS, pueden ser añadidas al mismo documento HTML o como un archivo separado que se referencia dentro del propio HTML. En este caso, agregaremos el código CSS dentro del documento HTML.

Todo el código CSS debe ir entre las etiquetas `<style>` y `</style>`, dentro de la sección del encabezado del documento HTML. CSS usa los llamados ‘selectores’, que apuntan a ciertos elementos del código HTML, los cuales podemos personalizar. Estos selectores cuentan con ciertas propiedades y valores, y tienen la siguiente estructura:

```
selector
{
  propiedad: valor;
}
```

Retomando el documento HTML, agregaremos algunos estilos de CSS a nuestra página. Primero configuraremos el selector `html` para cambiar la fuente de todos los textos a ‘Helvética’ (con `font-family`) y alinear los párrafos al centro (con `text-align`):

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 KYOMI Web Server</title>
  <meta name='viewport' content='width=device-width, initial-scale=1'>
  <link rel='icon' href='data:, '>
  <style>
    html
    {
      font-family: Helvetica;
      display: inline-block;
```



```
margin: 0px auto;
text-align: center;
}
</style>
</head>
<body>
<h1>ESP32 KYOMI Web Server</h1>
<p>ESTADO GPIO 26</p>
<p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a>
<a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p>
<p>ESTADO GPIO 27</p>
<p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a>
<a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p>
</body>
</html>
```

Guarda los cambios para ver el resultado:



También cambiaremos el estilo de los botones. En el siguiente selector (para los botones de encendido) definiremos algunas propiedades para personalizar el color de relleno (en RGB hexadecimal), los bordes (su grueso en pixeles, por ejemplo, 2px), el color del texto, el espaciado alrededor del botón, la decoración del texto, tamaño del texto, sus márgenes y la forma a la que cambiará el cursor al acercarlo al botón. Algunos sitios web te permiten personalizar botones y otros elementos CSS, para generarte automáticamente el código que va en el selector.



Observa que el nombre del selector es el mismo que el nombre que colocamos en las clases que creamos para los botones:

```
.BOTON1
{
background-color: #4CAF50;
border: none;
color: white;
padding: 16px 40px;
text-decoration: none;
font-size: 30px;
margin: 2px;
cursor: pointer;
}
```

En el caso de la clase *BOTON2* (para los botones de apagado), el estilo sería:

```
.BOTON2
{
background-color: #555555;
border: none;
color: white;
padding: 16px 40px;
text-decoration: none;
font-size: 30px;
margin: 2px;
cursor: pointer;
}
```

El HTML completo queda de la siguiente forma:

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 KYOMI Web Server</title>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='icon' href='data:, '>
    <style>
        html
        {
            font-family: Helvetica;
```



```
display: inline-block;
margin: 0px auto;
text-align: center;
}

.BOTON1
{
background-color: #4CAF50;
border: none;
color: white;
padding: 16px 40px;
text-decoration: none;
font-size: 30px;
margin: 2px;
cursor: pointer;
}

.BOTON2
{
background-color: #555555;
border: none;
color: white;
padding: 16px 40px;
text-decoration: none;
font-size: 30px;
margin: 2px;
cursor: pointer;
}
</style>
</head>

<body>
<h1>ESP32 KYOMI Web Server</h1>
<p>ESTADO GPIO 26</p>
<p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a>
<a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p>
```



```
<p>ESTADO GPIO 27</p>
<p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a>
    <a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p>
</body>
</html>
```

Guarda los cambios para ver el resultado:



Este documento HTML lo usaremos para montar un servidor web sencillo con el ESP32.

## Servidor Web básico con ESP32

El ESP32 generalmente se configura para actuar como servidor Host, y las peticiones de sus clientes pueden controlar el estado de los GPIO's o solicitar datos de algún sensor.

Cuando el ESP32 funciona como servidor web, se conecta al administrador de tu red (el módem o el router) junto con el resto de dispositivos (celulares, PC's, impresoras, etc.). Estos dispositivos podrán enviarle solicitudes al ESP32 a través de HTTP para que este les entregue una respuesta.

Con el siguiente ejercicio aprenderemos a montar un servidor web sencillo para controlar un par de GPIO's.

### Ejercicio 17

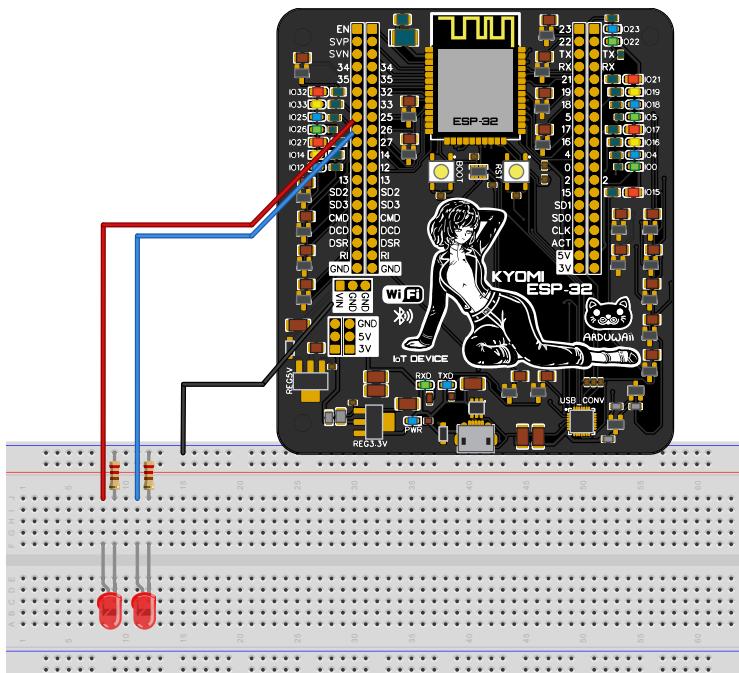
En este ejercicio controlaremos el estado digital (*HIGH* o *LOW*) del GPIO26 y 27 a través de la página web que diseñamos anteriormente. Podremos controlar los GPIO's desde el navegador web de cualquier dispositivo conectado a nuestra red.



**Materiales:**

- 2 LEDs (opcional)
- 2 resistencias de 220Ω (opcional)
- 3 jumpers o cables tipo Dupont (para los LEDs)
- plantilla de experimentos

El circuito es el siguiente:



Veamos el código. Primero debemos incluir la librería `WiFi.h` y declarar un par de variables `const char` que guardarán el nombre de la red Wifi a la que nos conectaremos y su contraseña (revisa ambos datos en tu router o módem). También configuraremos nuestro servidor web con el puerto 80 (`server(80)`). Aunque también podemos usar otros puertos disponibles, comúnmente se toma el 80:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);
```

Después vamos a declarar un par de variables de cadena de caracteres (`String`) y otra tipo `char` que nos servirán más adelante:

```
String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;
```



En otra variable tipo *String* vamos a guardar todo el código HTML de la página web que diseñamos previamente. El código debe quedar como una sola cadena de caracteres que podamos colocar entre “”:

```
String PAGINA_WEB="<!DOCTYPE html> <html> <head> <title>ESP32 KYOMI Web Server</title> <meta name='viewport' content='width=device-width, initialscale=1'> <link rel='icon' href='data:, '> <style> html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1 { background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 { background-color: #555555; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style> </head> <body> <h1>ESP32 KYOMI Web Server</h1> <p>ESTADO GPIO 26</p> <p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a> <a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p> <p>ESTADO GPIO 27</p> <p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a> <a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p> </body> </html>";
```

para darle este formato al código HTML, tendríamos que eliminar todos los saltos de línea del documento, pero existen sitios web que pueden convertir el código HTML a una sola cadena de caracteres, esto nos ahorrará mucho tiempo:

The screenshot shows a browser window titled "HTML Conversion for the Arduino IDE". The page contains instructions about escaping double quotes and using variables for serving webpages. It features a code editor with the original HTML code and a "Convert" button. Below the editor, a message states that the converted code should compile fine in the Arduino IDE. The final converted code is shown in a scrollable text area at the bottom.

This will escape any double quotes ("") and split it up so you can add in variables for each @ you place in the code. You can then use it as a String variable for serving webpages in boards like the ESP8266 or the Ethernet Shield.

```
<!DOCTYPE html>
<html>
<head>
<title>ESP32 KYOMI Web Server</title>
<meta name='viewport' content='width=device-width, initialscale=1'>
<link rel='icon' href='data:, '>

<style>
html {

```

Convert

The code below 'should' compile fine in the Arduino IDE.

```
String html ="<!DOCTYPE html> <html> <head> <title>ESP32 KYOMI Web Server</title> <meta name='viewport' content='width=device-width, initialscale=1'> <link rel='icon' href='data:, '> <style> html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1 { background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 { background-color: #555555; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style> </head> <body> <h1>ESP32 KYOMI Web Server</h1> <p>ESTADO GPIO 26</p> <p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a> <a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p> <p>ESTADO GPIO 27</p> <p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a> <a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p> </body> </html>";
```

Finalmente declaramos un par de variables para los GPIO26 y 27:

```
const int GPIO26 = 26;
const int GPIO27 = 27;
```



Dentro de `setup()` vamos a configurar la comunicación serial, los GPIO's como salidas y que estos arranquen en `LOW`:

```
Serial.begin(115200);
pinMode(GPIO26, OUTPUT);
pinMode(GPIO27, OUTPUT);
digitalWrite(GPIO26, LOW);
digitalWrite(GPIO27, LOW);
```

Luego crearemos una pequeña rutina para conectar al ESP32 a nuestra red. Primero mostraremos un mensaje en el Monitor Serie que indicará que el ESP32 se está conectando a la red que especificamos en `ssid`:

```
Serial.print("Conectando a:");
Serial.println(ssid);
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
}
```

el ESP32 intentará establecer la conexión mediante `WiFi.begin()`, con los valores que establecimos en `ssid` y `password`. El ciclo `while` simplemente imprime puntos en el Monitor Serie hasta que el estado de `WiFi.status()` sea igual a `WL_CONNECTED`, momento en el que la conexión será exitosa.

Para terminar, enviaremos un par de mensajes al Monitor Serie y también la dirección IP que asignó el módem o router a nuestro ESP32, usando `WiFi.LocalIP()`:

```
Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.LocalIP());
server.begin();
```

también iniciaremos el ESP32 en modo servidor con `server.begin()`.

Dentro de `loop()`, al inicio colocaremos la siguiente instrucción, que servirá para ‘escuchar’ las peticiones de todos los clientes que se conecten a nuestro servidor:

```
WiFiClient client = server.available();
```



Luego colocaremos un `if` cuya condición depende del valor de `client`, y ejecutará sus instrucciones siempre y cuando haya clientes conectados:

```
if(client)
{
    Serial.println("Cliente conectado");
    while(client.connected())
    {
        if(client.available())
        {
            CADENA_ENCABEZADO = client.read();
            Serial.write(CADENA_ENCABEZADO);
            ENCABEZADO += CADENA_ENCABEZADO;
            if(CADENA_ENCABEZADO == '\n')
            {
                if(PETICION_CLIENTE.Length() == 0)
                {
                    RESPUESTA_NAVEGADOR();
                    ACCIONES_HTML(ENCABEZADO);
                    client.println(PAGINA_WEB);
                    client.println();
                    break;
                }
                else {PETICION_CLIENTE = "";}
            }
            else if(CADENA_ENCABEZADO != '\r')
            {
                PETICION_CLIENTE += CADENA_ENCABEZADO;
            }
        }
        ENCABEZADO = "";
        client.stop();
        Serial.println("Cliente desconectado");
        Serial.println("");
    }
}
```



este *if* contiene la rutina más importante del servidor. Ahora veremos cómo funciona cada una de sus partes:

- Primero, si el valor de *client* nunca indica que hay clientes conectados, el servidor no hará nada. En caso contrario, si al menos un cliente se conecta, el *if* se ejecutará.
- Cuando se conecte un cliente, se enviará un mensaje al Monitor Serie que indicará lo ocurrido.
- Luego, el ciclo *while(client.connected())* se ejecutará mientras el cliente permanezca conectado a nuestro servidor.
- Dentro del *while*, el *if(client.available())* se ejecutará cuando el cliente (el navegador web) comience a enviar datos de petición HTTP a nuestro servidor. Con *CADENA\_ENCABEZADO = client.read()* vamos a guardar estos datos de petición y luego los imprimiremos en el Monitor Serie. Estos datos de petición se componen de varias líneas de texto que va recibiendo el servidor para formar un mensaje, por lo tanto, el ciclo *while* se ejecuta varias veces para recibir el mensaje de petición completo, pero hay que acumular todos estos mensajes en la variable *ENCABEZADO*, debido a que *CADENA\_ENCABEZADO* cambia su valor con cada línea de texto que recibe de la petición HTTP. El mensaje de petición completo que guardamos en *ENCABEZADO* lo usaremos después.

Cuando pruebes el programa, verás el mensaje de petición HTTP que envía el navegador cliente:

The screenshot shows a terminal window titled "COM". The text area contains the following data, with the first few lines highlighted by a red rectangle:

```
Conectado
Dirección IP:
192.168.0.105
Cliente conectado
GET / HTTP/1.1
Host: 192.168.0.105
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,in
Sec-GPC: 1
Accept-Encoding: gzip, deflate
Accept-Language: es-419,es;q=0.9
```

Below the highlighted text, the message continues:

```
Cliente desconectado
```

At the bottom of the window, there are several buttons and settings:

- Checkboxes for "Autoscroll" and "Mostrar marca temporal".
- A dropdown menu for "Nueva línea".
- A dropdown menu for "115200 baudio".
- A button labeled "Limpiar salida".

Peticiones de este tipo son las que estará enviando el navegador para recibir una respuesta de nuestro servidor.

- Cuando el cliente haya terminado de enviar la petición HTTP, hará un último salto de línea vacío en su mensaje de petición (*\n*). Es este salto de línea el que indica que el cliente ahora va esperar una respuesta del servidor.



```
COM3
Conectado
Dirección IP:
192.168.0.105
Cliente conectado
GET / HTTP/1.1
Host: 192.168.0.105
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/*
Sec-GPC: 1
Accept-Encoding: gzip, deflate
Accept-Language: es-419,es;q=0.9
[REDACTED]
Cliente desconectado
< >
 Autoscroll  Mostrar marca temporal Nueva línea 115200 baudio Limpiar salida
```

- El `if(CADENA_ENCABEZADO == '\n')` se ejecutará cuando se detecte el salto de línea en el mensaje de petición. Hace esta comprobación cada vez que cambia el texto guardado en `CADENA_ENCABEZADO`. Dentro de este `if`, vamos a colocar las respuestas que enviará el servidor al cliente.
- El `if(PETICION_CLIENTE.Length() == 0)` se ejecutará cuando el texto que esté guardado en `PETICION_CLIENTE` tenga longitud 0, es decir, cuando se detecte el salto de línea vacío que mencionamos anteriormente. La variable `PETICION_CLIENTE` también guardará el texto que se encuentre en `CADENA_ENCABEZADO`.

Digamos que aún no ocurre el salto de línea vacío del mensaje de petición, entonces `if(PETICION_CLIENTE.Length() == 0)` no se cumple y se ejecutaría `else{PETICION_CLIENTE = "";}`  en su lugar (para limpiar a `PETICION_CLIENTE`). Con `PETICION_CLIENTE` vacío, la ejecución va a continuar hasta `else if(CADENA_ENCABEZADO != '\r')`, que se va a ejecutar siempre que el valor de `CADENA_ENCABEZADO` sea distinto del carácter de retorno de carro (`\r`), que es el salto de línea que ocurre cuando presionamos la tecla Intro, por ejemplo. Cuando se ejecute este `else if`, `PETICION_CLIENTE` va a tomar el valor actual que tenga `CADENA_ENCABEZADO`. En algún momento, `CADENA_ENCABEZADO` tendrá el valor del salto de línea vacío que esperábamos y `PETICION_CLIENTE` también tendrá el mismo valor.

**NOTA:** El condicional `else if` también proporciona una ruta alternativa al igual que el `else` común, con la diferencia que `else if` también debe cumplir con una condición dada.

- Con el salto de línea guardado en `PETICION_CLIENTE`, el `if(PETICION_CLIENTE.Length() == 0)` se ejecutará. Primero correrá la subrutina `RESPUESTA_NAVEGADOR()`, que le enviará una respuesta HTTP genérica al cliente. La subrutina `ACCIONES_HTML(ENCABEZADO)` es la que



controlará a los GPIO's dependiendo de la petición que se guardó en *ENCABEZADO*, de hecho, el valor de *ENCABEZADO* lo compartiremos con esta subrutina. Cuando terminen las subrutinas, y como parte de la respuesta de nuestro servidor, enviaremos el documento HTML al cliente para que despliegue la página web, usando *client.println(PAGINA\_WEB)* (recuerda que en *PAGINA\_WEB* tenemos guardado el código HTML). Finalmente, para cerrar la respuesta del servidor, también debemos enviar un salto de línea vacío al cliente, usando *client.println()*, así sabrá el navegador web que la respuesta ha terminado (similar a su petición HTTP). La instrucción *break* la usamos para terminar directamente el ciclo *while*.

- Con el *while* finalizado, usamos *ENCABEZADO = ""* para vaciarlo y el cliente se podrá desconectar (una vez que le entreguemos una respuesta), para esto se utiliza *client.stop()*. Finalmente enviamos un par de mensajes al Monitor Serie para indicar lo sucedido.

Toda la rutina anterior se volverá a ejecutar si se detectan nuevos clientes o nuevas peticiones. Ahora veamos cómo trabajan el resto de las subrutinas.

La subrutina *RESPUESTA\_NAVEGADOR* envía una mensaje de respuesta genérico al navegador cliente, ya que, cuando el cliente haga la petición para obtener alguno de los hipervínculos asignados a los botones de nuestro documento HTML, no habrá otro documento web para devolver, es por eso que siempre se devuelve una respuesta HTTP genérica más el documento de la página web principal (enviado por *client.println(PAGINA\_WEB)*):

```
void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}
```

la instrucción *WiFiClient client = server.available()* la volvemos a colocar para poder usar las funciones *client.println()*.

La subrutina *ACCIONES\_HTML* se encarga de cambiar el estado de los GPIO's 26 y 27 dependiendo de la petición que fue guardada en *ENCABEZADO*. Esta subrutina toma el valor de *ENCABEZADO* que fue compartido en *Loop()*, que luego es guardado en una variable local que también se llama *ENCABEZADO*.



```
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /26/on")>=0)
    {
        Serial.println("GPIO 26 ENCENDIDO");
        digitalWrite(GPIO26, HIGH);
    }
    else if(ENCABEZADO.indexOf("GET /26/off")>=0)
    {
        Serial.println("GPIO 26 APAGADO");
        digitalWrite(GPIO26, LOW);
    }
    else if(ENCABEZADO.indexOf("GET /27/on")>=0)
    {
        Serial.println("GPIO 27 ENCENDIDO");
        digitalWrite(GPIO27, HIGH);
    }
    else if(ENCABEZADO.indexOf("GET /27/off")>=0)
    {
        Serial.println("GPIO 27 APAGADO");
        digitalWrite(GPIO27, LOW);
    }
}
```

cuando pruebes el programa y presiones uno de los botones en la página web, la petición HTTP va a solicitar el hipervínculo que asignamos a dicho botón mediante un método **GET**, que va aparecer en el mensaje de petición:

The screenshot shows a terminal window titled "COM3". The window displays an incoming HTTP request from a client connected to port COM3. The request is highlighted with a red box. The request details are as follows:

```
Clientes conectado
GET /26/on HTTP/1.1
Host: 192.168.0.105
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
Sec-GPC: 1
Referer: http://192.168.0.105/
Accept-Encoding: gzip, deflate
Accept-Language: es-419,es;q=0.9

GPIO 26 ENCENDIDO
Cliente desconectado
```

At the bottom of the terminal window, there are several status indicators and settings: "Autoscroll" (checked), "Mostrar marca temporal" (unchecked), "Nueva línea" (dropdown menu), "115200 baudio" (dropdown menu), and "Limpiar salida".



en este caso, la petición HTTP solicita el hipervínculo `/26/on` a través de `GET`. Suponiendo que `RESPUESTA_NAVEGADOR` ya le envío al cliente parte de la respuesta genérica, en el mensaje de petición guardado en `ENCABEZADO` buscaremos `GET /26/on`. En el `if(ENCABEZADO.indexOf("GET /26/on")>=0)`, la condición lo que hace es comprobar que el índice de caracteres que contiene a `GET /26/on` sea mayor o igual a 0, es decir, que exista este mensaje dentro de `ENCABEZADO`. Si se cumple la condición, va a cambiar el estado del GPIO26 a `HIGH` y enviará un mensaje al Monitor Serie.

Lo mismo ocurre con el resto de `else if`, solo que buscarán los otros hipervínculos en el mensaje de petición del cliente, que cambiará cuando se presionen los otros botones.

Esta es la estructura básica de un servidor en el ESP32. Gran parte de los cambios que realicemos (al montar otro tipo de servidores), los haremos en la subrutina `ACCIONES_HTML`, ya que en `RESPUESTA_NAVEGADOR()` y la rutina de `Loop()` no hay mucho que cambiar.

El código completo queda de la siguiente forma:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;

String PAGINA_WEB = "<!DOCTYPE html> <html> <head> <title>ESP32 KYOMI Web Server</title> <meta name='viewport' content='width=device-width, initialscale=1'> <link rel='icon' href='data:, '> <style> html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1 { background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 { background-color: #555555; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style> </head> <body> <h1>ESP32 KYOMI Web Server</h1> <p>ESTADO GPIO 26</p> <p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a> <a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p> <p>ESTADO GPIO 27</p> <p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a> <a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p> </body> </html>";

const int GPIO26 = 26;
const int GPIO27 = 27;
```



```
void setup()
{
    Serial.begin(115200);
    pinMode(GPIO26, OUTPUT);
    pinMode(GPIO27, OUTPUT);
    digitalWrite(GPIO26, LOW);
    digitalWrite(GPIO27, LOW);

    Serial.print("Conectando a:");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }

    Serial.println("");
    Serial.println("Conectado");
    Serial.println("Dirección IP: ");
    Serial.println(WiFi.localIP());
    server.begin();
}

void Loop()
{
    WiFiClient client = server.available();
    if(client)
    {
        Serial.println("Cliente conectado");
        while(client.connected())
        {
            if(client.available())
            {
                CADENA_ENCABEZADO = client.read();
                Serial.write(CADENA_ENCABEZADO);
            }
        }
    }
}
```



```
ENCABEZADO += CADENA_ENCABEZADO;
if(CADENA_ENCABEZADO == '\n')
{
    if(PETICION_CLIENTE.length() == 0)
    {
        RESPUESTA_NAVEGADOR();
        ACCIONES_HTML(ENCABEZADO);
        client.println(PAGINA_WEB);
        client.println();
        break;
    }
    else {PETICION_CLIENTE = "";}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}
}

ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}

void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}
```



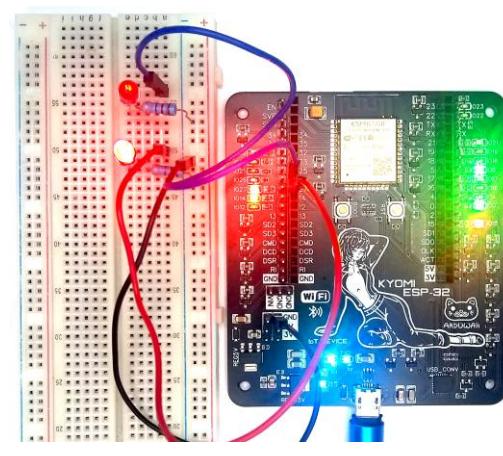
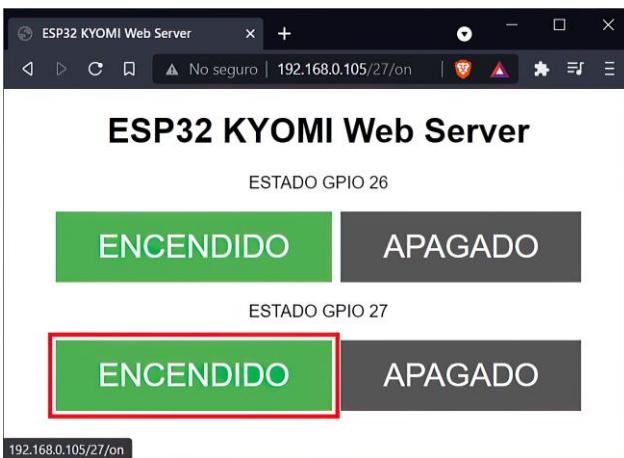
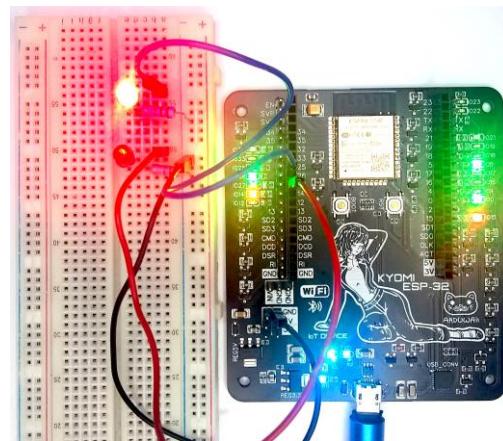
```
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /26/on")>=0)
    {
        Serial.println("GPIO 26 ENCENDIDO");
        digitalWrite(GPIO26, HIGH);
    }
    else if(ENCABEZADO.indexOf("GET /26/off")>=0)
    {
        Serial.println("GPIO 26 APAGADO");
        digitalWrite(GPIO26, LOW);
    }
    else if(ENCABEZADO.indexOf("GET /27/on")>=0)
    {
        Serial.println("GPIO 27 ENCENDIDO");
        digitalWrite(GPIO27, HIGH);
    }
    else if(ENCABEZADO.indexOf("GET /27/off")>=0)
    {
        Serial.println("GPIO 27 APAGADO");
        digitalWrite(GPIO27, LOW);
    }
}
```

Prueba el programa. Para acceder a la página web del servidor, copia la dirección IP que fue asignada a tu ESP32 y pégala en el navegador:

The screenshot displays two windows. On the left is a terminal window titled 'COM3' showing the text 'Conectado' and 'Dirección IP: 192.168.0.105'. On the right is a web browser window titled 'ESP32 KYOMI Web Server' with the URL '192.168.0.105'. The page content includes the text 'ESTADO GPIO 26' with 'ENCENDIDO' and 'APAGADO' buttons, and 'ESTADO GPIO 27' with 'ENCENDIDO' and 'APAGADO' buttons.



Cuando presiones los botones de la página, los LEDs de los GPIO's deberán encenderse o apagarse:



## Manejando HTML desde el IDE de Arduino

Hemos visto que `client.println()` nos permite enviar las respuestas HTTP que el navegador espera (para luego interpretarlas), incluyendo a la página web. Al separar el documento HTML en partes, podemos manejar ciertos elementos del mismo para hacer que aparezcan o no en la página web, cuando la enviamos al cliente como respuesta.

Las variables o textos que enviamos a través de `client.println()` deberán ser cadenas de caracteres tipo `String`.

### Ejercicio 18

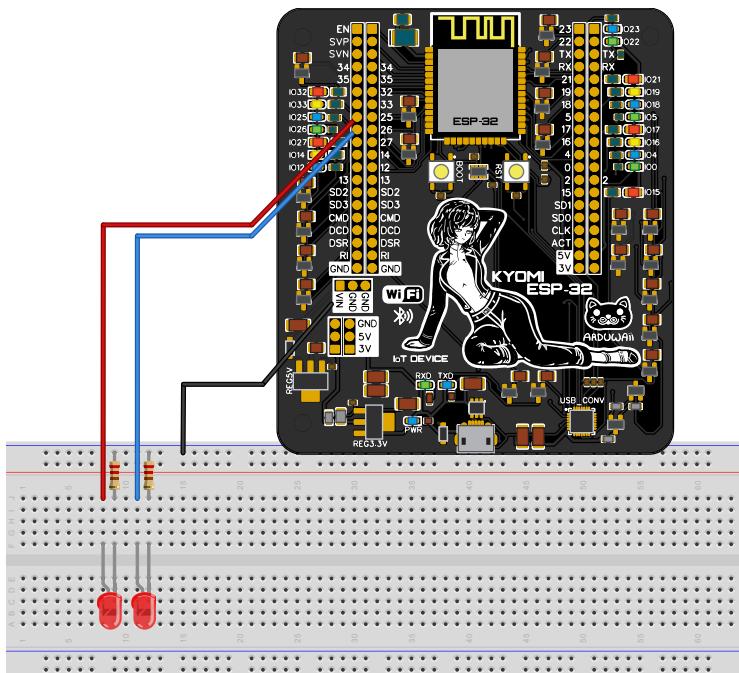
Retomando el ejercicio anterior, vamos a modificar algunas partes del mismo para hacer que desaparezcan los botones de encendido o apagado según el estado actual de los GPIO's.



**Materiales:**

- 2 LEDs (opcional)
- 2 resistencias de 220Ω (opcional)
- 3 jumpers o cables tipo Dupont (para los LEDs)
- plantilla de experimentos

El circuito es el siguiente:



Veamos el código. Las primeras variables las dejaremos sin cambios:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;
```

Después agregaremos un par de variables `String` que nos ayudarán más adelante:

```
String ESTADO_GPIO26="off",ESTADO_GPIO27="off";
```



Para poder controlar la aparición de los botones, tendremos que dividir nuestro documento HTML en 5 secciones diferentes:

1. En la primera sección, tendremos todas las instrucciones HTML hasta el texto de título `<p>ESTADO GPIO 26</p>`:

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 KYOMI Web Server</title>
    <meta name='viewport' content='width=device-width, initialscale=1'>
    <link rel='icon' href='data:, '>

    <style>
        html
        {
            font-family: Helvetica;
            display: inline-block;
            margin: 0px auto;
            text-align: center;
        }
        .BOTON1
        {
            background-color: #4CAF50;
            border: none;
            color: white;
            padding: 16px 40px;
            text-decoration: none;
            font-size: 30px;
            margin: 2px;
            cursor: pointer;
        }
        .BOTON2
        {
            background-color: #555555;
            border: none;
            color: white;
            padding: 16px 40px;
            text-decoration: none;
            font-size: 30px;
            margin: 2px;
            cursor: pointer;
        }
    </style>

</head>
<body>
    <h1>ESP32 KYOMI Web Server</h1>
    <p>ESTADO GPIO 26</p>
```

2. En la segunda sección irían las instrucciones de los botones para el GPIO26:

```
<p><a href='/26/on'><button class='BOTON1'>ENCENDIDO</button></a>
<a href='/26/off'><button class='BOTON2'>APAGADO</button></a></p>
```

pero la dejaremos vacía por ahora.



3. La tercera sección sólo va a contener el texto de título del GPIO27:

```
<p>ESTADO GPIO 27</p>
```

4. En la cuarta sección ocurre algo similar a la segunda, debería contener las instrucciones de los botones para el GPIO27:

```
<p><a href='/27/on'><button class='BOTON1'>ENCENDIDO</button></a><br><a href='/27/off'><button class='BOTON2'>APAGADO</button></a></p>
```

pero igualmente la dejaremos vacía.

5. En la quinta sección colocaremos lo que resta del documento HTML:

```
</body>
</html>
```

Con el documento HTML separado, creamos unas variables *String* para guardar cada sección:

```
String PAGINA_WEB_SECCION_1="<!DOCTYPE html> <html> <head> <title>ESP32 KYOMI Web
Server</title> <meta name='viewport' content='width=device-width,
initialscale=1'> <link rel='icon' href='data:,> <style> html { font-family:
Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1
{ background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2
{ background-color: #555555; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style>
</head> <body> <h1>ESP32 KYOMI Web Server</h1> <p>ESTADO GPIO 26</p>";
```

```
String PAGINA_WEB_SECCION_2="";
```

```
String PAGINA_WEB_SECCION_3="<p>ESTADO GPIO 27</p>";
```

```
String PAGINA_WEB_SECCION_4="";
```

```
String PAGINA_WEB_SECCION_5="</body> </html>";
```

El contenido de la sección 2 y 4 es el que vamos a modificar según las peticiones del cliente y el estado de los GPIO's. El documento HTML tuvo que ser dividido por que los mensajes de la respuesta HTTP se envían uno tras otro. Si el documento HTML entero se envía a través de una sola variable, no podremos modificar ninguna de sus partes. Es por eso que hay que separarlo, para modificar las partes de interés, pero siguiendo su secuencia original, para que el navegador cliente sea capaz de interpretar la respuesta HTTP entera.

Finalmente declaramos las variables para el GPIO26 y 27:

```
const int GPIO26 = 26;
const int GPIO27 = 27;
```

En *setup()* no será necesario hacer cambios:

```
Serial.begin(115200);
pinMode(GPIO26, OUTPUT);
pinMode(GPIO27, OUTPUT);
digitalWrite(GPIO26, LOW);
```



```
digitalWrite(GPIO27, LOW);
Serial.print("Conectando a:");
Serial.println(ssid);
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
}
Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.localIP());
server.begin();
```

Dentro de `loop()` mantendremos todo igual, excepto algunas instrucciones de `if(PETICION_CLIENTE.length() == 0)`:

```
WiFiClient client = server.available();
if(client)
{
    Serial.println("Cliente conectado");
    while(client.connected())
    {
        if(client.available())
        {
            CADENA_ENCABEZADO = client.read();
            Serial.write(CADENA_ENCABEZADO);
            ENCABEZADO += CADENA_ENCABEZADO;
            if(CADENA_ENCABEZADO == '\n')
            {
                if(PETICION_CLIENTE.length() == 0)
                {
                    RESPUESTA_NAVEGADOR();
                    ACCIONES_HTML(ENCABEZADO);
                    if(ESTADO_GPIO26=="off")
                    {
                        PAGINA_WEB_SECCION_2=<p><a href='/26/on'><button
class='BOTON1'>ENCENDER</button></a></p>";
                }
            }
        }
    }
}
```

`href='/26/on'><button`



```
        }

        else if(ESTADO_GPIO26=="on")
        {
            PAGINA_WEB_SECCION_2=<p><a href='/26/off'><button class='BOTON2'>APAGAR</button></a></p>";
        }

        if(ESTADO_GPIO27=="off")
        {
            PAGINA_WEB_SECCION_4=<p><a href='/27/on'><button class='BOTON1'>ENCENDER</button></a></p>";
        }

        else if(ESTADO_GPIO27=="on")
        {
            PAGINA_WEB_SECCION_4=<p><a href='/27/off'><button class='BOTON2'>APAGAR</button></a></p>";
        }

        client.println(PAGINA_WEB_SECCION_1);
        client.println(PAGINA_WEB_SECCION_2);
        client.println(PAGINA_WEB_SECCION_3);
        client.println(PAGINA_WEB_SECCION_4);
        client.println(PAGINA_WEB_SECCION_5);
        client.println();
        break;
    }

    else {PETICION_CLIENTE = "";}
}

else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}

ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}
```



Dentro del `if(PETICION_CLIENTE.Length() == 0)` ahora tenemos otros 4 `if`, que controlan a los botones de la página web. Las variables `ESTADO_GPIO26` y `ESTADO_GPIO27` están inicializadas en `off`, entonces el `if(ESTADO_GPIO26=="off")` y el `if(ESTADO_GPIO27=="off")` se ejecutarán. Por ejemplo, cuando `if(ESTADO_GPIO26=="off")` se cumple, el GPIO26 está en `LOW` y la variable `PAGINA_WEB_SECCION_2` tendrá las siguientes instrucciones HTML:

```
<p><a href='/26/on'><button class='BOTON1'>ENCENDER</button></a></p>
```

que habilitan el botón “Encender” para poder cambiar el estado del GPIO a `HIGH`. Algo similar ocurre en `if(ESTADO_GPIO27=="off")`.

Caso contrario, si `ESTADO_GPIO26` o `ESTADO_GPIO27` cambian a `on` (los GPIO's estarían en `HIGH`), `PAGINA_WEB_SECCION_2` o `PAGINA_WEB_SECCION_4` tendrían las instrucciones HTML para habilitar los botones de “Apagar” y así poder cambiar el estado de los GPIO's a `LOW` (esto ocurre en los `else if`). Los valores de `ESTADO_GPIO26` y `ESTADO_GPIO27` cambian en la subrutina `ACCIONES_HTML`.

Usaremos `client.println()` para enviar el contenido de las variables `PAGINA_WEB_SECCION_?` (en orden) y que el navegador las reciba en el mensaje de respuesta, para que las interprete como una página web completa.

La subrutina `RESPUESTA_NAVEGADOR` se mantiene sin cambios.

```
void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}
```

A la subrutina `ACCIONES_HTML` sólo hay que agregarle los cambios para `ESTADO_GPIO26` y `ESTADO_GPIO27` según el estado de los GPIO's. Por ejemplo, en `if(ENCABEZADO.indexOf("GET /26/on")>=0)`, el GPIO26 cambia a `HIGH` y `ESTADO_GPIO26` cambiará su valor a `on` (para habilitar el botón “Apagar”).

```
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /26/on")>=0)
    {
        Serial.println("GPIO 26 ENCENDIDO");
        digitalWrite(GPIO26, HIGH);
        ESTADO_GPIO26="on";
    }
}
```



```
}

else if(ENCABEZADO.indexOf("GET /26/off")>=0)
{
    Serial.println("GPIO 26 APAGADO");
    digitalWrite(GPIO26, LOW);
    ESTADO_GPIO26="off";
}

else if(ENCABEZADO.indexOf("GET /27/on")>=0)
{
    Serial.println("GPIO 27 ENCENDIDO");
    digitalWrite(GPIO27, HIGH);
    ESTADO_GPIO27="on";
}

else if(ENCABEZADO.indexOf("GET /27/off")>=0)
{
    Serial.println("GPIO 27 APAGADO");
    digitalWrite(GPIO27, LOW);
    ESTADO_GPIO27="off";
}

}
```

El código completo queda de la siguiente forma:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;

String ESTADO_GPIO26="off",ESTADO_GPIO27="off";

String PAGINA_WEB_SECCION_1="<!DOCTYPE html> <html> <head> <title>ESP32 KYOMI Web Server</title> <meta name='viewport' content='width=device-width, initialscale=1'> <link rel='icon' href='data:,> <style> html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1 { background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 { background-color: #555555; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style> </head> <body> <h1>ESP32 KYOMI Web Server</h1> <p>ESTADO GPIO 26</p>";
```



```
String PAGINA_WEB_SECCION_2="";
String PAGINA_WEB_SECCION_3=<p>ESTADO GPIO 27</p>;
String PAGINA_WEB_SECCION_4="";
String PAGINA_WEB_SECCION_5=</body> </html>;

const int GPIO26 = 26;
const int GPIO27 = 27;

void setup()
{
    Serial.begin(115200);
    pinMode(GPIO26, OUTPUT);
    pinMode(GPIO27, OUTPUT);
    digitalWrite(GPIO26, LOW);
    digitalWrite(GPIO27, LOW);

    Serial.print("Conectando a:");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("Conectado");
    Serial.println("Dirección IP: ");
    Serial.println(WiFi.localIP());
    server.begin();
}

void Loop()
{
    WiFiClient client = server.available();
    if(client)
    {
```



```
Serial.println("Cliente conectado");
while(client.connected())
{
    if(client.available())
    {
        CADENA_ENCABEZADO = client.read();
        Serial.write(CADENA_ENCABEZADO);
        ENCABEZADO += CADENA_ENCABEZADO;
        if(PETICION_CLIENTE.Length() == 0)
        {
            RESPUESTA_NAVEGADOR();
            ACCIONES_HTML(ENCABEZADO);
            if(ESTADO_GPIO26=="off")
            {
                PAGINA_WEB_SECCION_2=<p><a href='/26/on'><button
class='BOTON1'>ENCENDER</button></a></p>";
            }
            else if(ESTADO_GPIO26=="on")
            {
                PAGINA_WEB_SECCION_2=<p><a href='/26/off'><button
class='BOTON2'>APAGAR</button></a></p>";
            }
            if(ESTADO_GPIO27=="off")
            {
                PAGINA_WEB_SECCION_4=<p><a href='/27/on'><button
class='BOTON1'>ENCENDER</button></a></p>";
            }
            else if(ESTADO_GPIO27=="on")
            {
                PAGINA_WEB_SECCION_4=<p><a href='/27/off'><button
class='BOTON2'>APAGAR</button></a></p>";
            }
            client.println(PAGINA_WEB_SECCION_1);
            client.println(PAGINA_WEB_SECCION_2);
            client.println(PAGINA_WEB_SECCION_3);
            client.println(PAGINA_WEB_SECCION_4);
            client.println(PAGINA_WEB_SECCION_5);
            client.println();
        }
    }
}
```



```
        break;
    }
    else {PETICION_CLIENTE = "";}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}
}
}
ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}

void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}

void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /26/on")>=0)
    {
        Serial.println("GPIO 26 ENCENDIDO");
        digitalWrite(GPIO26, HIGH);
        ESTADO_GPIO26="on";
    }
    else if(ENCABEZADO.indexOf("GET /26/off")>=0)
    {
```

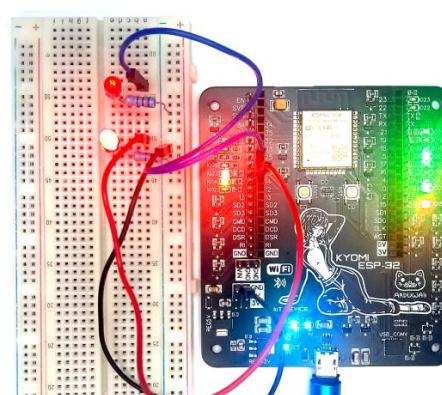
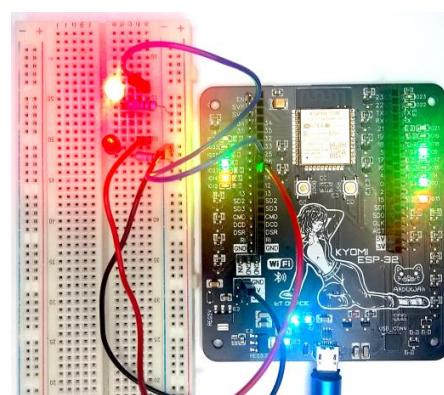


```
Serial.println("GPIO 26 APAGADO");
digitalWrite(GPIO26, LOW);
ESTADO_GPIO26="off";
}

else if(ENCABEZADO.indexOf("GET /27/on")>=0)
{
    Serial.println("GPIO 27 ENCENDIDO");
    digitalWrite(GPIO27, HIGH);
    ESTADO_GPIO27="on";
}

else if(ENCABEZADO.indexOf("GET /27/off")>=0)
{
    Serial.println("GPIO 27 APAGADO");
    digitalWrite(GPIO27, LOW);
    ESTADO_GPIO27="off";
}
}
```

Ahora prueba el programa. Verás que sólo aparecerá un botón para cada LED de los GPIO's, que cambiará para encenderlos o apagarlos según su estado actual:

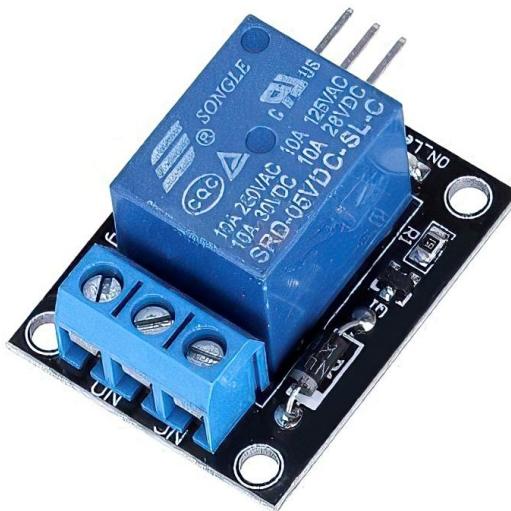




**NOTA:** En cualquier documento HTML que realices, evita usar comillas dobles ("") en cualquiera de las instrucciones, ya que esto causa conflictos al momento de pegar el código en una variable *String*. Usa comillas simples (') en estos casos.

## Aplicación práctica: control de relevadores

Ahora hagamos un ejercicio de aplicación. Partiremos del ejercicio 18 para crear un control sencillo para un módulo relevador, que podremos usar para encender o apagar cargas grandes, como un foco o un motor.



Módulo relevador de un solo canal.

Si has usado este tipo de módulos con algún Arduino clásico, entonces su funcionamiento no te será desconocido.

El relevador es un componente electromecánico capaz de abrir o cerrar un interruptor interno a través de un mecanismo, que es accionado por una bobina que trabaja con bajo voltaje:

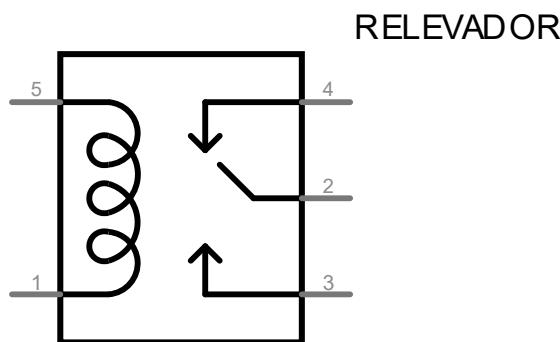


Diagrama eléctrico de un relevador común.



Eléctricamente, la bobina que acciona al interruptor y el propio interruptor están aislados entre sí, por lo que podemos usar el relevador para controlar el encendido o apagado de una carga que trabaje con alto voltaje, aislándola de la etapa de bajo voltaje que activa a la bobina del relevador (en este caso el ESP32).

Comúnmente, el interruptor del relevador tiene tres terminales: un polo común, la terminal normalmente abierta o NA (NO en inglés) y la terminal normalmente cerrada o NC.

Entre la terminal normalmente abierta y el polo común, el interruptor está abierto hasta que se acciona el relevador, mientras que entre la terminal normalmente cerrada y el polo común, el interruptor está cerrado hasta que se acciona el relevador.

Ahora veamos el ejercicio de aplicación.

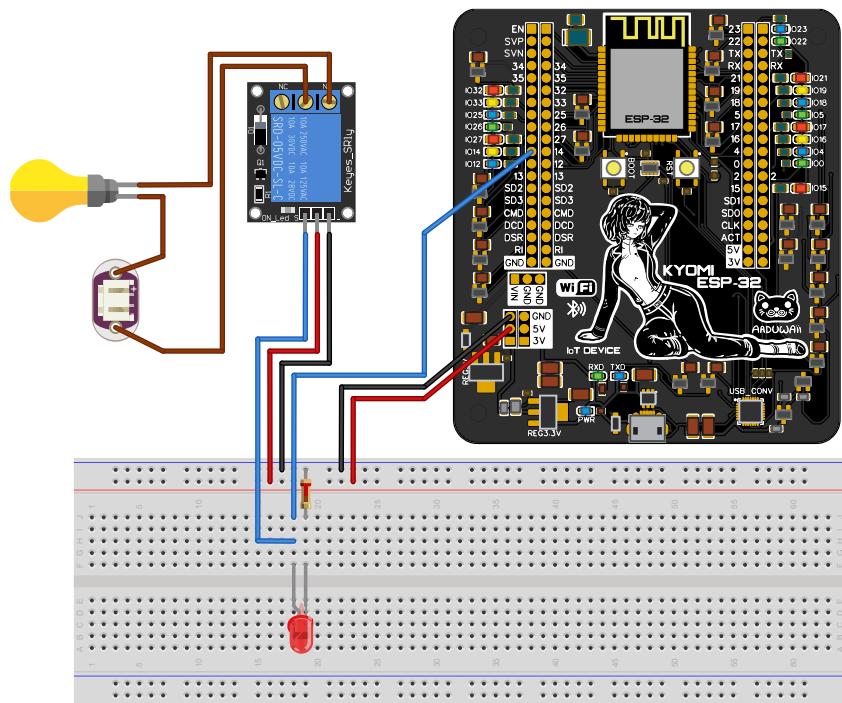
## Ejercicio 19

En este ejercicio controlaremos el encendido y apagado de un foco incandescente de 120V. Crearemos un servidor web sencillo que controle un GPIO, que a su vez controlará un módulo relevador. Usaremos la terminal normalmente abierta del relevador.

Materiales:

- 1 LED (opcional)
- 1 resistencia de 220Ω (opcional)
- 1 módulo relevador de 5V individual o de un solo canal
- 1 foco incandescente de 120V, preferiblemente de 25W o menos (también puedes usar un foco LED o un foco fluorescente/ahorrador)
- 1 socket de rosca para foco de 120V
- 1 clavija de 120V sencilla
- 6 jumpers o cables tipo Dupont (1 para el LED)
- cable de 120V (para el circuito de alto voltaje)
- plantilla de experimentos

El circuito es el siguiente:



**CUIDADO:** trabajaremos con alto voltaje, así que protege el circuito del relevador con cinta aislante o dentro de una caja pequeña, para evitar una descarga eléctrica. Los puntos ‘calientes’ del circuito se encuentran debajo del módulo relevador, en sus terminales de bloque (donde se atornillan los cables) y en las terminales del socket.

El documento HTML que hemos estado utilizando los simplificaremos un poco en este ejercicio:

```
<!DOCTYPE html>
<html>
<head>

<title>CONTROL DE RELEVADOR</title>
<meta name='viewport' content='width=device-width, initial-scale=1'>
<link rel='icon' href='data:, '>

<style>
html
{
    font-family: Helvetica;
    display: inline-block;
    margin: 0px auto;
    text-align: center;
}
```



```
}

.BOTON1
{
background-color: #4CAF50;
border: none;
color: white;
padding: 16px 40px;
text-decoration: none;
font-size: 30px;
margin: 2px;
cursor: pointer;
}

.BOTON2
{
background-color: #555555;
border: none;
color: white;
padding: 16px 40px;
text-decoration: none;
font-size: 30px;
margin: 2px;
cursor: pointer;
}

</style>

</head>
<body>
<h1>CONTROL DE RELEVADOR</h1>
ESTE ESPACIO LO RESERVAREMOS PARA EL TEXTO DE ESTADO
Y EL BOTON DE CONTROL
</body>
</html>
```

Los estilos CSS los conservaremos y cambiaremos el título de la pestaña y de la página. Ahora sólo usaremos un botón, pero esta parte del documento la agregaremos en el código de Arduino (como en el ejercicio anterior).



Cuando pruebes el programa final, la apariencia de la página web será la siguiente:



Ahora veamos el código. La librería `WiFi.h` y las primeras variables ya las conoces:

```
#include <WiFi.h>
const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;
```

Después agregaremos una variable `String` para guardar el estado del relevador:

```
String ESTADO_RELLEVADOR="off";
```

Esta vez separaremos el documento HTML en tres secciones (cada una con su variable `String`): la primera irá desde el inicio hasta el texto de título `<h1>CONTROL DE RELEVADOR</h1>`, la segunda la reservaremos para las instrucciones del botón y el texto del estado del relevador, la tercera sección tendrá el resto del documento HTML:

```
String PAGINA_WEB_SECCION_1=<!DOCTYPE html> <html> <head> <title>CONTROL DE
RELEVADOR</title> <meta name='viewport' content='width=device-width,
initialscale=1'> <link rel='icon' href='data:,> <style> html { font-family:
Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1
{ background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 {
background-color: #555555; border: none; color: white; padding: 16px 40px; text-
```



```
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style>
</head> <body> <h1>CONTROL DE RELEVADOR</h1>";
String PAGINA_WEB_SECCION_2="";
String PAGINA_WEB_SECCION_3="</body> </html>";
```

Finalmente declaramos la variable para el GPIO que controlará al relevador:

```
const int GPIO_RELEVADOR = 14;
```

En `setup()` configuramos la comunicación serial y el GPIO del relevador:

```
Serial.begin(115200);
pinMode(GPIO_RELEVADOR, OUTPUT);
digitalWrite(GPIO_RELEVADOR, LOW);
```

la rutina de conexión Wifi es la misma que hemos estado utilizando:

```
Serial.print("Conectando a:");
Serial.println(ssid);
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
}
Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.localIP());
server.begin();
```

Dentro de `loop()` tendremos la misma rutina para el servidor, pero haremos unos cambios dentro de `if(PETICION_CLIENTE.Length() == 0):`

```
WiFiClient client = server.available();
if(client)
{
    Serial.println("Cliente conectado");
    while(client.connected())
    {
        if(client.available())
        {
            CADENA_ENCABEZADO = client.read();
```



```
Serial.write(CADENA_ENCABEZADO);
ENCABEZADO += CADENA_ENCABEZADO;
if(CADENA_ENCABEZADO == '\n')
{
    if(PETICION_CLIENTE.Length() == 0)
    {
        RESPUESTA_NAVEGADOR();
        ACCIONES_HTML(ENCABEZADO);
        if(ESTADO_RELEVADOR=="off")
        {
            PAGINA_WEB_SECCION_2=      "<p>Relevador      desactivado</p>      <p><a href='/14/on'><button class='BOTON1'>ENCENDER</button></a></p>";
        }
        else if(ESTADO_RELEVADOR=="on")
        {
            PAGINA_WEB_SECCION_2=      "<p>Relevador      activado</p>      <p><a href='/14/off'><button class='BOTON2'>APAGAR</button></a></p>";
        }
        client.println(PAGINA_WEB_SECCION_1);
        client.println(PAGINA_WEB_SECCION_2);
        client.println(PAGINA_WEB_SECCION_3);
        client.println();
        break;
    }
    else {PETICION_CLIENTE = "";}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}
}
}
ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}
```



En `if(PETICION_CLIENTE.Length() == 0)` tendremos un `if` y un `else if` para habilitar el botón “Encender” o “Apagar”, dependiendo del valor de `ESTADO_RELEVADOR` (similar al ejercicio 18). Además de habilitar los botones, también mostraremos un mensaje que indique el estado del relevador. Por ejemplo, cuando esté desactivado, mostraremos el mensaje que indica que está desactivado y habilitaremos el botón para activarlo, usando las siguientes instrucciones HTML:

```
<p>Relevador desactivado</p> <p><a href='/14/on'><button class='BOTON1'>ENCENDER</button></a></p>
```

Usaremos `client.println()` para enviar el contenido de las tres secciones `PAGINA_WEB_SECCION_?`. La subrutina `RESPUESTA_NAVEGADOR` se mantiene sin cambios:

```
void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}
```

La subrutina `ACCIONES_HTML` es similar a la de los ejercicios anteriores, pero ahora, el control del GPIO del relevador dependerá de la petición `GET /14/on` y `GET /14/off`. Cuando el GPIO cambia a `HIGH`, el relevador cerrará el interruptor, si está en `LOW`, el interruptor permanecerá abierto:

```
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /14/on")>=0)
    {
        Serial.println("RELEVADOR ENCENDIDO");
        digitalWrite(GPIO_RELEVADOR, HIGH);
        ESTADO_RELEVADOR ="on";
    }
    else if(ENCABEZADO.indexOf("GET /14/off")>=0)
    {
        Serial.println("RELEVADOR APAGADO");
        digitalWrite(GPIO_RELEVADOR, LOW);
        ESTADO_RELEVADOR ="off";
    }
}
```



El código completo queda de la siguiente forma:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;

String ESTADO_RELLEVADOR="off";

String PAGINA_WEB_SECCION_1=<!DOCTYPE html> <html> <head> <title>CONTROL DE
RELEVADOR</title> <meta name='viewport' content='width=device-width,
initialscale=1'> <link rel='icon' href='data:,> <style> html { font-family:
Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1
{ background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2
{ background-color: #555555; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style>
</head> <body> <h1>CONTROL DE RELLEVADOR</h1>;
String PAGINA_WEB_SECCION_2="";
String PAGINA_WEB_SECCION_3=</body> </html>";

const int GPIO_RELLEVADOR = 14;

void setup()
{
    Serial.begin(115200);
    pinMode(GPIO_RELLEVADOR, OUTPUT);
    digitalWrite(GPIO_RELLEVADOR, LOW);

    Serial.print("Conectando a:");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
}
```



```
}

Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.localIP());
server.begin();
}

void Loop()
{
    WiFiClient client = server.available();
    if(client)
    {
        Serial.println("Cliente conectado");
        while(client.connected())
        {
            if(client.available())
            {
                CADENA_ENCABEZADO = client.read();
                Serial.write(CADENA_ENCABEZADO);
                ENCABEZADO += CADENA_ENCABEZADO;
                if(CADENA_ENCABEZADO == '\n')
                {
                    if(PETICION_CLIENTE.length() == 0)
                    {
                        RESPUESTA_NAVEGADOR();
                        ACCIONES_HTML(ENCABEZADO);
                        if(ESTADO_RELEVADOR=="off")
                        {
                            PAGINA_WEB_SECCION_2=      "<p>Relevador      desactivado</p>      <p><a href='/14/on'><button class='BOTON1'>ENCENDER</button></a></p>";
                        }
                        else if(ESTADO_RELEVADOR=="on")
                        {
                            PAGINA_WEB_SECCION_2=      "<p>Relevador      activado</p>      <p><a href='/14/off'><button class='BOTON2'>APAGAR</button></a></p>";
                        }
                    }
                }
            }
        }
    }
}
```



```
        client.println(PAGINA_WEB_SECCION_1);
        client.println(PAGINA_WEB_SECCION_2);
        client.println(PAGINA_WEB_SECCION_3);
        client.println();
        break;
    }
    else {PETICION_CLIENTE = "";}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}
}
ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}

void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}

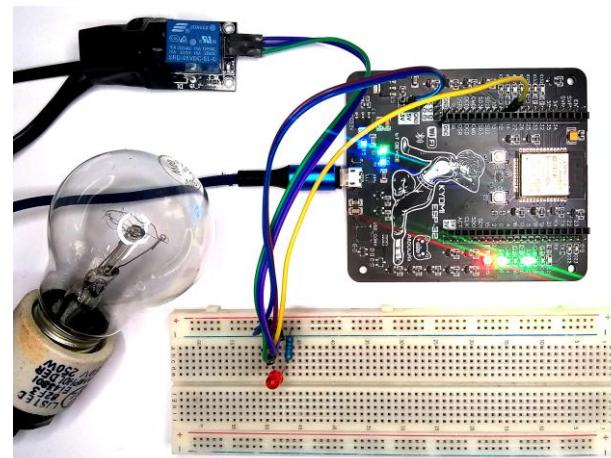
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /14/on")>=0)
    {
        Serial.println("RELEVADOR ENCENDIDO");
        digitalWrite(GPIO_RELEVADOR, HIGH);
    }
}
```



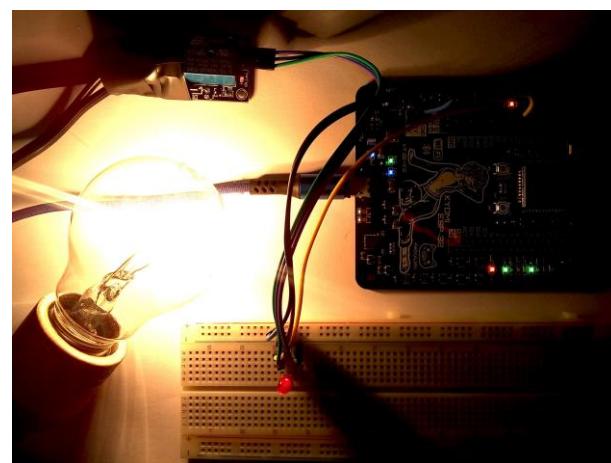
```
ESTADO_RELÉVADOR = "on";  
}  
else if(ENCABEZADO.indexOf("GET /14/off")>=0)  
{  
    Serial.println("RELÉVADOR APAGADO");  
    digitalWrite(GPIO_RELÉVADOR, LOW);  
    ESTADO_RELÉVADOR = "off";  
}  
}
```

Ahora prueba el programa. El mensaje de estado y el botón cambiarán cuando actives o desactives el relevador.

The screenshot shows a web browser window titled "CONTROL DE RELÉVADOR". The address bar indicates the URL is "No seguro | 192.168.0.105". The page content displays the message "Relevador desactivado" and features a large green button labeled "ENCENDER". At the bottom left, there is a text input field containing the URL "192.168.0.105/14/on".



The screenshot shows the same web browser window after the relay has been turned on. The message now reads "Relevador activado" and the "ENCENDER" button has been replaced by a dark grey "APAGAR" button. The URL at the bottom left is now "192.168.0.105/14/off".





## Servidor protegido por contraseña

Podemos agregar credenciales a nuestro servidor web para que sólo sea accesible mediante un usuario y contraseña.

Agregar credenciales a nuestro servidor es muy sencillo, así que vamos a retomar el ejercicio 19 para agregarle protección con usuario y contraseña.

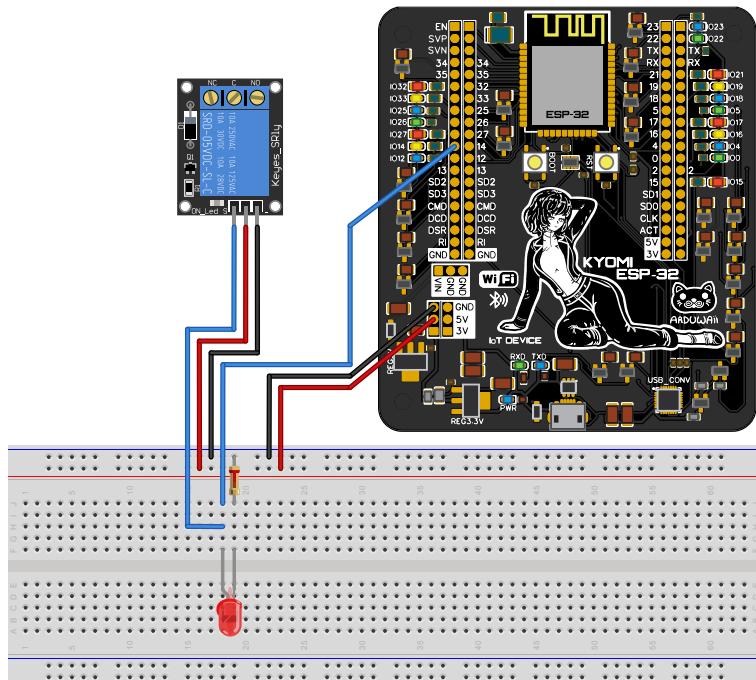
### Ejercicio 20

Modificaremos el ejercicio 19 para que sólo sea accesible con el usuario ‘SERVIDOR’ y la contraseña ‘1234abcd’. Esta vez omitiremos el circuito de alto voltaje, pero conservaremos el relevador para comprobar el funcionamiento.

#### Materiales:

- 1 LED (opcional)
- 1 resistencia de 220Ω (opcional)
- 1 módulo relevador de 5V individual o de un solo canal
- 6 jumpers o cables tipo Dupont (1 para el LED)
- plantilla de experimentos

El circuito es el siguiente:



La página web del servidor la dejaremos sin cambios, agregaremos las credenciales directamente en el código Arduino.



Veamos el código. Las variables iniciales permanecerán iguales, incluyendo a las secciones de la página web:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;
String ESTADO_RELEVADOR="off";

String PAGINA_WEB_SECCION_1=<!DOCTYPE html> <html> <head> <title>CONTROL DE
RELEVADOR</title> <meta name='viewport' content='width=device-width,
initialscale=1'> <link rel='icon' href='data:, '> <style> html { font-family:
Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1
{ background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2
{ background-color: #555555; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style>
</head> <body> <h1>CONTROL DE RELEVADOR</h1>;
String PAGINA_WEB_SECCION_2="";
String PAGINA_WEB_SECCION_3=</body> </html>;

const int GPIO_RELEVADOR = 14;
```

En `setup()` tampoco tendremos que hacer cambios:

```
Serial.begin(115200);
pinMode(GPIO_RELEVADOR, OUTPUT);
digitalWrite(GPIO_RELEVADOR, LOW);

Serial.print("Conectando a:");
Serial.println(ssid);
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
}
Serial.println("");
```



```
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.localIP());
server.begin();
```

Dentro de `loop()` tendremos la rutina típica del servidor, pero las credenciales las agregaremos en `if(PETICION_CLIENTE.length() == 0):`

```
WiFiClient client = server.available();
if(client)
{
    Serial.println("Cliente conectado");
    while(client.connected())
    {
        if(client.available())
        {
            CADENA_ENCABEZADO = client.read();
            Serial.write(CADENA_ENCABEZADO);
            ENCABEZADO += CADENA_ENCABEZADO;
            if(CADENA_ENCABEZADO == '\n')
            {
                if(PETICION_CLIENTE.length() == 0)
                {
                    if(ENCABEZADO.indexOf("U0VSVkLET1I6MTIzNGFiY2Q=") >= 0)
                    {
                        RESPUESTA_NAVEGADOR();
                        ACCIONES_HTML(ENCABEZADO);

                        if(ESTADO_RELEVADOR=="off")
                        {
                            PAGINA_WEB_SECCION_2=<p>Relevador desactivado</p> <p><a href='/14/on'><button class='BOTON1'>ENCENDER</button></a></p>";
                        }
                        else if(ESTADO_RELEVADOR=="on")
                        {
                            PAGINA_WEB_SECCION_2=<p>Relevador activado</p> <p><a href='/14/off'><button class='BOTON2'>APAGAR</button></a></p>";
                        }
                    }
                }
            }
        }
    }
}
```



```
    client.println(PAGINA_WEB_SECCION_1);
    client.println(PAGINA_WEB_SECCION_2);
    client.println(PAGINA_WEB_SECCION_3);
    client.println();
    break;
}
else
{
    client.println("HTTP/1.1 401 Unauthorized");
    client.println("WWW-Authenticate: Basic realm=\"Secure\"");
    client.println("Content-Type: text/html");
    client.println();
    client.println("<html>Credenciales incorrectas</html>");
}
}
else {PETICION_CLIENTE = "";}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}
}
}
ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}
```

En `if(PETICION_CLIENTE.Length() == 0)` ahora tendremos otro `if` y un `else` que se ejecutarán dependiendo de si la autenticación del usuario y contraseña fue correcta. Esta vez no enviaremos directamente la respuesta con la página web al cliente, primero tendrá que ingresar sus credenciales para que el servidor comience a enviar respuestas.

En `if(ENCABEZADO.indexOf("U0VSVkLET1I6MTIzNGFiY2Q=") >= 0)` colocaremos la misma rutina que envía la página web al servidor (y que también controla los botones). Este `if` se ejecuta cuando el usuario y contraseña ingresados en el navegador web son correctos. Si no se cumple esta



condición, un `else` envía una respuesta genérica al cliente que indica que las credenciales ingresadas no son correctas, para volver a pedirlas. La condición de `if(ENCABEZADO.indexOf("U0VSVkLET1I6MTIzNGFiY2Q=") >= 0)` indica que, en la petición del cliente que guardamos en `ENCABEZADO`, debe encontrarse el valor `U0VSVkLET1I6MTIzNGFiY2Q=`, el cual contiene al usuario ‘SERVIDOR’ y la contraseña ‘1234abcd’, codificados en base 64.

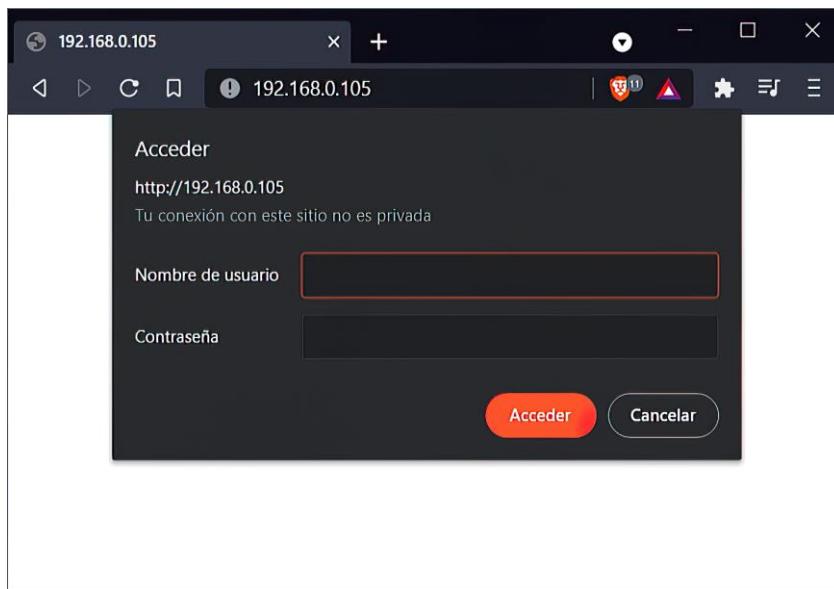
Podemos convertir las credenciales a base 64 en muchos sitios web, pero deben tener el siguiente formato: **Usuario:Contraseña**. En este caso tendríamos **SERVIDOR:1234abcd**, que es lo que colocaremos en el convertidor:

The screenshot shows a browser window titled "64 Base64 Encode - Online Encoder". The URL is "base64code.com/encode". The main title is "Base64 Decode" with "Encode" underlined. Below it is a breadcrumb navigation: "Home > Base64 > Encode". A horizontal menu bar includes "Decode", "Encode" (which is highlighted), "URL Decoder", "String Utilities", and "HTML Escape". The main content area has a heading "Encode to Base64 format". A text input field contains "SERVIDOR:1234abcd". To the right of the input field is a checkbox labeled "URL Safe" and a large red-bordered "Encode" button. Below the input field is a text output field containing "U0VSVkLET1I6MTIzNGFiY2Q=". The entire screenshot is framed by a red border.

El resultado de la conversión (`U0VSVkLET1I6MTIzNGFiY2Q=`) es el que colocaremos en la condición `if(ENCABEZADO.indexOf(" ") >= 0)`. Cuando el navegador cliente envíe la primera petición a nuestro servidor, el servidor le indicará al cliente que también envíe las credenciales



codificadas dentro del mensaje de petición. En el navegador veremos que el servidor nos va a pedir ingresar el usuario y la contraseña que especificamos:



El resto de la rutina de `Loop()` permanecerá igual.

La subrutina `RESPUESTA_NAVEGADOR` y `ACCIONES_HTML` también se mantienen sin cambios:

```
void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}
```

```
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /14/on")>=0)
    {
        Serial.println("RELEVADOR ENCENDIDO");
        digitalWrite(GPIO_RELEVADOR, HIGH);
        ESTADO_RELEVADOR ="on";
    }
}
```



```
else if(ENCABEZADO.indexOf("GET /14/off")>=0)
{
    Serial.println("RELEVADOR APAGADO");
    digitalWrite(GPIO_RELEVADOR, LOW);
    ESTADO_RELEVADOR = "off";
}
}
```

El código completo queda de la siguiente forma:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(80);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;
String ESTADO_RELEVADOR="off";

String PAGINA_WEB_SECCION_1=<!DOCTYPE html> <html> <head> <title>CONTROL DE
RELEVADOR</title> <meta name='viewport' content='width=device-width,
initialscale=1'> <link rel='icon' href='data:, '> <style> html { font-family:
Helvetica; display: inline-block; margin: 0px auto; text-align: center; } .BOTON1
{ background-color: #4CAF50; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 {
background-color: #555555; border: none; color: white; padding: 16px 40px; text-
decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style>
</head> <body> <h1>CONTROL DE RELEVADOR</h1>";

String PAGINA_WEB_SECCION_2="";
String PAGINA_WEB_SECCION_3="</body> </html>";

const int GPIO_RELEVADOR = 14;

void setup()
{
    Serial.begin(115200);
    pinMode(GPIO_RELEVADOR, OUTPUT);
    digitalWrite(GPIO_RELEVADOR, LOW);
```



```
Serial.print("Conectando a:");
Serial.println(ssid);
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
}
Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.LocalIP());
server.begin();
}

void Loop()
{
    WiFiClient client = server.available();
    if(client)
    {
        Serial.println("Cliente conectado");
        while(client.connected())
        {
            if(client.available())
            {
                CADENA_ENCABEZADO = client.read();
                Serial.write(CADENA_ENCABEZADO);
                ENCABEZADO += CADENA_ENCABEZADO;
                if(CADENA_ENCABEZADO == '\n')
                {
                    if(PETICION_CLIENTE.Length() == 0)
                    {
                        if(ENCABEZADO.indexOf("U0VSVkLET1I6MTIzNGFiY2Q=") >= 0)
                        {
                            RESPUESTA_NAVEGADOR();
                            ACCIONES_HTML(ENCABEZADO);
                        }
                    }
                }
            }
        }
    }
}
```



```
        if(ESTADO_RELÉ=="off")
    {
        PAGINA_WEB_SECCION_2=<p>Relé desactivado</p> <p><a href='/14/on'><button class='BOTON1'>ENCENDER</button></a></p>";
    }
    else if(ESTADO_RELÉ=="on")
    {
        PAGINA_WEB_SECCION_2=<p>Relé activado</p> <p><a href='/14/off'><button class='BOTON2'>APAGAR</button></a></p>";
    }

    client.println(PAGINA_WEB_SECCION_1);
    client.println(PAGINA_WEB_SECCION_2);
    client.println(PAGINA_WEB_SECCION_3);
    client.println();
    break;
}
else
{
    client.println("HTTP/1.1 401 Unauthorized");
    client.println("WWW-Authenticate: Basic realm=\"Secure\"");
    client.println("Content-Type: text/html");
    client.println();
    client.println("<html>Credenciales incorrectas</html>");
}
else
{
    PETICIÓN_CLIENTE = "";
}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICIÓN_CLIENTE += CADENA_ENCABEZADO;
}
}
}
```



```
ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}

void RESPUESTA_NAVEGADOR()
{
    WiFiClient client = server.available();
    client.println("HTTP/1.1 200 OK");
    client.println("Content-type:text/html");
    client.println("Connection: close");
    client.println();
}

void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /14/on")>=0)
    {
        Serial.println("RELEVADOR ENCENDIDO");
        digitalWrite(GPIO_RELEVADOR, HIGH);
        ESTADO_RELEVADOR ="on";
    }
    else if(ENCABEZADO.indexOf("GET /14/off")>=0)
    {
        Serial.println("RELEVADOR APAGADO");
        digitalWrite(GPIO_RELEVADOR, LOW);
        ESTADO_RELEVADOR ="off";
    }
}
```



Prueba el programa. Cuando ingreses el usuario y contraseña correctos, la página web se desplegará y funcionará normalmente. Si el usuario o la contraseña no son correctos, el navegador te volverá a pedir los datos o mostrará un mensaje indicando que las credenciales son incorrectas:

The image shows two side-by-side browser windows. The left window displays a login form with 'Nombre de usuario' set to 'SERVIDOR' and 'Contraseña' masked. The right window shows the resulting page titled 'CONTROL DE RELEVADOR' with a green button labeled 'ENCENDER'.

The image shows two side-by-side browser windows. The left window shows a successful login with 'Nombre de usuario' set to 'usuario' and 'Contraseña' masked. The right window shows an unsuccessful login attempt where both fields are empty.

## Accede a tu servidor desde cualquier lugar

Hasta ahora hemos utilizado nuestra red local para acceder al servidor del ESP32, pero podemos configurarlo para que podamos acceder desde cualquier navegador web en cualquier parte del mundo. Esto lo haremos a través de 'túneles', que establecerán la conexión con nuestro servidor, pero necesitaremos una computadora para administrar las conexiones del túnel.



Si montas un servidor permanente, ten en cuenta que la computadora que administra el túnel tendría que estar funcionando todo el tiempo, para no perder la comunicación con nuestro servidor, pero por ahora solo haremos unas pruebas.

Primero tenemos que configurar el servicio de túnel que usaremos para la conexión con nuestro servidor. En este caso usaremos ngrok, que es un servicio de túneles TCP gratuito y nos permitirá establecer una conexión por internet con nuestro servidor.

Para poder usar ngrok, debemos crear una cuenta gratuita:

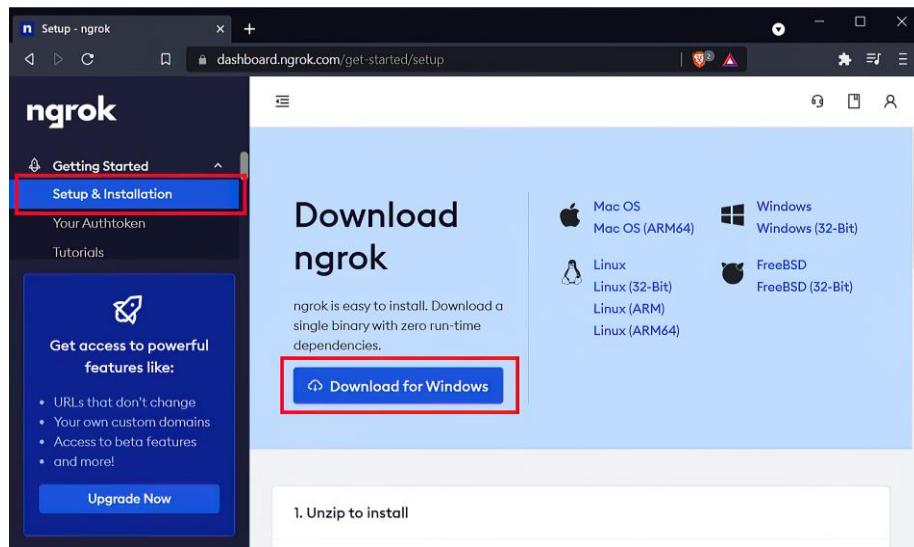
The screenshot shows the ngrok homepage. At the top, there's a navigation bar with links for 'How it works', 'Pricing', 'Enterprise solutions', 'Docs', 'Download', 'Login', and a prominent 'Sign up' button, which is highlighted with a red box. Below the navigation, there's a large heading 'Public URLs for testing your c' (with 'testing your' underlined). A subtext below it says: 'Spend more time programming. One command for an instant, secure URL to your localhost server through any NAT or firewall.' To the right, there's a small window showing a terminal session with ngrok running and a browser window showing a test page. At the bottom, there's a link 'https://dashboard.ngrok.com/signup'.

Una vez creada, debemos copiar y guardar el token de autenticación que nos generará automáticamente (siempre podrás volver a consultarla, no cambia):

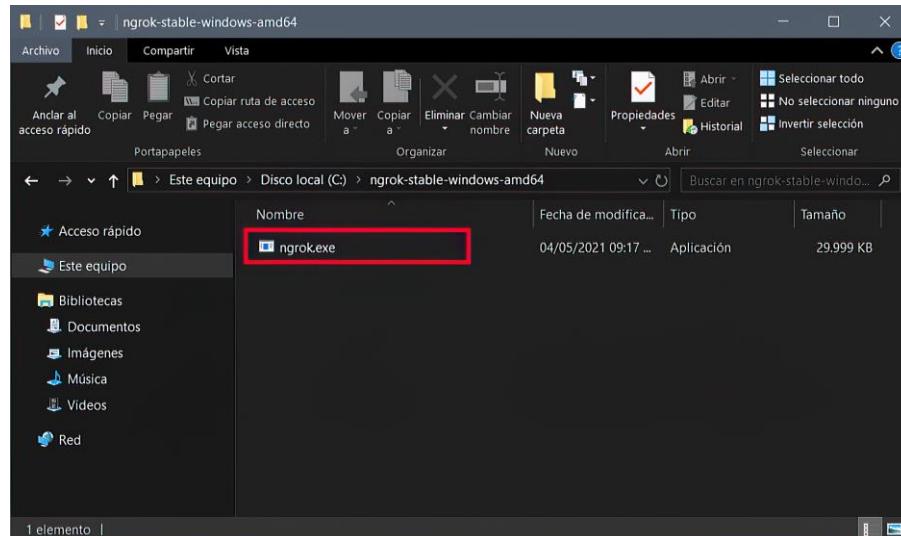
The screenshot shows the 'Your AuthToken' page from the ngrok dashboard. On the left, there's a sidebar with 'Getting Started' and 'Your AuthToken' selected (highlighted with a blue box). Below that, there's a section for 'Setup & Installation' and 'Tutorials'. In the main area, there's a heading 'Your AuthToken' with the subtext: 'This is your personal AuthToken. Use this to authenticate the ngrok agent that you downloaded.' Below this, there's a text input field containing the token 'lwHFgREDVVzroEff8q24h4Qo72M\_AZKxg7g3BNegI7u/' (highlighted with a red box), and a 'Copy' button next to it. At the bottom, there's a 'Command Line' section with the subtext: 'Authenticate your ngrok agent. You only have to do this once. The AuthToken is saved in the default configuration file.'



Después debemos descargar el programa necesario para habilitar el túnel de conexión:



En este caso descargaremos la versión para Windows, pero también está disponible para MacOS y Linux. Se descargará un archivo comprimido. Simplemente debes descomprimirlo para obtener un único archivo ejecutable:



Por ahora guardaremos el token de autenticación y el programa ejecutable, pero los usaremos en el siguiente ejercicio. Primero veamos otras instrucciones útiles para nuestro documento HTML.

## Set de caracteres para HTML y más sobre CSS

Los documentos HTML que hemos estado utilizando no han contenido ningún texto con acentos o con la letra ñ. Si agregamos alguno de estos caracteres al documento tal y como está, no se mostrarán



correctamente, ya que no hemos definido un set o conjunto de caracteres que incluya acentos y tildes.

Los sets de caracteres más comunes, que incluyen tildes y acentos, son UTF-8 e ISO-8859-1, pero el más recomendado es UTF-8. La configuración del set de caracteres se realiza dentro de la sección de metadatos `<meta>`. Por ejemplo, en los documentos HTML anteriores hemos estado utilizando la siguiente configuración en `<meta>`:

```
<meta name='viewport' content='width=device-width, initial-scale=1'>
```

para configurar el set UTF-8, basta con agregar lo siguiente:

```
<meta name='viewport' content='width=device-width, initial-scale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8'>
```

en el caso de ISO-8859-1, sería:

```
<meta name='viewport' content='width=device-width, initial-scale=1' http-equiv='Content-Type' content='text/html; charset=ISO-8859-1' />
```

Con esto ya podremos usar caracteres con acentos y tildes sin problemas en nuestra página web.

Ahora veamos otro selector bastante útil de CSS, que nos permitirá cambiar el fondo de nuestra página web. Con el selector `body` podemos cambiar varios aspectos del cuerpo de la página, pero nos centraremos en la modificación del fondo, reemplazándolo por una imagen:

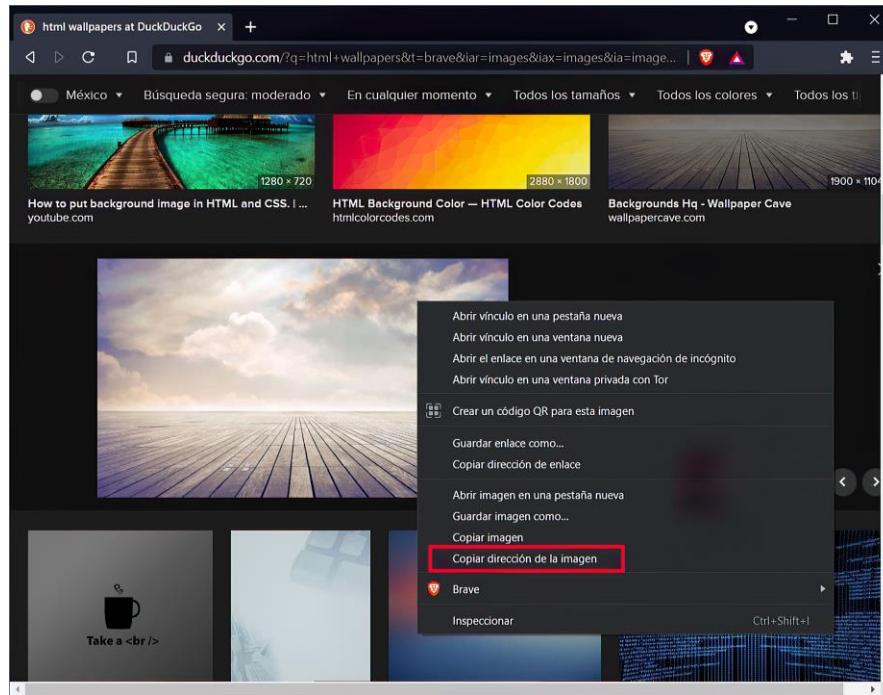
```
body
{
    background-image: url('PEGA AQUÍ LA UBICACIÓN DE TU IMAGEN O LA URL');
    background-repeat: no-repeat;
    background-attachment: fixed;
    background-size: 100% 100%;
}
```

en `background-image` colocaremos la ubicación de donde cargaremos la imagen para el fondo. Con `background-repeat: no-repeat` indicamos que la imagen no debe repetirse, que llene todo el espacio de la página una sola vez. Con `background-attachment: fixed` indicamos que la imagen debe estar fija y con `background-size: 100% 100%` indicamos que las proporciones de ancho y alto de la imagen deben ser del 100% (o escala 1:1).

La imagen de fondo puede ser un archivo local o del internet. Si es un archivo local, tendríamos que almacenarlo en la memoria del ESP32 o en una memoria microSD conectada al mismo. Lo más práctico es utilizar una imagen de internet, que sea descargada por el navegador cliente cuando acceda a nuestra página web.



Simplemente buscamos una imagen y copiamos y pegamos su URL o enlace en *background-image*:



```
background-image: url('https://external-content.duckduckgo.com/iu/?u=http%3A%2F%2Fwallpapercave.com%2Fwp%2FaJ5IFty.jpg&=1&nofb=1');
```

La red a la que conectemos nuestro servidor web debe tener acceso a internet (para descargar la imagen). Procura utilizar imágenes de resolución alta, al menos de 1280x700 pixeles, esto en el caso de pantallas anchas. Si tu página web está diseñada para pantallas verticales, usa una imagen de al menos 600x1024 pixeles.

Ahora hagamos un ejercicio para montar un servidor web que sea accesible desde internet.

## Ejercicio 21

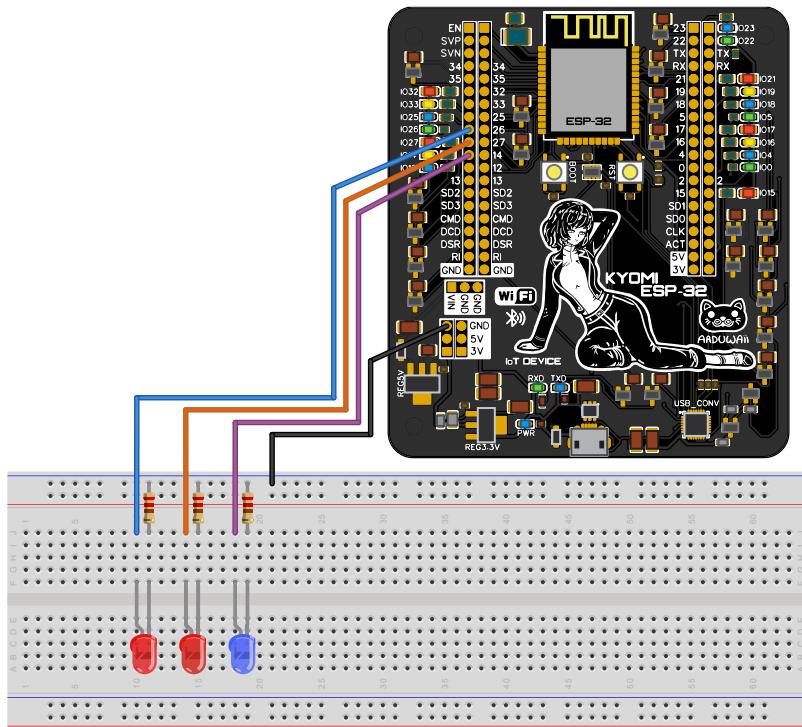
En este ejercicio montaremos un servidor web que pueda ser controlado desde cualquier parte del mundo, a través de internet. Controlaremos el encendido y apagado de tres LEDs. Usaremos ngrok para configurar los túneles de conexión por internet.

Materiales:

- 3 LEDs (opcional)
- 3 resistencias de 220Ω (opcional)
- 4 jumpers o cables tipo Dupont (para los LEDs)
- plantilla de experimentos



El circuito es el siguiente:



Vamos a personalizar el documento HTML para que la página web tenga una imagen de fondo (obtenida de internet), cambiaremos el color de los botones de encendido a rojo y el color de los botones de apagado a azul (todo esto en los estilos CSS). También estableceremos el set de caracteres UTF-8 y cambiaremos la fuente de texto a ‘Corbel’ (en el selector `html`):

```
<!DOCTYPE html>

<html>
  <head>
    <title>CONTROL DE GPIOs</title>
    <meta name='viewport' content='width=device-width, initial-scale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8' />
    <link rel='icon' href='data:, '>
    <style>
      html
      {
        font-family: Corbel;
        display: inline-block;
        margin: 0px auto;
        text-align: center;
      }
    </style>
  </head>
  <body>
    <h1>Controlador de GPIOs</h1>
    <p>Este dispositivo controla cuatro LEDs conectados a los pinos 13, 14, 15 y 16 del ESP-32. Puedes pulsar los botones para encender o apagar los LEDs correspondientes.</p>
    <button id='led1'>Encender LED 1</button>
    <button id='led2'>Apagar LED 1</button>
    <button id='led3'>Encender LED 2</button>
    <button id='led4'>Apagar LED 2</button>
  </body>
</html>
```



```
body
{
    background-image: url('https://external-
content.duckduckgo.com/iu/?u=http%3A%2F%2Fwallpapercave.com%2Fwp%2FaJ5IFty.jpg&f
=1&nofb=1');
    background-repeat: no-repeat;
    background-attachment: fixed;
    background-size: 100% 100%;
}

.BOTON1
{
    background-color: #FF4D4D;
    border: none;
    color: white;
    padding: 16px 40px;
    text-decoration: none;
    font-size: 30px;
    margin: 2px;
    cursor: pointer;
}

.BOTON2
{
    background-color: #99CCFF;
    border: none;
    color: white;
    padding: 16px 40px;
    text-decoration: none;
    font-size: 30px;
    margin: 2px;
    cursor: pointer;
}
</style>
</head>
```



```
<body>  
    <h1>Control GPIOs por internet</h1>  
    ESTE ESPACIO LO RESERVAREMOS PARA LOS TEXTOS DE ESTADO  
    Y PARA LOS BOTONES DE CONTROL  
</body>  
</html>
```

Ahora veamos el código. Las siguientes variables son las mismas que hemos utilizado en ejercicios anteriores, con la diferencia que el puerto del servidor ahora será el 1000, ya que ngrok no funciona con el puerto 80:

```
#include <WiFi.h>  
  
const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";  
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";  
WiFiServer server(1000);  
  
String ENCABEZADO, PETICION_CLIENTE = "";  
char CADENA_ENCABEZADO;
```

Agregaremos tres variables *String* para guardar los estados de los GPIO's (nos servirán para la habilitación de los botones):

```
String ESTADO_GPIO14="off", ESTADO_GPIO27="off", ESTADO_GPIO26="off";
```

Al igual que en el ejercicio 19, solo tendremos que dividir el documento HTML en tres secciones:

```
String PAGINA_WEB_SECCION_1="<!DOCTYPE html> <html> <head> <title>CONTROL DE GPIOs  
</title> <meta name='viewport' content='width=device-width, initial-scale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8' /> <link rel='icon' href='data:,'\> <style> html { font-family: Corbel; display: inline-block; margin: 0px auto; text-align: center; } body { background-image: url('https://external-content.duckduckgo.com/iu/?u=http%3A%2F%2Fwallpapercave.com%2Fwp%2FaJ5IFty.jpg&f=1&nofb=1'); background-repeat: no-repeat; background-attachment: fixed; background-size: 100% 100%; } .BOTON1 { background-color: #FF0000; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 { background-color: #002DCF; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style> </head> <body> <h1>Control GPIOs por internet</h1>";  
String PAGINA_WEB_SECCION_2="";  
String PAGINA_WEB_SECCION_3="</body> </html>";
```

Las siguientes variables auxiliares las usaremos después:

```
String PAGINA_WEB_SECCION_2_BOTON1="", PAGINA_WEB_SECCION_2_BOTON2="",  
PAGINA_WEB_SECCION_2_BOTON3="";
```



También declararemos los GPIO's para los LEDs:

```
const int GPIO_14=14, GPIO_27=27, GPIO_26=26;
```

Esta vez aginaremos una IP fija a nuestro servidor, para evitar que el router o modem asigne otra dinámicamente en algún momento. Para esto declaramos unas variables *IPAddress*, que contengan la IP fija, la compuerta de red, máscara de subred, el DNS primario y el secundario:

```
IPAddress Local_IP(192, 168, 0, 200);
IPAddress gateway(192, 168, 200, 200);

IPAddress subnet(255, 255, 0, 0);
IPAddress primaryDNS(8, 8, 8, 8);
IPAddress secondaryDNS(8, 8, 4, 4);
```

en este caso usaremos la IP 192.168.0.200 y la compuerta de red 192.168.200.200. La máscara de subred, el DNS primario y secundario serán 255.255.0.0, 8.8.8.8 y 8.8.4.4 respectivamente (generalmente estos no cambian). Puedes cambiar la IP o la compuerta de red si tienes problemas al conectarte con el servidor.

En *setup()* vamos a configurar la comunicación serial y los GPIO's de los LEDs para que arranquen en *LOW*:

```
Serial.begin(115200);
pinMode(GPIO_14, OUTPUT);
pinMode(GPIO_27, OUTPUT);
pinMode(GPIO_26, OUTPUT);
digitalWrite(GPIO_14, LOW);
digitalWrite(GPIO_27, LOW);
digitalWrite(GPIO_26, LOW);
```

Luego usaremos *WiFi.config* para establecer la configuración de la IP, la compuerta de red, la máscara de subred, el DNS primario y secundario. Despues colocaremos la rutina de conexión que ya conoces:

```
WiFi.config(Local_IP, gateway, subnet, primaryDNS, secondaryDNS);

Serial.print("Conectando a:");
Serial.println(ssid);
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
{
    delay(500);
```



```
    Serial.print(".");
}

Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.localIP());
server.begin();
```

Dentro de `loop()` tendremos la rutina normal del servidor. Al igual que en el ejercicio 20, protegeremos el servidor con el usuario ‘SERVIDOR’ y la contraseña ‘1234abcd’:

```
WiFiClient client = server.available();

if(client)
{
    Serial.println("Cliente conectado");
    while(client.connected())
    {
        if(client.available())
        {
            CADENA_ENCABEZADO = client.read();
            Serial.write(CADENA_ENCABEZADO);
            ENCABEZADO += CADENA_ENCABEZADO;
            if(CADENA_ENCABEZADO == '\n')
            {
                if(PETICION_CLIENTE.length() == 0)
                {
                    if(ENCABEZADO.indexOf("U0VSVkLET1I6MTIzNGFiY2Q=") >= 0)
                    {
                        client.println("HTTP/1.1 200 OK");
                        client.println("Content-type:text/html");
                        client.println("Connection: close");
                        client.println();

                        ACCIONES_HTML(ENCABEZADO);

                        if(ESTADO_GPIO14=="off")
                        {

```



```
PAGINA_WEB_SECCION_2_BOTON1=<p>EL GPIO14 está apagado</p> <p><a href='/14/on'><button class='BOTON1'>ENCENDER</button></a></p>";
}

else if(ESTADO_GPIO14=="on")
{

PAGINA_WEB_SECCION_2_BOTON1=<p>EL GPIO14 está encendido</p> <p><a href='/14/off'><button class='BOTON2'>APAGAR</button></a></p>";
}

if(ESTADO_GPIO27=="off")
{
PAGINA_WEB_SECCION_2_BOTON2=<p>EL GPIO27 está apagado</p> <p><a href='/27/on'><button class='BOTON1'>ENCENDER</button></a></p>";
}

else if(ESTADO_GPIO27=="on")
{
PAGINA_WEB_SECCION_2_BOTON2=<p>EL GPIO27 está encendido</p> <p><a href='/27/off'><button class='BOTON2'>APAGAR</button></a></p>";
}

if(ESTADO_GPIO26=="off")
{
PAGINA_WEB_SECCION_2_BOTON3=<p>EL GPIO26 está apagado</p> <p><a href='/26/on'><button class='BOTON1'>ENCENDER</button></a></p>";
}

else if(ESTADO_GPIO26=="on")
{
PAGINA_WEB_SECCION_2_BOTON3=<p>EL GPIO26 está encendido</p> <p><a href='/26/off'><button class='BOTON2'>APAGAR</button></a></p>";
}

PAGINA_WEB_SECCION_2=PAGINA_WEB_SECCION_2_BOTON1+PAGINA_WEB_SECCION_2_BOTON2+PAGINA_WEB_SECCION_2_BOTON3;

client.println(PAGINA_WEB_SECCION_1);
client.println(PAGINA_WEB_SECCION_2);
client.println(PAGINA_WEB_SECCION_3);
client.println();
break;
}
```



```
        else
        {
            client.println("HTTP/1.1 401 Unauthorized");
            client.println("WWW-Authenticate: Basic realm=\"Secure\"");
            client.println("Content-Type: text/html");
            client.println();
            client.println("<html>Credenciales incorrectas</html>");
        }
    }
    else {PETICION_CLIENTE = "";}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}
}
}
ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}
```

En `if(ENCABEZADO.indexOf("U0VSVklET1I6MTIzNGFiY2Q=") >= 0)` ya no usaremos la subrutina `RESPUESTA_NAVEGADOR()` para enviar la respuesta HTTP genérica, sino que la enviaremos directamente desde este `if`. Esto se debe a que, en `RESPUESTA_NAVEGADOR()`, volvíamos a declarar la función `WiFiClient client = server.available()` para usar las funciones `client.println()`, pero esto provoca conflictos al enviar la respuesta HTTP al cliente cuando no trabajamos con el puerto 80.

El resto de `if` y `else if` funcionan de manera similar a como los aplicamos en el ejercicio 18, para colocar las instrucciones HTML que habilitan los botones (y el mensaje de estado del GPIO). Pero esta vez, las instrucciones de los botones son guardadas individualmente en las variables `PAGINA_WEB_SECCION_2_BOTON1`, `PAGINA_WEB_SECCION_2_BOTON2` y `PAGINA_WEB_SECCION_2_BOTON3`. El contenido de estas variables se junta en `PAGINA_WEB_SECCION_2`, para poder mostrar los botones de los tres GPIO's.



La subrutina `ACCIONES_HTML` funciona de manera similar a la del ejercicio 18, solo que esta vez controlamos 3 GPIO's con la petición `GET` y usamos las variables `ESTADO_GPIO14`, `ESTADO_GPIO27` y `ESTADO_GPIO26` para cambiar a los botones de control:

```
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /14/on")>=0)
    {
        Serial.println("GPIO14 encendido");
        digitalWrite(GPIO_14, HIGH);
        ESTADO_GPIO14="on";
    }
    else if(ENCABEZADO.indexOf("GET /14/off")>=0)
    {
        Serial.println("GPIO14 apagado");
        digitalWrite(GPIO_14, LOW);
        ESTADO_GPIO14="off";
    }

    else if(ENCABEZADO.indexOf("GET /27/on")>=0)
    {
        Serial.println("GPIO27 encendido");
        digitalWrite(GPIO_27, HIGH);
        ESTADO_GPIO27="on";
    }
    else if(ENCABEZADO.indexOf("GET /27/off")>=0)
    {
        Serial.println("GPIO27 apagado");
        digitalWrite(GPIO_27, LOW);
        ESTADO_GPIO27="off";
    }

    else if(ENCABEZADO.indexOf("GET /26/on")>=0)
    {
        Serial.println("GPIO26 encendido");
    }
}
```



```
digitalWrite(GPIO_26, HIGH);
ESTADO_GPIO26="on";
}
else if(ENCABEZADO.indexOf("GET /26/off")>=0)
{
Serial.println("GPIO26 apagado");
digitalWrite(GPIO_26, LOW);
ESTADO_GPIO26="off";
}
}
```

El código completo queda de la siguiente forma:

```
#include <WiFi.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
WiFiServer server(1000);

String ENCABEZADO, PETICION_CLIENTE = "";
char CADENA_ENCABEZADO;

String ESTADO_GPIO14="off", ESTADO_GPIO27="off", ESTADO_GPIO26="off";

String PAGINA_WEB_SECCION_1="<!DOCTYPE html> <html> <head> <title>CONTROL DE GPIOs
</title> <meta name='viewport' content='width=device-width, initialscale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8' /> <link rel='icon' href='data:,'
<style> html { font-family: Corbel; display: inline-block; margin: 0px auto; text-align: center; } body { background-image: url('https://external-content.duckduckgo.com/iu/?u=http%3A%2F%2Fwallpapercave.com%2Fwp%2FaJ5IFTy.jpg&f=1&nofb=1'); background-repeat: no-repeat; background-attachment: fixed; background-size: 100% 100%; } .BOTON1 { background-color: #FF0000; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } .BOTON2 { background-color: #002DCF; border: none; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer; } </style> </head> <body> <h1>Control GPIOs por internet</h1>";
String PAGINA_WEB_SECCION_2="";
String PAGINA_WEB_SECCION_3="</body> </html>";
String PAGINA_WEB_SECCION_2_BOTON1="", PAGINA_WEB_SECCION_2_BOTON2="",
PAGINA_WEB_SECCION_2_BOTON3="";

const int GPIO_14=14, GPIO_27=27, GPIO_26=26;
```



```
IPAddress Local_IP(192, 168, 0, 200);
IPAddress gateway(192, 168, 200, 200);

IPAddress subnet(255, 255, 0, 0);
IPAddress primaryDNS(8, 8, 8, 8);
IPAddress secondaryDNS(8, 8, 4, 4);

void setup()
{
    Serial.begin(115200);
    pinMode(GPIO_14, OUTPUT);
    pinMode(GPIO_27, OUTPUT);
    pinMode(GPIO_26, OUTPUT);
    digitalWrite(GPIO_14, LOW);
    digitalWrite(GPIO_27, LOW);
    digitalWrite(GPIO_26, LOW);

    WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS);

    Serial.print("Conectando a:");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("Conectado");
    Serial.println("Dirección IP: ");
    Serial.println(WiFi.localIP());

    server.begin();
}
```



```
void Loop()
{
    WiFiClient client = server.available();
    if(client)
    {
        Serial.println("Cliente conectado");
        while(client.connected())
        {
            if(client.available())
            {
                CADENA_ENCABEZADO = client.read();
                Serial.write(CADENA_ENCABEZADO);
                ENCABEZADO += CADENA_ENCABEZADO;
                if(CADENA_ENCABEZADO == '\n')
                {
                    if(PETICION_CLIENTE.length() == 0)
                    {
                        if(ENCABEZADO.indexOf("U0VSVkLET1I6MTIzNGFiY2Q=") >= 0)
                        {
                            client.println("HTTP/1.1 200 OK");
                            client.println("Content-type:text/html");
                            client.println("Connection: close");
                            client.println();
                            ACCIONES_HTML(ENCABEZADO);

                            if(ESTADO_GPIO14=="off")
                            {
                                PAGINA_WEB_SECCION_2_BOTON1=<p>EL GPIO14 está apagado</p> <p><a href='/14/on'><button class='BOTON1'>ENCENDER</button></a></p>";
                            }
                            else if(ESTADO_GPIO14=="on")
                            {
                                PAGINA_WEB_SECCION_2_BOTON1=<p>El GPIO14 está encendido</p> <p><a href='/14/off'><button class='BOTON2'>APAGAR</button></a></p>";
                            }
                        }
                    }
                }
            }
        }
    }
}
```



```
if(ESTADO_GPIO27=="off")
{
    PAGINA_WEB_SECCION_2_BOTON2=<p>El GPIO27 está apagado</p> <p><a href='/27/on'><button class='BOTON1'>ENCENDER</button></a></p>";
}

else if(ESTADO_GPIO27=="on")
{
    PAGINA_WEB_SECCION_2_BOTON2=<p>El GPIO27 está encendido</p> <p><a href='/27/off'><button class='BOTON2'>APAGAR</button></a></p>";
}

if(ESTADO_GPIO26=="off")
{
    PAGINA_WEB_SECCION_2_BOTON3=<p>El GPIO26 está apagado</p> <p><a href='/26/on'><button class='BOTON1'>ENCENDER</button></a></p>";
}

else if(ESTADO_GPIO26=="on")
{
    PAGINA_WEB_SECCION_2_BOTON3=<p>El GPIO26 está encendido</p> <p><a href='/26/off'><button class='BOTON2'>APAGAR</button></a></p>";
}

PAGINA_WEB_SECCION_2=PAGINA_WEB_SECCION_2_BOTON1+PAGINA_WEB_SECCION_2_BOTON2+PAGINA_WEB_SECCION_2_BOTON3;

client.println(PAGINA_WEB_SECCION_1);
client.println(PAGINA_WEB_SECCION_2);
client.println(PAGINA_WEB_SECCION_3);
client.println();
break;
}
else
{
    client.println("HTTP/1.1 401 Unauthorized");
    client.println("WWW-Authenticate: Basic realm=\"Secure\"");
    client.println("Content-Type: text/html");
    client.println();
    client.println("<html>Credenciales incorrectas</html>");
}
```



```
        }
    else {PETICION_CLIENTE = "";}
}
else if(CADENA_ENCABEZADO != '\r')
{
    PETICION_CLIENTE += CADENA_ENCABEZADO;
}
}
ENCABEZADO = "";
client.stop();
Serial.println("Cliente desconectado");
Serial.println("");
}
```

```
void ACCIONES_HTML(String ENCABEZADO)
{
    if(ENCABEZADO.indexOf("GET /14/on")>=0)
    {
        Serial.println("GPIO14 encendido");
        digitalWrite(GPIO_14, HIGH);
        ESTADO_GPIO14="on";
    }
    else if(ENCABEZADO.indexOf("GET /14/off")>=0)
    {
        Serial.println("GPIO14 apagado");
        digitalWrite(GPIO_14, LOW);
        ESTADO_GPIO14="off";
    }

    else if(ENCABEZADO.indexOf("GET /27/on")>=0)
    {
        Serial.println("GPIO27 encendido");
        digitalWrite(GPIO_27, HIGH);
```



```
ESTADO_GPIO27="on";
}

else if(ENCABEZADO.indexOf("GET /27/off")>=0)
{
    Serial.println("GPIO27 apagado");
    digitalWrite(GPIO_27, LOW);
    ESTADO_GPIO27="off";
}

else if(ENCABEZADO.indexOf("GET /26/on")>=0)
{
    Serial.println("GPIO26 encendido");
    digitalWrite(GPIO_26, HIGH);
    ESTADO_GPIO26="on";
}

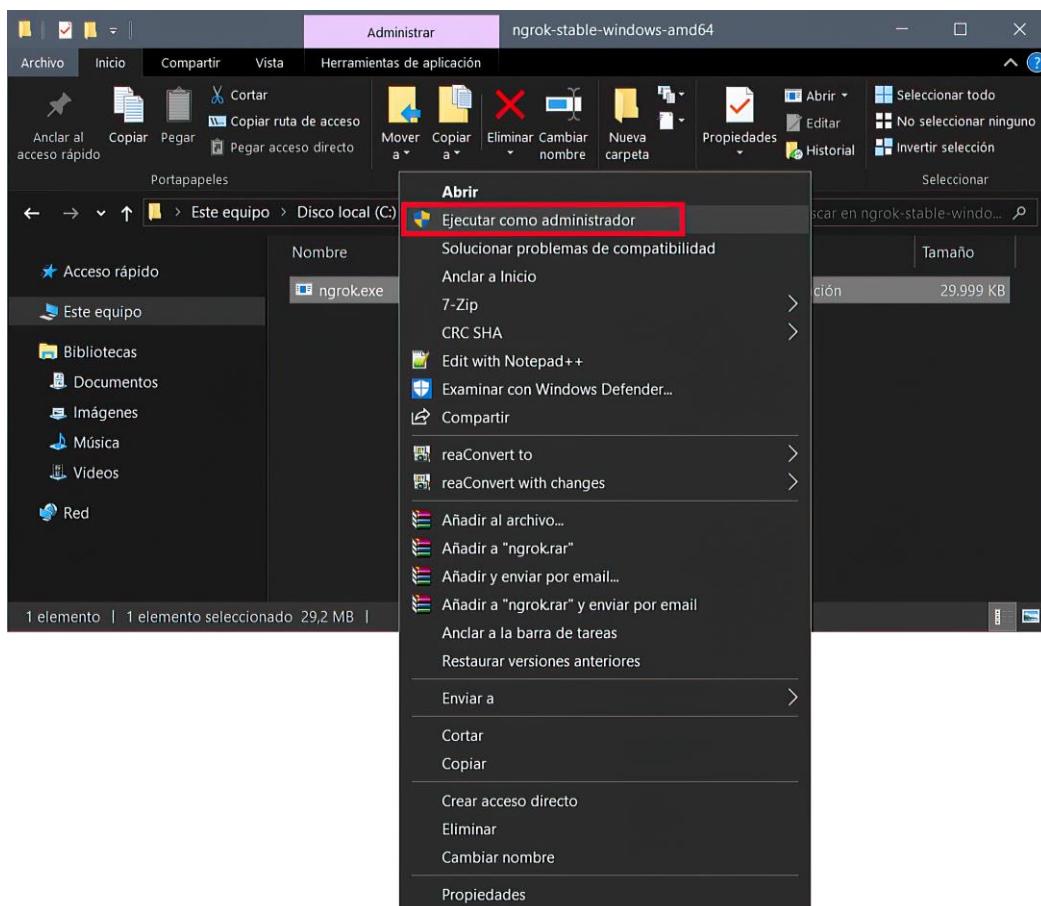
else if(ENCABEZADO.indexOf("GET /26/off")>=0)
{
    Serial.println("GPIO26 apagado");
    digitalWrite(GPIO_26, LOW);
    ESTADO_GPIO26="off";
}
}
```

Antes de conectar el servidor a internet, primero pruébalo en tu red local para verificar su funcionamiento. Esta vez tendrás que colocar la IP junto con el puerto del servidor:





Si el servidor funciona correctamente, entonces podremos conectarlo a internet. Para esto debemos abrir el programa que descargamos de ngrok, ejecutándolo como administrador:



Una vez abierto, deberás colocar la IP local de tu servidor (con el puerto) y el token de autenticación de ngrok en la línea de comandos:

```
C:\ngrok-stable-windows-amd64\ngrok.exe
ngrok http -subdomain=baz 8080  # port 8080 available at baz.ngrok.io
ngrok http foo.dev:80          # tunnel to host:port instead of localhost
ngrok http https://localhost   # expose a local https server
ngrok tcp 22                   # tunnel arbitrary TCP traffic to port 22
ngrok tls -hostname=foo.com 443 # TLS traffic for foo.com to port 443
ngrok start foo bar baz        # start tunnels from the configuration file

VERSION:
2.3.40

AUTHOR:
inconsnreveable - <alan@ngrok.com>

COMMANDS:
authtoken  save authtoken to configuration file
credits    prints author and licensing information
http      start an HTTP tunnel
start     start tunnels by name from the configuration file
tcp       start a TCP tunnel
tls       start a TLS tunnel
update    update ngrok to the latest version
version   print the version string
help      Shows a list of commands or help for one command

ngrok is a command line application, try typing 'ngrok.exe http 80'
at this terminal prompt to expose port 80.
C:\ngrok-stable-windows-amd64\ngrok.exe
```



para hacerlo, debes seguir el siguiente formato:

```
ngrok tcp TU_IP_LOCAL:PUERTO_DEL_SERVIDOR --auth token TU_TOKEN_DE_AUTENTICACION
```

```
C:\ Administrador: C:\ngrok-stable-windows-amd64\ngrok.exe
ngrok http -subdomain=baz 8080  # port 8080 available at baz.ngrok.io
ngrok http foo.dev:80          # tunnel to host:port instead of localhost
ngrok http https://localhost   # expose a local https server
ngrok tcp 22                   # tunnel arbitrary TCP traffic to port 22
ngrok tls -hostname=foo.com 443 # TLS traffic for foo.com to port 443
ngrok start foo bar baz       # start tunnels from the configuration file

VERSION:
2.3.40

AUTHOR:
inconshreveable - <alan@ngrok.com>

COMMANDS:
auth token    save auth token to configuration file
credits        prints author and licensing information
http           start an HTTP tunnel
start          start tunnels by name from the configuration file
tcp            start a TCP tunnel
tls             start a TLS tunnel
update         update ngrok to the latest version
version        print the version string
help           Shows a list of commands or help for one command

ngrok is a command line application, try typing 'ngrok.exe http 80'
at this terminal prompt to expose port 80.
C:\ngrok-stable-windows-amd64\ngrok tcp 192.168.0.200:1000 --auth token 1wHFgREDWzroEff8q24h4Qo72M_AZKxg7g3E
```

Una vez colocado, presiona la tecla ‘Enter’ y el túnel de conexión empezará a funcionar:

```
C:\ Administrador: C:\ngrok-stable-windows-amd64\ngrok.exe - ngrok tcp 192.168.0.200:1000 --auth token 1wHFgRE... - (Ctrl+C to quit)
ngrok by @inconshreveable

Session Status      online
Account             [REDACTED] (Plan: Free)
Version             2.3.40
Region              United States (us)
Web Interface       http://127.0.0.1:4040
Forwarding          tcp://8.tcp.ngrok.io:15489 -> 192.168.0.200:1000

Connections          ttl     opn      rt1      rt5      p50      p90
                      0       0       0.00    0.00    0.00    0.00
```

Se generará una dirección web, que es la que usaremos para acceder a nuestro servidor desde cualquier lugar, pero debes cambiar a `tcp://` por `http://`, por ejemplo:

```
tcp://8.tcp.ngrok.io:15489 → http://8.tcp.ngrok.io:15489
```

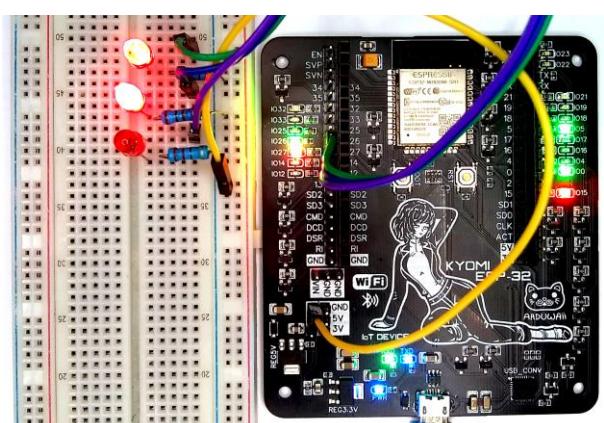
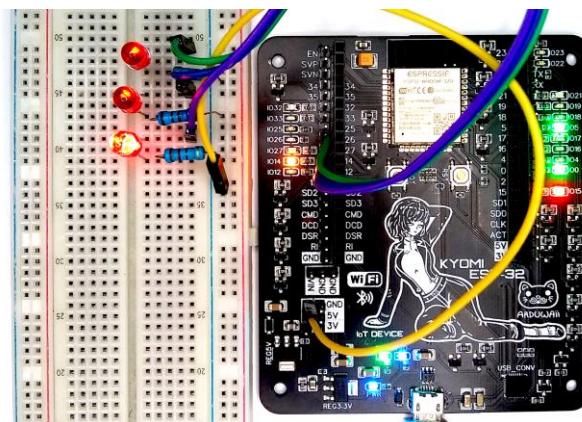
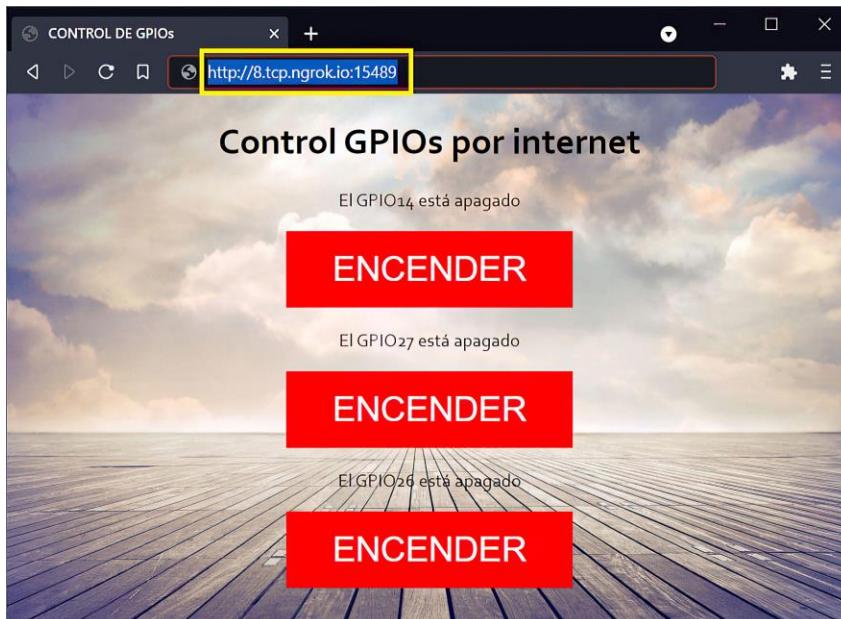
El programa de ngrok debe estar abierto y funcionando para no perder la conexión con nuestro servidor, y por supuesto, el ESP32 debe estar funcionando también.



## Capítulo 4: Wifi y servidores

Web

Ahora prueba el programa accediendo al servidor con la dirección web generada. El apagado y encendido de los LEDs tardará un poco dependiendo de la velocidad de tu conexión a internet:





## Aplicación práctica: servidor asíncrono con sensor de temperatura y humedad (DHT11)

Cuando trabajamos con sensores, es importante mostrar los resultados de sus mediciones en la página web de nuestro servidor y que se actualicen cada cierto tiempo. Una solución sería enviar directamente los valores a nuestra página mediante instrucciones HTML, pero tendríamos que actualizarla o refrescarla cada cierto tiempo, lo cual es una solución poco elegante. Lo más ideal es implementar un ‘script Java’ (JavaScript), para que se encargue de realizar peticiones a nuestro servidor y que actualice la parte específica de la página web en donde se muestran los datos. Entonces se dice que el servidor trabaja de manera asíncrona, porque sólo ciertos elementos están cambiando.

Los scripts de Java son capaces de controlar elementos del documento HTML en forma dinámica y sin tener que modificarlo directamente (similar a CSS), pero también permiten incrustar elementos complejos como animaciones, sonido o videos.

### Otras funciones para HTML y JavaScript básico

Ya hemos visto como colocar una imagen de fondo a nuestra página web mediante CSS, pero también podemos agregar otras imágenes en el cuerpo de la misma, iconos por ejemplo. Para colocar una imagen en nuestro documento, usaremos la siguiente etiqueta HTML:

```
<img src='PEGA AQUÍ LA UBICACIÓN DE TU IMAGEN O LA URL' alt='ALIAS O NOMBRE DE REFERENCIA' width='TAMAÑO HORIZONTAL EN PX' height='TAMAÑO VERTICAL EN PX'>
```

en `src` colocamos la ubicación de donde cargaremos la imagen. Al igual que con la imagen de fondo de la página, también podemos usar un archivo local o una imagen de internet. En `alt` colocaremos un nombre o un alias para la imagen, que básicamente nos servirá de referencia para usarlo en alguna otra instrucción. En `width` y `height` configuramos el tamaño horizontal y vertical de la imagen en pixeles.

La etiqueta `<img>` podemos usarla en cualquier parte del cuerpo de la página donde queramos insertar la imagen, preferiblemente en un párrafo, por ejemplo:

```
<p>
  <img src='https://image.flaticon.com/icons/png/512/1035/1035618.png' alt='TEMPERATURA' width='80' height='80'>
</p>
```

Otras opciones para personalizar nuestra página web pueden ser cambiar el color de un texto en específico y su tamaño de fuente.



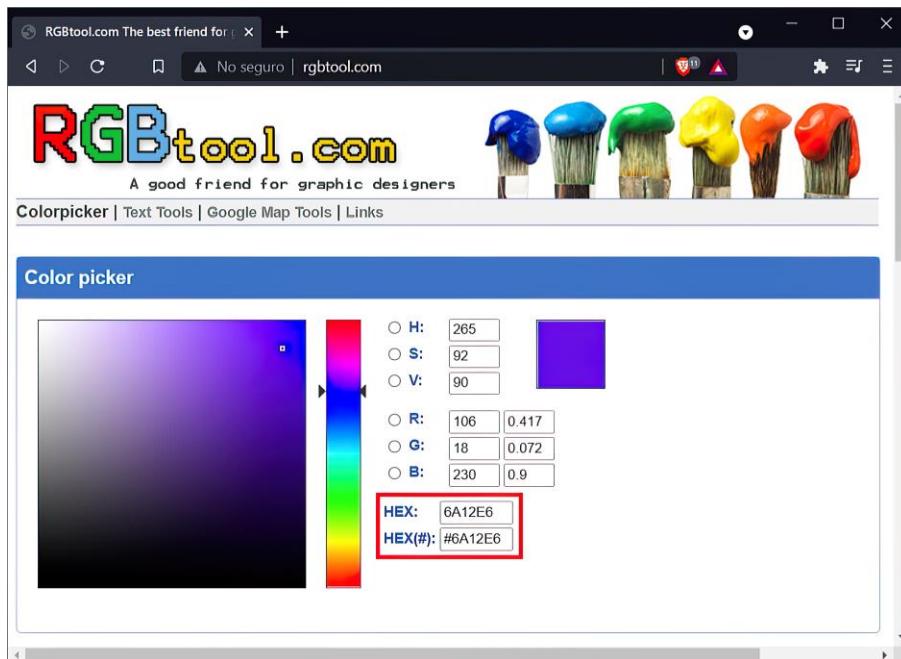
Estas propiedades las podemos agregar a las etiquetas de título, como las de tipo `<h1>` y `</h1>`, o a las etiquetas de párrafo. En el caso de una etiqueta de título, estas propiedades solo afectarían al texto de título, pero cuando se aplican a un párrafo, las propiedades establecidas van a afectar a todo el texto del párrafo. Para agregar las propiedades de color y fuente, debemos usar la instrucción `style`, junto con todas las propiedades que queramos agregar (separadas por `' ; '`). Por ejemplo, en el caso de las etiquetas de título tendríamos:

```
<h1 style='color:COLOR EN RGB;'> TEXTO </h1>
```

para el caso de los párrafos, sería:

```
<p style='color:COLOR EN RGB; font-size: TAMAÑO EN PIXELES;'> TEXTO </p>
```

en `color` colocamos el color del texto en código RGB hexadecimal. En muchos sitios web hay herramientas que nos permiten elegir colores y obtener su código hexadecimal:



en `font-size` colocamos el tamaño de fuente en pixeles. En las etiquetas de título no se puede cambiar el tamaño de fuente. Con el siguiente documento HTML puedes probar estas propiedades:

```
<!DOCTYPE HTML>
<html>
<head>
  <meta name='viewport' content='width=device-width, initialscale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8' />
</head>
```



```
<body>
    <h1 style='color:#00992B;'>Título en verde</h1>

    <p style='color:#E61227; font-size: 32px;'>
        Párrafo 1: ROJO a 32px
    </p>

    <p style='color:#1612E6; font-size: 25px;'>
        Párrafo 2: AZUL a 25px
    </p>
</body>
</html>
```



Una función muy importante de HTML son las etiquetas de marca `<span>` y `</span>`. Estas etiquetas nos permiten marcar un elemento de la página (un texto, por ejemplo), para que podamos referenciarlo en otra parte del documento. Básicamente sirven para darle un identificador a ciertos elementos o partes de nuestro documento HTML. Esto es bastante útil para modificar su contenido al trabajar con JavaScript, ya que necesitaremos estos identificadores. Podemos colocar párrafos y otros elementos entre las etiquetas de marca para darles un identificador, pero normalmente se usan para referenciar un texto sencillo que podamos modificar con JavaScript.

Para usar las etiquetas de marca, tenemos lo siguiente:

```
<span id='NOMBRE DEL IDENTIFICADOR'>
    CONTENIDO DE LA SECCION
</span>
```



en `id` colocamos el nombre que le daremos al identificador de la marca. Si en `CONTENIDO DE LA SECCION` colocamos un texto entre % (por ejemplo `%TEXTO%`), cuando trabajemos con el servidor asíncrono a través de JavaScript, podremos buscar ese texto en específico para modificarlo desde el servidor. También podemos agregar propiedades tipo `style` a `<span>` para cambiar el aspecto de `CONTENIDO DE LA SECCION`, por ejemplo:

```
<span id='NOMBRE DEL IDENTIFICADOR' style='color:COLOR EN RGB; font-size: TAMAÑO  
EN PIXELES;>  
  
CONTENIDO DE LA SECCION  
  
</span>
```

Al igual que CSS, JavaScript tiene su propio lenguaje de programación, y puede ser incluido en el mismo documento HTML. Para agregar el código JavaScript, lo haremos entre las etiquetas `<script>` y `</script>`, que deben estar en la sección `<html>` y `</html>`, junto al encabezado y el cuerpo.

Por sí mismo JavaScript es muy extenso, así que sólo veremos las instrucciones necesarias para hacer peticiones al servidor y actualizar elementos de nuestra página web.

Con la instrucción `setInterval` podremos ejecutar una función genérica o subrutina cada cierto intervalo de tiempo (dado en milisegundos), junto con todas las instrucciones que contenga:

```
setInterval(function(){AQUÍ VAN LAS INSTRUCCIONES DE LA FUNCIÓN}, TIEMPO);
```

En la función genérica `function` de `setInterval`, colocaremos las instrucciones para realizar peticiones al servidor de manera periódica. La siguiente rutina envía peticiones al servidor para recibir una respuesta, y reemplazarla en cierto lugar del documento HTML:

```
var xhttp = new XMLHttpRequest();  
  
xhttp.onreadystatechange = function()  
{  
    if (this.readyState == 4 && this.status == 200)  
    {  
        document.getElementById('IDENTIFICADOR').innerHTML = this.responseText;  
    }  
};  
  
xhttp.open('GET', '/HIPERVÍNCULO DE PETICIÓN', true);  
  
xhttp.send();
```

con `var` declaramos una variable llamada `xhttp`, que está igualada a la función `XMLHttpRequest()`, la cual nos permitirá llamar a las instrucciones con las que haremos las peticiones HTTP a nuestro servidor. Con esto



estamos convirtiendo a la variable `xhttp` en un objeto dependiente de la función `XMLHttpRequest()`.

En `xhttp.onreadystatechange` usamos el objeto `xhttp` para definir una función de tipo `onreadystatechange`, la cual se ejecuta cuando cambia el estado de alguna propiedad tipo `readyState`. En el siguiente `if` se hace esta comprobación: si `this.readyState` es igual a 4 (que se envió una petición al servidor y la respuesta que regresó está lista) y si `this.status` es igual a 200 (que la petición fue correcta), entonces la instrucción `document.getElementById` buscará el identificador o etiqueta de marca llamada `IDENTIFICADOR` en nuestro documento HTML, para reemplazar su contenido con la respuesta recibida del servidor (que está ‘guardada’ en `this.responseText`). Básicamente la rutina del `if (this.readyState == 4 && this.status == 200)` se encarga de recibir la respuesta del servidor para colocarla en la página web.

La petición al servidor la haremos con `xhttp.open('GET', '/HIPERVÍNCULO DE PETICIÓN', true)` y con `xhttp.send()`, donde en `xhttp.open` crearemos el mensaje de petición con el método `GET`, que incluye un hipervínculo de referencia para especificar la información que queremos pedirle al servidor, similar a como lo hacíamos con `href` en los ejercicios anteriores. Finalmente, con `xhttp.send()` enviaremos la petición.

Por ejemplo, con la siguiente rutina actualizaremos el valor de `DATO_ACTUALIZAR` cada 1000ms, haciendo una petición al servidor con el hipervínculo `/Actualizar-Dato`:

```
<body>
    <span id='DATO'>%DATO_ACTUALIZAR%</span>
</body>
<script>
setInterval(function(){
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function()
    {
        if (this.readyState == 4 && this.status == 200)
        {
            document.getElementById('DATO').innerHTML = this.responseText;
        }
    };
    xhttp.open('GET', '/Actualizar-Dato', true);
    xhttp.send();},1000);
</script>
```



Antes de pasar al ejercicio de aplicación, necesitarás unas cuantas librerías para poder realizar el programa:

- Las librerías “*DHT Sensor*” y “*Adafruit Unified Sensor Driver*” nos permitirán usar el sensor de temperatura y humedad DHT11 para el ejercicio de aplicación. Puedes descargarlas en .zip desde los siguientes enlaces:

**DHT Sensor:**

<https://codeload.github.com/adafruit/DHT-sensor-library/zip/master>

**Adafruit Unified Sensor Driver:**

[https://codeload.github.com/adafruit/Adafruit\\_Sensor/zip/master](https://codeload.github.com/adafruit/Adafruit_Sensor/zip/master)

- Las librerías “*ESPAsyncWebServer*” y “*Async TCP Library for ESP32*” las necesitaremos para poder usar las funciones del servidor asíncrono en el ESP32. Puedes descargarlas en .zip desde los siguientes enlaces:

**ESPAsyncWebServer:**

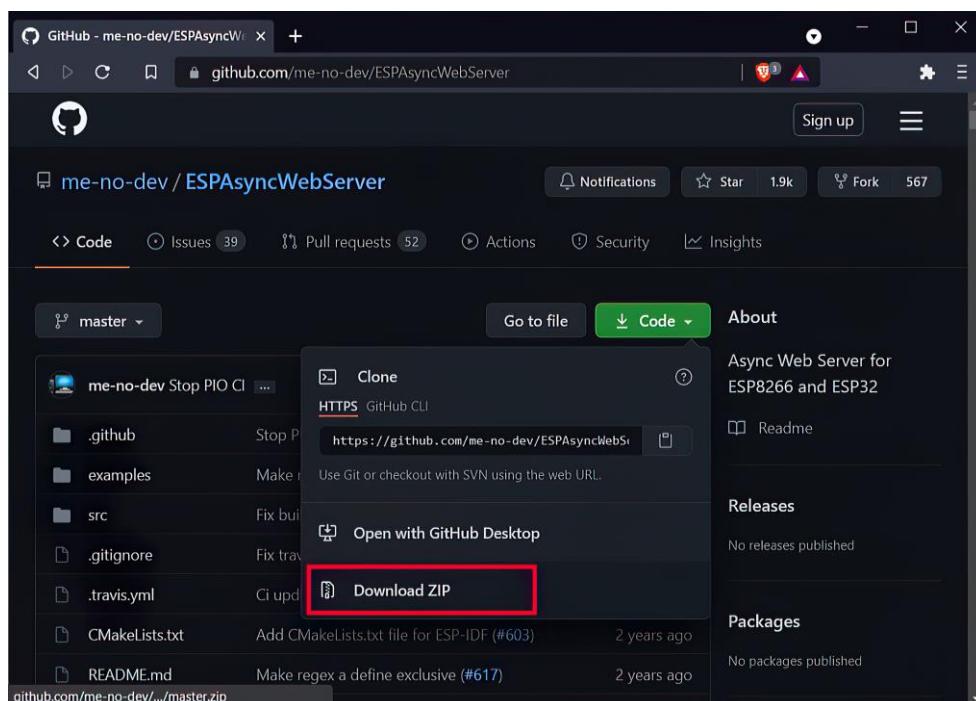
<https://codeload.github.com/me-no-dev/ESPAsyncWebServer/zip/master>

**Async TCP Library for ESP32:**

<https://github.com/me-no-dev/AsyncTCP/archive/refs/heads/master.zip>

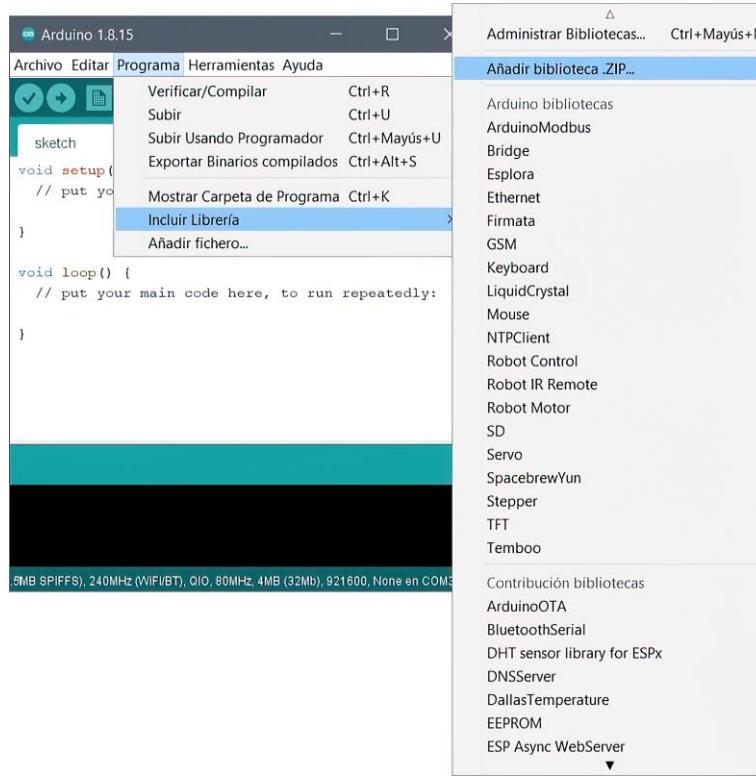
## Básicos: añadir librerías nuevas al IDE de Arduino

Las librerías nos permiten agregar funciones de sensores o tarjetas que no se encuentren integradas en nuestro IDE de Arduino. Las librerías las puedes encontrar en muchos sitios web, pero normalmente se encuentran en repositorios del sitio Github. En Github puedes encontrar las librerías desempaquetadas o incluso su código fuente, pero lo importante es descargarlas en formato comprimido (.zip):

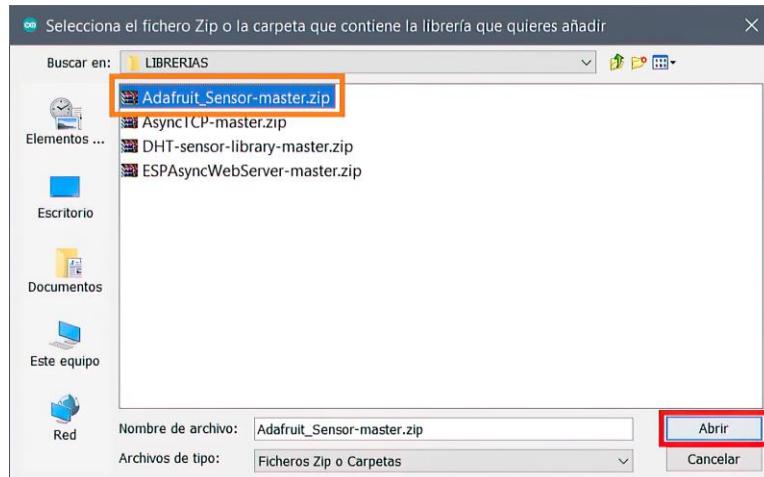




Una vez hayas descargado la librería en .zip, en el IDE de Arduino ve a la pestaña Programa > Incluir librería y da clic en ‘Añadir biblioteca .ZIP’:



Ahora sólo busca y abre la librería que descargaste:



Con esto habrás agregado la librería nueva y podrás usar las funciones que incluye.

Veamos el ejercicio de aplicación.



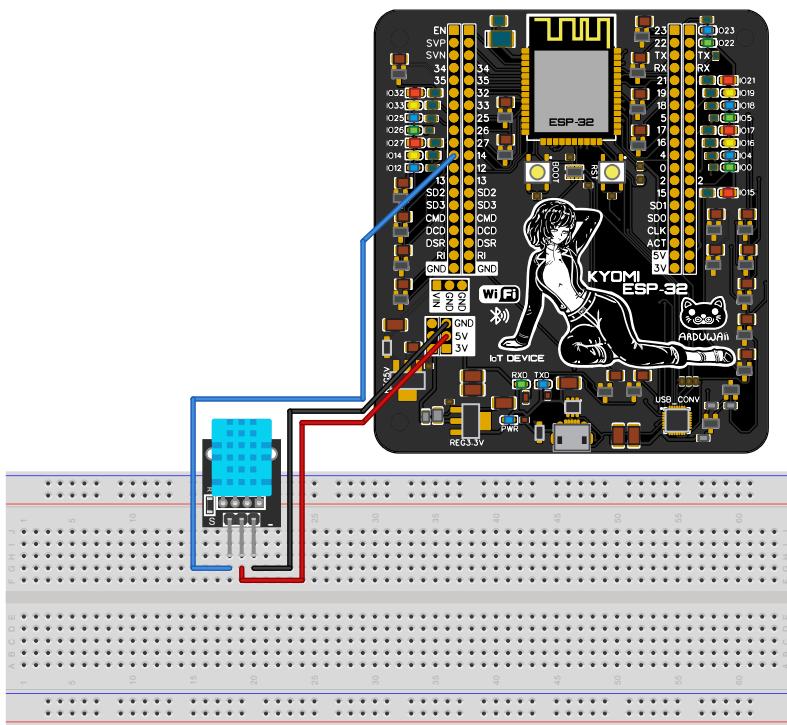
## Ejercicio 22

En este ejercicio usaremos el sensor de temperatura y humedad DHT11 para enviar sus lecturas a una página web, las cuales se van a actualizar cada 2 segundos. Configuraremos el ESP32 como servidor asíncrono para actualizar las lecturas del DHT11 en la página web, mediante JavaScript.

### Materiales:

- 1 módulo sensor DHT11 (como el KY-015), también puedes usar el DHT22
- 3 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:



Vamos a utilizar el siguiente documento HTML para nuestra página web:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Servidor DHT11</title>
  <meta name='viewport' content='width=device-width, initialscale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8' />
  <link rel='icon' href='data:, '>
```



```
<style>
    html
    {
        font-family: Corbel;
        display: inline-block;
        margin: 0px auto;
        text-align: center;
    }
    body
    {
        background-image:
url('https://i.pinimg.com/originals/9e/0b/e9/9e0be9abf2f2dfe0399f85cdc32681d2.jpg');
        background-repeat: no-repeat;
        background-attachment: fixed;
        background-size: 100% 100%;
    }
</style>
</head>

<body>
    <h1 style='color:#BAEFFF;'>Servidor de temperatura y Humedad DHT11</h1>
    <p style='color:#FFE6B3; font-size: 32px;'>
        <img src='https://image.flaticon.com/icons/png/512/1035/1035618.png' alt='TEMPERATURA' width='80' height='80'>
        Celsius:
        <span id='Lectura_temp_C'>%TEMPERATURA_C%</span>
        °C
        Fahrenheit:
        <span id='Lectura_temp_F'>%TEMPERATURA_F%</span>
        °F
    </p>
    <p style='color:#CCD4FF; font-size: 32px;'>
        <img src='https://image.flaticon.com/icons/png/512/777/777610.png' alt='HUMEDAD' width='80' height='80'>
        <span id='Lectura_humedad'>%HUMEDAD%</span>
        %
    </p>
</body>
```



```
</body>
<script>
setInterval(function()
{
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function()
    {
        if (this.readyState == 4 && this.status == 200)
        {
            document.getElementById('Lectura_temp_C').innerHTML = this.responseText;
        }
    };
    xhttp.open('GET', '/Temperatura-En-C', true);
    xhttp.send();
},2000);

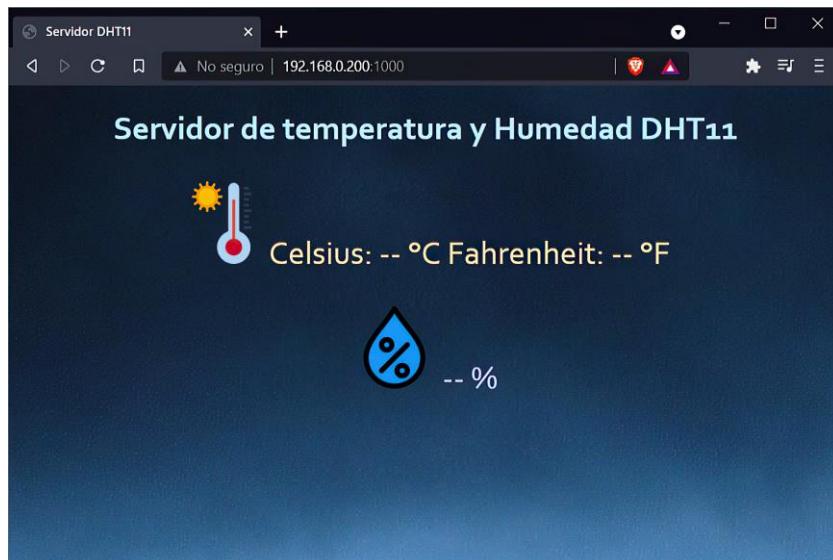
setInterval(function()
{
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function()
    {
        if (this.readyState == 4 && this.status == 200)
        {
            document.getElementById('Lectura_temp_F').innerHTML = this.responseText;
        }
    };
    xhttp.open('GET', '/Temperatura-En-F', true);
    xhttp.send();
},2000);

setInterval(function()
{
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function()
    {
        if (this.readyState == 4 && this.status == 200)
```



```
{  
    document.getElementById('Lectura_humedad').innerHTML = this.responseText;  
}  
};  
xhttp.open('GET', '/Humedad', true);  
xhttp.send();  
,2000);  
</script>  
</html>
```

Tendremos tres `setInterval()` con sus funciones respectivas para cambiar el contenido de tres marcas `<span>`, cuyos identificadores son `Lectura_temp_C`, `Lectura_temp_F` y `Lectura_humedad`. En estas marcas `<span>`, los textos `TEMPERATURA_C`, `TEMPERATURA_F` y `HUMEDAD` serán reemplazados con las lecturas de grados Celsius, Fahrenheit y porcentaje de humedad entregadas por el DHT11 (a través de peticiones HTML), usando los hipervínculos `/Temperatura-En-C`, `/Temperatura-En-F` y `/Humedad`. Todas las peticiones al servidor se hacen cada 2000ms (2 segundos). Puedes personalizar el resto del documento cambiando las imágenes para los iconos de temperatura y humedad, la imagen de fondo o el tamaño y color de los textos:



Veamos el código de Arduino. Primero incluiremos las librerías para el sensor DHT11 y para las funciones del servidor asíncrono:

```
#include <WiFi.h>  
#include "ESPAsyncWebServer.h"  
#include <Adafruit_Sensor.h>  
#include <DHT.h>
```



después declararemos las variables necesarias para realizar la conexión a nuestra red. Esta vez usaremos la instrucción `AsyncWebServer` para establecer el puerto de nuestro servidor. También usaremos una IP fija como en el ejercicio 21, en este caso será 192.168.0.200:

```
const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
AsyncWebServer server(1000);

IPAddress Local_IP(192, 168, 0, 200);
IPAddress gateway(192, 168, 200, 200);

IPAddress subnet(255, 255, 0, 0);
IPAddress primaryDNS(8, 8, 8, 8);
IPAddress secondaryDNS(8, 8, 4, 4);
```

Luego vamos a declarar el GPIO que usaremos para comunicar al sensor DHT11 (en este caso el GPIO14). También agregaremos unas cuantas variables para guardar las lecturas del DHT11. Despues configuraremos el sensor mediante la instrucción `DHT dht(GPIO_DHT, DHT11)`, donde colocaremos el pin de comunicación y el tipo de sensor (`DHT11` o `DHT22` según el sensor que tengas):

```
const int GPIO_DHT=14;
float temperatura_C=0, temperatura_F=0, humedad=0;
DHT dht(GPIO_DHT, DHT11);
```

Finalmente, vamos a guardar todo el documento HTML de nuestra página web en una variable `const char` de tipo array vacío, ya que no podremos usar una variable `String` (más adelante veremos por qué). El array vacío nos permitirá guardar muchos caracteres en la variable `char`. Esta vez no tendremos que dividir el documento HTML por secciones:

```
const char PAGINA_WEB[]="<!DOCTYPE HTML> <html> <head> <title>Servidor
DHT11</title> <meta name='viewport' content='width=device-width, initialscale=1'
http-equiv='Content-Type' content='text/html; charset=UTF-8' /> <link rel='icon'
href='data:, '> <style> html { font-family: Corbel; display: inline-block; margin:
0px auto; text-align: center; } body { background-image:
url('https://i.pinimg.com/originals/9e/0b/e9/9e0be9abf2f2dfe0399f85cdc32681d2.jpg');
background-repeat: no-repeat; background-attachment: fixed; background-size:
100% 100%; } </style> </head> <body> <h1 style='color:#BAEFFF;'>Servidor de
temperatura y Humedad DHT11</h1> <p style='color:#FFE6B3; font-size: 32px;'> <img
src='https://image.flaticon.com/icons/png/512/1035/1035618.png'
alt='TEMPERATURA' width='80' height='80'> Celsius: <span
id='Lectura_temp_C'>%TEMPERATURA_C%</span> °C Fahrenheit: <span
id='Lectura_temp_F'>%TEMPERATURA_F%</span> °F </p> <p style='color:#CCD4FF; font-
size: 32px;'> <img src='https://image.flaticon.com/icons/png/512/777/777610.png'</p>
```



```
alt='HUMEDAD' width='80' height='80'> <span id='Lectura_humedad'>%HUMEDAD%</span>
% </p> </body> <script> setInterval(function() { var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() { if (this.readyState == 4 && this.status
== 200) { document.getElementById('Lectura_temp_C').innerHTML =
this.responseText; } }; xhttp.open('GET','/Temperatura-En-C', true);
xhttp.send(); },2000); setInterval(function() { var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() { if (this.readyState == 4 && this.status
== 200) { document.getElementById('Lectura_temp_F').innerHTML =
this.responseText; } }; xhttp.open('GET','/Temperatura-En-F', true);
xhttp.send(); },2000); setInterval(function() { var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() { if (this.readyState == 4 && this.status
== 200) { document.getElementById('Lectura_humedad').innerHTML =
this.responseText; } }; xhttp.open('GET', '/Humedad', true); xhttp.send();
},2000); </script> </html>";
```

En `setup()` se ejecutarán la mayoría de las instrucciones del servidor. Primero vamos a configurar la comunicación serial, el pin del sensor DHT11 y luego inicializaremos el sensor con `dht.begin()`:

```
Serial.begin(115200);
pinMode(GPIO_DHT, INPUT);
dht.begin();
```

después colocaremos la rutina de conexión que ya conoces:

```
WiFi.config(Local_IP, gateway, subnet, primaryDNS, secondaryDNS);
Serial.print("Conectando a:");
Serial.println(ssid);
WiFi.begin(ssid, password);
while(WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
}
Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.localIP());
```

luego, con las siguientes instrucciones manejaremos las peticiones del navegador cliente y le enviaremos las respuestas:

```
server.on("/", HTTP_GET, [](AsyncWebRequest *request)
{
    request->send_P(200, "text/html", PAGINA_WEB, ENVIAR_DATOS);
});
```



```
server.on("/Temperatura-En-C", HTTP_GET, [](AsyncWebServerRequest *request)
{
    request->send_P(200, "text/plain", TEMPERATURA_DHT_C().c_str());
});

server.on("/Temperatura-En-F", HTTP_GET, [](AsyncWebServerRequest *request)
{
    request->send_P(200, "text/plain", TEMPERATURA_DHT_F().c_str());
};

server.on("/Humedad", HTTP_GET, [](AsyncWebServerRequest *request)
{
    request->send_P(200, "text/plain", HUMEDAD_DHT().c_str());
});
```

por ejemplo, en `server.on("/", HTTP_GET, [](AsyncWebServerRequest *request)`, el servidor va esperar una petición con el hipervínculo `/`, que es cuando el cliente pide el archivo raíz o página web principal a nuestro servidor. Cuando nuestro servidor recibe esta petición, enviará una respuesta con la instrucción `request->send_P(200, "text/html", PAGINA_WEB, ENVIAR_DATOS)`. En esta respuesta enviaremos el contenido de `PAGINA_WEB` y un resultado que entrega la subrutina `ENVIAR_DATOS` (luego para qué funciona). Ya que la respuesta enviada es de tipo `text/html`, si usamos valores `String` habría un error en la conversión de texto a HTML. El código '200' es el mismo que usamos en `this.status` (del documento HTML) para indicar que la petición fue correcta.

Para enviar las lecturas del sensor, por ejemplo, en `server.on("/Temperatura-En-C", HTTP_GET, [](AsyncWebServerRequest *request)`, el servidor va esperar una petición con el hipervínculo `/Temperatura-En-C` cuando el cliente solicite el dato de temperatura en grados Celsius. Mediante `request->send_P(200, "text/plain", TEMPERATURA_DHT_C().c_str())` enviaremos la respuesta con el dato de temperatura, que es entregado por la subrutina `TEMPERATURA_DHT_C`. La respuesta es de tipo texto plano (`text/plain`), así que el valor enviado puede ser de tipo `String`, aunque será convertido a `char` mediante `c_str()`.

Finalmente arrancaremos el servidor para terminar la estructura de `setup()`:

```
server.begin();
```

En `Loop()` prácticamente no colocaremos nada.



La subrutina `ENVIAR_DATOS` es la que se encargará de reemplazar los valores de temperatura y humedad en la página web (específicamente en las respuestas enviadas):

```
String ENVIAR_DATOS(const String& var)
{
    if(var == "TEMPERATURA_C")
    {
        return TEMPERATURA_DHT_C();
    }
    else if(var == "TEMPERATURA_F")
    {
        return TEMPERATURA_DHT_F();
    }
    else if(var == "HUMEDAD")
    {
        return HUMEDAD_DHT();
    }
    return String();
}
```

En el documento HTML usamos `var` para crear la variable `xhttp`, que funcionaba como un objeto dependiente de `XMLHttpRequest()`. En esta variable también se encuentra la petición que hace el cliente al servidor. Cuando el servidor recibe la petición, en la subrutina `ENVIAR_DATOS()`, los `if` y `else if` buscarán dentro de la petición (en forma de variable `var`) el texto que reemplazarán por las lecturas del sensor. En el caso de la temperatura Celsius, el `if(var == "TEMPERATURA_C")` buscará en la petición el texto `TEMPERATURA_C`, que es el mismo que colocamos entre % en el documento HTML (entre las etiquetas `<span id='Lectura_temp_C'>%TEMPERATURA_C%</span>`). Este texto en específico es el que será reemplazado con el valor devuelto por la subrutina `TEMPERATURA_DHT_C()`, mediante `return TEMPERATURA_DHT_C()`. Algo similar ocurre con el resto de `else if`, que reemplazan los valores de temperatura Fahrenheit y humedad.

Las subrutas `TEMPERATURA_DHT_C()`, `TEMPERATURA_DHT_F()` y `HUMEDAD_DHT()` obtienen las lecturas del sensor DHT11, para devolverlas a la subrutina `ENVIAR_DATOS()` y a `server.on` en forma de `String`.



Por ejemplo, en la subrutina `TEMPERATURA_DHT_C`, dentro de la variable `temperatura_C` guardamos el valor de la lectura Celsius, entregado por `dht.readTemperature()`. Si el valor de `temperatura_C` es `nan` (indeterminado), entonces el valor devuelto a la respuesta HTTP será `--`. Si no es el caso, la rutina devolverá el valor de `temperatura_C`. También enviaremos estas lecturas al Monitor Serie. Generalmente, los valores `nan` aparecen cuando el sensor aún no está listo para enviar las lecturas. El DHT11 es un sensor lento, y tarda al menos dos segundos en generar lecturas nuevas (es el tiempo que usamos en `setInterval()`):

```
String TEMPERATURA_DHT_C()
{
    temperatura_C = dht.readTemperature();
    if(isnan(temperatura_C))
    {
        Serial.println("Lectura errónea de temp. Celsius");
        return "--";
    }
    else
    {
        Serial.println(temperatura_C);
        return String(temperatura_C);
    }
}

String TEMPERATURA_DHT_F()
{
    temperatura_F = dht.readTemperature(true);
    if(isnan(temperatura_F))
    {
        Serial.println("Lectura errónea de temp. Farenheit");
        return "--";
    }
    else
    {
        Serial.println(temperatura_F);
        return String(temperatura_F);
    }
}
```



```
String HUMEDAD_DHT()
{
    humedad = dht.readHumidity();
    if(isnan(humedad))
    {
        Serial.println("Lectura errónea de humedad");
        return "--";
    }
    else
    {
        Serial.println(humedad);
        return String(humedad);
    }
}
```

Una observación importante es que, con el sensor DHT11, podemos obtener la temperatura Fahrenheit cuando agregamos el parámetro `true` a `dht.readTemperature`, quedando como `dht.readTemperature(true)`. Con `dht.readHumidity` obtenemos el porcentaje de humedad del ambiente.

El código completo queda de la siguiente forma:

```
#include <WiFi.h>
#include "ESPAsyncWebServer.h"
#include <Adafruit_Sensor.h>
#include <DHT.h>

const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
AsyncWebServer server(1000);

IPAddress Local_IP(192, 168, 0, 200);
IPAddress gateway(192, 168, 200, 200);

IPAddress subnet(255, 255, 0, 0);
IPAddress primaryDNS(8, 8, 8, 8);
IPAddress secondaryDNS(8, 8, 4, 4);

const int GPIO_DHT=14;
float temperatura_C=0, temperatura_F=0, humedad=0;
DHT dht(GPIO_DHT, DHT11);
```



```
const char PAGINA_WEB[]="<!DOCTYPE HTML> <html> <head> <title>Servidor DHT11</title> <meta name='viewport' content='width=device-width, initialscale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8' /> <link rel='icon' href='data:, '> <style> html { font-family: Corbel; display: inline-block; margin: 0px auto; text-align: center; } body { background-image: url('https://i.pinimg.com/originals/9e/0b/e9/9e0be9abf2f2dfe0399f85cdc32681d2.jpg'); background-repeat: no-repeat; background-attachment: fixed; background-size: 100% 100%; } </style> </head> <body> <h1 style='color:#BAEFFF;'>Servidor de temperatura y Humedad DHT11</h1> <p style='color:#FFE6B3; font-size: 32px;'> <img src='https://image.flaticon.com/icons/png/512/1035/1035618.png' alt='TEMPERATURA' width='80' height='80'> Celsius: <span id='Lectura_temp_C'>%TEMPERATURA_C%</span> °C Fahrenheit: <span id='Lectura_temp_F'>%TEMPERATURA_F%</span> °F </p> <p style='color:#CCD4FF; font-size: 32px;'> <img src='https://image.flaticon.com/icons/png/512/777/777610.png' alt='HUMEDAD' width='80' height='80'> <span id='Lectura_humedad'>%HUMEDAD%</span> % </p> </body> <script> setInterval(function() { var xhttp = new XMLHttpRequest(); xhttp.onreadystatechange = function() { if (this.readyState == 4 && this.status == 200) { document.getElementById('Lectura_temp_C').innerHTML = this.responseText; } }; xhttp.open('GET','/Temperatura-En-C', true); xhttp.send(); },2000); setInterval(function() { var xhttp = new XMLHttpRequest(); xhttp.onreadystatechange = function() { if (this.readyState == 4 && this.status == 200) { document.getElementById('Lectura_temp_F').innerHTML = this.responseText; } }; xhttp.open('GET','/Temperatura-En-F', true); xhttp.send(); },2000); setInterval(function() { var xhttp = new XMLHttpRequest(); xhttp.onreadystatechange = function() { if (this.readyState == 4 && this.status == 200) { document.getElementById('Lectura_humedad').innerHTML = this.responseText; } }; xhttp.open('GET', '/Humedad', true); xhttp.send(); },2000); </script> </html>";
```

```
void setup()
{
    Serial.begin(115200);
    pinMode(GPIO_DHT, INPUT);
    dht.begin();

    WiFi.config(Local_IP, gateway, subnet, primaryDNS, secondaryDNS);
    Serial.print("Conectando a:");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
}
```



```
Serial.println("");
Serial.println("Conectado");
Serial.println("Dirección IP: ");
Serial.println(WiFi.localIP());

server.on("/", HTTP_GET, [](AsyncWebRequest *request)
{request->send_P(200, "text/html", PAGINA_WEB, ENVIAR_DATOS);});

server.on("/Temperatura-En-C", HTTP_GET, [](AsyncWebRequest *request)
{request->send_P(200, "text/plain", TEMPERATURA_DHT_C().c_str());});

server.on("/Temperatura-En-F", HTTP_GET, [](AsyncWebRequest *request)
{request->send_P(200, "text/plain", TEMPERATURA_DHT_F().c_str());});

server.on("/Humedad", HTTP_GET, [](AsyncWebRequest *request)
{request->send_P(200, "text/plain", HUMEDAD_DHT().c_str());});
server.begin();
}

void Loop()
{

}

String ENVIAR_DATOS(const String& var)
{
if(var == "TEMPERATURA_C")
{
    return TEMPERATURA_DHT_C();
}

else if(var == "TEMPERATURA_F")
{
    return TEMPERATURA_DHT_F();
}
```



```
else if(var == "HUMEDAD")
{
    return HUMEDAD_DHT();
}
return String();}

String TEMPERATURA_DHT_C()
{
    temperatura_C = dht.readTemperature();
    if(isnan(temperatura_C))
    {
        Serial.println("Lectura errónea de temp. Celsius");
        return "--";
    }
    else
    {
        Serial.println(temperatura_C);
        return String(temperatura_C);
    }
}

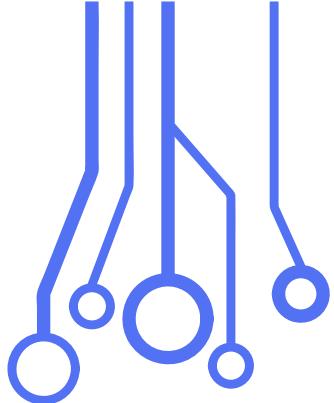
String TEMPERATURA_DHT_F()
{
    temperatura_F = dht.readTemperature(true);
    if(isnan(temperatura_F))
    {
        Serial.println("Lectura errónea de temp. Farenheit");
        return "--";
    }
    else
    {
        Serial.println(temperatura_F);
        return String(temperatura_F);
    }
}
```



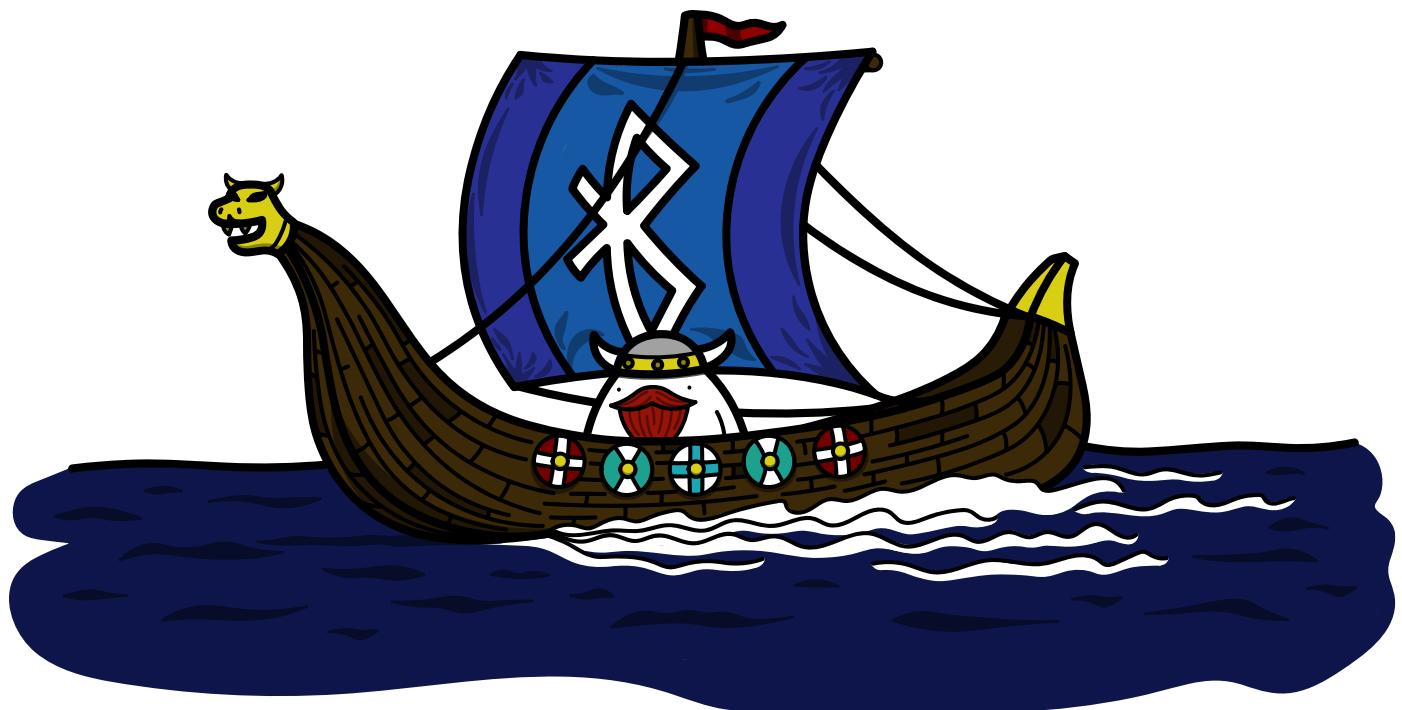
```
String HUMEDAD_DHT()
{
    humedad = dht.readHumidity();
    if(isnan(humedad))
    {
        Serial.println("Lectura errónea de humedad");
        return "--";
    }
    else
    {
        Serial.println(humedad);
        return String(humedad);
    }
}
```

Prueba el programa. Puedes acceder al servidor localmente con la IP fija o conectarlo a ngrok como en el ejercicio 21 (para poder acceder desde internet). Las lecturas que se muestran en la página web cambiarán cada dos segundos, si usas ngrok puede que tarden un poco más:





# Capítulo 5: Bluetooth clásico y BLE





## ¿Qué es Bluetooth?

Bluetooth es una tecnología de comunicación inalámbrica que puede transmitir datos de un dispositivo a otro, en una conexión punto a punto. Tiene un bajo consumo energético y es capaz de enviar y recibir muchos tipos de datos, desde mensajes simples hasta video y audio. Es común encontrarlo en dispositivos complejos como teléfonos inteligentes o computadoras, pero también en dispositivos sencillos, como teclados inalámbricos o llaves electrónicas.

La comunicación Bluetooth puede ser de tipo ‘clásica’ o ‘Bluetooth de baja energía’ (BLE). El ESP32 soporta ambos tipos.

## Bluetooth de baja energía (BLE)

Esta tecnología viene integrada desde la especificación Bluetooth 4.0. Fue diseñada para comunicar dispositivos con fuentes de energía muy limitadas, que generalmente se alimentan con pilas pequeñas (de tipo botón) o que deben mantenerse en comunicación por un tiempo prolongado. Comúnmente, BLE se encuentra en relojes inteligentes, controles remotos, sensores o llaves Bluetooth, que transmiten unos pocos datos al dispositivo cliente o destino.

La arquitectura Bluetooth BLE se compone de varias capas, pero utiliza dos protocolos de comunicación principales: GAP y GATT.

### Protocolo GAP

El protocolo GAP (Perfil Genérico de Acceso) es el que permite que nuestro dispositivo Bluetooth sea visible para los clientes Bluetooth cercanos.

En GAP, a los dispositivos Bluetooth se les puede asignar uno de cuatro roles durante la conexión:

- **EMISOR** (Broadcaster): es quien transmite paquetes de información a los ‘observadores’ para hacerse presente.
- **OBSERVADOR** (Observer): es quien escanea el entorno en busca de ‘emisores’, pero sin conectarse a éstos.
- **PERIFÉRICO** (Peripheral Device): son dispositivos simples que pueden establecer conexiones con los dispositivos ‘centrales’ para enviarles datos, pero actuarán como dispositivos ‘esclavos’ y no podrán administrar la conexión.
- **DISPOSITIVO CENTRAL** (Central Device): es quien permite y administra la conexión con el ‘periférico’, además de controlarlo.

La asignación roles se puede ejemplificar de la siguiente manera: al encender un teclado Bluetooth, este funciona como emisor para hacerse



visible ante los observadores. El dispositivo observador será un celular. Cuando el celular busca dispositivos Bluetooth, funciona como observador. Al detectar al emisor, el observador podrá enviarle una solicitud de conexión y el emisor cambiará a modo periférico, para aceptar la conexión. Ahora el periférico está conectado y el observador pasa al modo de dispositivo central, bloqueando cualquier otra conexión entrante (para que otro celular no se conecte al mismo tiempo, por ejemplo). El dispositivo central actúa como ‘cliente’ y puede solicitarle datos al periférico, que actúa como ‘servidor’.

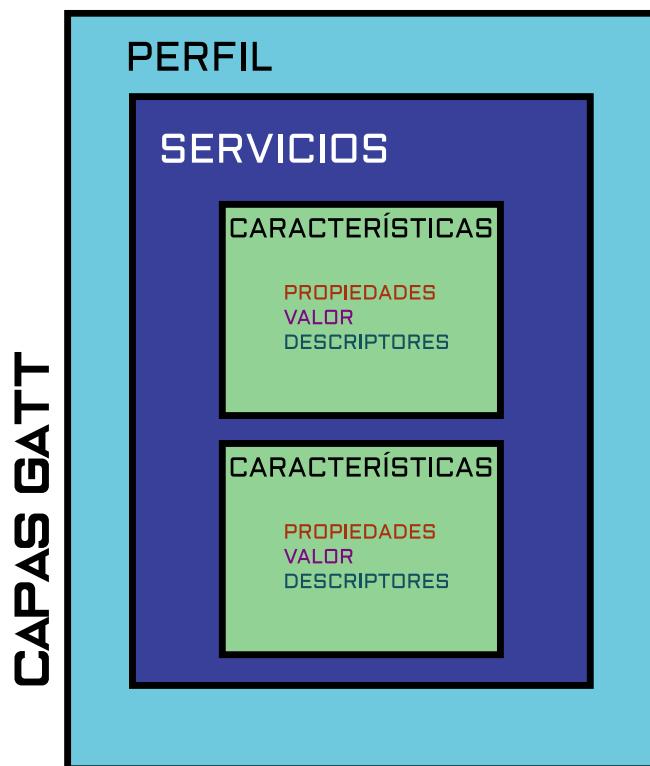
Una vez termina el proceso de conexión con GAP, ambos dispositivos podrán empezar a transmitir y recibir información usando el protocolo GATT.

## Protocolo GATT

El protocolo GATT (Perfil de Atributo Genérico) es usado por los dispositivos BLE para la transmisión y recepción de datos, en paquetes de información pequeños.

En este protocolo, la información va organizada en las capas de datos llamadas ‘servicios’ y ‘características’. Los dispositivos BLE pueden tener distintos servicios y características, dependiendo del tipo de datos que son capaces de transmitir.

Las capas de datos del protocolo GATT se organizan de la siguiente manera:





**PERFIL:** es la capa más externa y contiene a los servicios del dispositivo BLE. Los servicios ‘estándar’ que pueda tener dependen del tipo de dispositivo y fabricante, por ejemplo, muchos sensores de presión arterial tienen servicios similares.

**SERVICIOS:** son los conjuntos de información que puede enviar y recibir el dispositivo BLE. Estos conjuntos se subdividen en características. Por ejemplo, un sensor de presión arterial puede tener un servicio para el dato de la presión, otro para los pulsos del corazón y también uno con el estado de la batería.

**CARACTERÍSTICAS:** contienen los datos individuales del servicio. Por ejemplo, un sensor acelerómetro puede tener un servicio con los datos de posición y otro con el estado de la batería. El servicio con los datos de posición se puede dividir en tres características: una que contenga el valor de posición en el eje X, otra el valor de posición en el eje Y y otra con el valor de posición en el eje Z. Las características también contienen ciertas propiedades y ‘descriptores’.

Los servicios y las características tienen un identificador universal único o UUID, que es un valor alfanumérico de 16 bits. Muchos UUID's forman parte del estándar Bluetooth, para identificar servicios y características comunes, pero también puedes crear tus propios UUID's, dependiendo de cómo programes a tu ESP32 en modo BLE.

El ESP32 comúnmente se usa como servidor BLE, pero también puede funcionar como cliente. También se puede configurar el BLE para conexiones especiales con múltiples dispositivos a la vez, aunque es una función poco utilizada.

**IMPORTANTE:** algo que no puede hacer BLE es transmitir grandes cantidades de datos, por ejemplo, transmitir audio. BLE está diseñado para enviar y recibir paquetes de información muy pequeños. Si necesitas transmitir muchos datos, debes optar por Bluetooth clásico.

## BLE en modo Servidor

Podemos crear un servidor BLE sencillo con el ESP32 para enviar datos a un cliente. Básicamente, debemos configurar un servicio con una característica que contenga el dato a enviar.

Con el siguiente ejemplo crearemos un servidor BLE. Primero debemos incluir las librerías necesarias para crear el servidor:

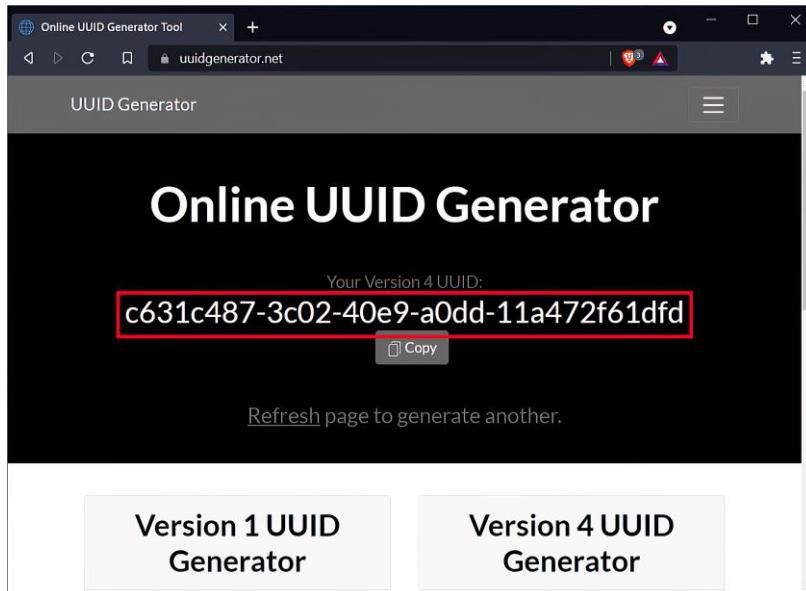
```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
```



Necesitaremos unas definiciones que contengan un UUID genérico para crear un servicio y otros UUID's para crear un par de características:

```
#define UUID_SERVICIO          "4fafc201-1fb5-459e-8fcc-c5c9c331914b"  
#define UUID_CARACTERISTICA_1   "beb5483e-36e1-4688-b7f5-ea07361b26a8"  
#define UUID_CARACTERISTICA_2   "c631c487-3c02-40e9-a0dd-11a472f61dfd"
```

En varios sitios web podremos generar UUID's aleatorios o elegir algunos predefinidos para servicios y características estándar:



Dentro de `setup()` vamos a configurar el servidor BLE. En `BLEDevice::init()` colocaremos el nombre de dispositivo de nuestro servidor BLE, que es el que se mostrará al escanear dispositivos Bluetooth con nuestro celular o computadora:

```
BLEDevice::init("Servidor ESP32");  
BLEServer *Servidor1 = BLEDevice::createServer();  
BLEService *Servicio_A = Servidor1->createService(UUID_SERVICIO);
```

con `BLEServer` creamos un servidor llamado `Servidor1`, al cual le agregaremos los servicios y características. Con `BLEService` vamos a crear un servicio llamado `Servicio_A` (que pertenece a `Servidor1`) y este servicio tendrá el UUID que definimos anteriormente (`UUID_SERVICIO`).

Ahora vamos a crear un par de características:

```
BLECharacteristic *Caracteristica_1 = Servicio_A->createCharacteristic  
(UUID_CARACTERISTICA_1,  
BLECharacteristic::PROPERTY_READ |  
BLECharacteristic::PROPERTY_WRITE);  
Caracteristica_1->setValue("Mensaje 1 del ESP32");
```



```
BLECharacteristic *Caracteristica_2 = Servicio_A->createCharacteristic  
(UUID_CARACTERISTICA_2,  
    BLECharacteristic::PROPERTY_READ |  
    BLECharacteristic::PROPERTY_WRITE);  
Caracteristica_2->setValue("Mensaje 2 del ESP32");
```

para crear la primera característica, usamos `BLECharacteristic` y le damos el nombre `Caracteristica_1`, que estará asociada a `Servicio_A` y tendrá uno de los UUID's que definimos anteriormente (`UUID_CARACTERISTICA_1`).

Con `BLECharacteristic::PROPERTY_READ` y `BLECharacteristic::PROPERTY_WRITE` habilitamos lectura y escritura de datos en la característica. La instrucción `setValue()` nos permite guardar un valor en la característica, por ejemplo, guardamos el texto `Mensaje 1 del ESP32` en `Caracteristica_1`. Usaremos la misma estructura para crear más características.

Finalmente, iniciaremos el servicio `Servicio_A` (con `Servicio_A->start()`) y configuraremos el servidor BLE para que empiece a enviar los datos:

```
Servicio_A->start();  
  
BLEAdvertising *enviarMensaje = BLEDevice::getAdvertising();  
enviarMensaje->addServiceUUID(UUID_SERVICIO);  
enviarMensaje->setScanResponse(true);  
enviarMensaje->setMinPreferred(0x06);  
enviarMensaje->setMinPreferred(0x12);  
  
BLEDevice::startAdvertising();
```

con la instrucción `BLEAdvertising` crearemos la función `enviarMensaje`. A esta función le asociaremos el servicio que creamos anteriormente, mediante `addServiceUUID(UUID_SERVICIO)`. Con `setScanResponse(true)`, la función `enviarMensaje` estará configurada para enviar respuestas a los clientes Bluetooth durante el escaneo de dispositivos. Las configuraciones con `setMinPreferred()` se agregan para mejorar la compatibilidad con dispositivos iPhone o con sistema IOS. Con `BLEDevice::startAdvertising`, el servidor BLE empezará a enviar paquetes de datos a los clientes Bluetooth.

En `loop()` no será necesario colocar instrucciones.

El código completo queda de la siguiente forma:

```
#include <BLEDevice.h>  
 #include <BLEUtils.h>  
 #include <BLEServer.h>
```



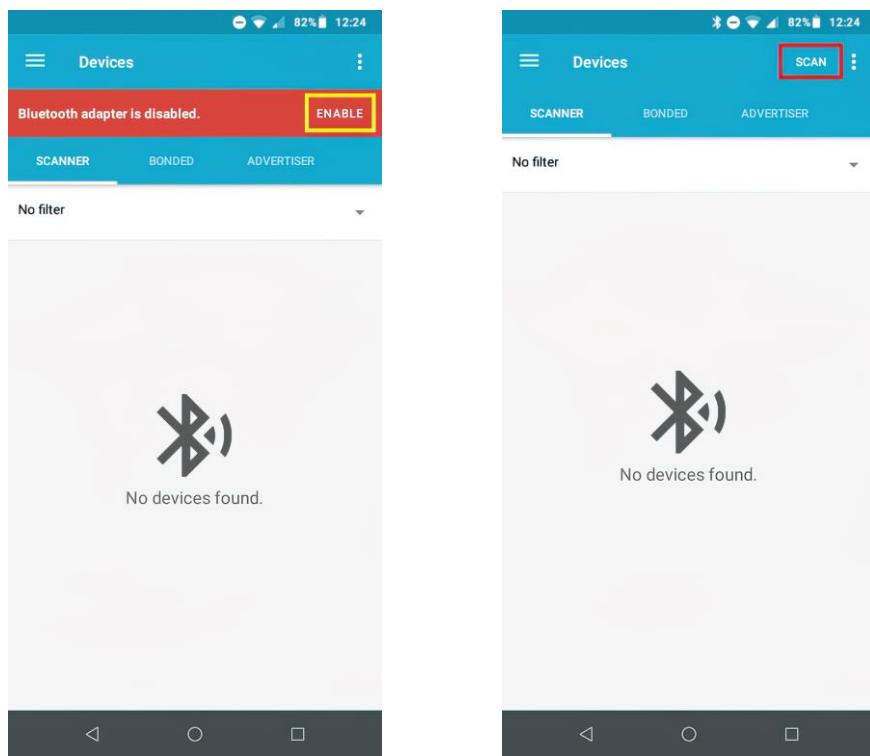
```
#define UUID_SERVICIO          "4fafc201-1fb5-459e-8fcc-c5c9c331914b"  
#define UUID_CARACTERISTICA_1   "beb5483e-36e1-4688-b7f5-ea07361b26a8"  
#define UUID_CARACTERISTICA_2   "c631c487-3c02-40e9-a0dd-11a472f61dfd"  
  
void setup()  
{  
    BLEDevice::init("Servidor ESP32");  
    BLEServer *Servidor1 = BLEDevice::createServer();  
    BLEService *Servicio_A = Servidor1->createService(UUID_SERVICIO);  
  
    BLECharacteristic *Caracteristica_1 = Servicio_A->createCharacteristic  
(UUID_CARACTERISTICA_1,  
     BLECharacteristic::PROPERTY_READ |  
     BLECharacteristic::PROPERTY_WRITE);  
    Caracteristica_1->setValue("Mensaje 1 del ESP32");  
  
    BLECharacteristic *Caracteristica_2 = Servicio_A->createCharacteristic  
(UUID_CARACTERISTICA_2,  
     BLECharacteristic::PROPERTY_READ |  
     BLECharacteristic::PROPERTY_WRITE);  
    Caracteristica_2->setValue("Mensaje 2 del ESP32");  
  
    Servicio_A->start();  
    BLEAdvertising *enviarMensaje = BLEDevice::getAdvertising();  
    enviarMensaje->addServiceUUID(UUID_SERVICIO);  
    enviarMensaje->setScanResponse(true);  
    enviarMensaje->setMinPreferred(0x06);  
    enviarMensaje->setMinPreferred(0x12);  
    BLEDevice::startAdvertising();  
}  
void Loop()  
{  
}
```

Ahora carga el programa a tu tarjeta. Para verificar su funcionamiento, usaremos un celular o Tablet con Bluetooth 4.0 y la aplicación ‘nRF Connect’, que está disponible para Android y iOS. En este caso usaremos un dispositivo Android.

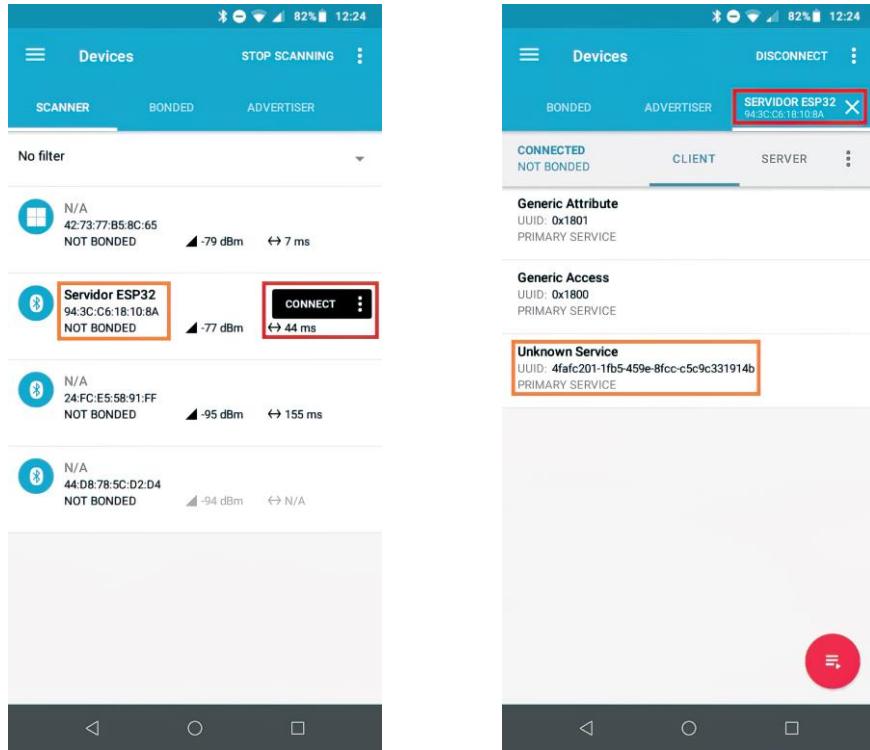


## Capítulo 5: Bluetooth clásico y BLE

Una vez descargada la aplicación, ábrela y activa el Bluetooth para escanear dispositivos cercanos:

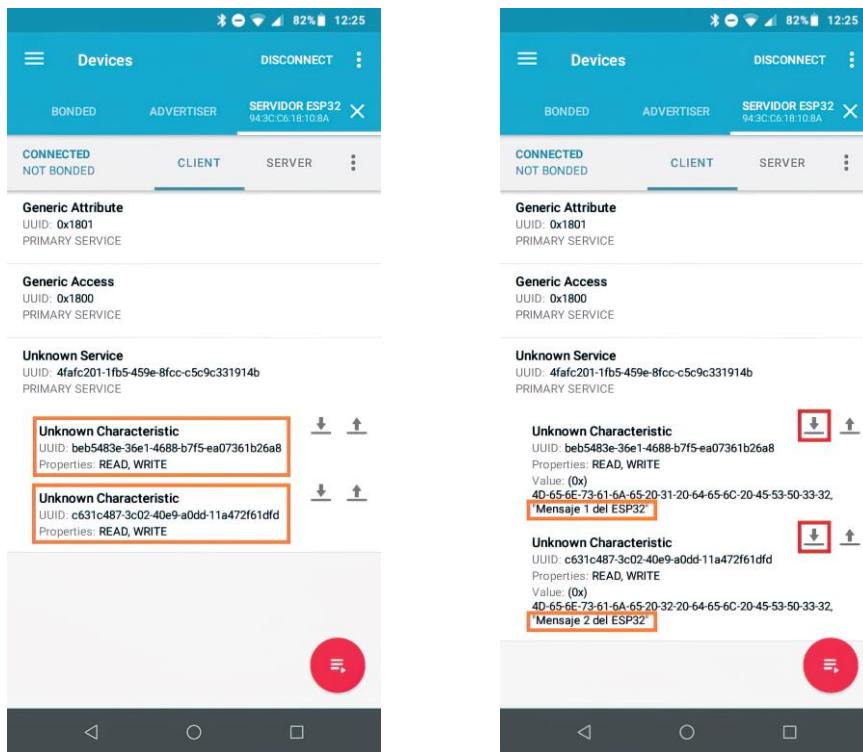


Cuando aparezca el ESP32, conéctalo para habilitar una nueva pestaña donde se mostrará el servicio que definimos en el programa, verás que aparecerá su UUID:





Abre el servicio para que se muestren las dos características que agregamos en el programa. Si despliegas cada característica, aparecerán los textos que guardamos con `setValue()`:



Por ahora el servidor BLE sólo envía paquetes de datos cuando el cliente los solicita (al desplegar las características en nRF Connect). Para enviar datos al cliente constantemente (sin que tenga que solicitarlos), debemos configurar el servidor en modo de ‘notificación’.

## BLE en modo Notificación

Al configurar nuestro servidor BLE en modo de notificación, podremos enviar datos al cliente cada cierto tiempo. Esto puede ser útil para enviar lecturas de sensores o valores de alguna variable.

Con el siguiente ejemplo vamos a configurar un servidor BLE en modo de notificación. Primero incluimos las librerías necesarias, esta vez agregaremos la librería `BLE2902.h` para usar las funciones de notificación:

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>
```



Crearemos un par de definiciones para guardar los UUID's que usaremos para un servicio y una característica:

```
#define UUID_SERVICIO      "4fafc201-1fb5-459e-8fcc-c5c9c331914b"  
#define UUID_CARACTERISTICA_1  "beb5483e-36e1-4688-b7f5-ea07361b26a8"
```

Colocaremos a `BLEServer` y `BLECharacteristic` en la declaración de variables para crear a `Servidor1` y `Caracteristica_1`, pero sin configurarlos (`NULL`):

```
BLEServer *Servidor1 = NULL;  
BLECharacteristic *Caracteristica_1 = NULL;
```

Las siguientes variables booleanas las usaremos más adelante, mientras que el valor de la variable `Dato` es el que estaremos enviando al cliente:

```
bool Dispositivo_conectado = false;  
bool Anterior_dispositivo_conectado = false;  
int Dato = 0;
```

Dentro de `setup()` configuraremos el servidor BLE. Primero habilitaremos la comunicación serial. En `BLEDevice::init` colocaremos el nombre de nuestro servidor BLE e iniciaremos `Servidor1` con `BLEDevice::createServer()`:

```
Serial.begin(115200);  
BLEDevice::init("Notificaciones ESP32");  
Servidor1 = BLEDevice::createServer();  
  
Servidor1->setCallbacks(new Llamadas_al_servidor());  
BLEService *Servicio_Notificaciones = Servidor1->createService(UUID_SERVICIO);
```

a `Servidor1` lo asociaremos con la clase `Llamadas_al_servidor()` (que es similar a una subrutina), para que la ejecute cuando un dispositivo intente conectarse al servidor (más adelante veremos cómo funciona). También vamos a crear un servicio llamado `Servicio_Notificaciones` y tendrá el UUID que definimos anteriormente (`UUID_SERVICIO`).

Luego vamos a configurar las propiedades de `Caracteristica_1` y le asignaremos su UUID:

```
Caracteristica_1 = Servicio_Notificaciones->createCharacteristic  
(UUID_CARACTERISTICA_1,  
BLECharacteristic::PROPERTY_READ  |  
BLECharacteristic::PROPERTY_WRITE |  
BLECharacteristic::PROPERTY_NOTIFY |  
BLECharacteristic::PROPERTY_INDICATE);  
  
Caracteristica_1->addDescriptor(new BLE2902());
```



Además configurar a `Caracteristica_1` con `BLECharacteristic::PROPERTY_READ` y `BLECharacteristic::PROPERTY_WRITE`, agregaremos las propiedades `BLECharacteristic::PROPERTY_NOTIFY` y `BLECharacteristic::PROPERTY_INDICATE` para enviar los datos de `Caracteristica_1` en modo notificación. También debemos configurar un descriptor tipo `BLE2902` para `Caracteristica_1` (con `Caracteristica_1->addDescriptor(new BLE2902())`) y así poder cambiar sus valores guardados en modo notificación.

Iniciaremos a `Servicio_Noticiones` con `Servicio_Noticaciones->start()`:

```
Servicio_Noticaciones->start();  
BLEAdvertising *enviarMensaje = BLEDevice::getAdvertising();  
enviarMensaje->addServiceUUID(UUID_SERVICIO);  
enviarMensaje->setScanResponse(false);  
enviarMensaje->setMinPreferred(0x0);  
BLEDevice::startAdvertising();  
Serial.println("Esperando clientes...");
```

nuevamente, con `BLEAdvertising` crearemos la función `enviarMensaje`, para asociarla a `Servicio_Noticaciones`. Esta vez vamos a configurar `setScanResponse()` como `false`. La configuración de `setMinPreferred()` también se usará para mejorar la compatibilidad con algunos dispositivos Bluetooth. Con `BLEDevice::startAdvertising`, el servidor BLE empezará a enviar paquetes de datos a los clientes Bluetooth. También enviaremos un mensaje al Monitor Serie durante la espera de clientes.

En `Loop()` vamos a colocar las instrucciones que enviarán los datos al cliente que esté conectado a nuestro servidor:

```
if(Dispositivo_conectado)  
{  
    Caracteristica_1->setValue(Dato);  
    Caracteristica_1->notify();  
    Dato++;  
    delay(1000);  
}  
if(!Dispositivo_conectado && Anterior_dispositivo_conectado)  
{  
    delay(500);  
    Servidor1->startAdvertising();  
    Serial.println("Enviando datos");  
    Anterior_dispositivo_conectado = Dispositivo_conectado;  
}
```



```
if(Dispositivo_conectado && !Anterior_dispositivo_conectado)
{
    Anterior_dispositivo_conectado = Dispositivo_conectado;
}
```

el primer `if` se ejecutará cuando `Dispositivo_conectado` sea igual a `true`, es decir, cuando un cliente se conecte al servidor BLE. Es entonces que `Caracteristica_1` tomará el valor de la variable `Dato` (`Caracteristica_1->setValue(Dato)`) y será enviado al cliente mediante `Caracteristica_1->notify()`. Luego se sumará un 1 a `Dato` y habrá una pausa de 1 segundo antes de volver a ejecutar este `if` (siempre y cuando no se desconecte el cliente).

En el segundo `if`, si `Dispositivo_conectado` ahora es igual a `false` y `Anterior_dispositivo_conectado` es igual a `true`, es decir, que el cliente actual se desconectó y luego se conectó un cliente nuevo o anterior, entonces se reiniciará la comunicación de `Servidor1` con `Servidor1->startAdvertising()` y `Anterior_dispositivo_conectado` tomará el lugar de `Dispositivo_conectado`.

En el tercer `if` ocurre una comprobación que es redundante a la del segundo `if`. Si `Dispositivo_conectado` es `true` y `Anterior_dispositivo_conectado` ahora es `false`, es decir, que el cliente actual no se ha desconectado pero el anterior sí, entonces `Anterior_dispositivo_conectado` tomará el lugar de `Dispositivo_conectado`.

La clase `Llamadas_al_servidor` se encarga de detectar a los clientes que se conecten al servidor BLE. Dentro de esta clase se encuentran las subrutinas con las funciones `onConnect` y `onDisconnect` (asociadas a `Servidor1`), que cambian el estado de `Dispositivo_conectado` a `true` o `false` cuando un cliente se conecta o desconecta del servidor:

```
class Llamadas_al_servidor: public BLEServerCallbacks
{
    void onConnect(BLEServer* Servidor1)
    {
        Dispositivo_conectado = true;
    };
    void onDisconnect(BLEServer* Servidor1)
    {
        Dispositivo_conectado = false;
    }
};
```

Esta clase se coloca antes de `setup()`.



El código completo queda de la siguiente forma:

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>

#define UUID_SERVICIO          "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define UUID_CARACTERISTICA_1   "beb5483e-36e1-4688-b7f5-ea07361b26a8"

BLEServer *Servidor1 = NULL;
BLECharacteristic *Caracteristica_1 = NULL;

bool Dispositivo_conectado = false;
bool Anterior_dispositivo_conectado = false;
int Dato = 0;

class Llamadas_al_servidor: public BLEServerCallbacks
{
    void onConnect(BLEServer* Servidor1)
    {
        Dispositivo_conectado = true;
    }

    void onDisconnect(BLEServer* Servidor1)
    {
        Dispositivo_conectado = false;
    }
};

void setup()
{
    Serial.begin(115200);
    BLEDevice::init("Notificaciones ESP32");
    Servidor1 = BLEDevice::createServer();

    Servidor1->setCallbacks(new Llamadas_al_servidor());
```



```
BLEService *Servicio_Noticiones = Servidor1->createService(UUID_SERVICIO);

Caracteristica_1 = Servicio_Noticiones->createCharacteristic
(UUID_CARACTERISTICA_1,
BLECharacteristic::PROPERTY_READ  |
BLECharacteristic::PROPERTY_WRITE | 
BLECharacteristic::PROPERTY_NOTIFY | 
BLECharacteristic::PROPERTY_INDICATE);

Caracteristica_1->addDescriptor(new BLE2902());

Servicio_Noticaciones->start();

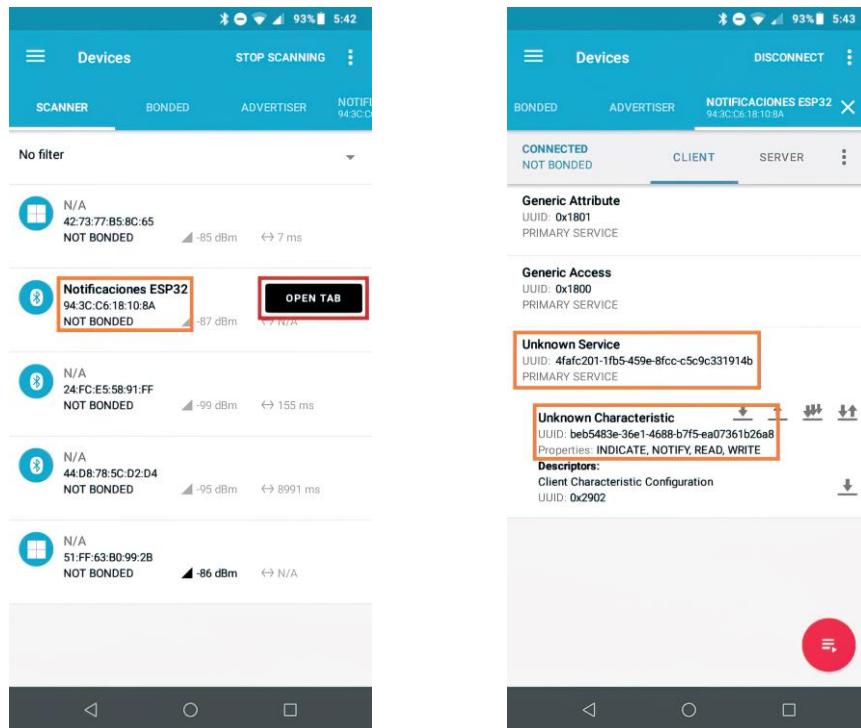
BLEAdvertising *enviarMensaje = BLEDevice::getAdvertising();
enviarMensaje->addServiceUUID(UUID_SERVICIO);
enviarMensaje->setScanResponse(false);
enviarMensaje->setMinPreferred(0x0);
BLEDevice::startAdvertising();
Serial.println("Esperando clientes...");
}

void Loop()
{
    if(Dispositivo_conectado)
    {
        Caracteristica_1->setValue(Dato);
        Caracteristica_1->notify();
        Dato++;
        deDelay(1000);
    }
    if(!Dispositivo_conectado && Anterior_dispositivo_conectado)
    {
        deDelay(500);
        Servidor1->startAdvertising();
        Serial.println("Enviendo datos");
        Anterior_dispositivo_conectado = Dispositivo_conectado;
    }
}
```



```
if(Dispositivo_conectado && !Anterior_dispositivo_conectado)
{
    Anterior_dispositivo_conectado = Dispositivo_conectado;
}
}
```

Carga el programa a tu tarjeta. En nRF Connect aparecerá el ESP32 con el nuevo nombre que le asignamos en el código, conéctalo para abrir la pestaña que muestra los servicios y características:

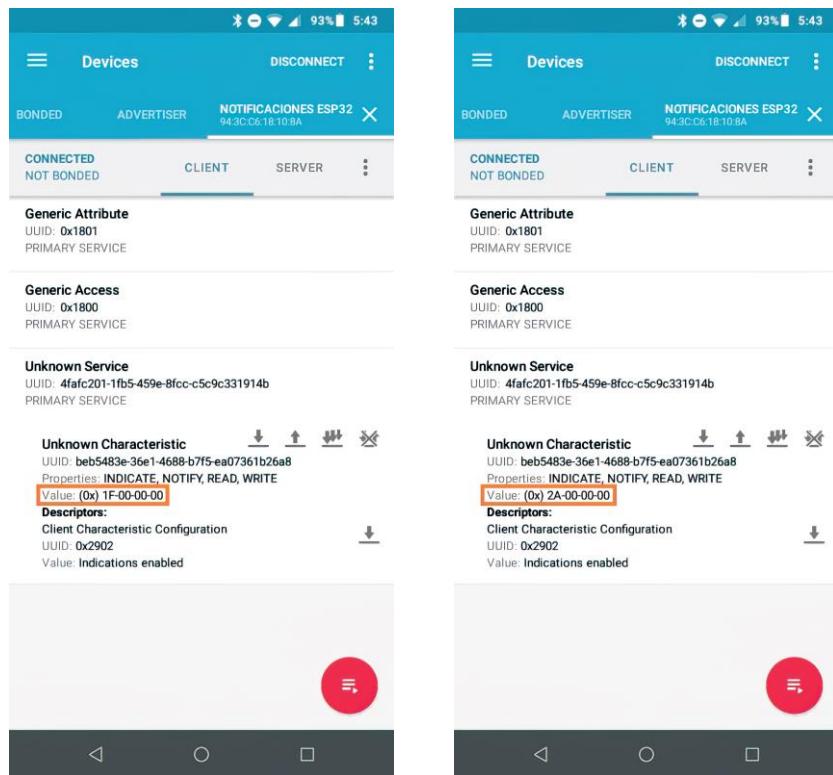


Al abrir la característica, verás que ahora tiene un descriptor asignado. Si presionas el botón marcado en rojo, el modo notificación se activará:





Con el modo notificación funcionando, verás que los valores de la característica cambian cada segundo (que son los valores de la variable *Dato*). Por ahora se muestran en hexadecimal, pero esto sirve para comprobar que el servidor BLE está enviando notificaciones:



## Aplicación práctica: servidor Bluetooth BLE con sensor de temperatura y humedad (DHT11)

Hemos visto como Bluetooth BLE nos permite montar un servidor para enviar datos a otros dispositivos Bluetooth, a través de una conexión de punto a punto. Las aplicaciones de BLE tienen que ver con la creación de dispositivos Bluetooth de bajo consumo energético, que cuenten con sensores que nos proporcionen información útil, como las pulseras de presión arterial.

En esta ocasión realizaremos un ejercicio de aplicación práctica, para montar un servidor BLE con el sensor de temperatura y humedad DHT11, similar a como lo manejamos en el ejercicio 22. Necesitarás las librerías para el sensor DHT11.



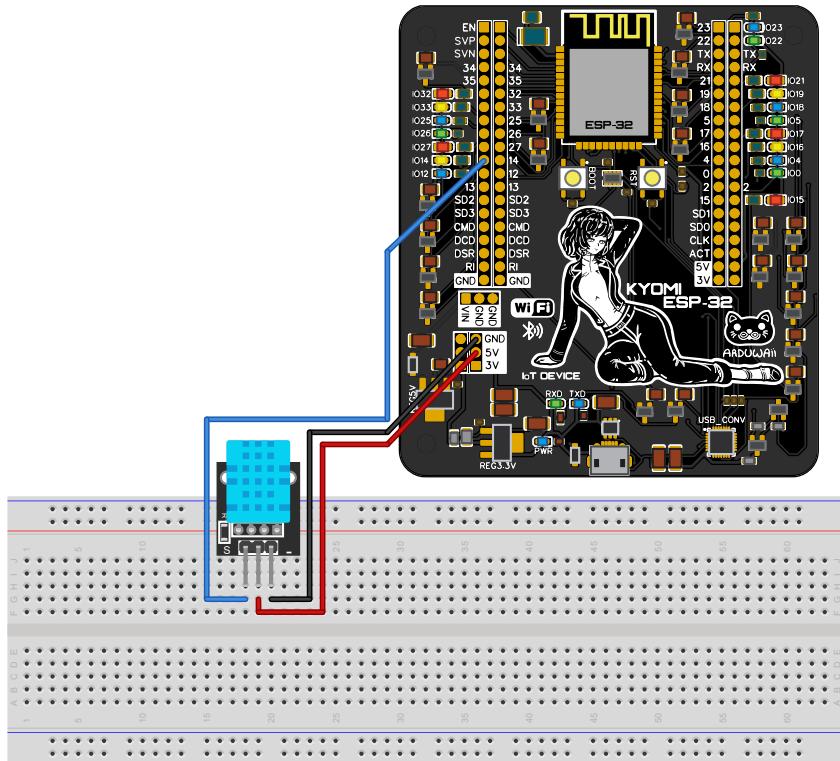
## Ejercicio 23

En este ejercicio montaremos un servidor Bluetooth BLE con un sensor DHT11, para que envíe sus lecturas de temperatura Celsius, Fahrenheit y porcentaje de humedad ambiental a un dispositivo cliente. Usaremos el modo notificación para enviar las lecturas al cliente cada 4 segundos.

Materiales:

- 1 módulo sensor DHT11 (como el KY-015), también puedes usar el DHT22
- 3 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:



Ahora veamos el código. Incluiremos las librerías necesarias para usar el BLE en modo notificación y para el sensor DHT11:

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include "DHT.h"
```



Usaremos una definición para guardar el UUID del servicio y otras tres para los UUID's de las características que guardarán la temperatura en Celsius, Fahrenheit y la Humedad:

```
#define UUID_SERVICIO "91bad492-b950-4226-aa2b-4ede9fa42f59"  
#define UUID_CARACTERISTICA_CELSIUS "234cd7a8-3225-4d09-b290-f84acb5791"  
#define UUID_CARACTERISTICA_FAHRENHEIT "61901b28-abb5-4ff0-a01d-1f706ef6ccfe"  
#define UUID_CARACTERISTICA_HUMEDAD "3cdcf7e4-72e6-4121-9bb1-0841d69a57dc"
```

Con `BLEServer` y `BLECharacteristic` creamos a `Servidor_DHT11` y las características `Caracteristica_CELSIUS`, `Caracteristica_FAHRENHEIT` y `Caracteristica_HUMEDAD`, pero sin configurar ninguna (`NULL`):

```
BLEServer* Servidor_DHT11 = NULL;  
BLECharacteristic* Caracteristica_CELSIUS = NULL;  
BLECharacteristic* Caracteristica_FAHRENHEIT = NULL;  
BLECharacteristic* Caracteristica_HUMEDAD = NULL;
```

Las siguientes variables `float` las usaremos para guardar las lecturas del DHT11 y en las variables `char` de tipo array (para 6 caracteres) también guardaremos estas lecturas (más adelante veremos por qué):

```
float T_Celsius=0, T_Fahrenheit=0, Humedad=0;  
char Temp_en_Celsius[6], Temp_en_Fahrenheit[6], P_Humedad[6];
```

Después vamos a declarar el GPIO que usaremos para comunicar al DHT11 (GPIO14). Configuraremos el sensor mediante `DHT dht(GPIO_DHT, DHT11)`, con su pin de comunicación y el tipo de sensor (`DHT11` o `DHT22`). La variable `Dispositivo_conectado` es la misma que usamos para el ejemplo de notificación:

```
const int GPIO_DHT = 14;  
DHT dht(GPIO_DHT, DHT11);  
bool Dispositivo_conectado = false;
```

La clase `Llamadas_al_servidor` detecta a los clientes que se conecten al servidor BLE. Funciona de la misma manera a como la aplicamos en el ejemplo de notificación. Recuerda que debes colocarla antes de la rutina `setup()`:

```
class Llamadas_al_servidor: public BLEServerCallbacks  
{  
    void onConnect(BLEServer* Servidor_DHT11)  
    {  
        Dispositivo_conectado = true;  
    };
```



```
void onDisconnect(BLEServer* Servidor_DHT11)
{
    Dispositivo_conectado = false;
}

};
```

Dentro de `setup()` configuraremos el servidor BLE. Habilitaremos la comunicación serial y el sensor DHT. En `BLEDevice::init()` colocamos el nombre de nuestro servidor BLE e iniciamos a `Servidor_DHT11` con `BLEDevice::createServer()`. También asociaremos a `Servidor_DHT11` con la clase `Llamadas_al_servidor`, para detectar a los clientes que se conecten:

```
dht.begin();
Serial.begin(115200);
BLEDevice::init("Servidor ESP32 con DHT11");
BLEServer *Servidor_DHT11 = BLEDevice::createServer();
Servidor_DHT11->setCallbacks(new Llamadas_al_servidor());
```

Luego vamos a crear un servicio llamado `Servicio1_DHT11`, que contendrá las características:

```
BLEService *Servicio1_DHT11 = Servidor_DHT11->createService(UUID_SERVICIO);
```

Ahora configuraremos las características que creamos anteriormente, cada una con su UUID:

```
Caracteristica_CELSIUS = Servicio1_DHT11->createCharacteristic
(UUID_CARACTERISTICA_CELSIUS,
BLECharacteristic::PROPERTY_READ  |
BLECharacteristic::PROPERTY_WRITE | 
BLECharacteristic::PROPERTY_NOTIFY | 
BLECharacteristic::PROPERTY_INDICATE);
Caracteristica_CELSIUS->addDescriptor(new BLE2902());
```

```
Caracteristica_FAHRENHEIT = Servicio1_DHT11->createCharacteristic
(UUID_CARACTERISTICA_FAHRENHEIT,
BLECharacteristic::PROPERTY_READ  |
BLECharacteristic::PROPERTY_WRITE | 
BLECharacteristic::PROPERTY_NOTIFY | 
BLECharacteristic::PROPERTY_INDICATE);
Caracteristica_FAHRENHEIT->addDescriptor(new BLE2902());
```



```
Caracteristica_HUMEDAD = Servicio1_DHT11->createCharacteristic  
(UUID_CHARACTERISTICA_HUMEDAD,  
BLECharacteristic::PROPERTY_READ |  
BLECharacteristic::PROPERTY_WRITE |  
BLECharacteristic::PROPERTY_NOTIFY |  
BLECharacteristic::PROPERTY_INDICATE);  
Caracteristica_HUMEDAD->addDescriptor(new BLE2902());
```

Iniciaremos a *Servicio1\_DHT11* y crearemos la función *enviarMensaje*, con la rutina que ya conoces:

```
Servicio1_DHT11->start();  
BLEAdvertising *enviarMensaje = BLEDevice::getAdvertising();  
enviarMensaje->addServiceUUID(UUID_SERVICIO);  
enviarMensaje->setScanResponse(false);  
enviarMensaje->setMinPreferred(0x0);  
BLEDevice::startAdvertising();  
Serial.println("Esperando clientes...");
```

En *Loop()* vamos a colocar una rutina similar a la que usamos en el ejemplo de notificación, pero sólo colocaremos el *if* principal para enviar datos, los *if* que comprobaban el estado de conexión de los clientes no los necesitaremos:

```
if(Dispositivo_conectado)  
{  
    T_Celsius = dht.readTemperature();  
    T_Fahrenheit = dht.readTemperature(true);  
    Humedad = dht.readHumidity();  
    dtostrf(T_Celsius, 3, 2, Temp_en_Celsius);  
    dtostrf(T_Fahrenheit, 3, 2, Temp_en_Fahrenheit);  
    dtostrf(Humedad, 3, 2, P_Humedad);  
  
    Caracteristica_CELSIUS->setValue(Temp_en_Celsius);  
    Caracteristica_CELSIUS->notify();  
  
    Caracteristica_FAHRENHEIT->setValue(Temp_en_Fahrenheit);  
    Caracteristica_FAHRENHEIT->notify();  
  
    Caracteristica_HUMEDAD->setValue(P_Humedad);  
    Caracteristica_HUMEDAD->notify();
```



```
Serial.print(T_Celsius);
Serial.println("°C");
Serial.print(T_Fahrenheit);
Serial.println("°F");
Serial.print(Humedad);
Serial.println("%");

if (isnan(T_Celsius) || isnan(T_Fahrenheit) || isnan(Humedad))
{
    Serial.println("Lectura retrasada o fallida");
    Caracteristica_CELSIUS->setValue("--");
    Caracteristica_CELSIUS->notify();

    Caracteristica_FAHRENHEIT->setValue("--");
    Caracteristica_FAHRENHEIT->notify();

    Caracteristica_HUMEDAD->setValue("--");
    Caracteristica_HUMEDAD->notify();
}

delay(4000);
}
```

El `if(Dispositivo_conectado)` se ejecutará cuando `Dispositivo_conectado` sea igual a `true`, es decir, cuando un cliente se conecte. Las lecturas de temperatura y humedad las obtendremos como en el ejercicio 22, cada una guardada en una variable `float`. Para que los datos enviados a la aplicación nRF Connect no aparezcan en hexadecimal, hay que convertirlos a una cadena de caracteres tipo `char`. Arduino tiene una instrucción dedicada para convertir números `float` a cadena de caracteres `char`:

`dtostrf(Variable a convertir, Cifras Izq, Cifras Der, Variable convertida);`  
en `Variable a convertir` colocaríamos la variable `float`, en `Cifras Izq` el número de cifras enteras que queramos incluir y en `Cifras Der` el número de cifras decimales. En `Variable convertida` se guardará el valor convertido. En este caso, las variables `char` que enviaremos al cliente son arrays de 6 caracteres: tres cifras a la izquierda, dos a la derecha y el punto decimal.  
Las lecturas de temperatura y humedad también las mostraremos en el Monitor Serie.



Similar al ejercicio 22, el `if (isnan(T_Celsius) || isnan(T_Fahrenheit) || isnan(Humedad))` se ejecuta cuando alguna de las lecturas del DHT11 es indeterminada (`nan`). Si esto ocurre, se envían un par de guiones cortos al cliente como respuesta.

Las respuestas del servidor BLE se envían cada 4 segundos (`delay(4000)`).

El código completo queda de la siguiente forma:

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include "DHT.h"

#define UUID_SERVICIO "91bad492-b950-4226-aa2b-4ede9fa42f59"
#define UUID_CARACTERISTICA_CELSIUS "234cd7a8-3225-4d09-b290-f84acb5791"
#define UUID_CARACTERISTICA_FAHRENHEIT "61901b28-abb5-4ff0-a01d-1f706ef6ccfe"
#define UUID_CARACTERISTICA_HUMEDAD "3cdcf7e4-72e6-4121-9bb1-0841d69a57dc"

BLEServer* Servidor_DHT11 = NULL;
BLECharacteristic* Caracteristica_CELSIUS = NULL;
BLECharacteristic* Caracteristica_FAHRENHEIT = NULL;
BLECharacteristic* Caracteristica_HUMEDAD = NULL;

float T_Celsius=0, T_Fahrenheit=0, Humedad=0;
char Temp_en_Celsius[6], Temp_en_Fahrenheit[6], P_Humedad[6];

const int GPIO_DHT = 14;
DHT dht(GPIO_DHT, DHT11);
bool Dispositivo_conectado = false;

class Llamadas_al_servidor: public BLEServerCallbacks
{
    void onConnect(BLEServer* Servidor_DHT11)
    {
        Dispositivo_conectado = true;
    };
}
```



```
void onDisconnect(BLEServer* Servidor_DHT11)
{
    Dispositivo_conectado = false;
}

};

void setup()
{
    dht.begin();
    Serial.begin(115200);
    BLEDevice::init("Servidor ESP32 con DHT11");
    BLEServer *Servidor_DHT11 = BLEDevice::createServer();
    Servidor_DHT11->setCallbacks(new Llamadas_al_servidor());

    BLEService *Servicio1_DHT11 = Servidor_DHT11->createService(UUID_SERVICIO);

    Caracteristica_CELSIUS = Servicio1_DHT11->createCharacteristic
    (UUID_CARACTERISTICA_CELSIUS,
     BLECharacteristic::PROPERTY_READ   |
     BLECharacteristic::PROPERTY_WRITE  |
     BLECharacteristic::PROPERTY_NOTIFY |
     BLECharacteristic::PROPERTY_INDICATE);
    Caracteristica_CELSIUS->addDescriptor(new BLE2902());

    Caracteristica_FAHRENHEIT = Servicio1_DHT11->createCharacteristic
    (UUID_CARACTERISTICA_FAHRENHEIT,
     BLECharacteristic::PROPERTY_READ   |
     BLECharacteristic::PROPERTY_WRITE  |
     BLECharacteristic::PROPERTY_NOTIFY |
     BLECharacteristic::PROPERTY_INDICATE);
    Caracteristica_FAHRENHEIT->addDescriptor(new BLE2902());

    Caracteristica_HUMEDAD = Servicio1_DHT11->createCharacteristic
    (UUID_CARACTERISTICA_HUMEDAD,
     BLECharacteristic::PROPERTY_READ   |
     BLECharacteristic::PROPERTY_WRITE  |
```



```
BLECharacteristic::PROPERTY_NOTIFY |  
BLECharacteristic::PROPERTY_INDICATE);  
Caracteristica_HUMEDAD->addDescriptor(new BLE2902());  
  
Servicio1_DHT11->start();  
BLEAdvertising *enviarMensaje = BLEDevice::getAdvertising();  
enviarMensaje->addServiceUUID(UUID_SERVICIO);  
enviarMensaje->setScanResponse(false);  
enviarMensaje->setMinPreferred(0x0);  
BLEDevice::startAdvertising();  
Serial.println("Esperando clientes...");  
}  
  
void Loop()  
{  
    if(Dispositivo_conectado)  
    {  
        T_Celsius = dht.readTemperature();  
        T_Fahrenheit = dht.readTemperature(true);  
        Humedad = dht.readHumidity();  
  
        dtostrf(T_Celsius, 3, 2, Temp_en_Celsius);  
        dtostrf(T_Fahrenheit, 3, 2, Temp_en_Fahrenheit);  
        dtostrf(Humedad, 3, 2, P_Humedad);  
  
Caracteristica_CELSIUS->setValue(Temp_en_Celsius);  
Caracteristica_CELSIUS->notify();  
  
Caracteristica_FAHRENHEIT->setValue(Temp_en_Fahrenheit);  
Caracteristica_FAHRENHEIT->notify();  
  
Caracteristica_HUMEDAD->setValue(P_Humedad);  
Caracteristica_HUMEDAD->notify();  
  
Serial.print(T_Celsius);  
Serial.println("°C");
```



```
Serial.print(T_Fahrenheit);
Serial.println("°F");
Serial.print(Humedad);
Serial.println("%");

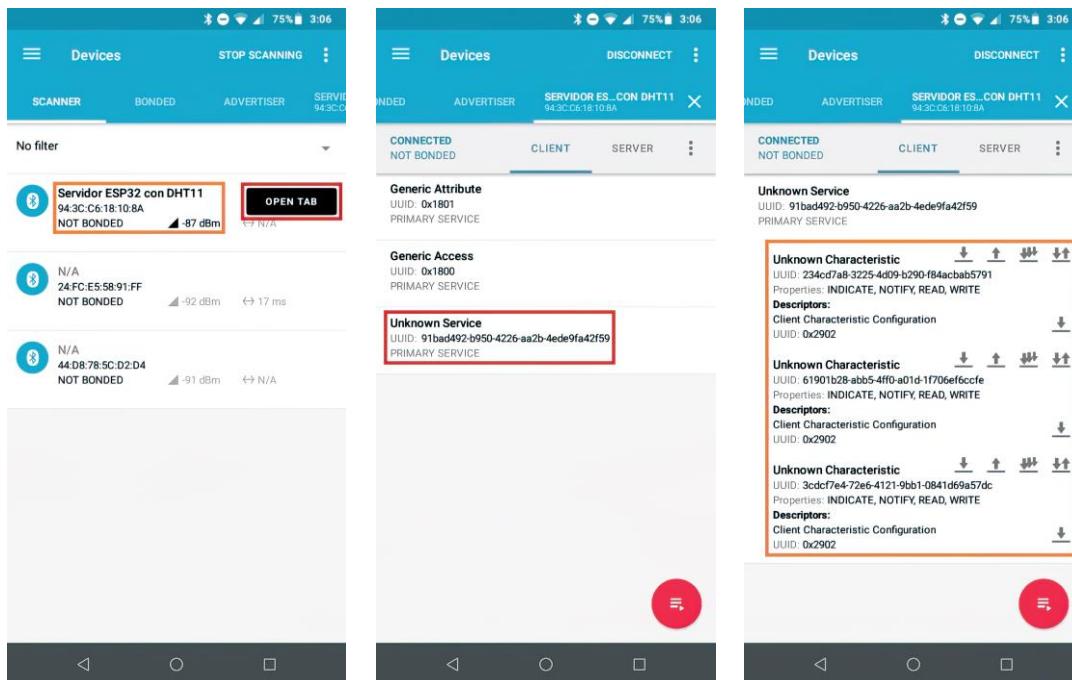
if (isnan(T_Celsius) || isnan(T_Fahrenheit) || isnan(Humedad))
{
    Serial.println("Lectura retrasada o fallida");
    Caracteristica_CELSIUS->setValue("--");
    Caracteristica_CELSIUS->notify();

    Caracteristica_FAHRENHEIT->setValue("--");
    Caracteristica_FAHRENHEIT->notify();

    Caracteristica_HUMEDAD->setValue("--");
    Caracteristica_HUMEDAD->notify();
}

delay(4000);
}
```

Prueba el programa. En nRF Connect conéctate al ESP32 para que puedas acceder al servicio y características que definimos:





Al abrir las características, se mostrarán las lecturas del DHT11 (activa el modo notificación para que los valores cambien). Puedes comparar las lecturas recibidas con las que se muestran en el Monitor Serie:

The image shows two side-by-side screenshots of the Bluefruit LE mobile application. Both screens display a 'Devices' list with a single entry: 'SERVIDOR ES... CON DHT11' (94:3C:C6:18:10:8A). The left screen shows the device as 'CONNECTED NOT BONDED' and has tabs for 'CLIENT' and 'SERVER'. Below the tabs, there are three 'Unknown Characteristic' entries. The first entry has a value of '29.90' highlighted in red. The second entry has a value of '85.82' highlighted in red. The third entry has a value of '62.00' highlighted in red. The right screen shows the device as 'CONNECTED NOT BONDED' and also has tabs for 'CLIENT' and 'SERVER'. It displays the same three 'Unknown Characteristic' entries, each with a value highlighted in red: '30.30', '86.54', and '64.00'. At the bottom of both screens is a red circular button with a white icon.

Below the app screenshots is a window titled 'COM3' showing serial port output. The text in the window is as follows:

```
86.54 °F
62.00%
30.30 °C
86.54 °F
62.00%
30.30 °C
86.54 °F
64.00%
30.30 °C
86.54 °F
64.00%
30.30 °C
86.54 °F
63.00%
```

At the bottom of the window are several buttons: 'Enviar' (Send), 'Autoscroll' (checked), 'Mostrar marca temporal' (unchecked), 'Nueva línea' (New Line), '115200 baudio' (Baud rate), and 'Limpiar salida' (Clear Output).

## BLE en modo Cliente

El ESP32 también puede funcionar como cliente Bluetooth, para conectarse a un servidor y empezar a recibir datos. En el modo cliente, básicamente hay que buscar las características del servidor con los



UUID's, extraer sus datos y guardarlos en algunas variables. Veremos un ejercicio práctico para crear un cliente BLE, pero necesitaremos un segundo ESP32 para montar el servidor del ejercicio 23.

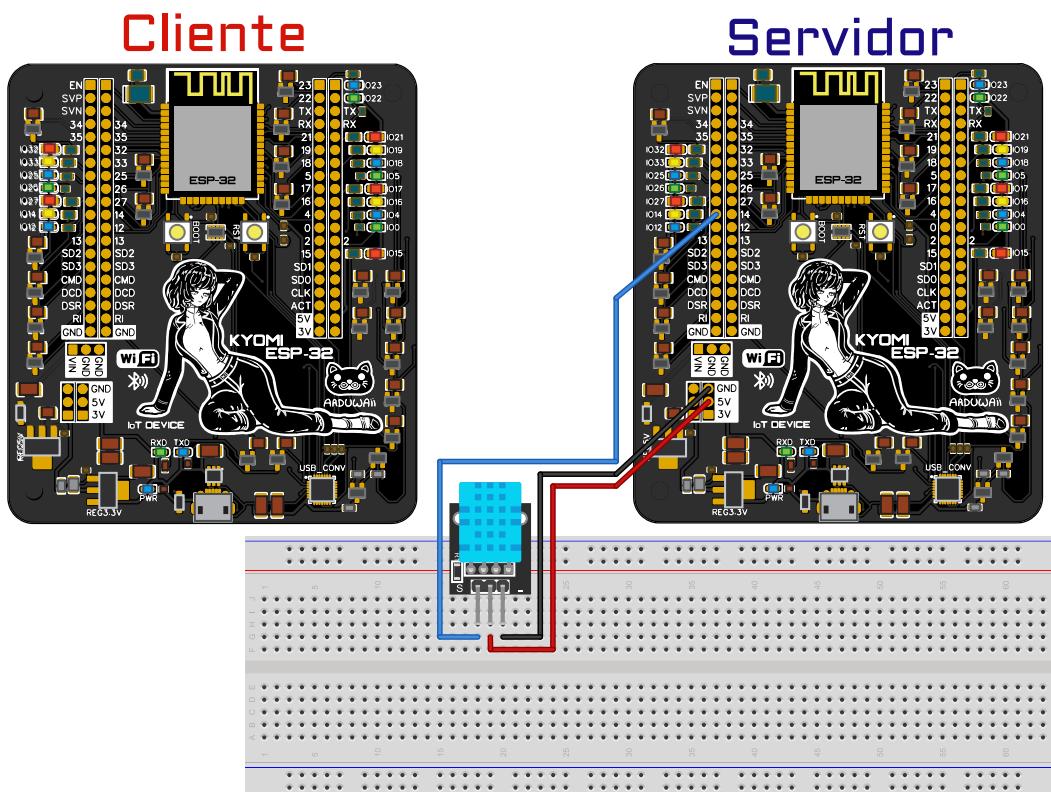
## Ejercicio 24

En este ejercicio vamos a configurar el ESP32 en modo cliente BLE, para realizar peticiones de datos a un servidor y mostrarlos en el Monitor Serie. Necesitarás un segundo ESP32 para montar el servidor del ejercicio 23 y así poder usar el otro ESP32 como cliente.

Materiales:

- 2 ESP32 (cualquiera compatible)
- 1 módulo sensor DHT11 (como el KY-015), también puedes usar el DHT22
- 3 jumpers o cables tipo Dupont
- plantilla de experimentos

El circuito es el siguiente:



Para el ESP32 servidor, simplemente carga el código del ejercicio 23. Ahora nos centraremos en el código para el ESP32 cliente.



Para el ESP32 cliente, sólo necesitaremos incluir una librería. Crearemos unas cuantas definiciones para el nombre del servidor y las UUID's de sus servicios y características. Tanto el nombre del servidor como las UUID's tienen que ser exactamente las mismas que le asignamos al servidor BLE:

```
#include "BLEDevice.h"

#define NOMBRE_SERVIDOR_DHT "Servidor ESP32 con DHT11"
#define UUID_SERVICIO "91bad492-b950-4226-aa2b-4ede9fa42f59"
#define UUID_CARACTERISTICA_CELSIUS "234cd7a8-3225-4d09-b290-f84acbab5791"
#define UUID_CARACTERISTICA_FAHRENHEIT "61901b28-abb5-4ff0-a01d-1f706ef6ccfe"
#define UUID_CARACTERISTICA_HUMEDAD "3cdcf7e4-72e6-4121-9bb1-0841d69a57dc"
```

Con `BLEAddress` crearemos el objeto `Direccion_SERVIDOR`, donde luego guardaremos la ‘dirección’ del servidor BLE. Con `BLERemoteCharacteristic` crearemos unas características para el cliente, similar a como lo hacíamos con el servidor, pero esta vez serán de tipo ‘remoto’ y las usaremos para guardar los valores de las características que le pediremos al servidor (por ahora no les daremos ninguna configuración):

```
BLEAddress *Direccion_SERVIDOR;
BLERemoteCharacteristic* CARACTERISTICA_CELSIUS_REMOTA;
BLERemoteCharacteristic* CARACTERISTICA_FAHRENHEIT_REMOTA;
BLERemoteCharacteristic* CARACTERISTICA_HUMEDAD_REMOTA;
```

Después vamos a declarar las variables necesarias. Las de tipo `char` las usaremos para guardar las lecturas recibidas del servidor, pero tienen que ser de tipo apuntador (por eso se coloca un \*), ya que algunas de las instrucciones que convierten los datos no funcionan bien con `char` simple:

```
const uint8_t notificacion_ON[] = {0x1, 0x0};

boolean Servidor_conectado = false;
boolean EstadoConexion = false;
char* temperatura_Celsius;
char* temperatura_Fahrenheit;
char* humedad;

boolean lectura_nueva_Celsius = false;
boolean lectura_nueva_Fahrenheit = false;
boolean lectura_nueva_humedad = false;
```

Las variables booleanas son variables auxiliares que usaremos después. La variable `notificacion_ON[]` es un array de tipo `uint8_t` (`int` de 8 bits que



se puede expresar en hexadecimal) que es usada por una de las instrucciones del cliente BLE, y por lo general se debe incluir en los programas para clientes BLE.

La clase `BUSQUEDA_SERVIDOR()` es similar a la que usamos en el servidor BLE, solo que ahora se encarga de buscar servidores en vez de clientes. Esta clase se manda a llamar cuando el ESP32 escanea el entorno en busca de servidores. Cuando encuentra un servidor, la subrutina `onResult()` guardará un valor en la variable `Dispositivo_encontrado`, la cual contiene el nombre del servidor BLE que fue detectado mediante `BLEAdvertisedDevice`. El `if` se va a ejecutar sólo si el nombre guardado en `Dispositivo_encontrado` es igual al nombre que establecimos en `NOMBRE_SERVIDOR_DHT`. Cuando esto ocurra, se detendrá el escaneo de dispositivos con `getScan()->stop()`, luego se va a extraer la ‘dirección’ del servidor BLE y se va a guardar en `Direccion_SERVIDOR`. La variable auxiliar `Servidor_conectado` va a cambiar a `true` y finalmente se va a enviar un mensaje al Monitor Serie. Esta clase y su rutina deben ser colocadas antes de `setup()`:

```
class BUSQUEDA_SERVIDOR: public BLEAdvertisedDeviceCallbacks
{
    void onResult(BLEAdvertisedDevice Dispositivo_encontrado)
    {
        if(Dispositivo_encontrado.getName() == NOMBRE_SERVIDOR_DHT)
        {
            Dispositivo_encontrado.getScan()->stop();
            Direccion_SERVIDOR = new BLEAddress(Dispositivo_encontrado.getAddress());
            Servidor_conectado = true;
            Serial.println("Se encontró el servidor, conectando...");
        }
    }
};
```

Dentro de `setup()` configuraremos el cliente BLE. Primero habilitaremos la comunicación serial. Con `BLEDevice::init` iniciamos el cliente BLE, pero no será necesario darle un nombre. Con `BLEScan` crearemos la función `escaneo_BLE` (similar a como lo hacíamos con `BLEAdvertising`), para iniciar la búsqueda de servidores con `getScan()` y llamar a la clase `BUSQUEDA_SERVIDOR()` cuando detecte algún dispositivo. La búsqueda de servidores durará 30 segundos (`escaneo_BLE->start(30)`):

```
Serial.begin(115200);
Serial.println("Iniciando cliente...");

BLEDevice::init("");
```



```
BLEScan* escaneo_BLE = BLEDevice::getScan();
escaneo_BLE->setAdvertisedDeviceCallbacks(new BUSQUEDA_SERVIDOR());
escaneo_BLE->setActiveScan(true);
escaneo_BLE->start(30);
```

Antes de pasar a `loop()`, vamos a crear la subrutina `conexion_Servidor()`. Esta subrutina tiene asignada a la variable `Direccion`, que contiene la dirección del servidor BLE, entregada por `BLEAddress`. Con `BLEClient` crearemos a `Cliente` (similar a cuando creábamos un servidor), luego la instrucción `Cliente->connect(Direccion)` hará la conexión con el servidor, usando el valor guardado en `Direccion`. Crearemos un servicio de tipo remoto llamado `SERVICIOREMOTO` (usando `BLERemoteService`), para guardar los datos del servicio del servidor. Para obtener estos datos, se usa la instrucción `getService()` junto con `UUIDSERVICIO` (debe ser el mismo UUID del servicio del servidor):

```
bool conexion_Servidor(BLEAddress Direccion)
{
    BLEClient* Cliente = BLEDevice::createClient();
    Cliente->connect(Direccion);
    Serial.println("Conectado al servidor.");

    BLERemoteService* SERVICIOREMOTO = Cliente->getService(UUIDSERVICIO);
    if(SERVICIOREMOTO == nullptr)
    {
        Serial.println("No se encontró el servicio...");
        return (false);
    }

    CARACTERISTICA_CELSIUSREMOTA = SERVICIOREMOTO-
>getCharacteristic(UUIDCARACTERISTICA_CELSIUS);
    CARACTERISTICA_FAHRENHEITREMOTA = SERVICIOREMOTO-
>getCharacteristic(UUIDCARACTERISTICA_FAHRENHEIT);
    CARACTERISTICA_HUMEDADREMOTA = SERVICIOREMOTO-
>getCharacteristic(UUIDCARACTERISTICA_HUMEDAD);

    if(CARACTERISTICA_CELSIUSREMOTA == nullptr ||
       CARACTERISTICA_FAHRENHEITREMOTA == nullptr ||
       CARACTERISTICA_HUMEDADREMOTA == nullptr)
    {
```



```
Serial.println("No se encontró la característica...");  
return false;  
}  
  
Serial.println("Se encontraron las características.");  
CARACTERISTICA_CELSIUS_REMOTA->registerForNotify(Notificacion_CELSIUS);  
CARACTERISTICA_FAHRENHEIT_REMOTA->registerForNotify(Notificacion_FAHRENHEIT);  
CARACTERISTICA_HUMEDAD_REMOTA->registerForNotify(Notificacion_HUMEDAD);  
return true;  
}
```

El `if(SERVICIOREMOTO == nullptr)` se ejecuta cuando no se encontró el servicio especificado en `SERVICIOREMOTO`, por lo general, esto ocurre cuando `UUIDSERVICIO` tiene algún error. Si se encontró el servicio, entonces crearemos unas características locales donde guardaremos los valores de las características del servidor.

Por ejemplo, `CARACTERISTICA_CELSIUS_REMOTA` guarda el valor de la característica del servidor que tiene el `UUIDCARACTERISTICA_CELSIUS` (el mismo que se usa en el servidor), mediante la instrucción `getCharacteristic`. El segundo `if` se ejecuta cuando no se encuentre a alguna de las características especificadas. Si se encontraron todas, entonces las características locales se configuran en modo notificación para estar recibiendo datos (con `registerForNotify`), que luego serán enviados a una subrutina. Por ejemplo, los datos de `CARACTERISTICA_CELSIUS_REMOTA` son enviados a la subrutina `Notificacion_CELSIUS`.

Ahora en `loop()`, básicamente colocaremos unos cuantos `if` que modificarán unas variables que establecen el estado de la conexión:

```
if(Servidor_conectado == true)  
{  
    if(conexion_Servidor(*Direccion_SERVIDOR))  
    {  
        Serial.println("Conexión correcta con el servidor");  
        CARACTERISTICA_CELSIUS_REMOTA->getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notificacion_ON, 2, true);  
        CARACTERISTICA_FAHRENHEIT_REMOTA->getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notificacion_ON, 2, true);  
        CARACTERISTICA_HUMEDAD_REMOTA->getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notificacion_ON, 2, true);  
        EstadoConexion = true;  
    }  
}
```



```
else
{
    Serial.println("Fallo en La conexión al servidor, reinicia el ESP32");
}
Servidor_conectado = false;
}

if(Lectura_nueva_Celsius && lectura_nueva_Fahrenheit && Lectura_nueva_humedad)
{
    Lectura_nueva_Celsius = false;
    Lectura_nueva_Fahrenheit = false;
    Lectura_nueva_humedad = false;
}
delay(1000);
```

El `if(Servidor_conectado == true)` se ejecuta cuando `Servidor_conectado` es igual a `true`, es decir, cuando nuestro cliente se ha conectado al servidor. Entonces `if(conexion_Servidor(*Direccion_SERVIDOR))` se ejecutará si la dirección que entrega `conexion_Servidor()` es igual a la que está guardada en `Direccion_SERVIDOR`, podría decirse que esto es para confirmar que la dirección del servidor sigue siendo la misma. Dentro de este `if`, se obtendrán los descriptores de cada característica del servidor mediante `getDescriptor` (aquí se usará la variable `notificacion_ON`), y luego `EstadoConexion` pasará a `true`. Si no se cumple este `if`, en el siguiente `else` se enviará un mensaje de error al Monitor Serie para que reinicies el ESP32, aunque este error ocurre cuando el servidor se desconecta porque se apagó o reinició. Si la conexión funcionó y se obtuvieron los descriptores, `Servidor_conectado` podrá volver a `false` (pero regresará a `true` en `BUSQUEDA_SERVIDOR`). El último `if` se ejecuta si alguna de sus variables de estado (`Lectura_nueva_Celsius`, por ejemplo) es igual a `true`, que ocurre cuando se recibe algún dato nuevo del servidor. Es entonces el estado de estas variables cambiará a `false`. Toda esta rutina se ejecuta cada 1000ms (`delay(1000)`).

Las siguientes subrutinas nos permitirán extraer los datos recibidos del servidor para mostrarlos en el Monitor Serie:

```
static void Notificacion_CELSIUS(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* DATOS, size_t length, bool isNotify)
{
    temperatura_Celsius = (char*)DATOS;
    Serial.print(temperatura_Celsius);
    Serial.println("°C");
```



```
    lectura_nueva_Celsius = true;
}

static void Notificacion_FAHRENHEIT(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* DATOS, size_t length, bool isNotify)
{
    temperatura_Fahrenheit = (char*)DATOS;
    Serial.print(temperatura_Fahrenheit);
    Serial.println("°F");
    lectura_nueva_Fahrenheit = true;
}

static void Notificacion_HUMEDAD(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* DATOS, size_t length, bool isNotify)
{
    humedad = (char*)DATOS;
    Serial.print(humedad);
    Serial.println("%");
    lectura_nueva_humedad = true;
}
```

Por ejemplo, cuando `conexion_Servidor()` devuelve los datos guardados en `CARACTERISTICA_CELSIUS_REMOTA`, en la subrutina `Notificacion_CELSIUS` se volverán a guardar temporalmente en la variable `DATOS` (de tipo `uint8_t`). En la variable `temperatura_Celsius` se convertirá el valor de `DATOS` a una cadena `char`, para luego enviarla al Monitor Serie y cambiar el estado de `lectura_nueva_Celsius` a `true`. El resto de las subrutinas funcionan de manera similar.

El código completo queda de la siguiente forma:

```
#include "BLEDevice.h"

#define NOMBRE_SERVIDOR_DHT "Servidor ESP32 con DHT11"
#define UUID_SERVICIO "91bad492-b950-4226-aa2b-4ede9fa42f59"
#define UUID_CARACTERISTICA_CELSIUS "234cd7a8-3225-4d09-b290-f84acb5791"
#define UUID_CARACTERISTICA_FAHRENHEIT "61901b28-abb5-4ff0-a01d-1f706ef6ccfe"
#define UUID_CARACTERISTICA_HUMEDAD "3cdcf7e4-72e6-4121-9bb1-0841d69a57dc"

BLEAddress *Direccion_SERVIDOR;
BLERemoteCharacteristic* CARACTERISTICA_CELSIUS_REMOTA;
```



```
BLERemoteCharacteristic* CARACTERISTICA_FAHRENHEIT_REMOTA;
BLERemoteCharacteristic* CARACTERISTICA_HUMEDAD_REMOTA;

const uint8_t notificacion_ON[] = {0x1, 0x0};

boolean Servidor_conectado = false;
boolean EstadoConexion = false;
char* temperatura_Celsius;
char* temperatura_Fahrenheit;
char* humedad;
boolean lectura_nueva_Celsius = false;
boolean lectura_nueva_Fahrenheit = false;
boolean lectura_nueva_humedad = false;

class BUSQUEDA_SERVIDOR: public BLEAdvertisedDeviceCallbacks
{
    void onResult(BLEAdvertisedDevice Dispositivo_encontrado)
    {
        if(Dispositivo_encontrado.getName() == NOMBRE_SERVIDOR_DHT)
        {
            Dispositivo_encontrado.getScan()->stop();
            Direccion_SERVIDOR = new BLEAddress(Dispositivo_encontrado.getAddress());
            Servidor_conectado = true;
            Serial.println("Se encontró el servidor, conectando...");
        }
    }
};

void setup()
{
    Serial.begin(115200);
    Serial.println("Iniciando cliente...");

    BLEDevice::init("");
    BLEScan* escaneo_BLE = BLEDevice::getScan();
    escaneo_BLE->setAdvertisedDeviceCallbacks(new BUSQUEDA_SERVIDOR());
```



```
escaneo_BLE->setActiveScan(true);
escaneo_BLE->start(30);
}

bool conexion_Servidor(BLEAddress Direccion)
{
    BLEClient* Cliente = BLEDevice::createClient();
    Cliente->connect(Direccion);
    Serial.println("Conectado al servidor.");

    BLERemoteService* SERVICIOREMOTO = Cliente->getService(UUID_SERVICIO);
    if(SERVICIOREMOTO == nullptr)
    {
        Serial.println("No se encontró el servicio...");
        return (false);
    }

    CARACTERISTICA_CELSIUSREMOTA = SERVICIOREMOTO-
>getCharacteristic(UUID_CARACTERISTICA_CELSIUS);
    CARACTERISTICA_FAHRENHEITREMOTA = SERVICIOREMOTO-
>getCharacteristic(UUID_CARACTERISTICA_FAHRENHEIT);
    CARACTERISTICA_HUMEDADREMOTA = SERVICIOREMOTO-
>getCharacteristic(UUID_CARACTERISTICA_HUMEDAD);

    if(CARACTERISTICA_CELSIUSREMOTA == nullptr ||
       CARACTERISTICA_FAHRENHEITREMOTA == nullptr ||
       CARACTERISTICA_HUMEDADREMOTA == nullptr)
    {
        Serial.println("No se encontró la característica...");
        return false;
    }
    Serial.println("Se encontraron las características.");
    CARACTERISTICA_CELSIUSREMOTA->registerForNotify(Notificacion_CELSIUS);
    CARACTERISTICA_FAHRENHEITREMOTA->registerForNotify(Notificacion_FAHRENHEIT);
    CARACTERISTICA_HUMEDADREMOTA->registerForNotify(Notificacion_HUMEDAD);
    return true;
}
```



```
void Loop()
{
    if(Servidor_conectado == true)
    {
        if(conexion_Servidor(*Direccion_SERVIDOR))
        {
            Serial.println("Conexión correcta con el servidor");

            CARACTERISTICA_CELSIUS_REMOTA->getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notificacion_ON, 2, true);

            CARACTERISTICA_FAHRENHEIT_REMOTA-
            >getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notificacion_ON, 2, true);

            CARACTERISTICA_HUMEDAD_REMOTA->getDescriptor(BLEUUID((uint16_t)0x2902))->writeValue((uint8_t*)notificacion_ON, 2, true);

            EstadoConexion = true;
        }
        else
        {
            Serial.println("Fallo en la conexión al servidor, reinicia el ESP32");
        }
        Servidor_conectado = false;
    }

    if(Lectura_nueva_Celsius && Lectura_nueva_Fahrenheit && Lectura_nueva_humedad)
    {
        Lectura_nueva_Celsius = false;
        Lectura_nueva_Fahrenheit = false;
        Lectura_nueva_humedad = false;
    }
    delay(1000);
}

static void Notificacion_CELSIUS(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* DATOS, size_t length, bool isNotify)
{
    temperatura_Celsius = (char*)DATOS;
    Serial.print(temperatura_Celsius);
```



```
Serial.println("°C");
Lectura_nueva_Celsius = true;
}

static void Notificacion_FAHRENHEIT(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* DATOS, size_t length, bool isNotify)
{
    temperatura_Fahrenheit = (char*)DATOS;
    Serial.print(temperatura_Fahrenheit);
    Serial.println("°F");
    Lectura_nueva_Fahrenheit = true;
}

static void Notificacion_HUMEDAD(BLERemoteCharacteristic*
pBLERemoteCharacteristic, uint8_t* DATOS, size_t length, bool isNotify)
{
    humedad = (char*)DATOS;
    Serial.print(humedad);
    Serial.println("%");
    Lectura_nueva_humedad = true;
}
```



Ahora prueba el programa. El ESP32 servidor lo puedes alimentar con una batería USB para no conectarlo a la computadora. Comprobaremos el funcionamiento del ESP32 cliente con el Monitor Serie:

The image shows two side-by-side screenshots of the Arduino Serial Monitor. Both windows have 'COM3' selected at the top left. The left window shows the initial connection steps: 'Iniciando cliente...', 'Se encontró el servidor, conectando...', 'Conectado al servidor.', 'Se encontraron las características.', and 'Conexión correcta con el servidor'. The right window shows a continuous stream of temperature data in both Fahrenheit and Celsius, with values ranging from 53.00°F/C to 92.12°F/C.

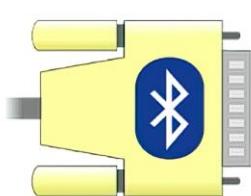
Si el cliente no se puede conectar al servidor la primera vez, reinicia ambos ESP32.

## Bluetooth clásico

El Bluetooth clásico es el que se usa en la mayoría de dispositivos Bluetooth con consumo energético común, como las bocinas inalámbricas, cámaras web, transmisores de video, entre otros periféricos. Su protocolo de comunicación es similar al GAP y GATT de BLE, pero la forma en que transmite los datos dependerá de la aplicación, por ejemplo, para el audio y video existen protocolos y reglas propias para transmitir sus datos. La ventaja de Bluetooth clásico sobre BLE, es que puede transmitir muchos más datos a la vez, ya que no está limitado por los paquetes de información pequeños que se usan en BLE.

El Bluetooth clásico que soporta el IDE de Arduino es muy simple, y se usa para enviar datos numéricos y texto a una terminal Bluetooth (parecido a usar el Monitor Serie). Las funciones más avanzadas, como transmitir audio, se reservan para otros IDE's compatibles con el ESP32, aunque también puedes desarrollar librerías propias para usar estas funciones dentro de Arduino.

Por ahora usaremos el Bluetooth clásico tipo 'serial', que viene integrado en el IDE de Arduino. Necesitaremos la aplicación 'Serial Bluetooth Terminal' para enviar y recibir datos desde nuestro celular o Tablet:



Serial Bluetooth Terminal

Kai Morich Tools

Everyone

Offers in-app purchases

This app is available for all of your devices



Esta aplicación solo está disponible para dispositivos Android. En iOS, la aplicación más parecida es 'HM10 Bluetooth Serial'.

Usar el Bluetooth serial es muy sencillo, así que haremos un ejercicio práctico para ver sus funciones.

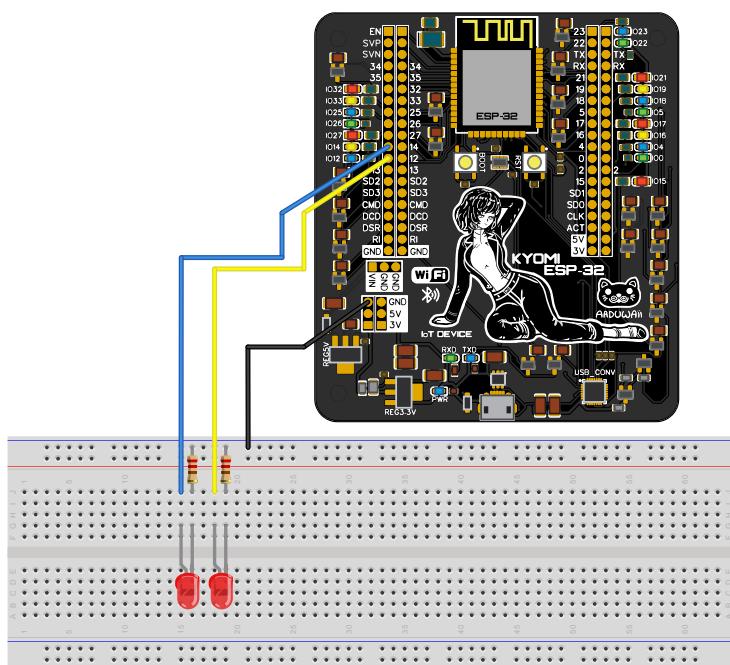
## Ejercicio 25

En este ejercicio usaremos el Bluetooth clásico en modo de terminal serial para controlar un par de GPIO's con LEDs, a través de comandos que enviaremos desde la aplicación Serial Bluetooth Terminal.

Materiales:

- 2 LEDs (opcional)
- 2 resistencias de 220Ω (opcional)
- 3 jumpers o cables tipo Dupont (para los LEDs)
- plantilla de experimentos

El circuito es el siguiente:



Veamos el código. Para usar el Bluetooth serial, solo necesitaremos incluir la librería `BluetoothSerial.h` e iniciar las funciones Bluetooth con `BluetoothSerial SerialBT`. También incluiremos unas variables auxiliares:

```
#include "BluetoothSerial.h"  
  
BluetoothSerial SerialBT;  
  
String ESTADO_LEDS="";  
const int GPIO12=12, GPIO14=14;
```



En `setup()` vamos a configurar la comunicación serial, los GPIO's para los LEDs y la comunicación Bluetooth. En `SerialBT.begin()` vamos a configurar el nombre de nuestro dispositivo Bluetooth, que es el que se va a mostrar cuando busquemos dispositivos con nuestro celular o Tablet:

```
Serial.begin(115200);
pinMode(GPIO12, OUTPUT);
pinMode(GPIO14, OUTPUT);
digitalWrite(GPIO12, LOW);
digitalWrite(GPIO14, LOW);
SerialBT.begin("ESP32 BT Serial");
```

Dentro de `loop()`, con la rutina que está en `if(SerialBT.available())`, guardaremos el mensaje que recibiremos del cliente Bluetooth. Desde la aplicación Serial Bluetooth Terminal, enviaremos un comando de texto que luego guardaremos en la variable `ESTADO_LEDS`, pero primero hay que ejecutar la rutina que irá recibiendo los caracteres de este mensaje. Cuando `if(SerialBT.available())` se cumple (siempre que la comunicación esté funcionando), se creará la variable temporal `Mensaje_Entrante`, que guardará un carácter del mensaje recibido por `SerialBT.read()`. Los caracteres que va recibiendo `Mensaje_Entrante` se van acumulando en la variable `ESTADO_LEDS` (en cada ciclo de `loop()`), siempre que se cumpla el `if(Mensaje_Entrante != '\n')`, es decir, que se irán acumulando estos caracteres hasta que se detecte un salto de línea (`\n`), que indica que el mensaje del cliente finalizó. Cuando se detecta el salto de línea, `ESTADO_LEDS` se vaciará en el `else`:

```
if(SerialBT.available())
{
    char Mensaje_Entrante=SerialBT.read();
    if(Mensaje_Entrante != '\n')
    {
        ESTADO_LEDS += String(Mensaje_Entrante);
    }
    else
    {
        ESTADO_LEDS = "";
    }
}
```

Con los siguientes `if` controlaremos los GPIO's, que cambiarán su estado de acuerdo al mensaje guardado en `ESTADO_LEDS`.



Estos *if* no se ejecutan hasta que se recibe el mensaje completo del cliente. Por ejemplo, *if(ESTADO\_LEDS=="GPIO12\_ON")* se ejecuta cuando el mensaje guardado en *ESTADO\_LEDS* es igual a *GPIO12\_ON*. Es entonces que se enviará un mensaje al Monitor Serie y a la terminal Bluetooth del cliente (con *SerialBT.println*), para finalmente cambiar el estado del GPIO12, en este caso a *HIGH*:

```
if(ESTADO_LEDS=="GPIO12_ON")
{
    SerialBT.println("GPIO12 Encendido");
    Serial.println("GPIO12 Encendido");
    digitalWrite(GPIO12,HIGH);
}

if(ESTADO_LEDS=="GPIO12_OFF")
{
    SerialBT.println("GPIO12 Apagado");
    Serial.println("GPIO12 Apagado");
    digitalWrite(GPIO12,LOW);
}

if(ESTADO_LEDS=="GPIO14_ON")
{
    SerialBT.println("GPIO14 Encendido");
    Serial.println("GPIO14 Encendido");
    digitalWrite(GPIO14,HIGH);
}

if(ESTADO_LEDS=="GPIO14_OFF")
{
    SerialBT.println("GPIO14 Apagado");
    Serial.println("GPIO14 Apagado");
    digitalWrite(GPIO14,LOW);
}

delay(10);
```

El código completo queda de la siguiente forma:

```
#include "BluetoothSerial.h"
BluetoothSerial SerialBT;
```



```
String ESTADO_LEDS="";
const int GPIO12=12, GPIO14=14;

void setup()
{
    Serial.begin(115200);
    pinMode(GPIO12, OUTPUT);
    pinMode(GPIO14, OUTPUT);
    digitalWrite(GPIO12, LOW);
    digitalWrite(GPIO14, LOW);
    SerialBT.begin("ESP32 BT Serial");
}

void Loop()
{
    if(SerialBT.available())
    {
        char Mensaje_Establecer=SerialBT.read();
        if(Mensaje_Establecer=='\n')
        {
            ESTADO_LEDS += String(Mensaje_Establecer);
        }
        else
        {
            ESTADO_LEDS = "";
        }
    }

    if(ESTADO_LEDS=="GPIO12_ON")
    {
        SerialBT.println("GPIO12 Encendido");
        Serial.println("GPIO12 Encendido");
        digitalWrite(GPIO12,HIGH);
    }

    if(ESTADO_LEDS=="GPIO12_OFF")
    {
        SerialBT.println("GPIO12 Apagado");
    }
}
```



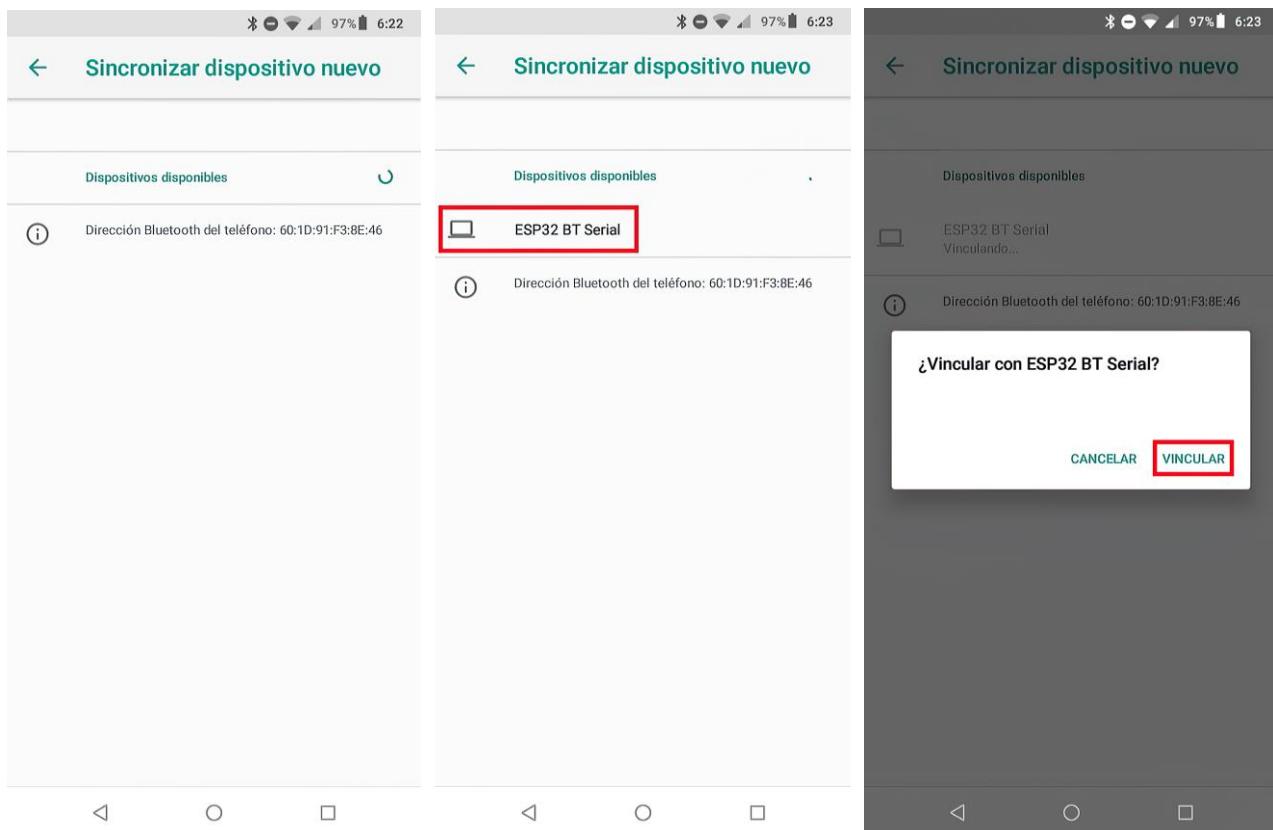
```
Serial.println("GPIO12 Apagado");
digitalWrite(GPIO12,LOW);
}

if(ESTADO_LEDS=="GPIO14_ON")
{
    SerialBT.println("GPIO14 Encendido");
    Serial.println("GPIO14 Encendido");
    digitalWrite(GPIO14,HIGH);
}

if(ESTADO_LEDS=="GPIO14_OFF")
{
    SerialBT.println("GPIO14 Apagado");
    Serial.println("GPIO14 Apagado");
    digitalWrite(GPIO14,LOW);
}

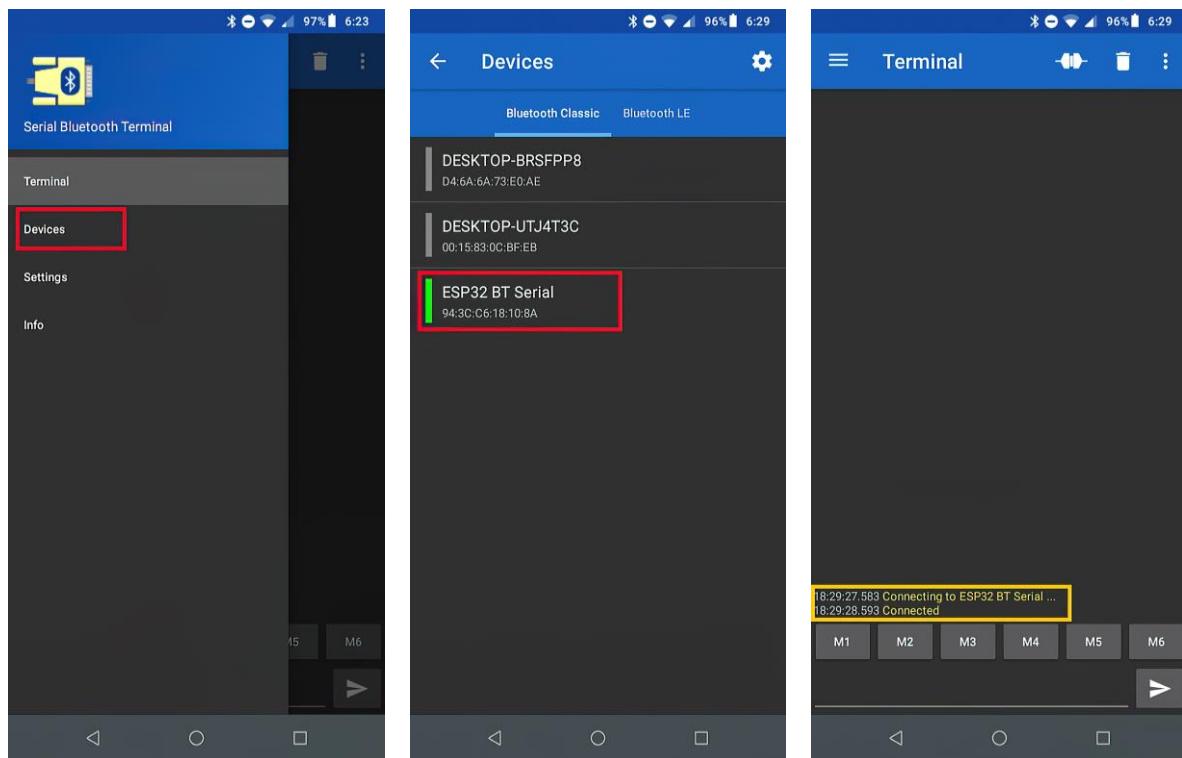
delay(10);
}
```

Ahora prueba el programa. Debes emparejar el ESP32 manualmente en tu dispositivo Android o IOS antes de abrir la aplicación:

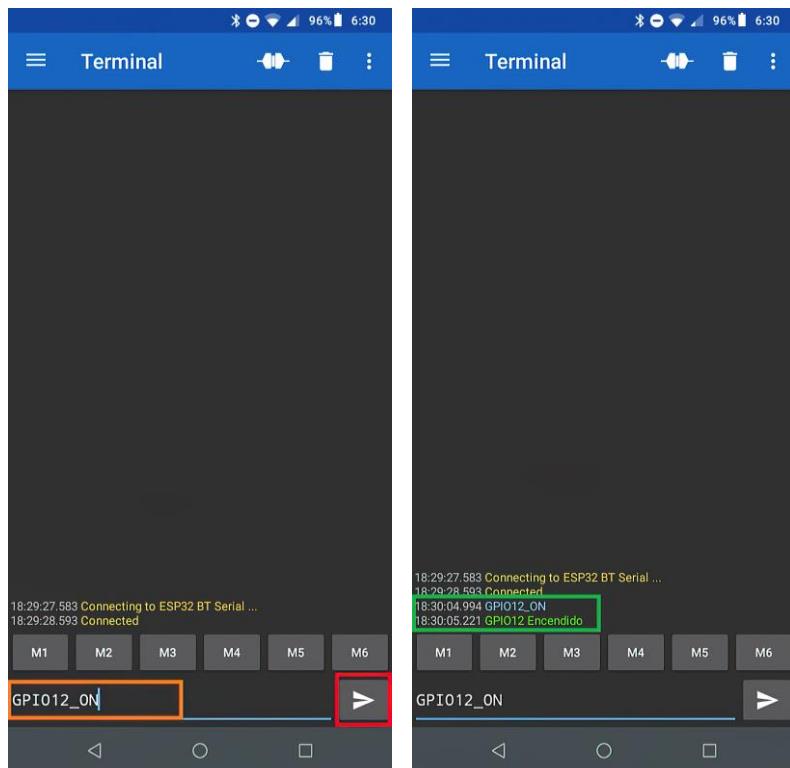




Una vez conectado, en Serial Bluetooth Terminal deberás seleccionar al ESP32 para poder comunicarlo con la terminal serial:



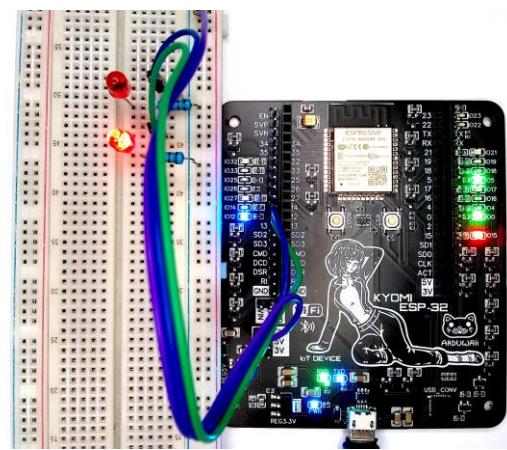
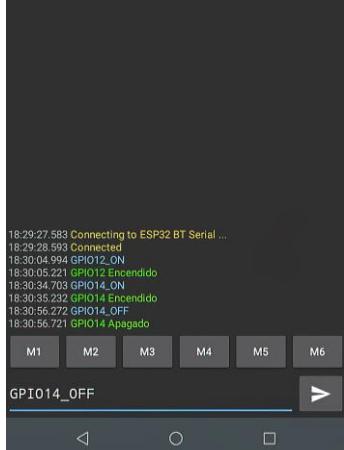
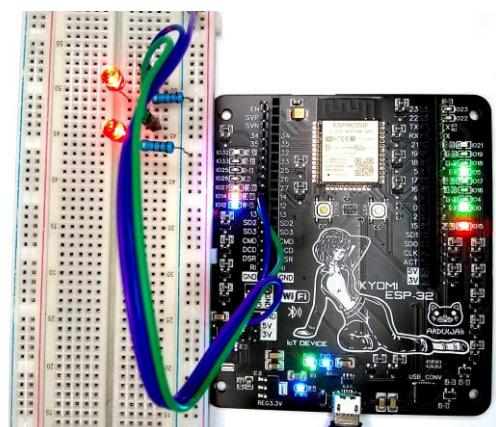
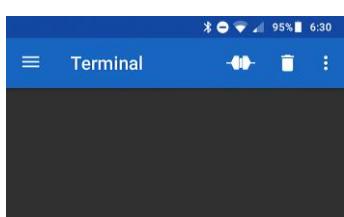
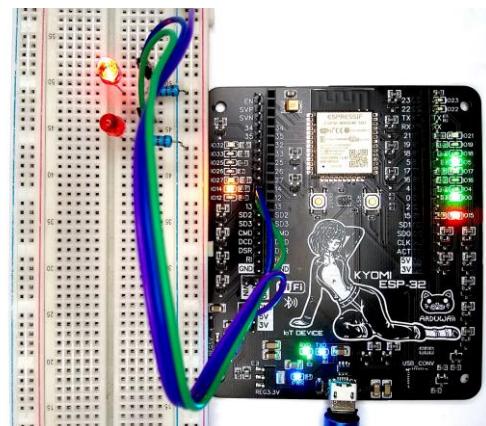
En la terminal serial podrás escribir los comandos para controlar a los LEDs, que son GPIO12\_ON, GPIO12\_OFF, GPIO14\_ON y GPIO14\_OFF:

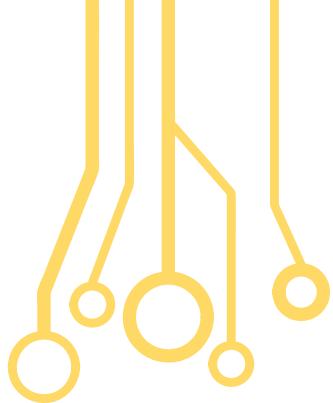




## Capítulo 5: Bluetooth clásico y BLE

Cuando envíes los comandos y cambies el estado de los LEDs, el mensaje que envía el ESP32 a la terminal también se va a mostrar:





# Capítulo 6: La última y nos vamos





## Cómo usar los dos núcleos del ESP32

Como se mencionó en el capítulo 1, el ESP32 cuenta con dos núcleos de procesamiento para ejecutar múltiples tareas, parecido a los núcleos del procesador de una computadora. Entre más núcleos tenga un procesador, va a poder ejecutar más tareas al mismo tiempo, aunque no necesariamente más rápido.

Los núcleos del ESP32 se organizan como núcleo 0 y núcleo 1. Generalmente, las rutinas `setup()` y `loop()` se ejecutan en el núcleo 1, dejando libre al núcleo 0.

Para usar los dos núcleos del ESP32, hay que asignarles alguna tarea. Esto se hace en tres pasos:

1. En la sección de declaración de variables, hay que declarar la tarea que luego asignaremos a un núcleo:

```
TaskHandle_t TAREA_1;  
en este caso, declaramos la tarea llamada TAREA_1.
```

2. Dentro de `setup()` vamos a configurar la tarea para asignarla a uno de los núcleos, usando la instrucción `xTaskCreatePinnedToCore()`:

```
xTaskCreatePinnedToCore(  
    RUTINA,  
    "ALIAS",  
    TAMAÑO DE LA PILA,  
    PARAMETROS,  
    PRIORIDAD,  
    &NOMBRE DE LA TAREA,  
    NUCLEO ASIGNADO);
```

en `RUTINA` colocamos el nombre de la rutina que contendrá las instrucciones a ejecutar, en `ALIAS` colocaremos un nombre de referencia para la tarea, el `TAMAÑO DE LA PILA` es la cantidad de memoria que le asignaremos a la tarea (por lo general, 10000 bytes serán suficientes), en `PARAMETROS` colocaremos algún parámetro que será transferido a nuestra tarea desde otra parte del programa (pero puedes dejarlo en `NULL` si no necesitas parámetros), en `PRIORIDAD` se coloca el número de prioridad que tiene la tarea (esto es importante cuando tienes muchas tareas asignadas a un mismo núcleo), en `NOMBRE DE LA TAREA` colocamos el nombre que definimos en `TaskHandle_t` y en `NUCLEO ASIGNADO` colocamos el número de núcleo al que le asignaremos esta tarea (núcleo 0 o 1).

Para este caso en particular, podemos configurar la tarea de la siguiente manera:

```
xTaskCreatePinnedToCore(  
    TAREA_NUCLEO_0,  
    "TAREA_n_1",  
    10000,  
    NULL,
```



```
1,  
&TAREA_1,  
0);
```

3. Finalmente hay que crear la rutina que contendrá las instrucciones de nuestra tarea:

```
void TAREA_1(void* pvParameters)  
{  
    //AQUÍ COLOCA LAS INSTRUCCIONES  
}
```

la variable `pvParameters` recibe parámetros internos del programa, así que siempre hay que colocarla. Para que esta rutina se ejecute constantemente (como `Loop()`) en lugar de una sola vez, tendremos que colocar un ciclo infinito. Podemos hacer esto de dos maneras, con un `for` sin configurar o con un `while` que tenga una condición `true`:

```
void TAREA_1(void* pvParameters)  
{  
    for(;;)  
    {  
        //AQUÍ COLOCA LAS INSTRUCCIONES  
    }  
}  
o bien  
void TAREA_1(void* pvParameters)  
{  
    while(true)  
    {  
        //AQUÍ COLOCA LAS INSTRUCCIONES  
    }  
}
```

Otra instrucción útil es `xPortGetCoreID()`, que entrega el número de núcleo en el que se está ejecutando el conjunto de tareas donde se encuentra esta instrucción, por ejemplo:

```
void TAREA_1(void* pvParameters)  
{  
    while(true)  
    {  
        Serial.println(xPortGetCoreID());  
    }  
}
```

donde el mensaje enviado al Monitor Serie sería ‘0’, que es el núcleo en el que se está ejecutando la rutina `TAREA_1`.

Ahora veamos un ejercicio sencillo.



## Ejercicio 26

En este ejercicio usaremos los dos núcleos del ESP32 para controlar dos secuencias de LEDs, cada una asignada a un núcleo.

Materiales:

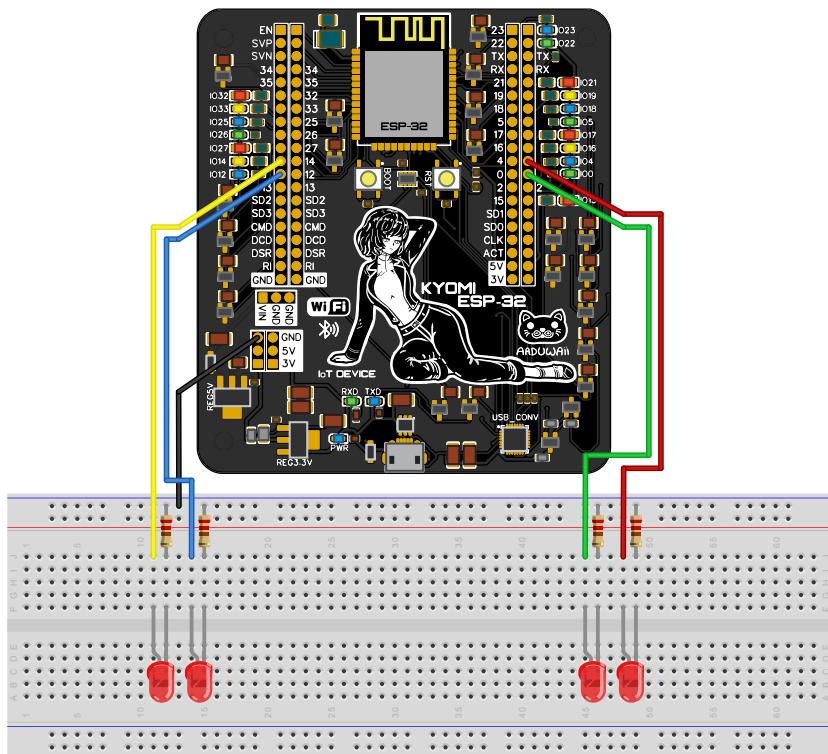
-4 LEDs (opcional)

-4 resistencias de 220Ω (opcional)

-5 jumpers o cables tipo Dupont (para los LEDs)

-plantilla de experimentos

El circuito es el siguiente:



Veamos el código. Primero vamos a crear un par de tareas que luego asignaremos a los núcleos, además de declarar las variables para los GPIO's que vamos a usar:

```
TaskHandle_t TAREA_nucleo_1;
TaskHandle_t TAREA_nucleo_2;

const int GPIO_1_TAREA_1=14,      GPIO_2_TAREA_1=12,      GPIO_1_TAREA_2=4,
GPIO_2_TAREA_2=0;
```



En `setup()` vamos a configurar la comunicación serial, los GPIO's y las tareas. La tarea `TAREA_nucleo_1` va a estar corriendo en el núcleo 0 y `TAREA_nucleo_2` va a estar corriendo en el núcleo 1:

```
Serial.begin(115200);  
  
pinMode(GPIO_1_TAREA_1, OUTPUT);  
pinMode(GPIO_2_TAREA_1, OUTPUT);  
  
pinMode(GPIO_1_TAREA_2, OUTPUT);  
pinMode(GPIO_2_TAREA_2, OUTPUT);  
  
digitalWrite(GPIO_1_TAREA_1, LOW);  
digitalWrite(GPIO_2_TAREA_1, LOW);  
  
digitalWrite(GPIO_1_TAREA_2, LOW);  
digitalWrite(GPIO_2_TAREA_2, LOW);  
  
  
xTaskCreatePinnedToCore(  
    TAREA_1,  
    "TAREA_n_1",  
    10000,  
    NULL,  
    1,  
    &TAREA_nucleo_1,  
    0);  
  
  
xTaskCreatePinnedToCore(  
    TAREA_2,  
    "TAREA_n_2",  
    10000,  
    NULL,  
    1,  
    &TAREA_nucleo_2,  
    1);
```

La rutina `TAREA_1` y `TAREA_2` pertenecen a `TAREA_nucleo_1` y `TAREA_nucleo_2` respectivamente.



Ambas rutinas se ejecutan como un `loop()`, pero `TAREA_2` usa un `for` infinito y `TAREA_1` un `while`:

```
void TAREA_1(void* pvParameters)
{
    while(true)
    {
        SECUENCIA_NUCLEO_0();
        Serial.print("Esto está corriendo en el núcleo: ");
        Serial.println(xPortGetCoreID());
    }
}

void TAREA_2(void* pvParameters)
{
    for(;;)
    {
        SECUENCIA_NUCLEO_1();
        Serial.print("Esto está corriendo en el núcleo: ");
        Serial.println(xPortGetCoreID());
    }
}
```

Las rutinas de las tareas también pueden llamar a otras subrutinas, en este caso, las subrutinas `SECUENCIA_NUCLEO_0` y `SECUENCIA_NUCLEO_1`, que contienen las secuencias de LEDs. La secuencia de LEDs de `SECUENCIA_NUCLEO_0` es más rápida que la de `SECUENCIA_NUCLEO_1`:

```
void SECUENCIA_NUCLEO_0()
{
    digitalWrite(GPIO_1_TAREA_1, HIGH);
    delay(500);
    digitalWrite(GPIO_2_TAREA_1, HIGH);
    delay(500);
    digitalWrite(GPIO_1_TAREA_1, LOW);
    delay(500);
    digitalWrite(GPIO_2_TAREA_1, LOW);
    delay(500);
}
```



```
void SECUENCIA_NUCLEO_1()
{
    digitalWrite(GPIO_1_TAREA_2, HIGH);
    delay(250);
    digitalWrite(GPIO_2_TAREA_2, HIGH);
    delay(250);
    digitalWrite(GPIO_1_TAREA_2, LOW);
    delay(250);
    digitalWrite(GPIO_2_TAREA_2, LOW);
    delay(250);
}
```

A `Loop()` lo dejaremos vacío. De hecho, podríamos eliminar a la rutina `TAREA_2` y colocar sus instrucciones en `Loop()` (e incluso eliminar su configuración), ya que `Loop()` se ejecuta en el núcleo 1:

```
void Loop()
{
}
```

El código completo queda de la siguiente forma:

```
TaskHandle_t TAREA_nucleo_1;
TaskHandle_t TAREA_nucleo_2;

const int GPIO_1_TAREA_1=14,      GPIO_2_TAREA_1=12,      GPIO_1_TAREA_2=4,
GPIO_2_TAREA_2=0;

void setup()
{
    Serial.begin(115200);
    pinMode(GPIO_1_TAREA_1, OUTPUT);
    pinMode(GPIO_2_TAREA_1, OUTPUT);
    pinMode(GPIO_1_TAREA_2, OUTPUT);
    pinMode(GPIO_2_TAREA_2, OUTPUT);
    digitalWrite(GPIO_1_TAREA_1, LOW);
    digitalWrite(GPIO_2_TAREA_1, LOW);
    digitalWrite(GPIO_1_TAREA_2, LOW);
    digitalWrite(GPIO_2_TAREA_2, LOW);
```



```
xTaskCreatePinnedToCore(  
    TAREA_1,  
    "TAREA_n_1",  
    10000,  
    NULL,  
    1,  
    &TAREA_nucleo_1,  
    0);  
  
xTaskCreatePinnedToCore(  
    TAREA_2,  
    "TAREA_n_2",  
    10000,  
    NULL,  
    1,  
    &TAREA_nucleo_2,  
    1);  
}  
  
void TAREA_1(void* pvParameters)  
{  
    while(true)  
    {  
        SECUENCIA_NUCLEO_0();  
  
        Serial.print("Esto está corriendo en el núcleo: ");  
        Serial.println(xPortGetCoreID());  
    }  
}  
  
void TAREA_2(void* pvParameters)  
{  
    for(;;)  
    {  
        SECUENCIA_NUCLEO_1();  
    }  
}
```



```
Serial.print("Esto está corriendo en el núcleo: ");
Serial.println(xPortGetCoreID());
}

void SECUENCIA_NUCLEO_0()
{
    digitalWrite(GPIO_1_TAREA_1, HIGH);
    delay(500);
    digitalWrite(GPIO_2_TAREA_1, HIGH);
    delay(500);
    digitalWrite(GPIO_1_TAREA_1, LOW);
    delay(500);
    digitalWrite(GPIO_2_TAREA_1, LOW);
    delay(500);
}

void SECUENCIA_NUCLEO_1()
{
    digitalWrite(GPIO_1_TAREA_2, HIGH);
    delay(250);
    digitalWrite(GPIO_2_TAREA_2, HIGH);
    delay(250);
    digitalWrite(GPIO_1_TAREA_2, LOW);
    delay(250);
    digitalWrite(GPIO_2_TAREA_2, LOW);
    delay(250);
}

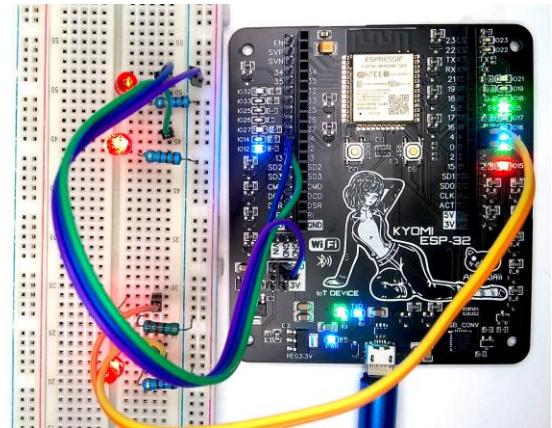
void Loop()
{
```



Prueba el programa. Verás que ambas secuencias de LEDs se ejecutan de manera independiente y a distinta velocidad:

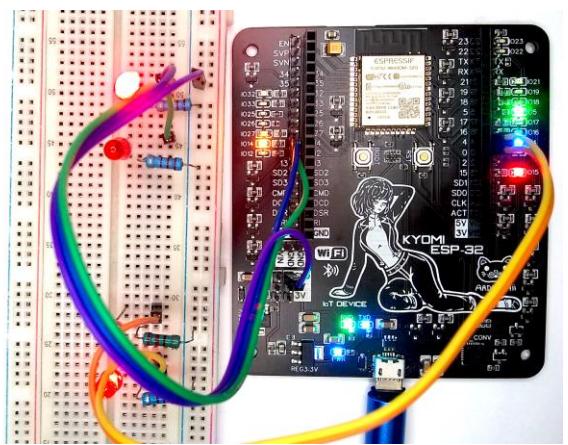
```
COM3
Esto está corriendo en el núcleo: 1
Esto está corriendo en el núcleo: 0
Esto está corriendo en el núcleo: 1
Esto está corriendo en el núcleo: 1

Autoscroll Mostrar marca temporal Nueva línea 115200 baudio Limpiar salida
```



```
COM3
Esto está corriendo en el núcleo: 1
Esto está corriendo en el núcleo: 0
Esto está corriendo en el núcleo: 1
Esto está corriendo en el núcleo: 1
Esto está corriendo en el núcleo: 0
Esto está corriendo en el núcleo: 1
Esto está corriendo en el núcleo: 1

Autoscroll Mostrar marca temporal Nueva línea 115200 baudio Limpiar salida
```



Puedes ejecutar más de dos tareas simultáneas en los dos núcleos, aunque necesitarás de algunos trucos de programación para lograrlo. Por ejemplo, las tareas creadas no necesariamente deben ser ciclos infinitos, así podrás crear más tareas y asignarles una prioridad, pero deberás llamarlas desde una rutina principal. Otra manera de ejecutar múltiples tareas es con la distribución de tiempos, es decir, asignar cierto tiempo de ejecución a cada tarea, parecido a lo que hacíamos con los temporizadores y la instrucción *millis()*.

Uno de los mejores métodos para ejecutar múltiples tareas al mismo tiempo es mediante el uso de un RTOS (sistema operativo en tiempo real), que es como el sistema operativo de tu computadora, que se encarga de distribuir y gestionar las tareas que ejecutará el procesador. Un RTOS es un sistema operativo bastante simple y pequeño, diseñado para correr en un microcontrolador. Uno de los más conocidos es FreeRTOS.

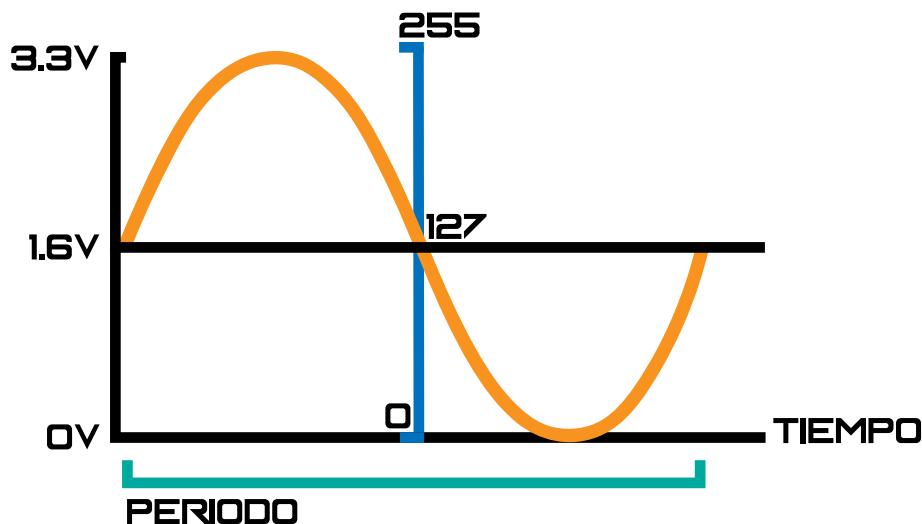


## El convertidor digital a analógico (DAC)

Como vimos en el capítulo 2, para convertir una señal analógica a valores digitales, se utiliza el convertidor analógico a digital (ADC). Con el convertidor digital a analógico (DAC) ocurre exactamente lo opuesto, convierte una serie de valores digitales (como los que obteníamos con el ADC) en una señal de voltaje analógico que puede variar con el tiempo.

Al igual que el ADC, el DAC tiene una resolución de bits (8 bits en este caso). Quiere decir que la señal de voltaje que puede generar (0V a 3.3V) se divide en 256 valores de amplitud (incluyendo el 0).

Digamos que queremos generar una señal sinusoidal de 2Hz con una amplitud máxima de 3.3V, entonces podemos partir del siguiente esquema:



Los 3.3V máximos se encontrarían en el valor 255 y los 0V mínimos en el 0. El voltaje medio de la señal se encuentra aproximadamente en el valor 127, que matemáticamente sería el 0 de la sinusoida, mientras que arriba del 0 estarían sus valores positivos y debajo sus valores negativos. Ya que usaremos la función matemática del seno, hay que sumar un 127 a su resultado para desplazar a la función hacia arriba, de lo contrario, también obtendríamos valores negativos, porque la función del seno entrega valores entre -1 y 1, que luego son multiplicados por una amplitud.

Con la función desplazada hacia arriba, el valor máximo de amplitud que podemos usar es 127, que sería el caso en el que el seno vale 1 y se multiplicaría por 127, a lo que después se le suma el 127 del desplazamiento, dando un total de 254. Por tratarse de una función periódica, la sinusoida tiene un periodo de tiempo (duración de un ciclo). En este caso queremos una sinusoidal de 2Hz, entonces su periodo es de 1/2Hz o 0.5 segundos (500ms). Matemáticamente, el periodo del seno es de 360°, si dividimos el periodo de tiempo entre los grados (0.5s/360),



obtendríamos 1.38ms, que es el tiempo que dura cada paso o grado. Con estos datos, podemos desarrollar el siguiente programa:

```
int GRADOS=0, SENO=0, AMPLITUD=127;  
  
void setup()  
{  
}  
  
void Loop()  
{  
    for(GRADOS=0; GRADOS<360; GRADOS++)  
    {  
        SENO=AMPLITUD*sin((GRADOS*PI)/180);  
        dacWrite(25,(127+SENO));  
        delayMicroseconds(1388);  
    }  
}
```

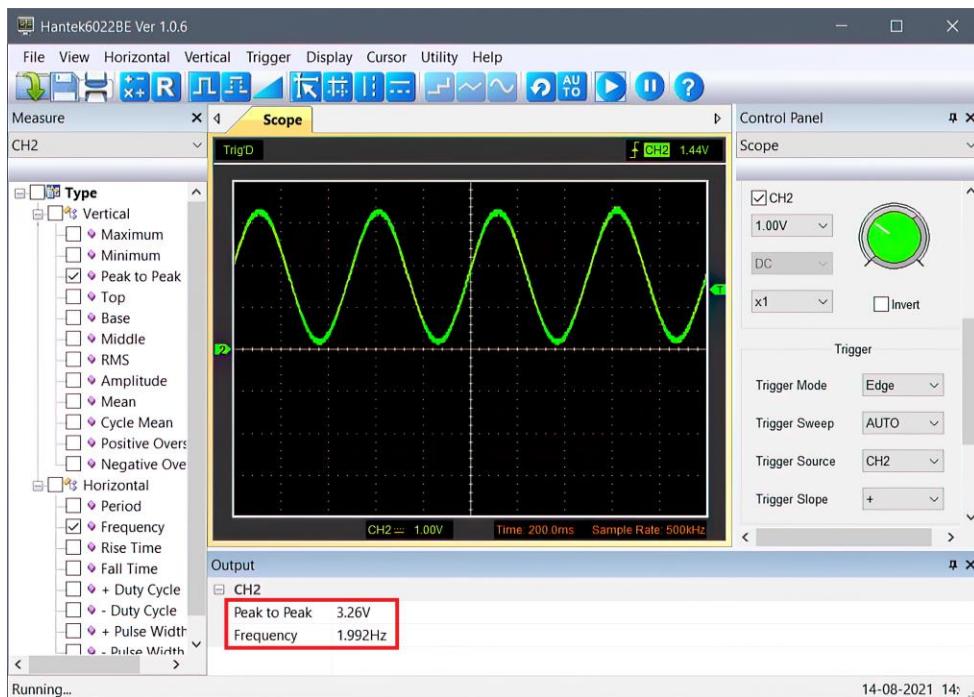
la instrucción `dacWrite()` es la que nos va a permitir usar el DAC. Recuerda que el ESP32 cuenta con dos GPIO's con salidas DAC independientes (revisa el pinout). En `dacWrite()` debemos colocar el GPIO que vamos a usar (en este caso el 25) y luego el valor de la señal (el resultado de `127+SENO`). El `for(GRADOS=0; GRADOS<360; GRADOS++)` va sumando los pasos de los grados para la función del seno.

La variable `SENO` contiene la función sinusoidal: `AMPLITUD` multiplica al resultado de la función `sin` para darle la amplitud, mientras que `sin` va cambiando su valor de acuerdo al producto de `GRADOS` por el número pi, pero hay que dividir este producto entre 180 para pasar su resultado a radianes.

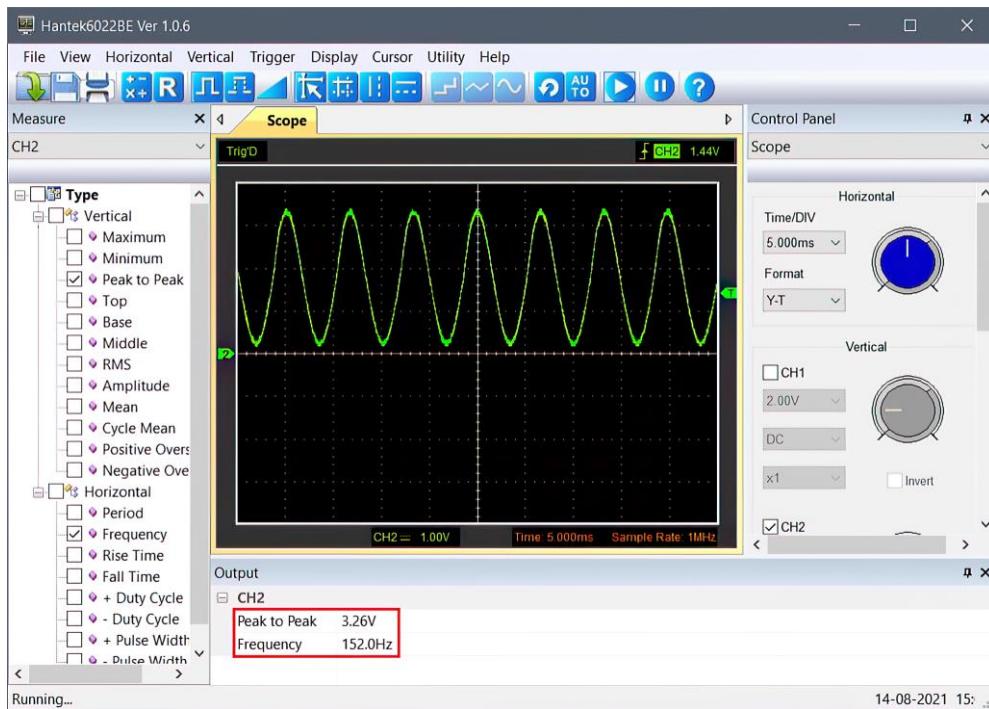
En `delayMicroseconds` vamos a colocar el tiempo que calculamos para cada grado (1.38ms o 1388μs), así controlaremos la frecuencia, ya que la suma de todas estas pausas o la duración total del `for` da como resultado 500ms (2Hz). Con esto generaríamos un ciclo de la señal, que se repetirá en cada ejecución del `Loop()`.



Si medimos la señal generada con un osciloscopio, veremos que el voltaje entre el pico superior e inferior de la señal casi llega a los 3.3V máximos, mientras que la frecuencia se aproxima a los 2Hz:



Si no tienes osciloscopio, más adelante haremos un ejercicio para medir las señales generadas usando el ADC. Si quitáramos a `delayMicroseconds`, forzaríamos la velocidad máxima de `loop()` para obtener la frecuencia máxima de `dacWrite()`, unos 152Hz aproximadamente:





Se pueden obtener frecuencias más altas si optimizamos el código, por ejemplo, usar un array que contenga todos los valores del seno para llevarlos directamente a `dacWrite()`, evitando tener que calcular cada uno.

Ahora hagamos un ejercicio para generar un par de señales con el DAC.

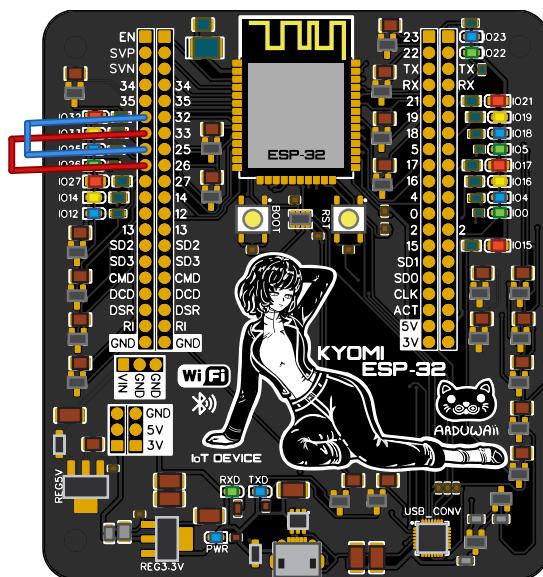
## Ejercicio 27

En este ejercicio vamos a generar dos señales distintas con los dos GPIO's DAC, que luego mediremos con dos GPIO's del ADC y mostraremos sus lecturas en el Graficador Serial. Usaremos el núcleo O del ESP32 para las mediciones del ADC.

Materiales:

-2 jumpers o cables tipo Dupont

El circuito es el siguiente:



Ahora veamos el código. Vamos a declarar unas cuantas variables y crearemos la tarea que hará la lectura de las señales. Generaremos una señal senoidal y una cosenoidal, ambas con amplitud máxima de 3.3V:

```
int GRADOS=0, SENO=0, COSENO=0, AMPLITUD=127;  
TaskHandle_t LECTURA_ADC;
```

En `setup()` vamos a configurar la tarea `LECTURA_ADC` para asignarla al núcleo O y también configuraremos la comunicación serial, esta vez a 250000 baudios para enviar los datos más rápido (no olvides seleccionar esta velocidad en el Graficador Serial):

```
xTaskCreatePinnedToCore(  
    LECTURA_Analogica,  
    "ADC",
```



```
10000,  
NULL,  
1,  
&LECTURA_ADC,  
0);  
  
Serial.begin(250000);
```

En `Loop()` vamos a generar la señal senoidal del ejemplo anterior, pero agregaremos la función `cos` para generar la señal cosenoidal, usando las mismas variables con ambas funciones (también funcionarán a 2Hz):

```
for(GRADOS=0; GRADOS<360; GRADOS++)  
{  
    SENO=AMPLITUD*sin((GRADOS*PI)/180);  
    COSENO=AMPLITUD*cos((GRADOS*PI)/180);  
    dacWrite(25,(127+SENO));  
    dacWrite(26,(127+COSENO));  
    delayMicroseconds(1388);  
}
```

La rutina `LECTURA_Analogica()` la usaremos para mostrar las lecturas del ADC en el Graficador Serial. El retraso de `delayMicroseconds` dura el triple del que se usó en `loop()`, para ver un poco más lento a las señales del Graficador Serial:

```
void LECTURA_Analogica(void* pvParameters)  
{  
    while(true)  
    {  
        Serial.print(analogRead(32));  
        Serial.print("\t");  
        Serial.println(analogRead(33));  
        delayMicroseconds(4164);  
    }  
}
```

Ya que `loop()` corre en el núcleo 1, la rutina `LECTURA_Analogica()` es independiente de la generación de las señales y no afecta a su comportamiento.



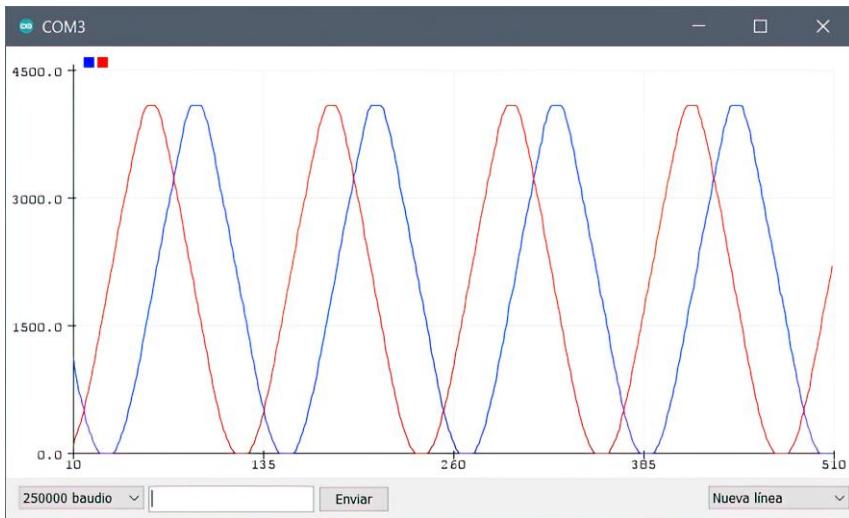
El código completo queda de la siguiente forma:

```
int GRADOS=0, SENO=0, COSENO=0, AMPLITUD=127;  
  
TaskHandle_t LECTURA_ADC;  
  
  
void setup()  
{  
    xTaskCreatePinnedToCore(  
        LECTURA_Analogica,  
        "ADC",  
        10000,  
        NULL,  
        1,  
        &LECTURA_ADC,  
        0);  
  
    Serial.begin(250000);  
}  
  
  
void Loop()  
{  
    for(GRADOS=0; GRADOS<360; GRADOS++)  
    {  
        SENO=AMPLITUD*sin((GRADOS*PI)/180);  
        COSENO=AMPLITUD*cos((GRADOS*PI)/180);  
        dacWrite(25,(127+SENO));  
        dacWrite(26,(127+COSENO));  
        delayMicroseconds(1388);  
    }  
}  
  
  
void LECTURA_Analogica(void* pvParameters)  
{  
    while(true)  
    {  
        Serial.print(analogRead(32));  
        Serial.print("\t");  
    }  
}
```



```
Serial.println(analogRead(33));  
delayMicroseconds(4164);  
}  
}
```

Prueba el programa. Verás el desplazamiento entre la señal senoidal y cosenoidal (de 90° aprox.):



## Guarda datos en la Flash mediante SPIFFS

El sistema de archivos SPIFFS (Serial Peripheral Interface Flash File System) nos permite guardar archivos pequeños en la memoria Flash del ESP32, similar a como lo hacemos en una computadora.

Con SPIFFS podemos guardar archivos de texto o documentos HTML en la Flash, para luego acceder a estos desde nuestro programa. La gran desventaja de SPIFFS es que no soporta subdirectorios, es decir, que todos los archivos que guardemos deben encontrarse en la ubicación raíz, por lo que no podremos organizarlos dentro de carpetas.

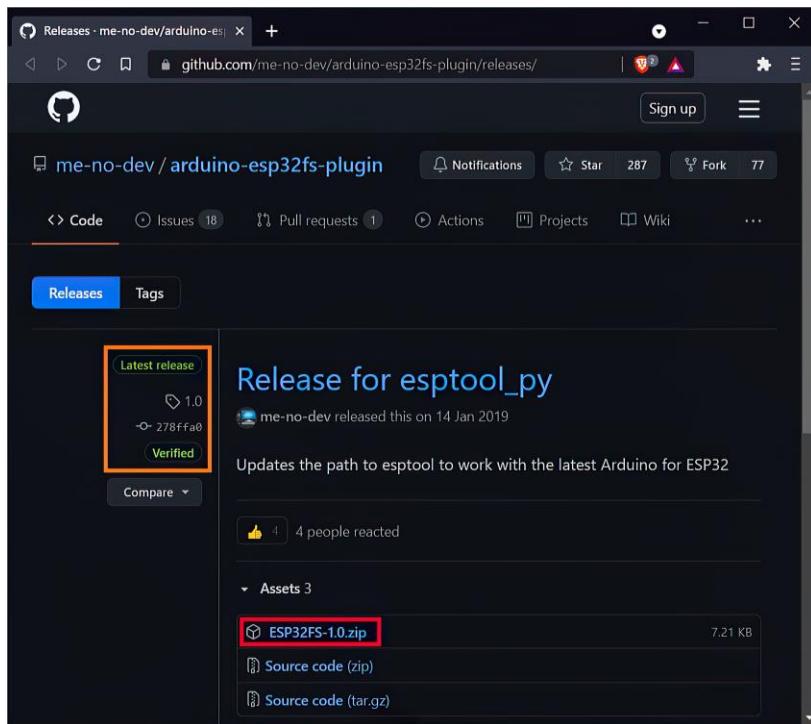
SPIFFS ya viene integrado en los paquetes de complementos para el ESP32, pero tendremos que instalar la herramienta que nos permitirá subir los archivos a la Flash.

Primero deberás descargar la herramienta ‘esp32fs-plugin’ del siguiente repositorio de GitHub:

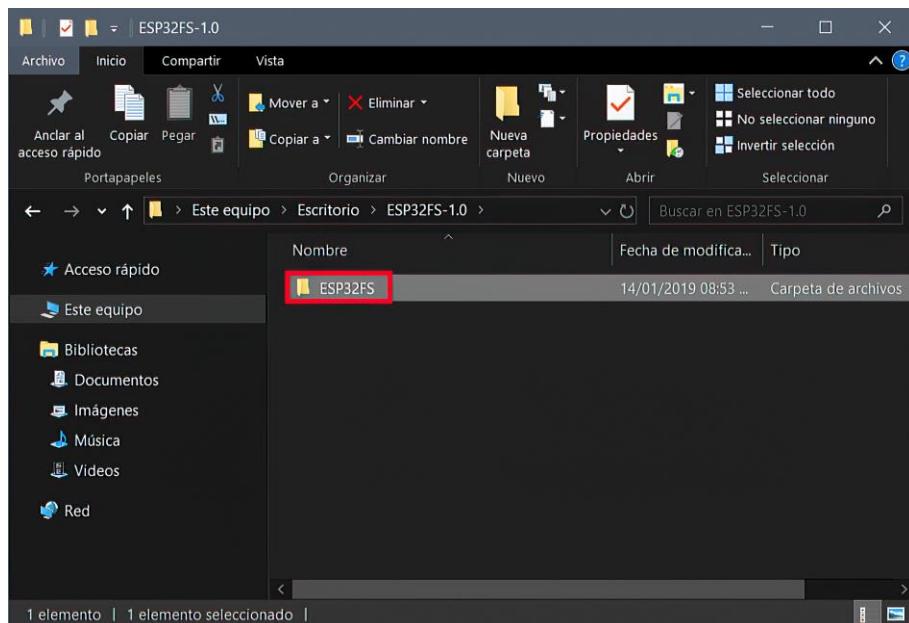
<https://github.com/me-no-dev/arduino-esp32fs-plugin/releases/>



recuerda descargarla en .zip:



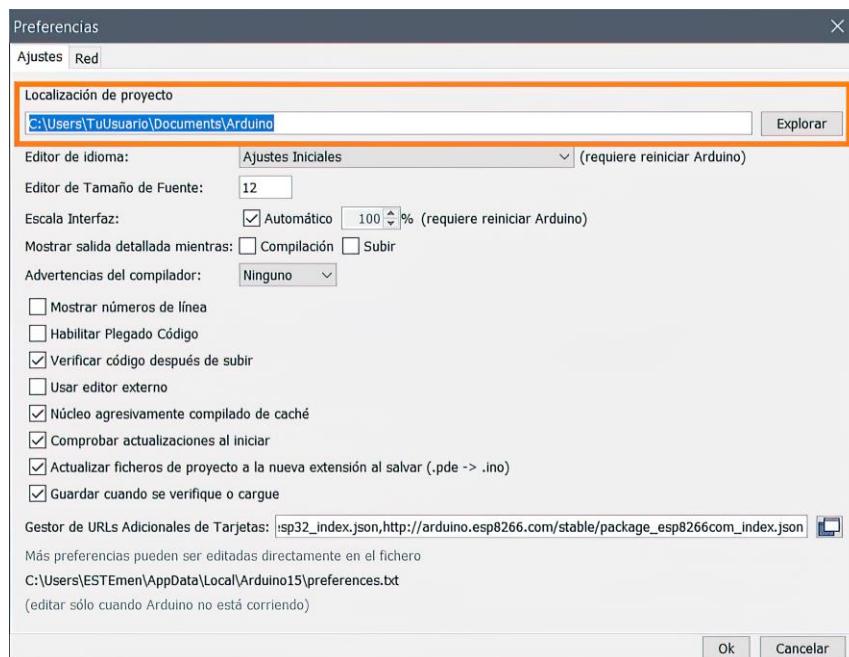
Una vez descargado, descomprime el zip para obtener la carpeta ‘ESP32FS’, que contiene los archivos necesarios para añadir la herramienta al IDE de Arduino:



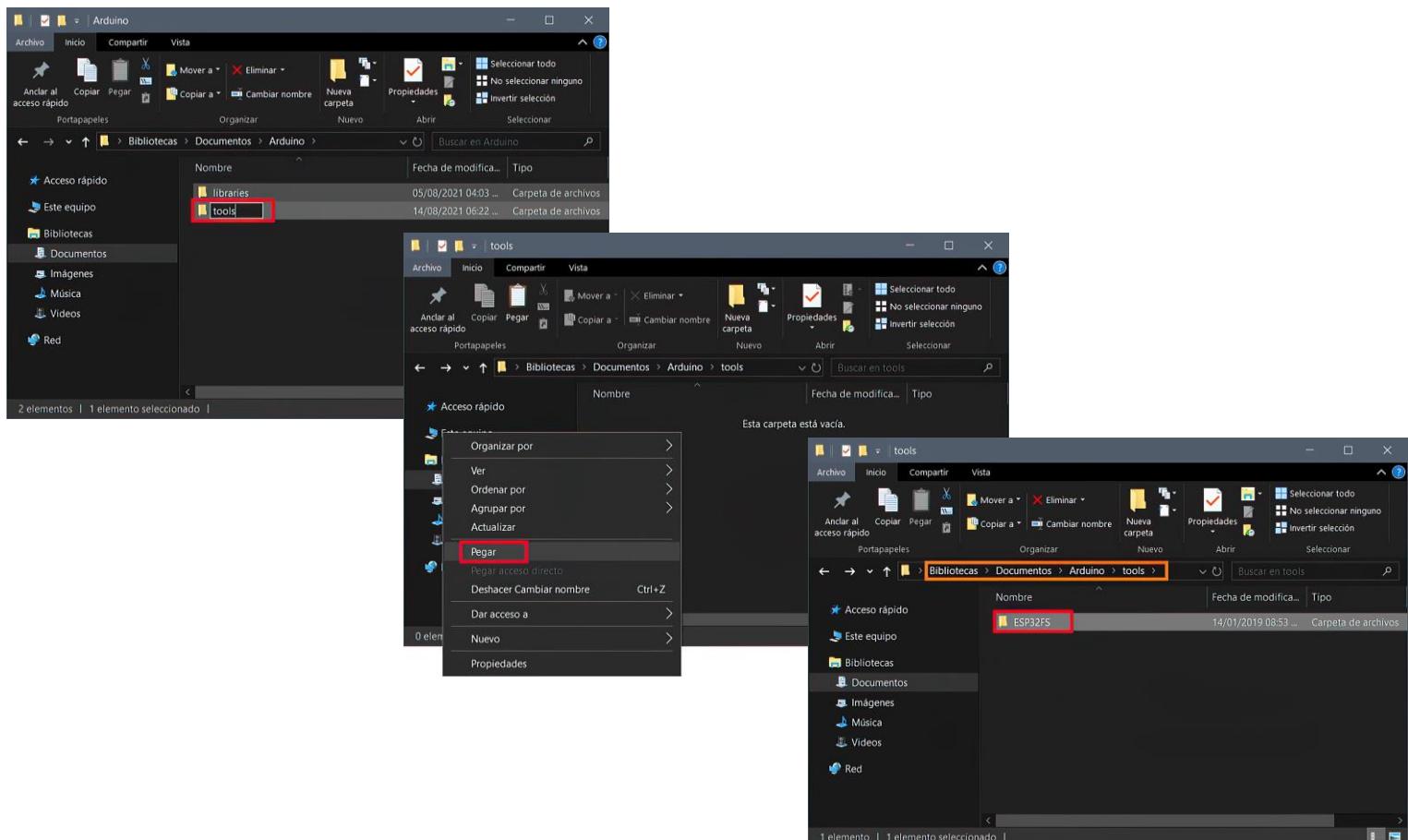
Deberás mover o copiar la carpeta ESP32FS a la carpeta ‘tools’, que se encuentra dentro de la carpeta de configuraciones de Arduino.



Puedes ver su ubicación en la pestaña Archivo > Preferencias:

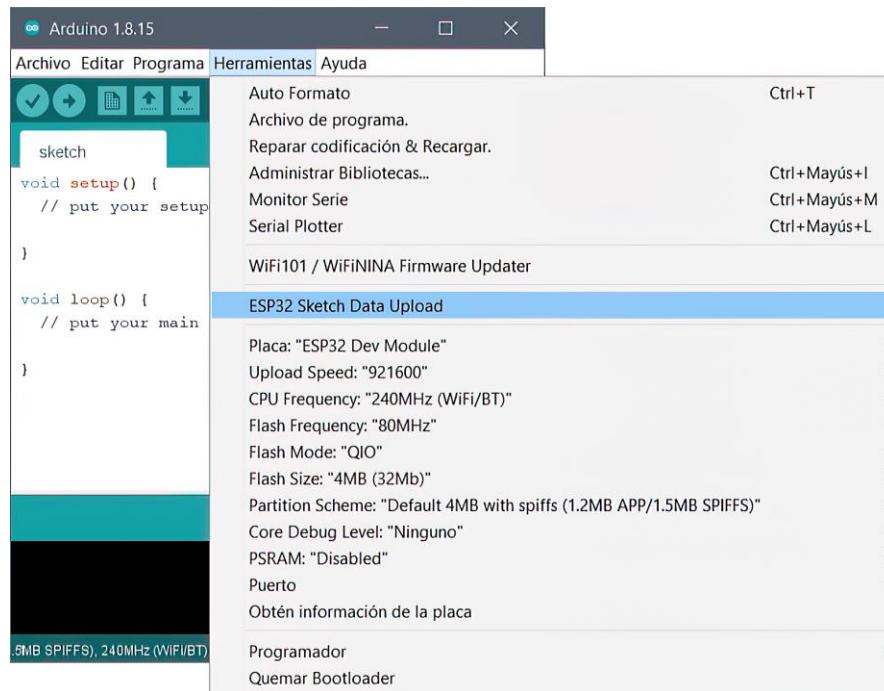


Si no tienes la carpeta tools en tu carpeta Arduino, puedes crearla y luego mover la carpeta ESP32FS a la misma:



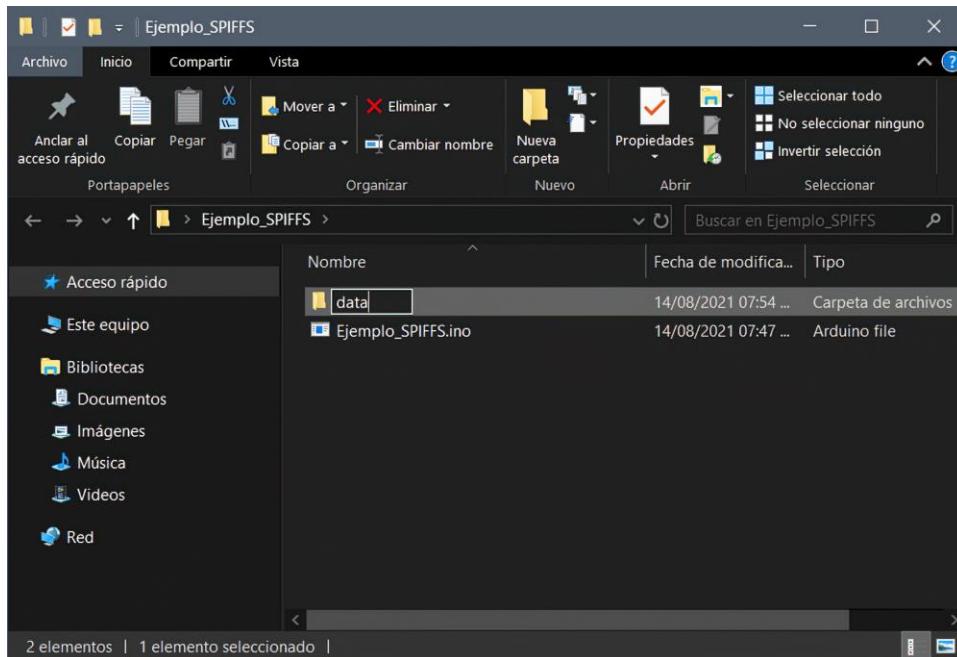


Una vez que tengas la carpeta **ESP32FS** en tools y vuelvas a iniciar el IDE de Arduino, verás una nueva opción en la pestaña de Herramientas:



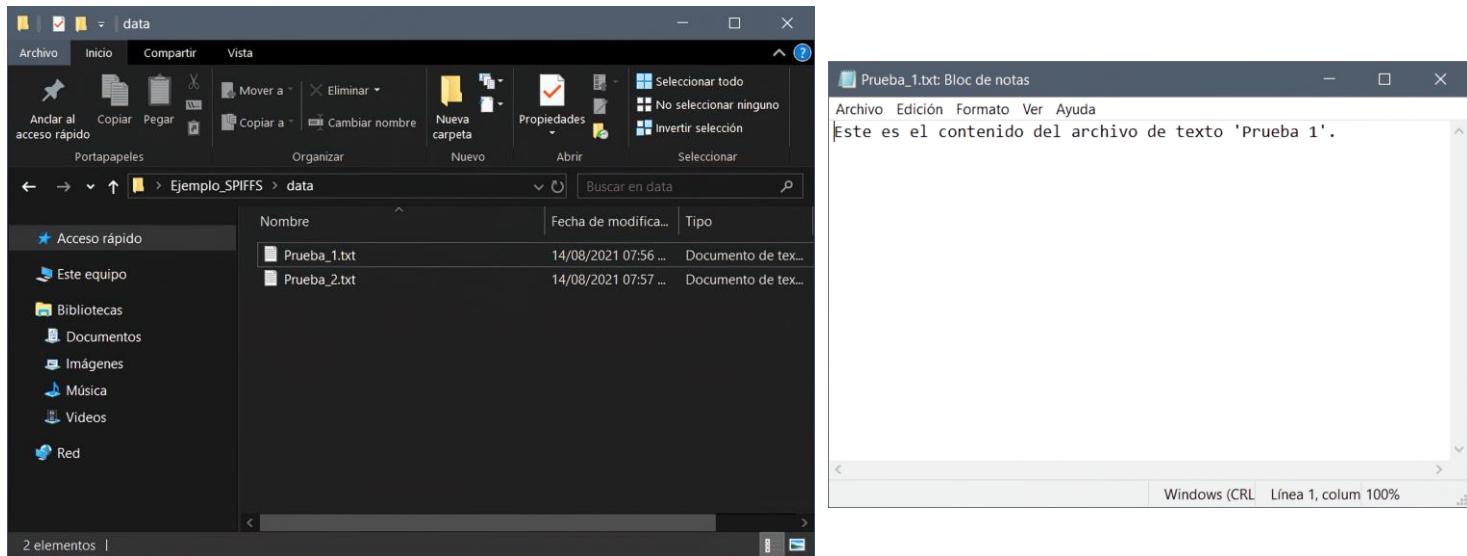
Con la herramienta ‘ESP32 Sketch Data Upload’ podremos subir y guardar archivos directamente en la Flash.

Ahora veamos un ejemplo. Digamos que queremos guardar un par de archivos de texto en la Flash para luego abrirlos. Antes de subir los archivos, primero tienes que crear una carpeta llamada ‘data’, que tiene que estar junto con el archivo .ino de tu programa:

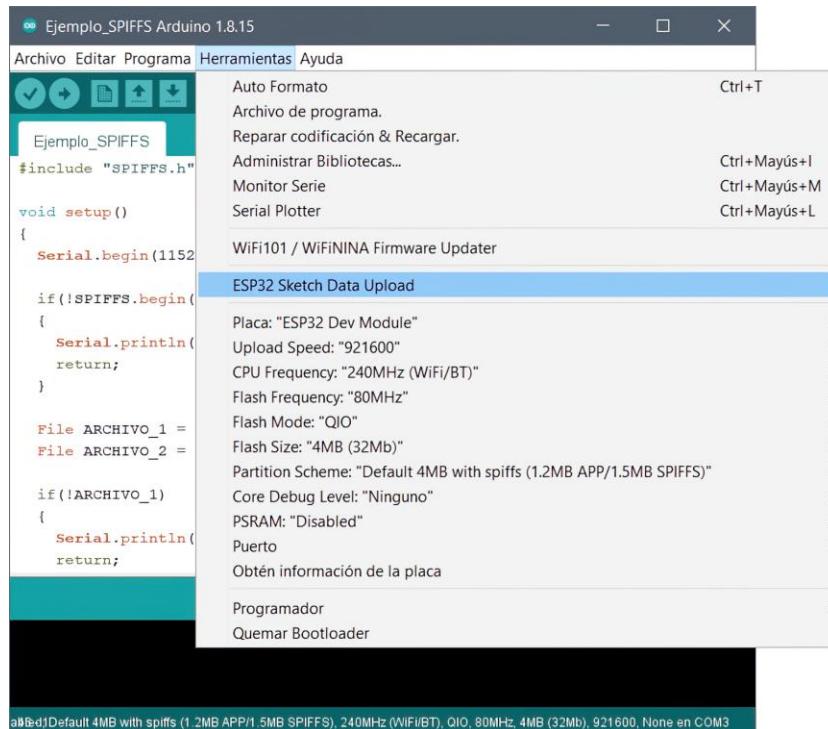




Dentro de la carpeta ‘data’ podrás colocar tus archivos de texto, pero recuerda que todos deben estar en la raíz de la misma, ya que SPIFFS no soporta subdirectorios:

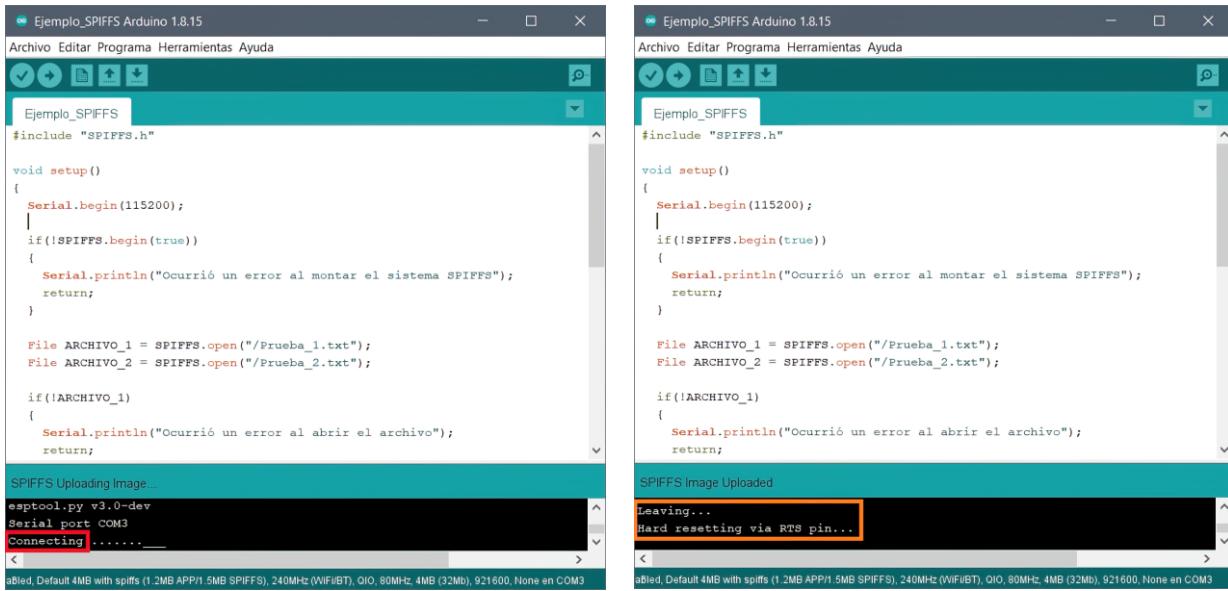


En el IDE de Arduino, tendrás que subir los archivos a la Flash antes de cargar tu programa a la tarjeta. Este proceso siempre formatea la Flash, por lo que cualquier archivo guardado anteriormente se perderá:





Al igual que al cargar un programa, deberás presionar el botón BOOT de la tarjeta cuando comience la comunicación:



Con los archivos ya guardados en la Flash, ahora veamos el código. Para usar las funciones del sistema SPIFFS, necesitaremos incluir la librería *SPIFFS.h*. En este ejemplo, prácticamente todo ocurre dentro de *setup()*:

```
#include "SPIFFS.h"

void setup()
{
    Serial.begin(115200);
    if(!SPIFFS.begin(true))
    {
        Serial.println("Ocurrió un error al montar el sistema SPIFFS");
        return;
    }

    File ARCHIVO_1 = SPIFFS.open("/Prueba_1.txt");
    File ARCHIVO_2 = SPIFFS.open("/Prueba_2.txt");

    if(!ARCHIVO_1)
    {
        Serial.println("Ocurrió un error al abrir el archivo");
        return;
    }
}
```



```
if(!ARCHIVO_2)
{
    Serial.println("Ocurrió un error al abrir el archivo");
    return;
}

Serial.println("Contenido de Prueba_1.txt:");
while(ARCHIVO_1.available())
{
    Serial.write(ARCHIVO_1.read());
}
ARCHIVO_1.close();

Serial.println("");
Serial.println("Contenido de Prueba_2.txt:");
while(ARCHIVO_2.available())
{
    Serial.write(ARCHIVO_2.read());
}
ARCHIVO_2.close();

void loop()
{}
```

El código funciona de la siguiente manera: después de configurar la comunicación serial, el `if(!SPIFFS.begin(true))` comprueba que el sistema SPIFFS haya iniciado correctamente (cuando `SPIFFS.begin` es `true`), pero si `SPIFFS.begin` es distinto de `true`, entonces SPIFFS no pudo iniciar y se enviará un mensaje de error al Monitor Serie. Si inició correctamente, el programa continuará.

Con la instrucción `File` crearemos los objetos `ARCHIVO_1` y `ARCHIVO_2`, a los cuales les asignaremos los archivos que guardamos en la Flash: `Prueba_1.txt` y `Prueba_2.txt` (que se encuentran en la raíz o `/`). Los siguientes dos `if` comprobarán si los archivos asignados a los objetos `ARCHIVO_1` y `ARCHIVO_2` fueron encontrados, si no es así, entonces se enviará un mensaje de error al Monitor Serie.



Con los ciclos `while` abriremos los archivos de texto para enviar su contenido al Monitor Serie. Por ejemplo, `while(ARCHIVO_1.available())` se ejecuta siempre y cuando `ARCHIVO_1` esté disponible y aún tenga información para mostrar, la cual se envía al Monitor Serie mediante `Serial.write(ARCHIVO_1.read())`. Cuando se haya mostrado todo el contenido, el ciclo `while` terminará y podremos cerrar el archivo de texto con la instrucción `ARCHIVO_1.close()`. El siguiente `while` funciona de manera similar.

Al abrir el Monitor Serie, el contenido de los archivos de texto se mostrará:

```
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:10944
load:0x40080400,len:6388
entry 0x400806b4

Contenido de Prueba_1.txt:
Este es el contenido del archivo de texto 'Prueba 1'.
Contenido de Prueba_2.txt:
Este es el contenido del archivo de texto 'Prueba 2'.
```

Autoscroll  Mostrar marca temporal  Nueva línea  Limpiar salida

Abrir archivos de texto plano es así de fácil, pero con otros tipos de archivos tendremos que recurrir a funciones diferentes.

Veamos un ejemplo más práctico. Una de las mejores formas para aprovechar la Flash, es usarla para guardar nuestros documentos HTML y CSS cuando trabajamos con un servidor web asíncrono.

En el capítulo 4, el código CSS lo colocábamos dentro del mismo documento HTML, pero podemos crear un archivo CSS por separado al que llamaremos desde el documento HTML principal. El código CSS que escribíamos entre las etiquetas `<style>` y `</style>` (dentro del documento HTML) podemos colocarlo tal cual en un archivo CSS.

Para crear un archivo CSS, podemos recurrir a cualquier editor de texto, como el bloc de notas.

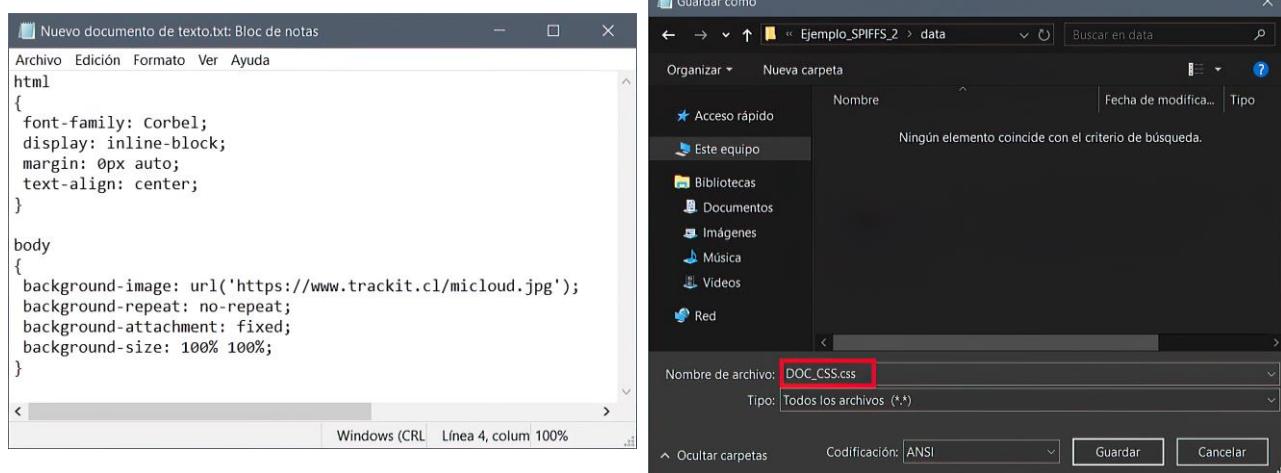


Por ejemplo, tenemos el siguiente código de estilos CSS:

```
html
{
    font-family: Corbel;
    display: inline-block;
    margin: 0px auto;
    text-align: center;
}

body
{
    background-image: url('https://www.trackit.cl/micloud.jpg');
    background-repeat: no-repeat;
    background-attachment: fixed;
    background-size: 100% 100%;
}
```

si trabajamos con el bloc de notas, debemos guardar el archivo con el código usando la extensión .css:



Ahora, en el documento HTML ya no usaremos las etiquetas `<style>` y `</style>`, sino que llamaremos al archivo CSS mediante una etiqueta `<link>`. Por ejemplo, tenemos el siguiente código HTML:

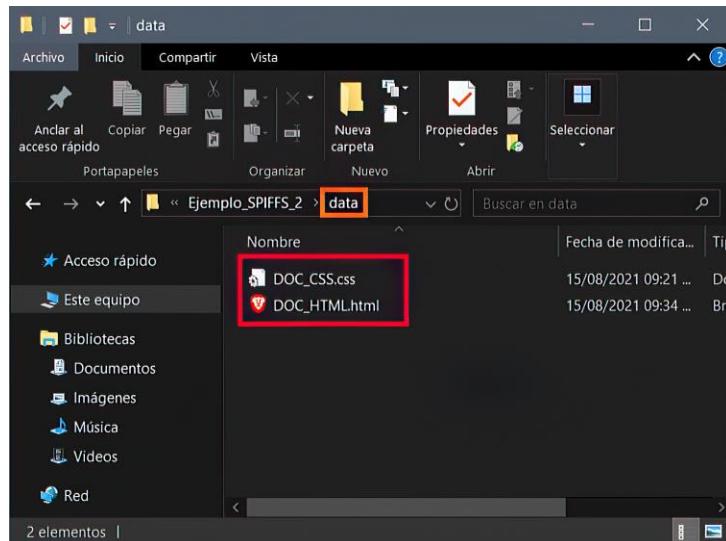
```
<!DOCTYPE HTML>
<html>
<head>
    <title>EJEMPLO SPIFFS</title>
```



```
<meta name='viewport' content='width=device-width, initialscale=1' http-equiv='Content-Type' content='text/html; charset=UTF-8' />  
<link rel='icon' href='data:, '>  
<link rel="stylesheet" type="text/css" href="DOC_CSS.css">  
</head>  
  
<body>  
    <h1 style='color:#BAEFFF; font-size: 32px;'>ESTE DOCUMENTO HA SIDO LEÍDO DESDE  
SPIFFS</h1>  
</body>  
</html>
```

la segunda etiqueta `<link>` se usa para llamar a una hoja de estilos CSS (`rel="stylesheet"`), que debe contener código tipo CSS (`type="text/css"`). En `href` debemos colocar el nombre con el que guardamos el archivo CSS, en este caso `DOC_CSS.css`. También guardaremos este documento HTML.

Los archivos CSS y HTML los colocamos en la carpeta data, que deberemos crear previamente (no olvides colocarla dentro de la carpeta de tu programa, junto con el archivo .ino):



Ahora veamos el código de Arduino. La estructura del programa es muy similar a la del ejercicio 22, solo que ahora no utilizaremos un `const char` para guardar la página web ni funciones para cambiar elementos de la misma, ya que no colocamos instrucciones JavaScript en el documento HTML:

```
#include <WiFi.h>  
#include "ESPAsyncWebServer.h"  
#include "SPIFFS.h"
```



```
const char* ssid = "AQUÍ VA EL NOMBRE DE TU RED";
const char* password = "AQUÍ VA LA CONTRASEÑA DE TU RED";
AsyncWebServer server(1000);

IPAddress Local_IP(192, 168, 0, 200);
IPAddress gateway(192, 168, 200, 200);

IPAddress subnet(255, 255, 0, 0);
IPAddress primaryDNS(8, 8, 8, 8);
IPAddress secondaryDNS(8, 8, 4, 4);

void setup()
{
    Serial.begin(115200);
    if(!SPIFFS.begin(true))
    {
        Serial.println("Ocurrió un error al montar el sistema SPIFFS");
        return;
    }

    WiFi.config(Local_IP, gateway, subnet, primaryDNS, secondaryDNS);
    Serial.print("Conectando a:");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("Conectado");
    Serial.println("Dirección IP: ");
    Serial.println(WiFi.localIP());

    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request)
    {request->send(SPIFFS, "/DOC_HTML.html", String(), false, ENVIAR_DATOS);});
}
```



```
server.on("/DOC_CSS.css", HTTP_GET, [](AsyncWebServerRequest *request)
{request->send(SPIFFS, "/DOC_CSS.css", "text/css");});

server.begin();
}

void Loop()
{
}

String ENVIAR_DATOS(const String& var)
{
    return String();
}
```

las modificaciones importantes están en `setup()`. El `if(!SPIFFS.begin(true))` es el mismo del ejemplo anterior. En el ejercicio 22 usábamos la siguiente instrucción para enviar la página web al cliente:

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request)
{request->send_P(200, "text/html", PAGINA_WEB, ENVIAR_DATOS);});
```

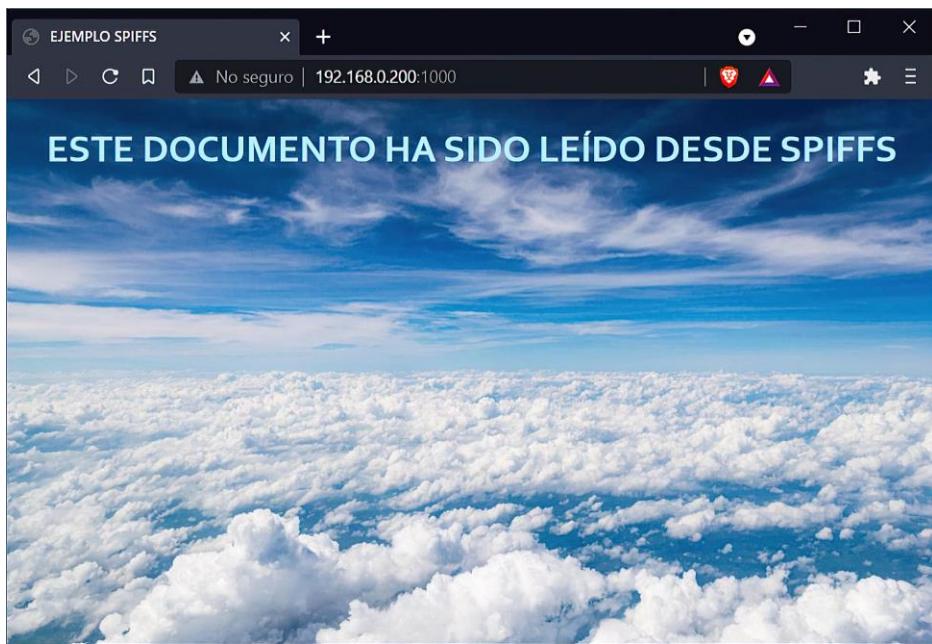
ahora la hemos reemplazado por las siguientes dos:

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request)
{request->send(SPIFFS, "/DOC_HTML.html", String(), false, ENVIAR_DATOS);});
server.on("/DOC_CSS.css", HTTP_GET, [](AsyncWebServerRequest *request)
{request->send(SPIFFS, "/DOC_CSS.css", "text/css");});
```

donde el primer `request->send` tiene el parámetro `SPIFFS` y envía el contenido del archivo `DOC_HTML.html` al cliente en forma de `String` (también manda a llamar a la subrutina `ENVIAR_DATOS()`, pero en este caso no hará nada). Como el documento HTML requerirá el contenido del archivo CSS, también hay que enviar este último al cliente con un segundo `request->send`, que abrirá el archivo `DOC_CSS.css` y lo mandará como `text/css`.



Cuando pruebas el programa, la página web será descargada de la memoria Flash y no desde una variable (no olvides subir los archivos a la Flash):



Manejar los archivos HTML y CSS por separado ofrece varias ventajas, por ejemplo, podemos modificar los estilos CSS directamente sin tocar el documento HTML, o también modificar algunos aspectos de la página web sin tener que hacerlo desde el código de Arduino. Claro que, al modificar los archivos, deberás volver a cargarlos a la Flash, pero no necesariamente tendrás que hacer lo mismo con el programa.

## Circuito de potencia sencillo

El ESP32, por sí solo, no es capaz de alimentar cargas grandes o dispositivos de gran consumo energético a través de los GPIO's. Para lograr esto, se requiere un circuito de potencia, el cual es controlado por las pequeñas señales de los GPIO's, para controlar a su vez una carga grande, como un motor. El circuito con relevador del ejercicio 19 podría considerarse como un circuito de potencia de corriente alterna, pero las únicas acciones de control que puede realizar son encender o apagar la carga.

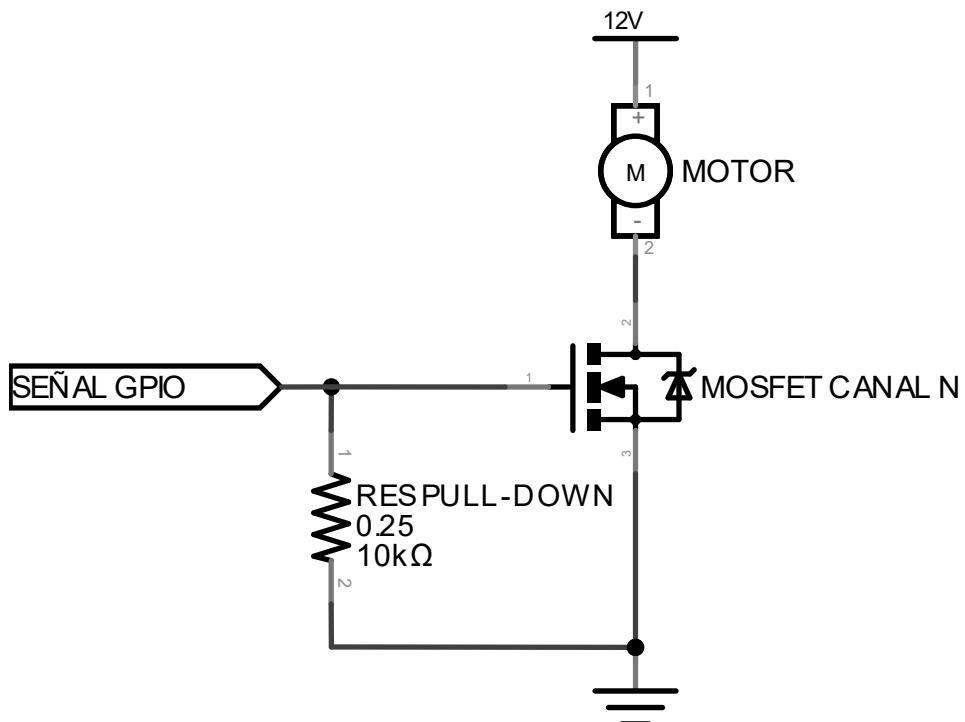
Los circuitos de potencia que trabajan con corriente directa pueden encender o apagar una carga, pero no solo eso, también permiten controlar la energía que consume, haciendo que trabaje entre el estado totalmente apagado y totalmente encendido. Esto se logra con el uso de transistores.



Podríamos controlar cargas de corriente directa con un relevador, pero ya se mencionó que está limitado al encendido y apagado simple. Al usar un transistor, podemos regular la corriente que pasa por la carga a mediante una señal de control.

Los transistores más utilizados para aplicaciones con microcontroladores son los de tipo MOSFET. Estos transistores fueron diseñados para actuar como interruptores rápidos, es decir, que permitan la conducción total de corriente o impidan su paso, pero a frecuencias muy altas. Este comportamiento es similar al de las señales digitales que generan los GPIO's. Ahora, si la señal que controla a un MOSFET es digital, solamente podríamos colocarlo en estado *high* (conducción de corriente) o en *low* (impedir el paso de corriente). En el capítulo 3, vimos que las señales PWM tienen cierta naturaleza digital, pero la modulación de su ciclo de trabajo es lo que permite variar el voltaje 'promediado' que realmente generan. Si controlamos el MOSFET con una señal PWM, obtendríamos un efecto similar, pero de mayor potencia.

Ahora bien, ¿cómo podemos usar un MOSFET para controlar una carga?, pues similar a como construimos el circuito con relevador. Existen MOSFET de canal N y canal P, pero nos centraremos en los de canal N. Los MOSFET tienen tres terminales: compuerta (Gate), drenaje (Drain) y fuente (Source). Por ejemplo, tenemos el siguiente circuito de potencia:



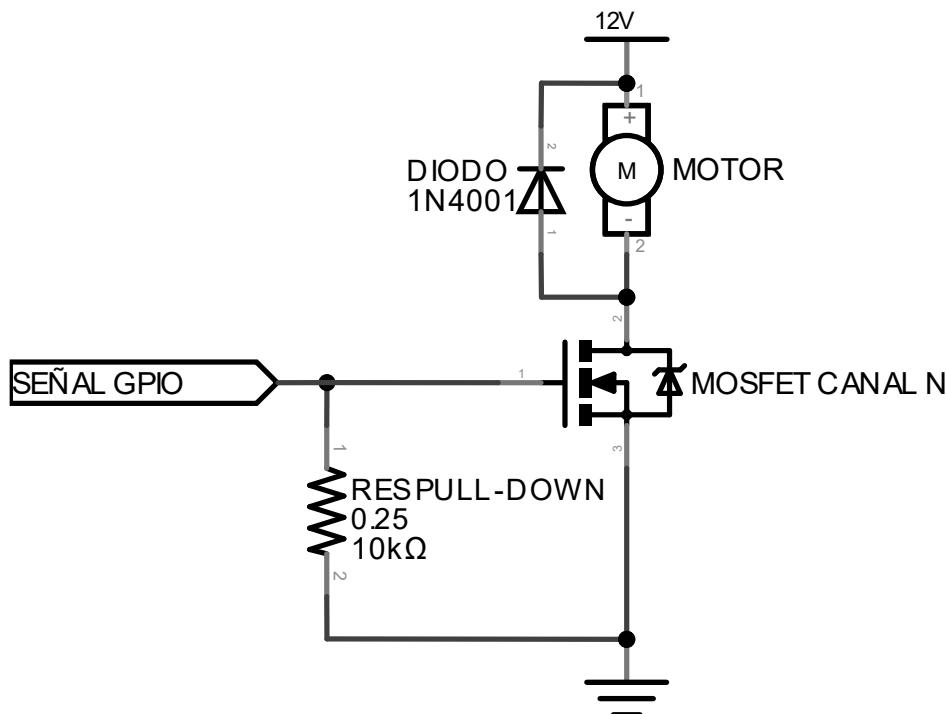


la compuerta del MOSFET (pin 1) está conectada al GPIO, la fuente (pin 3) está conectada a negativo o tierra y el drenaje (pin 2) está conectado a una de las terminales del motor de corriente directa. Este motor está alimentado por 12V. Podemos asemejar a las terminales de fuente y drenaje con las terminales de un interruptor.

Cuando el MOSFET está desactivado (aplicando un *Low* en la compuerta), no circula corriente de drenaje a fuente (como un interruptor abierto). Al activar el MOSFET (aplicando un *high* en la compuerta), la corriente circula de drenaje a fuente (como un interruptor cerrado) y se completa el circuito, haciendo funcionar al motor. Pero esto ocurre muchas veces por segundo al aplicar una señal PWM en la compuerta, obteniendo el efecto de voltaje variable en el motor.

La resistencia PULL-DOWN (conectada a la compuerta) siempre se recomienda colocarla, ya que la compuerta de los MOSFET puede retener unas cuantas cargas eléctricas que podrían activarlo.

En los circuitos de potencia como el anterior, si se van a usar para controlar motores, se debe colocar un diodo en paralelo e invertido entre las terminales del motor:



este diodo es una protección para el MOSFET y evita que se dañe por la fuerza contraelectromotriz (voltaje inverso) generada por las cargas que puede retener el motor (cuando se detiene), las cuales circularán en sentido contrario. El diodo provee una trayectoria para que estas cargas circulen por él, disipándolas en forma de calor.



El MOSFET que vayas a utilizar dependerá del tipo de carga que controlarás. El IRFZ44N es un MOSFET de canal N muy popular, que funciona bien en aplicaciones que trabajen hasta con 50V:



Hagamos un ejercicio práctico para aplicar el circuito de potencia anterior.

## Ejercicio 28

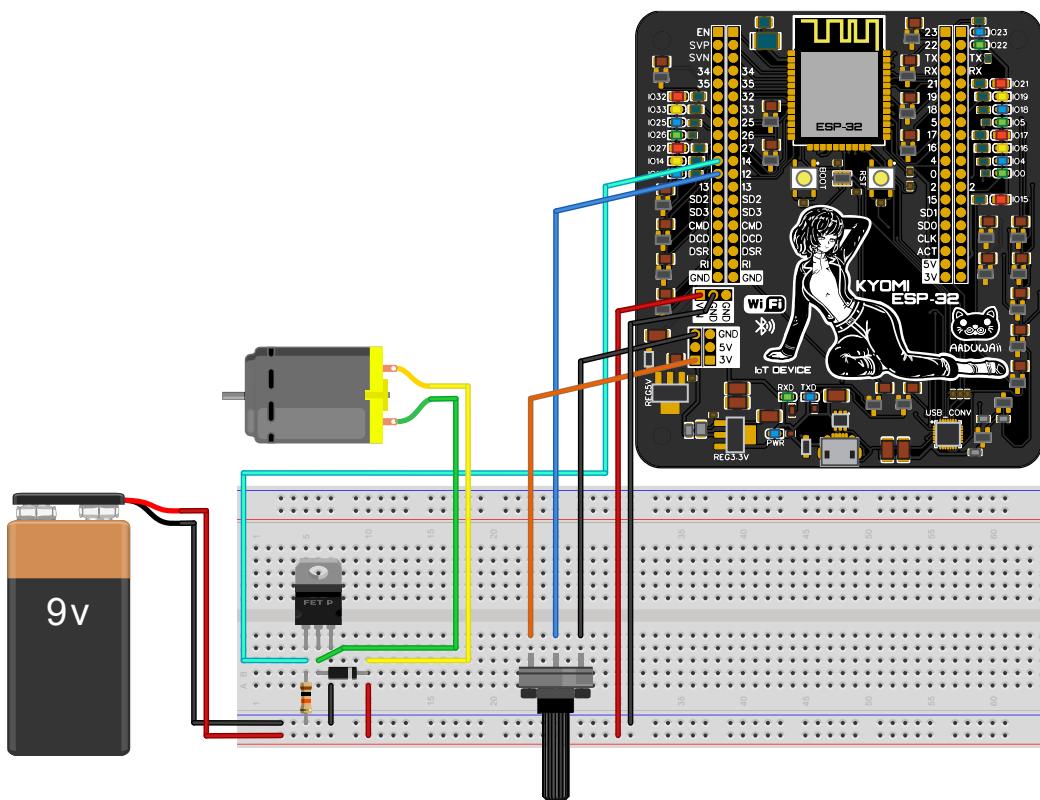
En este ejercicio construiremos un controlador de velocidad para un motor pequeño. El ESP32 leerá una señal analógica controlada por un potenciómetro, utilizando el ADC. Con los valores de la conversión se modulará una señal PWM, que a su vez controla a un circuito de potencia.

Materiales:

- 1 potenciómetro de 5KΩ
- 1 resistencia de 10KΩ
- 1 motor de corriente directa pequeño de 12V
- 1 batería de 9V o una fuente de 12V
- 1 diodo rectificador 1N4001, 1N4002, 1N4003 o 1N4004
- 1 MOSFET IRFZ44N
- 12 jumpers o cables tipo Dupont
- plantilla de experimentos



El circuito es el siguiente:



La batería de 9V puedes reemplazarla por una fuente de 12V. Al ESP32 lo alimentaremos con la misma fuente que alimenta al circuito de potencia. Recuerda que puedes alimentar tu tarjeta KYOMI con 6V hasta 12V a través del pin VIN.

Veamos el código. El programa es muy sencillo, simplemente configuraremos una señal PWM a 20KHz y 12bits, la cual regulará su ciclo de trabajo mediante los valores que entrega el ADC, que resultan de la lectura del voltaje del potenciómetro:

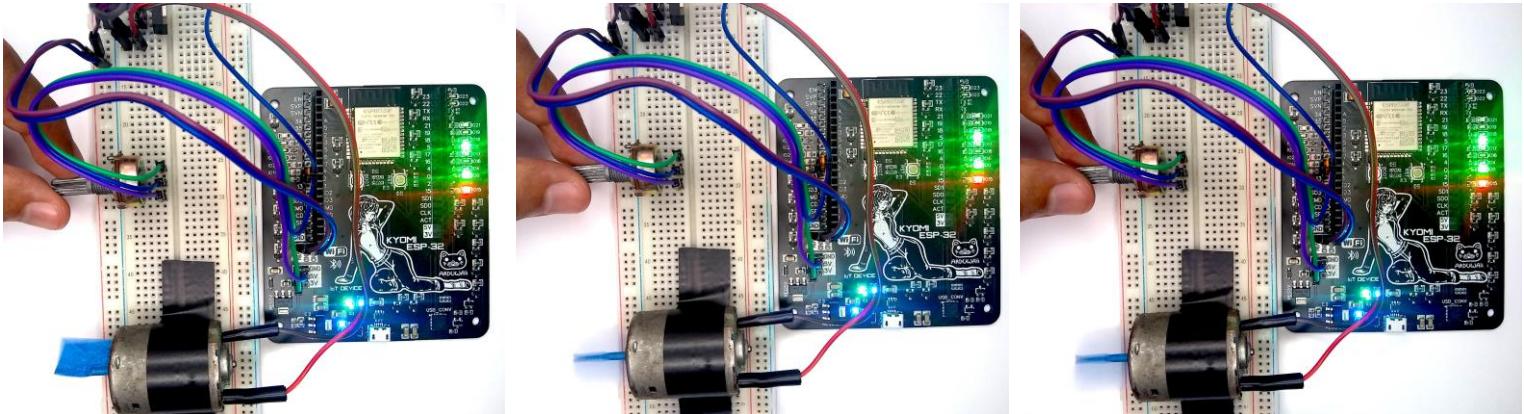
```
const int GPIO_MOTOR=14, GPIO_CONTROL=12, FRECUENCIA=20000, RESOLUCION=12;

void setup()
{
    LedcSetup(0, FRECUENCIA, RESOLUCION);
    LedcAttachPin(GPIO_MOTOR, 0);
    pinMode(GPIO_MOTOR, OUTPUT);
    pinMode(GPIO_CONTROL, INPUT_PULLDOWN);
    digitalWrite(GPIO_MOTOR, LOW);
}
```



```
void Loop()
{
    LedcWrite(0,analogRead(GPIO_CONTROL));
}
```

Prueba el programa. Cuando lo hayas cargado, podrás desconectar tu tarjeta del puerto USB, ya que será alimentada con el pin VIN:



Este circuito de potencia lo puedes usar para controlar otro tipo de cargas (en lugar de un motor), por ejemplo, un LED de alta potencia, una celda Peltier, una resistencia, etc. Pero este circuito está limitado por la potencia que puede tolerar el MOSFET y por la fuente de voltaje.

## Fuentes de alimentación para tu tarjeta

La opción más común para alimentar al ESP32 y darle portabilidad, es usar un PowerBank o batería USB. Con tu tarjeta KYOMI tienes la posibilidad de usar fuentes de voltaje de 6V a 12V, que pueden ser baterías, fuentes con transformador o fuentes conmutadas. Veamos algunas opciones para alimentar tu tarjeta sin depender del puerto USB.

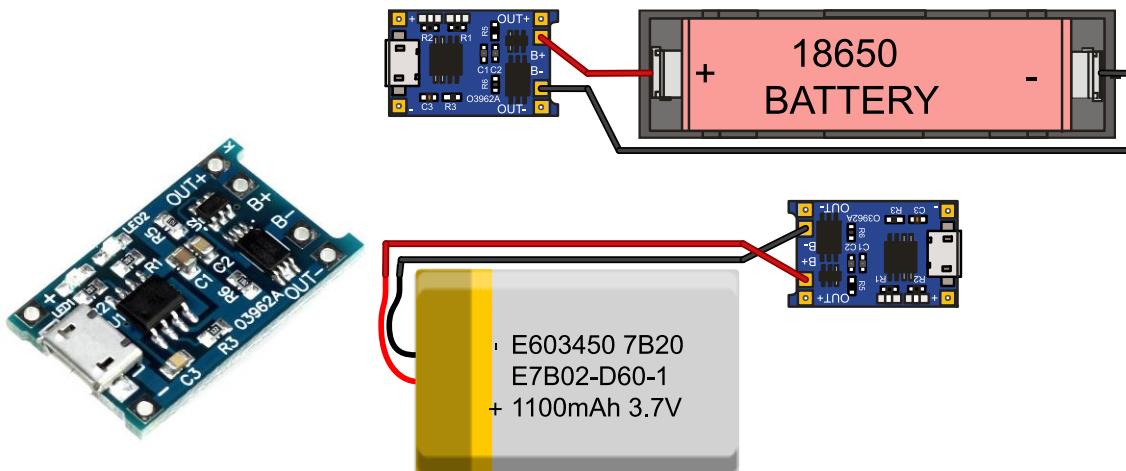
### Baterías de litio

Las baterías de litio las puedes encontrar en distintos factores de forma, tamaños, voltajes y composición química. Las baterías de litio comunes entregan 3.7V nominales y su química puede ser de polímero de litio (LiPo) o de iones de litio (ION). Las baterías de polímero de litio tienen factores de forma distintos según su capacidad, pero las de iones de litio tienen factores de forma más estándar, como las de tipo 18650.



Batería de iones de litio tipo 18650 (izq.) y batería de polímero de litio de 1.2 amperios/hora (der.), ambas entregan 3.7V.

Digamos que tienes una batería 18650 o una LiPo. Generalmente este tipo de baterías no cuentan con un sistema de protección, así que necesitarás añadir uno para que no recortes su vida útil. Con las baterías de litio debes tener en cuenta 4 medidas de protección: cargarlas a corriente constante, protegerlas contra cortocircuito, protegerlas contra sobrecarga y contra sobredescarga. Un módulo de protección muy popular es el TP4056, que permite cargar baterías de litio de 3.7V mediante USB y además cuenta con todas las protecciones mencionadas:



Módulo TP4056 y cómo conectarlo.

Una vez que nuestra batería cuente con protección, tendremos que elevar su voltaje, ya que mínimo necesitamos 6V para alimentar nuestra tarjeta KYOMI por el pin VIN. La opción más sencilla es utilizar un módulo elevador de voltaje (conocidos como Boost o Step-Up). Existen muchos tipos de módulos Boost, que varían de acuerdo a la potencia que pueden entregar y a su voltaje máximo de entrada y salida.

El ESP32 no consume mucha energía, así que puedes usar módulos Boost de baja potencia como el XL6009 o el MT3608. El módulo XL6009 trabaja con 3V a 32V de entrada y puede ajustarse para entregar hasta 35V en

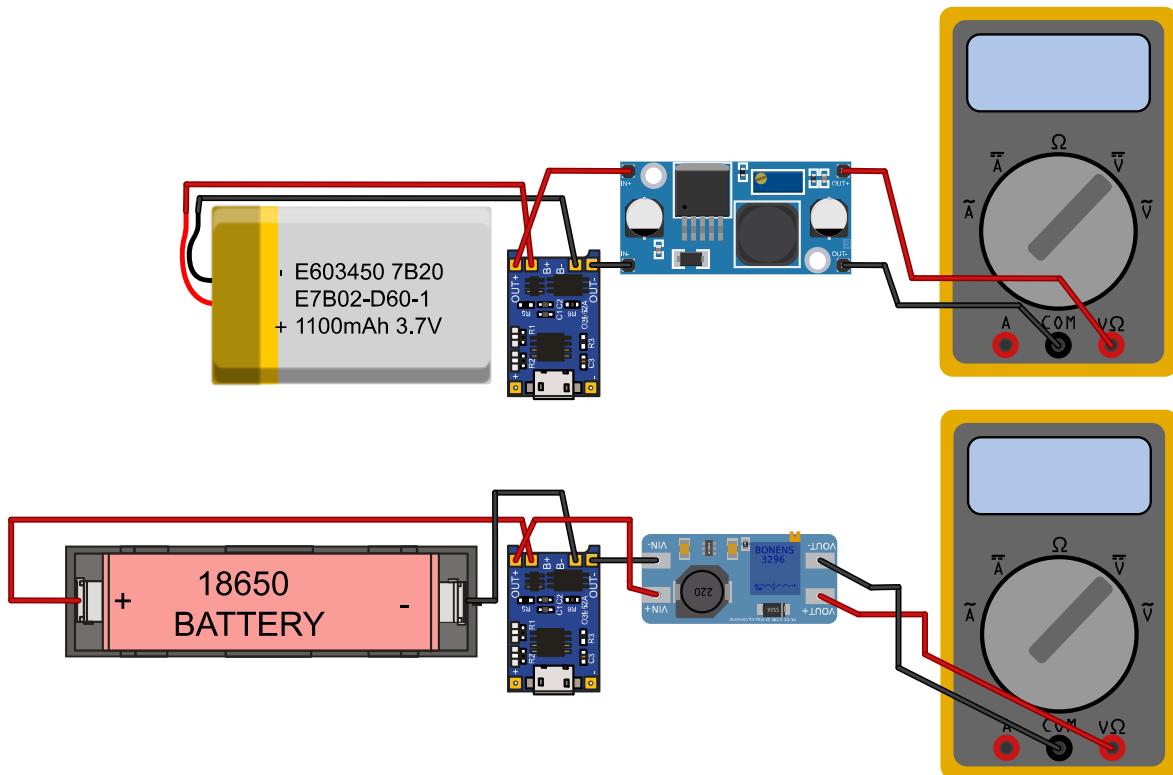


su salida, mientras que el MT3608 puede trabajar con 2V hasta 24V de entrada para ajustar su salida hasta 28V:



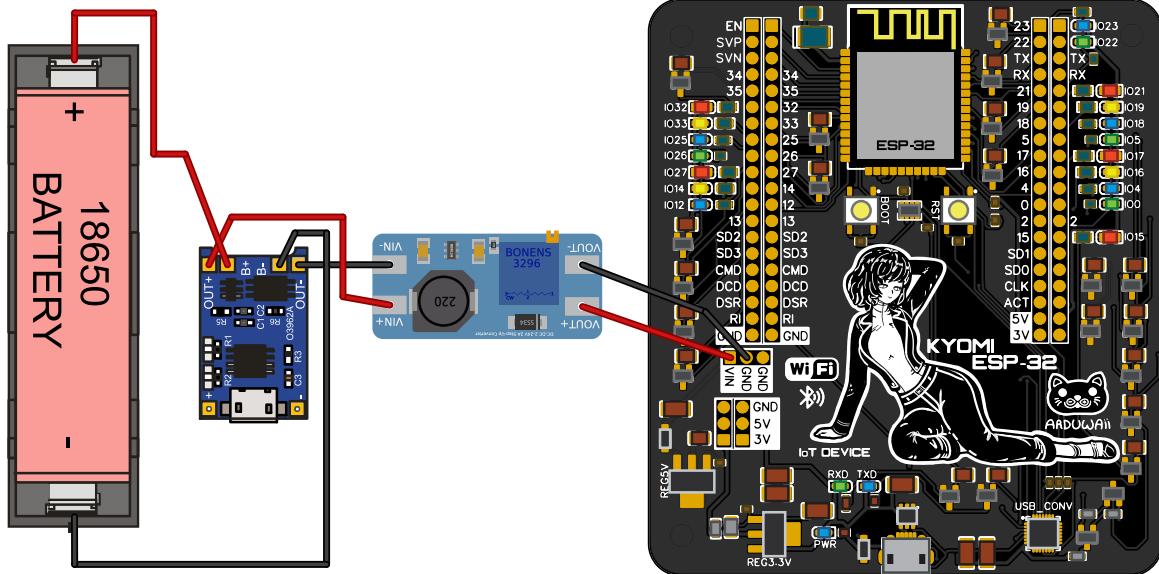
Módulo elevador de voltaje MT3608 (izq.) y XL6009 (der.).

Antes de conectar el módulo Boost a tu tarjeta, ajusta su voltaje de salida girando el potenciómetro de precisión (con un destornillador pequeño de punta plana) y usa un multímetro para medir el voltaje ajustado:





Cuando vayas a conectar la salida del módulo Boost a tu tarjeta (al pin VIN y a un pin de GND), revisa la polaridad de las conexiones para no dañarla o dañar al módulo:



El pin VIN tiene un diodo de protección contra polaridad inversa, pero se puede dañar si dejas invertida la conexión durante mucho tiempo.

Con baterías de litio, el ESP32 puede tener una autonomía de varias horas, considerando que las baterías 18650 tienen una capacidad estándar de 2200mAh (miliamperios/hora) y las baterías LiPo medianas tienen una capacidad de al menos 1000mAh.

## Baterías comunes

También tenemos la posibilidad de utilizar baterías comunes con nuestra tarjeta, como las baterías de 9V, pilas AA, AAA, C o D. Como sabes, estas pilas son desechables, y no necesitan un sistema de protección, ya que prácticamente no se pueden recargar. Lo que debes considerar al utilizar pilas AA, AAA, C y D es que debes colocarlas en serie para obtener los 6V mínimos. Estas pilas tienen un voltaje nominal de 1.2V a 1.5V, así que necesitaríamos al menos 5 pilas para sumar 6V, pero se recomienda usar 6 pilas para tener cierto margen de voltaje extra.

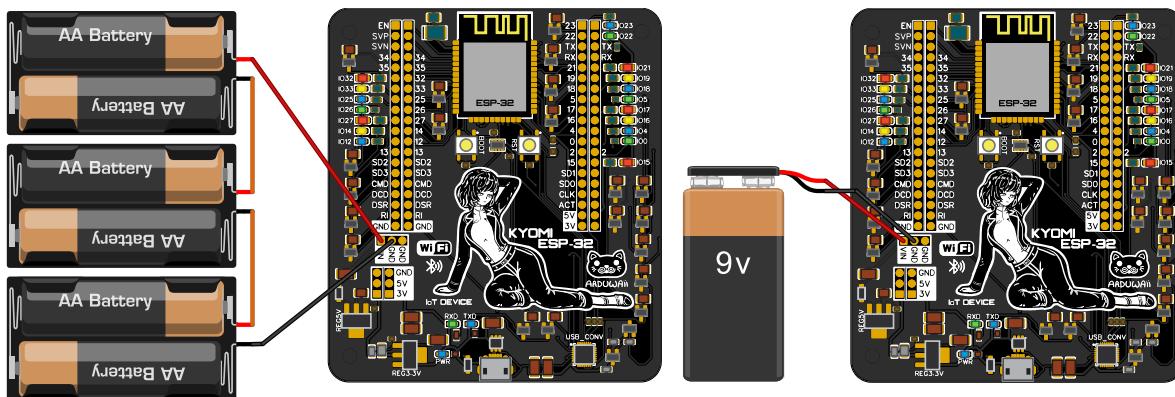
La diferencia entre las pilas AA, AAA, C y D (además de su tamaño) está su capacidad. Las pilas AA normalmente tienen una capacidad de 400mAh a 900mAh (según su composición química), pero existen pilas especiales con capacidades mayores a los 2000mAh. Las pilas AAA tienen capacidades que van de los 100mAh hasta los 1200mAh, de acuerdo a su composición química. Las pilas C pueden ir de los 2000mAh hasta los 8000mAh de capacidad, mientras que las pilas D van de los 2000mAh



hasta 18000 mAh. Las baterías de 9V tienen capacidades entre 200mAh hasta 1100mAh (esto también depende de su composición química).

Al conectar pilas en serie, se suman sus voltajes, pero no su capacidad. Para aumentar la capacidad, debes conectar las pilas en paralelo. Por ejemplo, consideremos que la capacidad de una pila AA es de 400mAh, si conectamos dos en serie, obtendríamos una batería de 3V a 400mAh, pero, si conectamos en serie a dos pares de pilas en paralelo, obtendríamos una batería de 3V a 800mAh.

Para alimentar nuestra tarjeta con pilas, lo mejor es usar portapilas conectados en serie o directamente optar por una batería de 9V:



Las pilas no son una opción muy conveniente, debido a la poca autonomía que obtendríamos y a la cantidad de pilas necesarias para alcanzar el voltaje mínimo (además de ser desechables). Tal vez la mejor opción serían las pilas C o D, ya que tienen capacidades mayores, pero son voluminosas y pesadas y aún necesitaríamos 6 pilas como mínimo. Lo mejor es usarlas para hacer pruebas rápidas de nuestros circuitos.

## Baterías de plomo/ácido y gel

La baterías de plomo/ácido y las baterías de gel son las que se utilizan en aplicaciones de alta potencia, como motocicletas, reguladores UPS, automóviles, lanchas eléctricas, etc.:

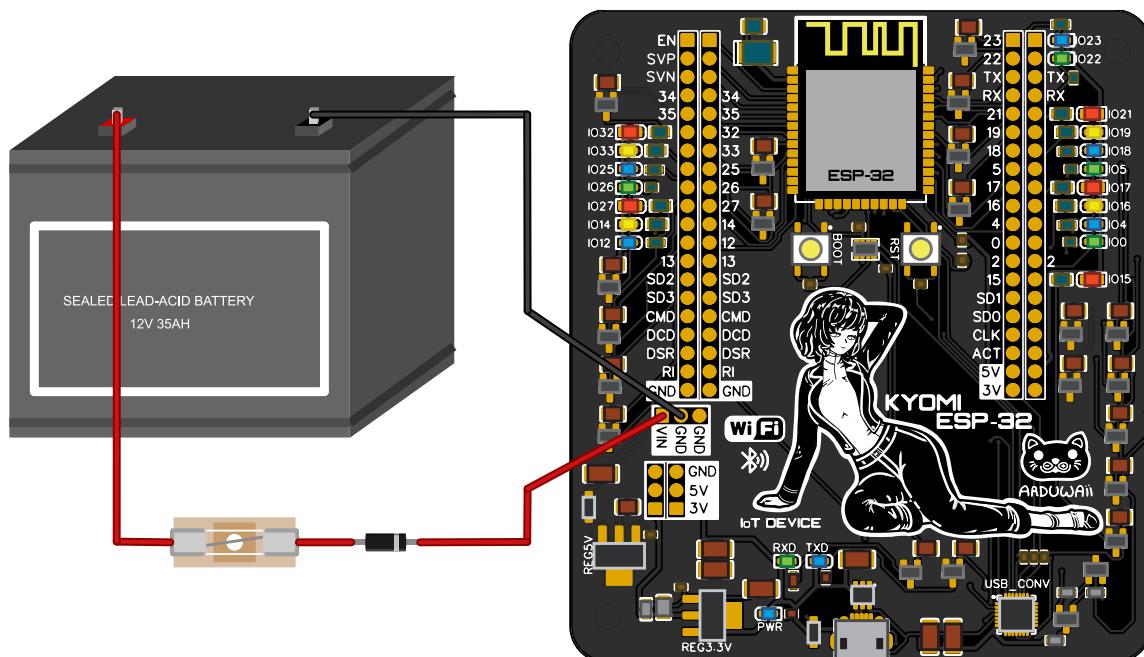


Baterías de plomo/ácido de distintas capacidades y voltajes.



Comúnmente, estas baterías tienen voltajes nominales de 6V, 12V y 24V, mientras que sus capacidades van de los 4Ah (amperios/hora) hasta los 12Ah, en las baterías pequeñas y medianas. Con estas capacidades, podemos obtener una autonomía de varios días o incluso semanas.

La opción más conveniente son las baterías de 12V, pero la capacidad elegida ya dependerá de tu aplicación. Este tipo de baterías no requieren de muchas protecciones, pero si es muy recomendable colocar un fusible en serie con el polo positivo, ya que pueden entregar corrientes muy altas que podrían dañar a la propia batería en caso de cortocircuitarla por accidente (también se recomienda colocar un diodo de protección):



Otro cuidado que debes tener en cuenta, es que la batería no se descargue por debajo de los 10V, porque en este punto se considera que ya está totalmente descargada. Para recargarla necesitarás un cargador externo.

Este tipo de baterías son muy útiles para darle cierta portabilidad a tus circuitos de potencia y a tu micro, pero ten en cuenta que son baterías grandes y pesadas.

## Fuentes conmutadas

Las fuentes conmutadas son las que podemos encontrar en cargadores o ‘eliminadores’, e incluso son las que utilizan la mayoría de los aparatos electrónicos modernos. Convierten la corriente alterna de la red eléctrica en corriente directa de bajo voltaje. Podemos encontrarlas en distintos tamaños, voltajes y capacidades de corriente, pero los voltajes



estándar más recomendados van de los 7V hasta los 12V, mientras que la capacidad de corriente elegida dependerá de tu aplicación o circuito:

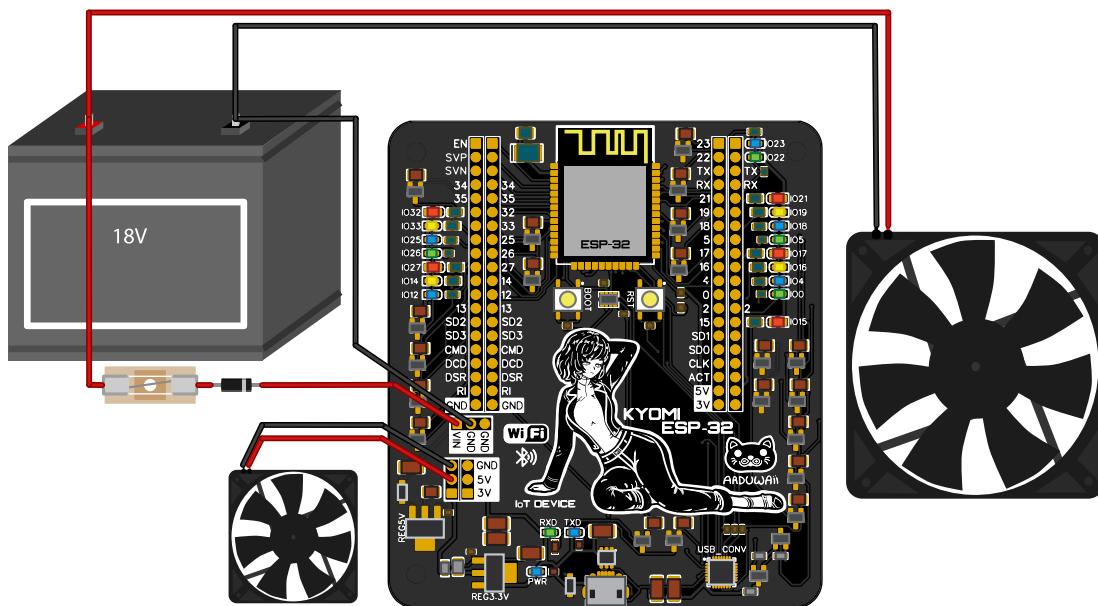


*Algunos ejemplos de fuentes conmutadas.*

Si alimentas tu tarjeta por el pin VIN, una fuente de 7V será suficiente, pero también puedes alimentarla por el puerto micro-USB con un cargador para celular. Las fuentes conmutadas ya cuentan con varias protecciones, así que solo tendrás que revisar la polaridad cuando las conectes al pin VIN.

## Cómo usar fuentes de más de 12V

Como se mencionó en el primer capítulo, el regulador de 5V ('REG5V' en tu tarjeta) usado para VIN técnicamente soporta hasta 20V de entrada, pero no se recomienda superar los 12V, ya que en este punto empieza a subir su temperatura. Puedes aplicar hasta 18V por VIN siempre y cuando tengas alguna manera de enfriar el regulador de 5V. Por ejemplo, si tienes una fuente de 18V, podrías conectar un ventilador de 5V pequeño a una de las salidas de voltaje de la tarjeta y colocarlo apuntando hacia el regulador, o bien, podrías usar un ventilador de 18V grande, conectado directamente a la fuente de voltaje:





## Capítulo 6: La última y nos vamos

Al añadir un sistema de enfriamiento para el regulador, también lo puedes aprovechar para el propio módulo ESP32 y así alargar su vida útil. Otra opción sería colocarle un disipador de calor pequeño al regulador, pero sólo deberá hacer contacto en su superficie plástica, para no cortocircuitar alguna parte metálica de otro componente cercano.



## ¡No dañes a tu ESP32!, toma en cuenta estos consejos

1. Siempre revisa la polaridad de tus conexiones cuando alimentes a tu tarjeta por VIN, porque el diodo de protección no es totalmente infalible.
2. No expongas a tu tarjeta al sol o al calor por mucho tiempo.
3. No expongas a tu tarjeta a los ambientes húmedos o al agua.
4. Usa un divisor de voltaje con los GPIO's cuando midas señales mayores a 3.3V
5. Añade un sistema de enfriamiento a tu tarjeta cuando vaya a estar funcionando por periodos de tiempo largos (días o semanas), especialmente cuando uses las funciones Wifi.
6. No trabajes sobre superficies metálicas o conductoras, los GPIO's están expuestos por debajo y podrías cortocircuitarlos.
7. Mantén aislada a tu tarjeta de los circuitos de alto voltaje, siempre usa relevadores u optoacopladores para controlarlos.
8. Los GPIO's no pueden alimentar cargas grandes, usa circuitos de potencia controlables.
9. Siempre activa las resistencias PULL-DOWN o PULL-UP internas (o usa unas externas) cuando configures los GPIO's como entradas digitales.
10. Las salidas de voltaje integradas sólo son para alimentar sensores, periféricos, pantallas o circuitos pequeños. Si les exiges demasiada potencia, puedes dañar los reguladores de voltaje.





## ¿Qué más aprender sobre el ESP32?

**MQTT:** un protocolo de comunicación ligero y rápido, diseñado para dispositivos IoT y sistemas conectados a internet.

**RTOS:** expresa al máximo a tu ESP32 con un sistema operativo en tiempo real. Ejecuta muchas tareas simultáneas en ambos núcleos.

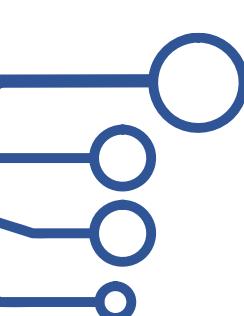
**I2S:** una interfaz de audio digital estéreo integrada en tu tarjeta. Aprovechala para tus proyectos con audio.

**LittleFS:** una versión más avanzada de SPIFFS, con soporte para subdirectorios y con velocidades de lectura/escritura más rápidas.

**LoRa:** una tecnología de comunicación inalámbrica similar a Wifi, enfocada a dispositivos de bajo consumo energético y para cubrir grandes distancias. No viene integrada en tu ESP32 (necesitarás un módulo adicional), pero la comunicación Wifi y Bluetooth complementan a LoRa para ampliar sus posibilidades.

**TWAI:** un protocolo de comunicación industrial muy utilizado. Reemplaza a la comunicación CAN clásica en muchas aplicaciones.

Entre muchas otras cosas más...



# Referencias

Kolban, N. (2018). *Kolban's Book on ESP32* (1ra ed.).

Santos, R. and Santos, S., 2021. *Learn ESP32 with Arduino IDE* (1ra ed.).

Espressif Systems. (2021). *ESP32 Series Datasheet* [Ebook], disponible en:

[https://www.espressif.com/sites/default/files/documentation/esp32\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf)

Espressif Systems. (2020). *ESP32 Technical Reference Manual* [Ebook]. disponible en:

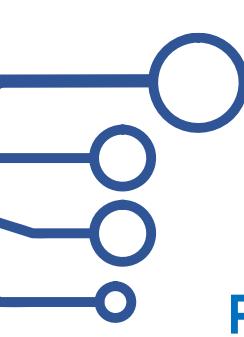
[https://www.espressif.com/sites/default/files/documentation/esp32\\_technical\\_reference\\_manual\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf)

Espressif Systems. (2020). *ESP32WR00M32D & ESP32WR00M32U Datasheet* [Ebook].:

[https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d\\_esp32-wroom-32u\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf)

Docs.espressif.com. 2021. *ESP-IDF Programming Guide*. [En Línea] disponible en:

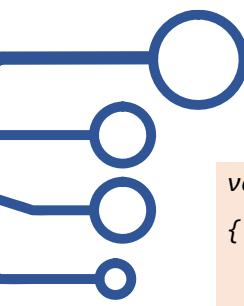
<https://docs.espressif.com/projects/espidf/en/latest/esp32/index.html>



# Anexos

## Programa DEMO

```
const int PINS_IQZ[8]={32,33,25,26,27,14,12};  
const int PINS_DER[11]={23,22,21,19,18,5,17,16,4,0,15};  
int n,m=0,k=0;  
const int FREC = 5000;  
const int RES = 8;  
  
void setup()  
{  
    for(n=0;n<=6;n++)  
    {  
        pinMode(PINS_IQZ[n],OUTPUT);  
        digitalWrite(PINS_IQZ[n],LOW);  
    }  
  
    for(n=0;n<=10;n++)  
    {  
        pinMode(PINS_DER[n],OUTPUT);  
        digitalWrite(PINS_DER[n],LOW);  
    }  
    esp_sleep_enable_timer_wakeup(10);  
}  
  
void loop()  
{  
    SECUENCIA_INDIVIDUALES();  
    SECUENCIA_TODOS();  
    IMPARES_PARES();  
    PWM_IQZ();  
    PWM_DER();  
    esp_deep_sleep_start();  
}
```



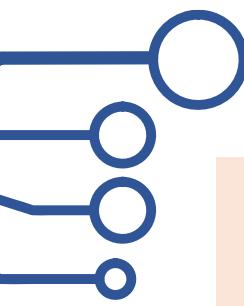
```
void SECUENCIA_INDIVIDUALES()
{
    for(n=0;n<=6;n++)
    {
        digitalWrite(PINS_IQZ[n],HIGH);
        delay(150);
        digitalWrite(PINS_IQZ[n],LOW);
        delay(150);
    }

    for(n=0;n<=10;n++)
    {
        digitalWrite(PINS_DER[n],HIGH);
        delay(150);
        digitalWrite(PINS_DER[n],LOW);
        delay(150);
    }
}

void SECUENCIA_TODOS()
{
    for(n=0;n<=6;n++)
    {
        digitalWrite(PINS_IQZ[n],HIGH);
        delay(100);
    }

    for(n=0;n<=10;n++)
    {
        digitalWrite(PINS_DER[n],HIGH);
        delay(100);
    }
}

delay(1000);
```



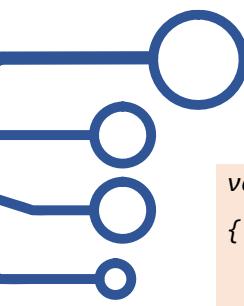
```
for(n=6;n>=0;n--)
{
    digitalWrite(PINS_IQZ[n],LOW);
    delay(100);
}

for(n=10;n>=0;n--)
{
    digitalWrite(PINS_DER[n],LOW);
    delay(100);
}

void IMPARES_PARES()
{
    while( m<=10)
    {
        for(n=1;n<=6;n=n+2){digitalWrite(PINS_IQZ[n],HIGH);}
        for(n=0;n<=6;n=n+2){digitalWrite(PINS_IQZ[n],LOW);}
        delay(330);
        for(n=0;n<=6;n=n+2){digitalWrite(PINS_IQZ[n],HIGH);}
        for(n=1;n<=6;n=n+2){digitalWrite(PINS_IQZ[n],LOW);}
        delay(330);
        m++;
    }
    for(n=0;n<=6;n++){digitalWrite(PINS_IQZ[n],LOW);}

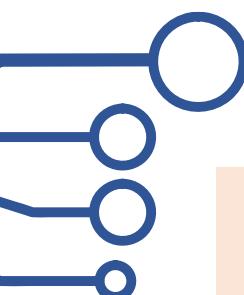
    while(k<=10)
    {

        for(n=0;n<=10;n++){digitalWrite(PINS_DER[n],HIGH);delay(30);digitalWrite(PINS_DER[n],LOW);}
        for(n=10;n>=0;n--)
        {digitalWrite(PINS_DER[n],HIGH);delay(30);digitalWrite(PINS_DER[n],LOW);}
        k++;
    }
}
```



```
void PWM_IQZ()
{
    for(n=0;n<=6;n++)
    {
        LedcSetup(n,FREC,RES);
        LedcAttachPin(PINS_IQZ[n],n);
    }
    for(n=0;n<=6;n++)
    {
        for(int i=0;i<=255;i++)
        {
            LedcWrite(n,i);
            deLay(3);
        }
        for(int i=255;i>=0;i--)
        {
            LedcWrite(n,i);
            delay(3);
        }
    }
}

void PWM_DER()
{
    for(n=0;n<=10;n++)
    {
        LedcSetup(12,FREC,RES);
        LedcAttachPin(PINS_DER[n],12);
    }
    for(n=0;n<3;n++)
    {
        for(int i=0;i<=255;i++)
        {
            LedcWrite(12,i);
            delay(3);
        }
    }
}
```



```
for(int i=255;i>=0;i--)
{
    LedcWrite(12,i);
    delay(3);
}
}
```



**NEZATRONICS MX**

*Esta obra es de libre distribución. Queda prohibida la copia total o parcial con fines de lucro.*