Universidad Politécnica de Cartagena

etsit escuela técnica superior de ingeniería de telecomunicación

Estudio de viabilidad para la creación de redes de sensores de bajo consumo y largo alcance de transmisión de datos en entornos abiertos

Trabajo Fin de Estudios

Grado en Ingeniería de Sistemas de Telecomunicaciones

Autor: Cristian Torres Ruiz
Director: Vicente Javier Garcerán Hernández

Autor:	Cristian Torres Ruiz
E-mail del autor:	torresctesp@gmail.com
Director:	Vicente Javier Garcerán Hernández
E-mail del director:	Vicente.Garceran@upct.es
Título del TFG:	Estudio de viabilidad para la creación de redes de sensores de bajo consumo y largo alcance de transmisión de datos en entornos abiertos.
Resumen:	El objetivo de este proyecto es el diseño de una serie de PCBs que tengan conexión M2M entre ellas para la transmisión de datos. Los datos transmitidos serán los parámetros obtenidos por una gran variedad de sensores, y serán transmitidos vía Wi-Fi utilizando la tecnología Lo-Ra.
Titulación:	Grado en Ingeniería de Sistemas de Telecomunicaciones
Departamento:	Departamento de Electrónica, Tecnología de Computadores y Proyectos
Fecha de Presentación:	Septiembre 2021

Agradecimientos

A mi madre y a mis hermanos, por haberme cuidado durante toda mi vida y porque sin ellos no sería quién soy actualmente.

A mis amigos del colegio, por haberme ayudado a desconectar cuando más lo necesitaba.

A mis amigos de la universidad, porque sois sin duda lo mejor que me llevo de esta etapa de mi vida.

A mi novia, por ayudarme con las pruebas del proyecto y por ser la parte más importante de mi vida.

A Vicente, por haberme propuesto este proyecto y por haber estado siempre para ayudarme.

INDICE

ÍNDICE DE IMÁGENES

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. Introducción

1.1. Justificación del proyecto

La tecnología ha experimentado un crecimiento exponencial en las últimas décadas debido a la globalización mundial. Según el Cisco Annual Internet Report (2018-2023), se prevé que el número de dispositivos en 2023 sea un 60% mayor que los dispositivos que había en 2018 (Fig. 1).

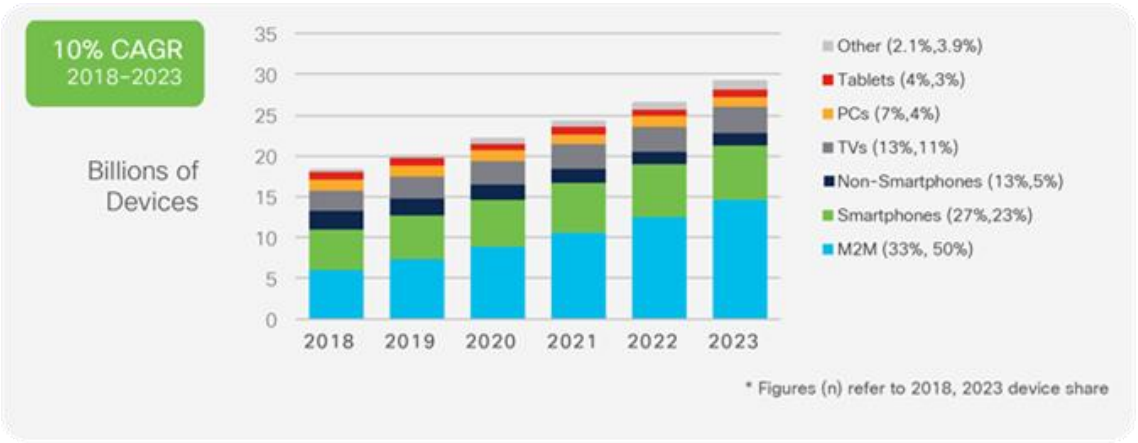

Figura 1. Crecimiento mundial de dispositivos y conexiones

El mayor responsable de esta previsión son los dispositivos M2M (machine to machine). Esta tecnología consiste en el intercambio de datos entre dos máquinas con el objetivo de automatizar diversas actuaciones en función de los datos recibidos, y se utiliza en industrias como la sanitaria, la automovilística, la energética, etc. Según Cisco, en 2023 habrá un crecimiento de un 241% en este tipo de dispositivos en comparación con los ya existentes en 2018 (Fig. 2).

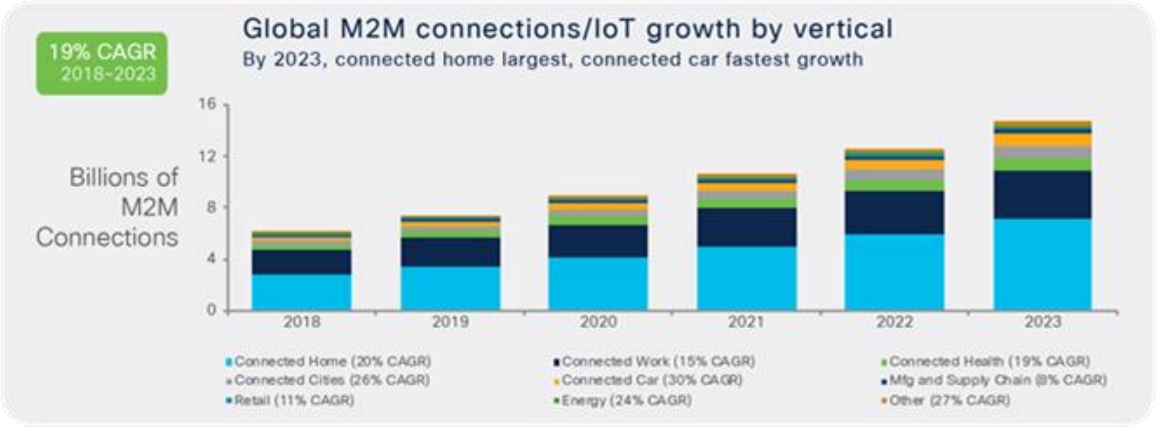

Figura 2. Crecimiento mundial de las conexiones M2M por sectores

Dentro de las conexiones M2M nos encontramos con las redes de sensores inalámbricas (Wireless Sensor Networks). Las redes de sensores inalámbricas consisten en un conjunto de dispositivos autónomos desplegados en un área de interés y cuyo objetivo es la toma de parámetros físicos y ambientales tales como temperatura, humedad, luz, lluvia, etc. Además, se considera una de las tecnologías clave para implementar el Internet de las cosas (IoT).

Estos dispositivos trabajan en áreas pequeñas y se intercomunican entre ellos para poder transmitir la información de la forma más rápida posible. Estas comunicaciones suelen ser bidireccionales, permitiendo configurar protocolos de actuación en función de los datos obtenidos. En el caso de que haya redes de actuación, se les conoce como redes de sensores y actuación inalámbricas (Wireless Sensors and Actuator Networks)

Uno de los principales beneficios del uso de estas redes es el bajo consumo que éstas necesitan, pudiendo tener una autonomía de 5 a 10 años con el uso de baterías. Esto nos permite desplegarlas en zonas de difícil acceso como integrados dentro de estructuras.

Dentro de las Telecomunicaciones, y más concretamente, en la rama de Electrónica es fundamental conocer las bases de la tecnología WSN y del funcionamiento de las conexiones M2M, así como la infinidad de posibilidades que nos ofrecen. En este proyecto se explicará paso a paso cómo podemos implementar esta tecnología, detallando todas las partes que tenemos que llevar a cabo para su correcta realización.

1.2. Objetivos

El objetivo principal de este proyecto es el desarrollo completo de una red de sensores, la cual pueda funcionar de forma autónoma mediante el uso de baterías durante el mayor tiempo posible. Esta red de sensores estará formada por varias placas que se dedicarán a transmitir o a recibir datos. Estas placas estarán formadas por un microcontrolador llamado ESP32, el cual nos ofrece las prestaciones de consumo y tamaño idóneas para el proyecto. Para configurar el funcionamiento del ESP32 utilizaremos la interfaz de Arduino para introducir el código necesario para su correcto funcionamiento. En las placas irán soldados sensores que miden diferentes parámetros como temperatura, humedad del aire, lluvia, luz UVA, gases (CO y CO_2), concentración de partículas en suspensión, presión atmosférica y humedad del suelo.

Todas estas PCBs, además, incluirán una pequeña placa soldada que utilizará la tecnología LoRa para realizar la transmisión y recepción de los datos. Estos datos se enviarán vía Wi-Fi a través del aire en la banda de 433MHz, e irán con la encriptación

que nos proporciona la tecnología LoRa. Con estas placas buscaremos maximizar la distancia necesaria entre transmisor y receptor sin que haya pérdida de datos.

1.3. Etapas del proyecto

En este TFG, en una primera fase, se analizarán las especificaciones y características de funcionamiento de microcontroladores que mejor se adapten para utilizarlos en un sistema de red de sensores conectados vía inalámbrica. Las especificaciones generales que debe cumplir esta red de sensores son:

- Versatilidad y multifuncionalidad.
- Escalabilidad
- Compatibilidad con sistemas de conectividad tipo LoRa, BLE y WiFi.
- Adaptación a diversos sensores, tanto digitales como analógicos.
- Bajo consumo de energía.
- Seguridad en la transmisión de datos y encriptación de los códigos en el microcontrolador.

Por otra parte, en la segunda fase de este proyecto se caracterizará y se estudiará el funcionamiento de sensores de temperatura, humedad, gases (CO, CO_2), concentración de partículas en suspensión, presión atmosférica, rayos UVA, detección de lluvia, nivel de humedad en suelo, etc.

Finalmente, una vez terminadas las fases anteriores, se diseñará un prototipo con PCB de conexión cableada y se probarán los diversos códigos implementados mediante el software adecuado para comprobar el funcionamiento de la red con los microcontroladores conectados a los sensores comprobando su funcionamiento en un entorno real.

1.4. Estructura

En el capítulo 2 se analizarán las diversas placas de desarrollo disponibles en el mercado que incluyen tecnología Arduino. Se hará una comparación entre ellas y se argumentará la elección del ESP32.

En el capítulo 3 se analizarán los sensores utilizados en el proyecto. Se incluirán sus prestaciones, sus requisitos de funcionamiento, el cableado necesario y la precisión de los datos obtenidos.

En el capítulo 4 se explicará la electrónica utilizada para la alimentación de la placa y de los sensores. Se analizarán los requisitos de alimentación de la placa, el tiempo de

trabajo del prototipo de forma autónoma, y los inconvenientes que han ocurrido en el proceso.

En el capítulo 5 se indicarán los pasos necesarios para la creación del prototipo. Se explicará el esquema general de todo el prototipo, cómo han sido diseñadas las diferentes partes y como se han creado mediante la soldadura correcta.

En el capítulo 6 se incluirán los códigos de Arduino necesarios para el correcto funcionamiento del prototipo. Se desglosará cada parte y se explicará su funcionamiento en el conjunto.

En el capítulo 7 se expondrán los resultados de las pruebas del prototipo en entornos outdoor. Se analizarán las pruebas y se establecerá una conclusión.

Al final del documento se incluyen todas las referencias utilizadas en esta redacción.

1.5. Novedades del proyecto en el mercado

Como hemos explicado en el punto 1.1, la implementación de los sistemas M2M está en auge y cada vez son más las empresas que ofrecen servicios de WSN. Estas empresas ofrecen tanto productos previamente diseñados y comprobados como productos diseñados mediante encargo por un cliente.

Sobre los productos previamente diseñados, si hacemos una investigación en el mercado nos encontramos con diversas opciones en tiendas online como Amazon.

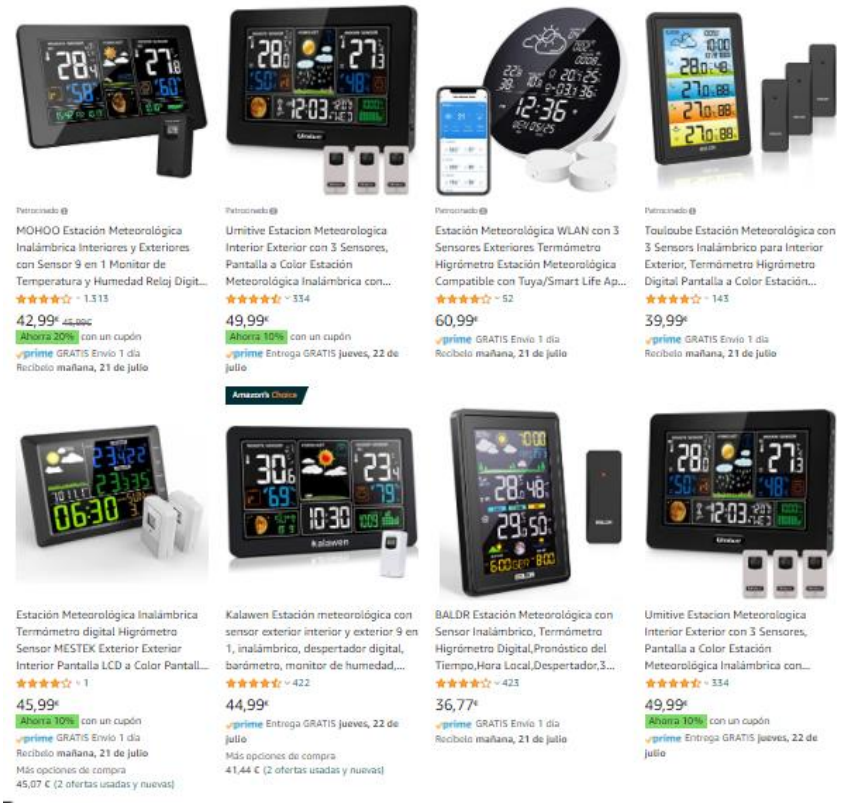

Figura 3. Opciones en el mercado.

Estas opciones nos ofrecen un aparato con todos los sensores incorporados y una pantalla para poder visualizar los datos de una forma más sencilla por una media de 45€. Si analizamos estos productos, vemos que se enfocan principalmente en rangos de áreas locales (aproximadamente 60m), y su principal uso es para transmitir información del ambiente en el hogar. Estos dispositivos funcionan con tecnología BLE (Bluetooth Low Energy), bluetooth de bajo consumo, que es idóneo para escenarios de distancias inferiores a 100 metros; y son alimentados mediante batería o pilas. La principal diferencia que tienen estos productos con el objeto de estudio de este proyecto es la tecnología utilizada y el rango de transmisión de datos, ya que el uso de tecnología Wi-Fi implica un mayor gasto de energía para poder llevarlo a cabo, lo que implica una optimización de las baterías. El uso de Wi-Fi nos permite unos rangos mayores, llegando a los 20km utilizando la tecnología LoRa.

Si hablamos de empresas que se especialicen en la tecnología M2M, nos encontramos con empresas nacidas en la UPCT que se están especializando en el uso de esta tecnología:

- Widhoc: es una spin-off que se dedica a la creación e instalación de sensores instalados en boyas que permiten medir el volumen de agua y su salinidad en

embalses, lo que permite conocer con exactitud la composición del agua que se utiliza para regar los cultivos y actuar en consecuencia.

- Qartech: es una spin-off surgida en la UPCT que se dedica a la investigación, diseño, desarrollo y comercialización de productos y servicios de sensorización e IoT con un alto componente de innovación. Ofrecen la posibilidad de implementar redes de sensores para cualquier tipo de aplicación que pida el cliente.

Como hemos podido observar, existe una gran oferta de empresas que cubren están necesidades aquí en la UPCT, lo que implica que la demanda de este tipo de servicios es elevada.

2. Análisis de microcontroladores

Antes de empezar a trabajar con los sensores y con el estudio de las alimentaciones, necesitamos realizar una investigación sobre el microcontrolador que vamos a utilizar para monitorizar todo el proceso. Como hemos indicado en la introducción, la tecnología que vamos a utilizar es Arduino, pero dentro de esta tecnología tenemos diferentes opciones: Arduino UNO, ESP2866, ESP32, etc. Todos estos microcontroladores utilizan la IDE de Arduino para ser configurados, pero tienen distintas condiciones de funcionamiento, distintas prestaciones, etc. Se va a realizar un análisis de los parámetros que nos interesan (consumo, tecnologías) y se va a realizar una conclusión de la placa más idónea para nuestro proyecto.

Para ello vamos a profundizar sobre los parámetros en los que nos vamos a basar para escoger un microcontrolador u otro. En este caso vamos a investigar sobre los modos de funcionamiento que puede tener un microcontrolador. Los modos de funcionamiento son distintos tipos de configuraciones con lo que la placa de Arduino puede funcionar, de tal forma que, si un modo requiere tener una mayor cantidad de funciones activadas, eso implica un mayor consumo de energía. Cómo hemos explicado en el punto 1.1, las redes de sensores inalámbricas se caracterizan por tener una autonomía de 5 años, por lo que necesitamos buscar el Arduino que menos consuma. Dentro de todos los modos de funcionamiento encontramos los siguientes ordenados de mayor a menor consumo.

- Modo activo o encendido: en este modo todas las funciones del microcontrolador están activadas y en funcionamiento.
- Modo modem-sleep: este modo es específico para las placas que tienen incorporados elementos Wi-Fi o Bluetooth. En este modo todo está activado a excepción de los relacionados con Wi-Fi o Bluetooth.

- Modo light-sleep: en este modo la CPU se encuentra pausada. La memoria RTC (Real Time Clock) y los periféricos RTC se encuentran encendidos. Se necesitan eventos de wake-up como interrupciones externas para reanudar el microcontrolador.
- Modo Deep-sleep: en este modo sólo se mantienen operativos la memoria y los periféricos RTC. Necesitamos un temporizador externo como el WatchDog Timer para poder despertar al microcontrolador.
- Modo hibernación: en este modo el oscilador interno del microcontroaldor se apaga. Sólo se mantiene encendido un temporizador RTC para poder despertar a la placa del modo hibernación.

Para nuestro prototipo nos centraremos en el modo activo y en el modo Deep-sleep, ya que utilizaremos el ya nombrado WDT para mantener el dispositivo en bajo consumo.

2.1. Arduino UNO

Arduino UNO es una placa de microcontrolador de código abierto desarrollado por la compañía Arduino. Como podemos ver en la Imagen 1, está formado por 14 pines digitales, 6 pines analógicos y los pines de alimentación. Se puede configurar con la IDE de Arduino.

Imagen 1. Arduino UNO.

En este proyecto, la principal característica en la que nos tenemos que fijar para saber si un dispositivo es adecuado o no es el consumo que tiene la placa de desarrollo.
Si investigamos, podemos ver que el consumo del Arduino UNO es de 50 mA cuando está encendido y de 30.8µA cuando está en Deep-sleep.

2.2. ESP8266

ESP8266 es una placa de microcontrolador de código abierto desarrollado por la compañía Arduino. Como podemos ver en la Imagen 2, está formado por 13 pines de GPIO, por pines de alimentación y de tierra y por otros pines con usos específicos como dispositivos LoRa, etc. Se puede configurar con la IDE de Arduino.

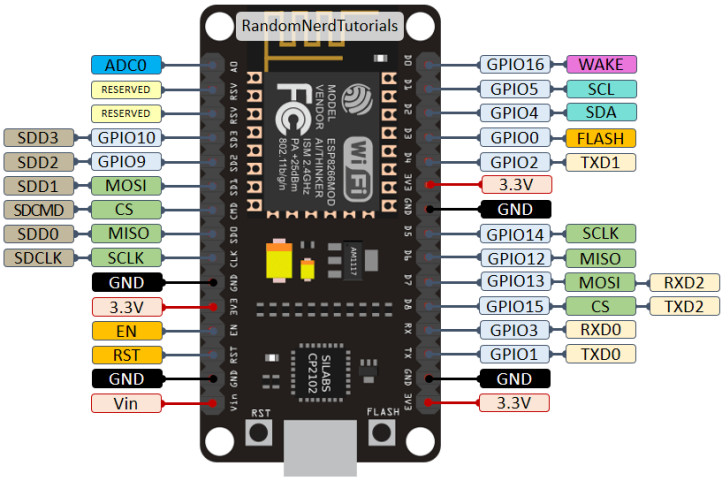

Imagen 2. ESP8266

En el datasheet del ESP8266 se nos indica que su consumo es de 15 mA cuando está encendido y de 20µA cuando está en Deep-sleep.

2.3. ESP32

ESP32 es una placa de microcontrolador de código abierto desarrollado por la compañía Arduino. Como podemos ver en la Imagen 3, está formado por 31 pines GIOP, por pines de alimentación y de tierra y por otros pines con usos específicos como dispositivos LoRa, etc. Se puede configurar con la IDE de Arduino.

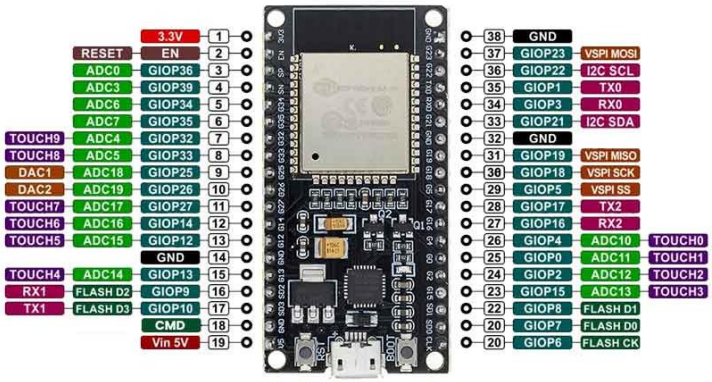

Imagen 3. ESP32

En el datasheet del ESP32 se nos indica que su consumo es de 20 mA a 25mA cuando está encendido y de 10µA cuando está en Deep-sleep.

2.4. Comparación

Una vez vistos estos tres microcontroladores, procedemos a comparar sus consumos:

Tabla 1. Comparación de microcontroladores.

Microcontrolador	Consumo modo normal	Consumo Deep-sleep mode
Arduino UNO	50mA	30.8µA
ESP8266	15mA	20µA
ESP32	20mA ~25mA	10µA

Como podemos ver en la Tabla 1, el ESP8266 tiene el menor consumo de los 3 cuando está en funcionamiento, mientras que el ESP32 tiene el menor consumo en modo Deep-sleep mode. Teniendo en cuenta que el dispositivo estará en Deep-sleep mode la mayor parte del tiempo, nos interesa más el microcontrolador que consuma menos en este modo de funcionamiento, por lo que escogeremos el ESP32.

Una vez escogido el mejor microcontrolador, vamos a profundizar un poco en él.

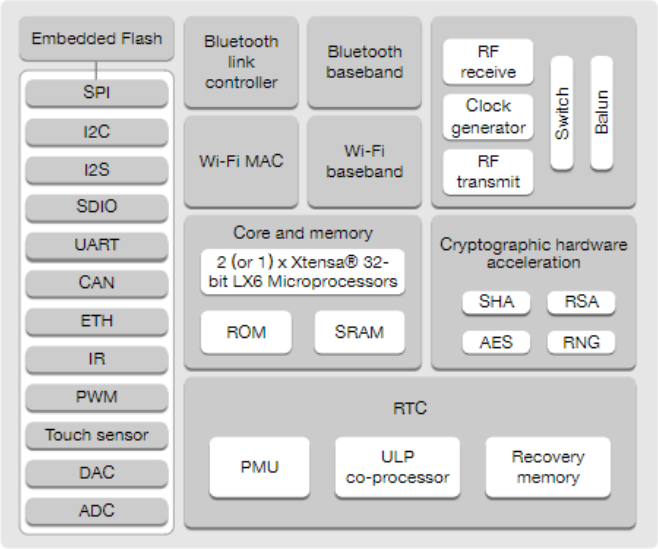

Figura 4. Diagrama de bloques.

En la Figura 4 podemos observar el diagrama de bloques que conforma el ESP32. Está dividido en diferentes áreas:

- Núcleo y memoria: el ESP32 utiliza un microprocesador Xtensa de 32 bits con 2 núcleos, dependiendo de la versión que se compre. Además, dispone de 520KB de RAM
- RTC: el ESP32 trabaja con un oscilador interno de 8MHz, además de un oscilador RC para el WDT. De forma externa, dispone de un oscilador de cristal de cuarzo que puede trabajar de 2 a 60 MHz.
- Interfaces periféricas: en nuestro modelo disponemos de 34 GPIOs programables. Permite conexión tanto por el estándar SPI como por el I^2C.
- Comunicaciones Wireless: el ESP32 es el microcontrolador de Arduino que incorpora tanto Wi-Fi como Bluetooth.

3. Análisis de dispositivos

En este proyecto vamos a utilizar diversos sensores que midan una serie de parámetros de interés. Para simplificar el montaje de las placas dividiremos los sensores en dos montajes independientes: una placa que estará situada en el suelo y otra placa que estará en un punto más elevado que la anterior. Esta distinción se ha realizado debido al tipo de sensores con los que trabajamos, ya que tenemos sensores que miden humedad en tierra y otros que miden el nivel de lluvia, por lo que deben estar más elevados. Primero analizaremos los sensores utilizados en la placa del suelo y después, en la placa más elevada.

3.1. Sensores: Placa inferior

3.1.1. Sensor de CO_2 y componentes orgánicos volátiles: CCS811

Un sensor de gas es un dispositivo que permite detectar la presencia en el aire de un determinado gas. En este caso se ha decidido medir algunos parámetros en concreto como el dióxido de carbono (CO_2) y componentes orgánicos volátiles (VOCs, Volatile Organic Compounds), para los cuales usaremos el CCS811.

Imagen 4. CCS811

Este sensor sólo está formado por una placa que tiene incluidos los elementos sensitivos. El funcionamiento está basado en la tecnología única de microplacas calientes de AMS, que permite una resolución muy alta para los sensores de gases, ciclos de trabajo muy rápidos y una reducción significativa en el consumo de potencia medio.

Está formado por un sensor y por un microcontrolador con un convertidor A/D para detectar la calidad del aire.

Con este funcionamiento nos permite medir rangos de CO_2 desde 400ppm hasta 32768ppm y de TVOC desde 0ppb hasta 32768ppb.

Como podemos ver en la Imagen 4 la placa tiene unas medidas de 1.38 x 2 cm y dispone de 8 patillas, las cuales son las siguientes:

- VCC (1): es la patilla de alimentación de la placa. Debe ser de 3.3V.
- GND (2): es la patilla de tierra de la placa.
- SCL (3): es la señal de reloj de entrada usada en la interfaz I^2C. Esta señal se encarga de indicar el período de envío de datos de los dispositivos esclavos (sensores).
- SDA (4): es la señal de datos de salida usada en la interfaz I^2C. Esta señal se encarga de enviar los datos cuando la señal de reloj se lo indique.

- WAK (5): es una entrada activa que debe situarse a nivel bajo para que el sensor empiece a funcionar.
- INT (6): es una salida activa que se establece en nivel bajo cuando la medición ha terminado.
- RST (7): es una patilla opcional.
- ADD (8): establece los bits de la dirección I^2C.

3.1.2. Sensor de CO y gases combustibles: MQ-9

Un sensor de gas es un dispositivo que permite detectar la presencia en el aire de un determinado gas. En este caso se ha decidido medir algunos parámetros en concreto como el monóxido de carbono (CO), metano, propano y LPG. Un sensor que nos permite medir estos parámetros es el MQ-9.

Imagen 5. MQ-9

Este dispositivo está formado por los elementos sensitivos y por un circuito integrado.
En el datasheet se nos indica que el sensor está basado en semiconductores y la detección de los gases se hace mediante el método de ciclo de alta y baja temperatura.

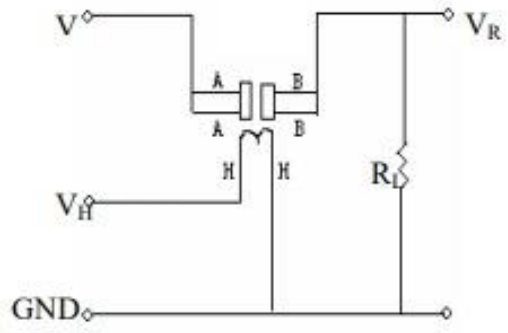

Figura 5. Esquema básico del sensor

En la Fig. 5 podemos ver el esquema en el que se basa este método. Necesitamos introducir dos voltajes, V_H se encarga de introducir la temperatura de trabajo del sensor y V_R es el valor de tensión de la resistencia R_L. Cuando la conductividad del sensor aumenta significa que la concentración de gases también ha aumentado. Esta variación de la conductividad de la unión p-n del semiconductor produce una serie de pares de electrones que se transfieren desde la banda de valencia a la banda de conducción. Al estar bajo la influencia de un campo eléctrico, se produce un pulso medible con la información sobre la energía de la radiación incidente original.

Esta salida analógica no nos muestra todavía la presencia de gases en el ambiente, sino que tenemos que normalizarlo para un valor de R_0 que es el promedio de las concentraciones de gases medidas por el sensor en un período de calibración. Este período de calibración se nos indica en el datasheet que debe ser de 2 días para una precisión exacta de los valores medidos, pero con 6 horas de calibración obtenemos una precisión suficientemente válida. El dispositivo tiene una salida analógica que nos permite obtener la concentración de gases con una precisión de 10 bits y una salida digital de 1 bit. Para configurar esta salida el sensor incluye un potenciómetro que marca el valor umbral del comparador, de tal forma que, si el valor medido es menor que el umbral se marcará un 0 y, si es mayor, un 1. Convirtiendo el valor obtenido en un valor de tensión, y calculando R_S/R_0, obtenemos la variación de la concentración de partículas en el aire respecto al valor normalizado de partículas en el aire. A través de este dato podemos saber el número de partículas de los diferentes elementos siguiendo la Figura 6, que es una gráfica que nos proporciona el fabricante:

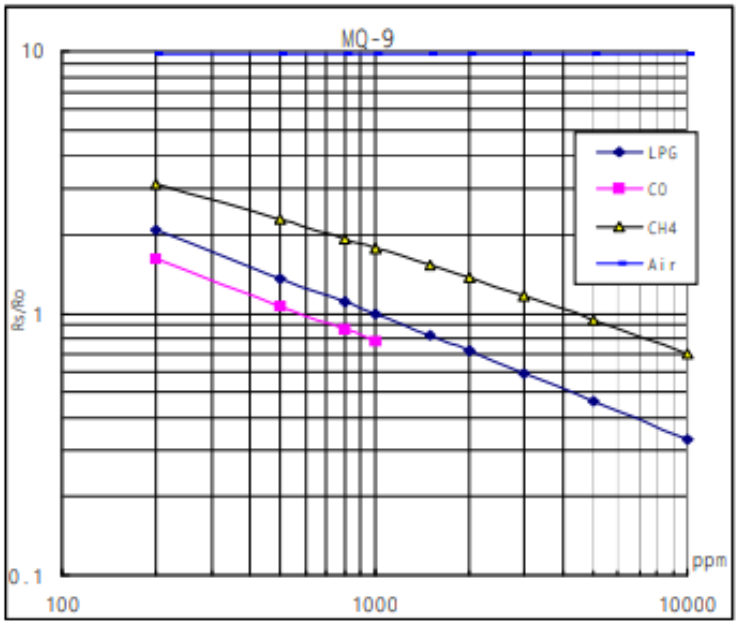

Figura 6. Gases en el aire en función de Rs/Ro

La placa mide 1.9 x 3.1 cm. Además, dispone de 4 patillas:

- V_{CC} (1): es la patilla de alimentación de la placa. Además, será el valor de referencia del amplificador.
- GND (2): es la patilla de tierra de la placa.
- DO (3): es la salida digital de la placa (la salida del comparador). Se debe introducir en un pin de entrada y salida digital del ESP32 (en nuestro caso no se utilizará).
- AO (4): es la salida analógica de la placa (la salida del divisor de tensión). Se debe introducir en un pin de entrada y salida analógico del ESP32.

Puesto que la lectura de los datos analógicos de este sensor es un valor de tensión, no será necesario incluir ninguna biblioteca adicional. El código se explicará en el capítulo 6.

3.1.3. Sensor de temperatura y humedad: DHT22

Un sensor de temperatura es un dispositivo que convierte la temperatura en una señal eléctrica, mientras que un sensor de humedad hace la misma conversión, pero con la humedad del aire. Para el análisis de estos dos parámetros se ha escogido el dispositivo DHT22.

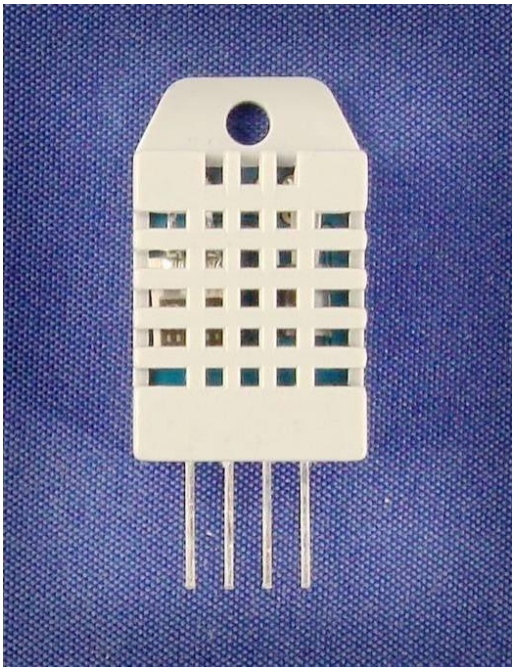

Imagen 6. DHT22

Este dispositivo está compuesto por los elementos sensitivos de temperatura y humedad y por un microcontrolador, conectados entre ellos por un bus de datos de 8 bits.
Los elementos sensitivos están formados por condensadores eléctricos de polímero, los cuales presentan diferentes valores capacitivos para diferentes valores de temperatura y humedad. Estos valores capacitivos son los que analiza el microcontrolador y, a partir de esos valores, genera la señal electrónica con los datos.
El microcontrolador contiene los parámetros necesarios para la calibración del dispositivo y, cuando el sensor es alimentado, calcula la salida digital utilizando esos parámetros y los valores de los condensadores eléctricos de polímero. Con este funcionamiento, las prestaciones que ofrece el DHT22 son las que se pueden observar en la Tabla 2:

Tabla 2. Datos técnicos del DHT22

Rango de operación:	Humedad: 0-100%RH; Temperatura: -40~80º Celsius
Precisión:	Humedad: ±2% RH (Max ±5%RH); Temperatura<±0.5ºC
Resolución o sensibilidad:	Humedad: 0.1%RH; Temperatura: 0.1ºC
Repetibilidad:	Humedad: ±1%RH; Temperatura: ±0.2ºC
Periodo de detección:	Media: 2 segundos

Además, como podemos ver en la Figura 7 proporcionada por el fabricante, el dispositivo es muy compacto, con unas medidas de 15.1x7.7x25.1(33.6 si incluimos las patillas) y con un consumo muy bajo, lo cual lo hace ideal para nuestro proyecto.

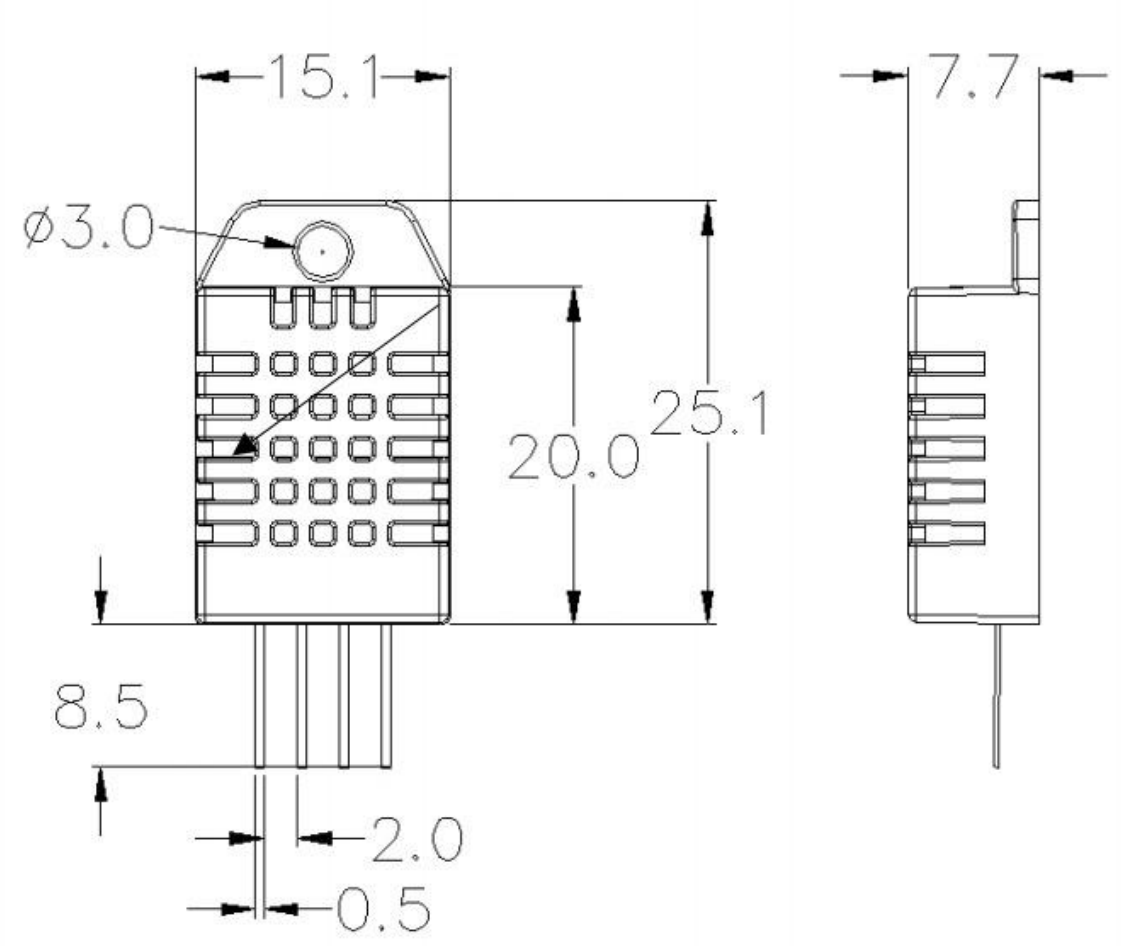

Figura 7. Medidas del sensor DHT22 (en mm)

Como podemos ver en la Figura 7, el DHT22 dispone de 4 patillas, las cuales en el propio datasheet del dispositivo se nos indica el conexionado necesario para hacerlo funcionar, el cual podemos observar en la Figura 8:

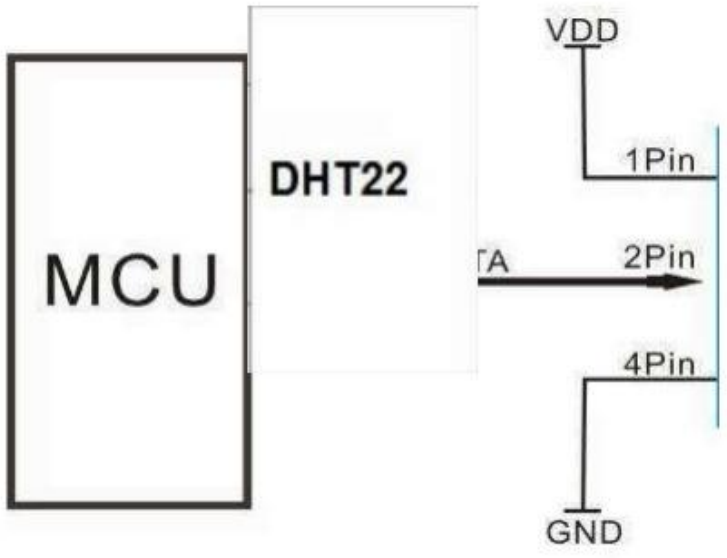

Figura 8. Conexionado del DHT22

- V_{DD} (1): es la alimentación del dispositivo. En el datasheet se nos indica que debe estar comprendido entre 3.3V y 6V DC. Además, se nos indica la posibilidad de incluir un condensador de 100nF entre V_{DD} y GND para eliminar componentes de rizado.
- DATA (2): a través de esta patilla se obtienen los datos de temperatura y humedad que mide el dispositivo. El procedimiento que sigue es el siguiente:

1. El MCU le envía la señal de inicio al DHT22. Para ello, el MCU se encarga de transformar el nivel de voltaje del bus de datos de un nivel 'high' a un nivel 'low'. Para este proceso el MCU necesita 1ms para asegurarse que el DHT22 ha recibido la señal de inicio. Posteriormente, el MCU esperará entre 20 y 40 µs para recibir la respuesta del DHT22.
2. El DHT22 envía la señal de respuesta al MCU. Cuando el DHT22 recibe la señal de inicio, éste envía una señal de bajo voltaje como respuesta que dura 80µs. Una vez enviado, vuelve a transformar el bus de datos del nivel 'low' al nivel 'high y tarda 80µs en preparase para enviar datos.
3. El DHT22 le envía datos al MCU.
    - NULL (3): este pin no necesita conexionado.
    - GND (4): es la tierra del dispositivo.

Para que la placa ESP32 analice los datos del pin 3, es necesario la instalación de dos bibliotecas llamadas DHT.h y DHT_U.h, las cuales se explicarán en el capítulo 6.

3.1.4. Sensor de humedad en suelo: HD-38

Un sensor de humedad de suelo es un dispositivo que convierte la humedad de una superficie en un valor de tensión. Para el análisis de este parámetro se ha escogido el HD-38.

Imagen 7. HD-38

Como podemos ver en la Imagen 7, este dispositivo está compuesto por dos partes: una sonda y una placa. La sonda es el elemento sensitivo que se introduce en el suelo. Cada patilla está formada por un electrodo, cuya resistencia dependerá de la humedad del suelo, es decir, a mayor humedad menor resistencia entre los electrodos. Esta resistencia (R_2) forma parte de un divisor de tensión junto a una resistencia fija (R_1), a la que se le introduce a la entrada el valor de alimentación (3.3V en nuestro caso).

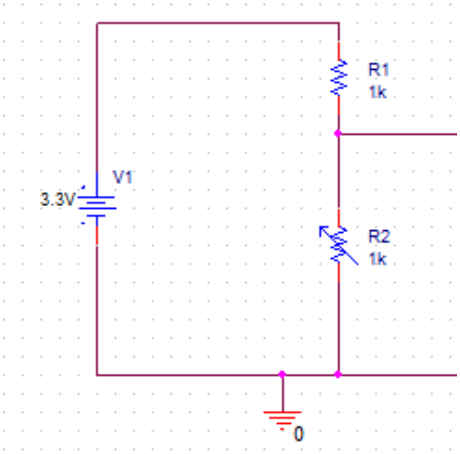

Figura 9. Esquema de un divisor de tensión

La ecuación que describe este circuito es $v_{out} = \frac{R_2}{R_1+R_2} v_{in}$:

Si $R_2 = 0$ (hay mucha humedad y actúa como un cortocircuito) a la salida obtendremos un valor de $v_{out}=0$.

Si $R_2 = \infty$ (hay poca humedad y actúa como un circuito abierto) a la salida obtendremos un valor de $v_{out}=(R_2/R_2)v_{in}=v_{in}=3.3V$. El resto de valores dependerá de la ecuación previamente dicha.

La placa dispone de dos salidas: una analógica y otra digital. La salida analógica es el valor de v_{out} calculado previamente, mientras que la salida digital es la salida de un amplificador operacional en modo comparador:

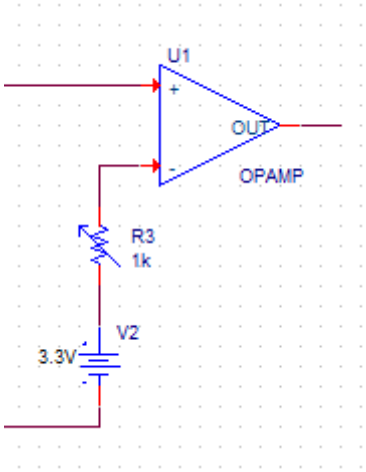

Figura 10. Esquema de un AO en modo comparador

En la patilla positiva del amplificador (la señal a comparar) se introduce la salida del divisor de tensión anterior y en la patilla negativa (la señal de referencia) se introduce un valor de tensión (valor umbral) que dependerá de un potenciómetro el cual puede ajustar el usuario. Si el valor de la patilla positiva supera el valor umbral, la salida será igual a 1, mientras qué si no lo supera, será 0.

Las patillas miden 8cm de largo, y la placa mide 1.4 x 3 cm. Además, dispone de 4 patillas:

- V_{CC} (1): es la patilla de alimentación de la placa. Además, será el valor de referencia del amplificador.
- GND (2): es la patilla de tierra de la placa.
- DO (3): es la salida digital de la placa (la salida del comparador). Se debe introducir en un pin de entrada y salida digital del ESP32 (en nuestro caso no se utilizará).
- AO (4): es la salida analógica de la placa (la salida del divisor de tensión). Se debe introducir en un pin de entrada y salida analógico del ESP32.

Puesto que la lectura de los datos analógicos de este sensor es un valor de tensión, no será necesario incluir ninguna biblioteca adicional. El código se explicará en el capítulo 6.

3.2. Sensores: Placa superior

3.2.1. Sensor de presión atmosférica: BMP280

Un sensor de presión atmosférica es un dispositivo que permite transformar en una señal eléctrica el valor de presión atmosférica en el aire. Para caracterizar este parámetro utilizaremos el BMP280.

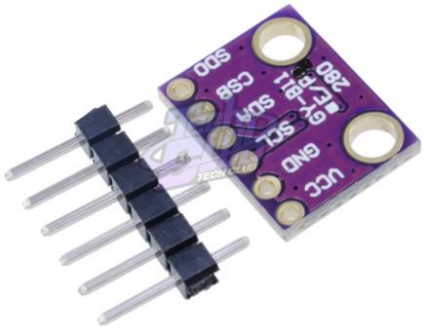

Imagen 8. BMP280

Como podemos ver en la Imagen 8, este dispositivo sólo está formado por una placa donde tiene incluidos los elementos sensitivos. En el datasheet se nos indica que el elemento sensitivo es un elemento piezorresistivo. En este tipo de sensores existe un puente Wheatstone con base de un material conductor (silicio, por ejemplo) que se extiende o se contrae bajo una determinada presión, de tal forma que varía la resistencia eléctrica del puente Wheatstone. Este cambio en la resistencia es controlado por un ASIC (Circuito Integrado de Aplicación Específica) mezclador de señal, el cuál realiza una conversión A/D con los valores de tensión medidos en el puente Wheatstone y obtiene los datos digitales de presión.

Este dispositivo nos permite 3 modos de funcionamiento:

- Sleep mode: en este modo no se realiza ninguna medida, y el dispositivo no consume apenas nada.
- Normal mode: en este modo el dispositivo realiza mediciones de forma periódica, manteniéndose en standby mientras no está en funcionamiento.
- Forced mode: en este modo se fuerza al sensor a realizar una única medición, volviendo a establecer el dispositivo en sleep mode.

Además, el dispositivo cuenta con un filtro IIR para minimizar las perturbaciones a corto plazo de los datos digitales. Con esta electrónica, las prestaciones que nos ofrece el BMP280 son las que podemos observar en la Tabla 3:

Tabla 3. Especificaciones del BMP280.

Parámetro	**BMP280**
Rango de V_{DD}	1.71 V → 5 V
Consumo	2.7 µA
Ruido RMS	1.3 Pa
Resolución	0.16 Pa
Interfaces	I^2C
Ratio de medición	Hasta 157 Hz
Medidas de la huella	2.0 mm x 2.5 mm

La placa dispone de 6 patillas:

- V_{CC} (1): es la patilla de la alimentación de la placa.
- GND (2): es la patilla de tierra de la placa.
- SCL (3): es la señal de reloj de entrada usada en la interfaz I^2C. Esta señal se encarga de indicar el período de envío de datos de los dispositivos esclavos (sensores).
- SDA (4): es la señal de datos de salida usada en la interfaz I^2C. Esta señal se encarga de enviar los datos cuando la señal de reloj se lo indique.
- CSB (5): esta patilla sirve para seleccionar el tipo de chip cuando se utiliza SPI. Para I^2C no hace falta que esté conectada.
- SDO (6): es la salida de datos serie utilizada en SPI. Para I^2C se debe conectar a tierra.

3.2.2. Sensor de lluvia: FC-3

Un sensor de lluvia es un dispositivo que permite cuantificar en un valor de tensión el nivel de lluvia en una superficie. Para caracterizar este parámetro utilizaremos el sensor de lluvia FC-3.

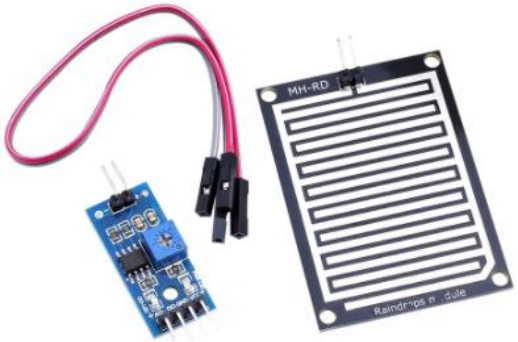

Imagen 9. FC-3

Como podemos ver en la Imagen 9, este dispositivo está compuesto por dos partes: una superficie y una placa. La superficie es el elemento sensitivo dónde se medirá el nivel de lluvia. En esta superficie tenemos dos pines. En cada pin encontramos un electrodo, cuya resistencia dependerá del nivel de lluvia, es decir, a mayor nivel de lluvia menor resistencia entre los electrodos. Esta resistencia (R_2) forma parte de un divisor de tensión junto a una resistencia fija (R_1), a la que se le introduce a la entrada el valor de alimentación (3.3V en nuestro caso).

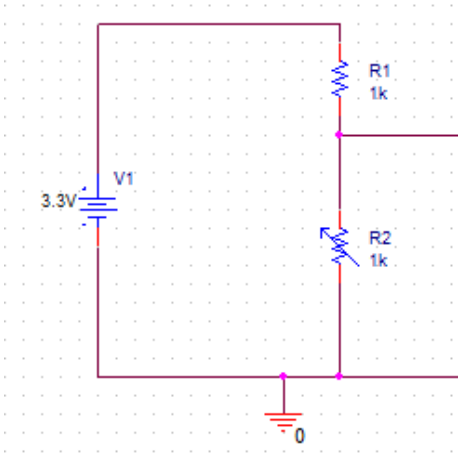

Figura 11. Esquema de un divisor de tensión

La ecuación que describe este circuito es $v_{out} = \frac{R_2}{R_1+R_2} v_{in}$:

Si $R_2 = 0$ (hay mucha lluvia y actúa como un cortocircuito) a la salida obtendremos un valor de $v_{out}=0$.

Si $R_2 = \infty$ (hay poca lluvia y actúa como un circuito abierto) a la salida obtendremos un valor de $v_{out}=(R_2/R_2)v_{in}=v_{in}=3.3V$. El resto de valores dependerá de la ecuación previamente dicha.

La placa dispone de dos salidas: una analógica y otra digital. La salida analógica es el valor de v_{out} calculado previamente, mientras que la salida digital es la salida de un amplificador operacional en modo comparador:

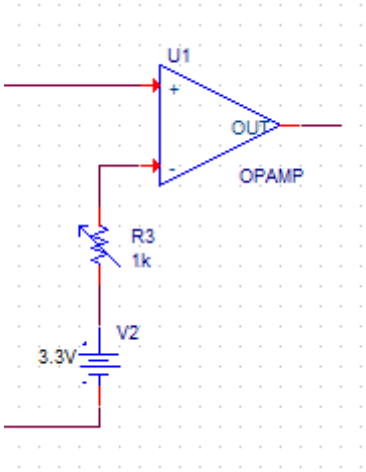

Figura 12. Esquema de un AO en modo comparador

En la patilla positiva del amplificador (la señal a comparar) se introduce la salida del divisor de tensión anterior y en la patilla negativa (la señal de referencia) se introduce un valor de tensión (valor umbral) que dependerá de un potenciómetro el cual puede ajustar el usuario. Si el valor de la patilla positiva supera el valor umbral, la salida será igual a 1, mientras qué si no lo supera, será 0.

Las medidas que nos indica el fabricante son de 5.4cm x 4cm para la superficie y de 3.1 cm x 1.42cm para la placa.

La placa dispone de 4 patillas:

- AO (1): es la salida analógica de la placa (la salida del divisor de tensión). Se debe introducir en un pin de entrada y salida analógico del ESP32.
- DO (2): es la salida digital de la placa (la salida del comparador). Se debe introducir en un pin de entrada y salida digital del ESP32 (en nuestro caso no se utilizará).
- GND (3): es la patilla de tierra de la placa.
- V_{CC} (4): es la patilla de alimentación de la placa. Además, será el valor de referencia del amplificador.

Puesto que la lectura de los datos analógicos de este sensor es un valor de tensión, no será necesario incluir ninguna biblioteca adicional. El código se explicará en el capítulo 6.

3.2.3. Sensor de luz UVA: VEML6075

Un sensor de luz es un dispositivo que permite convertir la intensidad lumínica en un valor de tensión. Para caracterizar este parámetro utilizaremos el VEML6075.

Imagen 10. VEML6075

Como podemos ver en esta Imagen 10, el dispositivo está formado por una placa donde tiene incluidos los elementos sensitivos. En el datasheet se nos indica que estos elementos sensitivos detectan rayos UVA y UVB y están formados por fotodiodos. Los fotodiodos son un tipo especial de diodos que generan corriente cuando son expuestos a la luz. A mayor cantidad de luz incidente mayor corriente eléctrica generada.
Para caracterizar esa señal de corriente se incluyen en la placa unos amplificadores y un circuito analógico/digital en un microchip CMOS. Con este equipamiento las prestaciones que nos ofrece el sensor son las que podemos observar en la Tabla 4:

Tabla 4. Características del VEML6075.

Voltaje de alimentación	Min: 1.7 V Max: 3.6 V
Temperatura de operación	Min: -40ºC Max: 85 ºC
Frecuencia de I^2C	Min: 10kHz Max: 400kHz
Responsividad UVA	0.93 counts/µW/cm^2
Responsividad UVB	2.1 counts/µW/cm^2

La placa tiene una huella de 1.7 x 1 cm y tiene 4 pines:

- GND (1): es la patilla de tierra de la placa.
- V_{CC} (2): es la patilla de la alimentación de la placa.
- SCL (3): es la señal de reloj de entrada usada en la interfaz I^2C. Esta señal se encarga de indicar el período de envío de datos de los dispositivos esclavos (sensores).
- SDA (4): es la señal de datos de salida usada en la interfaz I^2C. Esta señal se encarga de enviar los datos cuando la señal de reloj se lo indique.

3.3. Dispositivo LoRa: SX1278

Para realizar la transmisión de los datos vía Wi-Fi es necesario un módulo transmisor y receptor. A pesar de que la placa ESP32 ya tiene incorporado un módulo Wi-Fi, el rango de este es muy pequeño. Para poder realizar una transmisión con un rango amplio y con un consumo relativamente bajo la tecnología LoRa es ideal.

LoRa es una tecnología de modulación del tipo 'amplio espectro', lo que le permite tolerar ciertos fenómenos como ruido, efecto Doppler, efecto multicamino, etc, manteniendo un consumo de energía muy bajo. Debido a que está diseñado para aplicaciones con sensores de IoT, es una tecnología idónea para nuestro estudio.

Dentro de las opciones que tienen tecnología LoRa, escogeremos la placa SX1278.

Imagen 11. SX1278

Este módulo permite transmisiones de hasta +20dBm (100mW), y tiene una sensibilidad de -146.5dBm ($2.24 * 10^{-15}$mW). Para una frecuencia de 433MHz implica una distancia de 368.375km en condiciones teóricas aplicando la fórmula de Friis.

$$p_R = p_T \cdot \left(\frac{\lambda}{4\pi r}\right)^2 \rightarrow r = \frac{c}{4 \cdot \pi \cdot f \sqrt{\frac{p_R}{p_T}}} = \frac{3 \cdot 10^8}{4 \cdot \pi \cdot 433 \cdot 10^6 \sqrt{\frac{10^{20/10}}{10^{-146.5/10}}}} = 368.3755km$$

En una aplicación real, tenemos múltiples interferencias que disminuyen la distancia máxima, por lo que el fabricante nos recomienda utilizarlas para transmisiones de máximo 20km.

El patillaje que tenemos en nuestra placa es el siguiente:

- 3.3V: es la alimentación de la placa. Como indica el nombre, debe ser de 3.3V.
- GND: es la tierra de la placa.
- RST: es el pin de Reset del dispositivo. Cuando se pone a 1 reinicia el dispositivo.

- DIO0: es un pin digital de entrada y salida que permite la configuración del software. Esta patilla es opcional
- DIOn: tienen la misa función que DIO0. En nuestro esquema no serán utilizados.
- SCK: es la señal de reloj de entrada de la interfaz SPI.
- MISO: es la señal de salida de datos de la interfaz SPI.
- MOSI: es la señal de entrada de datos de la interfaz SPI.
- NSS: es la señal de entrada de detección del chip de la interfaz SPI.

4. Análisis de sistemas de alimentación

Una vez vistos todos los sensores que vamos a utilizar en el proyecto sólo nos falta saber cómo se van a alimentar estos dispositivos. En este TFG se busca un bajo consumo de los dispositivos utilizados, por lo que el principal objetivo será maximizar el tiempo de funcionamiento del dispositivo cuando esté conectado a una batería.
Si buscamos en las especificaciones de nuestro dispositivo ESP32, podemos ver que necesitamos alimentarlo entre 2.55 V y 3.6 V para que funcione correctamente. Nuestra fuente de alimentación debe mantenerse dentro de ese rango el mayor tiempo posible, independientemente de lo que utilicemos para conseguirlo. Una vez visto esto, pasamos a analizar diferentes opciones de alimentación:

4.1. Banco de energía

Imagen 12. Banco de energía.

En términos de capacidad, la primera opción que se nos viene a la cabeza es una batería portátil, las cuales podemos encontrar en el mercado por un precio de 20€ con capacidades de 26800mAh, que en teoría nos serviría para poder mantener alimentado el sistema por décadas. Sin embargo, el principal problema que presenta este tipo de alimentación para este tipo de aplicaciones es que ya de por si la batería tiene

incorporado un sistema de regulación de voltaje que lo modifica de 3.7 V a 5 V y, al enchufarlo a un ESP32, se vuelve a regular el voltaje de 5 V a 3.3 V. Estas conversiones consumen energía permanentemente, haciendo que el consumo de la batería sea mucho mayor. Además, algunas baterías tienen incorporadas un sistema de desconexión automática cuando no detectan cargas en sus puertos. Debido al bajo consumo del ESP32, se puede activar este sistema, dejando a la placa sin alimentación.

4.2. Baterías convencionales

Imagen 13. Pilas convencionales.

Otra opción es usar pilas convencionales o de níquel-metalhidruro. Estas pilas son las más comunes que se utilizan en aplicaciones de la vida cotidiana como alimentar un mando, una lámpara, un despertador, etc. El problema viene de que estas pilas ofrecen un voltaje medio de 1.2 V por cada pila, es decir, en el primer ciclo tiene un voltaje de 1.4 V y conforme va desgastándose la pila va disminuyendo el voltaje hasta un valor de 1 V. Si utilizamos dos pilas de este tipo tendríamos un voltaje de 2.8 V, que está por encima de los 2.55 V que necesitamos como mínimo para alimentar la placa, pero esta alimentación enseguida se quedaría por debajo del mínimo, así que esta opción no sirve. Por el otro lado, si utilizamos 3 pilas en serie tendríamos 4.2 V de carga, por lo que sobrealimentaríamos la placa, quemando el circuito. Otra opción sería utilizar un conversor de tensión para poder tener a la entrada de la placa una tensión dentro del rango, pero nos encontramos el mismo problema que en el apartado anterior: estos conversores consumen mucha energía para poder mantener el voltaje a la salida, por lo que el tiempo de vida del circuito se acorta. Además, a largo plazo, este tipo de alimentación empieza a fallar debido a que nuestra aplicación necesita picos de carga elevados para transmitir vía Wi-Fi, y en estos picos las pilas de Ni-MH dejan de funcionar.

4.3. Baterías de Litio

Imagen 14. Pilas de Litio.

Otra opción posible, y es la que utilizaremos en nuestras placas, son las baterías de Litio. La principal ventaja es que en el primer ciclo tenemos una carga máxima de 1.6 V, por lo que utilizando dos de ellas obtendríamos un voltaje de 3.2 V, que entra dentro del rango de operación del ESP32, evitando así el uso de conversores de tensión. Además, debido al uso de Litio, el problema de los picos de carga al utilizar la tecnología Lo-Ra se solucionan ya que consiguen proporcionar esa energía sin desplomar la tensión de la batería. Cada una de estas pilas tiene una carga de 1200 mAh, lo cual nos permite mantener el sistema funcionando durante varios años perfectamente, incluyendo los picos de caga para la transmisión Wi-Fi.

5. Diseño del prototipo

Una vez vistos todos los dispositivos que vamos a utilizar en este proyecto, vamos a diseñar las placas que vamos a utilizar en él. Primero se hará una breve definición de algunos elementos extras utilizados en cada una de las placas y luego se explicará cada una de ellas de forma individual.

5.1. Elementos adicionales.

En las placas hemos utilizado algunos componentes necesarios para poder realizar los prototipos de forma adecuada. Estos elementos son los siguientes:

- Zócalos: para facilitarnos el trabajo en algunas pruebas y para conectar y desconectar los dispositivos hemos soldado zócalos del tamaño de todos los elementos utilizados. En estos zócalos hemos introducidos todos los componentes.
- Patillas: los dispositivos LoRa y la mayoría de los sensores venían incluidos con unas patillas que había que soldarlos a las placas para poder trabajar con ellos. Estas patillas se conectan con los zócalos para que haya continuidad entre la placa y los dispositivos.
- Cables: para alimentar los dispositivos y para la transmisión y recepción de los datos hemos tenido que utilizar cables. Estos cables han sido soldados a la proto board con un soldador eléctrico y estaño.
- Protoboard: para poder situar todos los dispositivos hemos utilizado una protoboard de doble cara. Por una de las caras hemos introducido todos los zócalos y los cables y, por la otra cara, hemos soldado todo.
- Regletas de dos puertos: para alimentar todo el circuito hemos soldado una regleta en cada protoboard dónde irá introducida la fuente de alimentación. Gracias a esto podemos cambiar la fuente de alimentación si preferimos hacer pruebas en el laboratorio o con pilas en entornos outdoor.

Una vez explicados todos los componentes adicionales utilizados, vamos a mostrar cada una de las placas, los dispositivos por los que están formados y cómo funcionan:

5.2. Placa transmisora inferior.

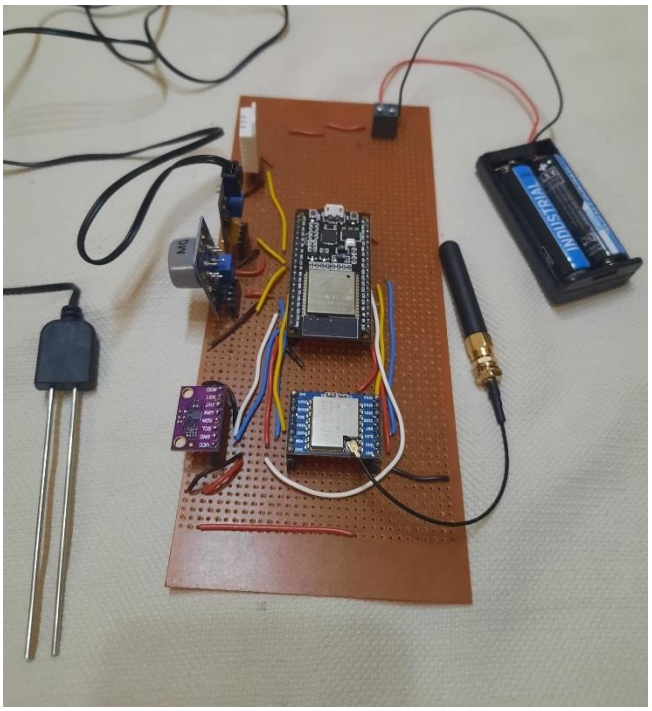

Imagen 15. Placa inferior. Cara de los dispositivos.

La primera placa que vamos a explicar es la placa transmisora que va situada a nivel del suelo. Como se ha explicado en puntos anteriores, se ha hecho esta distribución porque hay algunos sensores que es recomendable que se ubiquen en lugares específicos. En este caso el sensor que hemos utilizado es de detección de humedad en suelo, por lo que tiene que ir conectado en una zona de tierra. Como podemos ver en la Imagen 15, tenemos los sensores, el microcontrolador y el dispositivo LoRa conectados a los zócalos que van soldados a la protoboard. En los puntos 2 y 3 está explicado el patillaje de todos los dispositivos. En la Imagen 16 mostramos la cara de las soldaduras de la placa y que hay conectado en cada rama de la protoboard:

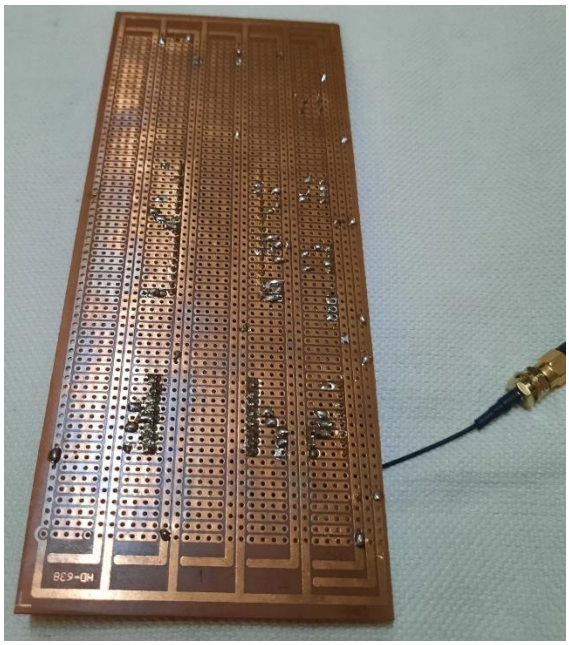

Imagen 16. Placa inferior. Cara de las soldaduras.

Cada sensor está conectado a las ramas de sus respectivas alimentaciones y a tierra, y también se conectan al ESP32 para transmitirle la información, ya sea de forma analógica o mediante la interfaz I^2C. El ESP32 se conecta con el dispositivo LoRa para transmitirle los datos de los sensores que tiene que enviar vía Wi-Fi y, por último, este los envía mediante la antena.

5.3. Placa transmisora superior.

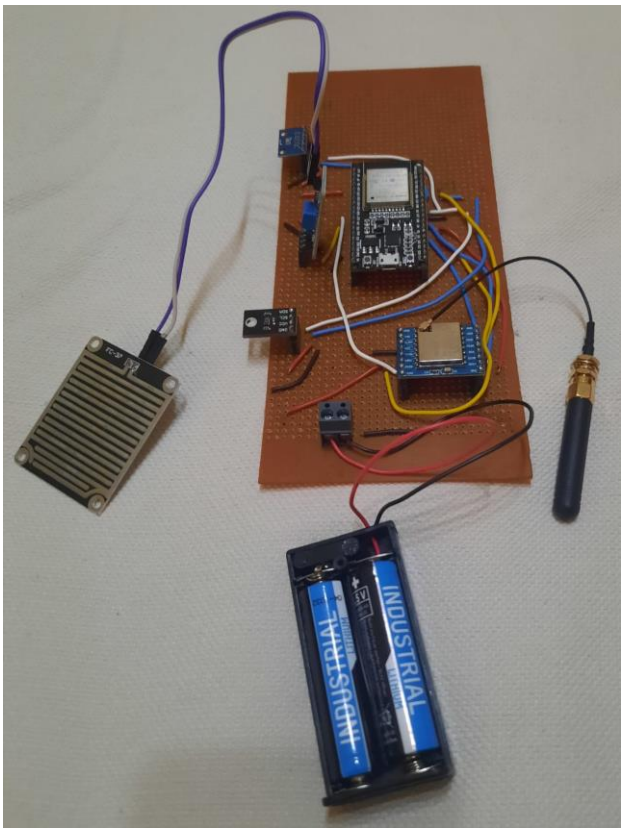

Imagen 17. Placa superior. Cara de los dispositivos.

Esta placa es la placa que va situada a una determinada altura. En este caso el sensor que determina la posición es un sensor de lluvia que hemos introducido en esta placa. Como podemos ver en la Imagen 17, tenemos los sensores, el microcontrolador y el dispositivo LoRa conectados a los zócalos que van soldados a la protoboard. En los puntos 2 y 3 está explicado el patillaje de todos los dispositivos. En la Imagen 18 mostramos la cara de las soldaduras de la placa y que hay conectado en cada rama de la protoboard:

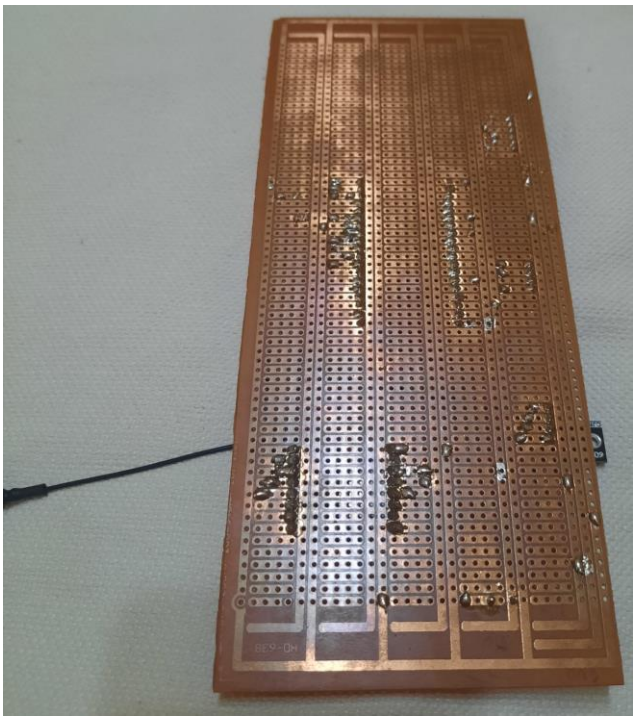

Imagen 18. Placa superior. Cara de las soldaduras.

Cada sensor está conectado a las ramas de sus respectivas alimentaciones y a tierra, y también se conectan al ESP32 para transmitirle la información, ya sea de forma analógica o mediante la interfaz I^2C. El ESP32 se conecta con el dispositivo LoRa para transmitirle los datos de los sensores que tiene que enviar vía Wi-Fi y, por último, este los envía mediante la antena.

5.4. Placa receptora.

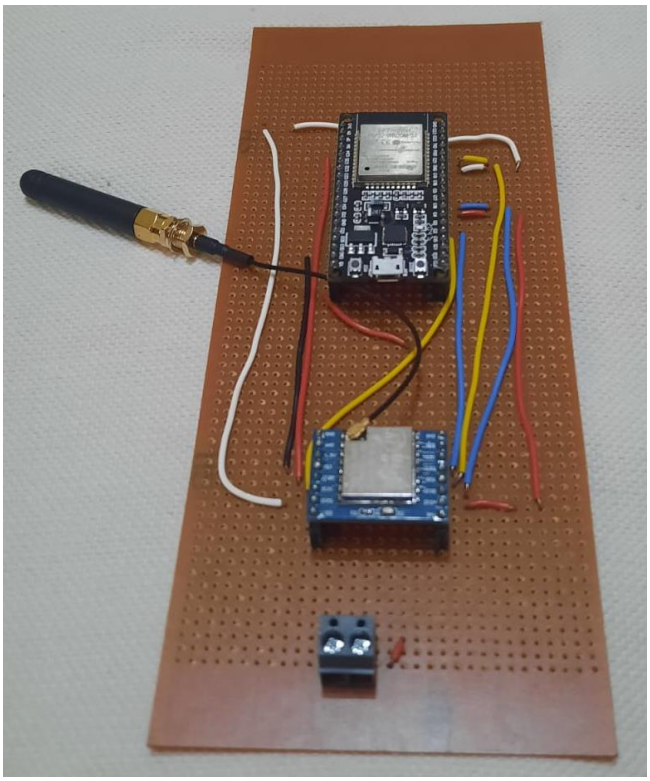

Imagen 19. Placa receptora. Cara de los dispositivos.

Como podemos ver en la Imagen 19 esta placa es mucho más sencilla que las anteriores, ya que sólo está formada por el ESP32 y por el dispositivo LoRa. Esto se debe a que esta placa sólo se dedica a recibir la información de las dos placas anteriores. El dispositivo LoRa procesa la información recibida por la antena y se la envía al ESP32 para que pueda informar de los datos obtenidos de los sensores. La distribución de la cara de las soldaduras aparece en la Imagen 20:

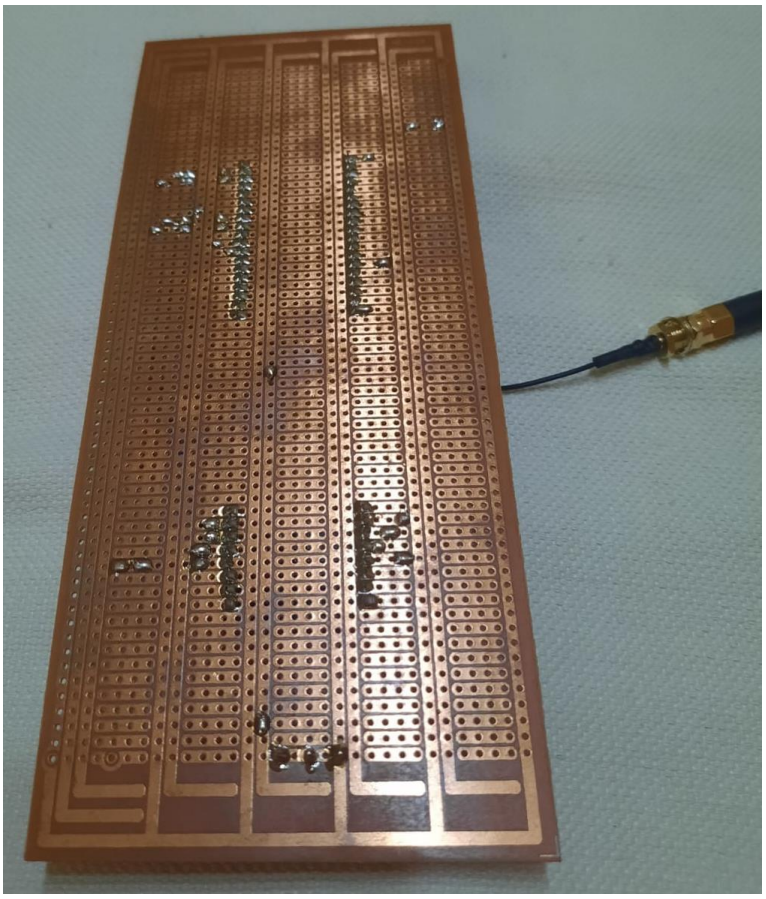

Imagen 20. Placa receptora. Cara de las soldaduras.

6. Códigos

Como se ha mencionado durante el trabajo, el lenguaje utilizado para la realización de este proyecto es Arduino. Para utilizarlo es necesario descargar la IDE de Arduino en su última versión. En nuestro caso utilizaremos la versión 1.8.15. El lenguaje de Arduino consta principalmente de 3 partes:

- Código antes del setup(): en esta parte del código se incluyen las variables generales y los include de las bibliotecas necesarias para el correcto funcionamiento del mismo. Como se ha explicado en el episodio 3, algunos sensores necesitan su propia biblioteca para obtener los códigos, mientras que otros, al ser una señal analógica, sólo necesitan una función determinada. En el programa de cada sensor se explicarán las bibliotecas utilizadas. Las variables generales consistirán en las inicializaciones de algunos parámetros que no varían durante todo el proceso, como el pin de lectura de los datos del ESP32 o el propio sensor.

- setup(): está función sólo se ejecuta una vez en todo el proceso. Para optimizar el uso de las baterías, incluiremos todo el código en esta parte, incluyendo las configuraciones de los sensores y la transmisión de los datos.

- loop(): está función es la parte del código que se ejecuta ininterrumpidamente mientras que la placa esté alimentada o conectada al puerto serie. Aquí sólo incluiremos la escucha por parte de la placa receptora.

Además, aparte de estas partes existen otras funciones llamadas métodos las cuales incluyen unos parámetros de entrada y un parámetro de salida, que puede ser de tipo void (vacío), entero, double, booleano, etc. Los métodos sirven para realizar funciones específicas que sean repetitivas en nuestro código y así evitar escribirlas varias veces. Para ejecutarlas, basta con "llamar" a la función, incluyendo los parámetros de entrada (en caso de ser necesarios) y de establecer un parámetro cómo salida de ese método (en caso de no ser de tipo void). Los métodos los hemos utilizado en el código para poder obtener los parámetros de interés que nos ofrecen los sensores de forma independiente.
Por último, para poder alargar el tiempo de funcionamiento vamos a utilizar el WatchDog Timer (perro guardián). El WDT es un bloque dentro de los microcontroladores que incluye un temporizador. Cuando el WDT se activa se encarga de poner el microcontrolador en Deep-sleep mode, lo que desactiva todas sus funciones a excepción de la del temporizador. Cuando se acaba el tiempo, el WDT reactiva el microcontrolador, ejecutando el código de nuevo. Para implementarlo en el ESP32 necesitamos incluir la biblioteca <esp_task_wdt.h>. Para poder ejecutarlo en el código necesitamos las siguientes líneas de código:

```
#define WDT_TIMEOUT x;
```

En esta línea inicializamos la variable WDT_TIMEOUT que utilizaremos para indicar cuanto tiempo debe permanecer el ESP32 en Deep-sleep Mode. X indica el tiempo en segundos.

```
esp_task_wdt_init(WDT_TIMEOUT, true);
esp_task_wdt_add(NULL);
```

En la primera línea le indicamos al WDT el tiempo que debe estar el dispositivo en Deep-sleep Mode. En la segunda le añadimos alguna tarea en especial en caso de ser

necesaria. Como no necesitamos ninguna, indicamos “null”. Estas líneas van en el setup().

```
int i = 0;
```

En esta línea creamos una variable que nos servirá para un bucle. Esta línea va antes del setup().

```
  if (i < 1) {
    esp_task_wdt_reset();
    i++;
 }
```

La función esp_task_wdt_reset(); nos sirve para resetear el temporizador del WDT. Es necesario un bucle para que se active, ya que si no se introduce el bucle no se resetea el temporizador.

Una vez visto las partes del código, se va a explicar el código asociado para cada placa. En el Anexo 1 se incluyen todas las clases utilizadas para este proyecto.

6.1. Placa inferior

Para simplificar el código y para poder diferenciar todas las partes entre sí, se han creado dos archivos para cada sensor y para LoRa y un programa principal llamado placa_inferior_datos que importe todos los programas y ejecute sus funciones.

Los dos archivos para cada sensor y para LoRa se corresponden con una extensión .cpp y otra .h.

El archivo con una extensión .h sólo incluye el nombre de las funciones de cada sensor, y será el que importe el código principal.

El archivo con una extensión .cpp será el que incluya todo el código asociado a la placa, como si del programa normal se tratara. Las condiciones para que funcione correctamente son las siguientes:

- Tiene que importar la biblioteca Arduino.h para poder trabajar como un Arduino.
- El programa principal debe importar el archivo.h.
- Tanto el archivo .cpp como el programa original deben importar las mismas bibliotecas en caso de necesitarlas.
- El archivo .cpp debe incluir las clases escritas en el archivo .h con su funcionamiento, ya que el programa principal las ejecutará cuando se las llame.

El programa principal consiste principalmente de 4 partes:

- Los include de las bibliotecas de Arduino necesarias para el correcto funcionamiento del código.
- Los include de las clases creadas por el usuario para cada sensor. Sólo es necesario importar los archivos .h para que las clases se utilicen correctamente.
- void setup(): aquí se establece la velocidad de comunicación en 115200 baudios para todos los sensores, se ejecutan las funciones que se corresponden con las configuraciones de los sensores, las funciones asociadas a las lecturas de los parámetros de sensores y del envío de los datos de forma ininterrumpida.
- void loop(): esta parte no se utiliza.

6.1.1. Sensor de CO_2 y componentes orgánicos volátiles: CCS811

Para el sensor CCS811 necesitamos importar la biblioteca Adafruit_CCS811.h, la cual incluye todas las funciones necesarias para trabajar con el CCS811.
Recordar que esta biblioteca también se tiene que importar en el programa principal para que pueda funcionar correctamente al haber creado un archivo .cpp y otro archivo .h.
La configuración de CCS_811.h sólo incluye la declaración de las clases configuracion_CCS_811(), getCO2() y getTVOC().
La configuración de CCS_811.cpp tiene varias partes.
Antes de configuración_CCS_811() debemos crear un objeto de la clase Adafruit_CCS811 para poder trabajar con el sensor. En nuestro caso será ccs.
En la parte de configuracion_CCS_811() confirmamos que se ha iniciado el código y esperamos a que el sensor esté listo para funcionar.
En las funciones getCO2() y getTVOC() incluimos el código necesario para poder leer estos parámetros del sensor. Este código consiste en un promedio de 100 muestras de estos parámetros. Luego se devuelven estos valores de tipo double y se igualan a un parámetro del código principal para que sea posible trabajar con ellos.
Todo el código se muestra en el apartado 10.1.4.

6.1.2. Sensor de CO y gases combustibles: MQ_9

Para el sensor MQ_9 no es necesario incluir ninguna biblioteca ya que la lectura es puramente analógica.
La configuración de MQ_9.h sólo incluye la declaración de las clases configuracion_MQ_9() y getGases().

La configuración de MQ_9.cpp tiene varias partes. En la parte antes de configuracion_MQ_9() incluimos esta línea de código que indica el pin donde se va a realizar la lectura de los datos analógicos:

const int MQ9Pin=4;

En este caso se indica el pin GIOP4, que se corresponde con el pad 26 del ESP32 .

En la parte de configuracion_MQ_9() simplemente confirmamos que se ha iniciado el código.

En getGases() declaramos el valor de R0 obtenido en la calibración. Después incluimos la lectura analógica del pin indicado previamente en el parámetro HDPin con la función analogRead(HDPin). Esta lectura obtiene un valor de 12 bits (entre 0 y 4095). Este valor lo normalizamos a 5V y lo dividimos entre R0 para obtener el coeficiente Rs/R0.

Todo el código se muestra en el apartado 10.1.5.

6.1.3. Sensor de temperatura y humedad: DHT22

Para el sensor DHT22 necesitamos importar 2 bibliotecas adicionales:

- Adafruit_Sensor: Esta biblioteca se necesita para todos los sensores creados por Adafruit. En ella se incluyen todos los parámetros necesarios para poder trabajar con los sensores.
- DHT_Sensor_Library: Esta biblioteca incluye todo lo necesario para poder extraer los datos del DHT22. Estos datos son transmitidos de una forma concreta, no como un único valor de tensión, por lo que son necesarias unas funciones especiales para obtenerlos. Estas funciones vienen incluidas en las bibliotecas DHT.h y DHT_U.h, las cuales importaremos en el código. Una vez hecho esto, podremos trabajar con el sensor.

Recordar que estas bibliotecas también se tienen que importar en el programa principal para que pueda funcionar correctamente al haber creado un archivo .cpp y otro archivo .h.

La configuración de DHT_22.h sólo incluye la declaración de las clases configuracion_DHT_22(), getTemperatura() y getHumedad().

La configuración de DHT_22.cpp tiene varias partes. En la parte antes de configuracion_DHT_22() declaramos el tipo de dht que tenemos (existen el DHT11 y el DHT22), el pin del que se van a leer los datos (el nº 15 que se corresponde con el pin 23) y creamos un objeto de la clase DHT, introduciendo como parámetros el pin de lectura de los datos y el tipo de dispositivo.

La primera línea se encarga de definir el tipo de DHT que tenemos (existe el DHT11 y el DHT22), la segunda línea declara el pin del que se van a leer los datos (el nº 15 se corresponde con el GIOP15, es decir, el pad 23) y la tercera línea crea un objeto de la clase DHT, introduciendo como parámetros el pin de lectura de los datos y el tipo de dispositivo.
En la parte de configuracion_DHT_22() simplemente confirmamos que se ha iniciado el código y activamos al sensor para que empiece a transmitir datos.
En las funciones getHumedad() y getTemperatura() incluimos el código necesario para poder leer estos parámetros del sensor. Además, incluimos una condición de que si el valor obtenido es NaN se devuelva un valor cero. Estas funciones devuelven los valores de temperatura y humedad y, en el código principal, un parámetro toma dicho valor haciendo que sea posible trabajar con él
Todo el código se muestra en el apartado 10.1.6.

6.1.4. Sensor de humedad en suelo: HD-38

Para el sensor HD-38 no es necesario incluir ninguna biblioteca ya que la lectura es puramente analógica.
La configuración de HD_38.h sólo incluye la declaración de las clases configuracion_HD_38() y getHumedad_suelo().
La configuración de HD_38.cpp tiene varias partes. En la parte antes de configuracion_HD_38() incluimos esta línea de código que indica el pin donde se va a realizar la lectura de los datos analógicos:
const int HDPin=0;
En este caso se indica el pin GIOP0, que se corresponde con el pad 25 del ESP32.
En la parte de configuracion_HD_38() simplemente confirmamos que se ha iniciado el código.
En getHumedad_suelo() incluimos la lectura analógica del pin indicado previamente en el parámetro HDPin con la función analogRead(HDPin). Esta lectura obtiene un valor de 12 bits (entre 0 y 4095).
Todo el código se muestra en el apartado 10.1.7.

6.1.5. LoRa

Para realizar esta parte del código nos hemos basado en el ejemplo LoRaSender que nos ofrece la propia biblioteca. En este código se importas las bibliotecas LoRa.h y SPI.h, necesarias para poder trabajar con la placa correctamente.

La configuración de LoRa.h sólo incluye la declaración de las clases configuracion_LoRa() y envio_placa_inferior(double temperatura, double humedad_aire, int humedad_suelo, double CO2, double TVOC, double RS).
La configuración de LoRa.cpp tiene varias partes.
En la parte de configuracion_LoRa() confirmamos que se ha iniciado el código y establecemos los pins de RST, NSS, DIO0, necesarios para el correcto funcionamiento de la placa. Después incluimos una condición para que se nos informe si la placa está conectada y operativa.
En envio_placa_inferior(double temperatura, double humedad_aire, int humedad_suelo, double CO2, double TVOC, double RS) introducimos como parámetros de entrada todos los valores que hemos obtenido en los get anteriores. Con estos valores realizaremos la transmisión LoRa.
Simplemente inicializamos el paquete mediante el comando LoRa.beginPacket(), introducimos con los comandos LoRa.print() y LoRa.println() toda la información de los parámetros de entrada y cerramos el paquete con LoRa.endPacket().
Todo el código se muestra en el apartado 10.1.8.

6.2. Placa superior

Para simplificar el código y para poder diferenciar todas las partes entre sí, se han creado dos archivos para cada sensor y para LoRa y un programa principal llamado placa_superior_datos que importe todos los programas y ejecute sus funciones.
Los dos archivos para cada sensor y para LoRa se corresponden con una extensión .cpp y otra .h.
El archivo con una extensión .h sólo incluye el nombre de las funciones de cada sensor, y será el que importe el código principal.
El archivo con una extensión .cpp será el que incluya todo el código asociado a la placa, como si del programa normal se tratara. Las condiciones para que funcione correctamente son las siguientes:

- Tiene que importar la biblioteca Arduino.h para poder trabajar como un Arduino.
- El programa principal debe importar el archivo.h.
- Tanto el archivo .cpp como el programa original deben importar las mismas bibliotecas en caso de necesitarlas.
- El archivo .cpp debe incluir las clases escritas en el archivo .h con su funcionamiento, ya que el programa principal las ejecutará cuando se las llame.

El programa principal consiste principalmente de 4 partes:

- Los include de las bibliotecas de Arduino necesarias para el correcto funcionamiento del código.
- Los include de las clases creadas por el usuario para cada sensor. Sólo es necesario importar los archivos .h para que las clases se utilicen correctamente.
- void setup(): aquí se establece la velocidad de comunicación en 115200 baudios para todos los sensores, se ejecutan las funciones que se corresponden con las configuraciones de los sensores, las funciones asociadas a las lecturas de los parámetros de sensores y del envío de los datos de forma ininterrumpida.
- void loop(): esta parte no se utiliza.

6.2.1. Sensor de presión atmosférica: BMP280

Para el sensor CCS811 necesitamos importar la biblioteca Adafruit_BMP280.h, la cual incluye todas las funciones necesarias para trabajar con el BMP280.
Recordar que esta biblioteca también se tiene que importar en el programa principal para que pueda funcionar correctamente al haber creado un archivo .cpp y otro archivo .h.
La configuración de BMP_280.h sólo incluye la declaración de las clases configuracion_BMP_280() y getPresion().
La configuración de CCS_811.cpp tiene varias partes.
Antes de configuración_CCS_811() debemos crear un objeto de la clase Adafruit_BMP_280 que se llamará bmp y otro de la clase Adafruit_Sensor que provenga del método bmp.getPressureSensor() para poder trabajar con el sensor.
En la parte de configuracion_BMP280() confirmamos que se ha iniciado el código y esperamos a que el sensor esté listo para funcionar.
Además, incluimos los valores base para el muestreo, los cuales incluyen el modo de operación, el tiempo de muestreo para la presión y el filtrado.
En la función getPresion() incluimos el código necesario para poder obtener la presión. Este código consiste en crear una variable llamada pressure_event de tipo sensors_event_t, la cual informa al sensor que tiene que devolver el valor de presión. Luego se devuelve este valor de tipo double y se iguala a un parámetro del código principal para que sea posible trabajar con él.
Todo el código se muestra en el apartado 10.2.2.

6.2.2. Sensor de lluvia: FC-3

Para el sensor FC-3 no es necesario incluir ninguna biblioteca ya que la lectura es puramente analógica.

La configuración de FC-3.h sólo incluye la declaración de las clases configuracion_ FC_3() y getLluvia().

La configuración de FC-3.cpp tiene varias partes. En la parte antes de configuracion_ FC_3() incluimos esta línea de código que indica el pin donde se va a realizar la lectura de los datos analógicos:

const int HDPin=33;

En este caso se indica el pin GIOP33, que se corresponde con el pad 8 del ESP32.

En la parte de configuracion_FC_3() simplemente confirmamos que se ha iniciado el código.

En el getLluvia() incluimos la lectura analógica del pin indicado previamente en el parámetro HDPin con la función analogRead(HDPin). Esta lectura obtiene un valor de 12 bits (entre 0 y 4095).

Todo el código se muestra en el apartado 10.2.3.

6.2.3. Sensor de luz UVA: VMLE6075

Para el sensor CCS811 necesitamos importar la biblioteca Adafruit_VEML6075.h, la cual incluye todas las funciones necesarias para trabajar con el VEML6075.

Recordar que esta biblioteca también se tiene que importar en el programa principal para que pueda funcionar correctamente al haber creado un archivo .cpp y otro archivo .h.

La configuración de BMP_280.h sólo incluye la declaración de las clases configuracion_VEML6075 (), getUVA(), getUVB() y getUVI().

La configuración de CCS_811.cpp tiene varias partes.

Antes de configuración_CCS_811() debemos crear un objeto de la clase Adafruit_VEML6075 que se llamará uv y que provenga del método Adafruit_VEML6075() para poder trabajar con el sensor.

En la parte de configuracion_VEML6075() confirmamos que se ha iniciado el código y esperamos a que el sensor esté listo para funcionar.

Además, incluimos los valores base para el muestreo, los cuales incluyen la constante de integración, el modo de operación y los coeficientes de calibración.

En las funciones getUVA(), getUVB() y getUVI() simplemente se devuelven los valores leídos mediante los métodos readUVA(), readUVB() y readUVI(). Luego estos valores

se igualan a varios parámetros del código principal para que sea posible trabajar con ellos.

En el código principal se ha incluido un bucle for para que las funciones de lectura de los datos se ejecuten varias veces con un retardo de 1 segundo entre ellos. Esto se ha implementado debido a que la librería VEML6075 necesita unos coeficientes para poder ajustar la lectura de los datos. Estos coeficientes necesitan un “entrenamiento” para poder mostrar un resultado más preciso, por lo que es necesario que el programa tome varias muestras espaciadas para ofrecer una mejor resolución. Además, conseguimos que haya un retardo entre la información de la placa superior y de la placa inferior, lo que permite que no se mezclen los datos en la placa receptora.

Todo el código se muestra en el apartado 10.2.4.

6.2.4. LoRa

Para realizar esta parte del código nos hemos basado en el ejemplo LoRaSender que nos ofrece la propia biblioteca. En este código se importas las bibliotecas LoRa.h y SPI.h, necesarias para poder trabajar con la placa correctamente.

La configuración de LoRa.h sólo incluye la declaración de las clases configuracion_LoRa() y envio_placa_superior(int lluvia, double presion, double UVA, double UVB, double UVI).

La configuración de LoRa.cpp tiene varias partes.

En la parte de configuracion_LoRa() confirmamos que se ha iniciado el código y establecemos los pins de RST, NSS, DIO0, necesarios para el correcto funcionamiento de la placa. Después incluimos una condición para que se nos informe si la placa está conectada y operativa.

En envio_placa_superior(int lluvia, double presion, double UVA, double UVB, double UVI) introducimos como parámetros de entrada todos los valores que hemos obtenido en los get anteriores. Con estos valores realizaremos la transmisión LoRa.

Simplemente inicializamos el paquete mediante el comando LoRa.beginPacket(), introducimos con los comandos LoRa.print() y LoRa.println() toda la información de los parámetros de entrada y cerramos el paquete con LoRa.endPacket().

Todo el código se muestra en el apartado 10.2.5.

6.3. Placa receptora

En este programa sólo hemos necesitado el programa principal. Este programa está basado en el ejemplo LoRaReceiver que nos ofrece la biblioteca LoRa. En este código se importas las bibliotecas LoRa.h y SPI.h, necesarias para poder trabajar con la placa correctamente.

El programa placa_receptora tiene varias partes.

En la parte de setup() confirmamos que se ha iniciado el código y establecemos los pins de RST, NSS, DIO0, necesarios para el correcto funcionamiento de la placa. Después incluimos una condición para que se nos informe si la placa está conectada y operativa.

En loop() introducimos el código necesario para poder recibir un paquete vía Wi-Fi. En este código se crea una variable llamada packetSize que viene del método LoRa.parsePacket(). Esto nos indica si se ha recibido un paquete o no. Si se ha recibido, se crea una variable llamada message de tipo String donde se irá incluyendo todo lo que se reciba a través del comando LoRa.read(), el cual se activa mientras que no se recibe el comando LoRa.endPacket(). Cuando se recibe todo el mensaje, se muestra al usuario por el puerto serie.

Todo el código se muestra en el apartado 10.3.1.

7. Puesta en funcionamiento. Resultados

Una vez escogidos todos los componentes que van a conformar nuestro prototipo, diseñadas y soldadas todas las PCBs y creados los códigos necesarios para el correcto funcionamiento de las placas, debemos comprobar si nuestro proyecto tiene el comportamiento esperado. Para ello, se realizarán las pruebas en un solar situado en Murcia, enfrente de Terra Natura, el cuál dispone de las condiciones necesarias para poder comprobar el funcionamiento real de las placas, ya que no tiene elementos que puedan interferir en nuestra prueba.

Imagen 21. Solar de pruebas.

En este solar hemos situado la estación receptora, montada en el coche, con la placa receptora conectada a un ordenador portátil para poder observar en el puerto serie los datos recibidos por la placa.

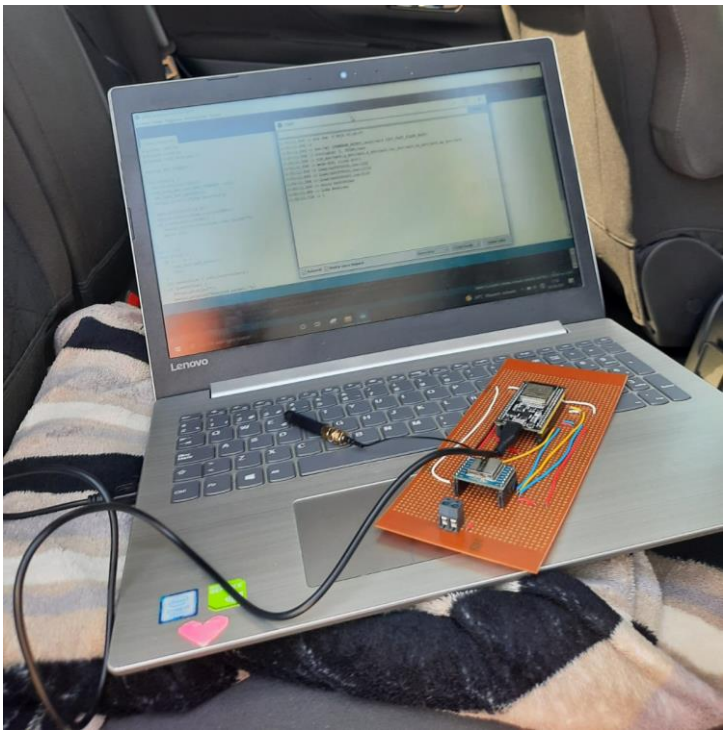

Imagen 22. Estación receptora.

Para las estaciones transmisoras, se ha situado la placa inferior en el suelo, clavando en la tierra el sensor de humedad de suelo para poder observar su funcionamiento, y la placa superior se ha quedado sostenida por nosotros en un punto más elevado:

Imagen 23. Placa transmisora inferior.

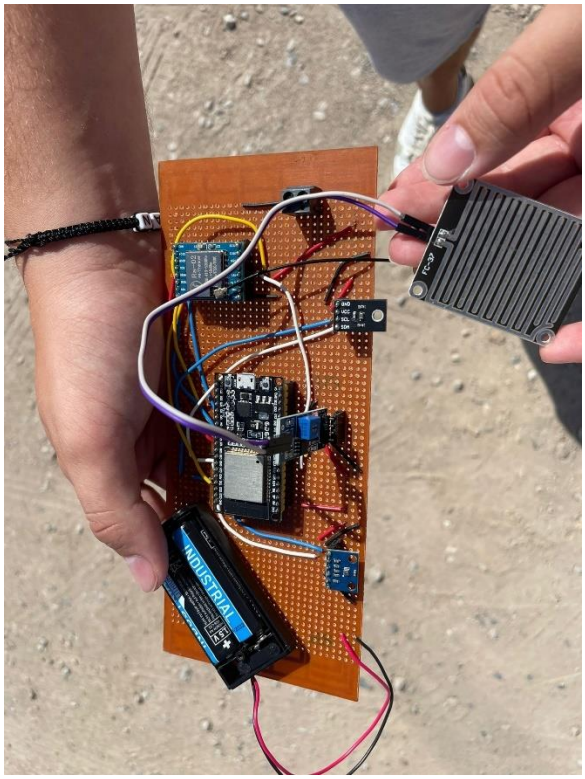

Imagen 24. Placa transmisora superior.

Con todas las placas montadas se han realizado dos tipos de pruebas para probar el correcto funcionamiento de ellas:

- Prueba de sensores: la primera prueba se ha realizado a una distancia de 3.2m entre las estaciones transmisoras y receptoras, con el fin de asegurar el buen funcionamiento de los sensores. En estas condiciones hemos alterado el estado de los sensores de humedad de suelo y de lluvia, mojando con agua los elementos sensitivos de ambos para observar cómo varía la información enviada. Recordemos que para estos sensores el valor de 4095 implica la presencia nula de agua en ellos. En estas condiciones, los datos obtenidos en la placa receptora son los que se pueden observar en la Imagen 25:

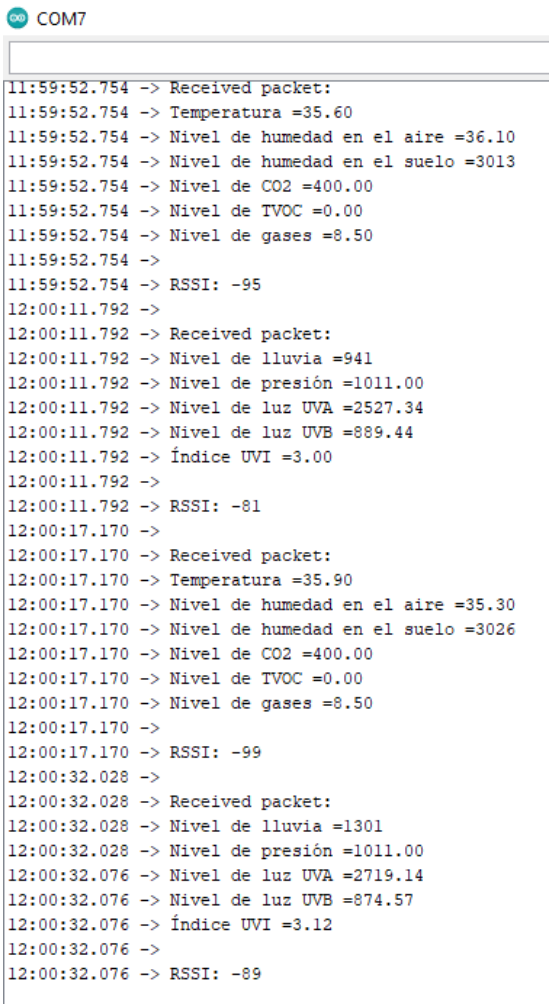

Imagen 25. Datos recibidos.

Como podemos observar, en la estación receptora se ha recibido la información transmitida por las placas correctamente. De la Imagen 25 podemos sacar varias conclusiones:

+ Funcionamiento del WatchDog Timer: Si observamos la Imagen 25, en el lado de la izquierda disponemos del instante temporal en el que se ha

escrito el texto en el puerto serie. En esta prueba hemos escogido un tiempo de WDT de 20 segundos, para poder comprobar de una forma más rápida su funcionamiento. Si nos fijamos en los instantes de tiempo, podemos comprobar que el tiempo entre la recepción de los paquetes ronda los 20 segundos, lo que demuestra el funcionamiento del WDT.

+ Sensores de humedad de suelo y de lluvia: En la Imagen 25 podemos leer los niveles de humedad de suelo y de lluvia que ha recibido la placa. En el primer paquete, estos niveles estaban situados en 3013 para la humedad del suelo y 941 para la lluvia. Cuando se ha recibido el segundo paquete ha habido una variación muy pequeña para la humedad en el suelo a 3026, pero el nivel de lluvia sí que ha sufrido una variación más notoria a 1301. Además, con otros parámetros ha ocurrido algo similar, cómo el nivel UVI, la temperatura y la humedad, pero con una variación muy pequeña.

- Prueba de distancia: Una vez realizada correctamente la prueba de los sensores, se ha comprobado la distancia dentro de la cual los datos se siguen enviando de forma correcta. Para ello se han cogido las dos placas transmisoras y se han ido alejando de la estación receptora. Mientras tanto, se ha ido observando en el ordenador cuándo se recibían paquetes y cuándo no. En esta prueba se ha llegado a una transmisión de 98.5 metros entre transmisor y receptor, punto en el cual a veces llegaba la transmisión y otras no. En la Imagen 26 podemos observar un caso de paquetes recibidos correctamente a esa distancia, pero en la Imagen 27 podemos observar otro caso en el que se perdían parte de los datos de un paquete:

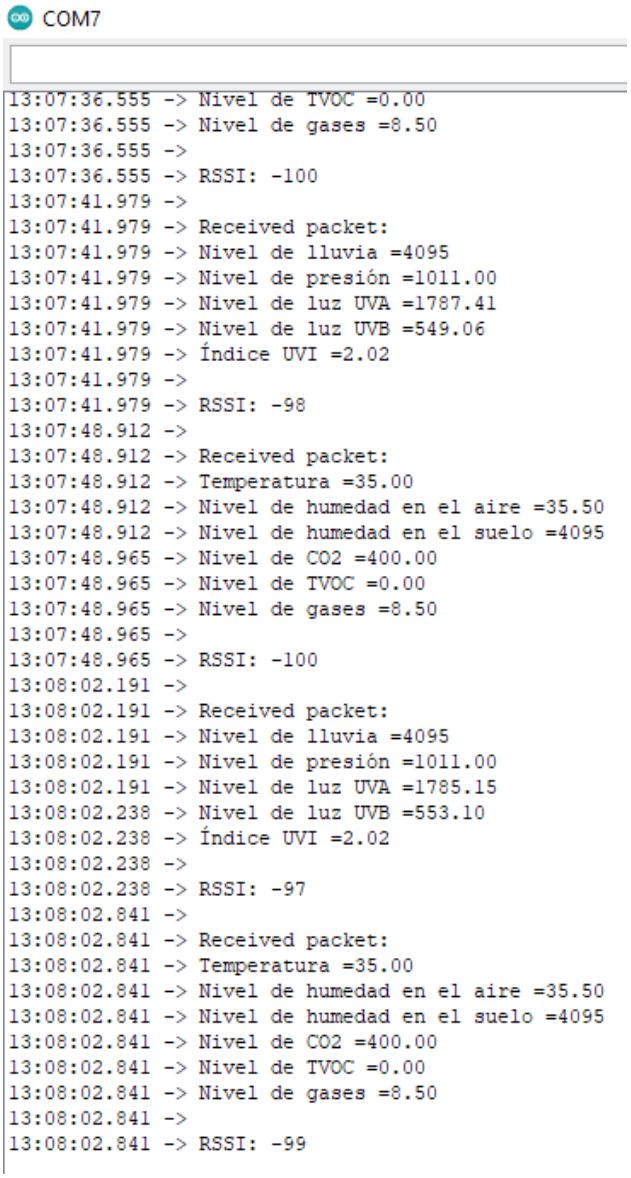

Imagen 26. Paquete de datos sin pérdidas.

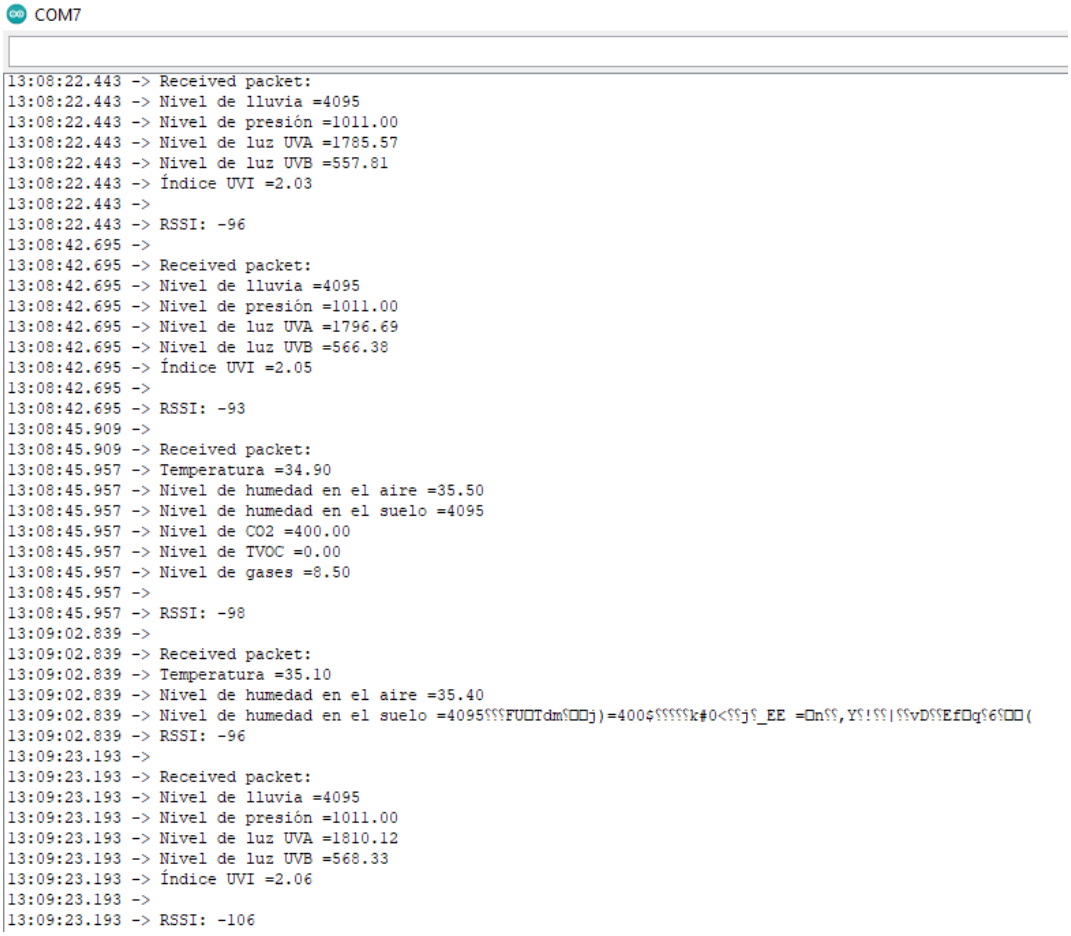

Imagen 27. Paquete de datos con pérdidas.

8. Conclusión y futuras líneas de investigación.

Observando los resultados de las pruebas, podemos concluir que se han logrado los objetivos propuestos en el punto 1.2, consiguiendo una correcta transmisión de todos los datos meteorológicos en una distancia máxima de 100 metros. Sin embargo, como se explicó en el punto 3.3, la tecnología LoRa es capaz de transmitir datos hasta una distancia de 20km, medida que está bastante lejos de la obtenida. Esto se debe a diversos factores, como el tipo de antena utilizado (se ha utilizado una antena omnidireccional, la cual radia hacia todas las direcciones con la misma potencia). Si utilizáramos antenas direccionales entre transmisores y receptor podríamos obtener distancias mucho mayores, ya que este tipo de antenas concentra la mayoría de la potencia de radiación en una única dirección. Otro factor que ha influido es la posición de las antenas, ya que si se hubiera podido situar las antenas transmisoras en un punto más elevado se podrían haber obtenido unas distancias mayores. Estas mejoras se podrían implementar en las placas adquiriendo el material necesario y trabajando en un

entorno más favorable. Otra forma de conseguir mejores resultados sería realizando una mayor inversión en los dispositivos utilizados en las PCBs. Con esto, no sólo podríamos aumentar de forma considerable la distancia de transmisión, sino que también se podría optimizar el tiempo de la batería y reducir el tamaño de las placas, pudiendo crear una versión comercial más compacta con mejores prestaciones para el cliente.

9. Bibliografía

Sánchez, C. (08 de febrero de 2019). Normas APA – 7ma (séptima) edición. Normas APA (7ma edición). https://normas-apa.org/

Cisco (2005). Cisco Annual Internet Report (2018–2023).

Tekniker (s.f) https://www.tekniker.es/es/redes-de-sensores

UPCT(s.f) https://www.upct.es/saladeprensa/notas.php?id=4884

Qartech (s.f) https://qartech.io/

Alldatasheet (s.f) para todos los datasheets de los sensores

Connor, N. (09 de marzo de 2020). ¿Cuál es el principio de funcionamiento de los detectores de semiconductores? Definición. https://www.radiation-dosimetry.org/es/cual-es-el-principio-de-funcionamiento-de-los-detectores-de-semiconductores-definicion/

Naylampmechatronics (s.f) https://naylampmechatronics.com/sensores-temperatura-y-humedad/47-sensor-de-humedad-de-suelo-fc-28.html

Aliexpress(s.f) para comprar los dispositivos y para la información de algunos sensores

Kistler (s.f) https://www.kistler.com/es/glosario/termino/sensor-de-presion-piezorresistivo/

Unicrom (s.f) https://unicrom.com/fotodiodo/

Amazon para búsqueda de productos similares en el mercado

Arduino UNO: https://es.wikipedia.org/wiki/Arduino_Uno

Rodrigo Hernández (09 de diciembre de 2019) ¿Qué es la tecnología LoRa y por qué es importante para IoT? https://www.thethingsnetwork.org/community/santa-rosa/post/que-es-la-tecnologia-lora-y-por-que-es-importante-para-iot

Radioshuttle(s.f) https://www.radioshuttle.de/es/medias-es/informaciones-tecnicas/esp32-alimentado-por-bateria/

RS(s.f) https://es.rs-online.com/web/

10. ANEXO 1: CÓDIGOS DE ARDUINO

10.1. Placa inferior

10.1.1. Placa_inferior_calibración

```
#include "MQ_9_cal.h"

void setup() {
 Serial.begin(115200);
 configuracion_MQ_9_cal();
}
void loop() {
 programa_MQ_9_cal();
}
```

10.1.2. MQ_9

MQ_9_cal.ccp

```
#include <Arduino.h>


const int MQ9Pin=4;


void configuracion_MQ_9_cal(){
  Serial.println("Sensor MQ-9 start");
}


void programa_MQ_9_cal(){


 float sensor_volt;
 float RS_air;
 float R0;
 float sensorValue;


   for(int x = 0 ; x < 100 ; x++)
 {
   sensorValue = sensorValue + analogRead(MQ9Pin);
 }
 sensorValue = sensorValue/100.0;
 sensor_volt = (sensorValue/1024)*5.0;
```

```
 RS_air = (5.0-sensor_volt)/sensor_volt;
 R0 = RS_air/9.9;
 Serial.print("Sensor_Value= ");
 Serial.println(sensorValue);
 Serial.print("sensor_volt = ");
 Serial.print(sensor_volt);
 Serial.println("V");
 Serial.print("R0 = ");
 Serial.println(R0);
 delay(1000);
}
```

MQ_9_cal.h

```
void configuracion_MQ_9_cal();
void programa_MQ_9_cal();
```

10.1.3. Placa_inferior_datos

```
#include <DHT.h>
#include <DHT_U.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_CCS811.h>
#include <SPI.h>
#include <LoRa.h>

#include "DHT_22.h"
#include "HD_38.h"
#include "CCS_811.h"
#include "MQ_9.h"
#include "LoRa.h"
#include <esp_task_wdt.h>

#define WDT_TIMEOUT 10
double temperatura = 0;
double humedad_aire = 0;
int humedad_suelo = 0;
double RS = 0;
```

```
double CO2 = 0;
double TVOC = 0;
int i = 0;
void setup() {
  Serial.begin(115200);
  esp_task_wdt_init(WDT_TIMEOUT, true);
  esp_task_wdt_add(NULL);
  configuracion_DHT_22();
  configuracion_HD_38();
  configuracion_CCS_811();
  configuracion_MQ_9();
  configuracion_LoRa();
  if (i < 1) {
    esp_task_wdt_reset();
    i++;
 }
 temperatura = getTemperatura();
 humedad_aire = getHumedad_aire();
 humedad_suelo = getHumedad_suelo();
 CO2 = getCO2();
 TVOC = getTVOC();
 RS = getGases();
 envio_placa_inferior(temperatura, humedad_aire, humedad_suelo, CO2, TVOC, RS);
}

void loop() {
}
```

10.1.4. CCS811

CCS_811.ccp

```
#include <Adafruit_CCS811.h>
#include <Arduino.h>

Adafruit_CCS811 ccs;

void configuracion_CCS_811(){
 Serial.println("CCS811 test");
```

```
  if(!ccs.begin()){
    Serial.println("Failed to start sensor! Please check your wiring.");
    while(1);
  }
  while(!ccs.available());
}
double getCO2(){
  double eCO2=0;
   if(ccs.available()){
    if(!ccs.readData()){
        for(int x = 0 ; x < 100 ; x++)
        {
        eCO2 = eCO2 + ccs.geteCO2();
    }
     eCO2 = eCO2/100;
    }
    else{
      Serial.println("ERROR!");
      while(1);
    }
  }
  return eCO2;
}

double getTVOC(){
  double TVOC=0;
   if(ccs.available()){
    if(!ccs.readData()){
        for(int x = 0 ; x < 100 ; x++){
        TVOC=TVOC+ccs.getTVOC();
    }
     TVOC = TVOC/100;
    }
    else{
      Serial.println("ERROR!");
      while(1);
    }
```

```
  }
  return TVOC;
}
```

CCS_811.h

```
void configuracion_CCS_811();
double getCO2();
double getTVOC();
```

10.1.5. MQ-9

MQ_9.cpp

```
#include <Arduino.h>

const int MQ9Pin=4;

void configuracion_MQ_9(){
 Serial.println("Sensor MQ-9 start");
}

double getGases(){

 float R0 = 0.54;
 int sensorValue = analogRead(MQ9Pin);
 double sensor_volt = ((float)sensorValue / 1024) * 5.0;
 double RS_gas = (5.0 - sensor_volt) / sensor_volt;
 return RS_gas;
}
```

MQ_9.h

```
void configuracion_MQ_9();
double getGases();
```

10.1.6. DHT_22

DHT_22.cpp

```
#include <DHT.h>
```

```
#include <DHT_U.h>

#include <Adafruit_Sensor.h>
#include <Arduino.h>
#define DHTTYPE DHT22

//#define DHTTYPE DHT21

const int DHTPin=15;

DHT dht(DHTPin, DHTTYPE);

void configuracion_DHT_22(){
   Serial.println("DHT22 test!");
   dht.begin();
}

double getTemperatura(){
  double temperatura = dht.readTemperature();
  if (isnan(temperatura)){
    return 0;
  }
  else return temperatura;
}

double getHumedad_aire(){
  double humedad = dht.readHumidity();
  if (isnan(humedad)){
    return 0;
  }
  else return humedad;
}
```

DHT_22.h

```
void configuracion_DHT_22();
double getTemperatura();
double getHumedad_aire();
```

10.1.7. HD_38

HD_38.cpp

```
#include <Arduino.h>

const int HDPin=0;
void configuracion_HD_38(){
  Serial.println("Sensor HD-38 start");
}

int getHumedad_suelo(){
  return analogRead(HDPin);
}
```

HD_38.h

```
void configuracion_HD_38();
int getHumedad_suelo();
```

10.1.8. LoRa

LoRa.cpp

```
#include <Arduino.h>

#include <SPI.h>
#include <LoRa.h>

void configuracion_LoRa() {
  while (!Serial);

  Serial.println("LoRa Sender");

  LoRa.setPins(26,27,14);
  Serial.println(LoRa.begin(434E6));
  if (!LoRa.begin(434E6)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
}
```

```
void envio_placa_inferior(double temperatura, double humedad_aire, int
humedad_suelo,
double CO2, double TVOC, double RS){
  LoRa.beginPacket();
  LoRa.print("Temperatura =");
  LoRa.println(temperatura);
  Serial.println(temperatura);
  LoRa.print("Nivel de humedad en el aire =");
  LoRa.println(humedad_aire);
  Serial.println(humedad_aire);
  LoRa.print("Nivel de humedad en el suelo =");
  LoRa.println(humedad_suelo);
  Serial.println(humedad_suelo);
  LoRa.print("Nivel de CO2 =");
  LoRa.println(CO2);
  Serial.println(CO2);
  LoRa.print("Nivel de TVOC =");
  LoRa.println(TVOC);
  Serial.println(TVOC);
  LoRa.print("Nivel de gases =");
  LoRa.println(RS);
  Serial.println(RS);
  LoRa.endPacket();
  delay(5000);
}
```

LoRa.h

```
void configuracion_LoRa();
void envio_placa_inferior(double temperatura, double humedad_aire, int
humedad_suelo, double CO2, double TVOC, double RS);
```

10.2. Placa superior

10.2.1. Placa_superior

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

```
#include <LoRa.h>
#include <Adafruit_BMP280.h>
#include <Adafruit_VEML6075.h>

#include "LoRa.h"
#include "BMP_280.h"
#include "FC_3.h"
#include "VMLE_6075.h"
#include <esp_task_wdt.h>

#define WDT_TIMEOUT 20

int lluvia = 0;
double presion = 0;
double UVA = 0;
double UVB = 0;
double UVI = 0;
int i = 0;
void setup() {
  Serial.begin(115200);
  esp_task_wdt_init(WDT_TIMEOUT, true);
  esp_task_wdt_add(NULL);
  configuracion_BMP_280();
  configuracion_FC_3();
  configuracion_VMLE_6075();
  configuracion_LoRa();
    if (i < 1) {
     esp_task_wdt_reset();
     i++;
 }
  lluvia = getLluvia();
  presion = getPresion();

    for(int j = 0; j < 3; j++){
     UVA = getUVA();
     UVB = getUVB();
     UVI = getUVI();
```

```
    delay(1000);
  }
   envio_placa_arriba(lluvia, presion, UVA, UVB, UVI);
}

void loop() {
}
```

10.2.2. BMP_280

BMP_280.cpp

```
#include <Wire.h>
#include <SPI.h>
#include <Adafruit_BMP280.h>
#include <Arduino.h>

Adafruit_BMP280 bmp; // use I2C interface
Adafruit_Sensor *bmp_pressure = bmp.getPressureSensor();

void configuracion_BMP_280(){

  Serial.println(F("BMP280 Sensor event test"));

  //if (!bmp.begin(BMP280_ADDRESS_ALT, BMP280_CHIPID)) {
  if (!bmp.begin()) {
    Serial.println(F("Could not find a valid BMP280 sensor, check wiring or "
                     "try a different address!"));
    while (1) delay(10);
  }

  /* Default settings from datasheet. */
  bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,     /* Operating Mode. */
                  Adafruit_BMP280::SAMPLING_X2,     /* Temp. oversampling */
                  Adafruit_BMP280::SAMPLING_X16,    /* Pressure oversampling */
                  Adafruit_BMP280::FILTER_X16,      /* Filtering. */
                  Adafruit_BMP280::STANDBY_MS_500); /* Standby time. */
```

```
  bmp_temp->printSensorDetails();
}

double getPresion(){
  sensors_event_t pressure_event;
  bmp_pressure->getEvent(&pressure_event);
  return pressure_event.pressure;
}
```

BMP_280.h

```
void configuracion_BMP_280();
double getPresion();
```

10.2.3. FC_3

FC_3.cpp

```
#include <Arduino.h>

const int FCPin=33;

void configuracion_FC_3(){
   Serial.println("Sensor FC-23 start");
}

int getLluvia(){
  return analogRead(FCPin);
}
```

FC_3.h

```
void configuracion_FC_3();
int getLluvia();
```

10.2.4. VEML_6075

VEML_6075.cpp

```
#include <Wire.h>
#include "Adafruit_VEML6075.h"
#include <Arduino.h>

Adafruit_VEML6075 uv = Adafruit_VEML6075();

void configuracion_VMLE_6075(){

  Serial.println("VEML6075 Full Test");
 if (! uv.begin()) {
   Serial.println("Failed to communicate with VEML6075 sensor, check wiring?");
 }
 Serial.println("Found VEML6075 sensor");

 // Set the integration constant
 uv.setIntegrationTime(VEML6075_100MS);
 // Get the integration constant and print it!
 Serial.print("Integration time set to ");
 switch (uv.getIntegrationTime()) {
   case VEML6075_50MS: Serial.print("50"); break;
   case VEML6075_100MS: Serial.print("100"); break;
   case VEML6075_200MS: Serial.print("200"); break;
   case VEML6075_400MS: Serial.print("400"); break;
   case VEML6075_800MS: Serial.print("800"); break;
 }
 Serial.println("ms");

 // Set the high dynamic mode
 uv.setHighDynamic(true);
 // Get the mode
 if (uv.getHighDynamic()) {
   Serial.println("High dynamic reading mode");
 } else {
   Serial.println("Normal dynamic reading mode");
 }

 // Set the mode
```

```
  uv.setForcedMode(false);
  // Get the mode
  if (uv.getForcedMode()) {
    Serial.println("Forced reading mode");
  } else {
    Serial.println("Continuous reading mode");
  }

  // Set the calibration coefficients
  uv.setCoefficients(2.22, 1.33,  // UVA_A and UVA_B coefficients
                     2.95, 1.74,  // UVB_C and UVB_D coefficients
                     0.001461, 0.002591); // UVA and UVB responses

}

double getUVA(){
  return uv.readUVA();
}

double getUVB(){
  return uv.readUVB();
}

double getUVI(){
  return uv.readUVI();
}
```

VEML_6075.h

```
void configuracion_VMLE_6075();
double getUVA();
double getUVB();
double getUVI();
```

10.2.5. LoRa

LoRa.cpp

```
#include <Arduino.h>

#include <SPI.h>
#include <LoRa.h>

void configuracion_LoRa() {
  while (!Serial);

  Serial.println("LoRa Sender");
  LoRa.setPins(17,16,22);
  Serial.println(LoRa.begin(434E6));
  if (!LoRa.begin(434E6)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
}

void envio_placa_superior(int lluvia, double presion, double UVA, double UVB, double
UVI){
  LoRa.beginPacket();
  LoRa.print("Nivel de lluvia =");
  LoRa.println(lluvia);
  Serial.println(lluvia);
  LoRa.print("Nivel de presión =");
  LoRa.println(presion);
  Serial.println(presion);
  LoRa.print("Nivel de luz UVA =");
  LoRa.println(UVA);
  Serial.println(UVA);
  LoRa.print("Nivel de luz UVB =");
  LoRa.println(UVB);
  Serial.println(UVB);
  LoRa.print("Índice UVI =");
  LoRa.println(UVI);
  Serial.println(UVI);
  LoRa.endPacket();
  Serial.println("Vabien");
```

```
  delay(5000);
}
```

LoRa.h

```
void configuracion_LoRa();
void envio_placa_superior(int lluvia, double presion, double UVA, double UVB, double
UVI);
```

10.3. Placa receptora

10.3.1. Placa_receptora

```
#include <SPI.h>
#include <LoRa.h>
#include <esp_task_wdt.h>

#define WDT_TIMEOUT 1

void setup() {
  Serial.begin(115200);
  esp_task_wdt_init(WDT_TIMEOUT, true);
  esp_task_wdt_add(NULL);
  Serial.println("LoRa Receiver");

  LoRa.setPins(17,16,22);
  Serial.println(LoRa.begin(434E6));
  if (!LoRa.begin(434E6)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
}
int i = 0;
void loop() {
  if (i < 1) {
      esp_task_wdt_reset();
      i++;
```

```
  }
  int packetSize = LoRa.parsePacket();
  if (packetSize) {
    Serial.println("");
    Serial.println("Received packet: ");
    String message = "";
    while (LoRa.available()) {
      message += (char)LoRa.read();
    }
    Serial.println(message);
    Serial.print("RSSI: ");
    Serial.println(LoRa.packetRssi());
  }
}
```