



UNIVERSIDAD
DE MÁLAGA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INDUSTRIAL
MASTER EN INGENIERÍA MECATRÓNICA

**Infraestructura de red de sensores inalámbricos auto-
configurables para fusión de datos en el borde**

Realizado por

Daniel Rodríguez Carrión

Tutorizado por

Óscar Guillermo Plata González

Andrés Rodríguez Moreno

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE
2023

Resumen en castellano

Actualmente estamos ante una época en la que predominan los avances tecnológicos y los dispositivos electrónicos que surgen de estos. Estos dispositivos están intrínsecamente conectados a redes inalámbricas, sea WiFi, radio, bluetooth... Esto permite que los usuarios de estos sistemas puedan comunicarse de manera casi instantánea desde cualquier parte del mundo y por diferentes vías, tanto persona a dispositivo como dispositivo a dispositivo. Por otro lado, surge la necesidad de tener un mayor control, mejor precisión y tiempos de respuesta adecuados, acompañado de un nivel de automatización suficiente para satisfacer las necesidades de la actual industria 5.0. Como respuesta, en parte a estas necesidades, aparece el Internet de las cosas o IoT (“Internet of Things”). En el IoT, todos los dispositivos están interconectados y son capaces de intercambiar información y cooperar entre ellos con el fin de realizar una tarea correctamente u optimizar algún proceso, conectándose con la nube o internet en caso de requerir recursos remotos o integrarse en una red más amplia.

En el presente trabajo, se pretenda desarrollar una infraestructura que permita el despliegue de dispositivos IoT inalámbricos que se conecten automáticamente a una red de sensores. Estos dispositivos se organizarán en pequeñas redes de área personal (PAN), que les permitan compartir información localmente para integrar la información obtenida por los sensores de los dispositivos próximos y así poder procesar *in situ* esta información para fusionarla y obtener una medida mejorada de sus sensores usando la información de sus vecinos. Se evaluará el rendimiento y eficiencia de nuevos dispositivos inalámbricos de muy bajo consumo para implementar estas redes: usaremos la microarquitectura RISC-V (SoC ESP32-C3) ejecutando un sistema operativo en tiempo real (FreeRTOS) para leer los sensores, procesar la información y gestionar las comunicaciones de la red.

Palabras clave

Red de sensores, comunicaciones IoT, MQTT, ESP-NOW, RTOS, ESP32-C3, RISC-

Abstract

We are currently facing a time in which technological advances and electronic devices that arise from it predominate. These devices are fully connected to wireless networks like WiFi, radio, Bluetooth... This allows users to communicate almost instantly from anywhere in the world and by different means, both person to device and device to device. On the other hand, the need arises to want to have greater control and a sufficient level of autonomy to satisfy the needs of the current industry 5.0, arising with this need the appearance of the Internet of things or IoT (“Internet of Things”). In the IoT, all devices are interconnected and are capable of exchanging information between them in order to perform a correct task or optimize a process that requires a larger network.

That is why, in the present work, it is intended to develop an infrastructure that allows the deployment of wireless devices that connect automatically to a sensor network. The sensors will be organized in small Personal Area Networks (PANs) that allow them to share information locally to integrate the data readings from nearby devices and thus be able to process this information in situ to merge it and obtain an improved measurement from their sensors using the information from their neighbors. The performance and efficiency of new very low consumption wireless devices to implement these networks will be evaluated: we will use the RISC-V microarchitecture (SoC ESP32-C3) running a real-time operating system (FreeRTOS) to read the sensors, process the information and manage network communications.

Keywords

Sensor network, IoT Communication, MQTT, ESP-NOW, RTOS, ESP32-C3, RISC-V

Índice general

1.	Introducción y objetivos propuestos	1
1.1.	Motivación	1
1.2.	Objetivos propuestos	2
1.3.	Organización del documento.....	3
2.	Estado del arte.....	5
3.	Plataforma hardware y framework de programación.....	9
3.1.	Dispositivos y herramientas utilizadas	12
3.2.	Introducción a la placa de desarrollo ESP32-C3.....	12
3.3.	Programando con ESP-IDF y repaso de FreeRTOS	13
3.4.	Gestión de recursos usando FreeRTOS.....	16
4.	Diseño e implementación de los protocolos de comunicación	19
4.1.	Protocolo de comunicación ESP-NOW	19
4.2.	Emparejamiento automático ESP-NOW	22
5.	Diseño e implementación de otras funcionalidades del dispositivo	31
5.1.	Lectura del convertidor analógico-digital	31
5.2.	Conexión al protocolo de comunicación WiFi.....	38
5.3.	Gestión de versiones. Actualización FOTA	43
5.4.	Manejo de la memoria no volátil NVS y RTC.....	48
6.	Resultados	51
6.1.	Programación del dispositivo	51
6.2.	Análisis de los resultados obtenidos.....	56
6.2.1.	Comparación del tamaño de los binarios	56
6.2.2.	Comparación de los tiempos de ejecución y rendimiento energético	57
7.	Conclusiones	67
7.1.	Conclusiones e implicaciones del proyecto	67
7.2.	Líneas futuras	68
8.	Anexo I. Entorno de desarrollo y configuración.....	69
9.	Anexo II. Estructura de la aplicación desarrollada en ESP-IDF.....	75
10.	Bibliografía	79

Lista de figuras

Ilustración 1. Esquema del "super-loop"	7
Ilustración 2. Descarga del plugin de Espressif	9
Ilustración 3. Añadiendo el repositorio de descarga.....	10
Ilustración 4. Aceptando la descarga del repositorio.....	10
Ilustración 5. Inicio de la instalación de ESP-IDF.....	11
Ilustración 6. Configuración de la descarga de ESP-IDF	11
Ilustración 7. Dispositivo ESP32-C3	13
Ilustración 8. Diagrama de flujo de la tarea principal.....	14
Ilustración 9. Esquema de ejecución multi-tarea.	15
Ilustración 10. Gestión de tareas en sistemas FreeRTOS (source: DigiKey, Introduction to RTOS - Solution to Part 3 (Task Scheduling))	15
Ilustración 11. Estados de las tareas usando FreeRTOS (source: DigiKey, Introduction to RTOS - Solution to Part 3 (Task Scheduling)).....	16
Ilustración 12. Flujo de tareas de la rutina principal.....	17
Ilustración 13. Diagrama de flujo de la función “keep_connection_task”	17
Ilustración 14. Esquema del modelo ESP-NOW vs modelo OSI (source: eMariete, Comunicación vía radio para ESP8266 y ESP32 con ESPNOW)	20
Ilustración 15. Esquema de conexión sender-receiver en ESP-NOW (source: RANDOM NERD TUTORIALS, ESP32: ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi)).....	22
Ilustración 16. Dirección MAC.....	23
Ilustración 17. Diagrama de flujo de la tarea "mantener conexión"	24
Ilustración 18. Función de respuesta de mensaje recibido del protocolo ESP-NOW	25
Ilustración 19. Uso de semáforos para controlar el flujo de un proceso	26
Ilustración 20. Semáforo siendo creado, pero sin poder tomarlo	26
Ilustración 21. Semáforo siendo liberado	26
Ilustración 22. Semáforo siendo tomado	27
Ilustración 23. Rutina de gestión de colas	27

Ilustración 24. Creación de la cola.....	27
Ilustración 25. Envío correcto a la cola. Primer dato introducido	28
Ilustración 26. Envío incorrecto a la cola. Segundo dato introducido	28
Ilustración 27. Recepción del dato de la cola en la función "espnow_send"	28
Ilustración 28. Desplazamiento del último valor introducido en la cola	28
Ilustración 29. Función espnow_send()	29
Ilustración 30. Trama de datos en una conversión del ADC en modo DMA (source: ESP-IDF Programming Guide, Analog to Digital Converter (ADC) Continuous Mode Driver)	31
Ilustración 31. Sensor de humedad de suelo SEN0308	33
Ilustración 32. Consumo del sensor de humedad.....	34
Ilustración 33. Característica del transistor BS170.....	35
Ilustración 34. Configuración del BS170.....	35
Ilustración 35. Datos obtenidos del sensor sin circuito de protección	35
Ilustración 36. Circuito de encendido con protección anti-rebotes.....	36
Ilustración 37. Señal de los sensores con circuito de protección	36
Ilustración 38. Resultados temporales de la lectura de la sonda.....	37
Ilustración 39. Diagrama de flujo de la lectura del ADC	37
Ilustración 40. Driver WiFi.....	38
Ilustración 41. Diagrama de conexión WiFi	39
Ilustración 42. Esquema de grupos de eventos	41
Ilustración 43. Diagrama de flujo de la conexión WiFi.....	41
Ilustración 44. Rutina de control de eventos WiFi	43
Ilustración 45. Proceso de conexión a servidor para mantenimiento OTA (source: DigiKey, How to Perform Over-the-Air (OTA) Updates Using the ESP32 Microcontroller and its ESP-IDF)	44
Ilustración 46. Driver OTA.....	45
Ilustración 47. Flujo de la rutina FOTA.....	47
Ilustración 48. Tabla de particiones del ESP32-C3	48
Ilustración 49. Inicialización de la memoria flash NVS	49
Ilustración 50. Escritura en la memoria flash NVS	50

Ilustración 51. Estructura del programa principal.....	53
Ilustración 52. Autoemparejamiento realizado por medio de los datos guardados en la memoria NVS.....	55
Ilustración 53. Comprobación de mensajes y envío de datos.	55
Ilustración 54. Recepción de mensajes PAN en dos dispositivos diferentes	56
Ilustración 55. Tamaño ocupado en la compilación del programa	57
Ilustración 56. Consumo en entorno de Arduino con ESP32-C3 (1).....	58
Ilustración 57. Consumo en entorno de Arduino con ESP32-C3 (2).....	58
Ilustración 58. Recepción de mensajes MQTT con el programa de Arduino con ESP32-C3 ..	59
Ilustración 59. Consumo en entorno de Arduino con ESP32 (1).....	60
Ilustración 60. Consumo en entorno de Arduino con ESP32 (2).....	60
Ilustración 61. Recepción de mensajes MQTT con el programa de Arduino con ESP32	61
Ilustración 62. Consumo en entorno de ESP-IDF con ESP32-C3 (1)	61
Ilustración 63. Consumo en entorno de ESP-IDF con ESP32-C3 (2)	62
Ilustración 64. Recepción de mensajes MQTT con el programa de ESP-IDF con ESP32-C3.62	62
Ilustración 65. Consumo en entorno de ESP-IDF con ESP32 (1)	63
Ilustración 66. Consumo en entorno de ESP-IDF con ESP32 (2)	63
Ilustración 67. Recepción de mensajes MQTT con el programa de ESP-IDF con ESP32.....	64
Ilustración 68. (a) Validación de la imagen en ESP-IDF. (b) Continuación de la validación e inicio del programa	64
Ilustración 69. Opción para impedir que se realice la validación de la imagen al resetear en ESP-IDF	65
Ilustración 70. Entorno de programación de Eclipse C/C++	69
Ilustración 71. Selección de dispositivo a utilizar	69
Ilustración 72. Fichero raíz del proyecto.	70
Ilustración 73. Vista general del archivo "sdkconfig"	71
Ilustración 74. Pestaña de validación de conectividad HTTP para FOTA	72
Ilustración 75. Invalidación certificado TLS	72
Ilustración 76. Selección de tabla de partición personalizada.	73
Ilustración 77. Gestión de nuevo firmware tras actualización FOTA	74

Lista de tablas

Tabla 1. Características de ESP-IDF vs Arduino IDE	8
Tabla 2. Campos del byte de control	20
Tabla 3. Campos del mensaje PAN	21
Tabla 4. Tipos de eventos OTA	45
Tabla 5. Campos del mensaje PAN	54
Tabla 6. Comparativa de los dispositivos que se han sometido a prueba	65
Tabla 7. Configuración de la tabla de particiones.....	74

Agradecimientos

A mi madre y a mis hermanos, quienes me han ayudado y me han brindado su inestimable compañía en momentos difíciles.

A mis amigos y esos momentos agradables para desconectar del día a día.

A mis tutores, quienes me han guiado a lo largo de todo el desarrollo del presente trabajo

Capítulo 1

Introducción y objetivos propuestos

1.1. Motivación

Los avances que se están dando en el campo de la tecnología de la comunicación, así como en la propia electrónica y paradigmas de programación cada vez más sofisticados resulta en una nueva era en la que la presencia de la tecnología es prácticamente necesaria [1]. El ser humano, en muchos puntos de su vida cotidiana, es dependiente de la tecnología para realizar alguna labor, así como la propia tecnología depende de otras para su correcto funcionamiento. Sin pretender realizar un análisis crítico sobre este tema, no cabe duda de que esta evolución nos permite tener un mayor control sobre determinados procesos que, por su naturaleza, podrían resultar complejos. Una de las herramientas que nos ha permitido llegar a este punto álgido de industrialización ha sido la comunicación. El objetivo de comunicarse entre distintas partes del planeta a través de dispositivos tecnológicos que funcionan en tiempo real ha dado como resultado un planeta hiperconectado, en el que se tiene acceso desde prácticamente cualquier lugar en el planeta a la mayor red de datos que existe en la actualidad, Internet.

Esta evolución, vista desde los primeros ordenadores que se conectaban a Internet (Arpanet, 1983) hasta nuestros días, ha sido exponencial. Han sido muchos los avances y dispositivos que se han ido desarrollando y que, actualmente, usamos en nuestra vida diaria y que tienen la propiedad de ser programables y de conectarse a esta red tan inmensa de datos para facilitar su uso e integración con el resto de tecnologías. A esta expansión de las conexiones a Internet en objetos de uso cotidiano, más allá de ordenadores, se le nombró Internet de las Cosas o IoT [2] (en inglés, “Internet of Things”), término acuñado por primera vez en 1999 por Kevin Aston¹.

Actualmente, el Internet de las cosas está integrado prácticamente a todos los niveles de nuestra vida cotidiana, desde la industria o agricultura hasta el día a día de las personas que habiten en regiones tecnológicamente avanzadas. Debido a esto, han ido surgiendo multitud de tecnologías y dispositivos que permiten a los usuarios introducirse de una forma más familiar a este mundo y comprender su funcionamiento. Este enfoque didáctico permite, además, adaptar las necesidades de cada individuo o compañía a un proyecto casero o industrial con plataformas de desarrollo como Arduino, Espressif o Raspberry Pi. Estos dispositivos permiten incluir a los usuarios al mundo de Internet de una forma más asequible y profesional gracias a su bajo coste y a las posibilidades que tienen.

¹ <https://www.visionofhumanity.org/what-is-the-internet-of-things/>

El presente trabajo permite analizar la inclusión de estas tecnologías emergentes que están al alcance de los usuarios gracias, en parte, a la accesibilidad y al bajo coste que tienen estos en el mercado. Además, en algunos casos la red de usuarios es tan amplia que permite mejorar el desarrollo de herramientas o sistemas entre la comunidad y favorecer así el crecimiento y la expansión del IoT. La creación de librerías, clases personalizadas, estructuras propias son algunos de los aspectos que permiten indicar que el mundo del Internet de las Cosas está en constante expansión.

1.2. Objetivos propuestos

Como parte del itinerario del máster en ingeniería mecatrónica, el presente trabajo se encuadra en el último proyecto en aras de conseguir el título del presente curso. El trabajo de fin de máster, en adelante, TFM, debe de cumplir con unos estándares de investigación, diseño, implementación y evaluación en todos los puntos necesarios para satisfacer en última instancia la calidad exigida en el presente trabajo. Por esto, se ha dividido el trabajo que se realizará a lo largo del TFM en partes independientes y con peso propio. Por todo esto y teniendo en cuenta que el TFM ocupa una carga lectiva de 10ETCs o 250 horas, se propone la siguiente metodología de trabajo:

Como punto de partida, se hará una investigación previa del estado del arte en el que nos encontramos con respecto al presente trabajo. Esto es, estudiar las capacidades y procedimientos varios que engloban al ESP32-C3 [3] [6] y su nueva arquitectura, el RISC-V, la cual pertenece a un proyecto que comenzó en el año 2010 por la Universidad de California, en Berkeley y desarrollado posteriormente por voluntarios y trabajadores de la industria fuera de la propia universidad. Aquí se pretende estudiar, sobre todo, la posibilidad de control en tiempo real con FreeRTOS [5] [7] [8] [9] de aplicaciones varias utilizando el nuevo microcontrolador y el entorno de programación Eclipse [4].

Tras el estudio preliminar, se pretende diseñar un algoritmo que se conecte al protocolo de comunicación ESP-NOW y WiFi, procese y envíe datos a una pasarela que para luego publicarlo por MQTT y estudiarlo. Debe de funcionar en tiempo real, utilizando para ello FreeRTOS. Dicho algoritmo pretende que sea lo más rápido posible para, no solo dotarlo de eficiencia computacional, sino que sea lo más energéticamente eficiente posible. La dinámica que se pretende obtener es la siguiente:

1. Inicio de una rutina de autoemparejamiento usando protocolo ESP-NOW a una pasarela maestra que está conectada a un servidor WEB y es capaz de enviar/recibir por MQTT datos desde y hacia dicho servidor.
2. Obtención de datos y posterior procesado.
3. Envío de los datos procesados a la pasarela maestra.
4. Apagado del dispositivo para ahorro energético.

Además, debe de contar con varios extras necesarios para un funcionamiento óptimo, como son:

- ✓ Posibilidad de cambiar parámetros como el tiempo en el que el dispositivo está en “Deep Sleep” o sueño profundo, para ahorro energético.
- ✓ Posibilidad de actualización FOTA [23] (“Firmware Over The Air”)
- ✓ Posibilidad de almacenamiento en memoria RTC/NVS [24] [25] [26] de parámetros de conexión varios, como la MAC o datos de configuración, así como datos obtenidos durante la ejecución del código.

Por ello, entre la investigación del estado del arte previo al trabajo, el diseño preliminar del algoritmo de conexión, procesamiento y envío y, finalmente, el diseño final del algoritmo de procesamiento y envío y la implementación física en el microcontrolador ESP32-C3 se pretende cumplir el cupo de las 250 horas de trabajo. Además, para cerrar el círculo del TFM, se debe de comprobar y corroborar el correcto funcionamiento del entorno, por lo que se computan las horas de ensayos en el laboratorio y en condiciones reales, para poder extraer conclusiones.

1.3. Organización del documento

Este documento estará organizado en diferentes capítulos para facilitar su seguimiento y lectura:

- **Capítulo 1: Introducción y objetivos propuestos.** En este primer capítulo se realiza una presentación de las motivaciones más importantes del proyecto, así como la introducción, los objetivos que se pretenden alcanzar en el mismo y la organización del documento.
- **Capítulo 2: Estado del arte.** En este segundo capítulo se detalla el estado actual de los paradigmas de programación que existen en microcontroladores (“super-loop” y RTOS) que se van a utilizar, así como los dispositivos que se van a utilizar. Enfocamos este segundo capítulo en abordar algunas diferencias entre estos paradigmas.
- **Capítulo 3: Entorno de desarrollo y herramientas de programación.** En cuanto al entorno de desarrollo y las herramientas que se han usado para programar, así como los aspectos físicos para realizar el proyecto, se verán en este capítulo. Se hará una pequeña introducción al entorno en el que se ha desarrollado la aplicación y se expondrán los elementos que se han utilizado en este proyecto.
- **Capítulo 4: Marco teórico del proyecto.** En este capítulo se desarrollará el marco teórico de las aplicaciones diseñadas y utilizadas a lo largo del proyecto. Se expande la idea de programación en tiempo real y se enseña al lector las APIs utilizadas para el funcionamiento del proyecto.
- **Capítulo 5: Análisis de los resultados obtenidos.** En esta sección se analiza la aplicación diseñada y se comentarán los resultados obtenidos. Además, se comparará

Capítulo 1 - Introducción y objetivos propuestos

con otros entornos para estudiar el rendimiento real que tiene esta herramienta novedosa.

- **Capítulo 6: Conclusiones.** Finalmente, se propone un resumen del trabajo realizado y se comentarán las implicaciones del mismo y líneas futuras del trabajo.
- **Capítulo 7: Anexos.** Se incluye una serie de anexos al documento para detallar aspectos relevantes del proyecto. El primero de ellos corresponde al entorno de desarrollo y su configuración y un segundo anexo relativo a la estructura que se ha aplicado en ESP-IDF al proyecto.

Capítulo 2

Estado del arte

Los cambios que ha habido en la última década en el campo del internet de las cosas (IoT) han conseguido que dispositivos sean dotados con sensores, actuadores y capacidades para conectarse a la red. La atención del campo ahora se está desplazando hacia las aplicaciones que, debido en parte a problemas relacionados con la escala de las implementaciones de redes de sensores complica el análisis de grandes cantidades de datos. Los datos recopilados, a menudo, son analizados para encontrar información específica en un momento determinado y que resulte significativo para actuar en consecuencia. Por ejemplo, los datos de los sensores corporales podrían analizarse para proporcionar alertas sanitarias tempranas. A menudo, tales patrones o eventos están fuera de lo común o son anómalos.

La detección de anomalías se puede definir como la detección de eventos, comportamientos o patrones inesperados en relación con un concepto de lo que es normal. Este enfoque, por ejemplo, se está usando para detectar intrusiones en los sistemas de información, algo cada vez más relevante en la computación en la nube actual. Los enfoques de detección de anomalías son populares en aplicaciones con grandes instalaciones de procesamiento y almacenamiento central, como las que se emplean para procesar *big data*. Sin embargo, su aplicación a sistemas livianos, como las WSN, aún está limitada debido a las severas limitaciones de recursos que plantean estos sistemas. Algunas limitaciones como la memoria, los altos costos de comunicación o el propio rendimiento de los dispositivos impiden el escenario en el que todos los nodos WSN envían toda la información a una instalación central para su almacenamiento y procesamiento. En cuanto a esta estudiar varios enfoques para mejorar estas limitaciones [27], se han propuesto métodos en entornos simulados en los que los resultados han sido realmente favorables en cuanto a rendimiento y operatividad se refiere. En este artículo se plantea la localidad como una característica más del sistema. Los datos locales se contrastan con la media recortada de los datos vecinos y se corrigen por la estabilidad de la vecindad, similar a la varianza. En este sentido, la noción de localidad puede intercambiarse entre espacio de datos y espacio geográfico. Por esto, realizar un estudio para la detección de anomalías de manera local es una mejora importante para los métodos de optimización existentes, ya que la mayoría de métodos apuntan a bases de datos de sistemas de información geográfica (SIG) con datos estacionarios y no a base de datos temporales. Aplicar estas técnicas a un sistema WSN no es trivial, debido al cálculo relativamente alto y al alto coste de comunicación que implicaría. Es por esto, en el que se debe de pensar en una implementación en un entorno real para así comprobar y aplicar esta filosofía.

La inclusión de entornos de programación más sofisticados, así como paradigmas de programación en tiempo real fuerzan al usuario a buscar alternativas que permitan mayor grado

de eficiencia, controlabilidad y robustez. Cuando hablamos de ESP-IDF (Espressif IoT Development Framework), hacemos referencia al framework oficial de desarrollo de la familia de microcontroladores de Espressif en asuntos relacionados con IoT o “Internet de las cosas”. Este framework oficial proporciona un gran conjunto de librerías, herramientas y APIs o “Application Programming Interfaces” para el desarrollo de los microcontroladores ESP32. Ofrece una amplia variedad de características que permiten el desarrollo de aplicaciones IoT, amén de proporcionar una plataforma flexible para su diseño.

Una de las principales ventajas del framework ESP-IDF es que soporta de manera nativa C/C++ como lenguajes de programación, lo cual permite una programación más eficiente y código de alto rendimiento. Además, ESP-IDF proporciona un conjunto de características y capacidades para crear aplicaciones de IoT, como:

- Soporte para conectividad Wi-Fi y Bluetooth
- Amplio conjunto de controladores para varios sensores, periféricos y protocolos de comunicación
- Compatibilidad con actualizaciones inalámbricas (OTA) y arranque seguro
- Compatibilidad con FreeRTOS, un sistema operativo en tiempo real para microcontroladores

Sin embargo, es bien sabido que la comunidad de ingenieros y programadores de esta familia de microcontroladores usan, de manera frecuente, el IDE de Arduino, el cual proporciona también un kit de desarrollo de software (SDK) diseñado para simplificar el proceso de programación de los microcontroladores ESP32 y ESP8266 utilizando la plataforma, el entorno de programación Arduino y una gran variedad de librerías asociadas.

La plataforma Arduino se ha vuelto popular por su facilidad de uso y su interfaz amigable para principiantes. Y ESP Arduino Core trae estas ventajas a los microcontroladores ESP32 y ESP8266. Por mencionar un par de ellas:

1. Permite utilizar el mismo lenguaje de programación y entorno de desarrollo de Arduino, incluido el IDE de Arduino, para escribir código para los microcontroladores ESP32 y ESP8266. Esto facilita que los principiantes y los aficionados comiencen a usar estos microcontroladores, especialmente si usó Arduino antes, sin tener que aprender un nuevo conjunto de herramientas.
2. ESP Arduino Core incluye una variedad de bibliotecas y API que facilitan el trabajo con el hardware de ESP32 y ESP8266, una variedad de sensores y protocolos de comunicación. Estas bibliotecas cubren todo, desde operaciones básicas de entrada/salida hasta funciones más avanzadas como WiFi o comunicación y criptografía Bluetooth.

Sin embargo, se debe recordar que la implementación de Arduino de los lenguajes de programación C/C++ utilizados en Arduino no es la versión completa del lenguaje. La

plataforma Arduino utiliza una versión simplificada del lenguaje C/C++ que se adapta específicamente a las necesidades de los proyectos basados en microcontroladores, utilizando un modelo de programación que se conoce como “super-loop”.

Cuando el programa arranca, ejecuta la función `setup()`, que se utiliza para realizar los ajustes iniciales necesarios para la lógica de nuestro programa y la inicialización habitual para el correcto funcionamiento de las librerías añadidas. Justamente después, se ejecutan las tareas de control en un bucle infinito llamando a la función `loop()`, donde se planifican secuencialmente las tareas que se ejecutan en el orden preestablecido en su aparición dentro de la función `loop()`, lo que condensa el tiempo de respuesta de cada tarea al tiempo de ejecución agregado del resto de tareas del bucle. Tiene como ventaja de que es un mecanismo de ejecución de tareas sencillo que no requiere un sistema operativo que realice la planificación.

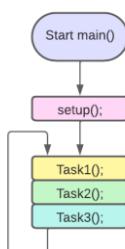


Ilustración 1. Esquema del "super-loop"

Esto a veces puede limitar la funcionalidad y la flexibilidad del código que se puede escribir. Si se quisiera cumplir unos requerimientos temporales estrictos para una o varias tareas, se debe de plantear otro esquema de programación que permita más maniobrabilidad. Una desventaja del sistema de gestión de tareas implementado en Arduino es que no se pueden ejecutar tareas de forma concurrente si al menos una de ellas lleva un tiempo de ejecución alto.

Por el contrario, ESP-IDF proporciona soporte de tiempo real gracias a una implementación ampliada de FreeRTOS, un sistema operativo en tiempo real para microcontroladores que permite a los desarrolladores programar tareas, establecer prioridades, utilizar temporizadores, colas de mensajes y otros mecanismos de sincronización. Además, la versión que implementa Espressif permite la programación multi-core, una ampliación sobre la versión Vanilla FreeRTOS en la que se basa.

Llegados a este punto, surge la pregunta de qué sistema de programación (*super-loop vs RTOS*) se debe de escoger para programar un dispositivo de la familia ESP32 o ESP8266 [10]. Cuando se trata de elegir entre ESP-IDF y ESP Arduino Core, hay algunas diferencias clave a tener en cuenta.

Como ya se mencionó, el ESP-IDF es compatible con los lenguajes de programación completos C/C++, lo que permite aprovechar al máximo las características que ofrece C/C++.

Si bien la plataforma Arduino también es compatible con C/C++, vale la pena señalar que no es una implementación completa de estos lenguajes. Además, ESP-IDF ofrece una variedad de

herramientas y funciones diseñadas específicamente para el desarrollo de IoT con microcontroladores ESP32 y ESP8266, incluidas las actualizaciones de firmware OTA (en inglés, “Over-the-Air”). Aunque las actualizaciones OTA también son posibles con ESP Arduino Core, el proceso no es tan sencillo como con ESP-IDF.

La principal ventaja de ESP IDF es que admite nuevas versiones de microcontroladores y funciones ESP32. Como Espressif fabrica ESP-IDF exactamente para los microcontroladores ESP32 y ESP8266, cuando lanzan una nueva versión de ESP32, se aseguran de que sea totalmente compatible con ESP-IDF Framework. Por ejemplo, actualmente ESP32-C5 y ESP32-C6 no son oficialmente compatibles con ESP Arduino Core y ya pueden generar problemas, como sucede si se necesita que un proyecto use WiFi de 5 GHz. En general, el ESP-IDF tiene varias características que no están disponibles o tienen un peor soporte en Arduino Core, que incluyen:

- Soporte nativo para FreeRTOS
- Herramientas de depuración y administración de memoria más eficientes
- Mejor soporte para CPU multinúcleo
- Actualizaciones más frecuentes y adopción más rápida de nuevas versiones de microcontroladores ESP
- Compatibilidad con funciones de hardware más avanzadas, como Bluetooth, Wi-Fi y modos de bajo consumo

ESP-IDF es una opción más poderosa y flexible para usuarios y proyectos que requieren características de hardware más avanzadas, una gestión de recursos más eficiente, se quieran usar las versiones más recientes de ESP32 y aprovechar al máximo el microcontrolador.

Tabla 1. Características de ESP-IDF vs Arduino IDE

ESP-IDF	Arduino Core
✓ Soporte nativo FreeRTOS	✗ Soporte FreeRTOS limitado
✓ Aplicaciones basadas en tareas	✗ Funciones setup() y loop()
✓ Multi-core por defecto	✗ Single-core por defecto
✓ Soporte para lanzamientos nuevos ESP32	✗ Soporte limitado para lanzamientos nuevos ESP32
✗ Menos amigables para principiantes	✓ Amigable para principiantes
✗ Comunidad reducida	✓ Comunidad muy amplia

Capítulo 3

Plataforma hardware y framework de programación

Una vez visto y analizado los puntos positivos de utilizar el SDK de Espressif, pasemos a repasar el entorno donde se va a desarrollar las diferentes aplicaciones en cuestión. Se utilizará el entorno de programación de Eclipse en su versión para C/C++. El propio IDE trae un plugin propio de Espressif que permite a los desarrolladores de código un uso fácil para el desarrollo de aplicaciones IoT basadas en el ESP32. Permite, entre otras cosas, herramientas más sofisticadas que simplifican y mejoran el desarrollo y la depuración de dichas aplicaciones. Se hará un breve repaso de cómo se procede a la instalación del IDE Eclipse:

- Como requisito fundamental, puesto que el IDE está diseñado y programado en Java, será tener instalado en nuestra computadora la última versión de Java (17 o superior, según Espressif) [12].
- Otro requisito parte de la propia arquitectura de ESP-IDF, puesto que corre script en Python. Por esto es recomendable tener una versión actualizada de Python [11].
- Finalmente, la versión más reciente de Eclipse IDE [13] para C/C++, donde programaremos el ESP32.

Para configurar el entorno de programación y añadirle el plugin del SDK de Espressif, hay que buscar en la barra de tareas el comando “Help” ubicado en la esquina superior derecha del entorno. Una vez lo seleccionemos, tenemos que buscar la opción “Install new software” para poder buscar el plugin necesario para programarlo en este entorno.

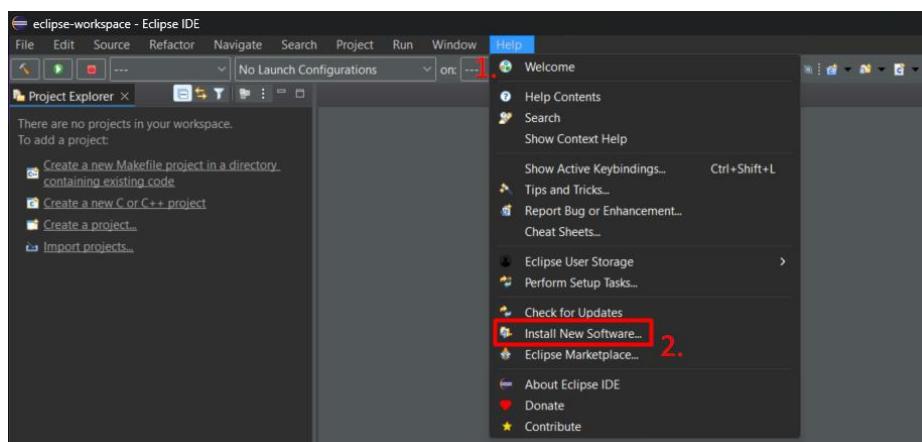


Ilustración 2. Descarga del plugin de Espressif

Capítulo 3 - Plataforma hardware y framework de programación

Una vez seleccionemos esta opción, nos aparecerá un recuadro similar al que se muestra en la siguiente ilustración, donde tendremos que llenar algunos campos:

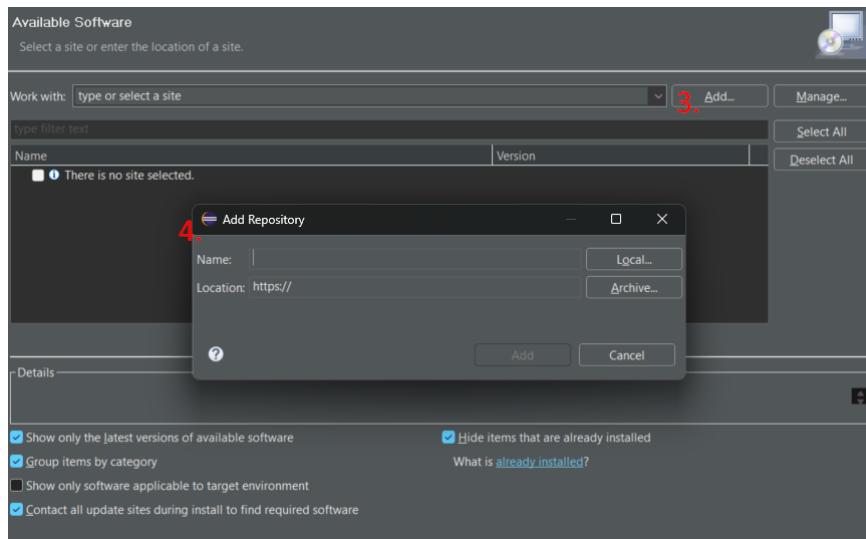


Ilustración 3. Añadiendo el repositorio de descarga.

- En el campo “Name” hay que añadir el nombre del SDK que vamos a instalar. En nuestro caso, “Espressif IDF Plugin for Eclipse”.
- En el campo “Location”, la URL para descargar los paquetes. Espressif nos da varios canales donde lanza versiones estables, betas y en desarrollo. Siempre se recomienda la versión estable, a no ser que se quiera desarrollar algún paquete específico.

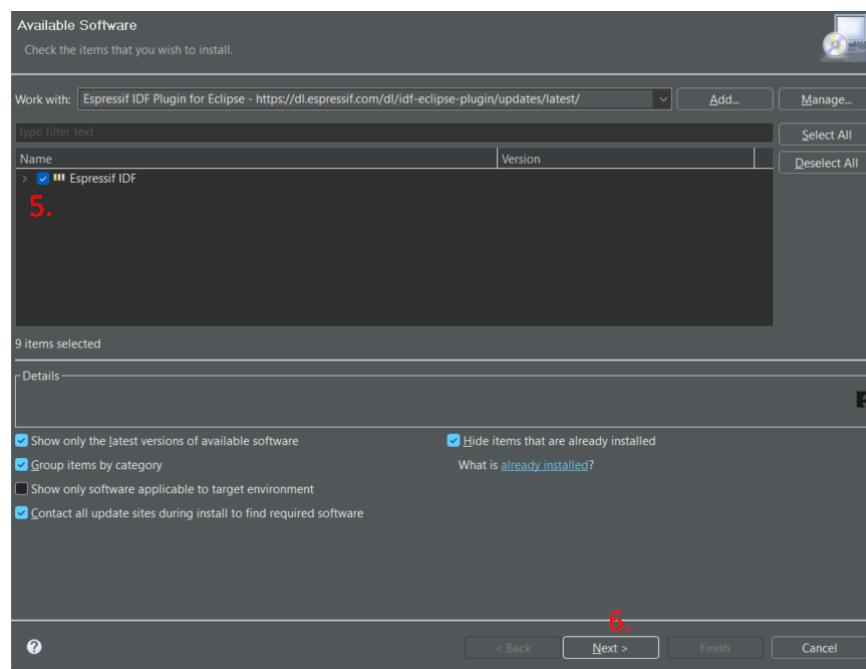


Ilustración 4. Aceptando la descarga del repositorio.

A partir de aquí, restaría aceptar los cuadros de diálogo que irán saliendo y aceptando permisos de confianza de los paquetes que son de terceros. Se reiniciará el IDE Eclipse. Una vez instalado el plugin, bajaremos ESP-IDF, que es una carpeta con todas las librerías, API's y ejemplos que nos da Espressif. Para ello, seguiremos los siguientes pasos:

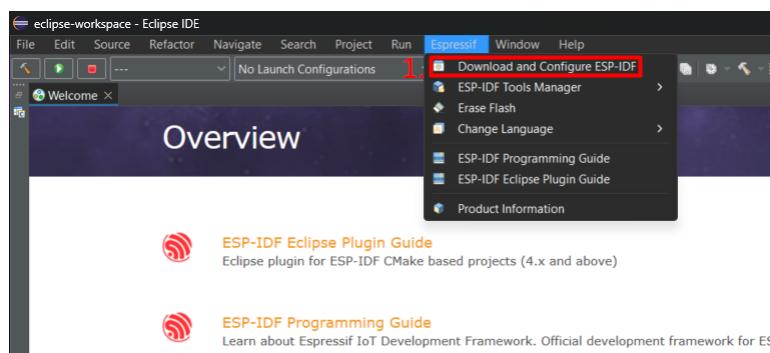


Ilustración 5. Inicio de la instalación de ESP-IDF

De nuevo en la barra de tareas nos saldrá la pestaña “Espressif”. De no salir, se tendrá que reiniciar el IDE Eclipse. Una vez seleccionada dicha pestaña, hay que irse a la opción “Download and Configure ESP-IDF”.

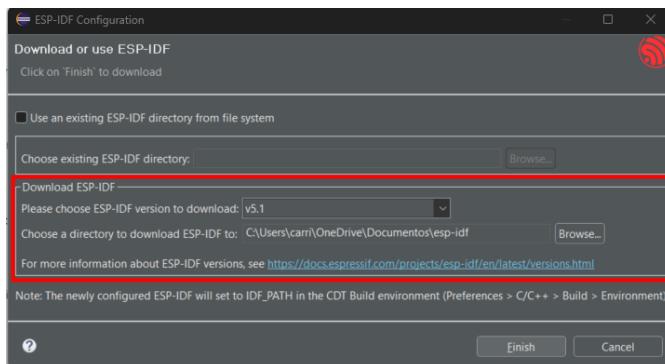


Ilustración 6. Configuración de la descarga de ESP-IDF

Si se ha descargado previamente algún directorio de ESP-IDF, se seleccionará la opción “Use an existing ESP-IDF directory from file System” que de manera automática nos lo iniciará e instalará en el IDE. Si se prefiere otras versiones de ESP-IDF, descargarla directamente seleccionando el directorio de descarga. Es posible que de manera automática nos indiquen si queremos que se instalen una herramienta del entorno ESP-IDF útiles para desarrollo y depuración y que, como es lógico, es de gran interés. En cualquier caso, si el usuario se va a la pestaña de “Espressif” y selecciona “ESP-IDF” Tools Manager” podrá instalarlo.

Una vez hecho todo lo anterior, el usuario ya está listo para poder programar con el IDE Eclipse cualquier aplicación que desee utilizando el SDK oficial de Espressif. En el Anexo I se pueden encontrar detalles de la configuración del SDK oficial de Espressif en el entorno de desarrollo de Eclipse.

3.1. Dispositivos y herramientas utilizadas

Más allá del entorno en el que se programe, se ha requerido de los siguientes elementos para poder realizar el trabajo de forma adecuada:

- Placas de desarrollo ESP32-C3 con procesador RISC-V como dispositivo cliente.
- Placa de desarrollo ESP32-WROOM como dispositivo de pasarela.
- Raspberry Pi 3 como dispositivo servidor.
- Baterías, placas solares, reguladores, componentes electrónicos pasivos.
- Multímetro digital.
- Sensor de humedad capacitivo SEN0308.

Además, como herramientas de desarrollo tanto para el análisis como para la implementación final han sido las siguientes:

- Framework de RTOS de Espressif para ESP32.
- Entorno de programación Eclipse/VSCode.
- MATLAB como parte para procesar los datos.
- Para la gestión de versiones: GitHub.
- Para la implementación de la aplicación IoT que permite gestionar y comunicarnos con los dispositivos IoT se ha utilizado NodeRED alojado en una Raspberry Pi.

3.2. Introducción a la placa de desarrollo ESP32-C3

Este proyecto está motivado a evaluar, entre otras muchas cosas, el comportamiento y el rendimiento del dispositivo ESP32-C3, que trae embebido un microcontrolador RISC-V. Este dispositivo, más pequeño que el tamaño de una moneda, cuenta con unas características realmente excepcionales, que lo convierten en un dispositivo realmente potente dentro de la familia de los ESP32.

Posee hasta 13 pines de entrada y salida que, sumado a su pequeño tamaño, lo convierte en un dispositivo de lo más conveniente para proyectos integrados de un tamaño reducido. Trae integrado un gestor de carga para baterías del tipo ion de litio, lo que permite conectar la batería directamente al dispositivo sin una interfaz de carga extra, satisfaciendo con la misma requisitos de seguridad y tamaño.

Como todos los dispositivos de la familia ESP32, su fuerte está en la conectividad. Este dispositivo soporta WiFi y Bluetooth 5 *Low Energy* (LE) en modo dual que reduce la dificultad de las conexiones y permitiendo conexiones incluso más estables.



Ilustración 7. Dispositivo ESP32-C3

El punto fuerte del dispositivo es su procesador, un RISC-V de 32 bits *Single-core*, diseñado pensando en implementaciones pequeñas, rápidas y de bajo consumo para el mundo real. Es un procesador que posee una frecuencia principal de 160MHz.

En el campo de la memoria tampoco se queda atrás, a pesar de su pequeño tamaño.

- ➔ Una memoria SRAM de 400KB
- ➔ Una memoria ROM de 384KB
- ➔ Una memoria Flash de 4MB
- ➔ Una memoria RTC SRAM de 8KB

Por otro lado, cuenta con una serie de puertos y periféricos que le permiten realizar una amplia variedad de aplicaciones:

- ➔ Posee hasta 13 puertos de entrada y salida
- ➔ Hasta 6 canales PWM configurables
- ➔ Un canal SPI, I2C, I2S y hasta dos UART
- ➔ Posee un transceptor infrarrojo (emisor/receptor) en hasta dos canales
- ➔ Controlador DMA con hasta 3 canales para transmitir y otros 3 canales para recibir
- ➔ Dos conversores analógicos ADC del tipo SAR con hasta 6 canales diferentes

La elección de este dispositivo en lugar de otros de la misma familia, incluso otros que puedan utilizar hasta un *dual-core* como el ESP32 (con un alto coste energético), recae en la rapidez de la nueva arquitectura y su eficiencia energética.

3.3. Programando con ESP-IDF y repaso de FreeRTOS

Como se comentó al principio, una de las características que un usuario novel con este entorno se puede encontrar a priori es la falta de información relativa a ESP-IDF que existe ya que, por lo general, se prefiere utilizar otros entornos más amigables de cara a un uso menos sofisticado.

Sin embargo, la propia plataforma de desarrollo nos ofrece una amplia variedad de ejemplos bastante útiles que nos permiten, amén de información de gran importancia en su propia plataforma online, programar utilizando este kit de desarrollo.

A pesar de que gran parte está programado en C, así como todas sus APIs, también nos permite programar utilizando C++. Por otro lado, la inclusión de las “Application Programming Interfaces” nos brinda la posibilidad de incrementar el nivel en la etapa de desarrollo, al tener más capacidades y libertad a la hora de programar un dispositivo de la familia ESP32. El presente trabajo se encuentra programado en gran medida en C.

Como cualquier programa escrito en C/C++, lo primero que nos encontramos en una función general donde se incluyen las inicializaciones y las funciones que el usuario quiera utilizar en su rutina. Esta función, habitualmente llamada “main”, establece el punto de partida para la ejecución de cualquier programa.

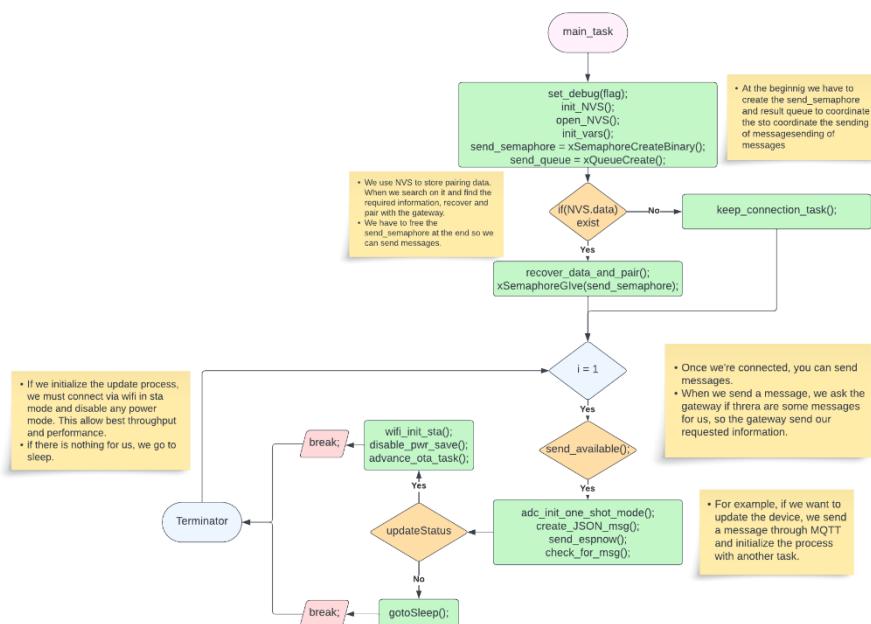


Ilustración 8. Diagrama de flujo de la tarea principal

En el entorno del SDK de Espressif, dicha función se llama “app_main” y constituye la función que cumple con un propósito idéntico a la función “main”. Como se vio en la tabla 1 de comparación entre el SDK que nos proporciona Espressif y la plataforma de Arduino, tenemos que en la segunda aparecen las funciones “setup()” y “loop()” como función de inicialización o ejecución única y de ejecución repetitiva. En el otro caso, la rutina “app_main” trabaja como una tarea creada por la CPU que previamente ha sido cargada en la tabla de particiones desde la memoria flash y que es gestionada por el gestor de tareas RTOS, “Real Time Operation System” o Sistema operativo en tiempo real. Este aspecto resulta fundamental, puesto que nos permite controlar tareas o procesos en tiempo real utilizando FreeRTOS.

La rutina “app_main” parte de tener una prioridad fija, una mayor que la mínima y una pila con

tamaño configurable. Además, permite volver a la aplicación en sí. De pasar, la pila se limpia y el sistema continuará corriendo otra tarea gestionada por RTOS. Gracias a esta funcionalidad, podemos realizar rutinas que se ejecuten de forma síncrona, con prioridad y en tiempo real.

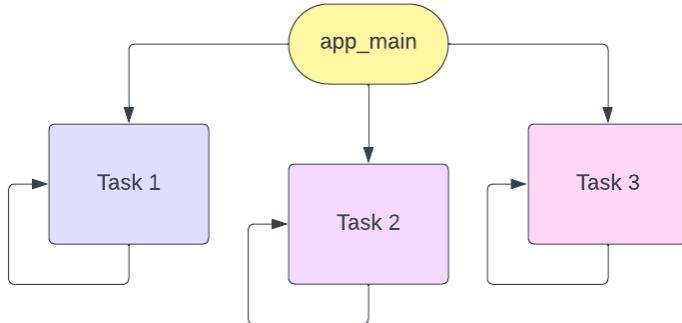


Ilustración 9. Esquema de ejecución multi-tarea.

Al paradigma de ejecución multitarea se le suele denominar “multithread” o “multitask”, ya que se lanzan procesos o “hilos” de forma paralela o secuencial al programa principal, en función del número de núcleos del que disponga la CPU.

Cada tarea se ejecuta de manera concurrente en su propia rutina de ejecución infinita, bien con un bucle “while” o “for”, si se desea que dicha tarea no se acabe nunca o bien suspendiéndola o terminando dicha tarea cuando se alcance una determinada condición. En sistema de un solo núcleo o “single-core”, la CPU debe de dividir las tareas en espacios temporales de tal modo que no se solapen y que parezca que se ejecutan de forma concurrente.

El gestor de tareas de un sistema operativo en tiempo real debe de asegurar qué tipo de tarea corre en cada sección de tiempo. Por ejemplo, asumiendo un solo núcleo en la CPU:

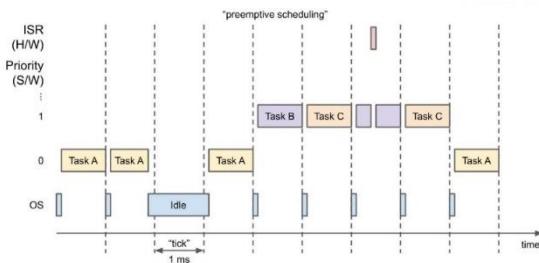


Ilustración 10. Gestión de tareas en sistemas FreeRTOS (source: DigiKey, Introduction to RTOS - Solution to Part 3 (Task Scheduling))

En FreeRTOS, el espacio temporal por defecto es conocido como “tick” y corresponde a 1ms.

En cada “tick”, el gestor de tareas decide qué tarea ejecutar en función de su prioridad. Si las tareas tienen la misma prioridad, se ejecutan por turnos, “round-robin fashion”. Si una tarea con una prioridad más alta que la tarea que se está ejecutando, se ejecutará inmediatamente sin esperar al siguiente “tick”. No obstante, cuando creamos nuestras tareas, podemos asignarles

diferentes prioridades e, incluso, cambiar de prioridad con la función “vTaskPrioritySet()”. Siguiendo este paradigma de programación, podemos encontrarnos en uno de estos cuatro estados:

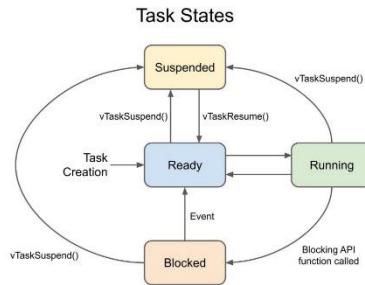


Ilustración 11. Estados de las tareas usando FreeRTOS (source: DigiKey, Introduction to RTOS - Solution to Part 3 (Task Scheduling))

Cuando se crea una tarea, entra en el estado Listo (“Ready-state”) e indica al gestor de tareas que está lista para arrancar. En cada “tick”, el gestor escoge una tarea para correr y que esté en este primer estado. Mientras la tarea está funcionando, está en el estado Corriendo (“Running-state”) y puede regresar al estado “Listo” por el propio gestor.

Las funciones que hacen que la tarea espere, como “vTaskDelay()”, colocan la tarea en el estado Bloqueado (“Blocked-state”). Aquí, la tarea está esperando que ocurra algún otro evento, como que expire el temporizador en “vTaskDelay()”. La tarea también puede estar esperando que otra tarea libere algún recurso, como un semáforo o una cola. Las tareas en estado Bloqueado permiten que se ejecuten otras tareas en su lugar.

Finalmente, una llamada explícita a “vTaskSuspend()” puede poner una tarea en el estado Suspensión (“Suspended-state”) (muy parecido a poner esa tarea en suspensión). Cualquier tarea puede poner cualquier tarea (incluida ella misma) en este estado. Una tarea solo puede regresar al estado Listo mediante una llamada explícita a “vTaskResume()” por parte de otra tarea.

3.4. Gestión de recursos usando FreeRTOS

La ventaja de utilizar un paradigma de programación basado en tareas, es que permite distribuir mejor la carga de trabajo entre los distintos núcleos que se dispongan o bien realizar los procesos de forma más rápida y eficiente que de manera concurrente, además de estar ejecutándose en tiempo real. Proporciona métodos para múltiples subprocesos o hilos, como mutexes, semáforos y temporizadores software, además de admitir prioridades de hilos. Entre sus principales ventajas nos encontramos las siguientes:

- La huella en memoria es pequeña.
- Gasto indirecto bajo

- Ejecución rápida y de bajo consumo
- Control total de la aplicación, así como compatibilidad con correntinas.
- Permite el seguimiento a través de macros de rastreo genérico para depuración.

Enfocándonos en este trabajo, se han utilizado una gran variedad de elementos para coordinar los procesos necesarios para su correcto funcionamiento.

En la siguiente ilustración se observa, por ejemplo, la tarea que se pretende lanzar para gestionar la conexión automática a la pasarela que se encarga de gestionar los mensajes ESP-NOW a MQTT:

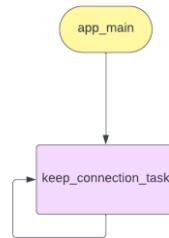


Ilustración 12. Flujo de tareas de la rutina principal

Esta tarea tendría el siguiente flujo de funcionamiento:

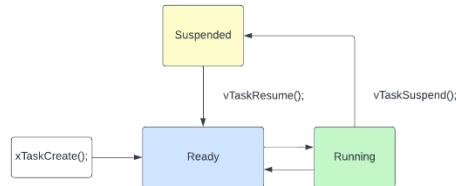


Ilustración 13. Diagrama de flujo de la función "keep_connection_task"

Analizando este diagrama, tenemos como punto inicial la creación de la tarea con la función “`xTaskCreate()`”. Esta función debe de recibir como parámetros lo siguiente:

- Código de la tarea “`pvTaskCode`”: puntero a la entrada de la función.
- Nombre de la tarea “`pcName`”: nombre descriptivo para procesos de depuración posteriores.
- Número de palabras para almacenar (como pila de la tarea) “`usStackDepth`”
- Puntero a parámetros para pasarle a la tarea “`pvParameters`”
- Prioridad de ejecución de la tarea “`uxPriority`”
- Variable para pasar la tarea creada fuera de la propia tarea “`pxCreatedTask`”

Cuando se ha logrado el objetivo de conectarse a la pasarela, se suspenderá la tarea en cuestión.

Capítulo 3 - Plataforma hardware y framework de programación

Capítulo 4

Diseño e implementación de los protocolos de comunicación

Se pretende desarrollar una infraestructura que permita el despliegue de dispositivos IoT inalámbricos que se conecten automáticamente a una red de sensores. Estos dispositivos se organizarán en pequeñas redes de área personal (PAN), que les permitan compartir información localmente para integrar la información obtenida por los sensores de los dispositivos próximos y así poder procesar *in situ* esta información para fusionarla y obtener una medida mejorada de sus sensores usando la información de sus vecinos.

En este capítulo se estudian las diferentes posibilidades que nos brinda el nuevo framework de ESP-IDF, así como un análisis de las APIs que incluye y la metodología RTOS para programación en tiempo real.

Sin embargo, antes de explicar este proceso, haremos una breve explicación de qué es ESP-NOW, para quien no esté familiarizado con dicho protocolo de comunicación.

4.1. Protocolo de comunicación ESP-NOW

El protocolo de comunicación ESP-NOW es, en esencia, un protocolo de comunicación vía radio que transmite en la banda de los 2.4GHz que permite conectar múltiples dispositivos mientras usa elementos del protocolo WiFi sin necesidad de un router para formar una red. Gracias a esto, permite realizar proyectos que requieran un muy bajo coste energético, aumentando en gran medida su autonomía y la latencia de datos, que aumenta de los 3-12 segundos a unos pocos milisegundos.

Este protocolo, completamente configurable en dispositivos de la familia ESP (ESP8266 y ESP32), permite además aumentar la distancia a la que se transmiten los datos, obteniendo resultados de transmisión prometedores a 500 metros de alcance, frente a los pocos metros que permite una red WiFi. Con respecto a la autonomía cabe destacar que, para conexiones WiFi, los dispositivos de la familia ESP gastan bastante tiempo en realizar la conexión, alrededor de unos pocos segundos, además de que en gasto energético el protocolo WiFi requiere de bastante energía para funcionar, puesto que consume entre 80 y 180mA [15]. Para una aplicación en la que se requiera un consumo menor para aumentar la autonomía, hay que prescindir de este protocolo. Al menos en parte.

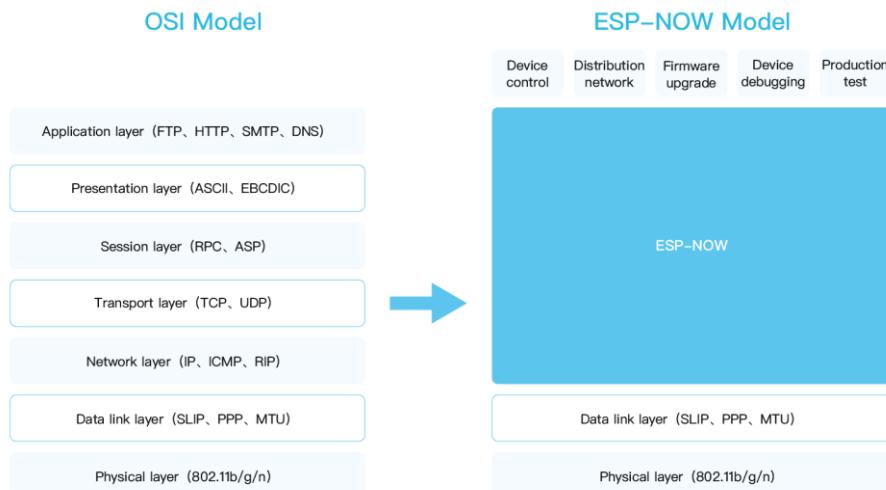


Ilustración 14. Esquema del modelo ESP-NOW vs modelo OSI (source: eMariete, Comunicación vía radio para ESP8266 y ESP32 con ESPNOW)

La eficiencia del protocolo ESP-NOW viene a que es un protocolo que no necesita una conexión como tal. A diferencia de los protocolos WiFi tradicionales, las cinco primeras capas superiores de OSI se simplifican en una sola capa en ESP-NOW, así los datos no tienen que pasar por una capa física, de enlace de datos, de red y la capa de transporte. Esto reduce significativamente el retraso causado por la pérdida de paquetes en una red que estuviera congestionada y permite un tiempo de respuesta más rápido. Para ello, es fundamental que los dispositivos que quieran enviar y recibir datos, estén conectados al mismo canal de radio, en el que se trasmitirá la dirección MAC que servirá de identificación inequívoca y los datos enviados o recibidos. Es por esto que se suele hablar de dos roles definidos en este protocolo según el flujo de datos, el dispositivo que envía los datos (“sender”) y el que los recibe (“receiver”), permitiendo además que el mismo dispositivo pueda trabajar con esos dos roles de manera simultánea.

El protocolo de ESP-NOW permite comunicarnos mediante mensajes cuya capacidad no debe de exceder los 250 bytes útiles. Se ha realizado un primer byte de control que posee la siguiente estructura y sirve para saber el tipo de mensaje que se ha recibido:

Tabla 2. Campos del byte de control

1bit CHECK	1brite RESERVED	4bits PAN ADDR	2bits MSG TYPE
-------------------	------------------------	-----------------------	-----------------------

El primer bit es de control y se usa para saber si el mensaje ha sido recibido o enviado correctamente o si hay mensajes para el dispositivo. El segundo bit queda reservado. Los cuatro siguientes bits corresponden a la dirección de la red personal. Los dos bits restantes al tipo de mensaje que se pueden recibir o enviar, que pueden ser los siguientes:

- ➔ Tipo de mensaje “00” corresponde a la función de autoemparejamiento. Aquí, el dispositivo intenta conectarse a la pasarela para poder enviar y recibir datos de la misma. Tiene la ventaja inmediata de que no requiere actividad por parte del usuario puesto que una vez se pierde la conexión, es el dispositivo quien intenta realizar de nuevo la conexión. Es una funcionalidad rápida, puesto que, tras un primer emparejamiento, la información se queda almacenada en la memoria no volátil y se reutiliza para futuras conexiones.
- ➔ Tipo de mensaje “01” corresponde al envío de datos al router o pasarela. Si se establece una dirección *PAN* > 2 se difundirá entre los diferentes dispositivos que estén en la misma red de área personal. Se ha reservado el *PAN* = 0 para la información que no se quiere difundir, solo enviar al servidor. También se ha reservado el *PAN* = 1 para usos futuros. En el envío de datos se puede aprovechar el bit CHECK para solicitar los mensajes que estén pendientes para el dispositivo.
- ➔ Tipo de mensaje “01” y “11” corresponde a la recepción de mensajes pendientes provenientes del servidor, mediante MQTT. Estos mensajes se envían uno a uno y nos mandan el mensaje final. El formato del mensaje está compuesto por el “topic” del mismo y el contenido útil o “payload”, separados entre ellos por una barra vertical.

$$\text{packet} = \text{topic}|\{\text{message}\}$$

- ➔ Tipo de mensaje “10” y “11” corresponde a la recepción de mensajes pendientes provenientes de dispositivos de la misma red de área personal. Estos mensajes se envían uno a uno y nos mandan el mensaje final. El formato del mensaje es el siguiente:

Tabla 3. Campos del mensaje PAN

Campo	Bytes
Tipo de mensaje	1
MAC	6
Antigüedad del mensaje (ms)	4
Cuerpo del mensaje en JSON	Hasta completar 250 bytes

4.2. Emparejamiento automático ESP-NOW

Como se indicó en la introducción del presente trabajo, un aspecto fundamental del mismo es que debe de ser autónomo. En lo que a la conexión se refiere, debe permitir conectarse de manera automática al dispositivo que recibe la información requerida. Esto se debe de hacer en el menor tiempo posible para asegurar también la eficiencia energética y robustez en la trasmisión de datos.

Se pretende realizar una rutina que se conecte automáticamente a una pasarela que servirá de unión al protocolo estándar del IoT, “Message Queuing Telemetry Transport” o MQTT. Para realizar esta rutina, que se ha realizado en definitiva como una tarea, hay que tener en cuenta que la pareja emisor-receptor deben de estar en el mismo canal de trasmisión y que, como identificador, se usará la dirección MAC, dado que cada ESP dispone de una única dirección MAC.



Ilustración 15. Esquema de conexión sender-receiver en ESP-NOW (source: RANDOM NERD TUTORIALS, ESP32: ESP-NOW Web Server Sensor Dashboard (ESP-NOW + Wi-Fi))

Para entender el funcionamiento de la tarea que se pretende diseñar, tenemos que tener claro lo que queremos hacer. En primer lugar, debemos de identificar tres situaciones distintas que resumen el estado de conexión al que pretendemos llegar:

- Petición de conexión “PAIR_REQUEST”. En este estado, enviamos una petición de conexión al dispositivo receptor a través de una MAC de broadcast del tipo “FF:FF:FF:FF:FF:FF”. En esta petición de mensaje se incluye un código que establece el tipo de mensaje que se está enviando, en este caso, del tipo Enlace (“Pairing”). El mensaje se envía a esta dirección y se esperará una respuesta por parte de la pasarela, en la que con un mensaje de respuesta en la que se incluye el mismo tipo Enlace, el dispositivo queda emparejado con la MAC correspondiente.

*Ilustración 16. Dirección MAC*

- Emparejamiento solicitado “PAIR_REQUESTED”. En este estado, si seguimos sin identificar el canal de trasmisión correcto, seguimos buscando el canal correcto incrementando el canal y reintentando la petición de conexión.
- Emparejamiento realizado “PAIR_PAIRED”. En este último estado, ya hemos conseguido conectarnos y suspendemos la tarea de conexión.

La tarea se tiene que localizar en una rutina con un bucle infinito, atendiendo a las consideraciones que se vieron al inicio de este apartado. No se puede realizar una tarea cuyo tiempo en estado “Running” exceda un determinado tiempo o incluso vaya más rápido. Por eso, es altamente recomendado poner determinados retrasos utilizando la función “vTaskDelay()” que permite que la tarea de feedback al TWDT (“Task Watchdog Timer” o temporizador de vigilancia de tareas). El temporizador de vigilancia de tareas (TWDT) se usa para monitorear tareas particulares, asegurando que puedan ejecutarse dentro de un período de tiempo de espera determinado. El TWDT observa principalmente las tareas inactivas de cada CPU, sin embargo, cualquier tarea puede suscribirse para ser supervisada por el TWDT.

El diagrama de flujo de la tarea que realiza esta función lo podemos ver a continuación:

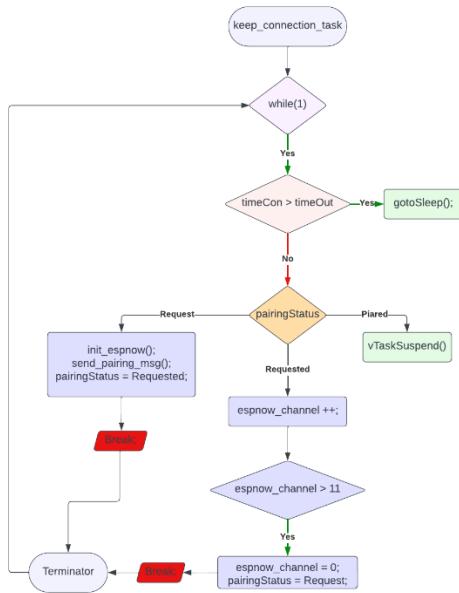


Ilustración 17. Diagrama de flujo de la tarea "mantener conexión"

Al observar las tareas inactivas de cada CPU, el TWDT puede detectar instancias de tareas que se ejecutan durante un período prolongado de tiempo sin rendirse. Esto puede ser un indicador de un código mal escrito que gira en un periférico, o una tarea que está atascada en un bucle infinito.

Antes de invocar esta tarea con el comando “`xTaskCreate()`” debemos de iniciar canal WiFi, pues será sobre esta banda por donde nos comunicaremos mediante el protocolo ESP-NOW. Una vez inicializada la tarea, su funcionamiento es el siguiente:

1. En primer lugar, si el tiempo de conexión t_{con} excede un tiempo determinado t_{out} , el dispositivo entra en un régimen de sueño profundo “Deep Sleep”, que es un modo de muy bajo consumo. Este modo estará un tiempo determinado hasta que se despierte y vuelva a intentar la conexión.
2. El estado de emparejamiento inicialmente estará en modo “PAIR_REQUEST”. En este punto, se realizan las siguientes operaciones:
 - 2.1. Iniciamos el protocolo ESP-NOW. Se inicializa el canal WiFi y se añade el receptor con la dirección MAC de broadcast. Se deben de registrar las funciones de respuesta “callback” que permiten enviar y recibir mensajes por ESP-NOW.
 - 2.2. Enviamos un mensaje de emparejamiento, indicando el tipo de mensaje y el número de identificación del dispositivo desde donde lo enviamos.

3. Una vez cambiado el estado a modo “PAIR_REQUESTED”, estado en el que se llega una vez enviado el mensaje de petición, esperamos a que la pasarela nos responda y nos emparejemos. De no ser así, vamos cambiando el canal de comunicación hasta que logremos una conexión estable.
4. Si nos encontramos conectados, “PAIR_PAIREDD”, ponemos la tarea en suspensión.

En la rutina establecida para recibir mensajes por ESP-NOW, se debe de tener en cuenta que podemos recibir de todo. En nuestro caso, deseamos recibir información relativa a datos que nos puedan enviar otros dispositivos para analizarlos (“DATA”) o información de emparejamiento (“PAIRING”). Si no hay datos para nosotros, es decir, no estamos esperando nada, salimos de esta función.

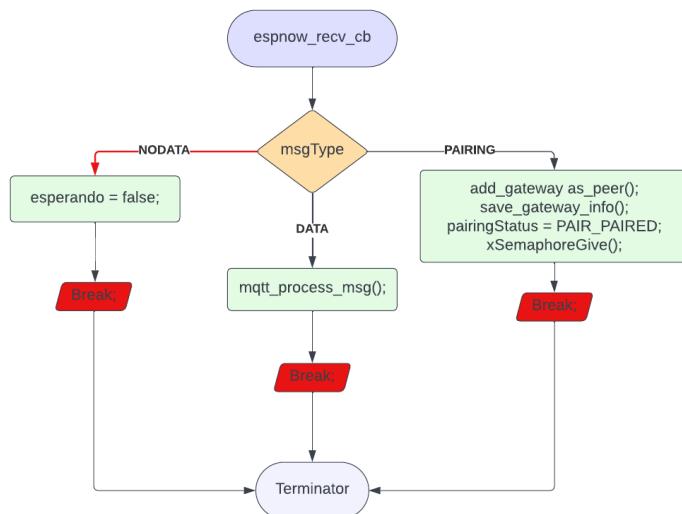


Ilustración 18. Función de respuesta de mensaje recibido del protocolo ESP-NOW

- Si recibimos por parte de la pasarela un mensaje cuyo tipo es de datos (“DATA”), debemos de procesar la información. Esta información está en formato JSON, por lo que hay que tratarla de una forma específica.
- Si recibimos por parte de la pasarela un mensaje cuyo tipo es de emparejamiento (“PAIRING”), añadimos la pasarela con su MAC correspondiente y el canal por el que nos ha respondido. En este punto, es conveniente almacenar la información para por ejemplo iniciar la conexión una vez nos despertemos de algún modo de bajo consumo o sueño profundo. En este caso, se dispone de la memoria no volátil, NVS, la cual es adecuada para almacenar valores que no se van a actualizar en cada reinicio del microcontrolador. Esto nos proporcionará un arranque aún más rápido cuando se vuelve a iniciar el dispositivo después de haber entrado en algún modo de sueño profundo. Se actualiza también el modo en el que nos encontramos una vez emparejados.

Un aspecto que cabe resaltar es la gestión de la tarea ante imprevistos como colisiones de mensajes o alguna gestión de los propios mensajes para evitar colas. Por suerte, FreeRTOS nos permite realizar operaciones de gestión de tareas que resultan bastante útiles para coordinar procesos, recordando a las reglas POSIX usadas en entornos UNIX.

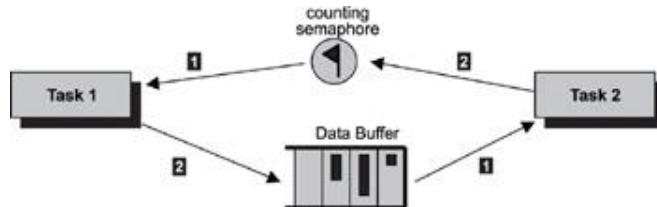


Ilustración 19. Uso de semáforos para controlar el flujo de un proceso

Por un lado, para la gestión de colisiones y sincronización se ha creado un semáforo binario en la rutina de inicialización de variables, al comienzo de la función “app_main”. Esto se realiza con la función “xSemaphoreCreateBinary()”, la cual inicializa una variable del tipo “SemaphoreHandle_t” con la que podemos referenciar dicho semáforo en otras funciones. Cada semáforo binario requiere poca RAM y se usa para mantener el estado de algún semáforo. Cuando pensamos en semáforos binarios podemos pensar en mutex, cuando comparten alguna característica, pero no son lo mismo. Un mutex es un semáforo binario que garantiza la exclusión mutua en las operaciones que se ejecutan sobre el semáforo general, puesto que tienen un mecanismo de herencia de prioridad. Los semáforos binarios son la mejor opción ante situaciones en las que es importante la sincronización, bien entre tareas o entre tareas y procesos.

Una vez se crea, se inicia en un estado vacío (sin asignar) lo que significa que se debe de dar el semáforo por primera vez con la función “xSemaphoreGive()”, antes de que se pueda tomarse posteriormente con la función “xSemaphoreTake()”.

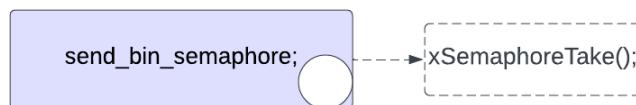


Ilustración 20. Semáforo siendo creado, pero sin poder tomarlo

Cuando recibimos un mensaje de tipo “PAIRING” en la rutina de recepción “espnow_recv_cb()”, liberamos el semáforo de envío de mensajes, por lo que ya estamos en disposición de enviar mensajes.



Ilustración 21. Semáforo siendo liberado

Cuando enviamos cualquier mensaje, primero tenemos que ver que podemos obtener el semáforo que se encarga de gestionar los envíos con la función “xSemaphoreTake()”. Si el semáforo no está disponible durante un tiempo determinado, liberamos el recurso y reiniciamos el dispositivo.



Ilustración 22. Semáforo siendo tomado

Por otro lado, podemos recurrir de nuevo a una herramienta de gestión de colas [16] que nos proporciona FreeRTOS. Cada cola requiere RAM que se utiliza para mantener el estado de la cola y para contener los elementos contenidos en la misma (el área de almacenamiento de la cola). Si se crea una cola con “xQueueCreate()”, la RAM necesaria se asigna automáticamente desde la pila de FreeRTOS. Esta se crea al comienzo de la aplicación, junto con la inicialización de todas las variables.

La cola en este caso se ha implementado para almacenar los resultados de los envíos, de tal manera que con esta información sabemos si ha sido enviado correctamente o no un determinado mensaje y actuar en consecuencia.



Ilustración 23. Rutina de gestión de colas

Las colas se suelen crear para permitir comunicar dos procesos, tareas o funciones diferentes. Las colas tienen un tamaño definido por el usuario que permite almacenar varios tipos de datos. Por defecto, cuando la cola se crea no contiene ningún tipo de dato.



Ilustración 24. Creación de la cola

En la función de retorno de mensajes enviados de ESP-NOW podemos saber si el mensaje ha sido enviado correctamente o no. Esta información se introduce en una cola mediante el comando “xQueueSend()”. El resultado del envío se coloca en el fondo de la cola y, al tener la cola vacía, también es el valor que está al frente de la cola.

Capítulo 4 - Diseño e implementación de los protocolos de comunicación

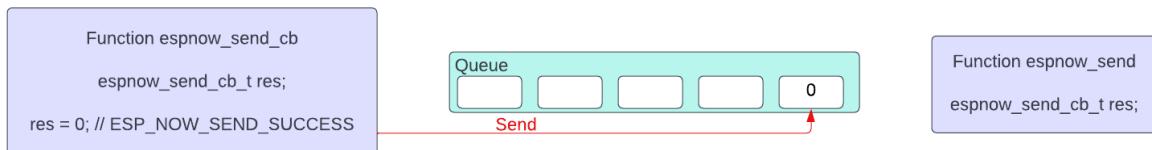


Ilustración 25. Envío correcto a la cola. Primer dato introducido

Si, por ejemplo, hubiera habido un fallo con el envío del mensaje, la función “espnow_send_cb” cambiaría el valor de la variable local antes de introducirla en la cola de nuevo. Ahora, la cola contiene copias de ambos valores escritos a la misma. El primer valor escrito continua al frente de la cola y el nuevo valor es insertado al final de la cola.

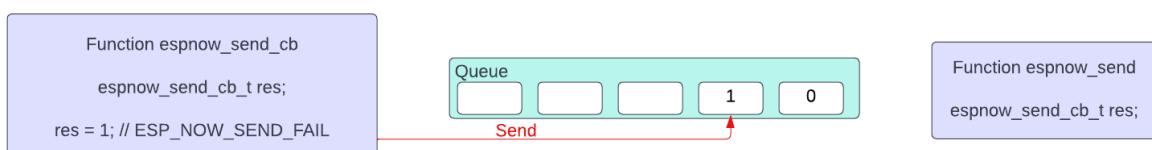


Ilustración 26. Envío incorrecto a la cola. Segundo dato introducido

Una vez enviado el mensaje, podemos recoger el resultado de la cola en otra función, “espnow_send” con “xQueueReceive()” y lo introduce en otra variable local diferente. El valor recibido por esta función es el valor que hay a la cabeza de la cola, es decir, el primer valor que la función “espnow_send_cb” escribió en la cola (*res = 0*).

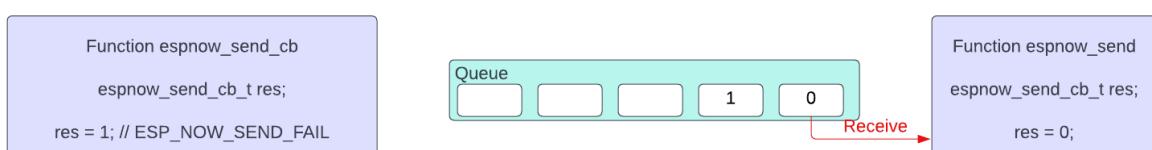


Ilustración 27. Recepción del dato de la cola en la función "espnow_send"

Cuando esto sucede, la función “espnow_send” elimina este elemento de la cola y deja el último valor introducido en la misma cola. Este valor será el próximo en leerse si se lee de nuevo la cola.

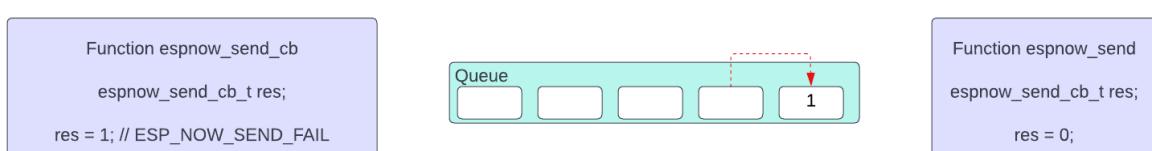


Ilustración 28. Desplazamiento del último valor introducido en la cola

En esta función se analiza el tiempo que se espera en la cola por un resultado de envío satisfactorio. Si se tarda en recibir un resultado un determinado tiempo, significaría que el envío ha tenido algún error.

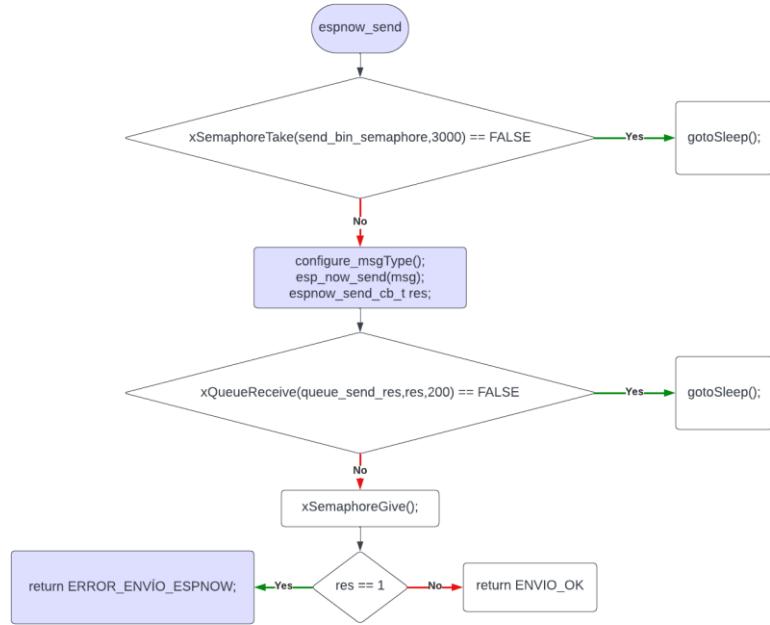


Ilustración 29. Función espnow_send()

Capítulo 5

Diseño e implementación de otras funcionalidades del dispositivo

La aplicación que se ha diseñado cuenta con varias partes que permiten un funcionamiento completo del sistema. Incluye el uso del módulo ADC, conexión a red mediante WiFi, el manejo de las memorias no volátiles y de tiempo real y un gestor de versiones OTA.

5.1. Lectura del convertidor analógico-digital

Para la lectura de sensores analógicos es usado frecuentemente convertidores analógicos-digitales [17] que permiten transformar la señal leída en un rango de tensiones interpretable por la CPU. En nuestro caso, se ha utilizado para leer y procesar los valores leídos por sondas de humedad de tierra.

El ESP32 contiene convertidores analógicos-digitales embebidos en sensor on-chip que permite medir señales analógicas de los pines dedicados a esta función, permitiendo además funcionar en varios escenarios:

- Generación de conversiones ADC usando el modo “one-shot”
- Generación de conversiones ADC usando el modo continuo

El modo continuo es un modo apropiado para lecturas rápidas, permitiendo una frecuencia de muestreo de hasta 2 MHz sin WiFi activo.

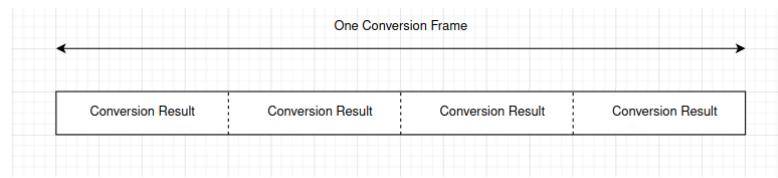


Ilustración 30. Trama de datos en una conversión del ADC en modo DMA (source: ESP-IDF Programming Guide, Analog to Digital Converter (ADC) Continuous Mode Driver)

Para aplicaciones en las que el WiFi sea necesario, los propios de Espressif recomiendan no usar frecuencias de muestreo superiores a 1 kHz, por lo que con el modo “one-shot” podemos obtener buenos resultados sin comprometer el resto de funcionalidades del dispositivo.

El ESP32-C3 tiene integrado dos ADC de 12 bits tipo SAR (“Successive Approximation Register”), formando un total de hasta 6 canales habilitados para conversiones analógicas. Sin

embargo, una limitación patente en la familia ESP está en que el módulo de conversión numero dos (ADC2) es usado por el módulo WiFi, por lo que funciones como “adc2_get_raw()” pueden fallar mientras “esp_wifi_start()” se ejecuta. Por lo que, en nuestra aplicación, se usará el módulo de conversión uno (ADC1) puesto que no entra en conflicto con otro módulo del microcontrolador. Este módulo dispone de hasta 5 canales ADC (desde el GPIO0 al GPIO4).

Para implementar un ADC hay que instanciarlo, utilizando para ello la API apropiada, “adc_oneshot_unit_init_cfg_t”, lo cual nos permitirá configurar los diferentes parámetros del conversor:

- El módulo utilizado “unit_id”, que puede ser el módulo uno (“ADC_UNIT_1”) o el módulo dos (“ADC_UNIT_2”)
- La fuente del reloj para la conversión “clk_src”
- Activar o desactivar el modo ULP (“Ultra Low Power”) “ulp_mode”

```
adc_oneshot_unit_handle_t adc1_handle;
adc_oneshot_unit_init_cfg_t init_config1 = {
    .unit_id = ADC_UNIT_1,
    .ulp_mode = ADC_ULP_MODE_DISABLE,
};

ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &adc1_handle));
```

Después de haber configurado el módulo, se debe de llamar a la función “adc_oneshot_new_unit()” con la configuración previamente creada, lo que generará una instancia del ADC si no ha habido ningún fallo.

Con la unidad creada, se puede configurar los canales que se van a usar. Para ello se puede invocar la estructura “adc_oneshot_chan_cfg_t” que contiene los siguientes parámetros:

- La atenuación requerida del ADC “atten”
- El ancho de banda del resultado de la conversión “bitwidth”

```
//-----ADC1 Config-----/
adc_oneshot_chan_cfg_t config = {
    .bitwidth = ADC_BITWIDTH_DEFAULT,
    .atten = ADC_ATTEN,
};

for(int i=0;i<lengthADC1_CHAN;i++) {
    ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle,
    adc_channel[i], &config));
}
```

Para aplicar los cambios, se debe de ejecutar la función “adc_oneshot_config_channel” con la configuración previamente hecha tantas veces como canales se estén utilizando.

Una vez esté todo configurado, habrá que llamar a la función “adc_oneshot_read()” para obtener el resultado de la conversión de los canales del ADC. El controlador de modo oneshot ADC funciona en una rutina de polling o sondeo, por lo que la función “adc_oneshot_read()” cunsultará la CPU hasta que la función regrese. Durante este período de tiempo, la tarea en la que reside el controlador de modo oneshot ADC no se bloqueará. Por lo tanto, la frecuencia del reloj es estable durante la lectura.

En este punto se puede calibrar el ADC para obtener una mejor medida. Para ello, se puede ajustar la lectura a una curva determinada por el usuario o bien siguiendo el esquema que nos propone la API. Para hacerlo con la API, simplemente tenemos que configurar la estructura “adc_cali_curve_fitting_config_t” que contiene los siguientes parámetros:

- La unidad utilizada por el conversor “unit_id”
- El canal por el que se va a tomar la medida “chan”
- La atenuación requerida del ADC “atten”
- El ancho de banda del resultado de la conversión “bitwidth”

```
adc_cali_curve_fitting_config_t cali_config = {
    .unit_id = unit,
    .atten = atten,
    .bitwidth = ADC_BITWIDTH_DEFAULT,
};

ret = adc_cali_create_scheme_curve_fitting(&cali_config, &handle);
```

La instancia de la calibración se incluye en “handle”, de tal modo que cuando queramos obtener un valor leído usando la función “adc_cali_raw_to_voltage” y le pasemos la instancia, nos devolverá un valor de tensión, en este caso calibrado, de la conversión realizada.

Un aspecto relevante que se pretende valorar es la autonomía del dispositivo, para lo cual debemos de asegurarnos de que el consumo sea mínimo.



Ilustración 31. Sensor de humedad de suelo SEN0308

El sensor con el que se pretende tomar medidas es el que se ilustra en la imagen 37. Es una sonda de humedad de suelo, la cual puede ir alimentada a tensiones de entre 3.3V a 5V. Este sensor cuenta con un pin de tensión, uno de salida de lectura y dos pines de tierra, uno que realiza la conexión a tierra del circuito y otro que tiene la función de apantallamiento o “shielding” para evitar interferencias.

En términos de consumo es poco eficiente, ya que es un sensor que consume unos 7mA y de pretender realizar cualquier proyecto que dependa de una batería, esta se gastaría rápidamente.



Ilustración 32. Consumo del sensor de humedad

Puesto que lo que se necesita es una lectura instantánea para procesarla y enviarla, se podría pensar en un sistema que activara el sensor exclusivamente cuando se requiera una lectura. En este sentido se ha diseñado un pequeño circuito que actúa como un interruptor que mantiene apagado el sensor hasta que se entra en la rutina de lectura. Una vez en esta rutina de lectura, se mantiene encendido un tiempo suficiente para asegurar que la lectura sea estable y coherente con la situación del sensor. Cuando se haya adquirido la lectura, se apagará el sensor para reducir al máximo el consumo.

El uso de transistores MOSFET como interruptores es adecuado puesto que consumen muy poco y además permiten operar a frecuencias y temperaturas que, según la aplicación, resultan más convenientes que los BJT. El aspecto que hace decantarse por un tipo u otro de MOSFET para aplicaciones particulares con microcontroladores recae en los niveles de activación lógicos o niveles TTL en los BJT o el voltaje umbral para los MOSFET. En este proyecto se pretende activar los sensores aplicando una activación mediante un pin GPIO a un transistor que cierre el circuito de alimentación de los sensores y así poder medir, por lo que necesitamos un transistor que permita umbrales de tensión de 3.3V.

El transistor BS170 [18] es un transistor de pequeña señal tipo MOSFET de canal N, bastante adecuado para este proyecto, puesto que tiene una tensión de activación umbral pequeña, perfecto para el uso con los GPIO de los microcontroladores actuales. Además, al ser de canal N, aseguramos que se encienda cuando se aplique una tensión en el terminal de compuerta adecuado.

ON CHARACTERISTICS (Note 1)					
	V _{GS(Th)}	0.8	2.0	3.0	V _{dc}
Gate Threshold Voltage (V _{DS} = V _{GS} ; I _D = 1.0 mAdc)					
Static Drain-Source On Resistance (V _{GS} = 10 Vdc, I _D = 200 mAdc)	r _{DS(on)}	—	1.8	5.0	Ω
Drain Cutoff Current (V _{DS} = 25 Vdc, V _{GS} = 0 Vdc)	I _{D(off)}	—	—	0.5	μA
Forward Transconductance (V _{DS} = 10 Vdc, I _D = 250 mAdc)	g _f	—	200	—	mmhos

Ilustración 33. Característica del transistor BS170

En este tipo de circuitos se debe de referenciar, en función del tipo de MOSFET utilizando, “aguas arriba” o “aguas abajo” o bien pull-up o pull-down, según sea necesario. Para un MOSFET tipo N, se debe de colocar una resistencia entre la compuerta y la fuente (pull-down) para evitar la lógica de compuerta flotante y nos de una salida no deseada desde el MOSFET durante los arranque y reinicios del microcontrolador.

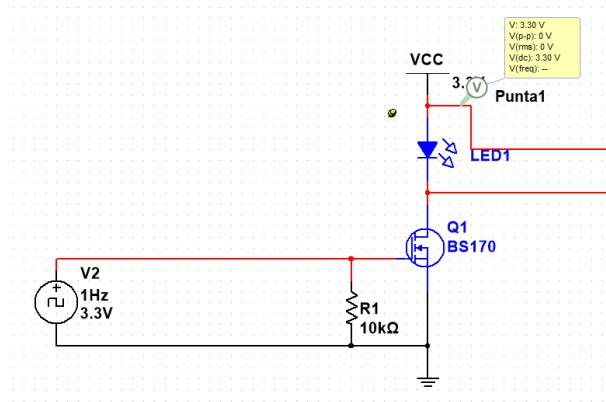


Ilustración 34. Configuración del BS170

Un problema de los que se percató a la hora de montar el circuito de la figura 40 y ponerlo a funcionar, fue que se detectó en la toma de medidas picos de tensión al abrir el circuito de casi 1V en la puerta del MOSFET que también aparece en el drenaje y, por lo tanto, afectará directamente a la lectura que podamos obtener del sensor de humedad. En la siguiente imagen se puede apreciar estos picos en la figura superior que representa ciclos de encendido y apagado del sensor.

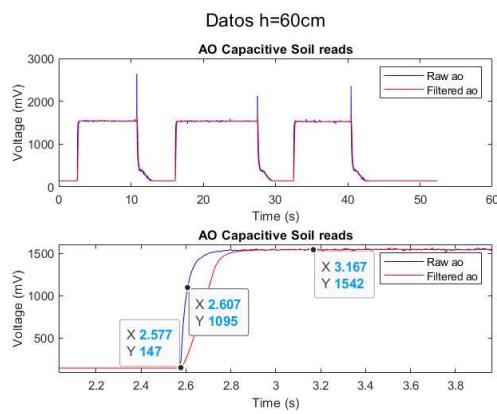


Ilustración 35. Datos obtenidos del sensor sin circuito de protección

Para solucionar esto, se ha colocado un condensador C_1 entre el GPIO de encendido y la alimentación y otro condensador C_2 entre el GPIO y tierra. El circuito resultante se aprecia en la siguiente ilustración.

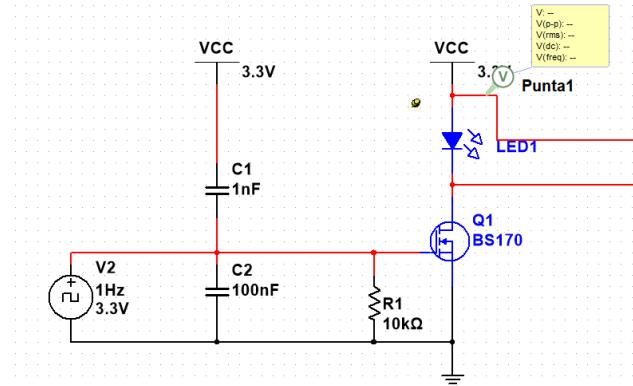


Ilustración 36. Circuito de encendido con protección anti-rebotes

Cuando no se aplica ninguna tensión por el pin GPIO, es decir, el circuito está abierto, la puerta del MOSFET está conectada a GND a través de la resistencia R_1 . En este estado, tanto el MOSFET como el sensor están apagados. Cuando se cierra el circuito, es decir, se aplica una tensión V_{CC} a la puerta, se enciende el MOSFET y se enciende el sensor. El condensador C_2 se carga a través de esta tensión de puerta y, cuando se deja de aplicar V_{CC} , C_2 se descarga a través de R_1 y la tensión en la puerta cae, apagando el MOSFET de manera progresiva. La combinación de C_1 y C_2 absorbe este pico de tensión, puesto que amortigua la caída de tensión al desconectar el MOSFET.

El resultado se puede ver en el osciloscopio, como en la siguiente figura, donde se ha reducido completamente ese pico tan indeseado de tensión.

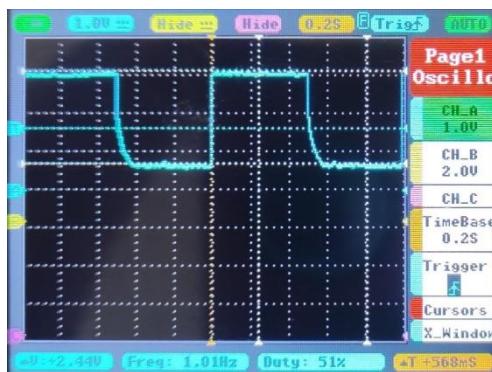


Ilustración 37. Señal de los sensores con circuito de protección

En cuanto al tiempo que tarda el sensor en estabilizar la medida, se ha realizado un experimento en el que se ha encendido y apagado el sensor sumergido en agua a diferentes alturas, para ver lo que tarda en estabilizarse la lectura. Por ejemplo, para la figura que se muestra a continuación, que corresponde a una inundación de la sonda hasta los 20cm, se ve que el tiempo de subida es de aproximadamente unos 30ms.

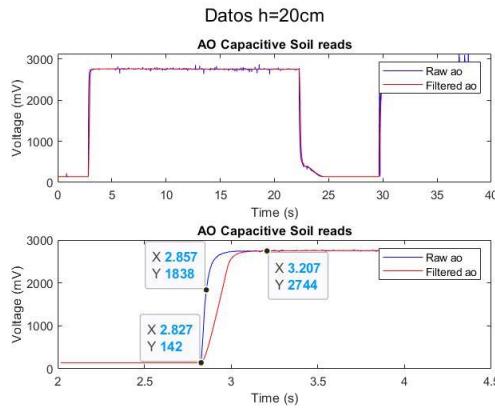


Ilustración 38. Resultados temporales de la lectura de la sonda

Es por ello que, si se quiere tomar una muestra en el que ya el sensor tenga una tensión estabilizada, habría que esperar un tiempo de establecimiento de, al menos, 120ms para que así las lecturas sean correctas.

El diagrama de flujo que se corresponde a esta parte del proyecto se puede estudiar a continuación, donde se ve la configuración del pin GPIO encargado de encender el sensor, la inicialización, configuración y calibración del módulo uno del ADC y el proceso de lectura del sensor, activando para ello el MOSFET que se encarga de encenderlo. Una vez finalizada la lectura, se apaga el sensor y de elimina el componente del ADC creado para impedir que el componente siga consumiendo.

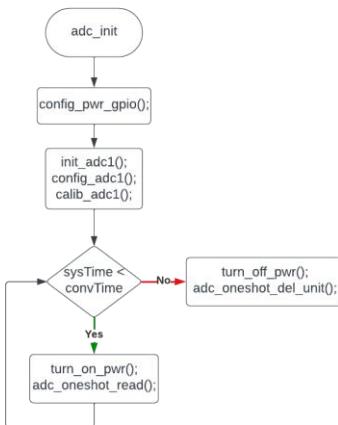


Ilustración 39. Diagrama de flujo de la lectura del ADC

5.2. Conexión al protocolo de comunicación WiFi

En este punto se va a estudiar cómo se establece una conexión WiFi [19] a Internet. En el apartado donde se estudió la conectividad con ESP-NOW se vio como se configuraba el WiFi para establecer una canal de comunicación para dar banda ancha al protocolo ESP-NOW. En este apartado, se pretende configurar una conexión WiFi que se utilizará para conectar el microcontrolador a Internet de cara, por ejemplo, a dar servicio FOTA.

Tanto para el caso en el caso de que simplemente queramos habilitar un canal WiFi para dar funcionalidad al protocolo ESP-NOW, como para conectarnos a Internet, debemos iniciar el driver WiFi [22] que nos trae el soporte de Espressif. Este driver posee unas llamadas a las funciones que están recogidas en el API que se pueden utilizar para interactuar con este componente. El driver es ejecutado en paralelo con el programa principal y se comunica con el mediante eventos.

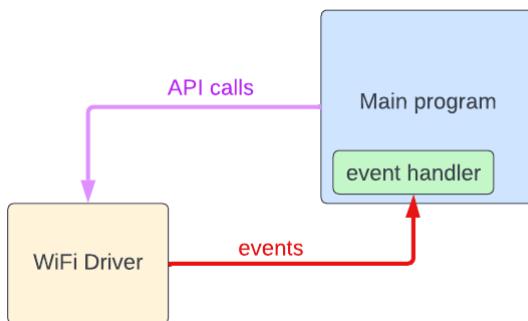


Ilustración 40. Driver WiFi

Primero de todo, se debe de inicializar la pila que almacenará tanto los eventos, la transmisión, configuración, los datos que se transmitan mediante TCP/IP. Para ello se debe de ejecutar una única vez el comando “`esp_netif_init()`” para dar soporte a este driver.

Para una configuración sencilla en la que simplemente queramos inicializar el canal WiFi, bastaría con indicar el lugar de almacenamiento de dicha pila “`esp_wifi_set_storage()`”, el modo de conexión, que puede ser AP (punto de acceso, “Access point”) o STA (estación, “Station”) o ambos con “`esp_wifi_set_mode()`”. Con esto ya podemos iniciar el WiFi con “`esp_wifi_start()`” y seleccionar el canal de comunicación “`esp_wifi_set_channel()`”

```

/* WiFi should start before using ESPNOW */
static void wifi_init(void) {
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
    ESP_ERROR_CHECK(esp_wifi_set_mode(ESPNOW_WIFI_MODE));
    ESP_ERROR_CHECK(esp_wifi_start());
#if CONFIG_ESPNOW_ENABLE_LONG_RANGE
    ESP_ERROR_CHECK(
        esp_wifi_set_protocol(ESPNOW_WIFI_IF,
        WIFI_PROTOCOL_11B|WIFI_PROTOCOL_11G|WIFI_PROTOCOL_11N|WIFI_PROTOCOL_LR)
    );
#endif
}

```

La conexión a Internet requiere de algo más de complejidad que lo anterior, puesto que depende de mecanismos de sincronización. En el programa principal habrá que crear una instancia que llame al driver que se encargue de gestionar el módulo WiFi cada vez que se solicite. Para que el driver funcione adecuadamente conjunto al programa principal y el resto de módulos, se debe de sincronizar la actividad. De manera pormenorizada, la conexión WiFi a Internet sigue el siguiente flujo:

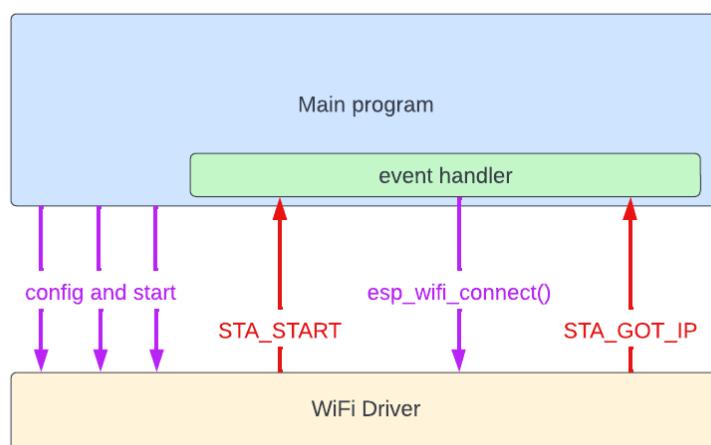


Ilustración 41. Diagrama de conexión WiFi

El funcionamiento del driver es el siguiente:

- El programa principal configura y empieza el driver con las llamadas de la API
- Tras haber completado las tareas internas, el driver notifica que ha empezado exitosamente el arranque del evento SYSTEM_EVENT_STA_START
- El gestor de eventos, tras haber recibido este evento, puede llamar a la función “esp_wifi_connect()” para indicarle al driver que se conecte a la red especificada durante la fase de configuración
- Cuando la conexión está completada y después de haber obtenido una dirección IP válida (si el servicio DHCP está disponible), el driver empieza el evento SYSTEM_EVENT_STA_GOT_IP
- El gestor de eventos puede informar al programa principal que la conexión ha sido completada

Estos mecanismos de sincronización utilizados en la conexión WiFi se denominan eventos. La API que gestiona el WiFi tiene una librería con los tipos de eventos existentes, “esp_wifi_types.h”. Estos eventos son bien gestionados por FreeRTOS.

Como se expuso en su momento, FreeRTOS dispone de una amplia variedad de mecanismos de gestión y sincronización de recursos. En el caso de la conexión WiFi, que es al final un módulo que está funcionando en paralelo con el programa principal y se comunica entre diferentes funciones, se necesita de una herramienta más compleja que los mutex, semáforos o colas, vistas anteriormente. Los eventos [20], en este sentido, permiten comunicación y sincronización entre tareas, lo cual es bastante conveniente para esta situación.

Un evento indica un acontecimiento importante, como es una conexión exitosa al protocolo WiFi para acceder a un punto de acceso. La API en la que se basan los eventos con FreeRTOS se denomina “esp_event”. Los eventos se lanzan en un bucle que funcionan como el puente entre los eventos y sus controladores. El origen del evento publica un evento a la rutina en bucle del evento usando la API y el controlador lo registra, en función del tipo de evento que se haya lanzado.

Conocidos como “Event bits” o bits de eventos [21] son similares a las banderas o “flags”, que son visibles para otras tareas:

- El usuario puede crear tantos “evento bits” como sea necesario
- Las tareas pueden activar (“set”) o desactivar (“unset”) los diferentes bits
- Una tarea puede pausar su ejecución esperando a la ejecución de uno o más bits de eventos

Cuando hacemos una agrupación de bits de eventos hablamos de grupos de eventos o “Event group”, de los que cada uno de estos eventos de grupos contienen 8 bits de eventos. Estos bits están numerados en función de su posición (BIT0, ..., BIT7).

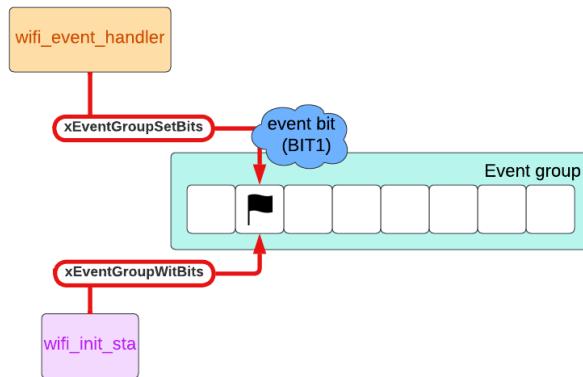


Ilustración 42. Esquema de grupos de eventos

El funcionamiento de la rutina “`wifi_init_sta`”, siguiendo este paradigma, es el que sigue:

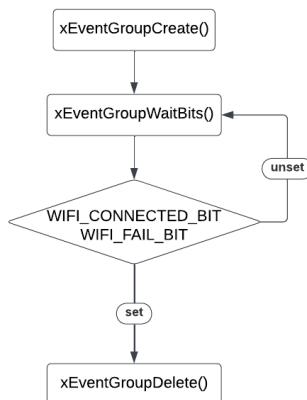


Ilustración 43. Diagrama de flujo de la conexión WiFi

- Se define un evento de grupo utilizando la función “`xEventGroupCreate()`”.
- Se inicia el canal WiFi y se registran los controladores de eventos para el comienzo de la conexión WiFi y la adquisición de una dirección IP válida
- Se rellena los parámetros de configuración WiFi necesarios (SSID, PASSWORD...). Una estructura parecida se muestra a continuación:

```
wifi_config_t wifi_config = {  
    .sta = {  
        .ssid = ESP_WIFI_SSID,  
        .password = ESP_WIFI_PASS,  
        .threshold.authmode = ESP_WIFI_SCAN_AUTH_MODE_THRESHOLD,  
        .sae_pwe_h2e = ESP_WIFI_SAE_MODE,  
    },  
};
```

- Se indica el modo de conexión WiFi, la configuración establecida con “esp_wifi_set_config()” y se inicia, como en la conexión vista para el canal ESP-NOW.
- Se espera hasta que la conexión se haya establecido utilizando los bits de evento “WIFI_CONNECTED_BIT” o conexión fallida en caso de haber superado un máximo de intentos de conexión “WIFI_FAIL_BIT”.
- La función “xEventGroupWaitBits()” nos devuelve el bit antes de la llamada de la función, por lo que podemos detectar el evento que ha fallado. Por ejemplo, parte del código de detección de bits se puede ver a continuación, donde se especifican los casos de éxito de conexión o fallo.

```
if (bits & WIFI_CONNECTED_BIT) {  
    ESP_LOGI(TAG, "connected to ap SSID:%s password:%s",  
            ESP_WIFI_SSID, ESP_WIFI_PASS);  
} else if (bits & WIFI_FAIL_BIT) {  
    ESP_LOGI(TAG, "Failed to connect to SSID:%s, password:%s",  
            ESP_WIFI_SSID, ESP_WIFI_PASS);  
} else {  
    ESP_LOGE(TAG, "UNEXPECTED EVENT");  
}
```

La rutina que controla los eventos, “wifi_event_handler”, tiene la siguiente estructura:

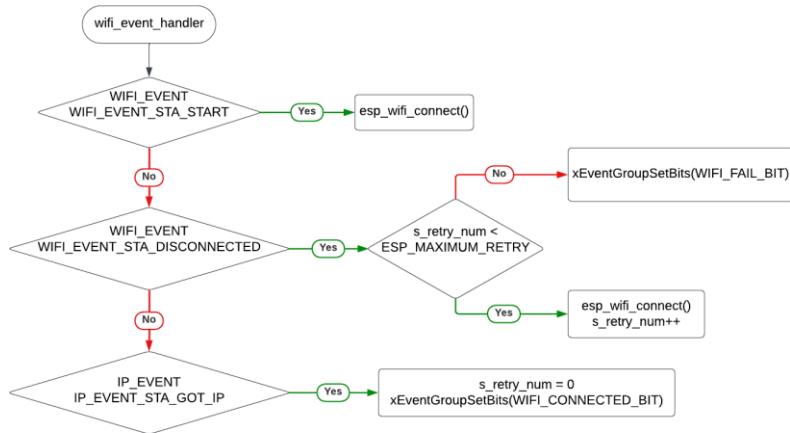


Ilustración 44. Rutina de control de eventos WiFi

- Si el evento base es “WIFI_EVENT” y el identificador es “WIFI_EVENT_STA_START”, indica que ha recibido una confirmación positiva de establecer conexión, luego, se procede a conectar el driver a Internet.
- Si el evento base es “WIFI_EVENT” y el identificador es “WIFI_EVENT_STA_DISCONNECTED”, indica que ha recibido una confirmación negativa de establecer conexión. En este punto se intenta establecer la conexión un número determinado de veces. En el caso de que el número de intentos supere un determinado umbral, se le indica al controlador que la conexión WiFi ha fallado, estableciendo el bit de grupo “WIFI_FAIL_BIT”.
- Si el evento base es “IP_EVENT” y el identificador es “IP_EVENT_STA_GOT_IP”, indica que hemos realizado una conexión favorable y que hemos adquirido una dirección IP válida. En este punto, reiniciamos el número de intentos y establecemos el bit de grupo “WIFI_CONNECTED_BIT”.

5.3. Gestión de versiones. Actualización FOTA

Cuando se plantea diseñar un producto que se enmarque dentro de la tecnología de Internet de las Cosas (IoT) se debe de evaluar la selección de componentes o plataformas con perspectiva a reducir el costo y aumentar el rendimiento y el diseño de aplicaciones que mejoren la conectividad. En este sentido, actualmente existen bastantes soluciones entre las que elegir, aunque siempre existe el reto de mantener el firmware actualizado mediante actualizaciones inalámbricas “por aire” u OTA (“Over the Air”). Esta función es crítica en cualquier aplicación que esté en desarrollo y actualmente los fabricantes de microcontroladores se aseguran de que entre dentro del soporte de los mismos para dar, sobre todo, la capacidad de mantenerse actualizado y poder mejorar el producto.

Como se ha visto, el ESP, tanto el ESP8266 como el ESP32 de Espressif permiten establecer conexiones inalámbricas tipo Bluetooth o WiFi, dándole la conectividad necesaria para establecer conexión con Internet. Esto nos permite conectarnos a algún servidor donde se aloje el firmware, descargarlo y actualizar el dispositivo.

Según el flujo de trabajo en el que se esté desarrollando el proyecto, podemos encontrarnos que el soporte puede variar en función de la solución que se plantee. Es el ejemplo de un equipo que se aprovecha de AWS (“Amazon Web Service”) IoT Core para hacer funcionar la solución OTA en ESP32. Sin embargo, Espressif nos brinda una solución potente para poder gestionar versiones. Normalmente, el proceso de configuración en un ESP32 para gestionar versiones OTA implica los siguientes pasos:

1. Configuración de la tabla de particiones del ESP32
2. Descargar el firmware que soporta OTA
3. Desarrollar una herramienta que actúe como servidor y trasmite el nuevo firmware
4. Descargar el último firmware en el ESP32
5. Cambiar a la nueva aplicación

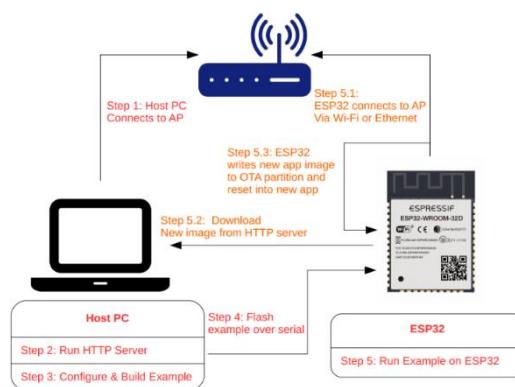


Ilustración 45. Proceso de conexión a servidor para mantenimiento OTA (source: DigiKey, How to Perform Over-the-Air (OTA) Updates Using the ESP32 Microcontroller and its ESP-IDF)

Como se vio en la introducción a ESP-IDF, debemos de configurar la tabla de particiones del ESP32 para que pueda albergar el nuevo firmware que se descargará y arrancará en una partición creada para este fin. Este paso es importante tenerlo ya que, de lo contrario, no funcionará.

En cuanto a la rutina que controla el proceso, se fundamenta en eventos, como se vio para establecer una conexión WiFi a Internet. En este caso, el único evento que se tiene que crear es la conexión HTTP/HTTPS al servidor OTA, puesto que será el evento que defina la posibilidad de actualizarse o no. La gestión de este evento es similar a la de la conexión WiFi, con algunas diferencias:

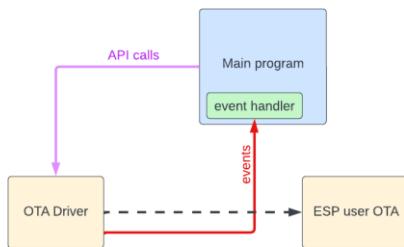


Ilustración 46. Driver OTA

El programa principal es quien inicializa el proceso de actualización enviando la dirección URL a un driver que gestionará las llamadas al API. El controlador se encargará de varios procesos:

1. Verificar la configuración al servicio OTA
2. Lanzar un proceso que descargue y realice la actualización OTA
3. Regresar al programa principal

Mientras que, en este caso, la funcionalidad que se le agregue al proceso entrará dentro de un bloque diseñado por el usuario para realizar una serie de funciones como:

1. Descargar el firmware en la partición creada en la tabla de particiones
2. Verificar la integridad de la imagen
3. Actualizar la partición
4. Reiniciar el dispositivo

Además, ESP-IDF nos permite realizar actualizaciones OTA certificadas, donde el usuario tendrá que suministrar los certificados del servidor donde se aloje el firmware para que la descarga pueda realizarse.

El driver que gestiona la actualización OTA, “ota_event_handler”, cuyo evento base es “ESP_HTTPS_OTA_EVENT” dispone de los siguientes eventos:

Tabla 4. Tipos de eventos OTA

Identificador de evento	Descripción
ESP_HTTPS_OTA_START	Empieza actualización OTA
ESP_HTTPS_OTA_CONNECTED	Se ha establecido una conexión con el servidor
ESP_HTTPS_OTA_GET_IMG_DESC	Se está leyendo la descripción del firmware
ESP_HTTPS_OTA_VERIFY_CHIP_ID	Se está verificando la identificación del chip para el firmware descargado

ESP_HTTPS_OTA_DECRYPT_CB	Llamada a la función para desencriptar el archivo protegido
ESP_HTTPS_OTA_WRITE_FLASH	Escribiendo en la memoria flash el nuevo firmware
ESP_HTTPS_OTA_UPDATE_BOOT_PARTITION	Actualizando la partición de arranque
ESP_HTTPS_OTA_FINISH	Finalizando la actualización OTA
ESP_HTTPS_OTA_ABORT	Abortado el proceso de actualización

En el programa principal se debe de configurar adecuadamente el cliente HTTP/HTTPS para poder realizar una conexión al servidor adecuada. En este sentido, una configuración similar a esta sería la adecuada:

```
esp_http_client_config_t config = {
    .url = FIRMWARE_UPGRADE_URL,
    .timeout_ms = OTA_RECV_TIMEOUT,
    .keep_alive_enable = true,
};
```

El campo “url” especifica la dirección completa, incluyendo el archivo y su extensión “.bin” del fichero al que queremos acceder. El tiempo de espera máximo de conexión lo podemos añadir al campo “timeout_ms” y si queremos establecer una conexión persistente, puesto que vamos a transferir archivos, debemos de activar la configuración “keep_alive_enable”.

La configuración OTA es similar a la anterior, a la que hay que pasarle la configuración de conexión más algún campo que se requiera, en función de la aplicación que se deseé crear.

```
esp_https_ota_config_t ota_config = {
    .http_config = &config,
    .http_client_init_cb = _http_client_init_cb, // Register a callback
    // to be invoked after esp_http_client is initialized
    //           .partial_http_download = true,
    //           .max_http_request_size = HTTP_REQUEST_SIZE,
};
```

Nos permite incluir una función de llamada, la cual podemos personalizar, así como incluir otras opciones como, por ejemplo, realizar una descarga parcial del fichero.

Una vez se ha configurado la conexión HTTP/HTTPS y la configuración OTA, el flujo que sigue el proceso de actualización es el siguiente:

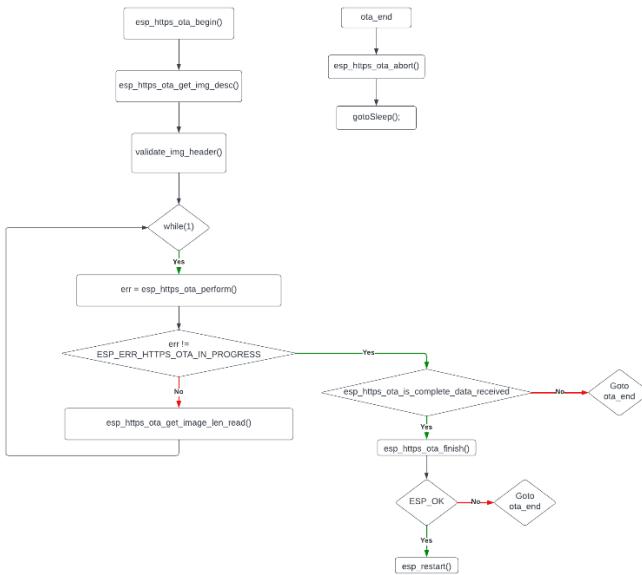


Ilustración 47. Flujo de la rutina FOTA

- La función “esp_https_ota_begin” recibe como parámetros la configuración especificada y un parámetro de tipo “esp_https_ota_handle_t” a la que se copiará la configuración y se hará referencia durante el proceso de actualización. Esto generará un evento de tipo “ESP_HTTPS_OTA_START” y, una vez se haya establecido conexión con el servidor, “ESP_HTTPS_OTA_CONNECTED”.
- Se lee la descripción de la imagen con “esp_https_ota_get_img_desc”, generando un evento del tipo “ESP_HTTPS_OTA_GET_IMG_DESC”
- Se valida la imagen para comprobar que pertenece al dispositivo que se desea actualizar con “validate_image_header”. Esto generará un evento del tipo “ESP_HTTPS_OTA_VERIFY_CHIP_ID”
- En el bucle de actualización, comenzamos con la función que realiza la actualización, “esp_https_ota_perform”. Esta función realiza la lectura del firmware desde el flujo recibido por HTTP y lo escribe en la partición OTA. Tiene la ventaja de devolver después de cada operación de lectura, por lo que permite monitorear el estado de la actualización OTA llamando a la función “esp_https_ota_get_image_len_read”, que permite obtener la longitud de la imagen leída hasta el momento. En este punto, tenemos el evento “ESP_HTTPS_OTA_WRITE_FLASH” activado.
- Una vez finalizada la actualización y hemos recibido todos los datos (de lo contrario nos daría un error), finalizamos la actualización con “esp_https_ota_finish” y con ello al evento “ESP_HTTPS_OTA_FINISH”. En este punto, se reinicia el dispositivo con el nuevo firmware cargado.

6. Si ha habido algún error al validar la imagen, bien porque esté corrupta o no se haya podido descargar, saltamos al identificador “ota_end”, donde se aborta el proceso con “esp_https_ota_abort”, generando un evento tipo “ESP_HTTPS_OTA_ABORT” e inducimos un estado de sueño profundo al microcontrolador

5.4. Manejo de la memoria no volátil NVS y RTC

Para almacenar los datos de emparejamiento o aquellos que se necesiten para realizar cálculos posteriores, se usan recursos de almacenamiento que no sean eliminados después de un sueño profundo o reinicio. El framework de Espressif nos da la posibilidad de controlar tanto la memoria no volátil NVS como la memoria RTC (“Real Time Clock”).

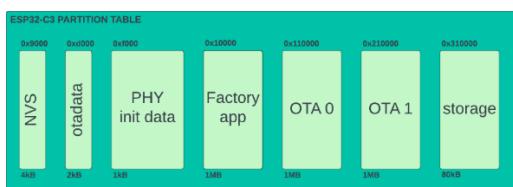


Ilustración 48. Tabla de particiones del ESP32-C3

El uso de un tipo de memoria u otro va en función de la frecuencia a la que queramos acceder a esta para escribir, sobre todo, puesto que la memoria flash se va degradando. Como los ESP32 disponen de memoria flash y, además, el ESP32-C3 dispone de memoria RTC, es bastante conveniente utilizarlas.

Por un lado, la memoria NVS utiliza una parte que el usuario debe de definir de la memoria flash de almacenamiento. Se pueden mezclar NVS,

FAT y SPIFFS en un proyecto, pero como todas las tecnologías son diferentes, todas necesitarán usar una partición diferente de la memoria flash. La vida útil del chip flash depende del tipo de chip flash y de su uso. Habitualmente el fabricante del chip suele especificar borrados de 10k a 100k por sector. Estos borrados dependerán de la tecnología que se utilice (flash, nvs, spiffs) y de cómo se utilice. Por otro lado, las actualizaciones OTA generalmente se escriben en una partición separada de NVS/SPIFFS/FAT y no se ven afectadas por el uso del resto del flash.

La biblioteca NVS permite almacenar datos en la propia memoria flash, pudiendo realizar cierta nivelación de desgaste. En el caso del ESP32-C3 dispone de 4MB de memoria flash que es utilizado para el software en sí, NVS y otros datos. Dado que no se pretende almacenar gran cantidad de datos, un espacio de 4kB será suficiente. En este sentido, NVS es utilizado para mantener datos como información sobre configuración que no cambia con frecuencia. Por ejemplo, la información de emparejamiento con la plataforma ESP-NOW. Desde un punto de vista energético, escribir en flash resulta más costoso, por lo que su acceso debe de reducirse al mínimo. Al arrancar la aplicación principal y de manera predefinida, el dispositivo inicializa la memoria flash.

Un diagrama de flujo adecuado para ver cómo se produce el inicio de los datos es el que se muestra a continuación:

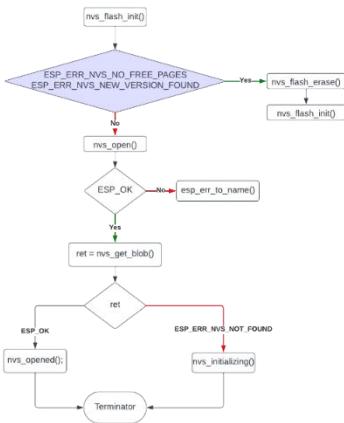


Ilustración 49. Inicialización de la memoria flash NVS

1. La memoria se inicializa con el comando “`nvs_flash_init`”. De no haber páginas libres en los módulos de memoria o haber encontrado una versión nueva de la propia memoria (en caso de actualizarse), la memoria se elimina “`nvs_flash_erase`” y se reinicia.
2. Para abrirlo con “`nvs_open`”, función que necesita saber a qué partición acceder, el modo de acceso que se quiere (“`NVS_READ`”, “`NVS_READWRITE`”) y el control (“`handle`”) tipo “`nvs_handle_t`”.
3. Cuando se haya abierto, si no ha habido ningún error, se recogen los datos de la memoria flash. Si es un conjunto de datos o estructuras numéricas, el comando que se debe de utilizar será “`nvs_get_blob`”. La biblioteca que gestiona la API de la memoria NVS tiene más posibilidades para gestionar el almacenamiento en función de las variables que se guarden.
4. Si se encuentran los datos, la memoria se guardan los datos en la variable que se indique en “`nvs_get_blob`”. De lo contrario, se debe de inicializar las variables para, posteriormente, almacenar los datos.

Para almacenar datos, la estructura es la siguiente:

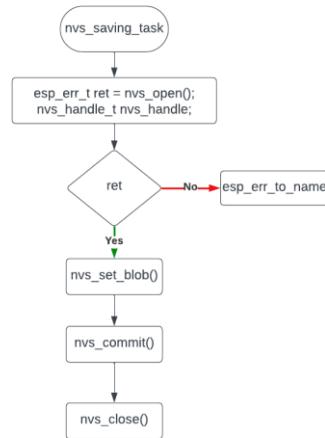


Ilustración 50. Escritura en la memoria flash NVS

1. La memoria se debe de abrir, indicando el modo de apertura que se desea. En este caso, como se desea escribir, “NVS_READWRITE”.
2. Una vez abierto, se añaden datos con “nvs_set_blob”.
3. Para almacenar los datos, se debe de llamar a la función “nvs_commit” para asegurarnos de que los datos son escritos en la memoria flash.
4. Una vez almacenado, se cierra la memoria flash con “nvs_close”

Por otro lado, para variables que necesiten sobrevivir a un reinicio o a un sueño profundo, Espressif recomienda el uso de la memoria RAM estática RTC del ESP32. Esta parte de la memoria sobrevive a los reinicios y al modo de sueño profundo, pero perderá el estado si se interrumpe la alimentación. Esta memoria no se desgastará al escribir en ella con frecuencia, y tampoco requiere un costo energético alto para acceder a la misma. Para poder almacenar valores en esta, basta con etiquetar la variable que se desee guardar con el atributo “RTC_DATA_ATTR”.

```
typedef struct{
    struct timeval sleep_enter_time;
}struct_rtcdata;
```

Capítulo 6

Resultados

En este capítulo se revisará el trabajo realizado con las diferentes APIs presentadas. Se han pretendido los objetivos iniciales, planteados en la introducción del proyecto, a saber:

- Inicio de una rutina de autoemparejamiento usando protocolo ESP-NOW a una pasarela maestra que está conectada a un servidor WEB y es capaz de enviar/recibir por MQTT datos desde y hacia dicho servidor.
- Obtención de datos y posterior procesado.
- Envío de los datos procesados a la pasarela maestra.
- Apagado del dispositivo para ahorro energético.

Además, debe de contar con varios extras necesarios para un funcionamiento óptimo, como son:

- ✓ Posibilidad de cambiar parámetros como el tiempo en el que el dispositivo está en “Deep Sleep” o sueño profundo, para ahorro energético.
- ✓ Posibilidad de actualización FOTA (“Firmware Over The Air”)
- ✓ Posibilidad de almacenamiento en memoria RTC/NVS de parámetros de conexión varios, como la MAC o datos de configuración, así como datos obtenidos durante la ejecución del código.

Se verá, por un lado, la aplicación y, por otro lado, diferentes comparaciones para ver si la aplicación utilizando el framework de ESP-IDF tiene mejoras con respecto al mismo en la plataforma de Arduino. Con respecto a la aplicación, se analizará el código realizado y se explicará el enfoque que se ha logrado implementar, puesto que, gracias a que se permite programar en C/C++, se pretendía alcanzar una aplicación que se pudiera adaptar a cualquier dispositivo.

6.1. Programación del dispositivo

Para que la programación de dispositivos sensores que usen nuestra infraestructura de comunicaciones y demás funcionalidades sea más sencilla, hemos optado por organizar el código de todas estas funcionalidades comunes en una clase C++. La idea es ocultar al programador los detalles del empleo de los protocolos de comunicación que hemos diseñado y que, si en un futuro hay que introducir mejoras o nuevas funcionalidades, el código programado para el dispositivo no necesite de cambios más allá de sustituir la implementación de nuestra librería (clase) autocontenido en un par de ficheros.

Capítulo 6 - Resultados

En el anexo II presentamos la estructura de ejemplo de aplicación que hemos desarrollado usando ESP-IDF, el framework de desarrollo para IoT de Espressif. Básicamente la aplicación tiene su código principal en un fichero main.cpp que será el que comienza a ejecutar cuando el dispositivo se enciende. Este código necesita incluir el fichero de cabecera de nuestra clase AUTOpairing_t y a partir de ahí, el dispositivo ya está listo para funcionar.

Tan sólo con declarar un objeto de nuestra clase e invocar la función begin() del mismo, comienza el emparejamiento con el router o pasarela que controla las comunicaciones. La primera vez que el dispositivo ejecuta, nuestra clase realiza una exploración para buscar la dirección y el canal donde está operando el router. A partir de ahí almacena esos datos en memoria no volátil para que las subsiguientes ejecuciones no necesiten realizar este proceso.

Para publicar datos, el dispositivo sensor simplemente llama a una función de nuestra clase indicando el tema (topic) asignado al mensaje y contenido del mensaje en sí. Si el dispositivo se ha unido a una red de área personal (PAN) ese mensaje también le llegará al resto de dispositivos vecinos en la misma red personal. Para unirse a la red el dispositivo simplemente tiene que usar una función de nuestra clase indicando la dirección PAN (número entero) que quiere usar.

El dispositivo puede proporcionar dos funciones propias para recibir mensajes, las configura simplemente usando sendas llamadas a nuestra clase. A partir de ese momento las funciones proporcionadas se invocarán respectivamente, cada vez que se reciba un mensaje de internet (redireccionado desde MQTT por el router/pasarela), y cada vez que se reciba un mensaje de un vecino de la red personal a la que se pertenezca.

De esta forma el programador que quiera utilizar nuestra infraestructura sólo se tiene que dedicar a programar la lógica que le permita obtener localmente los datos de sus sensores y las funciones que procesan la información que llega de los vecinos o de internet. Toda la gestión de esas comunicaciones en ambos sentidos la proporciona de forma totalmente transparente nuestra clase resultado de este trabajo.

La clase también proporciona funciones para almacenar y recuperar configuraciones del dispositivo de memoria no volátil (FLASH o RTC memory) y funciones para realizar actualizaciones del programa a través de WiFi (OTA) de forma sencilla. En el ejemplo de aplicación que hemos desarrollado el dispositivo es capaz de recibir mensajes desde una aplicación internet que le ordenan la actualización o que le envían parámetros de configuración como el tiempo de reposo en muy bajo consumo (deep-sleep) entre diferentes ejecuciones. Por último, mencionar que nuestra clase también automatiza el apagado del dispositivo (entrada en modo de ahorro de energía, deep-sleep) después de que todos los mensajes pendientes hayan sido enviados y recibidos.

Como se puede apreciar en esta breve exposición, todos los detalles de programación del dispositivo propias de Espressif, comunicación ESP-NOW, gestión de tareas FreeRTOS, sincronización de tareas, etc. están resueltas dentro de la clase y ocultas al programador que se

tiene que enfocar sólo en la funcionalidad que quiere darle a su dispositivo sensor y cómo integrar la información que se recibe del resto de dispositivos de su red personal.

El pseudocódigo del programa que hemos desarrollado a modo de ejemplo usando nuestra clase se puede ver a continuación:

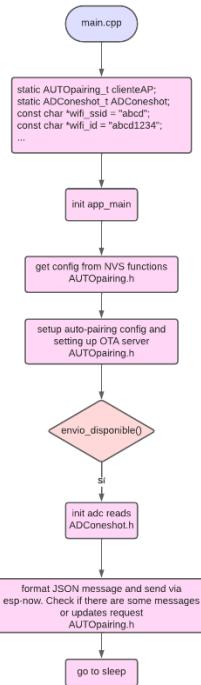


Ilustración 51. Estructura del programa principal

El código nos permite, a nivel de usuario, poder configurar los aspectos más importantes de la conexión, el SSID de punto de acceso al que queremos conectarnos cuando utilicemos WiFi y la contraseña, así como la URL desde donde tenemos alojado el archivo para la actualización OTA. Además, inicializamos la estructura que se usará para guardar los datos en la memoria no volátil. El funcionamiento es el siguiente:

1. Obtenemos la configuración de la memoria no volátil. Si no está inicializada, se dan valores por defecto y se actualiza la memoria. Inicializamos la estructura que tendrá los datos de los valores leídos por el conversor analógico-digital.
2. Inicializamos los parámetros más importantes de la conexión al servidor de actualización: la URL del archivo, el SSID y la contraseña. También indicamos el tiempo máximo de espera para la desconexión, en caso de que no se logre conectar con la pasarela. Inicializamos el tiempo en el que el dispositivo estará en el modo de sueño profundo, el canal en el que se empezará el escaneo, la identificación para la red de área personal “PAN” y la función “callback” para mensajes MQTT o de dispositivos que estén en la misma PAN.
3. Empezamos el escaneo. Si están los datos en memoria se recuperan, de lo contrario, se

inicia el escaneo automático.

4. Si se ha establecido la conexión, se prepara el escaneo de datos del convertidor analógico-digital, el mensaje con formato JSON y se envía. A la hora de enviar, se construye un mensaje en el que se incluyen unos bits iniciales que permiten indicarle a la pasarela si hay mensajes por parte del servidor o bien de otro dispositivo que esté dentro del PAN.
 - a. Si hay un mensaje entrante proveniente del servidor, se hace pasar a través de una función “callback” que procesará el mensaje que está en formato JSON. Si el mensaje es una petición de actualización, por ejemplo. se actualizará el firmware.
 - b. Si hay un mensaje entrante del PAN, este se procesará, sabiendo que el mensaje contiene los siguientes campos:

Tabla 5. Campos del mensaje PAN

Campo	Bytes
Tipo de mensaje	1
MAC	6
Antigüedad del mensaje (ms)	4
Cuerpo del mensaje en JSON	Hasta completar 250 bytes

El byte de control, el que indica el tipo de mensaje que es recibido, lo sigue la dirección MAC del dispositivo que envía el mensaje. Además, como pasa por la pasarela, queda registrado el instante en el que se envió dicho mensaje (un “timestamp”), útil para ver cuán útil es el mensaje. Después de este campo, sigue el cuerpo del mensaje, en formato JSON. La totalidad del mensaje es de 250 bytes, puesto que es la longitud máxima en bytes que se puede enviar por el protocolo de comunicación ESP-NOW.

5. Una vez enviado el mensaje y recibido el mensaje correspondiente, si procede, el dispositivo entrará en sueño profundo.

El resultado del programa final pasaría por ser una aplicación intuitiva y reconfigurable de cara a conectar múltiples dispositivos para intercambiar datos e interoperar para optimizar algún tipo de proceso que tengan que realizar en común, por ejemplo, algoritmos de optimización o de control que dependan de varios datos de dispositivos diferentes.

Por sí solo, un dispositivo simplemente envía datos a una pasarela y esta lo envía a un servidor para procesarlo, quedando como un registro individual desde el cual poder operar. Sin embargo, el hecho de interconectar dispositivos de la misma familia por medio de un protocolo de bajo consumo sin necesidad de utilizar Internet para que se puedan recibir los datos aumenta las posibilidades y las opciones de expandir a otras aplicaciones que requieran una mayor operatividad, versatilidad y precisión. Veamos cómo funciona la aplicación, en tiempo real.

Inicialmente la aplicación realiza un intento de conexión mediante una tarea que se ejecuta en un hilo de la CPU. Esta tarea realiza la actividad de autoemparejamiento estudiada en el apartado 4.2. Sin embargo, si la información de la pasarela se encuentra almacenada previamente en la memoria no volátil del microcontrolador, esta se iniciará. Esto aumentará el rendimiento del sistema al ser un emparejamiento más rápido, incrementando la eficiencia del dispositivo y durabilidad.

```
I (646) * mainApp: Emparejamiento recuperado de la memoria NVS del usuario
I (656) * mainApp: D9:47:D1:E6:E2:E0
I (656) * mainApp: en el canal 11 en 665.422974 ms
I (666) * funcion recivo: ADD PASARELA PEER
I (666) * funcion recivo: LIBERADO SEMAFORO ENVIO: EMPAREJAMIENTO
I (1076) * mainApp: LIBERANDO SEMAFORO CONEXION
```

Ilustración 52. Autoemparejamiento realizado por medio de los datos guardados en la memoria NVS

Una vez conectado, se produce la lectura de los datos. Este proceso realiza la calibración y la ejecución del ADC, obteniendo resultados en bruto o que pueden ser filtrados, en función de los requerimientos que nos encontremos en nuestro proyecto. Tras esta lectura, formateamos el mensaje con formato JSON y lo enviamos. Será un semáforo lo que nos indique que podemos enviar la información necesaria a la pasarela. En la imagen inferior se ve la formación del mensaje y el envío a la pasarela.

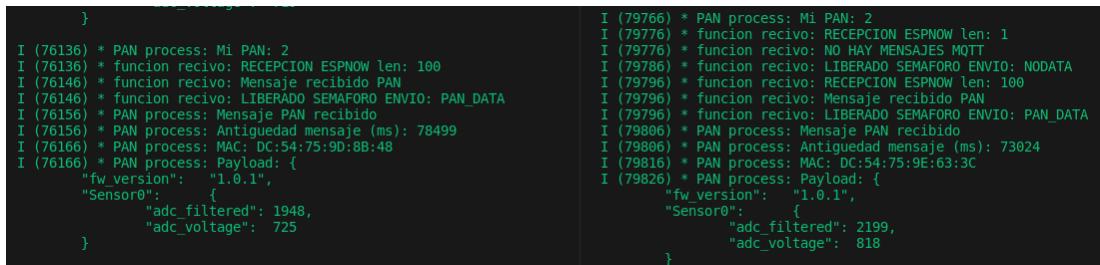
- ➔ Esperamos a que se haya realizado la conexión y enviamos una trama a la pasarela para ver si hay algún mensaje MQTT o PAN pendiente para recibir.
- ➔ Se envía nuestro mensaje y se espera un mensaje de recepción que indique si se ha realizado el envío satisfactoriamente o no.
- ➔ En caso de que no haya mensajes personales, desconectamos el dispositivo.

```
I (1326) * task conexion: ESPERANDO SEMAFORO ENVIO
I (1336) * task conexion: Sending message type = 10001001 DATOS PAN: 2 & SOLICITA MENSAJES
I (1336) * task conexion: Longitud del mensaje: 89
I (1346) * task conexion: mensaje: {
    "fw_version": "1.0.1",
    "Sensor0": {
        "adc_filtered": 1752,
        "adc_voltage": 655
    }
}

I (1356) * task conexion: ESPERANDO RESULTADO ENVIO
I (1356) * funcion envio: ENVIO ESPNOW status: OK
I (1356) * funcion envio: MAC: D9:47:D1:E6:E2:E0
I (1356) * funcion envio: >> Exito de entrega
I (1376) * task conexion: LIBERADO SEMAFORO ENVIO: FIN ENVIO
I (1386) * funcion recivo: RECEPCION ESPNOW len: 1
I (1396) * funcion recivo: NO HAY MENSAJES PENDIENTES
I (1406) * funcion recivo: LIBERADO SEMAFORO ENVIO: NODATA
I (1406) * funcion recivo: RECEPCION ESPNOW len: 1
I (1406) * funcion recivo: NO HAY MENSAJES PENDIENTES
I (1416) * funcion recivo: LIBERADO SEMAFORO ENVIO: NODATA
I (1416) * task conexion: ESPNOW_STATUS = 0
```

Ilustración 53. Comprobación de mensajes y envío de datos.

Otro aspecto fundamental es la recepción de mensajes PAN, con dispositivos que se encuentren en la misma red personal. Al enviar la trama de comprobación de mensajes, indicamos que también deseamos saber si hay mensajes que nos pertenezcan. Estos mensajes serán personales y contarán con el tiempo desde que fue recibido en la pasarela por el otro dispositivo, la MAC de origen y el contenido explícito del mensaje en cuestión. Con esta información en nuestro dispositivo, podemos obtener nuevas mediciones, realizando cálculos, ajustes, calibraciones, algoritmos de control o cualquier otro proceso para optimizar los datos que ya se hayan obtenido por cuenta propia. En la siguiente ilustración se puede observar cómo se ha recibido este mensaje, pudiendo a posteriori realizar cualquier operación relativa a los datos recibidos y los ya existentes en nuestro dispositivo.



The image shows two terminal windows side-by-side. Both windows have a black background with white text. The left window has a timestamp at the top: "2023-09-12T10:45:00Z". It contains several log entries starting with "I" followed by a timestamp and a message. The messages include receiving a PAN message from "Mi PAN", receiving a PAN message from "RECEPCION ESPNOW", sending a PAN message to "LIBERADO SEMAFORO ENVIO: PAN_DATA", and receiving a PAN message from "MAC: DC:54:75:9E:63:3C". The right window also has a timestamp at the top: "2023-09-12T10:45:00Z". It contains similar log entries, including receiving a PAN message from "Mi PAN: 2", sending a PAN message to "LIBERADO SEMAFORO ENVIO: NODATA", and receiving a PAN message from "MAC: DC:54:75:9E:63:3C". Both windows show a series of log entries with increasing timestamps, indicating a sequence of events over time.

Ilustración 54. Recepción de mensajes PAN en dos dispositivos diferentes

6.2. Análisis de los resultados obtenidos

Inicialmente expusimos una comparativa entre entornos tales como Arduino, bien conocido por su comunidad y su facilidad de uso para personas iniciadas en el mundo de la programación, y con el entorno que nos proporciona Espressif para programar sus dispositivos. Se ha diseñado una aplicación similar programada con el SDK de Espressif en Arduino, para ver si realmente hay alguna ventaja, más allá de las posibilidades que nos ofrece el propio entorno de desarrollo. Esta aplicación no cuenta con las funciones encargadas de actualizarse ni con el uso del ADC y, aunque en las pruebas que a continuación se muestran la aplicación diseñada con el SDK de Espressif sí cuenta con estas funcionalidades en el código, no se han utilizado, por lo que solo se tendrá en cuenta en el apartado relativo al tamaño de los ficheros.

Se pretende analizar ambos códigos en los siguientes terrenos: el tamaño de los binarios que se crean a la hora de depurar el código, tiempos de ejecución y rendimiento energético.

6.2.1. Comparación del tamaño de los binarios

En informática, el tamaño es fundamental para poder obtener una aplicación liviana y optimizar en aspectos de almacenamiento. Esto radica directamente en el coste de la memoria, la cual en algunos casos no puede ser superior a determinados megabytes (incluso kilobytes o algunos pocos bytes, según la aplicación). En este sentido, se podría pretender buscar una aplicación que genere un archivo que ocupe poco, permitiendo al dispositivo utilizar el resto de la memoria para otras funcionalidades.

Además, como se vio en el apartado correspondiente al estudio de la memoria del ESP32-C3, disponemos de unos 4MB, por lo que nos acota las dimensiones del fichero a este espacio. Se han compilado los dos programas sin ningún tipo de optimización de código.

➔ Tamaño en Arduino

Enfocándonos en el entorno de Arduino, y al depurar el código correspondiente a este, vemos que en la ventana inferior que nos proporciona el IDE de Arduino nos sale que se han utilizado 708550 bytes (generando un binario de unos 710KB) del espacio de almacenamiento del programa, un 54% del total, unos 1310720 bytes (aproximadamente 1,3MB). Las variables globales, por su lado, usan unos 37KB, un 11% de la memoria dinámica.

➔ Tamaño en Espressif SDK

Utilizando el entorno de Espressif, tenemos comandos como “idf.py size-components”, el cual nos permite saber el tamaño total de los componentes que estamos utilizando. En el caso del programa realizado en el entorno de desarrollo de Espressif, se ha incluido algunos componentes extras para que pueda funcionar el sistema al archivo “CMakeLists.txt”, como son los requerimientos de FreeRTOS, el WiFi o la memoria flash. Estos componentes, al añadirse, generan automáticamente un registro que, aunque no se use, ocupan un cierto espacio. Con esto, obtenemos los siguientes tamaños para el fichero generado.

```
Total sizes:
Used stat D/IRAM: 99102 bytes ( 222194 remain, 30.8% used)
  .data size: 10604 bytes
  .bss size: 18344 bytes
  .text size: 70154 bytes
Used Flash size : 931398 bytes
  .text: 755546 bytes
  .rodata: 175596 bytes
Total image size: 1012156 bytes (.bin may be padded larger)
```

Ilustración 55. Tamaño ocupado en la compilación del programa

El uso de la memoria RAM dinámica para datos e instrucciones del 30.8%, un 19.8% superior que en el IDE de Arduino. Por su parte, el binario ocupa casi 1MB de memoria.

6.2.2. Comparación de los tiempos de ejecución y rendimiento energético

Por otro lado, y dado que lo que se pretende es estudiar la eficiencia de los códigos, los mecanismos que tienen cada uno para ejecutarse, el “super-loop” de Arduino y la programación en tiempo real utilizando FreeRTOS en dos placas de la misma familia. Por un lado, un ESP32 con un microprocesador Tensilica Xtensa LX6 y el mismo ESP32-C3 con el procesador que incorpora como novedad, el RISC-V.

Los test se han realizado con los mismos dispositivos y sin sensores conectados al ADC. Para obtener el consumo, se han realizado pruebas en vacío de los ESP32 y ESP32-C3 conectándolo a un sensor de corriente de alta precisión INA219 y a una fuente de alimentación de 9V para los ESP32 y por USB (5V) en el caso de los ESP32-C3, ya que por USB los ESP32 no llegaban a funcionar junto con el INA219 (es recomendable, de hecho, alimentarlo con un voltaje que

Capítulo 6 - Resultados

esté entre los 6 o 7 voltios para impedir pérdidas de potencia en el regulador de tensión que trae incorporado. Los resultados han sido procesados en MATLAB.

→ Tiempos de ejecución y consumo en Arduino con ESP32-C3

En el entorno de desarrollo de Arduino nos encontramos con que el programa se ejecuta durante unos 460 milisegundos, tiempo durante el cual realiza la operación de configuración, emparejamiento y envío de los datos adquiridos.

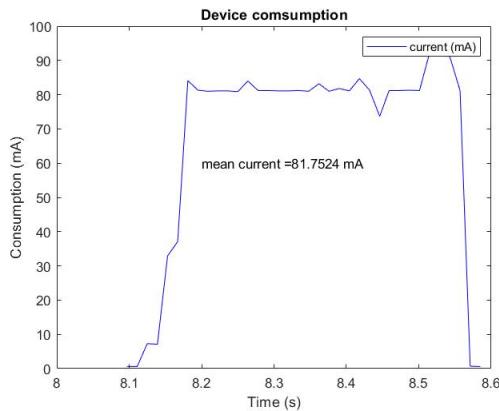


Ilustración 56. Consumo en entorno de Arduino con ESP32-C3 (1)

En la ilustración anterior se observa un gráfico que mide el consumo de corriente en miliamperios del dispositivo a lo largo de un ciclo de ejecución del programa. La corriente media en este tiempo de ejecución es de unos 81.75mA, después de lo cual se reduce prácticamente a cero al entrar en el modo de sueño profundo.

Un aspecto importante lo vemos en la siguiente ilustración, relativo a la potencia consumida, en miliwatios (mW). Normalmente, por las características de las baterías, se suele medir el consumo de corriente en los dispositivos en miliamperios (mA) para, con el tiempo, saber cuánto nos puede durar una batería en condiciones operativas. Sin embargo, la potencia también es útil conocerla, puesto que con esta podemos saber si, por ejemplo, el dispositivo gasta más o se calienta más.

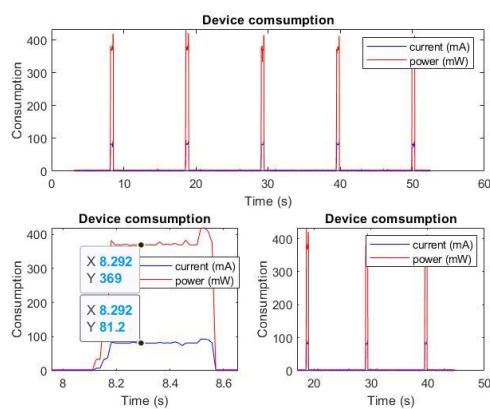


Ilustración 57. Consumo en entorno de Arduino con ESP32-C3 (2)

El dispositivo se alimenta a una tensión máxima de 5V por medio de un cable USB, una batería externa o un par de pilas AA. Incluso podríamos pensar en un sistema de alimentación por placas solares fotovoltaicas. Sin embargo, entre las pérdidas por el conductor, el propio sistema de alimentación y ruido, en general, al final llega una tensión nominal media de unos 4.2V. Si tenemos en cuenta un instante en el que la corriente sea de 81.2mA, obtendríamos una potencia de 369mW casi constante a lo largo de toda la ejecución del software. Y eso, a la larga, podría acabar afectando al dispositivo. Si bien es cierto que hay ciertos picos de alta intensidad correspondientes a los períodos de conexión al WiFi, que es por un instante muy pequeño, podría igualmente acabar afectando al dispositivo a la larga si no se protegiera adecuadamente.

La siguiente imagen es, en conjunto a la ilustración anterior, el proceso en el que se reciben los mensajes. Atendamos cuántos mensajes se envían, su momento y cuánto tardan en recibirse.

En la ilustración 57 notamos que hay cinco señales que indican que ha habido algún proceso de envío. En la imagen que sigue se puede observar los mensajes llegando al servidor con un tiempo entre los envíos de 10 segundo.

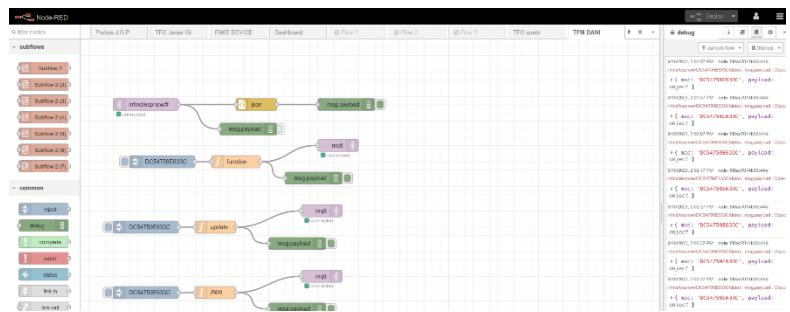


Ilustración 58. Recepción de mensajes MQTT con el programa de Arduino con ESP32-C3

➔ Tiempos de ejecución y consumo en Arduino con ESP32

Cuando programamos el ESP32 en el entorno de desarrollo de Arduino nos encontramos con que el programa se ejecuta durante 460 milisegundos, aproximadamente, tiempo durante el cual realiza la operación de configuración, emparejamiento y envío de los datos adquiridos. Por defecto, si no se utiliza o ejecuta tareas en RTOS definidas por el usuario, el ESP32 estará ejecutando las tareas en un solo núcleo a la máxima frecuencia de CPU, a unos 240MHz. Si bien es cierto que el ESP32-C3 tiene una frecuencia principal de 160MHz, sus tiempos son realmente competitivos. Sin embargo, en el aspecto de rendimiento y consumo no tiene nada que hacer frente al ESP32-C3.

Capítulo 6 - Resultados

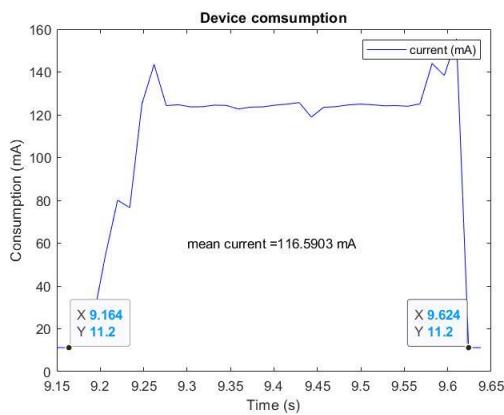


Ilustración 59. Consumo en entorno de Arduino con ESP32 (1)

En estado de sueño profundo la corriente consumida alcanza unos 11.2mA, con un consumo máximo de casi 140mA. La corriente media en ese tiempo de ejecución es de 116.59mA, unos 35mA más que con la versión de Arduino para el ESP32-C3. Comparándolo con su versión más nueva, se podría decir que este dispositivo nunca duerme.

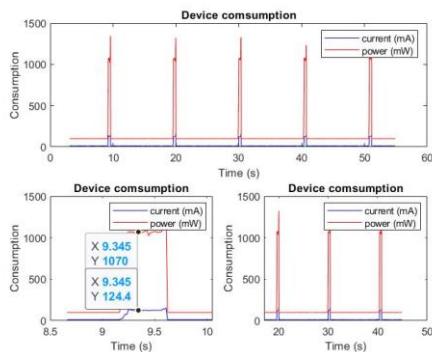


Ilustración 60. Consumo en entorno de Arduino con ESP32 (2)

De manera análoga y estudiando la potencia, con una potencia media de 116mA tenemos un consumo de entorno a unos 1000mW, unos 700mW más que en la versión de Arduino para el ESP32-C3 caso anterior. Este consumo para un dispositivo tan pequeño es inadmisible y, teniendo en cuenta que se desea un dispositivo que sea de bajo consumo, se hace realmente complicado encontrarle un hueco en el que se pueda incluir como dispositivo eficiente. Que la potencia alcance picos de unos cuantos watos conduce a que el sistema se caliente y se produzcan pérdidas en el mismo, sin mencionar la degradación del dispositivo a medio y largo plazo.

Así como se ilustró la recepción de los mensajes en los dos casos anteriores, esta vez se observa en el recuadro rojo de la imagen que está a continuación como han ido llegando los mensajes a la plataforma cada 10 segundos, aproximadamente.

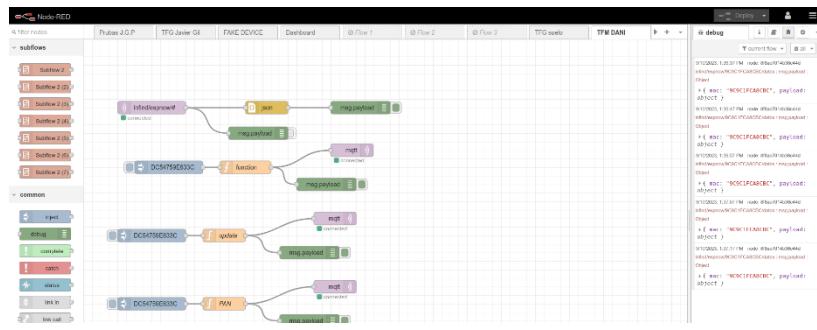


Ilustración 61. Recepción de mensajes MQTT con el programa de Arduino con ESP32

➔ Tiempos de ejecución y consumo en Espressif SDK con ESP32-C3

En el entorno de desarrollo de Espressif nos encontramos con que el programa se ejecuta durante unos 947 milisegundos, tiempo durante el cual realiza la operación de configuración, emparejamiento y envío de los datos adquiridos.

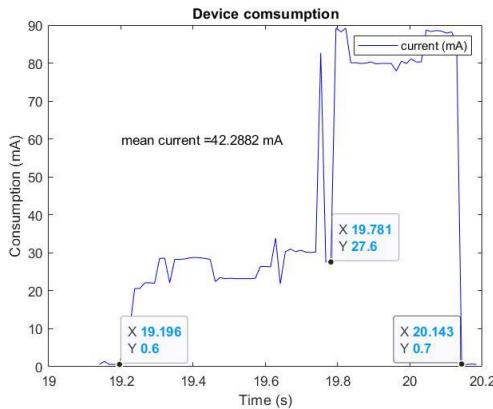


Ilustración 62. Consumo en entorno de ESP-IDF con ESP32-C3 (1)

En la ilustración anterior se observa un gráfico que mide el consumo de corriente en miliamperios del dispositivo a lo largo de un ciclo de ejecución del programa. La corriente media en este tiempo de ejecución es de unos 42.29mA, después de lo cual se reduce prácticamente a cero al entrar en el modo de sueño profundo. Es decir, prácticamente unos 40mA menos en cuanto a consumo. El desarrollo de la primera parte de la señal corresponde a la configuración de las tablas de particiones y configuraciones intrínsecas del microcontrolador. La segunda parte de la señal es la ejecución del programa, durante el cual se abre la memoria NVS, se extraen los datos, se procesan y se envían. Esta segunda parte se realiza en 362ms, similar al tiempo realizado en Arduino.

De manera análoga y estudiando la potencia, si tenemos en cuenta que la intensidad media es de unos 47mA, la potencia del dispositivo será de aproximadamente 200mW, prácticamente la mitad de potencia que en el caso anterior. Esta disminución de potencia, a la larga, incrementará la vida útil del dispositivo, así como su consumo. Si bien es cierto que hay ciertos picos de alta intensidad correspondientes a los períodos de conexión al WiFi, que es por un instante muy

Capítulo 6 - Resultados

pequeño, podría igualmente acabar afectando al dispositivo a la larga si no se protegiera adecuadamente.

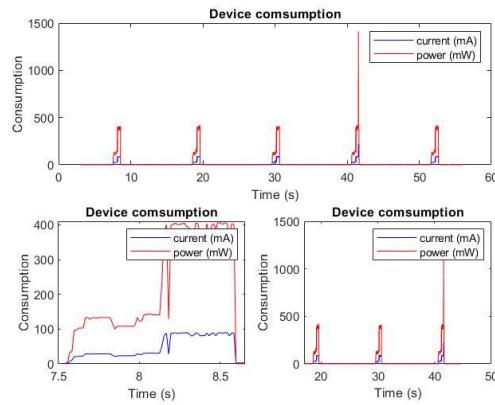


Ilustración 63. Consumo en entorno de ESP-IDF con ESP32-C3 (2)

Igualmente, analizando el momento en el que se reciben los mensajes podemos observar la fiabilidad y robustez del sistema. La siguiente imagen es, en conjunto a la ilustración anterior, el proceso en el que se reciben los mensajes. Atendamos cuántos mensajes se envían, su momento y cuanto tardan en recibirse.

En la ilustración anterior notamos que hay cinco señales que indican que ha habido algún proceso de envío. En la imagen que sigue se puede observar los mensajes llegando al servidor con un tiempo entre los envíos de 10 segundos.

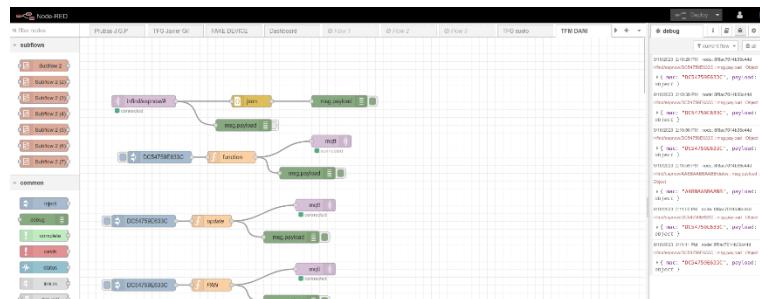


Ilustración 64. Recepción de mensajes MQTT con el programa de ESP-IDF con ESP32-C3

➔ Tiempos de ejecución y consumo en Espressif SDK con ESP32

Cuando programamos el ESP32 en el entorno de desarrollo de ESP-IDF nos encontramos con que el programa se ejecuta durante unos 1.2 segundos, similar al mismo tiempo obtenido para el ESP32-C3. Durante este tiempo realiza la operación de configuración, emparejamiento y envío de los datos adquiridos. Si bien es cierto que el ESP32-C3 tiene una frecuencia principal de 160MHz y un único núcleo, sus tiempos son realmente competitivos. Sin embargo, en el aspecto de rendimiento y consumo ha mejorado con respecto al caso anterior, en el que se ejecutaba el programa en el entorno de Arduino.

En estado de sueño profundo la corriente consumida alcanza unos 11.2mA, sigue sin irse a dormir del todo. La corriente media en el tiempo de ejecución es de 66.67mA.

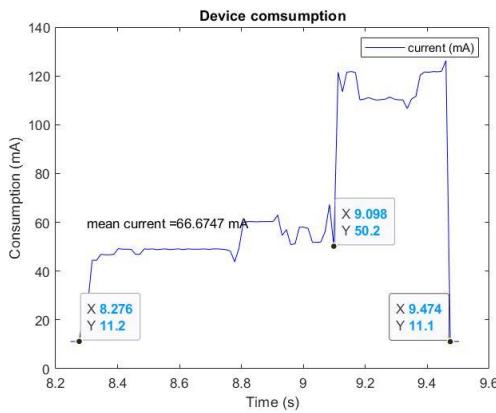


Ilustración 65. Consumo en entorno de ESP-IDF con ESP32 (1)

De manera análoga y estudiando la potencia, con un consumo medio de 66.67mA a lo largo del funcionamiento de la rutina obtenemos una potencia media de unos 600.03mW, casi 400mW menos que en la versión de Arduino para el ESP32. Este consumo se ve mejorado con respecto al esquema anterior, pero para un dispositivo tan pequeño es inadmisible y, teniendo en cuenta que se desea un dispositivo que sea de bajo consumo, se hace realmente complicado encontrarle un hueco en el que se pueda incluir como dispositivo eficiente.

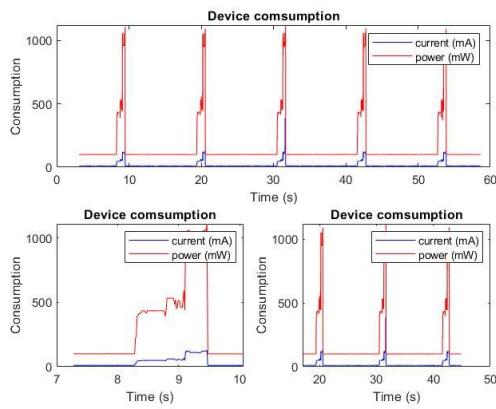


Ilustración 66. Consumo en entorno de ESP-IDF con ESP32 (2)

Así como se ilustró la recepción de los mensajes en los dos casos anteriores, esta vez se observa en el recuadro rojo de la imagen que está a continuación como han ido llegando los mensajes a la plataforma cada 10 segundos.

Capítulo 6 - Resultados

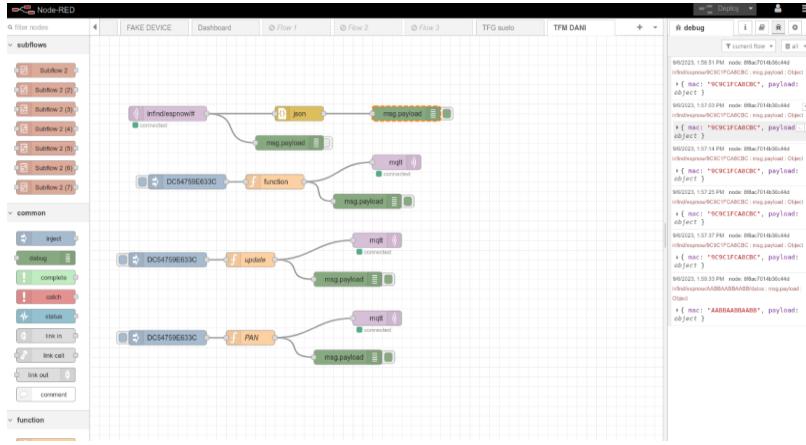


Ilustración 67. Recepción de mensajes MQTT con el programa de ESP-IDF con ESP32

En las gráficas que ilustran el consumo de corriente de los dispositivos ESP32 y ESP32-C3 usando ESP-IDF, se puede ver que la señal dispone de dos partes bien diferenciadas. La primera parte corresponde a la validación de la imagen firmware que se ha cargado el dispositivo a la hora de arrancarlo o despertarse de un sueño profundo.

```
I (24) boot: ESP-IDF v5.1-dev-4726-gdf9310ada2 2nd stage bootloader
I (24) boot: compile time Sep 11 2023 00:39:40
I (25) boot: chip revision: v0.4
I (29) boot.esp32c3: SPI Speed : 80MHz
I (34) boot.esp32c3: SPI Mode : DIO
I (38) boot.esp32c3: SPI Flash Size : 4MB
I (43) boot: Enabling RNG early entropy source...
I (49) boot: Partition Table:
I (52) boot: #> Label Usage Type ST Offset Length
I (59) boot: 0 nvs WiFi data 01 02 00000000 00000000
I (67) boot: 1 phy init RF data 01 01 00000000 00000000
I (74) boot: 2 factory factory app 00 00 00000000 00177000
I (82) boot: End of partition table
I (86) esp_image: segment 0: paddr=00010020 vaddr=3c0c0020 size=27060h (161376) map
I (118) esp_image: segment 1: paddr=00037688 vaddr=3fc91000 size=027ch ( 10188) load
I (128) esp_image: segment 2: paddr=00039e5c vaddr=40388000 size=001bch ( 25028) load
I (128) esp_image: segment 3: paddr=00040020 vaddr=42000000 size=b1238h (72556) map
I (239) esp_image: segment 4: paddr=000f1268 vaddr=403861bc size=0d40h ( 44352) load
I (247) esp_image: segment 5: paddr=000fbfa8 vaddr=50000000 size=00028h (   40) load
I (251) boot: Loaded app from partition at offset 0x10000
I (251) boot: Disabling RNG early entropy source...
I (268) cpu_start: Unicore app
I (268) cpu_start: Pro cpu up.
I (276) cpu_start: Pro cpu start user code
I (276) cpu_start: cpu freq: 160000000 Hz
I (276) cpu_start: Application information:
I (279) cpu_start: Project name: espnow_project
I (285) cpu_start: App version: 1.0.1
I (298) cpu_start: Compile time: Sep 11 2023 00:39:37
I (296) cpu_start: ELF file SHA256: a210bd6d770d74b6...
I (302) cpu_start: ESP-IDF: v5.1-dev-4726-gdf9310ada2
I (308) cpu_start: Mini chip rev: v0.3
I (313) cpu_start: Max chip rev: v0.99
I (318) cpu_start: Chip rev: v0.4
I (323) heap_init: Initializing, RAM available for dynamic allocation:
I (330) heap_init: At 3FC07F50 len 0000447C0 (273 KiB): DRAM
I (336) heap_init: At 3FC0D710 len 000002950 (10 KiB): STACK/DRAM
I (343) heap_init: At 500000038 len 00001FC8 ( 7 KiB): RTRAM
I (350) spi_flash: detected chip: generic
I (354) spi_flash: flash io: dio
I (358) sleep: Configure to isolate all GPIO pins in sleep state
I (364) sleep: Enable automatic switching of GPIO sleep configuration
I (371) app_start: Starting scheduler on CPU0
I (376) main_task: Started on CPU0
I (376) main_task: Calling app_main()
```

(a)

(b)

Ilustración 68. (a) Validación de la imagen en ESP-IDF. (b) Continuación de la validación e inicio del programa

En esta etapa, se comprueba toda la configuración que se le ha dado al dispositivo, así como la gestión de la tabla de particiones e inicializaciones como de la CPU o del HEAP.

Esta comprobación se puede desactivar en el menú de configuración de ESP-IDF en la opción “Skip image validation when exiting deep sleep”:

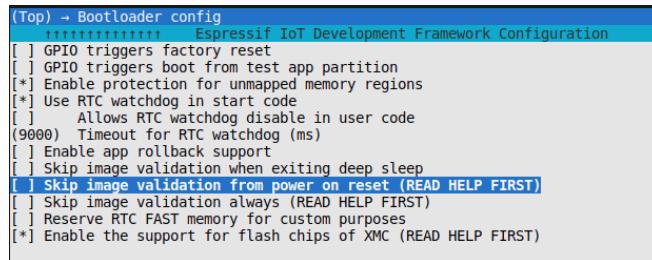


Ilustración 69. Opción para impedir que se realice la validación de la imagen al resetear en ESP-IDF

Esta comprobación inicial puede durar entre 400 y 600 ms.

La segunda parte de la señal corresponde al funcionamiento de la rutina programada. Esto es así porque es el momento en el que se activa el módulo WiFi para dar servicio a ESP-NOW, elevando el consumo hasta casi los 80mA. Esta segunda parte tarda 362ms para el dispositivo ESP32-C3 y 376ms para el dispositivo ESP32.

A modo de resumen de los resultados obtenidos, se muestra la siguiente tabla. Se observa que, en general, el rendimiento en ESP-IDF es bastante mejor. Hay mejoras en cuanto a tiempos de ejecución y consumo, llegando a reducirse considerablemente los tiempos.

Tabla 6. Comparativa de los dispositivos que se han sometido a prueba

Device	Framework	Execution time (ms)	Average consumption (mA)	Max. power (mW)
ESP32-C3	Arduino	460	81.75	369
ESP32		465	116.59	1000
ESP32-C3	ESP-IDF	947 (529 + 362)	42.29	200
ESP32		1100 (822 + 376)	66.67	600

- ➔ La primera columna considera el tiempo total del dispositivo desde que se despierte del sueño profundo hasta que se vuelve a apagar. En el caso de los dispositivos programados en ESP-IDF, el tiempo se ha dividido entre la comprobación del firmware inicial y la ejecución del programa.
- ➔ La segunda columna ilustra el consumo de corriente medio a lo largo del tiempo total del dispositivo, desde que este se despierta del sueño profundo hasta que se vuelve a apagar.
- ➔ La última columna indica la potencia consumida, en la que se ha tenido en cuenta la corriente media consumida y la tensión a la que se están alimentando los dispositivos, mediante USB 5V (4.2V nominales) para el ESP32-C3 y 9V para el ESP32.

Capítulo 6 - Resultados

La realidad de los procesadores ARC, MIPS y Tensilica ocupan nichos específicos. En cuanto al desarrollo de esta arquitectura no consideraron cuanta energía entregar puesto que, al disponer de varios núcleos de CPU ARM, su gasto energético se dispara. Por un lado, el núcleo del ESP32 es un Tensilica Xtensa LX6 de 32 bits con doble núcleo que opera entre 160 o 240 MHz. Por otro lado, el núcleo del ESP32-C3 es un RISC-V con un único núcleo de 32 bits y con una frecuencia máxima de 160 MHz. Este procesador se utiliza como procesado de consumo ultrabajo (*Ultra-Low-Power Processor*, ULP), lo que implica que se puede utilizar incluso si está completamente apagado, lo que permite que el chip absorba energía y continúe trabajando a muy bajo coste energético. Esto implica que el procesador no disponga de una alta velocidad de procesamiento, puesto que al ser un único núcleo no podrá exigirle una carga desmesurada de trabajo.

En la tabla observamos como justamente el consumo entre un dispositivo y otro se dispara, al no tener el control sobre los recursos que consume el microcontrolador. El ESP32 lo va a dar todo a una alta velocidad, inclusive si ello implica usar sus dos núcleos en tareas que lo requiera, a un coste energético inapropiado para tareas que requieran de un bajo consumo. Por otro lado, el ESP32-C3, aun teniendo menos velocidad y un único núcleo, sabe optimizar sus recursos, brindándonos un dispositivo funcional incluso con un gasto energético mínimo.

En cuanto a tiempos, el uso de la herramienta ESP-IDF junto con FreeRTOS y usando la misma filosofía que en el código de Arduino, es decir, sin optimización con las herramientas de ESP-IDF, es similar. Tal vez la diferencia más notable viene en el apartado energético. El uso de una plataforma u otra es fundamental, puesto que se han reducido los consumos un 42.48% para el caso del ESP32-C3 y un 42.81% en la versión del ESP32. Es decir, se ha reducido el consumo en prácticamente la mitad. Con respecto a los tiempos, teniendo en cuenta el tiempo de la rutina programada, al usar ESP-IDF ha conseguido reducir un 21.3% con respecto a la versión de Arduino.

Capítulo 7

Conclusiones

En este capítulo se desarrollarán algunas conclusiones a las que podemos llegar con las pruebas que se han ido realizando a lo largo de la vida de este proyecto.

7.1. Conclusiones e implicaciones del proyecto

Hemos conseguido cumplir con los objetivos planteados al comienzo de este trabajo. El objetivo principal consistía en estudiar las capacidades y versatilidad de un dispositivo conectado IoT de muy bajo consumo para la implementación de redes de sensores auto-configurables y capaces de compartir información en el borde. Escogimos el nuevo módulo ESP32-C3 con micro-arquitectura RISC-V, para evaluar sus características comparándolas con versiones anteriores de esta familia de dispositivos basadas en otras micro-arquitecturas (RISC, Tensilica Xtensa). También hemos comparado dos modelos de programación bien diferenciados, el hyper-loop clásico de Arduino y la programación de tareas en tiempo real que ofrece un sistema operativo de tiempo real como el que proporciona ESP-IDF, framework de programación basado en FreeRTOS.

Esta infraestructura que hemos desarrollado, servirá para implementar fácilmente aplicaciones que permitan explotar la capacidad de cálculo de los dispositivos IoT desplegados en el borde para procesar e integrar la información en el mismo lugar donde se produce: la red de sensores. Se ha estudiado un entorno y un paradigma de programación diferente orientado al mundo del IoT y, en particular a los dispositivos de Espressif ESP32. Se han desarrollado varios programas de test y prueba, así como topologías de circuitos diferentes para poder controlar de forma más eficiente el periodo en el que algún sensor está activado. Los programas que se han ido desarrollando han ido pasando desde un primer estado primitivo, en el que se estaba iniciando al mundo de la programación en tiempo real en el lenguaje C. Otra fase, por comentar, pasó por el desarrollo de la aplicación siguiendo los consejos y recomendaciones que nos propone Espressif. Finalmente, tras haber pasado tiempo estudiando y revisando los códigos anteriores, se ha planteado una solución eficiente en C++ utilizando clases. Este punto final permite modularidad, versatilidad, robustez y sobre todas las cosas, eficiente. Era importante el punto de que la aplicación tuviera una interfaz sencilla de configurar, más allá del código generado, y se ha logrado, como también se ha logrado cierta robustez a la hora de enviar y recibir mensajes.

Sin embargo, el estudio de la eficiencia energética ha resultado en algo realmente importante, puesto que estábamos llevando al estrado a dos dispositivos iguales, con la misma filosofía de código, pero con diferente implementación. Se ha visto que programar con Arduino resulta más

sencillo para un usuario medio. Eso radica en una curva de aprendizaje menos escarpada y asequible para proyectos en los que no sean necesarios requerimientos energéticos. Sin embargo, si necesitamos un producto que requiera una batería y esto sea un elemento determinante para asegurar una buena durabilidad, tendríamos que prescindir de Arduino y plantearnos programar en tiempo real, usando por ejemplo el entorno propuesto por Espressif. Se ha visto que el consumo eléctrico ha disminuido un 42,7% en la corriente que necesita el dispositivo, con un incremento en la velocidad de hasta tres segundos. Además, se calienta menos, aumentando la durabilidad del dispositivo en entornos algo agresivos con la temperatura. Si bien es cierto que la curva de aprendizaje es más compleja, resulta conveniente para proyectos en los que el tiempo y la energía son requerimientos importantes del sistema.

Con respecto a los tiempos, teniendo en cuenta el tiempo de la rutina programada, al usar ESP-IDF ha conseguido reducir un 21.3% con respecto a la versión de Arduino.

Esto indica que el uso óptimo y control de las características de un dispositivo potente se traduce en un incremento de la eficiencia del mismo. Sin lugar a dudas, esta herramienta conjunto al procesador RISC-V han aumentado las expectativas de bajo consumo y potencia al realizar cálculos poco exigentes.

Finalmente, y a nivel personal, la realización de este proyecto ha implicado la adquisición de nuevos conocimientos y habilidades sobre aspectos relativos a la programación en tiempo real, el universo cada vez más creciente del Internet de las Cosas y el conocimiento apropiado para aplicarlo. Con el análisis de los distintos entornos y paradigmas de programación he concluido en qué aspectos resulta más conveniente utilizar un entorno u otro, así como elaborarlo todo desde cero ha sido también un reto, desde el proceso en el que se idea hasta la implementación en sí misma.

7.2. Líneas futuras

Este proyecto se ha planteado con la idea futura de estudiar con el procesador RISC-V varios algoritmos de fusión sensorial y programación de infraestructuras de red de sensores inalámbricos auto-configurables para fusión de datos en el borde. La línea futura más inmediata, tras este primer análisis preliminar, debería de ser la implementación de algoritmos que permita realizar la fusión de datos a través de varios dispositivos que pertenezcan a la misma red de área personal.

Sin dudas, una línea futura es la depuración del código y seguir mejorando el rendimiento de la aplicación, así como métodos más eficientes de emparejamiento y conectividad. Estudiar aspectos como el retardo, gestión de mensajes, seguridad y robustez en el protocolo de comunicación es una tarea inmediata, así como realizar pruebas con aplicaciones que requieran más complejidad computacional para evaluar la eficacia de estos núcleos de ultra bajo consumo. Los aspectos físicos y de hardware también son mejorables, y se irán depurando en trabajos futuros.

Anexos

Anexo I. Entorno de desarrollo y configuración

Cuando nos adentramos en el IDE de Eclipse, la apariencia que tiene es similar al de otros softwares de desarrollo embebido del tipo VS Code o Atmel Studio. En la ilustración 68 se ve la pantalla del programa, donde tenemos diferentes partes:

A la derecha en rojo tenemos el directorio principal, donde están los ficheros y archivos de configuración, así como los generados en el proceso de construcción del programa (“build”). A diferencia de Arduino, aquí podemos tocar directamente la configuración del dispositivo a programar, su comportamiento, añadir o quitar librerías o incluso monitorear en tiempo real el código para su depuración.

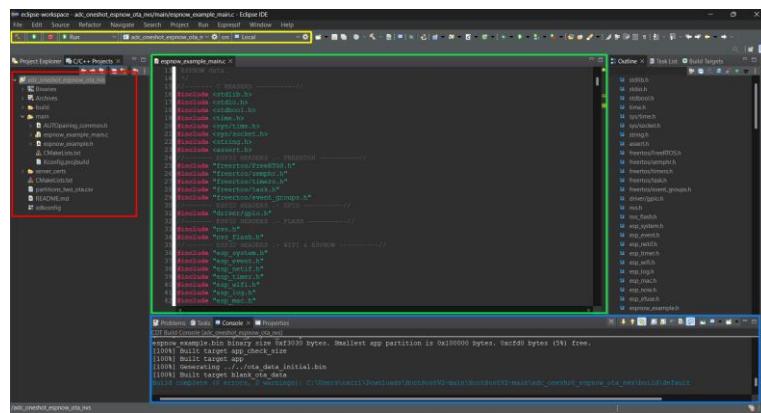


Ilustración 70. Entorno de programación de Eclipse C/C++

En la barra de herramientas de la parte superior, en amarillo, tenemos diferentes botones que nos permite depurar y montar el código en nuestro dispositivo (“build & run”). Es importante haber configurado previamente el objetivo del compilador si está en modo de carga de firmware. Esto se puede realizar en la siguiente pestaña:

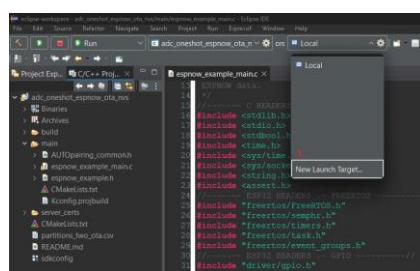


Ilustración 71. Selección de dispositivo a utilizar

Anexos - Anexo I. Entorno de desarrollo y configuración

En este menú, debemos escoger “ESP Target” y seleccionar nuestro dispositivo. Una vez hecho esto, se podrá cargar el software realizado.

El código se programa en la pestaña que se abre en el medio, en la ilustración 68 en verde. Es importante aclarar que, al estar programando en C, debemos de importar todas las librerías necesarias. Es interesante hacerse librerías propias, como se verá más adelante.

Finalmente, señalado en un recuadro azul, observamos la pestaña de depuración donde nos saldrán los avisos o errores que vayan apareciendo. También nos permite configurar y ver el puerto serial JTAG del dispositivo para ver aspectos del código que se desee depurar.

Antes de continuar con el siguiente apartado, donde se pretende hacer una abstracción al código e ir explicando las diferentes APIs y metodologías seguidas para el funcionamiento del dispositivo, se pretende ilustrar la configuración previa que se debe de hacer para lograr determinadas funcionalidades extras, como son, por ejemplo, la posibilidad de realizar una actualización y conectarse a internet.

Observando el directorio raíz, se ven varios ficheros.

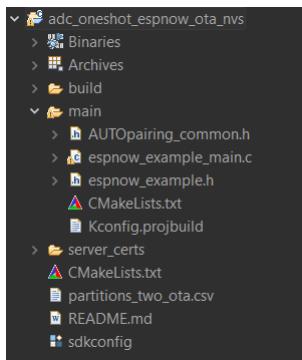


Ilustración 72. Fichero raíz del proyecto.

Estudiemos punto a punto qué es y para qué sirve cada documento.

- ➔ La carpeta “build” contiene, entre otras cosas, todos los archivos generados a la hora de compilar el programa principal. Además, contiene (si se ha seleccionado), el binario que nos hará falta para realizar la actualización OTA. Contiene todas las librerías que se han incluido en el proyecto y directorios necesarios para su funcionamiento.
- ➔ La carpeta “main” contiene el archivo principal en formato “.c” o “.cpp” según el lenguaje en el que se esté programando. También hay cabeceras personales creadas para un fin específico. Es notable ver que hay un archivo “CMakeList.txt” que contiene información sobre los directorios y las fuentes de las cabeceras incluidas en el código fuente.

- ➔ En el directorio raíz tenemos un “CMakeList.txt” genérico, que apunta a todo el proyecto. Su contenido es el siguiente:

```
# The following lines of boilerplate have to be in your project's CMakeLists
# in this exact order for cmake to work correctly
cmake_minimum_required(VERSION 3.16)
# This example uses an extra component for common functions such as Wi-Fi and Ethernet
# connection.
set(EXTRA_COMPONENT_DIRS
$ENV{IDF_PATH}/examples/common_components/protocol_examples_common)
set(PROYECT_VER "2")
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(espnnow_example)
```

En este archivo de compilación tenemos que asegurarnos de varias cosas:

1. Tenemos que tener apuntado el directorio principal de los componentes instanciados en el proyecto, si los hubiera. Estos son archivos de configuración o códigos fuente que nos permite configurar algo en particular.
2. Para la actualización FOTA tenemos que colocar una versión de nuestro proyecto. Esto lo podemos hacer con el comando “set(POJECT_VER “num”)”. Esto se comprobará cada vez que se ejecute la rutina de actualización y, de ser igual, no se actualizará.
3. Se incluirán todas las cabeceras que se necesiten en el proyecto.
4. Se le dará nombre al proyecto.

Otro archivo que tenemos para configurar el proyecto, en concreto el SDK, es el “sdkconfig”. Con el IDE Arduino esto se hacía de forma automática, aunque había algunos parámetros que sí que se podían configurar. Con este archivo, configuraremos completamente el comportamiento del dispositivo.

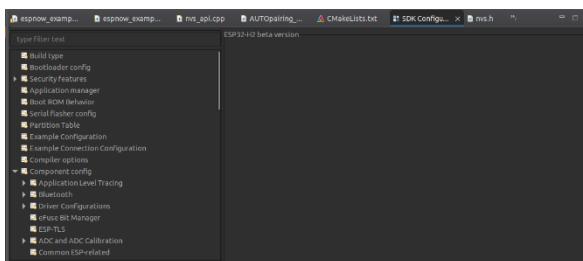


Ilustración 73. Vista general del archivo "sdkconfig"

Particularizando en este proyecto, debemos de configurar algunos puntos para lograr una mejor funcionalidad. En primer lugar, para la posibilidad de actualizar por medio de FOTA, debemos de permitir conexiones HTTP al servidor donde se aloja el binario. Para ello nos vamos al punto “ESP HTTPS OTA” y verificamos que la casilla “Allow HTTP fot OTA” esté validada.

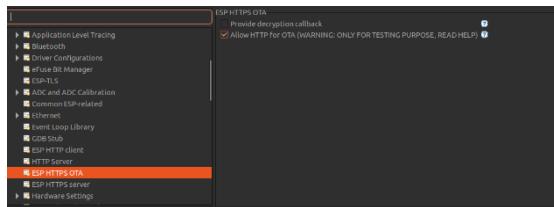


Ilustración 74. Pestaña de validación de conectividad HTTP para FOTA

En el caso de que nuestro servidor de alojamiento del binario tenga certificado, se lo tendremos que pasar al dispositivo para que se valide. De lo contrario y siempre que tengamos el control de la red (es decir, un entorno privado) debemos de invalidar la seguridad de la conexión y la validación de los certificados TLS. Por ello, tenemos que asegurarnos que las opciones “Allow potentially insecure options” y “Skip server certificate verification by default” estén activadas en la pestaña ESP-TLS, dentro de “Component config”.

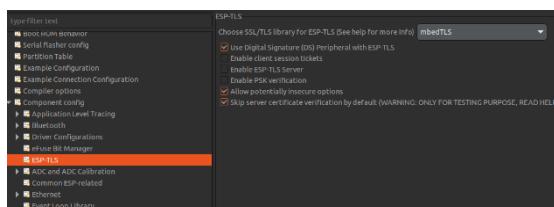


Ilustración 75. Ininvalidación certificado TLS

Finalmente, en el apartado “Partition table” [14] debemos de seleccionar cómo queremos que se almacenen los datos en nuestro dispositivo. Esto es algo potente puesto que podemos controlar el espacio y su gestión nosotros mismos, en función de nuestras necesidades. Por ejemplo, como deseamos realizar actualizaciones OTA de forma esporádica, tenemos que asegurarnos de almacenar los datos del nuevo firmware en algún lugar. Por defecto, la opción de tabla de particiones suele estar en “Single factory app, no OTA”, lo cual quiere decir que no hay espacio para almacenar información relativa al firmware nuevo. De quererlo, podríamos hacer una partición para tal propósito.

Por suerte, Espressif nos permite hacerlo de varias maneras.

1. Por un lado, tienen la opción “Factory app, two OTA definitions”. Esta primera opción nos permite tener una partición primaria denominada “factory”, donde almacenaremos la aplicación por defecto. Además, traerá dos particiones OTA, del mismo tamaño que la partición “factory”, donde se almacenará el nuevo firmware. Para saber qué aplicación inicializar, el gestor de arranque consulta el registro con etiqueta “otadata” que contiene las actualizaciones OTA. En función de su contenido, se ejecutará la aplicación por defecto “factory” o el de las particiones OTA.

2. Por otro lado, nos permite realizar nosotros mismos una tabla de particiones, pudiendo seleccionar los parámetros que mejor se adapten al uso que le daremos a la aplicación. Esta tabla será un fichero generado por nosotros mismos en formato CSV en el que se contengan los siguientes puntos:
 - 2.1. Campo de nombre: se especificará la etiqueta que tendrá esa partición. No es algo relevante para el funcionamiento del ESP32, pero sí que, como todo, sea clarificativo. Lo que se debe de tener en cuenta es que no debe de superar 16bytes de longitud, incluyendo el carácter de terminación.
 - 2.2. Campo de tipo: se debe de determinar si será un campo de aplicación (“app”, 0x00) o de datos (“data”, 0x01). También pueden ser números de 0 a 254 (0x00 a 0xFE), aunque los tipos del 0 al 63 (0x00 al 0x3F) están reservados para funciones internas del ESP32.
 - 2.3. Campo de subtipo: campo de 8 bits de longitud que especifica la tipología dada a una partición del tipo “app” o “data”.
 - 2.3.1. Si el tipo es “app”, los subtipos que se pueden especificar serán “factory” (0x00), “ota_0” (0x01) hasta “ota_15” (0x1F) o “test” (0x20). El subtipo “factory” es donde se almacena la aplicación por defecto y la primera que inicia el gestor de arranque, a menos que exista una partición del tipo data/ota, en cuyo caso se leerá esa partición para determinar que partición se debe de arrancar.
 - 2.3.2. Si el tipo es “data”, los subtipos que se pueden especificar serán “ota” (0x00), “phy” (0x01), “nvs” (0x02) y “nvs_keys” (0x04). Los más relevantes son el subtipo “ota”, que almacena la información sobre la partición OTA seleccionada y “nvs”, que es la partición dedicada al “Non-Volatile Storage” (o bien, almacenamiento no volátil).
 - 2.4. Finalmente debemos de darle un offset, es decir, el punto de memoria a partir del cual empieza la partición y un tamaño de partición total.

Una sola memoria flash del ESP32 puede contener varias aplicaciones, así como muchos tipos diferentes de datos (datos de calibración, sistemas de archivos, almacenamiento de parámetros, etc.). Por esta razón, una tabla de particiones se actualiza con un offset predeterminado de 8kB (0x8000 o 32768 en decimal) en la actualización.

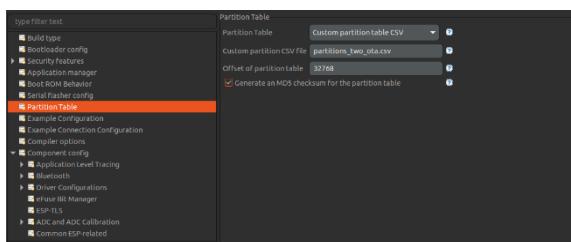


Ilustración 76. Selección de tabla de partición personalizada.

Anexos - Anexo I. Entorno de desarrollo y configuración

En nuestro caso, se ha dispuesto de una tabla de partición con las siguientes características, suficientes para almacenar y gestionar todas las variables, procesos y parámetros que se deseen, así como para darle la funcionalidad de actualizarse vía OTA.

Tabla 7. Configuración de la tabla de particiones.

Nombre	Offset	Tamaño
nvs	0x9000	4kB
otadata	0xd000	2kB
phy_init	0xf000	1kB
factory	0x10000	1MB
ota_1	0x110000	1MB
ota_2	0x210000	1MB
storage	0x310000	80kB

Como se observa en la siguiente imagen, que muestra el proceso después de una actualización, se ve como se ha gestionado la tabla de particiones y la imagen nueva se ha almacenado en la partición “ota_1”. Después de esto, se carga el contenido de dicha partición con la versión del firmware recién añadida.

```
I (64) boot: Partition Table:
I (68) boot: #& Label           Usage      Type Sf Offset  Length
I (75) boot: 0 nvs              WiFi data   01 02 00009000 00004000
I (83) boot: 1 otadata          OTA data    01 00 00000000 00002000
I (90) boot: 2 phy_init         RF data    01 01 00007000 00001000
I (98) boot: 3 factory          factory app 00 00 00010000 00100000
I (105) boot: 4 ota 0            OTA app    00 10 00100000 00100000
I (113) boot: 5 ota 1            OTA app    00 11 00210000 00100000
I (120) boot: 6 storage          unknown data 01 82 00310000 00080000
I (128) boot: End of partition table
I (132) esp image: segment 0: paddr=00110020 vaddr=3c0c0020 size=279a0h (162208) map
I (133) esp image: segment 1: paddr=001379c8 vaddr=3fc90c00 size=024dch ( 9436) load
I (134) esp image: segment 2: paddr=00139eac vaddr=40380000 size=001ech ( 24940) load
I (161) esp image: segment 3: paddr=00140e20 vaddr=42000020 size=b12ch (727852) map
I (221) esp image: segment 4: paddr=001f1054 vaddr=40380100 size=00900h ( 43272) load
I (226) esp image: segment 5: paddr=001f4e04 vaddr=500000010 size=00010h ( 16) load
I (227) boot: Loaded app from partition at offset 0x110000
```

Ilustración 77. Gestión de nuevo firmware tras actualización FOTA

Anexo II. Estructura de la aplicación desarrollada en ESP-IDF

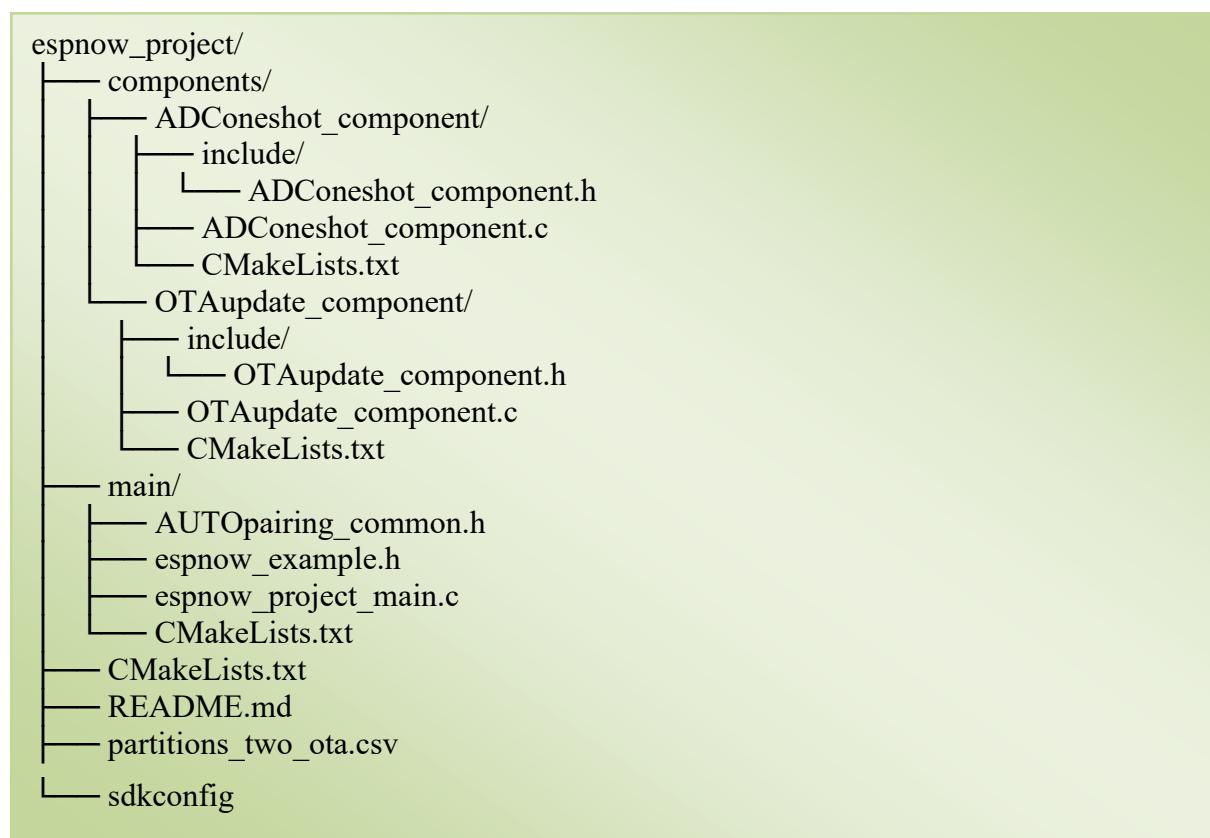
Con el framework de ESP-IDF se trabaja de manera nativa en C, aunque se permite realizar códigos con C++. Para esto último, bastaría con nombrar el archivo a “.cpp” y recordarle al compilador que se va a utilizar dicho lenguaje. Para esto último, debemos de añadir al archivo de configuración “CMakeLists.txt” la información de la compilación que queramos que realice, así como especificar dentro de nuestro código fuente que el “app_main” será programado en C++.

- ➔ Para añadir la información relativa al compilado, basta con añadir al “CMakeLists.txt” general la línea “set(CMAKE_CXX_STANDARD 17)”, especificando la versión que se quiera utilizar.
- ➔ Para el código fuente, como por defecto se trabaja con C, se debe indicar en el documento que dicha función se define en otro lugar y usa la convención de llamada del lenguaje C. Para esto, el modificador extern “C” también se puede aplicar a varias declaraciones de función en un bloque. En el archivo “main.cpp”, al inicio, hay que añadir la siguiente instrucción:

```
extern "C"
{
    void app_main(void);
}
```

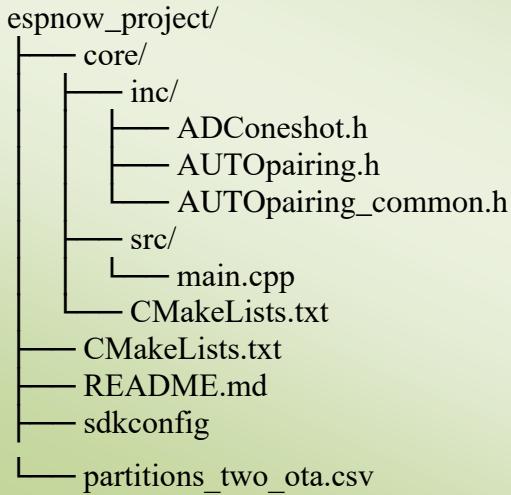
Teniendo en cuenta estos puntos, y a modo de buenas prácticas, se organiza el trabajo en carpetas para tener identificado adecuadamente cada componente personal. Se recomienda tener una estructura similar a la que se muestra para que el código sea exportable y que sean pocas las configuraciones que se tengan que realizar en el código fuente del proyecto.

En un primer boceto, la estructura raíz del directorio realizado en C acabó teniendo la siguiente estructura:



Sin embargo, la ventaja de programar en C++ es que permite utilizar un paradigma de programación orientada a objetos. Esto es una metodología especial de programación software en la que un programa informático se diseña como la interrelación entre un conjunto de instancias conocidas como objetos. Estos objetos son entidades que tienen un estado que está descrito por sus variables internas que se conocen como atributos o propiedades y que tienen comportamientos que están plasmados en funciones o métodos que se utilizan para manipular las variables que describen el estado del objeto. La programación orientada a objetos al igual que la programación estructurada sigue el método de programación imperativa, es decir, el programador escribe los métodos de forma que se a través de un algoritmo se dicta la funcionalidad de los métodos. Permite, entre otras cosas, realizar una programación orientada a la multiplataforma, reutilización de código. De la misma manera que se ha realizado un esquema en C, se puede presentar la misma distribución para un código que se ensambla en C++.

La estructura raíz del proyecto tiene la siguiente estructura:



Las funciones que se encargaban, por ejemplo, de la conexión por ESP-NOW al dispositivo que hacía la función de pasarela, así como el proceso de configuración o ajustes previos y el proceso de actualización OTA, se ha incluido en una clase dentro de la librería “AUTOpairing.h”. Por otro lado, el funcionamiento del convertidor ADC se ha modelado en la librería “ADConeshot.h”. La estructura es más sencilla y permite modular el código en diferentes clases que permiten modelar el comportamiento del dispositivo de una manera más eficiente.

- La librería “AUTOpairing.h” se ha diseñado para albergar una clase que componen el funcionamiento del auto emparejamiento del dispositivo que se pretende conectar a la pasarela y las funciones que modelan la configuración y conexión al servidor que alojan el fichero para realizar la actualización FOTA.
- La librería “ADConeshot.h” modela el comportamiento del convertidor analógico-digital. Permite ser configurado, seleccionar el canal y varios parámetros en función de las necesidades que sean requeridas.

Este modelado en clases permite, en definitiva, modular y realizar un código más sencillo, cuya funcionalidad se reduce exclusivamente a lo que se pretende realizar. El código funcional está escondido al usuario, de tal modo que este solamente se preocupa de configurar los diferentes parámetros funcionales que necesite.

El código está visible en un repositorio disponible [28].

Anexos - Anexo II. Estructura de la aplicación desarrollada en ESP-IDF

Bibliografía

- [1]. Guerrero, J. C. (s. f.). GUÍA PARA LA IMPLANTACIÓN DE TECNOLOGÍAS EN LA INDUSTRIA 4.0 (11a Parte). es.linkedin.com.
<https://es.linkedin.com/pulse/gu%C3%A1a-para-la-implantaci%C3%B3n-de-tecnolog%C3%ADas-en-industria-joaquin-8>
- [2]. ¿Qué es la Industria 4.0 y cómo funciona? | IBM. (s. f.). <https://www.ibm.com/es-es/topics/industry-4-0>
- [3]. Beetle-ESP32-C3 RISC-V Core Development Board Wiki - DFRobot. (s. f.).
https://wiki.dfrobot.com/SKU_DFR0868_Beetle_ESP32_C3
- [4]. Build and Flash with Eclipse IDE - ESP32 - — ESP-IDF Programming Guide release-v4.2 documentation. (s. f.). <https://docs.espressif.com/projects/esp-idf/en/release-v4.2/esp32/get-started/eclipse-setup.html>
- [5]. DiegoPaezA. (s. f.). ESP32-freeRTOS/main.c at master · DiegoPaezA/ESP32-freeRTOS. GitHub. <https://github.com/DiegoPaezA/ESP32-freeRTOS/blob/master/adc/main.c>
- [6]. ESP32-C3-DevKitM-1 - ESP32-C3 - — ESP-IDF Programming Guide latest documentation. (s. f.). <https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html#pin-layout>
- [7]. Esp32tutorialslab. (2023). ESP32 ESP-IDF FreeRTOS Tutorial: Learn to Create Tasks. ESP32 ESP-IDF. <https://esp32tutorials.com/esp32-esp-idf-freertos-tutorial-create-tasks/>
- [8]. Richard Barry. Real Time Engineers Ltd., 2016 "Mastering the FreeRTOS™ Real Time Kernel. A Hands-On Tutorial Guide". Disponible en:
https://freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf
- [9]. Amazon Web Services. Amazon.com, Inc., 2017 "The FreeRTOS™ Reference Manual". Disponible en: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf
- [10]. ESP-IDF vs Arduino Core: Which Framework to Choose in 2023. (2023, 18 marzo). Disponible en: <https://www.espboards.dev/blog/esp-idf-vs-arduino-core/>
- [11]. Download Python. (s. f.). Python.org. <https://www.python.org/downloads/>
- [12]. Download the Latest Java LTS Free. (s. f.).
<https://www.oracle.com/java/technologies/downloads/>

Anexos - Bibliografía

- [13]. Eclipse IDE for C/C++ Developers | Eclipse Packages. (s. f.).
<https://www.eclipse.org/downloads/packages/release/2022-09/r/eclipse-ide-cc-developers>
- [14]. Partition Tables - ESP32 - — ESP-IDF Programming Guide latest documentation. (s. f.). <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/partition-tables.html>
- [15]. ESP32 alimentado por batería | Protocolo de red radioshuttle. (s. f.).
<https://www.radioshuttle.de/es/medias-es/informaciones-tecnicas/esp32-alimentado-por-bateria/>
- [16]. Admin. (2023, 16 junio). FreeRTOS Tutorial #5 ->Using Queue » ControllersTech. ControllersTech. <https://controllerstech.com/freertos-tutorial-5-using-queue/>
- [17]. Analog-to-Digital Converter (ADC) — ESP-FAQ documentation. (s. f.).
<https://espressif-docs.readthedocs-hosted.com/projects/espressif-esp-faq/en/latest/software-framework/peripherals/adc.html>
- [18]. alldatasheet.es. (s. f.). BS170 PDF, BS170 Descripción electrónicos, BS170 Datasheet, BS170 View: ALLDATASHEET: <https://pdf1.alldatasheet.es/datasheet-pdf/view/121776/ONSEMI/BS170.html>
- [19]. Luca, B. (2017, 16 enero). ESP32 (6) – How to connect to a wifi network. lucadentella.it. <https://www.lucadentella.it/en/2017/01/16/esp32-6-collegamento-ad-una-rete-wifi/>
- [20]. Event Loop Library - ESP32 - — ESP-IDF Programming Guide Latest Documentation. (s. f.). https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp_event.html
- [21]. FreeRTOS. (2023, 13 julio). FreeRTOS event bits, event groups and event flags. <https://www.freertos.org/FreeRTOS-Event-Groups.html>
- [22]. Wi-Fi Driver - ESP32 - — ESP-IDF Programming Guide Latest Documentation. (s. f.). <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html#wifi-programming-model>
- [23]. Jain, M. (2021, 25 diciembre). OTA Updates Framework - The ESP Journal. Medium. <https://blog.espressif.com/ota-updates-framework-ab5438e30c12>
- [24]. Simple Pre-Purchase questions (RTC/Memory/Flash/NVS) - ESP32 Forum. (s. f.-b). <https://www.esp32.com/viewtopic.php?p=29525#p29525>
- [25]. Non-volatile Storage Library - ESP32 - — ESP-IDF Programming Guide Latest Documentation. (s. f.). https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/nvs_flash.html

- [26]. ESP32: Best way to store data frequently? (s. f.). Stack Overflow.
<https://stackoverflow.com/questions/63780850/esp32-best-way-to-store-data-frequently>
- [27]. Bosman, H., Iacca, G., Tejada, A., Wörtche, H., & Liotta, A. (2017). Spatial anomaly detection in sensor networks using neighborhood information. *Information Fusion*, 33, 41-56. <https://doi.org/10.1016/j.inffus.2016.04.007>
- [28]. Danrodcar. (s. f.). GitHub - danrodcar1/TFM_DRC: repositorio del TFM de Daniel Rodríguez Carrión, basado en una pasarela de conexión automática mediante protocolo ESP-NOW. GitHub. https://github.com/danrodcar1/TFM_DRC.git