Universidad de Valladolid

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Módulo inalámbrico de señalización luminosa en vehículos de emergencias

Autor:

Jesús Martín Pelayo

Tutor:

Jose Manuel Mena Rodríguez

Dpto. de Tecnología Electrónica

Valladolid, Octubre 2019

Resumen :

Los problemas actuales de las señales luminosas en vehículos de emergencia derivan de la comunicación entre el panel de señalización y el elemento de mando. Por ello, se ha desarrollado un sistema con el fin de solventarlos, cambiando completamente las características de los sistemas actuales.

Así, nos hemos basado en el chip ESP32, un microcontrolador que ofrece conectividad Wi-Fi, bluetooth y BLE, para evitar las comunicaciones cableadas entre el elemento de mando y el de señalización.

Usando el módulo ESP-WROOM-32, se ha desarrollado un módulo con comunicación BLE, capaz de hacer las veces de servidor del mismo y de trabajar como esclavo con el fin de que un smartphone, trabajando como maestro, ofrezca al usuario mayor control sobre la señal luminosa.

Por lo tanto, se ha desarrollado hardware receptor, software para el ESP32 y una app para el smartphone, creando un sistema completo interconectado mediante BLE.

The current problems of the light signals in emergency vehicles derive from the communication between the signaling panel and the control element. Therefore, a system has been developed in order to solve them, completely changing the characteristics of current systems.

Thus, we have relied on the ESP32 chip, a microcontroller that offers Wi-Fi, Bluetooth and BLE connectivity, to avoid wired communications between the master element and the signaling element.

Using the ESP-WROOM-32 module, a module with BLE communication has been developed, capable of serving as a server and working as a slave so that a smartphone, working as a master, offers the user greater control over the light signal.

Therefore, receiver hardware, software for the ESP32 and an app for the smartphone have been developed, creating a complete system interconnected by BLE.

Keywords:

- Señalización
- Inalámbrica
- Controlador
- BLE (Bluetooth Low Energy)
- Android

Índice del contenido

Índice de Ilustraciones

Índice de Ilustraciones

Acrónimos usados:

ATT	Attribute Protocol
BLE	Bluetooth Low Energy
CS	Chip Select(ed)
FHSS	Frequency Hopping Spread Spectrum
GAP	Generic Access Profile
GATT	General Attribute Profile
GPIO	General Purpose Input/Output
IDE	Integrated Development Enviorement
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
MTU	Maximum Transmission Unit
PDU	Protocol Data Unit
SCK	Serial Clock
SIG	Special Interest Group
UART	Universal Asynchronous Receiver Transmitter
UUID	Universally Unique Identifier

1 Introducción y estructura del proyecto

1.1 Introducción y justificación:

En la carretera hay situaciones excepcionales en las cuales intervienen diversos vehículos especiales con el fin de indicar al resto de usuarios que la situación requiere de un nivel de atención mayor del normal.

Ejemplos de estas situaciones son la reparación de las vías, en las que interviene personal y maquinaria que muchas veces requiere para su trabajo "cortar" uno o varios carriles, e incluso habilitar temporalmente el uso de carriles en sentido contrario al ordinario para la vía.

Otra situación particular es el traslado de maquinaria agrícola o "piezas" de un tamaño importante, provocando en ambos casos la invasión de parte del carril adyacente al que se usa para la circulación.

En ambos ejemplos, se hace uso de *vehículos especiales*, de modo que se pueda advertir al resto de conductores, a través de señales luminosas y/o sonoras, de la presencia de estas particularidades con suficiente antelación para adecuarse a ellas y así evitar posibles accidentes.

Esto, requiere del diseño de medios de comunicación con el resto de los vehículos, de modo que sean capaces de crear un canal para la difusión del mensaje sin comprometer la seguridad de los receptores.

Para lograrlo, hay diversas normativas que regularizan los elementos exteriores a la cabina de los vehículos de manera que su construcción y trabajo no impliquen riesgos para los usuarios de la vía bajo las condiciones de trabajo que encaran: posible viento sobre paneles, condiciones de baja luminosidad en las que se puede llegar a provocar deslumbramiento, condiciones de baja visibilidad debidas a niebla, temporales, etcétera.

Una de las dificultades, es evitar cualquier elemento similar a un cable o cuerda que pueda quedar en el exterior y, por alguna circunstancia, pueda desprenderse del vehículo y llegar a provocar problemas a otros usuarios de la vía.

Esto, es de mayor importancia en el caso de las señales luminosas y/o sonoras controladas, ya que requieren de varios hilos para enviar las señales al panel exterior desde el elemento de mando, que se encuentra en el interior de la cabina del conductor.

Por lo general, el cableado entre el *controlador* y la *señal luminosa* puede hacerse por el interior del vehículo, lo que reduce el problema a 2 posibles vías:

1. Lograr realizar el cableado por el interior de la estructura del vehículo, un trabajo que requiere desmontar partes del vehículo para realizar la instalación, y que a veces condiciona dónde colocar la señal luminosa, debido al punto hasta el que cablearse internamente.
2. Realizar un cableado por dentro de la cabina, tras estudiar la forma en que realizar este tipo de instalación no comprometa la seguridad de aquellos que estén en el vehículo en caso de accidente, y en caso de ser posible, lograr una solución con cierta estética.

Estos métodos, aunque válidos, tienen una mejor solución en la actualidad gracias a tecnologías que han avanzado en los últimos años, como es el caso del Bluetooth con su estándar v4.0: el Bluetooth de baja energía (BLE desde sus siglas en inglés).

Gracias a la tecnología Bluetooth, podemos reducir el cableado a tan solo 1 hilo, alimentación (ya que el chasis del propio vehículo está conectado a tierra), pues el resto de las señales de control son transmitidas mediante radiofrecuencia.

1.2 Definiciones previas:

Dado que se va a hacer uso de ciertos términos durante todo el proyecto y con el fin de facilitar el seguimiento del mismo, vamos a fijar ahora a qué hacemos referencia cuando usamos cada uno de ellos.

- Controlador: usaremos este término para hacer referencia al módulo receptor desarrollado y al que se envían los ciclos de trabajo desde la app Android del smartphone.
- Estado: llamamos estado a cómo se encuentra cada uno de los 8 led de la señal luminosa.
- Ciclo de trabajo: conjunto de estados que se repiten de forma cíclica

1.3 Objetivos:

El objetivo de este proyecto es, por tanto, crear un sistema controlador para un panel luminoso, minimizando el cableado y en el cual el elemento de control pueda ser independiente del elemento receptor conectado al panel.

Esto requiere del desarrollo de 2 partes:

1. Hardware controlador: se desarrolla el hardware que dará soporte al elemento receptor del controlador, al cual enviaremos los datos a reproducir en el panel luminoso.

2. Software controlador: este, a su vez, se divide en 2 partes:
   a. *Software ESP32*: software sobre la placa Devkit v1 con el fin de que pueda trabajar independientemente del elemento emisor y, a disposición del mismo, junto con él durante la emisión/recepción de datos.
   b. *Software Smartphone:* ya que se usará un smartphone, al alcance de cualquier posible usuario del proyecto, como elemento emisor (*maestro*) del mensaje a transmitir mediante la señal luminosa, y receptor del estado actual de la misma desde los datos expuestos por el ESP32.

Así mismo, otro de los objetivos es la independencia del hardware que hace las veces de dispositivo maestro, es decir, el smartphone, pues cualquiera que el futuro usuario de este proyecto tenga, debe poder hacer uso de la app, de modo que esto le permita una comunicación correcta con el hardware receptor.

1.4 Estado del arte:

En la actualidad, la gran mayoría de controladores de paneles luminosos externos preparados para montaje en vehículos, se componen de 2 elementos: una centralita con botonera, desde la cual poder elegir los estados de los led del panel luminoso, y un cableado para las señales entre la botonera y el propio panel luminoso, como se ve en la *Ilustración 1.*

Ilustración 1: Centralita y detalle de sus conexiones

Estos equipos, además de tener unos precios elevados, que oscilan desde los 50€ hasta los 245€, requieren una *posición fija* en el salpicadero u otra parte de la cabina del conductor, ocupando un espacio importante y que debe estar al alcance del usuario.

Por otro lado, presentan limitaciones en cuanto a la programación de los paneles led que se van a controlar, puesto que lo más común es encontrar que estas centralitas tienen predefinidos un grupo fijo de "ciclos de trabajo" o estados para los diferentes leds, que el usuario *no* puede modificar. Así mismo, es raro encontrar centralitas con pantallas que permitan visualizar más de 1 solo estado del panel luminoso por vez, o poder controlar con precisión el tiempo que permanece en cada estado dicho panel (en el caso de aquellas que sí nos permiten modificar este parámetro).

Por último, otro de los problemas es la dificultad de testear que se corresponde la señalización exterior con lo que los elementos de visualización nos muestran en la centralita, ya que al estar esta fija dentro de la cabina, no permite visualizar a la vez los led de la misma y el panel luminoso exterior.

Ilustración 2: Centralita instalada en salpicadero

Todo esto, nos lleva a buscar soluciones a las diversas limitaciones que se ven en los dispositivos actuales del mercado, fácilmente solucionables a través de una conexión inalámbrica y un software que permita al usuario reprogramar a su antojo los estados de los leds en cada momento.

Por otro lado, se pueden ver grandes diferencias entre los paneles luminosos existentes en el mercado, tanto en cuanto a cantidad de módulos led, como voltajes de funcionamiento, intensidad lumínica, distancia efectiva de visualización, forma… sin embargo, la gran mayoría busca un elemento base: alta luminosidad con el menor consumo posible.

Ilustración 3: Barra led direccional de 4 módulos de 16w cada uno

Ilustración 4: Barra estilo "eurotipe", 9w por cada módulo. Visible a 1000m

Ilustración 5: Barra led recargable con autonomía de 55h

Así, la cantidad de opciones en mercado ofrece una amplia versatilidad según las necesidades que el cliente pueda tener, sin embargo, los precios de estas son elevados, lo que nos ha llevado a la decisión de crear nuestro propio módulo a modo de prototipo, con el fin de validar el trabajo relativo al controlador, y a su vez tener un equipo con que realizar pruebas para estudiar la viabilidad de crear un módulo luminoso propio más adelante.

1.5 La señal luminosa:

Se ha montado, en conjunto con el resto del sistema, un módulo luminoso conformado por un panel con 8 bloques de tiras led para la señalización de los estados del ciclo de trabajo.

Así, dado que esta señal luminosa se ha desarrollado completamente desde cero, vamos a repasar el trabajo realizado para su consecución, pasando desde los cálculos y las simulaciones, hasta las especificaciones finales de la misma.

Primeramente, vamos a diferenciar entre led, tira led y módulo led, de modo que:

- Led: led individual dentro del conjunto del montaje.
- Tira led: unidad indivisible de 3 leds, dispuestos tal y como se envían para montaje, en serie.
- Módulo led: conjunto formado por 3 tiras_led, en paralelo.

Por lo tanto, utilizando los datos que nos da el fabricante de las tiras led…

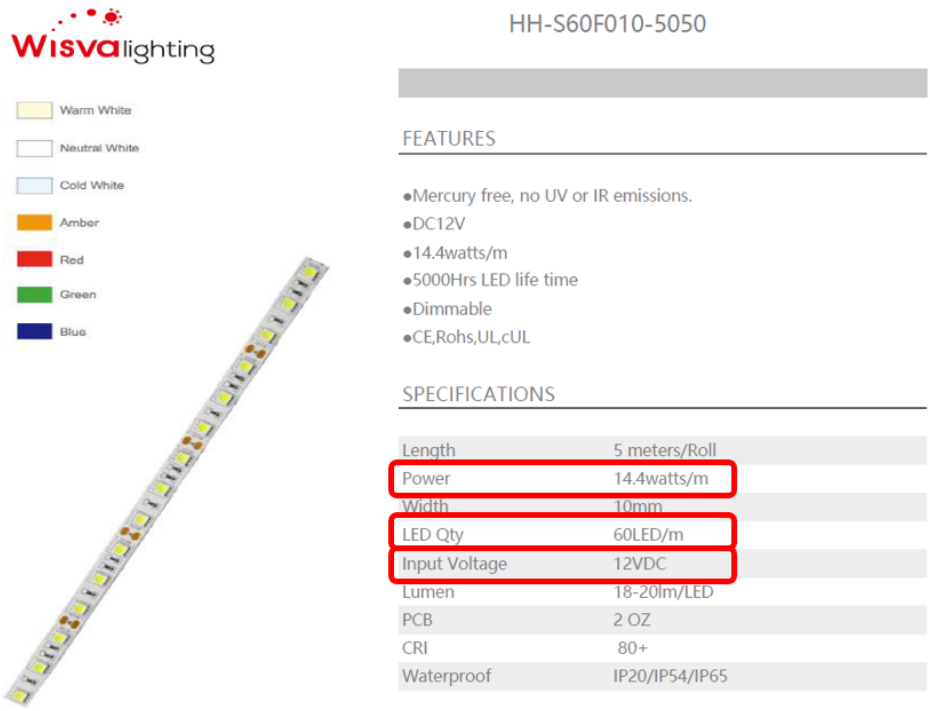


Ilustración 6: Datos del fabricante de las tiras led empleadas

El panel estará formado por 8 “módulos led” cada uno de los cuales estará constituido por 3 tiras en paralelo de leds en serie, de modo que cada uno de los módulos constará de 9 leds.

Utilizando los datos del datasheet, vemos que:

- Vin = 12 (v)
- P = 14’4 (w/m)
- Leds = 60 (led/m)
- Leds = 18-20 (lumen/led)

De este modo, teniendo en cuenta la forma de construcción que se prevé para el panel controlado, tenemos lo siguiente:

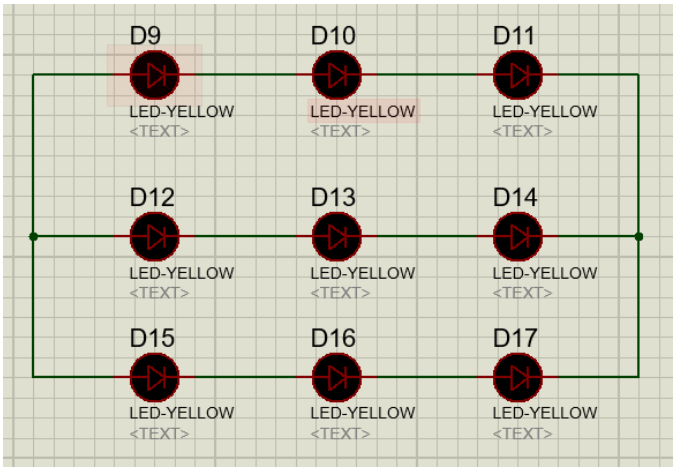


Ilustración 7: Esquema de los módulos led simulados

Utilizando los datos, teniendo en cuenta que la tensión será de 12 (v) (Vin) en el ánodo del diodo (parte izquierda común de la figura), y 0 (v) en su cátodo (parte derecha común de la figura), común a las 3 filas de leds, calculamos.

Podemos realizar los siguientes cálculos:

$$P_{led} = \frac{P_{metro}}{LED_{metro}} \rightarrow P_{led} = \frac{14.4}{60} \rightarrow P_{led} = 0.24(w)$$

$$P_{tira\_led} = 3 \cdot Leds \cdot P_{led} \rightarrow P_{tira\_led} = 3 \cdot 0.24 \rightarrow P_{tira\_led} = 0.72(w)$$

$$P_{modulo} = 3 \cdot P_{tira\_led} \rightarrow P_{modulo} = 3 \cdot 0.72 \rightarrow P_{modulo} = 2.16(w)$$

$$R_{carga\_transistor} = \frac{V^2}{P_{modulo}} \rightarrow R_{carga\_transistor} = \frac{12^2}{2.16} \rightarrow R_{carga\_transistor} = 66.667\ (\Omega)$$

$$I_{carga} = \frac{P_{modulo}}{V_{alimentacion}} \rightarrow I_{carga} = \frac{2.16}{12} \rightarrow I_{carga} = 180\ (mA)$$

Por lo tanto, teniendo en cuenta que cuanto mayor sea la corriente de colector que se le pide a un transistor, más difícilmente entra en saturación, vamos a

utilizar una impedancia óhmica de 60 (Ω) como carga de estos por cada módulo.

Por lo tanto, desde lo anterior, podemos calcular cual va a ser la Ic (usaremos transistores controlados desde el voltaje y la corriente en las patillas de salida de la EEPROM) necesaria como mínimo en cada polo de la etapa de potencia, de manera conservadora para con el componente al mayorar su valor a través del uso de la resistencia de carga calculada:

$$I_{carga} = \sqrt{P_{modulo}/R_{carga}} \rightarrow I_{carga} = \sqrt{2.16/60} \rightarrow I_{carga} = 189.737\ (mA)$$

Se han realizado mediciones con tiras similares de leds en otro color, pues el fabricante no especifica si en los 14,4 (w/m) y 60 (leds/m) se tienen en cuenta las resistencias de polarización de cada led, de modo que las mediciones a 12(v), nos dan un consumo de cada módulo led de 162(mA), lo que hace que cada uno tenga una carga óhmica aproximada de 74.074 (Ω), que es un valor menos conservador de cara a los cálculos de la IC que va a requerirse.

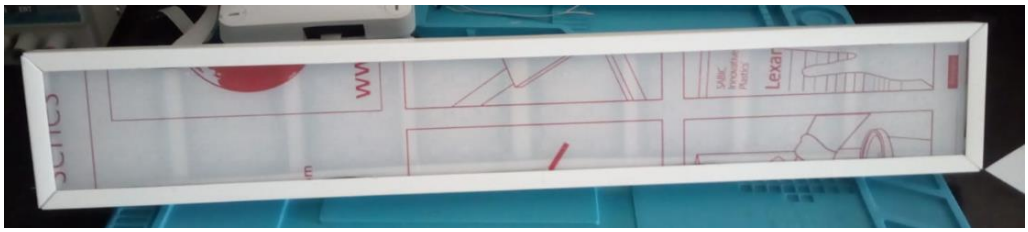


Ilustración 8: Barra led implementada

Nuestra señal luminosa requiere para su funcionamiento las siguientes condiciones:

- ·Voltaje: 12 (v)
- ·Corriente : 1.3(A)
- ·Consumo: 15.6(W)

Por otro lado, sus dimensiones son 540x84x19 mm

1.6 Estructura de la memoria:

- ### Capítulo 1: Introducción y estructura del proyecto

  En este primer apartado, se da la visión que lleva a la idea de desarrollar este proyecto, así como la justificación del mismo, los objetivos buscados en principio e información del mercado actual de controladores luminosos. Por otro lado, se incluye un apartado en el cual se da una breve descripción de los diferentes temas desarrollados en este documento.

- ### Capítulo 2: BLE: Bluetooth Low Energy

  En esta parte del documento, hacemos hincapié en la tecnología que es base de la solución ideada. Se definen tanto el protocolo como sus principales atributos y características en uso.

- ### Capítulo 3: ESP32

  Capítulo centrado en el corazón de nuestro proyecto: el chip ESP32, centrándonos la variante usada, el WROOM-32. Este módulo incluye en sí memoria flash, 2 núcleos, antena de recepción, Bluetooth, Wi-Fi… de modo que explicaremos brevemente sus capacidades y uso en este proyecto.

- ### Capítulo 4: Hardware del controlador

  En esta parte del documento se explican los diferentes componentes que se usan en el controlador, junto a una justificación de cada uno de ellos.

- ### Capítulo 5: Software del controlador

  En este capítulo se desarrollan y justifican el software desarrollado tanto para la placa ESP32 Devkit v1, como de la app Android para lograr los objetivos fijados previamente.

- ### Capítulo 6: Conclusiones y líneas futuras de trabajo

  Capítulo en que se hace un repaso de los objetivos propuestos en inicio y el grado en que se ha logrado cada uno de ellos, así como un nuevo análisis de cómo podría mejorarse y ampliar las capacidades del proyecto desde su estado actual. También se hace hincapié en otros

usos de la placa desarrollada, así como las repercusiones de la misma en el ámbito de la seguridad vial.

- ## Capítulo 7: Bibliografía

  En este apartado, se incluye la bibliografía utilizada en el desarrollo de este proyecto.

- ## Anexos

  Documentación base de los elementos hardware utilizados, así como el código de todo el software del proyecto y los planos de la placa desarrollada, incluyendo costes aproximados de la misma por unidad y tipo.

2 Bluetooth y Bluetooth Low Energy

En este capítulo, vamos a desarrollar brevemente los por qué de la creación de la tecnología bluetooth y cómo ha evolucionado esta hasta la especificación BLE, que es la base de todo nuestro sistema.

Para ello, primero hemos de tener en mente cómo son las diferentes topologías de red existentes que podemos implementar, así como las diferentes formas de comunicación que podemos usar. Esto nos ayudará a comprender las bases en que se apoya.

2.1 Topologías compatibles con Bluetooth:

Dado que la tecnología bluetooth está dentro del campo de las comunicaciones inalámbricas orientadas a WPAN, con el fin de proveer una comunicación dinámica y sencilla con capacidad de soportar transporte de datos y audio aprovechando la banda libre, se fijaron ciertos aspectos, como su base en una metodología maestro/esclavo para gestionar la comunicación.

Así, cada interfaz de radio usa una dirección propia y única de 48bits, y permite unir un maestro con hasta 7 esclavos a la par, creándose así una red ad-hoc que recibe el nombre de *piconet* . Este tipo de redes puede extenderse a través de *scatternets*, que son redes producidas por dispositivos que se conectan entre sí a pesar de que pertenecen a 2 piconets distintas.

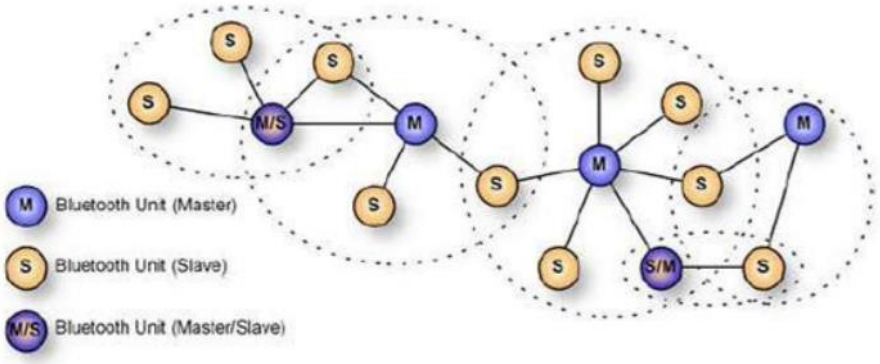


Ilustración 9:Ejemplo piconet y scattrernet

La comunicación, basada en la metodología maestro/esclavo, obliga a que sea el primero quien deba iniciar siempre los enlaces, pero una vez establecido el mismo. Hemos de tener en cuenta que la comunicación entre esclavos no existe, ya que solo un maestro es capaz de crear la señal de reloj con que todos

los dispositivos de la piconet se sincronizan y la información para realizar los saltos de frecuencia.

Puede observarse así que las piconets provocan una estructura de funcionamiento *punto a punto*, controlada por el maestro, que por su parte puede trabajar bajo una estructura multipunto entre varias piconets (es decir, en una scatternet).

Por otro lado, un esclavo en una scatternet puede funcionar como tal en varias picoredes diferentes, pero solo puede ser maestro en 1 de ellas.

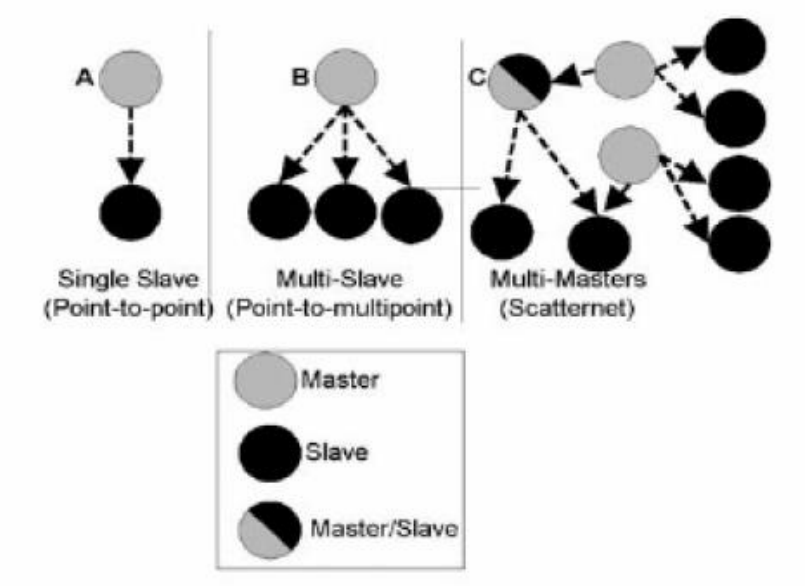


Ilustración 10: Modelos de comunicación entre piconets y scatternets

Podemos tener una visión generalizada de qué ofrece cada una de estas tipologías a través de la siguiente tabla, realizada a partir de los datos expuestos en la web oficial de bluetooth:

	BLE	Bluetooth clásico
Punto a Punto (comunicación 1 a 1)		
Tiempo de mantenimiento	< 6 ms	100 ms
N.º máx. de conexiones	ilimitadas (teóricamente)	7
Velocidad de datos	hasta 2Mb/s	hasta 3 Mb/s
Máxima cantidad de datos	251 byte	1021 byte

Broadcast (comunicación 1 a m)		
Máxima cantidad de datos	Canal primario: 31 byte Canal Secundario: 255 byte * Pueden encadenarse paquetes para mensajes mayores	NO APLICABLE
Seguridad	Formato Beacons	
Mesh (comunicación n a m)		
Nodos máximos	32767	NO APLICABLE
Subnets máximas	4096	
Máxima cantidad de datos	29 byte	

Tabla 1: Topologías de comunicación en Bluetooth

2.2 Modos de comunicación mediante Bluetooth:

Los dispositivos bluetooth pueden comunicarse mediante *broadcasting* (transmisión) o *connections* (conexión). Cada una de estas formas tiene sus ventajas y limitaciones, de modo que vamos a ver brevemente ambas:

- Broadcasting: permite enviar datos a cualquier dispositivo de escaneo, o a un receptor cualquiera en el rango de escucha.

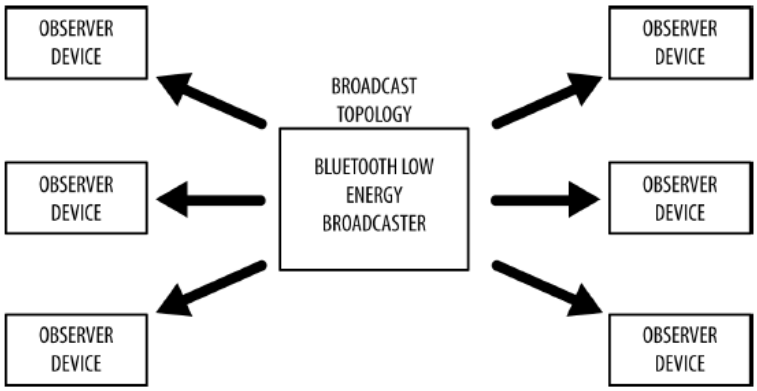


Ilustración 11: Topología broadcasting

De este modo, se definen 2 funciones: emisión y escucha, según se emitan/reciban datos. El emisor envía paquetes de datos de forma periódica a toda la red con capacidad de recibirlos, mientras que el

receptor hace escuchas a la misma frecuencia de emisión. Las grandes trabas de esta topología es que carece de seguridad en cuanto a la privacidad al no limitarse quienes reciben los datos, y la unidireccionalidad de los datos (salvo que se establezcan períodos de emisión y escucha entre los dispositivos).

- Connections: permite la comunicación de datos en ambos sentidos, a diferencia del broadcasting. Las conexiones permiten el intercambio de datos periódicamente entre 2 dispositivos de manera privada. En esta tipología hay 2 roles:
    - Maestro: busca periódicamente invitaciones de conexión, eligiendo cuando se conecta a la que desea. Establecida la conexión, establece la forma de comunicación: la frecuencia de trabajo (sincronía), tamaños de los paquetes de datos…
    - Esclavo: dispositivo que envía invitaciones de conexión periódicas y recibe conexiones como respuesta a ellas. Deja de enviar estas invitaciones una vez tiene establecida una conexión.

  Establecida la conexión, los dispositivos pueden comunicarse estableciendo un flujo de datos bidireccional.

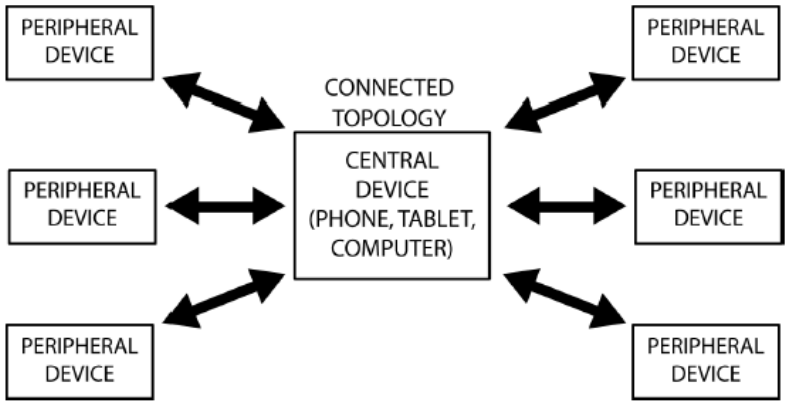


Ilustración 12: Topología Connections

Si atendemos a estos roles, vemos que inicialmente tenía restricciones en cuanto a quien iniciaba la comunicación, sin embargo, la especificación 4.1 eliminó estas, permitiendo que:

- Un dispositivo pueda actuar como dispositivo central y periférico a la vez.
- Conectar un dispositivo central a varios periféricos a la par.
- Conectar un periférico a varios dispositivos centrales.

Sin embargo, la mayor ventaja es la capacidad de controlar detalladamente los datos enviados, a través del uso de protocolos como el GATT, que organiza los datos en servicios y características, permitiendo que coexistan varios servicios diferentes en un servidor, cada uno con uncas características propias o comunes, que definen los datos que se comunican en las mismas. Por otro lado, este funcionamiento ofrece un cifrado seguro y una sincronización más eficiente entre los diferentes dispositivos, lo que resulta en un menor consumo.

Así, podemos conjugar los diferentes tipos de conexiones en una red bluetooth de muchas formas diferentes, con la única limitación de la tecnología que cada dispositivo tenga implementada:

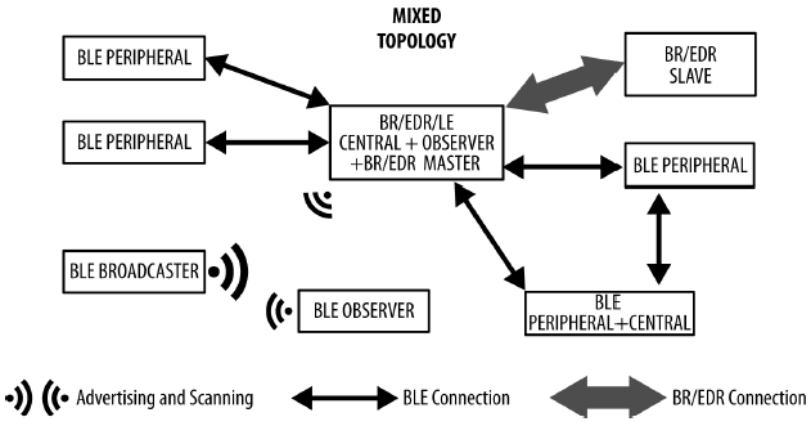


Ilustración 13: Ejemplo de comunicación Bluetooth-BLE en la red bluetooth

2.3 Qué es Bluetooth:

Bluetooth es un protocolo estandarizado para WPAN, redes inalámbricas que trabajan bajo radiotransmisión en la banda de los 2.4GHz. Regularizado por la IEEE 802.15.1, tanto en la capa hardware como en la software, modelos de uso, perfiles de usuario… está orientado al envío de pequeños paquetes datos a corta distancia.

Los dispositivos Bluetooth pueden clasificarse en varias clases, en función de la potencia de transmisión, siendo las reflejadas en la siguiente tabla:

Clase	Potencia máxima permitida		Alcance aproximado
	(mW)	(dB)	
1	100 mW	20 dBm	100 m

2	2,5 mW	4 dBm	5-10 m
3	1 mW	0dBm	1 m
4	0,5 mW	-3 dBm	0,5 m

Tabla 2: Clases de dispositivos Bluetooth

Hemos de conocer que todas las clases son compatibles entre ellas, así como interconectables, lo que ofrece ventajas como el aumento de la cobertura de una clase 2 al conectarse con una clase 1, debido a la mayor sensibilidad del dispositivo de clase 1 y su mayor potencia de transmisión.

Desde sus inicios en el 1994 y posterior estandarización por la IEEE en 2002, ha evolucionado en pos de reducir su consumo, mejorar la velocidad de transferencia, conectividad… Hasta que, en el 2010, "renace" con su versión 4.0 el estándar Bluetooth Low Energy, más conocido como BLE.

Esta evolución, ha traído consigo un aumento del ancho de banda muy a tener en cuenta:

Versión	Ancho de Banda
1.2	1Mbit/s
2.0 + EDR	3Mbit/s
3.0 + HS	24Mbit/s
4.0	24Mbit/s

Tabla 3: Evolución del tamaño de datos en el protocolo bluetooth

Para entender correctamente el funcionamiento de los dispositivos Bluetooth, hemos de tener en cuenta que la banda de los 2.4GHz, al ser una *"banda libre"* es el lugar en que conviven múltiples tecnologías (radioaficionados, ZigBee…), provocando la creación de interferencias.

La solución adoptada para afrontar este problema fue ensanchar el espectro mediante la técnica de "espectro disperso", para que la distorsión no afecte a frecuencias dominantes al haber ampliado el rango de las mismas. Esto, de manera conjunta con el *"salto de frecuencias"* (cambio de la frecuencia de

trabajo de forma pseudoaleatoria, a razón de hasta 1600 saltos/s), conforman la solución final.

Se hace uso así de un total de 79 frecuencias, a intervalos de 1MHz para aportar seguridad y robustez, lo que supone una ocupación real desde los 2.402 GHz hasta los 2.480 GHz en el espectro ISM.

2.4 Arquitectura Bluetooth:

La tecnología Bluetooth se basa en una arquitectura orientada a la comunicación entre las capas *radio* de 2 dispositivos diferentes, ambos con especificaciones simétricas para facilitar el intercambio de datos entre ambos.

Esto requiere un hardware específico, siendo mínima la presencia de los módulos siguientes:

- Banda Base: hace referencia a la banda de frecuencias creadas por un transductor (u otro generador de señales) de modo que no es necesario adaptar sin que sea necesario adaptarlas al medio de transmisión. En Bluetooth, la banda base combina la conmutación de circuitos y paquetes, asegurando el orden de estos últimos.
- Modulo Radio: encargado de modular y transmitir la señal.
- Modulo antena: usado para aumentar el rango de emisión y la sensibilidad del sistema en la banda de frecuencias.

Según la web oficial de bluetooth, regulada por el SIG, en la actualidad el bluetooth clásico sólo soporta topologías de red del tipo *punto a punto*, inclusive redes *piconet*.

2.5 Pila de protocolo Bluetooth:

Es la calve de esta tecnología, pues describe el lenguaje común de todos los dispositivos que quieran hacer uso de este método de comunicación, para permitir que todos logren entenderse.

Se diseñó buscando una máxima reutilización de los protocolos existentes, así como que la posibilidad de aplicar el Bluetooth a dispositivos que ya estaban en mercado.

En la pila, los datos fluyen a su través, con la salvedad de la información de audio, que es enviada directamente desde la banda base hacia la aplicación con alto grado de prioridad para garantizar el tiempo real.

Esta pila o stack, está formada por varias capas, divisibles en:

- Grupo de transporte
- Grupo de protocolos middleware
- Grupo de aplicación

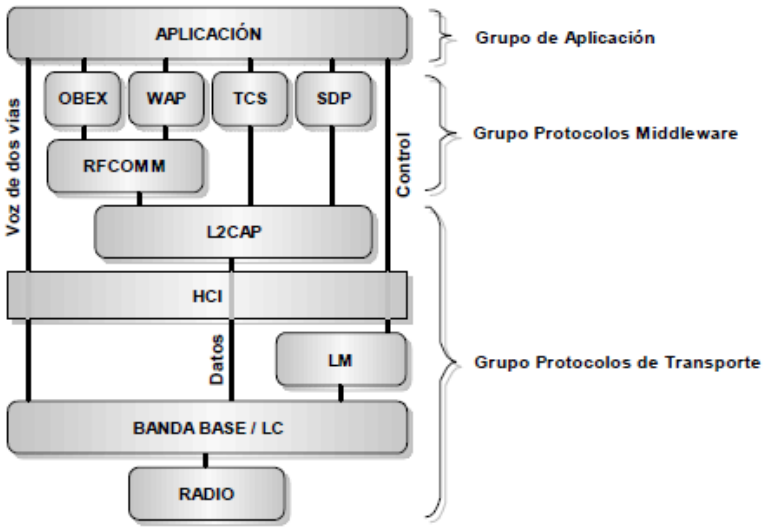


Ilustración 14: Pila del protocolo Bluetooth

Todos estos protocolos están orientados al reemplazo del cableado, control de la transmisión… pero hemos de tener en cuenta la clave: todo protocolo bluetooth implementa, como mínimo, los protocolos LMP, L2CAP y SPD, aunque también es muy habitual encontrar en la mayoría de dispositivos bluetooth los protocolos HCI y RFCOMM. Por ello, vamos a hacer hincapié en todos estos protocolos:

- LMP: Link Manager Protocol. Se usa para establecer y controlar el enlace de radio entre 2 dispositivos. Se implementa en el controlador.

- L2CAP: Logical Link Control and Adaptation Protocol. Su implementación permite multiplexar varias conexiones lógicas entre 2 dispositivos que usan protocolos diferentes y de niveles superiores. Así, también se encarga de la segmentación y reensamblado de los paquetes de datos, pudiéndose configurar el tamaño de estos.
    - En su modo básico, usa una MTU fija de 672bytes y ofrece un tamaño de paquete configurable de hasta 64Kb.

- La especificación Bluetooth añade 2 modos adicionales a este protocolo, que dejan obsoletos los modos de *"retransmisión"* y *"control de flujo"*:
  - Modo de retransmisión mejorado: el ERTM (Enhanced Retransmission Mode) mejora el modo de retransmisión ofreciendo un canal L2CAP seguro.
  - Modo streamisng: el SM (Streaming Mode) es un modo sin retransmisión ni control de flujo, no confiable.
- La confiabilidad en cualquiera de estos modos puede ser garantizada, opcionalmente, en la capa inferior BDR/EDR configurando el número de retransmisiones y el tiempo de espera antes de descartar paquetes. Esta misma capa se encarga del orden de los mismos.

- SDP: Service Discovery Protocol. Permite a un dispositivo descubrir los servicios ofrecidos por otros, asi como parámetros asociados a dichos servicios. Cada servicio es identificado mediante un UUID.

- RFCOMM: Radio Frecuency Communications. Es un protocolo de reemplazo de cable que ofrece transporte de datos binarios imitando señales de control del estándar RS-232 haciendo uso de la capa de banda. Ofrece un flujo confiable similar a TCP (Transmission Control Protocol), como si de un puerto serie se tratase.

- HCI: Host Controller Interface. Permite la comunicación estandarizada entre la pila host y el controlador. Aunque hay varios interfaces, los más comunes son USB y UART. En las comunicaciones bluetooth, suele estar implementado en el microprocesador y, aunque es opcional en ese caso, se suele prever como interfaz de comunicación interna.

Si volvemos sobre el esquema antes mostrado, las funciones de cada capa se resumen en:

- Radio: modulación y demodulación de datos para la emisión y/o recepción de los mismos por el aire.

- Banda base y LC (Link Controller): controlan los enlaces físicos vía radio, ensamblando paquetes y generando el salto de frecuencia.

- LM (Link Manager): controla y configura los enlaces entre dispositivos

- TCS: Telephone Control Service: encargado de proporcionar servicios de telefonía.

- OBEX y WAP: ofrecen protocolos a niveles superiores que los requieran.

2.6 Formato de los paquetes en Bluetooth:

Durante la comunicación, se usa un *"canal ranurado"*, donde cada ranura tiene una duración limitada de 625 µs. Por lo general, un paquete hace uso de una única ranura, pero está prevista su posible extensión hasta a 3 ó 4 ranuras.

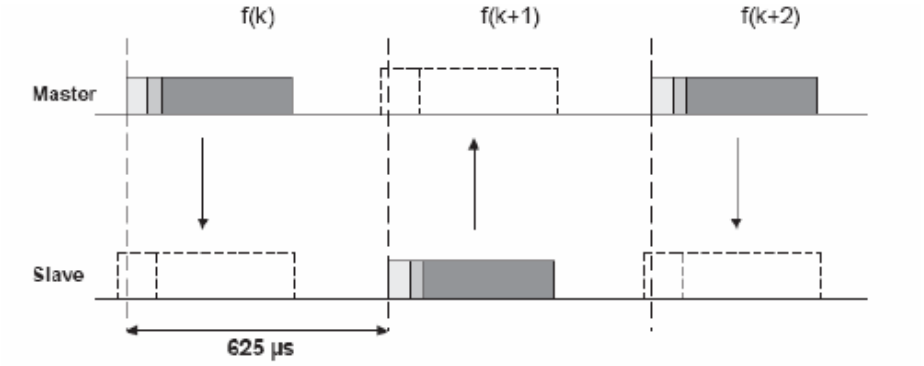


Ilustración 15: Tiempos de las tramas en el protocolo bluetooth

La figura siguiente muestra el formato general del contenido de una ranura de tiempo, trasmitida al aire en una WPAN de Bluetooth. El paquete abarca un código de acceso de tamaño fijo. La cabecera de paquete, también de tamaño fijo, que se utiliza para manejar la transmisión en una WPAN; y una carga de datos de tamaño variable, que transporta información de capas superiores. Debido al tamaño tan reducido de estos paquetes, se necesita que una capa superior más larga sea dividida en segmentos antes de que sea transmitida al aire.

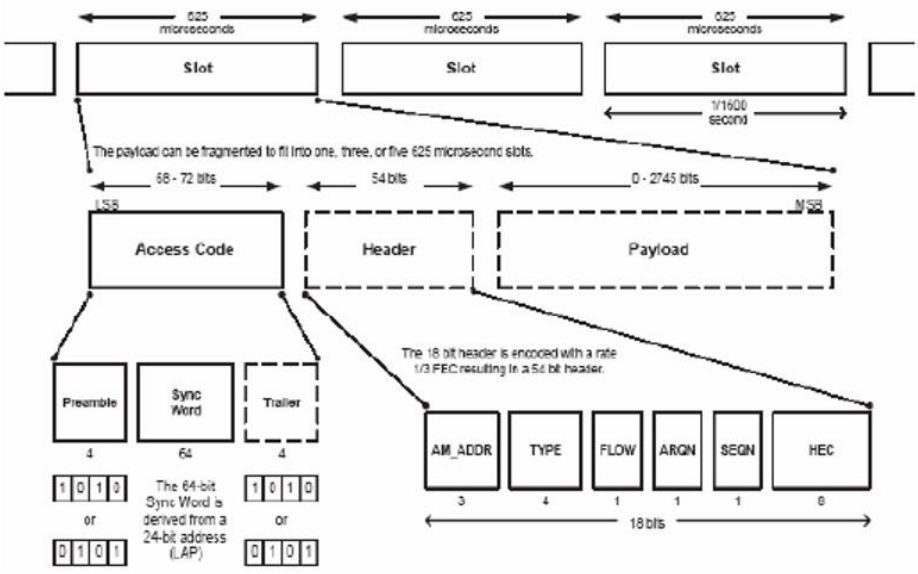


Ilustración 16: Formato de los paquetes bluetooth

Conviene profundizar un poco más en la formalización de los paquetes que se transmiten vía bluetooth, sobre todo para conocer la estructura de la cabecera:

- Código de acceso (72b): para sincronización, identificación (diferenciándose las diferentes WPAN a través de él) y comprensión
- Cabecera (54b): con información del control de enlace en 6 campos:
  - AM_ADDR: dirección temporal de 3 bits para diferenciar los dispositivos activos en la piconet. La dirección 000 es la del broadcast.
  - Tipo: define el tipo de paquete enviado y los slots que ocupa.
  - Flow (flujo): su bit de control indica cuando el buffer del receptor está lleno y debe pararse la transmisión.
  - ARQN: bit de reconocimiento de paquetes recibidos
  - SEQN: señal de 1 bit en onda cuadrada para evitar la retransmisión en receptor.
  - HEC: código de redundancia para comprobar errores en la transmisión.
- Campo de datos (hasta 2746 b): datos que contienen la información que se desea transmitir.

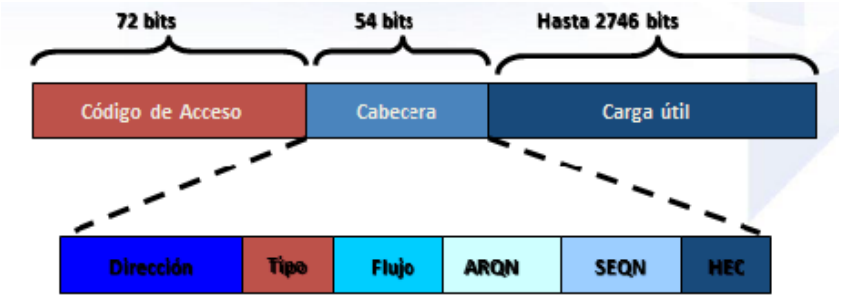


Ilustración 17: Detalle de un paquete bluetooth

2.7 Diferencias entre Bluetooth y Bluetooth Low Energy:

El BLE es la especificación 4.0 desarrollada por el SIG, complementaria al bluetooth clásico y como evolución del mismo con el fin de reducir el consumo de este mediante la reducción del tiempo de establecimiento de conexión, peor manteniendo el aprovechamiento de las bandas ISM.

La reducción del tiempo de conexión trae consigo la transmisión de paquetes de datos menores, ya que su idea base no es mantener una conexión prolongada en el tiempo para la transmisión de grandes paquetes de datos entre dispositivos.

Para verlo, vamos a partir de la siguiente tabla:

Device	BR/EDR (classic Bluetooth) support	BLE (Bluetooth Low Energy) support
Pre-4.0 Bluetooth	Yes	No
4.x Single-Mode (Bluetooth Smart)	No	Yes
4.x Dual-Mode (Bluetooth Smart Ready)	Yes	Yes

Tabla 4: Bluetooth vs. BLE

En ella, vemos como la tecnología Bluetooth posee 2 especificaciones desde su versión 4.0:

- Bluetooth Clásico: tecnología específica para dispositivos con alta demanda de pequeñas transmisiones. Existe desde la especificación 1.0.
- BLE: tecnología orientada a aplicaciones que requieren comunicación de pequeñas cantidades de datos de forma puntual o periódica. Aparece en la versión 4.0.

Así, a pesar de que la raíz del BLE es el Bluetooth clásico, del cual es una versión optimizada, el BLE en realidad tiene unas metas completamente diferentes. De hecho, BLE no usa los 79 canales de bluetooth clásico, separado 1MHz entre sí, sino 40 canales separados 2MHz, e incluso redefine las clases existentes de bluetooth de la siguiente manera:

Clase	Potencia máxima permitida	
	(mW)	(dB)
1	100 mW	20 dB
2	10 mW	10 dB
3	2,5 mW	4 dB
4	1 mW	0 dB

Tabla 5: Potencia de emisión en BLE

Las pilas de los protocolos bluetooth clásico y BLE son diferentes pero, dado que la revisión 4.0 del bluetooth era una actualización del mismo, busca conservar la capacidad de comunicación con los dispositivos que ya implementaban el clásico.

Esto provoca que ambos no sean compatibles de forma directa, ya que el protocolo de transporte de datos en el aire, los protocolos superiores y los niveles superiores son diferentes e incompatibles entre ambas tecnologías.

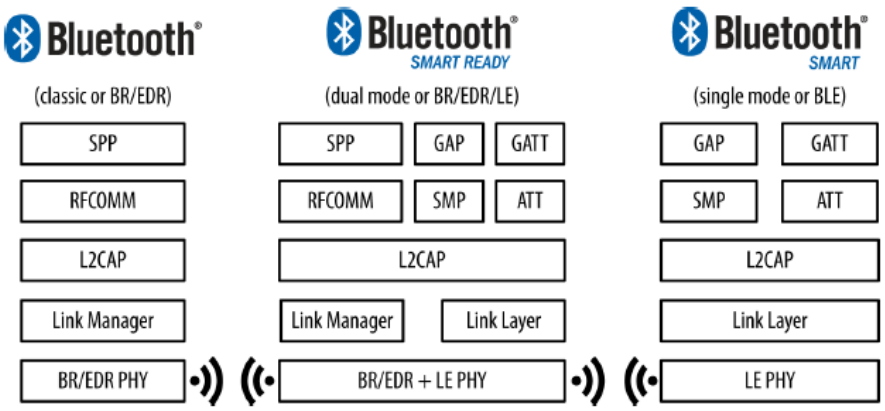


Ilustración 18: Forma de comunicación Bluetooth-BLE

Esto también provoca que haya diferencias entre las características que ofrecen ambos protocolos, aumentando las topologías que podemos implementar al trabajar con BLE.

Finalmente, mientras que el protocolo clásico ofrece transmisión rápida y cíclica de pequeños paquetes de datos hasta a 720kbps y la capacidad para la coexistencia de gran cantidad de dispositivos en un mismo entorno sin que se produzcan interferencias, el BLE ofrece un consumo muy reducido, buen funcionamiento en tiempo real si hay un bajo número de nodos interconectados y bajos tiempos de conexión/desconexión.

2.8 Bluetooth Low Energy:

Como ya se ha mencionado con anterioridad, las pilas de los protocolos Bluetooth clásico y BLE son diferentes, sin embargo, este último implementa muchos bloques recogidos en el clásico.

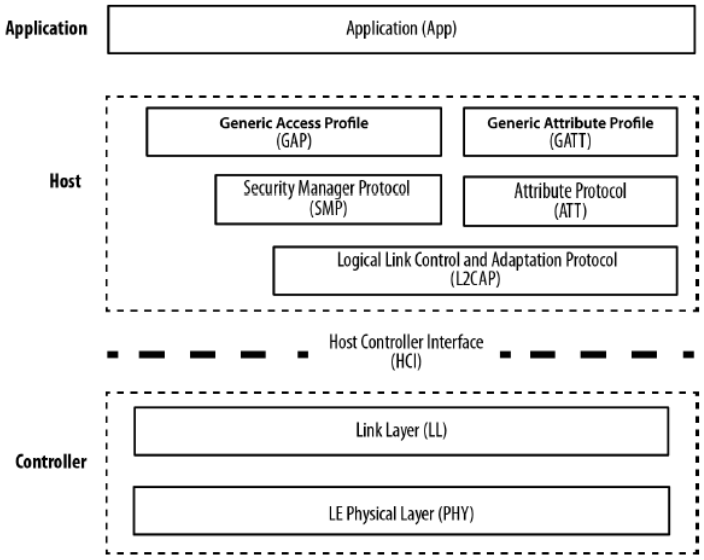


Ilustración 19: Pila del protocolo BLE

Por ello, vamos a comentar solamente los protocolos propios de los dispositivos BLE:

- ATT (Atribute Protocol): protocolo sin estado, basado en cliente/servidor y que usa atributos presentados por el dispositivo. En BLE, todo dispositivo es cliente, servidor o ambos, sin importar quién es maestro y quien esclavo.
  Bajo este protocolo, los servidores tienen sus datos organizados en forma de atributos, cada uno de los cuales está identificado mediante su UUID de 16 bits, tiene sus permisos y valores.
- GATT: basado en el protocolo ATT, crea una jerarquía y abstrae los datos, definiendo la forma de intercambio de los mismos entre aplicaciones. Manteniendo la arquitectura de ATT, encapsula los datos en servicios, cada uno de ellos consistente de una o más características.
- GAP: dicta la forma en que interactúan los dispositivos entre sí, desde la forma de realizar el descubrimiento de dispositivos, la conexión, el establecimiento de la seguridad… para lograr el intercambio de datos entre pares. Así se regula y estandarizan las operaciones de bajo nivel.

El GAP es un protocolo orientado a la gestión de los niveles inferiores de la comunicación, definiendo los procedimientos de emisión, descubrimiento de dispositivos, modos de conexión… convirtiéndose así en un protocolo obligatorio en todo BLE.

Por otro lago, el GATT se centra en los datos y su comunicación, en vez de en la conexión ente los dispositivos.

Merece la pena estudiar más a fondo los protocolos GATT y GAP.

2.9 GAP:

Usado como uno de los puntos de entrada de muchos dispositivos BLE en su nivel más bajo, proporciona una API funcional para desarrolladores de aplicaciones. GAP define aspectos como:

- Roles: cada dispositivo puede tener uno o varios, imponiendo restricciones y requisitos de funcionamiento.
- Modos: puntualiza el concepto de rol, siendo un estado en que un dispositivo puede cambiar una cierta cantidad de tiempo con un fin.
- Procedimientos: secuencias de acciones que permite a un dispositivo alcanzar un objetivo determinado.
- Seguridad: se basa en el protocolo SM, definiendo los modos y procedimientos de seguridad que se establecerán entre los pares, además de características adicionales.
- Formatos GAP adicionales: usado como marcador de posición para definiciones de formatos de datos relacionados con los modos de procedimiento definidos en el resto del protocolo.

El protocolo Gap define 4 tipos de roles diferentes:

- Broadcaster: optimizado para transmisiones regulares, de modo que se envían datos de forma pública en la red, como “paquetes de publicidad”, accesibles por cualquier dispositivo que pueda escucharlos.
- Observador: óptimo para aplicaciones de sólo recepción, está orientado a recopilar datos de la transmisión, dentro de los paquetes publicados en la red por el broadcaster.
- Central: este rol corresponde al maestro de la capa de enlace. El central comienza escuchando los paquetes publicados en la red por otros dispositivos y luego decide con cual comunicarse, ofreciendo la capacidad de mantener conexiones con varios dispositivos a la par.
- Periféricos: se corresponde con esclavos en la capa de enlace. Publica paquetes para que los dispositivos con el rol central puedan encontrarlos y, llegado el caso, decidir realizar una conexión.

Cada dispositivo puede funcionar con varios roles a la par, sin restricciones.

Hemos de tener en cuenta que no existe conexión entre las funciones de cliente y servidor BLE GATT, con los roles GAP. En un GATT, cualquier dispositivo puede ser cliente, servidor o ambos, dependiendo esto **solo** de la dirección del flujo de los datos, mientras que bajo roles GAP, un servidor será un dispositivo periférico que envía datos a otro central a disposición de este último. Así, los roles GAP son fijos, a diferencia de los de un GATT.

Los modos de procedimiento de transmisión y observación definidos en GAP establecen la forma unidireccional de envío de datos. Debido a dicha unidireccionalidad, el emisor nunca sabe si los datos han llegado al receptor (observador), que por su lado, escucha de forma temporal o indefinida, también sin la garantía de que llegue algún mensaje.

2.10GATT:

EL GATT establece en detalle la forma de intercambio de todo el perfil y datos de usuario en una conexión BLE ocupándose, a diferencia del GAP, de las interacciones de bajo nivel con los dispositivos.

Así, todos los perfiles BLE se basan en GATT, siendo obligado que todos los datos pasen bajo su formato y reglas. Esos datos están jerarquizados en grupos denominados *servicios*, que agrupan en sí datos de usuario, que reciben el nombre de *características*.

Bajo este protocolo, los dispositivos pueden adoptar diversos roles:

- Cliente: se corresponde con el cliente ATT enviando solicitudes al servidor y recibiendo respuestas. Este, desconoce los atributos del servidor de forma previa, de modo que primero pregunta por ello durante el descubrimiento del servicio. Tras dicho descubrimiento, el cliente puede empezar a trabajar con los atributos.
- Servidor: se corresponde de igual forma con el servidor ATT, recibiendo solicitudes de un cliente, que atiende devolviendo respuestas.

Cabe señalar que, debida la independencia entre GAP y GATT, tanto un periférico GAP como un central GAP pueden trabajar como servidor GATT, o incluso como ambos a la par.

Dentro del GATT, llamamos atributo a la entidad de datos más pequeña que podemos tener, conformando la forma de información más pequeña que podemos organizar. Conceptualmente, siempre pertenecen al servidor.

Los permisos son metadatos que especifican las operaciones ATT ejecutables sobre una característica en particular. Existen diversos permisos diferentes que puede tener una característica:

- Ninguno: impide lectura y/o escritura sobre la característica por parte del cliente.
- Lectura: permite que el cliente lea la característica.

- Escritura: permite que el cliente escriba sobre la característica.
- Lectura y escritura: ofrece al cliente los 2 permisos anteriores.

Los servicios GATT agrupan características relacionadas conceptualmente en un conjunto diferenciable dentro del servidor GATT, conformando así dichas características la definición del propio servicio. Dentro de cada servicio, cada una de las características constituyentes del mismo cuenta con su propia UUID. Las características de un servicio pueden agregar referencias a otros, lo que puede ayudar a ahorrar memoria, evitar la duplicación de datos en el servidor GATT y facilitar su diseño.

Por su lado, las características son contenedores de datos conformados siempre por 2 atributos: la declaración de la característica, que proporciona metadatos sobre el usuario actual de los datos, y el valor de la característica, que es otro atributo en cuyo campo están contenidos los datos en uso. Los valores de las características pueden ir acompañados de descriptores, que expanden los metadatos contenidos en la declaración de la característica para conocimiento del cliente.

Todas las características GATT son siempre parte de un servicio.

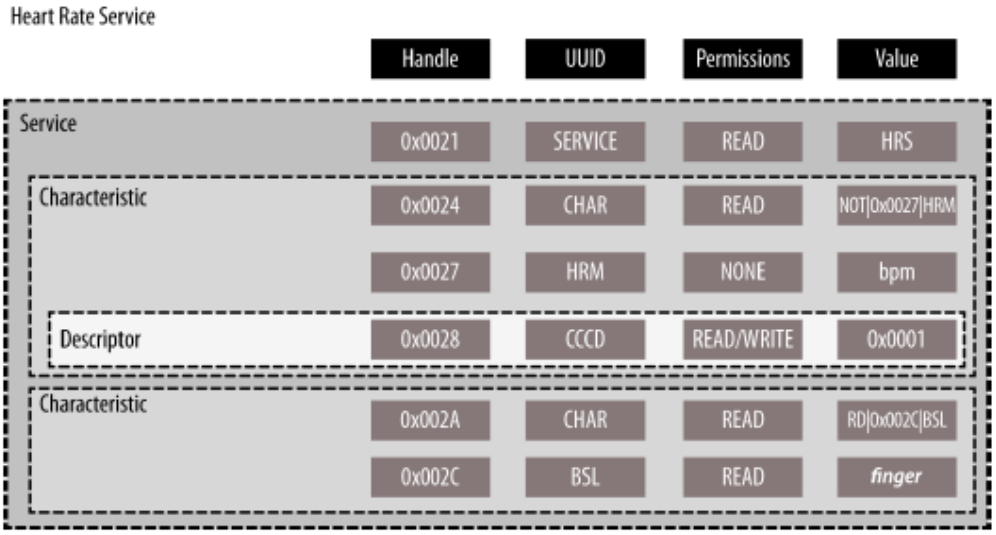


Ilustración 20: Ejemplo de servicio GATT

Hemos de saber que GATT tiene su propio servicio, obligado en todos los servidores GATT, conformado por una característica y que no puede ser leída o escrita. Este servicio no puede ser leído o escrito, y sólo se comunica al cliente a través de características.

Un cliente debe habilitar indicaciones para esa característica del servidor antes que nada para poder estar al tanto en los valores de cualquier atributo. Si el

servidor sufre cambios estructurales, esto es comunicado al cliente para que ambos puedan mantener de forma correcta la comunicación.

```
#define SERVICE_UUID                              "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"

#define CHARACTERISTIC_UUID_ESTADO                "6E400003-B5A3-F393-E0A9-ECECECEBBBBB"
#define CHARACTERISTIC_UUID_TIEMPO                "6E400003-B5A3-F393-E0A9-ACACACABBBBB"
#define CHARACTERISTIC_UUID_CICLO                 "6E400003-B5A3-F393-E0A9-CCCCCCCBBBBB"

#define CHARACTERISTIC_UUID_RECIBE_CICLO          "6E400003-B5A3-F393-E0A9-FCCFCCFCCBBB"
#define CHARACTERISTIC_UUID_RECIBE_NOMBRE_TIEMPO  "6E400003-B5A3-F393-E0A9-FECFECFECBBB"
```

Ilustración 21: Ejemplos UUID utilizados

El uso del GATT permite que un servidor cree varios servicios, cada uno con las características que se consideren necesarias o lógicamente agrupadas, como se puede ver en la siguiente imagen:

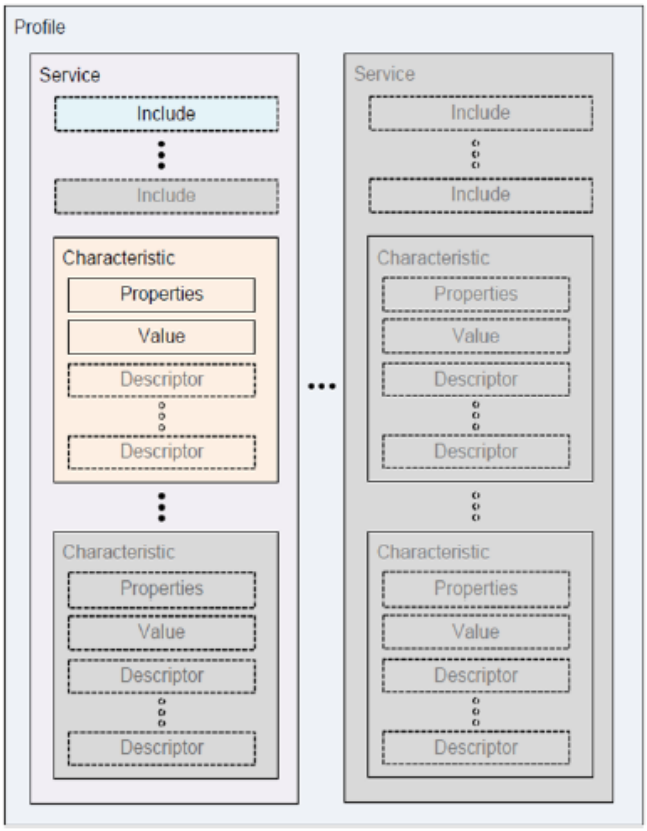


Ilustración 22: Jerarquía del GATT

De este modo, los dispositivos se anuncian en una red con el fin de establecer conexión con otros mediante el GAP, mientras que hacen uso del GATT para realizar dicha comunicación, pues es en él donde se definen los servicios y características. Una diferencia importante entre GAP y GATT es que mientras que el primero es unidireccional, el segundo es bidireccional.

3 Microcontrolador ESP32

El ESP32 es un System On Chip desarrollado por Espressif Systems, del cual hay varios modelos con distintas características.

Espressif define al Esp32 como un elemento orientado a aportar conectividad a microcontroladores que no dispongan de ella, sirviendo como vía a la red y el más actual IoT. Así mismo, fue desarrollado con diversos propósitos, como la capacidad de ejecutar código en tiempo real.

Hemos de diferenciar entre chip ESP32 y tarjeta o módulo ESP32, pues esta última incluye no solo el chip, sino todo un sistema desarrollado donde se incluyen memoria, microprocesador, oscilador a modo de reloj y, según el modelo, una antena para ampliar el rango del chip, todo ello montado en un PCB.

Los chips desarrollados por Espressif son lo que se muestran en la tabla siguiente:

Ordering code	Core	Embedded flash	Connection	Package
ESP32-D0WDQ6	Dual core	No embedded flash	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 6*6
ESP32-D0WD	Dual core	No embedded flash	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 5*5
ESP32-D2WD	Dual core	16-Mbit embedded flash (40 MHz)	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 5*5
ESP32-S0WD	Single core	No embedded flash	Wi-Fi b/g/n + BT/BLE Dual Mode	QFN 5*5

Tabla 6: Familia ESP32

Desde esta tabla, podemos identificar que hay chips con 1 ó 2 núcleos y la inclusión de o no de una memoria flash embebida.

La nomenclatura de estos chips define sus características principales bajo la codificación siguiente:

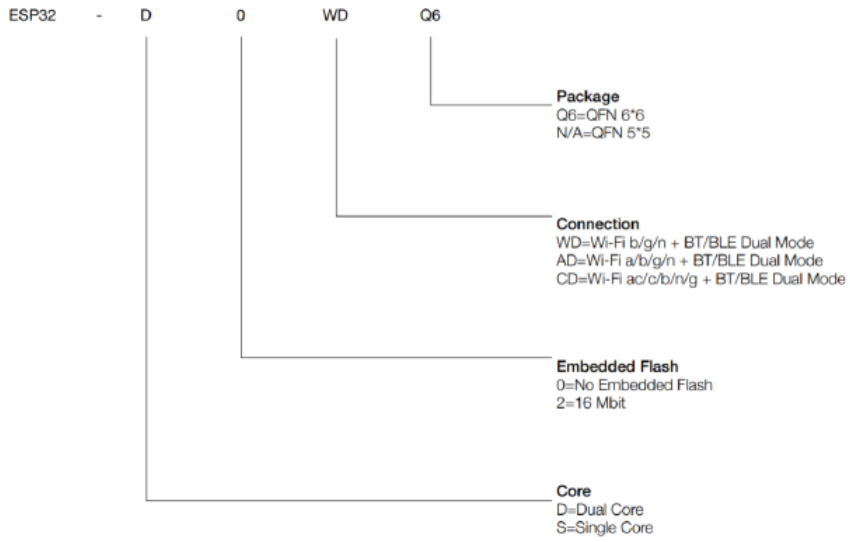


Ilustración 23: Significado del código en la familia ESP32

3.1 Especificaciones del ESP32:

El chip ESP32 es mucho más complejo de lo que la nomenclatura indicada en la ilustración 4 nos deja entrever.

Si hacemos uso de los documentos oficiales de Espressif, nos encontramos con la siguiente imagen, un diagrama de bloques en el que se muestran sus principales funciones:

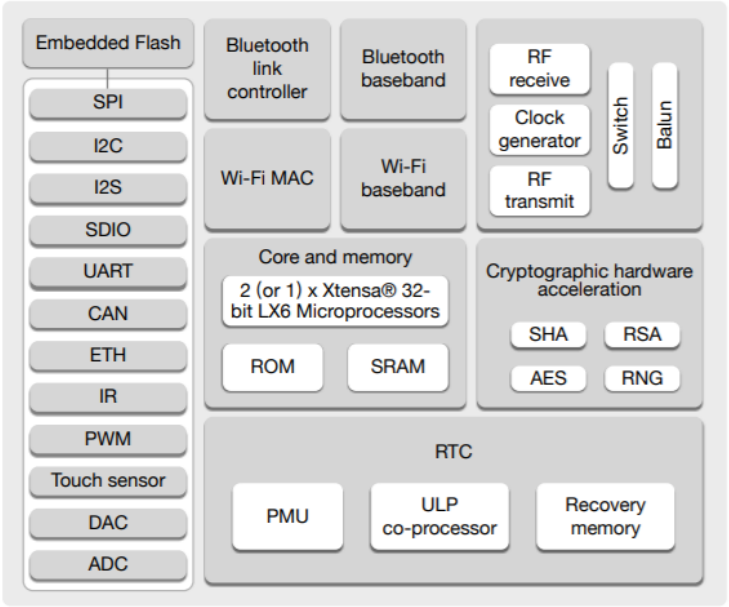


Ilustración 24: Diagrama de bloques del chip ESP32

En la figura, podemos apreciar que el ESP32 permite la coexistencia de Bluetooth y Wi-Fi , además de permitir trabajar con Bluetooth o BLE.

También se ve un RTC, que dispone de un coprocesador. La presencia de este permite programación de watchdogs, como veremos más adelante.

Por otro lado, se aprecia la presencia de una memoria flash embebida, así como varias funciones, entre las que destacan comunicación SPI, I2C e I2S, varios convertidores DAC y ADC, modulación PWM e incluso sensores táctiles.

Es de interés ver más en detalle algunos de estos bloques.

3.1.1 Memoria:

El chip cuenta con varias memorias internas:

- ROM (448KB): para funciones de los núcleos y boot.
- SRAM (520KB): para datos e instrucciones.
- SRAM del RTC (8KB): accesible desde el modo "deep-sleep", esta es de acceso rápido y puede usarse por el núcleo principal.
- SRAM del RTC (8KB): de "velocidad lenta", para el acceso del coprocesador en el modo "depp-sleep".
- FLASH: embebida, va de 0 a 4MB según el modelo.

Cabe decir que, según el modelo en uso, puede ampliarse acoplando una memoria flash externa de hasta 16MB o una SRAM de hasta 8KB.

3.1.2 Procesadores:

El/los microprocesadores Tensilica Xtensa con que cuenta el chip ESP32 son de 32-bit, con frecuencias de reloj que les permiten trabajar hasta a 240MHz.

Están preparados para realizar conversiones ADC y DAC, aprovechar la memoria RTC y sus funcionalidades, verificar el estado de sus pines…

Por otro lado, este chip incorpora un procesador de muy bajo consumo, que puede funcionar cuando la CPU entra en modo "deep-sleep", incluso haciendo uso de la "memoria lenta" del RTC, permitiendo una notable reducción del consumo.

3.1.3 Conectividad:

Como se ha mencionado con anterioridad, este chip posee tanto tecnología Bluetooth, como tecnología Wi-Fi. Es importante tener en cuenta que las versiones más actuales soportadas por este chip son el estándar Wi-Fi 802.11 b/g/n y hasta el estándar 4.2 de Bluetooth.

3.1.4 Elementos de radiofrecuencia:

El chip ESP32 cuenta con un receptor de 2.4GHz, capaz de demodular y digitalizar con alta resolución las señales recibidas. Dicho receptor incluye filtros RF, elementos de autoajuste de ganancia… con el fin de adaptarse a las condiciones de la señal dentro de la banda de los 2.4GHz.

Por otro lado, hay un transmisor en la misma banda de frecuencias, para realizar el trabajo contrario al receptor con las mismas características de resolución. Utiliza una antena para la transmisión de señales, que amplifica a través de un amplificador de potencia basado en CMOS.

Por último, hay un generador de onda cuadrada a 2.4GHz para transmisión y auto calibración.

3.1.5 Timers:

Este chip proporciona 4 temporizadores de propósito general y 64-bits, programables en sentido ascendente o descendente y controlables por nivel o flanco.

3.1.6 Watchdogs:

Encontramos entre 2 y 3 watchdogs diferentes, según la cantidad de núcleos del ESP32 que estemos utilizando.

Todos estos watchdog están destinados a recuperar el hilo del programa ante bloqueos durante la ejecución del mismo, mediante la ejecución de diferentes trabajos que pueden ir desde resetear la CPU, hasta algo que el usuario decida programar.

3.1.7 Relojes del sistema:

Hay varios relojes en el chip, pudiendo diferenciarlos en varios tipos:

- Reloj de la CPU: usa un oscilador externo a 40MHz, conectado a un PLL para aumentar su frecuencia hasta los 160MHz. Cuenta con un oscilador interno de 8KHz.

- RTC: basado en un oscilador de 150KHz y que cuenta con otro oscilador interno, de 8MHz.

3.1.8 Seguridad:

En este chip podemos encontrar implementados varios estándares para garantizar la seguridad de las conexiones Wi-Fi y Bluetooth, limitados por las versiones que soporta de cada uno. Entre ellos, podemos reconocer WPA, WPA2 y WPAI.

3.1.9 Periféricos y servicios:

El chip ESP32 cuenta con varios periféricos integrados, entre los cuales podemos destacar:

- GPIOs: el ESP32 cuenta con 34 GPIO, reprogramables a través de la gestión de los registros adecuados. Entre ellos, encontramos pines digitales, digitales y analógicos e incluso algunos con capacidad táctil.
- Conversores ADC: con una resolución de 12-bits, son capaces de trabajar incluso en el modo deep-sleep a través del coprocesador de bajo consumo
- Conversores DAC: para convertir hasta 2 señales analógicas en digitales, con una resolución de 8-bits.
- Sensores internos de temperatura, efecto Hall y "touch"
- UART interno para la comunicación de datos de forma asíncrona
- Controlador infrarrojo para comunicación.
- Interfaz para crear PWM en diferentes pines
- Protocolos I2C, I2S y SPI

3.1.10 Modos de funcionamiento:

El ESP32 puede trabajar de varias formas diferentes, cada una orientada a dar más o menos servicios con el fin de mantener controlado el consumo. Estos modos de funcionamiento son los siguientes:

- Activo: posibilita escuchar, recibir y enviar.
- Modem-sleep: desconecta Wi-Fi y bluetooth, pero mantiene operativa la CPU y se pueden configurar los relojes.
- Light-sleep: pausa la CPU para que pase a trabajar el coprocesador del RTC. La CPU puede activarse a través de eventos que requieran de ella.

- Deep-sleep: mantiene activos solamente la memoria y el RTC, que se encarga de almacenar los datos Wi-Fi y bluetooth.
- Hibernación: deshabilita el coprocesador, quedando el chip a la espera de ser despertado por acción de un evento externo a través de los GPIO o un evento programado del RTC.

3.2

3.3 Módulos ESP32 de Espressif:

Conocidos los diferentes chips creados por Espressif, vamos a ver los diferentes módulos desarrollados por la misma, cuyo resumen de propiedades puede verse en la tabla siguiente:

–	Key Components				Dimensions [mm]		
Module	Chip	Flash	RAM	Ant.	L	W	D
ESP32-WROOM-32	ESP32-D0WDQ6	4MB	–	MIFA	25.5	18	3.1
ESP32-WROOM-32D	ESP32-D0WD	4MB	–	MIFA	25.5	18	3.1
ESP32-WROOM-32U	ESP32-D0WD	4MB	–	U.FL	19.2	18	3.2
ESP32-SOLO-1	ESP32-S0WD	4MB	–	MIFA	25.5	18	3.1
ESP32-WROVER	ESP32-D0WDQ6	4MB	4MB	MIFA	31.4	18	3.2
ESP32-WROVER-I	ESP32-D0WDQ6	4MB	4MB	U.FL	31.4	18	3.5

Tabla 7: Módulos desarrollados por Espressif usando el ESP32

3.3.1 ESP-WROOM-32:

Este módulo utiliza el chip ESP32-DOWDQ6, lo que implica que tiene 2 núcleos pero ninguna memoria flash embebida.

Esto se solventa en el módulo al incluir una de 4MB, ampliable a 8 o 16MB y particionable durante la configuración software para determinar los tamaños de memoria de programa y memoria de datos.

Así mismo, incluye una antena MIFA integrada en el PCB.

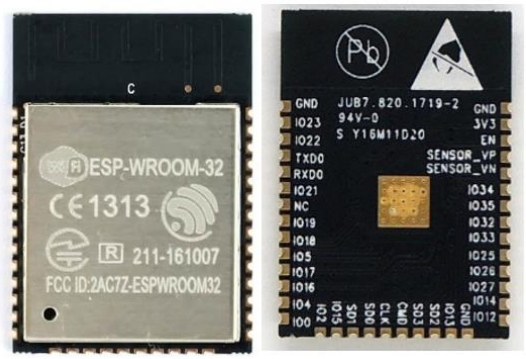


Ilustración 25: ESP-WROOM-32

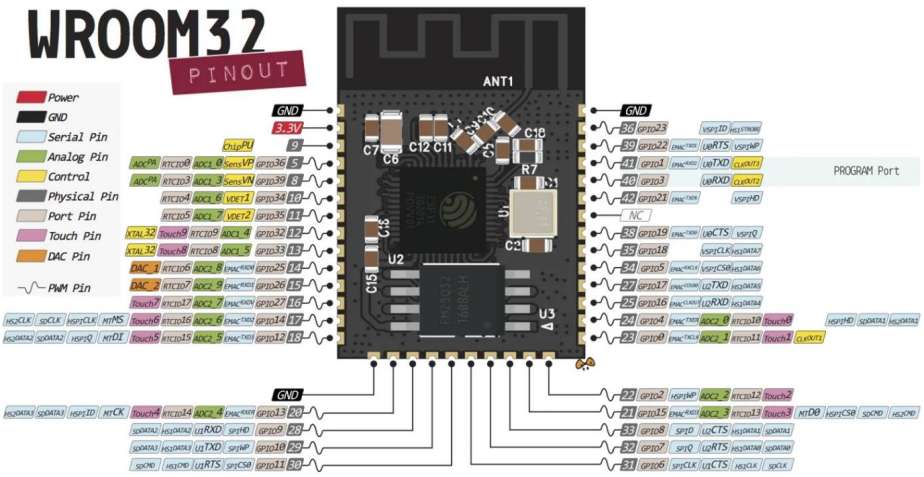


Ilustración 26: Pinout ESP-WROOM-32

3.3.2 *ESP-WROOM-32D:*

Prácticamente similar al ESP-WROOM-32, la diferencia clave entre ambos es el chip ESP32 que montan, pues el módulo WROOM–32D utiliza el chip ESP32-D0WD.

Ilustración 27:ESP-WROOM-32D

3.3.3 *ESP-WROOM-32U:*

Usando el chip ESP32-D0WD, este módulo tiene características similares al ESP-WROOM-32D, con la salvedad de que no monta una antena integrada en el PCB, sino una IPEX/U.FL, lo que permite que sea más compacto que los módulos anteriores.

Ilustración 28:ESP-WROOM-32U

3.3.4 *ESP-SOLO-1:*

Este módulo es una versión simplificada del módulo ESP-WROOM-32D, utilizando el chip ESP32-SOWD, lo que implica que tiene 1 único núcleo, con una frecuencia de trabajo de 160MHz.

Ilustración 29:ESP-SOLO-1

3.3.5 ESP-WROVER y WROVER-I:

Este módulo es el más avanzado y contiene una RAM de 4MB, además de una memoria flash externa, también de 4MB. La diferencia entre ambos reside en el tipo de antena que montan.

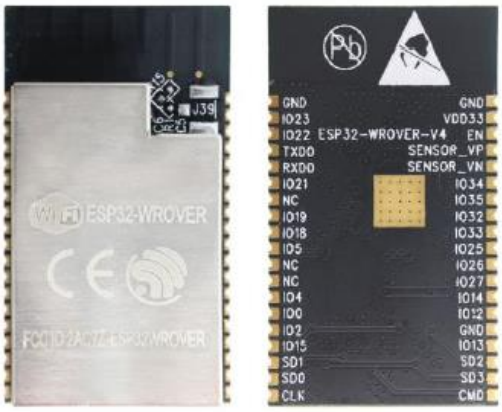


Ilustración 30:ESP-WROVER

3.4 Kits de desarrollo:

Debido a la versatilidad de los chips ESP32 y los módulos que la propia Espressif sacó a mercado a finales de 2016, muchas empresas desarrollaron sus propios *kits de desarrollo*, utilizando los diferentes módulos adaptándolos a sus necesidades con el fin de ofrecer productos diferentes en el mercado.

Debido a esto, hay una gran cantidad de kits de desarrollo diferentes, de modo que solo haremos un repaso de los principales desarrollados por Espressif. Dejamos para más adelante el usado en el proyecto.

3.4.1 ESP32-PICO-KIT:

Es el kit más pequeño, incluyendo antena LDO, botones para reset y boot y soporte USB/UART.

Su tamaño es de 20x52 mm, lo que le hace idóneo para aplicaciones portátiles, pudiendo alimentarse con una pila.

Algo importante a tener en cuenta sobre esta placa, es que monta un módulo ESP especial, el ESP-PICO-D4, con 4MB de memoria flash.

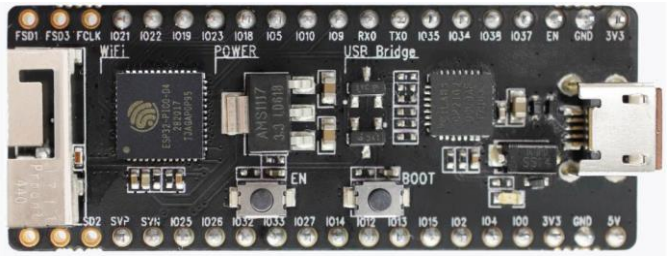


Ilustración 31:ESP32-PICO-KIT v4

3.4.2 ESP32-LyraT:

Este kit está especialmente diseñado para el desarrollo de aplicaciones de audio, permitiendo reconocimiento por voz y estando dotada de botones para la reproducción de sonido y jacks 3.5 para salidas de audio.

Usa un módulo ESP-WROVER-32 y viene preparada para la introducción de tarjetas microSD, con el fin de almacenar audio.

Este kit de desarrollo se utiliza en altavoces inteligentes y aplicaciones Smart Home.

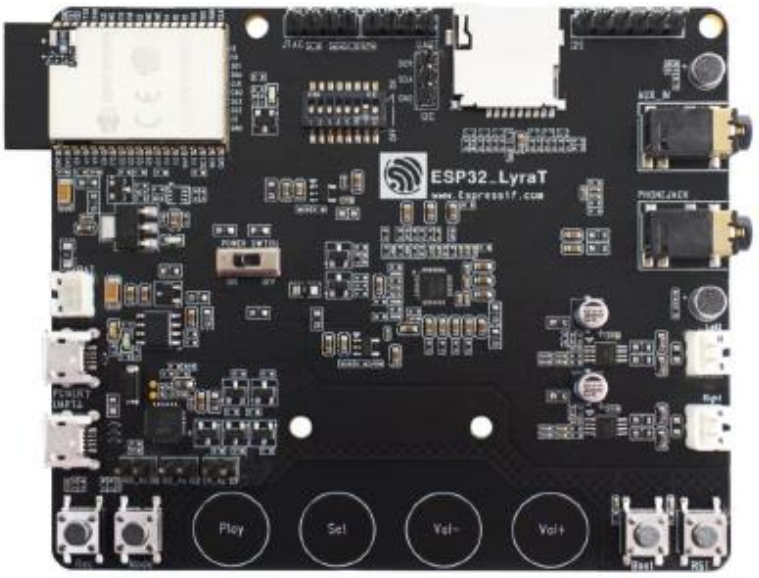


Ilustración 32:ESP-LyraT

3.4.3 *ESP32-WROVER-KIT:*

Este es el kit más potente, utilizando el módulo ESP-WROVER-32. Cuneta con un interfaz JTAG, pulsadores, lector de tarjetas microSD, puerto USB y pines preparados para la conexión de una pantalla LCD o dispositivos de vídeo. Es la más versátil de Espressif.

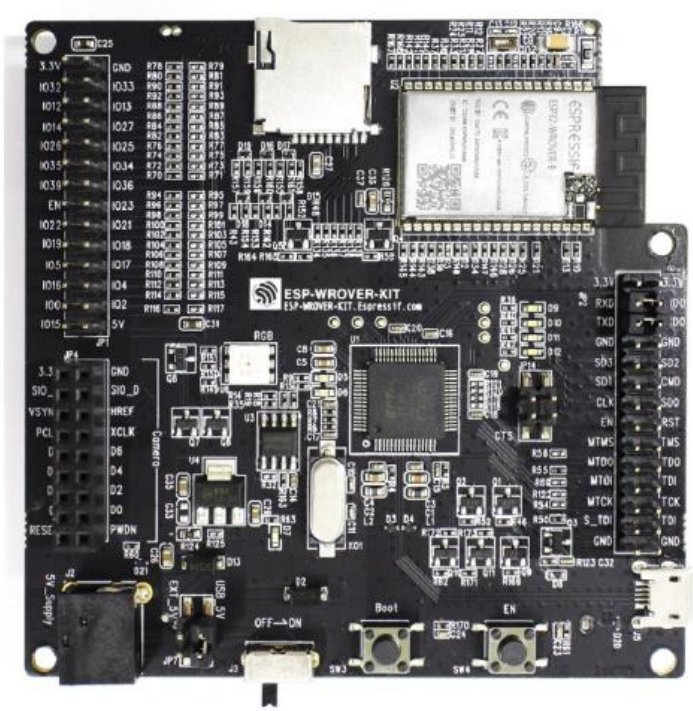


Ilustración 33:ESP32-WROVER-KIT

3.4.4 *Módulo Devkit v1:*

El módulo Devkit v1 es un sistema embebido basado en una placa ESP-WROOM-32, que incluye diversos drivers como el CP2102, un convertidor serie que nos permite gestionar la conexión mediante USB con el ESP32, en modo USB/UART, con una velocidad de hasta 3Mbit/s..

También incluye un regulador de tensión AMS1117 3.3v, lo que proporciona una tensión estable al nivel requerido por el ESP32.

Sin embargo, la característica que ha sido decisiva en cuanto a la elección del mismo, es el número de pines con acceso al ESP-WROOM-32, puesto que a pesar de existir versiones con un mayor número de pines, en el desarrollo de este proyecto basta con los que este proporciona (se requieren tan solo 15), manteniendo a su vez un reducido tamaño, lo que posibilita crear un módulo esclavo receptor compacto (las dimensiones de

28.6 x 51.65 mm), lo que le fácilmente acoplable a cualquier panel exterior de dimensiones reducidas sin que las rebase.

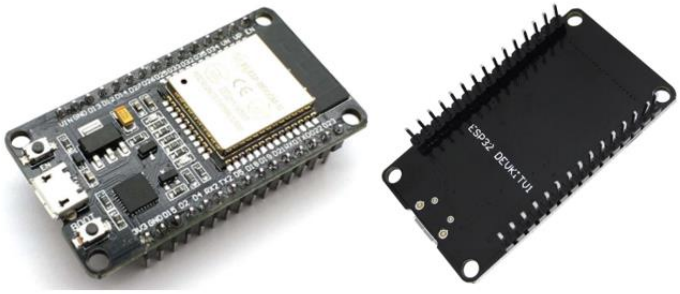


Ilustración 34:Módulo Devkit v1

Este módulo cuenta con 30 pines, cuyas características principales pueden verse en la documentación aportada en los anexos.

4 Hardware del controlador

4.1 Devkit v1 vs. Arduino:

Ahora que ya hemos vistos las características de las diferentes palcas de desarrollo de Espressif, cabe la posibilidad de plantearnos si hay otros medios de satisfacer las necesidades requeridas para este proyecto. En el mercado podemos encontrar diferentes placas de desarrollo más o menos conocidas, la mayoría de ellas basadas en microcontroladores con diferentes capacidades.

Una de las más conocidas es *arduino*.

Por ello, en este punto vamos a justificar la elección del Devkit v1, menos conocido y utilizado, frente con una placa cuyo uso está mucho más extendido y ofrezca características similares de procesamiento. Por ello, vamos a mostrar las características de varios arduino diferentes que podrían llegar a utilizarse.

4.1.1 Arduino NANO 33 IOT:

Este módulo no puede ser alimentado de forma directa a 5(v) ni recibir señales de más de 3.3(v) en ninguno de sus pines, sin embargo, nos ofrece comunicación BLE, Wi-Fi y bluetooth clásico mediante el módulo NINA-W10, basado en el ESP32 de Espressif.

Dicho módulo utiliza una antena tipo PIFA y está conectado mediante un bus SPI al microcontrolador SAMD21G18A (cerebro de toda la placa) y mediante puerto serie a varios pines.

Puede trabajar a una frecuencia de hasta 48MHz y cuenta con memoria FLASH de 256KB y SRAM de 32KB. Dispone de 14 pines digitales, 8 analógicos y es capaz de proporcionar señal PWM en 8 pines. Las dimensiones de esta placa son de 18 x 45 mm. Su precio base es de 16€.

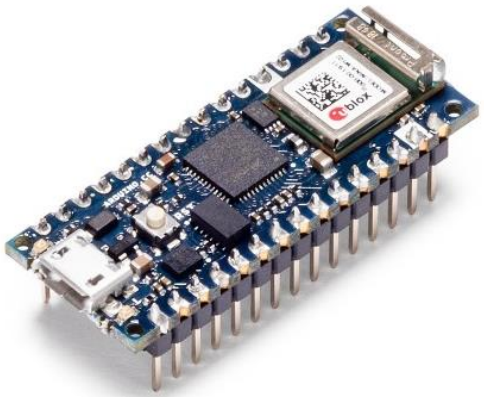


Ilustración 35: Arduino NANO 33 IOT

4.1.2 Arduino NANO 33 BLE:

Esta placa, tampoco puede trabajar suministrándosele una tensión superior a los 3.3v, ni trabajar con señales con voltajes mayores a este en ninguno de sus pines.

Por otro lado, aunque nos permita trabajar a mayores velocidades que la NANO 33 IOT (hasta 64MHz), sacrifica las comunicaciones Wi-Fi y bluetooth clásico, quedando tan solo la comunicación BLE bajo el estándar 5.0, a través del módulo NINA B306, igualmente basado en el ESP32 de Espressif.

Dicho módulo utiliza una antena tipo PIFA y proporciona los servicios de comunicación mencionados al microcontrolador nRF 52840.

Esta placa monta una memoria FLASH de 1MB y SRAM de 256KB, mejorando las características que ofrecía la NANO 33 IOT en este aspecto. Dispone de 14 pines digitales, 8 analógicos y es capaz de proporcionar señal PWM en 6 pines. El precio de esta placa es de 19.5€.

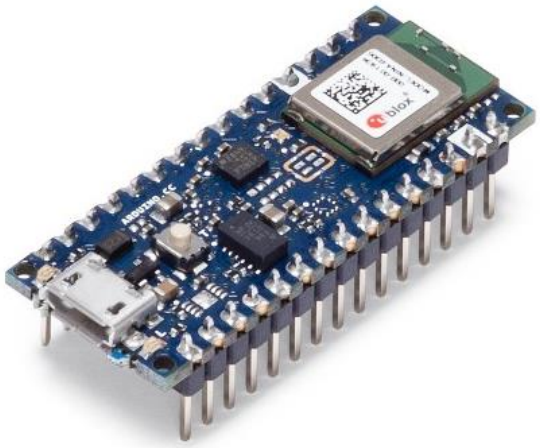


Ilustración 36:Arduino NANO BLE

4.1.3 Arduino NANO:

El arduino NANO es una de las primeras placas arduino creadas en busca de minimizar el espacio ocupado pero manteniendo las máximas características posibles de sus predecesores (como el ardunio UNO).

Así, se basa en un ATmega328 que trabaja con una frecuencia de reloj de 16MHz, pudiendo ser alimentado a 5(v). Dispone de 22 pines digitales (de los cuales 6 pueden proporcionar PWM) y 8 analógicos, todos ellos capaces

de proporcionar señales de hasta 40(mA) Por otro lado, esta tarjeta cuenta con una memoria FLASH de 32KB y una SRAM de 2KB.

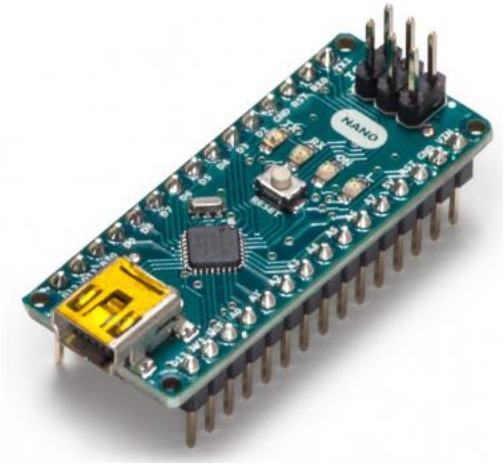


Ilustración 37:Arduino NANO v3

El mayor problema de este módulo es que, debido a cuándo fue desarrollado, no cuenta con ningún tipo de conectividad por sí mismo, lo que provocaría la necesidad de usar conjuntamente con él un módulo Bluetooth HC-06 para suplir esta carencia y así disponer de dicha conectividad, obteniéndose un conjunto similar al NANO 33 BLE.

Esto, además de reducir la cantidad de pines que quedarían disponibles, dificultaría la integración de todos los módulos en un espacio mínimo, además del rutado de la placa en que se montan todos ellos y encarecer todo el sistema en 3 ó 4€, según el distribuidor.

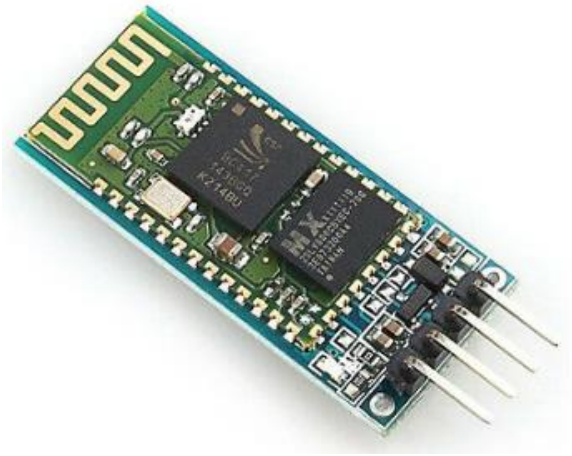


Ilustración 38: Módulo Bluetooth HC-06

Podemos hacer una comparación clara en la siguiente tabla:

	ESP32 Devkitv1	Arduino NANO 33 IOT	Arduino NANO 33 BLE	Arduino NANO
Núcleos	2	2	2	2
Arquitectura	32 bits	32 bits	32 bits	8 bits
Frecuencia Reloj	ajustable 80 - 240 MHz	hasta 48 MHz	hasta 64 MHz	hasta 16 MHz
Wi-Fi	802.11 B/G/N	802.11 B/G/N	NO	NO
Bluetooth	Si	Si	NO	NO
BLE	Si	Si	Si	NO
RAM	512 KB	32 KB	256 KB	2 KB
FLASH	4 MB	256 KB	1 MB	32 KB
GPIO	25	25	14	22
Interfaces	UART, SPI, I2C I2S, CAN	UART, SPI, I2C I2S, CAN	UART, SPI, I2C I2S, CAN	UART, SPI, I2C I2S, CAN
ADC	6	8	6	-
DAC	2	2	2	-
Vin	5-12 v	3,3 v	3,3 v	5-12 v
Imax por pin	12 mA	40 mA	40 mA	40 mA
Antena tipo	MIFA	PIFA	MIFA	-
Precio	7,50 €	16 €	19,50 €	20 €

Tabla 8: Comparación módulos arduino vs. Devkit v1

Desde ella, podemos ver no solo que cualquiera de las placas de arduino vale más del doble que la Devkit v1, sino que además, todas ellas usan procesadores con frecuencias de reloj muy inferiores, memorias de menor tamaño, voltajes de operación más comprometidos… Lo que, sumado a que todos ellos usan módulos RF basados en el chip ESP32, original de Espressif, nos ha llevado a la elección de la Devkit v1.

4.2 Elementos constituyentes:

Para lograr crear un sistema con las características requeridas para la consecución de los objetivos prefijados, se ha realizado un estudio previo basándonos en los resultados obtenidos de la creación de un controlador anterior, similar en características a los existentes en el mercado actual.

De ése *primer acercamiento*, solo se ha conservado la conexión del panel luminoso y el par Darlington con que las señales emitidas nos permiten controlar la "etapa de potencia" que supone la carga y sus diversos estados.

De este modo, se ha rediseñado toda la parte de control, dotándola a su vez de los elementos que le permiten la comunicación deseada, así como los módulos de alimentación y memoria.

Por lo tanto, los elementos que se combinan para dar solución a nuestro problema son los siguientes:

- Módulo Devkit v1
- Micro-SD card adapter
- Buck
- ULN2803AG

Si bien, se han desarrollado varias placas diferentes, con el fin de poder depurar y testear el desarrollo del trabajo a medida que este ha avanzado.

Ilustración 39: Izq-Drch: Prototipo Inicial, Placa Testeo, Placa Final

4.3 Lector micro-SD:

A pesar de que el ESP32 cuenta con una memoria interna, es preferible utilizar una memoria FLASH externa, a la que podamos acceder libremente en el instante en que sea necesario y no comprometa la cantidad de datos que se puedan recibir para controlar el sistema al desconocer los requerimientos del usuario para el mismo.

El módulo micro-SD adapter, nos permite acceder a los datos internos de una tarjeta microSD y manejarlos a conveniencia a través del bus SPI, aunque es posible usar otros interfaces como I2C o UART en caso de ser necesario.

El uso de este módulo mejora la modularidad de nuestro sistema, facilitando posibles labores de reparación en caso de que el módulo receptor resulte dañado.

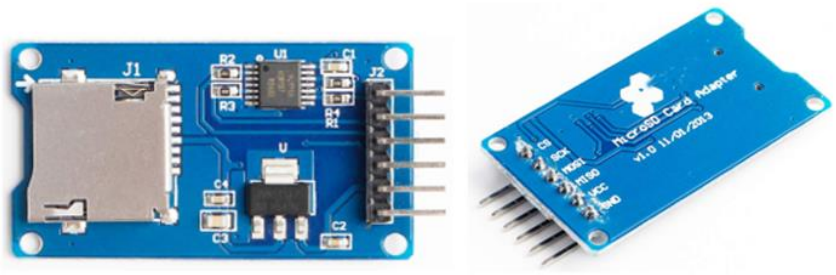


Ilustración 40: Módulo micro-SD adapter

El esquema de conexionado de este módulo se recoge en la siguiente tabla:

Pin micro-SD adapter	Pin Devkit v1
CS	D5
SCK	D18
MOSI	D23
MISO	D19
Vcc	3.3v
GND	GND

Tabla 9: Conexionado módulo micro-SD adapter

Junto con este módulo, se hace uso de una tarjeta microSD, formateada al sistema FAT32 para facilitar su integración y uso.

4.4 Buck:

Este módulo se usa para para reducir el voltaje de entrada de la alimentación general (12 (V)) a tan solo 5(V) para alimentar el módulo Devkit v1, que cuenta por sí mismo con un AMS1117 a 3.3(V) para regular el voltaje de entrada al chip ESP32.

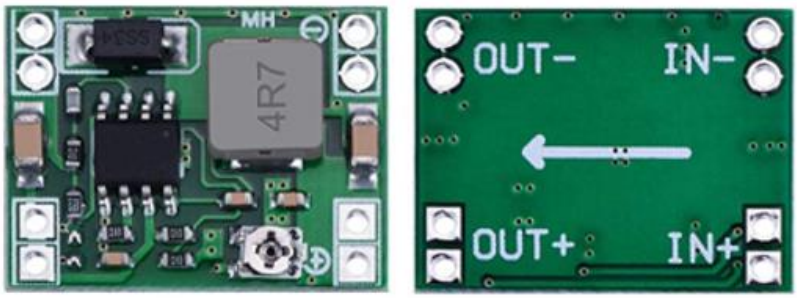


Ilustración 41:Módulo Buck

Las características de este módulo se recogen en la siguiente tabla:

	Mínimo	Máximo
Vin	4.5 (V)	28 (V)
Vout	0.8 (V)	20 (V)
Iout	1.8 (A) *típico	3 (A)
Eficiencia	-	96%
Rizado onda salida	-	30 (mV)
Temp. Funcionamiento	-45 ºC	85 ºC

Tabla 10: Características principales del Buck elegido

4.5 ULN2803AG:

El ULN2803AG es un componente electrónico que monta varios pares Darlington en su interior, lo que en este proyecto nos sirve para controlar la conexión/desconexión de leds del panel luminoso a través del funcionamiento en modo de "colector abierto" del transistor de cada una de las segundas etapas de cada Darlington.

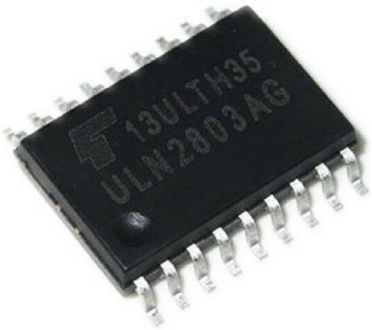


Ilustración 42: ULN2803AG

Esto se debe a que la corriente de base necesaria para lograr que cada par entre en corte/saturación (permitiendo el encendido/apagado de los led del panel) es menor que la necesaria en caso de usar 1 único transistor, debido a que la primera etapa del montaje amplifica la corriente de base en la segunda, como se aprecia en la siguiente imagen:

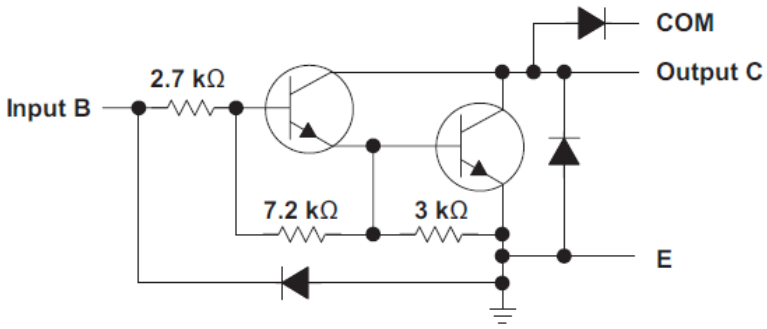


Ilustración 43:Par Darlington ULN2803

Podemos ver este componente como un interruptor controlado desde las señales de salida de los pines del ESP32 Devkit v1, conectadas a las bases de las primeras etapas de cada par.

5 Software del Controlador

5.1 Software ESP32 Devkit v1:

El controlador basado en el módulo Devkit v1, puede programarse de forma similar a un Arduino, sin embargo, las limitaciones del IDE original de Arduino son un problema de cara a editar ciertos valores para la compilación del software generado, como por ejemplo las particiones para la memoria interna y la memoria de datos del ESP32.

Por ello, se ha utilizado el IDE " VisualStudio Code", de Windows, en el que se ha hecho uso del entorno "platformio", ampliamente usado.

Platformio es un entorno de código abierto, en el que podemos encontrar librerías en las que se incluyen múltiples placas basadas en el ESP32, así como los directorios de los documentos que usa el compilador para cargar los programas, donde sí se nos permite editar las particiones deseadas para la memoria de programa y la memoria de datos del chip ESP32 (entre otros muchos aspectos).

Teniendo lo anterior en cuenta, la programación es similar a la que se haría en una placa Arduino, basándose en los bucles "setup()" y "loop()", desde los que iremos desarrollando el siguiente diagrama de flujo:

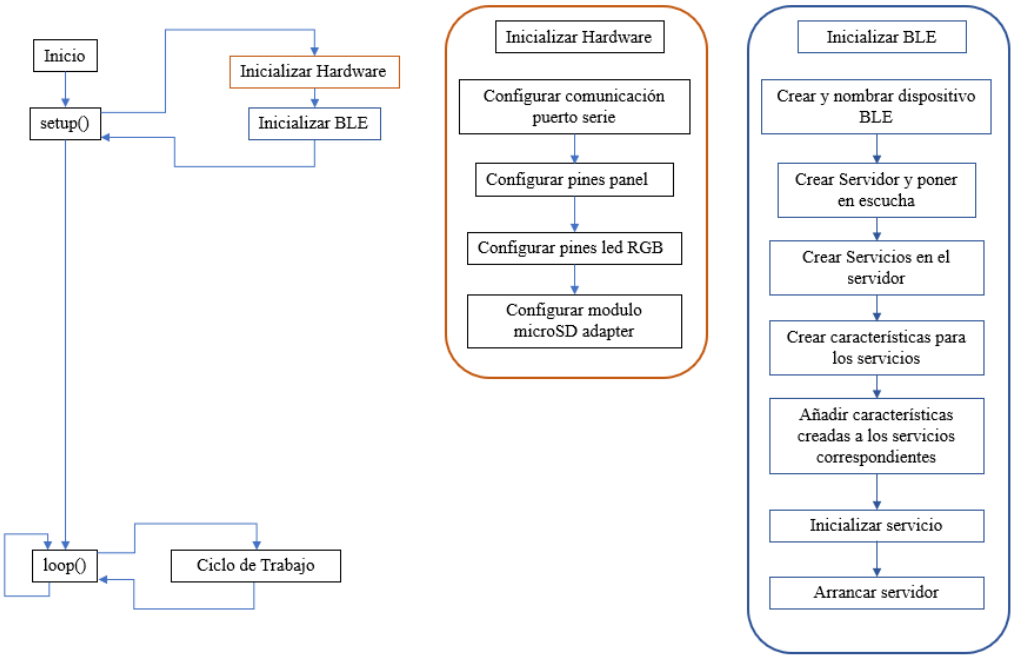


Ilustración 44: Diagrama fe flujo base del programa del módulo receptor

5.2 Void Setup:

Antes que nada, debemos tener en cuenta que la base de este controlador es usar los datos almacenados en la tarjeta micro-SD desde los paquetes enviados a través de la app android con el fin de leerlos para controlar el panel luminoso conforme a los parámetros contenidos en estos, entre los que están el tiempo entre estados y los propios estados de los leds.

Dichos datos se almacenan en forma de ficheros de texto en la tarjeta, de modo que el módulo devkit se encarga de toda su gestión (abrirlos, leerlos, cerrarlos, modificarlos) y, a través de la información contenida en ellos, llevar a saturación o corte los pares Darlington del ULN2803AG, encendiendo o apagando los módulos led de la señal luminosa.

Así mismo, usa esos datos para comunicar el estado de cada bloque de leds en la red BLE que crea a través del ESP32, para que cualquier dispositivo que se encuentre en el alcance de la misma y decida conectarse con este dispositivo pueda conocer el estado que el panel luminoso está emitiendo.

Por lo tanto, podemos resumir las tareas del Devkit v1 en las siguientes:

- Creación del servidor: junto con el servicio y las características del mismo.
- Gestión de datos: tanto su emisión en la red para dar a conocer qué trabajo está realizando en cada instante, como para poder recibir nuevas instrucciones de funcionamiento.
- Gestión del módulo micro-SD adapter: con el fin de almacenar en ella los parámetros de trabajo que el usuario le proporciona y poder recuperarlos para funcionar de forma autónoma en ausencia de un maestro.
- Control del panel luminoso: a través del ULN2803.
- Control del RGB: para indicar la existencia de conexión (o falta de la misma) del módulo receptor con un maestro.

5.2.1 Void "inicializacion_hardware()"

Durante la inicialización hardware, definimos los pines que se van a usar y la forma en que se utilizarán durante la ejecución del algoritmo desarrollado, así como la velocidad con que debe realizarse la comunicación a través del puerto serie con el computador.

Dado que el módulo microSD adapter es externo al módulo Devkit v1, también se inicializa en este momento, tras definir las conexiones pin a pin entre ambos módulos, especificadas con anterioridad, y comprobar que la conexión entre ambos módulos es correcta.

Los pines que gestionaremos en el software son los siguientes:

Pin Devkit	Uso	Pin
D15	LED RGB	R
D2		G
D4		B
D5	microSD adapter	CS
D18		SCK
D23		MOSI
D19		MISO
D13	ULN2803	led 1
D12		led 2
D14		led 3
D27		led 4
D26		led 5
D25		led 6
D33		led 7
D32		led 8

Tabla 11: Pines utilizados del Devkit v1

Realizado lo anterior, el hardware está completamente definido y preparado para su uso durante cada uno de los ciclos de ejecución. Sin embargo, dado que una de las funciones principales del módulo controlador desarrollado es la comunicación vía BLE, hemos de definir todo lo relativo a éste.

5.2.2 *Void "inicializacion_BLE():"*

Como ya se ha explicado, la comunicación BLE se basa en el uso de los protocolos GAP y ATT (además del GATT, basado en ATT).

Así, lo primero que debemos hacer es crear el dispositvo, es decir, declarar su existencia en la red, asignándole un nombre con el que será reconocido dentro de la misma. Esto se logra creando un objeto de la clase *BLEDevice* al cual asignamos nombre con que se mostrará en la red BLE a través del método *init()*.

Hecho esto, estamos en posición de *crear* un servidor con base en el dispositivo declarado, de modo que pasamos ello. Será el ESP32 quien actuará como tal, quedando el smartphone como cliente de este a decisión propia de si desea conectarse con uno u otro dispositivo. Para ello,

simplemente se hace uso del método *createServer()* sobre el objeto *BLEDevice* que creamos con anterioridad.

Realizado esto, el ESP32 habrá dispuesto un servidor que, sin embargo, no estará disponible en la red para mantener comunicaciones a través de la misma hasta que lo inicialicemos, es decir, hemos de "abrirlo al público", ponerlo en escucha, con el fin de que los posibles clientes puedan identificarlo en la res para ser captados y atendidos por él cuando estos lo deseen.

Para ello, se crea un nuevo objeto, de la clase *setCallback*s, que se encargará de gestionar las peticiones del cliente referentes a la conexión cuando este las requiera.

Hemos de tener en cuenta que esto se debe a la topología empleada en el desarrollo del proyecto, basada en *connections*, pues desde un principio se decidió trabajar así con el fin de evitar carga computacional a los receptores, ya que en redes con metodología broadcast (además del impedimento del flujo de datos bidireccional) son los receptores quienes deben comprobar si los mensajes retransmitidos por la red son o no para ellos.

Hasta este punto, lo que hemos desarrollado es la estructura del GAP, definiendo cual será el dispositivo que emplee este protocolo para la comunicación, sin embargo, queda ahora realizar toda la programación referida a los datos que van a mediar durante la comunicación entre dispositivos, es decir, lo referente al GATT.

Creado el servidor, pasamos a crear el servicio dentro de dicho servidor. Es ahora cuando hemos de empezar a tener en cuenta los UUID. Esto se debe a que los servidores son entidades públicas que el dispositivo puede decidir exponer o no en la red BLE en cada momento, y los propios dispositivos son identables por su nombre para que los clientes puedan acceder a ellos, provocando que ninguno de estos elementos requiera de UUID propio. Así, cada uno de los servicios que un servidor implemente sí los requiere, ya que es a través de dicho identificador que el cliente decide qué servicios, de entre los existentes dentro del servidor, quiere utilizar.

Cabe recordar que los servicios no son más que conjuntos características (es decir, de propiedades), de modo que estos en realidad no aportan nada más que un "entorno" para que las características que en él se incluyan formen parte de un todo común, una entidad diferenciable de otras, si bien

podemos encontrar una misma característica presente en varios servicios diferentes.

Esto provoca que las características también requieren de un UUID propio, de modo que una vez tratadas de la forma adecuada, estas proporcionen datos que, aún dentro del servicio, puedan identificarse individualmente.

De este modo, teniendo en cuenta que todo el trabajo a desarrollar mediante comunicación BLE entre los dispositivos conectados en cada instante está referido a un mismo modo de funcionamiento, se ha decidido crear un único servicio: *servicio_transferencia.*

Por otro lado, durante la comunicación vamos a poder diferenciar entre datos que recibe el servidor para conocer como debe ser controlado el panel luminoso, y datos que el controlador ya posee y puede enviar al cliente con el fin de que este conozca cómo se está trabajando en el instante actual.

Es ahora cuando debemos recordar que las características de un servicio pueden tener diferentes propiedades, que definen las posibilidades de interactuar de las mismas con el cliente. Así, las propiedades principales (además del UUID propio de cada una), más notables son:

- Propiedad de Lectura: permite que el cliente acceda a la característica con el fin de leer el valor que esta tiene.
- Propiedad de Escritura: permite al cliente dar valores a la característica. Hemos de tener en cuenta que esta propiedad requiere de su propia gestión de los datos recibidos, luego hace uso de objetos del tipo *BLECharacteristicCallbacks.*
- Propiedad de Notificación: envía una señal al cliente cada vez que se modifica el valor de la característica, permitiendo que este se subscriba a ella y tenga actualizado el valor de esta.

Así, diferenciando entre datos emitidos y datos recibidos desde el punto de vista del servidor, podríamos acordar usar 3 características con las propiedades de lectura y escritura, una para cada parte diferenciable en los datos de funcionamiento:

- Nombre del ciclo de trabajo
- Tiempo entre estados
- Ciclo de trabajo

Estas 3 características serían la base con que desarrollar el control del panel luminoso y poder mantener informado al cliente, de modo que el este podría modificarlas a su antojo en el momento que desee provocando una variación de los datos con que trabaja el servidor. Sin embargo, hemos de tener en cuenta que la emisión de datos y la recepción y tratamiento de los mismos requiere de un cierto tiempo.

Estos tiempos provocarían que nuestro controlador pudiera presentar "tiempos muertos" en el mejor de los casos, debido a la necesidad de atender tareas de mayor prioridad , o provocar que el controlador *se quedase colgado* o perdiera datos, en el peor de ellos.

Esto nos ha llevado a utilizar 5 características, 3 de emisión y 2 de recepción, con el fin de poder tratar los datos emitidos hacia el cliente por separado, y poder recibir nuevos datos desde el mismo de una forma fácilmente tratable.

Así, las 5 características implementadas en nuestro servicio son las siguientes:

- Característica_envio_estado_actual: con las propiedades de lectura y notificación, para que el cliente pueda conocer el estado del panel luminoso en cada instante.
- Característica_envio_tiempo: con las mismas propiedades que el anterior, y orientado a mostrar cual es el tiempo deseado entre 2 estados diferentes en la señal luminosa.
- Característica_envio_nombre_ciclo: al igual que las anteriores, con las propiedades de lectura y notificación, con el fin de que el cliente pueda conocer el nombre del ciclo de trabajo que está ejecutando el módulo esclavo y tener así una idea de cómo van a evolucionar los estados de los módulos led del panel luminoso.
- Característica_recibo_ciclo: con la propiedad de escritura, para que el cliente pueda enviar un nuevo ciclo de trabajo
- Caracteristica_recibo_nombre_tiempo: con la misma propiedad y finalidad que la anterior, esta vez para que el cliente edite el tiempo entre estados, y escriba el nombre del ciclo que desea ejecutar (a fin de poder conocerlo más adelante en caso de ser necesario).

Podemos hacernos una idea sencilla de todo este sistema de comunicación creado a través de la siguiente imagen:

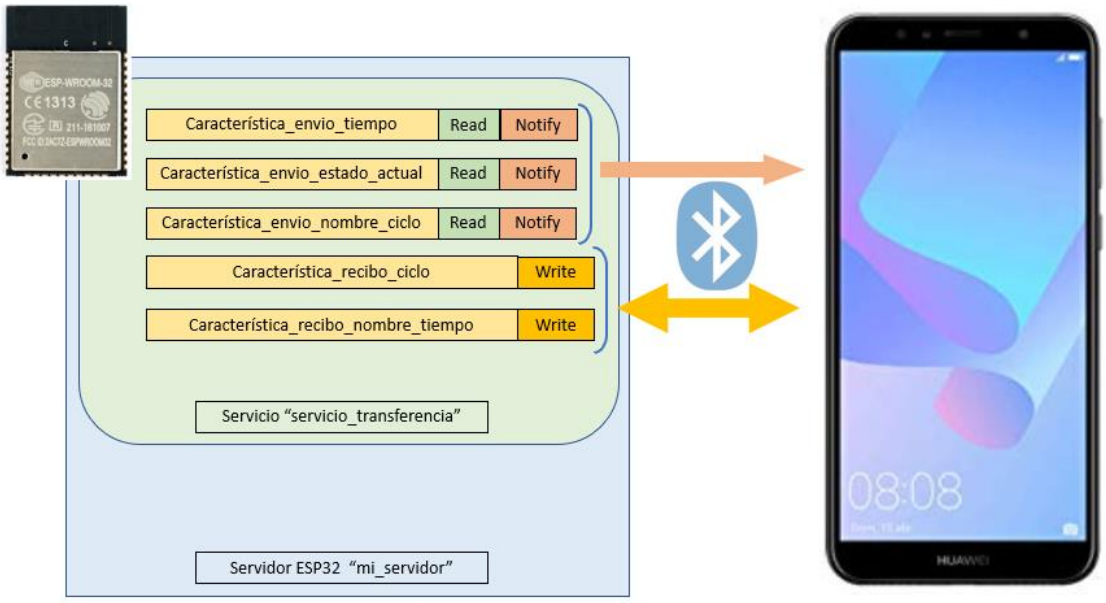


Ilustración 45: Esquema de comunicación dispositivo receptor-smartphone

Como ya se ha dicho con anterioridad, cuando el cliente quiere interactuar con el servidor, es necesaria la presencia de un objeto que se encargue de hacer las gestiones requeridas por el cliente, para que medie entre este y el servidor durante la transferencia de datos.

Es por esto que las características con la propiedad de escritura requieren de uno de estos objetos cada una, orientados a tratar adecuadamente los datos recibidos desde el cliente.

Una vez creadas las características a través de los objetos de la clase *createCharacteristic* de cada una, hemos de incluirlas en cada uno de los servicios que vayan a hacer uso de ellas.

Realizado esto, solo queda arrancar servicio y después servidor, a través de los métodos *start()* y *getAdvertising()*.

5.3 Void Loop:

Para el desarrollo de este bucle, vamos a seguir el siguiente diagrama de flujo:

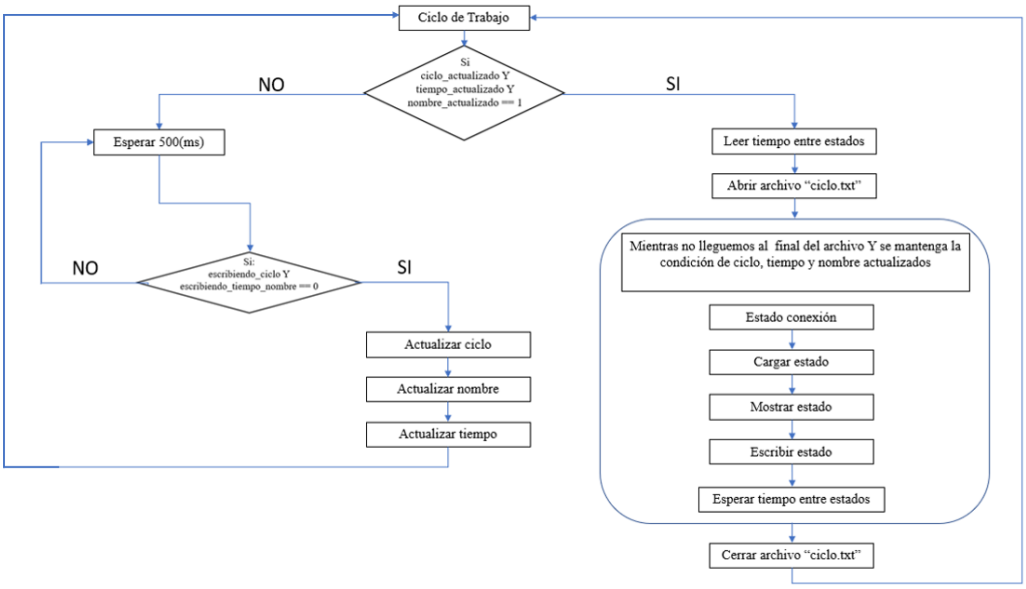


Ilustración 46: Diagrama de flujo del bucle loop()

5.3.1 Void "ciclo_de_trabajo()"

Este método es el único que se ejecuta en el loop, siendo el encargado de seguir las directrices marcadas en el diagrama de flujo de la ilustración anterior.

En él, lo primero que hacemos es comprobar si los datos que tenemos están o no actualizados, con el fin de mostrar siempre por el panel luminoso los últimos datos enviados por el maestro, es decir, los almacenados en los ficheros "*ciclo*", "*nombre*" y "*tiempo*". Para ello nos fijamos en los valores de las variables booleanas siguientes:

- Ciclo_actualizado
- Tiempo_actualizado
- Nombre_actualizado

Estas variables son utilizadas a modo de "*semáforos*" con el fin de determinar por qué rama del bucle *"ciclo_de_trabajo"* debemos ir en función de si están a 1 (true), indicando que uno u otro dato del modo de trabajo actual está actualizado, o 0 (false), indicando lo contrario.

Su necesidad se debe a que buscamos un funcionamiento autónomo del sistema en ausencia de maestro, lo que resulta en la necesidad de almacenar los datos enviados por este en instantes (o incluso conexiones) anteriores para trabajar con ellos.

Así, hemos de hacer uso no solo de 3 ficheros, sino de 6, de modo que 3 de ellos queden a modo de repositorio para almacenar los datos que escriba el cliente del servidor en las "*recibo_ciclo"* y "*recibo_nombre_tiempo*".

Estos ficheros-repositorio servirán para permitir un trabajo ininterrumpido sobre la señal luminosa, ya que mientras se están emitiendo señales de control hacia la misma, el cliente puede manejar las características con propiedad de escritura para indicar un nuevo modo de trabajo, cuya gestión se realizará por medio de los manejadores ligados a las características.

Estos 3 ficheros extra reciben los nombres de "*nuevo_ciclo", "nuevo_nombre"* y " *nuevo_tiempo*", y se tratan de forma conjunta con las variables booleanas usadas a modo de semáforo, todo ello por medio de los manejadores de las características ya mencionadas.

Así, las características "recibo_ciclo" y "recibo_nombre_tiempo" están ligadas con los métodos "nuevo_ciclo" y "nuevo_nombre", encargados de gestionar y guardar los datos recibidos desde el cliente en forma de cadena.

Ahora bien, hemos dicho que vamos a trabajar con 3 ficheros, debido a que se necesitan 3 datos para tener una forma de trabajo, sin embargo se hace uso tan solo de 2 características. Esto se debe a 2 motivos principales:

- A mayor número de características a gestionar, mayor tiempo.
- Hay datos de tamaño fijo (tiempo entre estados) y datos de tamaño variable (ciclo de trabajo y nombre del ciclo de trabajo).

Si bien podríamos enviar todos los datos a través de una única característica, sería difícil diferenciar qué parte de la trama se corresponde con cada uno de los datos del modo de trabajo. Esto se debe a que el usuario puede decidir crear ciclos de tantos estados como desee y darles el nombre que mejor le parezca, provocando que el tamaño de estos datos sea completamente imprevisible y, por lo tanto, sea una mejor opción enviarlos en partes separadas.

Sin embargo, el tiempo entre estados es un dato mucho más fácil de manejar, ya que la finalidad de este controlador es emitir mensajes

luminosos de un tiempo determinado que, al igual que los controladores actuales, provoca unos tiempos entre estados bajos.

Esto nos permite limitar su tamaño en todo el paquete de transmisión, habiéndose decidido que, además de que el tiempo que se transmitirá entre los dispositivos estará en milisegundos, tendrá un máximo de 5 caracteres, por lo que el tiempo máximo entre 2 estados será de 99999(ms), es decir, de casi 1 minuto y 40 segundos, mucho mayor de lo común.

Así, dado que el tamaño de este dato está limitado, podemos agruparle con cualquiera de los otros, de modo que podamos rescatarlo en el servidor para almacenarlo en el fichero correspondiente.

Para simplificar el trabajo, se decidió que, dado que el ciclo de trabajo puede ser significativamente mayor que el nombre del ciclo, se agruparían tiempo y nombre de ciclo por un lado, y ciclo por otro. Por la misma razón y dado que es conocido el tamaño de la trama que corresponde con el tiempo entre estados, se decidió que este dato sería la primera parte de la trama que se escribirá en la característica *"recibo_nombre_tiempo"*, de modo que tras rescatar esos primero 5 caracteres de lo escrito en esta, el resto serán los correspondientes al nombre del ciclo de trabajo.

En este momento, durante la ejecución del programa, ya sabremos si tenemos datos actualizados o no, de modo que vamos a desarrollar primero el caso en que sí lo estén.

Teniendo en cuenta que el diseño de los ciclos de trabajo hace que estos sean fijos una vez creados, así como equiespaciada la duración de cada estado en el tiempo, lo primero que se hace cada vez que se tiene un ciclo actualizado, es leer estos datos, pues serán inalterables mientras el cliente no decida que se debe realizar un trabajo diferente.

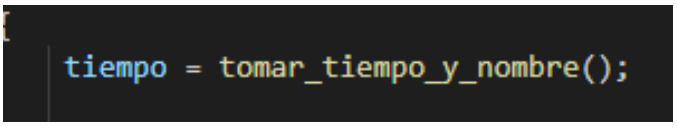


Ilustración 47: Llamada a la función tomar_tiempo_y_nombre()

De todo esto se encarga un módulo del tipo int, que no solo nos devuelve el tiempo entre estados, sino que también se encarga de toda la gestión necesaria para obtener este dato, lo que se subdivide en varios trabajos:

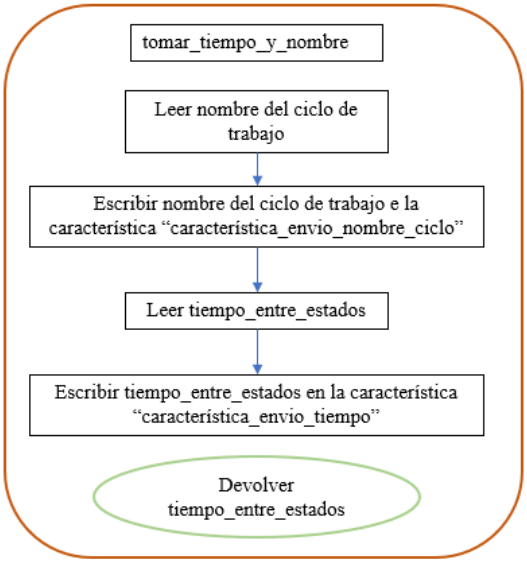


Ilustración 48: Diagrama de flujo de la función tomar_tiempo_y_nombre

Para realizar estas tareas, el programa se encarga de la apertura, lectura y almacenamiento de los datos que hay en los ficheros correspondientes.

Así mismo, también se encarga de escribir los datos almacenados en los mismos en las características previstas para que el cliente pueda acceder a ellos y leerlos. Finalmente, devuelve el valor del tiempo entre estados para que, durante el ciclo actual, pueda usarse.

Los ficheros usados para almacenar los datos anteriores están ideados de modo que solo almacenen en su interior 1 línea de caracteres, con un máximo de 5 para el fichero *"tiempo"* y un número dependiente del nombre del ciclo que se esté usando en cada momento (y por lo tanto indeterminado) en el fichero "*nombre*". Así, basta con recorrerlos a través del método propio *leer_linea*, que haciendo uso de la función *available()* para rescatar todos los datos dentro de cada uno de ellos, nos devuelve el contenido en forma de *string*.

El propio método *leer_linea* realiza la apertura del archivo que se le indique en la llamada, así como el cierre del mismo, todo ello a través de la variable tipo *file* "archivo".

Una vez realizados estos pasos, se abre el fichero donde se almacenan los estados del ciclo de trabajo, almacenados a razón de uno por línea. Esto nos permite que, a través de la condición *archivo.available()*, se pueda leer dicho archivo mientras no se haya llegado al final del mismo (además de las condiciones extra refrentes a la actualización de los datos) en un bucle.

Una vez dentro de dicho bucle, el trabajo pasa a controlar varias partes diferentes:

- Led RGB: para mostrar si hay o no un dispositivo maestro conectado al servidor, haciendo que este muestre los colores amarillo (servidor sin cliente) o verde (servidor con un cliente conectado a él). Todo ello se hace en función del valor de una variable booleana (deviceConnected) cuyo valor depende directamente del objeto *BLEServerCallbacks* de la clase *MyServerCallbacks*, ligado al propio servidor y que comprueba si el mantenedor de conexión del mismo con un cliente está conectado (onConnect) o desconectado (onDisconnect). Como se explicó con anterioridad en el *setup()*, en la parte referida a la inicialización BLE, la comprobación de presencia de cliente se hace a través del mantenedor creado mediante la función *setCallbacks* con el fin de atender a las peticiones de conexión del cliente.
- Panel Luminoso: a través de los datos leídos desde el fichero *"ciclo.txt"*.

EL control del panel se hace en varias etapas, ya que , a diferencia de los ficheros correspondientes al nombre del ciclo de trabajo y tiempo entre estados, este contiene diferentes líneas con el fin de poder leerlas una a una, siendo cada línea una cadena de 8bits, es decir, un byte, donde cada bit se corresponde con el estado de un panel led.

Se procede así a la lectura línea a línea, una por vez del fichero, de modo que una vez tomado un estado del ciclo de trabajo, este es cargado en el panel luminoso y escrito en la característica "caracteristica_envio_estado_actual", con el fin de que el cliente pueda leerlo. Finalmente, se espera el tiempo entre estados tomado con anterioridad antes de proceder a leer la siguiente línea (siempre y cuando las condiciones para permanecer dentro del bucle se mantengan).

Todo esto provoca algo a tener en cuenta: los ciclos con tiempos ente estados muy elevados provocarán que la carga de un nuevo ciclo se vea tras terminar el tiempo entre 2 estados previo, al igual que la presencia o no de un cliente dentro del servidor. Por ello, además de por razones de visualización debidas a las características del entorno de trabajo para las que se ha diseñado el controlador, se recomienda que los tiempos entre estados no sean superiores a 5 segundos, siendo preferible que su duración oscile entre 0.4 y 0.8 segundos.

Una vez realizadas todas estas operaciones, se habrá mostrado todo el ciclo de trabajo o se estarán incumpliendo las condiciones de actualidad de los datos en uso, por lo que se procede a cerrar el archivo*"ciclo.txt"* y el fin del módulo *"ciclo_ trabajo"* por la parte de ejecución cuando los estados están actualizados.

Sin embargo, cualquier variación introducida por el cliente con el fin de modificar algún aspecto del ciclo de trabajo, requiere que todos los datos actuales de trabajo sean actualizados, lo que nos llevará por la rama izquierda del diagrama de flujo del bucle *loop()*.

Haber entrado en esta rama del módulo *ciclo_trabajo* nos indica que los datos con que se estaban trabajando están desactualizados, pero esto no impide que el cliente esté trabajando en el envío de nuevos datos, bien porque el usuario haya cambiado de idea, bien porque se haya equivocado al enviar los primeros. Esto nos crea un nuevo problema: gestionar el acceso a los ficheros en que se van a almacenar los datos.

Así, se ve la necesidad de volver a hacer uso del método de *"banderas"* a través de variables booleanas que hagan las veces de *semáforo* con el fin de conocer si se están reescribiendo los datos en los ficheros *"nuevo_ciclo", "nuevo_nombre"* y *"nuevo_tiempo"* o ya se ha realizado la escritura de los mismos. Dichas variables reciben los nombres de *"escribiendo_ciclo"* y *"escribiendo_tiempo_nombre"*, haciendo referencia a las características de donde obtienen la información a guardar en los ficheros.

Para ello, lo primero que hacemos con el fin de favorecer la finalización de una posible sobreescritura de los ficheros de actualización de la forma de trabajo, es esperar 500 milisegundos con el fin de evitar intentar acceder

a los datos mientras están siendo cargados desde lo escrito por el cliente en las características.

Solamente cuando ambas *banderas* tienen un valor 0 (false), pasaremos a la actualización de los ficheros, consistente en la escritura carácter a carácter de los datos de los ficheros "*nuevo_xxxx*" en su homólogo "*xxxx"*. En otro caso, se esperarán 500 milisegundo más y repetirá la comprobación de estado de estas *banderas*.

Hecho todo esto, se procede a actualizar las variables de los semáforos booleanos, de modo que en la siguiente comprobación dentro de ña función *"ciclo de trabajo"* se pueda volver a acceder a la rama que los trata.

Hemos de tener en cuenta ahora que la escritura de los ficheros "*nuevo_xxxx*" es gestionada a través de los mantenedores de las características de escritura ligados a las mismas en el bucle *inicializar_BLE*, donde se realiza el tratamiento de los datos contenidos en ellas de modo que:

- Los datos de la característica en que se recibe el ciclo de trabajo son escritos en el fichero en forma de octetos, a razón de 1 octeto por línea.
- Los datos de la característica *recibo_nombre_tiempo* contienen en su trama las informaciones para los ficheros *nuevo_tiempo* y *nuevo_nombre*, siendo los 5 primeros caracteres de la trama los que hacen referencia al tiempo y deben escribirse en el archivo *nuevo_tiempo* y el resto, hacen referencia al nombre del ciclo, y deben escribirse en el otro dichero.

Es en este mismo momento cuando se hace la gestión de los valores de las variables booleanas *escribiendo_ciclo* y *escribiendo_nombre_tiempo*, además de todo el trabajo referente a la apertura, vaciado y cierre de los archivos según se necesita.

5.4 Software app Android:

Para el desarrollo de la aplicación Android con que comunicar los módulos maestro y esclavo, se ha usado el IDE "Android Studio", pero antes de entrar a explicar el código creado, conviene tener en mente cómo es el ciclo de vida de una app android.

5.4.1 *Ciclo de vida de una aplicación Android:*

Las apps android se basan en el uso de pantallas con el nombre de "activity". Al lanzar una, se crea la misma y se carga la vista (layout), que tiene ligada.

En la vista de cada activity, aparecen los elementos cuya visualización hemos habilitado, con los cuales podemos interactuar con el fin de provocar la ejecución de diversos métodos definidos para dicha activity con el fin de obtener uno u otro resultado.

Hemos de tener en cuenta que podemos crear una activity desde otra, pero esto aumenta el stack de activities si no se hace uso del método finish() para liberar un activity de la pila de ejecución, de modo que podemos tener varias activity trabajando en varios planos diferentes, pero solo visualizaremos la que está en la parte superior de la pila.

Todo ello, provoca que los recursos de que disponemos en el smartphone se vayan distribuyendo en función de las activity operativas en cada instante.

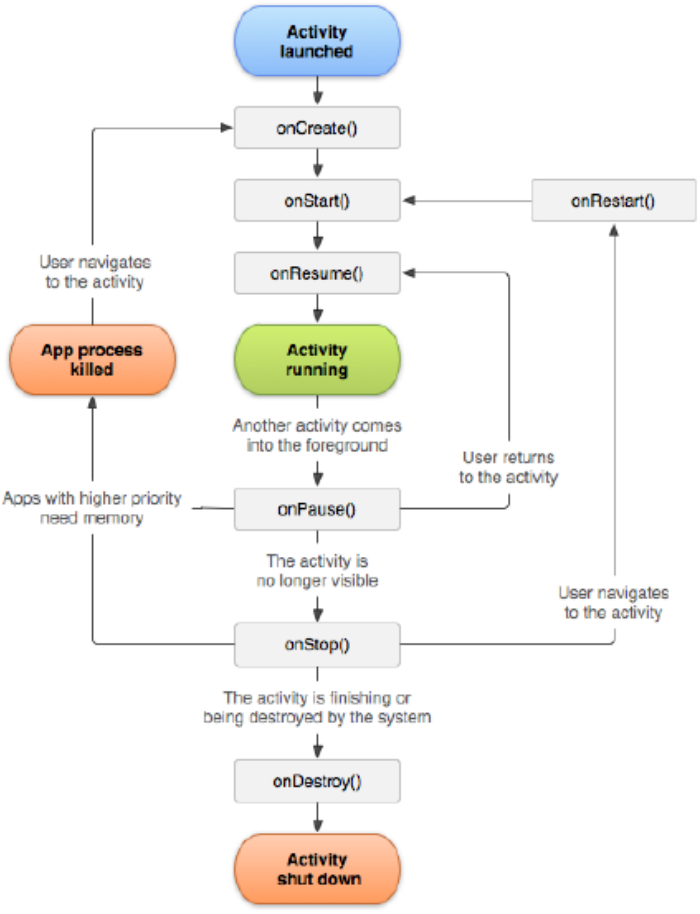


Ilustración 49: Ciclo de vida de una activity en una app Android

5.4.2 *Desarrollo del programa:*

El programa consta de 6 activitys, pudiendo llegar a una u otra según el siguiente diagrama de flujo:

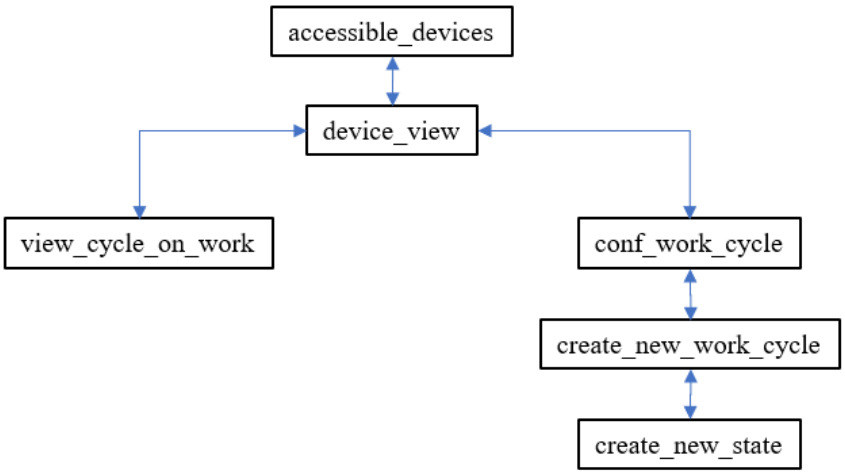


Ilustración 50: Flujo entre activities

Para entender mejor el funcionamiento de la app, vamos a desarrollar el código de cada activity junto con una vista del layout diseñado para cada una, pero antes vamos a hacer una breve descripción de lo que nos ofrecerá cada una de ellas:

- Accessible_devices: en esta vista se nos mostrará un listado de los dispositivos BLE existentes en la red que alcanzamos a descubrir. Este listado muestra el nombre del dispositivo y el estado de conexión del mismo.
- Device_view: esta activity es un acceso a un dispositivo con que deseamos realizar un enlazamiento (establecer conexión). Así, nos mostrará el nombre del dispositivo y su estado de conexión. Si dicho estado es “conectado”, nos mostrará la configuración con que está trabajando en ese instante, actualizando el estado de la señal luminosa a medida que este cambia durante el ciclo de trabajo.
- View_cycle_on_work: esta vista es meramente informativa, con el fin de poder ver todos los estados constituyentes de un cierto ciclo de trabajo. Sirve así para que el usuario pueda ver cómo va a evolucionar lo mostrado por la señal luminosa sin necesidad de estar esperando a que ocurra un ciclo completo, siendo de especial interés cuando la señal estaba trabajando de forma previa a la conexión.

- Conf_work_cycle: esta activity, tal y como indica su propio nombre, nos permite configurar el ciclo de trabajo que deseamos mostrar por el panel luminoso. Así, nos da acceso a los diferentes ciclos existentes y que podemos cargar (mostrándolos a modo de lista deslizante), tanto como la opción de crear nuestro propio ciclo en función de las necesidades, y a la edición del tiempo entre estados deseado.
- Create_new_work_cycle: esta activity está creada con el fin de dar la posibilidad al usuario de crear un ciclo de trabajo completamente nuevo, estado a estado, y guardarlo con el nombre que desee para su posterior uso. A medida que se van agregando estados, estos se van mostrando en esta misma activity, en una lista ordenada de los mismos.
- Create_new_state: esta activity es la que ofrece al usuario la capacidad de gestionar el último detalle de todo ciclo de trabajo: los estados individuales de cada módulo led para un determinado estado. Así, le da la opción al usuario de definir qué módulos desea encender del panel luminoso de 8 leds, y guardar así cada uno de los estados constituyentes del ciclo de trabajo que desea crear.

Pasaremos ahora a hacer una descripción más detallada de cada una de estas activity, desglosando las partes más destacables del código que sustenta cada una, así como las funciones y métodos detrás de cada uno de los elementos que nos muestra el layout. Por otro lado y con el fin de evitar repetir explicaciones, todas las activity tienen ciertos métodos comunes además de los propios del ciclo de vida de la activity:

- init(): este método es llamado por el método onCreate y sirve para definir las variables que serán usadas para atender a los diferentes elementos de los layouts de cada activity. En caso de haber botones, se hace una llamada al método *prepare_buttons().*
- prepare_buttons(): este método se usa con el único fin de modularizar los trabajos a realizar. En él, se declaran sobre las variables ligadas a botones la necesidad de atención a estas variables debido a que anuncian eventos con tratamientos específicos.
- onClick(): en este método se declara cómo debe tratarse cada uno de los eventos provocados al pulsar en elementos con "*escuchas"* sobre ellos, como los botones. Constan de un switch en que se diferencian los eventos en función del id del elemento que los anuncia, y tantos case como se necesitan, donde se definen las acciones a realizar para tratar el evento disparado.

5.4.3 *Accessible_devices:*

La primera activity es *accessible_devices*, ligada al layout de mismo nombre y encargada de buscar y listar los dispositivos al alcance del smartphone, es decir, los servidores BLE encontrados y a los que se puede acceder como cliente.

Esta pantalla contiene una "vista lineal" (linear layout) central, con la propiedad de hacer "scroll" en ella para poder contemplar todos los dispositivos encontrados. Así mismo, contiene 2 botones, uno para realizar una nueva búsqueda de servidores BLE a los que poder conectarnos, y otro que nos lleva a un video en youtube en el que se explica el funcionamiento de la misma.

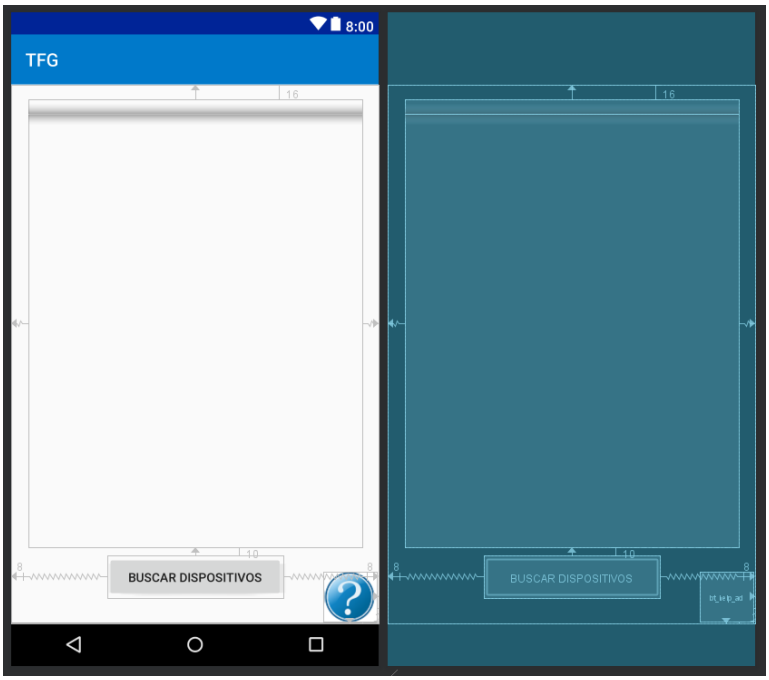


Ilustración 51: Layout accessible_devices

Cabe decir que el botón de ayuda en youtube se encuentra en todas las activity, en la misma posición dentro del layout, con el fin de facilitar al usuario su localización. Sin embargo, esto provoca la necesidad de dar a cada uno de los botones "*help*" de cada activity un *id* (identifier) diferente, con el fin de que durante la ejecución del programa se haga uso del correcto.

Por ello, todos los elementos de cada activity llevan en su sobrenombre un acrónimo referente a la activity a la que pertenecen, siendo “ad” el de “accessible_devices”, de modo que todos los elementos del activity tienen un id de la forma “elemento_nombre_ad”, siendo “elemento” otro acrónimo que hace referencia al funcionamiento del mismo.

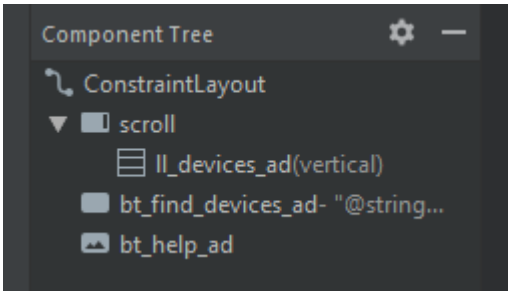


Ilustración 52: Elementos del layout accessible_devices

Si pasamos ahora a la parte central del activity, vemos que su código está estructurado en torno a la búsqueda de dispositivos BLE que hagan las veces de servidor. Para ello, en el método onCreate se llama a init(), desde donde se lanza una búsqueda nada más arrancar la app.

Esto nos genera un flujo de datos referentes a los diferentes dispositivos que se anuncian en la red BLE, de modo que el método *find_devices* recoge las señales emitidas por y usa un objeto de la clase *BroadcastReceiver* con mismo nombre, encargado de leer los datos emitidos por este para su identificación en la red.

Con ellos, se crea un objeto del tipo BluetoothDevice, que añadimos a una lista ordenada (Array List) creada previamente por init(), con el nombre “*devices”*. Así mismo, se añade este dispositivo encontrado usando el layout *item_lv_view_device* como plantilla, rellenando sus campos con los datos correspondientes.

Esto nos genera una vista, que es un elemento (item) que podemos mostrar en el *linear layout*, asignándole además propiedades, como por ejemplo la posibilidad de asignar un *onClickListener*, que nos permitirá conocer si se ha pulsado sobre ese elemento del linear layout. Definiremos las acciones a realizar por dicho *onClickListener* sobre el elemento que contiene, es decir, el dispositivo que se muestra a través de la vista creada.

Realizado todo esto, se pasa a la búsqueda de dispositivos de forma activa, pues ya está definida la forma en que se deben tratar los dispositivos

encontrados. Para ello, se hace uso del objeto de la clase *BluetoothAdapter* y se inicia la búsqueda que, con el fin de evitar se prolongue de manera indefinida (con la cantidad de energía que consume), será finalizada tras 10 segundos.

Como se ha comentado con anterioridad, esta activity cuenta con un botón orientado a realizar una búsqueda activa de dispositivos a petición del usuario. Este evento arranca cuando se pulsa en el botón, lo que provoca la cancelación de cualquier búsqueda que pueda estarse realizando, borra la lista de dispositivos que había antes de la pulsación, y limpia el linear layout en que estos se mostraban, dejando todo preparado para la llamada al método *find_devices* con el fin de hacer una nueva búsqueda.

Finalmente, hemos de tener en cuenta las transiciones con esta activity, siendo 2 los casos posibles:

- Ir hacia la activity *device_view*: al pulsar sobre cualquiera de los dispositivos existentes en el linear layout
- Volver desde la activity *device_view*: al pulsar el botón *"back"* de la activity *device_view*.

Estos casos tienen tratamientos diferentes, puesto que ir hacia la otra activity requiere de un intento y establece un fin: interactuar con el dispositivo. Además, esa nueva activity es la encargada de establecer la conexión entre los dispositivos cliente y servidor, lo que hace que requiera de los datos referentes al dispositivo seleccionado.

Por otro lado, volver de la activity significa que ya hemos acabado de trabajar con ese dispositivo, deshaciendo la conexión entre cliente y servidor (en caso de que se hubiese llegado a establecer), ya sea por el deseo de cambiar el dispositivo con que se desea realizar la conexión, sea por otras causas.

Así se lanza el método *onActivityResult*, que en función del valor que devuelve la activity *device_view* antes de que la app pase a *accessible_devices*, ejecutará unos trabajos u otros, siendo estos una nueva búsqueda si el resultado es correcto, o esperar a nuevas órdenes del usuario en otro caso.

5.4.4 *Device_view:*

Esta activity puede comunicarse con 3 activity diferentes:

- Accessible_devices: activity desde la que llegamos y que nos pasa los datos del dispositivo bluetooth elegido por el usuario con el fin de disponer de los medios necesarios para conectarnos con él.
- View_cycle_on_work: activity que, mediante el nombre del ciclo de trabajo en cada instante, permite visualizar todos los estados del mismo en el orden de ejecución, con el fin de que el usuario de la app tenga una idea clara de cómo es el ciclo mostrado a través de la señal luminosa.
- Conf_work_cycle: activity que nos permite configurar una nueva forma de controlar la señal, pudiendo definir a través de ella los parámetros de ciclo de trabajo y tiempo entre estados.

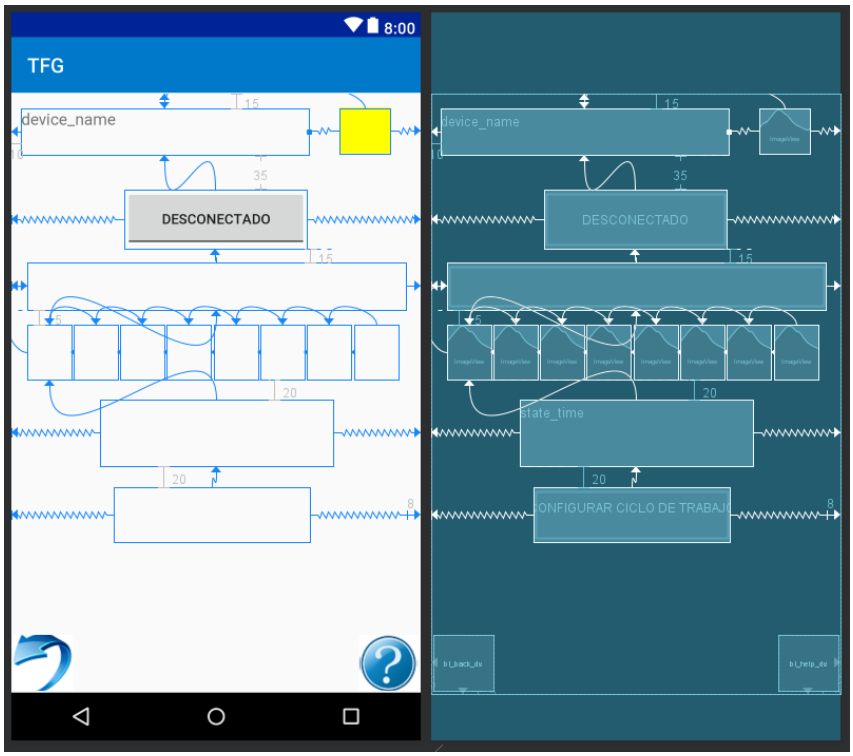


Ilustración 53: Layout device_view

Cabe tener en cuenta que esta pantalla evoluciona en función de la existencia o no de conexión entre el smartphone y el servidor, puesto que la existencia de dicha conexión provoca el flujo de información base para el control de todo nuestro sistema.

Así, cuando el usuario llega a esta pantalla tan solo ve los botones de ayuda y atrás, un *toggle button* referido al estado de conexión entre los

dispositivos, el nombre del dispositivo y una imagen que igualmente indica el estado de la conexión. Sin embargo, el árbol de componentes muestra que hay muchos más elementos:

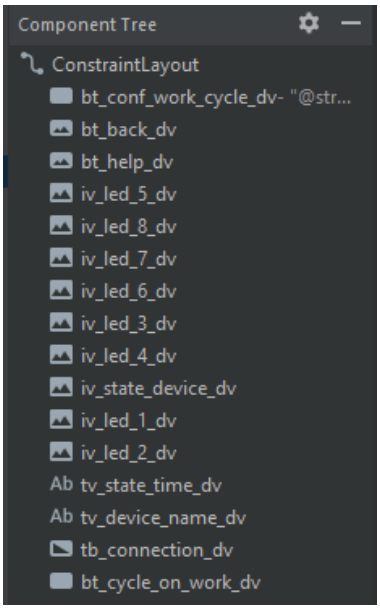


Ilustración 54: Elementos del layout device_view

La diferencia entre los que ya se ven y los que no está en el atributo *visibility* de los mismos, predefinido como "invisible" en el layout, pero que será modificado a "visible" cuando exista conexión entre los dispositivos y puedan recibirse por lo tanto los datos necesarios para rellenar estos campos.

```
android:id="@+id/iv_led_5_dv"
android:layout_width="45dp"
android:layout_height="55dp"
android:layout_marginStart="2dp"
android:background="@color/button_gone"
android:contentDescription="Show the state of the led pannel x"
android:visibility="invisible"
app:layout_constraintStart_toEndOf="@+id/iv_led_4_dv"
app:layout_constraintTop_toTopOf="@+id/iv_led_4_dv"
app:srcCompat="@color/button_gone"
tools:srcCompat="@color/button_gone" />
```

Ilustración 55: Detalle del código del layout device_view

Así, una vez conectados con el controlador de la señal luminosa, aparecen nuevos elementos:

- Botón *cycle_on_work*: que muestra el nombre del ciclo de trabajo en uso actual y permite acceder a la activity *view_cycle_on_work*.

- Botón *conf_work_cycle*: que nos lleva a la activity de mismo nombre para poder editar el trabajo desarrollado por el controlador sobre el panel.
- Text view *state_time*: que nos muestra el tiempo entre estados con que se está trabajando. Se da en milisegundos.
- Image view: referentes a los 8 módulos led del panel luminoso, que se muestran en gris si están apagados o en amarillo si están encendidos, en función de los datos que hay en la característica correspondiente del servicio BLE ofrecido por el servidor.

Dado que esta activity es en realidad la que mayor carga de trabajo realiza y en la que se editan los elementos que intervienen en la comunicación, conviene también conocer cómo trabaja, los diferentes métodos editados para que dicha comunicación funcione como nosotros deseamos. Así, en esta activity se ha declarado un objeto de la clase BluetoothGattCallback, con el nombre de *generalGatt*, para el que se han editado métodos con el fin de gestionar sus utilidades. Así, vamos a diferenciar 4 métodos dentro de este objeto:

- onCharacteristicWrite: este método está relacionado con la propiedad *write* de las características, de modo que permite la edición de los valores de las mismas en el servicio. Dado que se ha planteado el uso de 2 características con propiedad e escritura (como se ha justificado con anterioridad), a este método se le debe invocar indicando el GATT a usar y la característica que deseamos editar.
- onCharacteristicRead: este método hace uso del GATT para acceder a la característica indicada, con la propiedad de lectura, y realizar la acción de lectura del valor almacenado en dicha característica. Con el fin de que las lecturas de datos se hagan de manera cíclica, este módulo se llama a sí mismo enviando nuevos valores sobre las características que deben leerse en cada momento en función de cuánto varían estas durante el ciclo de trabajo. Esto también provoca alteraciones de los elementos que muestran los datos referidos en dichas características, como por ejemplo la variación de estados de los diferentes leds a lo largo de los diferentes estados del ciclo de trabajo.
- onServicesDiscovered: este método se encarga de leer los UUID de las características existentes en los servicios de un GATT, de modo que los almacenamos en un arraylist con el fin de poder acceder a ellos y gestionarlos de forma individual según los datos presentes en las distintas características.

- onConnectionStageChange: método que, a través del objeto de la clase BluetoothGatt y los valores de estado del GattCallback (status y newState) se encarga de gestionar la visibilidad de los elementos antes especificados, así como de hacer llamadas a los métodos anteriores, a fin de descubrir los servicios disponibles en el servidor y las características constituyentes de los mismos.
  En caso de que el estado sea "conectado" se lanza un hilo encargado de leer la primera característica de interés: el nombre del ciclo de trabajo que el dispositivo está reproduciendo, pues por cómo se ha definido el ciclo de lectura, esto provocará el arranque del ciclo de lectura del resto de características.
  En caso de que el estado del dispositivo sea "desconectado", también se gestiona la invisibilidad de los elementos que, al no existir conexión, quedarían inútiles por la falta de datos que mostrar a través de ellos.

Teniendo todo esto en mente, podemos entender el funcionamiento de esta activity, basado en el estado de conexión entre los dispositivos cliente y servidor de la red BLE conectados.

Cuando esta activity es lanzada por la función *startActivityForResult* a través del objeto *intent* de accessible_devices, el método *onCreate* hace uso de la función *init()*, que además de declarar los elementos del layout, recoge los datos enviados por la activity padre (accessible_devices) a través de la función *take_data()*. También crea el arraylist en que se almacenarán las características disponibles en el servicio ofrecido por el servidor.

Realizado esto, la activity queda a la espera de una conexión con el dispositivo, que requerirá de la gestión del objeto de la clase *BluetoothGattCallback*. Dicha conexión se gestiona una vez pulsado el toggle button "*Desconectado"*, que es el único que funciona haciendo uso de la función *onCheckedChangeListener* en vez de la función *onClickListener*.

La función *onCheckedChangeListener* nos permite editar el método *onCheckedChanged*, que en función del estado del toggle button asociado a ella, permite realizar unas u otras acciones.
Así, cuando se acciona el toggle button Y mientras no se ha establecido conexión entre los 2 dispositivos, se pasa el texto mostrado en el toggle button a "*Conectando…"*, y se llama a un manejador bluetooth (*BluetoothManager*), que toma servicios de bluetooth.

Se hace así uso de un objeto de la clase *BroadcastReceiver*, cuyo método *onReceive* se edita de modo que cada vez que se encuentre un dispositivo bluetooth, tome de este los datos con que lo diferenciaremos del resto, con el fin de establecer conexión con el dispositivo deseado. La necesidad de esto es que pueden coexistir (y de hecho en este proyecto ocurre) diferentes dispositivos que implementes servicios que usen las mismas características de igual forma, de modo que dicha características y servicios comparten su UUID. Esto requiere que se diferencie entre los dispositivos a través del nombre del propio dispositivo como método de filtrado de los dispositivos similares pero con los cuales no se desea establecer conexión. En el caso de nuestro programa, se usa tan solo como una comprobación.
Si todo esto es correcto, se llama a la función *readStuff()*, que se encarga de parar la búsqueda de otros dispositivos y conectarse con el que se indica, cambiando de nuevo el texto del toggle button a "Conectado" y disparando el generalGatt.

Por otro lado si estábamos conectados y se pasa a desconectar, se gestiona la invisibilidad de los elementos correspondientes.

Como se ha dicho anteriormente en este apartado, esta activity está relacionada con 3 más. Si bien las interacciones se limitan a el paso de parámetros o datos entre ellas, hemos de tener en cuenta cuáles son esos datos requeridos por cada una de las activity, así como la forma en que se accede a ellas.
Para con la activity *accessible_devices* sólo hay un flujo de datos referente al dispositivo con que trabajar, pues los "datos de vuelta" como *result* del activity solo hacen referencia a si todo ha ido bien o no. Al volver hacia la activity *accessible_devices*, la activity *device_view* se saca del stack mediante el método *finish().*

Por otro lado, los datos entre *device_view* y *view_cycle_on_work* es similar al que hay entre *accessible_devices* y *device_view*, ya que solo se envían datos de referencia con que realizar un cierto trabajo en la activity inicializada.
Sin embargo, cuando pasamos desde *device_view* a *conf_work_cycle*, el start activity sí se realiza con el fin de obtener unos datos de vuelta. Esto se debe a que toda la conexión y trabajo con las características del servicio se realiza en *device_view*, mientras que la generación de los datos que se desean escribir en dichas características se hace en *conf_work_cycle*. Esto provoca que, aunque no se envíen parámetros a *conf_work_cycle*, esta activity sí nos devuelva unos parámetros. Estos se tratan en el método

onActivityResult, de modo que en caso de que el resultado sea OK, se recogen los datos (mediante la función *take_data_for_new_conf*) para escribirlos en las características del servicio, mientras que si el resultado es CANCELED, tan solo se mantiene la lectura de las características con dicha propiedad.

5.4.5 *View_cycle_on_work:*

La función final de esta activity ya se ha mencionado con anterioridad, y dado que la forma en que obtiene los datos a mostrar en el linear layout central de la misma es similar a cómo se hace en la activity *conf_work_cycle* (más compleja y de mayor interés en lo que a trabajo se refiere), carece de sentido explicar aquí eso, debido a que sería contar 2 veces exactamente lo mismo.

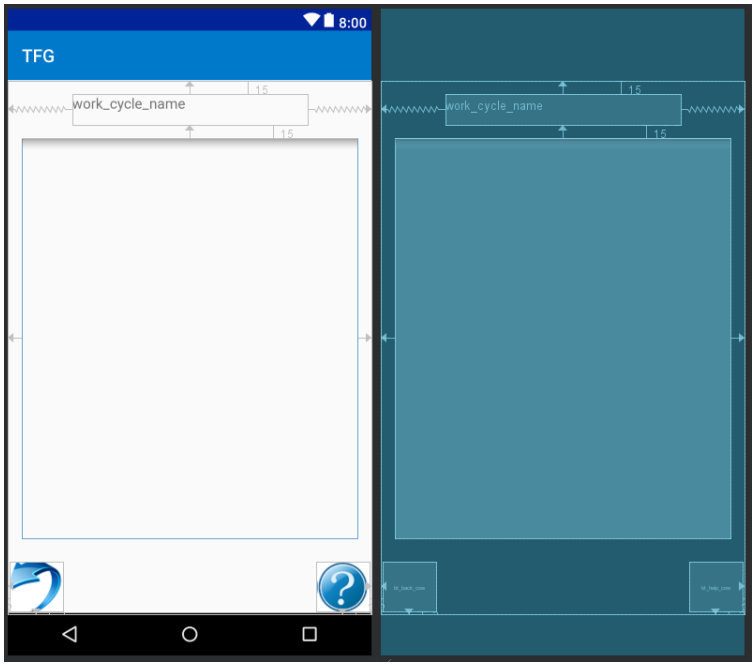


Ilustración 56: Layout view_cycle_on_work

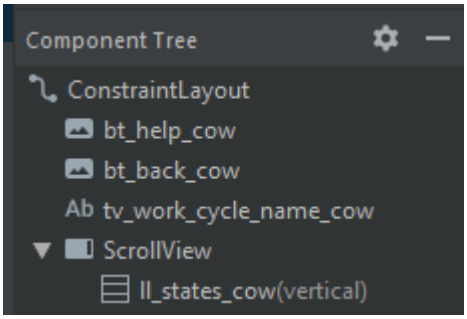


Ilustración 57: Elementos del layout view_cycle_on_work

Merece la pena conocer que el linear layout de este activity se apoya, al igual que hace *accessible_devices* en otro layout (*item_ll_view*_state) que hace las veces de item de este con el fin de mostrar el estado de encendido o apagado de cada uno de los módulos led de la señal luminosa en cada estado del ciclo de trabajo.

Cuando desde esta activity se vuelve a *device_view*, dado que es meramente informativa, simplemente se hace uso del método *finish()*.

5.4.6 *Conf_work_cycle:*

Esta activity, cuya funcionalidad base ya se ha comentado con anterioridad (aunque el nombre es auto explicativo), consta principalmente de 2 grandes linear layout, estando el segundo bajo la acción del primero.

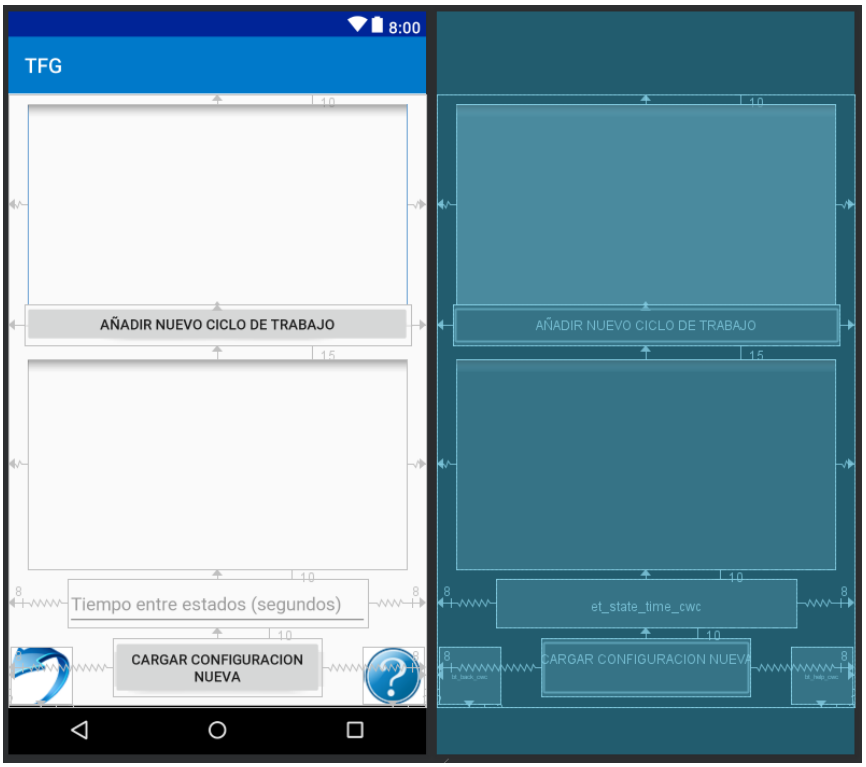


Ilustración 58: Layout conf_work_cycle

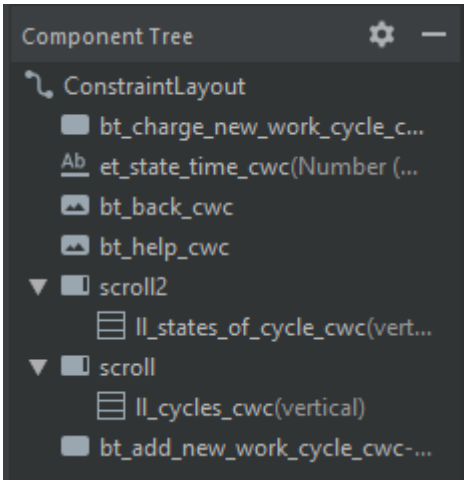


Ilustración 59: Elementos del layout conf_work_cycle

Ambos linear layout cuentan con la opción de scroll, y son rellenados con items conformados a partir de otros layouts (*item_ll_list_work_cicles* e *item_ll_viesw_state*).

El primero de los linear layout sirve para mostrar los diferentes ciclos existente en la app y que podemos decidir cargar en controlador. Para ello, se hace uso de la función *fill_linear_layout_cycles()*, donde se hace una lista con los archivos xxxx.txt existentes en la app, pues todos ellos son ciclos de trabajo.

Así mismo, se crea un item para cada uno de estos ciclos, definiéndose la posibilidad de seleccionar cada uno de estos items de forma similar a los dispositivos en la activity *accessible_devices*, de modo que la activity pueda conocer cuál ha sido seleccionado por el usuario y debe ser mostrado en el segundo linear layout (para que el usuario pueda seleccionar así el ciclo que más le conviene visualizando los estados constituyentes de cada ciclo existente según va eligiendo uno u otro).

Para rellenar el segundo linear layout, cuyo funcionamiento es similar al de la activity *view_cycle_on_work*, primero se carga el fichero seleccionado en una lista ordenada, de modo que cada estado del ciclo de trabajo pasa a ser 1 elemento de dicha lista, constituido por 8bits, cada uno haciendo referencia a si el módulo led con que se corresponde está encendido (1) o apagado (0) en ese estado del ciclo.

Una vez cargada la lista, es recorrida elemento a elemento, creando una vista por cada uno de ellos según la forma definida en el layout *item_ll_view_state*, de modo que cada uno de los 8 bits definen el color del image view con que se muestra el estado cada módulo led de la señal luminosa. También se rellenan los campos de información sobre el número de cada estado, a fin de que el usuario sepa el orden exacto en que se mostrarán estos.

De forma similar a cómo se gestionan los archivos en el módulo receptor, aquí son tratados a través de objetos de la case InputStreamReader, con las propiedades adecuadas para su uso, y leídos línea a línea, para almacenar cada una como un elemento de la lista (tal y como se ha indicado anteriormente).

Los otros elementos de este activity son un edit text editable, en que el usuario puede escribir el tiempo entre estados que desea, *en milisegundos* para la nueva configuración del trabajo, y un botón con el que verificará que los elementos elegidos en el instante que es pulsado son los deseados para la nueva configuración del sistema y un botón que nos lleva a otra activity (*create_new_work_cycle*), donde podremos definir un nuevo ciclo de trabajo si ninguno de los disponibles convence al usuario.

Si bien el edit text es un elemento "casi pasivo" del que solamente recuperamos los datos introducidos por el usuario, el botón *Configurar Ciclo* tiene una mayor interacción.

Cuando se pulsa este botón se toman los datos de las seleccionados, de modo que se puedan hacer con ellos los paquetes de datos que enviaremos al módulo receptor, es decir, aquellos que la activity *device_view* escribirá en las características. Hemos de recordar que, dado que se usan 2 características y se desean enviar 3 datos, los paquetes tienen un formato predefinido (por la forma en que los espera el receptor para su correcto tratamiento). Así, se crean estos paquetes siguiendo el patrón previamente acordado para los mismos y se pasan a la activity que los escribirá en las características.

Finalmente, el botón *Añadir nuevo ciclo de trabajo* nos lleva a la activity antes mencionada a través del método ya conocido *startActivityForResult()*, del que esperamos un nuevo ciclo de trabajo (lo que provocaría que se deba recargar el primer linear layout de este activity)

en caso de que el usuario acabe creando uno, o nada, en caso de que en algún momento del proceso decida desechar ese nuevo ciclo en creación.

5.4.7 *Create_new_work_cycle:*

Esta activity, es en realidad un paso intermedio para la creación de un ciclo, pues si bien permite visualizar cada uno de los estados constituyentes del nuevo ciclo que se está creando a través de ella, es mediante la activity *create_new_state* (accesible desde esta a través del botón *"Añadir nuevo estado"*) como se definen los estados individuales del ciclo, editando qué módulos led se requiere que estén encendidos o apagados para ese estado.

Así, esta activity consta (principalmente) de un edit text, para que el usuario pueda definir el nombre con que desea guardar el ciclo de trabajo que está creando, un linear layout en el que se mostrarán (de forma similar a activitys anteriores) los estados constituyentes del ciclo, y varios botones, orientados a cerrar y almacenar el ciclo en construcción (GUARDAR), descartarlo (DESCARTAR) y añadir un nuevo estado (añadir nuevo estado).

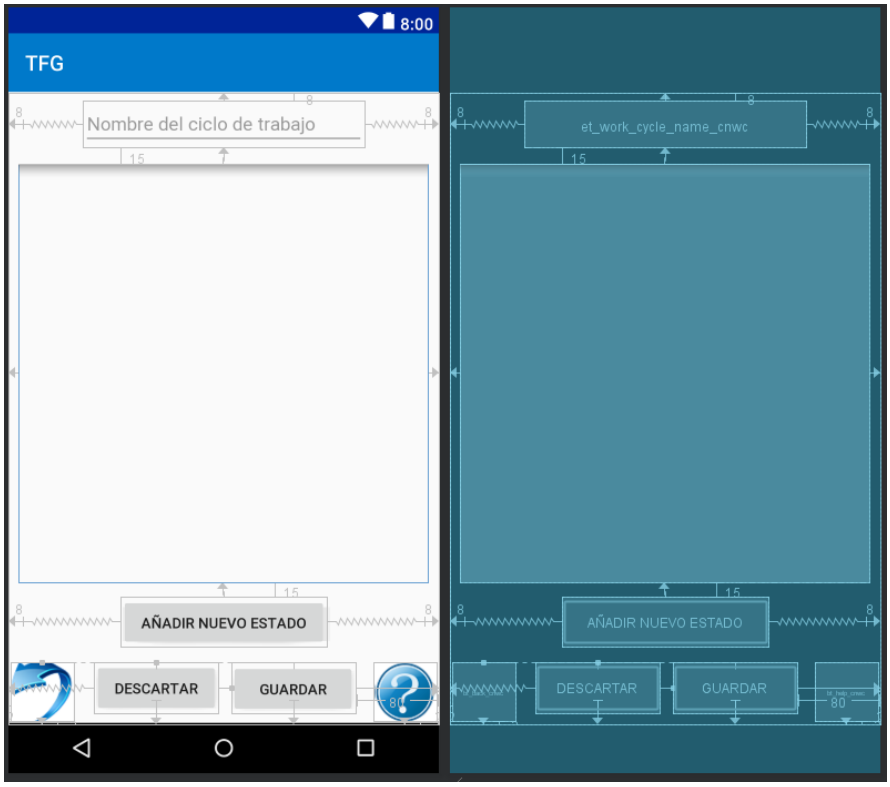


Ilustración 60: Layout create_new_work_cycle

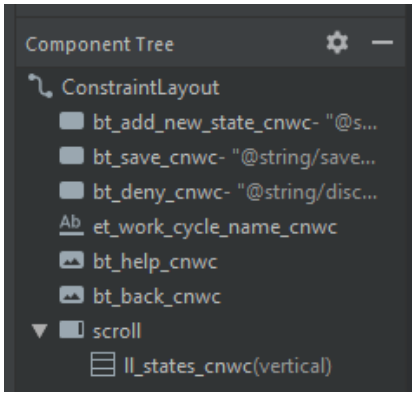


Ilustración 61: Elementos del layout create_new_work_cycle

La mayor novedad de esta activity es que, debido a que los estados se van almacenando en un fichero de texto, este debe ser pasado a *create_new_state* para que pueda añadir los nuevos estados, de modo que en *create_new_work_cycle* solamente se gestiona la lectura del mismo y su exposición a través de items en el linear layout (con la forma de *item_ll_state_view*).

5.4.8 *Create_new_state:*

Esta última activity nos muestra el elemento constituyente más pequeño de control de nuestro sistema: un estado del panel luminoso.

Así, a través de los 8 toggle buttons podemos definir so deseamos que un módulo del panel se encienda o permanezca apagado en un cierto estado del ciclo de trabajo.

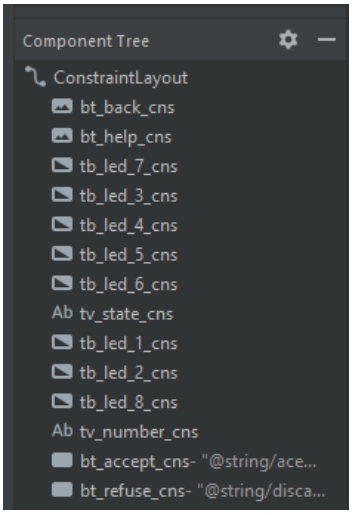


Ilustración 62: Elementos del layout create_new_State

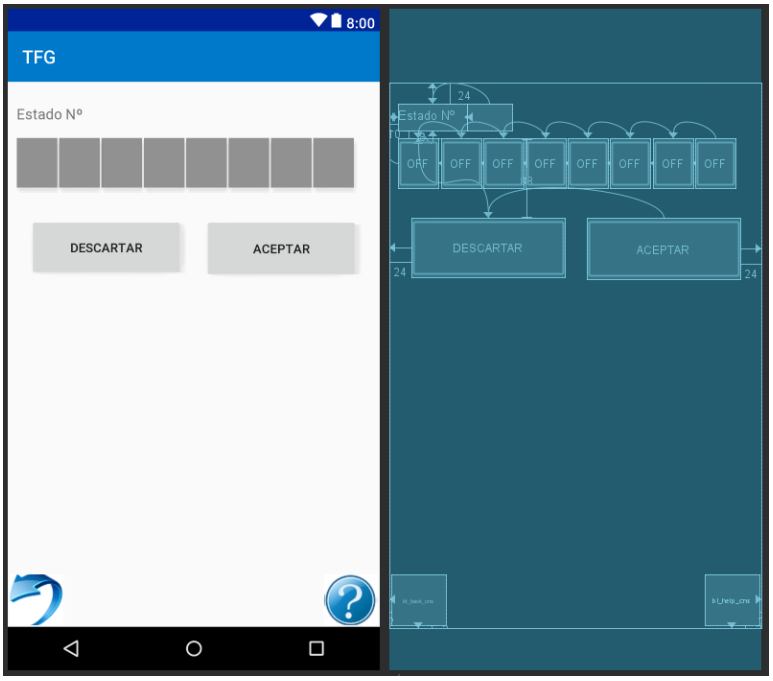


Ilustración 63: Layout create_new_state

En esta activity también disponemos de 2 botones para decidir si el estado debe ser descartado (DESCARTAR) o almacenado (GUARDAR), lo que conlleva la escritura del valor de estado de los toggle buttons en el fichero de texto, cuyo nombre nos envía la activity *create_new_work_cycle*, a fin de actualizar el fichero a medida que se van añadiendo estados.

5.4.9 *Otros layouts:*

Como se ha mencionado en puntos anteriores, hay activitys que usan varios layouts, a modo de plantillas con que rellenar campos de diversos elementos. En el caso de esta app, si bien hay 6 activitys, se hace uso de 9 layouts diferentes, puesto que existen 3 orientados a su uso como vistas base en linear layout o listview de otras activitys.

Vamos a echar un breve vistazo a estos layout:

- Item_lv_view_device: consta de 3 campos, aunque uno de ellos (un text view) es invisible, pues su utilidad se centra en proporcionar la capacidad de indexar el elemento que use este layout como patrón de visualización. Los otros 2 son un image view, para proporcionar un espacio en que mostrar un color que haga referencia al estado de conexión entre cliente y servidor, y un text view en que se

mostrará el nombre del dispositivo que contiene el servidor (es decir, el módulo receptor basado en ESP32).

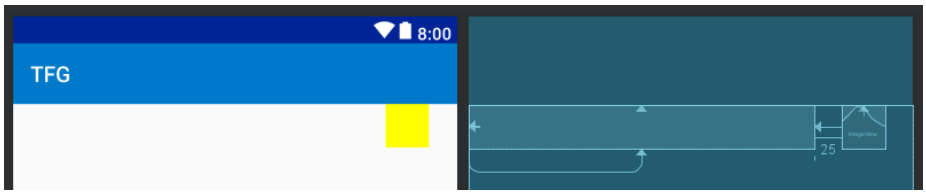


Ilustración 64: Layout item_lv_view_device

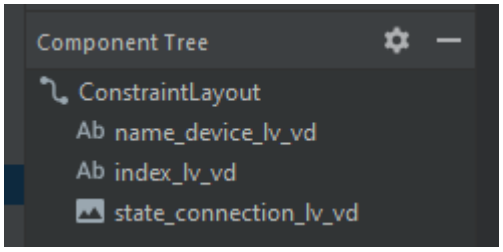


Ilustración 65: Elementos del layout item_lv_view_device

- Item_ll_work_cycles: este layout es el más simple, pues tiene una forma similar al anterior, pero eliminando el image view, ya que solo se desea mostrar el nombre de los ciclos de trabajo existentes, siendo por ello necesario sólo 2 edit text, puesto que el segundo de ellos vuelve a trabajar como índice.

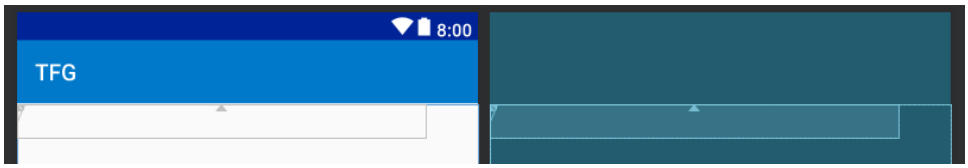


Ilustración 66: Layout item_ll_work_cycles

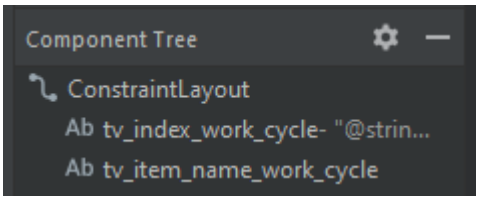


Ilustración 67: Elementos del layout item_ll_work_cycles

- Item_ll_view_state: tiene una forma similar al layout de la activity *create_new_state*, con la salvedad de que se eliminan los botones aceptar y descartar, y se sustituyen los toggle buttons por image views, en los que cargar la imagen que indique el encendido o apagado del módulo led en cada estado.

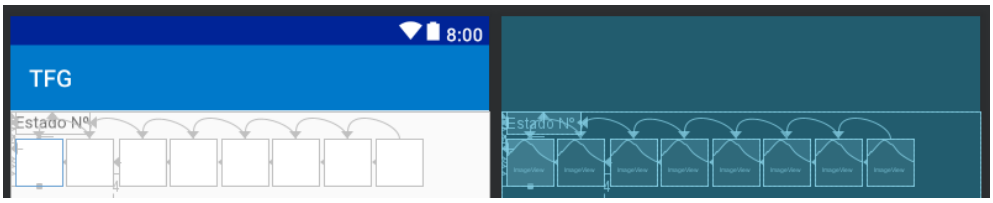


Ilustración 68: Layout item_ll_view_state

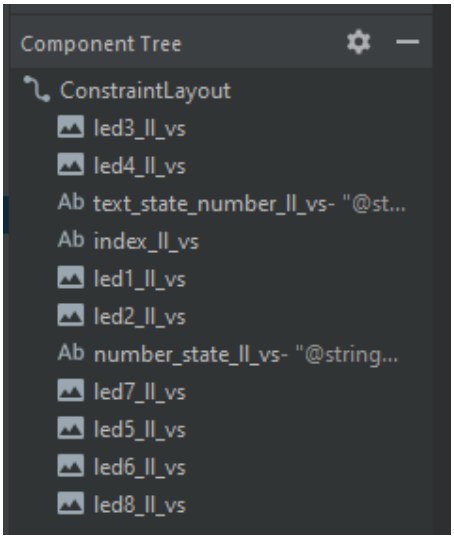


Ilustración 69: Elementos del layout item_ll_view_state

5.4.10 *Ciclos de trabajo predefinidos:*

En la app, hay varios ciclos de trabajo precargados, con el fin de que se pueda comprobar el correcto funcionamiento del sistema. Estos son algunos ejemplos de esos ciclos:

- Exterior interior: es un ciclo de trabajo que enciende de manera intermitente los módulos led exteriores (1, 2, 7 y 8) o interiores (3,4,5 7 6).

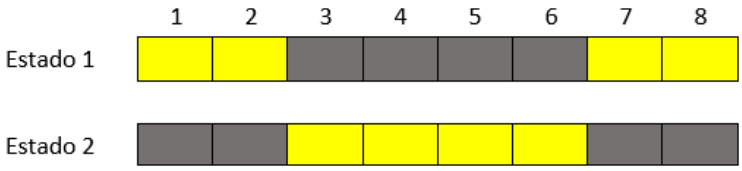


Ilustración 70: Ciclo exterior-interior

- Todo nada: este ciclo de trabajo hace parpadear toda la señal luminosa, encendiendo o apagando los 8 módulos led en cada estado del ciclo.

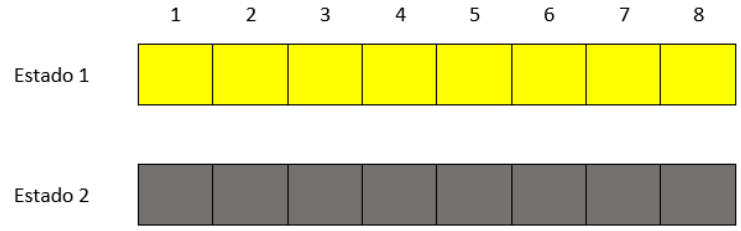


Ilustración 71: Ciclo todo-nada

- Izquierda derecha: mantiene encendida la mitad del panel luminoso que corresponde en cada momento, haciendo que permanezcan siempre 4 módulos led encendidos y 4 apagados.

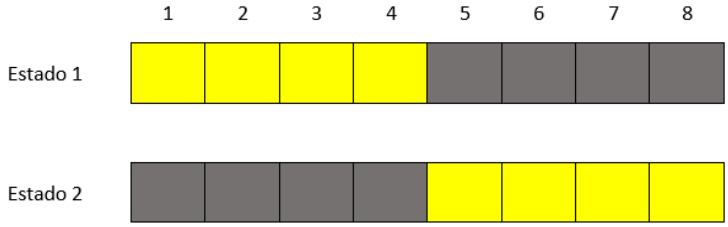


Ilustración 72: Ciclo izquierda-derecha

- Pares impares: enciende de manera alternativa los módulos led pares e impares.

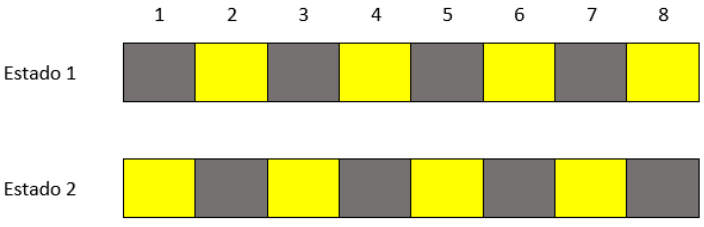


Ilustración 73: Ciclo pares-impares

6 Conclusiones y líneas futuras de trabajo

6.1 Conclusiones:

Recordando los objetivos que se fijaban al principio de este proyecto, se pretendía crear un sistema de control de un panel luminoso con el cual pudiésemos minimizar el cableado, lograr reprogramar a necesidad los paneles led en cada instante y controlar el tiempo entre cambios de estado en los paneles.

Si lo tomamos por partes:

- Se ha desarrollado un dispositivo receptor mediante tecnología BLE, lo que permite la comunicación entre controlador y panel sin hilos para las señales (descontando los necesarios para alimentación, obligados por el panel luminoso).
- Se ha creado una aplicación Android capaz de funcionar en cualquier dispositivo con una versión "Android Oreo" (2017) o superior, con la que comunicarnos con el módulo receptor.
- La app android permite crear nuestros propios ciclos de trabajo, en los que definimos los estados de los paneles led de la señal luminosa en cada momento y guardarlos, logrando así flexibilidad en cuanto a los mensajes a mostrar.
- Continuando con la app, esta nos permite fijar con una elevada precisión el tiempo que deseamos que permanezca la señal luminosa en cada estado.
- El uso de un dispositivo móvil evita la estaticidad de las centralitas actuales y facilita la comparación de lo mostrado en la señal luminosa con lo mostrado en el dispositivo de control y visualización (smartphone) para verificar el correcto funcionamiento del sistema.
- El elemento receptor tiene un menor consumo que una centralita, incluso durante el envío de datos al dispositivo maestro estando ambos conectados entre sí.

Teniendo en cuenta todo esto, podemos concluir que se han conseguido los objetivos fijados inicialmente, mejorándose ciertos aspectos que, si bien podrían haberse mantenido tal y como están en la actualidad, no está de más mejorar. Por contrapartida, debido a los tiempos de refresco, no puede lograrse un control que mantenga una visualización sincronizada entre lo que muestra el panel y lo que se muestra en el smartphone para tiempos entre estados por debajo de los 0.35 segundos.

6.2 Vías futuras de trabajo y desarrollo:

Si bien la posibilidad de comunicarnos con un módulo receptor por vez como máximo limita el control que tenemos en sistemas con más de un receptor, el módulo ESP32 Devkit v1 puede programarse igualmente como maestro, e incluso funcionar como cliente de otro módulo que haga las veces de servidor.

Esto, permitiría mejorar varias cosas:

- Distancia máxima de comunicación: debido al bajo consumo de estos módulos en sí mismos, no es inviable colocar varios de modo que, podamos mezclar la tipología estrella propia de la comunicación Bluetooth, con una cadena de transmisión entre varios módulos, que pasarían a hacer las veces de repetidores, emitiendo/recibiendo datos. Cabe decir que este trabajo requeriría de una fuerte sincronización de todo el sistema
- Control de varios receptores por vez: siguiendo el punto anterior, y aprovechando la capacidad de un maestro de mantener la capa de enlace con varios esclavos, podría desarrollarse un sistema en el cual, un módulo hiciese las veces de controlador general, mostrando al smartphone toda la extensión de la red creada. Así, podrían controlarse diversos puntos de la misma desde un controlador "central", conectable al smartphone.

Por otro lado, la app mostrada es también mejorable, siendo uno de los primeros cambios a tener en cuenta la portabilidad de la misma a bases de datos MSQL, de modo que puedan reeditarse los ciclos de trabajo ya existentes.

También puede resultar interesante crear una nueva activity desde la cual ver todos los dispositvicos receptores, así como las características que emiten en broadcast, de modo que se puedan visualizar los estados actuales de cada señal luminosa al mantener la conexión con varios a la vez.

Los cambios anteriores requerirían de la creación de una barra de menú deslizable para mejorar la movilidad dentro de la app al haber aumentado las funciones de la misma.

6.3 Otros posibles usos del hardware desarrollado:

La posibilidad de manejar diversos paneles luminosos desde un solo controlador es fácilmente aprovechable sin salir del entorno vial para el cual está planteado este proyecto.

Por ejemplo, de cara a la ejecución de obras en las que se necesite dar paso por una carretera en uno u otro sentido durante períodos de tiempo equiespaciados, los malentendidos entre operarios pueden provocar que se de paso a vehículos desde ambos sentidos, provocando una situación en la que pueden acaecer accidentes.

Si eliminamos de la ecuación la necesidad de comuncar a varios operarios y que estos se entiendan correctamente, se sortea la psoibilidad de provocar un accidente.

En otras palabras, si un solo operario controla ambos sentidos de circulación, es menos probable que se malinterprete a sí mismo y permita la circulación en ambos sentidos a la par.

Por otro lado, la placa de testeo desarrollada, al incluir 8 led luminosos, puede ser facilmente utilizada académicamente para visualizar el trabajo con servidores, servicios y características en una red BLE.

7 Bibliografía

1. Datasheet ESP32
   https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

2. Software | Espressif Systems
   https://www.espressif.com/en/products/software.

3. espressif/arduino-esp32, GitHub
   https://github.com/espressif/arduino-esp32

4. ESP32 – techtutorialsx,
   https://techtutorialsx.com/category/esp32/.

5. Comparación módulos ESP32 – ESP8266
   https://blog.bricogeek.com/noticias/electronica/comparativa-y-analisis-completo-de-los-modulos-wifi-esp8266-y-esp32/

6. Technical Reference Manual ESP32
   https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

7. Soporte al desarrollador Android
   https://developer.android.com/guide/topics

8. Consultas bases BLE
   https://www.silabs.com/community/wireless/bluetooth/knowledge-base.entry.html/2018/05/30/ble_basics_masters-i4n9

9. Wikipedia
   https://es.wikipedia.org/wiki/Bluetooth
   https://en.wikipedia.org/wiki/Generic_access_profile

10. Android
    https://developer.android.com/guide/topics/connectivity/bluetooth-le.html
    https://developer.android.com/training/constraint-layout/?hl=es-419

11. Adafruit
    https://learn.adafruit.com/introduction-to-bluetooth-low-energy/introduction

12. Bluetooth
    https://www.bluetooth.com

https://www.bluetooth.com/bluetooth-technology/radio-versions/
https://www.bluetooth.com/specifications/gatt/generic-attributes-overview

13. Spakfun
    https://learn.sparkfun.com/tutorials/bluetooth-basics

14. Solid gear group
    https://solidgeargroup.com/detectar-dispositivos-bluetooth-android?lang=es

15. Stack Overflow
    https://es.stackoverflow.com/questions/4447/leer-datos-desde-dispositivo-bluetooth

16. Randomnerdtutorials:
    https://randomnerdtutorials.com/esp32-pinout-reference-gpios/

17. Led Iberica:
    https://ww.lediberica.es

18. Randomnerdtutorials:
    https://randomnerdtutorials.com/esp32-pinout-reference-gpios/

19. Getting Started with Bluetooth Low Energy: Tools and techniques for low-power networking. De Kevin Townsend, Carles Cufí, Akiba & Robert Davidson.

8 Anexos

8.1 Presupuestos placas:

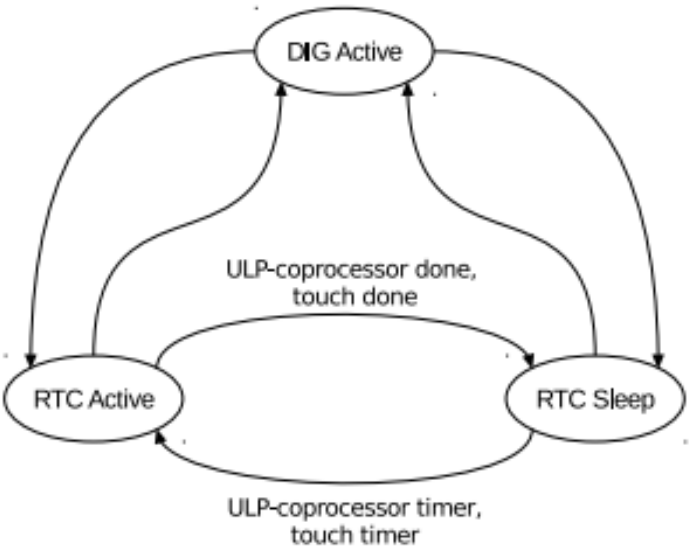
Placa FINAL conectable a panel						
Material	**Código Fabricante**	**Unidades por equipo**	**Precio unitario**	**%IVA**	**Subtotal**	**Precio CON IVA**
Modulo Devkit v1	xxx	1	7,5	1,575	5,925	7,5
Modulo micro-SD adapter	xxx	1	1,2	0,252	0,948	1,2
Tarjeta memoria micro-SD	xxx	1	7,5	1,575	5,925	7,5
Led RGB	LL-509RGBM2E-004	1	0,52	0,1092	0,4108	0,52
Modulo Buck	xxx	1	1,34	0,2814	1,0586	1,34
Tornillos M3 + separador nylon negro para sujeción	B3X4/BN1062 + HP-10	6	0,25	0,0525	1,4475	1,5
ULN2803AG	xxx	1	0,26	0,0546	0,2054	0,26
Conector jack 5mm hembra	FC681465P	1	0,2	0,042	0,158	0,2
Placa PCB 4,9 x 9,92 cm	xxx	1	2,43	0,5103	1,9197	2,43
Conector T81-1-20-R1 20 pines macho	T821120A1R100CEU	2	0,56	0,1176	1,0024	1,12
Conector T812-1-20 20 pines hembra	T812120A101CEU	2	0,37	0,0777	0,6623	0,74
			Total	4,6473	19,6627	**24,31**

Placa TESTEO leds 12v						
Material	**Código Fabricante**	**Unidades por equipo**	**Precio unitario**	**%IVA**	**Subtotal**	**Precio CON IVA**
Modulo Devkit v1	xxx	1	7,5	1,575	5,925	7,5
Modulo micro-SD adapter	xxx	1	1,2	0,252	0,948	1,2
Tarjeta memoria micro-SD	xxx	1	7,5	1,575	5,925	7,5
Led RGB	LL-509RGBM2E-004	1	0,52	0,1092	0,4108	0,52
Modulo Buck	xxx	1	1,34	0,2814	1,0586	1,34
Tornillos M3 + separador nylon negro para sujeción	B3X4/BN1062 + HP-10	4	0,25	0,0525	0,9475	1
ULN2803AG	xxx	1	0,26	0,0546	0,2054	0,26
Conector jack 5mm hembra	FC681465P	1	0,2	0,042	0,158	0,2
Placa PCB 4,9 x 9,92 cm	xxx	1	2,43	0,5103	1,9197	2,43
Leds 3mm	OSY5JA3Z74A	8	0,06	0,0126	0,4674	0,48
Resistencias	xxx	8	0,01	0,0021	0,0779	0,08
			Total	0	18,0433	**22,51**

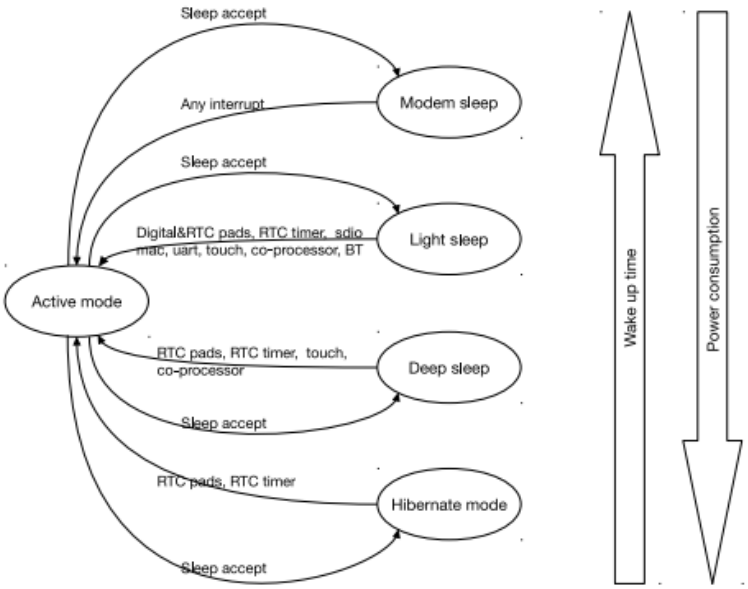
Panel leds 12v						
Material	**Código Fabricante**	**Unidades por equipo**	**Precio unitario**	**%IVA**	**Subtotal**	**Precio CON IVA**
Metacrilato 54x8,4 cm	xxx	1	7,5	1,575	5,925	7,5
Madera precortada	xxx	1	8	1,68	6,32	8
Cableado	xxx	1	0,8	0,168	0,632	0,8
Tiras led (metros)	HH-YLGI-5050FWNA	2,4	6,37	1,3377	13,9503	15,288
Canteado en plástico	xxx	1	2,5	0,525	1,975	2,5
			Total	5,2857	28,8023	**34,088**

8.2 Otras características del ESP32:

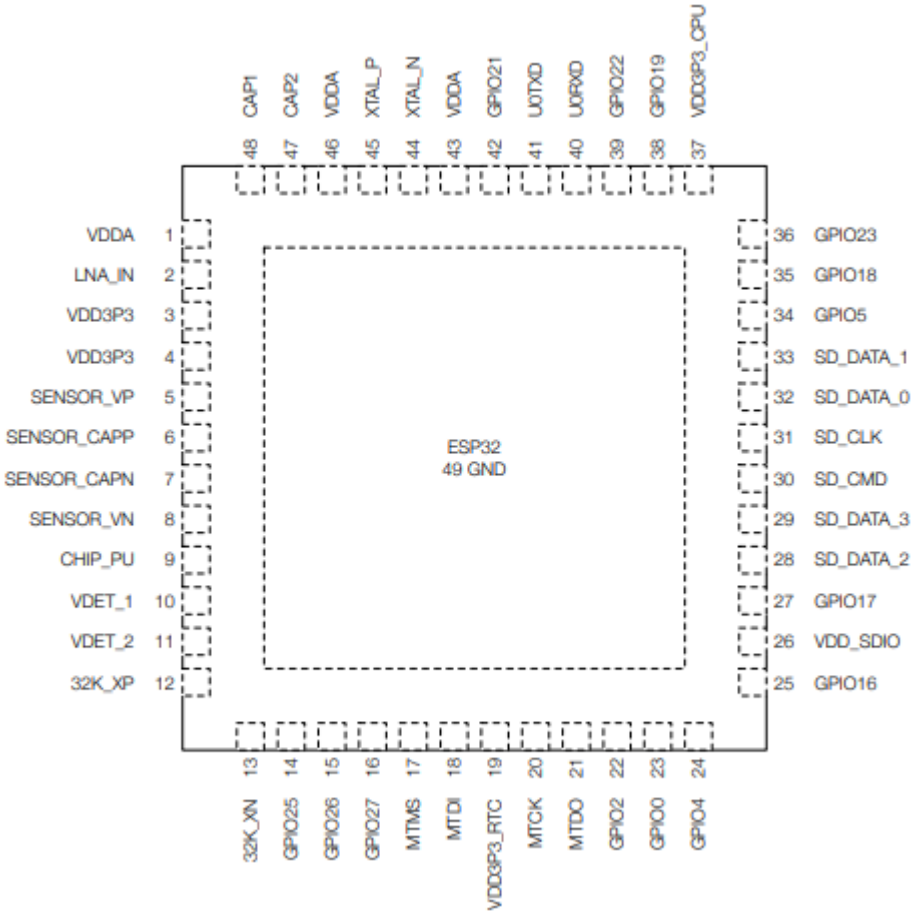
Estados del RTC del ESP32



Modos de funcionamiento del ESP32

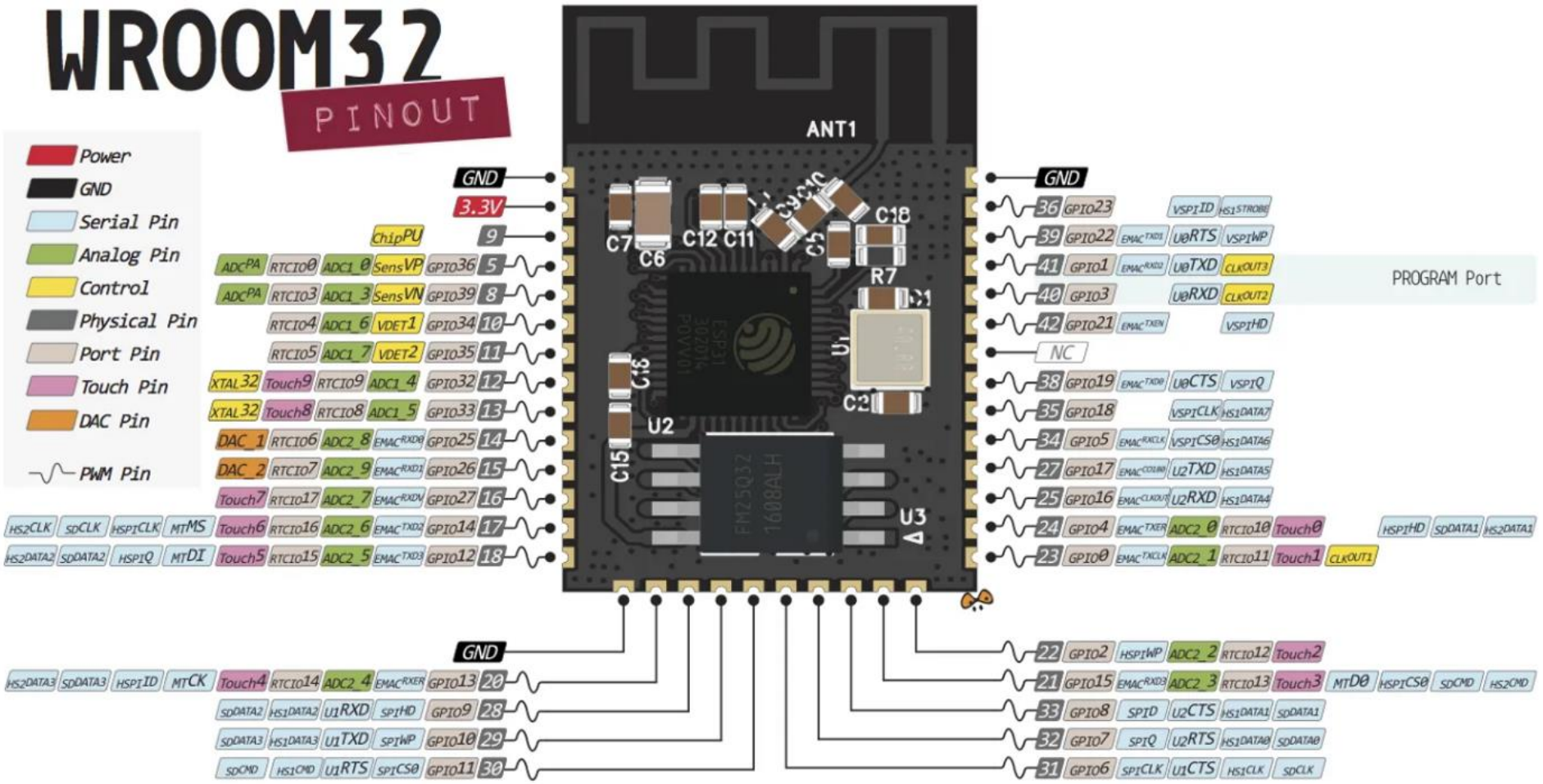


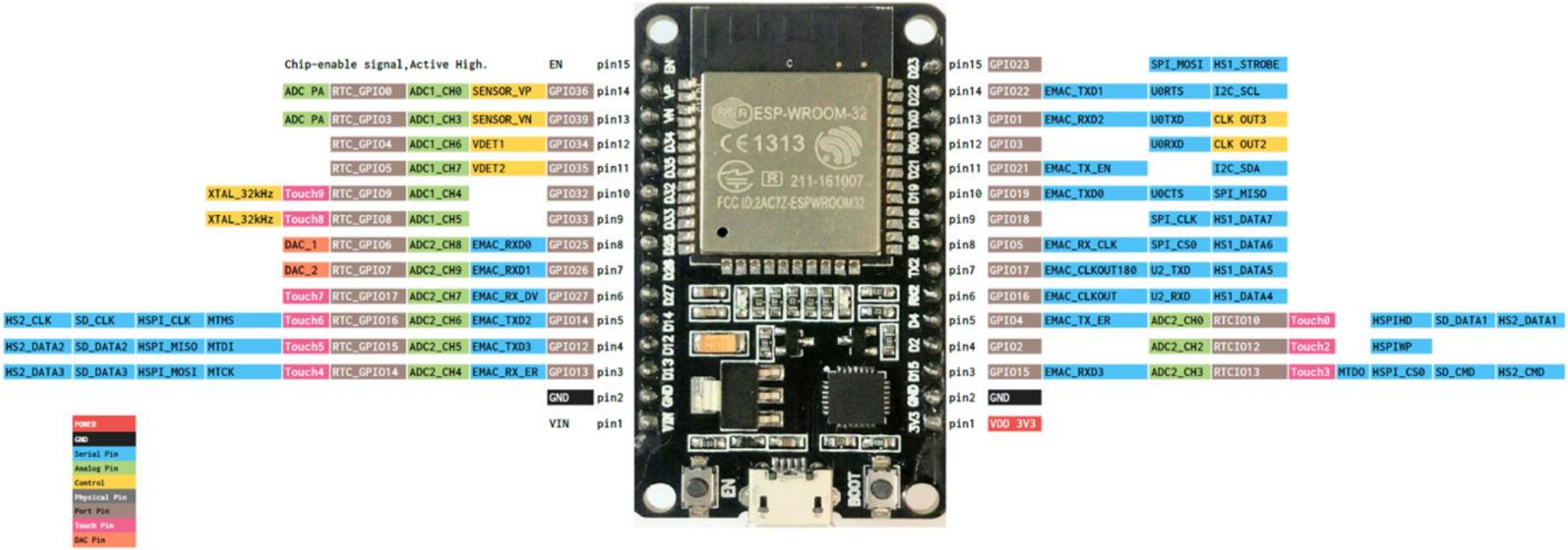
8.3 Pineado del ESP32



IO_MUX

Pin No.	Power Supply Pin	Analog Pin	Digital Pin	Power Domain	Analog Function1	Analog Function2	Analog Function3	RTC Function1	RTC Function2	Function1	Type	Function2	Type	Function3	Type	Function4	Type	Function5	Type	Function6	Type	Drive Strength (2'd2: 20 mA)	At Reset	After Reset
1	VDDA			VDDA supply in																				
2		LNA_IN		VDD3P3																				
3	VDD3P3			VDD3P3 supply in																				
4	VDD3P3			VDD3P3 supply in																				
5		SENSOR_VP		VDD3P3_RTC	ADC_H	ADC1_CH0		RTC_GPIO0		GPIO36	I			GPIO36	I								oe=0, ie=0	oe=0, ie=0
6		SENSOR_CAPP		VDD3P3_RTC	ADC_H	ADC1_CH1		RTC_GPIO1		GPIO37	I			GPIO37	I								oe=0, ie=0	oe=0, ie=0
7		SENSOR_CAPN		VDD3P3_RTC	ADC_H	ADC1_CH2		RTC_GPIO2		GPIO38	I			GPIO38	I								oe=0, ie=0	oe=0, ie=0
8		SENSOR_VN		VDD3P3_RTC	ADC_H	ADC1_CH3		RTC_GPIO3		GPIO39	I			GPIO39	I								oe=0, ie=0	oe=0, ie=0
9		CHIP_PU		VDD3P3_RTC																				
10		VDET_1		VDD3P3_RTC		ADC1_CH6		RTC_GPIO4		GPIO34	I			GPIO34	I								oe=0, ie=0	oe=0, ie=0
11		VDET_2		VDD3P3_RTC		ADC1_CH7		RTC_GPIO5		GPIO35	I			GPIO35	I								oe=0, ie=0	oe=0, ie=0
12		32K_XP		VDD3P3_RTC	XTAL_32K_P	ADC1_CH4	TOUCH9	RTC_GPIO9		GPIO32	I/O/T			GPIO32	I/O/T							2'd2	oe=0, ie=0	oe=0, ie=0
13		32K_XN		VDD3P3_RTC	XTAL_32K_N	ADC1_CH5	TOUCH8	RTC_GPIO8		GPIO33	I/O/T			GPIO33	I/O/T							2'd2	oe=0, ie=0	oe=0, ie=0
14			GPIO25	VDD3P3_RTC	DAC_1	ADC2_CH8		RTC_GPIO6		GPIO25	I/O/T			GPIO25	I/O/T					EMAC_RXD0	I	2'd2	oe=0, ie=0	oe=0, ie=0
15			GPIO26	VDD3P3_RTC	DAC_2	ADC2_CH9		RTC_GPIO7		GPIO26	I/O/T			GPIO26	I/O/T					EMAC_RXD1	I	2'd2	oe=0, ie=0	oe=0, ie=0
16			GPIO27	VDD3P3_RTC		ADC2_CH7	TOUCH7	RTC_GPIO17		GPIO27	I/O/T			GPIO27	I/O/T					EMAC_RX_DV	I	2'd2	oe=0, ie=0	oe=0, ie=1
17			MTMS	VDD3P3_RTC		ADC2_CH6	TOUCH6	RTC_GPIO16		MTMS	I0	HSPICLK	I/O/T	GPIO14	I/O/T	HS2_CLK	O	SD_CLK	I0	EMAC_TXD2	O	2'd2	oe=0, ie=0	oe=0, ie=1
18			MTDI	VDD3P3_RTC		ADC2_CH5	TOUCH5	RTC_GPIO15		MTDI	I1	HSPIQ	I/O/T	GPIO12	I/O/T	HS2_DATA2	I1/O/T	SD_DATA2	I1/O/T	EMAC_TXD3	O	2'd2	oe=0, ie=1, wpd	oe=0, ie=1, wpd
19	VDD3P3_RTC			VDD3P3_RTC supply in																				
20			MTCK	VDD3P3_RTC		ADC2_CH4	TOUCH4	RTC_GPIO14		MTCK	I1	HSPID	I/O/T	GPIO13	I/O/T	HS2_DATA3	I1/O/T	SD_DATA3	I1/O/T	EMAC_RX_ER	I	2'd2	oe=0, ie=0	oe=0, ie=1
21			MTDO	VDD3P3_RTC		ADC2_CH3	TOUCH3	RTC_GPIO13	I2C_SDA	MTDO	O/T	HSPICS0	I/O/T	GPIO15	I/O/T	HS2_CMD	I1/O/T	SD_CMD	I1/O/T	EMAC_RXD3	I	2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
22			GPIO2	VDD3P3_RTC		ADC2_CH2	TOUCH2	RTC_GPIO12	I2C_SCL	GPIO2	I/O/T	HSPIWP	I/O/T	GPIO2	I/O/T	HS2_DATA0	I1/O/T	SD_DATA0	I1/O/T			2'd2	oe=0, ie=1, wpd	oe=0, ie=1, wpd
23			GPIO0	VDD3P3_RTC		ADC2_CH1	TOUCH1	RTC_GPIO11	I2C_SDA	GPIO0	I/O/T	CLK_OUT1	O	GPIO0	I/O/T					EMAC_TX_CLK	I	2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
24			GPIO4	VDD3P3_RTC		ADC2_CH0	TOUCH0	RTC_GPIO10	I2C_SCL	GPIO4	I/O/T	HSPIHD	I/O/T	GPIO4	I/O/T	HS2_DATA1	I1/O/T	SD_DATA1	I1/O/T	EMAC_TX_ER	O	2'd2	oe=0, ie=1, wpd	oe=0, ie=1, wpd
25			GPIO16	VDD_SDIO						GPIO16	I/O/T			GPIO16	I/O/T	HS1_DATA4	I1/O/T	U2RXD	I1	EMAC_CLK_OUT	O	2'd2	oe=0, ie=0	oe=0, ie=1
26	VDD_SDIO			VDD_SDIO supply out/in																				
27			GPIO17	VDD_SDIO						GPIO17	I/O/T			GPIO17	I/O/T	HS1_DATA5	I1/O/T	U2TXD	O	EMAC_CLK_OUT_180	O	2'd2	oe=0, ie=0	oe=0, ie=1
28			SD_DATA_2	VDD_SDIO						SD_DATA2	I1/O/T	SPIHD	I/O/T	GPIO9	I/O/T	HS1_DATA2	I1/O/T	U1RXD	I1			2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
29			SD_DATA_3	VDD_SDIO						SD_DATA3	I0/O/T	SPIWP	I/O/T	GPIO10	I/O/T	HS1_DATA3	I1/O/T	U1TXD	O			2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
30			SD_CMD	VDD_SDIO						SD_CMD	I1/O/T	SPICS0	I/O/T	GPIO11	I/O/T	HS1_CMD	I1/O/T	U1RTS	O			2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
31			SD_CLK	VDD_SDIO						SD_CLK	I0	SPICLK	I/O/T	GPIO6	I/O/T	HS1_CLK	O	U1CTS	I1			2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
32			SD_DATA_0	VDD_SDIO						SD_DATA0	I1/O/T	SPIQ	I/O/T	GPIO7	I/O/T	HS1_DATA0	I1/O/T	U2RTS	O			2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
33			SD_DATA_1	VDD_SDIO						SD_DATA1	I1/O/T	SPID	I/O/T	GPIO8	I/O/T	HS1_DATA1	I1/O/T	U2CTS	I1			2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
34			GPIO5	VDD3P3_CPU						GPIO5	I/O/T	VSPICS0	I/O/T	GPIO5	I/O/T	HS1_DATA6	I1/O/T			EMAC_RX_CLK	I	2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
35			GPIO18	VDD3P3_CPU						GPIO18	I/O/T	VSPICLK	I/O/T	GPIO18	I/O/T	HS1_DATA7	I1/O/T					2'd2	oe=0, ie=0	oe=0, ie=1
36			GPIO23	VDD3P3_CPU						GPIO23	I/O/T	VSPID	I/O/T	GPIO23	I/O/T	HS1_STROBE	I0					2'd2	oe=0, ie=0	oe=0, ie=1
37	VDD3P3_CPU			VDD3P3_CPU supply in																				
38			GPIO19	VDD3P3_CPU						GPIO19	I/O/T	VSPIQ	I/O/T	GPIO19	I/O/T	U0CTS	I1			EMAC_TXD0	O	2'd2	oe=0, ie=0	oe=0, ie=1
39			GPIO22	VDD3P3_CPU						GPIO22	I/O/T	VSPIWP	I/O/T	GPIO22	I/O/T	U0RTS	O			EMAC_TXD1	O	2'd2	oe=0, ie=0	oe=0, ie=1
40			U0RXD	VDD3P3_CPU						U0RXD	I1	CLK_OUT2	O	GPIO3	I/O/T							2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
41			U0TXD	VDD3P3_CPU						U0TXD	O	CLK_OUT3	O	GPIO1	I/O/T					EMAC_RXD2	I	2'd2	oe=0, ie=1, wpu	oe=0, ie=1, wpu
42			GPIO21	VDD3P3_CPU						GPIO21	I/O/T	VSPIHD	I/O/T	GPIO21	I/O/T					EMAC_TX_EN	O	2'd2	oe=0, ie=0	oe=0, ie=1
43	VDDA			VDDA supply in																				
44		XTAL_N		VDDA																				
45		XTAL_P		VDDA																				
46	VDDA			VDDA supply in																				
47		CAP2		VDDA																				
48		CAP1		VDDA																				
Total Number	8	14	26																					


WROOM32
PINOUT
Power
GND
Serial Pin
Analog Pin
Control
Physical Pin
Port Pin
Touch Pin
DAC Pin
PWM Pin
ANT1
PROGRAM Port
GND
3.3V
ChipPU
9
ADCPA RTCIO0 ADC1_0 SensVP GPIO36 5
ADCPA RTCIO3 ADC1_3 SensVN GPIO39 8
RTCIO4 ADC1_6 VDET1 GPIO34 10
RTCIO5 ADC1_7 VDET2 GPIO35 11
XTAL32 Touch9 RTCIO9 ADC1_4 GPIO32 12
XTAL32 Touch8 RTCIO8 ADC1_5 GPIO33 13
DAC_1 RTCIO6 ADC2_8 EMACRXD0 GPIO25 14
DAC_2 RTCIO7 ADC2_9 EMACRXD1 GPIO26 15
Touch7 RTCIO17 ADC2_7 EMACRXDV GPIO27 16
HS2CLK SDCLK HSPICLK MTMS Touch6 RTCIO16 ADC2_6 EMACTXD2 GPIO14 17
HS2DATA2 SDDATA2 HSPIQ MTDI Touch5 RTCIO15 ADC2_5 EMACTXD3 GPIO12 18
GND
HS2DATA3 SDDATA3 HSPIID MTCK Touch4 RTCIO14 ADC2_4 EMACRXER GPIO13 20
SDDATA2 HS1DATA2 U1RXD SPIHD GPIO9 28
SDDATA3 HS1DATA3 U1TXD SPIWP GPIO10 29
SDCMD HS1CMD U1RTS SPICS0 GPIO11 30
GND
36 GPIO23 VSPIID HS1STROBE
39 GPIO22 EMACTXD1 U0RTS VSPIWP
41 GPIO1 EMACRXD2 U0TXD CLKOUT3
40 GPIO3 U0RXD CLKOUT2
42 GPIO21 EMACTXEN VSPIHD
NC
38 GPIO19 EMACTXD0 U0CTS VSPIQ
35 GPIO18 VSPICLK HS1DATA7
34 GPIO5 EMACRXCLK VSPICS0 HS1DATA6
27 GPIO17 EMACCOL0 U2TXD HS1DATA5
25 GPIO16 EMACCLKOUT3 U2RXD HS1DATA4
24 GPIO4 EMACTXER ADC2_0 RTCIO10 Touch0 HSPIHD SDDATA1 HS2DATA1
23 GPIO0 EMACTXCLK ADC2_1 RTCIO11 Touch1 CLKOUT1
22 GPIO2 HSPIWP ADC2_2 RTCIO12 Touch2
21 GPIO15 EMACRXD3 ADC2_3 RTCIO13 Touch3 MTDO HSPICS0 SDCMD HS2CMD
33 GPIO8 SPID U2CTS HS1DATA1 SDDATA1
32 GPIO7 SPIQ U2RTS HS1DATA0 SDDATA0
31 GPIO6 SPICLK U1CTS HS1CLK SDCLK
C7
C6
C12
C11
C18
R7
C1
C2
C16
C15
U1
U2
U3
FM25Q32
1608ALH


Chip-enable signal,Active High. EN pin15
ADC PA RTC_GPIO0 ADC1_CH0 SENSOR_VP GPIO36 pin14
ADC PA RTC_GPIO3 ADC1_CH3 SENSOR_VN GPIO39 pin13
RTC_GPIO4 ADC1_CH6 VDET1 GPIO34 pin12
RTC_GPIO5 ADC1_CH7 VDET2 GPIO35 pin11
XTAL_32kHz Touch9 RTC_GPIO9 ADC1_CH4 GPIO32 pin10
XTAL_32kHz Touch8 RTC_GPIO8 ADC1_CH5 GPIO33 pin9
DAC_1 RTC_GPIO6 ADC2_CH8 EMAC_RXD0 GPIO25 pin8
DAC_2 RTC_GPIO7 ADC2_CH9 EMAC_RXD1 GPIO26 pin7
Touch7 RTC_GPIO17 ADC2_CH7 EMAC_RX_DV GPIO27 pin6
HS2_CLK SD_CLK HSPI_CLK MTMS Touch6 RTC_GPIO16 ADC2_CH6 EMAC_TXD2 GPIO14 pin5
HS2_DATA2 SD_DATA2 HSPI_MISO MTDI Touch5 RTC_GPIO15 ADC2_CH5 EMAC_TXD3 GPIO12 pin4
HS2_DATA3 SD_DATA3 HSPI_MOSI MTCK Touch4 RTC_GPIO14 ADC2_CH4 EMAC_RX_ER GPIO13 pin3
GND pin2
VIN pin1
pin15 GPIO23 SPI_MOSI HS1_STROBE
pin14 GPIO22 EMAC_TXD1 U0RTS I2C_SCL
pin13 GPIO1 EMAC_RXD2 U0TXD CLK OUT3
pin12 GPIO3 U0RXD CLK OUT2
pin11 GPIO21 EMAC_TX_EN I2C_SDA
pin10 GPIO19 EMAC_TXD0 U0CTS SPI_MISO
pin9 GPIO18 SPI_CLK HS1_DATA7
pin8 GPIO5 EMAC_RX_CLK SPI_CS0 HS1_DATA6
pin7 GPIO17 EMAC_CLKOUT180 U2_TXD HS1_DATA5
pin6 GPIO16 EMAC_CLKOUT U2_RXD HS1_DATA4
pin5 GPIO4 EMAC_TX_ER ADC2_CH0 RTCIO10 Touch0 HSPIHD SD_DATA1 HS2_DATA1
pin4 GPIO2 ADC2_CH2 RTCIO12 Touch2 HSPIWP
pin3 GPIO15 EMAC_RXD3 ADC2_CH3 RTCIO13 Touch3 MTDO HSPI_CS0 SD_CMD HS2_CMD
pin2 GND
pin1 VDD_3V3
POWER
GND
Serial Pin
Analog Pin
Control
Physical Pin
Port Pin
Touch Pin
DAC Pin
ESP-WROOM-32
211-161007
FCC ID:2AC7Z-ESPWROOM32
EN
BOOT