Proyecto Fin de Carrera
Ingeniería Electrónica, Robótica y Mecatrónica

Puesta en marcha de un sistema de pruebas del estándar de comunicaciones LoRa

Autor: Raúl Morillo Ciuciumán
Tutor: Manuel Ángel Perales Esteve

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Proyecto Fin de Carrera
Ingeniería Electrónica, Robótica y Mecatrónica

Puesta en marcha de un sistema de pruebas del estándar de comunicaciones LoRa

Autor:
Raúl Morillo Ciuciumán

Tutor:
Manuel Ángel Perales Esteve
Profesor titular

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Proyecto Fin de Carrera: Puesta en marcha de un sistema de pruebas del estándar de comunicaciones LoRa

Autor: Raúl Morillo Ciuciumán

Tutor: Manuel Ángel Perales Esteve

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Antes de comenzar el trabajo de fin de grado, me gustaría reconocer brevemente al gran número de personas que han hecho posible la conclusión de su redacción.

En primer lugar, a mi familia, quienes me han motivado y apoyado durante todos los años de mi vida en el ámbito académico, proyectando valores necesarios como la perseverancia, constancia y aspiración necesaria para decidirme a aumentar mis conocimientos. A mi madre y mis abuelos, por facilitarme el transcurso del grado con sus cuidados y cariños, a mi hermana, y especial mención a mi padre, quien despertó desde pequeño, mi curiosidad por las tecnologías y la electrónica, introduciéndome en este maravilloso mundo de la ingeniería.

Por otro lado, a mi pareja y amigos, tanto a los que han estado conmigo durante la infancia y adolescencia, como a aquellos tantos que he conocido en el transcurso por la universidad, porque, aunque de formas distintas, todos han contribuido a que me desarrolle personal y profesionalmente. El trabajo en equipo, las discusiones técnicas y, sobre todo, las risas, me renovaban las energías y avivaban las ganas de continuar.

De la misma manera, quiero agradecer a todos los docentes que me habéis acompañado estos años, enseñándonos de distintas formas los conocimientos teóricos y habilidades necesarias para aplicarlos. Gracias, por entender los momentos de dificultad por los que todos hemos pasado, por animarnos a continuar y potenciar nuestras capacidades. No hay un gran profesional sin que exista un gran maestro detrás.

Por todo ello, considero a este trabajo un culmen de todos los conocimientos, experiencias y sobre todo la cantidad de excelentes personas que han estado a mi alrededor, ayudándome a afrontar los problemas que surgían en el camino e impulsándome a continuar mejorando. Gracias a las personas anteriormente citadas, siento mucho cerrar con este proyecto lo que ha sido, sin duda alguna, la mejor etapa de mi vida.

Muchas gracias.

Raúl Morillo Ciuciumán

Sevilla, 2020

Resumen

Este proyecto trata de introducir y descubrir las características de las redes IoT. Haciendo énfasis en un caso de uso concreto: una red LPWAN haciendo uso del estándar de comunicación mediante modulación de onda de largo alcance y bajo consumo, LoRa, implementado en un dispositivo microcontrolador fabricado por Pycom, llamado LoPy4.

Tras una introducción acerca del internet de las cosas y las redes LPWAN, se va a detallar las características del estándar LoRa, como de su protocolo de red global LoRaWAN. Por otro lado, se detalla el uso, las especificaciones y librerías incluidas en el microcontrolador LoPy4, así como se muestran ejemplos de programación y uso de este dispositivo, tanto haciendo uso de LoRa, como de sus distintos periféricos.

Por último, se realizan experimentos de alcance y potencia del LoPy4 al momento de establecer redes privadas basadas en LoRa.

Palabras clave: IoT, LoRa, LoPy4, Pycom, MicroPython, LPWAN

Abstract

This project tries to introduce and discover the characteristics of IoT networks. Emphasizing a specific use case: LPWAN network using the communication standard through long-range wave modulation and low consumption, LoRa, implemented in a microcontroller device manufactured by Pycom, called LoPy4.

After an introduction about the Internet of Things and LPWAN networks, the characteristics of the LoRa standard will be detailed, as well as its global LoRaWAN network protocol. On the other hand, the use, specifications and libraries included in the LoPy4 microcontroller are detailed, as well as examples of programming and use of this device, both making use of LoRa, and its different peripherals.

Finally, LoPy4 scope and power experiments are performed when establishing LoRa-based private networks.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

La tecnología hizo posible las grandes poblaciones; ahora las grandes poblaciones hacen que la tecnología sea indispensable.

- José Krutch-

La tecnología ha avanzado mucho en el siglo XXI, muchas veces impresiona ver hasta dónde pueden llegar los avances que cada día se producen por todo el mundo. No parece existir un límite para este tipo de innovaciones. Hoy en día, la tecnología está presente en prácticamente todos los ámbitos del planeta. Desde la comunicación, gracias a las redes e internet, hasta la agricultura, automatizando procesos de riego o arado. Desde la medicina, con brazos robóticos que operan con mucha más precisión que un cirujano, hasta la música y sus efectos. Es indiscutible la necesidad real que tenemos hoy en día los seres humanos de la tecnología, para llevar a cabo nuestras labores rutinarias.

Es por esto, que existe una demanda abultada sobre los dispositivos, cada vez más modernos, con nuevas características actualizadas según los requerimientos del mercado actual. Se ha creado por lo cual, un proceso constante de reinvención de las tecnologías que se emplean cada día.

La consecuencia de este desarrollo exponencial de la tecnología, es la incorporación de la electrónica digital y la computación a la mayor parte de objetos cotidianos. Por ejemplo, frigoríficos, luces, cafeteras, sistemas de calefacción y refrigeración, relojes, termómetros, altavoces… Un sin fin de objetos, que cada día son más, los que se suman a esta nueva moda digital.

Figura 1: Maquinilla de afeitar con conexión BLE a app móvil

El principal propósito de la incorporación de computación en los objetos cotidianos es la de automatizar los procesos que llevan a cabo, y/o la monitorización de ellos mismos. Es decir, en el caso del frigorífico, se le incorpora una pantalla táctil desde la cual puede ver y ajustar las diferentes temperaturas, muestra la hora y puedes programar alarmas, activar el dispensador de cubos de hielo e incluso tiene conexión Bluetooth para conectarse con la app móvil del fabricante y poder cambiar los ajustes desde el dispositivo. Si hace siglos era impensable la existencia de un frigorífico, hace décadas eran impensables todas estas nuevas funcionalidades.

Hay que mencionar, que como previamente se comentó, todo esto viene de la mano de la constante mejora de los dispositivos electrónicos digitales, los cuales nos permiten llevar a cabo esta computación.

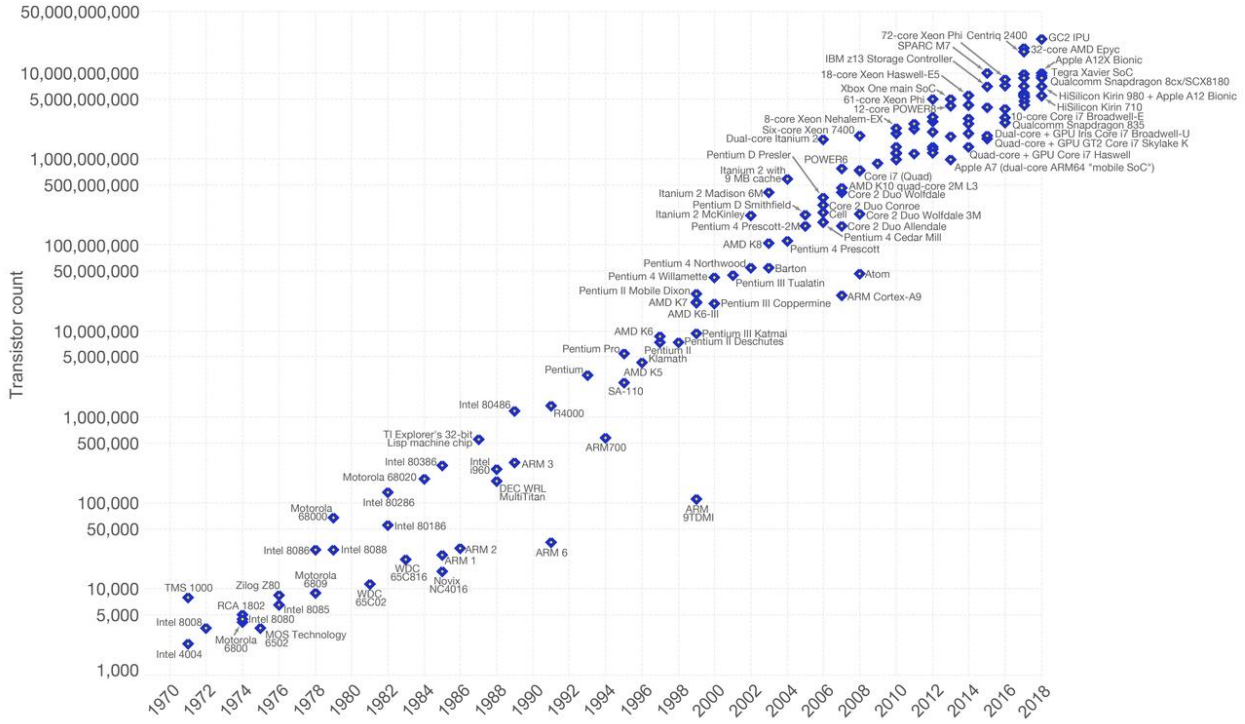


Figura 2: Gráfico de la evolución de número de transistores por procesador

Si observamos la Figura 2, podemos apreciar la evolución de los procesadores con el paso del tiempo. En medio siglo, han pasado de tener 4004 transistores por procesador, a aproximadamente 40 millones de transistores, en un tamaño de muchísimo más pequeño, del orden de mm^2. Este crecimiento ha permitido desarrollar más cálculos, más complejos y más rápido, siendo este avance el gran culpable de la revolución digital.

Gracias aparición de los microprocesadores, se ha conseguido desarrollar microcontroladores capaces de gestionar múltiples procesos de comunicación, actuación o monitorización. Están compuestos por diferentes bloques, como el microprocesador, la memoria, los buses de datos, la unidad de cálculo y periféricos como convertidores analógico-digitales, módulos de entrada y salida, etc. Son capaces de alojar sentencias de código de programación en su memoria programable, por lo que pueden utilizarse para llevar a cabo infinitas aplicaciones

Con el paso del tiempo y gracias a la aparición de los microcontroladores, miles de objetos hoy en día están bajo las órdenes de estos dispositivos. Gracias a la programación de estas placas de desarrollo, es posible integrar otros periféricos como tarjetas WiFi o Bluetooth, que permiten conectar los objetos a la red o a un dispositivo móvil. Es así como poco a poco se consolidó el concepto de IoT (Internet of Things).

2 INTERNET OF THINGS

Si tu negocio no está en internet, tu negocio no existe

- Bill Gates -

Este nuevo concepto, surgió entre 2008 y 2009. Debido a la convergencia de las tecnologías inalámbricas y los dispositivos microelectrónicos y micromecánicos. Es una red de comunicación M2M (Machine to machine) que engloba conexión que de millones de dispositivos entre ellos con el fin de facilitar nuestro día a día gracias a la monitorización y automatización de los procesos. Actualmente, para 2021, se prevé que haya cerca de 50.000 millones de dispositivos IoT en funcionamiento en todo el mundo.

Para entenderlo mejor, con IoT se trata de conectar la mayor parte de dispositivos inteligentes a una red común para poder interactuar todos con todos y tener control de ellos desde un mismo punto de acceso. Por ejemplo, un pastor, desde su casa, podría ser capaz de localizar a cada una de sus ovejas gracias a un dispositivo IoT con GPS en el collar, o cerrar y abrir las puertas de sus corrales gracias a dispositivos IoT con accionadores instalados en las puertas.

2.1. Concepto de IoT

Para entender el término IoT, debemos dividirlo en las 2 partes que componen su nombre:

2.1.1 Internet

La primera, es nuestra ya conocida red de redes, o internet. No hace falta mencionar que permite tener a millones de dispositivos conectados entre sí para compartir información, acceder a recursos, servicios y páginas web a través de protocolos muy estructurados en distintas capas. Internet hace que un sistema IoT tenga cobertura mundial, es decir, el pastor podría hacer lo mismo que desde su casa en cualquier parte del mundo.

Internet suele ser el objetivo final en las aplicaciones IoT, pero para llegar a la red de redes, muchas veces la información viaja por otros medios físicos hasta que alguno de sus nodos intermedios con conexión a internet, pone en manos de la red los datos requeridos. Antes de que esto suceda, la información ha podido viajar por muchos medios como pueden ser Bluetooth, modulación en radiofrecuencia como LoRa o Sigfox, comunicación satélite… La combinación de los distintos métodos de comunicación, hacen posible que la información llegue a internet y así el usuario final puede acceder a ella desde cualquier parte del mundo.

Igualmente, a parte de la monitorización de datos remota, también podemos revertir el flujo, y enviar órdenes a través de internet a las puertas de enlace con el resto de métodos de conexión, para que así la orden llegue al nodo terminal y ejecute la acción sobre el sistema.

2.1.2 Las cosas

La segunda parte, como bien viene implícita en un nombre (Internet of Things), son las "cosas". Desde un coche hasta una lavadora, pasando por una cafetera. Estos dispositivos deben estar dotados de sensores y circuitos integrados que permitan su monitorización, actuación y comunicación, ya sea entre ellos, o con internet. A estos dispositivos se les conoce como nodos IoT.

Estos nodos IoT o dispositivos inteligentes deben ser capaces de:

- Monitorear cualquier tipo de información (altitud, presión, temperatura, velocidad…).
- Controlar algún tipo de acción a partir del monitoreo (enviar o recuperar información, encender o apagar un interruptor, controlar un motor…).
- Automatizar un proceso para realizar una rutina programable.
- Optimizar los recursos.

Hoy en día, casi cualquier objeto puede ser candidato a formar parte de la red IoT, y ya existen aplicaciones IoT en prácticamente todos los ámbitos posibles.

En resumen, IoT trata de conectar los diferentes objetos inteligentes entre sí y con servidores de internet para compartir información y conseguir así analizar los datos y optimizar los procesos de cada una de las diferentes aplicaciones.

2.2. Objetivos de IoT

IoT se enfrenta a varios retos y dificultades para conseguir desarrollar una completa red de dispositivos funcionando y comunicándose correctamente. Estos son los objetivos a alcanzar para una red IoT ideal:

- Conseguir optimizar el consumo eléctrico de los dispositivos, que, al tener sensores y circuitos integrados, consumen más energía.
- Administrar bien el ancho de banda y conseguir más y mejores enlaces para no colapsar la red debido a la enorme cantidad de dispositivos conectados enviando datos simultáneamente.
- Disminuir el tamaño de los sensores y actuadores para mejorar la portabilidad y funcionalidad de los dispositivos inteligentes.

Sin duda queda un largo camino para conseguir escalar la red a niveles mundiales, pero son objetivos que se alcanzarán con el paso del tiempo, como siempre ha sucedido en la tecnología.

2.3. Adversidades de IoT

Sin embargo, IoT se ve perjudicada por factores que impiden un desarrollo más sencillo:

- Al ser dispositivos concretos y específicos, la posibilidad de actualización de los mismos es más complicadas y en caso de falla es más difícil de reparar.
- La privacidad se ve invadida, ya que para beneficiarse de los beneficios de dispositivos IoT, habrá que brindar información que puede ser de carácter personal y confidencial.
- La vulnerabilidad se ve muy afectada debido a la poca seguridad informática desarrollada en estos productos.

Poco a poco, IoT va perfeccionando la manera de conectarse con otros dispositivos entre las diferentes capas. Por ejemplo, Sigfox o LoRaWAN, son redes privadas con protocolos de seguridad creadas precisamente para intentar superar dichas adversidades.

2.4. Medios de conexión

Para conectar los dispositivos IoT entre ellos, no tiene que ser necesariamente por internet a través de WiFi o Ethernet. Existen otro tipo de comunicaciones que se usan para ello, empleando otras capas físicas como pueden ser la radiofrecuencia o la conexión vía satélite. Por ejemplo, BLE (Bluetooth Low Energy), Sigfox, LTE (Long Term Evolution), ZigBee… son distintos métodos de conexión que poseen diferentes características. Según el tipo de aplicación, conviene más usar una tecnología u otra.

A continuación, veremos una comparativa de diferentes métodos de comunicación según su alcance y ancho de banda.

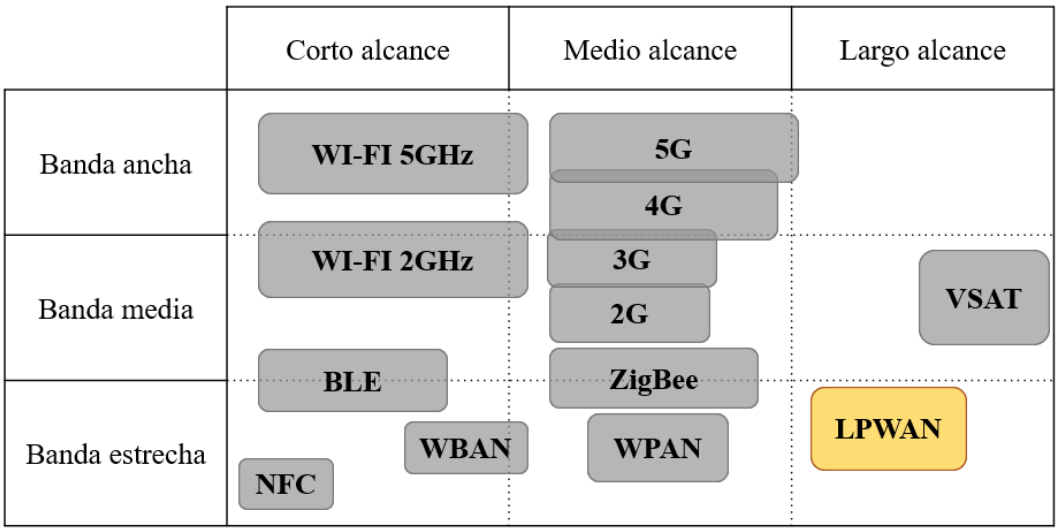


Figura 3: Comparación según alcance y ancho de banda de diferentes tecnologías de comunicación

Como se puede apreciar en la Figura 3, cada una de las tecnologías cubre un tipo de necesidades. Por ejemplo, una red WiFi de 5GHz trabaja con un gran ancho de banda, por lo que logra una gran velocidad de transmisión de datos. Sin embargo, la conexión satélite VSAT, es bastante más lenta, pero permite una mayor cobertura. Según el objetivo de la aplicación que se desarrolle, es conveniente usar un método de conexión u otro, ya que sería un malgasto energético realizar una conexión de 20m de longitud con la tecnología 5G, pudiendo usar BLE.
Como se puede apreciar en la tabla de la imagen, se ha resaltado LPWAN (Low-power wide area network) ya que las analizaremos más a fondo. Son redes de banda estrecha, con una transmisión de datos lenta, pero con largo alcance y poco consumo. Las redes LPWAN más importantes son LoRa y Sigfox.

3 LoRa

Se ha vuelto terriblemente obvio que nuestra tecnología ha superado nuestra humanidad.

- Albert Einstein-

Como ya mencionamos anteriormente, una de las tecnologías para implementar redes IoT de largo alcance es "LoRa" (Long range), una red LPWAN (Low-power wide area network). Este tipo de red de bajo consumo y área extensa, cuenta con grandes ventajas para conseguir desarrollar comunicaciones inteligentes entre dispositivos lejanos, con muy poco consumo de energía. Esto permite implementar gracias a LoRa, redes IoT con los nodos muy separados entre sí para monitorizar y actuar sobre sistemas alejados una distancia del orden de kilómetros de la puerta de enlace.

Es una opción perfecta si nuestra aplicación requiere de libertad de movimiento y no necesita una transferencia de datos extremadamente rápida.

3.1. Origen

La modulación CSS, de la que hablaremos posteriormente, se usaba durante décadas en comunicaciones militares y espaciales debido a su gran alcance y robustez. LoRa se basó en este tipo de modulación para implementarla en la banda ISM (Industrial, Scientific and Medical) internacional y así crear un método de conexión estándar y global.

LoRa fue el primer uso comercial de bajo coste basado en la modulación de radio-frecuencia CSS. Fue desarrollada por la start-up francesa *Cycleo* en 2008, aunque más tarde sería adquirida y patentada por *Semtech*, una compañía californiana creada en 2015.

Semtech, además de poseer la patente de LoRa, fabrica chips dotados de sistemas de comunicación LoRa y es fundadora de *Lora Alliance*. Esta alianza o asociación de empresas tecnológicas se dedican a desarrollar, coordinar y mantener una red masiva IoT llamada LoRaWAN. Su labor también consiste en dotar de una certificación a los fabricantes de dispositivos capaces de soportar el protocolo LoRaWAN, para garantizar un uso estandarizado y correcto de este protocolo. Destacamos entre los miembros de *LoRa Alliance* a *IBM*, *Cisco*, *Singtel* y *Swisscom,* entre muchos otros.

A día de hoy, existen cerca de 1 millón de dispositivos ya conectados a la red LoRaWAN y cada vez son más los fabricantes de dispositivos IoT que apuestan por esta tecnología.

3.2. LoRa PHY

Se conoce como LoRa PHY a la capa física que implementa este tipo de conexión. Es una tecnología inalámbrica que utiliza la modulación en radio-frecuencia para transmitir información de un nodo a otro. Esta frecuencia es mucho menor que la de WiFi, permitiendo así un alcance mucho más extenso, del orden de kilómetros, aunque sea sacrificando ancho de banda.

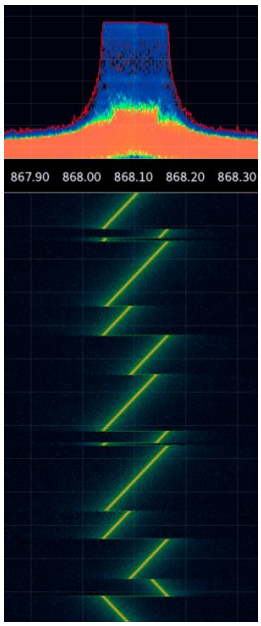


Figura 4: Espectro de un mensaje LoRa

LoRa se basa en un tipo de modulación denominada CSS (*Chirp Spread Spectrum*), donde la información digital es transmitida como variaciones lineales de frecuencia de la señal transmitida. Estas variaciones de frecuencia tienen como valor inicial y final los límites de ancho de banda de cada canal. Como podemos ver en la Figura 4, se aprecia un mensaje LoRa, comprendido en un canal de 125 kHz, entre 868.4 MHz y 868.525 MHz.

Tabla 1: Características de LoRa. Comparativa Europa vs. América

	Europa	**América**
Banda de frecuencia	867-869 MHz	902-928 MHz
Canales	10	72
Ancho de banda del canal	125 kHz	500 kHz
Spreading Factor (SF)	7 - 12	7 - 10
Velocidad de datos	250 bps – 50 kbps	980 bps – 21.9 kbps

El Spreading Factor, previamente citado en la Tabla 1, define la velocidad de emisión del mensaje. Esto es, mientras más lento se emita el mensaje (SF-12), más robusto es y con más probabilidad llegue al nodo destino. Mientras más rápido se emita el mensaje (SF-7), más rápida será a conexión con el nodo receptor.

Es decir, teniendo un ancho de banda de canal fijo, podemos emitir la información más rápido, pero más pequeña será la rampa en frecuencia y por lo tanto más difícil de detectar será. Sin embargo, mientras más alarguemos la rampa del bit enviado, más opciones de captar el bit exitosamente, aunque tardes más tiempo en enviarlo.

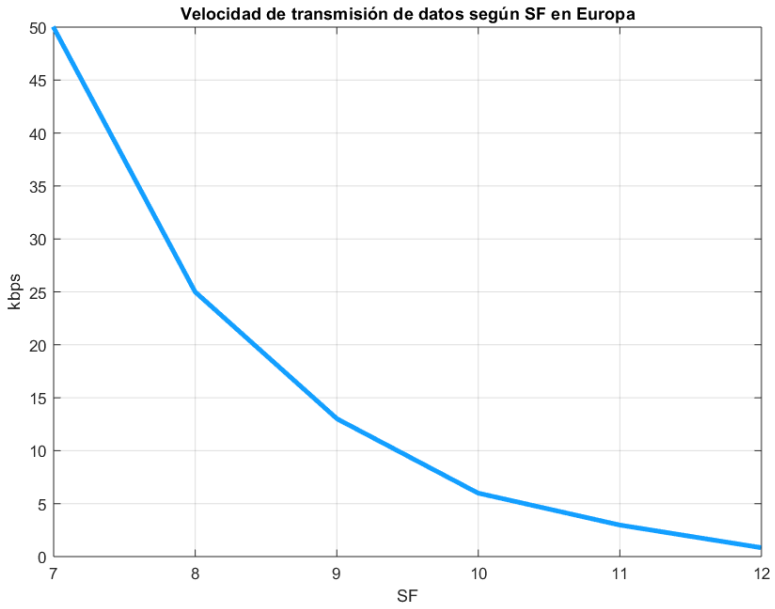


Figura 5: Gráfico de velocidad de transmisión según Spreading Factor

3.3. Estructura de un mensaje LoRa

La estructura de las tramas que viajan por la capa física LoRa PHY, siguen el siguiente esquema:

Figura 6: Esquema de la estructura de una trama LoRa

- Preámbulo: Cumple la función de sincronización y contiene datos sobre la conexión, como el S-F. Su longitud puede variar y se establece al inicio de la transmisión. Suele ser de 12 símbolos, pero LoRaWAN usa 8. La longitud del encabezado del receptor y del emisor debe ser la misma, si es desconocida, se debe configurar al máximo.
- Cabecera: Contiene información sobre la carga útil: longitud del paquete, tasa de codificación y validación del CRC. Para una conexión privada (ajena a LoRaWAN), estos parámetros deben ser configurados manualmente. Al final, contiene un CRC (Cyclic Redundancy Check) para la corrección de errores. Ambos ocupan una longitud de 20 bits.
- Datos: Carga útil enviada. En LoRaWAN, existen 2 capas más entramadas en esta sección. La capa MAC, donde se encapsula la carga útil en un mensaje con información de identificación del dispositivo. Finalmente, una capa de aplicación con información de la dirección, información de control y el puerto de dicha aplicación.
- CRC: Finalmente, 2 bytes de CRC para la corrección de errores en la recepción de la trama.

3.4. LoRaWAN

Tal y como se mencionó previamente, la *Lora Alliance* se encarga de mantener y dar soporte a los distintos fabricantes para mantener una red mundial utilizando la tecnología de comunicación LoRa. Su objetivo es promover y desplegar una red IoT global. Por ello, esta asociación se encarga de certificar el correcto uso y funcionamiento de los diferentes dispositivos que emplean esta tecnología. Para ello, se creó LoRaWAN, un estándar que define los protocolos de esta red masiva.

Las principales características de dicho estándar son las siguientes:

- Conexiones seguras y bidireccionales
- Largo alcance
- Bajo consume de energía
- Baja velocidad de datos
- Baja frecuencia de transmisión
- Servicios de localización

La red trabaja a unas velocidades de transmisión entre 0.3 kbps y 50 kbps con el fin de mantener el bajo consumo de los dispositivos y asegurar la capacidad de transmisión de datos de la red. Es un servidor central de LoRaWAN el que se encarga de asignar la velocidad de transmisión de datos de cada dispositivo de manera individual, mediante un sistema ADR (Adaptative Data Rate), el cual asigna dinámicamente dicha velocidad según el esquema de nodos conectados a la red. La frecuencia y velocidad de transmisión de datos asignada a los canales entre los dispositivos finales y las puertas de enlace viene condicionada por el coste del mensaje, en función de la distancia, duración de la transmisión

La arquitectura de LoRaWAN sigue una topología en forma de estrella. Los nodos finales, encargados de transmitir información se conectan a unas puertas de enlace normalmente conectadas directamente a internet, para tener los datos a disposición de la capa de aplicación del usuario final.

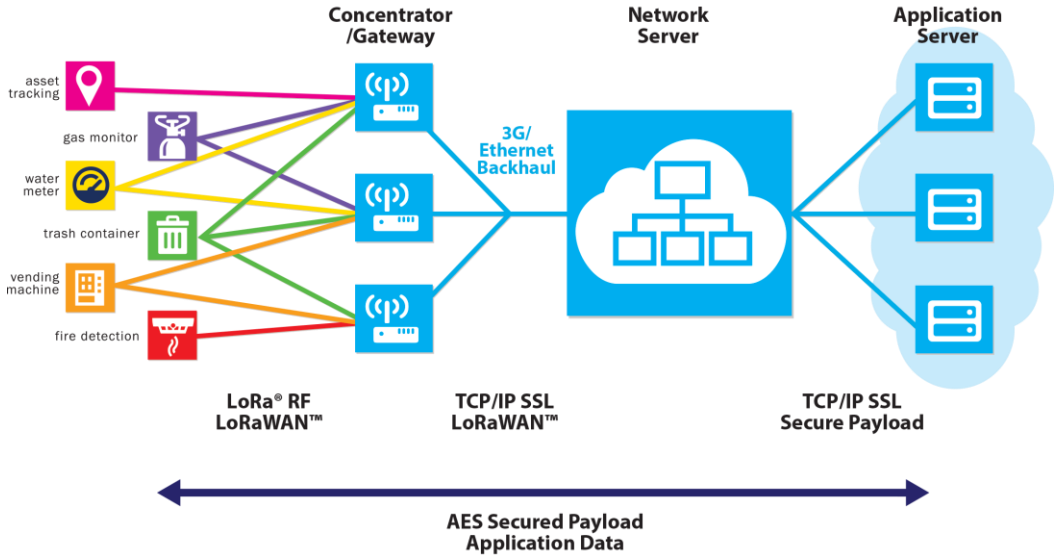


Figura 7: Esquema de conjunto de red IoT

A parte de encriptar los mensajes para hacer más segura la red, también se encarga de filtrar mensajes duplicados, corregir errores de transmisión, actualizar el software de los dispositivos mediante canales de multidifusión para lograr una robustez y fiabilidad máxima de una red IoT mundial.

En definitiva, LoRa es la capa física que da soporte al protocolo de comunicación LoRaWAN, que dirige y gestiona las conexiones entre distintos nodos finales y puertas de enlace para poner en manos de un servidor central la información para así poder ser recatada por el usuario final.

3.5. LoRaWAN vs. Sigfox

Ambas tecnologías persiguen el mismo fin, una red LPWAN robusta y centralizada para dar lugar a aplicaciones IoT con la máxima cobertura posible. Dentro de este tipo de redes de largo alcance y bajo consumo, existen muchas tecnologías como pueden ser Ingenu RPMA, LTE CAT-M, NB-IoT, EC-GSM... Sin embargo, LoRaWAN y Sigfox son las tecnologías más usadas.

Tal y como hemos visto previamente, LoRaWAN es un estándar administrado por Lora Alliance cuyo objetivo es certificar el uso de la tecnología LoRa en los diferentes fabricantes de dispositivos IoT. Sin embargo, Sigfox es una compañía privada cuyo propósito es ser un proveedor global de redes IoT.

Sigfox, tiene origen en Toulouse, Francia gracias a la necesidad de conectar smartwaches que enviaban pequeñas cantidades de datos cada cierto tiempo. Esto sucedió en 2010, y a día de hoy posee cobertura en más de 1.3 millones de km^2 en 22 países, hacienda uso de la banda ISM (Industrial, Scientific and Medical Band) en las mismas frecuencias que LoRa (868MHz en Europa y 902MHz en Estados Unidos). La compañía Sigfox gestiona y administra completamente la comunicación entre el nodo terminal y la puerta de enlace, por lo que es muy sencilla su instalación por parte del usuario. Este protocolo utiliza una única API lo que hace que tenga muy poca configuración y es aún más sencillo de usar. Para poder usar Sigfox, hay que pagar una suscripción.

LoRaWAN, sin embargo, tiene como objetivo definir y certificar el estándar en los distintos fabricantes para que puedan utilizarla, no es un proveedor de red como Sigfox. Esto permite poder crear redes privadas adaptadas a nuestras necesidades de comunicación, o contratar proveedores de LoRaWAN para poder conectarse, como por ejemplo Orange en España, donde se encarga del mantenimiento de la red.

Diferencias técnicas:

Tabla 2: Comparativa entre LoRaWAN y Sigfox

	LoRaWAN	**Sigfox**
Banda de frecuencia	433/868/780/915 MHz en ISM	868/902 MHz en ISM
Alcance Urbano	2-5 km	3-10 km
Alcance Rural	15-20 km	30-50 km
Capacidad del paquete	Configurable	12 bytes
Límite de transmisión	~ 20 mensajes/hora	140 mensajes/día (7 max por hora)
API	Compleja y configurable	Simple y estándar
Comunicación	Estrella Bidireccional	Estrella Unidireccional

Por ello, para una aplicación compleja, en la que se quiera actuar sobre un determinado sistema, por la bidireccionalidad y la configuración de la API, LoRa es la solución adecuada. Por otro lado, si queremos sencillez, comodidad y alcance, son factores en los que Sigfox es superior.

4 PLACA DE DESARROLLO LOPY4

Puedes centrarte en las barreras o bien en escalar el muro y redefinir el problema.

- Tim Cook -

Este es el dispositivo que trataremos de analizar en este proyecto, con el fin de evaluar su eficacia a la hora de implementar pequeñas aplicaciones con LoRa y el resto de sus funcionalidades. En definitiva, vamos a tratar de usar el LoPy4 para crear una pequeña red IoT, utilizando sus posibilidades de comunicación como son LoRa, Sigfox, BLE y WiFi.

Adicionalmente, se analizará generalmente las principales funcionalidades y características de esta potente placa de desarrollo de 5x2x3 cm.

Figura 8: Chip LoPy4

Figura 9: LoPy4 junto con Expansion board 3.0 y antena conectada

4.1. Pycom

Una start-up procedente de Eindhoven, en los Países Bajos, es la creadora de estas placas de desarrollo. Su nombre es Pycom, y ya tiene conectados a redes IoT más de 230.000 dispositivos. Su principal objetivo es producir placas de desarrollo, fáciles de programar y conectar, a través de las distintas capas físicas que soportan sus productos, múltiples nodos de una forma sencilla, rápida y eficaz.

Sin duda, la labor de Pycom hace que sea mucho más fácil llegar a crear redes IoT, con un simple dispositivo de 5 cm de tamaño podemos transmitir y recibir la información que necesitemos para conseguir nuestros objetivos de una manera más eficaz.

Uno de sus productos, y en el que nos vamos a centrar, es la placa de desarrollo LoPy4 que fue lanzada al mercado en 2018. Es sucesora de la antigua placa de desarrollo LoPy (2016), que ya contaba con conexión Bluetooth, WiFi y LoRa, pero no de Sigfox.

4.2. Características de LoPy4

La placa de desarrollo LoPy4, es uno de los dispositivos que ofrece Pycom para conectar los distintos dispositivos en una red inteligente.

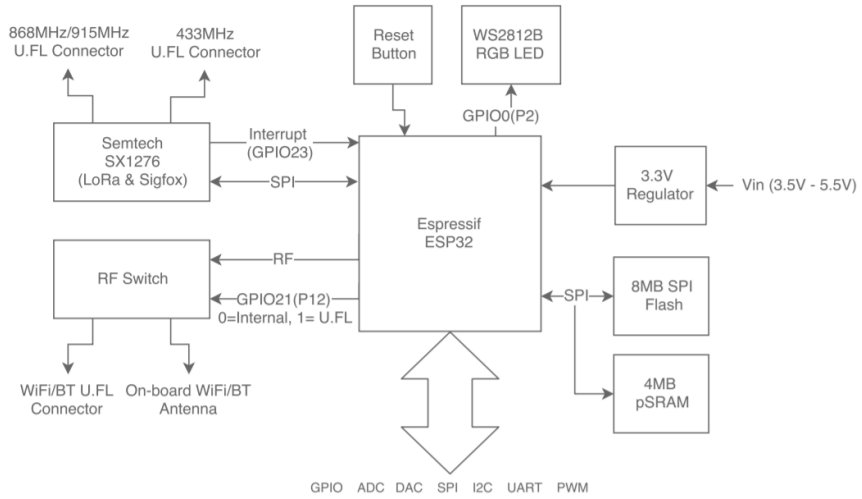


Figura 10: Esquema de los distintos módulos del LoPy4

Como se puede apreciar en la Figura 10, el LoPy cuatro posee diferentes módulos o periféricos capaces de llevar a cabo diferentes funciones, dotando así a esta placa de desarrollo de múltiples opciones a la hora de dar soluciones a aplicaciones IoT. Vamos a ir comentando los diferentes módulos del LoPy4:

4.1.1 CPU

Tal y como se muestra en la Figura 10, el núcleo del LoPy4 es su microcontrolador ESP32, creado y desarrollado por la empresa asiática Espressif Systems. Este microcontrolador low-cost y low-power, posee un microprocesador LX6 Xtensa® dual–core de 32 bits, de hasta 600 DMIPS.

Compatible con sistemas multihilos Python, aceleración por cálculo en punto flotante y un coprocesador que puede manejar el GPIO, el ADC y la mayoría de periféricos internos durante el modo “deep sleep” del LoPy4, donde consume ~ 0.1W.

4.1.2 Memoria

La placa LoPy4 posee una memoria RAM de 4 MB + 520 kB del modo “deep sleep”. A parte, existe una memoria flash interna de 8 MB, ampliables con la placa de adaptación *Expansion board 3.0* la cual posee una ranura para tarjetas microSD de hasta 32GB.

4.1.3 Periféricos del ESP32

LoPy4 es un microcontrolador muy competitivo en el mercado debido a lo completo que es en lo que a periféricos se refiere. Gracias a ellos, con LoPy4 se pueden controlar una inmensa cantidad de aplicaciones, ya que posee muchos pins de entrada/salida, convertidores, comunicaciones inalámbricas y puerto serie, interrupciones... Los principales periféricos disponibles son los siguientes:

4.1.3.1 GPIO

El módulo GPIO (General Purpose Input/Output) controla los 23 pins disponibles de uso general que dispone la placa de desarrollo LoPy4. Gracias a ellos podemos usar nuestros sensores y actuadores, conectados a sus correspondientes pins detallados en los siguientes apartados. Estos pins pueden ser configurados tanto de entrada como salida, tomando valores digitales o analógicos.

Tabla 3: Funcionalidad de los pins del LoPy4

Numero	Función por defecto	I/O	PWM	ADC	DAC
P0	RX0 (para programarla)	I	x	x	x
P1	TX0 (para programarla)	O	✓	x	x
P2	RGB Led	O	✓	x	x
P3	TX1	O	✓	x	x
P4	RX1	I	✓	x	x
P8	SD card DAT0	I/O	✓	x	x
P9	SDA (I2C) o LED (exp. board)	I/O	✓	x	x
P10	SCL (I2C) o CLK (SPI) o Botón (exp. board)	I/O	✓	x	x
P11	MOSI (SPI)	I/O	✓	x	x
P12	Antena WiFi/BLE	I/O	✓	x	x
P13		I	x	✓	x
P14	MISO (SPI)	I	x	✓	x
P15		I	x	✓	x
P16		I	x	✓	x
P17		I	x	✓	x
P18		I	x	✓	x
P19		I/O	✓	✓	x
P20		I/O	✓	✓	x
P21	DAC	I/O	✓	x	✓
P22	DAC	I/O	✓	x	✓
P23	SD card SCLK	I/O	✓	x	x

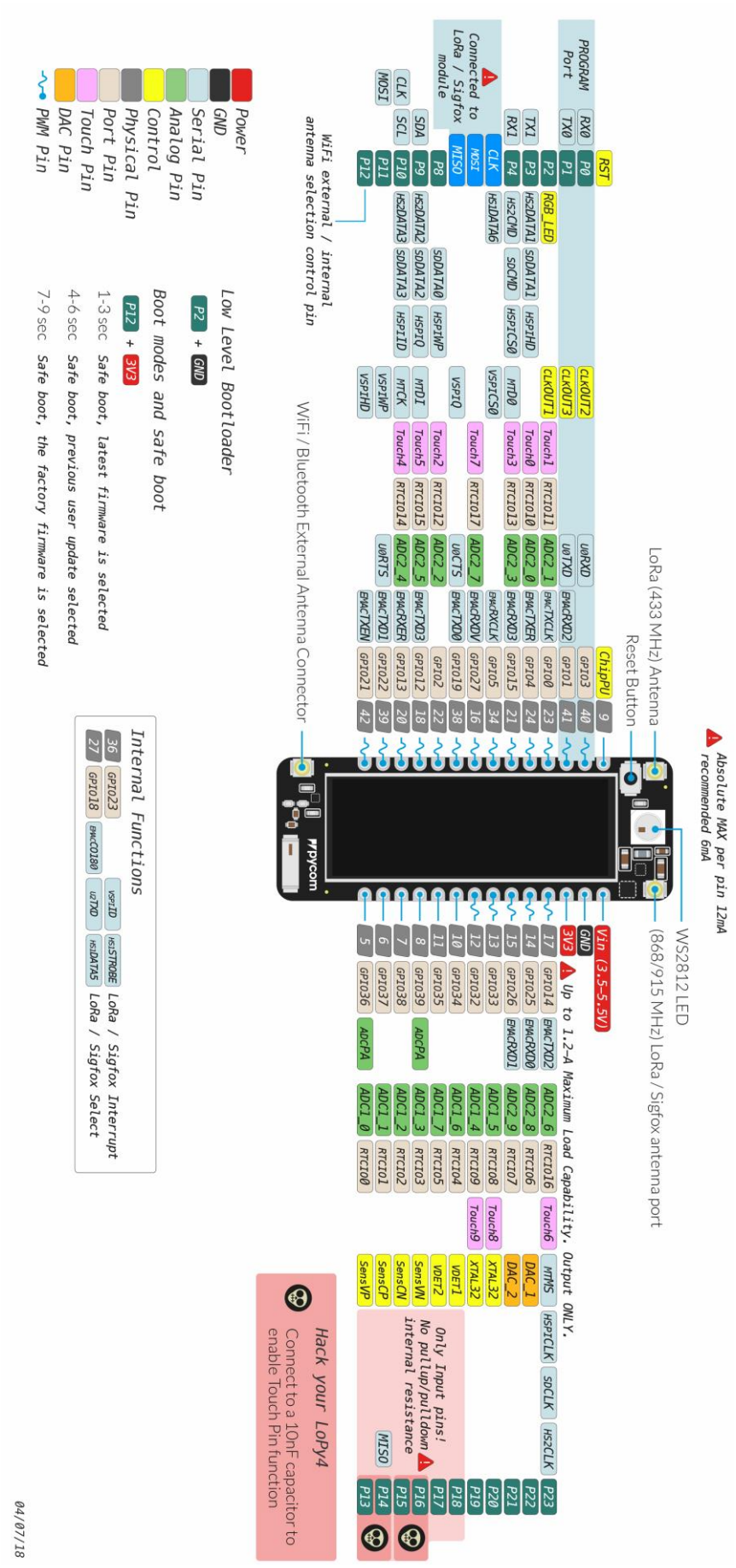


Figura 11: Mapa de pins del LoPy4

4.1.3.2 PWM

Existen 4 temporizadores PWM distintos para configurar cada uno a una frecuencia determinada entre 1 Hz y 78 kHz. Cada uno de estos temporizadores tiene 8 canales distintos para establecer diferentes duty cycles asignados a diferentes pines, usando un mismo temporizador.

Prácticamente todos los pines están habilitados para funcionar como PWM.

4.1.3.3 ADC / DAC

Los pins válidos para usar el conversor analógico – digital (ADC) van desde el P13 hasta el P20, como se puede apreciar en la Tabla 3. Este se puede configurar con una resolución entre 9 y 12 bits.

El rango de valores analógicos que el ADC es capaz de soportar, está comprendido entre 0 y 1.1 V máximo. Este valor puede aumentar hasta 3.3 V si se establece una atenuación de 11 dB en el canal del conversor, pero se debe tener en cuenta que voltajes superiores podrían dañar el dispositivo.

Por otro lado, el conversor digital – analógico (DAC) solo se puede usar en los pins P21 y P22. Este módulo es capaz de generar tanto valores constantes como senoidales de frecuencias entre 125 Hz y 20 kHz y valores de amplitud de 3.3 V máximos de pico.

4.1.3.4 SPI / I2C / UART

El LoPy4 soporta estos 3 estándares de comunicación física en serie:

- SPI: se configura en los pins P10 (CLK), P11 (MOSI) y P14 (MISO).
- I2C: protocolo de 2 cables físicos en los pins P10 (SDA) y P11 (SCL).
- UART: generalmente, habilitada en los pins P0 (RX) y P1 (TX) aunque pueden ser configurables para otros pines.

4.1.4 Conectividad

El dispositivo LoPy4 posee 4 módulos de comunicación inalámbrica:

4.1.4.1 WiFi

Dispone de un módulo WiFi 802.11 b/g/n/e/i de 2.4 GHz con capacidad de hasta 150 Mbps. Gracias a esto, se puede programar el dispositivo como una puerta de enlace entre LoRa e internet.

4.1.4.2 LoRa

Posee el microchip *Semtech SX1276* que cumple con las siguientes características:

- Rango de frecuencias: 137 – 1020 MHz.
- SF (Spreading Factor): 6 – 12.
- Ancho de banda del canal: 7.8 – 500 kHz.
- Velocidad de transmisión efectiva: 0.018 – 37.5 kbps.
- Sensibilidad: -111 a -148 dBm.

4.1.4.3 Sigfox

Tiene certificación en las regiones de Europa (RCZ1), Estados Unidos (RCZ2), Sudamérica, Australia y Nueva Zelanda (RCZ3), y está a la pendiente las de Corea y Japón (RCZ4). El módulo posee las siguientes características según la zona:

Tabla 4: Características de Sigfox según región

Región	RCZ1	RCZ2	RCZ3	RCZ4
Velocidad de transmisión	100 bps	600 bps	100 bps	600 bps
Sensibilidad de transmisión	+14 dBm	+20 dBm	+14 dBm	+20 dBm
Sensibilidad de recepción	-126 dBm	-126 dBm	-126 dBm	-126 dBm
Frecuencia de subida	86813 kHz	902200 kHz	920800 kHz	923200 kHz
Frecuencia de bajada	869525 kHz	905200 kHz	922300 kHz	922200 kHz

4.1.4.4 BLE

Por último, contiene un módulo Bluetooth con soporte para las especificaciones de BLE (Bluetooth Low Energy). Esto es perfecto para aplicaciones de corto alcance, ya que se puede establecer una conexión de muy bajo consumo de una manera muy sencilla.

4.1.5 Consumo

El LoPy4 puede alimentarse con una Fuente de 3.3V – 5.5V, aunque internamente regula este voltaje a 3.3V. Se puede aproximar según la tarea que esté llevando a cabo:

Tabla 5: Consumo aproximado del LoPy4 según aplicación

Tarea	Corriente	Consumo
Normal	30 mA	99 mW
Trasmisión LoRa	105 mA	346.5 mW
Transmisión Sigfox	60 mA	198 mW
WiFi	100 mA	330 mW
BLE	94 mA	310 mW
Deep Sleep	19.5 mA	64.35 mW

4.1.6 Programación

La programación del dispositivo LoPy4, se puede llevar a cabo de diferentes maneras.

- UART: el módulo ejecuta un lenguaje REPL interactivo, en el los puertos de la UART0, configurada en 115200 baudios. Con un plugin para Atom o Visual Studio y gracias a la *expansion board 3.0* puedes subir fácilmente el código al dispositivo. Dicha placa de expansión, tiene pins accesibles para conectar fácilmente periféricos, lector de tarjeta microSD, varios LEDs y puerto microUSB para conectar la UART al PC como puerto serie.
- WiFi: a través de una aplicación para smartphones puedes acceder al firmware de Pybytes instalado en el LoPy4 y así configurar la OTA del WiFi. Una vez conectados al WiFi del módulo, se puede subir de dos maneras:
    - Telnet: en el puerto 23 se ejecuta un servidor Telnet, que funciona igual que la UART.
    - FTP: el módulo también ejecuta un servidor FTP por el que se puede copiar desde y al dispositivo, en una tarjeta SD que debe tener conectada. Para conectarte necesitas usar un protocolo sin encriptar (FTP puro) con las siguientes credenciales: Usuario: “micro”; Contraseña: “python”.

5 PROGRAMACIÓN DEL LOPY4

La mayoría del software actual es muy parecido a una pirámide egipcia, con millones de ladrillos puestos unos encima de otros sin una estructura integral, simplemente realizada a base de fuerza bruta y miles de esclavos

- Alan Kay -

La programación del microcontrolador LoPy4 es muy sencilla gracias a las múltiples librerías que están disponibles en nuestro dispositivo. Es muy fácil conectar la placa de desarrollo al PC, programarla, cargarle el archivo y ya estaría configurada para empezar a usarla en nuestra aplicación.

El lenguaje que reconoce el dispositivo es MicroPython, una adaptación de Python para microcontroladores. Este cuenta con la mayoría de funcionalidades de alto rendimiento de Python, con la capacidad de ejecutarlas con solo 16kB de memoria RAM.

A continuación, se va a detallar cómo conectar y empezar a programar el LoPy4, así como las principales funciones necesarias para empezar a desarrollar aplicaciones IoT.

5.1 Configuración del entorno de desarrollo

Para empezar a programar el LoPy4 vía USB, es necesario conectar la placa al PC mediante un cable microUSB-USB.

Tenemos que elegir un entorno de desarrollo. Si queremos correr el código y acceder a la consola de la placa de desarrollo, necesitamos utilizar una de las 2 únicas soluciones posibles:

- El plugin “Pymakr” instalado en Atom.
- La extensión “Pymakr” instalada en Visual Studio Code.

En este caso, se ha usado la versión para Visual Studio Code para Windows.

Una vez instalado, debemos cambiar los ajustes del proyecto, para seleccionar el puerto serie por el que el microcontrolador ha sido conectado con el PC. Para ello hay que seguir los siguientes pasos:

1. Abrir el Administrador de dispositivos de Windows, y buscar nuestro dispositivo en los puertos serie:

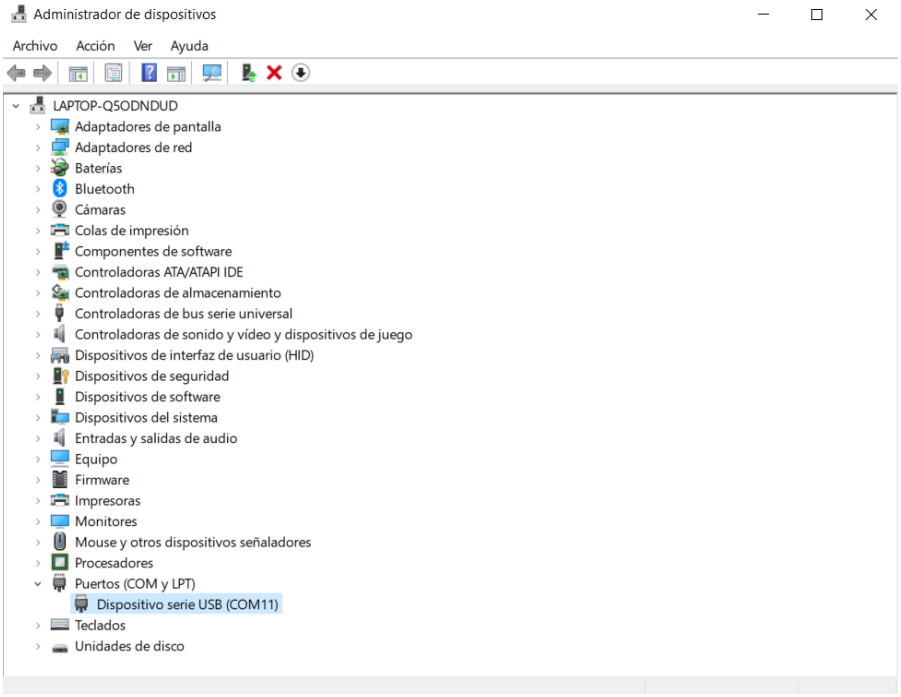


Figura 12: Puerto Serie del LoPy4, en el Administrador de Dispositivos

2. Una vez instalado la extensión para Visual Studio Code, en la parte inferior, hacer click en “All commands” y ejecutar “Global Settings”.

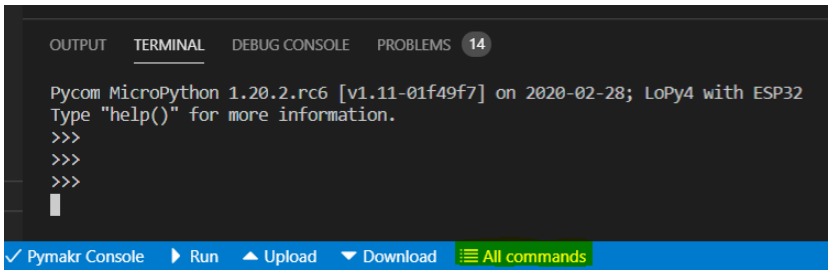


Figura 13: Botón "All Commands"

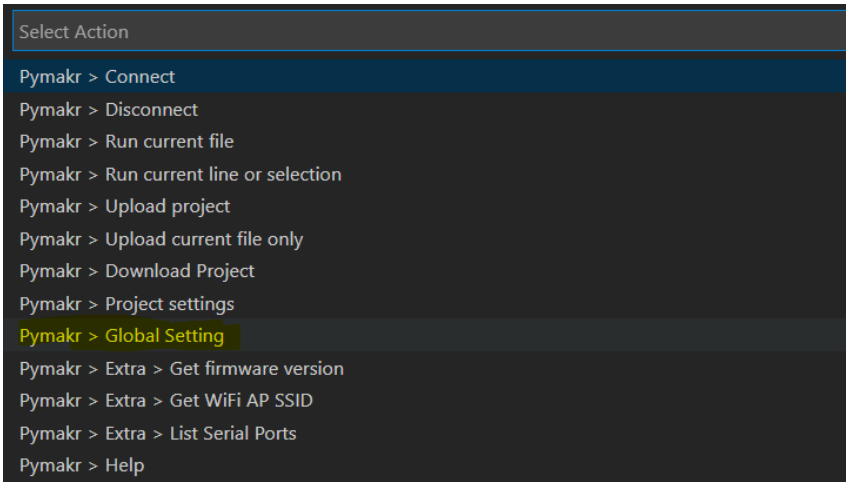


Figura 14: Pop-up de comandos y comando Global Setting

3. Cambiar la configuración de dirección del puerto al correcto. En este caso:

```
"address": "COM11"
```

4. Guardar el archivo y reiniciar la conexión.

Se puede probar el correcto funcionamiento de la conexión con el LoPy4 ejecutando las siguientes líneas desde la consola de comandos:

```
>> import pycom
>> pycom.heartbeat(False)
>> pycom.RGBLED(0xFF0000)
```

Si el LED RGB ha dejado de parpadear y se ha puesto de color rojo, todo ha ido bien y podemos empezar a programar nuestra placa de desarrollo.

5.2 Programación por módulos

5.2.1 Pycom

Para empezar, vamos a ver algunas funciones de la librería *pycom.* Estas son muy sencillas y adecuadas para iniciarse programando la placa de Desarrollo LoPy4. Para importar la librería, se escribe la siguiente sentencia:

```
import pycom
```

Algunas de las funciones más importantes de dicha librería son:

- Habilitar/Deshabilitar el parpadeo del LED RGB.

```
pycom.heartbeat(True/False)
```

- Cambiar color del LED RGB.

```
pycom.rgbled(color)
```

Parámetro	Comentario
color	Color que se quiera establecer en hexadecimal. (Ej.: Rojo, 0xFF0000)

- Leer el número de pulsos por tiempo en un pin. Devuelve una lista con la duración de los pulsos leídos.

```
pycom.pulses_get(pin, timeout)
```

Parámetro	Comentario
pin	Pin que se desee leer. Este debe haber sido previamente declarado como IN o DRAIN. Véase programación del módulo GPIO.
timeout	Tiempo en milisegundos que debe transcurrir sin cambios para finalizar la lectura.

- Activar/Desactivar parpadeo en el arranque.

```
pycom.heartbeat_on_boot([boolean])
```

- Activar/Desactivar WiFi en el arranque.

```
pycom.wifi_on_boot([boolean])
```

- Establecer modo WiFi en el arranque.

```
pycom.wifi_mode_on_boot(mode)
```

Parámetro	Comentario
mode	Modo de uso. Puede tomar los siguientes valores: • *WLAN.STA* para poder conectar el dispositivo a una red WiFi. • *WLAN.AP* para poder crear una red WiFi desde el dispositivo. Si se establece esta opción, es obligatorio configurar un **SSID**. • *WLAN.APSTA* para poder conectarte y crear a una red WiFi, ambas.

- Establecer SSID de conexión al arranque.

```
pycom.wifi_ssid_sta([ssid]) #Para buscar un punto de acceso WiFi con ese SSID.
pycom.wifi_ssid_ap([ssid]) #Para crear un punto de acceso WiFi con ese SSID.
```

- Establecer contraseña de conexión al arranque.

```
pycom.wifi_pwd_sta([key]) #Contraseña del punto de acceso buscado
pycom.wifi_pwd_ap([key]) #Contraseña del punto de acceso creado
```

5.2.2 GPIO

A continuación, veremos las principales funciones de MicroPython que programan el módulo GPIO de nuestra placa de desarrollo. Todas estas funciones se alojan en la clase *Pin* de la librería *machine*, que debe ser importada inicialmente de la siguiente manera:

```
from machine import Pin
```

Las funciones para programar los pins para el uso del GPIO son las siguientes:

- Crear una instancia de pin de entrada/salida.

```
pin = Pin(id, mode, pull, *)
```

Parámetro	Comentario
id	Número del pin a configurar. Entre el P0 y P23. También es posible referencial a pins de la expansión board, de la manera *Pin.exp_board.GXX* donde XX corresponde al número del pin.
mode	Modo en el que se quiere configurar el pin. Puede ser: • *Pin.IN* para usarlo de entrada. • *Pin.OUT* para usarlo de salida. • *Pin.OPEN_DRAIN* para usarlo de entrada y salida.
pull	Puede ser: • *None* • *Pin.PULL_UP* para obtener una resistencia pull-up (HIGH en reposo) • *Pin.PULL_DOWN* o pull-down (LOW en reposo)
*	Establece el valor inicial del pin. 1 para HIGH o 0 para LOW.

- Leer el valor simultáneo del pin.

```
pin()
```

- Establecer un valor digital al pin (1 / 0 o True / False).

```
pin.value(val)
```

- Leer el ID del pin.

```
pin.id()
```

- Cambiar el valor del pin al opuesto.

```
pin.toogle()
```

- Mantener el valor del pin hasta que se desactive. Útil para guardar el valor en un reinicio del sistema.

```
pin.hold(hold)
```

Parámetro	Comentario
hold	Habilita guardar el valor del pin. Puede ser *True* o *False.* Solo se puede establecer en los pins: P2-P4, P6, P8-P10, P13-P23.

- Habilitar función de callback.

```
pin.callback(trigger, [handler, arg])
```

Parámetro	Comentario
trigger	Evento que dispara la llamada de la función. Pueden ser los siguientes: • *Pin.IRQ_FALLING* cuando pase de nivel alto a nivel bajo. • *Pin.IRQ_RISING* cuando pase de nivel bajo a nivel alto. • *Pin.IRQ_LOW_LEVEL* cuando esté en nivel bajo. • *Pin.IRQ_HIGH_LEVEL* cuando esté en nivel alto.
handler	Nombre de la función a llamar cuando se produzca el evento.
arg	Argumento que se le pasa a la función.

5.2.3 ADC Y DAC

5.2.3.1 Conversor Analógico/Digital

El conversor analógico - digital que incluye el dispositivo tiene una resolución de 9 a 12 bits y se le puede ajustar la atenuación en 0dB, 2.5dB, 6dB y 11dB. Todas las funciones se encuentran en la clase *ADC* de la librería *machine*.

```
from mahine import ADC
```

Las funciones que se usan para programar un conversor Analógico/Digital son:

- Crear una instancia de ADC.

```
adc = ADC()
```

- Habilitar el módulo del conversor analógico digital. Se le pasa como parámetros el número de bits de resolución (9 – 12).

```
adc.init([bits=9])
```

- Deshabilitar el módulo.

```
adc.deinit([bits=9])
```

- Crear un canal del conversor analógico digital en el pin seleccionado.

```
adc_c = adc.channel(pin='P16', attn=ADC.ATTN_0DB)
```

Parámetro	Comentario
pin	Cualquiera de los pins válidos para el ADC. Ver tabla
attn	Nivel de atenuación. Este parámetro puede adoptar las siguientes constantes: • *ADC.ATTN_0DB* para 0dB • *ADC.ATTN_2_5DB* para 2.5dB • *ADC.ATTN_6DB* para 6dB • *ADC.ATTN_11DB* para 11dB

- Leer el valor del canal.

```
val = adc_c()
val = adc_c.value()
```

- Devuelve el valor del canal convertido en mili Voltios.

```
volts = adc_c.voltage()
```

5.2.3.2 Conversor Digital/Analógico

La conversión de digital a analógico es igual de simple, acompañado de las funciones de la clase *DAC* de la librería *machine*, previamente mencionada. Se debe de tener en cuenta que la salida estará comprendida entre 0 y 3.3V y que solo se podrán usar los pins P21 y P22.

Las funciones para manejar el DAC son:

- Crear una instancia del conversor digital analógico.

```
dac = machine.DAC(pin)
```

Parámetro	Comentario
pin	Cualquiera de los pins válidos para el ADC. Pueden ser P21 o P22.

- Habilitar el módulo DAC.

```
dac.init()
```

- Deshabilitar el módulo DAC

```
dac.deinit()
```

- Establecer un valor analógico.

```
dac.write(valor)
```

Parámetro	Comentario
valor	Valor es un numero comprendido entre 0 y 1. Siendo 1, 3.3V.

- Establecer una señal sinusoidal con los valores de amplitud y frecuencia determinados y un offset de $V_{DD}/2$.

```
dac.tone(frecuencia, amplitud)
```

Parámetro	Comentario
frecuencia	Numero entero comprendido entre 122 Hz y 20000 Hz. Solo en saltos de 122 Hz.
amplitud	Entero que indica la amplitud de la señal sinusoidal, acorde a la siguiente tabla:

Amplitud	Voltaje Pico a Pico
0	~3 Vpp
1	~1.5 Vpp
2	~0.8 Vpp
3	~0.4 Vpp

5.2.4 PWM

La programación de un pin con modulación por ancho de pulso o PWM (Pulse Width Modulation), es muy sencilla. Los distintos métodos se encuentran en la clase *PWM* de la librería *machine*.

```
from machine import PWM
```

Las funciones que se usan para establecer PWMs son las siguientes:

- Crea un temporizador PWM

```
pwm = PWM(id, frequency=500)
```

Parámetro	Comentario
id	Identificador de la instancia de temporizador PWM. Es un entero de valor entre 0 y 3.
frequency	Frecuencia del temporizador del PWM. Es un entero entre 1 (1Hz) y 78000 (78kHz).

- Crea un canal para el temporizador seleccionado y lo asigna a un pin.

```
pwm_c = pwm.channel(id, pin='P12', duty_cycle=0.5)
```

Parámetro	Comentario
id	Identificador de la instancia de temporizador PWM. Es un entero de valor entre 0 y 3.
pin	Cualquiera de los pins válidos para el PWM. Ver Tabla 3.
duty_cycle	Valor del Duty cycle. Es un número decimal entre 0 y 1

- Cambia el valor del duty cycle. Como argumento se le pasa el valor del duty cycle (0 – 1).

```
pwm_c.duty_cycle(dc)
```

5.2.5 Timers

Programar los temporizadores del Lopy4 es extremadamente sencillo y se pueden crear tantas instancias de temporizador como se desee. La clase *Timer* se define en la librería *machine*:

```
from machine import Timer
```

Existen 3 vertientes definidas según el objetivo de estos temporizadores:

5.2.5.1 Temporizador como cronómetro

Sirve para medir el paso del tiempo. Las funciones que controlan el uso del cronómetro son:

- Crear instancia de temporizador como cronómetro.

```
crono = Timer.Chrono()
```

- Dispara el cronómetro.

```
crono.start()
```

- Lee el valor actual del cronómetro.

```
valor = crono.read()
```

- Lee el valor actual del cronómetro en milisegundos.

```
valor_ms = crono.read_ms()
```

- Lee el valor actual del cronómetro en microsegundos.

```
valor_us = crono.read_us()
```

- Para el cronómetro.

```
crono.stop()
```

- Reinicia el cronómetro a 0.

```
crono.reset()
```

5.2.5.2 Temporizador para dormir el dispositivo

- Para crear un retardo en el programa inferior a 10 000 µs (10 ms), se utiliza la siguiente función.

```
Timer.sleep_ms(valor)
```

Donde ***valor*** es el número de microsegundos a pausar.

Para retardos superiores a 10 ms, no se garantiza la completa exactitud del temporizador ya que otros subprocesos se están ejecutando durante su llamada. Para ello, usamos la función definida en la librería *time*, que abordaremos en otro subapartado.

5.2.5.3 Temporizador para crear una interrupción

Sirve para definir una acción a realizar una vez transcurrido cierto tiempo. El sistema de interrupciones del Lopy4 permite encolar como máximo 16 de ellas. Las funciones que definen el proceso son las siguientes:

- Crear una instancia de interrupción por alarma

```
Timer.Alarm([handler=funcion, {s, ms, us}, arg=argumento, periodic=Fase])
```

Parámetro	Comentario
handler	Función que se quiera ejecutar cuando se dispare la alarma.
{s, ms, us}	Vector de tiempo que puede ser establecido tanto en segundos (flotante), como en milisegundos (entero) o microsegundos (entero). Solo se puede establecer uno de los 3 valores.
arg	Argumento que es pasado a la función. Si se especifica *None*, recibe el objeto que disparó la alarma.
periodic	Si se establece en True, la alarma se dispara periódicamente en el tiempo.

5.2.6 SD

Para poder utilizar el módulo SD, se ha de tener en cuenta que solo reconoce tarjetas microSD con capacidad inferior a 32 GB, siempre en formato FAT16 o FAT32. Si se desea usar el módulo SD sin la placa expansión board, los pines de conexión son:

- Pin *P8* para transmisión de datos.
- Pin *P23* para el reloj del sistema.
- Pin *P4* para la CMD

Para manejar archivos, ya sea para editar o leer datos de ellos, mediante el uso de una memoria extraíble tipo Micro-SD, debemos utilizar 2 librerías. Se debe importar la clase *SD* de la librería *machine* y la librería *os*, que contiene los métodos para explorar los archivos del sistema operativo.

```
from machine import SD
import os
```

Los principales métodos existentes para el manejo de archivos en tarjeta microSD son:

- Crear una instancia del módulo microSD.

```
sd = SD()
```

- Montar los archivos de la tarjeta microSD en el directorio */sd.*

```
os.mount(sd, '/sd')
```

- Leer el contenido de un directorio.

```
os.listdir('/sd')
```

- Cargar un archivo en una variable.

```
archivo = open(dir, opcion)
```

Parámetros	Comentario
dir	Dirección del archivo a cargar. Ejemplo: *archivo=open('/sd/texto.txt','r')*
opcion	Forma de cargar el archivo: • 'w' para escritura • '*r'* para lectura

- Escribir texto en un archivo.

```
f.write('texto a escribir')
```

- Mostrar el contenido de un archivo.

```
f.readall()
```

- Guardar y cerrar un archivo

```
f.close()
```

5.2.7 WIFI

El módulo WIFI que contiene LoPy4 puede ser programado de distintas maneras con infinidad de aplicaciones. Se van a estudiar sin entrar en mucho detalle 2 de ellas, como son la conexión WLAN (Wireless Local Area Network) y el protocolo de comunicación COAP (Constrained Application Protocol)

5.2.7.1 WLAN

Los métodos que se usan para configurar las conexiones WiFi se encuentran en la clase *WLAN* de la librería *network*.

```
from network import WLAN
```

No se van a detallar la mayor parte de métodos, sin embargo, veremos los más importantes para establecer una conexión. Estas son las siguientes:

- Iniciar el módulo WiFi y ajustar los parámetros de WLAN.

```
  wlan = WLAN(mode, [ssid=None, auth=None, channel=1, antenna=WLAN.INT_ANT,
power_save=False, hidden=False, bandwidth=WLAN.HT40, max_tx_pwr, country=NA,
protocol=(1,1,1)])
```

Parámetros	Comentario
mode	Modo de uso. Puede tomar los siguientes valores: • *WLAN.STA* para poder conectar el dispositivo a una red WiFi. • *WLAN.AP* para poder crear una red WiFi desde el dispositivo. Si se establece esta opción, es obligatorio configurar un **SSID**. • *WLAN.STA_AP* para poder conectarte y crear a una red WiFi, ambas.
ssid	Nombre (SSID) de la red WiFi creada.
auth	Debe pasarse 2 parámetros, (**sec**, **key**). También se puede establecer como None para no añadir seguridad. El primer parámetro, **sec**, es el tipo de seguridad y puede ser: • *WLAN.WEP* • *WLAN.WPA2* • *WLAN.WPA2_ENT* El segundo parámetro, **key**, es la propia contraseña del punto de acceso.
channel	Solo si estamos en modo *WLAN.AP.* Debe ser un número entre 1 y 11.
antenna	Elegir entre antena interna (*WLAN.INT_ANT*) o la antena externa (*WLAN.EXT_ANT*).
power_save	Solo si estamos en modo *WLAN.STA.* Habilita o deshabilita el ahorro de energía.
hidden	Solo si estamos en modo *WLAN.AP.* Si está en *True*, oculta el SSID.
bandwidth	Habilita o deshabilita la protección MITM de conexiones seguras.
max_tx_power	Máximo valor de potencia de transmisión. Puede tomar valores entre 8 (2dBm) y 78 (20dBm).
country	Cadena de 2 caracteres que representa el país (Ej.: “ES” para España, “FR” para Francia).
protocol	Configura el tipo de protocolo WiFi que soporta. (bool PHY11_B, bool PHY11_G, bool PHY11_N).

- Buscar puntos de acceso WiFi. Devuelve una lista de vectores nombrados con información de los puntos de acceso encontrados.

```
wlan.scan([ssid=None, bssid=None, channel=0, show_hidden=False, type=WLAN.SCAN_ACTIVE, scantime=120ms])
```

Parámetros	Comentario
ssid	Si es establecido, solo busca un SSID en concreto. *None* para buscar cualquiera.
bssid	Si es establecido, solo busca un BSSID en concreto. *None* para buscar cualquiera.
channel	Si es establecido, busca un canal en concreto. Si se establece en 0, busca en todos los canales.
show_hidden	Si es establecido, busca también los SSIDs ocultos.
type	Tipo de búsqueda a realizar. Puede ser: • *WLAN.SCAN_ACTIVE* donde se envía una petición de prueba. • *WLAN.SCAN_PASSIVE* donde cambia a un canal en concreto y espera una señal.
scantime	Tiempo de búsqueda empleado en cada canal.

- Desconectar un punto de acceso.

```
wlan.disconnect()
```

- Comprobar si está conectado. Devuelve *True*, si lo está.

```
wlan.isconnected()
```

- Establecer los parámetros de IP.

```
wlan.ifconfig(config)
```

Parámetros	Comentario
config	Si se establece config=('dhcp'), será el servidor DHCP el que establezca estos parámetros. Si se establece config=(ip, mascará de red, puerta de enlace, servidor DNS) se configura como estática. Por ejemplo, *wlan.ifconfig(config=('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))*

- Establecer conexión WiFi en modo promiscuo.

```
wlan.promiscuous ([bool])
```

- Configurar una función de callback solo si se ha activado el modo promiscuo.

```
wlan.callback(trigger, [handler=Null, arg=Null])
```

Parámetro	Comentario
trigger	Evento por el que se quiere disparar la función de callback. Puede tomar los siguientes valores: • *WLAN.EVENT_PKT_MGMT* cuando se reciba un paquete de gestión. • *WLAN.EVENT_PKT_CTRL* cuando se reciba un paquete de control. • *WLAN.EVENT_PKT_DATA* cuando se reciba un paquete de datos. • *WLAN.EVENT_PKT_DATA_MPDU* cuando se reciba un paquete de datos MPDU. • *WLAN.EVENT_PKT_DATA_AMPDU* cuando se reciba un paquete de datos AMPDU. • *WLAN.EVENT_PKT_MISC* cuando se reciba un paquete misceláneo. • *WLAN.EVENT_PKT_ANY* cuando se reciba cualquier paquete.
handler	Función que se desea lazar
arg	Argumentos pasados a la función

5.2.7.2 COAP

COAP es un protocolo de la capa de aplicación que permite a dispositivos de bajo consumo y poca capacidad de procesamiento, puedan conectarse a internet utilizando mensajes REST (haciendo uso de los métodos primitivos: GET, POST, PUT, DELETE) cortos y sencillos. Esto es esencial para desarrollar el concepto de IoT.

Las funciones para gestionar la comunicación COAP se encuentran en la clase *Coap* de la librería *network*.

```
from network import Coap
```

Vamos a tratar el caso de cliente y servidor COAP a través del LoPy4. Los métodos más relevantes para manejar estas conexiones, son los siguientes:

- Inicializar el módulo COAP.

```
Coap.init(adress, *, port=5683, service_discovery=False)
```

Parámetro	Comentario
adress	Dirección IP del dispositivo como cadena de caracteres.
port	Puerto donde iniciar el módulo.
service_discovery	Si se establece como *True*, se habilita la escucha en la dirección de multidifusión 224.0.1.187, por defecto está en *False*.

5.2.7.2.1 Cliente COAP

- Obtener socket de Coap creado en el puerto previamente seleccionado.

```
coap_socket=Coap.socket()
```

- Enviar una petición a un servidor COAP. Devuelve el ID de la petición.

```
id_pet=Coap.send_request(uri_host, method, *, uri_port=5683, uri_path, content_format, payload, token, include_options=true
```

Parámetro	Comentario
uri_host	Dirección IP del servidor como cadena de caracteres.
method	Método a usar en la petición. Este puede ser: • *Coap.REQUEST_GET* • *Coap.REQUEST_POST* • *Coap.REQUEST_PUT* • *Coap.REQUEST_DELETE*
uri_port	Puerto del servidor COAP.
uri_path	Ruta del recurso a usar en el lado del servidor. Si no se establece ninguna, no habrá uri_path.
content_format	Formato del contenido. Si no se establece, no tendrá esta opción. Este puede tomar los siguientes valores: • *Coap.MEDIATYPE_TEXT_PLAIN* • *Coap.MEDIATYPE_APP_LINK_FORMAT* • *Coap.MEDIATYPE_APP_XML* • *Coap.MEDIATYPE_APP_OCTET_STREAM* • *Coap.MEDIATYPE_APP_RDF_XML* • *Coap.MEDIATYPE_APP_EXI* • *Coap.MEDIATYPE_APP_JSON* • *Coap.MEDIATYPE_APP_CBOR*
payload	Datos útiles a enviar. Si no se establece, no habrá payload.
token	Token opcional para pasar. Si no se establece, no habrá token.
include_options	Si se establece como *True*, se habilitan las opciones previamente citadas. Por defecto está en *True*.

- Configurar una función de callback.

```
Coap.register_response_handler(función_callback)
```

Donde la función de callback que se le pasa como parámetro, debe tener los siguientes argumentos:

```
función_callback(code, id_param, type_param, token, payload)
```

Argumento	Comentario
code	Código de respuesta del mensaje recibido.
id_param	ID de la comunicación. Sirve para relacionar respuestas o detectar mensajes duplicados.
type_param	Para saber si es solicitud (confirmable o no) o respuesta (reconocimiento de confirmable o reenvío por error).
token	Información del mensaje enviada por el cliente y se replica en la respuesta.
payload	Carga útil del mensaje.

5.2.7.2.2 Servidor COAP

- Crear un recurso y lo añade al módulo COAP para operar como servidor. Los argumentos son los siguientes:

```
Coap.add_resource(uri, *, media_type=-1, max_age=-1, value=0)
```

Argumento	Comentario
uri	Nombre y ruta del recurso. Es una cadena de caracteres.
media_type	Tipo de datos del recurso. Si no se especifica, no se asocia ningún tipo. Puede tomar los siguientes valores: • *Coap.MEDIATYPE_TEXT_PLAIN* • *Coap.MEDIATYPE_APP_LINK_FORMAT* • *Coap.MEDIATYPE_APP_XML* • *Coap.MEDIATYPE_APP_OCTET_STREAM* • *Coap.MEDIATYPE_APP_RDF_XML* • *Coap.MEDIATYPE_APP_EXI* • *Coap.MEDIATYPE_APP_JSON* • *Coap.MEDIATYPE_APP_CBOR*
max_age	Tiempo máximo en segundos en los que el recurso se considera nuevo.
value	Valor por defecto del recurso. Por defecto es 0.

- Borrar un recurso creado en la ruta *uri*.

```
Coap.remove_resource(uri)
```

- Leer un recurso creado en *uri.*

```
recurso=Coap.get_resource(uri)
```

- Leer una petición entrante al socket del módulo Coap y envía una respuesta si es necesario. Llamar cuando se reciba un paquete.

```
Coap.read()
```

5.2.8 Bluetooth

Las funciones disponibles para el uso y configuración del módulo Bluetooth del LoPy4 se encuentran en la clase *Bluetooth* de la librería *network*.

```
from network import Bluetooth
```

Actualmente, solo está disponible el modo BLE (Bluetooth Low Energy). Las funciones para su uso y configuración son las siguientes:

- Iniciar el módulo bluetooth con los siguientes parámetros:

```
bt=Bluetooth([id=0,    mode=Bluetooth.BLE,        antenna=Bluetooth.INT_ANT,
modem_sleep=True, pin=None, privacy=True, secure_connections=True, mtu=200])
```

Parámetro	Comentario
id	Siempre 0 ya que solo existe un periférico Bluetooth
mode	Solo está disponible el modo BLE, por lo tanto, su único valor posible es *Bluetooth.BLE*
antenna	Sus posibles valores son: • *Bluetooth.INT_ANT* para usar la antena Bluetooth interna. • *Bluetooth.EXT_ANT* para usar la antena Bluetooth externa. Se debe configurar el pin P12 como salida para este modo. Ver sección GPIO.
modem_sleep	Establecer en *True* para que se duerma el módulo cuando no transmita ni reciba.
secure	Habilita o deshabilita la conexión con el servidor GATT.
pin	Número de 6 dígitos para conectarse al servidor GATT. Si este se establece, se activan las funciones de seguridad del mismo.
privacy	Habilita o deshabilita la configuración de seguridad local configurando la dirección como aleatoria o pública.
secure_connections	Habilita o deshabilita la protección MITM de conexiones seguras.
mtu	Longitud máxima de un paquete ATT. Puede tomar valores entre 23 y 200.

- Desactivar el módulo Bluetooth

```
bt.deinit()
```

- Iniciar búsqueda de anuncios BLE.

```
bt.start_scan(tiempo)
```

Parámetro	Comentario
tiempo	Tiempo que durará la búsqueda en segundos. Si se establece un número negativo, busca indefinidamente.

- Comprobar si está buscando anuncios. Devuelve *True* si está buscando.

```
val = bt.isscanning()
```

- Detener la búsqueda de anuncios BLE.

```
bt.stop_scan()
```

- Obtiene el anuncio BLE detectado durante la búsqueda. Devuelve un vector con información sobre dicho anuncio.

```
adv=bt.get_adv()
```

Valor del vector	Comentario
mac	Dirección MAC del emisor. Tiene una longitud de 6 bytes.
addr_type	Solo está disponible el modo BLE, por lo tanto, su único valor posible es *Bluetooth.BLE*
adv_type	Tipo de anuncio recibido. Puede ser de los siguientes tipos: • *Bluetooth.CONN_ADV* • *Bluetooth.CONN_DIR_ADV* • *Bluetooth.DISC_ADV* • *Bluetooth.NON_CONN_ADV* • *Bluetooth.SCAN_RSP*
rssi	Número entero que muestra la intensidad de la señal con la que ha llegado el anuncio.
data	Contiene los 31 bytes del anuncio BLE. Para leer los distintos tipos de datos, se utiliza el método *resolve_adv_data(data, data_type).*
pin	Número de 6 dígitos para conectarse al servidor GATT. Si este se establece, se activan las funciones de seguridad del mismo.
privacy	Habilita o deshabilita la configuración de seguridad local configurando la dirección como aleatoria o pública.
secure_connections	Habilita o deshabilita la protección MITM de conexiones seguras.
mtu	Longitud máxima de un paquete ATT. Puede tomar valores entre 23 y 200.

- Obtener una lista con los anuncios BLE detectados durante la búsqueda. Similar a *get_adv().*

```
advs = bt.get_advertisements()
```

- Para leer los datos de un anuncio BLE.

```
data = bt.resolve_adv_data(data, data_type)
```

Parámetro	Comentario
data	Datos que queremos leer. En el caso del ejemplo, sería *adv.data.*
data_type	Tipo de dato que queremos leer. Si no existe el tipo de dato seleccionado, devuelve None. Puede tomar los siguientes valores: • *Bluetooth.ADV_FLAG* • *Bluetooth.ADV_16SRV_PART* • *Bluetooth.ADV_T16SRV_CMPL* • *Bluetooth.ADV_32SRV_PART* • *Bluetooth.ADV_32SRV_CMPL* • *Bluetooth.ADV_128SRV_PART* • *Bluetooth.ADV_128SRV_CMPL* • *Bluetooth.ADV_NAME_SHORT* • *Bluetooth.ADV_NAME_CMPL* • *Bluetooth.ADV_TX_PWR* • *Bluetooth.ADV_DEV_CLASS* • *Bluetooth.ADV_SERVICE_DATA* • *Bluetooth.ADV_APPEARANCE* • *Bluetooth.ADV_ADV_INT* • *Bluetooth.ADV_32SERVICE_DATA* • *Bluetooth.ADV_128SERVICE_DATA* • *Bluetooth.ADV_MANUFACTURER_DATA*

- Establecer una conexión con un dispositivo BLE.

```
bt.connect(mac_adr, timeout)
```

Parámetro	Comentario
mac_adr	Dirección MAC del dispositivo a conectarse.
timeout	Tiempo máximo para intentar realizar la conexión.

- Terminar una conexión BLE.

```
bt.disconnect_client()
```

- Configurar una función de callback

```
bt.callback(trigger = None, handler = None, arg = None)
```

Parámetro	Comentario
trigger	Evento por el que se quiere disparar la función de callback. Puede tomar los siguientes valores: • *Bluetooth.CHAR_READ_EVENT* • *Bluetooth.CHAR_WRITE_EVENT* • *Bluetooth.NEW_ADV_EVENT* • *Bluetooth.CLIENT_CONNECTED* • *Bluetooth.CLIENT_DISCONNECTED* • *Bluetooth.CHAR_NOTIFY_EVENT*
handler	Función que se desea lazar
arg	Argumentos pasados a la función

- Crear un anuncio BLE:

```
bt.set_advertisement([name=None,manufacturer_data=None,service_data=None,
service_uuid=None])
```

Parámetro	Comentario
name	Cadena de caracteres que da nombre al anuncio BLE
manufacturer_data	Datos que se publican en el anuncio.
service_data	Datos del servicio que se publica en el anuncio.
service_uuid	UUID del servicio publicado.

- Configurar los parámetros del anuncio BLE

```
bt.set_advertisement_params(adv_int_min=0x20,adv_int_max=0x40,adv_type=Bluetooth.ADV_TYPE_IND,own_addr_type=Bluetooth.BLE_ADDR_TYPE_PUBLIC,channel_map=Bluetooth.ADV_CHNL_ALL,adv_filter_policy=Bluetooth.ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY)
```

Parámetro	Comentario
adv_int_min	Valor de intervalo de anuncio mínimo, para anuncios indirectos y directos de bajo ciclo de trabajo.
adv_int_max	Valor de intervalo de anuncio máximo, para anuncios indirectos y directos de bajo ciclo de trabajo.
adv_type	Tipo de anuncio. Puede tomar los siguientes valores: • *Bluetooth.ADV_TYPE_IND* • *Bluetooth.ADV_TYPE_DIRECT_IND_HIGH* • *Bluetooth.ADV_TYPE_SCAN_IND* • *Bluetooth.ADV_TYPE_NONCONN_IND* • *Bluetooth.ADV_TYPE_DIRECT_IND_LOW*
own_adr_type	Tipo de dirección del dispositivo. Puede tomar los siguientes valores: • *Bluetooth.ADV_BLE_ADDR_TYPE_PUBLIC* • *Bluetooth.ADV_BLE_ADDR_TYPE_RANDOM* • *Bluetooth.ADV_BLE_ADDR_TYPE_RPA_PUBLIC* • *Bluetooth.ADV_BLE_ADDR_TYPE_RPA_RANDOM*
channel map	Mapa del canal de anuncio. Puede tomar los siguientes valores: • *Bluetooth.ADV_CHNL_37* • *Bluetooth.ADV_CHNL_38* • *Bluetooth.ADV_CHNL_39* • *Bluetooth.ADV_CHNL_ALL*
adv_filter_policy	Tipo de filtrado de anuncios. Estos son los tipos de filtro: • *Bluetooth.ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY* • *Bluetooth.ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY* • *Bluetooth.ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST* • *Bluetooth.ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST*

- Activar el envío de anuncios BLE.

```
bt.advertise([Enable])
```

- Configurar o leer la potencia de envío de la antena BLE.

```
bt.tx_power(type, level)
```

Parámetro	Comentario
type	Puede tomar los siguientes valores: • *Bluetooth.TX_PWR_CONN* para mantener una conexión. • *Bluetooth.TX_PWR_ADV* para mandar avisos BLE. • *Bluetooth.TX_PWR_SCAN* para buscar avisos BLE. • *Bluetooth.TX_PWR_DEFAULT* para usarlo genéricamente.
level	Nivel de potencia de la señal. Si no se incluye este parámetro, devolverá el valor actual en vez de configurarlo. Este valor puede ser: • *Bluetooth.TX_PWR_N12* para una potencia de -12dBm • *Bluetooth.TX_PWR_N9* para una potencia de -9dBm • *Bluetooth.TX_PWR_N6* para una potencia de -6dBm • *Bluetooth.TX_PWR_N3* para una potencia de -3dBm • *Bluetooth.TX_PWR_0* para una potencia de 0dBm • *Bluetooth.TX_PWR_P3* para una potencia de 3dBm • *Bluetooth.TX_PWR_P6* para una potencia de 6dBm • *Bluetooth.TX_PWR_P9* para una potencia de 9dBm

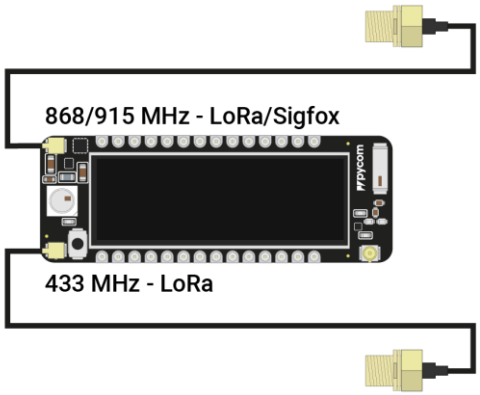
5.2.9 LoRa

La configuración del módulo LoRa en Lopy4 es muy sencilla gracias a los métodos de la clase *Lora* de la librería *network*.

```
from network import Lora
```

Estas son las funciones para gestionar una comunicación Lora:

No se debe olvidar conectar la antena al dispositivo antes de inicializar el modulo Lora, ya que de lo contrario podría ocasionar daños en el LoPy4 .



- Habilitar el módulo LoRa, con sus principales parámetros de configuración:

```
lora=LoRa(mode=LoRa.LORAWAN, [regionLoRa.EU868, frequency, tx_power, bandwidth=LoRa.BW_125KHZ, sf=7, preamble=8, coding_rateLoRa.CODING_4_5, power_mode=LoRa.ALWAYS_ON, tx_iq=False, rx_iq=False])
```

<table>
<tr><th>Parámetro</th><th>Comentario</th></tr>
<tr><td>mode</td><td>Modo de funcionamiento de LoRa. Puede ser:
LoRa.LORA si se trata de una red privada.
LoRa.LORAWAN si queremos seguir los estándares de LoRaWAN.</td></tr>
<tr><td>region</td><td>Región donde se va a usar el dispositivo para ajustar la frecuencia específica. Puede tomar los siguientes valores:
LoRa.AS923
LoRa.AU915
LoRa.EU868
LoRa.US915
LoRa.CN470
LoRa.IN865</td></tr>
<tr><td>frequency</td><td>Permite ajustar la frecuencia concreta dentro de los rangos permitidos en cada zona.
Tabla 3: Frecuencias de LoRa según región<table><tr><th>Región</th><th>Rango de frecuencias</th></tr><tr><td>AS923</td><td>920 - 925 MHz</td></tr><tr><td>AU915</td><td>915 - 928MHz</td></tr><tr><td>CN470</td><td>470 - 510 MHz</td></tr><tr><td>EU868</td><td>863 - 870 MHz</td></tr><tr><td>IN865</td><td>865 - 867 MHz</td></tr><tr><td>US915</td><td>902 - 928 MHz</td></tr></table></td></tr>
<tr><td>tx_power</td><td>Se establece la potencia de envío en dBm</td></tr>
<tr><td>bandwith</td><td>Ajusta el ancho de banda de cada canal. Puede tomar los siguientes valores:
LoRa.BW_125KHZ
LoRa.BW_250KHZ
LoRa.BW_500KHZ</td></tr>
<tr><td>sf</td><td>Ajusta el Spreading Factor deseado. Puede tomar valores enteros entre 7 y 12.</td></tr>
<tr><td>preamble</td><td>Ajusta el número de símbolos del preámbulo. Su valor por defecto es 8.</td></tr>
</table>

coding_rate	Número de bits útiles en función de los bits generados. Puede tomar los siguientes valores: • *LoRa.CODING_4_5* • *LoRa.CODING_4_6* • *LoRa.CODING_4_7* • *LoRa.CODING_4_8*
power_mode	Ajusta el modo de uso de la radio LoRa, con las siguientes posibilidades: • *LoRa.ALWAYS_ON*: la radio siempre está activa para recibir paquetes. • *LoRa.TX_ONLY*: la radio se duerme en cuanto no está enviando. • *LoRa.SLEEP*: la radio está siempre dormida hasta que no se cambie el modo.
tx_iq	Habilita la inversión IQ del TX
rx_iq	Habilita la inversión IQ del RX
adr	Solo para el modo LoRaWAN. Habilita el ADR.
public	Solo para el modo LoRaWAN. Selecciona una palabra de sincronización pública.
tx_retries	Solo para el modo LoRaWAN. Especifica el número de reintentos del transmisor.
device_class	Solo para el modo LoRaWAN. Selecciona entre: • *LoRa.CLASS_A* • *LoRa.CLASS_C*

- Devolver un valor que indica cuál ha sido el último evento sucedido. Tras ser llamada se borra el registro.

```
event=lora.events()
```

Valor devuelto	Comentario
LoRa.RX_PACKET_EVENT	Se ha recibido un paquete.
LoRa.TX_PACKET_EVENT	Se ha enviado un paquete
LoRa.TX_FAILED_EVENT	Ha fallado el envío de un paquete

- Crear una llamada a función cuando se active algún evento LoRa.

```
lora.callback(trigger=(LoRa.RX_PACKET_EVENT | LoRa.TX_PACKET_EVENT), handler=lora_cb)
```

Parámetro	Comentario
trigger	Especifica con qué eventos será llamada la función.
handler	Nombre de la función a ser llamada.

- Devuelve un vector nombrado con información acerca del último paquete recibido.

```
vec=lora.stats()
```

Valor del vector	Comentario
rx_timestamp	Marca de tiempo interno (microsegundos) desde que se recibió el último paquete.
rssi	Fuerza de la señal recibida en dBm.
snr	Relación señal/ruido en dBm.
sfrx	Muestra el Spreading Factor del último paquete recibido.
sftx	Muestra el Spreading Factor del último paquete transmitido.
tx_trials	Intentos del último paquete enviado (solo para LoRaWAN).
tx_power	Potencia de señal en dBm de la última transmisión.
tx_time_on_air	Tiempo en el aire del último paquete transmitido.
tx_counter	Número de paquetes transmitidos.
tx_frequency	Frecuencia usada en la última transmisión.

5.2.10 Otros métodos de utilidad.

5.2.10.1 Librería *time*

En esta librería, se detallan varias funciones muy útiles para programar el LoPy4. Está presente en la mayoría de ejemplos de programación que más adelante se detallan. Para poder usar estos métodos, se debe importar la librería de la siguiente manera:

```
import time
```

Las funciones más importantes de dicha librería son:

- Dormir el LoPy4 durante *s* segundos.

```
time.sleep(s)
```

- Dormir el LoPy4 durante *ms* milisegundos.

```
time.sleep_ms(ms)
```

- Dormir el LoPy4 durante *us* microsegundos.

```
time.sleep_us(us)
```

- Segundos que han pasado desde el arranque. Devuelve un entero equivalente al número de segundos transcurridos.

```
tiempo_s = time.time()
```

- Vector de tiempo con la fecha actual. Devuelve un vector nombrado con la siguiente estructura:

  (año, mes, día del mes, hora, minutos, segundos, día de la semana, día del año)

Si se le pasa un entero de número de segundos como argumento, se traduce ese número de segundos al vector de tiempo.

```
tiempo_vec = time.gmtime(seg)
```

5.2.10.2 Librería *socket*

Esta librería la vamos a usar para crear los sockets necesarios para el envío y recepción de mensajes. En nuestro caso, lo usaremos para las transmisiones LoRa. Sus métodos se encuentran alojados en la librería *socket*, que se importa de la siguiente manera:

```
import socket
```

Los métodos más importantes que usaremos en los ejemplos de esta librería, son los siguientes:

- Crear un socket según la dirección de la tecnología, tipo y protocolo.

```
socket.socket(familia, tipo, protocolo)
```

Parámetro	Comentario
familia	Tecnología del LoPy4 que usará el socket. Pueden ser las siguientes: • *socket.AF_INET* • *socket.AF_LORA* • *socket.AF_SIGFOX*
tipo	Tipo de socket a crear. Puede tomar los siguientes valores: • *socket.SOCK_STREAM* para conexiones TCP. • *socket.SOCK_DGRAM* para conexiones UDT. • *socket.SOCK_RAW* para conexiones sin protocolo (LoRa).
protocolo	Se establece el protocolo a usar, si *socket.IPPROTO_UDP* para UDP o *socket.IPPROTO_TCP* para TCP.

- Enviar datos al socket.

```
socket.send(datos)
```

- Recibir y guardar *N* bytes del buffer del socket.

```
datos = socket.recv(N)
```

6 EJEMPLOS DE PROGRAMACIÓN

Our whole life is solving puzzles.

- Ernő Rubik -

A Continuación, se van a exponer varios ejemplos de programación de los distintos módulos que tiene LoPy4. Se va a intentar poner ejemplos de la mayoría de casos de uso de estos módulos. Finalmente, se añaden un ejemplo de simulación de aplicación IoT con un nodo LoRa y otro actuando de puerta de enlace hasta un servidor COAP.

4.3. Pycom

En este ejemplo se programa el módulo usando la librería *pycom* para programar el LED RGB del dispositivo, y la librería *time* para dormir la placa durante medio segundo.

```
import pycom
import time

#Desactivamos el heartbeat para usar el LED RGB
pycom.heartbeat(False)

#Vector con colores en RGB
colors=[0xFF5733,0x8E44AD,0x3498DB,0x27AE60,0xF1C40F,0xE67E22]

while True:

    # Recorremos el vector de colores cambiando cada medio segundo
    for color in colors:
      pycom.rgbled(color)
      time.sleep(1)
```

4.4. GPIO

Para este módulo, se ha desarrollado un ejemplo en el que vamos alternando el estado del LED situado en el pin P9 de la *expansion board*, cada vez que se pulse el botón situado en el pin P10 de la *expansion board*. Como se puede apreciar, se usa una función de callback para cambiar el estado del led, detectando los flancos de subida del pin P10 (botón).

```
from machine import Pin
import time

#Función de callback
def cambio(led):
    #Si el led está encendido, se apaga. Si no, se enciende.
    if led():
        led.value(0)
    else:
        led.value(1)

#Inicializamos ambos pines: P9 como salida (LED en expansion board)
#                           P10 como entrada (boton en expansion board)
led = Pin('P9', mode = Pin.OUT)
boton = Pin('P10', mode = Pin.IN)

#Configuramos la función de callback cuando se pulse el boton
boton.callback(trigger=Pin.IRQ_FALLING, handler=cambio, arg=led)

#Dormimos el LoPy4 mientras no pase nada
while True:
    time.sleep(-1)
```

4.5. Timers

En el siguiente ejemplo, vemos como se programa un timer como cronómetro. Se simulan 2 tiempos de retardo para esperar a leer los valores del cronómetro y ver cómo ha salido el experimento. Finalmente se calculan los errores cometidos en las medidas.

Por otro lado, la programación del timer como alarma con función de callback se usará en otro ejemplo de programación, por lo que no será detallado en este apartado.

```
  from machine import Timer
  import time

  #Inicializamos el cronómetro
  crono = Timer.Chrono()

  #Definimos los retardos
  ret1=2
  ret2=2.5

  #Iniciamos el cronómetro
  crono.start()
  time.sleep(ret1) #Simulamos un tiempo de espera
  lec1 = crono.read() #Leemos el valor intermedio
  time.sleep(ret2)
  crono.stop()
  total = crono.read()

  #Calculamos errores
  error1=100*abs(ret1-lec1)/ret1
  error2=100*abs(ret2-(total-lec1))/ret2
  errort=100*abs(ret1+ret2-total)/(ret1+ret2)

  print("\nExacto: Total -> ", (ret1+ret2)," ; Paso 1 -> ",ret1," ; Paso 2 ->
" , ret2 )
  print("\nMedido: Total -> ", total, "; Paso 1 -> ", lec1,"; Paso 2 -> ",
(total-lec1))
  print("\nError: Total -> ",errort,"% ; Paso 1 -> ", error1,"%  ; Paso 2 -> "
, error2,"%")
```

4.6. DAC

En el siguiente ejemplo, se configura el conversor digital-analógico para sacar por el pin P21, distintos valores analógicos entre 0 y 3.3V.

Primero, se crea una rampa ascendente, que en 10 segundos pasa de 0 a 3.3V en 10 pasos (1 por segundo). Después, se crea una onda sinusoidal de 1kHz y 3.3Vpp y se mantiene 5 segundos. Finalmente, otra onda sinusoidal de 50kHz y 1.5Vpp, nuevamente mantenida durante 5 segundos.

```
from machine import DAC
import time

#Instancia del DAC
dac = DAC('P21')

while True:

    #Creamos rampa ascendente de 0 a 3.3V AC durante 10 segundos
    dac.write(0)
    dac.init()
    for i in range(1,10,1):
        time.sleep(1)
        dac.write(i/10)
    dac.deinit()

    #Creamos señal sinusoidal de frecuencia 1kHz y 3.3Vpp (0dBm)
    dac.tone(1000, 0)
    dac.init()
    time.sleep(5)
    dac.deinit()

    #Creamos señal sinusoidal de frecuencia 50kHz y 1.5Vpp (-6dBm)
    dac.tone(50000, 1)
    dac.init()
    time.sleep(5)
    dac.deinit()
```

4.7. PWM

Se ha tratado de crear un ejemplo visual para el PWM sin necesidad de disponer de cualquier actuador para ver su funcionamiento. En este caso, se ha usado como salida PWM el pin *P9*, que está conectado al LED de la *expansion board 3.0*. Esto permirá ver un cambio de intensidad en el brillo del LED, según se el duty cycle del PWM.

Para ello, se han creado 2 rampas, una ascendente y otra descendente del duty cycle para similar una sinusoide de brillo del LED.

```
from machine import PWM
import time

#Instancia del PWM con frecuencia de 1kHz
pwm = PWM(0, frequency=1000)

#Creamos un canal de PWM en el pin P9
pwm_c = pwm.channel(0, pin='P9', duty_cycle=0)

while True:

    #Rampa ascendente del duty cycle del PWM
    for dc in range(0,10,1):
        time.sleep(0.25)
        pwm_c.duty_cycle(dc/10)
        print("\nDC -> ",(dc/10))

    #Rampa descendente del duty cycle del PWM
    for dc in range(0,10,1):
        time.sleep(0.25)
        pwm_c.duty_cycle(1-dc/10)
        print("\nDC -> ",(1-dc/10))
```

4.8. BLE

En primer lugar, se ha desarrollado un ejemplo para enviar un anuncio BLE. Se ha hecho un experimento de potencia de antenna Bluetooth interna.

```
  from network import Bluetooth
  import time

  #Instancia del Bluetooth
  bt = Bluetooth()

  while True:

      #Creamos un anuncio BLE
      bt.set_advertisement(name="Hola   soy   LoPy4",   manufacturer_data="Mis
Datos")

      #Cambiamos potencia de antenna a 9dBm
      bt.tx_power(Bluetooth.TX_PWR_ADV, Bluetooth.TX_PWR_P9)

      #Emitimos el anuncio
      bt.advertise(True)

      time.sleep(0.5)
```

Se ha comprobado con una APP de Google Play el resultado de los anuncios creados.

Figura 15: Datos leídos del anuncio BLE

En la Figura 15, se puede apreciar en el campo correspondiente al nombre del anuncio, en este caso el segundo, cuyo valor hexadecimal “48 6F 6C 61 20 73 6F 79 20 4C 6F 50 79 34”, convertido a cadena de caracteres, sería “Hola soy LoPy4”, y los datos útiles (o manufacturados), en este caso el penúltimo campo, la secuencia hexadecimal es “4D 69 73 20 44 61 74 6F 73”, que corresponde a la cadena de caracteres “Mis Datos”.

En el segundo ejemplo, se trata de crear un escaner de anuncios BLE mostrando la información de estos mismos.

```
from network import Bluetooth
import time
import ubinascii

#Instancia del Bluetooth
bt = Bluetooth()

#Escaneamos infinitamente
bt.start_scan(-1)

while True:

    #Leemos anuncios
    advs = bt.get_advertisements()

    #Por cada anuncio leemos los datos
    for adv in advs:
        if adv:

            #Convertimos los datos de cada anuncio
            nombre=bt.resolve_adv_data(adv.data, Bluetooth.ADV_NAME_CMPL)
            man_dat=bt.resolve_adv_data(adv.data,
Bluetooth.ADV_MANUFACTURER_DATA)
            dir_mac=ubinascii.hexlify(adv.mac)

            #Mostramos los datos leidos
            print('\n-----------------------------------------------------')
            print("\nNombre del anuncio = ", nombre)
            print("\nDatos = ", man_dat)
            print("\nDireccion MAC = ", dir_mac)
            print('\n-----------------------------------------------------')

    time.sleep(1)
```

Se ha comprobado que efectivamente recibe los anuncios BLE de los dispositivos cercanos. Se ha emulado uno desde al APP de Google Play, *Bluetooth LE Tool*, un anuncio con una cadena de caracteres “Prueba” como carga útil.

```
-----------------------------------------------------

Nombre del anuncio =  Galaxy Note10+

Datos =  b'\x00\x00Prueba'

Direccion MAC =  b'798a106df5e5'

-----------------------------------------------------
```

4.9. Nodo terminal con ejemplos de LoRa (envio), ADC y Timer como alarma.

Se ha desarrollado una simulación de aplicación IoT. Para ello, en este primer ejemplo veremos cómo actúa un nodo terminal LoRa, el cual se encarga de recibir los datos de un sensor y transmite la información vía LoRa, hacia otro nodo LoRa que actúe de puerta de enlace.

En este ejemplo, se han creado 2 clases, las cuales crean instancias de temporizador como alarma:

- La primera clase, *lectura_adc*, instancia, se dispara cada segundo y su función de callback, *lectura,* se encarga de leer el valor analógico del pin *P16,* donde se ha conectado un micrófono. También, para evitar falsas alarmas, compara cada medida con las 5 anteriores para determinar si ha sido realmente una señal sustancial, y active un flag de aviso de superación de umbral de sonido. Hay que recalcar, que solo empieza el muestreo una vez se haya completado la emisión del mensaje LoRa.
- La segunda clase, *transmision_LoRa*, instancia, se dispara cada 5 segundos y su función de callback es *evia_paquete.* Esta envía un mensaje LoRa y cambia el flag de envió, para informar a la clase de lectura del ADC que ya puede resetear su flag de activación.

Por último, en el bucle principal, duerme el LoPy4 y se despierta cada 250ms para comprobar el estado del flag de activación del ADC y cambiar el LED RGB del dispositivo.

```
  from machine import Timer
  from machine import ADC
  from network import LoRa
  import socket
  import time
  import pycom

  class transmision_LoRa:
      #Instanciamos e iniciamos la alarma para enviar paquetes LoRa
      def __init__(self):
          self.flag = 1
          self.__temp_trans_lora    =    Timer.Alarm(self._envia_paquete,    5,
periodic=True)

      #Función callback encargada de enviar paquete
      def _envia_paquete(self, temp_trans_lora):

          #Enviamos el paquete con el flag del ADC
          s.setblocking(True)
          s.send(lec_adc.flag)
          s.setblocking(False)

          #Cambiamos flag de envio para resetear flag del muetsreo
          if self.flag == 0:
              self.flag = 1
          else:
              self.flag = 0


  class lectura_adc:
      #Instanciamos e iniciamos la alarma para leer el ADC, y variables.
      def __init__(self):
          self.flag = 0
          self.data=[0,0,0,0,0]
          self.transflag = trans_LoRa.flag
          self.__temp_lec_adc = Timer.Alarm(self._lectura, 0.3, periodic=True)
```

```
    #Función callback encargada de leer el ADC
    def _lectura(self, temp_lec_adc):
        #Leemos el valor
        self.valor = adc_c.voltage()

        #Se compara con las anteriores medidas
        for i in self.data:
            if self.valor<i:
                #Si es menor que cada elemento, flag activo
                self.flag = 1
                continue
            #Si es igual o mayor que las anteriores uestras
            else:
                #y se ha trasnmitido ya el paquete, reset del flag
                if (self.transflag_ant!=trans_LoRa.flag):
                    self.flag = 0
                break

        #Recosntruimos el vector de registro de muestreo y variables
        for i in range(1,5):
            self.data[5-i]=self.data[4-i]
        self.data[0]=self.valor
        self.transflag_ant = trans_LoRa.flag

#Funcion de callback de LoRa para monitorizar envios
def lora_cb(lora):
    evento = lora.events()
    if evento & LoRa.TX_FAILED_EVENT:
        print('Error de envio')
    if evento & LoRa.TX_PACKET_EVENT:
        print('Paquete enviado')

#Desactivamos el heartbeat para usar el LED RGB
pycom.heartbeat(False)

#Instancia y canal del ADC en el pin P16
adc = ADC()
adc.vref(1100)
adc_c = adc.channel(pin='P16', attn=ADC.ATTN_6DB)

#Instancia LoRa y socket
lora   =   LoRa(mode=LoRa.LORA,   region=LoRa.EU868,   tx_power=14,   sf=12,
bandwidth=LoRa.BW_250KHZ)
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
s.setblocking(False)
lora.callback(trigger=(LoRa.TX_FAILED_EVENT     |     LoRa.TX_PACKET_EVENT),
handler=lora_cb)

#Arranque transmision LoRa
trans_LoRa = transmision_LoRa()
lec_adc = lectura_adc()

while True:
    time.sleep(0.25)
    if lec_adc.flag:
        pycom.rgbled(0x00FF00)  # Verde si el flag esta activado
    else:
        pycom.rgbled(0xFF0000)  # Rojo si no esta activado
```

4.10. Nodo puerta de enlace con LoRa (recepción), WLAN y COAP.

El siguiente ejemplo complementa el anterior, completando así la aplicación IoT. En este caso, se reciben los mensajes LoRa del primer nodo y se envía su carga útil vía WiFi a un servidor COAP. Se ha tratado de emular un nodo puerta de enlace o gateaway, entre LoRa e internet.

En el código, primero se encuentran las funciones callback de eventos LoRa y COAP, las cuales sirven para monitorizar el correcto envío y recepción de mensajes. Tras ello, el hilo de sondeo encargado de detectar si ha entrado un mensaje COAP nuevo. Se inicia la conexión WLAN a un router cercano, conectado a internet para, finalmente, inicializar los módulos COAP y LoRa.

El bucle principal del programa se encarga de leer los mensajes LoRa y comprobar si su contenido es el esperado. En caso afirmativo se envía la carga útil a través de COAP al servidor de internet.

```
from network import LoRa
from network import WLAN
from network import Coap
import socket
import pycom
import time
import uselect
import _thread

#Función de callback de LoRa
def lora_cb(lora):
    evento = lora.events()
    #Si se recibe un paquete, LED RGB verde
    if evento & LoRa.RX_PACKET_EVENT:
        pycom.rgbled(0x00FF00)

#Función de callback de respuestas COAP y plot de datos de interés
def callback_respuestas_COAP(code, id_param, type_param, token, payload):
    print("ID: {}".format(id_param))
    print("Payload: {}".format(payload))

#Hilo de socket COAP
def hilo_socket_COAP(sondeo, coap_socket):
    while True:
        #leemos los sockets de la encuesta poll
        sockets = sondeo.poll()
        #para cada socket del sondeo, comprobamos si es un mensaje de entrada
y es del módulo COAP
        for s in sockets:
            sock = s[0]
            event = s[1]
            if(event & uselect.POLLIN):
                if(sock == coap_socket):
                    #si es así, leemos el mensaje
                    Coap.read()

#Desactivamos el heartbeat para usar el LED RGB
pycom.heartbeat(False)
```

```
  #Instancia e inicialización de LoRa
  lora   =   LoRa(mode=LoRa.LORA,   region=LoRa.EU868,   tx_power=14,   sf=12,
bandwidth=LoRa.BW_250KHZ)
  s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
  lora.callback(trigger=(LoRa.RX_PACKET_EVENT     |     LoRa.TX_PACKET_EVENT),
handler=lora_cb)

  # Instancia WLAN y conexión WiFi
  wlan = WLAN(mode=WLAN.STA)
  wlan.connect(ssid='MIWIFI_2G_geit', auth=(WLAN.WPA2, 'i9x2fj5cwbfw'))
  while not wlan.isconnected():
      time.sleep_ms(50)
      pycom.rgbled(0xEE00EE)  # LED RGB Morado -> conectando
  pycom.rgbled(0x00FFBA)  # LED RGB Verde claro -> conectado

  # Inicio módulo COAP
  Coap.init(str(wlan.ifconfig()[0]), service_discovery=True)
  #Server COAP a la escucha de las respuestas del otro server
  Coap.register_response_handler(callback_respuestas_COAP)
  #Obtenemos token COAP
  socket_servidor_COAP = Coap.socket()
  #Creamos objeto de sondeo
  sondeo = uselect.poll()
  #Registramos el socket de COAP en el sondeo como entrada
  sondeo.register(socket_servidor_COAP, uselect.POLLIN)
  #Iniciamos el hilo que escucha el sondeo
  _thread.start_new_thread(hilo_socket_COAP, (sondeo, socket_servidor_COAP))


  while True:
      #leemos mensaje recibido en socket de LoRa
      mensaje = s.recv(64)
      #si es el mensaje que esperamos, enviamos los datos vía COAP
      if mensaje==b'0' or mensaje==b'1':
          Coap.send_request("192.168.1.132",  Coap.REQUEST_GET,  uri_port=5683,
payload=mensaje, include_options=False)
      else:
          #si no se ha recibido uno válido, apagamos LED
          pycom.rgbled(0x000000)
```

7 TESTS DE LORA

Ninguna cantidad de experimentación puede probar definitivamente que tengo razón; pero un solo experimento puede probar que estoy equivocado.

- Albert Einstein -

Para medir la cobertura que ofrece LoPy4 para redes LPWAN, se han realizado varios experimentos con mensajes del estándar LoRa, en una red privada (sin usar los servicios de LoRaWAN), para medir los distintos parámetros que manejan este tipo de conexiones.

Se va a tratar de estimar la distancia máxima a la que podemos transmitir vía LoRa RAW, variando la potencia de la antena del nodo emisor y el Spreading Factor.

Para poder ser llevado a cabo de una manera sencilla, se ha desarrollado una conexión entre 2 dispositivos LoPy4 conectados vía LoRa dónde:

- Un nodo emisor, instalado en un punto estático, cada 3 segundos envía mensajes LoRa en bucle, variando la potencia de transmisión entre 2dB, 8dB, 14dB y 20dB, y el Spreading Factor entre 8, 10 y 12.
- Un nodo receptor, se va desplazando y comprobando si recibe o no los distintos mensajes de las diferentes potencias y SFs. Mediante el botón de la *Expansion board 3.0*, se va cambiando el Spreading Factor para poder leer los distintos anuncios.

El objetivo es ir aumentando el radio de cobertura con el nodo receptor, midiendo cada km qué potencias llegan con los diferentes Spreading Factors.

4.11. Código del nodo emisor

```
from machine import Timer
from machine import ADC
from network import LoRa
import socket
import time
import pycom

#Funcion de callback de LoRa para monitorizar envios
def lora_cb(lora):
    evento = lora.events()
    if evento & LoRa.TX_FAILED_EVENT:
        print('Error de envio')
    if evento & LoRa.TX_PACKET_EVENT:
        pycom.rgbled(0x00FF00)
        print('Paquete enviado')

#Desactivamos el heartbeat para usar el LED RGB
pycom.heartbeat(False)

# Instancia LoRa y socket
lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868, tx_power=14, sf=10,
bandwidth=LoRa.BW_250KHZ)
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
lora.callback(trigger=(LoRa.TX_FAILED_EVENT | LoRa.TX_PACKET_EVENT),
handler=lora_cb)




while True:
    for power in [2, 8, 14, 20]:
        for sf in [8, 10, 12]:
            pycom.rgbled(0x000000)
            lora.tx_power(power)
            lora.sf(sf)
            msj = "Test PW:{}".format(power)
            s.send(msj)
            print(msj)
            print(lora.stats())
            time.sleep(4)
```

4.12. Código del nodo receptor

```
from network import LoRa
import pycom
import socket
import time
from machine import Pin

#Función de callback de LoRa
def lora_cb(lora):
    evento = lora.events()
    #Si se recibe un paquete, LED RGB verde
    if evento & LoRa.RX_PACKET_EVENT:
        print(lora.stats())
        pycom.rgbled(0x00FF00)

#Función de callback del botón para cambiar el SF
def cambio(sf):
    time.sleep(0.3)
    sf=lora.stats()[3] #Pillamos el SF actual
    sf=sf+2
    if sf>12 or sf<8:
        sf=8
    if sf == 12:
        pycom.rgbled(0xF4D03F0) #Amarillo
    if sf == 10:
        pycom.rgbled(0x6C3483) #Morado
    if sf == 8:
        pycom.rgbled(0x2E4053) #Oscuro
    time.sleep(1)
    lora.sf(sf)

#Desactivamos el heartbeat para usar el LED RGB
pycom.heartbeat(False)
sf=12
boton = Pin('P10', mode = Pin.IN)
boton.callback(trigger=Pin.IRQ_FALLING, handler=cambio, arg=sf)

# Instancia e inicialización de LoRa
lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868, tx_power=14, sf=10,
bandwidth=LoRa.BW_250KHZ)
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
lora.callback(trigger=(LoRa.RX_PACKET_EVENT | LoRa.TX_PACKET_EVENT),
handler=lora_cb)

while True:
    #leemos mensaje recibido en socket de LoRa
    time.sleep(1)
    mensaje = s.recv(64)
    if mensaje == b'Test PW:2':
        pycom.rgbled(0x00FF00)
    if mensaje == b'Test PW:8':
        pycom.rgbled(0xFF0000)
    if mensaje == b'Test PW:14':
        pycom.rgbled(0x0000FF)
    if mensaje == b'Test PW:20':
        pycom.rgbled(0xFFFFFF)

    print(mensaje)
```

4.13. Experimentos

Se han llevado a cabo diferentes experimentos según su medio de propagación:

7.1.1 Zona Urbana

Se han establecido distintos puntos de conexión a diferentes distancias, con el fin de comprobar la cobertura que ofrece el dispositivo en un medio urbano, es decir, con muchas señales de interferencia obstaculizando la conexión. El alcance máximo al que se han recibido mensajes LoRa es de 5 km.

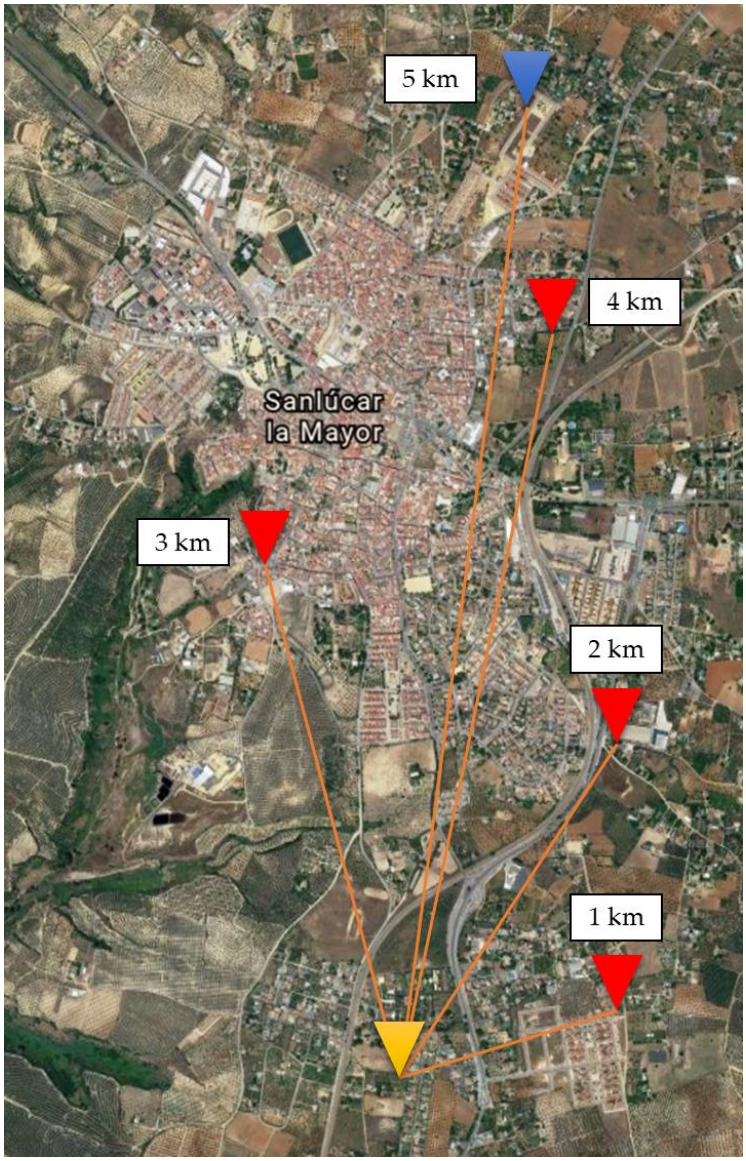


Figura 16: Mapa de experimentos en zona Urbana

Los diferentes resultados obtenidos según la potencia de transmisión, son los siguientes:

7.1.1.1 Potencia TX = 2 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
0.1 km	8	7 dB	-90 dBm
	10	7 dB	-93 dBm
	12	9 dB	-95 dBm
1 km	8	- 9 dB	-120 dBm
	10	-7 dB	-122 dBm
	12	-7 dB	-121 dBm
2 km	8	-	-
	10	-	-
	12	-17 dB	-132 dBm

7.1.1.2 Potencia TX = 8 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
0.1 km	8	7 dB	-85 dBm
	10	7 dB	-92 dBm
	12	8 dB	-86 dBm
1 km	8	-2 dB	-112 dBm
	10	-2 dB	-115 dBm
	12	-2 dB	-117 dBm
2 km	8	-	-
	10	-13 dB	-128 dBm
	12	-13 dB	-127 dBm
3 km	8	-	-
	10	-	-
	12	-18 dB	-132 dBm

7.1.1.3 Potencia TX = 14 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
0.1 km	8	6 dB	-81 dBm
	10	7 dB	-79 dBm
	12	9 dB	-82 dBm
1 km	8	1 dB	-108 dBm
	10	0 dB	-108 dBm
	12	1 dB	-112 dBm
2 km	8	-8 dB	-121 dBm
	10	-10 dB	-124 dBm
	12	-7 dB	-121 dBm
3 km	8	-13 dB	-128 dBm
	10	-13 dB	-125 dBm
	12	-10 dB	-125 dBm
4 km	8	-	-
	10	-14 dB	-120 dBm
	12	-12 dB	-123 dBm

7.1.1.4 Potencia TX = 20 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
0.1 km	8	5 dB	-77 dBm
	10	7 dB	-72 dBm
	12	8 dB	-80 dBm
1 km	8	3 dB	-106 dBm
	10	4 dB	-105 dBm
	12	3 dB	-110 dBm
2 km	8	-10 dB	-123 dBm
	10	-8 dB	-118 dBm
	12	-3 dB	-117 dBm
3 km	8	-8 dB	-120 dBm
	10	-12 dB	-121 dBm
	12	-11 dB	-124 dBm
4 km	8	-	-
	10	-12 dB	-121 dBm
	12	-10 dB	-122 dBm
5 km	8	-	-
	10	-	-
	12	-17 dB	-128 dBm

7.1.2 Zona Rural

En este caso, los puntos de conexión son bastante más alejados del emisor, ya que al ser un medio mucho más libre de interferencias y obstáculos, la señal llega más lejos. El alcance máximo en el que se han recibido mensajes LoRa es de 15 km.

Figura 17: Mapa de experimentos en zona Rural.

Los diferentes resultados obtenidos según la potencia de transmisión, son los siguientes:

7.1.2.1 Potencia TX = 2 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
6 km	8	-	-
	10	-	-
	12	-13 dB	-129 dBm
8 km	8	-	-
	10	-	-
	12	-17 dB	-127 dBm

7.1.2.2 Potencia TX = 8 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
6 km	8	-	-
	10	-13 dB	-125 dBm
	12	-12 dB	-122 dBm
8 km	8	-	-
	10	-12 dB	-124 dBm
	12	-10 dB	-124 dBm
11 km	8	-	-
	10	-11 dB	-121 dBm
	12	-10 dB	-121 dBm

7.1.2.3 Potencia TX = 14 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
6 km	8	-7 dB	-117 dBm
	10	-8 dB	-114 dBm
	12	-6 dB	-118 dBm
8 km	8	-8 dB	-120 dBm
	10	-8 dB	-119 dBm
	12	-2 dB	-116 dBm
11 km	8	-6 dB	-115 dBm
	10	-7 dB	-115 dBm
	12	-5 dB	-117 dBm

7.1.2.4 Potencia TX = 20 dBm

Distancia	Spreading Factor	SNR	Intensidad de señal
6 km	8	-4 dB	-113 dBm
	10	-3 dB	-115 dBm
	12	0 dB	-113 dBm
8 km	8	-4 dB	-113 dBm
	10	-2 dB	-112 dBm
	12	-2 dB	-115 dBm
11 km	8	-5 dB	-122 dBm
	10	-4 dB	-123 dBm
	12	-3 dB	-120 dBm
15 km	8	-	-
	10	-7 dB	-127 dBm
	12	-6 dB	-128 dBm

8 CONCLUSIÓN

El conocimiento de los hombres no va más allá de su experiencia.

- John Locke -

Cómo hemos podido analizar durante todo el proyecto de fin de grado, el microcontrolador LoPy4, gracias a los múltiples periféricos y las opciones de conectividad que nos brinda, es un dispositivo ideal para implementar aplicaciones IoT. Por ejemplo, se le podría conectar algún sensor de humo, o alguna cámara programada para detectar fuego, e instalarlo en zonas estratégicas vigilando que no se provoque ningún incendio en el lugar deseado. En caso que el sensor detecte fuego, envía un mensaje vía LoRaWAN a otro dispositivo receptor conectado a internet, que da la voz de alarma a los bomberos. Esta aplicación tiene una cobertura de muchos kilómetros si se trata de zona rural, prácticamente no requiere mantenimiento y su coste es ínfimo (50 €).

Tal y como se ha visto durante el proyecto, nos hemos centrado en el uso de la tecnología LoRa. Se ha tratado de medir cuán valioso es el microcontrolador LoPy4 para generar este tipo de conexiones. Es por ello, y según los resultados obtenidos en los experimentos realizados en la sección anterior, se puede llegar a varias conclusiones:

1. LoRa funciona mucho mejor en zonas rurales. Es capaz de llegar a un alcance muy superior al de las zonas urbanas. Esto quiere decir, que es muy sensible en cuanto a las interferencias y el ruido electromagnético urbano, ya que como se puede apreciar, en zona urbana, el SNR medio está en torno a los -12 dB, mientras que, en zona rural, este se establece en aproximadamente -6 dB.
2. Aumentar el Spreading Factor, tal y como se introdujo en la sección de LoRa, hace que se envíe una señal más robusta, capaz de obtener un mayor alcance a pesar de mantener una conexión más lenta. Esto se puede apreciar muy bien en los experimentos, pues en todos los casos, con SF = 12, se llega a más longitud sea cual sea la potencia de la antena. Con SF = 8, no siempre se conseguía recibir el mensaje.
3. Para conseguir una mejor experiencia con los mensajes LoRa, es conveniente aumentar el SF, así como la potencia de la antena. Esto hará que se envíen mensajes mucho más robustos y potentes, más detectables por el nodo receptor. Sin embargo, lo ideal sería ajustar según el tipo de aplicación, una combinación de SF + potencia de transmisión óptimas, para reducir el consumo del dispositivo LoPy4, ya que muchas veces el nodo transmisor se encuentra conectado a una batería, que se desea mantener con carga el mayor tiempo posible.

Con las conclusiones previamente mostradas, y todo lo ya comentado respecto al dispositivo, se puede afirmar que, sin duda alguna, Pycom da la talla como fabricante de dispositivos IoT, ya que LoPy4 tiene todo lo que necesita un microcontrolador destinado a este tipo de aplicaciones. Monitorización, conexión y actuación, los 3 pilares necesarios que el dispositivo cubre con total comodidad, sin mencionar lo sencilla que es su programación.

REFERENCIAS

[1] Documentación sobre LoPy4 de Pycom [En línea]

https://docs.pycom.io/

[2] LoPy4 Datasheet. [En líena]

https://docs.pycom.io/gitbook/assets/specsheets/Pycom_002_Specsheets_LoPy4_v2.pdf

[3] Documentación de Micropython [En línea]

http://docs.micropython.org/

[4] Seneviratne, Pradeeka. Begining LoRa Radio Networks with Arduino. Build Long Range, Low Power Wireless IoT Networks. 1st Edition.

[5] Documentación sobre LoRa [En línea]

https://botrueactivities.com/comparativa-entre-sigfox-y-lorawan/

http://www.techplayon.com/lora-long-range-network-architecture-protocol-architecture-and-frame-formats/

https://www.seeedstudio.com/blog/2020/05/08/lora-and-lorawan-what-is-the-difference-and-how-to-apply-lora-and-lorawan-into-applications/

https://es.wikipedia.org/wiki/LoRaWAN

ÍNDICE DE CONCEPTOS

GLOSARIO

1. IoT – Internet of Things
2. M2M – Machine to Machine
3. LoRa – Long Range
4. LoRaWAN – Long Range Wide Area Network
5. LPWAN – Low Power Wide Area Network
6. VSAT – Very Small Aperture Terminal
7. BLE – Bluetooth Low Energy
8. LTE – Long Term Evolution
9. ISM – Industrial, Scientific and Medical band
10. CSS – Chirp Spread Spectrum
11. SF – Spreading Factor
12. ADR – Adaptative Data Rate
13. CRC - Cyclic Redundancy Check
14. IP – Internet Protocol
15. CoAP – Constrained Application Protocol

ANEXO A: CÓDIGO DEL SERVIDOR COAP

Se ha desarrollado un ejemplo de servidor CoAP en un PC para comprobar si funciona correctamente las funciones de dicho protocolo en el dispositivo LoPy4. Para ello, se ha instalado el framework de JavaScript, Node.js y npm para poder ejecutar las dependencias necesarias de dicho servidor. Para ello, los pasos a seguir son:

1. Instalación de Node.js

https://nodejs.org/es/

2. Creación del repositorio con las dependencias del paquete CoAP de npm. Para ello, abrimos el terminal en el directorio deseado y escribimos el siguiente comando:

```
npm install coap –save
```

Esto instalará todos los paquetes y dependencias necesarias para la creación del servidor coap.

3. Escritura del simple script que se muestra a continuación.

```
const coap  = require('coap')

coap.createServer(function(req, res) {
  res.end('Mensaje COAP recibido')
  console.log('MENSAJE RECIBIDO: ' +req.payload+ '\n')
}).listen(function() {
  console.log('server started')
})
```

4. Ejecutamos el script escribiendo en el terminal `npm start` en el directorio seleccionado. Esto creará el servidor en el puerto predeterminado 5683.