



Universidad
Zaragoza



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Prácticas

Cuaderno de Prácticas.

Autor

García Usón, Alba

Redes de Sensores Electrónicos
Escuela de Ingeniería y Arquitectura de Zaragoza
Año académico: 2021 – 2022

Contenido

1. ESP32 como plataforma de prototipado rápido de sensores.....	3
1.1. Configuración del ADC	3
1.2. Configuración del ADC con Timer.	4
1.3. Configuración del PWM.	6
1.4. Configuración de comandos por puerto serie.	6
1.5. Configuración del sensor inercial por I2C.	8
2. Diseño basado en sistema operativo de tiempo real (RTOS).	11
2.1. Implementación con RTOS de la lectura de un sensor inercial.	11
3. Comunicación, gestión y representación de datos de sensores con Python.	14
3.1. Comprobación recepción de datos.	14
3.2. Almacenar los datos en fichero .txt	14
3.3. Procesar y graficar en tiempo real.	15
4. Comunicaciones WIFI desde microcontrolador. Stack IP y subida de datos a la nube.....	18
4.1. Socket TCP para mandar la hora mediante start y stop	18
4.2. Servidor WEB que muestre la hora y sea capaz de resetearla	20
4.3. Subir fichero JSON a un servidor FTP	24
4.4. Subir datos usando MQTT.....	26
5. Comunicaciones BLE y Bluetooth.....	29
5.1. Escanea beacons y reporta en formato JSON-SENML.	29
5.2. Advertising iBeacon	32
6. Comunicaciones Lora y LoraWAN	34
6.1. Ping-pong Lora	34
6.2. Envío datos a TTN.....	37

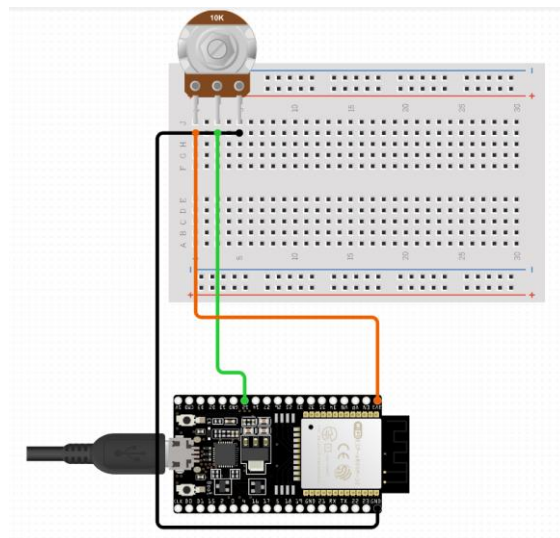
1. ESP32 como plataforma de prototipado rápido de sensores.

En esta sesión de la asignatura se plantea como objetivo familiarizarse con el entorno del ESP32. De esta forma, aprenderemos a manejarnos con diferentes tipos de periféricos como pueden ser entradas y salidas digitales, ADC, UART para visualización por puerto serie y timers.

Los puntos clave para el desarrollo de esta práctica son la utilización de timers para generar una interrupción cada 10 segundos, momento en el cual se deberá leer el ADC y mostrarlo por pantalla; utilizar la UART para el manejo de la lectura del ADC y del PWM; y familiarizarse con un sensor inercial y el protocolo de comunicación que utilice, ya sea SPI o I2C y mande sus datos vía UART.

1.1. Configuración del ADC

Antes de aprender a utilizar los timers, es necesario conocer el funcionamiento del conversor analógico-digital que se encuentra integrado en el kit de desarrollo del ESP32. Se va a utilizar un potenciómetro conectado al pin 34, correspondiente al ADC, para comprobar su funcionamiento.



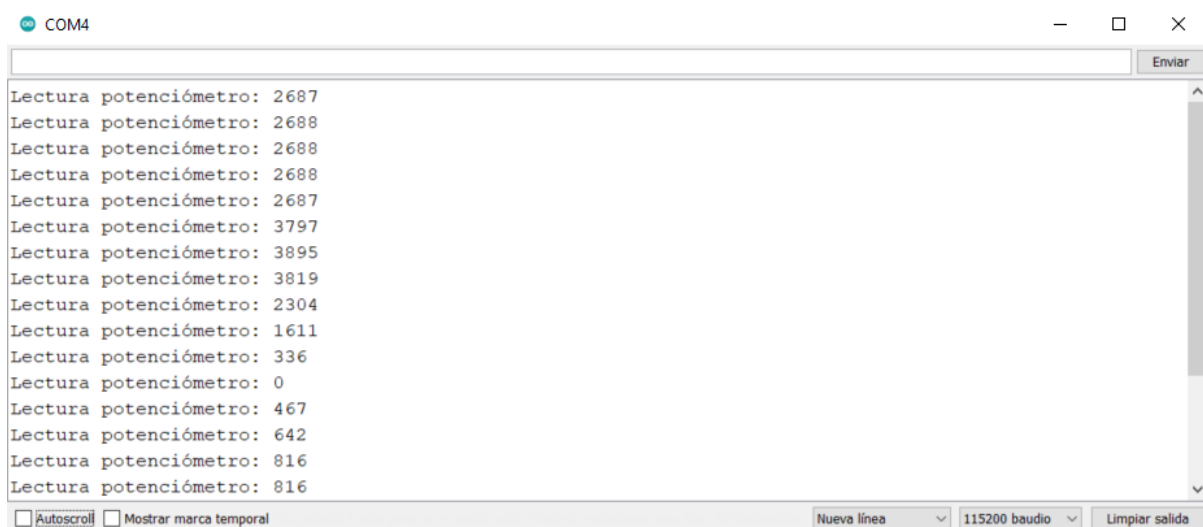
Una vez conectado el potenciómetro, prepararemos un programa que lea el valor del ADC y lo muestre por consola cada segundo, utilizando en este caso un delay. Para ello primero creamos una función para leer el potenciómetro:

```
int leePot() {  
    int resul = analogRead(pinPot);  
    return resul;  
}
```

Desde el loop, llamaremos a esta función e imprimiremos el resultado cada segundo:

```
void loop() {  
    valADC = leePot();  
    Serial.print("Lectura potenciómetro: ");  
    Serial.println(valADC);  
    delay(1000);  
}
```

El resultado que encontramos es que, cada segundo, se imprime por pantalla el valor del ADC, el cual varía en función de cómo se encuentre el potenciómetro.



1.2. Configuración del ADC con Timer.

La interrupción realizada en el apartado anterior se consigue a través de un delay. Esto supone que el kit de desarrollo no realiza ningún otro tipo de función durante el tiempo establecido. Para evitar que estas interrupciones afecten al resto del programa, se utilizan los timer, que permiten temporizar las diferentes funciones sin afectar a la ejecución del resto del programa.

Mantendremos la función `leePot()` del apartado anterior y simplemente cambiaremos la forma de llamarla. Para ello tenemos que configurar primero las variables necesarias para utilizar el timer:

```
volatile int interruptCounter;  
int totalInterruptCounter;  
hw_timer_t * timer = NULL;  
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
```

En este caso tenemos dos variables de tipo entero:

- `interruptCounter`: contador utilizado por si alguna de las interrupciones toma más tiempo del esperado y se producen más interrupciones. Además, es de tipo `volatile` porque se comparte entre la rutina de interrupción y el bucle principal.
- `totalInterruptCounter`: contador que cuenta todas las interrupciones que se han ido realizando.

Tenemos también un puntero timer de tipo `hw_timer_t` para configurar el temporizador, y de un multiplexor que nos permite mantener la sincronización entre el bucle principal y la rutina de interrupción al modificar la variable `interruptCounter`.

```
void IRAM_ATTR onTimer() {  
    portENTER_CRITICAL_ISR(&timerMux);  
    interruptCounter++;  
    portEXIT_CRITICAL_ISR(&timerMux);  
}
```

Además, se genera la función de la rutina de interrupción que debe tener atributo `IRAM_ATTR` para colocar el código en la RAM. Esta función incrementa el código de interrupciones cada vez que ocurre una interrupción.

```

void setup() {
  Serial.begin(115200);
  timer = timerBegin(0, 80, true);
  timerAttachInterrupt(timer, &onTimer, true);
  timerAlarmWrite(timer, 10000000, true);
  timerAlarmEnable(timer);
}

```

Por otra parte, en el setup inicializamos el puerto serie y el temporizador, devolviendo un puntero a una estructura de tipo `kw_timer_t`. Esta función recibe como argumentos el número de temporizador que queremos usar, un valor multiplicador, en este caso 80 porque la frecuencia de trabajo del ESP32 es de 80 MHz y, de esta forma, podemos trabajar en microsegundos y, por último, un indicador de si el contador cuenta hacia arriba (verdadero) o hacia abajo (falso).

Antes de habilitar el temporizador, se vincula a una función de manejo que se ejecuta cuando se genera la interrupción (`timerAttachInterrupt`) que tiene como argumentos el puntero del temporizador, la dirección de la función `onTimer()` y el tipo de interrupción, de flanco (verdadero) o de nivel (falso).

Lo siguiente es el `timerAlarmWrite`, que establece el número de ticks del procesador (1MHz son 1 millón de ticks por segundo) que se utiliza para las interrupciones, por ello utilizamos el valor de 10000000, y el `true` establece que el contador se reinicia y, por tanto, es periódico.

Por último, habilitamos con el `timerAlarmEnable` la interrupción creada.

Una vez configurado el timer, lo utilizamos en el loop para llamar a la función `leePot()` cada 10 segundos:

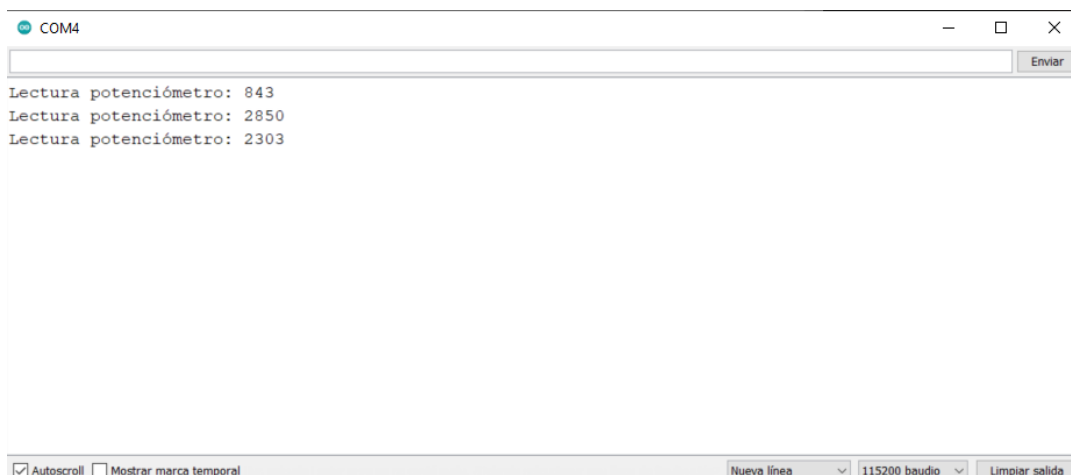
```

void loop() {
  if (interruptCounter > 0) {
    portENTER_CRITICAL(&timerMux);
    interruptCounter--;
    portEXIT_CRITICAL(&timerMux);
    //Serial.println(totalInterruptCounter);
    totalInterruptCounter++;
    valADC = leePot();
    Serial.print("Lectura potenciómetro: ");
    Serial.println(valADC);
  }
}

```

De esta forma, se realiza una interrupción cada 10 segundos, leemos el valor del potenciómetro y lo imprimimos por pantalla.

El resultado es que cada 10 segundos se imprime por pantalla el valor del ADC, que sigue dependiendo del potenciómetro.



1.3. Configuración del PWM.

En este apartado aprenderemos a configurar las salidas PWM del ESP32. Tendremos que configurar un PWM con una frecuencia de 5kHz y un ciclo de trabajo proporcional a la lectura del ADC.

El pin seleccionado en nuestro caso será el 12, configurándolo de la siguiente forma:

```
#define LED_GPIO 12 |
#define PWM1_Ch 0
#define PWM1_Res 12
#define PWM1_Freq 5000
```

De esta forma, el canal del PWM será el 0, la resolución será de 12 bits, la misma que el ADC, y la frecuencia serán 5000 Hz, 5kHz. El LED_GPIO será el pin de salida del PWM.

En el setup, añadimos la configuración de forma que el canal PWM se saque por el pin establecido y que el PWM se configure con los valores mencionados anteriormente:

```
ledcAttachPin(LED_GPIO, PWM1_Ch);
ledcSetup(PWM1_Ch, PWM1_Freq, PWM1_Res);
```

Para establecer el ciclo de trabajo en función del ADC se utiliza la siguiente función:

```
ledcWrite(PWM1_Ch, valADC);
```

Con esta función, le establecemos al canal del PWM el ciclo de trabajo leído en el ADC.

El resultado lo podemos ver midiendo con un osciloscopio las salidas 12 y 34, el del PWM y el del ADC respectivamente.

1.4. Configuración de comandos por puerto serie.

En este punto tendremos que configurar el programa de forma que, según la petición realizada por pantalla, nuestro controlador realice diferentes acciones, siendo estas:

- a. ADC: Envíe la lectura del ADC actual
- b. ADC(x): envíe la lectura del ADC cada x segundos. Si x=0, deja de mandar datos
- c. PWM(x): comanda el duty cycle del módulo PWM con números del 0 al 9.

Para ello tenemos que utilizar las diferentes funciones de los String y del Serial.

En el loop tendremos que poner un if (Serial.available() > 0 para poder introducir texto en el puerto serie, y dentro de este, las diferentes lecturas y comparaciones de dichas lecturas para acceder a las diferentes funciones:

```

void loop() {
  funcionEscritura();
  if (Serial.available() > 0) {
    escritura = Serial.readStringUntil('\n');
    Serial.println(escritura);
    if (escritura == "actADC") {
      ValorEscritura = 0;
      Serial.print("Envio de la lectura del ADC actual: ");
      valADC = leePot();
      Serial.println(valADC);
    }
    else if (escritura.substring(0, 4) == "ADC(" && escritura.substring(5, 6) == ")") {
      Serial.print("Envio de lectura cada ");
      Serial.print(escritura.substring(4, 5));
      Serial.println(" segundos");
      ValorEscritura = 1;
      totalInterruptCounter = 0;
      interruptCounter = 0;
    }
    else if (escritura.substring(0, 4) == "PWM(" && escritura.substring(5, 6) == ")") {
      Serial.print("Duty cycle: ");
      Serial.println(escritura.substring(4, 5));
      ValorEscritura = 2;
    }
    else
      ValorEscritura = 0;
  }
}

```

En este programa entraremos en diferentes secciones en función del texto que recibe el puerto serie. Además, tenemos la función `funcionEscritura()` que se ejecuta al principio del loop en función de los valores recibidos. En esta función podemos ejecutar las acciones requeridas para cada caso:

```

void funcionEscritura() {
  if (ValorEscritura == 1) {
    if (escritura.substring(4, 5).toInt() != 0) {
      ADCx(escritura.substring(4, 5).toInt());
    }
  }
  if (ValorEscritura == 2) {
    PWMx(escritura.substring(4, 5).toInt());
  }
}

```

Se llama a las funciones `ADCx` y `PWMx` teniendo el número `x` recibido por el puerto serie como input en la función.

Las funciones serían las siguientes:

```

void ADCx(int t) {
  if (interruptCounter > 0) {
    portENTER_CRITICAL(&timerMux);
    interruptCounter--;
    portEXIT_CRITICAL(&timerMux);
    //Serial.println(totalInterruptCounter);
    totalInterruptCounter++;
    if (totalInterruptCounter == t) {
      valADC = leePot();
      Serial.println(valADC);
      totalInterruptCounter = 0;
    }
  }
}

void PWMx(int x) {
  int y = map(x, 0, 9, 0, 4095);
  ledcWrite(PWM1_Ch, y);
}

```

La función `ADCx` utiliza el contador `totalInterruptCounter`, en este caso, se producen interrupciones cada segundo (configurado en el setup) y se acumulan en `totalInterruptCounter`, cuando el número de interrupciones llega al valor asignado por el puerto serie, `totalInterruptCounter` se resetea y se lee e imprime el valor del potenciómetro por el puerto serie.

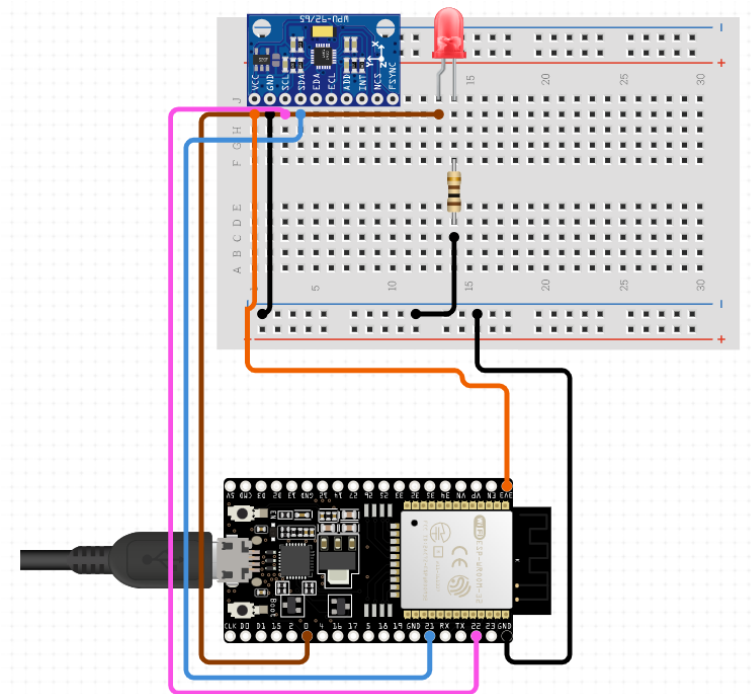
La función PWMx simplemente hace un mapeo para que los valores del 0 al 9 se transformen según la resolución (12 bits) del PWM y se escriban como ciclo de trabajo en el PWM.

El resultado es el siguiente:

```
COM4
PWM(4)
Duty cycle: 4
actADC
Envio de la lectura del ADC actual: 2318
PWM(9)
Duty cycle: 9
ADC(4)
Envio de lectura cada 4 segundos
2374
1870
1263
487
487
ADC(0)
Envio de lectura cada 0 segundos
```

1.5. Configuración del sensor inercial por I2C.

Finalmente, en esta sección tendremos que configurar un sensor inercial conectado por I2C de forma que muestree la aceleración cada 100 ms y mande los datos por el puerto serie cada segundo, activando un LED durante 200 ms.



Para ello utilizaremos una librería que permita controlar el acelerómetro MPU9250, configurando el sensor según dicha librería, además del Wire para poder configurar la comunicación I2C:


```
#include <MPU9250_asukiaaa.h>
#include <Wire.h>

#define SDA_PIN 21
#define SCL_PIN 22

MPU9250_asukiaaa sensor;
```

Para configurar el sensor y el Wire utilizamos los siguientes comandos de la librería:

```
Wire.begin(SDA_PIN, SCL_PIN);

sensor.setWire(&Wire);
sensor.beginAccel();
```

En este caso sólo iniciaremos la aceleración del sensor ya que es lo único que vamos a utilizar.

Por otra parte, como tendremos que temporizar tres funciones distintas (leer, escribir en pantalla y apagar el led después de 200 ms), añadiremos dos contadores nuevos además del totalInterruptCounter, contadorSerial y contadorLed:

```
volatile int interruptCounter;
int totalInterruptCounter;
int contadorSerial;
int contadorLed;
hw_timer_t * timer = NULL;
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
```

Para esto creamos una función que incluya todo llamada leeSensor a la que llamaremos desde el loop:

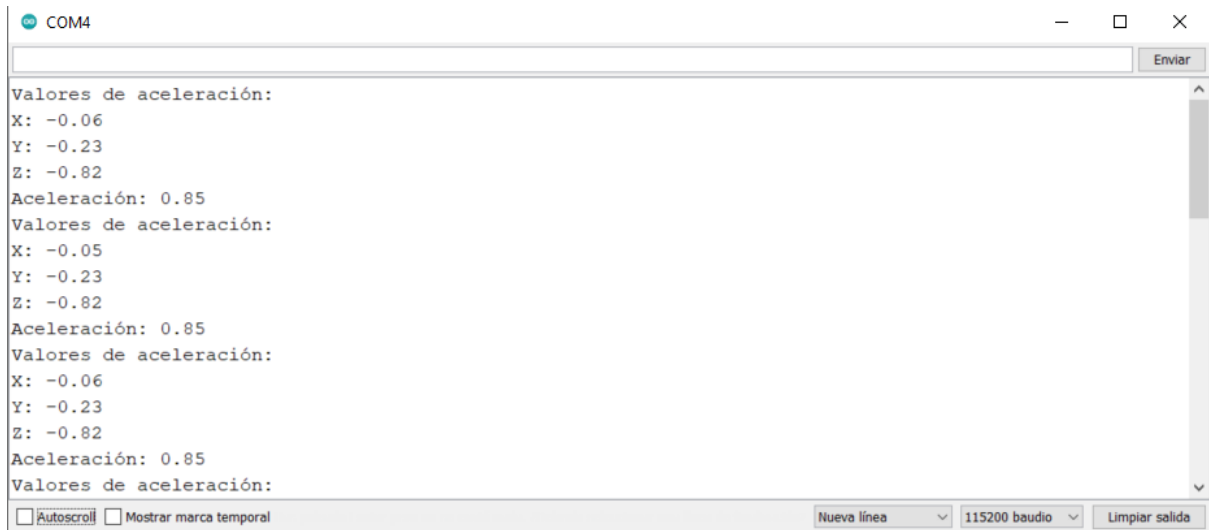
```
void leeSensor() {
    if (interruptCounter > 0) {
        portENTER_CRITICAL(&timerMux);
        interruptCounter--;
        portEXIT_CRITICAL(&timerMux);
        totalInterruptCounter++;
        contadorSerial++;
        contadorLed++;
        if (totalInterruptCounter == 100) {
            sensor.accelUpdate();
            totalInterruptCounter = 0;
        }
        if (contadorSerial == 1000) {
            Serial.println("Valores de aceleración:");
            Serial.println("X: " + String(sensor.accelX()));
            Serial.println("Y: " + String(sensor.accelY()));
            Serial.println("Z: " + String(sensor.accelZ()));
            Serial.println("Aceleración: " + String(sensor.accelSqrt()));
            if (digitalRead(12) == LOW) {
                digitalWrite(12, HIGH);
            }
            contadorSerial = 0;
        }
        if (digitalRead(12) == HIGH && contadorLed == 1200) {
            digitalWrite(12, LOW);
            contadorLed = 0;
        }
    }
}

void loop() {
    leeSensor();
}
```

Leeremos la aceleración del sensor con sus valores x, y, z y el módulo de la aceleración, imprimiéndolos por pantalla, cuando el contadorSerial llega a 1000 ms. Para poder medir en ms, hemos tenido que cambiar la configuración del timerAlarmWrite a 1000, en lugar de 1000000.

Además, pasados 1200 ms se apagará el Led si este estaba encendido.

Los resultados son los siguientes:



```
COM4
Valores de aceleración:
X: -0.06
Y: -0.23
Z: -0.82
Aceleración: 0.85
Valores de aceleración:
X: -0.05
Y: -0.23
Z: -0.82
Aceleración: 0.85
Valores de aceleración:
X: -0.06
Y: -0.23
Z: -0.82
Aceleración: 0.85
Valores de aceleración:
```

☐ Autoscroll ☐ Mostrar marca temporal Nueva línea 115200 baudio Limpiar salida

2. Diseño basado en sistema operativo de tiempo real (RTOS).

Esta práctica se centra en aprender el uso y conceptos asociados a un sistema operativo en tiempo real (RTOS) y diseñar un firmware basado en el mismo.

Trabajamos con una librería llamada ESP32_FreeRtos.

De esta forma, adaptaremos el sistema desarrollado en el punto 5 de la práctica anterior sustituyendo las interrupciones y contadores con tareas que tienen orden de prioridad.

2.1. Implementación con RTOS de la lectura de un sensor inercial.

La tarea que se va a implementar utilizando RTOS es la adquisición de datos de aceleración de nuestro sensor inercial, de forma que el sensor lea la aceleración cada 100 ms y cada segundo, se imprima el valor en pantalla. Al imprimir el valor en pantalla, se debe encender durante 200 ms un led.

Lo primero que debemos hacer es configurar el sistema RTOS:

```
#if CONFIG_FREERTOS_UNICORE
#define ARDUINO_RUNNING_CORE 0
#else
#define ARDUINO_RUNNING_CORE 1
#endif
```

Nuestra librería FreeRTOS funciona a través de tareas, de forma que se crean diferentes tareas independientes entre sí que se ejecutan de forma continua. Para temporizar estas tareas tenemos delays para cada tarea específica, de forma que el delay sólo se aplica a una tarea, sin afectar a las demás. Por ello, crearemos una tarea para cada una de las acciones que debemos realizar, muestreo del acelerador, envío de datos al puerto serie y encendido y apagado del led, que se definen de la siguiente forma:

```
void Muestreo( void *pvParameters );
void EnvioDatos( void *pvParameters );
void Led( void *pvParameters );
```

Además de definir las, tenemos que configurarlas en el setup:

```
void setup() {

    xTaskCreatePinnedToCore(
        Muestreo
        , "Muestreo"
        , 1024
        , NULL
        , 3
        , NULL
        , ARDUINO_RUNNING_CORE);

    xTaskCreatePinnedToCore(
        EnvioDatos
        , "EnvioDatos"
        , 1024
        , NULL
        , 2
        , NULL
        , ARDUINO_RUNNING_CORE);

    xTaskCreatePinnedToCore(
        Led
        , "Led"
        , 1024
        , NULL
        , 1
        , NULL
        , ARDUINO_RUNNING_CORE);

}
```

Como puede observarse, las tareas se crean con el comando `xTaskCreatePinnedToCore`. Debemos pasar el puntero a la función (Muestreo, Envío datos o Led), el nombre de la tarea (Muestreo, EnvíoDatos o Led), el tamaño de pila la pila de tareas en bytes (1024), el puntero que se utilizará como parámetro para la tarea que se está creando (NULL), la prioridad de la tarea (3, 2, 1), un identificador mediante el cual se puede hacer referencia a la tarea creada (NULL), y el núcleo donde correrá la tarea (ARDUINO_RUNNING_CORE).

Preparamos las funciones, añadiendo comentada una pequeña descripción de la tarea.

```
void Led(void *pvParameters) // This is a task.
{
    (void) pvParameters;

    /*
     Led
     Enciende un Led cada segundo, lo apaga tras 200 ms y se vuelve a encender cada segundo.
     El Led debe estar conectado al pin 12
    */

    pinMode(12, OUTPUT);

    for (;;)
    {
        vTaskDelay(1000 / portTICK_RATE_MS);
        digitalWrite(12, HIGH);
        vTaskDelay(200 / portTICK_RATE_MS);
        digitalWrite(12, LOW);
    }
}
```

Utilizamos los `vTaskDelay` para temporizar las acciones, en este caso el Led se espera 1000 ms para encenderse y se apaga tras otros 200 ms. Los `vTaskDelay` se definen en ticks, por lo que dividimos el valor en ms que queremos por el `portTICK_RATE_MS`.

De forma similar, tendremos las otras dos funciones:

```
void Muestreo(void *pvParameters)
{
    (void) pvParameters;

    /*
     Muestreo
     Lee la aceleración de un sensor MPU9250 cada 100 ms.
    */

    Wire.begin(SDA_PIN, SCL_PIN);

    sensor.setWire(&Wire);
    sensor.beginAccel();
}

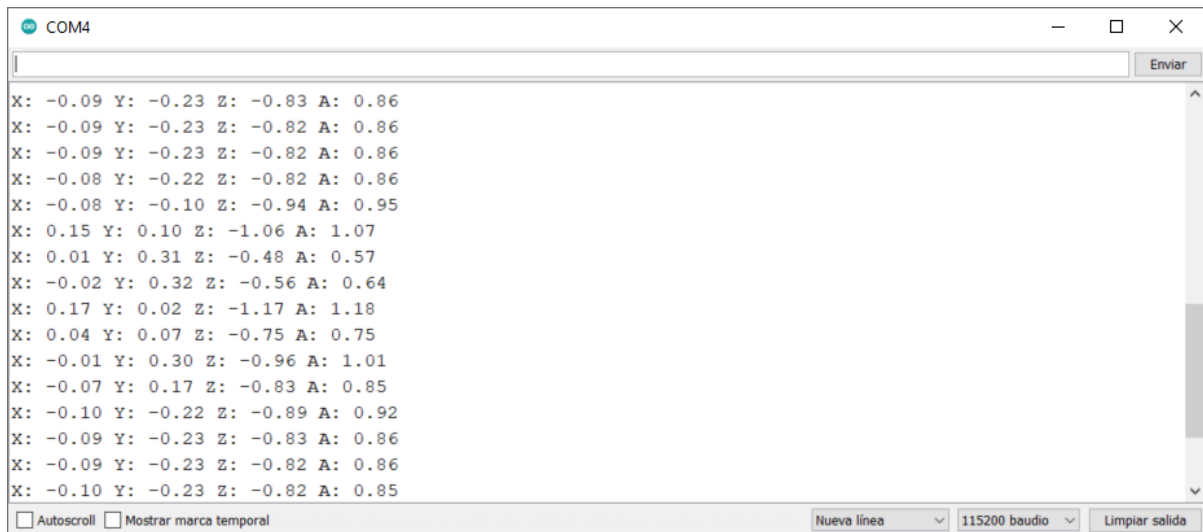
void EnvioDatos(void *pvParameters)
{
    (void) pvParameters;

    /*
     EnvioDatos
     Imprime en pantalla la aceleración de un sensor MPU9250 cada segundo
    */

    Serial.begin(115200);

    for (;;)
    {
        Serial.print("X: ");
        Serial.print(String(sensor.accelX()));
        Serial.print(" Y: ");
        Serial.print(String(sensor.accelY()));
        Serial.print(" Z: ");
        Serial.print(String(sensor.accelZ()));
        Serial.print(" A: ");
        Serial.print(String(sensor.accelSqrt()));
        Serial.println();
        vTaskDelay(1000 / portTICK_RATE_MS);
    }
}
```

El resultado es igual que el obtenido en la práctica 1.5, sin embargo, utilizando este tipo de sistema vemos que nos podemos evitar ocupar memoria con contadores para cada función que queremos utilizar.



COM4

Enviar

```
X: -0.09 Y: -0.23 Z: -0.83 A: 0.86
X: -0.09 Y: -0.23 Z: -0.82 A: 0.86
X: -0.09 Y: -0.23 Z: -0.82 A: 0.86
X: -0.08 Y: -0.22 Z: -0.82 A: 0.86
X: -0.08 Y: -0.10 Z: -0.94 A: 0.95
X: 0.15 Y: 0.10 Z: -1.06 A: 1.07
X: 0.01 Y: 0.31 Z: -0.48 A: 0.57
X: -0.02 Y: 0.32 Z: -0.56 A: 0.64
X: 0.17 Y: 0.02 Z: -1.17 A: 1.18
X: 0.04 Y: 0.07 Z: -0.75 A: 0.75
X: -0.01 Y: 0.30 Z: -0.96 A: 1.01
X: -0.07 Y: 0.17 Z: -0.83 A: 0.85
X: -0.10 Y: -0.22 Z: -0.89 A: 0.92
X: -0.09 Y: -0.23 Z: -0.83 A: 0.86
X: -0.09 Y: -0.23 Z: -0.82 A: 0.86
X: -0.10 Y: -0.23 Z: -0.82 A: 0.85
```

☐ Autoscroll ☐ Mostrar marca temporal

Nueva línea 115200 baudio Limpiar salida

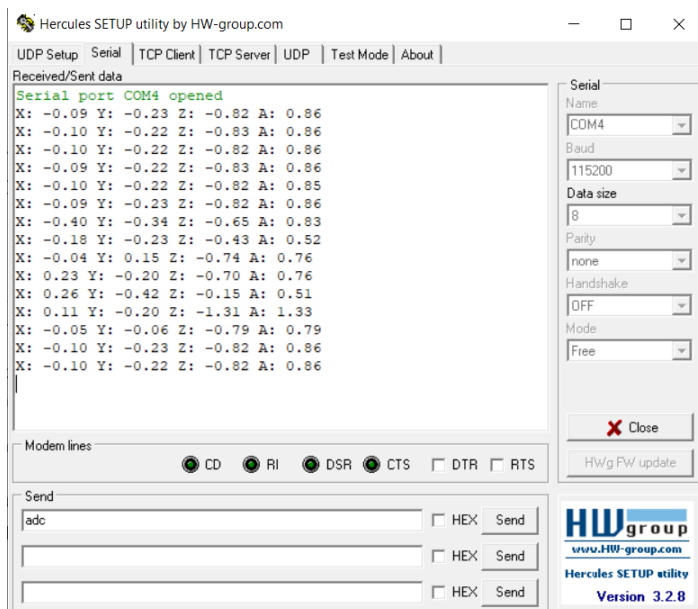
3. Comunicación, gestión y representación de datos de sensores con Python.

En esta sesión de prácticas, el objetivo fundamental es familiarizarse con Python como lenguaje de programación, aprender a gestionar datos procedentes de comunicaciones serie, trabajar con estos datos en formato .txt para almacenarlos en ficheros, y operar con los datos y representarlos gráficamente.

Para ello, partiremos de la práctica anterior donde se utiliza un sensor inercial que muestrea la aceleración y manda los datos vía UART cada 100 ms.

3.1. Comprobación recepción de datos.

En primer lugar, para comprobar que la información se envía correctamente por el puerto serie desde el esp32 utilizaremos HERCULES.



3.2. Almacenar los datos en fichero .txt

Una vez comprobado el correcto funcionamiento del puerto serie se crea un programa capaz de acceder al puerto serie, muestre por pantalla los datos en tiempo real y los almacene en un fichero .txt separando cada variable con “,” y con retorno de carro al final de cada muestra.

```
import serial
import os

PuertoSerie = serial.Serial('COM4', baudrate=115200, timeout=1.0)
file = open ("./practica3.4.txt", "w")

while True:
    try:
        esp32 = PuertoSerie.readline()
        #print(esp32)
        datosCaracter = ""
        for valor in esp32:
            datosCaracter = datosCaracter + chr(valor)
            file.write(datosCaracter.replace(" ", ","))
            print(str(datosCaracter))
        except KeyboardInterrupt:
            break

PuertoSerie.close()
file.close()
```

En primer lugar, se ha creado la variable PuertoSerie para definir las características de este puerto. Para ello se ha llamado a la función serial y se ha definido el puerto COM4 con un baudrate de 115200, tal y como se inicializa en el esp32.

Se crea también el archivo de texto practica3.4.txt.

En el bucle principal del código se lee el puerto serie con la función PuertoSerie.readline(), todo lo que leamos del puerto serie estará contenido en la variable esp32, la cual hay que tratar para eliminar la información no deseada, es decir, las tabulaciones y retornos de carro, separar cada elemento por “;” y escribirlo tanto por consola como en el archivo .txt

```
Python 3.8.5 (default, Sep 3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license()" for more information.

IPython 7.19.0 -- An enhanced Interactive Python.

In [1]: runfile('G:/./shortcut-targets-by-id/1vU1T4mBrhb3u29jx_INOqZwdrPJRZP2/
RSENSE-720251/2_Pract/P3/Practica3.4.py', wdir='G:/./shortcut-targets-by-id/
1vU1T4mBrhb3u29jx_INOqZwdrPJRZP2/RSENSE-720251/2_Pract/P3')
X: 0.00 Y: 0.00 Z: 0.00 A: 0.00

X: -0.10 Y: -0.22 Z: -0.83 A: 0.86
X: -0.10 Y: -0.22 Z: -0.82 A: 0.86
X: -0.10 Y: -0.22 Z: -0.83 A: 0.86
X: -0.10 Y: -0.23 Z: -0.82 A: 0.86
X: -0.10 Y: -0.22 Z: -0.83 A: 0.86
X: -0.10 Y: -0.22 Z: -0.83 A: 0.86
X: -0.10 Y: -0.22 Z: -0.83 A: 0.86
X: -0.10 Y: -0.22 Z: -0.83 A: 0.86
X: -0.10 Y: -0.22 Z: -0.82 A: 0.86
X: -0.10 Y: -0.22 Z: -0.82 A: 0.86
X: -0.10 Y: -0.22 Z: -0.82 A: 0.86
X: -0.10 Y: -0.22 Z: -0.82 A: 0.86
X: -0.10 Y: -0.22 Z: -0.82 A: 0.85
```

```
practica3.4.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
X:;0.00;Y:;0.00;Z:;0.00;A:;0.00

X:;-0.10;Y:;-0.22;Z:;-0.83;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.82;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.83;A:;0.86
X:;-0.10;Y:;-0.23;Z:;-0.82;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.83;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.83;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.83;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.83;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.82;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.82;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.82;A:;0.86
X:;-0.10;Y:;-0.22;Z:;-0.82;A:;0.85
```

3.3. Procesar y graficar en tiempo real.

Una vez conseguida la información a través del puerto serie, se debe acumular durante 5 segundos, calculando el promedio y la desviación estándar para representarlos en tiempo real.

En este caso, no solo es necesario importar la librería serie, sino que también se debe importar la librería numpy para manejarse con números y realizar cálculos estadísticos, la librería time para manejar los timers, la librería plt para realizar gráficos y la librería drawnow para realizar actualizaciones en tiempo real.

```
# 1.- DEFINIMOS EL PUERTO
PuertoSerie = serial.Serial('COM4', baudrate=115200, timeout=1.0)

#2.- LEEMOS DEL PUERTO SERIE LIMPIANDO LA INFORMACIÓN NO DESEADA
# Y CREAMOS UN TIMER PARA QUE ACUMULE LOS DATOS CADA 5 SEGUNDOS

signal = {'X' : [], 'Y' : [], 'Z' : [], 'A' : []}
mean = {'X' : [], 'Y' : [], 'Z' : [], 'A' : []}
std = {'X' : [], 'Y' : [], 'Z' : [], 'A' : []}
```

Igual que en el apartado anterior, se debe configurar el puerto y leerlo, limpiando la información no deseada, en este caso, solo queremos guardar los parámetros, que meteremos dentro de una variable de tipo diccionario denominada signal. Crearemos dos variables más de este tipo para la media y la desviación estándar.

```
tiempo = []
tiempoCero = time.time()
tiempoTomadeDatos = 5.0
```

Generamos, también, un acumulador de tiempo, inicializamos tiempoCero como time.time() (parecido a millis() de Arduino) y elegimos un tiempo de toma de datos igual a 5 segundos.

```

while True:
    tiempoInicio = time.time()
    tiempoActual = tiempoInicio
    while(tiempoTomadeDatos >= (tiempoActual-tiempoInicio)):
        esp32 = PuertoSerie.readline()
        datosCaracter = ""
        for valor in esp32:
            datosCaracter = datosCaracter + chr(valor)
        prueba = datosCaracter.replace(" ", ";").replace("\n", "").replace("\r", "").split(";")
        listaTexto =list(prueba)
        signal['X'].append(float(listaTexto[1]))
        signal['Y'].append(float(listaTexto[3]))
        signal['Z'].append(float(listaTexto[5]))
        signal['A'].append(float(listaTexto[7]))
        tiempoActual = time.time()
    npX = np.array(signal['X'])
    npY = np.array(signal['Y'])
    npZ = np.array(signal['Z'])
    npA = np.array(signal['A'])
    mean['X'].append(npX.mean())
    mean['Y'].append(npY.mean())
    mean['Z'].append(npZ.mean())
    mean['A'].append(npA.mean())
    std['X'].append(npX.std())
    std['Y'].append(npY.std())
    std['Z'].append(npZ.std())
    std['A'].append(npA.std())

    tiempo.append(tiempoActual - tiempoCero)

drawnow(Figuras)

```

En el bucle principal, inicializamos el tiempo de inicio con `time.time()` y el tiempo actual igual a tiempo de inicio. Por lo que, mientras que el tiempo de toma de datos sea mayor o igual a la diferencia entre el tiempo actual y el tiempo de inicio, leeremos el puerto serie, limpiaremos los caracteres no deseados y utilizaremos una lista de texto para guardar el nombre de la variable y su parámetro.

Como solo queremos tener los parámetros y no sus nombres, que ya vienen definidos en la variable diccionario, pasaremos únicamente los parámetros en formato float, realizando un `append` para que se vayan acumulando en ese transcurso de tiempo.

Para poder calcular la media y la desviación típica, es necesario convertir cada conjunto de parámetros de cada variable del diccionario en un array. Para ello, se utiliza la librería `numpy`. De esta forma, cada conjunto de parámetros queda dentro de un array.

Finalmente, para realizar el sumatorio de la media y la desviación estándar de cada parámetro se utiliza la función `append`, realizando la media o la desviación estándar del array que se requiere, metiéndolo en la variable diccionario correspondiente.

Se realiza también un sumatorio del tiempo para poder representar las variables en función del tiempo y se llama a la función `drawnow`, pasándole el argumento `Figuras` que es una función generada para graficar cada uno de los parámetros con su media y su desviación estándar.

```

def Figuras():
    plt.subplot(2,2,1)
    plt.ylim(-2,2)
    plt.title('X', fontsize=12, fontweight='bold', color='k')
    plt.ylabel('Valor [g]', fontsize=12, fontweight='bold', color='k')
    plt.xlabel('Tiempo [s]', fontsize=12, fontweight='bold', color='k')
    plt.plot(tiempo, mean['X'], label = 'media')
    plt.plot(tiempo, std['X'], label = 'desv. est.')
    plt.legend()

    plt.subplot(2,2,2)
    plt.ylim(-2,2)
    plt.title('Y', fontsize=12, fontweight='bold', color='k')
    plt.ylabel('Valor [g]', fontsize=12, fontweight='bold', color='k')
    plt.xlabel('Tiempo [s]', fontsize=12, fontweight='bold', color='k')
    plt.plot(tiempo, mean['Y'], label = 'media')
    plt.plot(tiempo, std['Y'], label = 'desv. est.')
    plt.legend()

```



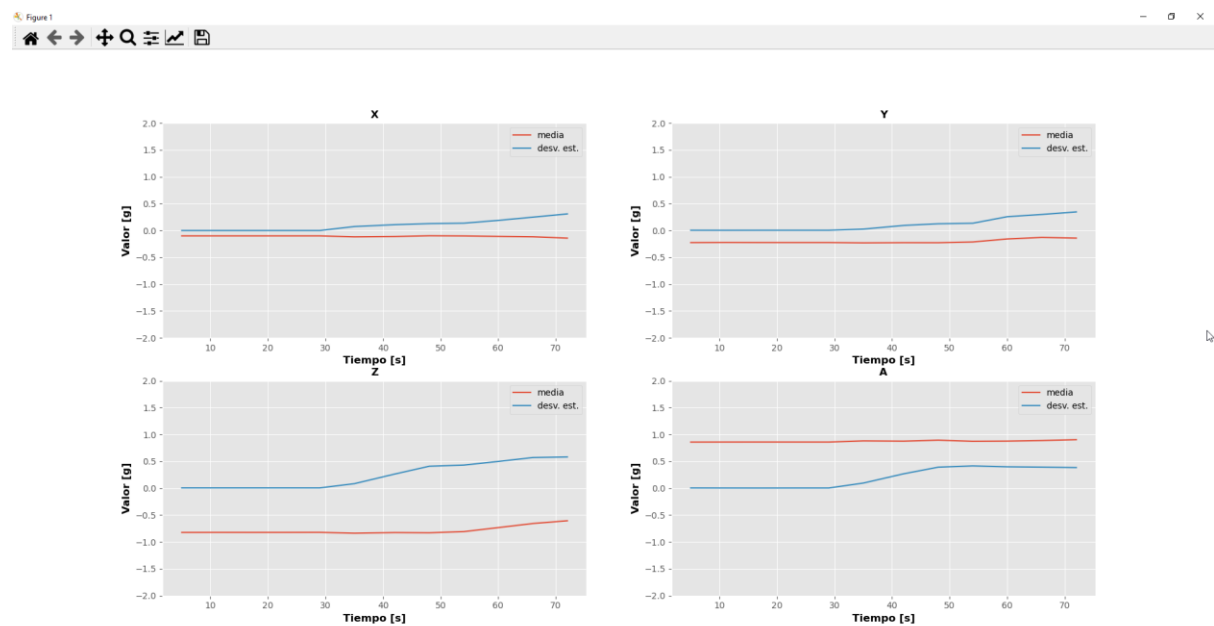
```

plt.subplot(2,2,3)
plt.ylim(-2,2)
plt.title('Z', fontsize=12, fontweight='bold', color='k')
plt.ylabel('Valor [g]', fontsize=12, fontweight='bold', color='k')
plt.xlabel('Tiempo [s]', fontsize=12, fontweight='bold', color='k')
plt.plot(tiempo, mean['Z'], label = 'media')
plt.plot(tiempo, std['Z'], label = 'desv. est.')
plt.legend()

plt.subplot(2,2,4)
plt.ylim(-2,2)
plt.title('A', fontsize=12, fontweight='bold', color='k')
plt.ylabel('Valor [g]', fontsize=12, fontweight='bold', color='k')
plt.xlabel('Tiempo [s]', fontsize=12, fontweight='bold', color='k')
plt.plot(tiempo, mean['A'], label = 'media')
plt.plot(tiempo, std['A'], label = 'desv. est.')
plt.legend()

```

El resultado obtenido es el siguiente:



4. Comunicaciones WIFI desde microcontrolador. Stack IP y subida de datos a la nube.

Para esta sesión de prácticas se deben intentar alcanzar como objetivos la puesta en marcha de los conocimientos de las redes inalámbricas WiFi, el manejo de comunicaciones IP a bajo nivel mediante sockets y el manejo de protocolos de alto nivel HTTP, NTP, FTP, MQTT y estándares como SENML.

4.1. Socket TCP para mandar la hora mediante start y stop

En este apartado se va a implementar un programa para aprender a manejarse con el protocolo TCP.

```
#include <WiFi.h>
#include "time.h"

const char* ssid      = "vodafoneBA1157";
const char* password   = "SRULAGD6RHFQ4M5K";

const IPAddress serverIP(192, 168, 0, 13); // La dirección que desea visitar
uint16_t serverPort = 1080;              // Número de puerto del servidor
```

Para poder utilizar este protocolo vía WiFi, es necesario, además de incluir la librería WiFi, asignar su nombre de usuario y contraseña. Otros dos parámetros a tener en cuenta para el manejo de protocolo TCP son el servidor IP que deseamos visitar, es decir, la IP del servidor que pondremos en la aplicación SocketTest, y el puerto del servidor.

```
const char* ntpServer = "europe.pool.ntp.org";
const long  gmtOffset_sec = 3600;
const int   daylightOffset_sec = 0;
```

Para obtener la fecha y la hora se va a utilizar el servidor NTP, se trata de un protocolo de internet para sincronizar los relojes a través de una red. Como estamos en Europa, se especifica la dirección del servidor NTP como "europe.pool.ntp.org", se ajusta la compensación UTC y, como estamos en UTC+1 es 3600 y al ser horario de invierno, la compensación de luz será cero.

```
void setup()
{
    Serial.begin(115200);
    timer = timerBegin(0, 80, true);
    timerAttachInterrupt(timer, &onTimer, true);
    timerAlarmWrite(timer, 1000, true);
    timerAlarmEnable(timer);
    WiFi.mode(WIFI_STA);
    WiFi.setSleep(false);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println("Connected");
    Serial.print("IP Address:");
    Serial.println(WiFi.localIP());

    configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
    //printLocalTime();
}
```

Para inicializar el esp32 como cliente NTP se usa configTime. Además, se inicializa el WIFI, el timer para generar las interrupciones necesarias y el puerto serie.

```

void printLocalTime()
{
    if (interruptCounter > 0) {
        struct tm timeinfo;
        if (!getLocalTime(&timeinfo)) {
            Serial.println("Hora no obtenida");
            return;
        }
        portENTER_CRITICAL(&timerMux);
        interruptCounter--;
        portEXIT_CRITICAL(&timerMux);
        Serial.println(&timeinfo, "%A, %d %B %Y %H:%M:%S");
        client.println(&timeinfo, "%A, %d %B %Y %H:%M:%S");
        //delay(1000);
    }
}

```

Además, se genera una función para imprimir la hora utilizando interrupciones cada segundo. Se utiliza la función interna de NTP, `getLocalTime`, para transmitir un paquete de solicitud a un servidor NTP y analizar el paquete de marca de tiempo en un formato legible, utilizando la estructura del tiempo como parámetro.

```

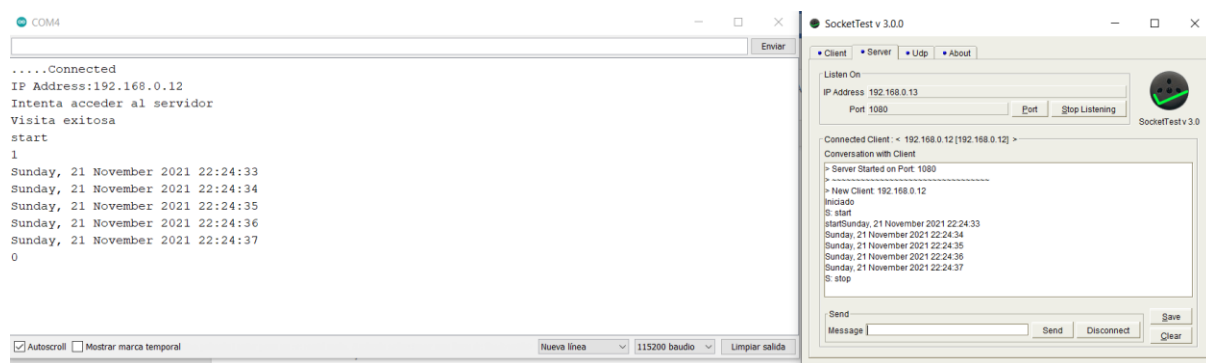
void accesoServidor()
{
    Serial.println("Intenta acceder al servidor");
    if (client.connect(serverIP, serverPort)) // Intenta acceder a la dirección de destino
    {
        Serial.println("Visita exitosa");
        client.println("Iniciado");
        ValorEscritura = 0;
        while (client.connected() || client.available()) // Si está conectado o se han recibido datos no leídos
        {
            if (client.available() > 0) // Si hay datos para leer
            {
                String escritura = client.readStringUntil('\n'); // Leer datos a nueva línea
                Serial.println(escritura);
                client.write(escritura.c_str()); // Devuelve los datos recibidos
                if (escritura.substring(0, 5) == "start") {
                    ValorEscritura = 1;
                    Serial.println(ValorEscritura);
                }
                while (ValorEscritura == 1) {
                    printLocalTime();
                    escritura = client.readStringUntil('\n');
                    if (escritura.substring(0, 4) == "stop") {
                        ValorEscritura = 0;
                        Serial.println(ValorEscritura);
                    }
                }
            }
        }
    }
}

```

Finalmente, se genera otra función “`accesoServidor`” que será llamada por el bucle principal. Esta función comprueba si el cliente, que somos nosotros, está conectado a la dirección IP y al puerto del servidor. Cuando se conecta, se muestra por pantalla y se utiliza la variable `ValorEscritura`, para mostrar por pantalla siempre que esta se encuentre a 1.

Cuando el cliente tiene datos que leer, observamos estos datos en el string `escritura` mediante `readStringUntil`, tal y como se hacía en la práctica 1 por consola, pero esta vez leyendo lo que recibe el cliente. Cuando lo que hemos recibido es `start`, se pone `ValorEscritura` a 1 y siempre que se encuentre así, se llama a la función `printLocalTime`, que se encarga de escribir por el puerto serie la fecha y la hora y de enviársela al servidor. Además, si el servidor envía al cliente la orden de `stop`, `valorEscritura` pasa a ser un 0 y, de esta forma, se deja de escribir en consola y de mandar al servidor la fecha y la hora.

Observamos los resultados a continuación:



Es posible apreciar como en la pestaña de “SocketTest” se inicia el servidor en el puerto 1080. Se detecta un nuevo cliente con IP 192.168.0.12 que es exactamente la misma dirección IP que tiene nuestro esp32. Una vez se ha conectado, observamos como en el terminal COM4 aparece “Visita exitosa” indicando que ya estamos conectados al servidor y, además, en “SocketTest” nos aparece el mensaje “Iniciado”. Es ahora cuando podemos enviar un mensaje desde el servidor al cliente. Al mandar el comando “start” desde el servidor al cliente, observamos como el cliente le devuelve al servidor tanto los caracteres que ha recibido como la fecha y la hora actual. En el terminal COM4 observamos como se ha recibido el comando “start” lo que pone la variable ValorEscritura a 1, haciendo que se escriba y se envíe la fecha y la hora. Cuando escribimos en el servidor el comando stop, observamos como la variable ValorEscritura pasa a ser un 0 y se deja de escribir y mandar la fecha y la hora.

4.2. Servidor WEB que muestre la hora y sea capaz de resetearla

En este apartado, se va a implementar un programa para aprender a manejar el protocolo HTTP. El objetivo que se persigue es conseguir mostrar la hora en tiempo real en un servidor WEB y que, al pulsar un botón, la hora se resetee a cero y empiece a contar desde ese momento.

Al igual que en el apartado anterior, se debe incluir la librería WiFi y asignar su nombre de usuario y contraseña. También se debe sincronizar la fecha y la hora mediante el servidor NTP y, además se deben incluir las librerías “ESPAsyncWebServer.h” y “SPIFFS.h” definiendo el objeto servidor de tipo AsyncWebServer en el puerto 80.

```
String proceso(const String& var) {
```

Para realizar el proceso de reseteo de la hora se genera una función de tipo String “proceso” que posee como argumento un String local de tipo var.

```
    if (var == "STATE" || var == "RESET") {  
        getLocalTime(&timeinfo);  
        h = timeinfo.tm_hour;  
        m = timeinfo.tm_min;  
        s = timeinfo.tm_sec;
```

Si este string es igual a state o a reset, se utiliza la función getLocalTime para transmitir un paquete de solicitud al servidor NTP y se guardan los valores de hora, minuto y segundo en unas variables auxiliares (h, m, s) que se van a utilizar para realizar el posterior reseteo.

```

if (flag) { //PULSAMOS RESET
    h_reset = h;
    m_reset = m;
    s_reset = s;
    flag = false;
}

```

Cuando se recibe un pulso en reset, los valores de reset de hora, minuto y segundo pasan a ser igual a los que había en la hora actual en el momento de recibir el pulso.

```

//Hora mostrada
h = h - h_reset;
m = m - m_reset;
s = s - s_reset;
if (s < 0) {
    s = s + 60;
    m = m - 1;
}

```

Ahora es cuando se calcula el tiempo reseteado. Se obtiene de forma que h va a ser igual a la diferencia entre la hora actual y la hora guardada en el momento de realizar el reset. Igual ocurre con los minutos y los segundos.

```

if (s < 10) { //Para añadir 0 a la izquierda
    hora_S = String("0" + String(s));
}
else {
    hora_S = String(":" + String(s));
}
if (m < 10) {
    hora_S = String("0" + String(m) + hora_S);
}
else {
    hora_S = String(":" + String(m) + hora_S);
}
if (h < 10) {
    hora_S = String("0" + String(h) + hora_S);
}
else {
    hora_S = String(String(h) + hora_S);
}
return hora_S;
}
return String();
}

```

Además, como lo que se manda es un String, hay que rellenar los ceros a la izquierda que faltan, que se realiza de forma que cuando los segundos son menores a 10 se añade un 0, y lo mismo con los minutos y las horas. Esta función devuelve al string la hora total.

En el setup se inicializa el protocolo SPIFFS, el WiFi y el servidor NTP con configTime.

La librería que hemos incluido, ESPAsyncWebServer, permite configurar las rutas donde el servidor escuchará las solicitudes HTTP entrantes y ejecutará funciones cuando se reciba una solicitud en esa ruta. Para eso se utiliza el método "on" en el objeto del servidor.

```
<!DOCTYPE html>
```

```
<html>
<head>
  <title>ESP32 Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,">
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta http-equiv="refresh" content="1">
</head>
<body>
  <h1>ESP32 Web Server</h1>
  <p>HORA: <strong> %STATE%</strong></p>
  <p><a href="/reset"><button class="button">RESET</button></a></p>
</body>
</html>
```

```
html {
  font-family: Helvetica;
  display: inline-block;
  margin: 0px auto;
  text-align: center;
}
h1{
  color: #0F3376;
  padding: 2vh;
}
p{
  font-size: 1.5rem;
}
button {
  display: inline-block;
  background-color: #008CBA;
  border: none;
  border-radius: 4px;
  color: white;
  padding: 16px 40px;
  text-decoration: none;
  font-size: 30px;
  margin: 2px;
  cursor: pointer;
}
```

Para poder configurar las rutas es necesario crear los archivos index.html y style.css que contienen un título, un botón y un marcador de posición, y el tamaño de fuente, estilo, color del botón y la alineación de la página, respectivamente, y subirlos pulsando Herramientas/ESP32 Sketch Data Upload.

```
// Route for root / web page
server.on("/", HTTP_GET, [] (AsyncWebServerRequest * request) {
  request->send(SPIFFS, "/index.html", String(), false, proceso);
});

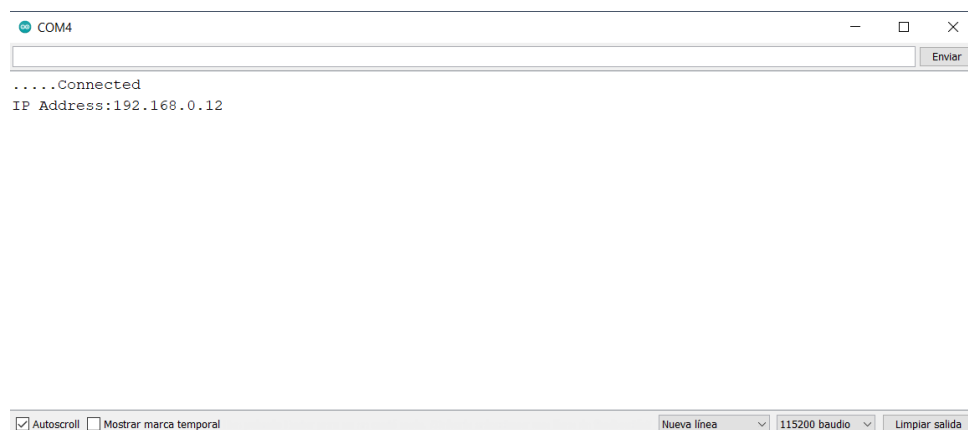
// Route to load style.css file
server.on("/style.css", HTTP_GET, [] (AsyncWebServerRequest * request) {
  request->send(SPIFFS, "/style.css", "text/css");
});
```

Cuando el servidor recibe una solicitud en la URL raíz, envía el archivo index.html configurado al cliente. Además, el cliente realizará también una solicitud para el archivo CSS.

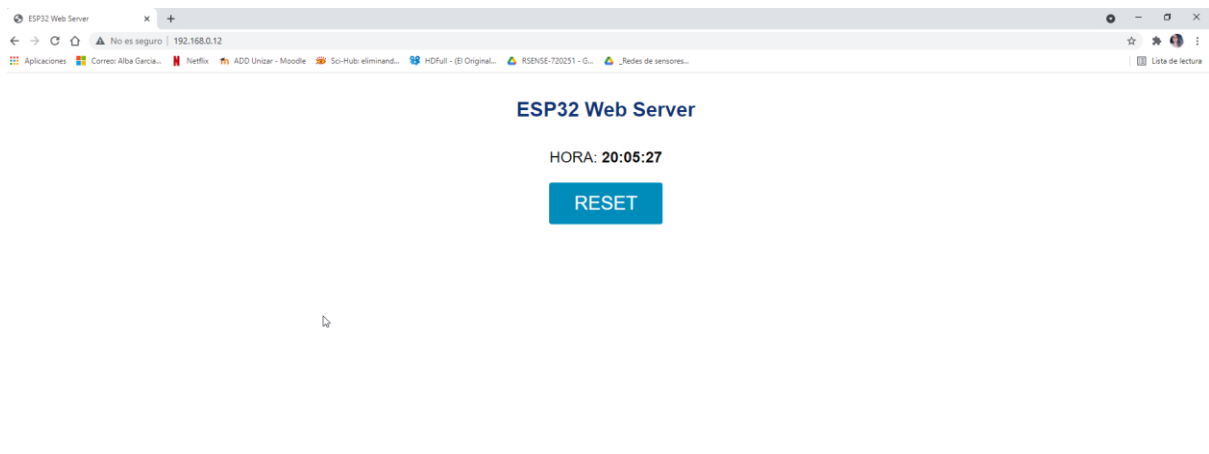
```
server.on("/reset", HTTP_GET, [] (AsyncWebServerRequest * request) {
  request->send(SPIFFS, "/index.html", String(), false, proceso);
  flag = true;
});
// Start server
server.begin();
```

Finalmente, se necesita definir lo que sucede en las rutas reset. Cuando se realiza esta solicitud, (al pulsar reset) la hora se pone a cero y no es hasta que se vuelve a la ruta raíz cuando empieza a contar la hora de nuevo.

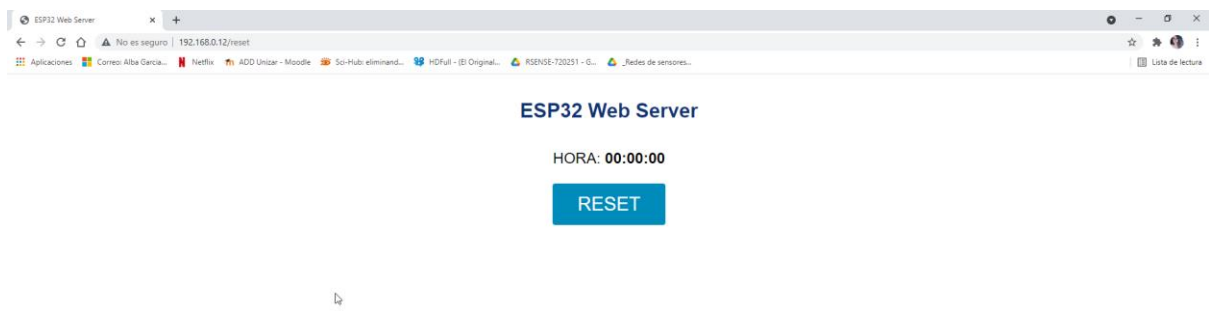
Al lanzar el código observamos por consola:



Por lo que nos vamos a conectar a esa dirección IP.



En ella observamos la hora en tiempo real. Si pulsamos reset, observaremos:



Vemos que el tiempo se ha reseteado, pero no corre, esto es porque en la ruta 192.168.0.12 aparece /reset. Debemos volver a la ruta raíz para que la hora reseteada comience a correr de nuevo.



4.3. Subir fichero JSON a un servidor FTP

El siguiente apartado de la práctica consiste en implementar un programa para aprender a manejar el servidor TCP. El objetivo que se persigue es conseguir generar un fichero json que se suba al servidor cada 10 segundos y contenga una temperatura aleatoria y la marca temporal.

Para este apartado es necesario incluir las librerías “arduinojson.h” y “esp32_ftpClient”, además de las utilizadas anteriormente.

```
char ftp_server[] = "155.210.150.77";
char ftp_user[]   = "rsense";
char ftp_pass[]   = "rsense";
ESP32_FTPClient ftp (ftp_server, ftp_user, ftp_pass, 5000, 2);
```

Como en los apartados anteriores, será necesario definir el usuario y la contraseña del WiFi y, para este caso, incluir la información necesaria para poder conectarse al servidor FTP de la universidad.

También se crean variables para el json, definiendo el documento que se guarda en la memoria estática, con una capacidad de memoria definida en función del tamaño del array y del tamaño del objeto.

```
const int capacity = JSON_ARRAY_SIZE(100) + 100 * JSON_OBJECT_SIZE(4);
StaticJsonDocument<capacity> doc;
char datosChar[capacity];

void almacenaDatos() {
    JsonObject obj1 = doc.createNestedObject();
    obj1["t"] = random(10, 30);
    obj1["u"] = "C";
    obj1["h"] = timeinfo.tm_hour;
    obj1["m"] = timeinfo.tm_min;
    obj1["s"] = timeinfo.tm_sec;
}
```

Se crea la función almacenaDatos en la que se crea una variable, obj1, de tipo JsonObject que posee los datos del JsonDocument asignado, doc. Todo ello se hace con createNestedObject. La información que almacena es una temperatura aleatoria entre 10 y 30, sus unidades, la hora, el minuto y el segundo del momento en el que se almacena.

```
void escribeFichero() {
    serializeJson(doc, datosChar);
    doc.clear();
}
```

Se crea también, la función escribeFichero para serializar un json document (doc) y crear un documento json sin espacios ni saltos de línea entre valores (datosChar).

```
void mandaFichero() {
    ftp.OpenConnection();
    ftp.ChangeWorkDir("/rsense/720251");
    ftp.InitFile("Type A");
    ftp.NewFile("720251_211119.json");
    ftp.Write(datosChar);
    ftp.CloseFile();
    ftp.CloseConnection();
}
```

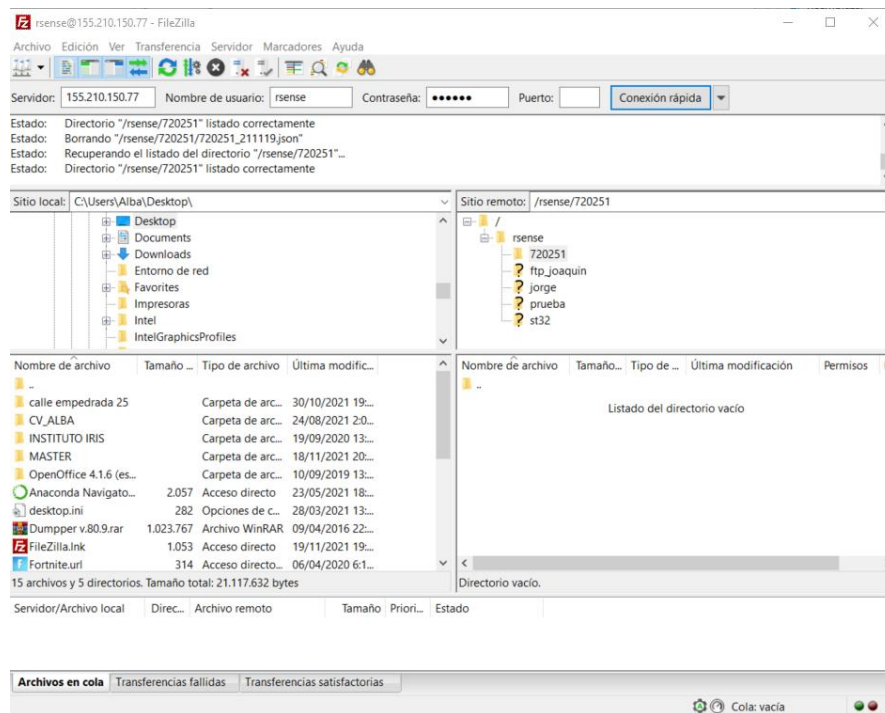
Finalmente, crea la función mandaFichero. En esta función se abre la conexión con el servidor FTP definido anteriormente, se cambia de directorio para abrir el que nosotros queremos, se inicia el

archivo con el tipo que queremos, se crea el archivo, se escriben en él los datos que queremos pasarle (datosChar), se cierra el archivo y se cierra la conexión.

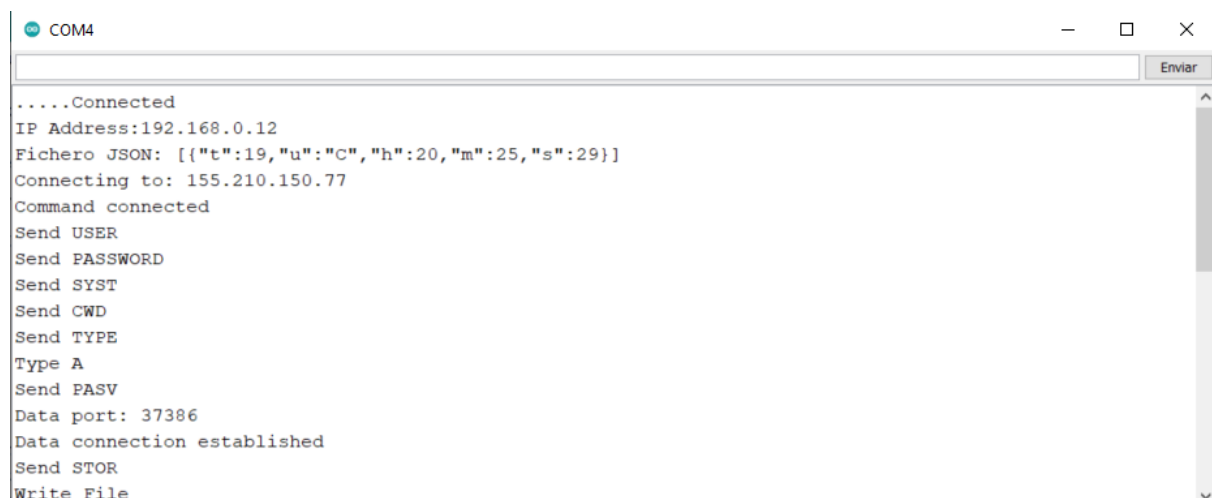
Como en apartados anteriores, se debe inicializar tanto el puerto serie, como las interrupciones, el WiFi y la configuración de la hora obtenida del servidor NTP.

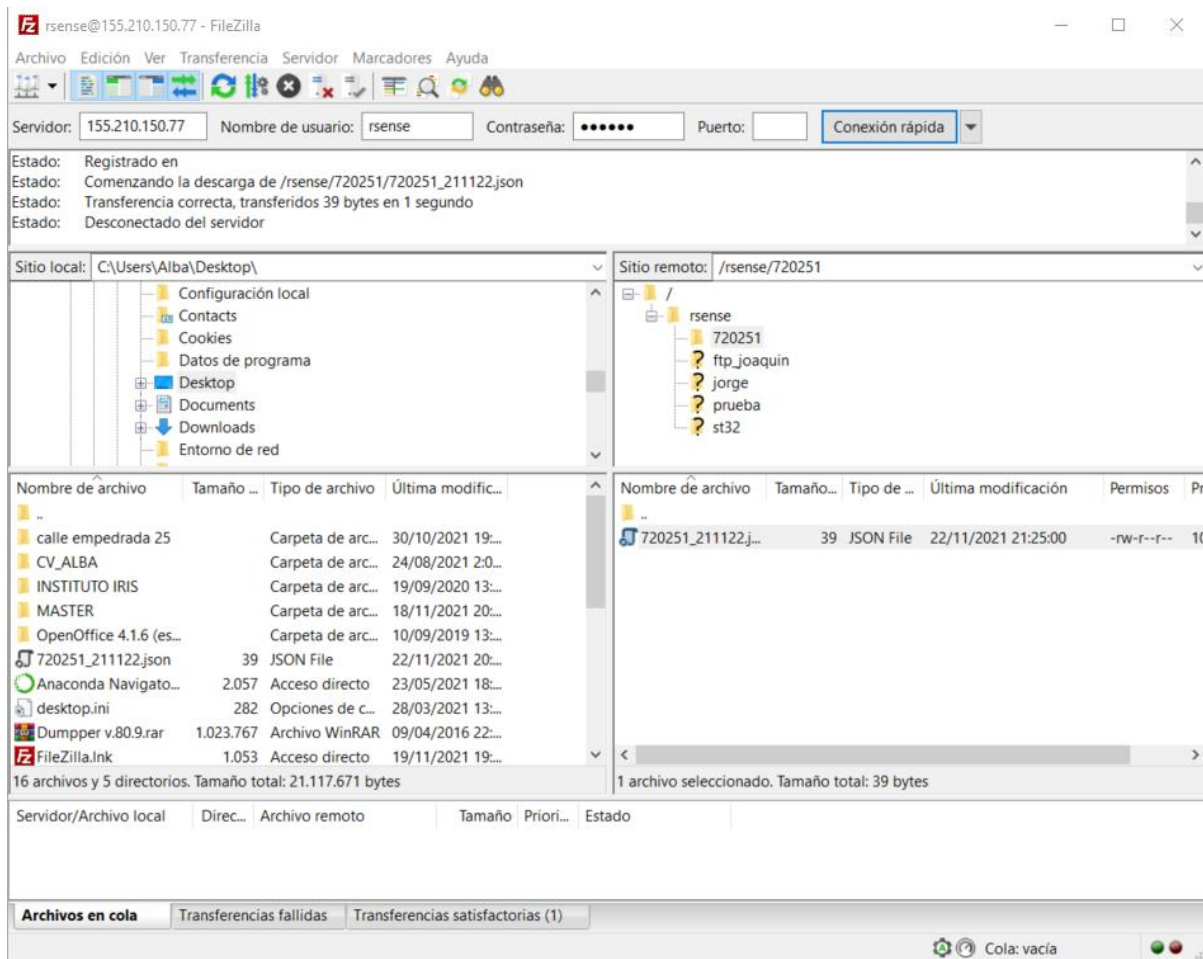
Además, en el bucle principal se llama a las funciones generadas anteriormente con una periodicidad de 10 segundos, momento en el cual se realiza la interrupción.

Para comprobar que todo ello funciona correctamente debemos entrar en el servidor de la universidad, en la carpeta rsense/720251 que es la que hemos definido para guardar nuestro archivo.json.



Al cargar el código observamos por consola lo siguiente y donde, cada 10 segundos, nos muestra un nuevo dato en el fichero JSON, que se envía al servidor FTP.





Comprobamos que el archivo enviado es el mismo que el que nos ha mostrado por consola.

```
1 [{"t":19,"u":"C","h":20,"m":25,"s":29}]
```

4.4. Subir datos usando MQTT

Finalmente, el último apartado de la práctica consiste en utilizar uno de los ejemplos de Adafruit donde se utiliza mqtt para realizar un contador y poder visualizarlo en la web de adafruitIO, en un feed asignado de Adafruit.

En este caso, en vez de configurar los parámetros en el código principal de Arduino, se configuran dentro de la librería config.h. Estos parámetros son el usuario y la contraseña del WiFi que se va a utilizar, el usuario y la key de Adafruit IO y otro conjunto de parámetros que nos permiten trabajar con esta aplicación.

En el código principal se crea un puntero “contador” que va a estar asociado al feed de adafruit io.

```
prueba->onMessage(Mensaje);
```

Como siempre, en el setup inicializamos el puerto serie y el WiFi. También se asignará a contador la función interna OnMessage a la que se le pasa como argumento nuestra propia función Mensaje, encargada de pasar el puntero data de Adrafruitio de forma que recibamos el valor que se envía desde el feed y lo imprima por pantalla.

```
void Mensaje(AdafruitIO_Data *data) {

    Serial.print("Recibido <- ");
    Serial.println(data->value());

}
```

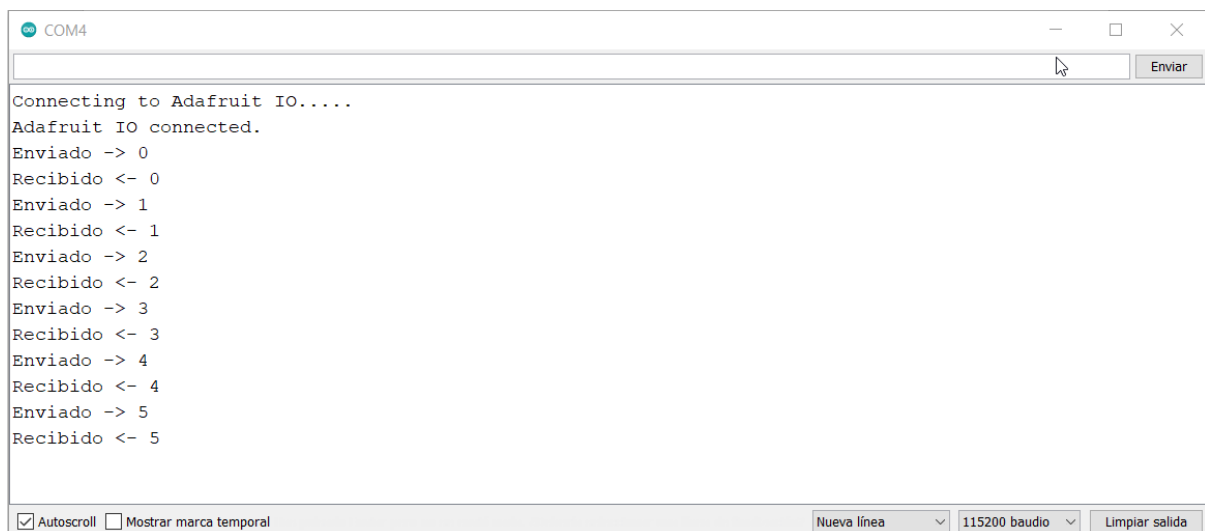
En el bucle principal se utiliza la función `millis` para actualizar el valor del contador cada 5 segundos, evitando, de esta forma, producir interrupciones, asignando a contador el contador guardado y aumentando la unidad.

```
if (millis() > (actualizacion + IO_LOOP_DELAY)) {
    Serial.print("Enviado -> ");
    Serial.println(contar);
    prueba->save(contar);

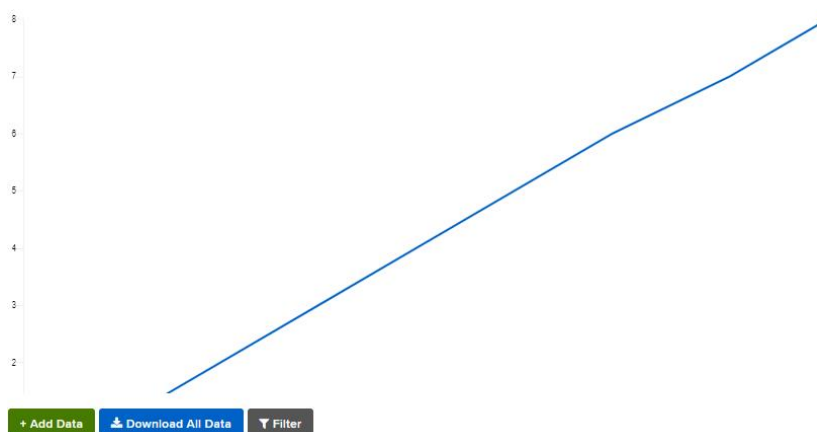
    contar++;

    actualizacion = millis();
}
```

Al comprobar el código observamos lo siguiente:



Podemos visualizar como el número que enviamos es el que recibimos, realizando un contador desde cero.



<div> <div> <div>◀ Prev</div> <div>First</div> </div> <div> <div>page 1 of 1</div> <div>Next ▶</div> </div> </div>		
Created at	Value	Location
2021/11/22 8:45:59PM	8	0, 0, 0
2021/11/22 8:45:54PM	7	0, 0, 0
2021/11/22 8:45:48PM	6	0, 0, 0
2021/11/22 8:45:43PM	5	0, 0, 0
2021/11/22 8:45:38PM	4	0, 0, 0
2021/11/22 8:45:33PM	3	0, 0, 0, 0, 0
2021/11/22 8:45:28PM	2	0, 0, 0, 0, 0
2021/11/22 8:45:23PM	1	0, 0, 0, 0, 0
2021/11/22 8:45:18PM	0	0, 0, 0, 0, 0

Además, si enviamos un número desde el terminal de adafruit, veremos como este lo recibimos en nuestra consola.

COM4

Enviar

Enviado -> 8
Recibido <- 8
Enviado -> 9
Recibido <- 9
Enviado -> 10
Recibido <- 10
Enviado -> 11
Recibido <- 11
Enviado -> 12
Recibido <- 12
Recibido <- 77
Enviado -> 13
Recibido <- 13
Enviado -> 14
Recibido <- 14
Enviado -> 15

☐ Autoscroll

☐ Mostrar marca temporal

Nueva línea

115200 baudio

Limpiar salida

5. Comunicaciones BLE y Bluetooth.

En esta sesión de prácticas se pretende conseguir familiarizarse con las comunicaciones BLE de forma que sea posible leer mensajes de BLE emitidos por un sensor y reportarlos en formato json y emitir mensajes de advertising vía BLE.

5.1. Escanea beacons y reporta en formato JSON-SENML.

En primer lugar, es necesario descargarse la aplicación Beacon Simulator en el móvil para, como su nombre indica, simular uno, de forma que nuestro esp32 sea capaz de detectarlo y decodificar su información en formato json.

Para este caso, el servidor será el móvil puesto que anuncia su existencia, puede ser controlado por otros dispositivos y contiene la información que se desea leer, mientras que el cliente será el esp32, capaz de escanear dispositivos cercanos y, cuando encuentra el servidor que está buscando, lea la información entrante.

Como esta práctica consiste en el manejo de BLE, se incluyen sus librerías. Además, cabe destacar que la información recibida desde el beacon se encuentra en formato hexadecimal, por lo que es necesario convertirla a formato decimal utilizando la variable `ENDIAN_CHANGE_U16`

```
#define ENDIAN_CHANGE_U16(x) (((x)&0xFF00) >> 8) + (((x)&0xFF) << 8))
```

Como en prácticas anteriores, también se crean variables y funciones para el json. En este caso, la función `almacenaDatos` contiene los valores de major, minor, UUID y TxPower. La función `escribeFichero` tendrá el mismo cometido que en apartados anteriores, serializar en formato json.

Siguiendo uno de los ejemplos de Arduino, se crea la clase `MyAdvertisedDeviceCallbacks` donde se define la función `onResult` a la que se le pasa el argumento `advertisedDevice` de tipo `BLEAdvertisedDevice`. En esta función se imprime por pantalla la información de publicidad del dispositivo.

```
class MyAdvertisedDeviceCallbacks : public BLEAdvertisedDeviceCallbacks
{
    void onResult(BLEAdvertisedDevice advertisedDevice)
    {
        Serial.printf("Advertised Device: %s \n", advertisedDevice.toString().c_str());
        if (advertisedDevice.haveManufacturerData() == true)
        {
            std::string strManufacturerData = advertisedDevice.getManufacturerData();

            uint8_t cManufacturerData[100];
            strManufacturerData.copy((char *)cManufacturerData, strManufacturerData.length(), 0);

            if (strManufacturerData.length() == 25 && cManufacturerData[0] == 0x4C && cManufacturerData[1] == 0x00)
            {
                Serial.println("Beacon Encontrado");
                oBeacon.setData(strManufacturerData);
                almacenaDatos();
                Serial.print("Fichero JSON: ");
                escribeFichero();
                Serial.println(datosChar);
            }
        }
        return;
    }
};
```

Cuando los datos publicitarios contienen algún tipo de información (`manufacturerData`), el tamaño es igual a 25, e l primer byte del elemento 4C y el segundo byte es 0, tenemos nuestro BEACON

ENCONTRADO, diferenciándose del resto de dispositivos escaneados. En este momento será cuando utilizemos las funciones de json para almacenar los datos y escribir el fichero en formato json.

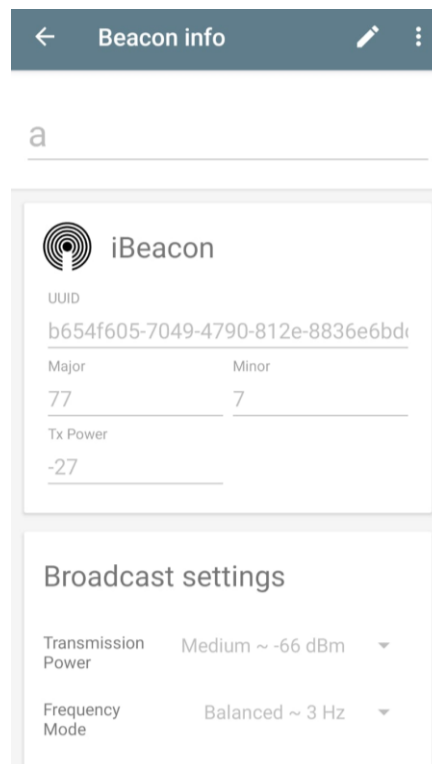
```
void setup()
{
  Serial.begin(115200);
  Serial.println("Scanning...");

  BLEDevice::init("Alba");
  pBLEScan = BLEDevice::getScan(); //create new scan
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
  pBLEScan->setActiveScan(true); //active scan uses more power, but get results faster
  pBLEScan->setInterval(100);
  pBLEScan->setWindow(99); // less or equal setInterval value
}
```

Como siempre, en el setup inicializaremos el puerto serie y nuestro dispositivo BLE, crearemos un escaneo y asignaremos al puntero, pBLEScan, la llamada a la clase que hemos creado, la activación del escaneo, el intervalo de escaneo y la ventana de escaneo.

En el bucle principal únicamente mostraremos por pantalla los resultados obtenidos durante el escaneo.

Para comprobar el código, creamos un beacon:



Antes de activar nuestro beacon, observamos que se detectan otro tipo de advertising, pero el objetivo es que nuestro dispositivo sea capaz de leer únicamente el nuestro.

```
COM4
Enviar
Scanning...
Advertised Device: Name: , Address: 73:f3:f3:db:f3:dd, serviceUUID: cbbfe0e2-f7f3-4206-84e0-84cbb3d09dfc
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 80:47:86:89:71:eb, manufacturer data: 750042040180668047868971eb824786897
Advertised Device: Name: , Address: 8c:79:f5:d2:a1:2f, manufacturer data: 7500420401207e190f000201370000000000
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 80:47:86:89:71:eb, manufacturer data: 750042040180668047868971eb824786897
Advertised Device: Name: , Address: 6f:89:d8:92:68:7b
Advertised Device: Name: Galaxy Watch Active2(1116) LE, Address: d8:9c:2c:0e:62:2c, appearance: 192, manufact
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 80:47:86:89:71:eb, manufacturer data: 750042040180668047868971eb824786897
Advertised Device: Name: , Address: 6f:89:d8:92:68:7b
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 6f:89:d8:92:68:7b
Advertised Device: Name: Galaxy Watch4 (HEFW), Address: 47:3c:d5:39:95:b2, appearance: 192, manufacturer data
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: Galaxy Watch4 (HEFW), Address: 47:3c:d5:39:95:b2, appearance: 192, manufacturer data
Advertised Device: Name: , Address: 80:47:86:89:71:eb, manufacturer data: 750042040180668047868971eb824786897
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 80:47:86:89:71:eb, manufacturer data: 750042040180668047868971eb824786897
Advertised Device: Name: , Address: 6f:89:d8:92:68:7b
Advertised Device: Name: Galaxy Watch Active2(1116) LE, Address: d8:9c:2c:0e:62:2c, appearance: 192, manufact
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Autoscroll Mostrar marca temporal Nueva línea 115200 baudio Limpiar salida
```

Al activar nuestro beacon, observamos que, aunque se detecten otro tipo de advertising, el esp32 es capaz de diferenciar el beacon creado del resto y únicamente lee la información del que nosotros queremos.

```
COM4
Enviar
Scanning...
Advertised Device: Name: , Address: 5d:8e:b4:e1:02:85, manufacturer data: 4c000215b654f60570494790812e8836e6b
Beacon Encontrado
Fichero JSON: [{"major":77,"minor":7,"UUID":"79d7bde6-3688-2e81-9047-497005f654b6","TxPower":-27}]
Advertised Device: Name: Galaxy Watch4 (HEFW), Address: 47:3c:d5:39:95:b2, appearance: 192, manufacturer data
Advertised Device: Name: , Address: 5d:8e:b4:e1:02:85, manufacturer data: 4c000215b654f60570494790812e8836e6b
Beacon Encontrado
Fichero JSON: [{"major":77,"minor":7,"UUID":"79d7bde6-3688-2e81-9047-497005f654b6","TxPower":-27}]
Advertised Device: Name: Galaxy Watch4 (HEFW), Address: 47:3c:d5:39:95:b2, appearance: 192, manufacturer data
Advertised Device: Name: , Address: 5d:8e:b4:e1:02:85, manufacturer data: 4c000215b654f60570494790812e8836e6b
Beacon Encontrado
Fichero JSON: [{"major":77,"minor":7,"UUID":"79d7bde6-3688-2e81-9047-497005f654b6","TxPower":-27}]
Advertised Device: Name: , Address: 6f:89:d8:92:68:7b
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 5d:8e:b4:e1:02:85, manufacturer data: 4c000215b654f60570494790812e8836e6b
Beacon Encontrado
Fichero JSON: [{"major":77,"minor":7,"UUID":"79d7bde6-3688-2e81-9047-497005f654b6","TxPower":-27}]
Advertised Device: Name: , Address: 56:1b:6a:d6:69:97, manufacturer data: 750001124150613036640000000000000000
Advertised Device: Name: , Address: 5d:8e:b4:e1:02:85, manufacturer data: 4c000215b654f60570494790812e8836e6b
Beacon Encontrado
Fichero JSON: [{"major":77,"minor":7,"UUID":"79d7bde6-3688-2e81-9047-497005f654b6","TxPower":-27}]
Advertised Device: Name: , Address: 6f:89:d8:92:68:7b
Advertised Device: Name: , Address: 5d:8e:b4:e1:02:85, manufacturer data: 4c000215b654f60570494790812e8836e6b
Autoscroll Mostrar marca temporal Nueva línea 115200 baudio Limpiar salida
```


5.2. Advertising iBeacon

Este apartado tiene como objetivo generar un anuncio con el esp32 y detectarlo con el móvil utilizando la aplicación nRFconnect. Para este caso, el móvil es el cliente puesto que escanea el dispositivo y el esp32 es el servidor puesto que se anuncia y contiene la información.

```
#define BEACON_UUID "8b21e714-4a01-11ec-81d3-0242ac720251"
```

Se sigue uno de los ejemplos de BLE, definiendo el UUID (Identificador único universal) de forma que acabe con el nip de la universidad, 720251.

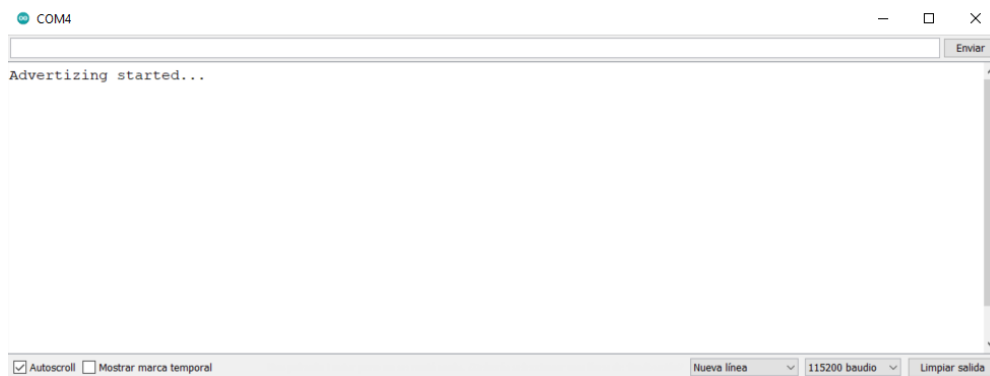
```
void setBeacon() {  
  
    BLEBeacon oBeacon = BLEBeacon();  
    oBeacon.setManufacturerId(0x4C00); // fake Apple 0x004C LSB (ENDI  
    oBeacon.setProximityUUID(BLEUUID(BEACON_UUID));  
    oBeacon.setMajor((bootcount & 0xFFFF0000) >> 16);  
    oBeacon.setMinor(bootcount & 0xFFFF);  
    BLEAdvertisementData oAdvertisementData = BLEAdvertisementData();  
    BLEAdvertisementData oScanResponseData = BLEAdvertisementData();  
  
    oAdvertisementData.setFlags(0x04); // BR_EDR_NOT_SUPPORTED 0x04  
  
    std::string strServiceData = "";  
  
    strServiceData += (char)26; // Len  
    strServiceData += (char)0xFF; // Type  
    strServiceData += oBeacon.getData();  
    oAdvertisementData.addData(strServiceData);  
  
    pAdvertising->setAdvertisementData(oAdvertisementData);  
    pAdvertising->setScanResponseData(oScanResponseData);  
}
```

En la función setBeacon, se configuran las diferentes variables que se desea enviar, la ID como 4C, el UUID como el nuestro propio, mayor y el menor.

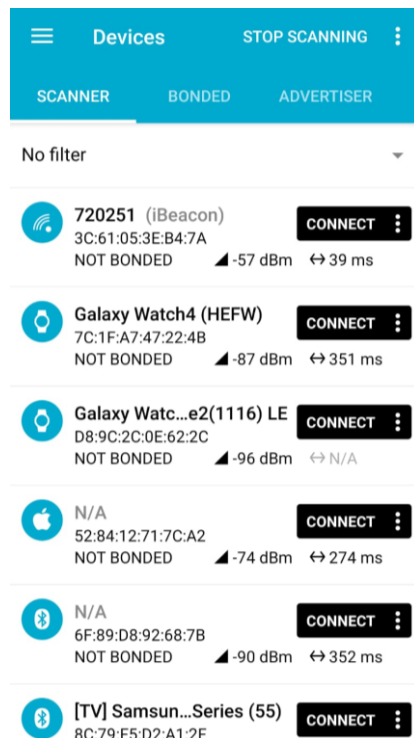
```
void setup() {  
  
    Serial.begin(115200);  
    Serial.printf("start ESP32 %d\n", bootcount++);  
  
    // Create the BLE Device  
    BLEDevice::init("720251");  
    // Create the BLE Server  
    BLEServer *pServer = BLEDevice::createServer();  
    pAdvertising = BLEDevice::getAdvertising();  
    BLEDevice::startAdvertising();  
    setBeacon();  
    // Start advertising  
    pAdvertising->start();  
    Serial.println("Advertisizing started...");  
    delay(100);  
}
```

Finalmente, en el setup se inicializa el puerto serie, el dispositivo Bluetooth, se crea el servidor Bluetooth que es el encargado de anunciar su existencia para poder ser encontrado por otros

dispositivos y contiene los datos que el cliente puede leer. Se configura el anuncio y su start. Se llama a la función setBeacon y se comienza a anunciar.



Tras cargar el código, observamos en el puerto serie como se comienza a anunciar el Beacon y, utilizando la aplicación nRFconnect observamos el dispositivo anunciado.



6. Comunicaciones Lora y LoraWAN

Para esta sesión de prácticas se han adquiridos dos módulos SX1278 LoRa RA-02 que trabajan en la banda de 433MHz.



El objetivo es familiarizarse con este tipo de protocolo con la finalidad de establecer comunicaciones ping-ping entre dos dispositivos, además de enviar datos utilizando el protocolo LoraWAN a un servidor TTN. Para ello, se ha seguido la wiki del siguiente enlace: https://bricolabs.cc/wiki/guides/lora_ttn

6.1. Ping-pong Lora

Para este primer apartado, se conectarán dos módulos LoRa a dos módulos esp32, de forma que los pines quedarán distribuidos de la siguiente forma:

```
#define SX1278_SCK 18    // GPIO18  -- SX1278's SCK
#define SX1278_MISO 19  // GPIO19  -- SX1278's MISO//SPIQ//SDO
#define SX1278_MOSI 23  // GPIO23  -- SX1278's MOSI//SPID//SDI
#define SX1278_CS 5     // GPIO5   -- SX1278's CS
#define SX1278_RST 14   // GPIO14  -- SX1278's RESET
#define SX1278_DIO 26   // GPIO26  -- SX1278's IRQ(Interrupt Request)

#define LORA_BAND 433E6
```

Además, es necesario definir la banda que utiliza el dispositivo, en este caso 433MHz.

```
void loop() {
  if (millis() - lastSendTime > interval) {
    sendMessage(message);
    lastSendTime = millis();
    interval = random(2000) + 1000;
    LoRa.receive();
  }
  int packetSize = LoRa.parsePacket();
  if (packetSize) {
    onReceive(packetSize);
  }
}
```

Se crean dos funciones, que son llamadas en el bucle principal, la primera de ellas envía el mensaje siempre que el intervalo fijado en la configuración sea menor a la diferencia entre el millis actual y el último millis utilizado para el envío de los datos. La segunda de ellas recibe la información del otro dispositivo, comprueba si la longitud es la adecuada y si está recibiendo la información del dispositivo correcto, mostrando los parámetros necesarios por pantalla.

```

void sendMessage(String outgoing) {
    LoRa.beginPacket();           // Inicio
    LoRa.write(destination);      // Escribir destino
    LoRa.write(localAddress);     // Escribir dirección dispositivo
    LoRa.write(msgCount);         // Escribir contador mensaje
    LoRa.write(outgoing.length()); // Longitud del mensaje saliente
    LoRa.print(outgoing);         // Escribir mensaje saliente
    LoRa.endPacket();             // Finalizar paquete y enviar

    Serial.println("Sending message " + String(msgCount) + " to address: " + String(destination));
    Serial.println("Message: " + message);
    Serial.println();
    delay(1000);
    msgCount++;
}

```

Si observamos con detenimiento la primera función, sendMessage, observamos como se inicia el paquete de LoRa, escribiendo la dirección de destino, la dirección local, contador de mensaje, la longitud del mensaje saliente y el mensaje saliente. Además, esto se muestra por pantalla.

```

void onReceive(int packetSize) {
    if (packetSize == 0) return;    // Si no hay paquete, se re

    // Lectura de bytes del paquete
    int recipient = LoRa.read();    // Dirección del recipiente
    byte sender = LoRa.read();      // Dirección del que envía
    byte incomingMsgId = LoRa.read();
    byte incomingLength = LoRa.read();

    String incoming = "";

    while (LoRa.available()) {
        incoming += (char)LoRa.read();
    }

    if (incomingLength != incoming.length()) { //Se realiza un check
        Serial.println("error: message length does not match length");
        incoming = "message length error";
        return;
    }

    // Se realiza un check para comprobar si el destinatario no es este
    if (recipient != localAddress && recipient != 0xFF) {
        Serial.println("This message is not for me.");
        incoming = "message is not for me";
        return; // skip rest of function
    }

    // Si el mensaje es para este dispositivo, se escribe por pantalla
    Serial.println("Received from: 0x" + String(sender, HEX));
    Serial.println("Sent to: 0x" + String(recipient, HEX));
    Serial.println("Message ID: " + String(incomingMsgId));
    Serial.println("Message length: " + String(incomingLength));
    Serial.println("Message: " + incoming);
    Serial.println("RSSI: " + String(LoRa.packetRssi()));
    Serial.println("Snr: " + String(LoRa.packetSnr()));
    Serial.println();
    delay(1000);
}

```

En la segunda función, leemos los bytes del paquete y comprobamos si recibimos información de algún paquete por medio de LoRa. Además, chequeamos la longitud del paquete y si el destinatario es el correcto. En caso de ser correcto, se escriben una serie de parámetros para comprobar el envío.

Para comprobar el correcto funcionamiento de la práctica, se han utilizado dos esp32 con dos módulos LoRa, cada uno configurado de una forma distinta.

```

////////////////////////CONFIG 1////////////////////////////////////////
byte localAddress = 8;      // Dirección dispositivo
byte destination = 18;     // Dirección de envío
int interval = 3000;       // Intervalo entre envíos
String message = "¡HOLA!"; // Mensaje
////////////////////////

|

////////////////////////CONFIG 2////////////////////////////////////////
byte localAddress = 18;    // Dirección dispositivo
byte destination = 8;     // Dirección de envío
int interval = 2000;      // Intervalo entre envíos
String message = "¿QUE TAL?"; // Mensaje
////////////////////////

```

Utilizando uno de ellos para visualizar por pantalla y el otro alimentado mediante una power bank se observa como se ha iniciado adecuadamente el dispositivo LoRa.

En este caso, el dispositivo envía el mensaje ¡HOLA! a la dirección 18 y recibe de 0x12 (dirección 18 en hexadecimal) el mensaje ¿QUE TAL?

```

LoRa init succeeded.
Sending message 0 to address: 18
Message: ¡HOLA!

```

```

Received from: 0x12
Sent to: 0x8
Message ID: 0
Message length: 10
Message: ¿QUE TAL?
RSSI: -117
Snr: 1.75

```

```

Sending message 1 to address: 18
Message: ¡HOLA!

```

```

Received from: 0x12
Sent to: 0x8
Message ID: 1
Message length: 10
Message: ¿QUE TAL?
RSSI: -117
Snr: 2.00

```

```

Sending message 2 to address: 18
Message: ¡HOLA!

```

```

Received from: 0x12
Sent to: 0x8
Message ID: 2
Message length: 10
Message: ¿QUE TAL?
RSSI: -117
Snr: 1.75

```

6.2. Envío datos a TTN

La finalidad de esta práctica es conseguir enviar información a un servidor TTN por medio del protocolo LoRaWAN.

Como los módulos LoRa adquiridos trabajan en la banda de 433MHz es necesario construir, en primer lugar, un Gateway.

Para ello se van a utilizar varias guías, entre las que podemos destacar <https://learn.sparkfun.com/tutorials/esp32-lora-1-ch-gateway-lorawan-and-the-things-network/all> y <https://www.hackster.io/Arn/single-channel-ttn-lora-gateway-and-nodes-with-esp32-sx1276-709612>.

Debemos descargar el código de ejemplo de puerta de enlace de un solo canal desde el repositorio de github que se muestra a continuación: <https://github.com/things4u/ESP-1ch-Gateway-v5.0--OLD>

Antes de cargar el código de ejemplo, se deben realizar algunas modificaciones como son: banda de frecuencia de trabajo, configuración de los pines de SPI, configuración WiFi y configuración de los pines del modem de lora.

Para ello, abrimos el archivo ESP-sc-gway.h y modificamos:

```
44 // Define the LoRa Frequency band that is used. TTN Supported values are 915MHz, 868MHz and 433MHz.
45 // So supported values are: 433 868 915
46 #define _LFREQ 433

81 // We support a few pin-out configurations out-of-the-box: HALLARD, COMPRESULT and TTGO ESP32.
82 // If you use one of these two, just set the parameter to the right value.
83 // If your pin definitions are different, update the loraModem.h file to reflect these settings.
84 // 1: HALLARD
85 // 2: COMRESULT pin out
86 // 3: ESP32 Wemos pin out
87 // 4: ESP32 TTGO pinning (should work for 433 and OLED too).
88 // 5: ESP32 TTGO EU433 MHz with OLED
89 // 6: Other, define your own in loraModem.h
90 #define _PIN_OUT 6

194 // Gateway Ident definitions
195 #define _DESCRIPTION "ESP Gateway Alba" // Name of the gateway
196 #define _EMAIL "720251@unizar.es" // Owner
197 #define _PLATFORM "ESP32"
198 #define _LAT 52.0
199 #define _LON 41.6399200
200 #define _ALT 212 // Altitude

261 wpa_s wpa[] = {
262   { "", "" }, // Reserved for WiFi Manager
263   { "vodafoneBA1157", "SRULAGD6RHFQ4M5K" },
264   { "Mia2", "25208230t" }
265 };
```

A continuación, abrimos loraModem.h y modificamos:

```

239 // -----
240 // Use your own pin definitions, and comment #error line below
241 // MISO 12 / D6
242 // MOSI 13 / D7
243 // CLK 14 / D5
244 // SS 16 / D0
245 struct pins {
246     uint8_t dio0 = 26;
247     uint8_t dio1 = 33;
248     uint8_t dio2 = 32;
249     uint8_t ss = 5;
250     uint8_t rst = 27; // Reset not used
251 } pins;
252 #define SCK 18
253 #define MISO 19
254 #define MOSI 23
255 #define SS 5
256 #define DIO0 26
257

```

Al subir el código a Arduino, abrimos la consola y observamos lo siguiente:

```

COM4

ESP32 defined, freq=433175000 EU433
ARDUINO_ARCH_ESP32 defined
SPIFFS init success
Assert=Do Asserts
debug=1
readConfig:: Starting
.SSID=  vodafoneBA1157
.PASS=  SRULAGD6RHFQ4M5K
.CH=  0
.SF=  9
.FCNT=  0
.DEBUG= 1
.PDEBUG=64
.CAD=1
.HOP=0
.NODE=  0
.BOOT= 2
.RESETS= 0
.WIFIS= 2
.VIEWS= 0
.REFR= 0
.REENTS= 0
.NTPETIM= 0
.NTPERR= 0
.NTPS= 2
.FILERECD= 0
.FILENO= 0
.FILENUM= 0
.EXPERT= 0
#
MAC: 78:e3:6d:0a:c6:50, len=17
WlanConnect:: Init para 0
0:1:3. WiFi connect SSID=vodafoneBA1157, pass=SRULAGD6RHFQ4M5K
A WlanStatus:: CONNECTED to vodafoneBA1157
Host esp32-0ac650 WiFi Connected to vodafoneBA1157 on IP=192.168.0.20
Local UDP port=1700
Connection successful
Gateway ID: 78E36DFFFF0AC650, Listening at SF9 on 433.18 Mhz.
setupOta:: Started
Ready
IP address: 192.168.0.20
Time: Friday 13:20:23
Gateway configuration saved
WWW Server started on port 80
-----
A readUdp:: NTP msg rcvd

```

Si nos conectamos a la IP que nos indica (192.168.0.20) podremos visualizar el portal web ESP Gateway config. Desde esta página es posible controlar los mensajes que llegan, las frecuencias y los factores de propagación. Además, es posible cambiar la configuración de la puerta de enlace, ajustar el canal y el factor de dispersión, entre otras posibilidades.

ESP Gateway Config

Version: V3.3.0 (08/02/2023)
 ESP Gateway Config: V3.3.0 (08/02/2023) (https://github.com/ESP8266/ESP8266)
 Contact: email:info@esp8266.com Tel: +34 912 022 13 02

[Documentation](#) | [Expert Mode](#) | [Log Files](#)

Package Statistics

Counter	C 0	C 1	C 2	Plp	Plp/s
Packages Download				0	0
Packages Upload Total				0	0
Packages Upload OK				0	0 %
SF revd	0	0	0	0	0 %
SF8 revd	0	0	0	0	0 %
SF9 revd	0	0	0	0	0 %
SF10 revd	0	0	0	0	0 %
SF11 revd	0	0	0	0	0 %
SF12 revd	0	0	0	0	0 %

Message History

Time	Note	C	Freq	SF	pRSSI
------	------	---	------	----	-------

Gateway Settings

Setting	Value	Set
CAD	ON	ON OFF
BOP	OFF	ON OFF
SF Setting	AUTO	-
Channel	0	-
Dispersal level	0	-
Dispersal pattern	SCN	CAD RX TX
USB Debug	PRE	MAI GUI RDO
WiFi/Bluetooth	OFF	ON OFF
Update Firmware		UPDATE
Format SPIFFS		FORMAT
Reboot	0	REBOOT
Reset and Reboot	0	RESET

Una vez tenemos nuestra puerta de enlace, debemos generar un código capaz de comunicarse con este Gateway para que envíe la información a la nube.

Para ello se debe hacer uso de las librerías siguientes:

```
#include <lmic.h>
#include <hal/hal.h>
#include <SPI.h>
```

Es necesario señalar que, como se está trabajando en la banda de 433 MHz hay que modificar la librería Imic para que trabaje en esta banda. Modificamos el archivo lorabase.h de la siguiente forma:

```
67 // Default frequency plan for EU 868MHz ISM band
68 // Bands:
69 // g1 : 1% 14dBm
70 // g2 : 0.1% 14dBm
71 // g3 : 10% 27dBm
72 //
73 //          freq          band      datarates
74 /*enum { EU868_F1 = 868100000, // g1 SF7-12
75          EU868_F2 = 868300000, // g1 SF7-12 FSK SF7/250
76          EU868_F3 = 868500000, // g1 SF7-12
77          EU868_F4 = 868850000, // g2 SF7-12
78          EU868_F5 = 869050000, // g2 SF7-12
79          EU868_F6 = 869525000, // g3 SF7-12
80          EU868_J4 = 864100000, // g2 SF7-12 used during join
81          EU868_J5 = 864300000, // g2 SF7-12 ditto
82          EU868_J6 = 864500000, // g2 SF7-12 ditto
83          };
84 enum { EU868_FREQ_MIN = 863000000,
85        EU868_FREQ_MAX = 870000000 };
86 */
87 //Modify the EU868Mhz band configure for EU433Mhz temporary use.
88 /*enum { EU868_F1 = 433175000, // g1 SF7-12
89          EU868_F2 = 433375000, // g1 SF7-12 FSK SF7/250
90          EU868_F3 = 433575000, // g1 SF7-12
91          EU868_F4 = 433775000, // g2 SF7-12
92          EU868_F5 = 433975000, // g2 SF7-12
93          EU868_F6 = 434175000, // g3 SF7-12
94          EU868_J4 = 434375000, // g2 SF7-12 used during join
95          EU868_J5 = 434575000, // g2 SF7-12 ditto
96          EU868_J6 = 434775000, // g2 SF7-12 ditto
97          };
98 enum { EU868_FREQ_MIN = 433000000,
99        EU868_FREQ_MAX = 435000000 };
100 */
```


Y descargamos el código de ejemplo siguiente:
https://cdn.sparkfun.com/assets/learn_tutorials/8/0/4/ESP-1CH-TTN-Device-ABP-v01.zip

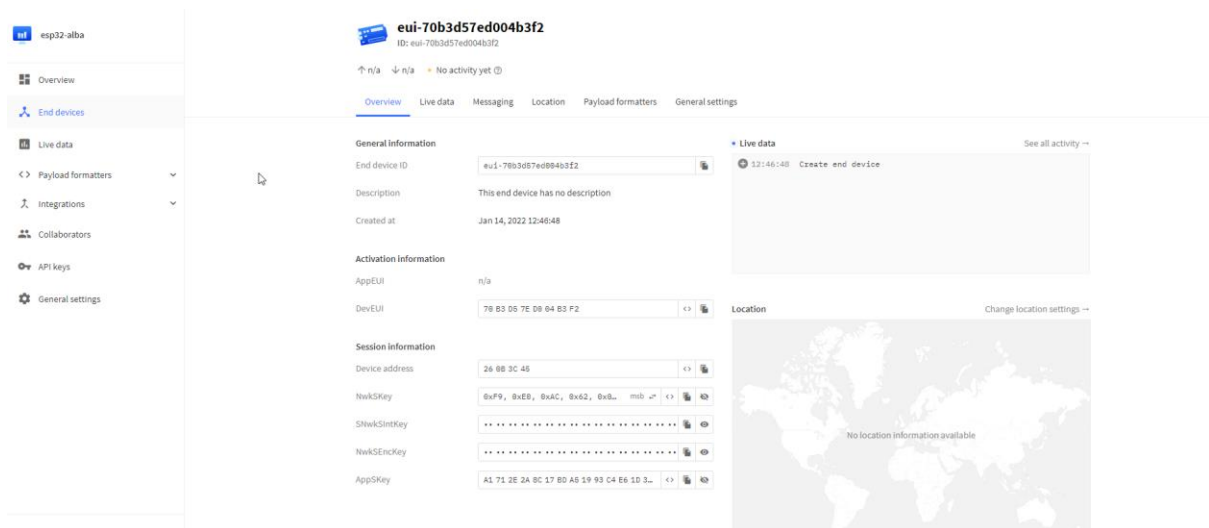
Además, debemos modificar el pineado del dispositivo lora:

```
62 // Pin mapping for the SparkX ESP32 LoRa 1-CH Gateway
63 const lmic_pinmap lmic_pins = {
64     .nss = 5,
65     .rxtx = LMIC_UNUSED_PIN,
66     .rst = 27,
67     .dio = {26, 33, 32},
68 };
```

Y convertirlo a un único canal:

```
181 /*#if defined(CFG_eu868) // EU channel setup
182 // Set up the channel used by the Things Network and compatible with
183 // our gateway.
184 // Setting up channels should happen after LMIC_setSession, as that
185 // configures the minimal channel set.
186 LMIC_setupChannel(0, 433175000, DR_RANGE_MAP(DR_SF12, DR_SF7), BAND_CENTI); // g-band
187 */
188 #if defined(CFG_eu868) // EU channel setup
189     int channel = 0;
190     int dr = DR_SF9;
191     for(int i=0; i<9; i++) {
192         if(i != channel) {
193             LMIC_disableChannel(i);
194         }
195     }
196 // Set data rate (SF) and transmit power for uplink
197 LMIC_setDrTxpow(dr, 14);
198 ...
```

Finalmente, debemos configurar la aplicación en TTN y agregar las claves requeridas en el nodo.



```
39 // LoRaWAN NwkSKey, network session key
40 // This is the default Semtech key, which is used by the early prototype TTN
41 // network.
42 static const PROGMEM u1_t NWKKEY[16] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6, 0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C };
43
44 // LoRaWAN AppSKey, application session key
45 // This is the default Semtech key, which is used by the early prototype TTN
46 // network.
47 static const u1_t PROGMEM APPKEY[16] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6, 0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C };
48
49 // LoRaWAN end-device address (DevAddr)
50 static const u4_t DEVADDR = 0x260B3C45;
```



```
// Transmit data from mydata
void do_send(osjob_t* j) {
    // Check if there is not a current TX/RX job running
    if (LMIC.opmode & OP_TXRXPEND) {
        Serial.println(F("OP_TXRXPEND, not sending"));
    } else {
        digitalWrite(LED_BUILTIN, HIGH); // Turn on LED while sending
        // Prepare upstream data transmission at the next possible time.
        LMIC_setTxData2(1, mydata, sizeof(mydata) - 1, 0);
        Serial.println(F("Packet queued")); // Paquete en cola
    }
}
```

Al iniciar, observamos que el paquete está en la cola para enviarse, pero al pulsar el botón aparece el mensaje de no enviado.

[illegible]