UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

MONITORIZACIÓN DE PARÁMETROS MEDIOAMBIENTALES CON SENSORES DE BAJO COSTE MEDIANTE APLICACIÓN ANDROID

TRABAJO FIN DE GRADO

Septiembre - 2022

AUTOR: Guillermo Cortés Orellana

DIRECTOR/ES: Miguel Onofre Martínez Rach

Salvador Hurtado Alia

RESUMEN

En los últimos años, los avances en el desarrollo de sensores y de diferentes microprocesadores, han habilitado nuevos métodos de trabajo, entre los que destaca el *Internet of Things*, en castellano, “internet de las cosas”.

Gracias a este nuevo concepto, es posible llevar a cabo la interconexión de objetos cotidianos con internet. Es, en definitiva, la conexión de internet más con objetos que con personas.

Los modelos IoT están presentes hoy en día en innumerables sectores, tales que: mantenimiento predictivo, control de calidad, movilidad, gestión y monitoreo automatizada y remota de los equipos etc. Todo ello se consigue haciendo uso de software avanzado (aplicaciones móviles) y diferentes chips electrónicos (como el ESP32).

Es en el monitoreo de equipos donde se encuentra ubicado el proyecto en cuestión, tratando de llevar a cabo un seguimiento y observación del medio ambiente a través de la interconexión de diferentes sensores con la tecnología anteriormente descrita e internet.

Por esa razón, como Trabajo de Fin de Grado (TFG), se ha realizado un proyecto focalizado en monitorizar datos de diferentes sensores (temperatura, humedad y distancia a objetos) conectados a un módulo ESP32, el cual, a su vez, los enviará haciendo uso de diferentes tecnologías inalámbricas (BLE y WiFi) a un smartphone para su posterior almacenamiento y representación haciendo uso de una aplicación Android diseñada específicamente para el proyecto:

PALABRAS CLAVE

- BLE
- Access Point
- Java
- C++
- HTML
- Android
- Arduino
- ESP32
- Sensores

ABSTRACT

In recent years, advances in the development of sensors and different microprocessors have enabled new working methods, among which the IoT (Internet of Things).

Thanks to this new concept, it is possible to carry out the interconnection of everyday objects with the Internet. It is, in short, the internet connection more with objects than with people.

IoT are present today in countless sectors, such as: predictive maintenance, control of quality models, mobility, automated and remote management and monitoring of equipment, etc. All this is achieved by using advanced software (mobile applications) and different electronic chips (such as ESP32).

It is in the monitoring of equipment where the project in question is located, trying to carry out monitoring and observation of the environment through the interconnection of different sensors with the technology described above and the internet.

For this reason, as a Final Degree Project, a project has been carried out focused on monitoring data from different sensors (temperature, humidity and distance to objects) connected to an ESP32 module, which, in turn, making use of different wireless technologies (BLE and WiFi) to a smartphone for later storage and representation using an-Android application specifically designed for the project:

KEYWORDS

- BLE
- Access Point
- Java
- C++
- HTML
- Android
- Arduino
- ESP32
- Sensors

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

En este primer capítulo se llevará a cabo la exposición del contexto y objetivos del proyecto, así como la memoria de este.

1.1. CONTEXTO

Actualmente nos encontramos inmersos en plena revolución tecnológica, donde, el desarrollo de diferentes técnicas y herramientas, unidas al uso cada vez más común de ciertas plataformas de desarrollo, están causando una transformación sin precedentes en el mundo tal y como lo conocemos. Además, el crecimiento del uso de diferentes elementos hardware, como sensores y microcontroladores, en consonancia con elementos software, como aplicaciones móviles, nos permiten llevar a cabo tareas de una forma más rápida y eficiente que utilizando modelos anteriores. Es en esta cooperación entre elementos hardware y software donde podemos situar al IoT.

Esta tecnología ha cambiado radicalmente la manera en la que nos comunicamos con los elementos físicos de nuestro entorno, permitiéndonos mejorar la gestión, control, mantenimiento y monitoreo de estos, a través de elementos hardware y software.

Son muchos los usos que se le pueden dar: desde amplios proyectos industriales (‘Industria 4.0’), hasta pequeños proyectos personales, como el control de un sistema de regadío, controles de seguridad, etc.

Por ello, podríamos situar este TFG en un punto intermedio entre un proyecto personal, el cual nos permita llevar a cabo una control y monitorización de diferentes parámetros de nuestro alrededor, y un proyecto más masificado, ya que se desarrollará un sistema totalmente escalable cuyo límite lo pondremos nosotros mismos.

Para cubrir esta necesidad, el presente proyecto tiene la pretensión de desarrollar un sistema que permita la comunicación y monitoreo en tiempo real entre los diferentes componentes del sistema y el usuario.

1.2. OBJETIVOS

El principal objetivo del presente proyecto es el desarrollo de un sistema hardware que permitirá captar valores de temperatura, humedad y distancia a objetos en el medio, enviándolos a una aplicación móvil para su posterior monitoreo, permitiendo así, captar dichos valores en tiempo real. Para una mayor claridad separaremos el objetivo principal en dos objetivos más concretos. Éstos serán: el desarrollo hardware y el desarrollo software.

El primer objetivo, equivalente al desarrollo hardware, consiste en la realización de un sistema compuesto por 3 sensores y un microprocesador, que permitirán en su conjunto captar diferentes valores del medio y enviarlos vía BLE o WiFi.

El segundo objetivo, correspondiente al desarrollo software, consistirá en el desarrollo de una aplicación móvil, en el entorno de desarrollo de Android, utilizada para monitorizar los valores obtenidos en el desarrollo hardware anterior.

Implícitamente, este Trabajo de Fin de Grado también tiene el objetivo de aplicar los conocimientos y aptitudes adquiridos a lo largo del Grado en Ingeniería en Tecnologías de Telecomunicación y aprender nuevas tecnologías que son ampliamente usadas por las empresas de todo el mundo para desarrollar aplicaciones móviles.

1.3. ESTRUCTURA DE LA MEMORIA

En este apartado se detallarán los objetivos que pretende cubrir el sistema descrito en la memoria:

- **Introducción:** descripción general del proyecto. Objetivos.
- **Estado del arte**: soluciones y tecnologías similares en la actualidad.
- **Análisis del proyecto**: definición y requisitos del proyecto, así como casos de uso de la aplicación.
- **Arquitectura de la solución**: composición general del sistema, así como de sus elementos hardware y software.
- **Implementación**: explicación del sistema hardware y de la aplicación software desarrollados.
- **Resultados**: funcionalidades principales del proyecto.
- **Conclusiones**: consecuencias obtenidas al elaborar el proyecto, así como posibles mejoras.
- **Bibliografía**: referencias utilizadas para la elaboración del proyecto.

2. ESTADO DEL ARTE

En este segundo capítulo se pretende analizar soluciones similares a las desarrolladas en el proyecto, disponibles en la industria. Además, también se llevará a cabo un análisis de diferentes tecnologías hardware y software disponibles para la elaboración de proyectos similares.

2.1. SOLUCIONES SIMILARES EN LA INDUSTRIA

Actualmente existen diferentes sistemas y aplicaciones que permiten lograr una funcionalidad como la que se describe en este proyecto.

Por una parte, podríamos destacar diferentes sistemas hardware presentes en la actualidad con diferentes sensores en su haber, que permiten captar determinadas magnitudes de nuestro entorno como, por ejemplo, las denominadas estaciones meteorológicas.

Uno de los modelos que más se asemeja a las funcionalidades desarrolladas en nuestro sistema, y de mayor reconocimiento en la actualidad, es el conocido **ThermoPro TP49**, caracterizado por la medida de temperatura y humedad en interiores.

Ilustración 1. ThermoPro TP49

Dicho dispositivo permite medir temperaturas y humedades en interiores, en grados Celsius y Fahrenheit con bastante precisión, haciéndolo idóneo en la utilización de lecturas fluctuantes como en un invernadero. Además, permite actualizar dichos valores cada 10 segundos y mostrarlos en su pantalla LCD. [1]

Otro de los dispositivos que posee una funcionalidad similar a la utilizada en el proyecto, es el conocido como telémetro digital, capaz de medir la distancia un objeto haciendo uso de la luz reflejada por un láser incorporado.

Actualmente existen diferentes maneras de medir la distancia a un objeto de manera digital. Sin embargo, uno de los modelos más utilizados, y de mayor reconocimiento en la actualidad, es el conocido **DATAPE DT50**, caracterizado por la medida de distancia a un objeto haciendo uso de la técnica anteriormente mencionada. [2]

Ilustración 2. DATAPE DT50

Dicho dispositivo mide la distancia a un objeto utilizando la luz reflejada. Funciona de forma muy parecida a un radar, pero utiliza láseres en lugar de ondas de radio. Una vez captada la magnitud, muestra la distancia medida en diferentes unidades, además de indicar diferentes medidas al mismo tiempo.

Caracterizado por su alta precisión, seguridad e interfaz intuitiva, el DATAPE DT50 es una de las mejores alternativas disponibles en la industria en cuanto a medidas digitales se refiere.

Utilizados en múltiples escenarios, destaca su uso habitual en los campos de golf, donde pueden montarse en carros o dispositivos manuales para que los jugadores puedan medir las distancias desde su ubicación actual a varios puntos de su entorno.

Por otra parte, existen múltiples aplicaciones móviles que nos permiten establecer una conexión con dispositivos haciendo uso del *Bluetooth Low Energy* (BLE), en castellano, “Bluetooth de baja energía”.

Una de las aplicaciones más conocidas para llevar a cabo una comunicación BLE, es la denominada **nRF Connect**, utilizada principalmente para leer valores de un dispositivo, haciendo uso de la tecnología inalámbrica anteriormente mencionada.

Ilustración 3. Logo aplicación nRF Connect

Dicha aplicación, se caracteriza por permitir llevar a cabo un escaneo y conexión con dispositivos, haciendo uso del BLE. Sin embargo, a diferencia del presente proyecto, no permite leer de manera secuencial los valores recibidos del dispositivo externo. [3]

Otra de las aplicaciones utilizadas para la recepción en dispositivos de valores captados por un sistema, es **Bluetooth Terminal**, utilizada para el envío y recepción de datos haciendo uso del puerto serie.

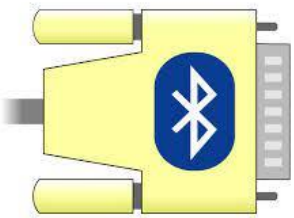


Ilustración 4. Logo aplicación Serial Bluetooth Terminal

Dicha aplicación, se caracteriza por permitir llevar a cabo un intercambio de información haciendo uso del puerto serie, mostrando por pantalla de manera secuencial los datos recibidos, pero sin hacer uso de una tecnología inalámbrica. [4]

2.2. TECNOLOGÍAS DE DESARROLLO ACTUALES

Actualmente existen numerosas tecnologías que permitirían desarrollar una solución como la que se propone en este proyecto. Para explicar las más populares, separaremos este subapartado en dos partes. El primero tratará de explicar las tecnologías más eficientes para el desarrollo de la parte hardware del sistema. El segundo, por su parte, mostrará las tecnologías actuales para desarrollar la parte software, relacionado con la aplicación móvil.

2.2.1. TECNOLOGÍAS HARDWARE

Se llevará a cabo un amplio análisis de las tecnologías hardware disponibles, centrándonos en los microprocesadores y sensores en la actualidad.

2.2.1.A. SENSORES

Un sensor es un dispositivo capaz de detectar magnitudes físicas o químicas, llamadas variables de instrumentación, y transformarlas en variables eléctricas.

Las variables de instrumentación pueden ser, por ejemplo: temperatura, intensidad lumínica, distancia, aceleración, inclinación, desplazamiento, presión, fuerza, torsión, humedad, movimiento, pH, etc.

Una magnitud eléctrica puede ser una resistencia eléctrica (como en una RTD), una capacidad eléctrica (como en un sensor de humedad o un sensor capacitivo), una tensión eléctrica (como en un termopar), una corriente eléctrica (como en un fototransistor), etc. [5]

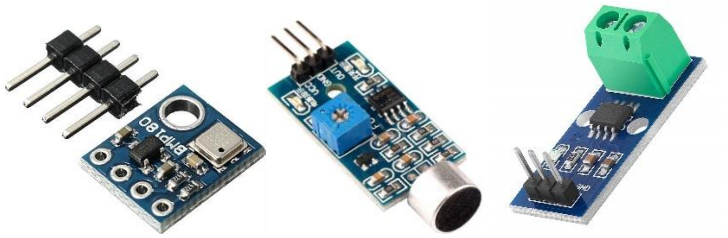


Ilustración 5. Ejemplo de sensores electrónicos de bajo coste

La inmensa mayoría de ellos suelen ser de dimensiones reducidas, y pueden clasificarse en función de los datos ofrecidos en su salida como:

- Digitales
- Analógicos

Los sensores digitales nos ofrecen una señal digital con dos estados como una salida de contacto libre de tensión o una salida en bus digital. Por su parte, los sensores analógicos, ofrecen en su salida señales de naturaleza analógica, formadas por un campo de valores instantáneos que varían en el tiempo, proporcionales a los efectos que se están midiendo.

A la hora de elegir un sensor para conectar en nuestro sistema, se debe tener en cuenta los valores que puede leer las entradas analógicas o digitales de la placa para poder conectarlo.

Por su parte, se debe verificar cómo leer el sensor mediante la programación, comprobar si existe una librería o es posible leerlo con los métodos disponibles de lectura de entrada analógica o digital.

Finalmente, se verificará cómo alimentar el sensor y comprobar si se puede hacer desde el propio sistema o desde una fuente exterior. Además, en función del número de sensores que queramos conectar es posible que el módulo utilizado no pueda alimentar todos.

A la hora de elegir un sensor a utilizar en un sistema, es preferible comprobar una serie de características que definen a este. En general, las más comunes son las siguientes:

- **Rango de medida**: dominio en la magnitud medida en el que puede aplicarse el sensor.
- **Precisión**: error de medida máximo esperado.
- **Offset o desviación de cero**: valor de la variable de salida cuando la variable de entrada es nula.
- **Sensibilidad**: suponiendo que es de entrada a salida y la variación de la magnitud de entrada.
- **Resolución**: mínima variación de la magnitud de entrada que puede detectarse a la salida.
- **Rapidez de respuesta**: puede ser un tiempo fijo o depender de cuánto varíe la magnitud a medir. Depende de la capacidad del sistema para seguir las variaciones de la magnitud de entrada.
- **Derivas**: son otras magnitudes, aparte de la medida como magnitud de entrada, que influyen en la variable de salida. Por ejemplo, pueden ser condiciones ambientales, como la humedad, la temperatura u otras como el envejecimiento (oxidación, desgaste, etc.) del sensor.
- **Repetitividad**: error esperado al repetir varias veces la misma medida.

2.2.1.B. SOC

Hablamos de un *System on a Chip*, SoC, en castellano, “sistema en chip”, para referirnos a un circuito integrado que incluye dentro de sí todo un sistema electrónico o informático. Es, como su nombre lo indica, un sistema completo que funciona integrado en un solo chip.

Los componentes que un SoC busca incorporar dentro de sí, incluyen, por lo general, una unidad central de procesamiento (CPU), puertos de entrada y salida, memoria interna, así como bloques de entrada y salida analógica. [6]

En la actualidad existe una gran variedad de dispositivos utilizados para proyectos IoT. Sin embargo, dos de los más utilizados en la actualidad, son el **ESP8266** y el **ESP32**.

ESP8266

Se trata un chip de bajo costo WiFi con un stack TCP/IP completo y un microcontrolador, fabricado por Espressif. [7], [8]

Ilustración 6. ESP8266

El ESP8266 normalmente viene integrado en un módulo. Esto es debido a que no tiene memoria Flash integrada.

Con un procesador Tensilica Xtensa LX106 a 80 MHz, RAM de instrucciones de 64 KB y de datos de 96 KB, 16 pines GPIO y un conversor ADC de 10 bit, hacen del ESP8266 como una de las opciones más recomendadas a elegir para llevar a cabo proyectos IoT de no muy alta complejidad.

ESP32

Se trata de un chip de bajo costo y consumo de energía, con tecnología Wi-Fi y Bluetooth de modo dual integrada.

Ilustración 7. ESP32

Definido como el sucesor del ESP8266, será el utilizado en el presente proyecto y cuyas características serán analizadas posteriormente en mayor profundidad.

2.2.2. TECNOLOGÍAS SOFTWARE

Se llevará a cabo un amplio análisis de las tecnologías software disponibles, centrándonos en los diferentes lenguajes de programación más utilizados para el desarrollo de aplicaciones Android, implementación de bases de datos locales y entornos de desarrollo.

2.2.2.A. LENGUAJES DE PROGRAMACIÓN

Actualmente existen numerosos lenguajes de programación empleados en el desarrollo de aplicaciones móviles. Sin embargo, a la hora de llevar a cabo la creación de una aplicación en la plataforma Android, podemos destacar dos: **Java** y **Kotlin**.

Java

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. [10]

Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir, que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

Ilustración 8. Icono de Java

Las características principales de Java son:

- Orientado a objetos.
- Multiplataforma.
- Compatible con librerías estándar y editores.
- Seguro.
- Lenguaje de código abierto.
- Cuenta con liberación de memoria.
- Admite subprocesos múltiples.
- Permite crear aplicaciones distribuidas.

La parte correspondiente con el diseño de la aplicación Android del proyecto ha sido programada íntegramente con este lenguaje de programación, el cual, a su vez, ha sido uno de los más utilizados a lo largo de la carrera del estudiante.

Kotlin

Kotlin es un lenguaje de programación estático, gratuito, de código abierto y de propósito general. Se trata de un sistema que combina características de programación funcional y programación orientada a objetos, el cual, trata de reducir la cantidad de código repetitivo como sucede con otros lenguajes. [11]

Ilustración 9. Icono de Kotlin

Las características principales de Kotlin son:

- Orientado a objetos.
- Multiplataforma.
- Corrutinas optimizan la programación asíncrona.
- Más seguro que sus predecesores.
- Permite escribir menos código para realizar la misma tarea.
- Fácilmente adaptable.
- Compatible con Java.

Podría decirse que se trata del lenguaje de programación de aplicaciones Android por excelencia en la actualidad.

Por su parte, podríamos destacar el lenguaje C++ como uno de los más utilizados en lo relacionado a la programación de microprocesadores.

C++

C++ es un lenguaje de programación compilado, multiparadigma, principalmente de tipo imperativo y orientado a objetos, incluyendo también programación genérica y funcional. [12]

Ilustración 10. Icono de C++

Las características principales de C++ son:

- Orientado a objetos
- Alto rendimiento
- Lenguaje actualizado
- Multiplataforma
- Rápido
- Didáctico

El lenguaje C++, es ampliamente utilizado a la hora de programar microcontroladores y otros dispositivos electrónicos.

La parte correspondiente con la programación del microprocesador del proyecto ha sido programada íntegramente con este lenguaje de programación, el cual, a su vez, ha sido uno de los más utilizados a lo largo de la carrera del estudiante.

Finalmente, a la hora de llevar a cabo cualquier tipo de desarrollo web, podemos destacar el **HTML** como uno de los lenguajes de marcado más utilizados en la actualidad.

HTML

HTML es el lenguaje con el que se define el contenido de las páginas web. Básicamente se trata de un conjunto de etiquetas que sirven para definir el texto y otros elementos que compondrán una página web, como imágenes, listas, vídeos, etc. [13]

Ilustración 11. Icono de HTML

Las características principales de HTML son:

- Lenguaje sencillo y más extendido.
- Fácil de aprender
- Código interpretable por los buscadores

2.2.2.B. BASES DE DATOS LOCALES

Una base datos constituye una herramienta capaz de almacenar datos de distinta índole, así como de conectarlos entre sí en una unidad lógica. Dicho almacenamiento y conexión, es posible implementarlo haciendo uso de servidores web ajenos al dispositivo origen. Sin embargo, también es posible llevarlo a cabo de manera local en el terminal que los recibe.

El término “datos locales” hace referencia a disponer de una conexión entre la aplicación y un archivo de base de datos en el equipo local. Esto es posible implementarlo con tecnologías como SQLite, definida en profundidad más adelante en el presente proyecto.

2.2.2.C. ENTORNOS DE DESARROLLO

Un entorno de desarrollo es un conjunto de procedimientos y herramientas que se utilizan para desarrollar un código fuente o programa. Este término se utiliza a veces como sinónimo de entorno de desarrollo integrado (**IDE**), que es la herramienta de desarrollo de software utilizado para escribir, generar, probar y depurar un programa. [14]

Los IDE pueden mejorar la productividad y el rendimiento del desarrollador al reducir el tiempo de configuración, aumentar la velocidad de desarrollo, mantener a los desarrolladores actualizados y otros. Las siguientes son algunas características típicas de un IDE:

- Editor de código fuente
- Depurador
- Compilador
- Finalización de código
- Ayuda de idioma
- Integraciones y complementos

A la hora de llevar a cabo el desarrollo de una aplicación Android, debemos destacar **Android Studio** como IDE recomendado por Google. [15]

Ilustración 12. Icono de Android Studio

Las características principales de Android Studio son:

- Fácil distribución de código
- Reutilización de código y recursos
- Permite compilar desde línea de comandos

Dicho IDE, ha sido el utilizado para programar lo relacionado con la aplicación móvil del proyecto.

Además, existen muchos otros entornos de desarrollo utilizados para la programación de aplicaciones, como **Visual Studio Code**. [16]

Ilustración 13. Icono de Visual Studio Code

Las características principales de Visual Studio Code son:

- Posibilidad de personalizar y agregar funcionalidades de forma modular y aislada.
- Programar diferentes lenguajes.
- Conectar con otros servicios.

Por su parte, en lo relacionado con la programación de microcontroladores, el IDE por excelencia es el de **Arduino**. [17]

Ilustración 14. Icono de Arduino

Las características principales de Arduino son:

- Puede ser utilizado por diferentes lenguajes de programación
- Permite cargar el programa ya compilado en la memoria flash del dispositivo

Dicho IDE, ha sido el utilizado a la hora de programar la parte relacionada con el microcontrolador del proyecto.

3. ANÁLISIS DEL PROYECTO

En este tercer capítulo se pretende llevar a cabo un análisis del proyecto, indicando una serie de requisitos (tanto hardware como software) que se han tenido en cuenta a la hora de elaborarlo. Además, se incluirá un amplio análisis de los casos de uso presentes.

3.1. DEFINICIÓN DEL PROYECTO

El objetivo principal de este Trabajo de Fin de Grado es la creación y desarrollo de un sistema compuesto por diferentes sensores de bajo coste y un microprocesador como unidad central, que permita captar, monitorizar y almacenar diferentes parámetros medioambientales en tiempo real, de una forma rápida, fácil y segura mediante el uso de una aplicación Android.

Dicha aplicación, será la encargada de procesar los datos recibidos del microprocesador a través de dos modos de comunicación inalámbricos diferenciados: BLE y WiFi, permitiendo el almacenamiento, consulta y gestión de estos. Asimismo, la aplicación permitirá al usuario establecer la frecuencia de muestreo con la que se recibirán los datos en nuestro dispositivo móvil y elegir entre uno de los modos de comunicación para recibirlos.

Así, podríamos recapitular las diferentes funcionalidades descritas anteriormente en el siguiente gráfico:

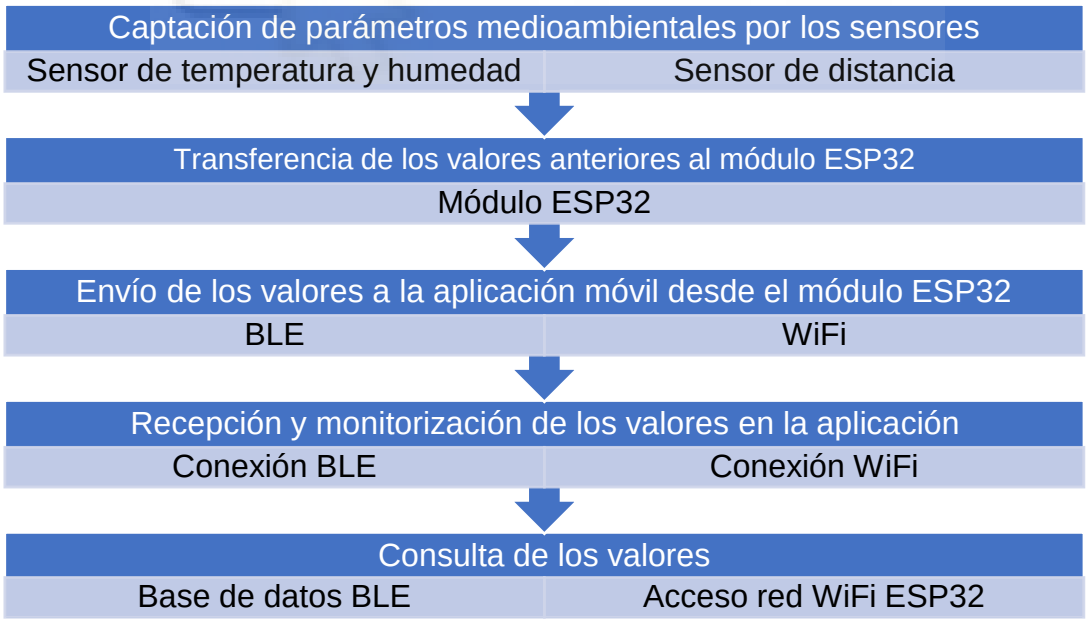


Ilustración 15. Gráfico de las diferentes funcionalidades del sistema

3.2. REQUISITOS HARDWARE

En lo referente a los hitos hardware del proyecto descritos anteriormente, el sistema desarrollado debe satisfacer una serie de necesidades.

Compuesto por un módulo ESP32, sensores de bajo coste, resistencias y el cableado correspondiente, la idea principal es captar 3 parámetros del entorno. A saber, de:

- Temperatura
- Humedad
- Distancia a objetos

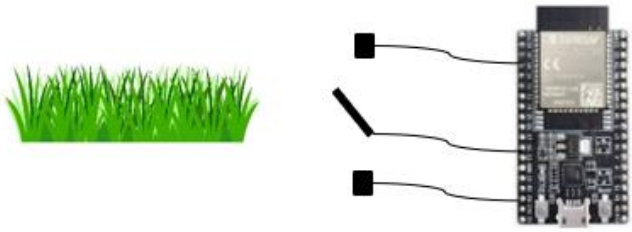


Ilustración 16. Captación de parámetros medioambientales por los sensores del sistema, conectados al módulo ESP32

Para ello, actualmente existen en la industria una amplia variedad de sensores capaces de llevar a cabo dicha funcionalidad.

Una vez captados los valores por los sensores, deberán transferirse a nuestro módulo para su posterior envío a la aplicación. El dispositivo seleccionado, el ESP32, es capaz de llevar a cabo un envío de información haciendo uso de diferentes tecnologías inalámbricas, permitiendo así, ofrecer al usuario diferentes alternativas a la hora de consultar la información medida.

Ilustración 17. Envío de datos desde el ESP32 haciendo uso de diferentes tecnologías

3.3 REQUISITOS SOFTWARE

En lo referente a los hitos software del proyecto descritos anteriormente, el sistema desarrollado debe satisfacer una serie de necesidades.

La aplicación Android desarrollada (programada con lenguaje Java), permite al usuario consultar los valores recibidos de los sensores. Para ello, deberá ser posible llevar a cabo diferentes tipos de conexiones.

Por una parte, será posible realizar un escaneo BLE, con el fin de permitir al usuario conectarse con un dispositivo para poder llevar a cabo un intercambio de información.

Por otro lado, si el dispositivo del usuario no dispone de tecnología BLE o si, simplemente opta por no utilizarla, deberá ser posible comprobar el estado de los sensores haciendo uso de una red WiFi creada por el propio ESP32.

En caso de haber realizado un intercambio de datos haciendo uso del BLE, la aplicación deberá ser capaz de mostrar por pantalla los valores recibidos, seleccionar el sensor a consultar, la frecuencia de muestreo de estos, así como poder consultar el histórico de las transacciones, pudiendo filtrar por diferentes parámetros.

En caso de haber seleccionado una conexión WiFi, el usuario podrá conectarse a la red en cuestión del propio microprocesador, para poder consultar de manera instantánea el estado de los sensores, además de poder seleccionar entre una serie de opciones, cada cuanto tiempo actualizar el valor de estos.

El siguiente esquema resume los diferentes requisitos software del sistema:

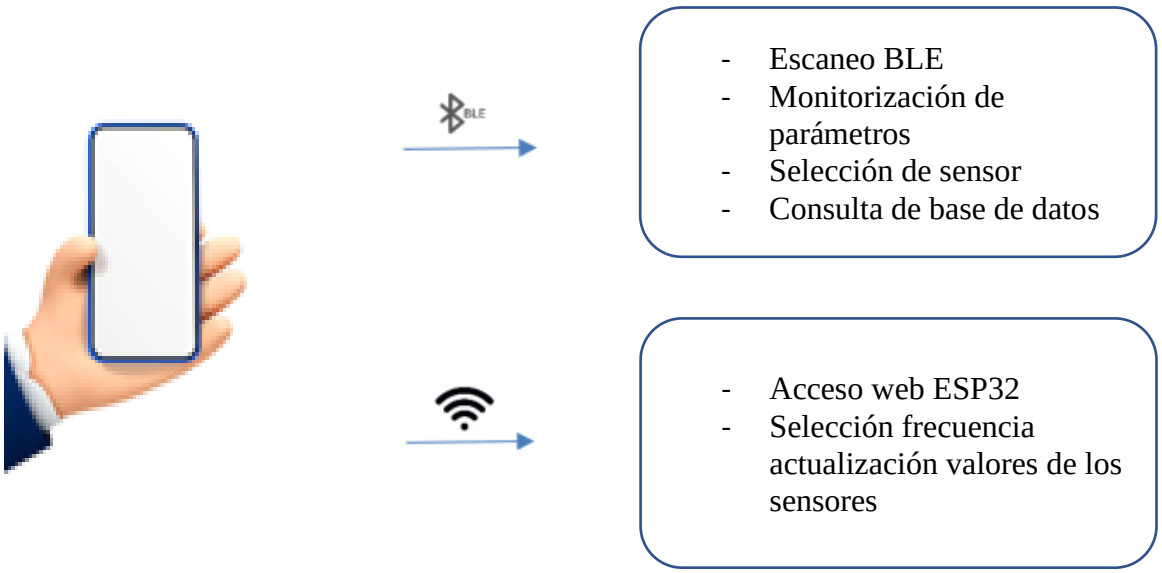


Ilustración 18. Esquema requisitos software del sistema

3.4. ANÁLISIS DE LOS CASOS DE USO

A partir de los diferentes requisitos definidos en los puntos anteriores, se diferencian los casos de uso del sistema a continuación:

Ilustración 19. Diagrama de los casos de uso

Captación de parámetros medioambientales

El sistema debe estar capacitado para obtener de manera continua los valores de diferentes parámetros medioambientales, así como para enviarlos haciendo uso de dos tecnologías inalámbricas disponibles, BLE y WiFi, permitiendo así al usuario acceder a ellos en todo momento.

Escaneo BLE

El usuario debe ser capaz de poder llevar a cabo un escaneo de dispositivos BLE, detectar el módulo ESP32 utilizado y conectarse a este para poder llevar a cabo un intercambio de información.

Consultar estado de los sensores vía BLE

Una vez llevado a cabo el correspondiente escaneo BLE y, tras conectarse al dispositivo en cuestión, el usuario tiene la opción de comprobar de manera instantánea el estado de los sensores, pudiendo monitorizar y almacenar el valor de cualquiera de ellos al mismo tiempo.

Consultar base de datos

Existe la posibilidad de comprobar el histórico de las transacciones realizadas entre la aplicación y el módulo ESP32, pudiendo filtrar los resultados por fecha, sensor etc. Además, en caso de ser necesario, podrá eliminarse la base de datos.

Consultar estado de los sensores vía WiFi

Por su parte, el usuario tendrá la posibilidad de acceder a la información de los sensores conectándose a la red WiFi que facilita nuestro sistema, pudiendo comprobar desde cualquier terminal el estado del dispositivo.

Establecer frecuencia de muestreo

En caso de haber llevado a cabo una conexión BLE con el sistema, el usuario tendrá la posibilidad de establecer la frecuencia de muestreo, es decir, seleccionar cada cuanto tiempo desea recibir los datos. Por su parte, en caso de haber accedido a la red WiFi creada por el ESP32, con el objetivo de comprobar el estado de nuestro sistema, el usuario dispondrá de diferentes opciones que le permitan refrescar el estado de los sensores de manera instantánea.

4. ARQUITECTURA DE LA SOLUCIÓN

En este cuarto capítulo se pretende llevar a cabo un análisis de las diferentes tecnologías hardware y software utilizadas para el desarrollo del proyecto. Además, se argumentará el motivo por el cual estas han sido seleccionadas para su posterior implementación.

4.1. ARQUITECTURA HARDWARE

En esta sección se explicarán en detalle las diferentes tecnologías y componentes hardware utilizados en la elaboración del proyecto.

4.1.1. ESP32

El módulo ESP32 constituye un *System On Chip*, diseñado por la empresa Espressif, pero fabricado por TSMC.

Capaz de ejecutar aplicaciones en tiempo real, hace del ESP32 uno de los dispositivos más interesantes disponibles en el mercado para la elaboración de proyectos IoT y otros tantos relacionados.

Chip, módulo y tarjeta de desarrollo

Es conveniente llevar a cabo una diferenciación entre los términos: chip, módulo y tarjeta de desarrollo, aunque comúnmente nos refiramos a él como 'módulo ESP32' o 'chip ESP32' de manera indiferente.

Hablamos de **módulo**, al referirnos al dispositivo que lleva el chip integrado, junto a una memoria flash, un cristal de 40 MHz y una posible antena (dependiendo del modelo).

Ilustración 20. Módulo ESP32

Por su parte, la **tarjeta de desarrollo** suele integrar el módulo anteriormente descrito, en una PCB, con el objetivo de brindar al sistema de otras tantas características y funcionalidades: conexión serie USB, alimentación USB, botones, pines soldados a la placa etc.

Ilustración 21. Tarjeta de desarrollo ESP32

Chips de la familia ESP32

Al igual que su predecesor, cuenta en su haber con diferentes modelos con varias características diferenciadas: [18]

Ordering code	Core	Embedded flash	Connection	Package
ESP32-DOWDQ6	Dual core	No embedded flash	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 6*6
ESP32-DOWD	Dual core	No embedded flash	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 5*5
ESP32-D2WD	Dual core	16-Mbit embedded flash (40 MHz)	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 5*5
ESP32-S0WD	Single core	No embedded flash	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 5*5

Tabla 1. Chips de la familia ESP32

La nomenclatura de los chips anteriores define sus características y sigue el siguiente esquema:

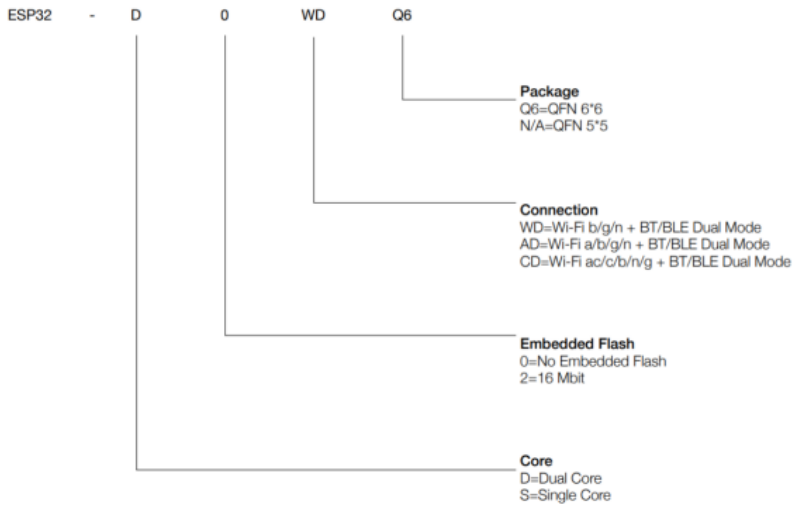


Ilustración 22. Nomenclatura de los chips ESP32

Así, podremos diferenciar entre chips con uno o dos núcleos, si cuentan con memoria flash embebida o no, tipo de conexión WiFi que soportan y tamaño del encapsulado.

Módulos de la familia ESP32 desarrollados por Espressif Systems

Seguidamente se listan los principales módulos de la familia ESP32 desarrollados por Espressif Systems:

-	Key Components				Dimensions [mm]		
Module	**Chip**	**Flash**	**RAM**	**Ant.**	**L**	**W**	**D**
ESP32-WROOM-32	ESP32-DOWDQ6	4 MB	-	MIFA	25.5	18	3.1
ESP32-WROOM-32D	ESP32-DOWD	4 MB	-	MIFA	25.5	18	3.1
ESP32-WROOM-32U	ESP32-DOWD	4 MB	-	U.FL	19.2	18	3.2
ESP32-SOLO-1	ESP32-SOWD	4 MB	-	MIFA	25.5	18	3.1
ESP32-WROVER	ESP32-DOWDQ6	4 MB	4 MB	MIFA	31.4	18	3.2
ESP32-WROVER-I	ESP32-DOWDQ6	4 MB	4 MB	U.FL	31.4	18	3.5

Tabla 2. Módulos de la familia ESP32

Actualmente existen en el mercado numerosos fabricantes que desarrollan módulos similares a los mencionados anteriormente con características y funcionalidades semejantes a las anteriores.

Módulo ESP32 seleccionado

Pese a existir una amplia gama de dispositivos que cumplan con las especificaciones necesarias para poder llevar a cabo el presente proyecto, el módulo seleccionado pertenece al fabricante Lilygo-TTGO y posee unas características similares al ESP32-WROVER-I descrito anteriormente:

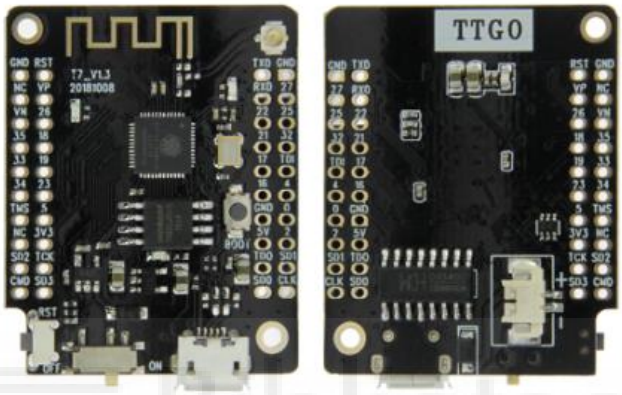


Ilustración 23. Dispositivo ESP32 seleccionado

Especificaciones detalladas ESP32

Seguidamente se detalla una descripción interna del módulo ESP32, funcionalidades y características principales [19].

La siguiente imagen muestra los principales bloques del dispositivo [20]:

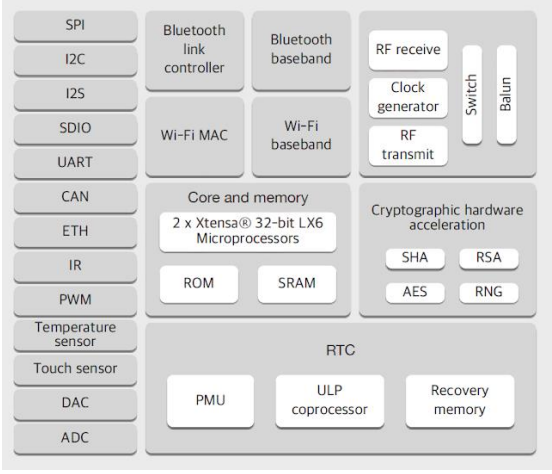


Ilustración 24. Diagrama de bloques del ESP32

Procesadores

El ESP32 monta uno o dos microprocesadores Xtensa de doble núcleo (o un solo núcleo) LX6 de 32 bits, que funciona a 160 o 240 MHz y funciona hasta 600 DMIPS.

Por su parte, incorpora un coprocesador de ultra baja potencia (ULP), capaz de trabajar cuando la CPU entra en modo deep-sleep de ahorro de energía. Puede ejecutar conversiones ADC, operaciones computacionales y verificar el estado de los pines con un consumo minúsculo.

Memoria

En cuanto a la memoria, el chip dispone de las siguientes memorias internas:

- Memoria ROM de 448 KiB, para funciones de núcleo y boot.
- Memoria SRAM de 520 KiB, para datos e instrucciones.
- Memoria RTC fast SRAM de 8 KiB, para almacenamiento de datos. Puede ser usada por el núcleo principal durante el boot desde el modo deep-sleep.
- Memoria RTC slow SRAM de 8KiB, para que el coprocesador acceda durante el modo deep-sleep.
- Memoria flash embebida, que, dependiendo del modelo, tiene un tamaño de 0 a 4 MiB.

Timers

Dispone de cuatro temporizadores de propósito general embebidos de 64 bits, basados en *prescalers* de 16 bits (de 2 a 65536). Los Timers son configurables en cuenta ascendente o descendente y pueden ser generados como interrupción por nivel o por flanco.

Watchdog

Dispone de tres Watchdog Timers, uno en cada núcleo, llamados Main Watchdog Timer (MWDT) y otro en el RTC, llamado RTC Watchdog Timer (RWDT). Están destinados a recuperar al dispositivo ante un fallo imprevisto, pudiendo desencadenar distintas acciones como el reinicio de la CPU, del núcleo o del sistema.

Relojes del sistema

El dispositivo cuenta con diferentes tipos de relojes del sistema.

En primer lugar, el reloj de la CPU, cuya fuente de reloj es por defecto un reloj de cristal externo de 40MHz que se conecta a un PLL para generar una frecuencia de 160Mhz. Además, cuenta con un oscilador interno de 8MHz. El programa puede elegir que reloj utilizar en el sistema, dependiendo de la aplicación.

En segundo lugar, un reloj de tiempo real (RTC), cuya fuente puede ser un reloj de un oscilador RC interno de 150KHz, de un oscilador interno de 8MHz, o del cristal externo al dispositivo.

Radiofrecuencia

El módulo ESP32 consta de los siguientes bloques de radiofrecuencia:

- Un receptor de 2.4 GHz que demodula la señal de radiofrecuencia y la convierte al dominio digital por medio de dos ADCs de alta resolución. Para adaptarse a las diferentes condiciones de la señal, el chip monta filtros de RF, control automático de ganancia, cancelación de continua y filtros de banda base integrados.

- Un transmisor de 2.4Ghz que modula la señal en banda base a señal de RF a 2.4Ghz, que posteriormente sale por la antena gracias a un amplificador de potencia basado en CMOS. Incluye calibraciones integradas que permiten cancelar las imperfecciones de ruido.

- Un generador de reloj que produce una señal en cuadratura de 2.4Ghz para transmisión y recepción de WiFi con todos los componentes integrados y calibración automática.

Seguridad

Para la conexión vía WiFi el dispositivo soporta los estándares de seguridad de 802.11 como WFA, WPA/WPA2 y WAPI.

Además, permite la encriptación de la memoria flash del dispositivo y aceleración criptográfica hardware como AES (Advanced Encryption Standard), SHA-2 (Secure Hash Algorithm 2), RSA, ECC (Elliptic Curve Cryptography) y RNG (Random Number Generator).

Modos de operación

El SoC puede estar en distintos modelos de operación en función de la energía utilizada y son los siguientes:

- **Activo**: todo encendido, el chip puede recibir, transmitir y escuchar.
- **Modem-sleep**: la CPU está operativa y el reloj es configurable. WiFi y Bluetooth están desactivados.
- **Light-sleep**: la CPU está pausada. La memoria y los periféricos RTC están activos junto al coprocesador ULP de bajo consumo. La CPU se activará ante un evento que requiera su uso.
- **Deep-sleep**: solo la memoria y los periféricos RTC están encendidos. Los datos de conexión WiFi o Bluetooth se almacenan en la memoria RTC.
- **Hibernación**: el oscilador interno de 8MHz y el coprocesador ULP están deshabilitados. El RTC *Timer* o un evento generado por los pines GPIO pueden despertar al chip del modo hibernación.

Periféricos y sensores

En primer lugar, los pines de propósito general. El módulo ESP32 tiene 34 pines GPIO a los que se le pueden asignar distintas funciones mediante la programación de los registros apropiados. Hay pines de distintos tipos: digitales, digitales y analógicos, y con función *touch*. La mayoría de los pines GPIO se pueden configurar como *pull-up, pull-down*, o alta impedancia.

El ESP32 dispone de dos conversores analógico-digital (ADC) de 12-bit SAR, que pueden soportar entre ambos hasta 18 canales de entrada analógica. Además, es capaz de funcionar cuando el dispositivo entra en modo de ahorro de energía, gracias al coprocesador de bajo consumo. También posee un conversor digital-analógico (DAC) de 8 bits que permite convertir dos señales digitales en analógicas.

El dispositivo cuenta con una serie de sensores internos como un sensor de temperatura o un sensor de efecto Hall, que permite detectar si existe campo magnético cerca del dispositivo.

Para la comunicación da datos serie el dispositivo cuenta con tres interfaces UART que proporcionan comunicación asíncrona de hasta 5Mbps. Otros protocolos de comunicación como I2C, I2S y SPI son soportados por el dispositivo.

Para finalizar el apartado de periféricos, el ESP32 también puede generar hasta dieciséis canales PWM (*pulse width modulation*) y posee una interfaz para la comunicación por bus CAN 2.0.

Conectividad

Posee soporte para tecnologías WiFi y Bluetooth.

Respecto al WiFi, soporta las tecnologías estándar Wifi 802.11 b/g/n. (802.11n a 2.4 GHz hasta 150 Mbit/s), mientras que la versión de Bluetooth soportada es la v4.2 BR/EDR y dispone además de **BLE (Bluetooth Low Energy)**.

Pinout ESP32

Seguidamente se muestra el *pinout* del módulo ESP32 utilizado:

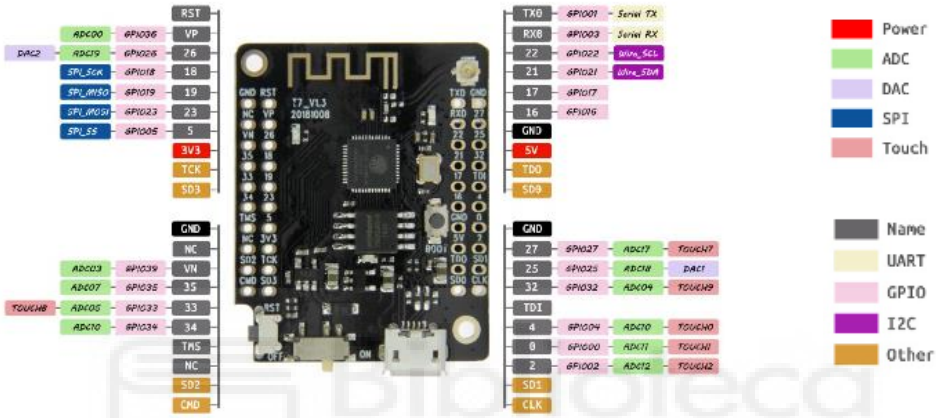


Ilustración 25. Pinout ESP32 utilizado

Cada uno de ellos tiene un propósito diferente:

- **Pines GPIO 34 y 39:** pines solamente de entrada. No pueden generar PWM ni actuar como salidas GPIO.
- **Pines GPIO6 al GPIO11:** conectados a la memoria flash SPI interna del chip, por lo que no se recomienda su uso ya que puede fallar al cargar el código en el dispositivo.
- **Pines para el ADC:** 0,2,4,12,13,14,15,25,26,27,32,33,34,35,36,39.
- **Pines GPIO25 y GPIO26:** pines que pueden usar el DAC.
- **Pines GPIO21 y GPIO22**: pines para la comunicación I2C.
- **Pines GPIO23 (MOSI), GPIO19(MISO), GPIO18(CLK), GPIO5(CS):** pines por defecto para la comunicación por SPI.

  Además, la tarjeta de desarrollo también posee pines de alimentación a 3V3 y 5V, tres pines conectados a GND (tierra) y dos pulsadores que se corresponden con los pines EN (reset) y GPIO0 (boot o gpio).

4.1.1.A. MODO BLE

El BLE, *Bluetooth Low Energy*, o en castellano, Bluetooth de baja energía, es una tecnología de conexión inalámbrica basada en el estándar Bluetooth 4.0.

Capaz de emitir ondas de radiofrecuencia en la banda de 2.4 GHz con alcance de hasta 100 metros, posee la gran ventaja de consumir muy poca energía, permitiendo así, emitir una señal durante meses o años sin recargar la batería del dispositivo que lo implementa. Además, incluye cifrado y seguridad configurable, por lo que podremos proteger nuestras conexiones de escucha externas.

Cómo se organizan los datos

La estructura de los datos en una conexión BLE, se realiza de forma jerárquica, disponiendo de una pila de protocolos en referencia a la capa OSI completamente nueva y orientada a conexiones sencillas [21]:

Capa física

Contiene la circuitería de comunicaciones capaz de realizar los procesos de modulación y demodulación de señales analógicas y posteriormente transformarlas en símbolos digitales. La tecnología BLE es capaz de utilizar hasta 40 canales de 2MHz en la banda ISM de 2.4 GHz. El estándar emplea la técnica de “saltos en frecuencia”, siguiendo una secuencia de saltos pseudoaleatorios entre los canales frecuenciales mencionados que ofrece un alto grado de robustez frente a interferencias.

Capa de enlace

Se encarga de gestionar características como los requerimientos temporales del estándar, chequeo de mensajes y reenvío de mensajes erróneos recibidos, gestión, filtrado de direcciones etc. Además, ofrece la definición de roles (*Advertiser*, *Scanner*, *Master and Slave*) que permiten identificar de forma lógica el rol de cada dispositivo en el proceso de comunicación. El nivel de enlace es del mismo modo responsable de procesos de control como el cambio de parámetros de la conexión o la encriptación.

HCI

HCI es un protocolo estándar que permite que la comunicación entre un host y un controlador se lleve a cabo a través de un interfaz serie. El estándar Bluetooth define HCI como el conjunto de comandos y eventos para la interacción de ambas partes (*host* y controlador).

Capa L2CAP

La capa L2CAP (*Logic Link Control and Adaptation Protocol*), se responsabiliza de dos tareas fundamentales en un proceso de comunicación. En primer lugar, el proceso de multiplexación, es decir, la capacidad de dar formato a mensajes provenientes de las capas OSI superiores y encapsularlos en paquetes estándar BLE, así como el proceso inverso.

Por otro lado, la fragmentación y recombinación. Paquetes que en nivel de aplicación suponen datagramas de gran cantidad de bytes son fragmentados correctamente adecuándose al MTU de BLE (27 bytes de *payload* máximo).

Para BLE, la capa L2CAP es la encargada de dar acceso y soporte a los dos protocolos fundamentales. Por un lado, ATT (*Attribute Protocol*), un protocolo basado en atributos presentados por dispositivo, con arquitectura cliente-servidor, que permite el intercambio de información. Por otro lado, SMP (*Security Manager Protocol*), protocolo que proporciona un *framework* para generar y distribuir claves de seguridad entre dos dispositivos.

En el nivel más alto de la capa de protocolos, encontraremos de forma paralela las capas GAP y GATT.

GAP

La capa **GAP** (*Generic Access Porfile*, o en castellano perfil de acceso genérico) se utiliza principalmente para controlar la conexión y transmisión de dispositivos. GAP hace que un dispositivo sea visible para otros dispositivos y determina si este dispositivo puede comunicarse con dispositivos interactivos.

Hay 4 tipos diferentes de transmisiones en la capa GAP:

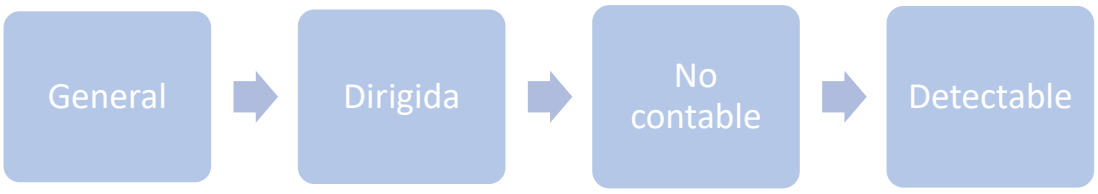


Ilustración 26. Tipos de transmisiones GAP

Cada vez que el dispositivo transmite, enviará el mismo mensaje en 3 canales de transmisión. Estos mensajes se denominan evento de difusión. A excepción de los mensajes dirigidos, se pueden seleccionar otros eventos de transmisión a intervalos que van desde 20ms a 10.28s. Generalmente, un dispositivo en una transmisión transmite cada segundo. El tiempo entre eventos de transmisión se denomina intervalo de transmisión, y el host puede controlar el intervalo.

En la mayoría de los casos, el dispositivo periférico se transmite para permitir que el dispositivo central se descubra a sí mismo y establezca una conexión GATT para intercambiar más datos. En algunos casos, no es necesario conectarse, siempre que el dispositivo periférico difunda sus propios datos. El objetivo principal de este método es permitir que los dispositivos periféricos envíen su propia información a múltiples dispositivos centrales.

Roles del dispositivo GAP

GAP define varios roles para dispositivos, los dos principales son: dispositivos periféricos (servidor periférico esclavo) y dispositivos centrales (cliente central-host).

- Equipo periférico esclavo: Este es generalmente un dispositivo de baja potencia muy pequeño o simple que se utiliza para proporcionar datos y conectarse a un dispositivo central más potente. Por ejemplo, pulseras inteligentes.

- Equipo central-host: El dispositivo central es relativamente potente y se utiliza para conectar otros dispositivos periféricos. Por ejemplo, un teléfono móvil.

Datos de difusión GAP

En GAP, los dispositivos periféricos transmiten datos de dos maneras:

- Carga de datos de publicidad (datos de transmisión).

- Carga de datos de respuesta de escaneo (respuesta de escaneo).

Cada tipo de datos puede contener hasta 31 bytes. Aquí la transmisión de datos es necesaria, ya que los periféricos deben transmitir al exterior constantemente, informar al equipo central de su existencia. La respuesta de escaneo es opcional.

El dispositivo central puede solicitar una respuesta de escaneo al dispositivo periférico. Contiene información adicional sobre el dispositivo, como el nombre del dispositivo.

Proceso de transmisión GAP

El flujo de trabajo de difusión de GAP se muestra a continuación:

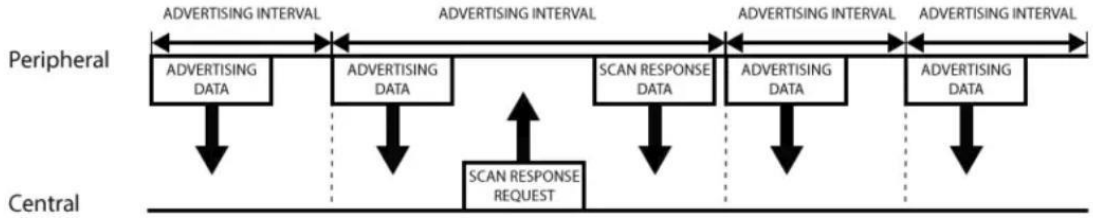


Ilustración 27. Flujo de trabajo difusión GAP

Se observa cómo funcionan los datos de transmisión y los datos de respuesta de escaneo. El dispositivo periférico establecerá un intervalo de transmisión y, en cada intervalo de transmisión, reenvía sus propios datos de transmisión. Cuanto más largo sea el intervalo de transmisión, más ahorro de energía.

Topología de red de difusión GAP

En la mayoría de los casos, el dispositivo periférico se transmite para permitir que el dispositivo central se descubra a sí mismo y establezca una conexión GATT para intercambiar más datos. En algunos casos, no es necesario conectarse, siempre que el dispositivo periférico difunda sus propios datos. El objetivo principal de esta manera es permitir que los dispositivos periféricos envíen su información a múltiples dispositivos centrales. Debido a que se basa en la conexión GATT, solo se puede conectar un dispositivo periférico a un dispositivo central.

La siguiente figura muestra el diagrama de topología de red en el modo de trabajo difusión:

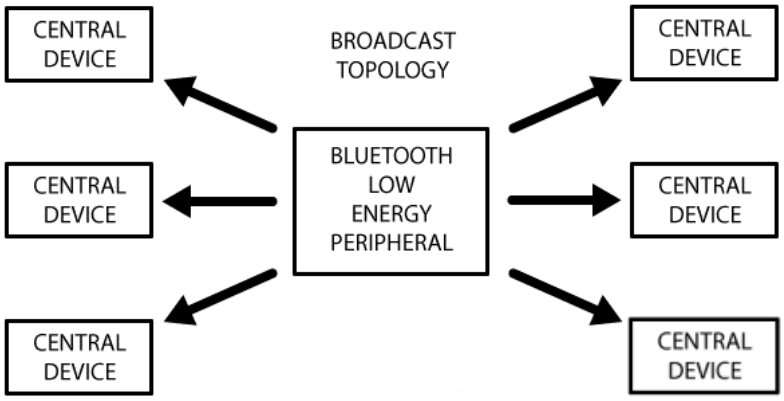


Ilustración 28. Diagrama de topología en red en modo difusión

GATT

El nombre completo de **GATT** es *Generic Attribute Profile,* en castellano, perfil de atributos genérico, que define dos dispositivos BLE para comunicarse a través de algo llamado Servicio y Característica. GATT utiliza el protocolo ATT (Protocolo de atributos). El protocolo ATT almacena los datos correspondientes al servicio y los restos de características en una tabla de búsqueda. La tabla de búsqueda secundaria utiliza ID de 16 bits como índice para cada elemento.

Una vez que los dos dispositivos han establecido una conexión, GATT comienza a funcionar, lo que también significa que debe completar el protocolo GAP anterior. Así, la conexión GATT primero debe pasar por el protocolo GAP.

Se presta especial atención a la conexión GATT, ya que las conexiones GATT son exclusivas. Es decir, un periférico BLE solo puede conectarse mediante un dispositivo central a la vez. Una vez que el periférico está conectado, dejará de transmitir de inmediato, de modo que es un periférico BLE y solo puede conectarse mediante un dispositivo central. Una vez que el periférico está conectado, dejará de transmitir inmediatamente, de modo que no sea visible para otros dispositivos. Cuando el dispositivo se desconecta, comienza a transmitir nuevamente.

Si el dispositivo central y los periféricos necesitan comunicación bidireccional, la única forma es establecer una conexión GATT.

Topología de red conectada GATT

La siguiente figura muestra la topología de la red de conexión GATT. Se aprecia como un dispositivo periférico solo puede conectarse a un dispositivo central, y un dispositivo central puede conectarse a múltiples dispositivos periféricos. Una vez se establece la conexión, la comunicación es bidireccional.:

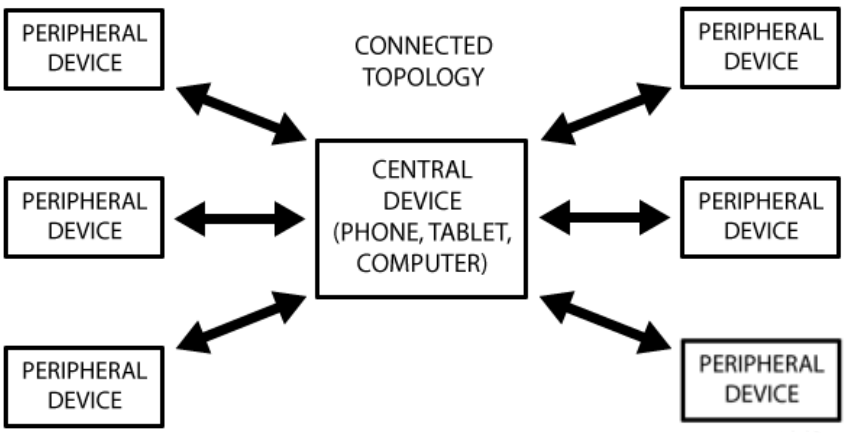


Ilustración 29. Topología de la red de conexión GATT

Comunicaciones GATT

Las dos partes de la comunicación del GATT son relación cliente-servidor. Como servidor GATT, el periférico mantiene la tabla de búsqueda de ATT y la definición de servicio y características. El dispositivo central es el cliente GATT (Cliente), que inicia las solicitudes al Servidor. Cabe señalar que todos los eventos de comunicación son iniciados por el cliente (también llamado maestro) y reciben la respuesta del servidor (también llamado esclavo).

Una vez que se establece la conexión, el dispositivo periférico sugerirá un intervalo de conexión (Intervalo de conexión) al dispositivo central, de modo que el dispositivo central intentará volver a conectarse en cada intervalo de conexión para verificar si hay nuevos datos. Sin embargo, este intervalo de conexión es solo una sugerencia. Es posible que su equipo central no siga estrictamente este intervalo.

La siguiente figura muestra el proceso de intercambio de datos entre un dispositivo periférico (servidor GATT) y un dispositivo central (cliente GATT). Puede ver que cada vez que el dispositivo maestro inicia una solicitud:

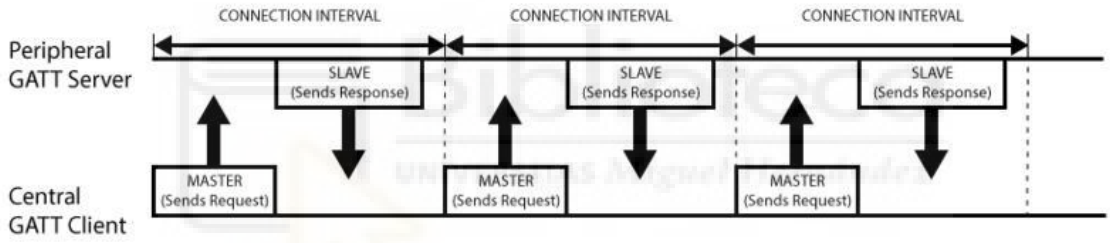


Ilustración 30. Intercambio de datos en una conexión GATT

Estructura GATT

La estructura de los datos se realiza de forma jerárquica y se divide en secciones. Estas secciones son perfiles, servicios y características. [22]

Perfil

Un perfil es un conjunto de servicios que están definidos por la Bluetooth SIG (*Bluetooth Special Interest Group*).

En realidad, no existe en el periférico BLE, es solo una colección de servicios predefinidos por Bluetooth SIG o el diseñador periférico. Por ejemplo, el perfil de frecuencia cardíaca.

Servicio

Los servicios son agrupamientos de datos simples, como por ejemplo la información de un sensor. Se dividen en características, y sirven para agruparlas.

El uso de los servicios sirve para, por ejemplo, un dispositivo que dispone de varios sensores. En lugar de crear un sólo servicio con todos los datos, se crea un servicio por sensor con el objetivo de tener los datos organizados.

Los servicios tienen un identificador único llamado UUID, el cual es de 16 bits para servicios oficiales, y de 128 bits para servicios personalizados. Al igual que los perfiles, están definidos por la Bluetooth SIG. Si el uso es privado, podemos generar nuestro propio UUID, haciendo uso de diferentes herramientas habilitadas para ello. [23]

Característica

El nivel más bajo en las transacciones GATT. La característica es la unidad de datos lógica más pequeña y puede contener un grupo de datos asociados (valor, descriptor. etc). De manera similar al servicio, cada característica se identifica de manera única por un UUID de 16 o 128 bits.

Tratar con periféricos BLE se realiza principalmente a través de las características. Es posible leer datos de características o escribir datos en ellas. De esta manera, se realiza una comunicación bidireccional.

La siguiente figura, muestra la estructura de datos en una comunicación GATT:

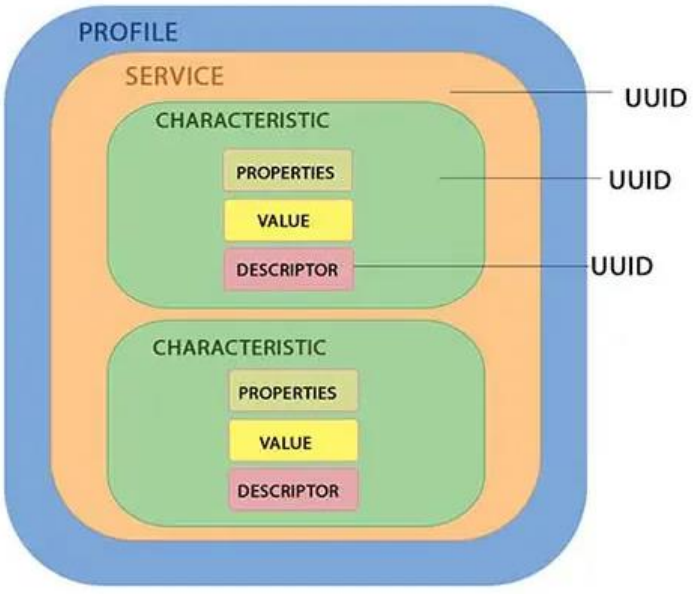


Ilustración 31. Estructura de datos GATT

4.1.1.B. MODO WI-FI

Como se ha especificado anteriormente, el dispositivo ESP32 dispone del estándar 802.11 b/g/n con una velocidad máxima de hasta 150Mbit/s a 2.4GHz. Además, dispone de varios modos de seguridad como WPA o WPA2, escáner de puntos de acceso activo y pasivo, y un modo promiscuo para monitorizar el envío de paquetes.

Cuando se trabaja con un dispositivo WiFi, es importante tener una noción básica de cómo funciona el dispositivo. En general, se trata de la comunicación TCP/IP sobre un enlace inalámbrico.

A continuación, se detallan los modos WiFi que permite el ESP32: []

Modo estación (STA)

El modo estación consiste en que el ESP conectado a una red, ya sea por un router o por algún dispositivo que genere una red local o por medio de un *access point*.

Cuando el ESP a esta conectado a la red local lo que hace es obtener una dirección IP y con esta, configura un punto de entrada donde se da acceso a los servicios que ofrecer el dispositivo y cualquier cliente que tenga acceso a la IP podrá acceder a dichos servicios.

Ilustración 32. ESP32 en modo estación

Modo punto de acceso (*Soft AP*)

En este modo el dispositivo crea su propia red local y el microcontrolador actúa como un punto de acceso para la red, esto permite que hasta 10 clientes (dispositivos) puedan conectarse por medio de esta red local al punto que se ha creado con el ESP, adicionalmente en esta red local se puede configurar el nombre de red, su identificador y la contraseña para que los dispositivos que se conecten puedan acceder de forma segura. Para este caso hay que definir el SSID y la contraseña a utilizar para crear el AP.

Es importante destacar que el dispositivo realiza conexiones directas con las estaciones que se conecten, por lo que, en el caso de conectar más de una estación, estas no se podrían comunicar entre ellas, sino a través del punto de acceso.

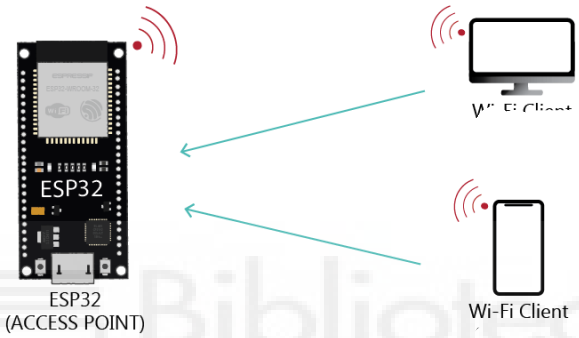


Ilustración 33. ESP32 en modo Access Point

El proyecto en cuestión implementa la funcionalidad WiFi en modo *Access Point*.

Modo combinado (AP + STA)

El dispositivo actúa como punto de acceso y a su vez está conectado a otra red como estación. Es un conjunto de los dos modos anteriores.

Ilustración 34. ESP32 en modo estación y Access Point

4.1.2. SENSORES UTILIZADOS

En la elaboración del sistema hardware se han utilizado dos sensores de bajo coste con el objetivo de medir diferentes parámetros medioambientales:

- Sensor de temperatura y humedad DHT11
- Sensor de distancia HC-SR04

DHT11

El sensor DHT11 constituye una de las mejores alternativas disponibles en el mercado en cuanto a medida de temperatura y humedad a bajo coste.

Definición

Se trata de un sensor digital de bajo costo y sencillo de utilizar. Integra un sensor capacitivo de humedad y un termistor para medir el aire circundante. Muestra los datos mediante una señal digital en el pin de datos.

A nivel de software, existen diferentes librerías disponibles en Arduino para llevar a cabo su programación haciendo uso del protocolo *single bus,* el cual consiste en un bus, un maestro y varios esclavos de una sola línea de datos en la que se alimentan.

Respecto a la parte hardware, consta de 4 pines diferenciados, los cuales van conectados a los pines del microprocesador:

1. Alimentación (VCC).
2. Señal (DATA).
3. No usado.
4. Tierra (GND).

Ilustración 35. Sensor DHT11

Especificaciones técnicas

Seguidamente se listan las principales especificaciones técnicas del sensor obtenidas del *datasheet* del dispositivo [25]:

- **Voltaje de Operación:** 3V - 5V DC.
- **Rango de medición de temperatura:** 0 a 50 °C.
- **Precisión de medición de temperatura:** ±2.0 °C.
- **Resolución Temperatura:** 0.1°C.
- **Rango de medición de humedad:** 20% a 90% RH.
- **Precisión de medición de humedad:** 5% RH.
- **Resolución Humedad:** 1% RH.
- **Tiempo de sensado:** 1 s.
- **Interfaz digital:** Single-bus (bidireccional).
- **Dimensiones:** 16*12*5 mm.
- **Peso:** 1 gr.
- **Esquema eléctrico:**

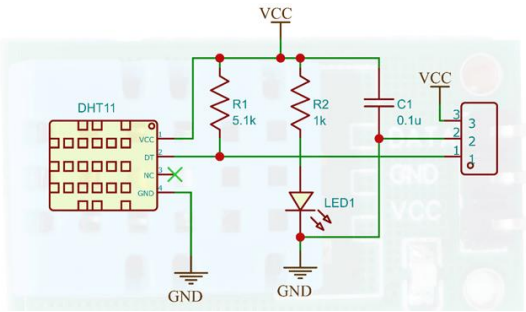


Ilustración 36. Esquema eléctrico sensor DHT11

HC-SR04

El sensor HC-SR04 constituye una de las mejores alternativas disponibles en el mercado en cuanto a medida de distancias a objetos a bajo coste.

Definición

Se trata de un sensor de distancia de bajo coste que utiliza ultrasonido para determinar la distancia de un objeto. Destaca por su pequeño tamaño, bajo consumo energético, buena precisión y excelente precio.

Consta de 4 pines diferenciados, los cuales van conectados a los pines del microprocesador:

1. Alimentación (VCC).
2. Disparo del ultrasonido (TRIG).
3. Recepción de ultrasonido (ECHO).
4. Tierra (GND).

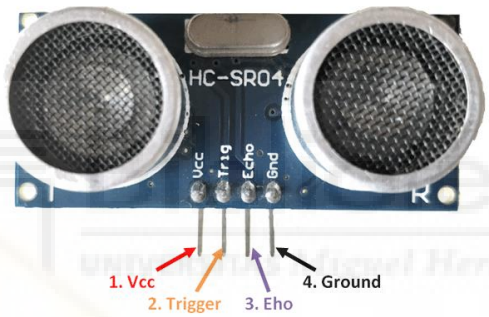


Ilustración 37. Sensor HC-SR04

Funcionamiento

El emisor piezoeléctrico emite 8 pulsos de ultrasonido (40KHz) después de recibir la orden en el pin TRIG. Seguidamente, las ondas de sonido viajan en el aire y rebotan al encontrar un objeto. A continuación, el sonido de rebote es detectado por el receptor piezoeléctrico. Seguidamente, el pin ECHO cambia a nivel Alto (5V) por un tiempo igual al que demoró la onda desde que fue emitida hasta que fue detectada. Finalmente, el tiempo del pulso ECO es medido por el microcontrolador y así se puede calcular la distancia al objeto.

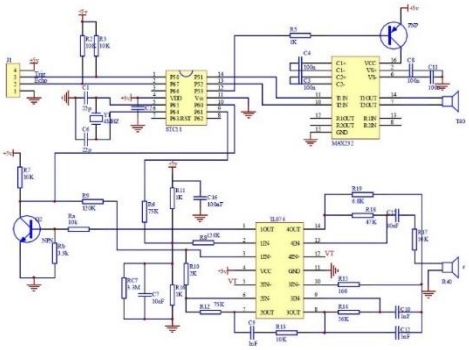
Por lo tanto, podremos calcular la distancia utilizando la siguiente fórmula:

$$\textbf{Distancia [m]} = \frac{\textbf{Tiempo del pulso ECO} \cdot \textbf{Velocidad del sonido}}{\textbf{2}}$$

Especificaciones técnicas

Seguidamente se listan las principales especificaciones técnicas del sensor obtenidas del *datasheet* del dispositivo [26]:

- **Voltaje de Operación**: 5 V DC.
- **Corriente de reposo:** < 2 mA Corriente de trabajo: 15Ma.
- **Rango de medición**: 2 cm a 450 cm.
- **Precisión**: +- 3mm Ángulo de apertura: 15°.
- **Frecuencia de ultrasonido**: 40 KHz.
- **Duración mínima del pulso de disparo TRIG (nivel TTL):** 10 µs.
- **Duración del pulso ECO de salida (nivel TTL):** 100-25000 µs.
- **Dimensiones:** 45*20*15 mm.
- **Tiempo mínimo de espera entre una medida y el inicio de otra**: 20ms
- **Peso**: 10g.
- **Esquema eléctrico y funcionamiento del sensor:**



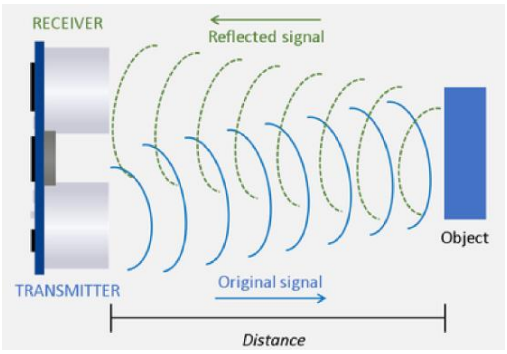


Ilustración 38. Esquema eléctrico y funcionamiento del sensor HC-SR04

4.1.3. OTROS COMPONENTES

Además del módulo ESP32 y los sensores anteriores, el sistema cuenta con otra serie de elementos comunes en la inmensa mayoría de circuitos electrónicos:

Cables de interconexión

Se han utilizado diferentes cables macho-macho para la interconexión de los diferentes elementos del circuito:

Ilustración 39. Cables macho-macho

Cable de alimentación micro USB

Se ha utilizado un cable micro USB para la alimentación del módulo ESP32 a través del ordenador portátil utilizado.

Ilustración 40. Cable micro USB

Resistencias

Ha sido necesaria una resistencia pull-up de valor 10 kOhm a la hora de llevar a cabo la conexión del sensor DHT11 con el resto del sistema.

Ilustración 41. Resistencia de 10 kOhm

Protoboard

El montaje de todo el circuito se ha hecho sobre una *protoboard*, elemento utilizado para la interconexión de los diferentes elementos del sistema.

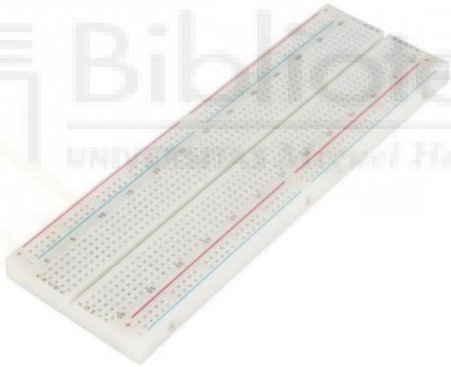


Ilustración 42. Protoboard

Alimentación

La alimentación del circuito se lleva a cabo a través del puerto micro USB del módulo ESP32, el cual, en este caso, está conectado al ordenador. Sin embargo, podría sustituirse por algún tipo de batería u otro elemento electrónico.

4.2. ARQUITECTURA SOFTWARE

En esta sección se explicarán en detalle las diferentes tecnologías y componentes software utilizados en la elaboración del proyecto.

4.2.1. JAVA

El sistema operativo Android ha sido diseñado con Java, por lo que, durante mucho tiempo, ha constituido el lenguaje de programación por defecto para el desarrollo de aplicaciones en esta plataforma. La sintaxis del lenguaje para el desarrollo de aplicaciones se fundamenta o deriva de otro lenguaje, el C++. Aunque actualmente son ya muchas las diferencias entre ambos lenguajes. Además, sirve para cualquier otro desarrollo, ya sea de sistemas *backend*, applets o cualquier otro sistema cliente-servidor que necesitemos implementar.

Ilustración 43. Mascota Java de desarrollo app

Se trata de un lenguaje de programación rápido, seguro, sencillo y ofrece un amplio rango de aplicaciones.

Seguidamente se muestran una serie de características más detalladas que las vistas en la sección '2.2.2.A. LENGUAJES DE PROGRAMACIÓN', relacionadas con el proyecto:

- No existen punteros.
- Definición de tipos (typedef).
- Orientado a objetos.
- Proporciona librerías para ser un lenguaje distribuido.
- Uso de excepciones (control de errores).

- Se trata de un lenguaje de arquitectura neutral.

- Lenguaje multithreaded. Java permite muchas actividades simultáneas en un programa. Los threads (a veces llamados, procesos ligeros), son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los threads contruidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++. El beneficio de ser miltithreaded consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real.

- Java se beneficia todo lo posible de la tecnología orientada a objetos. No intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librería nuevas o actualizadas no paralizan las aplicaciones actuales (siempre que mantengan el API anterior).

- En resumen, las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o de sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del kernel del sistema operativo. Además, para evitar modificaciones por parte de los crackers de la red, implementa un método bastante seguro de autentificación por clave pública.

La elección de este lenguaje de programación a la hora de programar la aplicación del proyecto se debe a que este ha sido uno de los más utilizados en la carrera del estudiante.

4.2.2. C++

Antes de comenzar hablando del propio lenguaje en sí, es conveniente indicar que Arduino utiliza un lenguaje de programación propio, el cual está basado en C++. Por lo tanto, comparte las principales ventajas de este lenguaje de programación. Además, en las versiones más recientes del IDE, es posible incluso enviarle las instrucciones directamente en C++ sin tener que traducirlas a su propio lenguaje para programar esta placa. Esta es la razón por la cual se ha optado por programar el módulo ESP32 con dicho lenguaje.

Se trata de un lenguaje de programación considerado como uno de los de mayor nivel en cuanto a posibilidades en el mundo de la manipulación de objetos. Además de ello, hoy por hoy, sigue siendo un lenguaje completamente actualizado y útil para mantener la estabilidad, seguridad y buen rendimiento en el desarrollo de proyectos.

Por su parte, C++ es uno de los lenguajes de programación con menos vulnerabilidades que podemos encontrar, con tan solo un 6% de código vulnerable.

Las características de C++ pueden ser muy útiles al programar los microcontroladores. Además, C++ está estandarizado (el estándar ANSI), es muy portable, así que el mismo código se puede utilizar muchas veces en diferentes proyectos. Lo que lo hace accesible para cualquiera que conozca este lenguaje sin reparar en el propósito de uso del microcontrolador. C es un lenguaje compilado, lo que significa que los archivos fuentes que contienen el código C se traducen a lenguaje máquina por el compilador. Todas estas características hicieron al C uno de los lenguajes de programación más populares.

A la hora de compilar y cargar un programa en Arduino se sigue un proceso similar al mostrado a continuación [28]:

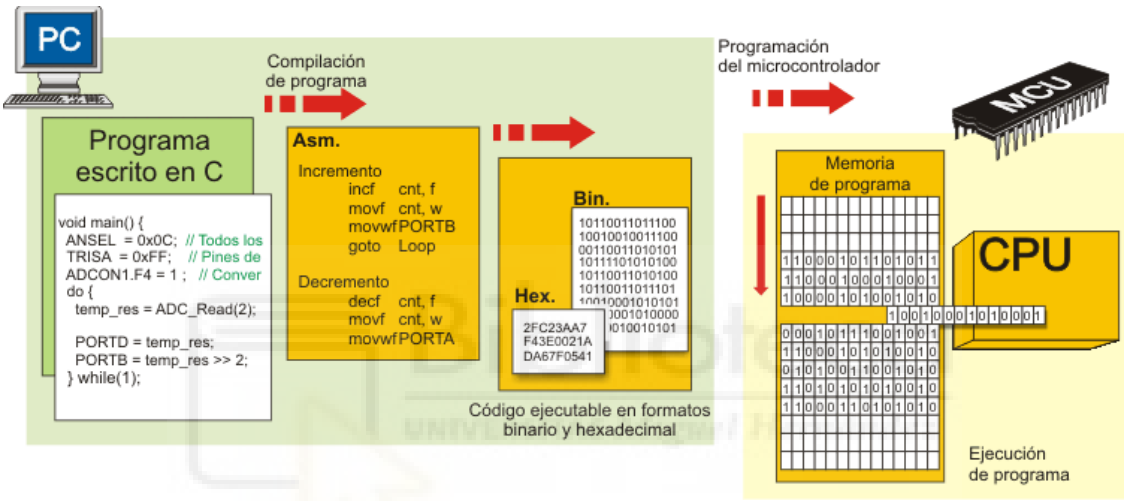


Ilustración 44. Proceso de compilación y carga de un programa en Arduino

La elección de este lenguaje de programación a la hora de programar el microprocesador del sistema del proyecto se debe a que este ha sido uno de los más utilizados en la carrera del estudiante.

4.2.3. HTML

HTML (*HyperText Markup Language, en castellano* Lenguaje de Marcas de Hipertexto*)* es el componente más básico de la Web. Define el significado y la estructura del contenido web. Además de HTML, generalmente se utilizan otras tecnologías para describir la funcionalidad y el comportamiento (JavaScript).

HTML hace referencia a cómo va ordenado el contenido de una página web, lo cual consigue haciendo uso de las marcas de hipertexto.

Así, "hipertexto" hace referencia a los enlaces que conectan páginas web entre sí, ya sea dentro de un único sitio web o entre sitios web. Los enlaces son un aspecto fundamental de la Web.

HTML utiliza "marcas" para etiquetar texto, imágenes y otro contenido para mostrarlo en un navegador Web. Las marcas HTML incluyen "elementos" especiales como:

**<head>, <title>, <body>, <header>, <footer>, <article>, <section>, <p>, , , <img>, <aside>, <audio>, <canvas>, <datalist>, <details>, <embed>, <nav>, <output>, <progress>, <video>, <ul>, <ol>, **

Un elemento HTML se distingue de otro texto en un documento mediante "etiquetas", que consisten en el nombre del elemento rodeado por "<" y ">". El nombre de un elemento dentro de una etiqueta no distingue entre mayúsculas y minúsculas. Es decir, se puede escribir en mayúsculas, minúsculas o una mezcla. Por ejemplo, la etiqueta <title> se puede escribir como <Title>, <TITLE> o de cualquier otra forma.

Seguidamente se muestran una serie de características más detalladas que las vistas en la sección '2.2.2.A. LENGUAJES DE PROGRAMACIÓN', relacionadas con el proyecto:

- Puede ser creado y editado con cualquier editor de texto.
- Lenguaje multiplataforma.
- Cada elemento de un documento HTML consta de una etiqueta de comienzo, un bloque de texto y una etiqueta de fin.
- Se trata de un lenguaje estático.
- Estándar reconocido por todo el mundo y cuyas normas define un organismo sin ánimo de lucro llamado World Wide Web Consortium, más conocido como W3C

4.2.4. JAVASCRIPT

JavaScript (JS) es un lenguaje de programación ligero, interpretado, o compilado justo-a-tiempo. Comúnmente es conocido como un lenguaje de secuencias de comandos para páginas web. [29]

Ilustración 45. Icono de Javascript

Cuando JavaScript se ejecuta en el navegador, no necesita de un compilador. El navegador lee directamente el código, sin necesidad de terceros.

Por tanto, se le reconoce como uno de los tres lenguajes nativos de la web junto a HTML (contenido y su estructura) y a CSS (diseño del contenido y su estructura).

Seguidamente se muestran una serie de, relacionadas con el proyecto:

- Lenguaje del lado del cliente.
- Orientado a objetos.
- De tipado débil o no tipado.
- Lenguaje interpretado.
- Funciones propias del lenguaje.
- Lenguaje dinámico. Responde a eventos en tiempo real.

4.2.5. SQLITE

SQLite es una herramienta de software libre, que permite almacenar información en dispositivos empotrados de una forma sencilla, eficaz, potente, rápida y en equipos con pocas capacidades de hardware, como puede ser una PDA o un teléfono móvil. [30]

Ilustración 46. Icono de SQLite

Seguidamente se muestran una serie de, relacionadas con el proyecto:

- La base de datos completa se encuentra en un solo archivo.
- Puede funcionar enteramente en memoria, lo que la hace muy rápida.
- Es totalmente autocontenida (sin dependencias externas).
- Cuenta con librerías de acceso para muchos lenguajes de programación.

Se trata de la herramienta utilizada en la elaboración de la base de datos de la aplicación móvil del proyecto.

4.3. HERRAMIENTAS

En esta sección se explicarán en detalle las diferentes tecnologías y herramientas de desarrollo utilizados en la elaboración del proyecto.

¿Qué es un entorno de desarrollo?

Un entorno de desarrollo es un conjunto de procedimientos y herramientas que se utilizan para desarrollar un código fuente o programa. Este término se utiliza a veces como sinónimo de entorno de desarrollo integrado (IDE), que es la herramienta de desarrollo de software utilizado para escribir, generar, probar y depurar un programa. También proporcionan a los desarrolladores una interfaz de usuario común (UI) para desarrollar y depurar en diferentes modos.

4.3.1. IDE ARDUINO

Se trata del entorno más utilizado para el desarrollo de aplicaciones con el ESP32.

Su fácil uso en cuanto a la programación se refiere, como su intuitiva interfaz, hacen de esta herramienta una de las opciones más atractivas disponibles en el mercado en lo que a la programación de microprocesadores se refiere. Además, cuenta con una amplia comunidad, encargada de mejorar la implementación del módulo ESP32 en el entorno de Arduino, así como de añadir progresivamente diferentes librerías que poder utilizar.

Las funciones que se definen en ESP-IDF son totalmente compatibles con el entorno Arduino, ya que ha sido el mismo fabricante Espressif Systems el que ha desarrollado Arduino Core para el ESP32, por lo que es una gran ventaja de cara a disponer de todas las funcionalidades que posee el dispositivo.

Tal y como se ha comentado previamente, se trata de un IDE bastante intuitivo, el cual permite configurar diferentes parámetros como el puerto al que conectar el dispositivo, frecuencia, partición de la memoria etc.

Ilustración 47. Vista general IDE Arduino

4.3.2. ANDROID STUDIO

Se trata del entorno oficial para el desarrollo de aplicaciones Android. Dedicado exclusivamente a la programación de aplicaciones para dispositivos Android, proporciona a Google un mayor control sobre el proceso de producción. Además, entre sus principales características cuenta con la posibilidad de ofrecer soporte para programar aplicaciones para *Android Wear* (sistema operativo para dispositivos corporales, como por ejemplo un *smartwatch*).

Por su parte, cuenta con herramientas *Lint*, utilizadas para detectar código no compatible entre arquitecturas diferentes o código confuso que no es capaz de controlar el compilador.

Otro de los aspectos más llamativos, es la integración de la herramienta Gradle, encargada de gestionar y automatizar la construcción de proyectos, como puede ser las tareas de *testing*, compilación o empaquetado.

La siguiente ilustración muestra el aspecto general del IDE de Android Studio:

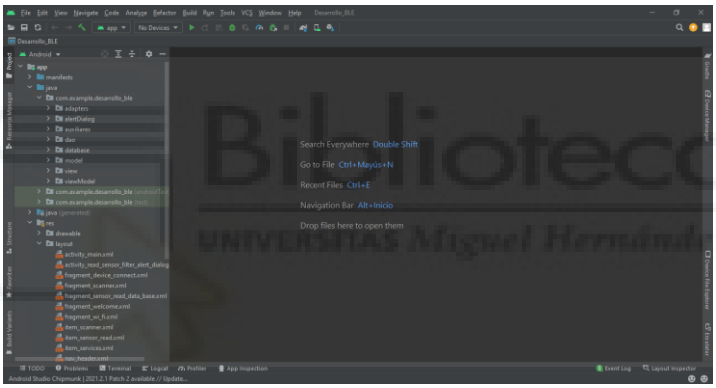


Ilustración 48. Vista general IDE Android Studio

Finalmente, otra de las características a destacar de la herramienta, es inclusión de un emulador que permite simular diferentes dispositivos (móviles, tablets, Smart TV etc.) para obtener una vista previa de la aplicación:

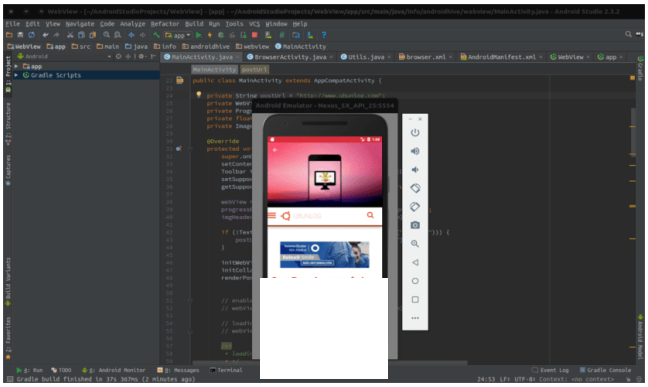


Ilustración 49. Simulador Android Studio

5. IMPLEMENTACIÓN

En este quinto capítulo se pretende explicar la implementación del sistema hardware desarrollado, compuesto por los sensores, cableado, resistencias y módulo ESP32 descritos anteriormente, así como de la aplicación Android utilizada para la monitorización y almacenamiento de los datos leídos.

5.1. IMPLEMENTACIÓN COMPONENTES HARDWARE

Seguidamente se explica la conexión de los diferentes sensores utilizados con el módulo ESP32 por separado, exponiendo los circuitos de interconexión, pines utilizados y programación.

Sensor de temperatura y humedad DHT11

Circuito eléctrico

La siguiente ilustración muestra la conexión del sensor DHT11 con el módulo ESP32:

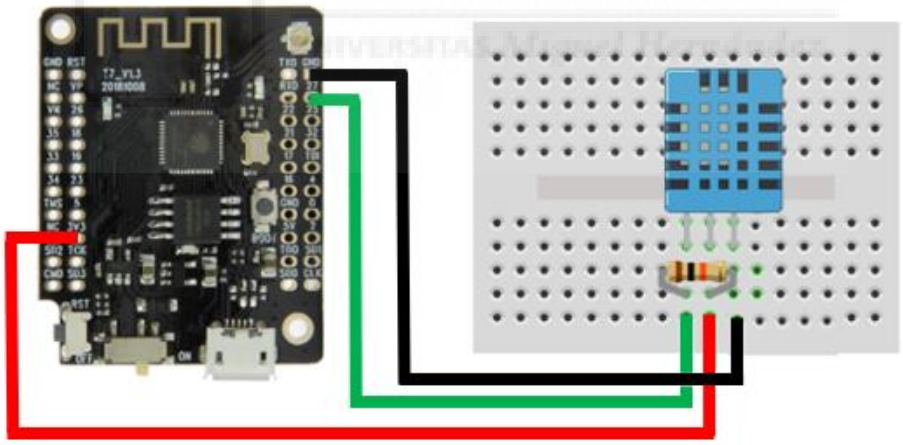


Ilustración 50. Circuito eléctrico sensor DHT11-ESP32

Pines del ESP32 utilizados

Pin de datos (DATA) → Pin 27 del ESP32

Pin de alimentación (VCC) → Pin de 3V3 del ESP32

Pin de masa (**GND**) → Pin GND del ESP32

Otros → Resistencia pull-up de 10 kOhm

Programación

A continuación, se explica el código utilizado en la programación del sensor, el cual, posteriormente será incorporado en el programa completo del sistema: [31]

Primeramente, a la hora de programar el sensor DHT11, es necesario incluir en el programa una librería propia para poder utilizarlo:

```
#include "DHT.h"
```

En segundo lugar, se definen el pin utilizado y la creación del sensor:

```
#define DHTPin 27
#define DHTTipo DHT11
```

Una vez llevados a cabo los pasos anteriores, se creará una instancia del sensor, pasándole como parámetros los datos anteriores:

```
DHT dht(DHTPin,DHTTipo);
```

Seguidamente se llevan a cabo dos configuraciones diferenciadas en todo sketch de Arduino:

Por una parte, se diferencia el *setup* de configuración, es decir, la parte del programa donde se llevarán a cabo las diferentes configuraciones previas del sistema. Solo se ejecutará una vez:

```
void setup() {
  // Damos de alta el puerto serial
  Serial.begin(9600);

  // Mostramos por pantalla mensaje de bienvenida
  Serial.println("Sensor DHT");

  // Declaración necesaria para poder trabajar con el sensor
  dht.begin();
}
```

Es posible distinguir tres partes diferentes:

1. Configuración de la velocidad de comunicación serial: utilizado para indicar a Arduino (al módulo ESP32) que se debe iniciar una comunicación con el ordenador a través del puerto serie estableciendo un valor de 9600 bits por segundo (baudios):

```
Serial.begin(9600);
```

2. Mensaje de bienvenida: se trata de un mensaje tipo, utilizado para comprobar a través del puerto serie que el sensor ha sido correctamente dado de alta:

```
Serial.println("Sensor DHT");
```

3. Inicio del sensor DHT11:

```
dht.begin();
```

Por otra parte, el *loop* del programa, es decir, la parte del código a ejecutar en bucle. Se trata del programa como tal:

```
void loop() {
  delay(2000);
  // Leemos los datos del sensor
  float t = dht.readTemperature();
  float h = dht.readHumidity();

  // Mostramos mensaje por pantalla en caso de ser errónea alguna de las medidas
  if(isnan(t) || isnan(h)){
    Serial.println("Lectura errónea");
    return;
  }

  // Mostramos por pantalla los valores de temperatura y humedad medidos
  Serial.println("Temperatura: ");
  Serial.println(t);
  Serial.println("Humedad: ");
  Serial.println(h);
  Serial.println("");
}
```

Es posible distinguir tres partes diferentes:

1. *Delay* inicial con valor 2000 ms y lectura de los valores de temperatura y humedad:

```
delay(2000);
// Leemos los datos del sensor
float t = dht.readTemperature();
float h = dht.readHumidity();
```

2. Mensaje indicativo en caso de error en la lectura del sensor:

```
if(isnan(t) || isnan(h)){
  Serial.println("Lectura errónea");
  return;
}
```

3. Mostrar por puerto serie el valor de los sensores:

```
Serial.println("Temperatura: ");
Serial.println(t);
Serial.println("Humedad: ");
Serial.println(h);
Serial.println("");
```

Finalmente se muestra el código completo utilizado en la programación del sensor DHT11:

```
// Librería necesaria para poder trabajar con el sensor
#include "DHT.h"

// Definición de pines y otras palabras reservadas
#define DHTPin 27
#define DHTTipo DHT11

// Instancia del sensor
DHT dht(DHTPin,DHTTipo);

void setup() {
  // Damos de alta el puerto serial
  Serial.begin(9600);

  // Mostramos por pantalla mensaje de bienvenida
  Serial.println("Sensor DHT");

  // Declaración necesaria para poder trabajar con el sensor
  dht.begin();
}

void loop() {
  delay(2000);
  // Leemos los datos del sensor
  float t = dht.readTemperature();
  float h = dht.readHumidity();


  // Mostramos mensaje por pantalla en caso de ser errónea alguna de las medidas
  if(isnan(t) || isnan(h)){
    Serial.println("Lectura errónea");
    return;
  }

  // Mostramos por pantalla los valores de temperatura y humedad medidos
  Serial.println("Temperatura: ");
  Serial.println(t);
  Serial.println("Humedad: ");
  Serial.println(h);
  Serial.println("");
}
```

Sensor de distancia HC-SR04

Circuito eléctrico

La siguiente ilustración muestra la conexión del sensor HC-SR04 con el módulo ESP32:

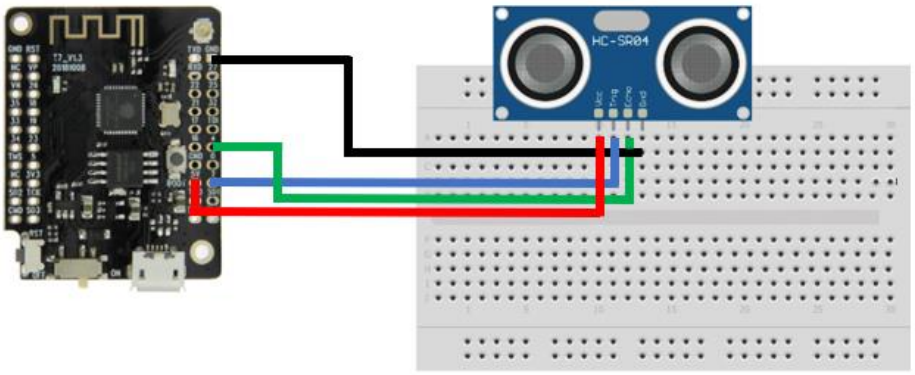


Ilustración 51. Circuito eléctrico sensor HC-SR04-ESP32

Pin del ESP32 utilizado

Pin de Trigger (TRIG) → Pin 2 del ESP32

Pin de Echo (ECHO) → Pin 4 del ESP32

Pin de alimentación (VCC) → Pin de 5V del ESP32

Pin de masa (**GND**) → Pin GND del ESP32

Programación

A continuación, se explica el código utilizado en la programación del sensor, el cual, posteriormente será incorporado en el programa completo del sistema: [32]

Primeramente, se definen los pines utilizados:

```
#define trig 2
#define eco 4
```

En segundo lugar, se definen dos variables que se utilizarán a la hora de calcular la distancia con respecto al objeto encontrado:

```
float duracion;
float distancia;
```

Seguidamente se llevan a cabo dos configuraciones diferenciadas en todo sketch de Arduino como las indicadas en el caso del sensor anterior:

Por una parte, se diferencia el *setup* de configuración, es decir, la parte del programa donde se llevarán a cabo las diferentes configuraciones previas del sistema. Solo se ejecutará una vez (en el programa general únicamente habrá un *setup* de configuración):

```
void setup() {
  // Damos de alta el puerto serial
  Serial.begin(9600);

  // Definimos los pines 'trig' y 'eco' como salida
  // y entrada repectivamente
  pinMode(trig, OUTPUT);
  pinMode(eco, INPUT);
}
```

Es posible distinguir dos partes diferentes:

1. Configuración de la velocidad de comunicación serial: utilizado para indicar a Arduino (al módulo ESP32) que se debe iniciar una comunicación con el ordenador a través del puerto serie estableciendo un valor de 9600 bits por segundo (baudios):

```
Serial.begin(9600);
```

2. Configurar los pines declarados anteriores como salida (trig) y entrada (eco):

```
pinMode(trig, OUTPUT);
pinMode(eco, INPUT);
```

Por otra parte, el *loop* del programa, es decir, la parte del código a ejecutar en bucle. Se trata del programa como tal:

```
void loop() {
  // Definimos un pulso alto y bajo en el pin 'trig'
  digitalWrite(trig, HIGH);
  delay(1);
  digitalWrite(trig, LOW);

  duracion = pulseIn(eco, HIGH);
  distancia = duracion /58.2;

  delay(1000);
}
```

Es posible distinguir tres partes diferentes:

1. Primeramente, se lanza un pulso alto en el pin 'trig', seguido de un pequeño *delay* y de un pulso bajo, con el objetivo de crear una señal cuadrada:

```
digitalWrite(trig, HIGH);
delay(1);
digitalWrite(trig, LOW);
```

2. Cálculo de la distancia al objeto:

```
duracion = pulseIn(eco, HIGH);
distancia = duracion /58.2;
```

3. *Delay* con valor 1000 ms:

```
delay(1000);
```

Finalmente se muestra el código completo utilizado en la programación del sensor HC-SR04:


```
// Definición de pines y otras palabras reservadas
#define trig 2
#define eco 4

float duracion;
float distancia;

void setup() {
  // Damos de alta el puerto serial
  Serial.begin(9600);

  // Definimos los pines 'trig' y 'eco' como salida
  // y entrada repectivamente
  pinMode(trig, OUTPUT);
  pinMode(eco, INPUT);
}

void loop() {
  // Definimos un pulso alto y bajo en el pin 'trig'
  digitalWrite(trig, HIGH);
  delay(1);
  digitalWrite(trig, LOW);

  duracion = pulseIn(eco, HIGH);
  distancia = duracion /58.2;

  delay(1000);
}
```

5.2. IMPLEMENTACIÓN FUNCIONALIDADES ESP32

En esta sección se explica el desarrollo de las dos tecnologías inalámbricas que ofrece el módulo ESP32 a la hora de enviar datos.

5.2.1. DESARROLLO BLE

A continuación, se explica el código utilizado en la programación del desarrollo BLE del módulo ESP32, el cual, posteriormente será incorporado en el programa completo del sistema: [33]

Primeramente, a la hora de programar la funcionalidad BLE, es necesario incluir en el programa una serie de librerías propia para poder utilizarlo:

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>
```

Seguidamente se instancia el servidor BLE y las características utilizadas:

```
BLEServer* pServer = NULL;
BLECharacteristic* pCharacteristic1 = NULL;
BLECharacteristic* pCharacteristic2 = NULL;
BLECharacteristic* pCharacteristic3 = NULL;
BLECharacteristic* pCharacteristic4 = NULL;
```

A continuación, se crean una serie de variables de control, así como otras relacionadas con el tiempo de emisión de los datos:

```
bool deviceConnected = false;
bool oldDeviceConnected = false;

int tiempo = 15000;
int tiempo_delay = 15000;
```

Por su parte, para poder arrancar un servidor BLE, es necesario levantar al menos un servicio y una característica, cada uno con su propio UUID, generado aleatoriamente en una página habilitada para ello.

En el proyecto en cuestión, será necesario instanciar cuatro servicios y características respectivamente, ya que, al disponer de tres sensores (temperatura, humedad y distancia) y de la frecuencia de muestreo, no será posible hacerlo únicamente con un servicio y una característica individual.

Seguidamente se muestra lo descrito anteriormente:

```
// Sensor 1 - DHT11 (Temperatura)
#define SERVICE_1_UUID          "2d22c77b-dbc3-4fb1-ba2a-8dfdeb115d6e"
#define CHARACTERISTIC_1_UUID   "7f425374-c825-41db-9cbd-c72bd17a8797"
```

```
// Sensor 2 - DHT11 (Humedad)
#define SERVICE_2_UUID          "f1c524f4-9867-4983-8eb9-5f81577cf2c2"
#define CHARACTERISTIC_2_UUID   "6fb17f9e-4c73-45a6-81fb-cb7dc5a7b108"

// Sensor 3 - HC-SR04 (Distancia)
#define SERVICE_3_UUID          "3460f0df-1f3d-4eee-b785-2fc9fec9d556"
#define CHARACTERISTIC_3_UUID   "b3bf19d2-070d-481e-a7a8-e00daa48f6de"

// Frecuencia de muestreo
#define SERVICE_4_UUID          "af5d710f-f0b4-4804-84c6-d0563c932b0e"
#define CHARACTERISTIC_4_UUID   "b0167c1c-91d6-4003-996b-4ec293acd456"
```

Otro de los puntos clave en el desarrollo BLE, es el uso de los *callbacks*.

Se tratan de porciones de código que permiten llamar a una función u otra dependiendo de la activación o no de un evento determinado.

En el desarrollo BLE se han utilizado dos diferenciados:

Por una parte, se ha utilizado un *callback* para indicar que se ha escrito en alguna característica, en este caso, en la correspondiente con la frecuencia de muestreo:

```
class MyCallbacks: public BLECharacteristicCallbacks
{

  // Función invocada cuando recibimos dato del cliente
  void onWrite(BLECharacteristic *pCharacteristic4)
  {
    // Almacenamos el valor de la característica 2
    std::string tiempo = pCharacteristic4->getValue();

    if (tiempo.length() > 0)
    {
      Serial.println("*********");
      Serial.print("Nueva 'Frecuencia de Muestreo': ");

      // Recorremos la cadena de caracteres (string)
      for (int i = 0; i < tiempo.length(); i++)
      {
        Serial.print(tiempo[i]);
      }

      Serial.println();
      Serial.println("*********");
    }
  }
};
```

Más allá de los comentarios propios mostrados por el monitor serie haciendo uso de la sentencia Serial.print, destacar lo siguiente:

El *callback* '**MyCallbacks'** utilizado, contiene una función 'onWrite' utilizada para escribir en la característica 4, correspondiente con la frecuencia de muestreo de los datos:

```
void onWrite(BLECharacteristic *pCharacteristic4)
```

Por otra parte, se ha utilizado otro *callback* para indicar si hay algún dispositivo conectado al módulo ESP32:

```
class MyServerCallbacks: public BLEServerCallbacks {

    // Funcion invocada cuando se conecte un cliente
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
      Serial.println("Dispositivo conectado");
    };

    // Funcion invocada cuando se desconecte un cliente
    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
      Serial.println("Dispositivo desconectado");
    }
};
```

Más allá de los comentarios propios mostrados por el monitor serie haciendo uso de la sentencia Serial.print, destacar lo siguiente:

El *callback* '**MyServerCallbacks'** utilizado, contiene un par de funciones: 'onConnect' y 'onDisconnect' utilizadas para indicar si hay algún dispositivo conectado al sistema:

```
void onConnect(BLEServer* pServer)

void onDisconnect(BLEServer* pServer)
```

Seguidamente se llevan a cabo dos configuraciones diferenciadas en todo sketch de Arduino:

Por una parte, se diferencia el *setup* de configuración, es decir, la parte del programa donde se llevarán a cabo las diferentes configuraciones previas del sistema. Solo se ejecutará una vez:

```
void setup() {
  Serial.begin(115200);
  delay(50);

  // BLE
      // 1. Inicializamos el Bluetooth con el nombre indicado
      BLEDevice::init("Módulo ESP32 - UMH");
```

```
// 2. Creamos SERVIDOR BLE
pServer = BLEDevice::createServer();
pServer->setCallbacks(new MyServerCallbacks()); // Funciones 'callback'


// 3.1. Definimos un SERVICIO con el UUID definido anteriormente
BLEService *pService1 = pServer->createService(SERVICE_1_UUID);

// 3.2. Definimos un SERVICIO 2 con el UUID definido anteriormente
BLEService *pService2 = pServer->createService(SERVICE_2_UUID);

// 3.3. Definimos un SERVICIO con el UUID definido anteriormente
BLEService *pService3 = pServer->createService(SERVICE_3_UUID);

// 3.4. Definimos un SERVICIO 2 con el UUID definido anteriormente
BLEService *pService4 = pServer->createService(SERVICE_4_UUID);


// 4. Creamos un servicio (1,2,3,4) asociado al servicio
// pService que acabamos de definir. Asociamos a este pService,
// una CARACTERÍSTICA y una propiedad/característica
// (READ + WRITE + NOTIFY + INDICATE)
pCharacteristic1 = pService1->createCharacteristic(
                    CHARACTERISTIC_1_UUID,
                    BLECharacteristic::PROPERTY_READ);

pCharacteristic2 = pService2->createCharacteristic(
                    CHARACTERISTIC_2_UUID,
                    BLECharacteristic::PROPERTY_READ);

pCharacteristic3 = pService3->createCharacteristic(
                    CHARACTERISTIC_3_UUID,
                    BLECharacteristic::PROPERTY_READ);

pCharacteristic4 = pService4->createCharacteristic(
                    CHARACTERISTIC_4_UUID,
                    BLECharacteristic::PROPERTY_READ
                    BLECharacteristic::PROPERTY_WRITE
                    );

// 5. Creamos descriptor (1,2,3,4) BLE
pCharacteristic1->addDescriptor(new BLE2902());
pCharacteristic2->addDescriptor(new BLE2902());
pCharacteristic3->addDescriptor(new BLE2902());
pCharacteristic4->addDescriptor(new BLE2902());

// 6. Damos un valor inicial a las características
pCharacteristic1->setValue(" ");
pCharacteristic2->setValue(" ");
pCharacteristic3->setValue(" ");
pCharacteristic4->setValue("2000");
```

```
    // 7. Una vez configurado el servicio,
    // lo iniciamos y escuchamos las solicitudes entrantes de los clientes
    pService1->start();
    pService2->start();
    pService3->start();
    pService4->start();

    // 8. Iniciamos el 'advertising' (indicamos que "estamos aquí").
    // Enviamos continuamente paquetes de publicidad para
    // que otros dispositivos puedan verlo y reconocerlo
    BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
    pAdvertising->addServiceUUID(SERVICE_1_UUID);
    pAdvertising->addServiceUUID(SERVICE_2_UUID);
    pAdvertising->addServiceUUID(SERVICE_3_UUID);
    pAdvertising->addServiceUUID(SERVICE_4_UUID);

    // 9. Buscamos respuestas a nuestro anuncio (Clientes interesados)
    pAdvertising->setScanResponse(false);
    pAdvertising->setMinPreferred(0x0);

    // 10. Iniciamos físicamente el 'advertising'
    BLEDevice::startAdvertising();
    Serial.println("\nEsperando dispositivos...");

    // 11. Configuramos 'callback' de nuestras caracteristicas
    pCharacteristic1->setCallbacks(new MyCallbacks());
    pCharacteristic2->setCallbacks(new MyCallbacks());
    pCharacteristic3->setCallbacks(new MyCallbacks());
    pCharacteristic4->setCallbacks(new MyCallbacks());
}
```

Es posible distinguir diferentes partes en el código anterior:

1. Definición del nombre del dispositivo, es decir, nombre que aparecerá en nuestro terminal móvil cuando se lleve a cabo un escaneo BLE:

```
BLEDevice::init("Módulo ESP32 - UMH");
```

2. Creación del servidor BLE. Cuando un cliente se conecte, devolverá el *callback*:

```
pServer = BLEDevice::createServer();
pServer->setCallbacks(new MyServerCallbacks());
```

3. Definición de los diferentes servicios utilizados, asociados a las UUID anteriores:

```
BLEService *pService1 = pServer->createService(SERVICE_1_UUID);
```

4. Creación de un servicio asociado al ‘pService’ definido anteriormente. Se le asignará una característica y una propiedad:

```
pCharacteristic2 = pService2->createCharacteristic(
                   CHARACTERISTIC_2_UUID,BLECharacteristic::PROPERTY_READ);
```

5. Creación de los diferentes descriptores:

```
pCharacteristic1->addDescriptor(new BLE2902());
```

6. Establecimiento de un valor inicial a las características definidas anteriormente:

```
pCharacteristic1->setValue(" ");
```

7. Inicio del servicio. Escucha continua de solicitudes:

```
pService1->start();
```

8. Inicio del *advertising*. Envío continuo de paquetes de publicidad:

```
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_1_UUID);
```

9. Búsqueda de respuesta al anuncio:

```
pAdvertising->setScanResponse(false);
pAdvertising->setMinPreferred(0x0);
```

10. Inicio físico del *advertising*:

```
BLEDevice::startAdvertising();
```

11. Configuración de los *callbacks* de las características:

```
pCharacteristic1->setCallbacks(new MyCallbacks());
```

Por otra parte, el *loop* del programa, es decir, la parte del código a ejecutar en bucle. Se trata del programa como tal:

```
void loop() {
  // BLE
        // Conectado
        if (deviceConnected) {

            //Actualizamos el valor de la cracterística 1 - Valor de temperatura
            pCharacteristic1->setValue(t);
            pCharacteristic1->notify();

            //Actualizamos el valor de la cracterística 2 - Valor de humedad
            pCharacteristic2->setValue(h);
            pCharacteristic2->notify();

            //Actualizamos el valor de la cracterística 3  - Valor de distancia
            pCharacteristic3->setValue(distancia);
            pCharacteristic3->notify();
```

```
    // Mostramos en el monitor serial los valores
    Serial.println("Secuencia de valores:");
    Serial.println(t);
    Serial.println(h);
    Serial.println(distancia);
    Serial.println("");

    //Actualizamos el valor de la cracterística 4 - 'delay'
    tiempo_delay = atoi(pCharacteristic4->getValue().c_str());
    Serial.println("Escala de tiempos:");
    Serial.println(tiempo_delay);
    Serial.println("\n");
    pCharacteristic4->notify();

    delay(tiempo_delay);
}

// Desconectado
if (!deviceConnected && oldDeviceConnected) {
    Serial.println("Dispositivo desconectado");
    pServer->startAdvertising(); // Reiniciamos el 'advertising'
    oldDeviceConnected = deviceConnected;
    delay(2000); // Damos a la pila de bluetooth la oportunidad de prepararse
}
```

Es posible distinguir dos partes en el código anterior:

1. En caso de haber un dispositivo conectado, se llevará a cabo una actualización de las diferentes características, correspondientes con el valor de los sensores, indicando que ha habido un cambio en uno de los atributos de esta:

```
pCharacteristic1->setValue(t);
pCharacteristic1->notify();
```

   En el caso de la característica 4, correspondiente con la frecuencia de muestreo, previamente se deberá llevar a cabo una conversión de esta para poder asociarla a la variable 'tiempo_delay' creada anteriormente:

```
tiempo_delay = atoi(pCharacteristic4->getValue().c_str());
pCharacteristic4->notify();
```

2. En caso de no haber ningún dispositivo conectado, se volverá a llevar a cabo un *advertising*:

```
pServer->startAdvertising();
```

5.2.2. DESARROLLO WI-FI

A continuación, se explica el código utilizado en la programación del desarrollo Wi-Fi del módulo ESP32, el cual, posteriormente será incorporado en el programa completo del sistema: [34]

Primeramente, a la hora de programar la funcionalidad BLE, es necesario incluir en el programa una serie de librerías propia para poder utilizarlo:

```
#include <WiFi.h>
#include <WebServer.h>
```

Seguidamente se definen las características de la red generada por el ESP32 en caso de funcionar en modo *Access Point* (AP):

```
const char* ssid     = "ESP32_UMH";
const char* password = "123456789";
```

A continuación, se define el número de puerto del servidor web. En este caso, en el 80:

```
WebServer server(80);
```

Después, se crean una serie de variables utilizadas a la hora de programar la web que implementará el ESP32:

```
unsigned long refresh_time = 0, compare = 0;
String mnj = "", datos = "";
```

En este punto, iría declarado el código correspondiente con la web que implementará el módulo del proyecto. Sin embargo, su explicación se llevará a cabo posteriormente.

Seguidamente se llevan a cabo dos configuraciones diferenciadas en todo sketch de Arduino:

Por una parte, se diferencia el *setup* de configuración, es decir, la parte del programa donde se llevarán a cabo las diferentes configuraciones previas del sistema. Solo se ejecutará una vez:

```
void setup() {
  Serial.begin(115200);
  delay(50);

  // WiFi (AP)
      // Configurando el AP
      Serial.print("Setting AP (Access Point)...");

      // Configuramos AP con el nombre de red y contraseña definidas anteriormente
      WiFi.softAP(ssid, password);
      IPAddress IP = WiFi.softAPIP();
```

```
    // Mostramos por el puerto serie el valor de la dirección IP
    Serial.print("AP IP address: ");
    Serial.println(IP);

    // Página web
    server.on("/", handleRoot);
    server.on("/vccread", vcc_data);
    server.on("/set", refresh_page);

    // Iniciamos el servidor
    server.begin();

}
```

Es posible distinguir tres partes importantes en el código anterior:

1. Configuración en modo AP del ESP32 haciendo uso de las características de la red definidas anteriormente:

```
WiFi.softAP(ssid, password);
IPAddress IP = WiFi.softAPIP();
```

   La red creada tiene la dirección IP **192.168.4.1**, por lo que, si queremos acceder a ella a través de nuestro navegador, basta con conectarnos a dicha red e introducir en el navegador web dicha dirección. [35]

2. Instancia de la web creada:

```
server.on("/", handleRoot);
server.on("/vccread", vcc_data);
server.on("/set", refresh_page);
```

3. Inicio del servidor web:

```
server.begin();
```

Por otra parte, el *loop* del programa, es decir, la parte del código a ejecutar en bucle. Se trata del programa como tal:

```
void loop() {
  // WiFi (AP)
        // Recibimos las peticiones de los clientes
        // y lanzamos las funciones de callback asociadas en el ruteo
        server.handleClient();
}
```

Se reciben las peticiones de los clientes y lanzan las funciones callback asociadas:

```
server.handleClient();
```

A continuación, y para finalizar esta sección, se describe la parte relacionada con la web asociada al módulo ESP32 descrita anteriormente.

La siguiente parte de código iría situada después de la zona de variables del programa, previa al 'void setup()':

```
// Web HTML utilizada a la hora de trabajar en modo AP
String webpage_start = "<!DOCTYPE html>"
                       "<html>"
                       "<style type=\"text/css\">"
                       ".button { background-color: #B7C6CA; border: 3pt color: white; padding: 15px 32px; text-align: center;"
                       "text-decoration: none; display: inline-block; font-size: 25px; }"
                       "</style>"
                       "<body style= background-color:#FFFFFF text=\"black\"><font face=\"Arial\"><h1><center><div>"
                       "<h1 style = \"font-size: 100px;\">Sensores ESP32 - Modo AP</h1>"
                       "<h2><style=\"color:black\"><font size=\"20\"><span id=\"vcc\"></span></h2>"
                       "<h3><button class=\"button\" onclick=\"send(1)\">1 Seg</button>"
                       "<button class=\"button\" onclick=\"send(2)\">3 Segs</button>"
                       "<button class=\"button\" onclick=\"send(3)\">5 Segs</button>"
                       "<button class=\"button\" onclick=\"send(4)\">10 Segs</button><br><br><h3>"
                       "</div>"
                       "<html><body style=\"zoom: 220%;\">"
                       "<script>"
                       "function send(state)"
                       "{"
                       "  var xhttp = new XMLHttpRequest();"
                       "  xhttp.onreadystatechange = function()"
                       "{"
                       " if (this.readyState == 4 && this.status == 200) {"
                       "  document.getElementById(\"state\").innerHTML = this.responseText;"
                       "}"
                       "};"
                       "xhttp.open(\"GET\", \"set?state=\"+state, true);"
                       "xhttp.send();"
                       "}"
                       "setInterval(function()"
                       "{"
                       "  get_vcc();"
                       "}, 1000);"
                       "function get_vcc()"
                       "{"
                       "  var xhttp = new XMLHttpRequest();"
                       "  xhttp.onreadystatechange = function()"
                       "  {"
                       "    if (this.readyState == 4 && this.status == 200) {"
                       "      document.getElementById(\"vcc\").innerHTML = this.responseText;"
                       "    }"
                       "  };"
                       "  xhttp.open(\"GET\", \"vccread\", true);"
                       "  xhttp.send();"
                       "}"
                       "</script>"
                       "</center>"
                       "</body>"
                       "</html>";
```

El código anterior se incluye dentro del sketch global del programa que se cargará en el ESP32. Sin embargo, dentro de la misma carpeta que lo contiene, existe otro archivo, también con terminación '.ino' (HTML.ino), en el que se definirán todas las funciones utilizadas en el desarrollo del código de a la web descrita anteriormente, así como otras utilizadas en la implementación del modo Wi-Fi del sistema.

Seguidamente se muestra el código contenido en el nuevo archivo ‘HTML.ino’:

```
void handleRoot()
{
  server.send(200, "text/html", webpage_start);
}

void vcc_data()
{
  if (millis() >= compare)
  {
    compare = millis() + refresh_time;

    // SENSOR DHT 11
    temperature_ap = dht.readTemperature();
    humidity_ap = dht.readHumidity();

    // SENSOR HC-SR04
    // 1º - Enviamos un pulso generado incialmente con un
    // nivel alto de 1 ms y después, de un nivel bajo
    digitalWrite(TrigPin, HIGH);
    delay(1);
    digitalWrite(TrigPin, LOW);

    // 2º - Obtenemos el tiempo que tarda en respondernos
    // al pulso anterior, mediante el EcoPin
   duracion_ap = pulseIn(EcoPin, HIGH);

    // 3º - Convertimos dicha duración a distancia,
    // dividiendo por una constante de valor 58.2
    distance_ap = duracion_ap / 58.2;

    TIME();

    datos = "<br>";
    datos += "Temperatura: <span style = \"color:orange\">" +
    String(temperature_ap) + " ºC</span>" + "<br>" + "<br>";
    datos += "Humedad: <span style = \"color:#0EBCE7\">" +
    String(humidity_ap) + " %</span>" + "<br>" + "<br>";
    datos += "Distancia: <span style = \"color:green\">" +
    String(distance_ap) + " cm</span>" + "<br>" + "<br>";
    datos += mnj+ "<br>" + "<br>"+ "<br>";
    datos += "<span style = \"color:#97948E\">Seleccionar fm:</span>";
    server.send(200, "text/plane", datos);
  }
}

   void TIME()
   {
     unsigned long tsegundos =  millis() / 1000;
     int horas = (tsegundos / 3600);
     int minutos = ((tsegundos - horas * 3600) / 60);
     int segundos = tsegundos - (horas * 3600 + minutos * 60);
```

```
  mnj = "<br>";
  mnj += "Tiempo activo: ";
  if (horas < 10) mnj += "0";
  mnj += String(horas) + ":";
  if (minutos < 10) mnj += "0";
  mnj += String(minutos) + ":";
  if (segundos < 10) mnj += "0";
  mnj += String(segundos);
}

void refresh_page()
{
  String act_state = server.arg("state");

  if (act_state == "1")
  {
    refresh_time = 800;
  }
  else if (act_state == "2")
  {
    refresh_time = 2800;
  }
  else if (act_state == "3")
  {
    refresh_time = 4800;
  }
  else if (act_state == "4")
  {
    refresh_time = 9800;
  }

  Serial.println(refresh_time);
}
```

Es posible distinguir diferentes partes importantes en el código anterior relacionado con el desarrollo de la web implementada:

1. El código HTML perteneciente al 'String **webpage_start**' contiene la estructura global de la web. Entre sus líneas, merece la pena destacar la definición de los botones destinados a seleccionar el intervalo de actualización de los valores, entre otros componentes:

```
"<h3><button class=\"button\" onclick=\"send(1)\">1 Seg</button>"
"<button class=\"button\" onclick=\"send(2)\">3 Segs</button>"
"<button class=\"button\" onclick=\"send(3)\">5 Segs</button>"
"<button class=\"button\" onclick=\"send(4)\">10 Segs</button><br><br><h3>"
```

2. Dentro del archivo ‘HTML.ino’:

   a) Definición de la función ‘handleRoot()’ utilizada en nuestro sketch dentro del ‘void setup()’ para llevar a cabo la creación de la página.

      En su interior se crea el servidor web pasándole el código HTML perteneciente al ‘String **webpage_start**’ anterior, entre otros parámetros:

```
void handleRoot()
{
  server.send(200, "text/html", webpage_start);
}
```

   b) Función ‘vcc_data()’, utilizada para actualizar el valor de los sensores en la web haciendo uso de los datos recibidos del sistema:

```
void vcc_data()
{
  if (millis() >= compare)
  {
    ...

    TIME();

    datos = "<br>";
    datos += "Temperatura: <span style = \"color:orange\">" + String(temperature_ap) + " °C</span>" + "<br>" + "<br>";
    datos += "Humedad: <span style = \"color:#0EBCE7\">" + String(humidity_ap) + " %</span>" + "<br>" + "<br>";
    datos += "Distancia: <span style = \"color:green\">" + String(distance_ap) + " cm</span>" + "<br>" + "<br>";
    datos += mnj+ "<br>" + "<br>"+ "<br>";
    datos += "<span style = \"color:#97948E\">Seleccionar frecuencia de muestreo:</span>";
    server.send(200, "text/plane", datos);
  }
}
```

      Dicha función además se encarga de actualizar y mostrar por pantalla el tiempo transcurrido desde que se accedió a la web. Para ello, es necesario llamar a la función ‘TIME()’ también descrita en el archivo en cuestión:

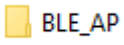
```
void TIME()
{
  unsigned long tsegundos =  millis() / 1000;
  int horas = (tsegundos / 3600);
  int minutos = ((tsegundos - horas * 3600) / 60);
  int segundos = tsegundos - (horas * 3600 + minutos * 60);

  mnj = "<br>";
  mnj += "Tiempo activo: ";
  if (horas < 10) mnj += "0";
  mnj += String(horas) + ":";
  if (minutos < 10) mnj += "0";
  mnj += String(minutos) + ":";
  if (segundos < 10) mnj += "0";
  mnj += String(segundos);
}
```

c) Finalmente ha de destacar el uso de la función ‘refresh_page()’ para llevar a cabo la actualización de los valores de los sensores en la página web cada vez que se pulse uno de los botones descritos anteriormente:

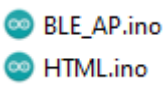
```
void refresh_page()
{
  String act_state = server.arg("state");

  if (act_state == "1")
  {
    refresh_time = 800;
  }
  else if (act_state == "2")
  {
    refresh_time = 2800;
  }
  else if (act_state == "3")
  {
    refresh_time = 4800;
  }
  else if (act_state == "4")
  {
    refresh_time = 9800;
  }

  Serial.println(refresh_time);
}
```


Así, tal y como se ha descrito anteriormente, la organización de archivos a la hora de implementar el modo **Soft AP** y desarrollar una web con cierta complejidad, es el siguiente:

1º. Carpeta que contiene el código del programa:



2º. Estructura interna de archivos:



5.3. IMPLEMENTACIÓN DE LA APLICACIÓN ANDROID

En esta sección se explica el desarrollo de la aplicación móvil desarrollada basada en Android, utilizada para la recepción, gestión, almacenamiento y monitorización de los datos captados por los sensores y enviados a través del módulo ESP32.

5.3.1. DESARROLLO ANDROID BASADO EN FRAGMENTS

¿Qué son los fragments?

Un ***fragment*** es una sección «modular» de interfaz de usuario embebida dentro de una actividad anfitriona, el cual permite versatilidad y optimización de diseño. Se trata de pequeñas actividades contenidas dentro de una actividad anfitriona, manejando su propio diseño (un recurso layout propio) y ciclo de vida. [36]

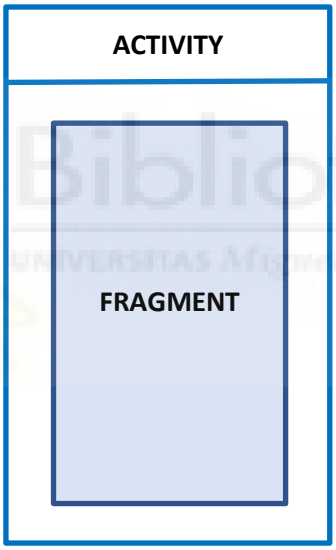


Ilustración 52. Representación gráfica de un fragment

La necesidad de usar *fragments* nace con la versión 3.0 (API 11) de Android debido a los múltiples tamaños de pantalla que estaban apareciendo en el mercado y a la capacidad de orientación de la interfaz. Estas características necesitaban dotar a las aplicaciones Android de la capacidad para adaptarse y responder a la interfaz de usuario sin importar el dispositivo.

Una de las principales características de estos elementos, es la posibilidad que ofrecen de reutilizar código, permitiendo así ahorrar tiempo a la hora de desarrollar la aplicación. Además, facilitan el despliegue de las aplicaciones en cualquier tipo de pantalla y/u orientación.

Otra de las ventajas que ofrecen estos elementos es que permiten crear diseños de interfaces de usuario de múltiples vistas, es decir, que son imprescindibles para generar actividades con diseños dinámicos, como por ejemplo el uso de pestañas de navegación, *expand and collapse*, *stacking*, etc.

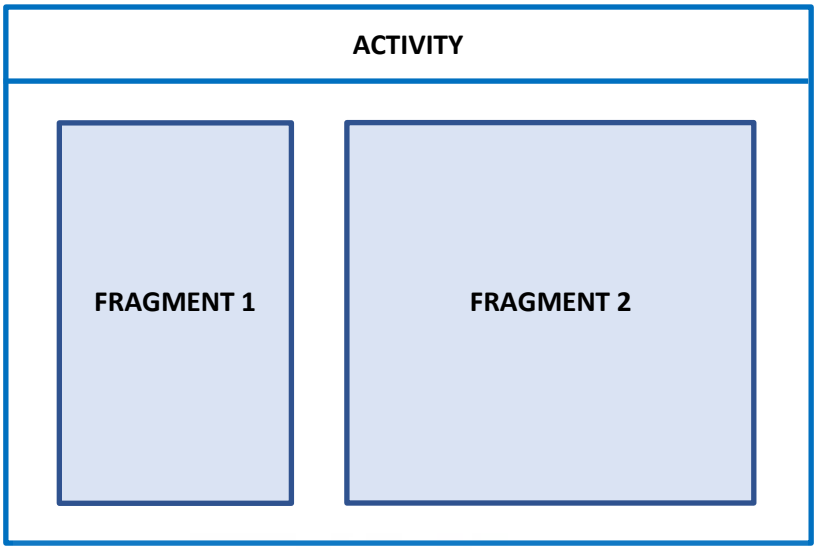


Ilustración 53. Vista con múltiples fragments

Por su parte, el uso de fragments implica que no sean necesarios cambios profundos en la jerarquía de vistas, permitiendo así al programador, evitar numerosos fallos de compilación, diseño y desarrollo

Ciclo de vida de un fragment

El ciclo de vida de un *fragment* depende del ciclo de vida de una actividad y además un *fragment* posee algunas características extras, ya que es muy común alterarlos en tiempo de ejecución. Sin embargo, cada fragment tiene su propio ciclo de vida.

Cuando una actividad está en su estado de reanudación, todos los *fragments* que esta contiene pueden actuar independientemente y pasar a sus otros estados sin ningún problema. Pero si en algún momento la actividad pasa a pausa, entonces todos los *fragments* siguen su comportamiento, al igual que si pasa a detención o se destruye.

Seguidamente se muestra una ilustración correspondiente con el ciclo de vida de un *fragment* mientras su actividad está en ejecución:

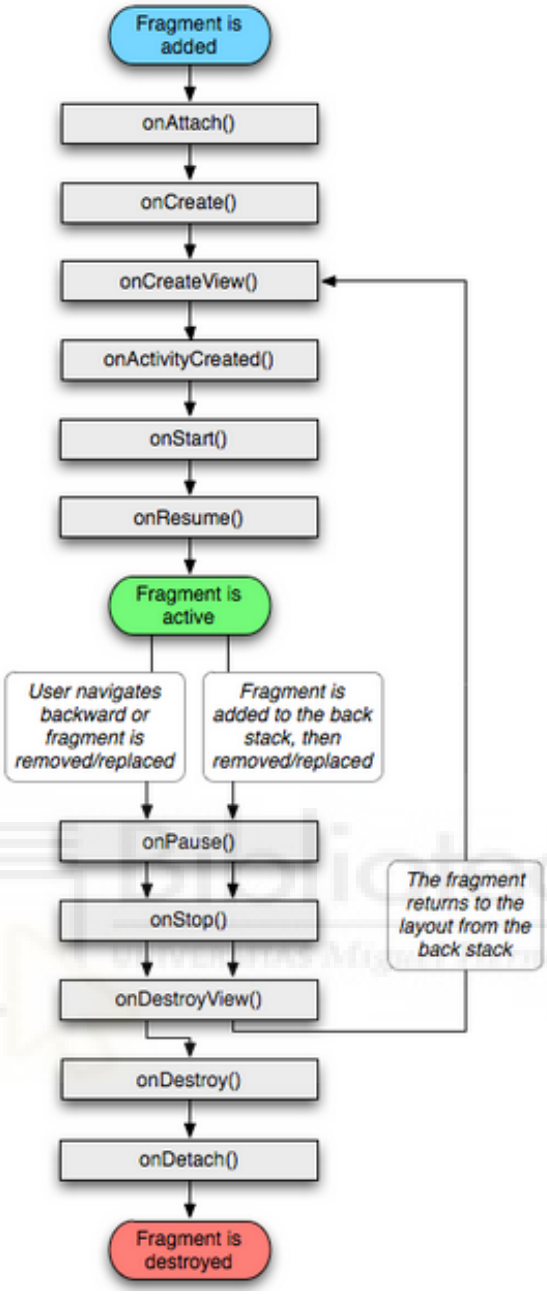


Ilustración 54. Ciclo de vida de un fragment

De la ilustración anterior podemos diferenciar una serie de métodos *callback* adicionales al ciclo de vida de la actividad:

onAttach

Invocado cuando el fragmento ha sido asociado a la actividad anfitriona.

onActiviyCreated

Ejecutado cuando la actividad anfitriona ya ha terminado la ejecución de su método onCreate().

onCreate

Método llamado cuando el fragmento se está creando. En él es posible inicializar todos los componentes que se desean guardar si el fragmento fue pausado o detenido.

onCreateView

Método llamado cuando el fragmento es dibujado por primera vez en la interfaz de usuario. En este método se lleva a cabo la creación del *view* que representa al fragmento para retornarlo hacia la actividad.

onStart

Método llamado cuando el fragmento está visible ante el usuario. Depende del método onStart() de la actividad para saber si la actividad se está mostrando.

onResume

Ejecutado cuando el fragmento está activo e interactuando con el usuario. Esta situación depende de que la actividad anfitriona esté primero en su estado *Resumed*.

onStop

Método llamado cuando un *fragment* ya no es visible para el usuario debido a que la actividad anfitriona está detenida o porque dentro de la actividad se está gestionando una operación de fragmentos.

onPause

Método llamado como el primer indicador de que el usuario está abandonando el fragmento (aunque no siempre significa que el fragmento se esté destruyendo). Generalmente, este es el momento en el que se deben confirmar los cambios que deban conservarse más allá de la sesión de usuario actual.

onDestroyView

Método llamado cuando la jerarquía de vistas a la cual ha sido asociado el fragmento ha sido destruida.

onDetach

Método llamado cuando el fragmento ya no está asociado a la actividad anfitriona.

Generalmente, de los métodos anteriores, el programador debe implementar:

- onCreate()
- onCreateView
- onPause()

5.3.2. ESTRUCTURA DE LA APLICACIÓN

La aplicación desarrollada sigue un patrón modular ***Model View ViewModel* (MVVM)**, en castellano, Modelo Vista VistaModelo, donde es posible diferenciar tres partes:

Modelo

Parte de la estructura constituida por los datos, el estado y la lógica de negocio.

Al no estar vinculado a la vista, es posible reutilizarlo en diferentes contextos.

Vista

Se trata de un componente vinculado con variables “observables” del código y “acciones” expuestas por el ViewModel de forma flexible.

VistaModelo

Es el responsable de ajustar el modelo y preparar los datos observables que necesita la vista. Además, permite que la viste pase eventos al modelo. Sin embargo, no está vinculado a la vista. Podría decirse que se trata de la capa de persistencia de los datos. Se mantiene creado a lo largo de la vida de la aplicación.

Finalmente, el patrón MVVM, vincula los componentes de la IU en los diseños con las fuentes de datos de la aplicación mediante un formato declarativo.

La siguiente ilustración representa el patrón MVVM:

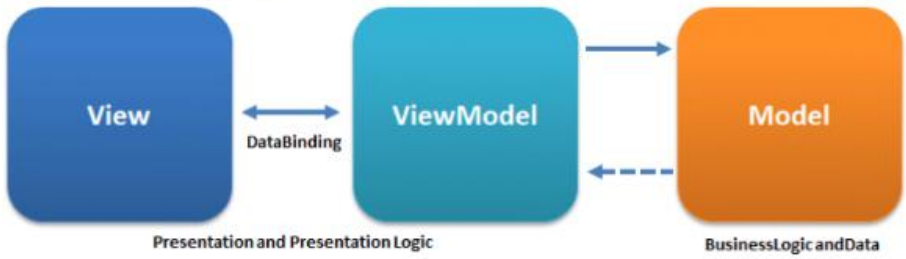


Ilustración 55. Patrón MVVM

5.3.3. GESTIÓN DE PERMISOS

En Android existen diferentes permisos a la hora de programar una aplicación, los cuales, podrían agruparse en tres bloques:

- Permisos en el momento de la instalación
- Permisos de tiempo de ejecución
- Permisos especiales

Respecto al desarrollo BLE, han sido necesarios diferentes permisos relacionados con el Bluetooth y la ubicación. En la parte de desarrollo Wi-Fi, el de acceso a la red. Y finalmente, otros tantos relacionados con el acceso al almacenamiento interno utilizado en la base de datos.

Todos los permisos han de ser declarados en el *manifest* (AndroidManifest.xml), documento en el cual se describe información esencial de la aplicación para las herramientas de creación Android, el sistema operativo Android y Google Play:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Además, algunos de ellos han de ser aceptados por el usuario en tiempo de ejecución, como es el caso de la ubicación y del acceso al almacenamiento. Para ello, una buena forma de hacerlo es solicitarlo según se lance la aplicación.

El siguiente código muestra el algoritmo utilizado en la solicitud del permiso de ubicación:

```
if(ActivityCompat.checkSelfPermission( context: this, Manifest.permission.ACCESS_FINE_LOCATION)
        != PackageManager.PERMISSION_GRANTED){
    listaPermisos.add(Manifest.permission.ACCESS_FINE_LOCATION);
}

if(listaPermisos.size() != 0){
    ActivityCompat.requestPermissions( activity: this, listaPermisos.toArray(new String[0]), requestCode: 0);
}
```

Una vez ejecutada la aplicación, en caso de no contar con el permiso anterior, aparecerá el siguiente mensaje:

Ilustración 56. Permiso ubicación

5.3.4. FRAGMENTS DE LA APLICACIÓN

Seguidamente se lleva a cabo un análisis de los diferentes *fragments* de la aplicación:

WelcomeFragment

Posiblemente se trate del *fragment* más sencillo de la aplicación. Su código sencillo y rápida implementación hacen de este componente un un elemento cuya única finalidad es hacer las veces de anfitrión de la aplicación:

```
public class WelcomeFragment extends Fragment {

    public WelcomeFragment() {
        // Required empty public constructor
    }

    public static WelcomeFragment newInstance(String param1, String param2) {
        WelcomeFragment fragment = new WelcomeFragment();
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_welcome, container,  attachToRoot: false);
    }
}
```

Su *layout* asociado consta de un *textview*, una *imageview* y un gif:

```
<TextView
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:gravity="center"
    android:text="ESP32 - BLE/WiFi"
    android:textColor="@color/white"
    android:textSize="35sp"
    android:translationZ="1dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

```
        <ImageView
            android:id="@+id/imageView8"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:adjustViewBounds="true"
            android:src="@drawable/sensores_def"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    <pl.droidsonroids.gif.GifImageView
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        app:layout_constraintTop_toBottomOf="@id/imageView8"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:src="@drawable/lens">
```

ScannerFragment

Fragment utilizado para llevar a cabo toda la conectividad BLE.

En un primer lugar es necesario obtener el adaptador del dispositivo, así como instanciar un objeto de tipo 'scanner' que se utilizará para llevar a cabo las tareas relacionadas con la conexión:

```
// Obtenemos el adaptador Bluetooth (solo hay uno en el sistema)
bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

// Instanciamos un objeto de tipo Scanner para el adaptador creado anteriormente
scanner = bluetoothAdapter.getBluetoothLeScanner();
```

Seguidamente es necesario definir unos filtros y características de escaneo:

```
    scanSettings = new ScanSettings.Builder()
            .setScanMode(ScanSettings.SCAN_MODE_LOW_POWER)
            .setCallbackType(ScanSettings.CALLBACK_TYPE_ALL_MATCHES)
            .setMatchMode(ScanSettings.MATCH_MODE_AGGRESSIVE)
            .setNumOfMatches(ScanSettings.MATCH_NUM_ONE_ADVERTISEMENT)
            .setReportDelay(0L)
            .build();
```

Además, es necesario configurar un *CallBack* que se ejecute cada vez que se encuentre un dispositivo que coincida con los criterios de búsqueda anteriores:

```
private final ScanCallback scanCallback = new ScanCallback() {

    @SuppressLint("MissingPermission")
    @Override
    public void onScanResult(int callbackType, ScanResult result) {
        BluetoothDevice device = result.getDevice();
        if (result.getScanRecord() != null && result.getDevice() != null) {
            viewModel.addDevice(new DeviceModel(device.getName(),device.getAddress(), conectado: false));
        }
    }

    @Override
    public void onBatchScanResults(List<ScanResult> results) { Integer integer =0; }

    @Override
    public void onScanFailed(int errorCode) { Integer integer =0; }
};
```

Tras pulsar sobre el botón ‘Start Scanner’, se llevará a cabo un escaneo. En caso de estar conectado a una red Wi-Fi, aparecerá un ‘AlertDialog’ utilizado para indicar que es necesario desconectarse de esta para evitar así un posible conflicto de intereses Wi-Fi/BLE. En caso de no estar conectado, comenzará el escaneo de dispositivos haciendo uso del método ‘**iniciarScanner()**’:

```
binding.btnStart.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        ConnectivityManager connectivityManager = (ConnectivityManager) getActivity()
                .getSystemService(getContext().CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connectivityManager.getActiveNetworkInfo();

        if(networkInfo != null && networkInfo.getType() == ConnectivityManager.TYPE_WIFI){

            AlertDialog.Builder builder = new AlertDialog.Builder(getContext());
            AlertDialog alert = builder.setTitle("¡Atención!")
                    .setMessage("Si está conectado a la red WiFi, desconéctese")
                    .setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                        }
                    })
                    .setNegativeButton("Abrir ajustes", new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            startActivity(new Intent(Settings.ACTION_BLUETOOTH_SETTINGS));
                        }
                    }).create();
            alert.show();

        }else{
            iniciarScanner();
        }
    }
});
```

Método ‘iniciarScanner()’ definido anteriormente:

```
@SuppressLint("MissingPermission")
private void iniciarScanner(){
    if(comprobarRequisitosPrevios()){
        if (scanner == null){
            scanner = bluetoothAdapter.getBluetoothLeScanner();
        }

        scanner.startScan( filters: null, scanSettings, scanCallback);
        Toast.makeText(getActivity().getApplicationContext(),
                "Escaneo iniciado", Toast.LENGTH_SHORT).show();
    }
}
```

Dicho método comprueba primeramente si se ha creado previamente un objeto de tipo ‘scanner’ para poder iniciar un escaneo. En caso de no estarlo, lo crea. Finalmente se inicia un escaneo con las características definidas anteriormente y el *CallBack* asociado.

Por su parte, cada vez que se encuentre un dispositivo que cumpla con los criterios de búsqueda anteriores, será añadido al ‘RecyclerView’ del fragment.

Un ‘RecyclerView es un ‘ViewGroup’ flexible que permite mostrar un conjunto de datos con capacidad de *scroll*.

Los ‘RecyclerViews’ tienen asociado un ‘adaptador’ cuya función es hacer de puente entre los datos y las vistas contenidas en la lista. Además del ‘adaptador’, los ‘RecyclerViews’ llevan asociado un ‘item’ de tipo .xml utilizado para definir la vista dentro de la lista del propio ‘Recycler’ (**el procedimiento a la hora de trabajar con cualquier ‘RecyclerView’ es siempre el mismo**):

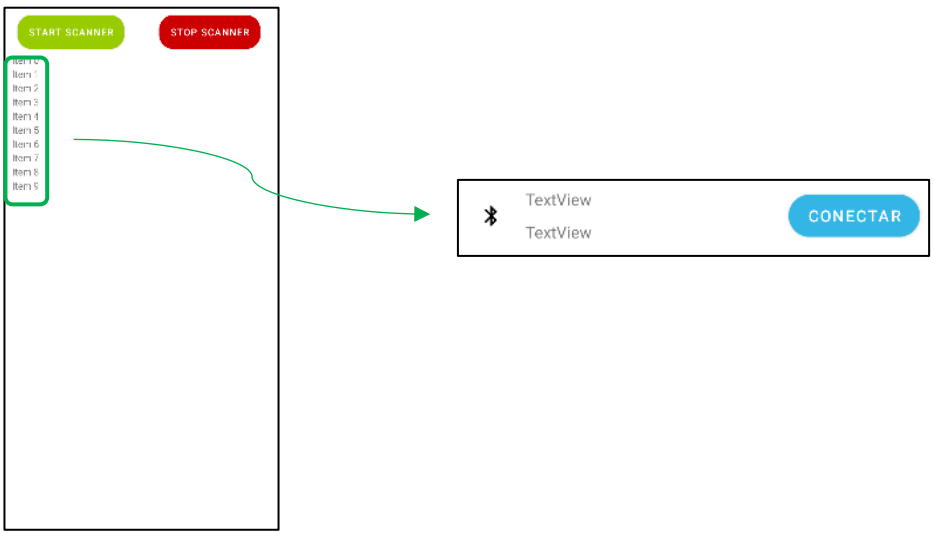


Ilustración 57. Formato RecyclerView

Así, cada vez que se encuentre un dispositivo, se añadirá en el 'RecyclerView' indicando su nombre y su dirección MAC:

```
viewModel.addDevice(new DeviceModel(device.getName(),device.getAddress(), conectado: false));
```

Una vez se muestren por pantalla los diferentes dispositivos encontrados durante el escaneo, será posible conectarse al que el usuario seleccione:

```
private void conectarDispositivo(int position){
    BluetoothDevice device = bluetoothAdapter.getRemoteDevice(viewModel
            .getDevices().getValue().get(position).getDireccionMac());
    connectedDevice = device;
    @SuppressLint("MissingPermission") BluetoothGatt gatt = device
            .connectGatt(getContext(), autoConnect: false, bluetoothGattCallback, TRANSPORT_LE);
}
```

Cada dispositivo encontrado posee un *Callback* (bluetoothGattCallback) asociado con diferentes métodos relacionados de la conexión, entre los cuales, conviene destacar el onServiceDiscovered(), utilizado para obtener los servicios del dispositivo y permitir, tras haber realizado la conexión, desplazarse al siguiente *fragment (*debido a la extensión de dicho método se sintetizan las partes más importantes, correspondientes con la obtención de los servicios y el 'AlertDialog' para cambiar de fragment):

Obtención de los servicios:

```
deviceConnectViewModel = new ViewModelProvider(requireActivity()).
        get(DeviceConnectViewModel.class);
deviceConnectViewModel.setBluetoothGatt(gatt);
ArrayList<GattServiceModel> gattServiceModels = new ArrayList<>();

for (int i = 2; i < services.size(); i++){
    gattServiceModels.add(new GattServiceModel(services.get(i)));
}

deviceConnectViewModel.setServices(gattServiceModels);
```

‘AlertDialog’ de cambio de fragment:

```
viewModel.connectDevice(connectedDevice.getAddress());
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
builder.setTitle("Conexión BLE realizada");
builder.setMessage("¿Desea ir a la pantalla 'Dispositivo conectado'?");
builder.setIcon(getContext().getDrawable(R.drawable.escaneo_ble));
builder.setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        // User clicked OK button
        Navigation.findNavController(getView()).navigate(R.id.device_connect_dest,
                args: null, new NavOptions.Builder()
                        .setPopUpTo(Navigation.findNavController(getView())
                                .getGraph().getStartDestinationId()
                                , inclusive: true).build());
        //Navigation.findNavController(getView()).navigate(R.id.device_connect_dest);
    }
});
builder.setNegativeButton("Cancelar", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        // User cancelled the dialog
    }
});
AlertDialog dialog = builder.create();
dialog.show();
```

Tras seleccionar la opción de ir a la ‘Pantalla de dispositivo conectado’ el usuario cambiará de *fragment*.

Se llevará a cabo un análisis en mayor profundidad de las diferentes pantallas relacionadas, en el apartado ‘6. RESULTADOS’.

DeviceConnectFragment

La funcionalidad de este fragment es la monitorización como tal de los valores de los sensores recibidos del módulo ESP32 haciendo uso de una conexión BLE.

Debido a su extensión, se han optado por diferenciar tres de los métodos más importantes.

Por una parte, el método ‘**readCharacteristic()**’, utilizado para la lectura de características:

```
@SuppressLint("MissingPermission")
public void readCharacteristic(BluetoothGattCharacteristic bluetoothGattCharacteristic){
    readCharacteristicQueue.add(new Runnable() {
        @Override
        public void run() {
            viewModel.getBluetoothGatt().readCharacteristic(bluetoothGattCharacteristic);

        }
    });
}
```

Por otra parte, el método '**continuousReadingCharacteristic()**', utilizado para la lectura continua de las características:

```
public void continuousReadingCharacteristic(){
    if(viewModel.getBluetoothGatt() != null) {
        for (GattServiceModel service : viewModel.getServices()) {
            if (service.isActivado()) {
                readCharacteristic(service.getService().getCharacteristics().get(0));
            }
        }
        nextCommand();

        bleHandler.postDelayed(runContinuousReading,
                (long) (viewModel.getFrecuenciaMuestreo().getValue()*1000));
    }
}
```

Finalmente, el método '**nextCommand()**', utilizado para ir seleccionando características de la cola implementada en caso de que se acumulen:

```
private void nextCommand(){
    if(readCharacteristicQueueBusy){
        return;
    }

    if (readCharacteristicQueue.size() > 0) {
        final Runnable bluetoothCommand = readCharacteristicQueue.poll();
        readCharacteristicQueueBusy = true;

        bleHandler.post(new Runnable() {
            @Override
            public void run() {
                try {
                    bluetoothCommand.run();
                } catch (Exception ex) {
                    // Log.e(TAG, String.format("ERROR: Command exception
                    readCharacteristicQueueBusy = false;
                    nextCommand();
                }
            }
        });
    }
}
```

Se llevará a cabo un análisis en mayor profundidad de las diferentes pantallas relacionadas, en el apartado '6. RESULTADOS'.

SensorReadDataBaseFragment. Filtro y borrado de la base de datos

Dentro del 'SensorReadDataBaseFragment' se han implementado dos funcionalidades adicionales:

- Filtro de búsqueda.
- Borrado de la base de datos.

Por una parte, el filtrado de la base de datos se realiza haciendo uso de una *activity* llamada '**ReadSensorFilterAlertDialog**' ligada a este *fragment*, la cual hace las veces de un 'AlertDialog' pero de una manera más compleja.

Primeramente, dentro del método 'onCreate()' se llama a un método definido dentro de la propia clase, denominado 'cargarFiltros()', encargado de inicializar los valores por defecto de las vistas de los filtros (sensores seleccionados y fecha desde/hasta):

```
private void cargarFiltros(){
    if(filterModel.isSensor1()){
        binding.swSensor1.setChecked(true);
    }
    if(filterModel.isSensor2()){
        binding.swSensor2.setChecked(true);
    }
    if(filterModel.isSensor3()){
        binding.swSensor3.setChecked(true);
    }

    if(filterModel.getDateDesde() != null){
        binding.etDateDesde.setText(DateConverter.getFecha(filterModel.getDateDesde()));
    }else{
        binding.etDateDesde.setText("");
    }
    if(filterModel.getDateHasta() != null){
        binding.etDateHasta.setText(DateConverter.getFecha(filterModel.getDateHasta()));
    }else{
        binding.etDateHasta.setText("");
    }
```

En segundo lugar, se define el estado de los tres *switches* correspondientes con los sensores que se desean filtrar. El siguiente código muestra la activación del primero de ellos, siendo un proceso análogo el seguido para la activación de los otros dos:

```
binding.swSensor1.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked) {
        filterModel.setSensor1(isChecked);
    }
});
```

Adicionalmente, también se declaran un par de métodos dentro de la misma clase. El primero de ellos utilizado para reiniciar los filtros de búsqueda:

```
private void resetFilters(){
    filterModel = new FilterModel();
    cargarFiltros();
}
```

El segundo de ellos, utilizado para actualizar los valores del filtrado (fecha desde/hasta y número de elementos a mostrar):

```
private boolean actualizarFiltro(){
    if (!binding.etDateDesde.getText().toString().equals("")){
        filterModel.setDateDesde(DateConverter.getDateFromString(binding.etDateDesde.getText().toString()));
    }else{
        filterModel.setDateDesde(null);
    }

    if (!binding.etDateHasta.getText().toString().equals("")){
        filterModel.setDateHasta(DateConverter.
                getDateWithHourFromString( value: binding.etDateHasta.getText().toString() + "/23:59:59"));
    }else{
        filterModel.setDateHasta(null);
    }

    if (!binding.etNumElements.getText().toString().equals("")){
        filterModel.setNumElementos(Integer.parseInt(binding.etNumElements.getText().toString()));
    }else{
        return false;
    }

    return true;
}
```

Finalmente, dentro de dicha actividad, se definen un par de botones utilizados para lanzar el filtrado o reiniciarlo:

```
binding.btnSetFiltro.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        if(actualizarFiltro()){
            Intent intent = new Intent();
            intent.putExtra( name: "filtro", filterModel);
            setResult(RESULT_OK, intent);
            finish();
        }
    }
});

binding.btnResetFiltro.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) { resetFilters(); }
});
```

Por otra parte, es posible llevar a cabo un borrado de la base de datos desde el 'SensorReadDataBaseFragment', esta vez, sin la necesidad de crear una *activity* ligada a este.

Así, gestionando el botón de borrado de la base de datos como un 'AlertDialog' convencional, será posible llevar a cabo la eliminación de los valores almacenados:

```
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
builder.setTitle("¿Desea eliminar la base de datos?")
        .setMessage("¡ATENCIÓN!, la base de datos se eliminará")
         setIcon(R.drawable.ic_warning)
        .setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
                viewModel.deleteTable();
                Toast.makeText(getActivity(), text: "Base de datos eliminada", Toast.LENGTH_SHORT).show();
            }
        })
        .setNegativeButton("Cancelar", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
                // User cancelled the dialog
            }
        }).create().show();
return true;
```

Como puede apreciarse, ligado al botón de 'Aceptar', queda definido el comando que elimina por completo la base de datos del dispositivo, con su correspondiente mensaje 'Toast' indicativo:

```
viewModel.deleteTable();
Toast.makeText(getActivity(), text: "Base de datos eliminada", Toast.LENGTH_SHORT).show();
```

Vistos ambos procesos, se aprecia que el relacionado con el filtrado de la base de datos resulta más complejo y elaborado que el de borrado, principalmente porque plantea la necesidad de aplicar los cambios de una actividad en un fragment diferente.

```
ActivityResultLauncher<Intent> alertDialogResult = registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        new ActivityResultCallback<ActivityResult>() {
            @Override
            public void onActivityResult(ActivityResult result) {
                if (result.getResultCode() == Activity.RESULT_OK) {
                    // There are no request codes
                    Intent data = result.getData();

                    FilterModel filterModel = (FilterModel) data.getSerializableExtra( name: "filtro");
                    viewModel.setFilterModel(filterModel);
                    viewModel.obtenerDatosFiltrados(getActivity());
                }
            }
        });
```

Por su parte, el *fragment* anterior tendrá la funcionalidad de mostrar los diferentes elementos filtrados de la actividad anterior:

```
private void initRecyclerView (){
    adapter = new SensorReadAdapter(sensorReadModelList);
    binding.rvSensorRead.setAdapter(adapter);
    binding.rvSensorRead.setLayoutManager(new LinearLayoutManager(getActivity()));
    adapter.notifyDataSetChanged();
}
```

Se llevará a cabo un análisis en mayor profundidad de las diferentes pantallas relacionadas, en el apartado '6. RESULTADOS'.

WiFiFragment

El reducido tamaño de este *fragment* permite mostrarlo al completo:

```
import ...

public class WiFiFragment extends Fragment {

    private FragmentWiFiBinding binding;
    private final String url = "http://192.168.4.1";

    public WiFiFragment() {
        // Required empty public constructor
    }

    @Override
    public void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        binding = FragmentWiFiBinding.inflate(inflater,container, attachToParent: false);

        binding.btnAbrirWiFi.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                startActivity(new Intent(Settings.ACTION_WIFI_SETTINGS));
            }
        });

        binding.btnComprobarSensores.setOnClickListener(new View.OnClickListener() {
            @SuppressLint("MissingPermission")
            @Override
            public void onClick(View view) {

                BluetoothAdapter bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
```

```
            if(bluetoothAdapter != null && bluetoothAdapter.isEnabled()){

                AlertDialog.Builder builder = new AlertDialog.Builder(getContext());
                AlertDialog alert = builder.setTitle("¡Atención!")
                        .setMessage("Si está conectado el adaptador Bluetooth del dispositiv...")
                        .setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {
                            public void onClick(DialogInterface dialog, int id) {
                            }
                        })
                        .setNegativeButton("Abrir ajustes", new DialogInterface.OnClickListener() {
                            public void onClick(DialogInterface dialog, int id) {
                                startActivity(new Intent(Settings.ACTION_BLUETOOTH_SETTINGS));
                            }
                        }).create();
                alert.show();

            }else{
                Uri uri = Uri.parse(url);
                Intent intent = new Intent(Intent.ACTION_VIEW, uri);
                startActivity(intent);
            }
        }
    });
```

A la hora de comprobar el estado de los sensores haciendo uso de una red Wi-Fi, se sigue un proceso muy similar al del 'ScannerFragment'.

Por una parte, se le indica al usuario que desconecte el Bluetooth en caso de tenerlo conectado para evitar un conflicto de interés.

Seguidamente, en caso de no tener una conexión BLE activa, se abre una nueva ventana fuera de la aplicación, haciendo uso del navegador web por defecto del dispositivo, para acceder a la url definida con anterioridad en el mismo fragment donde se mostrarán los datos:

```
private final String url = "http://192.168.4.1";
```

Se llevará a cabo un análisis en mayor profundidad de las diferentes pantallas relacionadas en el apartado '6. RESULTADOS'.

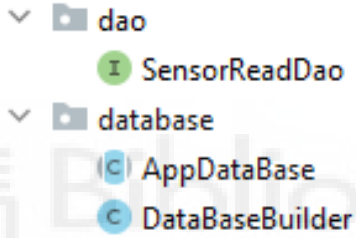
5.3.5. DISEÑO DE LA BASE DE DATOS

Para almacenar los valores obtenidos de los sensores, se ha optado por hacer uso de una base de datos local mediante el motor de base de datos **SQLite**.

En primer lugar, es necesario añadir en el Gradle del proyecto las dependencias de la librería 'Room', la cual permite simplificar la tarea de trabajar con bases de datos en Android, implementando una capa intermedia entre la base de datos y el resto de la aplicación:

```
// Room -> librería utilizada para implementar bases de datos
def room_version = "2.4.2"
implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"
```

Seguidamente, se muestra la estructura de organización de los archivos relacionados:



Por una parte, la clase '**AppDataBase**' será la encargada de proveer diferentes herramientas para poder acceder a la base de datos:

```
package com.example.desarrollo_ble.database;

import androidx.room.Database;
import androidx.room.RoomDatabase;
import androidx.room.TypeConverters;

import com.example.desarrollo_ble.auxiliares.DateConverter;
import com.example.desarrollo_ble.dao.SensorReadDao;
import com.example.desarrollo_ble.model.SensorReadModel;

@Database(entities = {SensorReadModel.class},version = 1)
@TypeConverters({DateConverter.class})
public abstract class AppDataBase extends RoomDatabase {
    public abstract SensorReadDao sensorReadDao();
}
```

El comando '@Database' será el encargado de la declaración de la base de datos con las diferentes tablas y entidades (entities) que se definan. En este caso, únicamente 'SensorReadModel' con los campos asociados:

Por otro lado, la clase '**DataBaseBuilder'**, será la encargada de crear la base de datos en caso de que esta no exista a partir de lo declarado en '@Database':

```
public class DataBaseBuilder {
    private static AppDataBase dataBase = null;

    public static AppDataBase getAppDataBase(Context context){
        if(dataBase == null){
            dataBase = Room.databaseBuilder(context, AppDataBase.class,
                    name: "BLE-Database").build();
        }
        return dataBase;
    }
}
```

Por su parte, también se implementa la interfaz '**SensorReadDao'**, lo que permitirá codificar diferentes métodos para acceder a la base de datos:

```
@Dao
public interface SensorReadDao {

    @Query("SELECT * FROM SensorReadModel ORDER BY dateTime DESC")
    List<SensorReadModel> getAll();

    @RawQuery
    List<SensorReadModel> getAllFiltered(SupportSQLiteQuery query);

    @Insert
    void insertRead(SensorReadModel sensorReadModel);

    @Query("DELETE FROM SensorReadModel")
    void deleteTable();
}
```

El uso de 'Room' facilita la creación de los métodos anteriores, permitiendo así no tener que ejecutar el código SQL, como en los siguientes casos:

- **@RawQuery**: consultas en tiempo de ejecución.
- **@Insert**: inserción básica de objetos en la tabla.

Sin embargo, algunos de ellos, al tratarse de operaciones más complejas, es preciso indicarle al código SQL qué se debe ejecutar en cada una:

- **@Query**: 'SELECT' ordenado de la base de datos.
- **@Query**: 'DELETE' de la tabla completa.

Finalmente, en el 'SensorReadViewModel' se guardará un objeto 'dao' (objeto que permite acceder a los datos) de la base de datos para posteriormente utilizarlo para hacer accesos a esta. El método 'getAppDataBase' devolverá un objeto 'AppDataBase' (ya creado o uno nuevo), del cual se obtendrá el 'dao':

```
sensorReadDao = DataBaseBuilder.getAppDataBase(getApplication().
        getApplicationContext()).sensorReadDao();
```

5.3.6. OTRAS FUNCIONALIDADES

Además de las características principales de la aplicación mencionadas anteriormente, conviene hacer referencia a diferentes hitos relacionados:

Splash screen

Un *Splash Screen* consiste en una pantalla inicial que muestra el logotipo de la aplicación durante uno o varios segundos y que puede o no realizar ciertas operaciones, como, por ejemplo, la carga de datos durante ese tiempo, para posteriormente realizar una transición hacia la aplicación en sí. [37]

Por una parte, es necesario crear un archivo .xml con el icono a mostrar en la pantalla:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@color/white"/>
    <item
        android:width="200dp"
        android:height="200dp"
        android:gravity="center">
        <bitmap
        android:gravity="fill"
        android:src="@drawable/icono_app_splash"/>
    </item>
</layer-list>
```

Seguidamente, se creará un tema relacionado con el icono anterior, configurando una serie de campos como el icono que implementa (el desarrollado anteriormente) y el color de fondo de la pantalla:

```
<!-- Tema Splash Screen -->
<style name="Theme.Splash" parent="Theme.Desarrollo_BLE.NoActionBar" >
    <item name="android:windowBackground">@drawable/splash_screen</item>
    <item name="colorPrimaryVariant">#FFFFFF</item>
</style>
```

Por su parte, se debe configurar que aparezca este tema como parte de la 'MainActivity' de manera temporal. En este caso, se ha optado por una duración de 2 segundos:

```
try {
    Thread.sleep( millis: 2000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
setTheme(R.style.Theme_Desarrollo_BLE_NoActionBar);
super.onCreate(savedInstanceState);
```

Además, también deberá indicarse en el *manifest* el tema inicial de la aplicación para que toda la configuración anterior pueda llevarse a cabo:

```
<activity
    android:name=".view.MainActivity"
    android:exported="true"
    android:theme="@style/Theme.Splash">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Fichero uuid_data.json

En un intercambio de datos haciendo uso de la comunicación BLE, unos de los valores leídos del controlador es la UUID de los servicios y de las características implementadas. Sin embargo, el formato de estas UUIDs no es aparentemente legible.

Haciendo uso de un archivo .json, llamado en la aplicación **uuid_data.json**, se ha creado una "especie de diccionario" en el cual se han introducido parejas clave-valor (key-value), siendo el primer parámetro la UUID de los diferentes servicios, características y unidades, y el segundo, el valor de esas unidades y los "nombres" para reconocer dichos elementos que nos proporciona el BLE:

```
{
  "2d22c77b-dbc3-4fb1-ba2a-8dfdeb115d6e": "Sensor 1",
  "7f425374-c825-41db-9cbd-c72bd17a8797": "Característica 1",
  "7f425374-c825-41db-9cbd-c72bd17a8797-unit": "°C",
  "f1c524f4-9867-4983-8eb9-5f81577cf2c2": "Sensor 2",
  "6fb17f9e-4c73-45a6-81fb-cb7dc5a7b108": "Característica 2",
  "6fb17f9e-4c73-45a6-81fb-cb7dc5a7b108-unit": "%RH",
  "3460f0df-1f3d-4eee-b785-2fc9fec9d556": "Sensor 3",
  "b3bf19d2-070d-481e-a7a8-e00daa48f6de": "Característica 3",
  "b3bf19d2-070d-481e-a7a8-e00daa48f6de-unit": "cm",
  "af5d710f-f0b4-4804-84c6-d0563c932b0e": "Servicio 4",
  "b0167c1c-91d6-4003-996b-4ec293acd456": "Característica 4"
}
```

Así, el primer ejemplo de estas parejas tendría por clave "2d22c77b-dbc3-4fb1-ba2a-8dfdeb115d6e" y por valor asociado "Sensor 1", mientras que el resto de parejas corresponderían al sensor 2, sensor 3 y la frecuencia de muestreo con sus UUIDs asociadas.

Por su parte, ligado al archivo .json anterior, se ha creado una clase auxiliar '**UuisListSingleton**' utilizada para leer de este:

```
public class UuidListSingleton {
    private static UuidListSingleton uuidList;
    private Map<String, String> uuidMap = new HashMap<>();

    public UuidListSingleton(Context context) {
        InputStream inputStream = context.getResources().
                openRawResource(R.raw.uuid_data);
        BufferedReader buffer = new BufferedReader
                (new InputStreamReader(inputStream));
```

```
            String cadena = "";
            String json = "";

            while(cadena != null){
                try {
                    json += cadena;
                    cadena = buffer.readLine();
                } catch (IOException e) {
                    cadena = null;
                }
            }
            uuidMap = new Gson().fromJson(json,uuidMap.getClass());
        }

    public Map<String, String> getUuidMap() { return uuidMap; }

    /**
     * Método utilizado para crear un objeto de la clase
     * UuidListSingleton en caso de no estar ya creado previamente
     * @return objeto del tipo UuidListSingleton
     */
    public static UuidListSingleton getUuidListSingleton(Context context){
        if(uuidList == null){
            uuidList = new UuidListSingleton(context);
        }
        return uuidList;
    }
}
```

Dicha clase es utilizada para evitar leer continuamente del objeto .json puesto que este se encuentra guardado en memoria. Así, se guardaría inicialmente una instancia de este objeto para posteriormente devolverla siempre que se necesite.

Por una parte, es utilizado en el método 'onBindViewHolder' del '**ServiceAdapter**' en el proceso encargado de mostrar en el 'RecyclerView' los diferentes servicios que posee el sistema:

```
@Override
public void onBindViewHolder(@NonNull ServiceAdapter.ViewHolder holder, int position) {
    GattServiceModel service = gattServiceModels.get(position);
    holder.tvSensor.setText(UuidListSingleton.getUuidListSingleton(context)
            .getUuidMap().get(service.getService().getUuid().toString()));

    if(service.isActivado()){
        holder.swService.setChecked(true);
    }
}
```

Por otra parte, en el método ‘onCharacteristicRead’ del ‘BluetoothGattCallback’ en el proceso encargado de leer las características de los sensores:

```
public void onCharacteristicRead (BluetoothGatt gatt, final BluetoothGattCharacteristic characteristic, int status){
    if(status == GATT_SUCCESS){

        activity.runOnUiThread(new Runnable() {
            @Override
            public void run() {
                if(deviceConnectViewModel.getFrecuenciaMuestreo().getValue() == -1){
                    deviceConnectViewModel.setFrecuenciaMuestreo(Double.parseDouble(characteristic.getStringValue( offset: 0))/1000);
                }else {

                    SensorReadModel sensor = new SensorReadModel(characteristic.
                            getUuid().toString(),
                            UuidListSingleton.getUuidListSingleton(activity).getUuidMap()
                                    .get(characteristic.getService().getUuid().toString()),
                            String.format("%.2f",ByteBuffer.wrap(characteristic.getValue())
                                    .order(ByteOrder.LITTLE_ENDIAN).getFloat()),
                            UuidListSingleton.getUuidListSingleton(activity).getUuidMap()
                                    .get(characteristic.getUuid().toString() + "-unit"),
                            new java.sql.Date(Calendar.getInstance().getTime().getTime()));

                    deviceConnectViewModel.addLectura(sensor);
                }
            }
        });
    }
}
```

Traducción de los diferentes campos de la aplicación al inglés

Dados los diferentes textos que aparecen en la aplicación (títulos, nombre de los botones, filtro de la base de datos etc.), inicialmente escritos en castellano, se ha introducido la posibilidad de una traducción automática al inglés.

Para ello, se ha creado un archivo **strings.xml (en)** en el que se traducen todos los textos utilizados del castellano al inglés.

El siguiente código muestra un ejemplo de algunas de estas traducciones:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">BLE and AP sensors</string>

    <!-- Menú -->
    <string name="conexiones_ble">BLE connexions</string>
    <string name="escaneo_dispositivos">Device scanning</string>
    <string name="dispositivo_conectado">Device connected</string>
    <string name="sensor_read_db">Check historial</string>
    <string name="opciones">Options</string>
```

Así, en caso de que el usuario cambie la configuración de su dispositivo (estableciendo el inglés como idioma predeterminado), al abrir de nuevo la aplicación, los textos de esta aparecerán traducidos.

DateConverter

Los dato de tipo ‘date’ utilizados en la aplicación no son objetos básicos. Sin embargo, es posible expresarlos como un tipo ‘long’.

La clase ‘**DateConverter**’ es utilizada para llevar a cabo conversiones específicas entre distintos tipos de dato, como por ejemplo, cuando es necesario convertir un ‘date’ en un ‘long’ para poder almacenarlo en la base de datos, ya que ‘Room’ no trabaja con datos complejos.

De manera análoga, también permite convertir datos de tipo ‘long’ a tipo ‘date’, como por ejemplo, cuando se obtiene información de la base de datos.

El siguiente código muestra la implementación de la clase, indicando los métodos diseñados:

```
public class DateConverter {
    private static DateFormat dfFechaHora = new SimpleDateFormat( pattern: "HH:mm:ss / dd-MM-YY");
    private static DateFormat dfHora = new SimpleDateFormat( pattern: "HH:mm:ss");
    private static DateFormat dfFecha = new SimpleDateFormat( pattern: "dd/MM/yy");
    private static DateFormat getDfFechaHoraFiltro = new SimpleDateFormat( pattern: "dd/MM/yy/HH:mm:ss");

    @TypeConverter
    public static Date fromTimestamp(Long value) { return value == null ? null : new Date(value); }
    @TypeConverter
    public static Long dateToTimestamp(Date date) { return date == null ? null : date.getTime(); }

    public static String getFechaHoraFromDate(Date value) { return dfFechaHora.format(value); }

    public static String getHoraFromDate(Date value) { return dfHora.format(value); }

    public static String getFecha(Date value){ return dfFecha.format(value); }

    public static Date getDateFromString(String value){
        try {
            return dfFecha.parse(value);
        } catch (ParseException e) {
            return null;
        }
    }

    public static Date getDateWithHourFromString(String value){
        try {
            return getDfFechaHoraFiltro.parse(value);
        } catch (ParseException e) {
            return null;
        }
    }
```

6. RESULTADOS

En este sexto capítulo se pretenden mostrar los resultados del sistema hardware desarrollado, compuesto por los sensores, cableado, resistencias y módulo ESP32 descritos anteriormente. Además, se llevará a cabo un recorrido por las diferentes vistas de la aplicación.

6.1. SISTEMA HARDWARE

En esta sección se mostrará en detalle el sistema hardware del proyecto.

Tal y como se ha mencionado con anterioridad, el fin del sistema implementado es la monitorización de diferentes parámetros medioambientales, como la temperatura, la humedad y la distancia a objetos, haciendo uso de sensores de bajo coste. En su desarrollo, se han utilizado los siguientes elementos:

- Sensor de temperatura
- Sensor de humedad
- Módulo ESP32
- Resistencias
- Cableado

Seguidamente se muestra una vista general del sistema final:

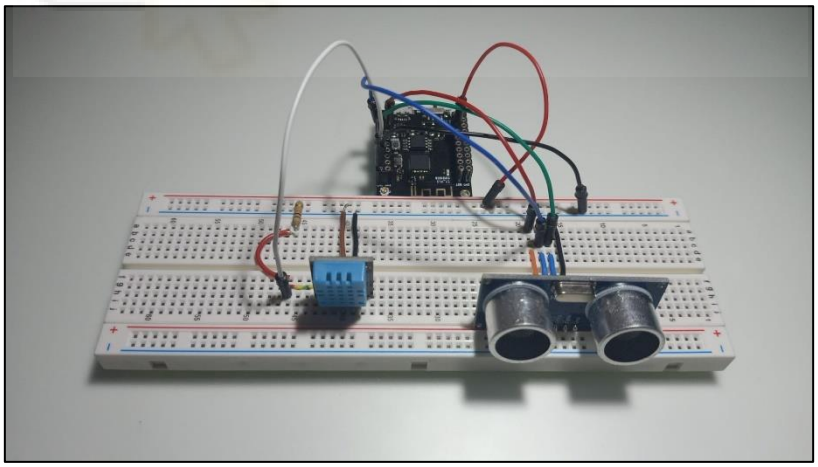


Ilustración 58. Vista general del sistema

En la elaboración de estas imágenes se ha omitido la toma de conexión del módulo ESP32 al ordenador con el fin de obtener una imagen más limpia y con mayor claridad.

Seguidamente se muestran los diferentes planos del sistema:

Alzado

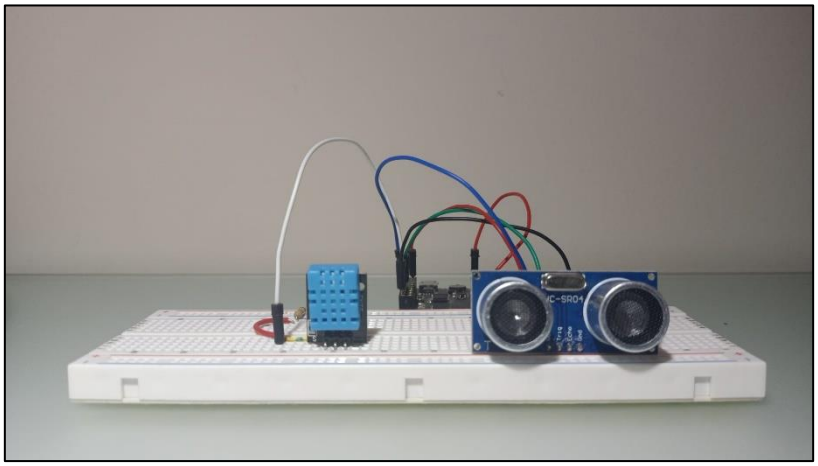


Ilustración 59. Alzado del sistema

Planta

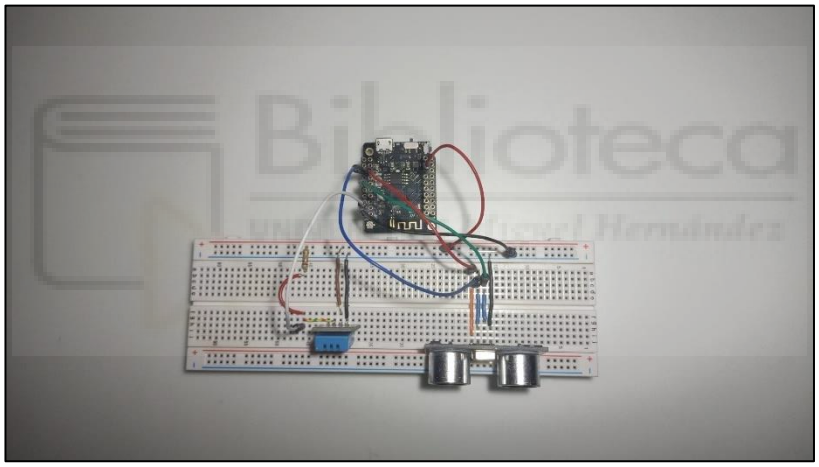


Ilustración 60. Planta del sistema

Perfil

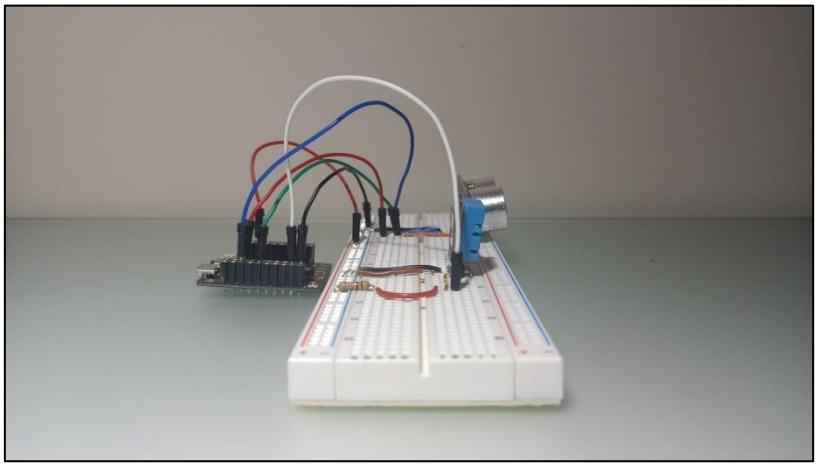


Ilustración 61. Perfil del sistema

6.2. VISTAS CONEXIÓN BLE

En esta sección se mostrarán en detalle las diferentes vistas de la aplicación Android desarrollada de la parte de BLE.

6.2.1. ICONO DE LA APLICACIÓN

No se trata como tal de una vista de la aplicación, sino del enlace inicial para lanzar y ejecutar la aplicación:

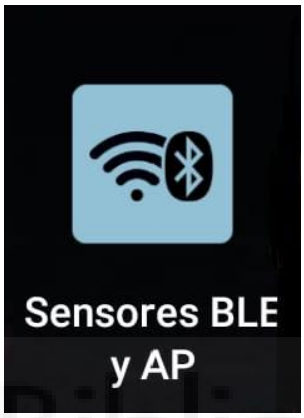
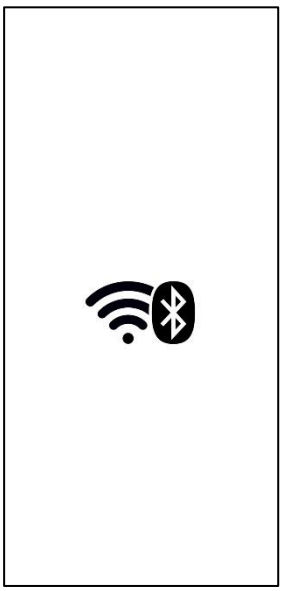


Ilustración 62. Icono de la aplicación

6.2.2. SPLASH SCREEN

Tras lanzar la aplicación, aparecerá de manera temporal el *splash screen* implementado:



Aparecerá de manera temporal el icono de nuestra aplicación mientras se carga el resto del contenido

Ilustración 63. Splash screen

6.2.3. WELCOME

Una vez lanzada la aplicación, aparecerá la primera de nuestras pantallas de bienvenida:

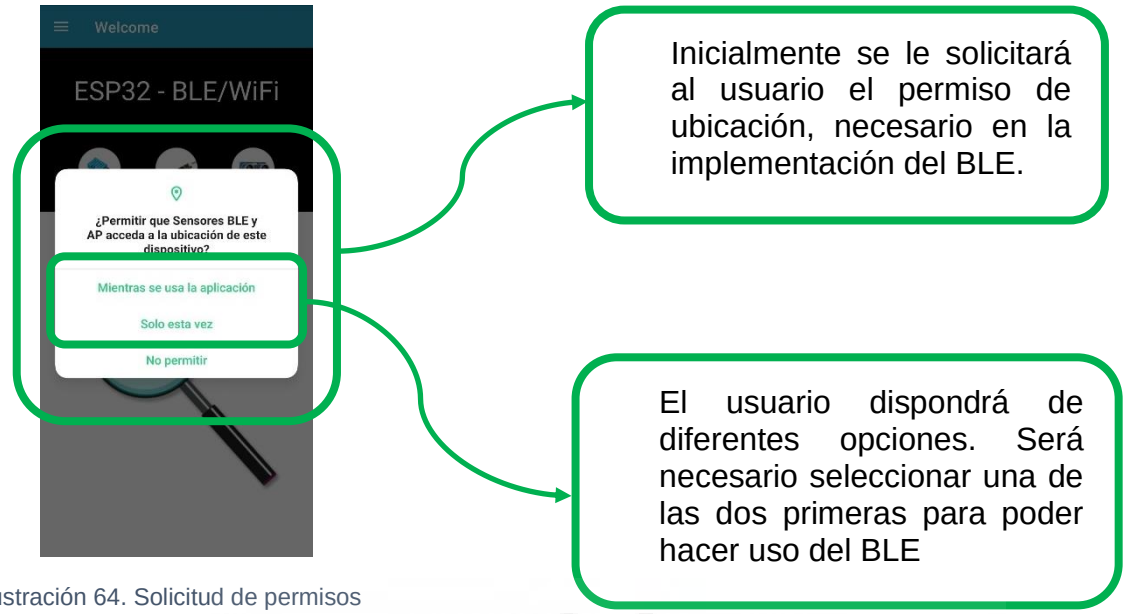


Ilustración 64. Solicitud de permisos

Seguidamente, tras la aceptación de los permisos requeridos, se podrá observar la pantalla de bienvenida como tal:

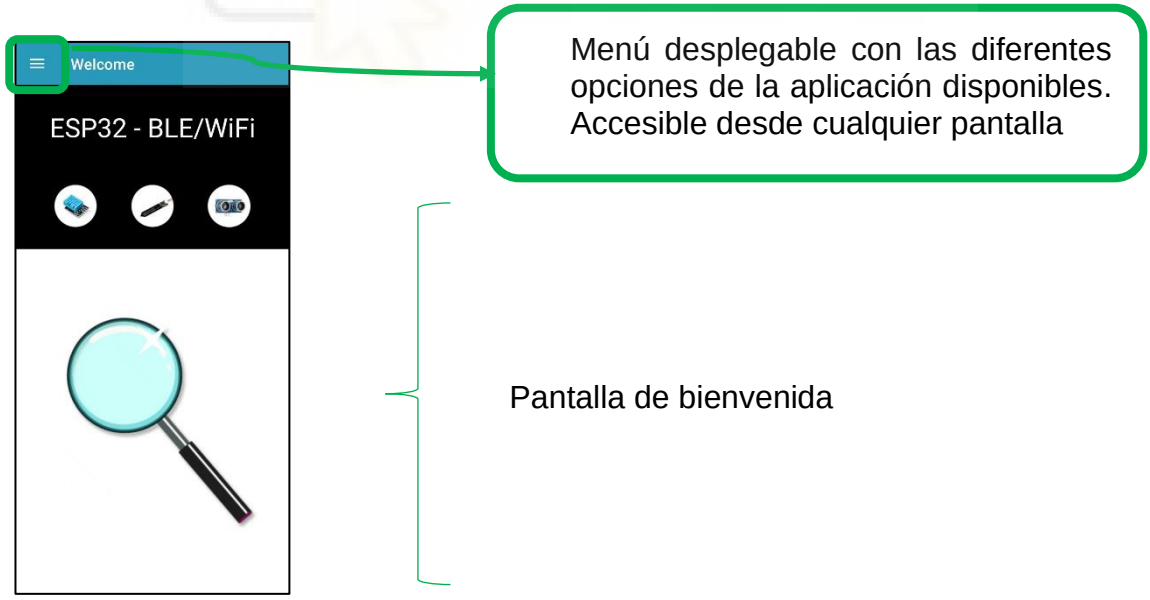


Ilustración 65. WelcomeFragment

Una vez seleccionado el menú desplegable, aparecerán las diferentes opciones que nos brinda la aplicación:

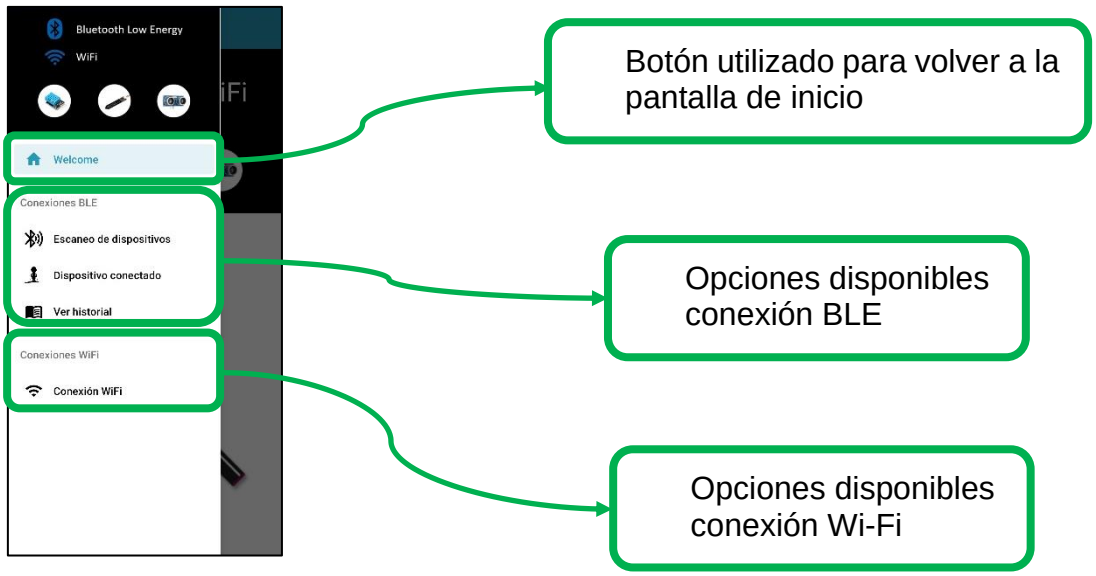


Ilustración 66. Menú desplegable

6.2.4. ESCANEO DE DISPOSITIVOS

Una vez se haya seleccionado la opción 'Escaneo de dispositivos', la aplicación nos redirigirá a la pantalla para comenzar nuestra búsqueda de dispositivos BLE:

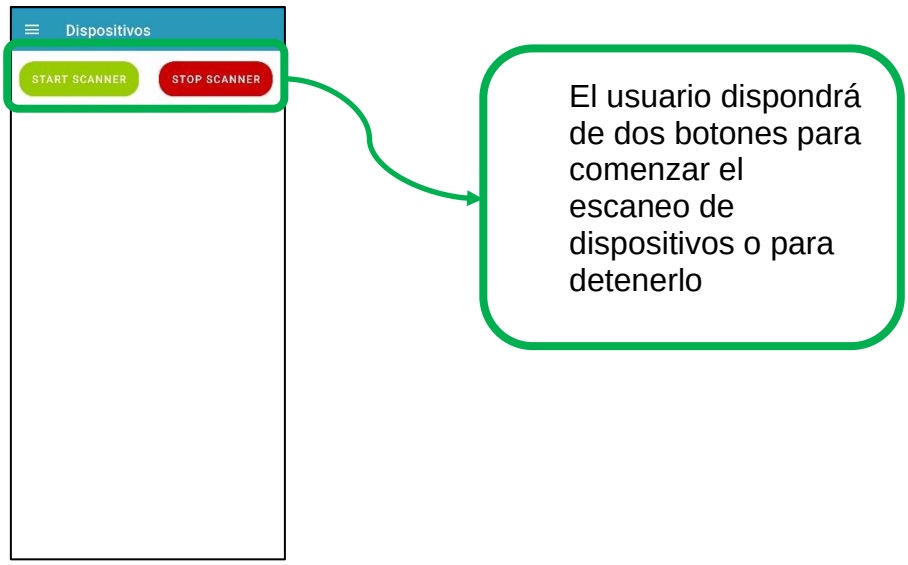


Ilustración 67. ScannerFragment

Tras pulsar sobre el botón ‘Start scanner’ para iniciar una búsqueda de dispositivos BLE, en caso de tener el Bluetooth del dispositivo y la ubicación del terminal desconectadas, aparecerán los siguientes avisos en la aplicación:

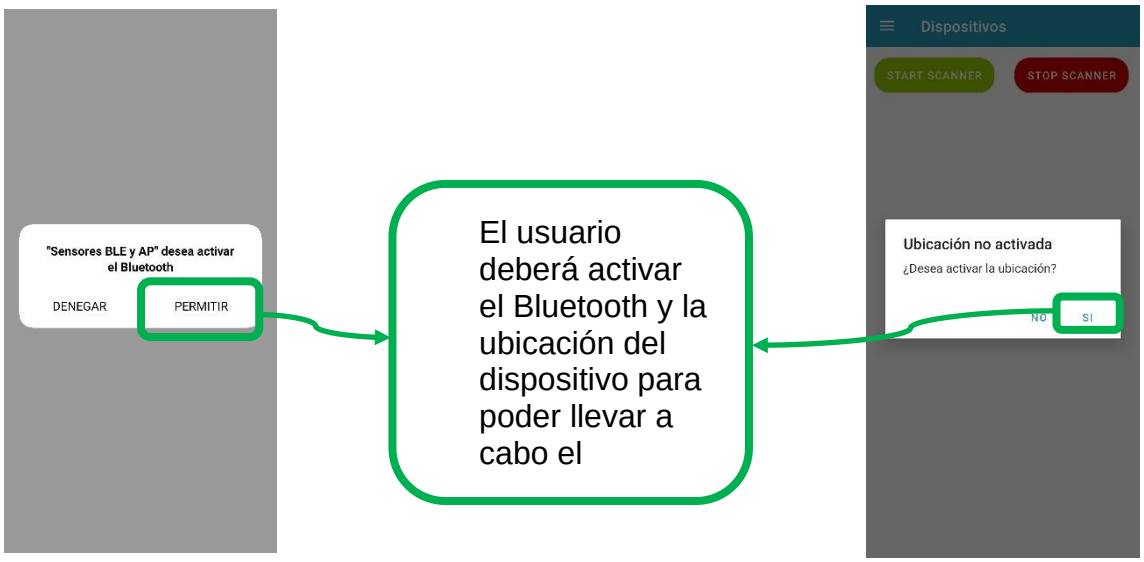


Ilustración 68. Encendido de Bluetooth

Ilustración 69. Encendido de ubicación

Una vez activados el Bluetooth y la ubicación respectivamente, el usuario podrá volver a pulsar sobre el botón ‘Start scanner’ con el objetivo de iniciar un escaneo de dispositivos BLE:

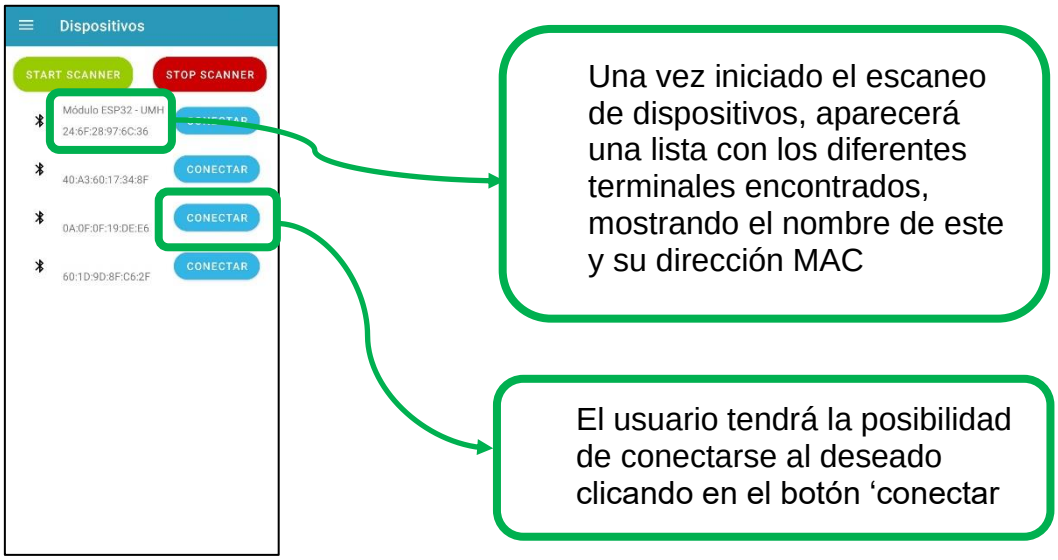


Ilustración 70. Escaneo de dispositivos

Tras seleccionar un dispositivo al que conectarse, aparecerá el siguiente mensaje en la aplicación:

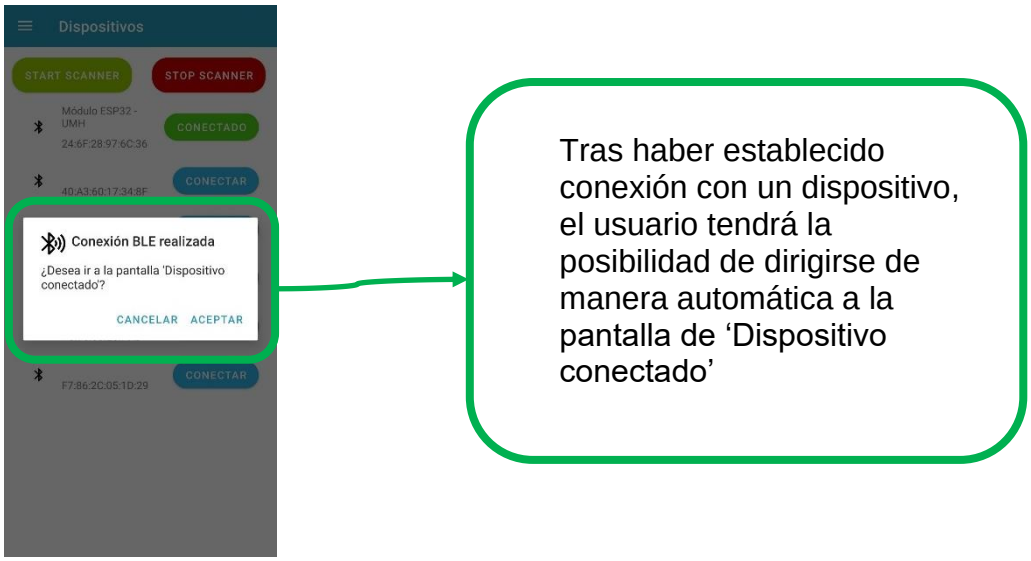


Ilustración 71. Conexión realizada

6.2.5. DISPOSITIVO CONECTADO

Una vez establecida la conexión con el dispositivo, en nuestro caso, con el 'Módulo ESP32 -UMH', aparecerá la pantalla de 'Dispositivo conectado' mostrando el siguiente mensaje:

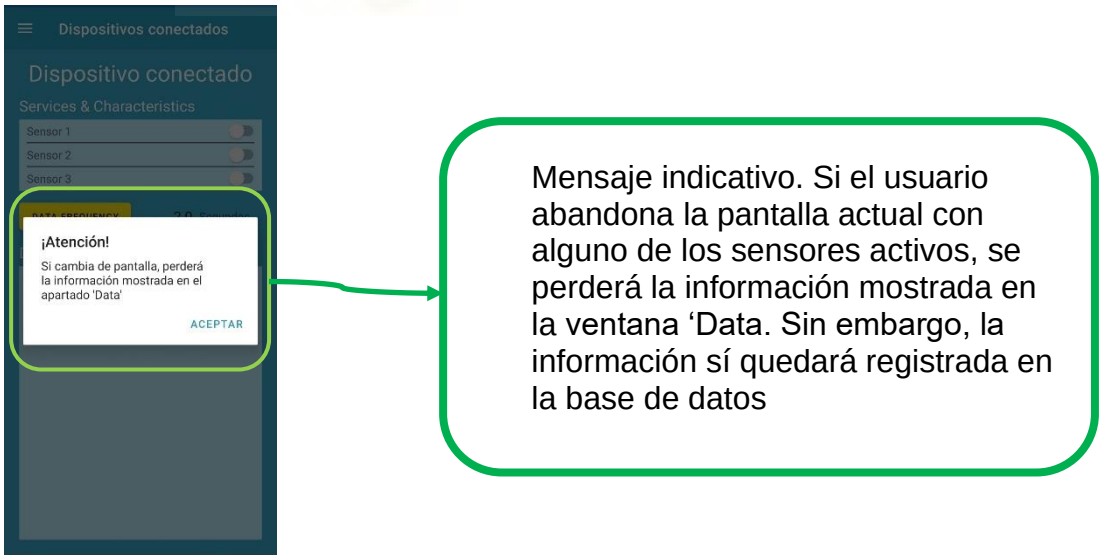


Ilustración 72. AlertDialog

Tras aceptar el mensaje anterior, el usuario verá lo siguiente:

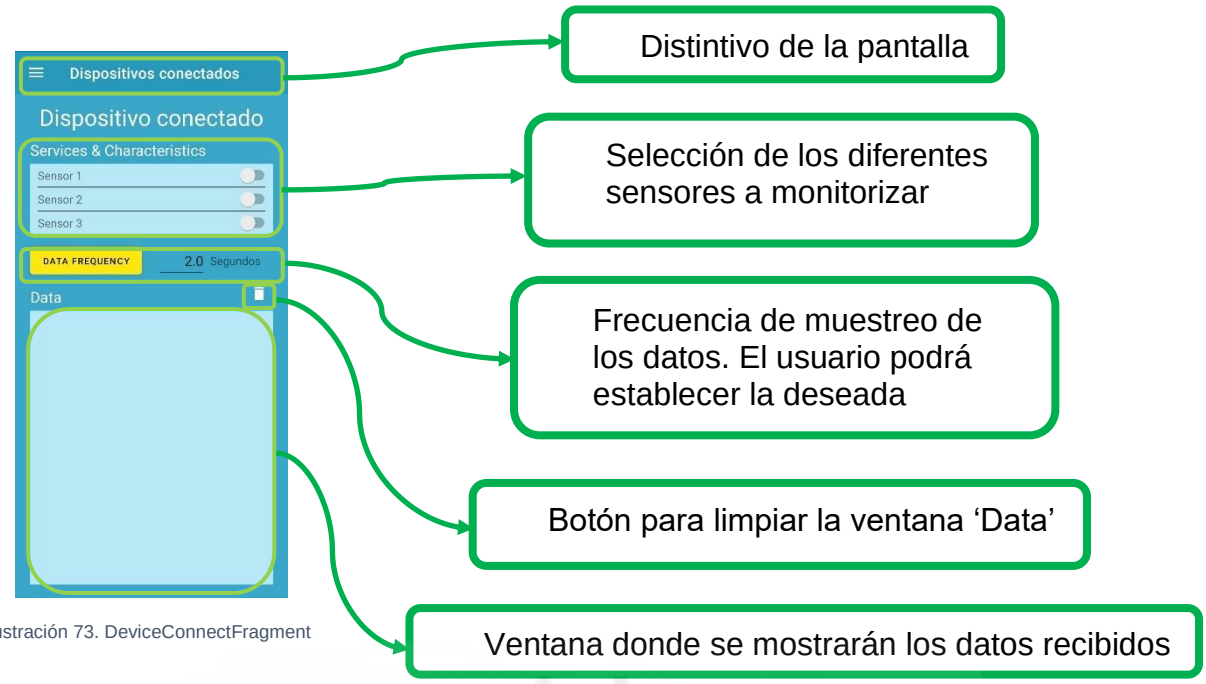


Ilustración 73. DeviceConnectFragment

Seguidamente se muestra un ejemplo de lo mostrado en la ventana 'Data' con los tres sensores activos, recibiendo información a una frecuencia de muestre de 2 segundos:

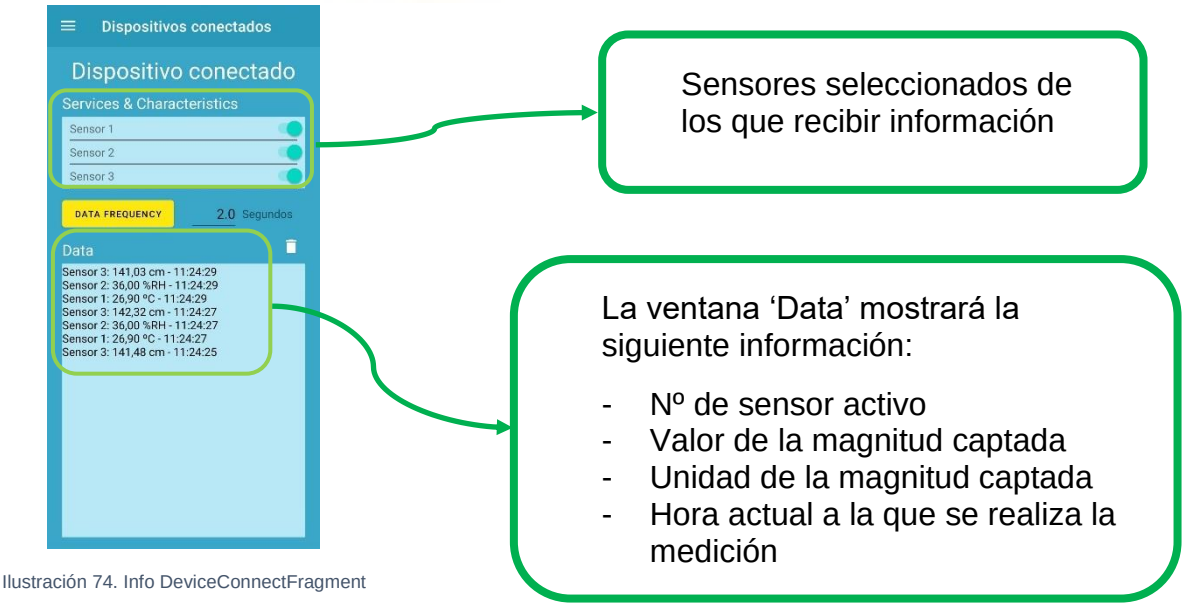


Ilustración 74. Info DeviceConnectFragment

6.2.6. VER HISTORIAL

Una vez llevada cabo la monitorización de los diferentes parámetros, estos serán almacenados en una base de datos local. Para acceder a ella, el usuario deberá seleccionar la opción ‘Ver historial’ del menú desplegable. Dicha acción le conducirá a la siguiente pantalla:

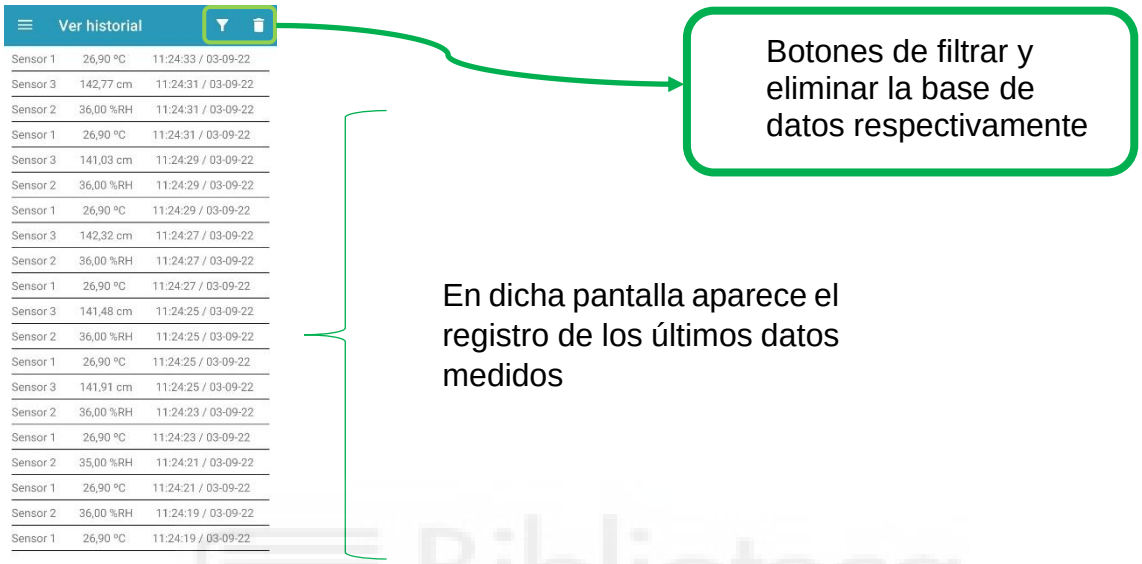


Ilustración 75. Historial

El usuario podrá llevar a cabo un filtrado exhaustivo de los valores almacenados:

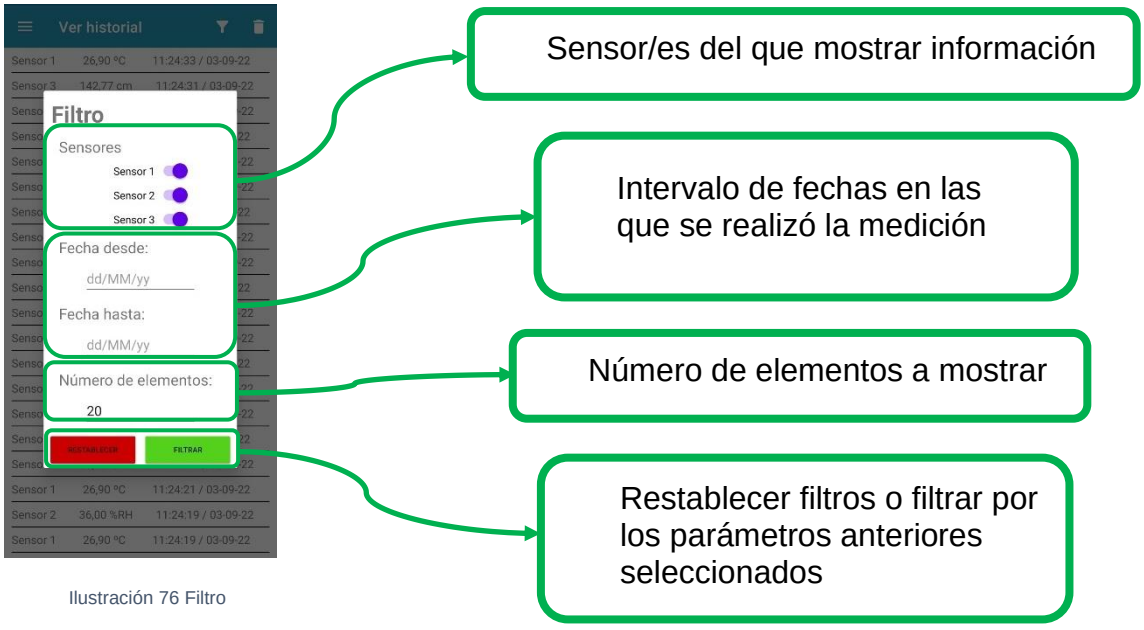


Ilustración 76 Filtro

Además, será posible eliminar la base de datos tras seleccionar el botón habilitado para ello:

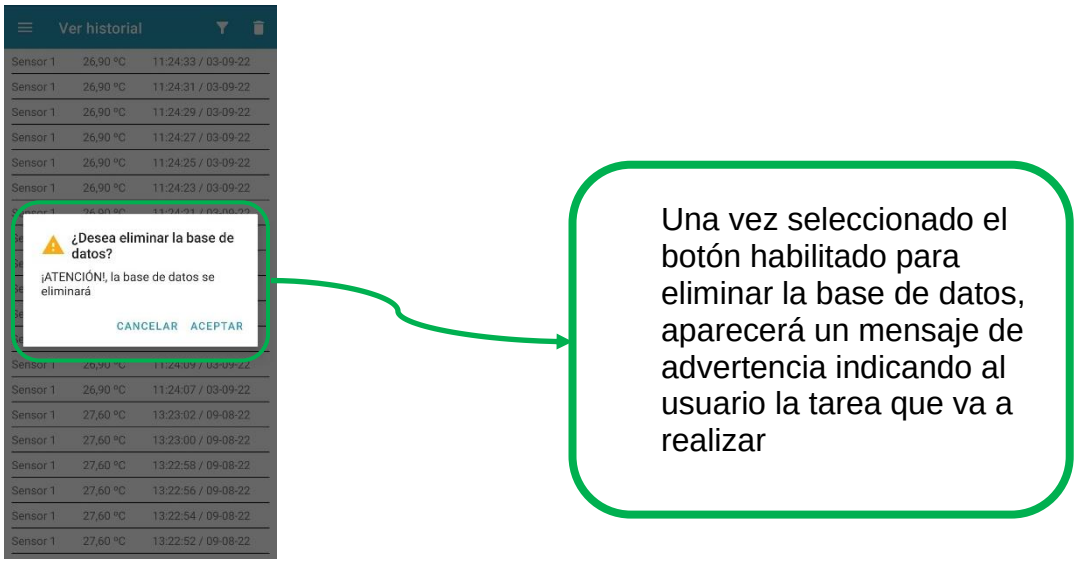


Ilustración 77. Aviso eliminación base de datos

6.3. VISTAS CONEXIÓN WI-FI

En esta sección se mostrarán en detalle las diferentes vistas de la aplicación Android desarrollada de la parte de Wi-Fi.

6.3.1. CONEXIÓN WI-FI

El usuario podrá comprobar el estado de los sensores haciendo uso de la red Wi-Fi generada por el ESP32 en modo AP tras seleccionar la opción 'Conexión WiFi' del menú desplegable:

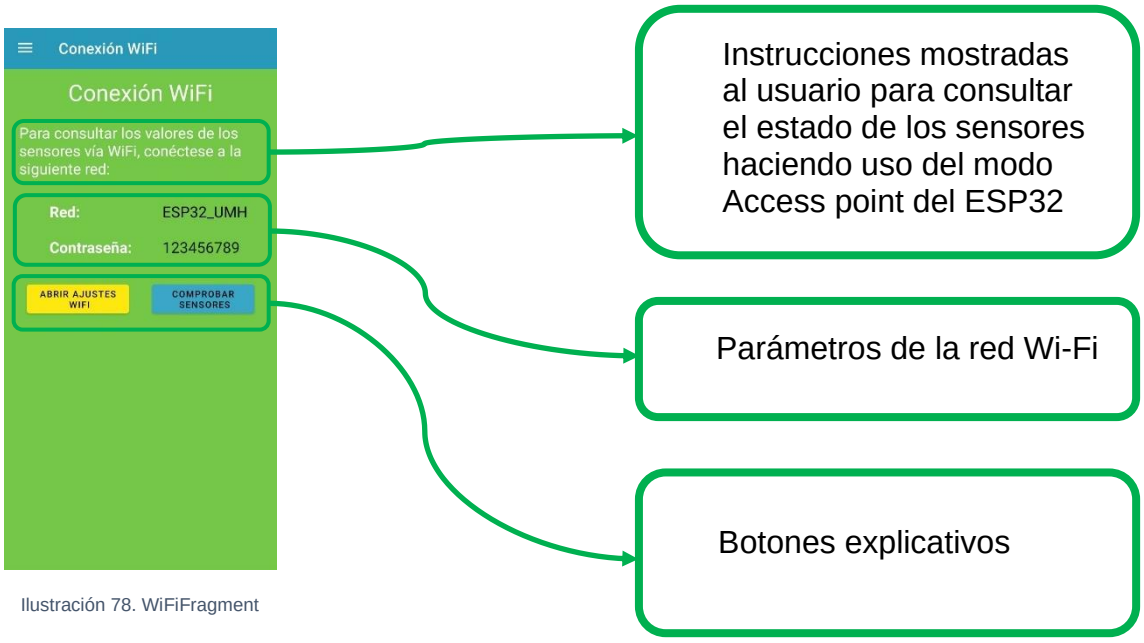


Ilustración 78. WiFiFragment

En caso de seleccionar la opción ‘Comprobar el estado de los sensores’ sin estar activado el Wi-Fi del dispositivo o sin estar conectado a la red en cuestión, se abrirá el navegador web por defecto del dispositivo y no se cargará ninguna información, dando como resultado la apertura de una ventana en el navegador web por defecto del dispositivo sin llegar a cargar nunca la información requerida.

Por su parte, en caso de cumplir con las especificaciones indicadas en la pantalla anterior, el usuario podrá llevar a cabo una completa monitorización de los diferentes valores captados por los sensores del sistema en tiempo real. Sin embargo, dicha información no quedará almacenada en la base de datos del dispositivo.

El usuario pulsará el botón ‘Comprobar sensores’ y se mostrará la siguiente pantalla haciendo uso del navegador web del dispositivo:

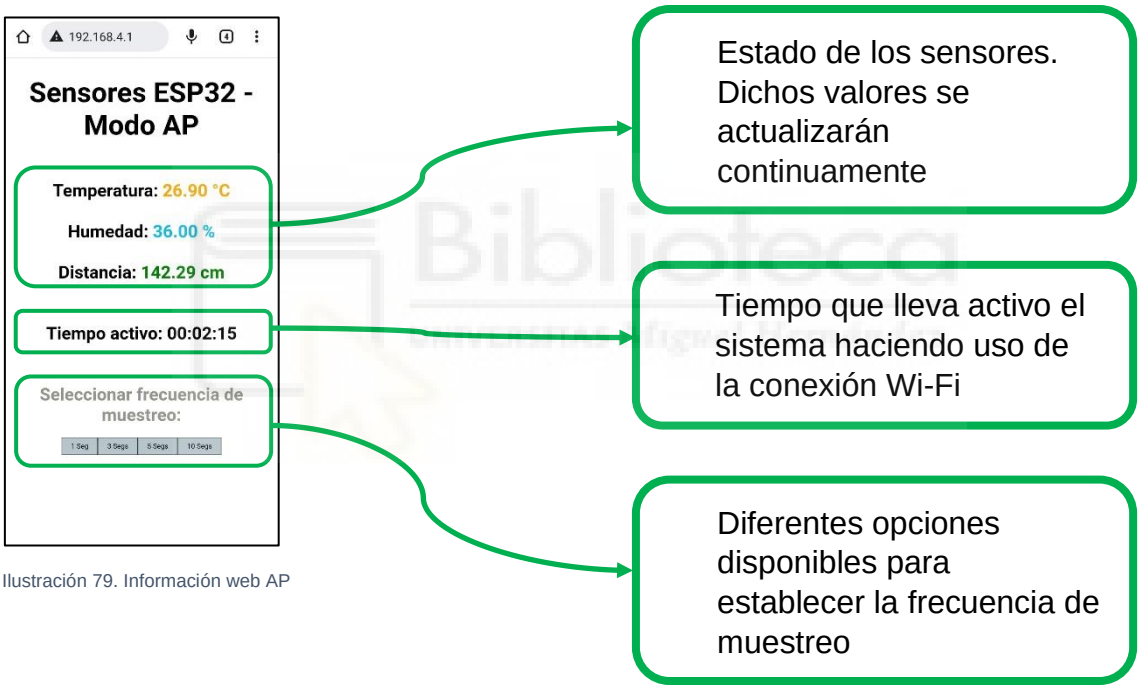


Ilustración 79. Información web AP

7. CONCLUSIONES

En este séptimo capítulo se recogen las conclusiones del proyecto, así como posibles desarrollos futuros para el sistema realizado.

7.1. CONCLUSIONES

Tras el desarrollo del presente trabajo de fin de grado, una de las conclusiones que podemos obtener es que se ha conseguido llevar a cabo el principal objetivo marcado al principio de esta memoria, el cual consistía en lograr el desarrollo e implementación de un sistema que permitiera monitorizar y llevar a cabo un control de diferentes parámetros medioambientales haciendo uso de una aplicación móvil.

Por su parte, desde el punto de vista personal, podemos destacar distintas conclusiones. En primer lugar, se ha comprendido la complejidad que hay detrás del desarrollo de una aplicación móvil, además de conseguir ampliar los conocimientos y habilidades en cuanto al desarrollo de aplicaciones móviles basadas en Android se refieren. Asimismo, ha sido posible poner a prueba diferentes conocimientos y habilidades desarrolladas durante el periodo de estudios de grado del estudiante.

Por último, he de comentar que este proyecto ha sido un gran reto que me ha aportado personalmente muchos conocimientos en un ámbito que me apasiona como es el desarrollo de aplicaciones móviles y diseño hardware de sistemas.

7.2. LÍNEAS FUTURAS

Seguidamente y para concluir el presente documento, se mencionan una serie de implementaciones futuras planteadas por el estudiante:

- Integrar en una misma PCB los diferentes componentes del sistema hardware sin necesidad de depender de una *protoboard*.
- Añadir una fuente de alimentación que permita al sistema funcionar sin depender de una conexión a un terminal portátil.
- Migrar la aplicación al lenguaje de programación Kotlin con el objetivo de estar a la vanguardia de las futuras actualizaciones.
- Mejorar la seguridad en las conexiones.
- Incluir en la aplicación un sistema de notificaciones pop-up programable para indicar diferentes eventos del sistema.

8. BIBILIOGRAFÍA

[1] Información sobre ThermoPro TP49

https://tecnoshop.bo/producto/temperatura-y-humedad-monitor-digital-para-interiores-thermopro-tp49

[2] Información sobre DATAPE DT50

https://www.reviewbox.es/telemetro-laser-mejores-opciones/

[3] Aplicación nRF Connect

https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&hl=es_419&gl=US

[4] Aplicación Serial Bluetooth Terminal

https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal&hl=es&gl=US

[5] Sensores electrónicos

https://aprendiendoarduino.wordpress.com/2016/11/06/electronica-sensores-actuadores-y-perifericos/

[6] SoC

https://engineering.purdue.edu/ECE/Academics/PMP/Areas/system-on-chip-design

[7] Web oficial de Espressif

https://www.espressif.com/

[8] Datasheet ESP8266

https://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[9] Datasheet ESP32

https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[10] Java

https://www.java.com/es/

[11] Kotlin

https://kotlinlang.org/

[12] C++

https://openwebinars.net/blog/que-es-cpp/

[13] HTML

https://developer.mozilla.org/es/docs/Web/HTML

[14] ¿Qué es un entorno de desarrollo?

https://www.arimetrics.com/glosario-digital/entorno-de-desarrollo

[15] IDE: Android Studio

https://developer.android.com/studio

[16] IDE: Visual Studio Code

https://code.visualstudio.com/

[17] IDE: Arduino

https://arduino.cc

[18] Chips de la familia ESP32

https://hmong.es/wiki/ESP32

[19] Características ESP32

https://soloarduino.blogspot.com/2017/03/que-es-un-esp32.html

[20] Diagrama de bloques del ESP32

https://hmong.es/wiki/ESP32

[21] Estructura de datos BLE

https://www.elt.es/ble-bluetooth-low-energy

[22] Estructura de datos GATT

https://programmerclick.com/article/8749346541/

[23] Generador de UUID

https://www.uuidgenerator.net/

[24] Modos de funcionamiento WIFi del ESP32

https://www.esploradores.com/modos-de-conexion-wifi/

[25] Datasheet sensor DHT11

https://www.electronicoscaldas.com/datasheet/DHT11_Aosong.pdf

[26] Datasheet sensor HC-SR04

https://datasheet4u.com/datasheet-pdf/ETC1/HC-SR04/pdf.php?id=1380138

[27] Características lenguaje de programación Java

http://www.itlp.edu.mx/web/java/Tutorial%20de%20Java/Intro/carac.html

[28] Proceso de compilación y carga de un programa en Arduino

https://aprendiendoarduino.wordpress.com/category/c/

[29] Javascript

https://www.javascript.com/

[30] SQLite

https://www.sqlite.org/index.html

[31] DHT11 y ESP32

https://randomnerdtutorials.com/esp32-dht11-dht22-temperature-humidity-sensor-arduino-ide/

[32] Conexión HC-SR04 y ESP32

https://microcontrollerslab.com/hc-sr04-ultrasonic-esp32-tutorial/

[33] Conexión BLE ESP32

https://microcontrollerslab.com/esp32-bluetooth-low-energy-ble-using-arduino-ide/

[34] Conexión Wi-Fi ESP32

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_wifi.html

[35] Direcciones IP ESP32 modo Soft AP

https://www.esploradores.com/access-point-servidor-web-nodemcu/

[36] Fragments

https://developer.android.com/guide/fragments?hl=es-419

[37] Splash screen

https://leanmind.es/es/blog/android-splash-screen-api/