UNIVERSIDAD POLITÉCNICA
DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Estudio de técnicas Over-The-Air en la Internet de las Cosas

TRABAJO FIN DE GRADO

Grado en Ingeniería Telemática

Autor: Javier Cano Asís
Directora: María Dolores Cano Baños

Cartagena, 10 de octubre de 2020

Agradecimientos

Este trabajo no habría sido posible sin el apoyo de mi directora, María Dolores Cano Baños, quien me ha dado la oportunidad de realizar este proyecto y me ha ofrecido su ayuda constante en estos tiempos tan difíciles.
También me gustaría agradecerles a Celia y a mis amigos todo su apoyo durante todos estos años, pues ellos me han acompañado durante este camino y han sido mi apoyo fuera de las clases.
No puedo terminar sin agradecer a mi familia y, en especial a mis padres, pues con ellos he compartido todo, los momentos felices y amargos, siendo un apoyo incondicional durante esta etapa académica que hoy culmina.

Si consigo ver más lejos
es porque he conseguido auparme
a hombros de gigantes

Isaac Newton.

Índice general

1. Introducción **1**
1.1. Descripción del proyecto 1
1.2. Objetivos y resumen del proyecto 1
1.3. Planificación temporal 2

2. Internet de las cosas **3**
2.1. Introducción a IoT 3
2.2. Aplicaciones 4
2.3. Futuro y barreras 5

3. Técnicas Over-The-Air **7**
3.1. Over-The-Air 7
3.1.1. Seguridad en OTA 8
3.2. Plataformas IoT 9

4. Dispositivos comerciales de IoT **11**
4.1. Kits de prototipado hardware para IoT 11
4.1.1. Análisis comparativo 11
4.1.2. Conclusiones 15
4.2. Sistemas operativos en IoT 15
4.2.1. Análisis comparativo 15
4.3. Preparación de las pruebas 17

5. Realización de pruebas **19**
5.1. Desarrollo de un firmware con capacidad de actualización OTA 19
5.1.1. Desarrollo del programa 19
5.1.2. Descarga y configuración de herramientas de desarrollo 21
5.1.3. Descarga y configuración de ESP-IDF 21
5.1.4. Componentes y conexiones necesarias 25
5.1.5. Prueba del programa 25
5.2. Prueba OTA comercial 29
5.2.1. Configuración inicial 30
5.2.2. Prueba de los servicios OTA 31

6. Conclusiones **35**

Bibliografía **37**

Lista de Acrónimos y Abreviaturas **39**

A. Código fuente **41**
A.1. main.c 41
A.2. wifi.c 45
A.3. wifi.h 47
A.4. Makefile 47
A.5. component.mk 47
A.6. github_cert.pem 48
A.7. update_info.json 48

Índice de figuras

1.1. Planificación temporal 2

2.1. Evolución del IoT 3
2.2. Aplicaciones del IoT 5

3.1. Actualización 'Over the air' 7
3.2. Plataformas IoT 9

4.1. Comparativa de las opciones de conectividad 12

5.1. Menú principal de configuración ESP-IDF 22
5.2. Configuración SDK tool 23
5.3. Configuración tabla de particiones de memoria 23
5.4. Selección partición de memoria para imagen OTA 24
5.5. Esquema de conexiones 25
5.6. Ejecución de la versión 0.1 del firmware desarrollado para ESP32 27
5.7. Actualización del firmware de la placa a la versión 0.2 28
5.8. Ejecución de la versión 0.2 del firmware desarrollado tras actualización OTA . 29
5.9. Se añade el dispositivo a la plataforma 30
5.10. Lista de dispositivos añadidos a la plataforma 31
5.11. Compilación y generación del archivo binario con la versión 1 31
5.12. Compilación y generación del archivo binario con la versión 2 32
5.13. Carga de binario a la plataforma 32
5.14. Repositorio de firmwares de la plataforma 33
5.15. Selección de grupos para actualización de firmware 33
5.16. Confirmación de grupos para actualización de firmware 34
5.17. Panel de información de eventos para el dispositivo Particle Argon 34

Índice de tablas

4.1. Tabla comparativa del análisis de plataformas hardware 14
4.2. Tabla comparativa del análisis de sistemas operativos en IoT 16
4.3. Compatibilidades entre arquitecturas de microcontroladores y sistemas operativos 16

5.1. Estructura de la memoria preparada para recibir actualizaciones OTA 24

Índice de Códigos

5.1. Instalación paquetes necesarios 21
5.2. Instalación de herramientas para compilar 21
5.3. Instalación API para ESP32 21
5.4. Configuración ESP-IDF 22
5.5. Compilación del proyecto y generación del binario v0.2 26
5.6. Borrado de memoria 26

A.1. main.c 41
A.2. wifi.c 45
A.3. wifi.h 47
A.4. Makefile 47
A.5. component.mk 47
A.6. github_cert.pem 48
A.7. update_info.json 48

1. Introducción

1.1. Descripción del proyecto

Las soluciones basadas en la Internet of Things (IoT) están ayudando a la digitalización de una gran cantidad de funcionalidades en áreas de aplicación como la atención sanitaria, la vigilancia, la agricultura o la automatización del hogar y la industria. Esta tendencia hará que se incremente notablemente el número de dispositivos, por lo que es necesario que las soluciones IoT estén bien diseñadas y sean escalables. Sin embargo, las limitaciones específicas de los dispositivos, la rápida evolución de la tecnología y el creciente ritmo de implantación de nuevos dispositivos en zonas de difícil acceso plantean cuestiones relativas a la sostenibilidad a largo plazo de las redes IoT desplegadas. Por ejemplo, a menudo se detectan problemas o errores de seguridad después del despliegue, lo que dificulta el funcionamiento de estos despliegues. La programación Over-The-Air (OTA) permite el uso de técnicas inalámbricas para la configuración y actualización de dispositivos.

1.2. Objetivos y resumen del proyecto

El objetivo de este TFE es estudiar de forma comparativa diferentes técnicas Over-The-Air sobre varios dispositivos IoT y realizar con ellos pruebas experimentales. Para ello, en el Capítulo 2 del proyecto se profundiza en qué es el IoT, en el Capítulo 3 se introduce qué son las técnicas Over-The-Air, en el Capítulo 4 se analizan diferentes plataformas hardware comerciales y sistemas operativos sobre los que poder desarrollar un producto IoT para así, tener un producto funcional sobre el que, en el Capítulo 5 poder realizar pruebas experimentales de distintas técnicas OTA, compararlas, analizar los resultados y elaborar conclusiones.

1.3. Planificación temporal

Para una mejor organización del proyecto, se ha estimado la siguiente planificación temporal reflejada en el diagrama de Gantt acorde a los temas tratados y al nivel de detalle que se espera adquirir:

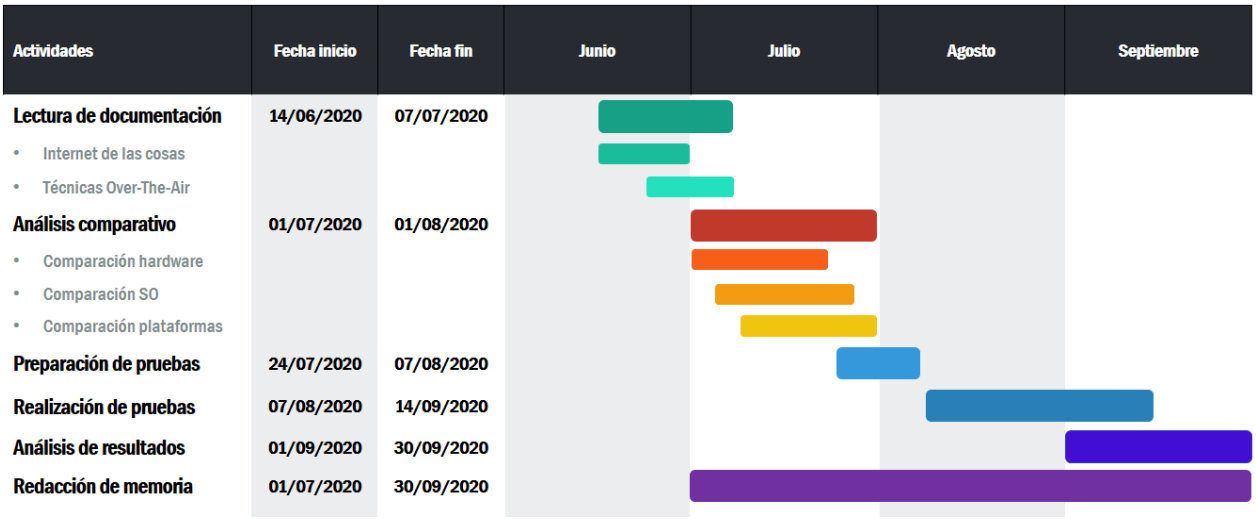

Figura 1.1: Planificación temporal

2. Internet de las cosas

2.1. Introducción a IoT

El Internet of Things (IoT) (en castellano, internet de las cosas) es un concepto que consiste en la interconexión de dispositivos a través de internet. Gracias a su pequeño tamaño y a su bajo coste, cada vez más sensores son integrados en todo tipo de dispositivos, siendo cualquier objeto susceptible de estar conectado a internet y de esta manera poder recopilar información e interactuar con otros elementos de la red.

El concepto del internet de las cosas surgió en 1999 con el término de "machine to machine" (máquina a máquina) y desde entonces, ha evolucionado tanto que el número de dispositivos conectados a internet ha experimentado un crecimiento exponencial, teniendo a día de hoy diez veces más objetos conectados a internet que usuarios.

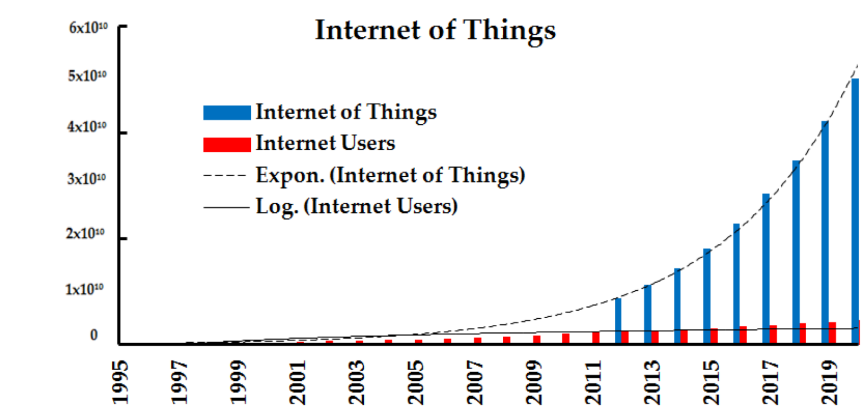

Figura 2.1: Evolución del IoT. Fuente:`https://www.mdpi.com/2079-8954/5/1/24`

Su éxito radica en su amplia utilidad y la facilidad que existe hoy en día para realizar un producto IoT, gracias a la gran oferta de kits de desarrollo y prototipado así como acceso a la información y proyectos ya desarrollados.

2.2. Aplicaciones

Con el crecimiento del IoT cada vez son más las utilidades y funcionalidades que nos aporta esta tecnología, tanto en el ámbito particular como en el empresarial. Entre ellas cabe destacar las siguientes categorías:

- **Domótica.** Desde hace unos años, la idea de hogares inteligentes está cada vez más presente en nuestras vidas. El poder controlar algunos electrodomésticos como luces o televisión con la voz o realizar ciertas automatizaciones como encender una luz cuando sea una determinada hora son algunas de las funcionalidades del IoT aplicado a la domótica.

- **Weareables.** Cada vez es más común el uso de werarebles (también conocidos como accesorios inteligentes), pequeños dispositivos energéticamente eficientes equipados con sensores, capaces de monitorizar, realizar mediciones y lecturas para mostrar datos de interés al usuario. Ejemplos de wearebles son las gafas de realidad virtual, smartwatches o pulseras inteligentes.

- **Salud.** Con el desarrollo de los weareables surgió una rama del IoT aplicada a la salud. Con estos dispositivos y otros sensores conectados a los pacientes, los médicos son capaces de medir constantes y tener un seguimiento en tiempo real sobre datos del paciente que podrían ser críticos, recibiendo incluso alertas si alguna de las constantes vitales sobrepasa cierto valor, siendo muy útil para prevenir eventos letales en pacientes de alto riesgo. Un ejemplo sería el uso de camas hospitalarias inteligentes que permitan el seguimiento de constantes vitales concretas como la temperatura, presión sanguínea o saturación de oxígeno en sangre.

- **Agricultura.** La agricultura es una actividad económica fundamental en la que se utilizan grandes extensiones de terreno para realizar cultivos. A menudo, el controlar estas extensiones de terreno requiere de muchos recursos humanos y tiempo, lo que hace que la agricultura se más ineficiente. Gracias al IoT, se pueden monitorizar parámetros de interés para obtener información sobre los cultivos y el suelo, información como humedad, acidez, temperatura, nivel de nutrientes, etc., que permite a los agricultores el hacer un uso más eficiente del agua o saber qué momento es mejor para iniciar la siembra/recogida de los frutos.

- **Smart cities.** El internet de las cosas nos posibilita recopilar datos sobre como se mueve una ciudad, para así establecer métodos eficientes para la gestión de los recursos. Una smart city permite optimizar la iluminación, transporte público, recogida de residuos, etc., lo cual reduce el gasto público y facilita la vida de las personas que viven en ella. Un ejemplo sería la monitorización de los contenedores de basura, lo que permite saber cuando está lleno o vacío para así evitar que el camión de recogida de residuos pase por ese punto, optimizando la ruta, ahorrando tiempo y combustible.

Figura 2.2: Aplicaciones del IoT. Fuente: `https://www.cistec.es/las-mil-y-una-aplicaciones-del-internet-de-las-cosas-iot/`

2.3. Futuro y barreras

El IoT es un campo que promete seguir desarrollándose y creciendo, pero se plantean unos desafíos que hay que abordar:

- **Seguridad y privacidad**. En los últimos años se han desarrollado e implementado complejos algoritmos de cifrado que garantizan confidencialidad, integridad y autenticación en las comunicaciones. En el IoT a menudo se maneja información sensible, lo que obliga a utilizar alguna de estas técnicas de cifrado si queremos asegurar la comunicación. El problema es que, debido a la complejidad de estos algoritmos, para su ejecución a menudo es necesario un hardware específico, el cual incrementa el tamaño y hace que consuma más energía, haciendo el producto menos eficiente energéticamente y, en el caso de que funcione a batería, haya que cargarlo con más frecuencia.
- **Interoperabilidad**. Ante el rápido crecimiento y desarrollo y sin una estandarización, resulta muy difícil que con la gran cantidad de fabricantes que hay en el mercado usando tecnologías heterogéneas, sea posible la interconexión de todos los dispositivos.
- **Falta de infraestructura**. Las redes de comunicaciones (en especial, internet) no están preparadas para gestionar los miles de millones de dispositivos que se espera que estén conectados en los próximos años. Ante este crecimiento exponencial se ha de mejorar la infraestructura de comunicaciones que tenemos hoy en día para que la idea de IoT a gran escala sea viable.

3. Técnicas Over-The-Air

3.1. Over-The-Air

Al desarrollar un proyecto IoT es habitual involucrar decenas o incluso cientos de dispositivos, los cuales son ubicados en sitios poco accesibles y, en ocasiones, muy distanciados entre ellos. El software que están ejecutando a menudo ha de actualizarse ya sea para incorporar un parche de seguridad, implementar una nueva funcionalidad, solucionar un bug o cambiar alguna configuración. Ante este escenario es inviable desplazarse y acceder físicamente al microcontrolador de cada placa para así reprogramarlo y meterle una nueva versión de software. Es aquí donde las actualizaciones vía OTA juegan un papel fundamental.

La programación OTA es el método a partir del cual un dispositivo puede actualizar su software sin estar conectado a ningún medio físico, usando una tecnología inalámbrica ('over the air') como puede ser WiFi o Bluetooth. En el campo del IoT, los nodos están suscritos a un canal y, ante una nueva actualización disponible estos son notificados y cuando se den las condiciones necesarias comenzará el proceso. Una de las grandes ventajas que esto nos aporta es que se puede hacer de manera remota y se puede configurar la actualización para elegir cuales de los dispositivos conectados a la red son actualizados. Se requiere que el dispositivo tenga un software y hardware que soporten esta característica pues no todos lo tienen. Habitualmente, al distribuir el firmware a los dispositivos se suele usar algún mecanismo de seguridad (por ejemplo, uso de certificados) para asegurar la autenticidad e integridad.

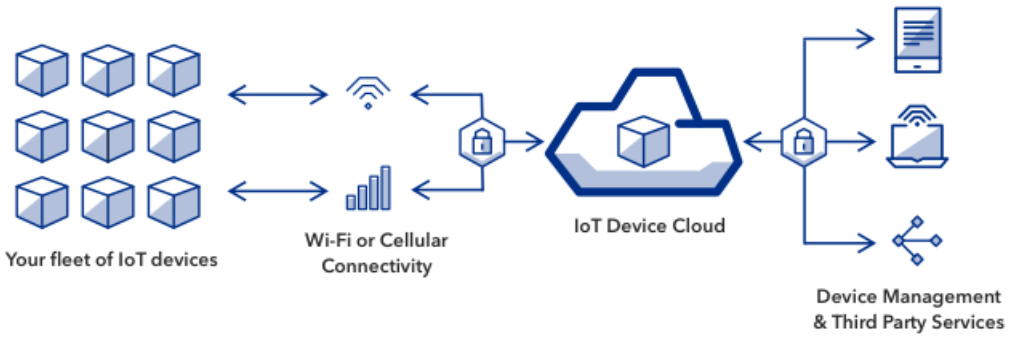

Figura 3.1: Actualización 'Over the air'. Fuente: `https://dzone.com/articles/how-to-choose-the-right-iot-cloud-platform`

3.1.1. Seguridad en OTA

El proceso de descargar un nuevo software por medio de una técnica OTA puede presentar una serie de riesgos. A menudo, la información tratada por esta serie de dispositivos suele ser confidencial y delicada, por lo que es imprescindible dotar a estas técnicas inalámbricas de métodos de seguridad para evitar fugas de información. A la hora de una actualización, el firmware descargado podría estar corrupto lo que dejaría al dispositivo inoperativo, alguien podría suplantar la identidad del servidor de confianza que almacena la actualización y de esta manera, hacerse con el control del dispositivo o incluso, tras el correcto almacenamiento de la imagen, cuando se reiniciara el dispositivo IoT, un corte de corriente podría dejarlo sin la configuración de red necesaria y ser irrecuperable. Por esto, a la hora de llevar a cabo una actualización, se suelen llevar a cabo unas medidas de seguridad que reducen esta serie de riesgos:

- **Realizar actualizaciones OTA gradualmente.** Cuando los proyectos con productos IoT involucran gran cantidad de dispositivos, es importante realizar la actualización del firmware gradualmente, de tal modo que el software puede ser probado por unos pocos para evitar grandes pérdidas en caso de que algo saliera mal y también evitar que el intento de acceso de todos los dispositivos al servidor de manera simultánea lo sobrecargara.

- **Descargar las actualizaciones con seguridad.** Una vez que el dispositivo es notificado de que hay una actualización disponible, tendrá que descargarla. Lo ideal sería que la nueva imagen se encontrara en un servidor dedicado accesible localmente, pero por desgracia la mayor parte de dispositivos necesitan el acceso a internet. Ante este escenario, es necesario establecer conexiones que soporten protocolos de seguridad como TLS para asegurar la conexión.

- **Autenticar la imagen OTA.** Conectarte al servidor correctamente no te garantiza que vayas a recibir la imagen correcta, ya que el archivo podría ser modificado o alterado. Un dispositivo debe ser capaz de autenticar que la imagen descargada es la misma que ha mandado el servidor. Para esto, la imagen descargada suele ser firmada por el servidor y se le realiza una función hash, de esta manera el cliente puede confirmar la integridad.

- **Recuperar el firmware si el dispositivo falla al arrancar con la nueva imagen.** El último paso para aplicar la actualización es reiniciar el dispositivo para iniciar con la nueva imagen. En este escenario, podría ser que algo fallara, lo que causaría que la placa no pudiera iniciarse correctamente. Ante esta amenaza, es importante al descargar la actualización no sobrescribir el firmware actual y descargar la imagen en otra partición de memoria, de esta manera siempre sería capaz de revertir los cambios y volver a un estado funcional a partir del cual volver a intentarlo.

3.2. Plataformas IoT

Las plataformas IoT surgieron como una capa middleware encargada de mediar entre las capas de hardware y aplicación, pero con los años sus funcionalidades han evolucionado y ahora a través de la mayoría de plataformas se pueden recopilar datos de todos los dispositivos conectados a ella a través de diferentes protocolos, realizar configuraciones, modificar la configuración, proporcionan herramientas de análisis de datos, procesamiento y actualizaciones de firmware inalámbricas, entre otras funcionalidades. Es este último punto el que nos interesa, ya que en grades proyectos la gestión de dispositivos se suele realizar a través de una plataforma IoT, y con ello, desde ahí programar actualizaciones y llevar un control de versiones de firmware de manera totalmente inalámbrica (OTA).

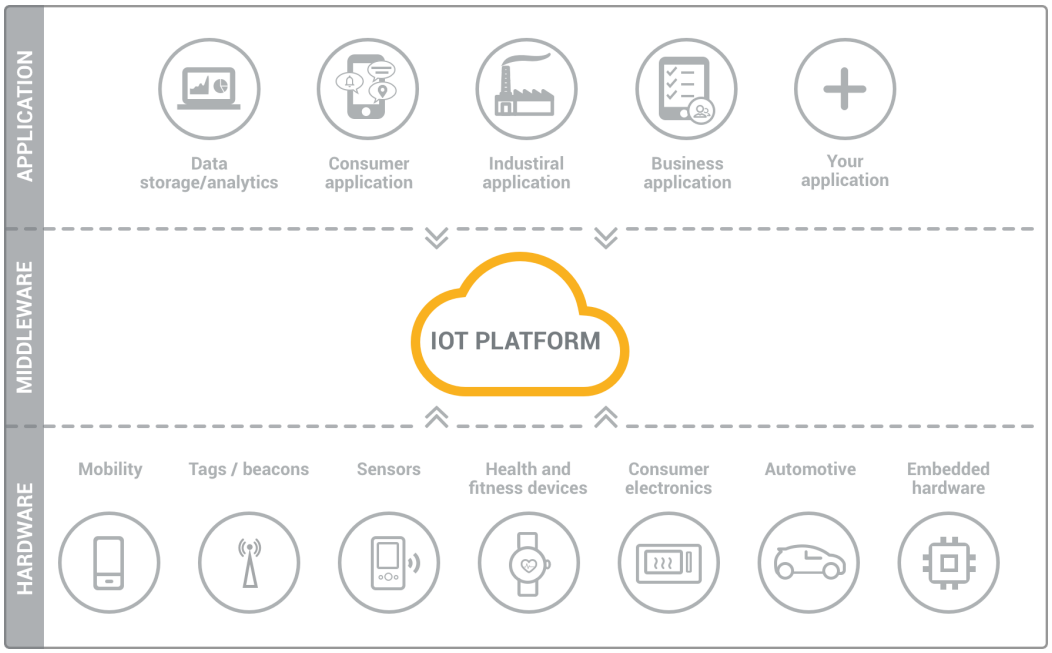

Figura 3.2: Plataformas IoT. Fuente: `https://www.kaaproject.org/blog/what-is-iot-platform`

4. Dispositivos comerciales de IoT

Tal y como se describía en la introducción, en este capítulo se analizarán distintas placas hardware y sistemas operativos. En la sección del hardware, se verá qué opciones hay en el mercado y qué nos ofrece cada una de ellas, para así más adelante poder elegir varios modelos de placas sobre las que realizar las pruebas. En la sección de análisis de sistemas operativos veremos qué son, por qué son imprescindibles para así, seleccionar varios modelos sobre los que realizar las pruebas.

4.1. Kits de prototipado hardware para IoT

4.1.1. Análisis comparativo

A la hora de comparar los kits de desarrollo hardware comerciales sobre los que poder implementar un programa, he analizado los dispositivos más usados en este ámbito, entre los que se encuentran placas de desarrollo más generales y otras más específicas, todas ellas aptas para ser usadas en IoT.

Todas ellas tienen una base común: un microcontrolador, encargado de automatizar procesos y procesar información, una memoria para almacenar información y otros periféricos que dotan al dispositivo de funciones más específicas. En este apartado analizaremos los siguientes campos:

- **Procesador principal**. Todas las placas de desarrollo incorporan un microcontrolador, el "cerebro" capaz de ejecutar las órdenes definidas en el programa. En él hemos analizado su procesador, la arquitectura que usan, así como a qué frecuencia trabajan, o, en otras palabras, la velocidad a la cual se ejecutan las instrucciones.

- **Memorias**. Otro dato a tener en cuenta son las memorias de las que dispone. Nos centraremos en la memoria de programa (ROM) y la memoria de acceso aleatorio (RAM).

- **Batería**. Para algunas soluciones/prototipos puede ser interesante que el producto disponga de batería, para hacerlo totalmente inalámbrico.

- **Conectividad**. Uno de los campos más importantes a analizar, ya que dependiendo de la finalidad del dispositivo y del entorno al que esté destinado debe soportar una tecnología u otra. Hay incluso algunos fabricantes que lanzan varios modelos parecidos cuyas únicas diferencias son el módulo de conectividad inalámbrica que implementa. Las más populares son:

- **WiFi**, en todos sus protocolos. Es una tecnología que, desde el punto de vista del IoT es ineficiente en el consumo de energía y en el aprovechamiento del ancho de banda. Se suele utilizar para conectar el dispositivo a internet y poder acceder a él desde cualquier parte del mundo, no para conectar diferentes nodos entre ellos. La reciente presentación del nuevo protocolo IEEE 802.11ah (también conocido como WiFi HaLow), orientado al internet de las cosas, marca un antes y un después en el uso de WiFi para IoT.
- **Ethernet**. Este estándar es utilizado para soluciones muy particulares en las que se dispone de una toma de red cercana. Con ella se pueden alcanzar grandes velocidades, aunque en IoT no se suele necesitar esta característica.
- **Zigbee**. Es un conjunto de protocolos, caracterizado por su bajo consumo (lo que permite crear soluciones con batería de larga duración) y su topología de red en malla en el que cada nodo está conectado a varios nodos y si alguno de ellos cae, se puede enviar el mensaje por otro camino. Estas características le hacen ideal para uso en domótica.
- **Bluetooth**. Es una tecnología en constante evolución que la banda de 2.4GHz por lo que su alcance es muy limitado. Con el lanzamiento de la versión 5 y más tarde BLE (Bluetooth Low Energy) es una especificación muy utilizada ya que permite funcionar como red en malla y consume muy poca energía.
- **2G/3G/LTE/5G**. Estas tecnologías están pensadas para soluciones industriales, donde el dispositivo no tiene acceso a otra red de comunicaciones ya que está en un sitio remoto o se encuentra, por ejemplo, en continuo movimiento. Tiene una cobertura del orden de los kilómetros.

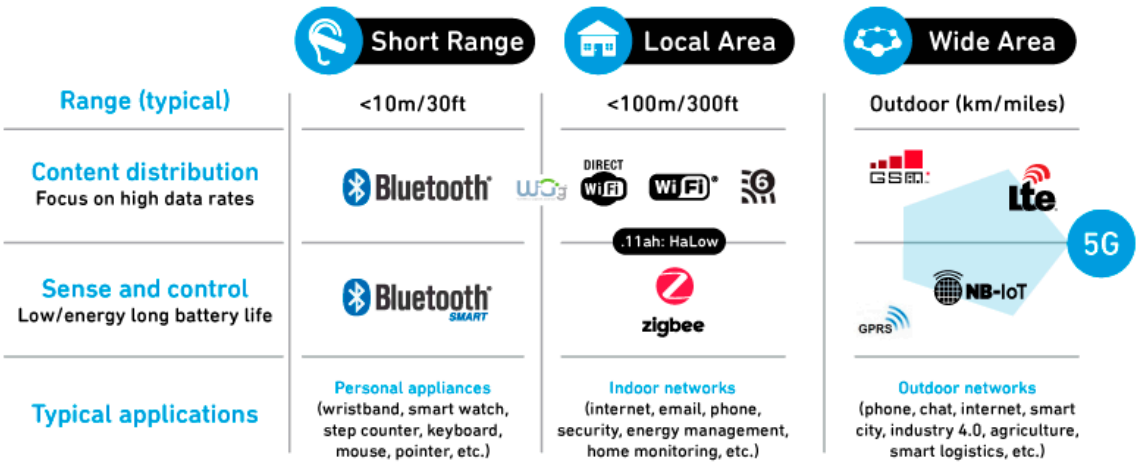

Figura 4.1: Comparativa de las opciones de conectividad. Fuente: `https://www.edn.com/iot-connectivity-wi-fi-and-bluetooth-are-winning/`

- **GPIO**. Las placas de desarrollo nos proporcionan una serie de pines de entrada/salida de propósito general que nos permiten conectar cualquier periférico, sensor o actuador externo. Dependiendo de las funciones externas que se necesiten será necesario disponer de más GPIOs o menos. También será importante conocer la tecnología que utilizan para saber si los periféricos externos que se conecten deben entregar/funcionar a 3.3V o 5V.
- **Resto de periféricos**. A menudo las placas incorporan otros periféricos que pueden ser útiles dependiendo de la finalidad. Entre ellos, se han analizado si la placa dispone de ADC (convertidor análogo/digital), UART (transmisor/receptor asíncrono universal), SPI (interfaz periférica serie), PWM (modulación por ancho de pulsos) o bus CAN (controlador de red de área).

En la Tabla 4.1 se recoge la tabla comparativa que se ha elaborado recogiendo las características analizadas en los principales dispositivos comerciales que hay en el mercado.

Tabla 4.1: Tabla comparativa del análisis de plataformas hardware

DEVICE	MAIN PROCESSOR			MEMORY	BATTERY	CONNECTIVITY	GPIO VOLTAGE	GPIO	PERIPHERAL				
	MODEL	WIDTH	CLOCK						ADC	UART	SPI	PWM	CAN
Raspberry PI Zero	ARMv8	64 bit	1.2 GHz	microsd, 512 MB RAM	-	Bluetooth 4.1, WIFI	3.3 V	40	0	2	1	2	-
Raspberry PI 3B	ARM-Cortex-A53	64 bit	1.4 GHz	microsd, 1GB RAM	-	WiFi, ethernet, Bluetooth 4.2	3.3 V	40	0	2	1	2	-
Raspberry PI 4B	ARM-Cortex A72	64 bit	1.5 GHz	microsd, 1-4GB RAM	-	WiFi, ethernet, Bluetooth 5	3.3 V	40	0	6	1	4	-
ESP32 DevKit	Xtensa Dual-Core	32 bit	160 MHz	4 MB flash, 512 kB RAM	-	WiFi, Bluetooth 4.2	3.3 V	25	6	3	2	2	-
ESP8266 NodeMCU	Xtensa Single-core	32 bit	80 MHz	2 MB flash, 160 kB RAM	-	WiFi	3.3 V	17	1	1	1	2	-
Arduino UNO	ATmega328P	8 bit	16 MHz	32 KB flash, 2 KB SRAM	-	WiFi	5 V	14	6	1	1	6	-
Arduino MKR 1000	SAMD21 Cortex-M0+	32 bit	48 MHz	256 KB flash, 32 KB SRAM	-	WiFi	3.3 V	8	7	1	1	12	-
Arduino MKR 1010 WIFI	SAMD21 Cortex-M0+	32 bit	48 MHz	256 KB flash, 32 KB SRAM	-	WiFi, bluetooth 4.2	3.3 V	8	7	1	1	12	-
Arduino MKR NB 1500	SAMD21 Cortex-M0+	32 bit	48 MHz	256 KB flash, 32 KB SRAM	-	bluetooth 4.2, WiFi, LTE	3.3 V	8	7	1	1	13	-
Arduino MKR WAN 1310	SAMD21 Cortex-M0+	32 bit	48 MHz	2MB flash, 32 KB SRAM	-	bluetooth 4.2, WiFi, LTE	3.3 V	8	7	1	1	13	-
Particle Argon	ARM Cortex-M4F	32 bit	64 MHz	1MB flash, 256KB RAM	-	WiFi, Bluetooth 5	3.3 V	20	6	1	1	8	-
Particle Boron	ARM Cortex-M4F	32 bit	64 MHz	1MB flash, 256KB RAM	-	2G/3G, LTE, BLE	3.3 V	20	6	1	1	8	-
Particle Photon	ARM Cortex M3	32 bit	64 MHz	1MB flash, 128KB RAM	-	WiFi	3.3 V	18	8	-	2	9	1
Nordic Thingy:52	ARM Cortex-M4F	32 bit	64 MHz	256 KB flash, 32KB RAM	1440 mAh	BLE, NFC	3.3 V	8	2	1	1	1	-
XMC4800 Infineon	ARM Cortex-M4	32 bit	144 MHz	1024KB Flash, 200KB RAM	-	WIFI, ethernet	3.3 V	20	6	1	1	6	6
NuMaker-IoT-M487	ARM Cortex-M4	32 bit	192 MHz	512KB flash, 160KB SRAM	-	WIFI, ethernet	1.8 - 3.6 V	16	16	6	4	16	2
MXChip DevKit AZ3166	ARM Cortex-M4	32 bit	100 MHz	3MB flash, 256 KB RAM	-	WIFI	3.3 - 5.5 V	3 1	2	2	1	1	-

4.1.2. Conclusiones

Son muchas las soluciones que nos ofrecen los fabricantes, todas ellas con una estructura común y pequeños detalles que las diferencian. Será necesario por ello, para la elección de un hardware específico sobre el que desarrollar un producto, analizar cuáles son las necesidades principales, qué tipo de conectividad se requiere, qué periféricos se necesitan para dotar al producto de las funcionalidades deseadas, qué potencia de procesamiento es la necesaria, etc.

4.2. Sistemas operativos en IoT

El Sistema Operativo (SO) es el encargado de administrar el hardware y software del sistema y asignar los recursos, incluyendo el procesamiento, la memoria y el almacenamiento. Existen sistemas operativos diseñados específicamente para satisfacer las demandas y especificaciones particulares de los dispositivos IoT. El SO es imprescindible para gestionar la seguridad, la comunicación, la administración remota (que usaremos más adelante para la programación vía OTA) y otras necesidades.

El SO juega un papel fundamental en una aplicación IoT ya que permite que los dispositivos y las aplicaciones se conecten entre sí y con otros sistemas, como plataformas y servicios en la nube. El SO de IoT también administra la potencia de procesamiento y otros recursos necesarios para recopilar, transmitir y almacenar datos.

4.2.1. Análisis comparativo

De los numerosos sistemas operativos que hay para IoT, he escogido aquellos más usados y he analizado varios campos que pueden ser interesantes a la hora de determinar por cuál decantarte en función de tus necesidades:

- **Autoría**. Nos indica si el software (en este caso sistema operativo) es de código abierto (y por ende se puede utilizar/editar de manera libre) o si por el contrario hace falta una licencia para su uso.
- **Memoria ROM/RAM mínima**. Dependiendo de la complejidad del sistema operativo y las funcionalidades que este incorpore, se requerirá un mínimo de prestaciones hardware para que se pueda ejecutar correctamente. Algunos sistemas operativos como TinyOS están diseñados para hardware muy básicos y otros como Windows 10 IoT requieren mayor memoria y capacidad de procesamiento.
- **Soporte para C y C++**. A pesar de que es inevitable que algunas partes del código del SO estén desarrolladas en ensamblador, se analiza si ofrece soporte para lenguajes de programación más alto nivel como C o C++.
- **Multihilo**. Para ejecutar varias tareas/servicios de manera concurrente se requiere que el sistema operativo tenga soporte para multihilo (multithreading).

- **Real-Time**. Un Real Time Operating System (RTOS) es un SO que administra los recursos del hardware, gestiona las aplicaciones y procesa los datos en tiempo real. RTOS define restricciones temporales como el tiempo de procesamiento de las tareas o la latencia de interrupción para determinar si el sistema operativo es en tiempo real o no. La diferencia clave entre RTOS y un SO de propósito general radica en su alto grado de fiabilidad y consistencia en el tiempo entre la aceptación y finalización de la tarea de la aplicación.

En base a estas características se ha creado la siguiente tabla:

Tabla 4.2: Tabla comparativa del análisis de sistemas operativos en IoT

OS	SOURCE	MIN RAM	MIN ROM	C SUPPORT	C++ SUPPORT	MULTI THREADING	REAL-TIME
RIOT OS	Open source	$\sim$ 1.5kB	$\sim$ 5kB	✓	✓	✓	✓
FreeRTOS	Open source	$\sim$ 300B	$\sim$ 2kB	✓	✓	✓	✓
Windows 10 IoT	Licensed	$\sim$ 256MB	$\sim$ 2GB	×	✓	✓	×
TinyOS	Open source	$<$ 1kB	$<$ 4kB	×	$\sim$	✓	✓
Zephyr	Open source	$<$ 5kB	$<$ 50kB	×	×	✓	✓
Contiki	Open source	$<$ 2kB	$<$ 30kB	$\sim$	×	$\sim$	$\sim$

✓Supported; $\sim$ Partially supported; × Not supported

Pero no todos los sistemas operativos para IoT son compatibles con todos los microcontroladores, algunos de ellos han sido diseñados para ciertas arquitecturas. A continuación se recoge la relación de compatibilidad entre arquitecturas de microcontroladores y los sistemas operativos analizados anteriormente:

Tabla 4.3: Compatibilidades entre arquitecturas de microcontroladores y sistemas operativos

OS	AVR	ARM	PIC32	PIC8	x86	XTENSA
RIOT OS	✓	✓	✓	✓	✓	✓
FreeRTOS	✓	✓	✓	✓	×	✓
Windows 10 IoT	✓	✓	×	×	✓	×
TinyOS	✓	✓	✓	✓	×	×
Zephyr	×	✓	×	×	✓	✓
Contiki	✓	✓	✓	✓	×	✓

✓ Supported; × Not supported

4.3. Preparación de las pruebas

Una vez ya hemos visto en qué consisten las técnicas OTA, para entenderlas mejor y probar una de las opciones que hay en el mercado, en el siguiente capítulo realizaré las pruebas en las que plantearé dos escenarios, uno de ellos desarrollando una solución OTA propia probándola sobre el dispositivo ESP32 DevKitC corriendo el sistema operativo freeRTOS, y otro probando los servicios la plataforma de IoT del fabricante Particle sobre el dispositivo Particle Argon.

5. Realización de pruebas

En este capítulo se realizarán las pruebas que antes se comentaban planteando dos escenarios, uno en el que se desarrollará un firmware con capacidad de actualización OTA, y otro en el que probaremos una solución comercial.

5.1. Desarrollo de un firmware con capacidad de actualización OTA

Como se adelantaba en la introducción, el propósito de esta sección será la de desarrollar usando el framework ESP-IDF[1] un software para la placa ESP32-DevKit, de esta manera entenderemos mejor cuáles son los requisitos y como se desarrolla una actualización OTA en una opción comercial. Todo el proyecto se ha desarrollado en una máquina Linux y puede encontrarse en el repositorio: `https://github.com/javicano8tfg/ota`

5.1.1. Desarrollo del programa

La idea es desarrollar un firmware que sea capaz de correr simultáneamente dos tareas[2], una que contenga un programa en el que hay un sensor y un LED conectado a la placa, y cada vez que el sensor detecta movimiento se enciende el LED (una especie de 'alarma'), y otra que compruebe continuamente si hay una nueva versión de firmware disponible, y en el caso de que la haya, la descargue y la instale. El programa se ha desarrollado en el lenguaje de programación `C`, todo el código se adjunta en el Apéndice A. Consta de los siguientes ficheros:

- `main.c`, que contiene el programa principal, configuración de ciertos parámetros e incluye las dependencias necesarias de ESP-IDF.
- `wifi.c`. Contiene funciones necesarias para realizar la conexión wifi.
- `wifi.h`. Header que contiene el prototipo de las funciones del fichero `wifi.c`, así como la definición del SSID y contraseña de la red wifi.
- `Makefile`. Archivo necesario para la compilación usando la herramienta `make`.

[1] ESP-IDF es el framework oficial de desarrollo para aplicaciones IoT de la compañía Espressif,

[2] Una tarea es, en el sistema operativo `freeRTOS` un programa independiente que desarrolla operaciones específicas.

- `component.mk`. Componente incluido en la compilación (en este caso nos indica la ruta del certificado)
- `update_info.json`, fichero en formato `.json` que se aloja en internet y es consultado por la placa para ver si hay una nueva versión de software y comenzar la actualización.
- `github_cert.pem`. Archivo que contiene el certificado del servidor donde se aloja el binario que permite que la descarga del fichero sea cifrada. Certificado de `www.github.com`.

Una vez comentado la función de cada fichero, pasaré a explicar la lógica del programa `main.c`:

1. Al principio se definen ciertas constantes que indican la versión del firmware que llevará ese programa, la url donde se encuentra el fichero `update_info.json`, configuración de los pines GPIO para la ejecución de un programa que se explicará más adelante, constante que modifica si el LED se enciende o parpadea cuando se detecta movimiento y otra variable que define cada cuántos segundos se debe comprobar si hay una nueva versión disponible.
2. A continuación se inicializa el wifi y se lanzan dos tareas, la que comprueba el estado del LED (`alarm_task`) y la que comprueba el estado del firmware (`check_firmware_task`).
3. La tarea de la 'alarma' es un bucle infinito que se queda comprobando cada segundo el estado del sensor conectado, y en el caso de que se detecte algún movimiento, se manda un mensaje por consola y, en función del valor de la variable `BLINK`, hará encenderse el LED o hará que parpadee para notificarnos visualmente.
4. La tarea que comprueba la versión del firmware que tenemos en la web, es otro bucle infinito que, en primer lugar obtiene el valor de los campos del fichero `update_info.json` ubicado en `URL_JSON` a través del protocolo HTTP. Si todos los campos del fichero tienen el formato correcto, comprueba si el campo `version` de `update_info.json` contiene una versión superior al firmware que está corriendo en ese momento (`firmware_servidor > FIRMWARE_ACTUAL`). En el caso de que la versión que se está ejecutando sea mayor o igual a la que hay en el servidor, se volvería a realizar esta misma comprobación pasados `DELAY_NEW_FIRMWARE` segundos. En el caso de que haya una versión más nueva disponible, se detendrá la ejecución de la tarea `alarm_task` y se comenzará la descarga cifrada del archivo binario `archivo`, especificado en `update_info.json`. Esta nueva imagen se almacena en una partición de la flash dedicada a la imagen OTA, para que en caso de que se descargue mal, se pierda la conexión a internet o se corte la alimentación, el dispositivo pueda seguir ejecutando el firmware que estaba corriendo hasta el momento. Una vez la descarga se ha realizado correctamente se reinicia la placa y esta arranca en la dirección de memoria donde se había guardado la imagen descargada, completando el proceso de actualización. Todo esto es realizado gracias a las APIs `esp_https_ota.h` y `esp_http_client.h` proporcionadas por el fabricante de la placa, Espressif.

5.1.2. Descarga y configuración de herramientas de desarrollo

Todo el entorno de desarrollo y las pruebas se han realizado en una máquina Linux. En este apartado se explicará como descargar y configurar las herramientas necesarias para poder desarrollar, compilar y flashear el software.

Es necesario instalar una serie de paquetes y dependencias para poder compilar con ESP-IDF:

Código 5.1: Instalación paquetes necesarios

```
javicano8@ubuntu:~$ sudo apt-get install gcc git wget libffi-dev libssl-dev
bison gperf python python-setuptools python-cryptography python-pip python-serial
python-future make libncurses-dev flex
```

A continuación, descargamos herramientas necesarias para el compilador y actualizamos las variables de entorno:

Código 5.2: Instalación de herramientas para compilar

```
javicano8@ubuntu:~$ wget dl.espressif.com/dl/xtensa-esp32-elf-linux64.tar.gz
javicano8@ubuntu:~$ mkdir -p ~/esp32
javicano8@ubuntu:~$ cd ~/esp32
javicano8@ubuntu:~/esp32$ tar -xzf ~/Downloads/xtensa-esp32-elf-linux64.tar.gz

javicano8@ubuntu:~/esp32$ export PATH="$HOME/esp32/xtensa-esp32-elf/bin:$PATH"
```

5.1.3. Descarga y configuración de ESP-IDF

Para continuar es necesario instalar las bibliotecas/API específicas para ESP32, proporcionadas por el fabricante y disponibles en su repositorio, se descargan con la herramienta `git`[3]. También es necesario añadir la ruta de esta API a las variables de entorno e instalar los paquetes de python necesarios:

Código 5.3: Instalación API para ESP32

```
javicano8@ubuntu:~/esp32$ git clone --recursive
https://github.com/espressif/esp-idf.git

javicano8@ubuntu:~/esp32$ export IDF_PATH = ~/esp32/esp-idf

javicano8@ubuntu:~/esp32$ python -m pip install --user -r
$IDF_PATH/requirements.txt
```

[3]Git es un software de control de versiones instalado en la Subsección 5.1.2

A continuación es necesario configurar ciertos parámetros de la API para adecuarla a nuestro programa. Para ello, nos situamos en la carpeta del proyecto y lanzamos el siguiente comando que nos abre la ventana de configuración que muestra la Figura 5.1.:

Código 5.4: Configuración ESP-IDF

```
javicano8@ubuntu:~$ cd ~/esp32/tfg_ota
javicano8@ubuntu:~/esp32/tfg_ota$ make menuconfig
```

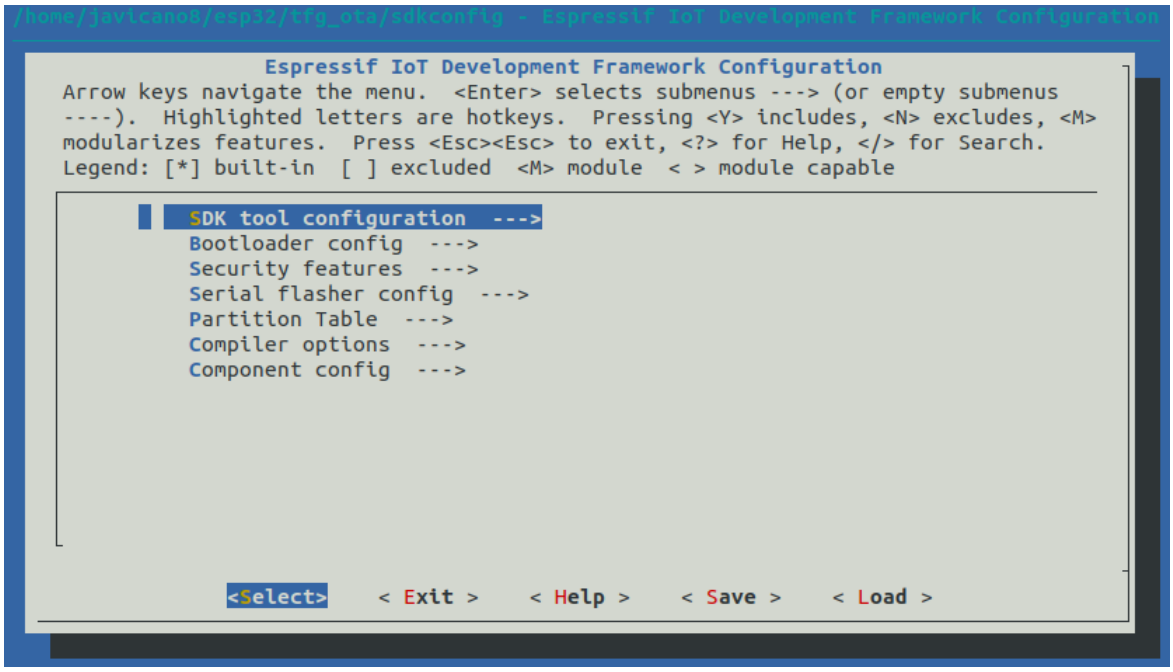

Figura 5.1: Menú principal de configuración ESP-IDF

A través de este menú, será necesario cambiar ciertas configuraciones. En primer lugar, en el apartado `SDK tool configuration` tendremos que seleccionar el puerto serie por defecto y fijar el tamaño de la memoria flash, en el caso de la placa de desarrollo ESP32-DevKitC, 4 MB, así tendremos espacio suficiente para almacenar la imagen descargada por OTA.

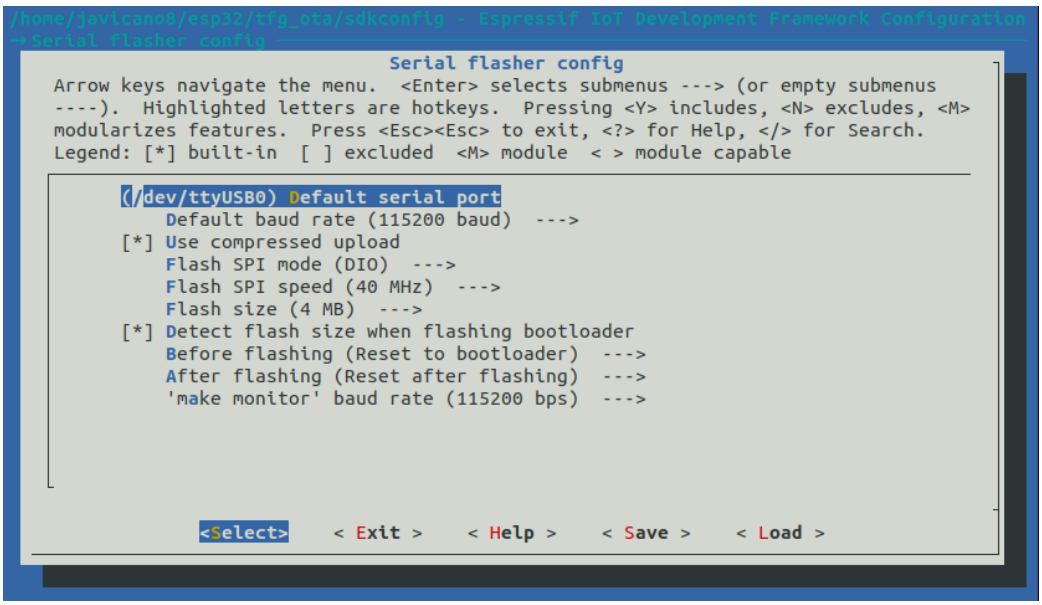

Figura 5.2: Configuración SDK tool

A continuación, en el apartado de `Partition table`, debemos seleccionar `Factory app, two OTA definitions`, tal y como se indica en la Figura 5.3 y Figura 5.4

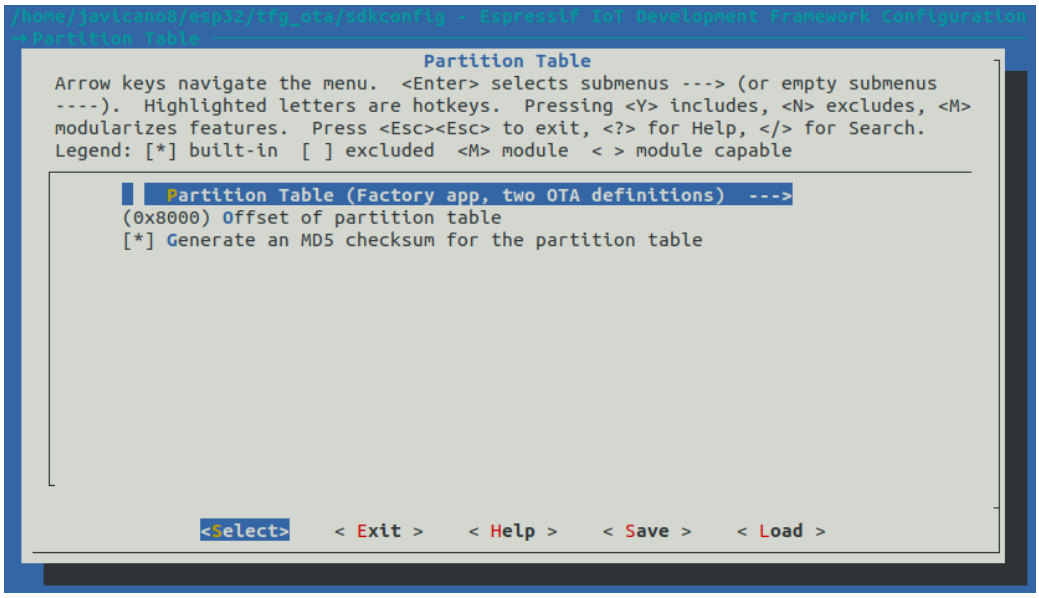

Figura 5.3: Configuración tabla de particiones de memoria

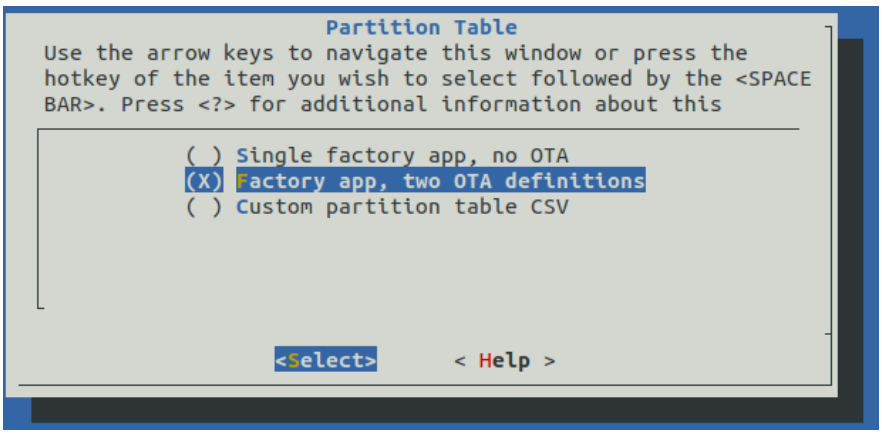

Figura 5.4: Selección partición de memoria para imagen OTA

Esto nos deja los 4 MB de memoria flash estructurada de la siguiente manera:

Tabla 5.1: Estructura de la memoria preparada para recibir actualizaciones OTA

NAME	TYPE	SUBTYPE	OFFSET	SIZE
bootloader	data	bootloader	0x1000	0x8000
nvs	data	nvs	0x9000	0x4000
otadata	data	ota	0xd000	0x2000
phy_init	data	phy	0xf000	0x1000
factory	0	0	0x10000	1M
ota_0	0	ota_0	0x110000	1M
ota_1	0	ota_1	0x210000	1M

Así, la aplicación inicial que flasheemos en la placa de desarrollo se almacenará en la partición `factory`. Con la API `esp_https_ota`, se descarga el binario (si procede), y se almacena en cualquiera de las particiones `ota_0` o `ota_1`, editando también la información del sector `otadata`. Tras resetear la placa, el bootloader[4] comprueba la información que hay escrita en `otadata`, y si hay algún programa almacenado en alguna partición OTA, arranca desde ahí. Si sucede cualquier error inesperado, se sigue almacenando la aplicación inicial en la partición `factory`, pues ese sector de la memoria no ha sido sobrescrito. Como medida de seguridad, tras actualizar cualquier sector de la memoria, se genera un MD5 checksum[5] que es comprobado cada vez que se iniciar el bootloader para asegurarse de la integridad de la información que hay en memoria.

[4]El bootloader es un gestor de arranque que determina qué se debe hacer al iniciar el microcontrolador.
[5]Un checksum es una función de redundancia que trata de detectar cambios no intencionados en datos, garantizando la integridad de la información.

5.1.4. Componentes y conexiones necesarias

Para la ejecución del programa es necesario disponer de una placa de desarrollo ESP32-DevkitC, un LED, una resistencia de 220 Ω y un sensor infrarrojo de movimiento PIR HC-SR501. Será necesario, previamente configurar el GPIO 32 como entrada y conectarlo al pin de señal de sensor PIR y el GPIO 14 como salida y conectarlo en serie con la resistencia y el LED, así como realizar las conexiones de alimentación y masa entre la placa y los otros dos componentes. Las conexiones deben quedar como en la Figura 5.17.

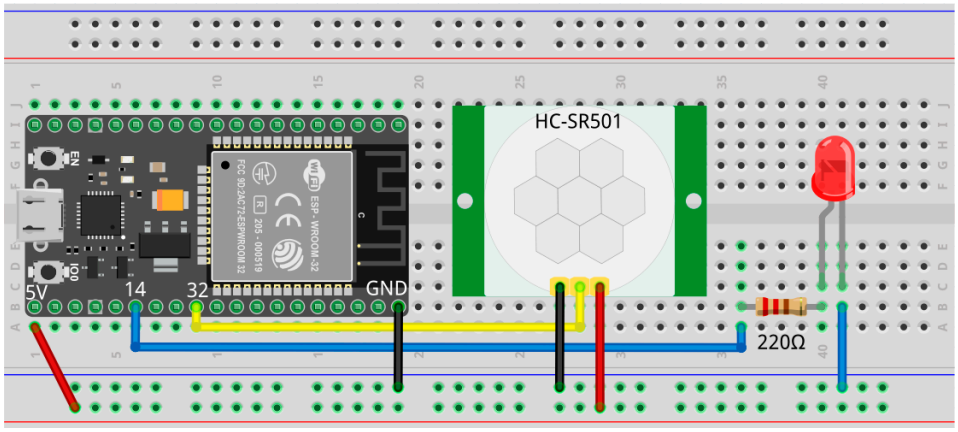

Figura 5.5: Esquema de conexiones

5.1.5. Prueba del programa

Una vez tenemos el código listo, las herramientas configuradas y las conexiones correctamente hechas, ya está todo listo para probar el programa. Para tener la capacidad de flashear por OTA, es necesario que la primera vez programemos la placa con el cable USB.
Ahora mismo, la jerarquía de directorios y ficheros es la siguiente:

```
home
└── esp32
    ├── esp-idf
    │   └── ..
    └── tfg_ota
        ├── Makefile
        └── main
            ├── main.c
            ├── wifi.c
            ├── wifi.h
            ├── github_cert.pem
            └── component.mk
```

Antes de nada, será necesario generar un el archivo binario de un firmware que queramos que sea el que se actualice. Para ello, del archivo `main.c` nos aseguramos de que el valor del parámetro `FIRMWARE_ACTUAL` sea `0.2` y el de `BLINK` sea `1`, indicando que para la versión `0.2` queremos que el LED parpadee al detectar movimiento. Ahora, es necesario compilar el proyecto para obtener el binario:

Código 5.5: Compilación del proyecto y generación del binario v0.2

```
javicano8@ubuntu:~$ cd ~/esp32/tfg_ota
javicano8@ubuntu:~/esp32/tfg_ota$ make flash
```

Esto genera el archivo binario `tfg_ota.bin` en la ruta `~/esp32/tfg_ota/build`. Inicialmente, el campo `version` del fichero `update_info.json` es `0.1`. Subimos el binario generado a internet (en mi caso, al repositorio github.com) y se actualiza la ruta absoluta del fichero en `update_info.json`, tal y como se encuentra en el Código A.7 .

Ahora, en nuestro programa `main.c`, introducimos la versión inicial del proyecto, para ello establecemos los parámetros `FIRMWARE_ACTUAL` a `0.1` y `BLINK` a `0`, borramos la memoria flash de la placa, programamos el nuevo firmware y nos conectamos a su puerto serie para ver el estado de la ejecución:

Código 5.6: Borrado de memoria

```
javicano8@ubuntu:~/esp32/tfg_ota$ make erase_flash flash monitor
```

```
VERSIÓN ACTUAL:  0.1
I (309) wifi:wifi driver task: 3ffc1540, prio:23, stack:3584, core=0
I (309) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE
I (309) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE
I (339) wifi:wifi firmware version: 5503362
I (339) wifi:config NVS flash: enabled
I (339) wifi:config nano formating: disabled
I (339) wifi:Init data frame dynamic rx buffer num: 32
I (349) wifi:Init management frame dynamic rx buffer num: 32
I (349) wifi:Init management short buffer num: 32
I (359) wifi:Init dynamic tx buffer num: 32
I (359) wifi:Init static rx buffer size: 1600
I (369) wifi:Init static rx buffer num: 10
I (369) wifi:Init dynamic rx buffer num: 32
W (389) phy_init: failed to load RF calibration data (0x1102), falling back to full calibration
I (519) phy: phy_version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 2
I (529) wifi:mode : sta (24:0a:c4:e8:09:84)
I (1869) wifi:new:<11,0>, old:<1,0>, ap:<255,255>, sta:<11,0>, prof:1
I (2849) wifi:state: init -> auth (b0)
I (2849) wifi:state: auth -> init (6c0)
I (2859) wifi:new:<11,0>, old:<11,0>, ap:<255,255>, sta:<11,0>, prof:1
I (5029) wifi:new:<11,0>, old:<11,0>, ap:<255,255>, sta:<11,0>, prof:1
I (5029) wifi:state: init -> auth (b0)
I (5029) wifi:state: auth -> assoc (0)
I (5039) wifi:state: assoc -> run (10)
I (5049) wifi:connected with OTA_TFG, channel 11
I (5049) wifi:pm start, type: 1

I (5789) event: sta ip: 192.168.137.80, mask: 255.255.255.0, gw: 192.168.137.1
WIFI conectado
Buscando una nueva versión de software...
Ya dispones de la última versión de firmware disponible (0.1)
Movimiento detectado
Movimiento detectado
Movimiento detectado
Movimiento detectado
Movimiento detectado
Movimiento detectado
Movimiento detectado
Movimiento detectado
Buscando una nueva versión de software...
Ya dispones de la última versión de firmware disponible (0.1)
```

Figura 5.6: Ejecución de la versión 0.1 del firmware desarrollado para ESP32

Como podemos observar, la placa nos muestra por consola cada vez que el sensor detecta un movimiento, y se comprueba cada minuto si en el servidor hay una versión más actual a la que está corriendo. Pese a estar el binario con la versión `0.2` subido, también es necesario reflejarlo en el campo `version` del `.json`, pues esto es lo que la aplicación comprueba. Una vez modificado e indicada la versión `0.2` en el `.json`, la placa comienza a actualizarse automáticamente, mostrando información sobre el desarrollador del firmware y la fecha:

```
Buscando una nueva versión de software...
Ya dispones de la última versión de firmware disponible (0.1)
Buscando una nueva versión de software...
Hay una versión nueva disponible! Actualizando a la v0.2
        Desarrollador: Javier Cano Asís
        Fecha: 30/10/2020
Descargando https://raw.githubusercontent.com/javicano8tfg/ota/master/tfg_ota.bin
I (192049) esp_https_ota: Starting OTA...
I (192049) esp_https_ota: Writing to partition subtype 16 at offset 0x110000
I (192509) esp_https_ota: esp_ota_begin succeeded
I (192509) esp_https_ota: Please Wait. This may take time
I (200389) esp_https_ota: Connection closed,all data received
I (200399) boot_comm: chip revision: 3, min. application chip revision: 0
I (200399) esp_image: segment 0: paddr=0x00110020 vaddr=0x3f400020 size=0x1d738 (120632) map
I (200499) esp_image: segment 1: paddr=0x0012d760 vaddr=0x3ffb0000 size=0x028b0 ( 10416)
I (200509) esp_image: segment 2: paddr=0x00130018 vaddr=0x400d0018 size=0x7d7fc (514044) map
0x400d0018: _stext at ??:?

I (200889) esp_image: segment 3: paddr=0x001ad81c vaddr=0x3ffb28b0 size=0x00d38 (  3384)
I (200899) esp_image: segment 4: paddr=0x001ae55c vaddr=0x40080000 size=0x00400 (  1024)
0x40080000: _iram_start at /home/javicano8/esp32/esp-idf/components/freertos/xtensa_vectors.S:1779

I (200899) esp_image: segment 5: paddr=0x001ae964 vaddr=0x40080400 size=0x14b08 ( 84744)
I (200969) boot_comm: chip revision: 3, min. application chip revision: 0
I (200969) esp_image: segment 0: paddr=0x00110020 vaddr=0x3f400020 size=0x1d738 (120632) map
I (201069) esp_image: segment 1: paddr=0x0012d760 vaddr=0x3ffb0000 size=0x028b0 ( 10416)
I (201079) esp_image: segment 2: paddr=0x00130018 vaddr=0x400d0018 size=0x7d7fc (514044) map
0x400d0018: _stext at ??:?

I (201469) esp_image: segment 3: paddr=0x001ad81c vaddr=0x3ffb28b0 size=0x00d38 (  3384)
I (201469) esp_image: segment 4: paddr=0x001ae55c vaddr=0x40080000 size=0x00400 (  1024)
0x40080000: _iram_start at /home/javicano8/esp32/esp-idf/components/freertos/xtensa_vectors.S:1779

I (201469) esp_image: segment 5: paddr=0x001ae964 vaddr=0x40080400 size=0x14b08 ( 84744)
I (201559) esp_https_ota: esp_ota_set_boot_partition succeeded
Reiniciando en 3, 2, 1...
I (205549) wifi:state: run -> init (0)
I (205549) wifi:pm stop, total sleep time: 186756651 us / 200497711 us
```

Figura 5.7: Actualización del firmware de la placa a la versión 0.2

Tras descargar la nueva versión en una de las particiones de la memoria dedicada a ello, la placa se reinicia y al iniciarse el bootloader arranca con el nuevo binario. Tal y como se muestra, ahora sí que tiene la última versión, y con ello, hemos actualizado la funcionalidad del programa de la alarma, el cual ahora hace que el LED parpadee cuando el sensor detecta algún movimiento.

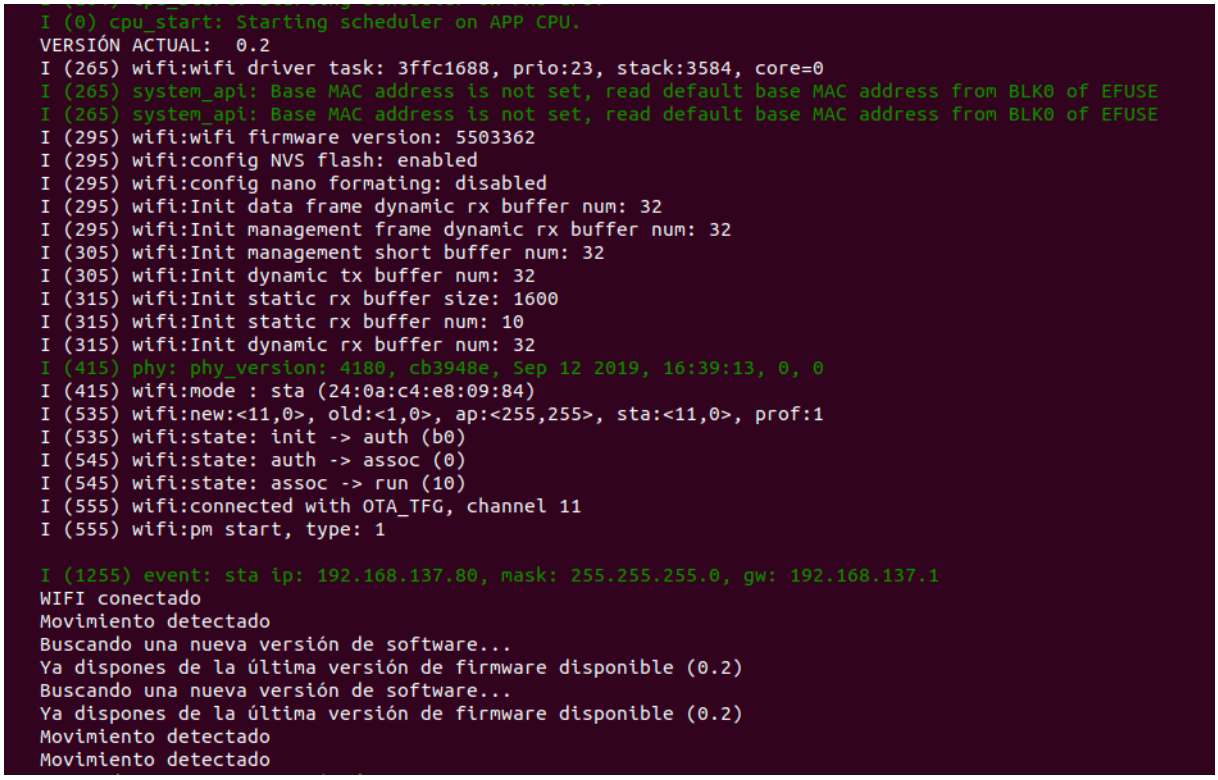

Figura 5.8: Ejecución de la versión 0.2 del firmware desarrollado tras actualización OTA

5.2. Prueba OTA comercial

En este segundo escenario se probará qué opciones nos ofrece una plataforma IoT, en particular la del fabricante Particle, y usaremos el dispositivo Particle Argon para realizar las pruebas.

5.2.1. Configuración inicial

A diferencia del primer escenario probado, en esta ocasión la configuración será mínima y a un mayor nivel, pues solo se ha de descargar la aplicación del fabricante[6] desde un dispositivo móvil, escanear el código QR que dispone, introducir nombre y contraseña del wifi y ya tendremos acceso total a los servicios:

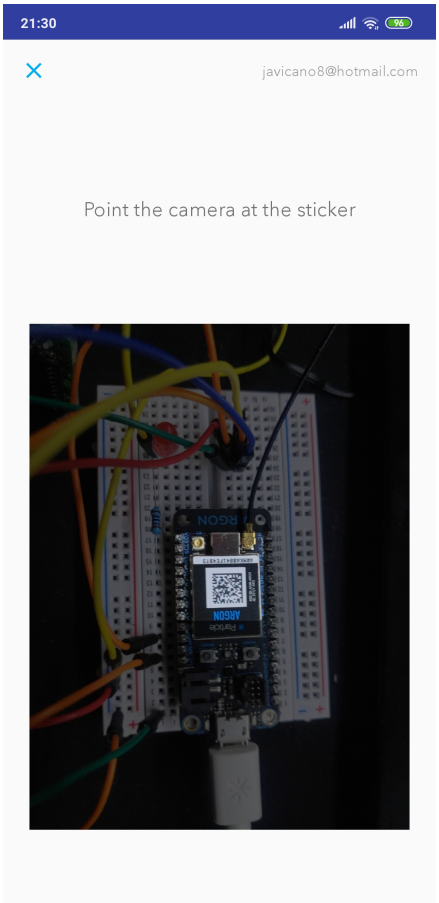

Figura 5.9: Se añade el dispositivo a la plataforma

De esta manera el dispositivo queda añadido a la plataforma y seremos capaces de gestionarlo de manera totalmente remota, obteniendo información sobre los eventos ocurridos en la placa o compilando y flasheando programas de manera totalmente inalámbrica.

[6]La aplicación se puede encontrar en `https://play.google.com/store/apps/details?id=io.particle.android.app`

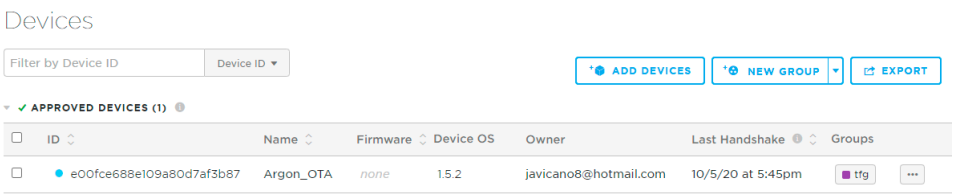

Figura 5.10: Lista de dispositivos añadidos a la plataforma

5.2.2. Prueba de los servicios OTA

Esta plataforma cuenta con un apartado dedicado para la gestión del firmware de todos los dispositivos que se hayan añadido previamente. En la pestaña 'Devices' se muestra el listado de equipos, el firmware actual que están corriendo así como la lista de grupos al que pertenecen. Agregar los dispositivos a grupos es una sencilla manera de indicar qué placas queremos que se actualicen, ya que se pueden programar actualizaciones para equipos que solo pertenezcan a un grupo en concreto. En este caso, agregamos la placa 'Particle Argon' al grupo 'tfg', como se muestra en la Figura 5.10

Después de esto deberemos desarrollar un programa para probar el correcto funcionamiento. Para ello utilizamos el `IDE` online[7] que nos permite compilar el programa y descargar el archivo binario con tan solo dos clicks. Para probar el funcionamiento haremos como en la sección pasada, desarrollaremos un programa con versión 1 (en este caso enciende el LED RGB que tiene la placa de color blanco) para más adelante actualizarlo a la versión 2 (se enciende el LED con colores aleatorios). Generamos ambos archivos binarios como se muestra en la Figura 5.11 y Figura 5.12

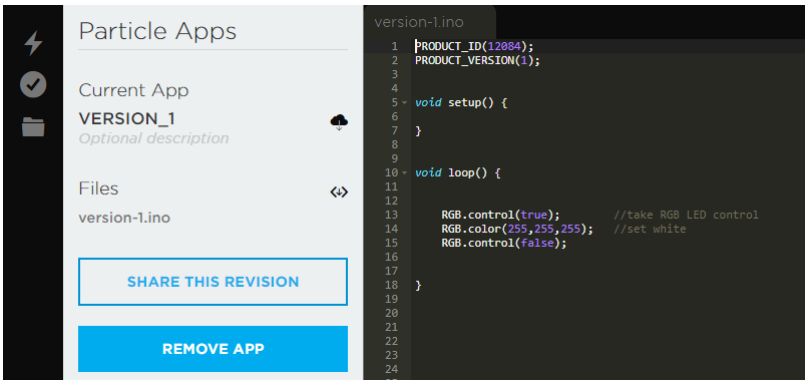

Figura 5.11: Compilación y generación del archivo binario con la versión 1

[7]IDE disponible en `https://build.particle.io/`

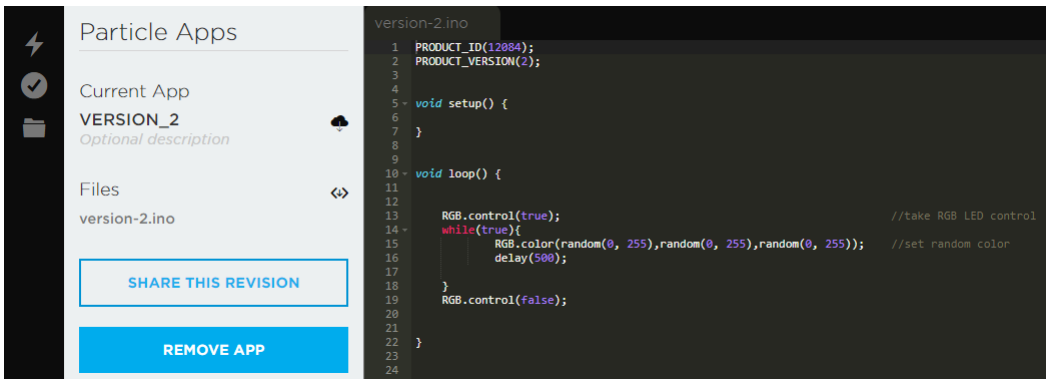

Figura 5.12: Compilación y generación del archivo binario con la versión 2

A continuación, a través del apartado 'Firmware' subimos ambos binarios indicando su información: qué versión son y una breve descripción:

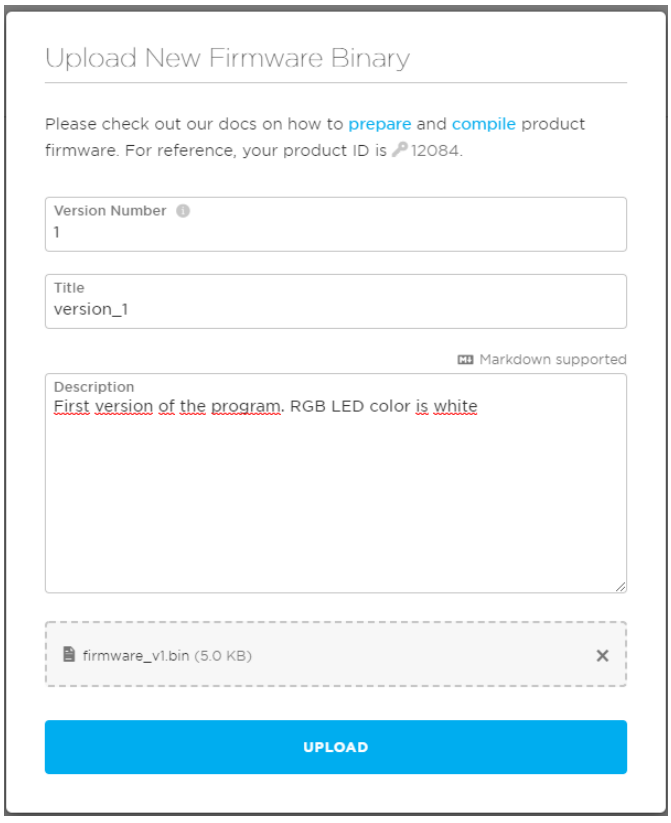

Figura 5.13: Carga de binario a la plataforma

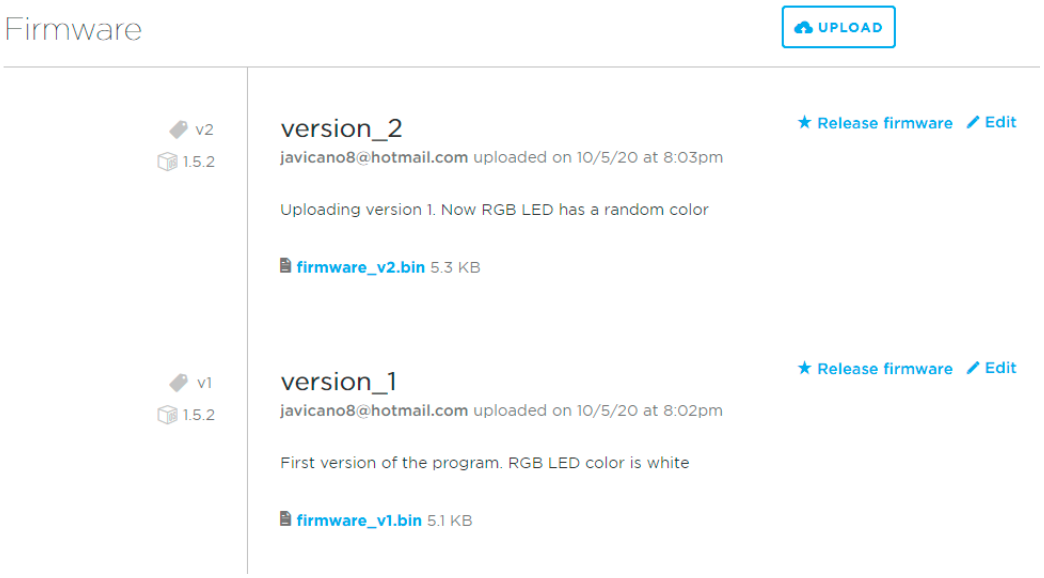

Figura 5.14: Repositorio de firmwares de la plataforma

Clickando en la opción 'Release firmware' nos permite lanzar este firmware y actualizar los grupos seleccionados con la versión deseada, en este caso lo haremos con la versión 1:

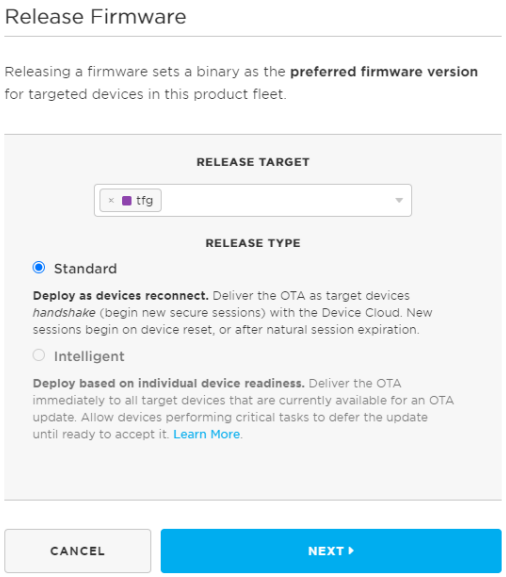

Figura 5.15: Selección de grupos para actualización de firmware

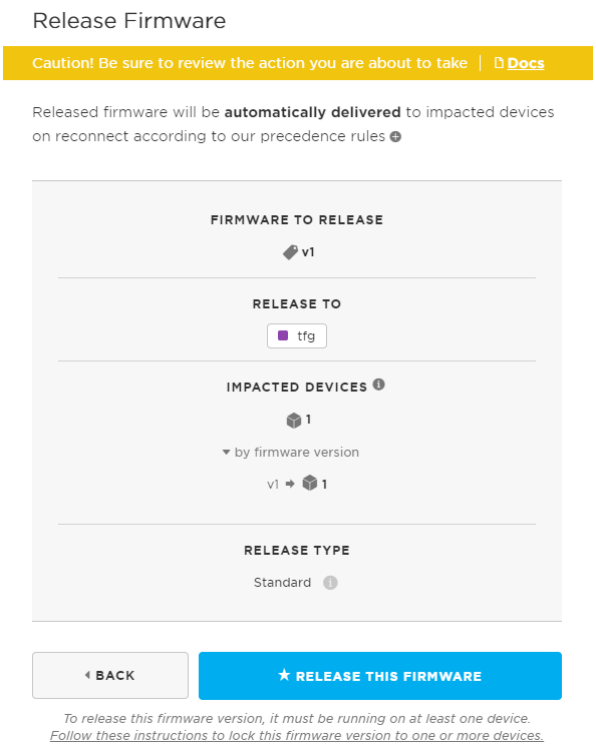

Figura 5.16: Confirmación de grupos para actualización de firmware

De la misma manera, si ahora lanzamos la versión 2, el dispositivo se actualiza incorporando el nuevo programa y los cambios son reflejados en el panel que nos muestra todos los eventos:

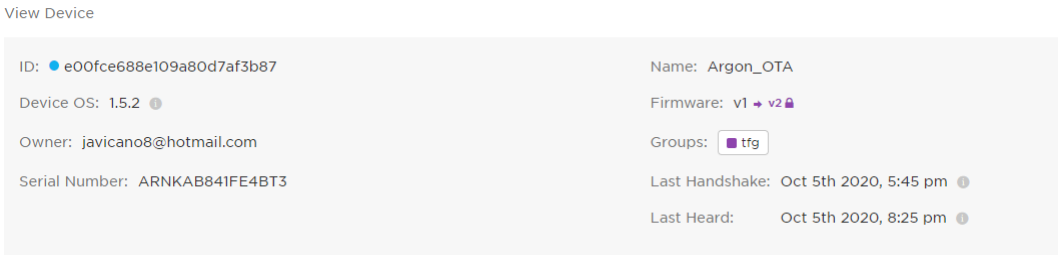

Figura 5.17: Panel de información de eventos para el dispositivo Particle Argon

6. Conclusiones

En este proyecto se han buscado dos objetivos, por un lado documentar de manera extensa cuáles son los equipos hardware y sistemas operativos más usados en el mundo IoT, analizando sus características y viendo qué nos ofrece cada uno, y por otro, probar de manera práctica diferentes técnicas OTA sobre distintos equipos para ver cómo se hace, qué implica y su dificultad.

Respecto a la documentación de los dispositivos, cabe mencionar que existen tantos modelos de dispositivos como aplicaciones puede darse al IoT, por ello, es necesario que a la hora de realizar un proyecto IoT, seamos capaces de analizar las necesidades técnicas de las que necesitamos que nuestro equipo disponga, para así, tras analizar una gran cantidad de equipos de la misma manera que en este trabajo se ha hecho, seamos capaces de elegir el que más nos convenga.

Por otro lado, la línea principal del proyecto era la de probar distintas técnicas OTA sobre distintos dispositivos IoT, y así se ha hecho. Por un lado, se ha desarrollado un firmware propio para el dispositivo 'Espressif ESP32-DevKitC' capaz de actualizarse vía OTA, en el que se ha aprendido como se realiza este proceso y qué elementos intervienen, y por otro lado, se ha probado la plataforma comercial IoT del fabricante 'Particle' con el dispositivo 'Particle Argon', que nos ha permitido comprobar la simpleza con la que esta actualización OTA puede hacerse, al alcance de todo el mundo. De ambas maneras hemos conseguido llevar a cabo actualizaciones OTA sobre distintos dispositivos, cada una tiene sus pros y sus contras: el firmware desarrollado propio requiere de una configuración a más bajo nivel, pero como ventaja ofrece un nivel de personalización increíble, pues se dispone de todo el código fuente y se pueden incluir las funcionalidades que se deseen. Por otro lado, en la solución comercial que se ha probado, como ventaja cabe destacar que la facilidad y el grado de detalle con la que está explicado cada apartado, hace que esta plataforma esté al alcance de todo el mundo y no se requieran apenas conocimientos previos para su uso. Por contra, pese a ser una plataforma muy completa, estamos limitados a usar las funcionalidades que el fabricante nos ofrece y, en ocasiones, para usar ciertas de ellas es necesario pagar.

Bibliografía

Atul, K. (2013). *How to view ssl certificate (pem file) using openssl?* https://onlineappsdba.com/index.php/2013/11/28/how-to-view-ssl-certificate-pem-file-using-openssl/.

Emmanuel, B., Oliver, H., Mesut, G., Matthias, W., y Thomas, S. (2013). Os for the iot - goals, challenges, and solutions. *Workshop Interdisciplinaire sur la Sécurité Globale*. https://hal.inria.fr/hal-00781769. Descargado de https://hal.inria.fr/hal-00781769

Espressif. (2020). *Esp-idf programming guide*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/.

freeRTOS. (2020). *Real-time operating system for microcontrollers*. https://www.freertos.org/.

Fritzing. (2020). *Open source software for the design of electronics hardware*. https://fritzing.org/.

Github. (2020). *Github website*. https://github.com/.

Heejung, Y., Muhammad Khalil, A., Mubashir, R., Husain, y Yousaf Bin, Z. (2018). Internet of things (iot): Operating system, applications and protocols design, and validation techniques. *Future Generation Computer Systems*. https://www.researchgate.net/publication/327138483. Descargado de https://www.researchgate.net/publication/327138483

Nick, L. (2018). A more secure and reliable ota update architecture for iot devices. *Texas Instrument journal*. https://www.ti.com/lit/wp/sway021/sway021.pdf?ts=1602005177497. Descargado de https://www.ti.com/lit/wp/sway021/sway021.pdf?ts=1602005177497

Particle. (2020). *Particle website*. https://www.particle.io/.

Sung Won, K., Oliver, H., Muhammad Khalil, Y., Afzaland Mohammed, y Yousaf Bin, Z. (2019). Internet of things (iot) operating systems management: Opportunities, challenges,and solution. *Sensors*. https://www.researchgate.net/publication/332415379. Descargado de https://www.researchgate.net/publication/332415379

Vlasios, T., Stamatis, K., Jan, H., David, B., y Catherine, M. (2014). *From machine-to-machine to the internet of things: Introduction to a new age of intelligence*. ilustrada.

Wikipedia. (2020). *Comparison of real-time operating systems.* `https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems`.

Lista de Acrónimos y Abreviaturas

ADC	Analog-to-Digital Converter.
API	Application Programming Interface.
BLE	Bluetooth Low Energy.
CAN	Controller Area Network.
ESP-IDF	Espressif IoT Development Framework.
GPIO	General Purpose Input/Output.
HTTP	Hypertext Transfer Protocol.
IDE	Integrated Development Environment.
IoT	Internet of Things.
LED	Light-Emitting Diode.
LTE	Long Term Evolution.
MB	MegaByte.
OTA	Over-The-Air.
PIR	Passive Infrared Sensor.
PWM	Pulse-Width Modulation.
QR	Quick Response.
RAM	Random Access Memory.
ROM	Read Only Memory.
RTOS	Real Time Operating System.
SDK	Software Development Kit.

SO Sistema Operativo.

SPI Serial Peripheral Interface.

SSID Service Set Identifier.

TFE Trabajo Final de Estudios.

TFG Trabajo Final de Grado.

TLS Transport Layer Security.

UART Universal Asynchronous Receiver-Transmitter.

USB Universal Serial Bus.

A. Código fuente

A.1. main.c

Código A.1: main.c

```
#include <string.h>
#include <time.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "cJSON.h"
#include "driver/gpio.h"
#include "esp_system.h"
#include "esp_log.h"
#include "esp_http_client.h"
#include "esp_https_ota.h"

#include "wifi.h"

#define FIRMWARE_ACTUAL 0.2
#define URL_JSON    "https://raw.githubusercontent.com/javicano8tfg/ota/master/update_info.json"
#define LED       GPIO_NUM_32
#define SENSOR      GPIO_NUM_14
#define BLINK     1
#define DELAY_NEW_FIRMWARE  60  // segundos que tarda en comprobar si hay un nuevo firmware


// Certificado .pem del servidor donde está alojado el binario, obtenido con:
// openssl s_client -showcerts -connect github.com:443
extern const char github_cert_pem_start[] asm("_binary_github_cert_pem_start");
extern const char github_cert_pem_end[] asm("_binary_github_cert_pem_end");

char dato_json[500];
xTaskHandle xAlarm_task;

esp_err_t _http_event_handler(esp_http_client_event_t *evento) {

  switch(evento->event_id) {
```

```
33            case HTTP_EVENT_ON_DATA:
34          if (!esp_http_client_is_chunked_response(evento->client)) {
35              strncpy(dato_json, (char*)evento->data, evento->data_len);
36          }
37          break;
38      default:
39          break;
40      }
41      return ESP_OK;
42  }
43
44
45
46  // Tarea con la funcionalidad de la alarma
47  void alarm_task(void *pvParameter){
48    bool on = true;
49    while(1) {
50    if(gpio_get_level(SENSOR)==1){
51      printf("Movimiento detectado\n");
52      if(BLINK){
53            if(on){
54          gpio_set_level(LED, 1);
55          on = false;
56        }
57        else{
58          gpio_set_level(LED, 0);
59          on = true;
60        }
61        }
62        else{
63        gpio_set_level(LED, 1);
64      }
65        }
66        else{
67            gpio_set_level(LED, 0);
68        }
69        vTaskDelay(1000 / portTICK_PERIOD_MS);
70    }
71  }
72
73
74  // Tarea que comprueba si hay actualización disponible cada DELAY_NEW_FIRMWARE↩
      ↪  segundos
75  void check_firmware_task(void *pvParameter) {
76
77    while(1) {
```

```
78
79      printf("Buscando una nueva versión de software...\n");
80
81      esp_http_client_config_t http_config = {
82        .url = URL_JSON,
83        .event_handler = _http_event_handler,
84      };
85
86      esp_http_client_handle_t http_client = esp_http_client_init(&http_config);
87      esp_http_client_perform(http_client);
88
89      cJSON *json = cJSON_Parse(dato_json);
90
91      cJSON *version = cJSON_GetObjectItemCaseSensitive(json, "version");
92      cJSON *archivo = cJSON_GetObjectItemCaseSensitive(json, "archivo");
93      cJSON *fecha = cJSON_GetObjectItemCaseSensitive(json, "fecha");
94      cJSON *autor = cJSON_GetObjectItemCaseSensitive(json, "autor");
95
96      if( version == NULL || archivo == NULL || fecha == NULL || autor == NULL){
97        printf("Faltan parámetros en el archivo de configuración JSON \n");
98        printf("Comprueba %s\n", URL_JSON );
99      }
100     else{
101       double firmware_servidor = version->valuedouble;
102
103       if(firmware_servidor < FIRMWARE_ACTUAL){printf("Tu versión de firmware ↩
    ↪ actual (%.1f) es superior a la que se encuentra en el servidor (%.1f) \n↩
    ↪ ", FIRMWARE_ACTUAL, firmware_servidor);}
104       else if (firmware_servidor == FIRMWARE_ACTUAL){ printf("Ya dispones de ↩
    ↪ la última versión de firmware disponible (%.1f) \n", FIRMWARE_ACTUAL);}
105       else {
106
107         printf("Hay una versión nueva disponible! Actualizando a la v%.1f\n", ↩
    ↪ firmware_servidor);
108         printf("  Desarrollador: %s\n", autor->valuestring );
109         printf("  Fecha: %s\n", fecha->valuestring );
110
111         vTaskDelete(xAlarm_task);
112
113         char *urlArchivo = archivo->valuestring;
114         printf("Descargando %s \n", urlArchivo);
115
116         esp_http_client_config_t ota_client_config = {
117           .url = urlArchivo,
118           .cert_pem = github_cert_pem_start,
119         };
```

```
120
121         if (esp_https_ota(&ota_client_config) == ESP_OK) {
122           printf("Reiniciando en 3,");
123           vTaskDelay((1000) / portTICK_PERIOD_MS);
124           printf(" 2,");
125           vTaskDelay((1000) / portTICK_PERIOD_MS);
126           printf(" 1");
127           vTaskDelay((1000) / portTICK_PERIOD_MS);
128           printf(".");
129           vTaskDelay((333) / portTICK_PERIOD_MS);
130           printf(".");
131           vTaskDelay((333) / portTICK_PERIOD_MS);
132           printf(".\n");
133           vTaskDelay((333) / portTICK_PERIOD_MS);
134
135           esp_restart();
136         }
137       }
138     }
139
140     esp_http_client_cleanup(http_client);
141           vTaskDelay((DELAY_NEW_FIRMWARE * 1000) / portTICK_PERIOD_MS);
142     }
143 }
144
145 void app_main() {
146
147   printf("VERSIÓN ACTUAL:  %.1f\n", FIRMWARE_ACTUAL );
148
149   gpio_set_direction(LED, GPIO_MODE_OUTPUT);
150
151   // Inicializa wifi
152   bool flag = false;
153   while(!flag){
154     flag = start_wifi();
155   }
156
157   wait_wifi();
158   printf("WIFI conectado \n");
159
160   // Lanza tarea de alarma
161   xTaskCreate(&alarm_task, "alarm_task", 2048, NULL, 5, &xAlarm_task);
162
163   // Lanza la tarea que comprueba si hay actualización disponible
164   xTaskCreate(&check_firmware_task, "check_firmware_task", 8192, NULL, 5, NULL↩
      ↪ );
```

```
165 }
```

A.2. wifi.c

Código A.2: wifi.c

```
1  #include "freertos/FreeRTOS.h"
2  #include "freertos/event_groups.h"
3  #include "esp_log.h"
4  #include "esp_wifi.h"
5  #include "esp_event_loop.h"
6  #include "nvs_flash.h"
7  #include <string.h>
8
9  #include "wifi.h"
10
11
12 static EventGroupHandle_t wifi_event_group;
13
14 static esp_err_t event_handler(void *ctx, system_event_t *evt){
15
16     switch(evt->event_id) {
17
18       case SYSTEM_EVENT_STA_START:
19     esp_wifi_connect();
20     break;
21
22   case SYSTEM_EVENT_STA_GOT_IP:
23     xEventGroupSetBits(wifi_event_group, BIT0);
24     break;
25
26   case SYSTEM_EVENT_STA_DISCONNECTED:
27     esp_wifi_connect();
28           break;
29
30   default: break;
31     }
32
33   return ESP_OK;
34 }
35
36
37 bool start_wifi(void) {
38
39     nvs_flash_init(); // necesario
```

```
40      wifi_event_group = xEventGroupCreate();
41      tcpip_adapter_init();
42      ESP_ERROR_CHECK(esp_event_loop_init(event_handler, NULL));
43      wifi_init_config_t wifi_init_config_default = WIFI_INIT_CONFIG_DEFAULT();
44      esp_wifi_init(&wifi_init_config_default);
45
46      esp_wifi_set_storage(WIFI_STORAGE_FLASH);
47      wifi_config_t wifi_config = {};
48      ESP_ERROR_CHECK(esp_wifi_get_config(ESP_IF_WIFI_STA, &wifi_config));
49
50      if (wifi_config.sta.ssid[0] == '\0' || wifi_config.sta.password[0] == '\0'↩
        ↪ ){
51        wifi_sta_config_t wifi_sta_config = {
52              .ssid = WIFI_SSID,
53              .password = WIFI_PASS,
54      };
55
56      wifi_config.sta = wifi_sta_config;
57
58      }
59
60      esp_wifi_set_mode(WIFI_MODE_STA);
61
62      esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config);
63      esp_wifi_start();
64
65    return true;
66  }
67
68  void wait_wifi()
69  {
70    xEventGroupWaitBits(wifi_event_group, BIT0, false, true, portMAX_DELAY);
71  }
```

A.3. wifi.h

Código A.3: wifi.h

```
#ifndef _WIFI_H_
#define _WIFI_H_

#define WIFI_SSID      "OTA_TFG"
#define WIFI_PASS      "OTA_TFG_JAVI"

bool start_wifi(void);
void wait_wifi();


#endif
```

A.4. Makefile

Código A.4: Makefile

```
#
# This is a project Makefile. It is assumed the directory this Makefile ↩
    ↪ resides in is a
# project subdirectory.
#

PROJECT_NAME := tfg_ota

include $(IDF_PATH)/make/project.mk
```

A.5. component.mk

Código A.5: component.mk

```
COMPONENT_EMBED_TXTFILES :=  ${PROJECT_PATH}/main/github_cert.pem
```

A.6. github_cert.pem

Código A.6: github_cert.pem

```
-----BEGIN CERTIFICATE-----
MIIEsTCCA5mgAwIBAgIQBOHnpNxc8vNtwCtCuF0VnzANBgkqhkiG9w0BAQsFADBs
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSswKQYDVQQDEyJEaWdpQ2VydCBIaWdoIEFzc3VyYW5j
ZSBFViBSb290IENBMB4XDTEzMTAyMjEyMDAwMFoXDTI4MTAyMjEyMDAwMFowcDEL
MAkGA1UEBhMCVVMxFTATBgNVBAoTDERpZ2lDZXJ0IEluYzEZMBcGA1UECxMQd3d3
LmRpZ2ljZXJ0LmNvbTEvMC0GA1UEAxMmRGlnaUNlcnQgU0hBMiBIaWdoIEFzc3Vy
YW5jZSBTZXJ2ZXIgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQC2
4C/CJAbIbQRf1+8KZAayfSImZRauQkCbztyfn3YHPsMwVYcZuU+UDlqUH1VWtMIC
Kq/QmO4LQNfE0DtyyBSe75CxEamu0si4QzrZCwvV1ZX1QK/IHe1NnF9Xt4ZQaJn1
itrSxwUfqJfJ3KSxgoQtxq2lnMcZgqaFD15EWCo3j/018QsIJzJa9buLnqS9UdAn
4t07QjOjBSjEuyjMmqwrIw14xnvmXnG3Sj4I+4G3FhahnSMSTeXXkgisdaScus0X
sh5ENWV/UyU50RwKmmMbGZJ0aAo3wsJSSMs5WqK24V3B3aAguCGikyZvFEohQcft
bZvySC/zA/WiaJJTL17jAgMBAAGjggFJMIIBRTASBgNVHRMBAf8ECDAGAQH/AgEA
MA4GA1UdDwEB/wQEAwIBhjAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIw
NAYIKwYBBQUHAQEEKDAmMCQGCCsGAQUFBzABhhhodHRwOi8vb2NzcC5kaWdpY2Vy
dC5jb20wSwYDVR0fBEQwQjBAoD6gPIY6aHR0cDovL2NybDQuZGlnaWNlcnQuY29t
L0RpZ2lDZXJ0SGlnaEFzc3VyYW5jZUVWUm9vdENBLmNybDA9BgNVHSAENjA0MDIG
BFUdIAAwKjAoBggrBgEFBQcCARYcaHR0cHM6Ly93d3cuZGlnaWNlcnQuY29tL0NQ
UzAdBgNVHQ4EFgQUUWj/kK8CB3U8zNllZGKiErhZcjswHwYDVR0jBBgwFoAUsT7D
aQP4v0cB1JgmGggC72NkK8MwDQYJKoZIhvcNAQELBQADggEBABiKlYkD5m3fXPwd
aOpKj4PWUS+Na0QWnqxj9dJubISZi6qBcYRb7TROsLd5kinMLYBq8I4g4Xmk/gNH
E+r1hspZcX30BJZr01lYPf7TMSVcGDiEo+afgv2MW5gxTs14nhr9hctJqvIni5ly
/D6q1UEL2tU2ob8cbkdJf17ZSHwD2f2LSaCYJkJA69aSEaRkCldUxPUd1gJea6zu
xICaEnL6VpPX/78whQYwvwt/Tv9XBZ0k7YXDK/umdaisLRbvfXknsuvCnQsH6qqF
0wGjIChBWUMo0oHjqvbsezt3tkBigAVBRQHvFwY+3sAzm2fTYS5yh+Rp/BIAV0Ae
cPUeybQ=
-----END CERTIFICATE-----
```

A.7. update_info.json

Código A.7: update_info.json

```
{
   "version": 0.2,
   "archivo": "https://raw.githubusercontent.com/javicano8tfg/ota/master/↩
       ↪ tfg_ota.bin",
   "fecha": "30/10/2020",
   "autor": "Javier Cano Asís"
}
```