Universidad Zaragoza
1542

Trabajo Fin de Grado

Red de Sensores y Actuadores IoT para Edificios Inteligentes

Sensor and Actuator Network for Smart Buildings

Autor

Paris Bielsa Godina

Directores

Roberto Casas Nebra
Álvaro Marco Marco

Titulación del autor

Grado en Ingeniería Electrónica y Automática

Escuela de Ingeniería y Arquitectura

2023

Resumen

Este trabajo propone el desarrollo de una red de sensores y actuadores IoT inalámbricos distribuidos en distintos lugares de un edificio capaces de recolectar variables ambientales o activar señales eléctricas o visuales según la necesidad del usuario final. Esto permitirá manejar numerosos dispositivos de un edificio de forma automatizada y remota para así poder observar variables ambientales del edificio y reducir el coste energético.

Para realizar este sistema se desea crear una red capaz de comunicar información de nodos situados por numerosos lugares del edificio. Adicionalmente se desea también poder observar dichas variables a través de una interfaz web de tal forma que el usuario pueda visualizarlas o modificarlas a su gusto. Finalmente se busca mantener cierta independencia entre el protocolo de comunicación de los nodos de la red y el protocolo de comunicación del servidor central, para así poder reducir las vulnerabilidades del propio sistema y evitar ataques externos.

A la hora de diseñar este sistema ha sido necesario estudiar las topologías y tecnologías existentes para la comunicación entre nodos de la red. Tras estudiar todas ellas se ha decidido utilizar el protocolo Zigbee que, a diferencia de otros protocolos como BLE, Thread, WiFi o Z-Wave, es un protocolo abierto y de bajo coste energético capaz de calcular dinámicamente la ruta ideal para transmitir la información.

Toda la información de los sensores y actuadores será transmitida hasta un servidor central que se encargará de almacenar dicha información, transmitirla al usuario final a través de software de visualización de datos y permitir su modificación a través de llamadas HTTP. Adicionalmente dicho servidor se encargará de mantener la independencia entre ambos protocolos para así minimizar las vulnerabilidades y proteger a los nodos de posibles ciberataques.

Respecto a este servidor nos hemos decantado por utilizar InfluxDB y Grafana para el almacenamiento y visualización de datos. InfluxDB es un motor de bases de datos especializado para datos temporales lo cual es óptimo para guardar datos enviados periódicamente por los sensores. Grafana es una interfaz web diseñada para visualizar detalladamente la información almacenada en bases de datos. Para transmitir órdenes a los actuadores ejecutamos rutinas de Node-Red mediante botones visibles en Grafana.

Adicionalmente hemos diseñado un gateway capaz de traducir la información recibida desde la red Zigbee al protocolo MQTT del servidor central, y viceversa utilizando un dongle Zigbee-USB y el programa Zigbee2MQTT.

Por último, los prototipos utilizan un microcontrolador de bajo consumo fabricado por Silicon Labs que es capaz de emitir y recibir datos a través de Zigbee. Dicho microcontrolador denominado EFR32MG24 tiene un coste de alrededor de los $10 y tiene un uso energético de tan solo 5mA a la hora de recibir información y de 20mA a la hora de transmitir información a 10dBm.

Índice

1 Introducción 6
1.1 Objetivos 7
1.2 Planificación 7
2 Análisis del estado del arte 9
2.1 Topologías IoT 9
2.2 Protocolos de tecnologías IoT 11
3 Selección de la tecnología 15
3.1 Selección de protocolo 15
3.2 Selección de microcontrolador 16
4 Diseño del hardware 18
4.1 Tipos de Sensores y Actuadores 18
4.2 Diseño de la arquitectura de los nodos 19
4.3 Elección de componentes 20
4.4 Diseño del esquemático 24
4.5 Diseño de la PCB 27
5 Diseño del firmware 30
5.1 Estructura Zigbee 30
5.2 Configuración Zigbee de los nodos 31
5.3 Desarrollo del firmware 33
6 Diseño del software 37
6.1 Gateway Zigbee 37
6.2 Servicio de control 38
6.3 Almacenamiento de datos 39
6.4 Interfaz con el usuario 40
6.5 Hardware utilizado 41
7 Validación de la red 42
7.1 Evaluación de la red 42
7.2 Complicaciones durante el desarrollo 45
7.3 Otros procesos de automatización 46
8 Conclusiones y futuro del proyecto 48
9 Bibliografía 49
ANEXO I: Código fuente proyecto TFG_Relay 51
ANEXO II: Código fuente proyecto TFG_Leaf 60
ANEXO III: Archivos de configuración Zigbee2MQTT 66

Fig 1: Número de dispositivos IOT [1] 6
Fig 2: Topología de Estrella 9
Fig 3: Topología de Árbol 10
Fig 4: Topología de Malla 10
Fig 5: Protocolos IoT 11
Fig 6: Tabla Comparando Protocolos IOT 14
Fig 7: Microcontrolador WBZ451PE 16
Fig 8: Módulo ESP32-C6-WROOM-1 17
Fig 9: Diagrama de Bloques Repetidor Zigbee 19
Fig 10: Diagrama de Bloques Dispositivo Final Zigbee 20
Fig 11: Pines del MGM240PA22VNA3 20
Fig 12: Lista de Materiales Repetidor Zigbee 23
Fig 13: Lista de Materiales Dispositivo Final Zigbee 23
Fig 14: Esquemático del circuito 24
Fig 15: Esquemático del Microcontrolador 24
Fig 16: Esquemático de la Alimentación por Batería 25
Fig 17: Esquemático de la Alimentación Alterna 26
Fig 18: Esquemático de las Entradas y Salidas 26
Fig 19: Esquemático de la Alimentación 3V3 y Filtrado 27
Fig 20: Vista superior de la PCB 27
Fig 21: Vista inferior de la PCB 28
Fig 22: Tabla de Componentes por Configuración 29
Fig 23: Configuración de la Estructura Zigbee 31
Fig 24: Pantalla de configuración de Zigbee 33
Fig 25: Pantalla de configuración de los pines 33
Fig 26: Diagrama de flujo dispositivo repetidor 35
Fig 27: Diagrama de flujo dispositivo final 36
Fig 28: Diagrama de la Arquitectura 37
Fig 29: Flujos Node-RED 38
Fig 30: Flujos de Automatización Node-RED 39
Fig 31: Diagrama de la Base de Datos 39
Fig 32: Pantalla de visualización de datos 40
Fig 33: Imagen Raspberry Pi Zero 2 W 41
Fig 34: Fotografía prototipo dispositivo final 42
Fig 35: Fotografía prototipo repetidor 42
Fig 36: Diagramas Iluminación y Ventilador Automáticos 43
Fig 37: Gráfica de la comparación de los sensores de temperatura 44
Fig 38: Diagramas termo automatizado y nevera inteligente 46
Fig 39: Diagrama sistema de climatización automatizado 47

1 Introducción

En la actualidad existen millones de dispositivos IoT capaces de comunicar datos entre el mundo digital y el mundo físico. Estos dispositivos pueden ser vistos en todo tipo de ámbitos, desde grandes industrias o ciudades hasta pequeñas oficinas u hogares.

Estos dispositivos IoT son simples dispositivos que se encargan de traducir información que recibe del mundo físico y enviarla al mundo digital y viceversa. Gracias a ellos podemos medir numerosos datos del mundo físico como la temperatura, humedad o la interacción con un pulsador o fin de carrera e interactuar con el mundo físico ya sea encendiendo un motor, conectando un relé o encendiendo un diodo.

El punto fuerte de estos dispositivos proviene de la unión de numerosos dispositivos, ya que nos permiten crear automatismos diseñados para la automatización de numerosas tareas de un edificio.

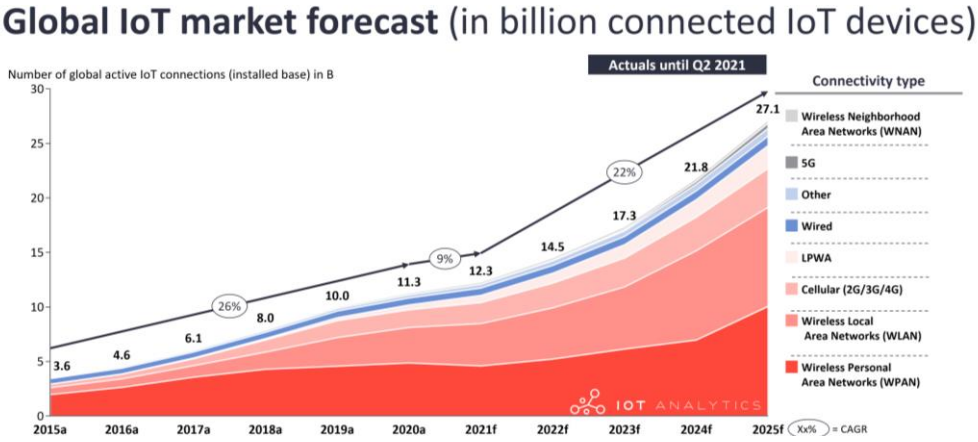

Fig 1: Número de dispositivos IOT [1]

Por ello este TFG plantea el diseño de una red de sensores y actuadores diseñados para edificios inteligentes que busca medir señales ambientales e interactuar con el mundo físico, minimizando el consumo energético de los nodos. Este sistema nos permitirá visualizar las señales e interactuar con el entorno desde una interfaz web y nos permitirá crear numerosos automatismos sin necesidad de conectarse a una interfaz web externa a nuestro edificio inteligente.

1.1 Objetivos

El objetivo principal de este TFG es de crear una red de dispositivos que nos permita crear un edificio inteligente con el que podamos visualizar datos ambientales del interior y del exterior del edificio y podamos encender o apagar dispositivos de forma automática según el estado de estas variables atmosféricas.

Para esto, se plantean los siguientes objetivos secundarios:

- Analizar las posibles tecnologías existentes
- Seleccionar las tecnologías óptimas para diseñar la red
- Diseñar varios dispositivos que permitan medir datos ambientales e interactuar con el entorno.
- Realizar la programación de los dispositivos para enviar estos datos a la red y recibir instrucciones de ella.
- Conectar un equipo central que nos permita interactuar con la red desde software.
- Implementar una solución de software que nos permita visualizar los datos recibidos por la red, crear automatismos y enviar instrucciones desde una interfaz gráfica

Como objetivo adicional, se plantea la implementación de varios casos de uso orientados a aumentar la eficiencia energética de un edificio inteligente y promover el uso de energías renovables, que permitirán validar el funcionamiento de la solución planteada y la consecución del objetivo principal del proyecto.

1.2 Planificación

Para llevar a cabo los objetivos planteados se establecen las siguientes etapas en el desarrollo del proyecto:

En la fase de **análisis del estado del arte** se realizará un análisis de las tecnologías existentes en la actualidad y que son relevantes en el proyecto: tipos de topología de comunicación, protocolos existentes para dispositivos IoT

En la fase de **selección de la tecnología** se analizarán las opciones disponibles para diseñar la red IoT. Se seleccionará la topología y el protocolo óptimo para realizar esta red, al igual que el microcontrolador que se utilizará en este trabajo.

En la fase de **diseño de hardware** realizaremos la selección de cada uno de los sensores y actuadores a utilizar, presentaremos las configuraciones de los prototipos y diseñaremos los circuitos impresos para nuestros prototipos.

En la fase de **diseño de firmware** expondremos el código que utilizan los microcontroladores para comunicarse con los otros nodos de la red y explicaremos algunas de las peculiaridades de cada tipo de sensor o actuador.

En la fase de **diseño de software** analizaremos el software que podemos usar para gestionar todas las tareas realizadas por el servidor central.

En la fase de **implementación de la red** presentaremos los usos que se han desarrollado para nuestros prototipos y expondremos posibles casos de usos más avanzados que se pueden desarrollar al conectar nuestros dispositivos a diversos electrodomésticos. Todos estos casos de uso se centrarán en la mejora del nivel de vida del usuario y el ahorro energético del edificio inteligente.

2 Análisis del estado del arte

En este apartado se van a analizar las distintas topologías, protocolos y tecnologías existentes que permitan la comunicación inalámbrica entre numerosos nodos IoT.

2.1 Topologías IoT

A la hora de mandar mensajes en una red IoT es importante conocer el trayecto que deberá recorrer la información para llegar desde el nodo inicial hasta el servidor central. La estructura que forman las conexiones entre los distintos nodos se denomina topología y de ella dependen tanto la velocidad como la distancia desde la que se puede enviar información.

Existen tres principales topologías que son viables para conectar dispositivos inalámbricos: topología de estrella, topología de árbol o topología de malla [2][3].

La **topología de Estrella** se compone de un nodo central (o coordinador) que se comunica directamente con cada uno de los nodos satélites. Los nodos satélites se comunican únicamente con el coordinador por lo que, si deseamos transmitir información entre dos nodos, dicha información deberá atravesar el coordinador.

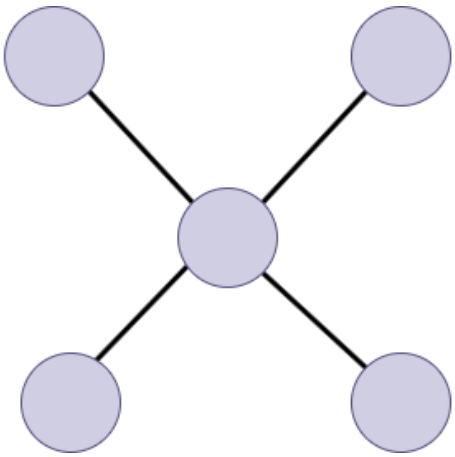

Fig 2: Topología de Estrella

La **topología de Árbol** se compone de un nodo principal (o coordinador) que contiene varios sub-nodos que a su vez actúan como coordinador de otros sub-nodos hasta conseguir cubrir la red deseada. Esta topología nos permite aumentar el área de la red ya que permite añadir nodos adicionales que aumenten el rango de la red. Sin embargo, todo nodo depende de sus nodos

superiores y quedará desconectado de la red si alguno de los nodos superiores sufre una baja.

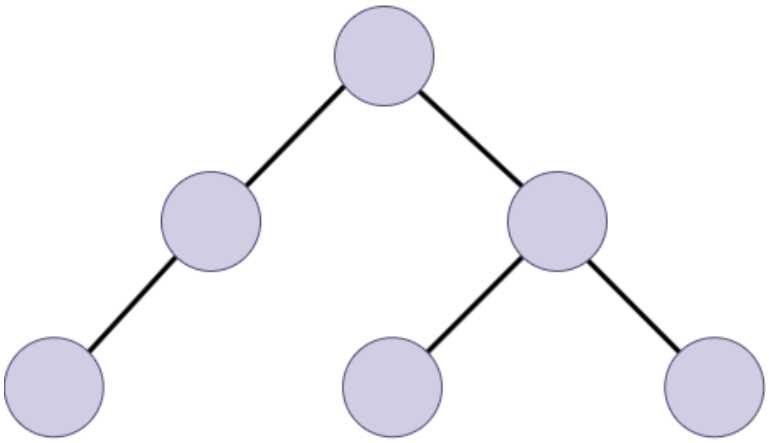

Fig 3: Topología de Árbol

La **topología de malla** consiste en conectar todos los nodos próximos entre sí de tal forma que se cree una red de nodos intercomunicados con varias rutas redundantes. Esta topología nos permite crear rutas dinámicas dependiendo del estado de la red y adicionalmente nos permite modificar las rutas en el caso en el que un nodo pierda se desconecte de la red. Sin embargo, esta topología supone una mayor carga en los nodos debido a su sistema dinámico de ruteo.

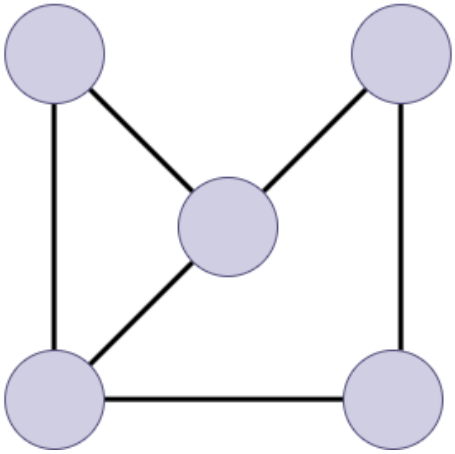

Fig 4: Topología de Malla

2.2 Protocolos de tecnologías IoT

Para transmitir la información existen numerosos protocolos capaces de gestionar la transmisión de datos y asegurar su correcta recepción. Entre los principales se encuentran: WiFi, Zigbee, LoRa, Z-Wave, Bluetooth (BLE) y Thread [4][5].

Fig 5: Protocolos IoT

El **protocolo WiFi** es por excelencia la tecnología de comunicación predominante en nuestro día a día, permitiendo que los dispositivos habilitados con WiFi puedan conectarse a Internet a través de un punto de acceso.

WiFi tiene habitualmente una topología de estrella, y utilizando repetidores puede adoptar una topología de árbol. Sin embargo existen dispositivos capaces de crear una red WiFi mallada, aunque su precio es superior al precio de los routers WiFi habituales. Sus principales ventajas son su fiabilidad y su disponibilidad en el mercado. Nos permite enviar una gran cantidad de información directamente hasta servidores a través de Internet. Sin embargo, debido al funcionamiento de su protocolo consume una gran cantidad de energía para establecer la conexión con el servidor y transmitir datos.

Bluetooth es otra de las tecnologías de comunicación más utilizada actualmente. Permite la transmisión de datos y se encuentra incorporado en la mayoría de los dispositivos móviles.

Bluetooth dispone también del protocolo **Bluetooth Low Energy (BLE)** presente en todos los dispositivos compatibles con Bluetooth 4.0 que tiene un

rango mucho menor que Bluetooth normal, pero tiene un consumo considerablemente inferior.

BLE puede funcionar tanto con las topologías de estrella, árbol y malla. Sin embargo, la topología de malla utiliza el mecanismo de ruteo de "malla de inundación (*Flooding Mesh*)" en la que los mensajes son enviados a todos y cada uno de los nodos en lugar de calcular la ruta ideal.

BLE tiene un coste y un consumo similar al de Zigbee, sin embargo, el sistema de ruteo que utiliza nos limita considerablemente la cantidad de nodos que puede haber en la red.

Zigbee es un protocolo de comunicación creado para funcionar principalmente en topología de malla, aunque también funciona en topologías de estrella y árbol. Tiene diseñado un sistema avanzado de ruteo dinámico con el que permite determinar la ruta más corta para transmitir información.

Zigbee cuenta con un rango prácticamente ilimitado ya que se pueden añadir nodos repetidores para aumentar el área de la red, cuenta también con un modo de bajo consumo para los nodos externos lo cual lo hace óptimo para redes de dispositivos IoT de bajo consumo. Sin embargo, Zigbee dispone de una velocidad de transferencia baja de alrededor de los 250Kbps.

Z-Wave es un protocolo similar a Zigbee especializado en las redes malladas de dispositivos IoT de bajo consumo. Z-Wave es un protocolo propietario desarrollado por Zensis que utiliza una frecuencia de comunicación de alrededor de unos 908.42Mhz. Al ser un protocolo propietario, los microcontroladores con Z-Wave deben cumplir una serie de requisitos estrictos para facilitar la intercomunicación entre dispositivos con Z-Wave.

Las principales ventajas de Z-Wave son su mayor rango y su independencia de las frecuencias de 2.5GHz comunes. Aunque sin embargo sus principales desventajas son su coste elevado debido a ser un protocolo propietario y una velocidad de transmisión mucho menor.

Thread es la tecnología más reciente creada para la transmisión de datos en sistemas IoT. Thread está diseñado para crear una red de topología de malla, aunque también nos permite utilizar otras topologías como la de estrella o árbol.

Thread forma parte del ecosistema de Matter y puede disponer de numerosos routers en la misma malla que reenvíen la información entre WiFi y Thread. Esto hace que Thread sea un protocolo ligeramente más complejo que otras alternativas.

Adicionalmente, al ser un protocolo tan novedoso, todavía no existen muchos microcontroladores en el mercado capaces de comunicarse utilizando Thread.

LoRa (Long Range) es una tecnología de radiofrecuencias propietaria que funciona a una frecuencia de unos 870MHz, que es una frecuencia de libre uso para los consumidores.

LoRaWAN es el protocolo que funciona con una topología de estrella y utiliza dispositivos LoRa para transmitir información para largas distancias de hasta más de 10 kilómetros con una velocidad de transmisión muy baja de alrededor de unos 10kbit/s.

LoRaWAN es ideal para la transmisión de datos IOT cuando los nodos se encuentran a distancias de entre 50 metros hasta varios kilómetros de distancia, como por ejemplo para la transmisión de medidas en ciudades inteligentes o la transmisión de datos en un campus universitario [6].

Tras estudiar los distintos protocolos se observa que cada protocolo ha sido creado para satisfacer distintas necesidades por lo que es importante conocer los requisitos de un proyecto para seleccionar el protocolo óptimo.

A continuación presentaremos una tabla en la que se indicarán los principales datos de cada protocolo.

Entre los datos presentados tenemos: el rango de la red representa la distancia máxima nominal a la que se pueden comunicar dos nodos (en m o km), la cantidad de nodos representa el número de nodos máximo nominal que permite una red, la velocidad de transmisión representa la velocidad media a la que se envían datos entre nodos (en Mbit/s o kbit/s), el coste económico

representa el precio aproximado al que se venden módulos compatibles con cada tecnología (en dólares), el coste energético representa la corriente nominal consumida por cada módulo, finalmente los dispositivos compatibles representa la cantidad de dispositivos vendidos anualmente que utilizan dichos protocolos (en millones, año 2019).

Tecnología	WiFi	BLE	Zigbee	Z-Wave	Thread	LoRa
Rango de la red	100m	30m	100m	100m	100m	10km
Cantidad de Nodos	--	32767	65536	232	300	1000
Velocidad de Transmisión	54Mb/s	1Mb/s	250kb/s	100kb/s	250kb/s	50kb/s
Coste económico	<10$	<5$	<15$	<10$	<15$	<15$
Coste energético	Alto	Bajo	Bajo	Bajo	Bajo	Bajo
Dispositivos Compatibles	3200 M	3500 M	420 M	120 M	420 M	45 M

Fig 6: Tabla Comparando Protocolos IOT

3 Selección de la tecnología

En este apartado seleccionaremos la tecnología principal para el funcionamiento de los numerosos nodos de la red. Para ello seleccionaremos el protocolo a utilizar junto con su topología y a continuación elegiremos el microcontrolador que más se adecua para el desarrollo de este trabajo.

3.1 Selección de protocolo

Para diseñar la red tenemos una lista de requisitos los cuales compararemos con los datos presentados en el apartado anterior para obtener el protocolo óptimo para la realización de este trabajo.

- La red debe ser capaz de recibir información de una gran cantidad de nodos, por lo que buscamos utilizar un protocolo que nos permita emparejar numerosos nodos. Para el desarrollo de una red para un edificio inteligente se estima la necesidad de tener más de 500 nodos conectados, por lo que Z-Wave y Thread no son protocolos viables.
- La red debe ser capaz de comunicarse con los nodos situados por varios puntos de un edificio inteligente, por lo que debe de tener un rango grande o ser capaz de crear una red mallada para aumentar el rango dinámicamente. La tecnología WiFi necesitará numerosos repetidores en cada planta del edificio inteligente para crear una red que cubra todo el edificio. BLE al tener un rango tan bajo necesitará que los nodos se encuentren muy próximos y por lo tanto aumentar la cantidad de nodos.
- Adicionalmente también buscamos un protocolo estándar y con gran cantidad de interoperabilidad para poder añadir sensores diseñados por terceros a nuestra red. En este aspecto destacan Zigbee, Z-Wave y Thread ya que son protocolos diseñados específicamente para la creación de redes IoT en la que se puedan usar dispositivos de distintos fabricantes.
- Finalmente buscamos un protocolo que ya esté establecido en el mercado con una gran cantidad de sensores y actuadores existentes. En este aspecto destacan principalmente BLE y WiFi ya que son los protocolos predominantes en el mercado. Sin embargo en el sector IoT existen también una gran cantidad de dispositivos que utilizan Zigbee, volviendo a ésta una tecnología capaz de competir con WiFi y BLE.

3.2 Selección de microcontrolador

Actualmente en el mercado existen numerosos microcontroladores capaces de recibir y transmitir información por Zigbee. Por ello hemos de seleccionar un microcontrolador que nos permita una gran cantidad de customización al menor coste tanto energético como económico posible.

Existen numerosos fabricantes que fabrican microcontroladores capaces de utilizar el protocolo Zigbee. De entre todos ellos los más destacables son los siguientes.

Microchip Technology cuenta con el SoC PIC32CX-BZ2 (y su módulo WBZ451PE) el cual es capaz de utilizar los protocolos de BLE 5.2 y Zigbee 3.0.

El módulo tiene un coste de alrededor de los 8 euros por unidad y un consumo energético de hasta 30mA transmitiendo datos, 20mA recibiendo datos y 0.12uA en modo XDS (Extreme Deep Sleep) [7].

Fig 7: Microcontrolador WBZ451PE

Sin embargo, a fecha del desarrollo de este TFG la disponibilidad del módulo WBZ451PE es prácticamente nula y es necesario comprarlos en grupos de 100. Es por ello que hemos decidido no utilizar este microcontrolador y decantarnos mejor por otros microcontroladores con mayor disponibilidad.

Silicon Labs es uno de los creadores de la tecnología Z-Wave y es uno de los principales fabricantes de microcontroladores Zigbee por lo que son lideres en el mercado en dispositivos de bajo consumo.

Disponen de los microcontroladores EFR32MG24 Series 2 disponibles tanto en SoC como en modulo y son capaces de utilizar los protocolos Thread, Zigbee y BLE [8].

Estos módulos tienen un coste de alrededor de unos 14 euros y un consumo energético de 5mA transmitiendo datos, 6mA recibiendo datos y 1.3uA en modo Deep Sleep.

Espressif Systems ha diseñado numerosos microcontroladores capaces de transmitir datos utilizando radio frecuencias. A día de la realización del análisis del estado del arte Espressif ha sacado el microcontrolador ESP32-C6 capaz de transmitir datos utilizando tanto WiFi, BLE, Thread y Zigbee.

El módulo ESP32-C6-WROOM-1 tiene un coste de unos 3 euros por unidad y tiene un consumo energético de 120mA transmitiendo datos, 73mA recibiendo datos y 7uA en modo Deep Sleep [9][10].

Fig 8: Módulo ESP32-C6-WROOM-1

Adicionalmente Espressif cuenta con documentación detallada sobre la funcionalidad de las distintas funciones de Zigbee y cuenta adicionalmente con numerosos ejemplos de creación de redes Zigbee usando sus microcontroladores [11].

Entre los microcontroladores estudiados se observa que los microcontroladores Espressif y de Silicon Labs buscan satisfacer mercados distintos. Los microcontroladores de Espressif nos ofrecen una solución compatible con el IDE de Arduino, es decir una solución que sea fácil de poner en marcha pero que no está preparada para casos de usos complejos, diseñado principalmente para el mercado “maker”. Sin embargo, los microcontroladores de SiLabs nos ofrecen una solución para el mercado profesional ya que nos permite desarrollar sistemas mucho más complejos optimizando también el consumo y el rendimiento de éstos. Por desgracia los microcontroladores de SiLabs cuentan con una curva de aprendizaje mucho más dura que los microcontroladores de Espressif debido a la enorme cantidad de opciones y configuraciones con las que cuenta.

Por estos motivos, nos hemos decantado por utilizar el microcontrolador de Silicon Labs EFR32MG24 Series 2 que, si bien requiere un esfuerzo de desarrollo inicial mayor, nos permitirá asegurar el funcionamiento deseado del programa.

4 Diseño del hardware

Como primer prototipo para este proyecto buscamos diseñar un circuito impreso multipropósito con la capacidad de soldar distintas configuraciones de componentes dependiendo del uso deseado. Una vez diseñado este prototipo, de cara a futuros proyectos, se puede utilizar como base para diseños mas específicos.

Para diseñar el hardware debemos primero seleccionar qué tipos de sensores y actuadores colocaremos en nuestro prototipo, ya que estos tendrán incidencia sobre otros componentes de la placa, como por ejemplo el tipo de alimentación o la necesidad de incorporar circuitería adicional. Seleccionando qué variables buscamos medir y cuáles son las formas con las que se interactuará con el entorno y con el usuario.

Diseñaremos la arquitectura que seguirá el prototipo, presentando dos configuraciones de componentes de forma que nuestro dispositivo actúe como repetidor de la red Zigbee o como dispositivo final de la red.

Una vez diseñada la arquitectura hemos de seleccionar los componentes a colocar según la configuración. Con esto presentaremos el Bill Of Materials e indicaremos el coste aproximado de cada una de las configuraciones.

A continuación, deberemos diseñar un esquemático en el que se puedan observar los componentes anteriormente mencionados y su localización en el circuito junto a los componentes pasivos que necesite.

Finalmente debemos diseñar la PCB física la cual debe sostener y permitir el cableado de todos los componentes de la forma más apropiada para minimizar el ruido y evitar los fallos del sistema.

4.1 Tipos de Sensores y Actuadores

A la hora de seleccionar los actuadores, buscamos encender o apagar dispositivos, ya sean electrodomésticos, bombillas o contactores. Así mismo, buscamos tener indicadores luminosos que nos permitan informar al usuario de un evento.

En el caso de los indicadores luminosos podemos utilizar diodos LED conectados directamente a un pin del microcontrolador para poder encenderlo, apagarlo, o regular su luminosidad.

Para poder conectar o desconectar una carga colocaremos un relé conectado al microcontrolador que esté diseñado para soportar la potencia necesaria para alimentar la carga.

Respecto a los sensores, los datos más oportunos a medir son la temperatura y humedad de una habitación, y la presencia de un ser humano enfrente de nuestro sensor. Para ello hemos de utilizar un sensor de temperatura y humedad que nos permite medir ambos valores con un mismo sensor, y un sensor PIR que utiliza ondas electromagnéticas infrarrojas para detectar la presencia y movimiento de un ser vivo.

4.2 Diseño de la arquitectura de los nodos

Una vez tenemos seleccionados los sensores y actuadores que colocar en nuestros prototipos. Debemos especificar las dos principales configuraciones que utilizaremos para nuestros prototipos.

La primera configuración funcionará como repetidor de la red Zigbee y se alimentará desde la red alterna. Tendrá sensores tanto de temperatura, humedad y presencia, e incorporará como actuadores un Indicador LED y un relé.

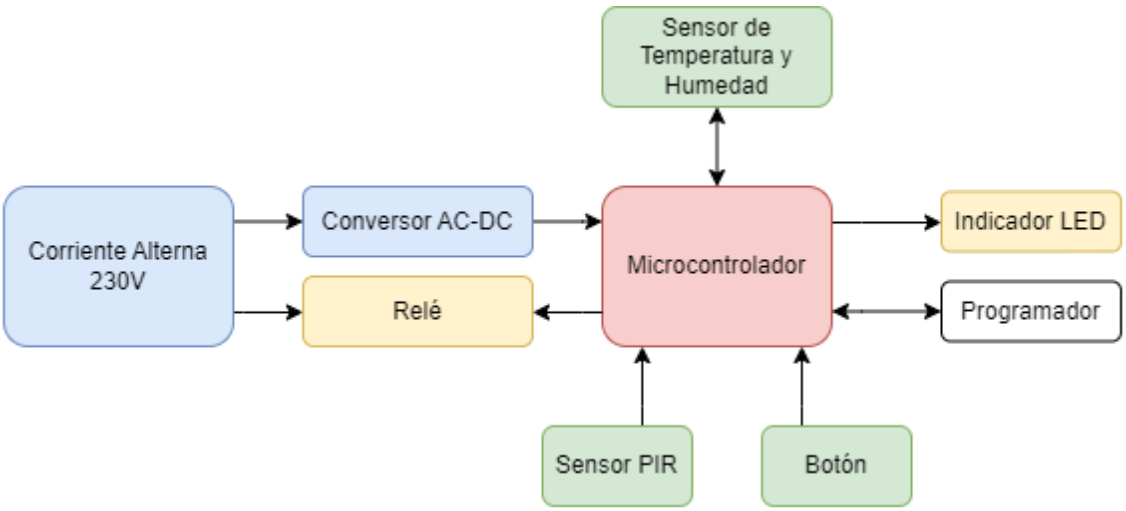

Fig 9: Diagrama de Bloques Repetidor Zigbee

La segunda configuración funcionará como dispositivo final de la red Zigbee, se encontrará normalmente en modo de bajo consumo y se alimentará desde una batería de litio que recargaremos desde un USB. Como sensores tendrá únicamente sensores de temperatura y humedad y como actuador un indicador LED.

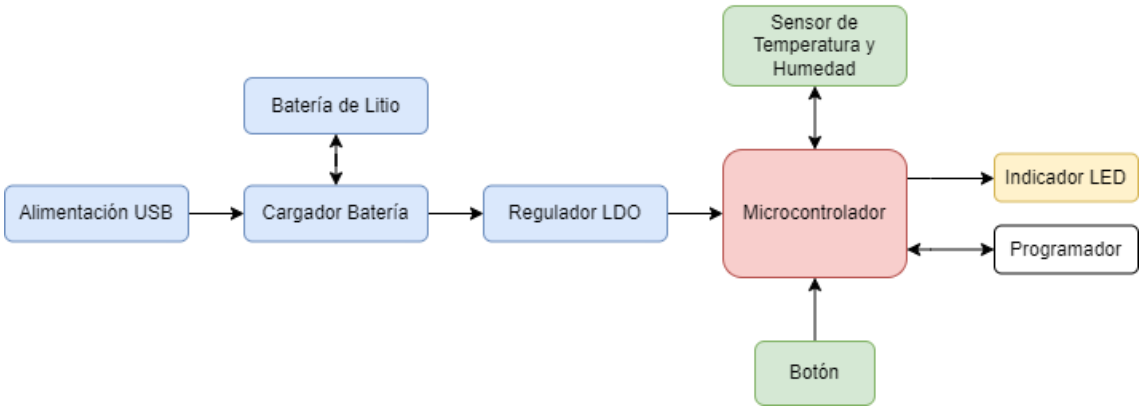

Fig 10: Diagrama de Bloques Dispositivo Final Zigbee

4.3 Elección de componentes

Como ya se ha mencionado anteriormente, el **microcontrolador** a utilizar es el MGM240PA22VNA3, un microcontrolador con Zigbee, BLE y Thread que se alimenta a 3.3V y 26 pines de entrada/salida. Este microcontrolador se programa utilizando los pines SWCLK, SWDIO, SWO y RESET, por lo que hemos de disponer de una cabecera de pines con estos cuatro pines, junto con tierra y 3.3V. Para reducir el ruido en la alimentación y proteger al microcontrolador, colocaremos varios condensadores entre 3,.3V y Gnd.

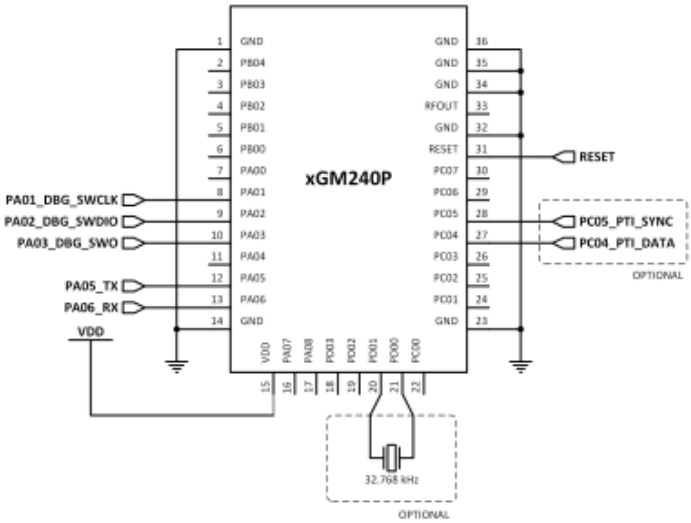

Fig 11: Pines del MGM240PA22VNA3

Para el **sensor de temperatura y humedad** hemos seleccionado el sensor DHT22 que se alimentará a 3.3V y se comunica con el microcontrolador mediante un único pin utilizando el protocolo One-Wire. Este sensor medirá la temperatura con una resolución de 0.1ºC y la humedad con una resolución de 0.1% y será capaz de medir la temperatura entre -40ºC y 80ºC.

El **sensor PIR** detectará la presencia de un ser humano en su campo de visión, indicándolo poniendo una señal a Vcc. Por ello hemos seleccionado el sensor PIR EKMC1604112. Este sensor se alimenta a 3.3V, consume unos 170uA y es capaz de detectar presencia de un ser humano hasta unos 6 metros de distancia.

El **botón** permitirá que el usuario pueda mandar señales directamente a la red Zigbee. Este botón irá conectado a un pullup interno del microcontrolador y conectará el pin a tierra cuando se encuentre pulsado.

Como actuador colocaremos un **diodo LED** THT para poder visualizar señales transmitidas por la red Zigbee, ya sea el estado del dispositivo, la señal de identificación o una señal mandada por el usuario. Junto al diodo hemos de colocar una resistencia para poder conectarlo directamente al microcontrolador.

Otro actuador que utilizaremos será un **relé** que conectará o desconectará una salida a la corriente alterna. Para reducir el consumo energético del sistema, utilizaremos un relé biestable de 3.3V que mantiene de forma pasiva la posición del relé. En las entradas del relé hemos de colocar un transistor bipolar con sus respectivas resistencias de polarización y diodos de protección para evitar daños a la hora de desactivar la bobina. El relé a usar será el relé G6SK-2-DC3, un relé capaz de soportar hasta 0.5 amperios de corriente alterna y a sus entradas de 3.3V irán conectados los transistores PMBT2222,215 y los diodos RS1M.

Las distintas configuraciones tienen distintas formas de alimentación, ya sea por corriente alterna o por batería de litio.

Cuando el circuito se encuentra **alimentado por la corriente alterna**, éste utilizará una fuente de alimentación AC-DC a 3.3V. Como medios de protección del circuito utilizaremos un fusible reseteable y un varistor, colocados a la entrada de la fuente de alimentación. La fuente de alimentación MP-LDE03-20B03 será capaz de ofrecer 700mA a 3.3V de corriente continua.

Los dispositivos que no puedan conectarse continuamente a la corriente alterna se **alimentarán utilizando una batería de litio**. Para ello hemos colocado un conector para dicha batería de litio, un regulador lineal LDO (AP7362-33SP-13) para transformar la tensión a 3.3V con un dropout de 190mV y una corriente

máxima de 1.5A, un cargador para la batería (MP2603EJ-LF-P) con sistema de protección de sobrecarga y sobrecalentamiento, y un conector micro-usb para alimentar y cargar la batería desde un USB a 5V.

Finalmente, para poder realizar pruebas cómodamente, ambas configuraciones contarán con una **alimentación 3.3V** utilizando un regulador externo al circuito.

Las resistencias y los condensadores que utilizaremos en este sistema serán de tipo SMD para poder soldarlos mediante placa caliente (Hot Plate) y serán todos de tamaño 1206 (3216 métricos).

A continuación, podemos observar la lista de materiales (Bill Of Materials) de cada una de las configuraciones e incluye todos los componentes y sus precios que se soldarán al prototipo final.

DETALLES DEL COMPONENTE		TOTAL	36,36€
Descripción corta del material (incluye Fabricante y referencia)	Precio	Cantidad	Subtotal
MGM240PA22VNA3	13,090€	1	13,090€
G6SK-2-DC3	4,250€	1	4,250€
NPN Transistor PMBT2222,215	0,101€	2	0,202€
Diode RS1M	0,030€	2	0,061€
MP-LDE03-20B03 3V3 Power Supply	5,650€	1	5,650€
Fusible Reseteable	0,128€	1	0,128€
Varistor 230V	0,940€	1	0,940€
LED Diode	0,024€	1	0,024€
DHT22 Thermo Sensor	2,480€	1	2,480€
THT Button	0,070€	1	0,070€
PIR Sensor EKMC1604112	9,020€	1	9,020€
Screw Terminal	0,108€	2	0,216€
Programming Header	0,184€	1	0,184€
Resistor 1206	0,005€	3	0,015€
Capacitor 1206	0,010€	3	0,030€

Fig 12: Lista de Materiales Repetidor Zigbee

DETALLES DEL COMPONENTE		TOTAL	18,29€
Descripción corta del material (incluye Fabricante y referencia)	Precio	Cantidad	Subtotal
MGM240PA22VNA3	13,090€	1	13,090€
3V3 LDO Regulator	0,500€	1	0,500€
Li-Ion Battery Charger	1,630€	1	1,630€
LED Diode	0,024€	1	0,024€
DHT22 Thermo Sensor	2,480€	1	2,480€
THT Button	0,070€	1	0,070€
Micro-Usb Connector	0,188€	1	0,188€
JST Receptacle	0,060€	1	0,060€
Programming Header	0,184€	1	0,184€
Resistor 1206	0,005€	2	0,010€
Capacitor 1206	0,010€	5	0,050€

Fig 13: Lista de Materiales Dispositivo Final Zigbee

4.4 Diseño del esquemático

Al diseñar el esquemático hemos dividido el documento en distintas partes, dependiendo del rol que tengan en el circuito.

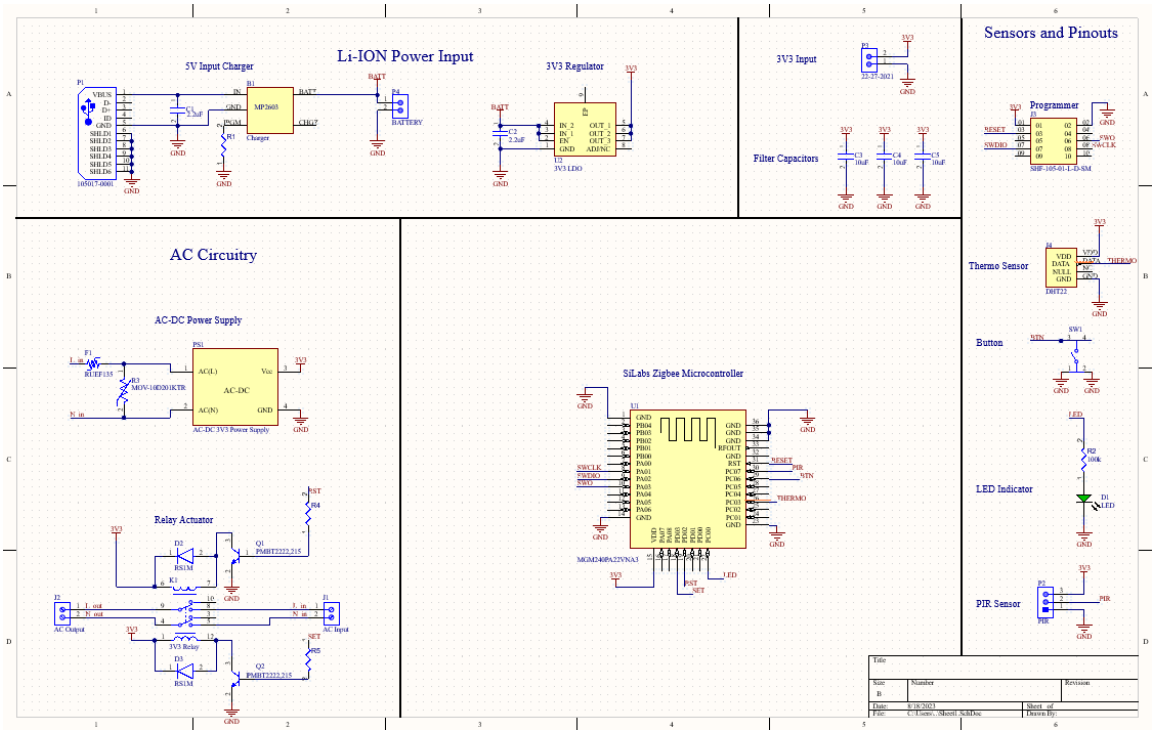

Fig 14: Esquemático del circuito

En el centro del esquemático vemos el microcontrolador MGM240PA22VNA3 indicando todos los pines que tiene conectados a los otros componentes del circuito.

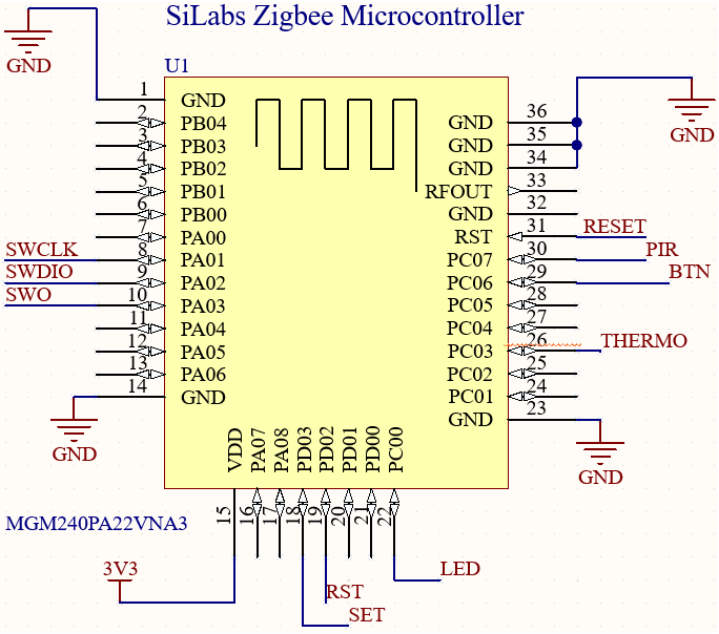

Fig 15: Esquemático del Microcontrolador

En la parte superior izquierda se encuentran todos los componentes relacionados con la alimentación a través de batería de litio. Esto incluye, el conector micro-usb, un condensador de filtrado, el cargador para la batería de litio junto con su resistencia que limita la corriente máxima de carga, el conector de la batería de litio, otro condensador de filtrado y finalmente el regulador lineal LDO.

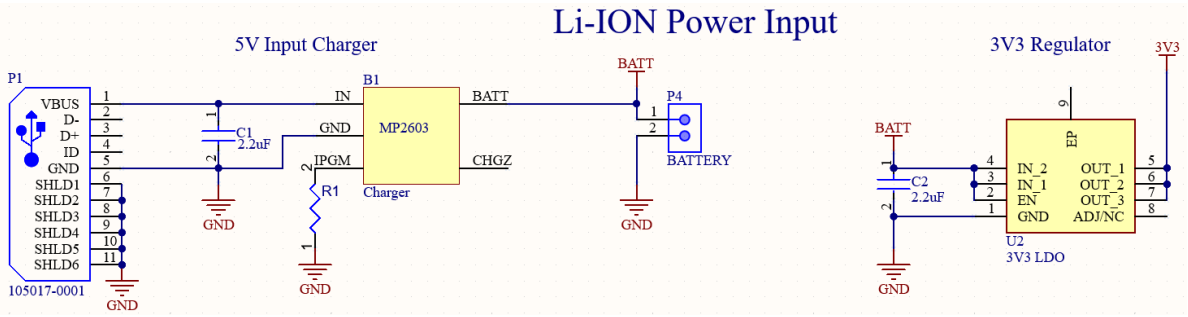

Fig 16: Esquemático de la Alimentación por Batería

En la parte inferior izquierda podemos ver los circuitos de corriente alterna, que incluye el conversor AC-DC junto con su varistor y fusible, los conectores de entrada y salida de corriente alterna, y todo el circuito del relé, que incluye sus diodos de protección, transistores y resistencias de polarización. Debido a que hemos escogido un relé biestable, este dispone de dos entradas de 3.3V para colocarlo en estado conectado o desconectado.

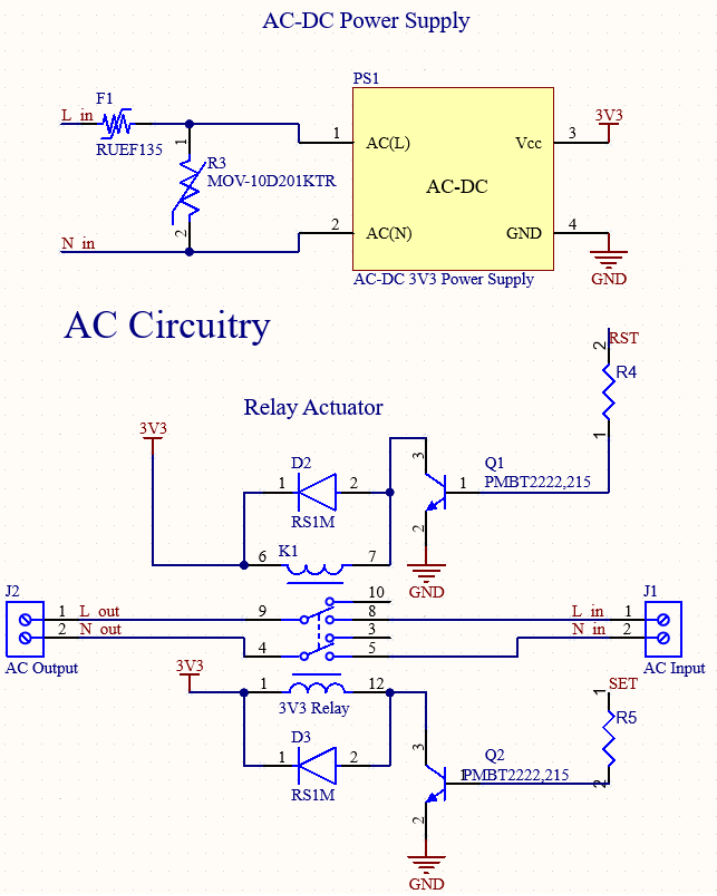

Fig 17: Esquemático de la Alimentación Alterna

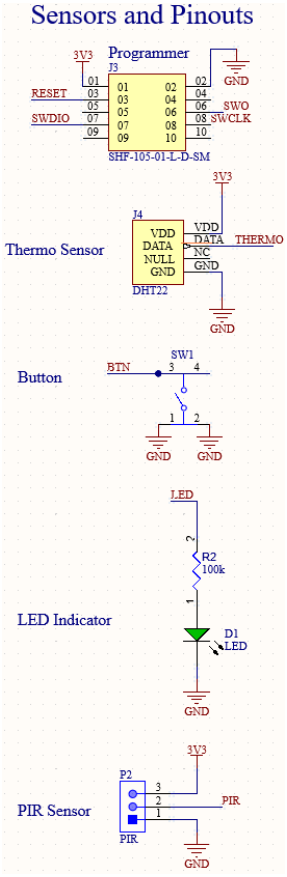

Fig 18: Esquemático de las Entradas y Salidas

En la parte de la derecha tenemos todas las entradas y salidas de señal del sistema. Esto incluye el conector de programación, el sensor de temperatura, el botón, led y sensor PIR. Todos estos se encuentran conectados directamente al microcontrolador para que pueda leer o escribir las señales de sus puertos.

Finalmente, en la parte superior disponemos de la entrada de alimentación de 3.3V y de tres condensadores de filtrado para eliminar todo el ruido en el rail de alimentación 3.3V.

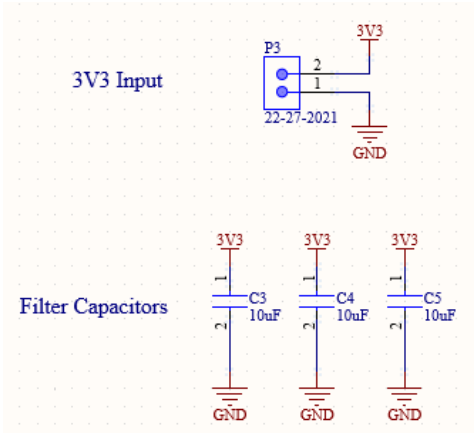

Fig 19: Esquemático de la Alimentación 3V3 y Filtrado

4.5 Diseño de la PCB

A la hora de diseñar la PCB hemos colocado los componentes de forma que se reduzca la distancia de los caminos que lleven corrientes elevadas. Es por ello que, al igual que en el esquemático, hemos agrupado los componentes que tengan roles similares en el sistema.

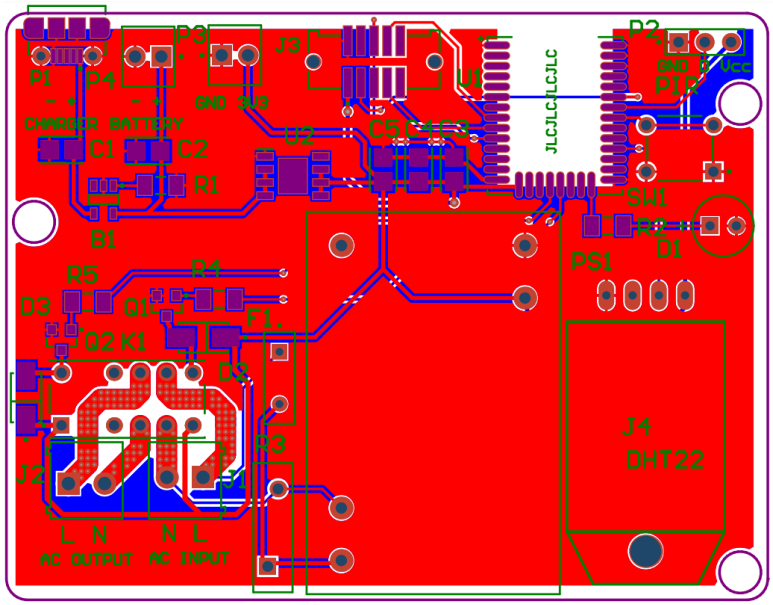

Fig 20: Vista superior de la PCB

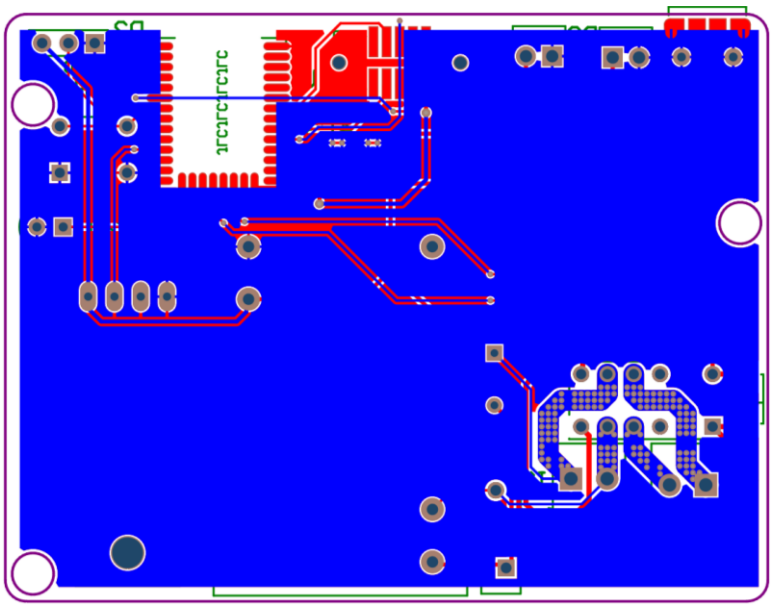

Fig 21: Vista inferior de la PCB

En la parte superior izquierda hemos colocado todos los componentes relacionados con la alimentación por baterías de litio, poniendo los conectores en el borde superior de la placa e indicando su polarización.

En la parte inferior izquierda hemos colocado los componentes relacionados con la corriente alterna y la alimentación por corriente alterna. La corriente máxima que puede atravesar el relé es de 2 amperios, por lo que, para minimizar la resistencia del circuito entre el relé y los conectores, hemos utilizado trazas de 2mm de ancho en ambas capas, y los hemos conectado mediante numerosas vías a lo largo del camino.

En la parte de la derecha hemos colocado los demás sensores y actuadores del circuito junto con el microcontrolador, situado en el borde superior, de forma que se minimicen las trazas de las señales analógicas y reducir el ruido en esas señales.

Una vez diseñadas las PCB deberemos soldar los componentes correspondientes a cada una de las configuraciones. La placa que funcionará como repetidor Zigbee tendrá todos los sensores y actuadores y los componentes relacionados a la alimentación por corriente alterna. La placa que funcionará como dispositivo final de Zigbee tendrá únicamente soldado el sensor de temperatura, el botón y el indicador LED, junto con los componentes necesarios para la alimentación mediante batería de Litio.

En la siguiente tabla podemos observar los componentes y sus indicadores según la configuración deseada.

CONFIGURACIÓN REPETIDOR ZIGBEE		CONFIGURACIÓN DISPOSITIVO FINAL ZIGBEE	
Component	Designators	Component	Designators
MGM240PA22VNA3	U1	MGM240PA22VNA3	U1
G6SK-2-DC3	K1	3V3 LDO Regulator	U2
NPN Transistor PMBT2222,215	Q1 Q2	Li-Ion Battery Charger	B1
Diode RS1M	D2 D3	LED Diode	D1
MP-LDE03-20B03 3V3 Power Supply	PS1	DHT22 Thermo Sensor	J4
Fusible Reseteable	F1	THT Button	SW1
Varistor 230V	R3	Micro-USB Connector	P1
LED Diode	D1	JST Receptacle	P4
DHT22 Thermo Sensor	J4	Programming Header	J3
THT Button	SW1	Resistor 1206	R1 R2
PIR Sensor EKMC1604112	P2	Capacitor 1206	C1 C2 C3 C4 C5
Screw Terminal	J1 J2		
Programming Header	J3		
Resistor 1206	R2 R4 R5		
Capacitor 1206	C3 C4 C5		

Fig 22: Tabla de Componentes por Configuración

5 Diseño del firmware

Para desarrollar el firmware debemos conocer la estructura del protocolo Zigbee y las configuraciones necesarias para hacer funcionar los nodos de nuestra red. Una vez conozcamos el protocolo, desarrollaremos el firmware de nuestros nodos de forma acorde con las especificaciones del protocolo Zigbee.

5.1 Estructura Zigbee

Para enviar información, los datos de Zigbee se encuentran divididos en distintos endpoints. Estos endpoints permiten crear cierta independencia a la hora de enviar información por Zigbee, permitiendo que distintas tareas publiquen información similar por Zigbee utilizando distintos endpoints.

Los endpoints a su vez se encuentran divididos en numerosos clusters. Cada clúster se encarga de gestionar un tipo de información diferente; es decir, el clúster de información básica se encarga de almacenar y transmitir la información básica del nodo, mientras que el clúster de información on/off se encarga de transmitir información sobre el estado de encendido o apagado de un periférico.

Cada clúster puede actuar como servidor, que almacena la información del estado del propio nodo y puede recibir comandos de otros nodos; o puede actuar como cliente, que únicamente puede enviar comandos o modificar atributos a clusters de tipo servidor de otros nodos.

Cada clúster de tipo servidor cuenta finalmente con atributos y comandos según el tipo de clúster. Los atributos nos permiten leer información del clúster, como el estado de encendido o apagado de un relé o el tiempo de encendido restante; mientras que los comandos nos permiten recibir mensajes de modificar las variables anteriormente mencionadas, como los comandos de encender o apagar un relé, y ejecutar tareas correspondientes al clúster, como el comando de identificar el dispositivo Zigbee.

Otra configuración que Zigbee pone también a nuestra disposición es la capacidad de reportar atributos de forma automática. Esta configuración nos permite especificar el valor mínimo de cambio para reportar un atributo, el tiempo máximo entre reportes y el tiempo mínimo entre reportes. Esto nos permite reportar atributos cada cierto tiempo o cada vez que su valor cambia o una combinación de éstas.

Toda la información sobre los tipos de clusters junto con sus atributos, comandos y tareas que ejecutan cada uno de ellos se puede leer en la documentación de los clusters de Zigbee ("Zigbee Cluster Library Specification") [12].

5.2 Configuración Zigbee de los nodos

Una vez estudiada la estructura de los datos de Zigbee hemos de elegir los endpoints, los clusters y los atributos a utilizar en este trabajo.

Como disponemos de dos configuraciones distintas de nuestro hardware, cada configuración tendrá su propia estructura de Zigbee. En la siguiente figura indicaremos los endpoints y los clusters utilizados en cada una de las configuraciones de hardware.

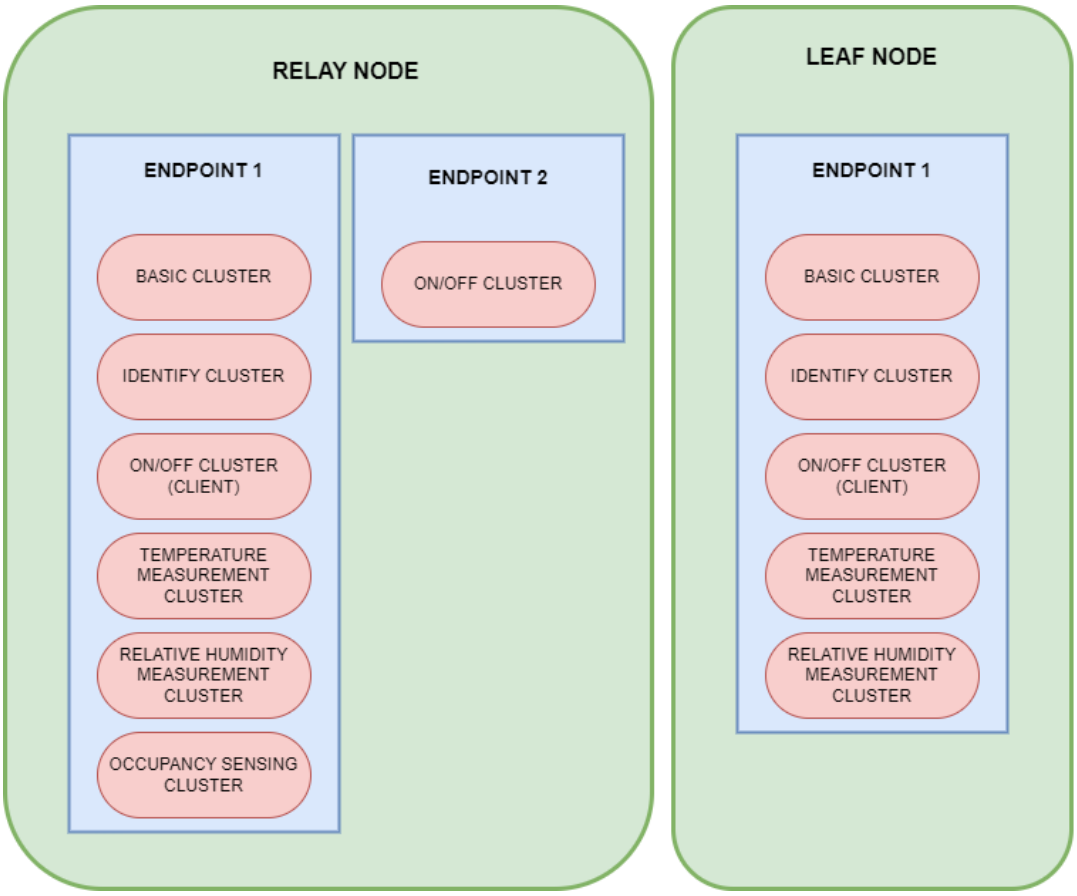

Fig 23: Configuración de la Estructura Zigbee

A continuación indicaremos más detalles sobre los endpoints, los clusters, los atributos y los comandos de Zigbee que utilizaremos.

- Endpoint 1:

En este primer endpoint vamos a almacenar todos los datos de los sensores que colocaremos en la placa (Sensor de temperatura y humedad, sensor PIR, botón). Adicionalmente utilizaremos el diodo LED para poder identificar el dispositivo.

Para ello utilizaremos los siguientes clusters:

- Basic: Información básica del dispositivo. Utilizando los atributos ZCLVersion (0x08), ManufacturerName ("Paris"), ModelIdentifier ("TFG_Relay" o "TFG_Leaf") y PowerSource (0x01 o 0x03).

- Identify: Permite mandar comandos para identificar el dispositivo. Utilizando su atributo IdentifyTime (0) y los comandos Identify, IdentifyQuery y IdentifyQueryResponse.

- On/Off: Este clúster será de tipo cliente y enviará comandos de On, Off y Toggle a otros nodos de la red.

- Temperature Measurement: Almacena información de medidas de temperatura (utilizando el sensor de temperatura y humedad). Utilizando los atributos MeasuredValue, MinMeasuredValue (0x8000) y MaxMeasuredValue (0x8000).

- Relative Humidity Measurement: Almacena información de medidas de humedad (utilizando el sensor de temperatura y humedad). Utilizando los atributos MeasuredValue, MinMeasuredValue (0xFFFF) y MaxMeasuredValue (0xFFFF).

- Occupancy Sensing: Almacena información de ocupación o de presencia detectada por el sensor PIR. Utilizando los atributos Occupancy, OccupancySensorType (0x00).

- Endpoint 2:

  En este segundo endpoint situaremos la salida de los actuadores. Para ello utilizaremos el clúster de On/Off y lo conectaremos mediante firmware al relé. Utilizando los atributos OnOff (0), GlobalSceneControl (1), OnTime (0) y OffWaitTime (0), y los comandos Off, On, Toggle, On with recall global scene y On with timed off.

5.3 Desarrollo del firmware

Para desarrollar el firmware utilizaremos Simplicity Studio, el entorno de desarrollo de Silicon Labs diseñado para programar sus microcontroladores. Este programa dispone de una interfaz gráfica para configurar los clusters de Zigbee por lo que toda la configuración del punto 5.2 se puede realizar mediante dicha interfaz gráfica. Adicionalmente utilizando componentes creados por SiLabs podemos configurar los pines del microcontrolador para cada uso que necesitemos darle, incluyendo interrupciones, modo del puerto y resistencias de pull-up/down.

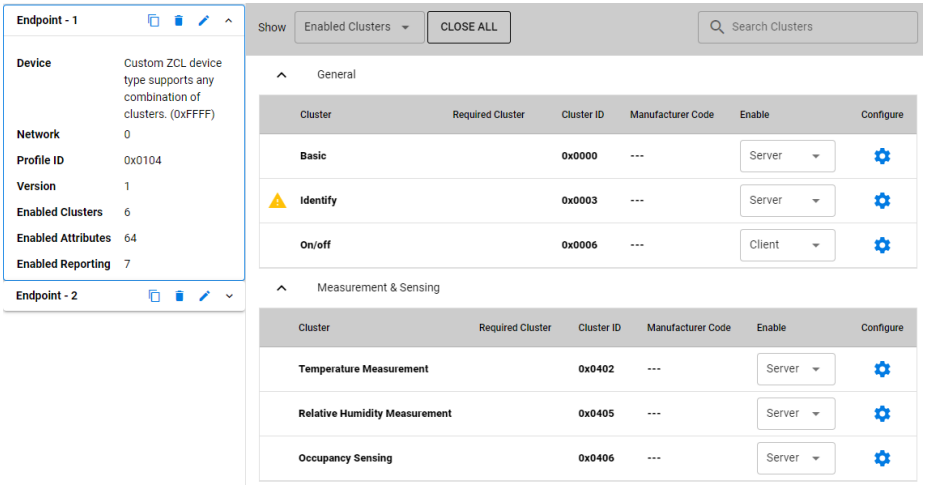

Fig 24: Pantalla de configuración de Zigbee

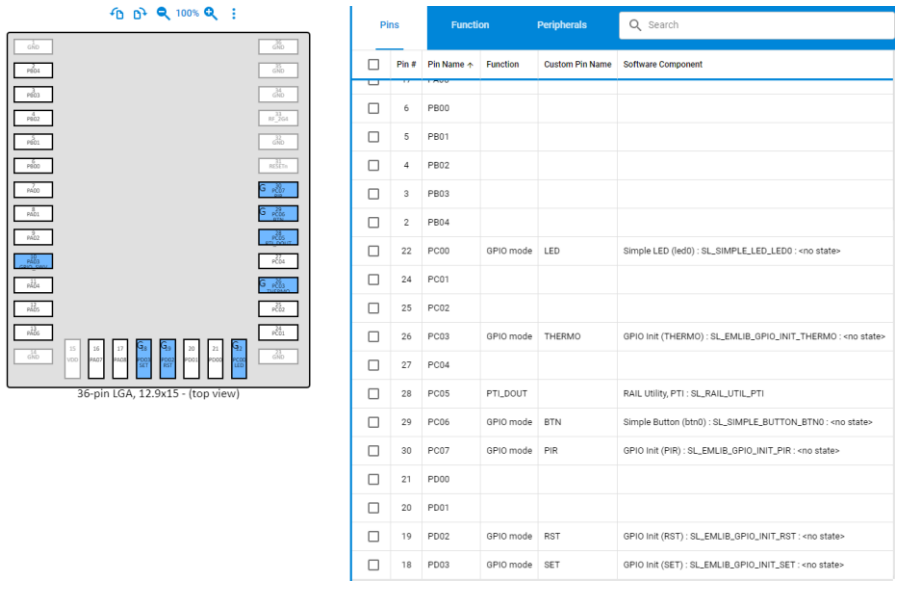

Fig 25: Pantalla de configuración de los pines

El código de este TFG se dividirá en dos proyectos, uno para los nodos que se alimenten mediante la red alterna y que actuarán como repetidores de la red, y otro para los nodos que se alimentan por baterías, se encontrarán

principalmente en estado de bajo consumo y actuarán como dispositivos finales de la red.

Cada uno de los proyectos consistirá de un archivo main que se encargará de lanzar la aplicación Zigbee, un archivo app que gestiona la red Zigbee, inicializa los periféricos y conecta cada uno de los clusters a su periférico correspondiente, y finalmente una serie de archivos que se encargan de gestionar cada uno de los periféricos, es decir actuar el relé o leer los datos de los sensores o pulsadores.

A continuación podremos observar los diagramas de flujo simplificados de ambos proyectos.

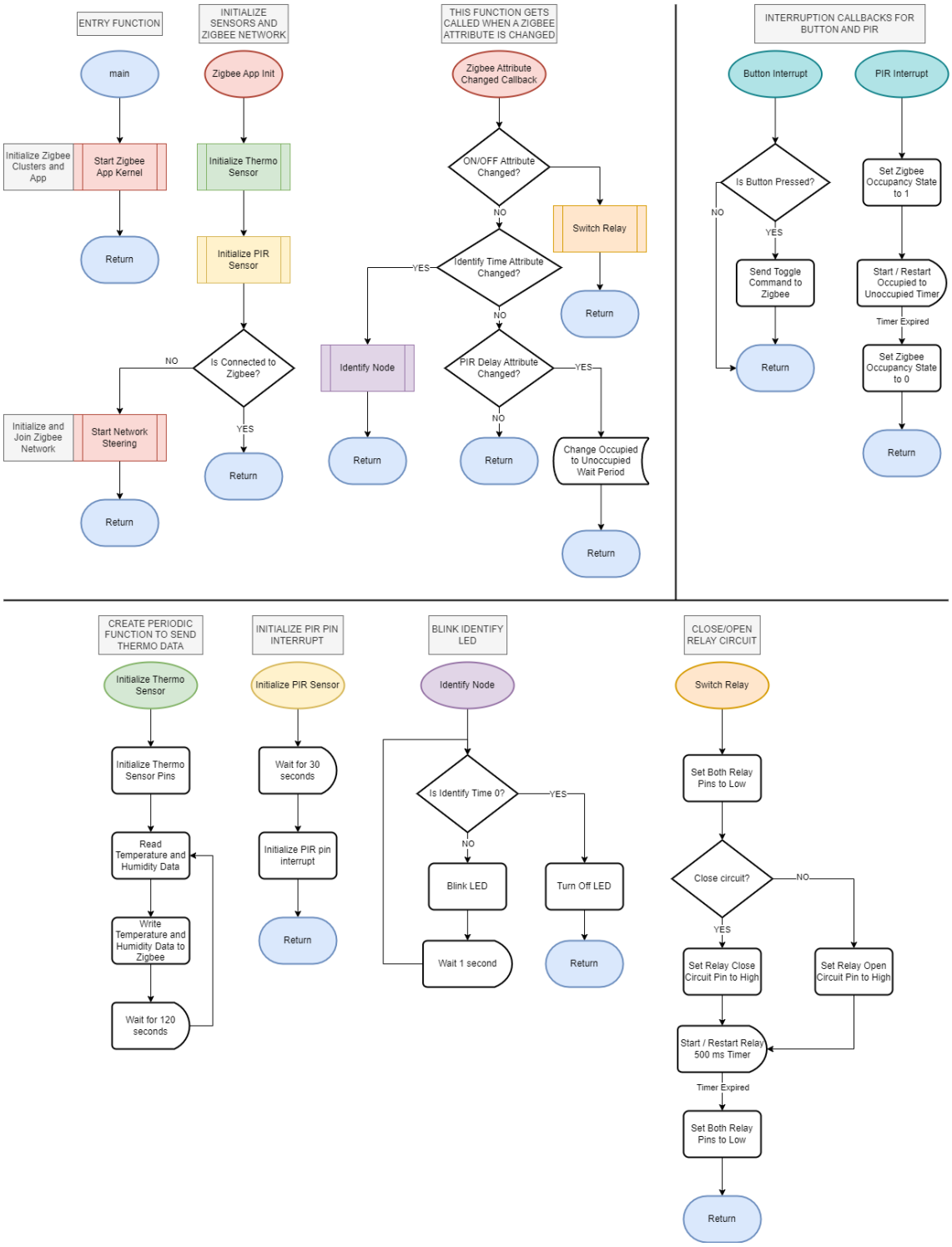

Fig 26: Diagrama de flujo dispositivo repetidor

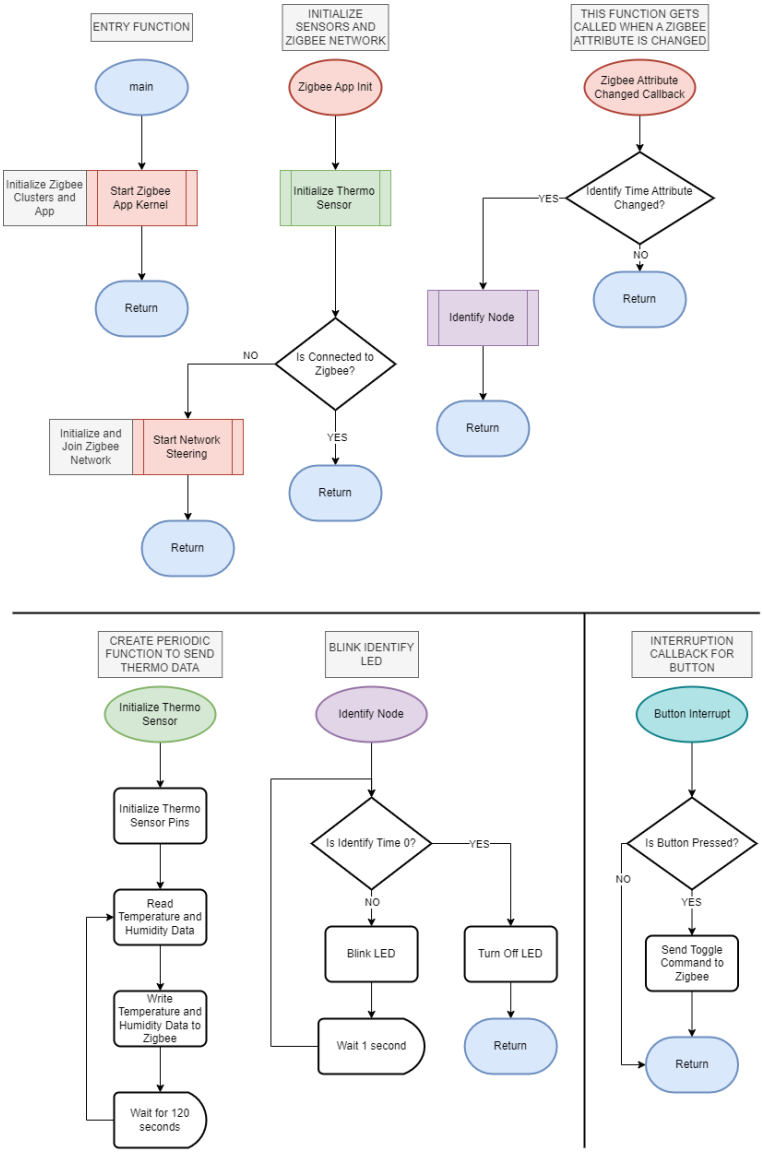

Fig 27: Diagrama de flujo dispositivo final

Los archivos de código fuente de ambos proyectos se pueden observar en más detalle en los Anexos I y II.

6 Diseño del software

Para permitir la conexión entre la red Zigbee y una interfaz gráfica para el usuario hemos de utilizar numerosos programas que nos permitan traducir los datos recibidos de Zigbee o enviados a Zigbee, almacenar dichos datos, y mostrarlos en una web.

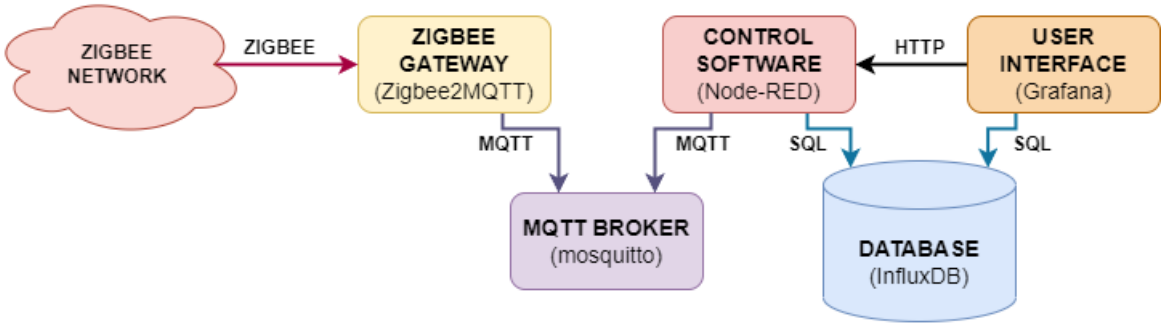

Fig 28: Diagrama de la Arquitectura

A continuación, hablaremos de los usos de cada uno de los programas a utilizar y la configuración necesaria para ponerlos en funcionamiento.

6.1 Gateway Zigbee

Para comunicarnos con la red Zigbee desde software necesitamos un gateway que nos permita conectar un ordenador a la red. Para ello hemos conectado un dongle USB que funcionará como controlador de la red a un servidor central y utilizaremos un programa para capaz de traducir los mensajes enviados por dicho dongle USB a un protocolo IP. Entre los programas diseñados para esto, los principales son ZHA (Zigbee Home Automation) y Zigbee2MQTT. ZHA [13] se conecta con Home Assistant para poder interactuar con la red directamente desde Home Assistant, Zigbee2MQTT [14] utiliza un broker MQTT externo para gestionar toda la configuración de la red, los datos recibidos de la red y los datos a enviar.

Hemos seleccionado Zigbee2MQTT ya que nos permite crear automatismos y gestionar la red con otros programas independientes de Home Assistant.

Para poder conectar este programa a nuestra red Zigbee hemos de conectar un dispositivo USB que funcionará como el coordinador de la red Zigbee. A continuación, instalaremos una serie de archivos de configuración (“converters”) que especifiquen los clusters y los endpoints utilizados por cada dispositivo Zigbee. Ya que los dispositivos Zigbee utilizados han sido creados a

medida, hemos de crear nuestros propios archivos de configuración para poder conectarlos a Zigbee2MQTT, los cuales se pueden observar en el Anexo III.

6.2 Servicio de control

Una vez conectada la red Zigbee a MQTT hemos de seleccionar un programa que nos permita crear automatismos y nos permita enviar los datos de una interfaz con el usuario a MQTT y viceversa.

Para esta tarea buscamos un programa que nos permita crear ese tipo de automatismos utilizando una interfaz gráfica y sin necesidad de tener conocimientos de programación, para así facilitar el diseño y la modificación de los automatismos por las personas responsables.

Es por ello que hemos escogido Node-RED [15], una herramienta de programación por flujos que nos permite crear automatismos simplemente moviendo y conectando los nodos de un flujo.

Los flujos creados para este TFG envían la información recibida por los sensores de la red a la base de datos, toman también datos de la interfaz con el usuario

Los flujos creados para este TFG recuperan los datos ambientales transmitidos por la red y los almacenan en la base de datos. Para interactuar con la red desde una interfaz gráfica hemos creado un flujo que traduce los datos recibidos por una petición HTTP a los dispositivos de la red. A continuación mostraremos los diagramas de flujo de los flujos mencionados.

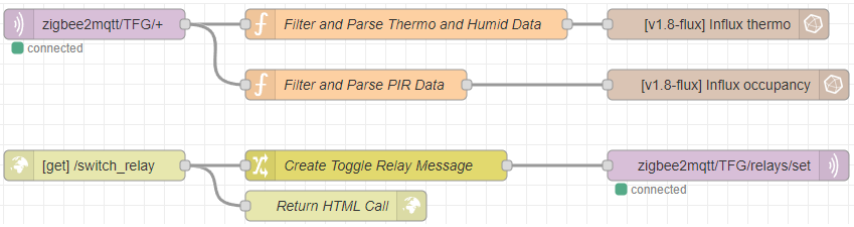

Fig 29: Flujos Node-RED

Adicionalmente se han desarrollado varios flujos que, de forma automática encienden o apagan el relé actuador según los datos recibidos por los sensores. Por ejemplo se ha diseñado un flujo que enciende una bombilla cuando el PIR detecta la presencia de un ser humano. También se ha implementado un flujo que conecta un ventilador cuando la temperatura de una

habitación es superior a una temperatura máxima indicada y lo desconecta cuando es inferior a una temperatura mínima indicada.

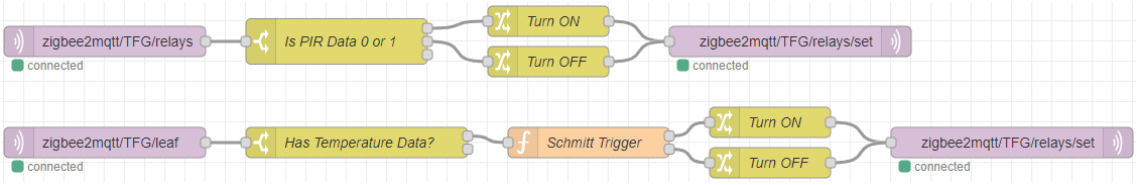

Fig 30: Flujos de Automatización Node-RED

6.3 Almacenamiento de datos

Para almacenar los datos enviados por los sensores hemos utilizado InfluxDB [16], una base de datos de series temporales especializada para almacenar datos de medidas continuas de sensores IoT. Las bases de datos de series temporales aprovechan el hecho de que los datos recibidos suelen ser periódicos y las medidas suelen ser continuas para reducir el tamaño de la marca de tiempo(timestamp) y de los datos.

En lugar de almacenar el timestamp de cada uno de los datos, las bases de datos de series temporales los agrupa según la periodicidad de éstos y almacena únicamente la variación en la periodicidad, la cual se puede almacenar en un menor número de bits.

Las bases de datos de series temporales aprovechan también que los datos suelen ser continuos para reducir el tamaño de los datos almacenados, almacenando únicamente la variación de estos datos y eliminando los ceros iniciales y finales.

Para este trabajo almacenaremos todos los datos medidos por los diversos sensores en una base de datos dedicada al TFG.

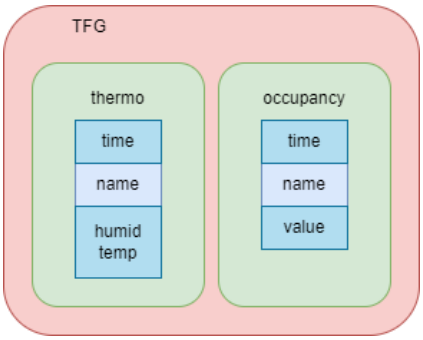

Fig 31: Diagrama de la Base de Datos

Los datos de los sensores de temperatura y humedad serán almacenados en la métrica "*thermo*" que contendrá el valor de tiempo "*time*", el tag de "*name*" indicando el nombre del nodo y los campos de "*temp*" y "*humid*" que almacenarán el valor numérico de la temperatura y humedad en grados centígrados.

Los datos de los sensores de proximidad serán almacenados en la métrica "*occupancy*" que contendrá el valor de tiempo "*time*", el tag de "*name*" indicando el nombre del nodo y el campo de "*value*" que indicará si ha detectado presencia (1) o no (0).

6.4 Interfaz con el usuario

Finalmente hemos de tener una interfaz gráfica que nos permita tanto tener botones que el usuario pueda pulsar para ejecutar flujos, como tener gráficas o indicadores que nos muestren la información enviada por los sensores. Para ello hemos utilizado Grafana [17], una herramienta diseñada para la visualización de datos que nos permite crear gráficos a partir de los datos de InfluxDB y (utilizando plugins de Grafana) enviar mensajes a Node-Red para ejecutar rutinas.

Para mostrar los datos de los sensores hemos creado una gráfica que nos muestra la evolución de la temperatura a lo largo del tiempo. Hemos añadido también un botón que nos permite encender o apagar el relé que se encuentra conectado a una bombilla. Finalmente hemos añadido también un indicador que muestra si el sensor PIR ha detectado la presencia de un ser vivo o no.

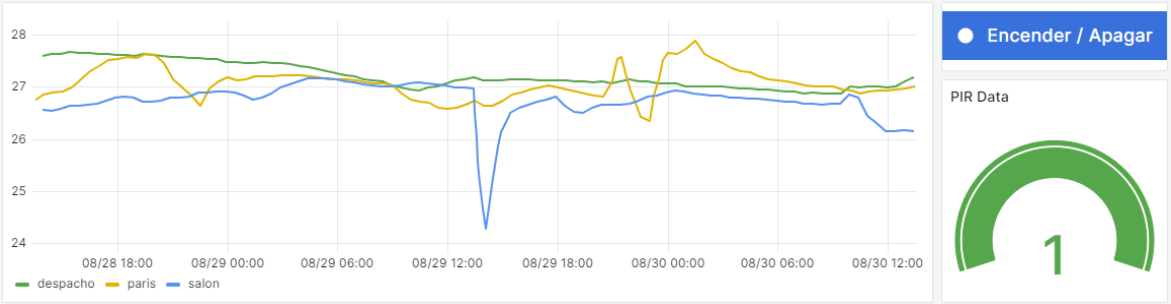

Fig 32: Pantalla de visualización de datos

6.5 Hardware utilizado

Como punto final en el diseño del software es importante mencionar el hardware utilizado para correr todos los programas mencionados.

Para correr las aplicaciones mencionadas buscamos un equipo con un consumo de energía mínimo y también con un precio bajo para poder utilizar equipos y asegurar la redundancia del sistema. Es por ello que nos hemos decantado por utilizar los ordenadores de la empresa Raspberry Pi. De entre todos sus micro-ordenadores, uno de los más baratos y con mayor eficiencia energética es la Raspberry Pi Zero 2 W [18], un equipo con un procesador de 4 núcleos con un consumo en reposo de 0.7W.

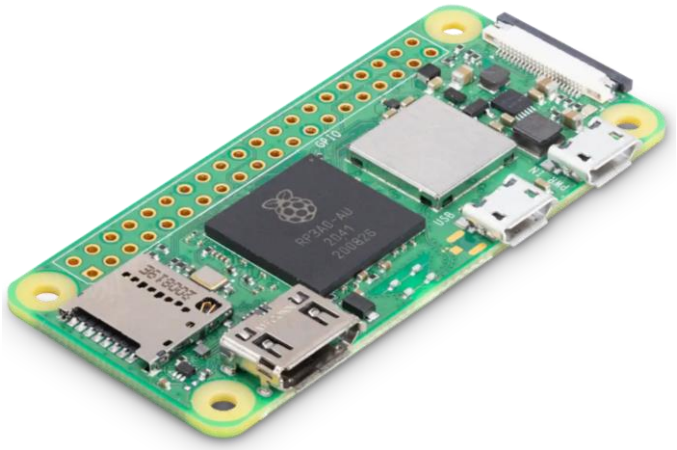

Fig 33: Imagen Raspberry Pi Zero 2 W

Ya que hemos utilizado varios micro-ordenadores para correr estos programas necesitamos utilizar alguna herramienta para dividir los programas entre los distintos equipos. Para esto hemos decidido correr los programas en contenedores y utilizar un orquestador de contenedores para distribuirlos y comprobar su correcto funcionamiento.

Para esto existen varias alternativas, pero las más destacables son Docker Swarm [19] o Kubernetes [20]. De entre estas dos alternativas, Docker Swarm es un programa mucho más ligero y viene instalado junto con Docker. Debido a que los equipos utilizados son tan poco potentes nos hemos decantado por utilizar la opción más ligera.

7 Validación de la red

Una vez tenemos diseñada toda la red, podemos empezar a poner a en funcionamiento todo el sistema para automatizar el edificio inteligente. Este sistema nos permite conectar o desconectar dispositivos según la temperatura o humedad medida por uno o varios sensores, según la detección de un ser humano por el sensor PIR, según la hora del día o según datos recibidos externamente a la red.

7.1 Evaluación de la red

El primer paso a la hora de evaluar la red es soldar los componentes y programar los microcontroladores con el firmware respectivo de cada configuración. Para soldar los componentes se ha utilizado pasta de soldadura y han sido soldados mediante Hot Plate.

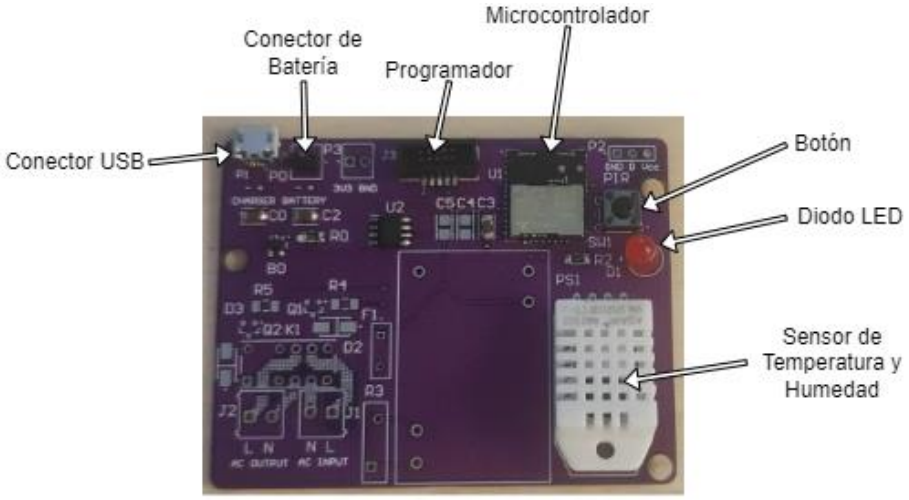

Fig 34: Fotografía prototipo dispositivo final

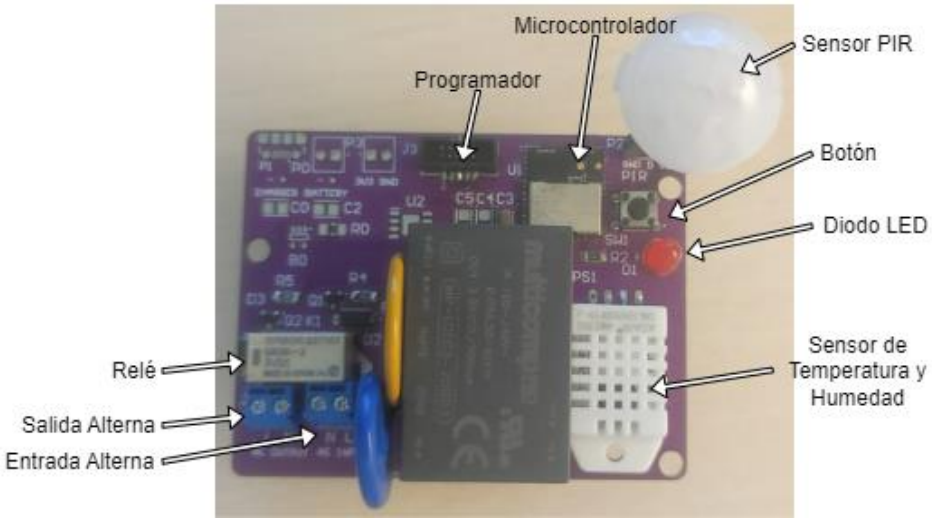

Fig 35: Fotografía prototipo repetidor

Una vez hemos soldado los componentes a cada uno de los prototipos es necesario instalarlos en sus ubicaciones correspondientes y conectar las entradas y salidas correspondientes.

Para crear una **iluminación automática** es necesario conectar el relé del nodo a los bornes de una bombilla, y colocar el sensor PIR en una ubicación que pueda visualizar correctamente el área a medir.

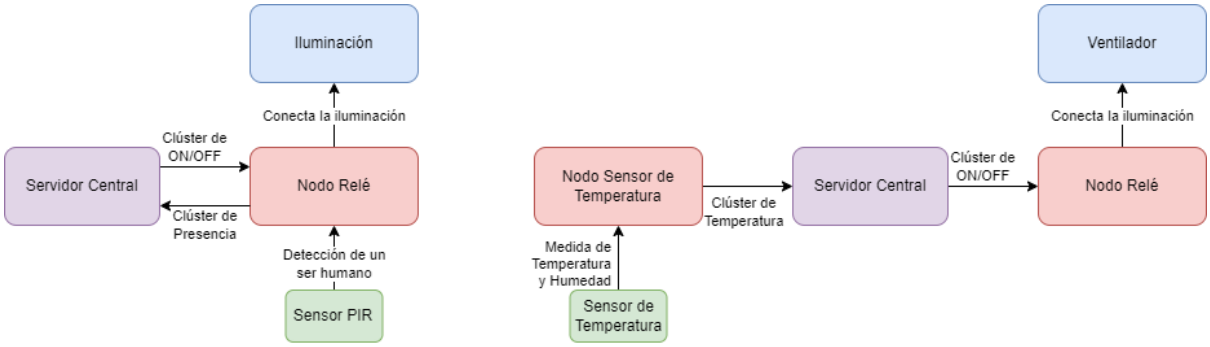

Fig 36: Diagramas Iluminación y Ventilador Automáticos

Para crear un **ventilador inteligente** es necesario conectar el relé del nodo repetidor a ambos bornes de un ventilador, y utilizar un segundo nodo con un sensor de temperatura colocado cerca de la ubicación que se desea controlar la temperatura.

Tal y como se ha indicado en el apartado *6.2 Software de Control* se han diseñado varios flujos que controlan de forma automática estos nodos de la red.

El sistema de **iluminación automática** recibe la señal del clúster de presencia del nodo y espera a que el sensor detecte la presencia de un ser humano. Una vez detectada la presencia se envía una señal de conectar el relé al clúster de ON/OFF. Tras 60 segundos sin detectar la presencia de ningún ser vivo, la señal del clúster de presencia pasa a false y se envía una señal para desconectar el relé.

Por otro lado el **ventilador inteligente** mide la señal del sensor de temperatura y pasa dicho valor por comparador de histéresis el cual conecta el ventilador una vez la temperatura supere una temperatura máxima deseada y no lo desconecta hasta que sea inferior a una temperatura mínima deseada. De esta forma la temperatura se encontrará siempre en el rango entre la temperatura máxima y la mínima.

Tras poner en marcha ambos automatismos somos capaces de evaluar el funcionamiento del sistema. Ambos automatismos funcionan de manera

correcta y al mismo tiempo nos permiten monitorizar el estado de las señales enviadas por la red, como los momentos en los que el sensor PIR ha detectado movimiento. De la misma forma el firmware de todos los prototipos funciona de manera correcta y las placas de circuito impreso han sido diseñadas para su correcto funcionamiento.

Para comprobar el correcto funcionamiento de los sensores de temperatura se han añadido a la red Zigbee otros sensores diseñados por terceros y se han comparado los valores obtenidos por ambos tipos de sensores al colocarse en la misma ubicación. Los sensores utilizados para comparar son los sensores de temperatura y humedad de Tuya, que al igual que los sensores que utilizamos en nuestro TFG tienen una fiabilidad de 0.5ºC y 3% para temperatura y humedad respectivamente pero tienen una resolución mucho más precisa que la de nuestro TFG, ofreciendo hasta dos decimales de precisión. Tras comparando ambos sensores a lo largo de 48 horas (desde las 22:00 hasta las 22:00) se observa que la diferencia entre temperaturas no supera los 0.4ºC, lo cual está muy por debajo de la discrepancia máxima de 1ºC entre los dos sensores. Por ello podemos concluir que los sensores DHT22 son capaces de medir la temperatura de manera correcta.

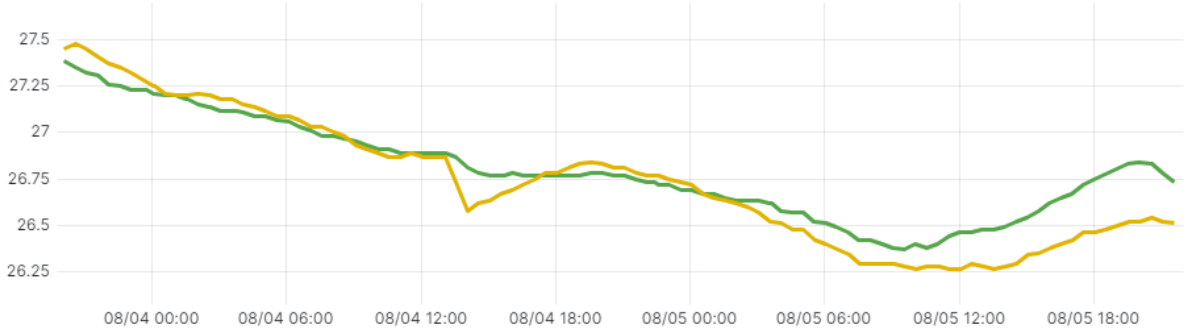

Fig 37: Gráfica de la comparación de los sensores de temperatura

Sin embargo, para las placas que se conectan directamente a la corriente alterna, sería oportuno disponer de un sistema de aislamiento para evitar el contacto con conductores que tengan una tensión elevada. Esto se puede solucionar colocando aislante en los conductores expuestos que contengan dicha tensión y creando una carcasa para el prototipo.

7.2 Complicaciones durante el desarrollo

Una vez tenemos la red funcionando podemos observar que su funcionamiento es adecuado. Sin embargo, este resultado ha sido obtenido tras solucionar numerosas complicaciones durante el desarrollo de la red, que se han ido solucionando una a una.

A la hora de diseñar el circuito impreso se ha tenido que pasar por numerosas revisiones para evitar errores y obtener una placa errónea. Por desgracia la placa final que se ha mandado a fabricar cuenta con varios errores debido a errores humanos a la hora de realizar la revisión. Varios de los componentes se encuentran modelados de forma errónea en la placa, ya sea su escala o que se encuentren invertidos. Estos errores se han podido solucionar de forma sencilla ya sea doblando los polos de los componentes o instalándolos en la capa inferior. Otro de los errores se trata de la falta de una traza que conecta dos pines, por lo que a la hora de solucionar el problema se han tenido que puentear soldando un cable externo. Afortunadamente en la versión final que se encuentra indicada en el apartado *4.5 Diseño de la PCB*, estos cambios han sido solucionados y son únicamente visibles en los prototipos.

En el desarrollo del firmware, debido a la falta de documentación extensa de los métodos de Zigbee por parte de Silicon Labs, hemos tenido que tomar como ejemplo varios de los proyectos de ejemplo que dispone el programa Simplicity Studio.

A la hora de realizar el desarrollo de software nos encontramos de nuevo el problema de la falta de documentación, ya que a la hora de crear los archivos de configuración de zigbee2mqtt, disponemos únicamente de 3 ficheros de ejemplo mínimos que no cuentan con detalles del funcionamiento de cada atributo. Para conocer el funcionamiento de los ficheros de configuración se han realizado ligeros cambios, uno a uno, para conocer cómo interactúa el programa con cada uno de los atributos.

Finalmente a la hora de evaluar la red se han encontrado varios errores. Ente ellos se encuentran problemas a la hora de soldar, ya que varios componentes necesitan soldarse por debajo del componente o tienen pines muy pequeños difíciles de soldar a mano. Para ello se ha utilizado pasta de soldadura y se ha realizado la soldadura mediante Hot Plate ya que no necesita acceso directo a los pines para soldar componentes.

Otro de los problemas que han sido mencionados en el apartado anterior se trata de la seguridad a la hora de alimentar el circuito a 230V, ya que la alimentación alterna se conecta mediante terminales de tornillo. Para solucionar esto, durante la etapa de pruebas se ha conectado a esta terminal un cable con una clavija para conectarla al enchufe cuando se desee poner el sistema en marcha. Una vez finalizadas las pruebas, el prototipo final se encuentra instalado directamente a la red eléctrica y para ello se ha desconectado la corriente en la sección en la que se estaba trabajando.

7.3 Otros procesos de automatización

Aparte de los casos de uso descritos, también existen numerosos casos de uso que se pueden implementar realizando ligeras modificaciones a los prototipos y conectándolos a electrodomésticos del edificio inteligente, ya sea un termo de agua, nevera, congelador o un sistema de climatización.

Podemos crear un **termo de agua automatizado** al conectar la resistencia interna del termo y el termómetro interno a nuestro prototipo. De esta forma podemos medir la temperatura interna del termo de agua y controlar el encendido o apagado de la resistencia interna del termo. Esto nos permite conectar la resistencia del termo durante las horas más baratas del día o durante las horas de mayor producción solar. Adicionalmente se conectará también la resistencia cuando la temperatura del agua sea inferior a una temperatura mínima y así evitar el enfriamiento excesivo del agua. Para aprovechar al máximo la energía producida por placas solares es recomendable conectar la resistencia durante cortos periodos de tiempo para que la energía consumida no supere la energía producida por las placas solares.

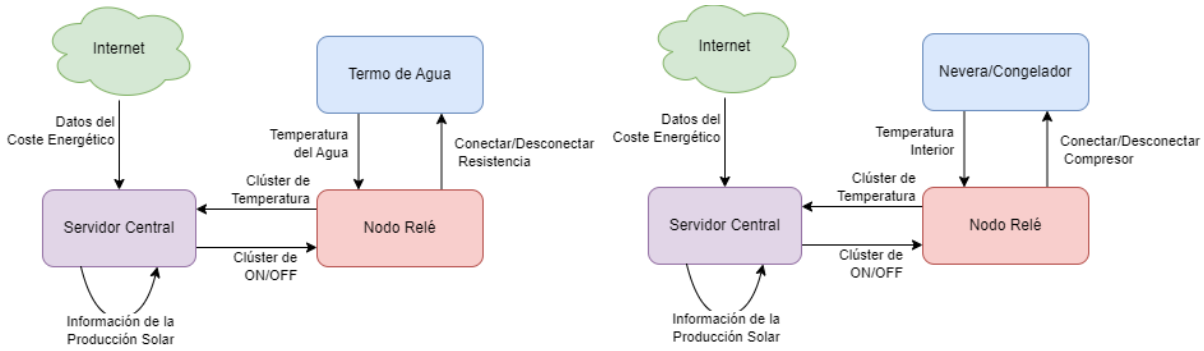

Fig 38: Diagramas termo automatizado y nevera inteligente

Otro caso de uso similar al anterior sería el de una **nevera o congelador inteligente**. Para desarrollar este sistema se debe conectar el compresor del

electrodoméstico al relé y colocar el sensor de temperatura y humedad en el interior de éste. Al igual que el sistema anterior podemos controlar el encendido o apagado del compresor según la hora más barata, la temperatura en el interior del electrodoméstico o la energía producida cada hora.

El último de los casos de uso es un **sistema de refrigeración y calefacción automatizado**. Utilizando sensores de temperatura y humedad colocados en distintas salas de un edificio, colocando sensores PIR y conectando los relés del sistema al sistema de climatización podemos encender, apagar o redirigir el flujo de aire de la climatizadora según los datos recibidos por los sensores.

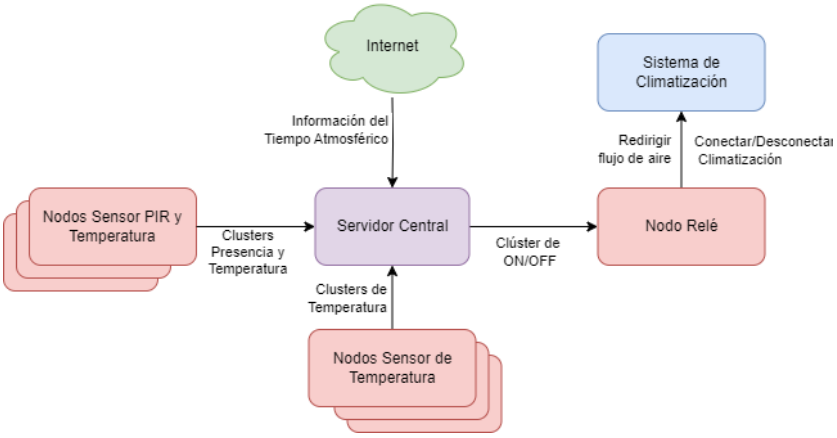

Fig 39: Diagrama sistema de climatización automatizado

Esto nos permite conectar la climatización de forma automática cuando algunas salas tienen su temperatura por debajo de una temperatura dada, desconectar la climatización cuando una sala se encuentra vacía o redirigir el flujo de aire según la temperatura o presencia de seres vivos en las distintas salas del edificio.

8 Conclusiones y futuro del proyecto

En este trabajo de fin de grado se ha creado una red de sensores y actuadores IoT que permiten automatizar el funcionamiento de un edificio inteligente.

Para ello hemos analizado el estado del arte, estudiando las principales tecnologías existentes y hemos seleccionado cuál de ellas es óptima para la red que buscamos diseñar. A continuación, hemos diseñado el hardware y firmware de varios prototipos capaces de enviar datos ambientales a la red Zigbee e interactuar con el entorno. Una vez han sido creados los nodos de la red, se ha conectado dicha red a un equipo y hemos utilizado varios programas como Zigbee2MQTT, Node-RED, InfluxDB y Grafana para enviar y recibir datos a la red y crear automatismos con los datos de la red.

Todo esto nos permite automatizar un edificio para reducir drásticamente el consumo de energía y promover el uso de energía solar para alimentar estos sistemas.

Sin embargo, este trabajo tiene todavía numerosas oportunidades de mejora. Entre ellas se encuentran las siguientes.

Crear sensores o actuadores específicos para cada caso de uso y de menor tamaño y complejidad para poder colocarlos en los distintos lugares deseados.

Eliminar los puntos únicos de fallo (single point of failure) que dispone el sistema. Específicamente añadir redundancia en el coordinador de la red, que en el caso de tener algún fallo, la red dejará de funcionar correctamente. Ya sea mediante varios coordinadores clonados o utilizando otras tecnologías que permitan el nivel de redundancia deseado.

Aislar los conductores que contengan tensiones demasiado elevadas, como la fase de entrada y salida de corriente alterna y los bornes del fusible reseteable, del varistor y de la fuente de alimentación.

Conectar la red creada en este TFG a otros dispositivos IoT creados por terceros y utilizarlos para aumentar la capacidad de la red.

Y finalmente diseñar e imprimir carcasas en las que poder colocar los prototipos y evitar el contacto directo con la electrónica.

9 Bibliografía

[1] Sinha S. (2023) 'State of IoT 2023'. Available at: https://iot-analytics.com/number-connected-iot-devices/

[2] Varghese S.G., Kurian C.P., George V.I., John A., Nayak V. and Upadhyay A., (2019) 'Comparative study of zigBee topologies for IoT-based lighting automation'. Available at: https://doi.org/10.1049/iet-wss.2018.5065/

[3] Elahi A. and Gschwender A. (2009) 'Introduction to the Zigbee Wireless Sensor and Control Network'. Available at: https://www.informit.com/store/zigbee-wireless-sensor-and-control-network-9780137134854?w_ptgrevartcl=Introduction+to+the+ZigBee+Wireless+Sensor+and+Control+Network+_1409785

[4] Lopez N.A.T, Morales J., Parado J., Pasaoa J.R. (2016) 'A Comparative Study of Thread Against Zigbee, Z-Wave, Bluetooth, and Wi-Fi as a Home-Automation Networking Protocol'. Available at: https://www.researchgate.net/publication/309669667_A_Comparative_Study_of_Thread_Against_ZigBee_Z-Wave_Bluetooth_and_Wi-Fi_as_a_Home-Automation_Networking_Protocol

[5] Learte Liarte J. (2018) 'Desarrollo de un ecosistema IoT para la mejora de la eficiencia energética de edificios' Available at: https://zaguan.unizar.es/record/76216/files/TAZ-TFM-2018-932.pdf

[6] Odongo G.Y., Musabe R. Hanyurwimfura D. and Bakari A.D. (2022) 'An Efficient LoRa-Enabled Smart Fault Detection and Monitoring Platform for the Power Distribution System Using Self-Powered IoT Devices'. Available at: https://ieeexplore.ieee.org/document/9817137

[7] Microchip 'PIC32CX-BZ2 and WBZ45 Family Data Sheet'. Available at: https://ww1.microchip.com/downloads/aemDocuments/documents/WSG/ProductDocuments/DataSheets/PIC32CX-BZ2-and-WBZ45-Family-Data-Sheet-DS70005504.pdf

[8] Silicon Labs 'MGM240P Multi-Protocol Wireless Module Data Sheet'. Available at: https://www.silabs.com/documents/public/data-sheets/mgm240p-datasheet.pdf

[9] Espressif 'ESP32-C6 Series Datasheet'. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-c6_datasheet_en.pdf

[10] Espressif 'ESP32-H2 Datasheet'. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-h2_datasheet_en.pdf

[11] Espressif 'ESP Zigbee SDK Programming Guide'. Available at: https://docs.espressif.com/projects/esp-zigbee-sdk/en/latest/esp32/

[12] Zigbee Alliance 'ZigBee Cluster Library Specification'. Available at: https://zigbeealliance.org/wp-content/uploads/2019/12/07-5123-06-zigbee-cluster-library-specification.pdf

[13] 'Zigbee Home Automation'. Available at: https://www.home-assistant.io/integrations/zha/

[14] 'Zigbee2MQTT'. Available at: https://www.zigbee2mqtt.io

[15] 'Node-RED'. Available at: https://nodered.org

[16] 'Influx Data Documentation'. Available at: https://docs.influxdata.com/influxdb/v1.8/

[17] 'Grafana Documentation'. Available at: https://grafana.com/docs/grafana/latest/

[18] 'Raspberry Pi Zero 2 W'. Available at: https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/

[19] 'Docker Documentation'. Available at: https://docs.docker.com/

[20] 'Kubernetes Documentation'. Available at: https://kubernetes.io/docs/home/

ANEXO I: Código fuente proyecto TFG_Relay

A continuación se observará el código fuente del proyecto responsable de gestionar los nodos repetidores de la red Zigbee.

El fichero **main.c** se actúa como fichero de entrada del programa y se encarga de inicializar y lanzar la app de Zigbee.

```
/***************************************************************************//**
 * @file main.c
 * @brief main() function.
 ******************************************************************************
 * # License
 * <b>Copyright 2021 Silicon Laboratories Inc. www.silabs.com</b>
 ******************************************************************************
 *
 * The licensor of this software is Silicon Laboratories Inc. Your use of this
 * software is governed by the terms of Silicon Labs Master Software License
 * Agreement (MSLA) available at
 * www.silabs.com/about-us/legal/master-software-license-agreement. This
 * software is distributed to you in Source Code format and is governed by the
 * sections of the MSLA applicable to Source Code.
 *
 ******************************************************************************/

#include "sl_component_catalog.h"
#include "sl_system_init.h"
#include "FreeRTOS.h"
#include "FreeRTOSConfig.h"
#include "task.h"
#include "sl_led.h"
#include "sl_simple_led_instances.h"
#include "app/framework/include/af.h"
//#undef SL_CATALOG_KERNEL_PRESENT
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
#include "sl_power_manager.h"
#endif
#if defined(SL_CATALOG_KERNEL_PRESENT)
#include "sl_system_kernel.h"
#else
#include "sl_system_process_action.h"
#endif // SL_CATALOG_KERNEL_PRESENT

#ifdef EMBER_TEST
#define main nodeMain
#endif

int main(void)
{
  // Initialize Silicon Labs device, system, service(s) and protocol stack(s).
  // Note that if the kernel is present, processing task(s) will be created by
  // this call.
  sl_system_init();

#if defined(SL_CATALOG_KERNEL_PRESENT)
  // Start the kernel.
  sl_system_kernel_start();
#else // SL_CATALOG_KERNEL_PRESENT
  while (1) {
    // Do not remove this call: Silicon Labs components process action routine
    // must be called from the super loop.
    sl_system_process_action();

    // Let the CPU go to sleep if the system allow it.
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
    sl_power_manager_sleep();
#endif // SL_CATALOG_POWER_MANAGER_PRESENT
  }
#endif // SL_CATALOG_KERNEL_PRESENT

  return 0;
}
```

El fichero **app.c** se encarga de gestionar las llamadas a las diversas otras clases, ya sea cuando se inicializa el programa o cuando recibe un cambio a algún atributo. Adicionalmente se encarga también de inicializar la conexión a la red Zigbee utilizando las librerías network-steering y zll-commissioning.

```
/***************************************************************************//**
 * @file app.c
 * @brief Callbacks implementation and application specific code.
 *******************************************************************************
 * # License
 * <b>Copyright 2021 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * The licensor of this software is Silicon Laboratories Inc. Your use of this
 * software is governed by the terms of Silicon Labs Master Software License
 * Agreement (MSLA) available at
 * www.silabs.com/about-us/legal/master-software-license-agreement. This
 * software is distributed to you in Source Code format and is governed by the
 * sections of the MSLA applicable to Source Code.
 *
 ******************************************************************************/

#include "app/framework/include/af.h"
#include "zll-commissioning.h"
#include "network-steering.h"
#include "find-and-bind-target.h"
#include "general.h"
#include "identify.h"
#include "relay.h"
#include "button.h"
#include "thermo.h"
#include "pir.h"

#define MAIN_ENDPOINT           1

static sl_zigbee_event_t commissioning_event;
static sl_zigbee_event_t finding_and_binding_event;

static void commissioning_event_handler(sl_zigbee_event_t *event)
{
  if (emberAfNetworkState() != EMBER_JOINED_NETWORK) {
    EmberStatus status = emberAfPluginNetworkSteeringStart();
    sl_zigbee_app_debug_println("%s network %s: 0x%02X", "Join", "start", status);
  }
}

static void finding_and_binding_event_handler(sl_zigbee_event_t *event)
{
  if (emberAfNetworkState() == EMBER_JOINED_NETWORK) {
    sl_zigbee_event_set_inactive(&finding_and_binding_event);

    sl_zigbee_app_debug_println("Find and bind target start: 0x%02X",
                                emberAfPluginFindAndBindTargetStart(MAIN_ENDPOINT));
  }
}

/** @brief Stack Status
 *
 * This function is called by the application framework from the stack status
 * handler.  This callbacks provides applications an opportunity to be notified
 * of changes to the stack status and take appropriate action. The framework
 * will always process the stack status after the callback returns.
 */
void emberAfStackStatusCallback(EmberStatus status)
{
  // Note, the ZLL state is automatically updated by the stack and the plugin.
  if (status == EMBER_NETWORK_DOWN) {
    sl_zigbee_app_debug_println("NETWORK DOWN");
  } else if (status == EMBER_NETWORK_UP) {
    sl_zigbee_app_debug_println("NETWORK UP");
    sl_zigbee_event_set_active(&finding_and_binding_event);
  }
}

/** @brief Init
```

```
 * Application init function
 */
void emberAfMainInitCallback(void)
{
  sl_zigbee_app_debug_println("STARTING...");
  sl_zigbee_event_init(&commissioning_event, commissioning_event_handler);
  sl_zigbee_event_init(&finding_and_binding_event, finding_and_binding_event_handler);

#ifdef RELAY
  initRelay();
#endif
#ifdef THERMO
  initThermo();
#endif
#ifdef PIR
  initPIR();
#endif


  sl_zigbee_event_set_active(&commissioning_event);
}

/** @brief Complete network steering.
 *
 * This callback is fired when the Network Steering plugin is complete.
 *
 * @param status On success this will be set to EMBER_SUCCESS to indicate a
 * network was joined successfully. On failure this will be the status code of
 * the last join or scan attempt. Ver.: always
 *
 * @param totalBeacons The total number of 802.15.4 beacons that were heard,
 * including beacons from different devices with the same PAN ID. Ver.: always
 * @param joinAttempts The number of join attempts that were made to get onto
 * an open Zigbee network. Ver.: always
 *
 * @param finalState The finishing state of the network steering process. From
 * this, one is able to tell on which channel mask and with which key the
 * process was complete. Ver.: always
 */
void emberAfPluginNetworkSteeringCompleteCallback(EmberStatus status,
                                                  uint8_t totalBeacons,
                                                  uint8_t joinAttempts,
                                                  uint8_t finalState)
{
  sl_zigbee_app_debug_println("%s network %s: 0x%02X", "Join", "complete", status);
}

/** @brief
 *
 * Application framework equivalent of ::emberRadioNeedsCalibratingHandler
 */
void emberAfRadioNeedsCalibratingCallback(void)
{
  sl_mac_calibrate_current_channel();
}

/** @brief Post Attribute Change
 *
 * This function is called by the application framework after it changes an
 * attribute value. The value passed into this callback is the value to which
 * the attribute was set by the framework.
 */
void emberAfPostAttributeChangeCallback(uint8_t endpoint,
                                        EmberAfClusterId clusterId,
                                        EmberAfAttributeId attributeId,
                                        uint8_t mask,
                                        uint16_t manufacturerCode,
                                        uint8_t type,
                                        uint8_t size,
                                        uint8_t* value)
{
  //sl_zigbee_app_debug_println("ATTRIBUTE CHANGED: %d %d %d", endpoint, clusterId,
attributeId);
#ifdef RELAY
  if (endpoint == RELAY_ENDPOINT
      && clusterId == ZCL_ON_OFF_CLUSTER_ID
      && attributeId == ZCL_ON_OFF_ATTRIBUTE_ID
```

```
      && mask == CLUSTER_MASK_SERVER)
    relayAttributechanged();
#endif
#ifdef LED
  if (endpoint == IDENTIFY_ENDPOINT
      && clusterId == ZCL_IDENTIFY_CLUSTER_ID
      && attributeId == ZCL_IDENTIFY_TIME_ATTRIBUTE_ID
      && mask == CLUSTER_MASK_SERVER)
    identifyAttributechanged();
#endif
#ifdef PIR
  if (endpoint == PIR_ENDPOINT
      && clusterId == ZCL_OCCUPANCY_SENSING_CLUSTER_ID
      && attributeId == ZCL_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY_ATTRIBUTE_ID
      && mask == CLUSTER_MASK_SERVER)
    updatePIRPeriodMS();
#endif
}


/** @brief On/off Cluster Server Post Init
 *
 * Following resolution of the On/Off state at startup for this endpoint, perform any
 * additional initialization needed; e.g., synchronize hardware state.
 *
 * @param endpoint Endpoint that is being initialized
 */
void emberAfPluginOnOffClusterServerPostInitCallback(uint8_t endpoint)
{
#ifdef RELAY
  if(endpoint == RELAY_ENDPOINT)
    relayAttributechanged();
#endif
}
```

El fichero **general.h** se encarga de definir los periféricos conectados en el prototipo, es decir se definirán únicamente las macros de los periféricos que deseemos inicializar y utilizar.

```
/*
 * general.h
 *
 *  Created on: Jul 25, 2023
 *      Author: paris
 */

#ifndef GENERAL_H_
#define GENERAL_H_

#define BUTTON
#define LED
#define THERMO
#define RELAY
#define PIR


#endif /* GENERAL_H_ */
```

Los ficheros **identify.h** e **identify.c** reciben llamadas directamente desde el fichero app.c y se encargan de inicializar y gestionar el parpadeo del diodo LED cuando se desee identificar el dispositivo.

```
/*
 * identify.h
 *
 *      Author: Paris
 */

#ifndef IDENTIFY_H_
#define IDENTIFY_H_

#define IDENTIFY_ENDPOINT           1
#define IDENTIFY_BLINK_S        2
```

```
void initIdentify();
void identifyAttributechanged();

#endif /* IDENTIFY_H_ */
```

identify.c:

```
/*
 * identify.c
 *
 *      Author: Paris
 */

#include "identify.h"
#include "app/framework/include/af.h"

void initIdentify()
{}

void identifyAttributechanged()
{
  uint16_t identifyTime;
  emberAfReadServerAttribute(IDENTIFY_ENDPOINT,
                             ZCL_IDENTIFY_CLUSTER_ID,
                             ZCL_IDENTIFY_TIME_ATTRIBUTE_ID,
                             (uint8_t *)&identifyTime,
                             sizeof(identifyTime));
  if (identifyTime <= 0)
    sl_led_turn_off(&sl_led_led0);
  else if (identifyTime % IDENTIFY_BLINK_S == 0) {
    sl_zigbee_app_debug_println("IDENTIFYING");
    sl_led_toggle(&sl_led_led0);
  }
}
```

Los ficheros **relay.h** y **relay.c** se encargan de la inicialización y de la comunicación con los pines conectados al relé. Al cambiar el estado del clúster ON/OFF se activa durante 500ms el pin correspondiente del relé y luego se desactiva para ahorrar energía.

```
/*
 * relay.h
 *
 *      Author: Paris
 */

#ifndef RELAY_H_
#define RELAY_H_

#define RELAY_ENDPOINT          2
#define RELAY_WAIT_PERIOD_MS    500

void initRelay();
void relayAttributechanged();

#endif /* RELAY_H_ */
```

relay.c:

```
/*
 * relay.c
 *
 *      Author: Paris
 */

#include "relay.h"
#include "app/framework/include/af.h"
#include "em_gpio.h"
#include "sl_emlib_gpio_init_RST_config.h"
#include "sl_emlib_gpio_init_SET_config.h"

sl_zigbee_event_t relay_switch_event;
sl_zigbee_event_t relay_reset_event;

void relay_event_handler(sl_zigbee_event_t *event);
```

```
void initRelay()
{
  sl_zigbee_event_init(&relay_switch_event, relay_event_handler);
  sl_zigbee_event_init(&relay_reset_event, relay_event_handler);
}

void relayAttributechanged()
{
  sl_zigbee_event_set_inactive(&relay_reset_event);
  sl_zigbee_event_set_active(&relay_switch_event);
}

void relay_event_handler(sl_zigbee_event_t *event)
{
  if (event == &relay_switch_event) {
      bool onOff;
      if (emberAfReadServerAttribute(RELAY_ENDPOINT,
                                     ZCL_ON_OFF_CLUSTER_ID,
                                     ZCL_ON_OFF_ATTRIBUTE_ID,
                                     (uint8_t *)&onOff,
                                     sizeof(onOff))
          == EMBER_ZCL_STATUS_SUCCESS) {
           //Set Corresponding GPIO to 1 and the other to 0
           if(onOff == true)
             {
               GPIO_PinOutClear(SL_EMLIB_GPIO_INIT_RST_PORT, SL_EMLIB_GPIO_INIT_RST_PIN);
               GPIO_PinOutSet(SL_EMLIB_GPIO_INIT_SET_PORT, SL_EMLIB_GPIO_INIT_SET_PIN);
             }
           else
             {
               GPIO_PinOutSet(SL_EMLIB_GPIO_INIT_RST_PORT, SL_EMLIB_GPIO_INIT_RST_PIN);
               GPIO_PinOutClear(SL_EMLIB_GPIO_INIT_SET_PORT, SL_EMLIB_GPIO_INIT_SET_PIN);
             }
           sl_zigbee_app_debug_println("ONOFF SWITCHED");

           sl_zigbee_event_set_delay_ms(&relay_reset_event, RELAY_WAIT_PERIOD_MS);
       }
  }
  else if (event == &relay_reset_event) {
      //Set Both GPIOs to 0
      GPIO_PinOutClear(SL_EMLIB_GPIO_INIT_RST_PORT, SL_EMLIB_GPIO_INIT_RST_PIN);
      GPIO_PinOutClear(SL_EMLIB_GPIO_INIT_SET_PORT, SL_EMLIB_GPIO_INIT_SET_PIN);
      sl_zigbee_app_debug_println("ONOFF RESET");
  }
}
```

Los ficheros **button.h** y **button.c** se encargan de definir la interrupción a lanzar cuando se pulsa el botón. Una vez pulsado se lanza una señal al coordinador de la red de Toggle del clúster de ON/OFF.

```
/*
 * button.h
 *
 *      Author: Paris
 */

#ifndef BUTTON_H_
#define BUTTON_H_

void sl_button_on_change(const sl_button_t *handle);

#endif /* BUTTON_H_ */
```

button.c:

```
/*
 * button.c
 *
 *      Author: Paris
 */

#include "button.h"
#include "general.h"
#include "app/framework/include/af.h"
#include "sl_simple_button.h"
```

```
#include "sl_simple_button_instances.h"

void sl_button_on_change(const sl_button_t *handle)
{
#ifdef BUTTON
  if (&sl_button_btn0 == handle) {
    if ( sl_button_get_state(handle) == SL_SIMPLE_BUTTON_PRESSED) {
        emberAfFillCommandOnOffClusterToggle();
        emberAfGetCommandApsFrame()->sourceEndpoint = 1;
        EmberStatus status = emberAfSendCommandUnicastToBindings();
        sl_zigbee_app_debug_println("%s: 0x%02X", "Send to bindings", status);
    } else if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_RELEASED) {
        sl_zigbee_app_debug_println("BUTTON RELEASED");
    }
  }
#endif
}
```

Los ficheros **thermo.h** y **thermo.c** se encargan de la inicialización de la tarea de medir temperatura y humedad. Esta tarea se ejecuta cada 2 minutos y mide la temperatura y la humedad mediante el sensor DHT22. Estos ficheros utilizan la librería dht22lib para comunicarse mediante el protocolo OneWire con el sensor de temperatura.

```
/*
 * thermo.h
 *
 *      Author: Paris
 */

#ifndef THERMO_H_
#define THERMO_H_

#define THERMO_WAIT_PERIOD_MS    120000

void initThermo();

#endif /* THERMO_H_ */
```

thermo.c:

```
/*
 * thermo.c
 *
 *      Author: Paris
 */

#include "thermo.h"
#include "app/framework/include/af.h"
#include "dht22lib.h"

sl_zigbee_event_t thermo_event;
sl_status_t result;
int16_t thermo1 = 0, humid1 = 0;

int16_t thermo2 = 0;
uint16_t humid2 = 0;

void thermo_event_handler(sl_zigbee_event_t *event);

void initThermo()
{
  dht22_init();
  sl_zigbee_event_init(&thermo_event, thermo_event_handler);

  sl_zigbee_event_set_active(&thermo_event);
}

void thermo_event_handler(sl_zigbee_event_t *event)
{
  sl_zigbee_event_set_delay_ms(&thermo_event, THERMO_WAIT_PERIOD_MS);

  result = dht22_getRHTdata(&thermo1, &humid1);
  if (result == 0x00){
      thermo2 = thermo1 * 10;
```

```
      humid2 = humid1 * 10;
      emberAfWriteServerAttribute(1, ZCL_TEMP_MEASUREMENT_CLUSTER_ID,
ZCL_TEMP_MEASURED_VALUE_ATTRIBUTE_ID, (uint8_t *)&thermo2, 0x29);
      emberAfWriteServerAttribute(1, ZCL_RELATIVE_HUMIDITY_MEASUREMENT_CLUSTER_ID,
ZCL_RELATIVE_HUMIDITY_MEASURED_VALUE_ATTRIBUTE_ID, (uint8_t *)&humid2, 0x21);
      sl_zigbee_app_debug_println("RESULT: %d, THERMO: %d, HUMID: %d", result, thermo1,
humid1);
  }
}
```

Finalmente los ficheros **pir.h** y **pir.c** se encargan de inicializar y gestionar las interrupciones lanzadas por el sensor PIR. Al inicializar el sensor es importante esperarse 30s para la inicialización del sensor PIR. Una vez han transcurrido estos 30 segundos el sensor pone su salida a tierra cada vez que detecta la presencia o el movimiento de algún ser humano en su campo de visión y se envía una señal a la red Zigbee indicando la presencia de dicho ser humano. De forma automática tras transcurrir un tiempo definido por el usuario "*unoccupied_wait_period_ms*" se envía una señal a la red Zigbee indicando la ausencia de ningún ser humano en su campo de visión.

```
/*
 * pir.h
 *
 *      Author: Paris
 */

#ifndef PIR_H_
#define PIR_H_

#define PIR_ENDPOINT 1

void initPIR();
void updatePIRPeriodMS();

#endif /* PIR_H_ */
```

pir.c:

```
/*
 * pir.c
 *
 *      Author: Paris
 */

#include "pir.h"
#include "app/framework/include/af.h"
#include "gpiointerrupt.h"
#include "sl_emlib_gpio_init_PIR_config.h"

#define INT_CALLBACK 7
#define PIR_WAIT_PERIOD_MS    30000

sl_zigbee_event_t pir_event;
sl_zigbee_event_t unoccupied_event;
uint32_t occupied_to_unoccupied_wait_period_ms = PIR_WAIT_PERIOD_MS;
uint8_t true_val = 1;
uint8_t false_val = 0;

void pir_event_handler(sl_zigbee_event_t *event);
void unoccupied_event_handler(sl_zigbee_event_t *event);
void pir_interrupt(uint8_t intNo, void* ctx);

void initPIR()
{
  sl_zigbee_event_init(&pir_event, pir_event_handler);
  sl_zigbee_event_init(&unoccupied_event, unoccupied_event_handler);
  sl_zigbee_event_set_delay_ms(&pir_event, PIR_WAIT_PERIOD_MS);
}

void updatePIRPeriodMS()
{
  uint16_t occupied_to_unoccupied_wait_period;
  if (emberAfReadServerAttribute(PIR_ENDPOINT,
```

```
                                ZCL_OCCUPANCY_SENSING_CLUSTER_ID,
                                ZCL_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY_ATTRIBUTE_ID,
                                (uint8_t *)&occupied_to_unoccupied_wait_period,
                                sizeof(occupied_to_unoccupied_wait_period))
     == EMBER_ZCL_STATUS_SUCCESS) {
      occupied_to_unoccupied_wait_period_ms = (uint32_t)occupied_to_unoccupied_wait_period *
1000;
  }
}

void pir_event_handler(sl_zigbee_event_t *event)
{
  updatePIRPeriodMS();
  unsigned int intPin = GPIOINT_CallbackRegisterExt(INT_CALLBACK, pir_interrupt, NULL);
  GPIO_IntClear(1 << intPin);
  GPIO_ExtIntConfig(SL_EMLIB_GPIO_INIT_PIR_PORT, SL_EMLIB_GPIO_INIT_PIR_PIN, intPin, true,
false, true);
  sl_zigbee_app_debug_println("PIR ENABLED");
}

void unoccupied_event_handler(sl_zigbee_event_t *event)
{
  emberAfWriteServerAttribute(PIR_ENDPOINT, ZCL_OCCUPANCY_SENSING_CLUSTER_ID,
ZCL_OCCUPANCY_ATTRIBUTE_ID, (uint8_t *)&false_val, 0x18);
}

void pir_interrupt(uint8_t intNo, void* ctx)
{
  uint8_t occupancy;
  if (emberAfReadServerAttribute(PIR_ENDPOINT,
                                ZCL_OCCUPANCY_SENSING_CLUSTER_ID,
                                ZCL_OCCUPANCY_ATTRIBUTE_ID,
                                (uint8_t *)&occupancy,
                                sizeof(occupancy))
     == EMBER_ZCL_STATUS_SUCCESS) {
      if(!(occupancy & 0x01)) {
          emberAfWriteServerAttribute(PIR_ENDPOINT, ZCL_OCCUPANCY_SENSING_CLUSTER_ID,
ZCL_OCCUPANCY_ATTRIBUTE_ID, (uint8_t *)&true_val, 0x18);
      }
      sl_zigbee_event_set_delay_ms(&unoccupied_event, occupied_to_unoccupied_wait_period_ms);
  }
}
```

ANEXO II: Código fuente proyecto TFG_Leaf

A continuación se observará el código fuente del proyecto responsable de gestionar los nodos repetidores de la red Zigbee.

El fichero **main.c** se actúa como fichero de entrada del programa y se encarga de inicializar y lanzar la app de Zigbee. Este dispositivo al ser de tipo Sleepy End Device pasará a modo de Sleep cuando no haya recibido ninguna interrupción por hardware o por software, haciendo que reduzca su consumo energético.

```
/***************************************************************************//**
 * @file main.c
 * @brief main() function.
 ******************************************************************************
 * # License
 * <b>Copyright 2021 Silicon Laboratories Inc. www.silabs.com</b>
 ******************************************************************************
 *
 * The licensor of this software is Silicon Laboratories Inc. Your use of this
 * software is governed by the terms of Silicon Labs Master Software License
 * Agreement (MSLA) available at
 * www.silabs.com/about-us/legal/master-software-license-agreement. This
 * software is distributed to you in Source Code format and is governed by the
 * sections of the MSLA applicable to Source Code.
 *
 ******************************************************************************/

#include "sl_component_catalog.h"
#include "sl_system_init.h"
#include "FreeRTOS.h"
#include "FreeRTOSConfig.h"
#include "task.h"
#include "sl_led.h"
#include "sl_simple_led_instances.h"
#include "app/framework/include/af.h"
//#undef SL_CATALOG_KERNEL_PRESENT
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
#include "sl_power_manager.h"
#endif
#if defined(SL_CATALOG_KERNEL_PRESENT)
#include "sl_system_kernel.h"
#else
#include "sl_system_process_action.h"
#endif // SL_CATALOG_KERNEL_PRESENT

#ifdef EMBER_TEST
#define main nodeMain
#endif

int main(void)
{
  // Initialize Silicon Labs device, system, service(s) and protocol stack(s).
  // Note that if the kernel is present, processing task(s) will be created by
  // this call.
  sl_system_init();

#if defined(SL_CATALOG_KERNEL_PRESENT)
  // Start the kernel.
  sl_system_kernel_start();
#else // SL_CATALOG_KERNEL_PRESENT
  while (1) {
    // Do not remove this call: Silicon Labs components process action routine
    // must be called from the super loop.
    sl_system_process_action();

    // Let the CPU go to sleep if the system allow it.
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
    sl_power_manager_sleep();
#endif // SL_CATALOG_POWER_MANAGER_PRESENT
  }
#endif // SL_CATALOG_KERNEL_PRESENT

  return 0;
}
```

El fichero **app.c** se encarga de gestionar las llamadas a las diversas otras clases, ya sea cuando se inicializa el programa o cuando recibe un cambio a algún atributo. Adicionalmente se encarga también de inicializar la conexión a la red Zigbee utilizando las librerías network-steering y zll-commissioning.

```
/***************************************************************************//**
 * @file app.c
 * @brief Callbacks implementation and application specific code.
 *******************************************************************************
 * # License
 * <b>Copyright 2021 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * The licensor of this software is Silicon Laboratories Inc. Your use of this
 * software is governed by the terms of Silicon Labs Master Software License
 * Agreement (MSLA) available at
 * www.silabs.com/about-us/legal/master-software-license-agreement. This
 * software is distributed to you in Source Code format and is governed by the
 * sections of the MSLA applicable to Source Code.
 *
 ******************************************************************************/

#include "app/framework/include/af.h"
#include "zll-commissioning.h"
#include "network-steering.h"
#include "find-and-bind-target.h"
#include "general.h"
#include "identify.h"
#include "button.h"
#include "thermo.h"

#define MAIN_ENDPOINT           1

static sl_zigbee_event_t commissioning_event;
static sl_zigbee_event_t finding_and_binding_event;

static void commissioning_event_handler(sl_zigbee_event_t *event)
{
  if (emberAfNetworkState() != EMBER_JOINED_NETWORK) {
    EmberStatus status = emberAfPluginNetworkSteeringStart();
    sl_zigbee_app_debug_println("%s network %s: 0x%02X", "Join", "start", status);
  }
}

static void finding_and_binding_event_handler(sl_zigbee_event_t *event)
{
  if (emberAfNetworkState() == EMBER_JOINED_NETWORK) {
    sl_zigbee_event_set_inactive(&finding_and_binding_event);

    sl_zigbee_app_debug_println("Find and bind target start: 0x%02X",
                                emberAfPluginFindAndBindTargetStart(MAIN_ENDPOINT));
  }
}

/** @brief Stack Status
 *
 * This function is called by the application framework from the stack status
 * handler.  This callbacks provides applications an opportunity to be notified
 * of changes to the stack status and take appropriate action. The framework
 * will always process the stack status after the callback returns.
 */
void emberAfStackStatusCallback(EmberStatus status)
{
  // Note, the ZLL state is automatically updated by the stack and the plugin.
  if (status == EMBER_NETWORK_DOWN) {
    sl_zigbee_app_debug_println("NETWORK DOWN");
  } else if (status == EMBER_NETWORK_UP) {
    sl_zigbee_app_debug_println("NETWORK UP");
    sl_zigbee_event_set_active(&finding_and_binding_event);
  }
}

/** @brief Init
```

```
 * Application init function
 */
void emberAfMainInitCallback(void)
{
  sl_zigbee_app_debug_println("STARTING...");
  sl_zigbee_event_init(&commissioning_event, commissioning_event_handler);
  sl_zigbee_event_init(&finding_and_binding_event, finding_and_binding_event_handler);

#ifdef THERMO
  initThermo();
#endif


  sl_zigbee_event_set_active(&commissioning_event);
}

/** @brief Complete network steering.
 *
 * This callback is fired when the Network Steering plugin is complete.
 *
 * @param status On success this will be set to EMBER_SUCCESS to indicate a
 * network was joined successfully. On failure this will be the status code of
 * the last join or scan attempt. Ver.: always
 *
 * @param totalBeacons The total number of 802.15.4 beacons that were heard,
 * including beacons from different devices with the same PAN ID. Ver.: always
 * @param joinAttempts The number of join attempts that were made to get onto
 * an open Zigbee network. Ver.: always
 *
 * @param finalState The finishing state of the network steering process. From
 * this, one is able to tell on which channel mask and with which key the
 * process was complete. Ver.: always
 */
void emberAfPluginNetworkSteeringCompleteCallback(EmberStatus status,
                                                  uint8_t totalBeacons,
                                                  uint8_t joinAttempts,
                                                  uint8_t finalState)
{
  sl_zigbee_app_debug_println("%s network %s: 0x%02X", "Join", "complete", status);
}

/** @brief
 *
 * Application framework equivalent of ::emberRadioNeedsCalibratingHandler
 */
void emberAfRadioNeedsCalibratingCallback(void)
{
  sl_mac_calibrate_current_channel();
}

/** @brief Post Attribute Change
 *
 * This function is called by the application framework after it changes an
 * attribute value. The value passed into this callback is the value to which
 * the attribute was set by the framework.
 */
void emberAfPostAttributeChangeCallback(uint8_t endpoint,
                                        EmberAfClusterId clusterId,
                                        EmberAfAttributeId attributeId,
                                        uint8_t mask,
                                        uint16_t manufacturerCode,
                                        uint8_t type,
                                        uint8_t size,
                                        uint8_t* value)
{

#ifdef LED
  if (endpoint == IDENTIFY_ENDPOINT
      && clusterId == ZCL_IDENTIFY_CLUSTER_ID
      && attributeId == ZCL_IDENTIFY_TIME_ATTRIBUTE_ID
      && mask == CLUSTER_MASK_SERVER)
    identifyAttributechanged();
#endif
}
```

El fichero **general.h** se encarga de definir los periféricos conectados en el prototipo, es decir se definirán únicamente las macros de los periféricos que deseemos inicializar y utilizar.

```
/*
 * general.h
 *
 *  Created on: Jul 25, 2023
 *      Author: paris
 */

#ifndef GENERAL_H_
#define GENERAL_H_

#define BUTTON
#define LED
#define THERMO


#endif /* GENERAL_H_ */
```

Los ficheros **identify.h** e **identify.c** reciben llamadas directamente desde el fichero app.c y se encargan de inicializar y gestionar el parpadeo del diodo LED cuando se desee identificar el dispositivo.

```
/*
 * identify.h
 *
 *      Author: Paris
 */

#ifndef IDENTIFY_H_
#define IDENTIFY_H_

#define IDENTIFY_ENDPOINT           1
#define IDENTIFY_BLINK_S        2

void initIdentify();
void identifyAttributechanged();

#endif /* IDENTIFY_H_ */
```

identify.c:

```
/*
 * identify.c
 *
 *      Author: Paris
 */

#include "identify.h"
#include "app/framework/include/af.h"

void initIdentify()
{}

void identifyAttributechanged()
{
  uint16_t identifyTime;
  emberAfReadServerAttribute(IDENTIFY_ENDPOINT,
                             ZCL_IDENTIFY_CLUSTER_ID,
                             ZCL_IDENTIFY_TIME_ATTRIBUTE_ID,
                             (uint8_t *)&identifyTime,
                             sizeof(identifyTime));
  if (identifyTime <= 0)
    sl_led_turn_off(&sl_led_led0);
  else if (identifyTime % IDENTIFY_BLINK_S == 0) {
    sl_zigbee_app_debug_println("IDENTIFYING");
    sl_led_toggle(&sl_led_led0);
  }
}
```

Los ficheros **button.h** y **button.c** se encargan de definir la interrupción a lanzar cuando se pulsa el botón. Una vez pulsado se lanza una señal al coordinador de la red de Toggle del clúster de ON/OFF.

```
/*
 * button.h
 *
 *      Author: Paris
 */

#ifndef BUTTON_H_
#define BUTTON_H_

void sl_button_on_change(const sl_button_t *handle);

#endif /* BUTTON_H_ */
```

button.c:

```
/*
 * button.c
 *
 *      Author: Paris
 */

#include "button.h"
#include "general.h"
#include "app/framework/include/af.h"
#include "sl_simple_button.h"
#include "sl_simple_button_instances.h"

void sl_button_on_change(const sl_button_t *handle)
{
#ifdef BUTTON
  if (&sl_button_btn0 == handle) {
    if ( sl_button_get_state(handle) == SL_SIMPLE_BUTTON_PRESSED) {
        emberAfFillCommandOnOffClusterToggle();
        EmberStatus status = emberAfSendCommandUnicastToBindings();
        sl_zigbee_app_debug_println("%s: 0x%02X", "Send to bindings", status);
    } else if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_RELEASED) {
        sl_zigbee_app_debug_println("BUTTON RELEASED");
    }
  }
#endif
}
```

Los ficheros **thermo.h** y **thermo.c** se encargan de la inicialización de la tarea de medir temperatura y humedad. Esta tarea se ejecuta cada 2 minutos y mide la temperatura y la humedad mediante el sensor DHT22. Estos ficheros utilizan la librería dht22lib para comunicarse mediante el protocolo OneWire con el sensor de temperatura.

```
/*
 * thermo.h
 *
 *      Author: Paris
 */

#ifndef THERMO_H_
#define THERMO_H_

#define THERMO_WAIT_PERIOD_MS    120000

void initThermo();

#endif /* THERMO_H_ */
```

thermo.c:

```
/*
 * thermo.c
 *
 *      Author: Paris
 */
```

```
#include "thermo.h"
#include "app/framework/include/af.h"
#include "dht22lib.h"

sl_zigbee_event_t thermo_event;
sl_status_t result;
int16_t thermo1 = 0, humid1 = 0;

int16_t thermo2 = 0;
uint16_t humid2 = 0;

void thermo_event_handler(sl_zigbee_event_t *event);

void initThermo()
{
  dht22_init();
  sl_zigbee_event_init(&thermo_event, thermo_event_handler);

  sl_zigbee_event_set_active(&thermo_event);
}

void thermo_event_handler(sl_zigbee_event_t *event)
{
  sl_zigbee_event_set_delay_ms(&thermo_event, THERMO_WAIT_PERIOD_MS);

  result = dht22_getRHTdata(&thermo1, &humid1);
  if (result == 0x00){
      thermo2 = thermo1 * 10;
      humid2 = humid1 * 10;
      emberAfWriteServerAttribute(1, ZCL_TEMP_MEASUREMENT_CLUSTER_ID,
ZCL_TEMP_MEASURED_VALUE_ATTRIBUTE_ID, (uint8_t *)&thermo2, 0x29);
      emberAfWriteServerAttribute(1, ZCL_RELATIVE_HUMIDITY_MEASUREMENT_CLUSTER_ID,
ZCL_RELATIVE_HUMIDITY_MEASURED_VALUE_ATTRIBUTE_ID, (uint8_t *)&humid2, 0x21);
      sl_zigbee_app_debug_println("RESULT: %d, THERMO: %d, HUMID: %d", result, thermo1,
humid1);
  }
}
```

ANEXO III: Archivos de configuración Zigbee2MQTT

Para añadir un dispositivo a zigbee2mqtt es necesario que exista un archivo de configuración o converter indicando los clusters y los atributos de cada dispositivo. Los archivos diseñados por terceros suelen tener ya creados sus converters y no es necesario ni buscarlos ni descargarlos. Sin embargo para nuestros dispositivos hemos de gestionar la configuración de los converters para los dispositivos que hemos creado.

Para configurar nuestro dispositivo hemos de crear un objeto "**definition**" con los datos de los dispositivos. El valor **zigbeeModel** es una lista de los modelos que compartirán este archivo de configuración, el modelo es un nombre dado por el fabricante del dispositivo Zigbee. Los valores de **model**, **vendor** y **description** son los nombres de modelo, fabricante y descripción que se visualizará en la pantalla de zigbee2mqtt. En los campos de **fromZigbee** y **toZigbee** indicaremos los atributos o comandos de los clusters que pueden enviar información desde Zigbee o a Zigbee respectivamente. El campo **exposes** indica los atributos que queremos visualizar directamente desde zigbee2mqtt; en nuestro caso, al no visualizar datos directamente desde zigbee2mqtt, no es necesario añadir ningún valor en este campo. Finalmente, en el campo **configure** se configurarán los endpoints y los tiempos de reporte de cada atributo de los clusters que busquemos reportar de forma automática.

A continuación, se muestran los archivos de configuración de los prototipos que han sido diseñados.

TFG_Relay.js:

```
const fz = require("zigbee-herdsman-converters/converters/fromZigbee");
const tz = require("zigbee-herdsman-converters/converters/toZigbee");
const exposes = require("zigbee-herdsman-converters/lib/exposes");
const reporting = require("zigbee-herdsman-converters/lib/reporting");
const extend = require("zigbee-herdsman-converters/lib/extend");
const e = exposes.presets;
const ea = exposes.access;
const definition = {
  zigbeeModel: ["TFG_Relay"], // The model ID from: Device with modelID 'lumi.sens' is not supported.
  model: "TFG_Relay", // Vendor model number, look on the device for a model number
  icon: "/icons/TFG.jpg",
  vendor: "Paris", // Vendor of the device (only used for documentation and startup logging)
  description: "", // Description of the device, copy from vendor site. (only used for documentation and startup logging)
  fromZigbee: [fz.on_off, fz.temperature, fz.humidity, fz.occupancy], // We will add this later
  toZigbee: [
    tz.factory_reset,
    tz.write,
    tz.identify,
    tz.on_off,
    tz.occupancy_timeout,
  ], // Should be empty, unless device can be controlled (e.g. lights, switches).
  configure: async(device, coordinatorEndpoint, logger) => {
    const endpoint1 = device.getEndpoint(1);
    const endpoint2 = device.getEndpoint(2);
    await reporting.onOff(endpoint2, {
      min: 0,
      max: 120,
      change: 1,
    });
    await reporting.temperature(endpoint1, {
      min: 0,
      max: 0,
      change: 1,
```

```
    });
    await reporting.humidity(endpoint1, {
      min: 0,
      max: 0,
      change: 1,
    });
    await reporting.occupancy(endpoint1, {
      min: 0,
      max: 120,
      change: 1,
    });
  },
  exposes: [e.temperature(), e.humidity(), e.switch(), e.occupancy()], // Defines
what this device exposes, used for e.g. Home Assistant discovery and in the
frontend
};
module.exports = definition;
```

TFG_Leaf.js:

```
const fz = require("zigbee-herdsman-converters/converters/fromZigbee");
const tz = require("zigbee-herdsman-converters/converters/toZigbee");
const exposes = require("zigbee-herdsman-converters/lib/exposes");
const reporting = require("zigbee-herdsman-converters/lib/reporting");
const extend = require("zigbee-herdsman-converters/lib/extend");
const e = exposes.presets;
const ea = exposes.access;
const definition = {
  zigbeeModel: ["TFG_Leaf"], // The model ID from: Device with modelID 'lumi.sens'
is not supported.
  model: "TFG_Leaf", // Vendor model number, look on the device for a model number
  icon: "/icons/TFG.jpg",
  vendor: "Paris", // Vendor of the device (only used for documentation and startup
logging)
  description: "", // Description of the device, copy from vendor site. (only used
for documentation and startup logging)
  fromZigbee: [fz.temperature, fz.humidity], // We will add this later
  toZigbee: [], // Should be empty, unless device can be controlled (e.g. lights,
switches).
  configure: async(device, coordinatorEndpoint, logger) => {
    const endpoint1 = device.getEndpoint(1);
    await reporting.temperature(endpoint1, {
      min: 0,
      max: 0,
      change: 1,
    });
    await reporting.humidity(endpoint1, {
      min: 0,
      max: 0,
      change: 1,
    });
  },
  exposes: [e.temperature(), e.humidity()], // Defines what this device exposes,
used for e.g. Home Assistant discovery and in the frontend
};
module.exports = definition;
```