ESCUELA DE INGENIERÍAS INDUSTRIALES
GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL

Integración de un Sistema de Visión para un Controlador Inteligente de Puertas de Garaje
Integration of a Vision System for a Smart Garage Door Controller

Realizado por
Alejandro Padilla Rodríguez
Tutorizado por
Alejandro Hidalgo Paniagua
Departamento
Departamento de Tecnología Electrónica

UNIVERSIDAD DE MÁLAGA
MÁLAGA, 22 de Mayo de 2025

Fecha defensa:
El Secretario del Tribunal

Agradecimientos

A Mamá y Papá, por la yesca y el pedernal.

A mi tío David, por enseñarme desde pequeño que bajo las cenizas hay rescoldos.

A mi Cori, por mantener la hoguera viva bajo la lluvia a mi lado.

A mi hermano Andrés, por la barbacoa que nos vamos a encajar.

Todo hombre es libre de elevarse tanto como pueda o quiera, pero es sólo el nivel en el que piensa lo que determina el grado en el que se elevará.

John Galt - La rebelión de Atlas

Resumen:

Integración de un módulo ESP32Cam en una centralita de gestión de accesos basada en ESP32. El módulo será capaz de comunicarse con la centralita a través de protocolo serie y MQTT, aprovechando distintas funciones de la cámara tales como la capacidad de procesamiento de imagen, retransmisión de vídeo en streaming, o detección automática de códigos QR. La integración se ha desarrollado usando C++ y una arquitectura de software basada en eventos, con el objetivo de lograr un dispositivo altamente escalable. Se ha programado también la funcionalidad necesaria para posibilitar la conexión automática a la red mediante lectura de códigos QR. Por último, se ha diseñado una carcasa compatible con impresión 3D usando programas CAD, lo que posibilita realizar pruebas de campo con el diseño final.

Palabras claves: Microcontroladores, Programación Orientada a Objetos, ESP32, Arquitectura de Software, Impresión 3D

Abstract: Integration of an ESP32-CAM module into an ESP32-based parking access management system. The module will be able to communicate with the control system via serial protocol and MQTT, taking advantage of various camera features such as image processing capabilities, video streaming, and automatic QR code detection.

The integration has been developed using C++ and an event-driven software architecture in order to achieve a highly scalable device. The necessary functionality has also been programmed to enable automatic network connection through QR code scanning.

Finally, a 3D-printable housing has been designed using CAD software, which will allow the final design to be tested in the field.

Keywords: Microcontrollers, Object Oriented Programming, ESP32, Software Architecture, 3D Printing

Siglas

API Application Programming Interface. 35

ASCII American Standard Code for Information Interchange. 26

CA Certification Authority. 24

CAD Computer-Aided Design. 6, 65

CAM Computer Aided Manufacturing. 71

CSV Comma-Separated Values. 32

DTR Data Terminal Ready. 17, 34

EEPROM Electrically Erasable Programmable Read-Only Memory. 52, 58, 59

EIA Electronic Industries Association. 33

ESP-IDF Espressif IoT Development Framework. 14–16, 19, 29, 32, 35, 74, 75

FDM Fused Deposition Modelling. 70

FIFO First In First Out. 39

FPS Frames Per Second. 2

FreeRTOS Free Real-Time Operating System. 37–39

FSM Finite State Machine. 42, 48, 61

GNU GNU's Not Unix. 18

GPIO General Purpose Input-Output. 31, 35, 58

GUI Graphic User Interface. 89

HTTP Hyper-Text Transfer Protocol. 25, 49

HTTPS Hyper-Text Transfer Protocol Secure. 1, 20, 22, 25, 46

IoT Internet of Things. 1, 3–6, 26, 27, 35, 36

IP Internet Protocol. 25, 42, 43, 55, 60

JSON JavaScript Object Notation. 22, 25, 26, 44, 51, 55, 56

LAN Local Access Network. 6

LDF Library Dependency Filter. 19

MAC Media Access Control. 41, 85

MQTT Message Queue Telemetry Transport. 1, 18, 20, 23, 24, 26, 27, 41, 54, 59, 61, 78–80, 82

OCR Optical Character Recognition. 90

OTA Over-The-Air. 18, 23, 32, 42, 47, 56

PLA PolyLactic Acid. 70, 71

PnP Plug and Play. 6

PSRAM Pseudo Static Random Access Memory. 18, 32, 33, 85

QR Quick Response. 45, 52, 53, 59, 62, 80, 81

RAII Resource Acquisition Is Initialization. 40

RAM Random Access Memory. 16

RGB Red-Green-Blue. 35

RS232 Recommended Standard 232. 33

RSA Rivest, Shamir, Adleman. 24

RTS Request To Send. 17, 34

SDLC Software Development Life Cycle. 9

SoC System-on-Chip. 3

SPIFFS Serial Peripheral Interface Flash File System. 16, 25, 43

SSID Service Set Identifier. 45, 48, 59, 76, 80

SSL Secure Sockets Layer. 23, 24, 46

STL STereoLitography. 70

TLS Transport Layer Secure. 25

TTB TheThingsBrain. 1, 5

UART Universal Asynchronous Receiver/Transmitter. 29, 31, 35, 74

UC Unidad Central. 6, 7, 10, 31, 34–36, 41, 44–48, 50–54, 56, 60, 62, 66, 67, 74, 76, 77, 79, 81, 82, 85

UDP User Datagram Protocol. 25

UI User Interface. 43, 76, 83

URL Uniform Resource Locator. 25, 42, 76

USB Universal Serial Bus. 31, 33

VCS Version Control System. 21

VGA Video Graphics Array. 2

VPN Virtual Private Network. 25, 43, 46

VSCode Visual Studio Code. 13, 14, 73, 76

Índice de contenidos

CAPÍTULO 1

Introducción

En un contexto marcado por la creciente expansión del Internet of Things (IoT), el presente proyecto se plantea como una solución tecnológica adaptable, económica y fácilmente desplegable en entornos reales. El núcleo del desarrollo reside en la integración de una cámara ESP32Cam, cuya incorporación permite la captura y transmisión remota de imágenes, dotando al sistema de nuevas capacidades. Esta funcionalidad se articula sobre una unidad central modular y reconfigurable TheThingsBrain (TTB), diseñada para gestionar accesos de forma remota y eficiente. La solución se sustenta en estándares de conectividad como Message Queue Telemetry Transport (MQTT) e Hyper-Text Transfer Protocol Secure (HTTPS), y se completa con una carcasa optimizada mediante diseño 3D, lo que facilita su instalación sin necesidad de infraestructuras complejas.

A través de una arquitectura de *software* basada en eventos y una metodología de desarrollo ágil, se ha conseguido una estructura robusta, escalable y segura, adecuada para futuros desarrollos. Las comunicaciones entre módulos se realizan a través de conexión serie, y, opcionalmente, de forma remota. Así se garantiza un flujo estable de datos entre los componentes, posibilitando su control y monitorización mediante herramientas externas.

Este documento no sólo expone de manera detallada el proceso técnico y metodológico seguido, sino que también adopta un enfoque eminentemente práctico, orientado a la validación funcional del sistema desarrollado. Para ello, se incluyen la planificación y ejecución de pruebas específicas que permiten verificar la viabilidad y el correcto funcionamiento del diseño propuesto.

1.1. Estado del arte

Se ha realizado una investigación sobre los posibles productos en el mercado asimilables a la solución propuesta. Se mencionan en este apartado las opciones tomadas en consideración para la consecución del proyecto.

1.1.1. Viabilidad del proyecto

Los requisitos necesarios para la selección de una cámara adecuada se resumen en los siguientes:

1. Tamaño y peso reducidos para reducir el volumen de la futura carcasa.
2. Costo reducido
3. Compatibilidad con resolución Video Graphics Array (VGA)
4. Velocidad de transmisión de 15 Frames Per Second (FPS)
5. Compatibilidad con zócalo de ESP32Cam

Para realizar esta integración, se han comparado distintos módulos programables compatibles con ESP32, que será el microprocesador designado como plataforma para desplegar el proyecto.

También se evalúa la posibilidad de hacer uso de módulos integrados de cámara y microprocesador como *ArduCam* o *MT9D111*.

El *hardware* evaluado se ha recopilado en la tabla 1.1

Identificador	**Ratio de transferencia**	**Costo total**	**Resolución**
OV2640	1600×1200: 15fps SVGA: 30fps CIF: 60fps	15 €	2 MPx
OV7725	VGA: 60fps	20 €	0.3 MPx
OV5642	QSXGA(2592x1944):15fps 1080p:30fps 1280x960:45fps 720p:60fps VGA(640x480):90fps QVGA(320x240):120fps	30 €	5 MPx
OV3660	(2048x1536):15fps 1080p: 20 fps 720p: 45 fps XGA(1024x768): 45 fps VGA(640x480): 60 fps QVGA(320x240): 120fps	25 €	3.2 MPx
ArduCAM Mini 2MP	VGA: 60fps	35 €	2 MPx
ArduCAM Mini 5MP Plus	(2592×1944): 15 fps 1080p (1920×1080): 30 fps 720p (1280×720): 60 fps VGA (640×480): 60 fps QVGA (320×240): 120 fps	40 €	5 MPx
MT9D111	15 fps a resolución completa	22 €	2 Mpx

Tabla 1.1: *Hardware* valorado para realizar la integración

Dados los requisitos del prototipo mencionados al comienzo de esta sección, se ha optado definitivamente por el uso de la cámara OV2640.

La opción seleccionada ofrece una resolución aceptable para la finalidad del proyecto, y es compatible con la placa de desarrollo ESP32Cam, cuyos detalles se ofrecen más adelante en la sección 3.1.1, del fabricante *Espressif*. Se diseña el prototipo para que sea capaz de transmitir tanto imágenes individuales como vídeo en *streaming*.

Espressif Systems es una empresa china *fabless* (sin fábricas en propiedad) fundada en 2008, especializada en el diseño y fabricación de System-on-Chip (SoC) y módulos inalámbricos que soportan múltiples tecnologías de telecomunicación. Sus productos, especialmente las familias ESP8266 y ESP32, son muy populares en el ámbito del IoT por su bajo consumo, versatilidad y coste accesible.

El resto de cámaras OVxxx también son compatibles con la mencionada placa de desarrollo, pero han sido descartadas por el elevado coste respecto al modelo escogido, que ya cumple con los requisitos especificados a menor coste.

En cuanto a las plataformas *ArduCAM* y *MT9D111*, dado que se encuentran implementadas en sus propias placas de desarrollo, obligan a prescindir de las funcionalidades de conectividad que sí incluye la placa ESP32Cam, o bien a realizar la integración con nuevos componentes que aporten estas funcionalidades, encareciendo el prototipo innecesariamente.

1.1.2. Aprimatic

Entre los conceptos más cercanos que se han podido encontrar en el mercado, destaca uno especialmente, por sus similitudes al proyecto desarrollado. Se puede observar una imagen del mencionado producto en la figura 1.1

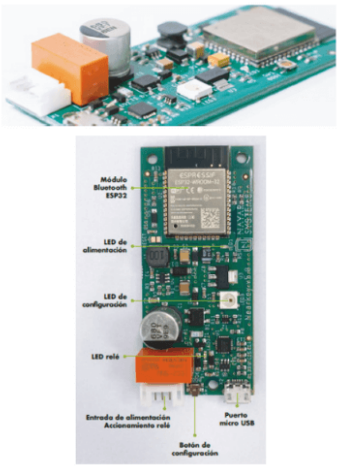


Figura 1.1: Dispositivo RX BTACCES. Imagen extraída de (Aprimatic, n.d.).

El dispositivo *Aprimatic RX BTaccess* se presenta como una alternativa portátil, barata, y adaptable para la gestión de puertas de garaje. Es una solución basada en ESP32, al igual que este proyecto.

Es capaz de gestionar los permisos de acceso en la nube, y cuenta con una *app* para ello. Sin embargo, su uso se encuentra limitado a un solo actuador, y necesita estar en rango de *Bluetooth* con el usuario para validar el acceso. También permite restricciones temporales.

La *app* de este producto necesita autenticación por parte del instalador para poder ser usada, ya que es necesario registrar la instalación y validarla con el fabricante para poder operar con ella.

1.1.3. IoT

Desde la aparición del concepto de Internet of Things (IoT) a finales de la década de 1990, el mercado ha ido incorporando progresivamente dispositivos conectados que permiten la recopilación y el intercambio de datos. Desde el despliegue de las redes 4G de última generación y la consecuente ampliación del ancho de banda disponible para el tratamiento de datos, esta tendencia se ha acelerado notablemente, poniendo a disposición de los consumidores cada vez más dispositivos alineados con este nuevo paradigma tecnológico.

Este concepto fue acuñado en 1999 por Kevin Ashton. Utilizó el término durante un período de tiempo en sus presentaciones en el MIT Auto-ID Center, lugar en el que trabajaba. No se popularizó hasta 2004-2005, período en el cual diversas publicaciones de prestigio comenzaron a hacer uso de él, al constatar que la exótica premisa de K. Ashton se estaba haciendo realidad.

(Ashton, 2009) :

> "If we had computers that knew everything there was to know about things—using data they gathered without any help from us—we would be able to track and count everything, and greatly reduce waste, loss and cost."
>
> "Si tuviéramos computadoras que supieran todo lo que hay que saber sobre las cosas, utilizando datos que recopilaran sin nuestra ayuda, podríamos rastrear y contar todo, y reducir enormemente el desperdicio, la pérdida y el costo" */traducción propia/*(Ashton, 2009, parr. 4).

Después, el concepto sería desarrollado por otras investigaciones, a medida que el Internet of Things a se convertía en una idea popular llamada a cambiar el mundo. En 2004 nace el concepto de *Internet-0*. (Gershenfeld, Krikorian y Cohen, 2004):

> "Internet-0 allows myriad devices to intercommunicate and interoperate: pill bottles can order refills from the pharmacy; light switches and thermostats can talk to lightbulbs and heaters; people can check on their homes from their offices. Existing technologies already allow many of these functions, but Internet-0 provides a single consistent standard..."
>
> "Internet-0 permite a miríadas de dispositivos inter-comunicarse e inter-operar: botes de pastillas que pueden solicitar recargas a la farmacia; interruptores de luz y termostatos que conversan con bombillas y calentadores; la gente puede monitorizar sus casas desde la oficina. Las tecnologías existentes ya permiten muchas de estas funcionalidades, pero Internet-0 provee un estándar consistente.../*traducción propia*/(Gershenfeld et al., 2004, p. 79)."

Esta definición de *Internet*-0 es la que más se aproxima al enfoque moderno que popularmente condensa el Internet of Things. Una propuesta que, gracias al despliegue de las redes de última generación, está materializándose ante nosotros, haciendo que nuestra tecnología pueda "conversar" en tiempo real con nuestro entorno. Las implicaciones

de estos avances son incalculables, a medida que se fusionan y complementan con otros avances tecnológicos en plena expansión hoy por hoy.

1.2. Antecedentes

Dentro de las posibles aplicaciones del IoT, las más populares se centran en la domotización del hogar. La llegada de plataformas abiertas como Home Assistant, Node-RED y la aparición de dispositivos fabricados compatibles con estos sistemas (Fabricantes como TP-Link, Xiaomi, Sonoff, Philips Hue y otras menos conocidas) ha permitido la instalación de redes IoT domésticas con capacidades que no hace tanto tiempo escapaban al bolsillo del público general.

Sin embargo, en la mayoría de casos se trata de dispositivos *stand alone*, cuya utilidad se limita a un usuario final, y con unas funciones muy concretas.

Este proyecto pretende implementar una solución aplicada a la gestión de estacionamientos, de forma que el sistema en su conjunto sea escalable y modificable. Para ello se trabajará sobre una plataforma de *hardware* TheThingsBrain (TTB) previamente diseñada, basada en ESP32, sobre la que se despliegan las mejoras de *hardware* y *software* que dotarán de mayor funcionalidad al sistema.

Según (Balali et al., 2020, Cap. 1.1, p. 1) el valor diferencial que aporta una red IoT es la conexión entre muchos dispositivos y unidades inteligentes, los cuales son capaces de comunicarse de manera fluida y eficiente para medir, recolectar y analizar los datos requeridos. La calidad de la propia red vendrá determinada por la capacidad de interacción entre los elementos que la componen.

Una aplicación IoT se puede estructurar tal y como se incluye en la figura 1.2.

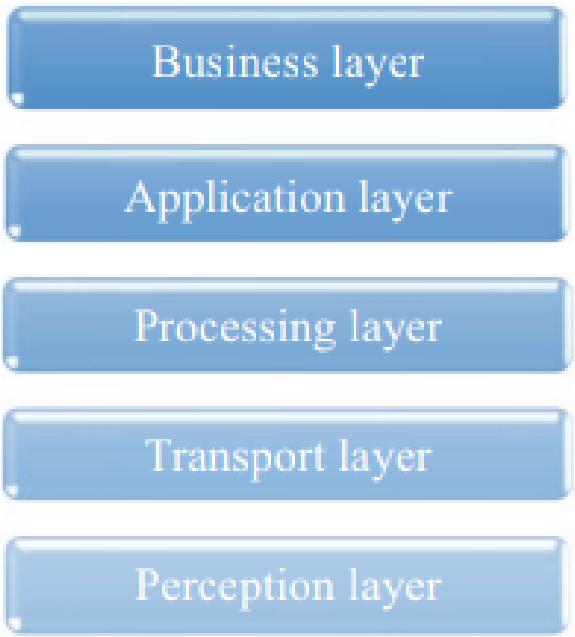


Figura 1.2: Estructura de diseño del IoT, tomado de (Jabraeil Jamali et al., 2020, Cap 1.1)

(Jabraeil Jamali et al., 2020, Cap. 1.1) expone que la estructura de una red IoT se puede caracterizar en 5 capas:

1. Todo el sistema IoT es gestionado por la capa de **negocio**. Esto incluye desde aplicaciones a lógica de negocio e incluso privacidad de los usuarios.

2. La capa de **aplicación** es responsable de proveer al usuario con servicios específicos. Recoge todas las posibles aplicaciones para el IoT.

3. La capa de **procesado** es también mencionada como capa intermedia. Puede almacenar, analizar, y procesar grandes cantidades de datos transportados. Además, puede gestionar y proveer de toda una capa de servicios derivados, tales como bases de datos, computación en la nube, o módulos de procesado de **big data**.

4. La capa de **transporte** transfiere la información recibida por la capa de percepción hasta la capa de procesamiento y viceversa, a través de protocolos como *Bluetooth*, WiFi, NFC, 5G, Local Access Network (LAN) u otros.

5. La capa de *percepción* es la capa física que contiene sensores de información ambiental. En el entorno, se detectan algunos parámetros físicos o se identifican otros objetos inteligentes.

Este proyecto se planteará de acuerdo a esta estructura básica.

1.3. Objeto del proyecto

El objetivo es ampliar las funcionalidades de un sistema de *hardware* Plug and Play (PnP), acorde a los principios del IoT, con capacidad para gestionar permisos de usuarios de forma remota, y fácilmente adaptable a las necesidades del usuario final. Se propone como un prototipo *hardware* de control de accesos, que no precisa de instalación especializada, reconfigurable y de bajo costo.

El desarrollo del presente proyecto parte de una centralita previamente implementada, la cual será referida en adelante como Unidad Central (UC). Este documento describe de manera detallada las etapas seguidas para la integración de una cámara en dicha unidad central con el objetivo de dotarla de capacidad para adquisición y procesamiento de imágenes. La solución propuesta permite tanto al usuario como a la propia UC activar la captura remota de imágenes, así como la transmisión de vídeo en modo *streaming*, validando así la viabilidad funcional del sistema desarrollado.

Además, se diseñará usando software Computer-Aided Design (CAD) una carcasa apta para impresión 3D, que posibilite su instalación de una forma sencilla para el usuario final.

La UC se presenta en la figura 1.3. Esta plataforma está basada en el microcontrolador ESP32, y se conecta al módulo ESP32Cam vía RS232. Esto constituye el nucleo sobre el que se estructura el proyecto. Una vez integrados ambos componentes, se conforma un único dispositivo funcional, capaz de satisfacer los objetivos planteados en el desarrollo del sistema.

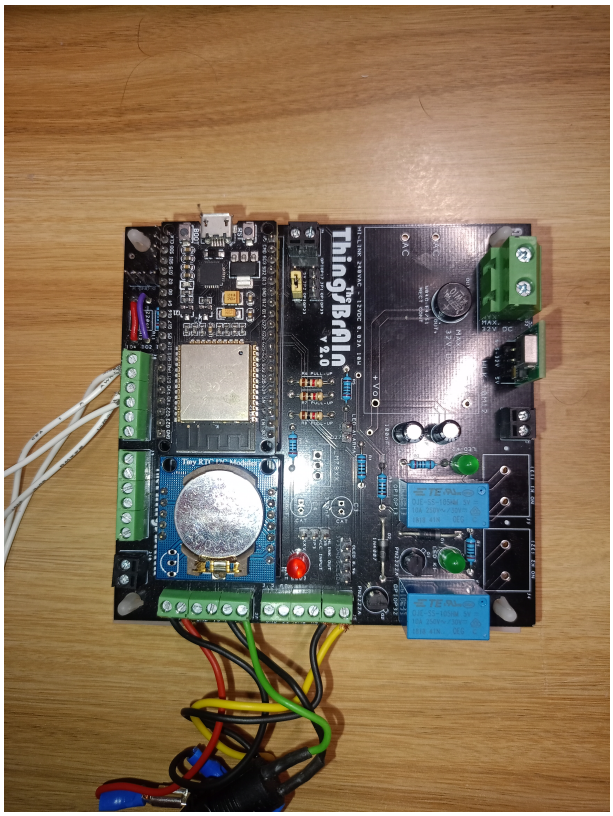


Figura 1.3: Imagen de la UC, que servirá de punto de partida para el desarrollo del proyecto.

1.4. Metodología de trabajo

Para garantizar el éxito de este proyecto, se ha recurrido a un conjunto de herramientas y metodologías de trabajo sólidamente establecidas y reconocidas en la industria.

1.4.1. Metodología Scrum

Según (Galiano, 2016, Cap. 1), la definición adecuada para la metodología *scrum* es la siguiente:

> Scrum está basado, por un lado, en la teoría del control empírico de procesos para la gestión de sistemas adaptativos complejos. Los tres pilares de este proceso son los siguientes:
>
> - **Transparencia:** los aspectos significativos del proceso tienen que ser conocidos por todo aquel que participa, lo cual conlleva que estos aspectos estén definidos mediante un estándar común, de forma que todo el mundo tenga la misma percepción de las características de cada aspecto (por ejemplo, la definición de acabado).
> - **Inspección:** todo proceso persigue un objetivo y, para llegar a ese objetivo, hace falta que los participantes en el proceso evalúen de manera continua sus resultados, y el proceso mismo, para detectar posibles desviaciones tan pronto como sea posible.
> - **Adaptación:** cuando se detecta una desviación, la respuesta debe ser la adaptación; es decir, la adopción de acciones o planes que, o bien ayuden a corregir la desviación, o bien reconfiguren el objetivo.

Scrum es una de las metodologías ágiles más utilizadas en la gestión y desarrollo de proyectos, especialmente en el ámbito de la ingeniería y el *software*. Su principal objetivo es facilitar la entrega incremental y continua de valor, permitiendo la adaptación rápida a los cambios y la mejora constante del producto y del proceso de trabajo. En el contexto de este documento, la adopción de *Scrum* ha permitido organizar el trabajo en ciclos cortos y repetitivos, denominados *sprints*, lo que ha favorecido la planificación, la revisión frecuente y la retroalimentación continua.

Los mencionados *sprints* son de duración fija (generalmente entre 2 y 4 semanas). Cada uno comienza con una reunión de planificación (*Sprint Planning*), donde se seleccionan las tareas a realizar. Diariamente, se celebra una breve reunión (*Daily Scrum*) para sincronizar el trabajo y detectar posibles bloqueos. Al final de cada uno de ellos, se realiza una revisión (*Sprint Review*) y una retrospectiva para identificar mejoras en el proceso.

1.4.2. Kanban

Para el seguimiento y la visualización del flujo de trabajo, se ha complementado *Scrum* con tableros *Kanban*.

Como explica a (Galiano, 2016), *Kanban* es un sistema para mostrar el estado de las tareas de un proyecto. Ideado por el industrial Taiichi Ohno en 1953. *Kanban* es la unión de dos conceptos: *kan*, que significa 'visual', y *ban*, que significa 'tarjeta'. Tarjetas visuales

situadas en un tablero y que indican en todo momento el estado de fabricación de una pieza, un producto o cualquier elemento que requiere de un proceso de acciones para su construcción.

Kanban es una herramienta visual que permite gestionar el estado de las tareas mediante columnas como *To Do*, *Doing*, *Testing* y *Done*. Cada tarea se representa como una tarjeta que se mueve entre las columnas según su estado de avance. Este sistema facilita la identificación de cuellos de botella, la asignación de responsabilidades y la transparencia en el progreso del proyecto.

En este proyecto, el uso de *Kanban* ha resultado especialmente útil para:

- Visualizar de manera clara el estado de cada tarea.
- Fomentar la colaboración y la comunicación dentro del equipo.
- Facilitar la priorización y la reasignación rápida de tareas en función de la carga de trabajo y los imprevistos.

En la figura 1.4 se muestra la herramienta utilizada para la gestión del tablero *Kanban* del proyecto. Este se ha integrado dentro de la herramienta *GitLab*, mencionada en la sección 2.1.4.

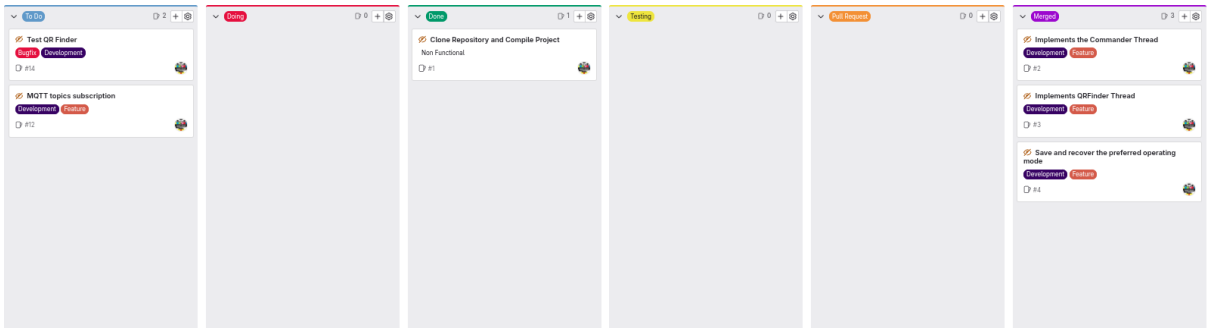


Figura 1.4: Captura de la herramienta usada para el tablero *Kanban* (también en anexo A.1)

La utilización de Kanban permite ajustar el desarrollo del proyecto al Software Development Life Cycle (SDLC) en cascada. Según explica (Villada Romero, 2023, Cap.3, Sec. 2.1):

> Este modelo se denomina también modelo lineal o ciclo de vida clásico. Es el modelo más antiguo que intenta dar un enfoque metodológico y riguroso al desarrollo del software, definiendo para ello distintas etapas involucradas en todo proceso. Las etapas que definen este modelo son:
>
> 1. Pre-análisis
> 2. Análisis
> 3. Diseño
> 4. Desarrollo
> 5. Pruebas
> 6. Implantación

7. Mantenimiento

Cada etapa está compuesta por un conjunto de actividades que la caracterizan y que son claves para la siguiente etapa.

El proceso comienza con una fase de pre-análisis, donde se realiza un estudio de viabilidad, se determina el alcance del sistema y se identifican las necesidades del proyecto. En esta etapa se evalúa la factibilidad económica, técnica y operativa, y se decide si el desarrollo del sistema es conveniente.

Superada la viabilidad, se procede a la planificación general de las actividades necesarias para el desarrollo. A continuación, en la etapa de análisis, se recogen y detallan los requisitos del sistema, elaborando una descripción comprensible tanto para el cliente como para el equipo de desarrollo.

Posteriormente, se clasifican los requisitos y se diseña la arquitectura del sistema, generando diagramas que describen su estructura y funcionamiento. La etapa de desarrollo implica la codificación y documentación del software, siendo fundamental que las fases previas estén correctamente ejecutadas para evitar retrasos y problemas durante la implementación.

Una vez finalizada la codificación, se realizan pruebas exhaustivas para verificar la integridad y funcionalidad del sistema. Tras superar las pruebas, el software se implanta y se capacita a los usuarios finales.

Finalmente, se lleva a cabo la fase de mantenimiento, que incluye la corrección de errores, la mejora continua del software y la incorporación de nuevas funcionalidades según las necesidades detectadas tras la puesta en marcha.

1.4.3. Pruebas

Todas las partes del proyecto se despliegan y comprueban en la propia placa de desarrollo ESP32Cam. De esta forma se ha garantizado que tanto software como hardware presenten el comportamiento esperado.

Para facilitar la ejecución de las pruebas, se ha desarrollado la opción de ejecutar un modo de depuración remota, que se suma al modo por defecto de depuración local. Este modo permite llevar a cabo las pruebas necesarias tanto en la UC como en el módulo ESP32Cam sin necesidad de que los dispositivos necesiten encontrarse en la misma ubicación.

El desarrollador puede alternar los modos de depuración a través de los ficheros de configuración del proyecto aportados por el entorno *platformIO*. Esta opción se explica más detalladamente en el apartado 2.1.2.

Las pruebas planteadas para el dispositivo se han desarrollado extensamente en el Capítulo 5.

En la figura 1.5 se muestra una imagen del módulo que integra el proyecto y la placa programadora utilizada. Este ultimo dispositivo es completamente opcional, ya que el módulo ESP32Cam es programable a través de puerto serie, activando su modo *bootloader* conectando su pin *GPIO0* a *GND*.

Figura 1.5: Imagen del módulo ESP32Cam y su correspondiente placa programadora

1.5. Contenido de la memoria

A continuación se resume el contenido en cada uno de los capítulos que componen la memoria de trabajo.

- ***Capítulo 2*: consideraciones previas.**
  En este capítulo se desarrollan aquellos conceptos previos más relevantes que serán usados recurrentemente durante el desarrollo del proyecto, y ayudan a entender su estructura y diseño, tales como herramientas o protocolos.

- ***Capítulo 3*: desarrollo de la solución**
  Se detallan los elementos relativos a *hardware* que conforman el proyecto y conectan las distintas entidades que interactúan en el software del proyecto. Se hace mención a métodos de conexión, uso de memoria en el dispositivo, o elementos relacionados con las comunicaciones.

- ***Capítulo 4*: diseño de la carcasa**
  Se explica en detalle las herramientas usadas y proceso de modelado de la carcasa que alojará el prototipo funcional. Esta será impresa en 3D con materiales resistentes, posibilitando en un futuro realizar pruebas de campo.

- ***Capítulo 5*:test y validación**
  En este apartado de la memoria, se hace referencia a las pruebas de funcionamiento realizadas a lo largo del proceso de desarrollo del prototipo, orientadas a verificar la correcta implementación del dispositivo.

- ***Capítulo 6*: conclusiones**
  En este apartado de la memoria, se hace una reflexión sobre el resultado del proyecto.
- ***Capítulo 7*: futuras líneas de trabajo**
  Se exponen diferentes posibilidades abiertas a la futura mejora del prototipo y posibles ramas de investigación y desarrollo del proyecto.

CAPÍTULO 2

Consideraciones previas

En este capítulo se presentan las consideraciones previas y los conceptos fundamentales necesarios para comprender tanto el alcance como el proceso de desarrollo del proyecto.

Se abordan conceptos clave, técnicas usadas, y desarrollos previos necesarios para el correcto funcionamiento del prototipo.

2.1. Entorno de desarrollo

En esta sección se procede a detallar la configuración del entorno sobre el que se ha desarrollado el proyecto a nivel de software. Se hace especial énfasis en las herramientas que han sido empleadas.

2.1.1. VSCode

Figura 2.1: Logotipo del entorno de desarrollo VSCode IDE

Visual Studio Code (VSCode), cuyo logotipo se muestra en la figura 2.1, es una herramienta fundamental para el desarrollo de proyectos de *software* debido a su versatilidad, ligereza y capacidad de personalización. Este editor de código fuente, desarrollado por *Microsoft*, permite trabajar eficientemente en una amplia variedad de lenguajes de programación y *frameworks*, gracias a su soporte multiplataforma y a su amplia biblioteca de extensiones. Entre sus características más destacadas se encuentran el resaltado de sintaxis, el autocompletado inteligente mediante *IntelliSense*, la navegación rápida por

el código, y la integración nativa con sistemas de control de versiones como *Git*, lo que facilita la colaboración y el seguimiento de cambios en proyectos individuales o en equipo.

Además, VSCode incluye un terminal integrado y potentes herramientas de depuración. Su sistema de extensiones permite adaptar el editor a cualquier flujo de trabajo, añadiendo compatibilidad con herramientas específicas, temas visuales y utilidades de análisis de código.

2.1.2. PlatformIO

Figura 2.2: Logotipo de PlatformIO

PlatformIO es una extensión para VSCode que ofrece un ecosistema completo y profesional orientado al desarrollo de sistemas embebidos y microcontroladores. En la figura 2.2 se muestre el logotipo identificativo de este *framwork*. Está especialmente diseñado para simplificar el trabajo con microcontroladores y placas de desarrollo, ofreciendo herramientas avanzadas como gestión automática de librerías, detección de puertos, depuración integrada y soporte multiplataforma. Además, integra la herramienta Espressif IoT Development Framework (ESP-IDF), desarrollada por *Espressif*, lo que permite trabajar de forma nativa con placas basadas en ESP32, como el ESP32Cam.

Entre las utilidades que ofrece, destacan varias que han sido empleadas de forma intensiva en este proyecto:

- Gestor de plataformas *hardware*
- Sistema de compilación avanzado
- Gestión de dependencias y librerías
- Soporte para pruebas unitarias y control de *hardware*
- Compatibilidad con entornos de desarrollo en la nube
- Capacidad para monitoreo por puerto serie

Una de las características más reseñables de *PlatformIO* en relación al proyecto es su capacidad para la gestión de dependencias, ya que permite instalar y administrar librerías de manera individual para cada proyecto, evitando así conflictos de versiones y facilitando la portabilidad del código entre diferentes entornos de desarrollo. A diferencia de otros entornos como *Arduino IDE*, donde las librerías se gestionan de forma manual y global, de forma que cualquier cambio puede afectar a todos los proyectos. *PlatformIO* ofrece un entorno más controlado y flexible, asegurando que cada proyecto utilice exactamente

las versiones necesarias de cada dependencia. Esto resulta especialmente útil en proyectos complejos o colaborativos, donde la consistencia y la trazabilidad de las dependencias son fundamentales para garantizar la estabilidad y el correcto funcionamiento del software.

Se suma a lo anterior la integración completa con ESP-IDF, el *framework* oficial de Espressif para el desarrollo profesional de software en microcontroladores de la familia ESP.

PlatformIO simplifica el proceso de configuración al automatizar gran parte de la instalación, ajuste y compilación del entorno mediante el archivo *platformio.ini*, cuya estructura y funcionamiento se detallan más adelante en el apartado 2.1.2.

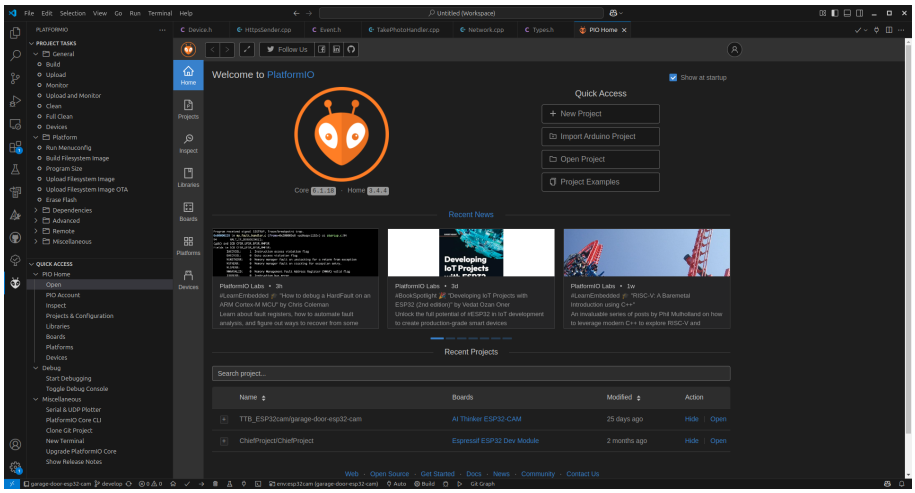


Figura 2.3: Captura del *plugin PlatformIO* instalada en Visual Studio Code

Platformio.ini

El archivo *platformio.ini* es el que dota al proyecto de una gran portabilidad entre distintas plataformas de *hardware*y *software*. Esto lo convierte en un elemento esencial, al definir cómo se debe construir, compilar, cargar y depurar el proyecto. Entre otras muchas posibilidades, el archivo *platformio.ini* recoge información de alto y bajo nivel.

A continuación, en la figura 2.4 se muestra el contenido del archivo *platformio.ini* para proceder a desgranar su contenido.

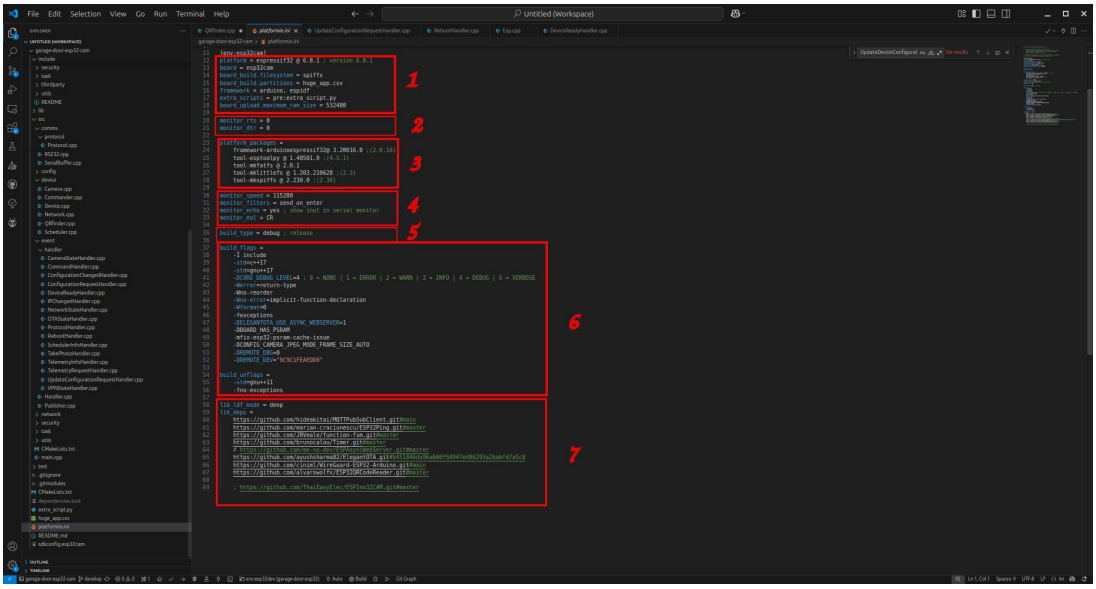


Figura 2.4: Captura del contenido del archivo de configuración `platformio.ini` (también en anexo A.2)

En la figura se han distinguido distintos bloques el contenido del archivo para facilitar la explicación del mismo.

Bloque 1
Este bloque define el entorno de desarrollo específico para una placa ESP32Cam. Los parámetros enmarcados son:

- *platform*: establece la versión específica del marco `Espressif32` que se usará.
- *board*: define el modelo de la placa
- *board_ build.filesystem*: usa SPIFFS, un sistema de archivos embebido.
- *board_ build.partitions*: carga una tabla de particiones personalizada. Esta tabla ha sido mencionada con anterioridad, en el apartado 3.1.3.
- *framework*: define los *frameworks* que serán implementados en el proyecto. Este proyecto ha sido configurado de manera que los *framework* tanto de *Arduino* como ESP-IDF coexistirán.
- *extra_ scripts*: esta directiva forma parte de la configuración avanzada de *PlatformIO* y permite inyectar *scripts Python* personalizados en momentos específicos del proceso de construcción del proyecto.

  El prefijo indica en qué momento de la compilación se ejecutarán los scripts indicados. Las opciones posibles son `pre` (antes de la compilación) y `post` (después de la compilación, y prefijo asumido por defecto).
- *board_ upload.maximum_ ram_ size*: esta opción se utiliza para definir el tamaño máximo de RAM disponible para la placa objetivo durante la compilación y carga del *firmware*.

Bloque 2
Sólo se señalizan dos variables dentro de este bloque, correspondientes a `monitor_rts = 0` y `monitor_dtr = 0`. Se trata de dos canales de datos, que forman parte del estándar RS232, como se explica en la sección 3.1.4 que conectan entre la placa de desarrollo y la cámara.

En el módulo ESP32Cam, las líneas Request To Send (RTS) y Data Terminal Ready (DTR) del puerto serie están conectadas directamente a los pines EN y GPIO0 del microcontrolador.

Cuando el monitor serie de *PlatformIO* está activo y mantiene activas estas señales, puede hacer que la placa se reinicie o entre en modo de programación en vez de ejecutar normalmente el código cargado.

Al establecer estos valores, se asegura que el monitor serie no active esas líneas y permite que la placa funcione correctamente tras la carga, mostrando la salida esperada por el terminal serie.

Bloque 3
El bloque de parámetros *platform_packages* sirve para especificar versiones concretas de las herramientas que se usarán durante el proceso de compilación, carga, depuración y manejo de sistemas. Esto permite controlar exactamente qué versiones de cada herramienta utiliza *PlatformIO*, evitando incompatibilidades o problemas por actualizaciones automáticas.

Bloque 4
Se han agrupado las directivas que permiten configurar los parámetros de comunicación serie:

- `monitor_speed:` velocidad de transmisión de datos.
- `monitor_filters:` establece un comportamiento de envío automático al presionar *Enter*.
- `monitor_echo:` activa el eco de los caracteres enviados, útil para depuración.
- `monitor_end:` define que cada línea enviada termine en retorno de carro o *Carriage Return*.

Bloque 5
El parámetro `build_type` hace referencia al tipo de compilación que se va a aplicar al proyecto. Se disponen de dos tipos principales de compilación disponibles: *debug* (para desarrollo y depuración) y *release* (para despliegue final, sin símbolos de depuración).

Bloque 6
El parámetro `build_flags` es de gran importancia, ya que permite definir un conjunto de opciones e indicadores que serán pasados directamente al compilador y al enlazador durante el proceso de construcción del *firmware*. Estos indicadores afectan al preproceso, compilación, ensamblado y enlace del código fuente en C y C++. A continuación, se describen funcionalmente los componentes más importantes incluidos en el bloque proporcionado:

- `-I include`: añade el directorio `include` a la lista de rutas de búsqueda de archivos de cabecera (*headers*) del compilador. Permite que los archivos fuente puedan incluir

cabeceras personalizadas.

- `std=gnu++17`: indica que se debe usar el estándar C++17, pero incluyendo además extensiones específicas del compilador GNU que no forman parte del estándar oficial de C++.
- `werror = return-type`: transforma los mensajes de *warning* del depurador en avisos de error de compilación cuando se detecta que una función no ha devuelto el tipo esperado.
- `fexceptions`: Opción de compilación que activa el soporte para el manejo de excepciones de C++.
- `-DCORE_DEBUG_LEVEL=4`: define la macro `CORE_DEBUG_LEVEL` con el valor 4, lo que habitualmente activa el nivel de depuración básico, que mostrará las trazas programadas en este nivel con la *macro* `LOG_ESPD{<Contexto>,<Mensaje>}`.

  El comentario asociado indica los posibles valores y su interpretación: `0 (NONE), 1 (ERROR), 2 (WARN), 3 (INFO), 4 (DEBUG), 5 (VERBOSE)`.
- `-DELEGANTOTA_USE_ASYNC_WEBSERVER=1`: define la macro con valor 1, lo que habilita la ejecuciín asíncrona de la librería OTA.
- `-DBOARD_HAS_PSRAM`: define la macro `BOARD_HAS_PSRAM`, para indicar la presencia de memoria PSRAM en la placa de desarrollo.
- `-DCONFIG_CAMERA_JPEG_MODE_FRAME_SIZE_AUTO`: deja definida la macro, habilitando así la selección automática del tamaño de fotograma en modo `streaming` para la cámara.
- `-DREMOTE_DBG=0`: define la macro `REMOTE_DBG` con valor 0 o 1, y es utilizada para habilitar las funcionalidades de depuración remota.
- `-DREMOTE_DEV="9C9C1FEAEDD0"`: se trata del identificador único del dispositivo remoto, que será utilizado para conectarse al *topic* de MQTT correspondiente para recibir mensajes de la contraparte. Lo referente al protocolo MQTT se explica con mayor detalle en el apartado 2.3

`build_unflags` este parámetro permite eliminar o anular ciertas indicadores de compilación que pudieran estar establecidas por defecto o en otras partes de la configuración. En la captura se observan los siguientes:

- -std=gnu++11: elimina la especificación del estándar gnu++11 para el compilador C++, permitiendo que se utilice otro estándar definido explícitamente (por ejemplo, gnu++17 en build_flags) sin dar lugar a incompatibilidades.
- -fno-exceptions: elimina el indicador que deshabilita el soporte para excepciones en C++. Así, queda habilitado el manejo de excepciones.

La deshabilitación de estos `flags` es necesaria para evitar incompatibilidades entre versiones de C++.

Bloque 7
Los últimos parámetros representan una de las aportaciones más distintivas de *platformIO*, y es su capacidad para gestionar dependencias de librerías externas automáticamente. *PlatformIO* es capaz de comprobar, descargar y compilar librerías tanto desde su propio

repositorio oficial como desde fuentes externas de forma automática. Esta característica es en buena parte responsable de la gran popularidad que este *framework* ha ganado a lo largo de los años.

- `lib_ldf_mode = deep`: Este parámetro configura el Library Dependency Filter (LDF) en modo `deep`. En este modo, *PlatformIO* analiza de manera recursiva todas las dependencias de las librerías utilizadas en el proyecto, incluyendo aquellas que son requeridas indirectamente por otras librerías. Esto garantiza que todas las dependencias necesarias se incluyan correctamente en el proceso de compilación, evitando errores por dependencias no resueltas.
- `lib_deps`: este parámetro especifica una lista de librerías externas que *PlatformIO* debe descargar y gestionar automáticamente para el proyecto. Se pueden incluir referencias directas a repositorios de GitHub (con ramas o *commits* específicos), lo que permite asegurar la reproducibilidad y estabilidad del entorno de desarrollo. Las librerías listadas serán instaladas automáticamente y estarán disponibles para su uso en el código fuente del proyecto. Además, se pueden comentar temporalmente algunas dependencias utilizando el carácter `#` o `;` al inicio de la línea.

Archivo *sdkconfig*

El archivo *sdkconfig* es un componente fundamental en los proyectos que utilizan ESP-IDF. Este archivo contiene la configuración de bajo nivel del *firmware*, incluyendo parámetros del sistema operativo embebido, opciones del *hardware*, controladores habilitados, niveles de *log*, configuración de *timers*, características de *WiFi* y *Bluetooth*, entre otros. En esencia, *sdkconfig* actúa como una base de configuración centralizada que define el comportamiento interno del *firmware* generado para el dispositivo embebido.

El archivo se genera y edita mediante la herramienta interactiva *menuconfig*, a la que se puede acceder desde *PlatformIO* mediante el comando `pio run -target menuconfig`. Esta utilidad presenta un menú basado en texto donde el desarrollador puede habilitar o deshabilitar funciones específicas del sistema y ajustar múltiples parámetros, aunque también es posible modificar manualmente el archivo.

Una vez guardados los cambios, el *framework* de *Espressif* utiliza *sdkconfig* para compilar el *firmware* de acuerdo con las opciones seleccionadas.

2.1.3. Node-RED

Figura 2.5: Logotipo de plataforma Node-RED

Según (Shanmugapriya et al., 2021):

> IBM Node-RED is a flow-based programming tool, for wiring together hardware devices, APIs and online services and is supported by the JS Foundation (...) The user can create various flows required, after constructing a flow the user can deploy and get his/her result. MQTT is the protocol used in this programming tool...
>
> IBM Node-RED es una herramienta de programación basada en flujos para cablear gráficamente dispositivos hardware, APIs y servicios online, y está apoyada por la fundación JS (...) el usuario puede crear los distintos flujos que se requieran, y después de construir un flujo el usuario podrá desplegarlo y obtener los resultados. El protocolo utilizado en esta herramienta de flujo es MQTT... [*traducción propia*]

(Shanmugapriya et al., 2021, Sección 2.2)

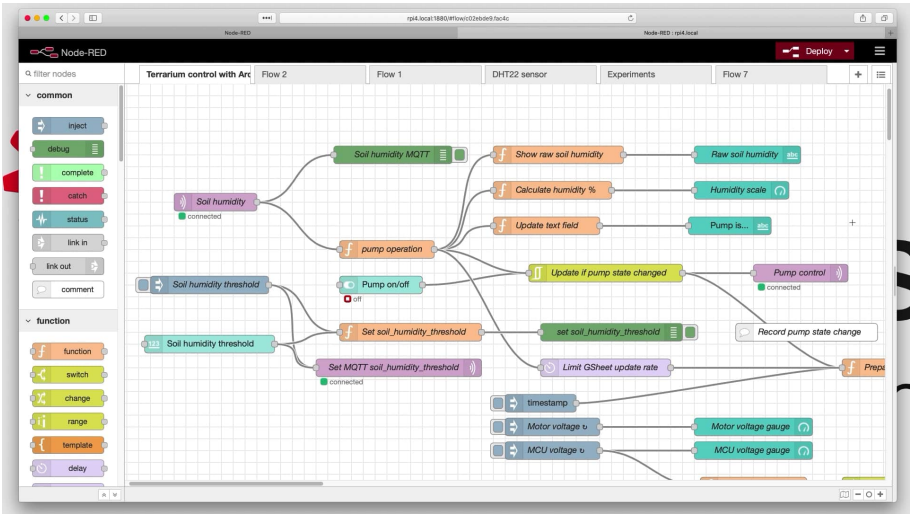


Figura 2.6: Ejemplo de un flujo en Node-RED. Adaptado de *1. Introduction to Node-RED*, por P. Dalmaris, 2024, Tech Explorations (https://techexplorations.com/guides/esp32/node-red-esp32-project/1-introduction-to-node-red/). Copyright 2024 por Tech Explorations.

El uso de Node-RED ha permitido llevar a cabo las pruebas necesarias para asegurar que el protocolo MQTT es funcional. Cuando el módulo ESP32Cam publica los mensajes pertinentes, estos son visibles a través de la vista de depuración de Node-RED.

De este modo, no solo es posible verificar que las comunicaciones funcionan correctamente, sino también analizar la estructura de las tramas enviadas y determinar su procedencia.

Además, Node-RED puede ser configurado para visualizar las imágenes capturadas y enviadas por HTTPS. Esto es posible gracias a la librería de terceros `node-red-contrib-image-output`.

En este proyecto se ha hecho uso de Node-RED tanto para la monitorización de las comunicaciones a través de protocolo MQTT, así como para previsualizar la imagen transmitida por el prototipo cuando se produce la solicitud de la misma. Este proceso se explica en detalle en el apartado 3.3.6.

Se puede consultar una imagen demostrativa en el anexo A.3.

2.1.4. Git

Para la gestión del desarrollo del proyecto, se ha hecho uso de Git como herramienta de control de versiones (Version Control System (VCS)).

Git es un sistema distribuido de control de versiones que rastrea las versiones de los archivos implicados. Suele utilizarse para controlar el código fuente de programadores que desarrollan software en colaboración. Entre los objetivos de diseño de Git se incluyen la velocidad, la integridad de los datos y la compatibilidad con flujos de trabajo distribuidos y no lineales (miles de ramas paralelas ejecutándose en distintos ordenadores). Como la mayoría de los otros sistemas de control de versiones distribuidos, y a diferencia de la mayoría de los sistemas cliente-servidor, Git mantiene una copia local de todo el repositorio, también conocido como "repo", con historial y capacidades de seguimiento de versiones, independiente del acceso a la red o de un servidor central.

Para el desarrollo del proyecto se ha elegido GitLab como plataforma principal de trabajo. Además de ofrecer un sistema de control de versiones basado en Git, GitLab integra diversas herramientas adicionales que han sido empleadas a lo largo del proyecto, como el tablero *Kanban* (descrita en el apartado 1.4.2) y *PlantUML* (analizada en detalle en el apartado 2.1.5). Esta integración facilita la gestión y el seguimiento eficiente de las distintas fases del desarrollo.

GitLab es una plataforma de gestión de repositorios de código basada en Git que permite a los equipos de desarrollo colaborar de manera eficiente en el ciclo de vida del desarrollo de software. Una de sus principales ventajas es que, a diferencia de otros servicios similares, GitLab puede ser autoalojado en la infraestructura propia de la organización, lo que proporciona un mayor control sobre la seguridad, la privacidad del código fuente.

Utilizar un sistema como GitLab ofrece beneficios clave, como la posibilidad de integración de más herramientas de gestión, automatización de pruebas y despliegues, y funcionalidades avanzadas de seguridad. Todo en un entorno centralizado y controlado por la propia organización. Esto facilita la trazabilidad, la colaboración y la privacidad del código, además de permitir la personalización y la aplicación de otras políticas internas.

2.1.5. PlantUML

PlantUML (UML es acrónimo de *Unified Modelling Language5*) es una herramienta de código abierto y una sintaxis para crear diagramas a partir de definiciones de texto plano. Utiliza una sintaxis simple para describir un tipo de diagrama, así como los elementos que forman el diagrama.

Usando esta sintaxis de texto plano, un servidor es el encargado de renderizar un diagrama a partir del texto entregado.

En este proyecto, todos los diagramas de secuencia y estado, mostrados en la sección 3.3.6, han sido generados usando esta herramienta.

Para una depuración eficaz del proyecto, se han utilizado, además de las herramientas previamente mencionadas, otras aplicaciones que integran en un solo entorno diversas utilidades especialmente útiles para el desarrollador.

2.2. Consideraciones de seguridad

Dado que este prototipo está diseñado para gestionar información que puede considerarse de carácter privado e incluso sensible (horarios de entrada y salida, matrículas de vehículos, accesos a viviendas o parques residenciales...) es indispensable establecer los criterios de seguridad que han de ser aplicados para la operación del dispositivo.

2.2.1. Configuración de seguridad

Se ha habilitado la posibilidad de configurar el dispositivo para operar bajo diferentes modos de seguridad, los cuales se describen en detalle en este apartado.

Para activar estos modos, el desarrollador puede definir las configuraciones de seguridad en varios archivos JSON, que posteriormente serán grabados directamente en la memoria. Para habilitar un modo de seguridad concreto, basta con establecer el parámetro `enabled: true;` en el archivo correspondiente.

Las opciones de seguridad disponibles incluyen: *https*, *ssl* y *vpn*. Además, en estos archivos se almacenan los certificados necesarios para la conexión segura. Al ser grabados en la memoria física, estos archivos permanecen accesibles tras el reinicio del dispositivo.

A continuación se describe el contenido de los archivos mencionados:

https.json
El contenido de este archivo se presenta en el listado 2.1.

```
1               {
2                 "user" : "poc",
3                 "pass" : "1233445*@",
4                 "enabled" : true
5               }
6
```

Listado 2.1: Contenido del archivo https.json

Este archivo incluye los datos correspondientes a las credenciales de acceso (usuario y contraseña), así como el *flag* que indica si el protocolo HTTPS se encuentra activo.

mqtt.json
El contenido de este archivo se presenta en el listado 2.2.

```
1             {
2               "user" : "ttb",
3               "pass" : "55487",
4               "ssl_enabled" : false,
5               "ca_cert": "Certificado de la autoridad de
  certificacion",
6               "client_crt" : "Certificado de cliente",
7               "client_key" : "Certificado privado"
8             }
9
```

Listado 2.2: Contenido del archivo mqtt.json

Este archivo contiene la información necesaria para el establecimiento de las conexiones con el *broker* de MQTT.

`user y pass` especifican las credenciales de acceso, permitiendo la autenticación del usuario en el sistema.

`ssl_enabled` es un valor booleano que indica si la comunicación segura mediante SSL está activada o no.

`ca_cert` contiene un certificado digital de la utilizado para verificar la autenticidad de las conexiones seguras establecidas con el dispositivo.

`client_crt` y `client_key` son el certificado y la clave privada del cliente, respectivamente. Estos archivos permiten la autenticación mutua entre el dispositivo y el *broker*, asegurando que ambos extremos de la comunicación sean legítimos y confiables.

ota.json
El contenido de este archivo se presenta en el código 2.3.

```
{
  "user" : "ttb",
  "pass" : "00254",
}

```

Listado 2.3: Contenido del archivo ota.json

El archivo, más sencillo que los anteriores, almacena las credenciales solicitadas para llevar a cabo la conexión con el servidor OTA.

vpn.json
El contenido de este archivo se presenta en el listado 2.4.

```
{
  "enabled" : false,
  "local_ip" : "10.0.10.4",
  "endpoint" : "waspbrain.es",
  "port" : 51822,
  "private_key" : "Clave_privada",
  "public_key" : "Clave_publica"
}

```

Listado 2.4: Contenido del archivo vpn.json

`enabled` es un valor booleano que indica si la funcionalidad de VPN está activada (*true*) o desactivada (*false*).

`local_ip` especifica la dirección IP local que el dispositivo utilizará dentro de la red VPN.

`endpoint` define el dominio o dirección del servidor VPN al que se conectará el dispositivo.

`port` indica el puerto de comunicación utilizado para establecer la conexión VPN.

`private_key` almacena la clave privada del dispositivo, necesaria para la autenticación segura y el cifrado de las comunicaciones dentro de la VPN.

`public_key` contiene la clave pública asociada, que puede ser utilizada por otros nodos o servidores para verificar la identidad del dispositivo durante el establecimiento de la conexión.

2.2.2. Certificado SSL

El certificado SSL es aplicado al protocolo Message Queue Telemetry Transport (MQTT) con el objetivo de cifrar las comunicaciones entre el dispositivo y el servidor de mensajería. De este modo, se garantiza la confidencialidad e integridad de los datos transmitidos, evitando que información sensible, como identificadores de usuario o comandos de control, pueda ser interceptada o alterada por terceros no autorizados.

En el caso de este proyecto, el protocolo a nivel de presentación de datos utilizado es SSL, y el algoritmo de cifrado asimétrico empleado es Rivest, Shamir, Adleman (RSA).

Cuando se habilita SSL en una comunicación MQTT, se establece un canal seguro entre el cliente (el dispositivo) y el servidor de mensajería (*broker*). Este proceso se basa en el uso de certificados digitales y criptografía de clave pública para autenticar a las partes y cifrar los datos transmitidos. El procedimiento técnico se desarrolla en varias fases:

1. Negociación del protocolo (*Handshake*): al iniciar la conexión, el cliente MQTT solicita establecer una sesión segura con el *broker*. El mismo responde enviando su certificado digital, que contiene su clave pública y está firmado por una Certification Authority (CA) confiable. Existen CA públicas, que emiten certificados reconocidos globalmente y son confiables para navegadores y usuarios en general, y CA privadas, usadas internamente en organizaciones para proteger redes y sistemas propios
2. Verificación del certificado: el cliente valida el certificado recibido comprobando que:
   - Está firmado por una CA reconocida.
   - No ha expirado ni ha sido revocado.
   - El nombre del servidor coincide con el certificado.
   - Si alguna de estas comprobaciones falla, la conexión se rechaza.
3. Intercambio de claves y establecimiento de sesión: una vez verificado el certificado, el cliente genera una clave de sesión simétrica, y la cifra utilizando la clave pública del *broker*. Solo el *broker*, que posee la clave privada correspondiente, puede descifrar este mensaje y obtener la clave de sesión.
4. Cifrado de la comunicación: a partir de este momento, todos los datos transmitidos entre el cliente y el *broker* se cifran utilizando la clave de sesión simétrica. Esto garantiza que, incluso si un atacante intercepta los paquetes, no podrá leer su contenido sin la clave correcta.

2.2.3. Conexión a VPN

En el archivo JSON correspondiente, mencionado en el apartado 2.2.1 se incluyen los datos necesarios para habilitar la conexión VPN que se ha desplegado experimentalmente para la realización del proyecto.

Entre estos datos, además de la clave de cifrado pública y privada, se incluye la Uniform Resource Locator (URL) de conexión y puerto. Para facilitar el proceso de depuración, se ha establecido una IP asignada ya en el archivo de configuración. Sin embargo, el servidor VPN es el que en condiciones de producción generará direcciones IP estáticas que serán asignadas a todos los dispositivos conectados a la misma red.

El servidor de VPN utilizado es *WireGuard*, que destaca por su eficiencia, simplicidad y alto nivel de seguridad. Otra de las características que hacen a *WireGuard* una opción deseable es que, debido a su ligereza y optimización, es la única opción disponible actualmente que posibilita la instalación de un servidor VPN en sistemas embebidos.

La configuración de cada dispositivo cliente se realiza mediante archivos `.conf` generados a partir de la información contenida en los archivos JSON, los cuales incluyen los pares de claves (pública y privada), la *ip* asignada al cliente, la clave pública del servidor, la dirección IP y el puerto de escucha del servidor (por defecto, el puerto User Datagram Protocol (UDP) 51822).

Durante el proceso de conexión, el cliente utiliza sus claves para autenticarse y cifra el tráfico dirigido al servidor usando la clave pública de este. El servidor, por su parte, valida los datos de conexión del cliente y, si está autorizado, permite el establecimiento del túnel seguro. Toda la comunicación entre el cliente y el servidor viaja cifrada de extremo a extremo, garantizando la confidencialidad e integridad de los datos transmitidos a través de la red VPN.

2.2.4. HTTPS

El dispositivo cuenta con capacidad de transmisión a través del protocolo HTTPS. Los certificados necesarios para establecer la comunicación segura se almacenan en el sistema de archivos Serial Peripheral Interface Flash File System (SPIFFS), que permite guardar archivos en la memoria flash del ESP32 de manera eficiente.

HTTPS es una extensión de HTTP que incorpora una capa de seguridad basada en Transport Layer Secure (TLS)- Esta capa garantiza el cifrado de los datos, la autenticación del servidor y la integridad de la información transmitida, lo que convierte a HTTPS en una elección ideal para aplicaciones sensibles, aplicable a el envío de imágenes desde una cámara remota. En el contexto del ESP32, el uso de HTTPS se implementa mediante librerías específicas: en este proyecto se utiliza *WiFiClientSecure* del núcleo de *Arduino*.

Esta funcionalidad ha sido aplicada a la transmisión de imágenes hacia el servidor. A través de este protocolo, la imagen puede ser retransmitida de dos formas:

1. **Array de bytes**: transmisión en formato array de bytes, que es el archivo generado por el sensor y se almacena en una variable tipo *uint8_t**. Este enfoque permite enviar los datos en su forma binaria original, sin necesidad de conversión, lo cual es más eficiente en términos de ancho de banda y consumo de recursos.

2. **Base64**: la codificación Base64 convierte datos binarios en una cadena American Standard Code for Information Interchange (ASCII) compuesta por caracteres alfanuméricos. Este método es útil cuando se requiere transmitir datos binarios a través de canales que solo admiten texto, como JSON, formularios MQTT u otros servicios sin soporte para contenido binario. No obstante, la conversión a *Base64* incrementa el tamaño de los datos, afectando al rendimiento de la transmisión en redes con ancho de banda limitado.

Dado que el formato *Base64* genera archivos más pesados, se ha configurado por defecto la transmisión de imágenes codificadas en *array* de *bytes*.

2.3. Protocolos

2.3.1. MQTT

Según se puede consultar en (Amazon Web Services, Inc., 2024), el protocolo MQTT es un protocolo de mensajería basado en estándares, o un conjunto de reglas, que se utiliza para la comunicación de un equipo a otro. Los sensores inteligentes, los dispositivos portátiles y otros dispositivos de IoT generalmente tienen que transmitir y recibir datos a través de una red con recursos restringidos y un ancho de banda limitado. Estos dispositivos IoT utilizan MQTT para la transmisión de datos, ya que resulta fácil de implementar y puede comunicar datos IoT de manera eficiente. MQTT admite la mensajería entre dispositivos a la nube y la nube al dispositivo.

En la figura 2.7 se muestra un diagrama de funcionamiento del protocolo.

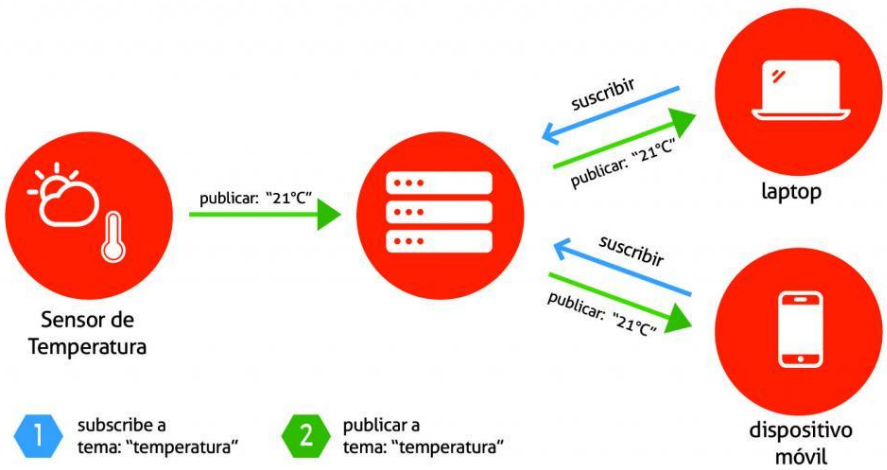


Figura 2.7: Topología de red en estrella basada en MQTT (Adaptada de `https://www.factor.mx/portal/wp-content/uploads/2018/10/mqtt1-1024x540.jpg`)

El protocolo MQTT funciona según los principios del modelo de publicación o suscripción. En la comunicación de red tradicional, los clientes y servidores se comunican directamente entre sí, donde los clientes solicitan recursos o datos del servidor y, a continuación, el servidor procesa y envía una respuesta. Sin embargo, MQTT utiliza un patrón

de publicación o suscripción para desacoplar el remitente del mensaje (editor), del receptor del mensaje (suscriptor), y un tercer componente, denominado *broker* de mensajes, controla la comunicación entre editores y suscriptores. El trabajo del *broker* consiste en filtrar todos los mensajes entrantes de los editores y distribuirlos correctamente a los suscriptores. Los componentes MQTT, en resumen, son los siguientes:

1. **Cliente MQTT:** cualquier dispositivo que publica o se suscribe a un *topic* MQTT. El módulo ESP32Cam se enmarca dentro de esta categoría.
2. ***Broker* MQTT:** sistema *back-end* que coordina los mensajes entre clientes. El *broker* es responsable de recibir y filtrar mensajes, identificar a los clientes suscritos a cada mensaje y hacérselos llegar.

La escalabilidad de una red de comunicación basada en MQTT, unida a sus bajos requisitos de procesamiento, la convierte en una opción muy popular. Es muy común encontrarla implantada en gran variedad de proyectos tanto IoT como de *software*. Como explica Hillar, 2017, Cap 1.2:

> A publisher that requires sending a message to hundreds of clients can do it with a single publish operation to a broker. The broker is responsible for sending the published message to all the clients that have subscribed to the appropriate topic. Because publishers and subscribers are decoupled, the publisher doesn't know whether there is any subscriber that is going to listen to the messages it is going to send. Hence, sometimes it is necessary to make the subscriber become a publisher too and to publish a message indicating that it has received and processed a message. The specific requirements depend on the kind of solution we are building. MQTT offers many features that make our lives easier in many of the scenarios we have been analyzing.

(Hillar, 2017, Cap 1.2)

> Un *publicador* que necesite enviar un mensaje a cientos de clientes puede hacerlo con una única operación de publicación a un *broker*. El *broker* se encarga de enviar el mensaje publicado a todos los clientes suscritos al tema correspondiente. Como los publicadores y los suscriptores están desacoplados, el publicador no sabe si hay algún suscriptor que vaya a escuchar los mensajes que va a enviar. Por ello, a veces es necesario hacer que el suscriptor se convierta también en publicador y publique un mensaje indicando que ha recibido y procesado un mensaje. Los requisitos específicos dependen del tipo de solución que estemos construyendo. MQTT ofrece muchas funcionalidades que nos facilitan la vida en muchos de los escenarios que hemos estado analizando.

(Hillar, 2017, Cap 1.2) [*Traducción propia*]

CAPÍTULO 3

Desarrollo de la solución

En esta sección se analizan en detalle los principios de desarrollo adoptados en este proyecto. Se abordan los criterios estructurales que otorgan cohesión al sistema y aseguran la compatibilidad tanto entre los distintos componentes integrados como con futuras actualizaciones que puedan incorporarse.

3.1. Hardware

A continuación se explican los elementos *hardwarae* más representativos para el proyecto, incluyendo el *hardware* utilizado, especificaciones técnicas, y consideraciones técnicas relevantes para el desarrollo del mismo.

3.1.1. ESP32Cam

El ESP32Cam es un módulo de desarrollo compacto y de bajo costo que combina un microcontrolador ESP32-S con conectividad *WiFi*, *Bluetooth* y la cámara digital escogida tipo OV2640. Esta integración le permite capturar, procesar y transmitir imágenes o video, convirtiéndolo en una solución ideal para proyectos de visión artificial, monitoreo remoto, y aplicaciones *iot* con capacidades de imagen.

Una de sus mayores ventajas es su capacidad de operar sin necesidad de microcontroladores adicionales. Puede programarse directamente desde el entorno *Arduino IDE*, o usando *frameworks* como ESP-IDF a través de interfaz Universal Asynchronous Receiver/Transmitter (UART). Además, existen múltiples librerías de código abierto que permiten ampliar sus funcionalidades. Las especificaciones del ESP32Cam facilitadas por el fabricante han sido recogidas en la tabla 3.1.

En el caso de nuestro proyecto, es esencial la capacidad de cómputo para mantener en funcionamiento los distintos hilos en paralelo que componen la arquitectura de software del proyecto.

Categoría	**Especificación**
Procesador	Dual-core 32-bit LX6 CPU, hasta 240 MHz, hasta 600 DMIPS
Memoria	520 KB SRAM interna, 4 MB PSRAM externa, 32 Mbit SPI Flash
Conectividad	Wi-Fi 802.11 b/g/n/e/i, Bluetooth 4.2 BR/EDR y BLE
Modos de operación	STA, AP, STA+AP, SmartConfig, AirKiss
Cámara soportada	OV2640 y OV7670, JPEG (OV2640), BMP, escala de grises
Interfaces	UART, SPI, I2C, PWM, ADC, DAC
UART disponibles	UART0 (GPIO1, GPIO3), UART2 (GPIO16, GPIO17)
Almacenamiento	Tarjeta TF de hasta 4 GB
GPIO	9 pines disponibles
Alimentación	5V, consumo de 6 mA (deep sleep) hasta 310 mA (con flash activo)
Frecuencia Wi-Fi	2.412 – 2.484 GHz
Antena	PCB integrada, ganancia 2 dBi
Sensibilidad Wi-Fi	Hasta -90 dBm (1 Mbps CCK), -67 dBm (MCS7)
Seguridad	WPA/WPA2/WPA2-Enterprise/WPS
Dimensiones	$27 \times 40.5 \times 4.5$ mm (± 0.2 mm)
Peso	10 g
Temperatura de operación	-20 °C a 85 °C
Condiciones de almacenamiento	-40 °C a 90 °C, $< 90\,\%$ humedad

Tabla 3.1: Especificaciones técnicas del módulo ESP32Cam

3.1.2. Conexión hardware

En la figura 3.1 se ofrece un ejemplo de cómo se va a efectuar la conexión física entre ambos componentes para posibilitar la comunicación por protocolo RS232 entre ambos.

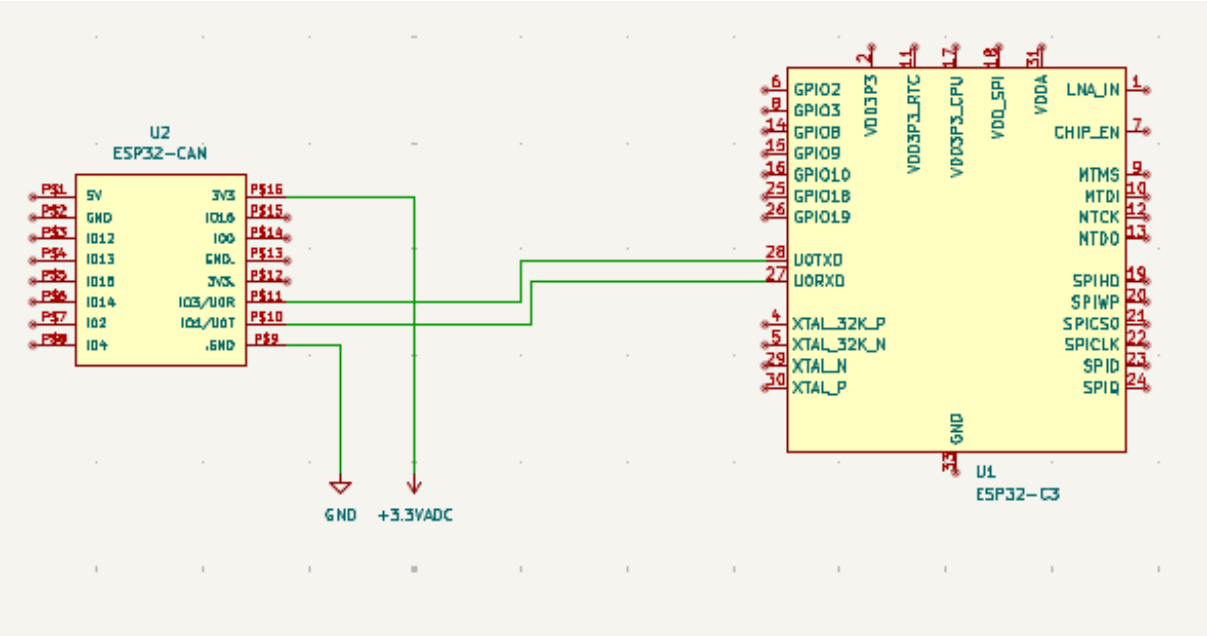


Figura 3.1: Ejemplo de diagrama de conexionado vía RS232 entre dos dispositivos

La conexión realizada entre ESP32Cam y la UC utiliza el puerto de conexión UART0 dentro del ESP32Cam.

Atendiendo al *datasheet* (Espressif Systems, 2020), el mencionado módulo cuenta con 2 puertos de comunicación UART, correspondientemente UART0 y UART2. Sin embargo, cuando el módulo hace uso de una cámara conectada, sólo el UART0 (General Purpose Input-Output (GPIO)1 y GPIO 3) está disponible para el usuario. En el caso del UART2, el pin Rx (GPIO 16) está dedicado a funciones relacionados con la comunicación con el módulo de cámara.

Al utilizarse la comunicación por UART0, los mensajes transmitidos entre la UC y el ESP32Cam son, a su vez, transmitidos por la propia interfaz Universal Serial Bus (USB) presente en la placa de desarrollo.

Dado que en condiciones normales de operación no se requiere la conexión mediante USB, esta limitación no representa un inconveniente significativo para el funcionamiento del sistema. Sin embargo, durante la fase de desarrollo, la interfaz USB resulta fundamental, ya que permite visualizar y monitorizar en tiempo real los paquetes de datos transmitidos a través de la UART0. Esta funcionalidad facilita considerablemente las tareas de depuración y ajuste del programa, contribuyendo a un desarrollo más eficiente y preciso.

3.1.3. Mapa de memoria

El módulo ESP32Cam cuenta con memoria física accesible por el usuario, cuya gestión eficiente es fundamental para el correcto funcionamiento y la flexibilidad del sistema. Para

aprovechar al máximo esta memoria, es necesario realizar una partición adecuada, que permita asignar espacios específicos a las distintas funcionalidades del dispositivo, como el almacenamiento de datos, la ejecución de aplicaciones o la realización de actualizaciones remotas.

Este proceso de particionado no solo optimiza el uso de los recursos disponibles, sino que también contribuye a la seguridad y estabilidad del sistema, permitiendo aislar áreas críticas y facilitar tareas de mantenimiento o recuperación ante fallos.

El mapa de memoria se encuentra redactado en formato `.csv`, dentro del archivo `huge_app.csv`. Su contenido se muestra en el listado 3.1.

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x5000,
otadata,  data, ota,     0xe000,  0x2000,
app0,     app,  ota\_0,   0x10000, 0x300000,
spiffs,   data, spiffs,  0x310000,0xE0000,
coredump, data, coredump,0x3F0000,0x10000,

```

Listado 3.1: Contenido del archivo huge_app.csv

El archivo `huge_app.csv` es una tabla de particiones utilizada en proyectos con microcontroladores ESP32, especialmente aquellos desarrollados con el *framework* ESP-IDF. Su función es definir cómo se distribuye la memoria flash interna del microcontrolador, estableciendo qué porciones se reservan para distintas funcionalidades del sistema. Esto incluye secciones como la aplicación principal (app0), almacenamiento no volátil (nvs), sistema de archivos (spiffs o littlefs), espacio reservado para actualizaciones Over-The-Air (OTA), y otros bloques personalizados según las necesidades del proyecto.

`Huge_app.csv` está pensada para proyectos de cierta complejidad o tamaño, ya que asigna un mayor espacio a la partición destinada a la aplicación principal. Esto resulta especialmente útil en *firmwares* que integran múltiples funcionalidades, librerías pesadas, procesamiento de imágenes, servidores web embebidos o manejo de archivos multimedia. En términos prácticos, esto permite que el binario generado durante la compilación no exceda el espacio reservado en la memoria *flash* y, por tanto, pueda ejecutarse sin errores por falta de memoria.

La inclusión de esta tabla de particiones en *PlatformIO* se realiza especificándola en el archivo de configuración *platformio.ini*, como se ha explicado en el apartado 2.1.2. Concretamente, en la línea

```
board_build.partitions = huge_app.csv
```

Durante el proceso de compilación, en este caso, el *framework* de *Espressif* tomará esta configuración y generará el *firmware* de acuerdo con el esquema de memoria definido. El archivo *huge_ app.csv* puede ser personalizado o reemplazado por versiones modificadas, ya que su formato es simple y legible Comma-Separated Values (CSV), lo que facilita la adaptación a requerimientos particulares. Este enfoque flexible permite un control detallado sobre el uso de la memoria *flash* y garantiza que los recursos del microcontrolador se utilicen de forma óptima y segura.

Además de la memoria flash principal, algunos modelos de la familia ESP32, como el ESP32Cam, usado en este proyecto, incorporan Pseudo Static Random Access Memory

(PSRAM), que amplía significativamente la capacidad de almacenamiento temporal del sistema. La PSRAM es especialmente útil para aplicaciones que requieren grandes volúmenes de datos en tiempo real, como el procesamiento de imágenes o el manejo de *buffers* de vídeo.

Aunque la PSRAM no se gestiona mediante la tabla de particiones de la memoria flash, su uso puede ser programado explícitamente mediante funciones específicas. Para habilitar el uso de la misma, sí que es necesaria la inclusión de otro *flag* en el archivo *platformio.ini*: `DBOARD_HAS_PSRAM` como se ha mencionado en el apartado 2.1.2.

El principal uso que se le aplica a la memoria física del ESP32Cam en este proyecto es, en primer lugar, el almacenamiento de credenciales de seguridad que posibiliten realizar transmisiones de datos cifradas, así como el almacenamiento de parámetros de red que permitan reanudar la conexión si se produce una parada en la alimentación del prototipo o cualquier evento que provoque la desconexión de la red temporalmente.

3.1.4. Estándar RS232

El estándar Recommended Standard 232 (RS232), fue introducido en 1960 por la Electronic Industries Association (EIA) en los Estados Unidos. Este estándar se desarrolló para proporcionar una base común para la transmisión de datos binarios en serie entre dispositivos como computadoras, módems y otros tipos de equipos periféricos.

El protocolo original era capaz de operar a velocidades de transferencia de datos de hasta 20 *kilobits* por segundo, en una longitud máxima de cable de 15 metros. Se caracteriza por usar una serie de señales de voltaje para representar datos binarios, con voltajes positivos que representan un 0 binario y voltajes negativos que representan un 1 binario.

Aunque existen numerosos protocolos de comunicación más modernos, como RS422, RS485 y USB, que ofrecen ventajas como mayores tasas de transferencia o mejor inmunidad al ruido eléctrico, RS232 continúa siendo una opción válida en muchos contextos gracias a una serie de características que favorecen su uso en aplicaciones específicas.

En primer lugar, la simplicidad de implementación de RS232 representa una ventaja significativa frente a otros protocolos más complejos. Su funcionamiento es relativamente fácil de comprender, lo que lo convierte en una alternativa accesible para desarrolladores e ingenieros que requieren una solución de comunicación fiable y con un bajo coste de desarrollo.

Asimismo, la compatibilidad con una amplia gama de dispositivos, tanto antiguos como modernos, otorga a RS232 un valor añadido en entornos donde la inter-operabilidad con hardware legado resulta imprescindible. En muchas ocasiones, RS232 constituye la única vía de comunicación viable con determinados equipos industriales o embebidos.

Otra de sus fortalezas radica en la capacidad de comunicación dúplex completo, es decir, la posibilidad de transmitir y recibir datos de manera simultánea.

A continuación, se describen los aspectos técnicos más relevantes:

- **Niveles de voltaje:** RS232 utiliza tensiones eléctricas específicas para representar los valores lógicos. Un voltaje comprendido entre –3 V y –15 V representa un “1” lógico (también denominado marca), mientras que un voltaje entre +3 V y +15 V

representa un "0" lógico (o espacio). Este esquema de voltaje inverso distingue a RS232 de otros estándares seriales más modernos.

- **Transmisión dúplex completo:** el protocolo permite comunicación bidireccional simultánea, es decir, los datos pueden ser enviados y recibidos al mismo tiempo, lo cual es ventajoso en aplicaciones que requieren interacción continua entre dispositivos.
- **Estructura de trama:** como se muestra en la figura 3.2, cada unidad de datos transmitida incluye una estructura estándar compuesta por un bit de inicio, seguido de 5 a 9 bits de datos, un bit de paridad opcional para detección de errores, y uno o dos bits de parada. Esta estructura se conoce como marco (*frame*) y permite que el receptor identifique correctamente los límites de cada *byte*.

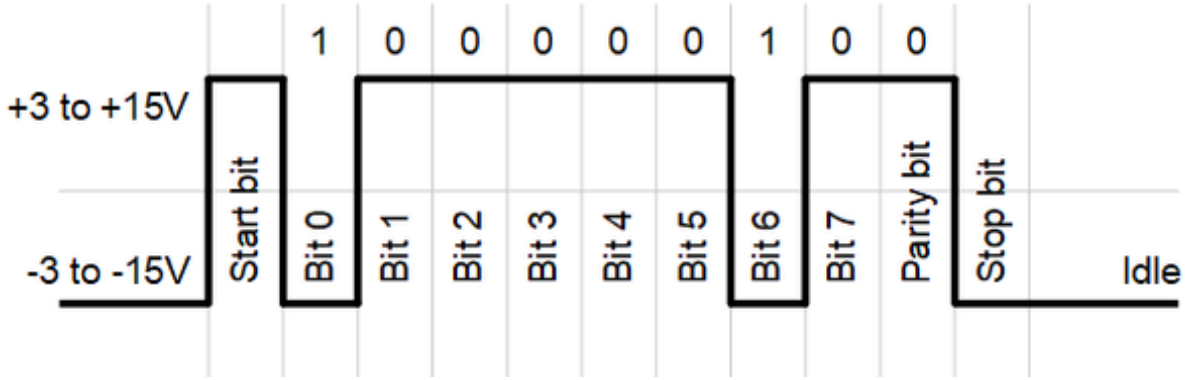


Figura 3.2: Estructura de la trama en protocolo RS232

- **Tasa de baudios:** la velocidad de transmisión en RS232 se especifica en baudios, que corresponde al número de transiciones de señal por segundo. Normalmente, esta tasa coincide con los *baudios* por segundo (bps) cuando no hay codificación adicional, y debe estar configurada de forma idéntica en ambos extremos del canal de comunicación. Se configura un *baudrate* de 115200bps para las comunicaciones entre ESP32Cam y UC.
- **Líneas de control:** para gestionar el flujo de datos y garantizar la integridad de la transmisión, RS232 incorpora múltiples líneas de control.
  - DTR (*Data Terminal Ready*)
  - DSR (*Data Set Ready*)
  - RTS (*Request to Send*)
  - CTS (*Clear to Send*)

  Estas señales permiten coordinar el inicio y la disponibilidad de la transmisión entre emisor y receptor, lo cual es especialmente útil en conexiones con dispositivos más complejos. Se ha hecho mención a las líneas DTR y RTS en el apartado 2.1.2.

  En el caso de este proyecto, se ha prescindido de estas líneas de control debido a la limitación de conexiones disponibles para comunicaciones en el módulo ESP32Cam.

3.1.5. Consideraciones técnicas

La transmisión de imágenes a demanda desde el módulo de cámara produce imágenes con una resolución de 640x480 px. Es decir, genera archivos en Red-Green-Blue (RGB) de un tamaño de aproximadamente 950 KB.

Si no se contara con conectividad *WiFi*, la transmisión desde el módulo de cámara hasta la UC se vería obligada a utilizar el puerto serie RS232.

La velocidad de conexión estándar es de 115200 Bd para el protocolo serie. Esta velocidad asegura una estabilidad de la señal suficiente para la interconexión entre distintos dispositivos, y garantiza hasta cierto punto la integridad de la información, especialmente importante a la hora de transmitir una imagen.

Haciendo el cálculo aproximado:

$$\frac{950\,\text{KB}}{115200\,\text{bps}} = 65{,}97\,\text{s} \tag{3.1}$$

Tomaría más de 1 minuto transmitir una única imagen (o *frame*) desde el módulo de cámara a la UC por puerto serie, suponiendo que las condiciones de la transmisión permitan aplicar la máxima velocidad.

Los tiempos de carga terminarían por bloquear procesos esenciales dentro del dispositivo, consumiendo recursos innecesariamente, y generando en el mejor de los casos unos tiempos de espera incómodos para el usuario final.

La existencia del módulo integrado *WiFi*, permite realizar la transmisión de imagen de forma independiente a la UC, evitando el consumo excesivo de recursos para llevar a cabo la transmisión si debiera ser a través de la UC.

Entre otras ventajas, cabe mencionar la posibilidad de visualización de la imagen en un servidor remoto prácticamente en tiempo real.

3.2. Software

3.2.1. Lenguaje

El lenguaje de programación utilizado es C++17. La elección de C++ como lenguaje principal para el desarrollo del sistema en ESP32Cam se fundamenta en múltiples ventajas técnicas y estratégicas, especialmente en el contexto de sistemas embebidos y dispositivos orientados al IoT. Además, C++ ofrece un equilibrio óptimo entre el control de bajo nivel necesario para interactuar directamente con el hardware del ESP32 (como UART, GPIO y periféricos de cámara) y las ventajas de un lenguaje orientado a objetos, lo que facilita la modularidad y escalabilidad del proyecto. Esta característica es particularmente relevante dado que se prevé una evolución futura del sistema, por lo que una arquitectura extensible resulta fundamental.

Cabe mencionar que al utilizar ESP-IDF como entorno de desarrollo, C++ se integra de manera nativa con las distintas Application Programming Interface (API) y librerías proporcionadas por *Espressif*, lo que permite aprovechar plenamente las capacidades del

hardware. Si bien alternativas como *Python* o *Arduino* pueden ofrecer una curva de aprendizaje inicial más amigable, resultan menos adecuadas en escenarios donde se requiere un mayor nivel de flexibilidad, robustez y personalización del código.

Por otro lado, *Python* es un lenguaje de más alto nivel que C++, lo que implica diferencias significativas en el rendimiento y la velocidad de ejecución de los procesos programados. El uso de C++ también permite implementar estructuras más complejas y mantener una gestión precisa de recursos. A esto se suma la reusabilidad del código. Dado que ambos módulos de *hardware*, tanto ESP32Cam como UC comparten tanto el lenguaje como la modularidad del código, ha sido posible reutilizar ciertas funcionalidades originalmente programadas en el módulo de UC para implementarlas en el propio ESP32Cam. Esta modularidad será de gran provecho en futuras mejoras, al facilitar la implementación de nuevas funcionalidades.

Asimismo, el soporte para programación genérica y la reutilización de componentes favorece un desarrollo más limpio y mantenible, especialmente considerando que el mantenimiento futuro del sistema recaerá en el mismo autor del proyecto. Por todo lo anterior, C++ se consolida como la opción más adecuada para este dispositivo IoT, combinando eficiencia, control y capacidad de evolución en el tiempo.

3.2.2. Librerías de terceros

Gracias a las capacidades para gestión de dependencias del entorno *PlatformIO*, mencionadas en el apartado 2.1.2, es posible listar de forma ordenada todas las librerías externas utilizadas en el proyecto. Todas ellas se mencionan en la tabla 3.2

Librería (.h)	**Descripción**	**Enlace a GitHub**
MQTTPubSubClient	Cliente MQTT para publicación y suscripción de mensajes en dispositivos embebidos	`https://github.com/hideakitai/MQTTPubSubClient.git`
ESP32Ping	Permite realizar pruebas de conectividad de red (ping) en el ESP32	`https://github.com/marian-craciunescu/ESP32Ping.git`
function-fsm	Implementación simple de máquinas de estados finitos en C++	`https://github.com/JRVeale/function-fsm.git`
Timer	Gestión y control de temporizadores en aplicaciones embebidas	`https://github.com/brunocalou/Timer.git`
ESPAsyncWebServer	Servidor web asíncrono para ESP32 y ESP8266	`https://github.com/me-no-dev/ESPAsyncWebServer.git`
ElegantOTA	Actualizaciones OTA (Over-The-Air) fáciles para ESP32/ESP8266	`https://github.com/ayushsharma82/ElegantOTA.git`
WireGuard-ESP32-Arduino	Implementación de cliente WireGuard VPN para ESP32	`https://github.com/ciniml/WireGuard-ESP32\protect\@normalcr\relax-Arduino.git`
ESP32QRCodeReader	Lectura y decodificación de códigos QR utilizando el ESP32	`https://github.com/alvarowolfx/ESP32QRCodeReader.git`

Tabla 3.2: Librerías utilizadas en el proyecto, descripción y enlace a GitHub

3.2.3. FreeRTOS

Dado que el proyecto requiere la ejecución simultánea de múltiples hilos, es fundamental disponer de una herramienta que permita gestionar eficazmente tareas en un entorno multitarea real.

Para ello, se utiliza la librería FreeRTOS, un sistema operativo de tiempo real ligero y

de código abierto diseñado para sistemas embebidos. FreeRTOS facilita la creación, planificación y sincronización de tareas, permitiendo que se ejecuten de manera concurrente sin que unas bloqueen a otras. Además, proporciona mecanismos avanzados de comunicación y control, como colas, semáforos y notificaciones entre tareas, optimizando así el aprovechamiento de los recursos del microcontrolador y mejorando la eficiencia y robustez del sistema.

Gracias a FreeRTOS, es posible gestionar múltiples procesos concurrentes de forma eficiente, lo que resulta especialmente útil en dispositivos embebidos como el ESP32Cam, donde distintas funcionalidades (como la captura de imágenes, la comunicación por red o el procesamiento de datos) deben ejecutarse sin interferencias y de manera coordinada.

Atendiendo a la definición de FreeRTOS en (Amazon Web Services, Inc., 2024)

> El kernel de FreeRTOS es un sistema operativo en tiempo real que admite numerosas arquitecturas. Sus fundamentos son ideales para crear aplicaciones de microcontroladores integrados. Proporciona:
>
> - Un programador multitareas.
> - Varias opciones de asignación de memoria (incluida la opción de crear sistemas asignados de forma totalmente estática).
> - Primitivos de coordinación entre tareas, como notificaciones de tareas, colas de mensajes, varios tipos de semáforos y búferes de transmisión y mensajes.
> - Compatibilidad para el multiprocesamiento simétrico (SMP) en microcontroladores multi-núcleo.

El uso de este recurso permite al dispositivo trabajar con distintas tareas de forma concurrente. También permite de esta manera optimizar el uso de los dos núcleos con los que cuenta el procesador. Esta estructura de tareas concurrentes se encuentra explicada en el siguiente apartado 3.3.

3.3. Diseño de la solución

A continuación se documentan los principales elementos que conforman la solución propuesta.

Se hace hincapié en aquellos aspectos que estructuran el flujo de información, estandarización de comunicaciones, y diagramas de funcionamiento.

3.3.1. Estructura del software

El proyecto se basa en una arquitectura orientada a eventos. Este tipo de arquitectura es definida por (Lazzari y Farias, 2023) de la siguiente forma:

> "In event-based architecture, components only publish data without knowing the other components or who will consume and react to the data, promoting the separation of computation and event publishing from any subsequent processing (Fiege et al., 2002).

> Furthermore, their communication is asynchronous in the producer/consumer model, and both are independent of each other (Falatiuk et al., 2019). Consequently, promoting loose coupling between components is why event-driven architecture has become predominant in large-scale distributed applications (Fiege et al., 2002)."
>
> "En una arquitectura basada en eventos, los componentes solo publican datos sin conocer a los demás componentes ni quién consumirá o reaccionará a esos datos, promoviendo así la separación entre la computación y la publicación de eventos respecto a cualquier procesamiento posterior (Fiege et al., 2002). Además, su comunicación es asíncrona, dentro del modelo productor/consumidor, y ambos son independientes entre sí (Falatiuk et al., 2019). Como consecuencia, la promoción del acoplamiento débil entre componentes es la razón por la cual la arquitectura orientada a eventos se ha vuelto predominante en aplicaciones distribuidas a gran escala (Fiege et al., 2002)."*[traducción propia]*(Lazzari L. y Farias K., 2023, pag. 340. apdo 2.2).

Siguiendo esta definición, la solución propuesta se compone de distintos elementos que interactúan entre ellos de distintas formas, con roles bien definidos dentro de los flujos de procesos. Los elementos de software básicos que se pueden categorizar en el código son los siguientes:

Tareas

Las tareas son procesos que se ejecutan de forma concurrente en los dos núcleos del ESP32Cam. En el apartado 3.3.6, con este motivo, se hace referencia a estas tareas indistintamente como "hilos" a lo largo de todo el documento. Dentro de la arquitectura del proyecto, se consideran parte de esta categoría las tareas o hilos denominados *Publisher*, *Commander*, *QRFinder* y *Network*.

Estas tareas se ejecutan de forma prácticamente simultánea gracias a las capacidades que proporciona FreeRTOS, el cual habilita un entorno multitarea sobre microcontroladores o microcontroladores.

El uso de FreeRTOS permite, entre otras funcionalidades, asignar tareas a núcleos específicos y definir manualmente el tamaño de sus respectivos *stacks* de memoria. Esto resulta especialmente útil en casos como el hilo *QRFinder*, que requiere una mayor cantidad de memoria debido a las operaciones de captura de imagen, en comparación con otros hilos como *Publisher*, que presentan una carga menor.

En resumen, las tareas son bucles que se ejecutan de manera simultánea y que gestionan funciones cíclicas del *software*, como podrían ser por ejemplo la recepción de mensajes por puerto serie, o la monitorización continua del estado de la conexión.

Eventos

Los eventos son sucesos en el tiempo previamente definidos en el código del dispositivo, que se generan en cualquier momento durante la ejecución del programa.

Su característica principal es la capacidad de ser encolados. Para ello, se ha implementado un *buffer* junto con la lógica necesaria para gestionar una cola de eventos bajo una estructura First In First Out (FIFO), en la que los elementos se insertan y procesan uno

a uno. Esta cola está gestionada íntegramente por el hilo *Publisher*.

Gracias a este enfoque, los eventos se procesan de forma secuencial y controlada, lo que garantiza la integridad del sistema al evitar accesos concurrentes a recursos compartidos por parte de distintos objetos.

Manejadores
Esta categoría de funciones se encarga de procesar la lógica de negocio asociada a los eventos. Cada vez que un evento es desencolado, se vincula de forma unívoca con un manejador específico.

El manejador recibe los datos adjuntos al evento, los interpreta y ejecuta las acciones correspondientes dentro del sistema. Estas acciones pueden incluir la publicación de nuevos eventos en la cola, el envío de mensajes informativos, la generación de trazas de depuración o, en algunos casos, no realizar ninguna operación.

En consecuencia, los manejadores son entidades pasivas que permanecen inactivos hasta que reciben el estímulo adecuado. Esta arquitectura permite desacoplar la generación de eventos de las acciones que estos desencadenan, facilitando un comportamiento asíncrono que optimiza el uso de los recursos en sistemas embebidos con capacidades limitadas, como el presente.

Planificador
El planificador de tareas de usuario está implementado en el hilo `Scheduler`, descrito en mayor detalle en el apartado 3.3.7.

Tal como se expone en dicha sección, este componente actúa como una interfaz accesible de forma global a lo largo del código. Su principal función es permitir la generación y cancelación de tareas definidas por el programador de manera independiente a la estructura centralizada de eventos que constituye el núcleo del sistema, contribuyendo así a una mayor modularidad del *software*.

Entidades no concurrentes
Dentro de esta categoría, en nuestro caso, se hace especial énfasis en el componente identificado como *Camera* en los diagramas de la sección 3.3.

La característica distintiva de este tipo de entidades, en contraste con los casos anteriores, es que su utilización de recursos de *hardware* está limitada estrictamente a su ámbito de vida, conforme a los principios de codificación Resource Acquisition Is Initialization (RAII). Es decir, los recursos se reservan únicamente durante el tiempo en que la entidad está instanciada.

Al finalizar dicho ámbito, el objeto se destruye automáticamente y los recursos se liberan de forma controlada. El caso de *Camera* es representativo de este comportamiento, ya que dicha entidad solo se instancia puntualmente por el manejador encargado de procesar eventos relacionados con la captura de imágenes o la retransmisión de vídeo en *streaming*.

3.3.2. Depuración remota

Se ha habilitado un modo de *depuración remota* en el dispositivo. La existencia de este modo permite la conectividad entre UC y Esp32Cam a distancia, aprovechando la propia estructura del protocolo MQTT.

En la lógica de negocio de cada uno de los hilos del dispositivo, se realiza una comprobación mediante instrucciones de pre-compilación para comprobar si el valor `REMOTE_DEBUG` está definido. Si lo está, se comprueba su valor, el cual es configurado por el desarrollador en *platformio.ini* como '1' o '0'. En segundo lugar, se debe indicar en el *flag* `DREMOTE_DEV` el valor de la Media Access Control (MAC) que representa al dispositivo.

Este último valor es esencial, ya que asegura que el dispositivo se suscribirá a un *topic* único e inconfundible. Esta dirección será la que define el *topic* de suscripción de MQTT desde el que el dispositivo recibe mensajes de la contraparte. La UC se configura de la misma forma.

Una vez comprobado si el modo de depuración es remota, las comunicaciones serán redirigidas al canal correspondiente. Si el modo remoto está habilitado, se le dará salida a través del hilo *Network*, usando el protocolo *mqtt*.

Dado que la estructura de los mensajes enviados, explicada más adelante en 3.3.5, se mantiene inmutable aunque cambie el canal de comunicación, las funcionalidades tanto de la UC como del ESP32Cam seguirán respondiendo de forma remota.

3.3.3. Modos de funcionamiento

El prototipo incluye varios modos de funcionamiento que pueden opcionalmente habilitarse de forma remota. Uno de ellos será establecido como `user_preferred_mode` (modo preferido por el usuario), que, tal y como se detalla en la sección 3.3.6 (apartado de modo *Configuration*), será el que indique el modo de arranque del prototipo una vez reconfigurado.

Los modos de arranque se recogen debajo de estas líneas en la tabla 3.3.

Id	Modo de configuración	Descripción
0	Config	Reinicio en modo configuración para cambiar parámetros de red
1	Normal with streaming	Funcionamiento normal con retransmisión de vídeo
2	Normal without streaming	Funcionamiento normal sin retransmisión de vídeo

Tabla 3.3: Modos de funcionamiento previstos para el prototipo

El programa se ha preparado de tal manera que en cualquier momento durante la ejecución del mismo se puede solicitar el cambio de modo al dispositivo a tiempo real.

Además, entre las opciones de configuración, es posible habilitar un modo de depuración remota, el cual se ha desarrollado en el apartado 3.2.2, es la que incluye todas las funciones necesarias para llevar a cabo la actualización.

El dispositivo se conectará a una IP y puertos preestablecidos, en una ruta con el formato `http://IP:PORT/update`, desde donde descarga los paquetes de datos pendientes.

3.3.4. Mantenimiento OTA

Una de las implementaciones con las que cuenta el prototipo es el alojamiento de un servidor de mantenimiento remoto, al que podrá acceder un cliente con las credenciales especificadas en el apartado 2.2, bajo el epígrafe **ota.json**. Esta característica permite actualizar de forma remota el *software* del ESP32Cam.

La FSM que opera dentro del hilo *network*, y cuyo diagrama se ha incorporado en el anexo A.4, es la que lanza el servidor OTA, que se encuentra integrado dentro de la propia placa.

En el fichero *platformio.ini*, además, se encuentra el *flag* `DELEGANTOTA_USE_ASYNV_WEBSERVER`, que permite al servidor OTA ejecutarse en un hilo concurrente, evitando así que el proceso de actualización de archivos se realice de forma secuencial.

El servidor es accesible a través de una URL con el formato `IP:PUERTO/update`. Por defecto, el puerto configurado es el 100.

En la figura 3.3 se puede apreciar una imagen de la interfaz accesible en la mencionada URL para la carga de archivos a través de OTA en el ESP32Cam.

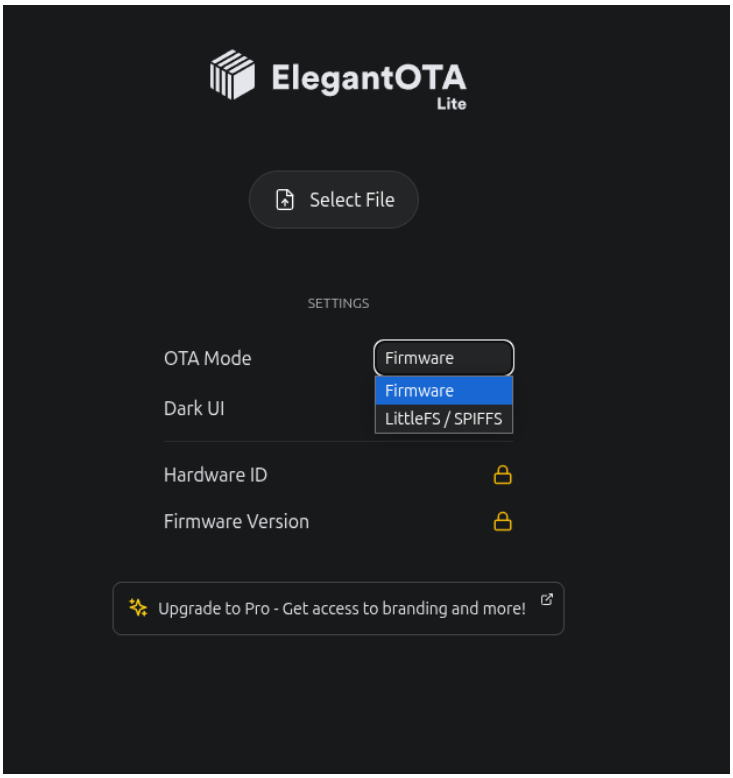


Figura 3.3: Captura de la interfaz gráfica servidor OTA

Se muestra también la posibilidad de ordenar la carga tanto del *firmware* como de los ficheros de configuración en el sistema SPIFFS, mencionados en el apartado 2.2. Esta acción se puede aplicar sobre los archivos de configuración desplegados en el sistema de ficheros implementados sobre la memoria *flash* usando tecnología SPIFFS.

También cabe la posibilidad, si la red VPN está habilitada, de acceder a este servicio desde fuera de la red local, conectando el cliente a la red VPN y accediendo a la IP asignada por la misma.

Ambas direcciones IP pueden ser consultadas por el usuario en la UI de la Unidad Central, como se muestra en las figuras 3.4 y 3.5.

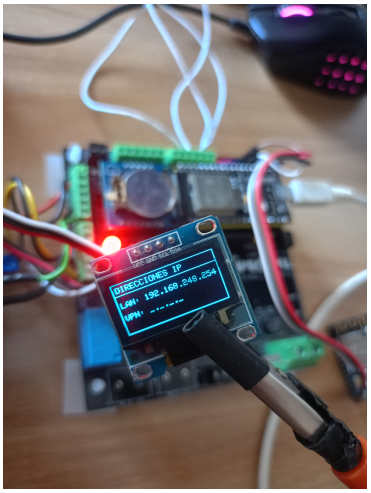


Figura 3.4: Captura de la UI de la Unidad Central mostrando al usuario las direcciones de IP asignadas a la propia UC

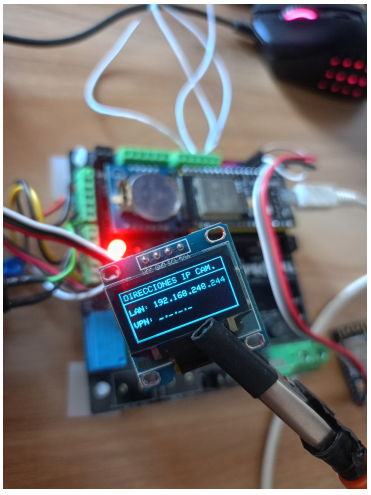


Figura 3.5: Captura de la UI de la Unidad Central mostrando al usuario las direcciones de IP asignadas al dispositivo ESP32Cam

3.3.5. Formato de mensajes

Para estandarizar las comunicaciones, todo el prototipo ha de seguir unas pautas establecidas, de forma que cualquier componente presente o futuro cuente con unos protocolos de comunicación claros a los que atenerse de forma que se asegure la escalabilidad y fiabilidad del proyecto.

La comunicación entre los distintos componentes del sistema se basa en el intercambio de mensajes de texto plano, estructurados utilizando el formato JavaScript Object Notation (JSON). De esta manera es posible permite datos de manera jerárquica y legible tanto por máquinas como por humanos.

Los mensajes JSON previstos contienen varios campos que son los que permiten al sistema clasificarlos para su correcto procesamiento. Estos campos varían según el tipo de mensaje que se está emitiendo, pero hay 2 de ellos que es necesario que estén presentes en todos los casos: campos `types` e `id`. Ambos se exponen en las correspondientes tablas 3.4 y 3.5.

Type	Descripción
0	cmd (command)
1	ack (acknowledgement)
2	inf (information)
3	tlm (telemetry)

Tabla 3.4: Tabla de types para mensajes transmitidos en formato JSON

En la tabla 3.4 se mencionan los distintos valores previstos para el campo *type* de los mensajes transmitidos entre UC y ESP32Cam. Si el valor no se encuentra entre los listados, se arrojará un error. A continuación se describen los valores listados:

- ***Command***: se encuentra reservado para aquellos mensajes que correspondan a comandos que ejecutan algún tipo de acción en el sistema. Cuando es detectado un mensaje de este tipo, será redirigido a una función `processCommand()`, que procesará el mensaje en función del contenido de su campo *id*.

- ***Acknowledgement***: este *type* se utiliza exclusivamente como señal de *handshake*, para notificar a la contraparte de que la instrucción ha sido recibida correctamente. Si el mensaje cuenta con este tipo, contará con otros campos a título informativo, que informarán sobre el proceso al que se responde, y el código de error correspondiente en caso de ser necesario.

- ***Information***: se reserva para aquellos mensajes emitidos a título informativo. Han sido usados principalmente como trazas de código para el desarrollador (p.e., al entrar en modo de funcionamiento de configuración, un mensaje de este tipo es publicado, notificando de la entrada en este modo de funcionamiento.

- ***Telemetry***: tipo reservado para la transmisión de información de telemetría relativa al estado actual del dispositivo, especialmente en cuanto a conectividad. Un mensaje de este tipo es enviado cuando el dispositivo realiza cambios en sus parámetros de

configuración de red. Su función es permitir en un futuro acceder a esta información de forma remota, para desarrollar nuevas funcionalidades para el dispositivo.

Sólo los mensajes de tipo *cmd* son procesados tras la recepción de los mismos, en función del contenido del campo *id*, requerido para la clasificación correcta del mensaje. Si este campo no es encontrado, el mensaje se descarta como uno inválido, de lo cual se informa mediante la publicación de un mensaje de error.

En la tabla 3.5 se describen los distintos valores que puede adoptar el campo *id* de los mensajes con valor `type:cmd`.

Command id	Descripción
0	Enter in configuration mode
1	Configure network
2	Telemetry request
3	Photo request

Tabla 3.5: Tabla de command ID para mensajes JSON con `type:cmd`

- ***Enter in configuration mode***: este comando siempre es emitido desde la UC hacia el módulo ESP32Cam, ordenando la entrada en modo de configuración al mismo. Como se observa en el anexo 3.17, el modo de configuración inicializa los procesos necesarios para configurar los nuevos parámetros de red a través de un código QR leído por la cámara.
- ***Configure network***: *configure network* es un tipo de comando usado exclusivamente para notificar a la UC de la necesidad de reconfigurar sus parámetros de red, publicado desde el hilo *QRFinder* del ESP32Cam, y que contiene la información necesaria para ello. Esta información es previamente capturada por la cámara a través del contenido en el código QR presentado (que es la SSID de red y contraseña).
- ***Telemetry request***: comando usado para activar el manejador correspondiente, cuya única función es devolver un mensaje de telemetría (tipo `tlm`) como los descritos en el apartado 3.3.5. Permite recabar datos de telemetría a demanda, y no únicamente durante la inicialización, donde la publicación de mensajes de telemetría se encuentra ya codificada por defecto.
- ***Photo request***: el comando *Photo request* es el que activa el modo de captura de imagen de la cámara en el módulo ESP32Cam a demanda del usuario o la UC. Cuando se recibe este comando, automáticamente se publica en la cola de eventos un evento tipo `take_photo`, y será el manejador correspondiente el que ordena la captura de imagen, su reenvío y posterior liberación de la memoria.

Mensajes de telemetría

En esta sección se explican en detalle los mensajes de telemetría emitidos por el módulo ESP32Cam. Se les dedica un apartado especial, dada la variedad de datos que pueden ser capturados y las implicaciones de los mismos en el funcionamiento del prototipo.

Los mensajes de telemetría se han dividido en distintas categorías según el contenido de los mismos, a fin de facilitar la visualización de los mismos en las tareas de depuración. Estas categorías se han esquematizado en la tabla 3.6.

Dado que todos ellos son mensajes de telemetría, se obviará el valor del campo `type`, que en todos los casos debe estar presente con un valor `type:tlm`.

Tipo de telemetría	Campos del mensaje	Tabla descriptiva
Protocolos de seguridad activos	ssl, https, vpn	Tabla 3.7
Modo de operación	camera	Tabla 3.8
Configuración de mantenimiento	ota_enabled	Tabla 3.9
Estado de conexión	state, ssid	Tabla 3.11

Tabla 3.6: Categorías de mensajes JSON con `type:tlm`

A continuación, se describe detalladamente la composición de los mensajes pertenecientes a cada una de las categorías en los siguientes epígrafes.

Telemetría de configuración de seguridad

Según se ha reflejado en la tabla 3.7, hay distintos campos que recogen información valiosa sobre el estado del dispositivo.

Campo	Descripción	Valores posibles
sec_settings: ssl	Certificado de seguridad SSL en uso	true | false
sec_settings: https	Certificado de seguridad HTTPS en uso	true | false
sec_settings: vpn	Activar VPN	true | false

Tabla 3.7: Campos que conforman los mensajes de telemetría relativos a la configuración de seguridad del dispositivo.

En este caso el mensaje de telemetría informa a la UC de el estado de los valores en el momento de la llegada del mensaje de telemetría. Estos valores son almacenados y procesados por la lógica interna de la UC.

Ejemplo real de mensaje de telemetría de configuración de seguridad publicado.

Telemetría de configuración de modo de operación

Los mensajes de telemetría relativos al modo de operación permiten tanto a la UC como al desarrollador saber en qué modo de operación se encuentra el dispositivo ESP32Cam. Los modos de operación previstos han sido mencionados previamente en la tabla 3.3.

Los campos disponibles para esta categoría de mensajes de telemetría se recogen en la tabla 3.8. Este tipo de mensaje de telemetría publica la información relativa al modo de operación del módulo ESP32Cam, que es recopilada por la UC y posteriormente procesada en su lógica interna. Los posibles valores recogidos en la tabla se correponden con los expuestos en la tabla 3.3.

Campo	Descripción	Posibles valores
op_mode: camera	Modo de operación actual	0 | 1 | 2

Tabla 3.8: Campos que conforman los mensajes de telemetría relativos al modo de operación del dispositivo.

Telemetría de configuración del modo de mantenimiento

El modo de mantenimiento indica si el dispositivo tiene habilitado el servidor OTA, el cual es necesario para recibir actualizaciones de forma remota. Es fundamental conocer el estado de esta funcionalidad, ya que, a través de las actualizaciones OTA, es posible modificar el *firmware* del ESP32Cam sin intervención física, incluyendo la actualización de los certificados de seguridad mediante este mismo mecanismo, el cual se describe en mayor profundidad en el apartado 3.3.4.

Los posibles valores que puede adoptar el campo `ota_enabled` se describen en la tabla 3.9.

Campo	Descripción	Posibles valores
maintenance: ota_enabled	Servidor OTA activado	true | false

Tabla 3.9: Campos que conforman los mensajes de telemetría relativos a la configuración de mantenimiento.

Una vez se publica este mensaje de telemetría, la UC recopila la información para procesarla en su lógica interna.

Telemetría sobre estado de conexión a la red

La telemetría sobre el estado de conexión a la red es la que mantiene informada a la UC sobre el estado de la conexión.

Para una correcta depuración se ha creado un enumerado específico que contiene los distintos estados de la FSM que monitoriza la conexión de red. Se han recogido los mismos en la tabla 3.10

Valor	Significado	Descripción
0	waiting_for_config	Conexión aún no establecida
1	disconnected	Desconectado de la red
2	connecting_wifi	En proceso de conexión
3	wifi_connected	Handshake establecido
4	wifi_timeout	Tiempo de espera de conexión a red rebasado
5	mqtt_timeout	Tiempo de espera de conexión a *broker* de MQTT rebasado
6	error	Error en la conexión
7	connecting_mqtt	En proceso de conexión a broker de MQTT
8	mqtt_connected	Conectado a broker MQTT
9	subscribing_topics	En proceso de suscripción a topics de MQTT preconfigurados
10	fully_connected	Conexión finalizada y estable

Tabla 3.10: Tabla de valores de estado de conexión para telemetría del estado de red

Los mencionados valores forman parte del mensaje de telemetría emitido por el ESP32Cam. Los campos que conforman este mensaje se muestran en la tabla 3.11.

Campo	Descripción	Posibles valores
net_state: ssid	Identificador de la red actual	<SSID >
net_state: state	Estado actual de la conexión	0 - 10

Tabla 3.11: Campos que conforman los mensajes de telemetría relativos al estado de conexión a la red del dispositivo.

Los posibles valores que aparecen en la tabla 3.11 asignados al campo `net_state:state` se corresponden con los recogidos en la tabla 3.10 situada más arriba. El mensaje contendrá el valor numérico dentro del campo *state*, que será procesado por la lógica interna de la UC.

A continuación, en la figura 3.6 se muestra una captura de como se deben visualizar en el monitor serie cada uno de los distintos mensajes de telemetría mencionados.

```
{"sec_settings":{"https":true,"ssl":false,"vpn":false},"type":3}
{"camera":{"mode":1},"type":3}
{"maintenance":{"ota_enabled":true},"type":3}
{"net_state":{"ssid":"TTBRed","state":0},"type":3}
```

Figura 3.6: Captura del monitor serie de VSCode que muestra la salida generada por el ESP32Cam tras el envío de los distintos datos de telemetría

Mensajes con acuse de recibo

El acuse de recibo (ACK) también cuentan con un formato estándar dentro del protocolo. En la figura 3.7 se muestra como la composición de un código de error incluye los campos `type, id, status_code` y `what`.

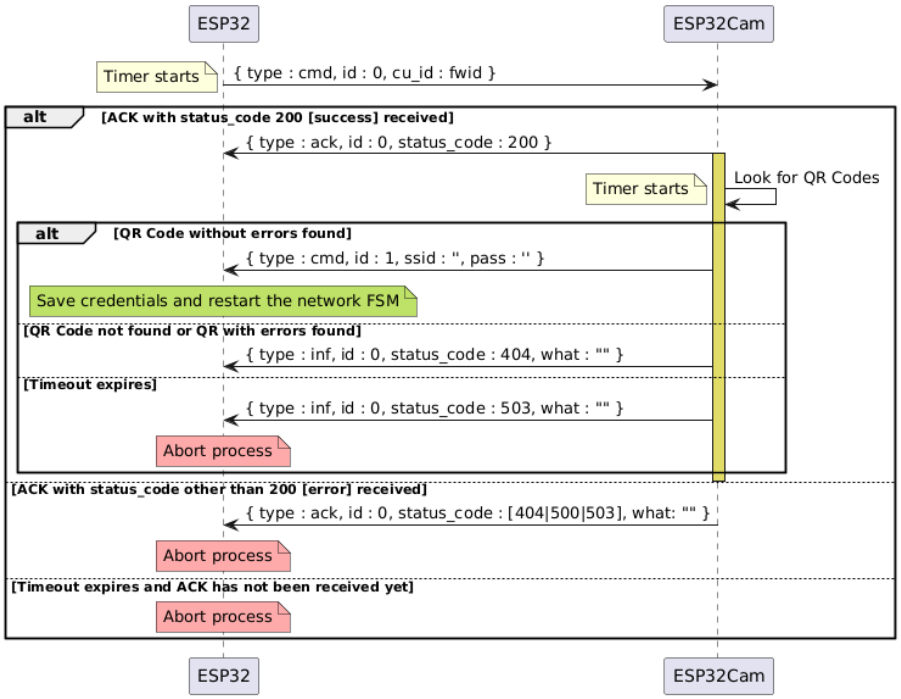


Figura 3.7: Diagrama de secuencia para reinicio de la red. Ejemplo de código de error.

Para mantener la coherencia de las comunicaciones dentro de todo el código, se ha optado por usar el estándar de códigos de estados de respuesta Hyper-Text Transfer Protocol (HTTP) en el campo `status_code`.

El campo *status_ code* deberá contener un valor de los que figuran en la tabla 3.12.

Response code	Descripción
200	Success
404	Not found
503	Service unavailable
500	Internal Server Error

Tabla 3.12: Códigos de respuesta según el estándar HTTP

Los campos que se incluyen en la estructura de este mensaje son los siguientes:

- `type`: su valor será el del tipo de comando al que se responde, y se corresponde con alguno de los indicados en la tabla 3.4.
- `id`: contiene la `id` del mensaje al que se acusa recibo. Por tanto, su valor se corresponde con alguno de los de la tabla 3.5.
- `status_code`: dependiendo del resultado de la transmisión, se rellena con un valor estado u otro. El contenido será acorde a alguno de los valores consignados en la tabla 3.12.
- `what` - en caso de devolverse algún valor en `status_code` distinto a 200 (*success*), este campo contendrá una descripción breve sobre el motivo del mismo. su contenido depende de la programación en el lado del cliente.

3.3.6. Diagramas de secuencia

A continuación se recogen y explican los diagramas que definen la arquitectura de software del dispositivo. Las figuras incluidas a continuación representan el flujo de eventos que han sido implementados en la lógica interna.

Los diagramas de secuencia son una forma de representar el flujo de comunicaciones en *software* multi-hilo, que es el caso de este proyecto.

Dado que hay varios procesos corriendo en paralelo, es necesaria una esquematización adecuada de los procesos que se ejecutan en el prototipo. Para ello, se usarán diagramas de secuencia.

Usando como ejemplo el diagrama mostrado en la figura 3.8 se procede a explicar cómo se han de interpretar este tipo de diagramas.

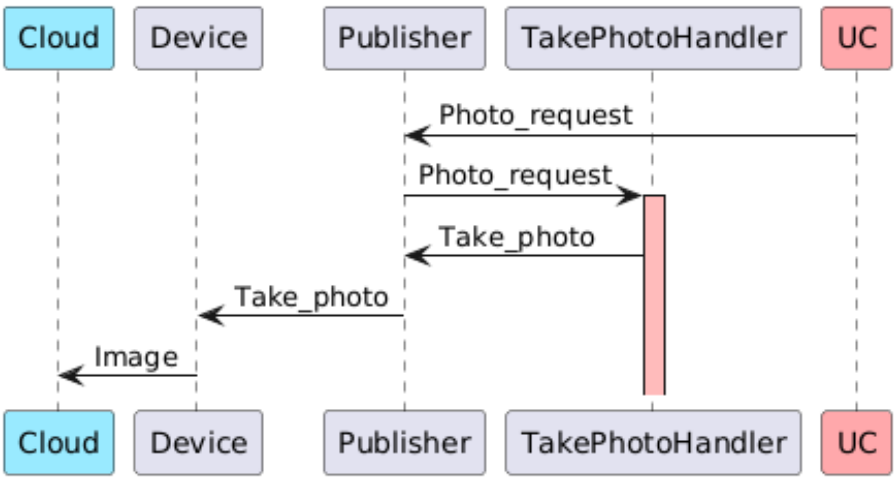


Figura 3.8: Ejemplo de diagrama de secuencia

En primer lugar, hay que identificar en el diagrama los distintos procesos que son ejecutados simultáneamente. Estos son representados por cada línea discontinua vertical. Los hilos UC y *Cloud*, que se encuentran coloreados, se representan como hilos para abstraer los procesos ejecutados simultáneamente al resto de hilos.

El orden cronológico de lectura es de arriba hacia abajo, y las flechas representan acciones ejecutadas en los distintos hilos que se comunican con otro hilo en el sentido de la flecha.

En el ámbito de este proyecto, la acción representada va a ser o bien la publicación de un mensaje en formato JSON como los explicados en 3.3.5, o bien el nombre de un evento que ha sido publicado.

En otros casos, la descripción sobre las flechas tiene como finalidad dar una denominación reconocible a la acción que está ocurriendo. En la figura de ejemplo, la UC envía un evento al hilo *publisher* del tipo `Photo_request`.

En algún momento posterior, como reacción a ese estímulo, el hilo *Publisher* desencola el evento `Photo_request`, y durante el proceso realiza una llamada al manejador *TakePhotoHandler*.

La columna coloreada que encontramos sobre la línea de un hilo representa el punto de inicialización del hilo correspondiente. Cuando no figura este elemento, se considera que el hilo está completamente inicializado y en ejecución desde algún momento cronológico anterior al proceso representado.

En este ejemplo, una vez el hilo *TakePhotoHandler* ya ha sido inicializado, publica el evento *take_photo*. La lógica interna del dispositivo será la que toma las acciones necesarias para terminar devolviendo la imagen hacia la nube, siguiendo su lógica interna, que no es relevante para el proceso representado en el diagrama.

Este ejemplo, con ligeras modificaciones para facilitar su comprensión, representa un proceso real que se describe en la sección 3.3.6.

Como última anotación, cabe mencionar que en los diagramas de secuencia también es posible incluir bloques que representan bifurcaciones condicionales, como las estructuras *if-else*.

En la figura 3.9 se muestra un ejemplo de cómo se representan este tipo de operaciones.

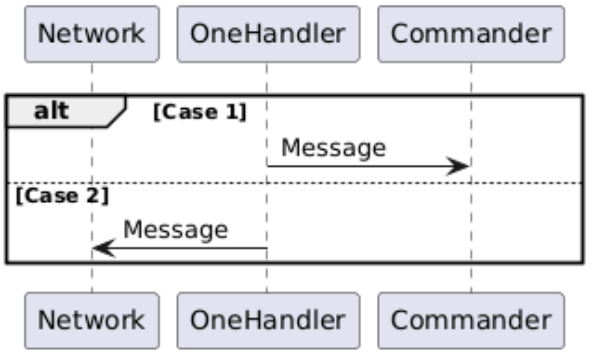


Figura 3.9: Ejemplo de bloque condicional en diagramas de secuencia

La secuencia se divide en dos bloques, identificables por la etiqueta **alt** (o *alternative*) que aparece en la esquina superior izquierda. Los procesos enmarcados dentro de cada uno de los dos bloques no ocurren ni secuencial ni simultáneamente. En cada ejecución del proceso, se atenderá a las condiciones indicadas en el espacio correspondiente, y se

ejecutará sólo uno de los cursos de acción disponibles. Este esquema se utiliza también para representar bucles *if-else* anidados.

Lanzamiento de modo configuración

Al activarse el modo de configuración, el dispositivo inicia una secuencia específica que lo pone en estado de búsqueda de un código QR. Este código contiene los nuevos parámetros necesarios para la conexión a la red.

La figura 3.10 ilustra en detalle este proceso, que abarca desde la solicitud de entrada en modo configuración hasta el reinicio automático del dispositivo, momento en el que este aplica los nuevos parámetros e intenta establecer la conexión con la red correspondiente.

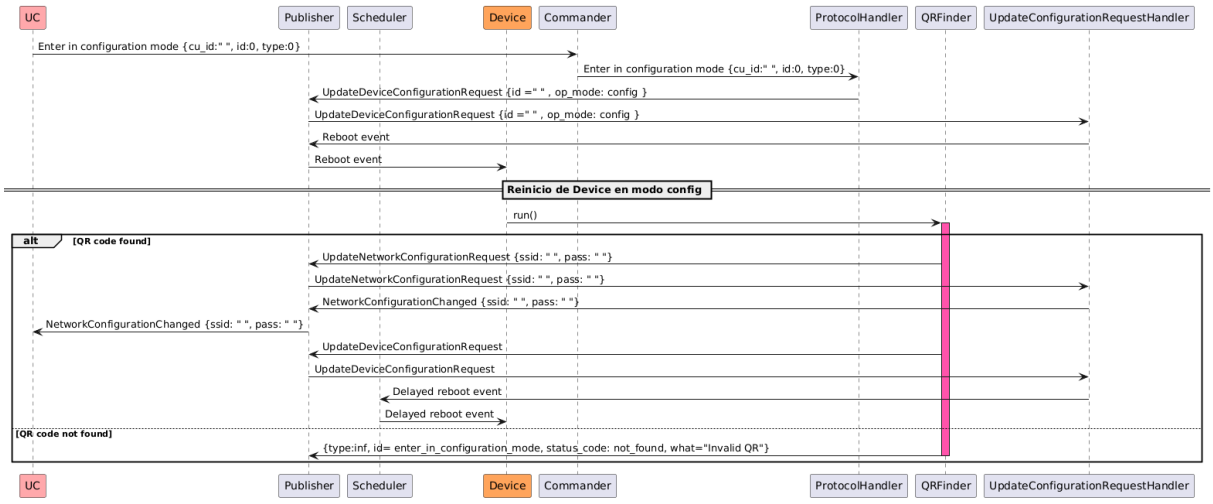


Figura 3.10: Diagrama de secuencia para reiniciar en modo de configuración (también en anexo A.11)

El diagrama comienza en el punto en el que la UC envía la solicitud para entrada en modo de configuración al módulo ESP32Cam. El mensaje enviado contiene los campos *type* e *id*, que son de enorme importancia, como ya se hizo mención en el apartado 3.3.5. Los mencionados campos permiten llamar al manejador correspondiente, denominado `ProtocolHandler`.

El manejador procesa el comando recibido, y publica el correspondiente evento `UpdateDeviceConfigurationRequest`, cuya principal función es la de almacenar los datos necesarios para el posterior arranque en la memoria EEPROM, especialmente el `user_preferred_mode` y el modo de operación `config`. De esto se encargará el manejador del evento, como se puede observar. Por último, se ordena el reinicio del Esp32Cam.

Cuando el dispositivo vuelva a iniciarse, habrán ocurrido dos procesos importantes: en primer lugar, el almacenamiento ya mencionado de el modo `user_preferred_mode`. Así, el prototipo podrá volver a este modo tras el proceso de configuración. En segundo lugar, el modo de inicio ahora es `config`. Cuando este modo es lanzado, el proceso de arranque del dispositivo también varía, como se muestra en la figura 3.17.

Tras iniciar el dispositivo en el nuevo modo, que se ha mencionado en el apartado 3.3.3, el hilo `QRFinder` es puesto en marcha, iniciando la captura del código QR.

A continuación se distinguen dos casos:

1. **Código encontrado:** en este caso, el hilo `QRFinder` procesa la información contenida en el código y publica el evento `UpdateNetworkConfiguration`, que será gestionado por el manejador `UpdateConfigurationRequestHandler`.

   A continuación, dicho manejador emite un nuevo evento destinado a la UC, indicando que se ha establecido una nueva configuración de red. La lógica interna de esta unidad se encarga entonces de ejecutar los procesos necesarios tras recibir dicha notificación.

   Posteriormente, `QRFinder` lanza un evento final, `UpdatedeviceConfiguration`, cuyo propósito es actualizar el valor de inicio almacenado correspondiente al parámetro `user_preferred_mode`. Este valor será el consultado por el código en la fase de inicio tras llevar a cabo la configuración.

   Finalmente, el manejador solicita el reinicio inmediato del sistema, con el objetivo de introducir una breve demora que asegure el final satisfactorio de cualquier envío de información a la UC antes de reiniciar el dispositivo.

2. **Código no encontrado:** si el código QR no se detecta dentro del tiempo establecido, el propio hilo `QRFinder` se cerrará automáticamente tras emitir un mensaje de tipo `inf` con el texto *Invalid QR*.

Emisión de mensajes de telemetría tras la inicialización

En la figura 3.11 se muestra la secuencia de envío de mensajes de telemetría cuando se termina de inicializar el dispositivo.

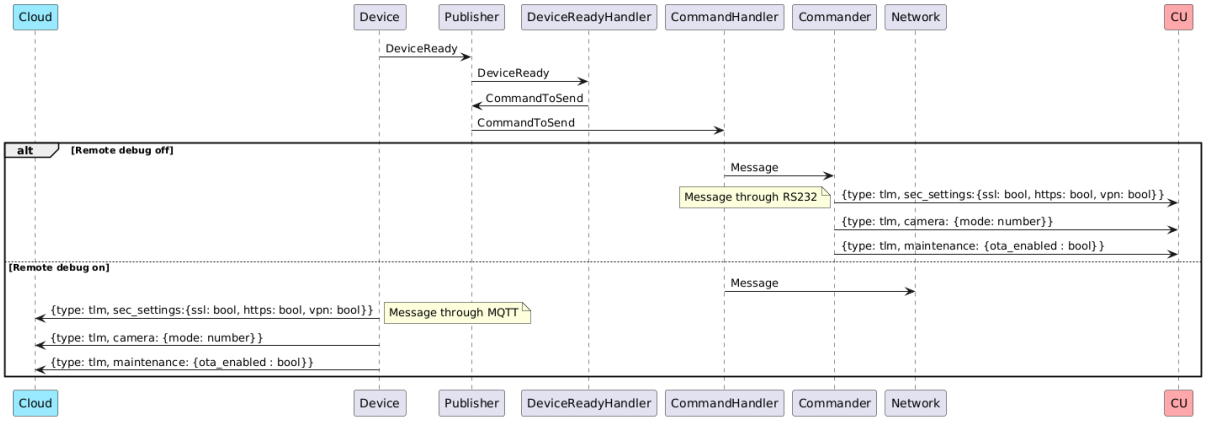


Figura 3.11: Diagrama de secuencia para el envío de telemetría cuando se produce *DeviceReady* (también en anexo A.6)

En primer lugar, el dispositivo (representado como el hilo *Device*), publica el evento `DeviceReady`, indicando que la inicialización ha terminado satisfactoriamente. Este evento se publica sólo cuando todos los hilos están ejecutándose de la forma esperada. El encargado de encolar el evento y publicarlo después es el hilo *Publisher*. Una vez esto ha ocurrido, se realiza la llamada al manejador correspondiente.

El manejador es el encargado de estructurar los mensajes correspondientes y publicarlos en el formato estipulado en el protocolo. La estructura y contenidos de los mensajes de telemetría se detalla en el apartado 3.3.5.

Dependiendo del modo de depuración activo (remoto o no), el mensaje de telemetría será finalmente emitido a través de protocolo MQTT (detallado en el apartado 3.2.3) o por serie RS232.

El diagrama representa exclusivamente la emisión de mensajes de telemetría tras la inicialización. En el siguiente apartado 3.3.6 se explica la segunda posibilidad para que se produzca el envío de telemetría, que es el envío a demanda.

Se han creado estas dos opciones para evitar que, por algún desfase en la inicialización de los dispositivos y en sus hilos *Commander*, detallados en el apartado 3.3.7, la información de telemetría se pierda.

Emisión de mensajes de telemetría a demanda

Como se ha explicado en el apartado anterior, la notificación de telemetría también se puede producir a demanda de la UC. En el siguiente diagrama 3.12 se muestra cómo ocurre este proceso.

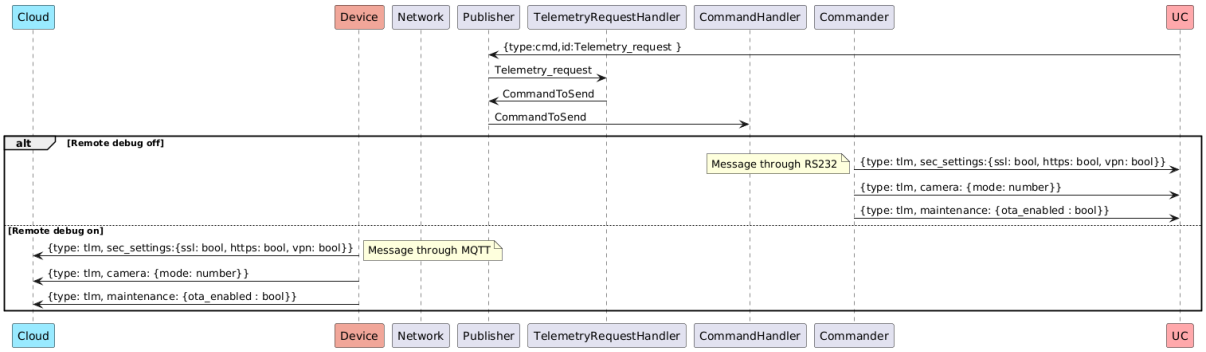


Figura 3.12: Diagrama de secuencia que muestra la secuencia del proceso que envía una notificación con información de telemetría a demanda (también en el anexo A.7)

En primer lugar, se puede comprobar que el estímulo para que se produzca la publicación del mensaje no proviene del dispositivo, como en el epígrafe anterior representaba en el diagrama 3.11.

En este caso, el evento se genera desde la propia UC, que es la que en algún momento del tiempo publicará la petición.

En cuanto al flujo del proceso, es la UC la que publica en un momento determinado el mensaje con `id = telemetry_request`. Una vez se desencola el evento, el *handler* correspondiente publica el evento que dispara el envío de los mensajes de telemetría por el canal correspondiente.

Telemetría de estado de la conexión

En la figura 3.13 se puede observar cómo se produce la notificación de telemetría cuando se producen cambios en el estado de la red.

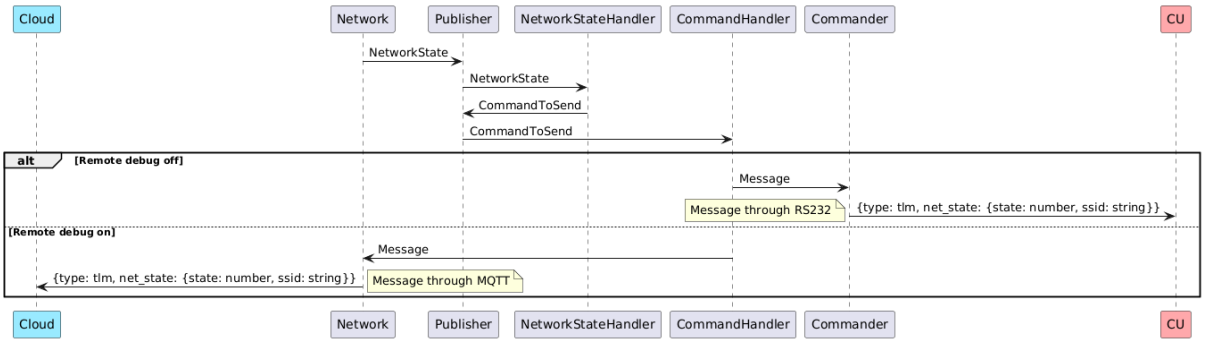


Figura 3.13: Diagrama de secuencia de notificaciones de telemetría sobre el estado de la conexión (también en anexo A.8)

El proceso es similar a los anteriores. El hilo *Network* emite un evento *NetworkState* que es encolado por el hilo *Publisher*. Una vez se desencola, es llamado el manejador correspondiente, encargado de componer el mensaje JSON de tipo *tlm* que será finalmente emitido.

En el caso de este tipo de telemetría, cabe destacar que su finalidad es informar del estado de la red, para lo que *NetworkStateHandler* recurre a un enumerado especialmente diseñado para este caso, y que se puede consultar en la tabla 3.10.

Notificación de cambio de IP Evento *IPChanged*

La figura 3.14 muestra el diagrama de secuencia que permite notificar el mensaje de telemetría cuando una nueva dirección Internet Protocol (IP) ha sido asignada al dispositivo al detectarse un cambio de red y la consecuente asignación de direcciones.

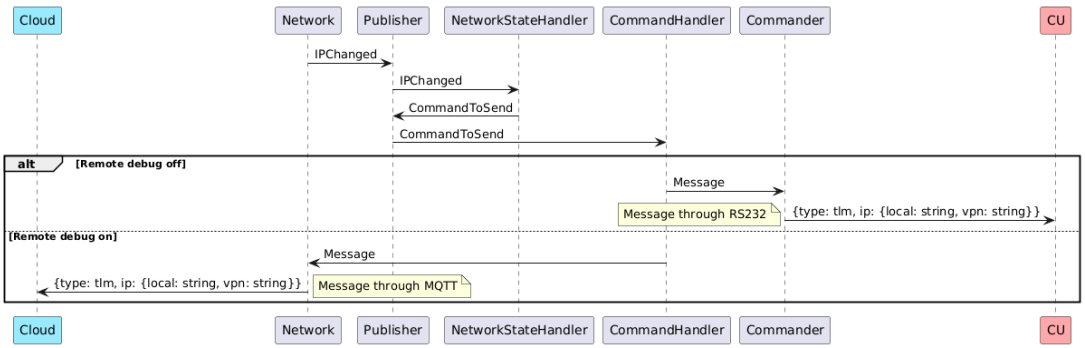


Figura 3.14: Diagrama de secuencia de telemetría de datos de conexión (también en A.9)

En primer lugar, el hilo *Network* notificará con un evento `IPchanged` que será encolado por el hilo *Publisher*. Cuando se gestione el evento y sea desencolado, se llamará al manejador `IPChangedHandler`. Este es el encargado de componer el mensaje en formato JSON con `type: tlm (telemetry)` que contiene los campos adecuados para publicar la información.

Como se ha visto en casos anteriores, el mensaje compuesto se notificará a través del hilo *Commander* o *Network* según el modo de depuración seleccionado.

Telemetría OTA

Dado que el dispositivo cuenta con capacidad de actualización Over-The-Air (OTA), se ha implementado la funcionalidad necesaria para monitorizar el estado de futuras actualizaciones. Esto posibilita actuar sobre distintos dispositivos simultáneamente, ya que serán identificados de forma única a través de su `device_id`, dotando de escalabilidad al proyecto.

En la figura 3.15 se muestra el proceso de solicitud de telemetría. El objetivo es publicar información en tiempo real y de forma remota sobre el estado de la actualización del dispositivo. Como la UC se encuentra suscrita al *topic* correspondiente al dispositivo actualizado, es posible hacer un seguimiento individualizado de la misma en cada dispositivo.

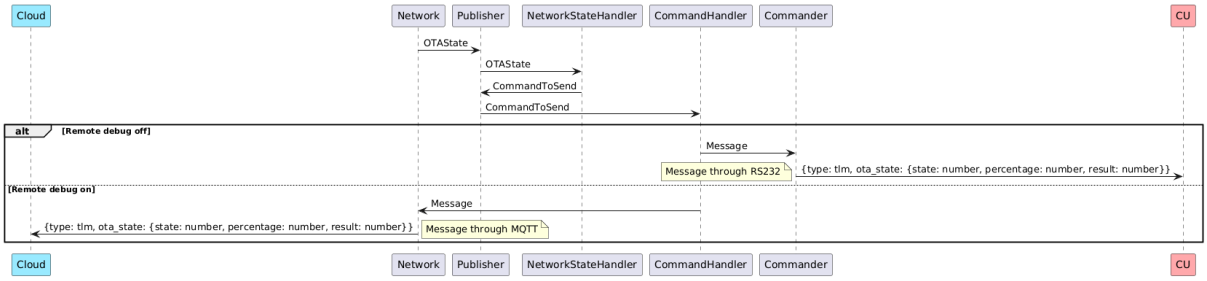


Figura 3.15: Diagrama de secuencia para solicitud de telemetría OTA (también en anexo A.10

El hilo *Network* emite un evento `OTAState`. Este evento se caracteriza por ser publicado desde *Network* ya con un campo `state` que contiene un valor asignado por el propio hilo *Network*.

Este valor puede ser:

- Ready
- Start
- Progress
- End

El resto de valores accesorios dependen del estado en el que *Network* configure el evento. Los campos mostrados en la figura 3.15 *percentage* y *result* son también configurados por el mencionado hilo. Si el estado es *progress* o en progreso, se realizará el cálculo del porcentaje. Si es *end* o finalizado, *result* será configurado como *success* o *failed* según el resultado de la carga.

Al desencolar el evento, el manejador `OTAStateHandler` procesa todos estos valores, componiendo un mensaje en formato JSON según la estructura ya comentada, que informará de toda esta información a través del canal correspondiente al modo de depuración configurado, como en los casos anteriores.

Comando para captura de imagen

En el diagrama 3.16 se explica de forma gráfica cual es el proceso por el cual se resuelve la solicitud de captura de imagen. En primer lugar, la *uc* envía un mensaje en formato *json*, con los campos `type = 0` (tipo *Command*) e `id = 3` (Id del comando `photo_request`).

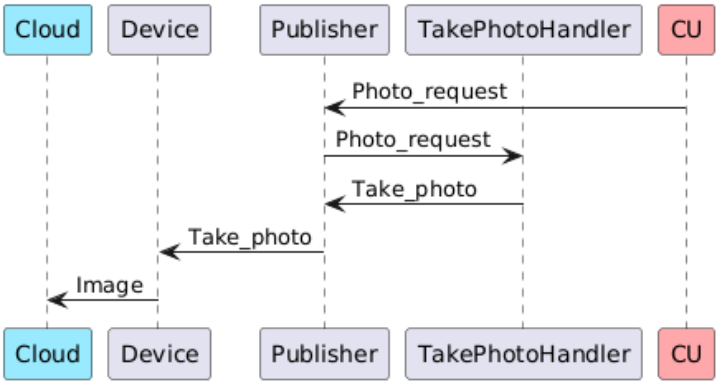


Figura 3.16: Secuencia para la captura de una imagen desde el módulo ESP32Cam

Se puede observar que el diagrama es más sencillo que los anteriores, debido a que una parte del proceso necesario para otras acciones no son necesarias en este caso.

Una vez se publica, el mensaje es interpretado directamente por el protocolo implementado. Al detectar una `id = 3`, directamente es llamado el manejador `TakePhotoHandler`, encargado de resolver la solicitud. El manejador entonces publica un evento `take_photo`, que finalmente es resuelto activando la lógica a bajo nivel que permite realizar la captura y retransmitirla hacia la nube usando la librería *esp_ cam.h*

Secuencia de arranque

En la figura 3.17 se muestra la secuencia ordenada de arranque del dispositivo ESP32Cam.

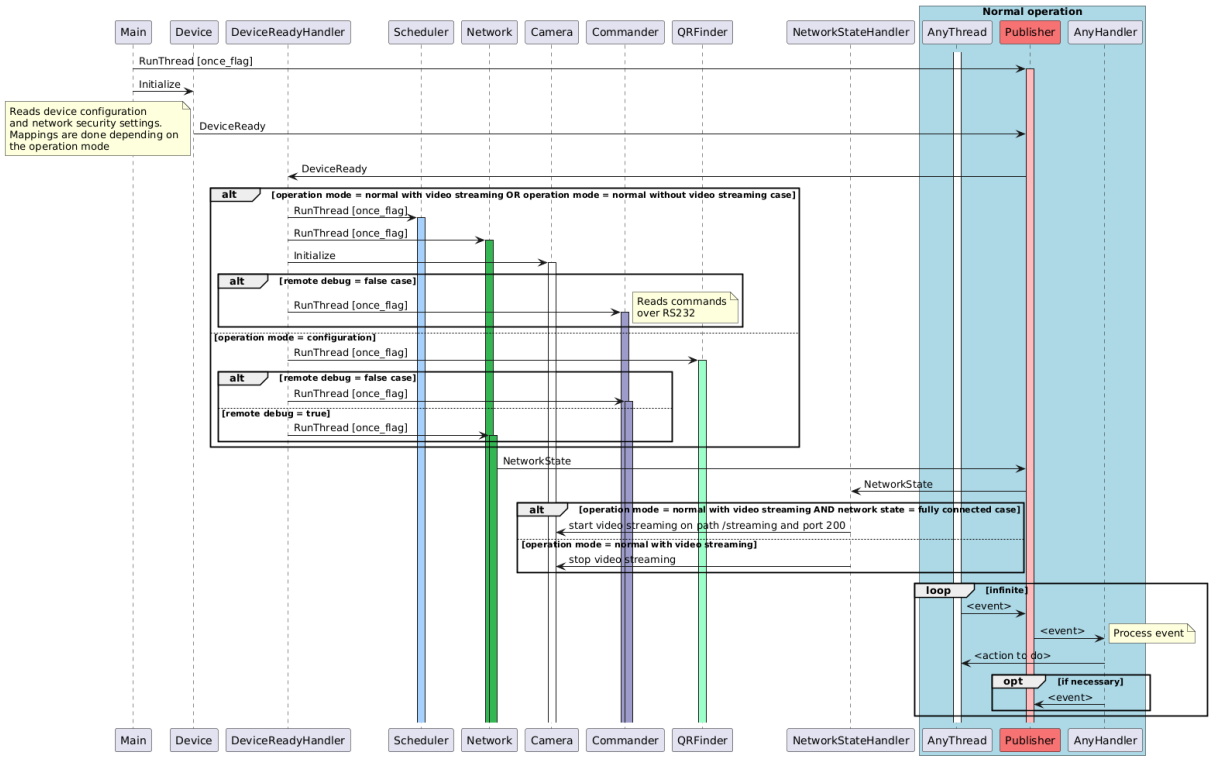


Figura 3.17: Diagrama de puesta en marcha del dispositivo (también en anexo A.5)

Si se sigue en un orden cronológico con las pautas referidas en el apartado 3.3.6, la secuencia es la siguiente:

- **Main →Publisher**: el punto de entrada del programa, situado en la función `main()`, inicializa el hilo **Publisher**. El flag *run_ once* permite asegurar que el hilo sólo es lanzado una vez por cada inicialización.
- **Main →Device**: sin esperar una respuesta por parte del hilo **Publisher**, que está en este momento llevando a cabo su inicialización, **main** lanza la inicialización de **Device. Device**, representado como un hilo más, en este punto comienza a cargar todos los parámetros de configuración del dispositivo (Tales como GPIO, manejadores, protocolos, certificados de seguridad en la EEPROM...). Como se ha anotado en el diagrama, la inicialización de estos valores dependerá del modo de operación cargado en el hardware previamente al arranque.
- **Device →Publisher**: una vez el dispositivo se ha inicializado por completo, envía el evento `DeviceReady` de al hilo **Publisher**. El mismo entonces desencolará el evento, para activar el manejador correspondiente. En este caso, es **DeviceReadyHandler**, que en última instancia actúa sobre el resto del sistema durante la inicialización.

- **DeviceReadyHandler →(Resto de hilos)**: en este punto, dependiendo del modo de operación configurado, se lanzarán unos hilos u otros. Este modo de operación actuará de distinta forma según si el dispositivo se ha reiniciado tras solicitar la configuración para conectarse a un nuevo SSID, o, en caso contrario, el último modo de funcionamiento almacenado en la EEPROM del dispositivo.

  a) **Modo *normal with streaming* O *normal without streaming***: en este caso, el modo de operación es *normal*. Cuando se da este caso, los hilos iniciados son, en orden cronológico:

     a) Scheduler

     b) Network

     c) Camera

     Se puede observar, dentro de un bloque condicional anidado, que la inicialización de **Commander** o **Network** depende de si el modo de depuración seleccionado es remoto o no. En el diagrama sólo se ha representado la opción ***[remote debug = false case]***, dejando por defecto el caso en el que el valor configurado es *true*.

     Si es seleccionado el modo de depuración remota, los paquetes de datos emitidos serán transmitidos a través del hilo **Network** hacia la nube usando protocolo MQTT.

     En caso contrario, el hilo **Commander** se encargará de enviar los mensajes a través de la conexión serie establecida en la UART.

  b) ***Modo Configuration***: al iniciar el dispositivo en este modo, se ejecuta directamente el hilo **QRFinder**, responsable de escanear el código QR mediante la cámara para obtener el nuevo SSID y la contraseña que se asignarán al dispositivo.

     Por razones de optimización, el dispositivo no lanzará ningún otro hilo en este modo de funcionamiento. Una vez almacenados los nuevos valores, el dispositivo volverá a iniciarse en el modo previo al de *Configuración*. Esto es posible porque los valores habrán sido almacenados previamente en la memoria EEPROM del dispositivo, como se ha mencionado anteriormente en el epígrafe 3.1.3.

     Esta funcionalidad es en esencia la que dota al dispositivo de capacidad *plug-and-play*. El usuario final sólo necesitará seleccionar el modo de configuración, y mostrar el código con las credenciales de conexión necesarias para que el sistema inicie la configuración automáticamente sin más interacción por su parte.

Tras resolver estos procesos, el hilo *Network* publica un evento tipo *NetworkState* para mantener a la otra unidad informada del estado de conexión a la red. Esta información será utilizada inmediatamente después, para gestionar el lanzamiento del modo *streaming*

a) ***Modo normal with video streaming AND network_ state = fully connected:*** dado este caso, el dispositivo se ha configurado para realizar la retransmisión en vídeo a través de la cámara.

Se puede observar que cronológicamente este es el último bloque condicional resuelto. Si la red se encuentra en estado `fully_connected`, al que se hizo referencia anteriormente en la tabla 3.10, y el modo configurado es el de *streaming* de vídeo, se lanzará el evento correspondiente para que la lógica interna del hilo *Camera* inicie la retransmisión a través del puerto 200 en la dirección IP correspondiente. La misma ya es conocida por la UC gracias a los mensajes de telemetría que se producen justo después de que el dispositivo publique el evento `deviceready`, y cuya generación se ha expuesto previamente en el apartado 3.3.6.

3.3.7. Entidades de software

Los hilos que componen el sistema cumplen cada uno una funcionalidad distinta dentro del dispositivo, interactuando entre ellos para ofrecer la funcionalidad buscada. En el caso de los hilos denominados *Scheduler* y *Camera*, no es posible clasificarlos técnicamente como hilos. Son entidades accesibles en todo momento. En este apartado se define todo el conjunto como "entidades".

A continuación, se explica en detalle el funcionamiento de cada uno de ellas.

Scheduler

Scheduler, como se ha mencionado, es una entidad accesible desde cualquier parte del código, motivo por el que se representa en el código como un hilo. Se trata de un planificador de tareas de usuario.

Esta entidad aporta una serie de funciones útiles para la escalabilidad del proyecto. Permite al desarrollador generar tareas de forma autónoma, que se comportarán como una pequeña máquina de estados dentro del código.

La entidad permite interactuar con estas tareas de diversas maneras, permitiendo ejecutarlas de forma diferida, encolarlas, y cancelarlas a discreción del desarrollador, sin necesidad de introducirlas en el resto de hilos, y pudiendo las mismas interactuar con el resto del sistema.

Las funciones con las que se ha dotado a este gestor de tareas permiten la creación de nuevas tareas, así como su cancelación de distintas formas.

Network

El hilo *Network* contiene una máquina de estados o Finite State Machine (FSM) que se encarga de mantener en funcionamiento la conexión a la red, gestionando en segundo plano la casuística necesaria para evitar una desconexión de la red no intencionada.

En la figura 3.18 se puede consultar el diagrama que representa la FSM que se aloja dentro de la tarea *Network*.

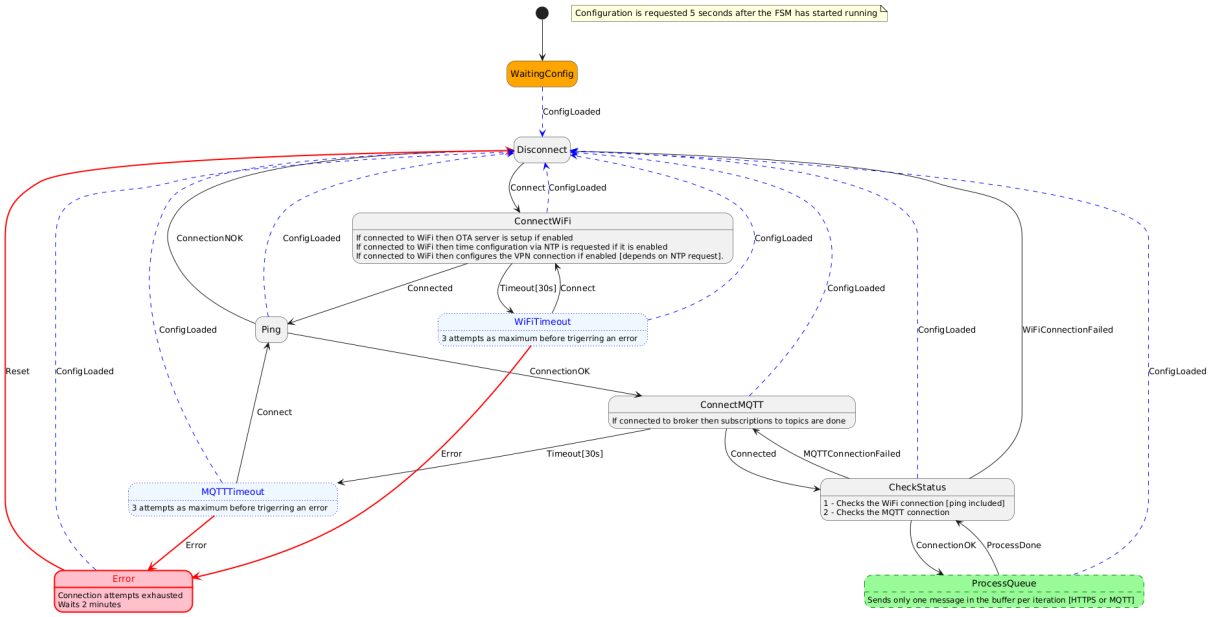


Figura 3.18: Máquina de estados residente en el hilo *Network* (también en anexo A.4)

Durante la ejecución del hilo, primero se verifica la conexión a la red *WiFi*, previamente a lanzar el resto de comprobaciones. Una vez la conexión se ha establecido, se inicia el protocolo MQTT. Cuando se ha iniciado la conexión al *broker* MQTT, se lleva a cabo la suscripción a los distintos *topics*. Mientras la red se encuentra en el estado *CheckStatus*, y la conexión sigue abierta, se procesa la cola de eventos, comunicando un evento por iteración.

Camera

Camera Tampoco se puede considerar un hilo. Se trata de una clase, que es instanciada sólo por el manejador *DeviceReadyHandler* durante la inicialización del dispositivo.

La clase *Camera* contiene los métodos necesarios para llevar a cabo las capturas de imágenes, liberar la memoria tras realizarlas, o llear a cabo la transmisión de vídeo por *streaming*.

Commander

Commander es el hilo responsable de gestionar todas las funcionalidades relacionadas con la recepción e interpretación de la información transmitida a través de la interfaz RS232.

Mientras el hilo está en ejecución, ejecuta un bucle infinito de lectura cada 20 ms. Cuando el *buffer* de recepción contiene un mensaje, se realiza una lectura del mensaje hasta el carácter `\n`. Entonces *Commander* publica el comando recibido, que será más adelante leído y gestionado por el hilo *Publisher*.

Publisher

La funcionalidad del hilo *Publisher* se enfoca en gestionar los eventos que recibe desde la cola de eventos, la cual es alimentada por `event::publish`. Esta interfaz puede ser accedida desde cualquier parte del código, lo que permite una integración flexible y eficiente en el sistema.

El hilo *Publisher* incorpora mecanismos de sincronización de tareas de C++, previniendo posibles bloqueos al producirse el acceso simultáneo de información al mismo objeto (Por ejemplo, dos hilos encolando simultáneamente un evento en *Publisher*.

QRFinder

QRFinder es el hilo encargado exclusivamente de la detección del código QR que contiene las credenciales de acceso a *WiFi*. Por ello, sólo es activado en caso de que el dispositivo entre en modo *Configuration*.

Una vez se lanza el hilo, se establece en primer lugar un tiempo de ejecución máxima tras el cual el hilo dejará de ejecutarse. De esta forma, se impide que el dispositivo quede perpetuamente buscando un código QR válido.

`MAX_QR_FINDER_TIME_MS` se ha establecido en 120000 ms (2 minutos)

Al iniciar el hilo, el ESP32Cam transmite un mensaje informativo tipo *Ack (Acknowledge)* que informa a la UC de que se ha iniciado el modo configuración correctamente.

Acto seguido, comienza el bucle de detección del código, que se mantendrá activo hasta que se detecte un código válido, o bien se agote el tiempo prefijado para la tarea, finalizando el hilo y volviendo al modo prefijado por el usuario.

Cuando se detecte un código válido, el propio hilo compone el mensaje con los campos *ssid*, contraseña, *type* e ID. El mensaje, acto seguido, es publicado.

Cabe destacar que, cuando esto se produce, el dispositivo ESP32Cam publica un evento *UpdateNetworkConfigurationRequest*. El manejador correspondiente entonces es el que modifica los parámetros almacenados en el dispositivo para que sean acordes a su nueva configuración, y se publica un evento *NetworkConfigurationChanged*, con el objetivo de que la UC almacene los nuevos parámetros y planifique el reinicio para conectar a la nueva red.

CAPÍTULO 4

Diseño de la carcasa

En esta sección se explica detalladamente el desarrollo de la carcasa del prototipo. La misma se ha elaborado diseñando un modelo de extensión `.STL`, compatible con impresión 3D, para contener toda la electrónica.

El objetivo final de este apartado es explicar cómo se ha diseñado una carcasa adecuada al prototipo, portátil, fácilmente instalable en una pared y que permita un acceso sencillo al interior de la misma.

En el diseño se han previsto futuras actualizaciones que puedan suponer la implementación de nuevos elementos de hardware.

4.1. Metodología empleada

En esta sección se describe detalladamente el proceso de elaboración de la carcasa del prototipo, así como las herramientas empleadas para su diseño y fabricación. Se explican los criterios considerados para asegurar la funcionalidad del componente.

4.1.1. Medidas

En primer lugar, se han tomado todas las medidas necesarias para el desarrollo del modelo, teniendo en cuenta que el dispositivo necesitará espacio tanto para ventilación como para conducción de cableado además del previsto paara distintos elementos de *hardware*.

Las medidas recogidas de forma previa a la fase de modelado son las mostradas a continuación en la tabla 4.1

Concepto	**Medida**
Diámetro de botones de apertura	23.45 mm
Profundidad de botones de apertura	67 mm
Diámetro de botón de interfaz	12.20 mm
Profundidad de botón de interfaz	32 mm
Dimensiones placa principal	104 x 100 mm
Espacio mínimo de alojamiento para los componentes de la placa principal	25 mm
Espacio de elevación bajo la placa principal	17 mm
Dimensiones de display (visible)	27 x 16 mm
Dimensiones de placa de display	28 x 28 mm
Separación entre perforaciones de la placa de CU	90 mm (ø4 mm)
Distancia agujeros placa display	23 mm (ø2 mm)
Grosor display (para encastrar)	1.6 mm
Diámetro exterior de eje engranado	6 mm
Tornillería de sujeción del compartimento para ESP32Cam	ø3 mm
Espacio de alojamiento para ESP32Cam	32 mm
Dimensiones placa ESP32Cam	40 x 28 mm
Espesor placa ESP32Cam	1 mm
Diámetro de cámara OV2640	8 mm
Medidas alojamiento para engranaje ESP32Cam	ø13 mm x 3.3 mm
Diámetro de rosca para antena de radio	ø6 mm
Dimensiones módulo GPS	26 x 36 mm
Diámetro de perforaciones de módulo GPS	ø3mm
Dimensiones módulo de antena GPS	28 x 28 mm

Tabla 4.1: Tabla de referencia de las dimensiones utilizadas en el diseño de la carcasa

4.1.2. Modelado 3D compatible con impresión 3D

La fabricación aditiva o impresión 3D, atendiendo a (Marcillo Parrales, Mero Lino y Ortíz Hernández, 2021, p. 153), se define como:

> La fabricación aditiva, también conocida como impresión 3D, es el proceso de fabricación de piezas a partir de modelos digitales 3D. Los fabricantes utilizan programas de diseño asistido por computadora (CAD) para construir un modelo digital que finalmente se corta en capas. La impresora 3D coloca las capas de abajo hacia arriba para construir el objeto. Si bien la fabricación de aditivos es aun relativamente nueva en los pisos de las fábricas, está causando un gran revuelo con su potencial como una forma más económica y eficiente

de hacer casi cualquier producto, desde piezas de automóviles estándar hasta piezas para uso en el espacio exterior.

Esta descripción se ajusta a la metodología de fabricación aplicada a la elaboración del prototipo. El diseño de la pieza se ha realizado en el software *Solidworks*.

4.1.3. Proceso de diseño

Modelado 3D con SolidWorks

En la figura 4.1 se muestra el logotipo de esta aplicación.

Figura 4.1: Logotipo de SolidWorks

El modelado 3D se realizará utilizando un software Computer-Aided Design (CAD), concretamente *SolidWorks* en su versión 2024. Esta elección se debe a la flexibilidad de su interfaz para el diseño tridimensional y a su amplia compatibilidad con diversos formatos. Además, SolidWorks permite simular ensamblajes de las piezas diseñadas, lo que facilita la detección de errores en el diseño antes de proceder a la impresión de los componentes.

En primer lugar, se ha diseñado en alzado el perfil de la caja. Posteriormente, es extruido y se le aplica un vaciado al cuerpo para obtener una carcasa hueca con 3 mm de grosor, para dotar de cierta resistencia al cuerpo de la carcasa.

Finalmente, usando distintos planos posicionados sobre las caras, se aplican las operaciones booleanas necesarias para practicar los huecos previstos para los botones de la interfaz, botones de apertura y cierre, pantalla, y agujeros para el paso de cables así como rejillas de ventilación.

En último lugar, se practican los huecos para las pestañas de sujección. Dado que el prototipo ha de ser capaz de mantener su propio peso, se han diseñado de manera que se inserten dentro de las propias paredes de la carcasa.

En las figuras 4.2 y 4.3 se muestran las capturas de la pieza de la carcasa frontal desarrollada en Solidworks.

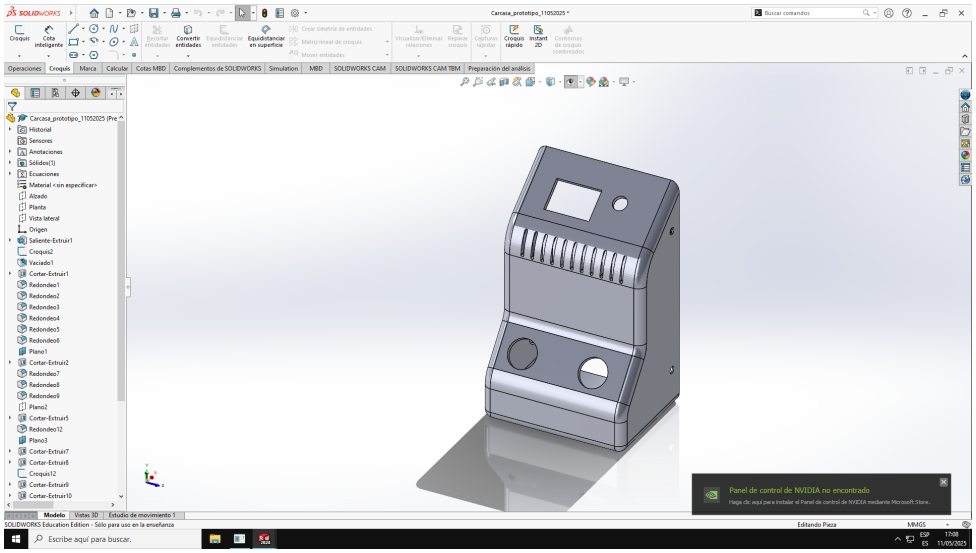


Figura 4.2: Vista frontal del modelo de la carcasa desarrollado en SolidWorks. (también en anexo A.12)

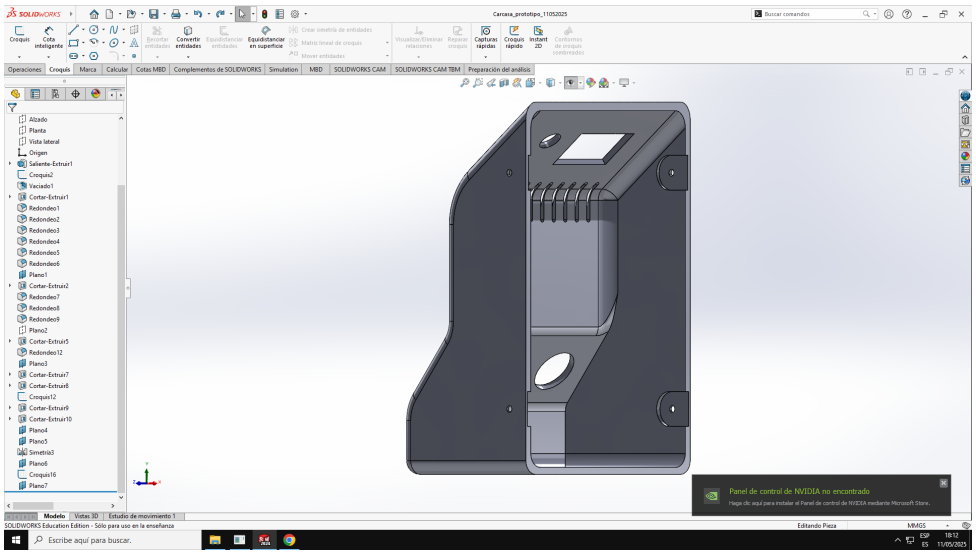


Figura 4.3: Vista trasera del modelo de la carcasa desarrollado en SolidWorks (también en anexo A.13)

En el caso de la tapa trasera, el modelado se ha iniciado desde su diseño en planta. Una vez establecidas las dimensiones, en primer lugar se aplica la pendiente de 30º en su borde superior. Esta inclinación es necesaria para que la tapa encaje perfectamente con el cuerpo de la carcasa frontal.

A continuación, se extruyen los soportes sobre los que se fijará la placa correspondiente a la UC, vaciándolos para permitir el paso de la tornillería. La altura de los soportes está diseñada para dejar un espacio suficiente que facilite la manipulación del cableado y favorezca la ventilación.

En las figuras 4.4 y 4.5 se muestra el modelo en 3D de la tapa trasera de la carcasa, que es la parte del prototipo que se fijará a la pared.

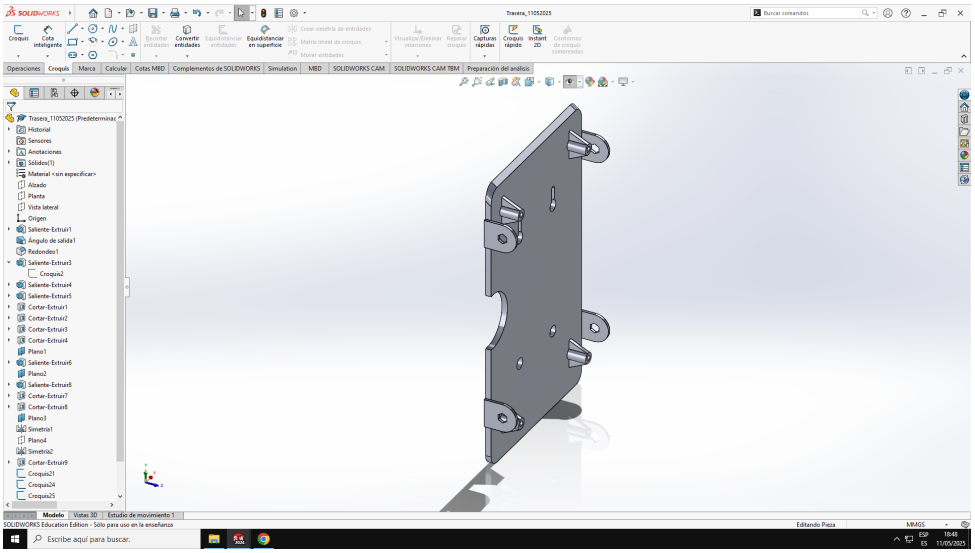


Figura 4.4: Vista frontal del modelo de la tapa trasera desarrollado en SolidWorks (también en anexo A.14)

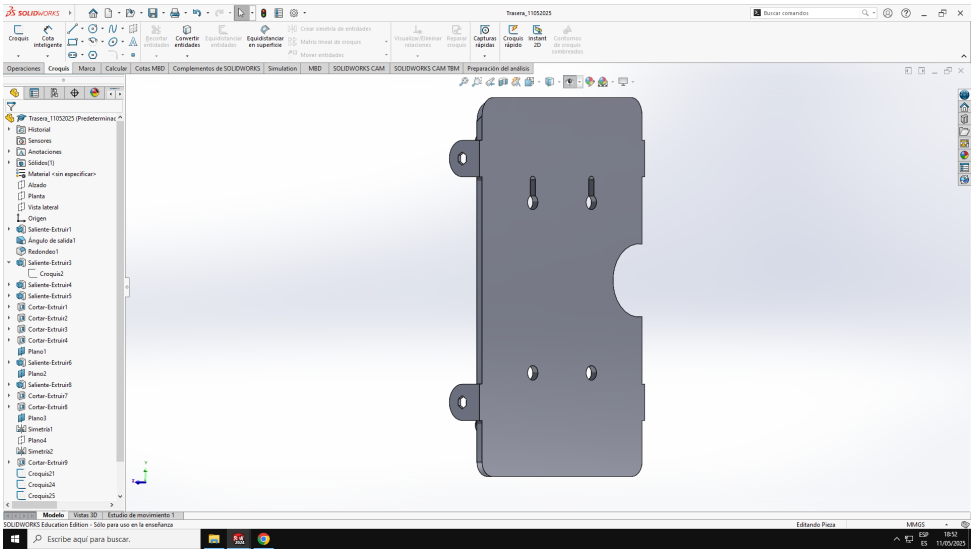


Figura 4.5: Vista trasera del modelo de la tapa trasera desarrollado en SolidWorks (también en anexo A.15)

También se ha diseñado el alojamiento del módulo ESP32Cam, de forma que en un momento ulterior del desarrollo del prototipo, resulte sencillo acoplar un eje motor que permita orientar la vista de la cámara de forma remota.

En las figuras 4.6 y 4.7 se muestra el modelo 3D planteado para impresión aditiva del compartimento que ha de alojar el módulo ESP32Cam. La pieza mostrada en la figura 4.8 es la que engranará con un servomotor situado en el interior de la carcasa, y que a su vez será conectado a la UC.

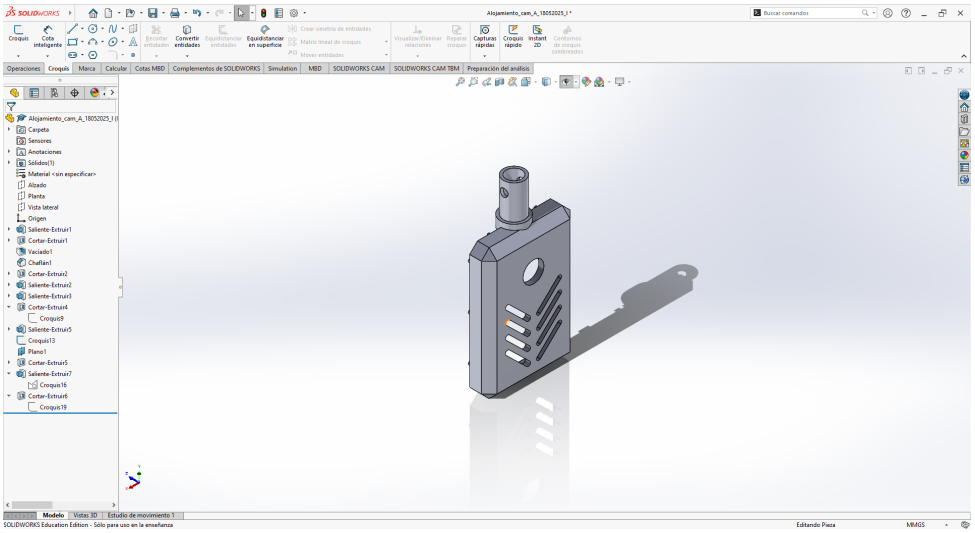


Figura 4.6: Vista de la mitad A del compartimento para alojar el módulo ESP32Cam (también en anexo A.16)

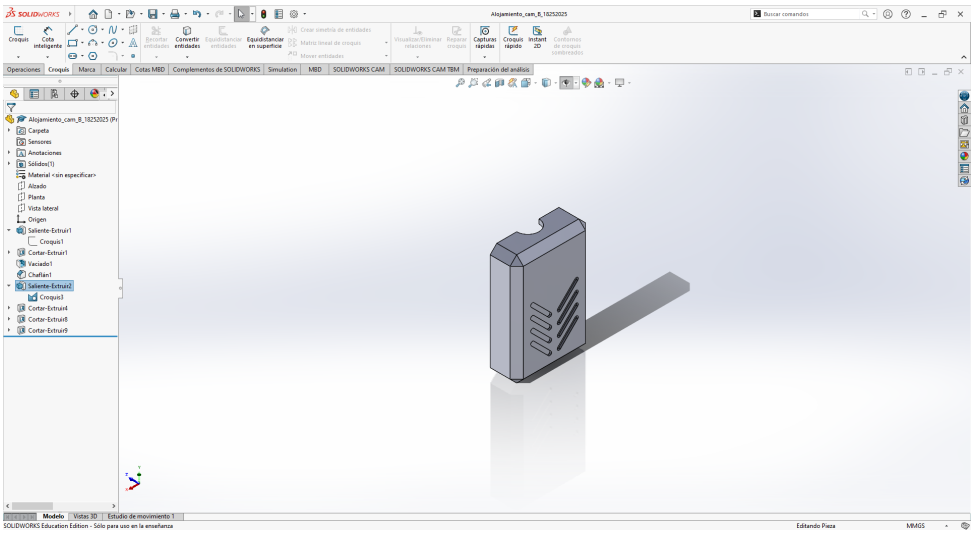


Figura 4.7: Vista de la mitad B del compartimento para alojar el módulo ESP32Cam (también en anexo A.17)

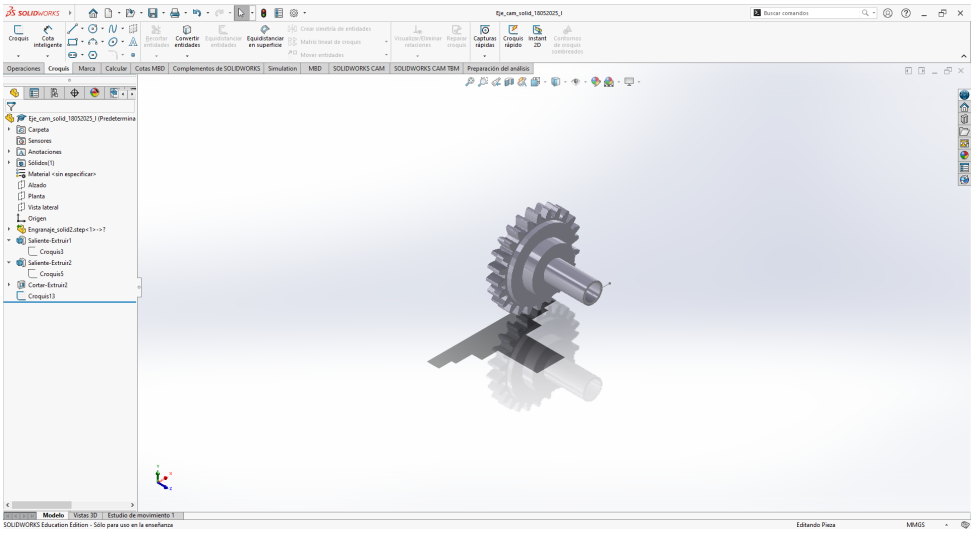


Figura 4.8: Pieza con engrane que conecta el compartimento ESP32Cam con el interior de la carcasa. (también en anexo A.18)

Para ofrecer una visión global de todos los elementos, a continuación se expone la figura 4.9, que muestra una vista explosionada del conjunto. Todas las piezas representadas han sido diseñadas para poder exportarse en un formato compatible con impresión 3D.

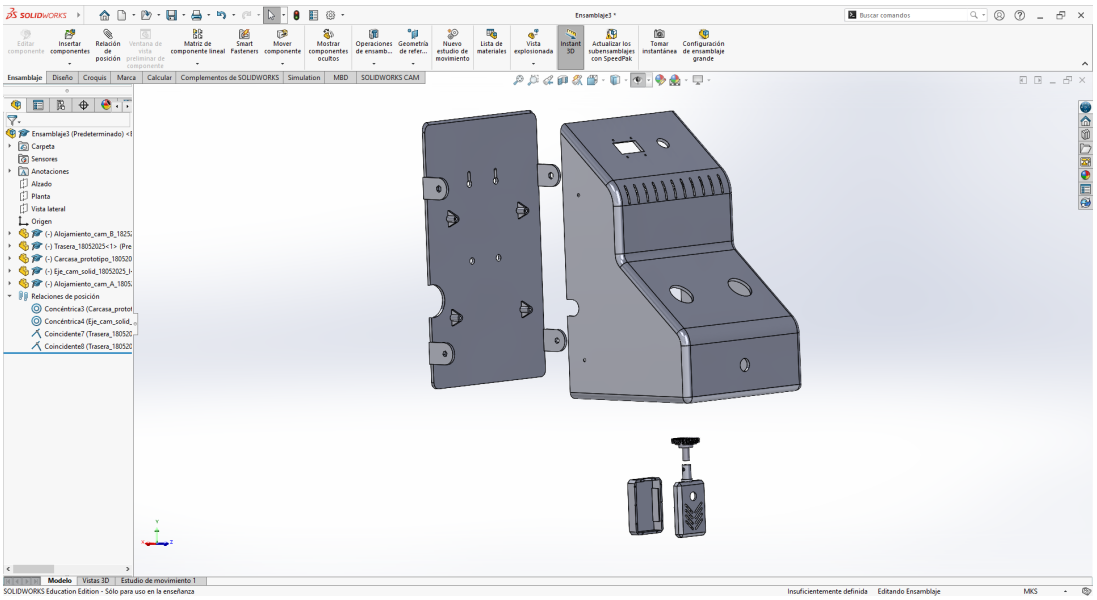


Figura 4.9: Vista del conjunto explosionado de piezas de componen la carcasa para el prototipo.(también en anexo A.19)

4.1.4. Software Slicer - UltiMaker Cura

En la figura 4.10 se muestra el logotipo identificativo del programa usado.

Figura 4.10: Logotipo de Ultimaker Cura

El siguiente paso es la exportación del modelo a un formato compatible con un software de laminado (o *Slicer*).

Ultimaker Cura es uno de ellos, de código abierto desarrollado por la empresa holandesa *Ultimaker*. Su función principal es convertir modelos tridimensionales digitales en instrucciones específicas que puede entender una impresora 3D, generando un archivo en formato G-code. Este proceso de "laminado" consiste en dividir el modelo en capas horizontales (*slices*) y calcular la trayectoria exacta que seguirá la boquilla de impresión para construir cada capa, incluyendo ajustes de temperatura, velocidad, relleno, soporte y más. *Cura* es ampliamente utilizado tanto en entornos educativos como profesionales, gracias a su interfaz intuitiva, su gran compatibilidad con impresoras 3D de distintas marcas y su capacidad de personalización avanzada.

Los archivos que se importan a Cura para preparar una impresión suelen estar en formato STereoLitography (STL). Este formato es un estándar en la impresión 3D porque describe exclusivamente la geometría de un objeto mediante una malla de triángulos que representa su superficie, sin incluir información de color, textura o escala. Esta simplicidad hace que los archivos sean ligeros, ampliamente compatibles con distintos programas de diseño y prácticamente universales entre impresoras 3D Fused Deposition Modelling (FDM). Cura los interpreta como la base para generar el G-code, ajustando parámetros físicos de impresión como altura de capa, grosor de pared, tipo de relleno, soportes, adherencia a la base, entre otros.

Se escoge utilizar *Cura* en lugar de otras opciones como *RepetierHost o Slic3r* principalmente debido a su interfaz, que ofrece las opciones necesarias para la configuración de la impresión de una forma accesible.

SolidWorks tiene la opción para exportar directamente los archivos modelados a formato `.STL`.

Características de la impresión

El material utilizado para la impresión será PolyLactic Acid (PLA), que es un material compatible con todas las impresoras 3D comerciales en el mercado, ampliamente utilizado en prototipado y de bajo coste.

En la figura 4.11 se puede previsualizar el modelo de la carcasaque será usado para la impresión.

El proceso de laminado genera un archivo con extensión .gcode, que contiene una secuencia de comandos en lenguaje máquina utilizados por el firmware de la impresora

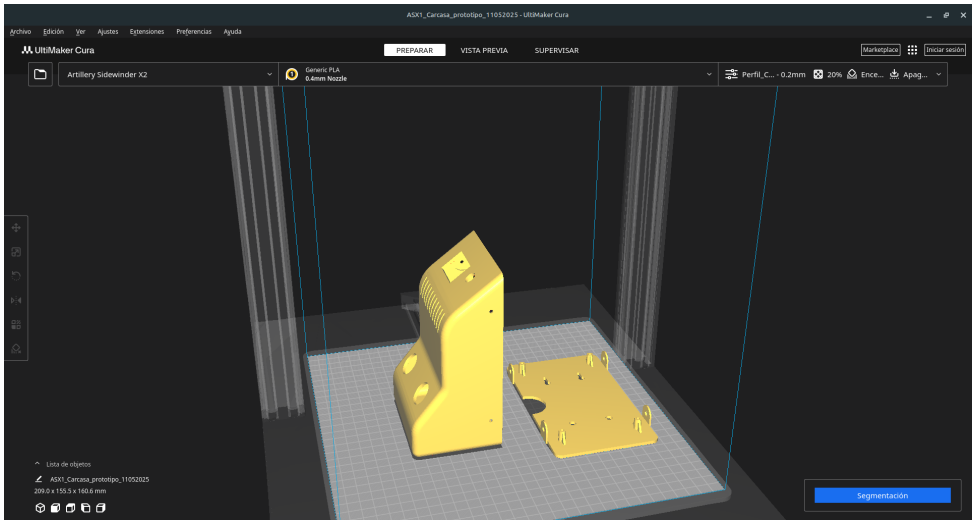


Figura 4.11: Captura de el modelo STL cargado en Cura para el laminado

3D para controlar de forma precisa los movimientos, temperaturas y demás parámetros operativos durante la fabricación.

G-code (código G) es un lenguaje de programación utilizado para controlar máquinas automatizadas, especialmente en Computer Aided Manufacturing (CAM), como impresoras 3D, fresadoras CNC y cortadoras láser. Este lenguaje consiste en una serie de instrucciones que indican a la máquina qué movimientos debe realizar, en qué coordenadas, a qué velocidad, y con parámetros tales como temperatura, velocidad de extrusión o cambios de herramienta. Cada línea del archivo G-code representa una acción específica.

El uso de PLA es preferente en la impresión 3D, debido a diversos factores. Entre ellos destacan su versatilidad, sostenibilidad y facilidad de uso. Se trata de un material muy accesible y fácilmente manipulable por la mayoría de impresoras.

El poli (ácido láctico) o ácido poliláctico (PLA) es un poliéster alifático termoplástico derivado de recursos renovables, de productos tales como almidón de maíz, tapioca, o caña de azúcar.

4.2. Planos técnicos

Todos los planos técnicos normalizados se han incluido en el anexo A.4.

CAPÍTULO 5

Test y validación

En este capítulos se detallan las pruebas de funcionamiento que se han realizado sobre el *software*, los resultados obtenidos y las herramientas a las que se ha recurrido para llevar a cabo todas las pruebas. se recogen capturas de los logs recibidos, fotografías del montaje preliminar del dispositivo, e información al respecto. El capítulo se ha estructurado en torno a las distintas funcionalidades que han sido desarrolladas.

5.0.1. Inicialización correcta del dispositivo

En primer lugar, se llevan a cabo las pruebas necesarias para la inicialización del dispositivo. Para garantizar que este proceso se realice correctamente, es fundamental asegurarse de que todos los hilos se hayan inicializado adecuadamente. Para ello, se han implementado trazas en el código que permiten, a través de la consola de VSCode, monitorear y verificar el arranque de cada hilo.

Los hilos que deberían ser inicializados son los que se muestran en la figura 3.17. Cada hilo contiene su propia traza que será procesada por la librería `< esp_log.h >` en tiempo de ejecución. En la tabla 5.1 se reflejan los tests llevados a cabo para verificar la correcta inicialización del dispositivo, y los mensajes mostrados en consola tras la programación del dispositivo.

Test	**Resultado esperado**	**Resultado obtenido**
Inicialización de hilo *Device*	Device: DeviceReady	Device: DeviceReady
Inicialización de hilo *Camera*	Worker thread started	Worker thread started
Inicialización de hilo *Scheduler*	Worker thread started	Worker thread started
Inicialización de hilo *Network*	Worker thread started	Worker thread started
Inicialización de hilo *Commander*	Worker thread started	Worker thread started

Tabla 5.1: Tabla de test de inicialización y resultados obtenidos.

Para poder comprobar que se produce la inicialización del hilo *Commander*, primero se debe deshabilitar el modo de depuración remota, ya que la inicialización del mismo se supedita a que el modo remoto esté desactivado.

Para ello se configura en el archivo *platformio.ini* el flag personalizado `-DREMOTE_DBG=0`

En la figura 5.1 se muestran los mensajes recibidos en consola a través de puerto UART.

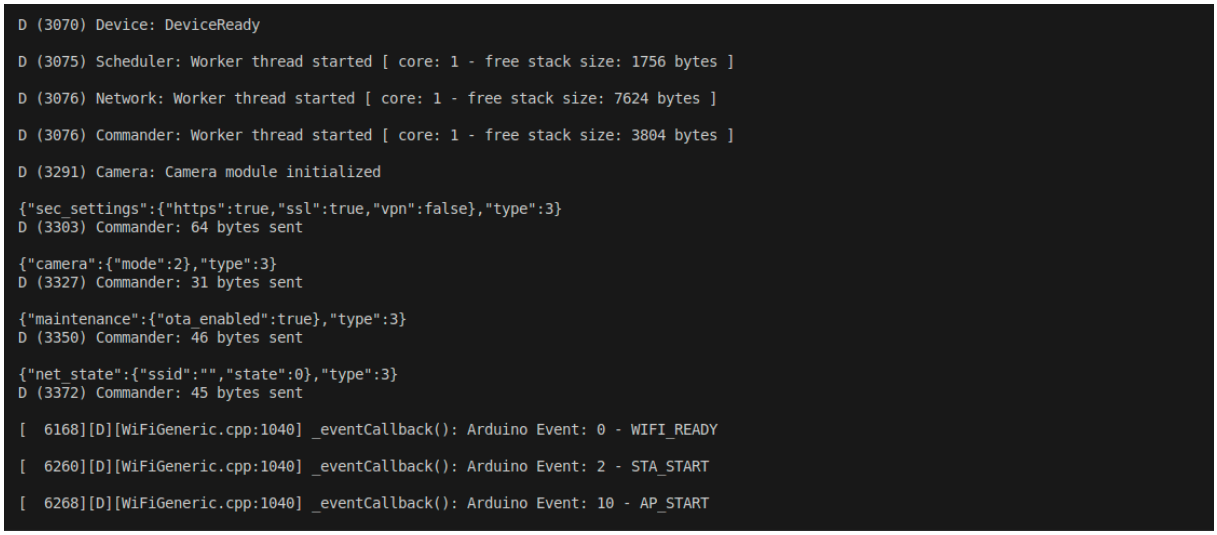


Figura 5.1: Logs de inicialización del dispositivo

Acto seguido, se comprueba que el hilo *Commander* está funcionando, mostrando el envío de telemetría con los parámetros de configuración inicial a la UC

5.0.2. Conexión establecida con la red

El siguiente paso consiste en verificar que, una vez iniciado el hilo *Network*, se establece correctamente la conexión a la red. Si se analiza detenidamente la máquina de estados implementada en dicho hilo —la cual se incluye en el anexo A.4—, se observa que, en caso de no lograrse la conexión, el hilo permanece en un bucle de configuración infinito a la espera de completarla. Este comportamiento es deliberado y tiene como propósito evitar el envío de información sensible a través de cualquier canal hasta que la conexión se haya establecido de manera segura y efectiva.

Para visualizar el comportamiento del hilo, el mismo cuenta con su propia codificación que notifica sobre el estado de la conexión a medida que recorre los distintos estados de la misma. Esta se encuentra ya definida dentro de la librería *esp_wifi.h*, que forma parte del *framework* de ESP-IDF. Estos tipos se han recogido en la tabla 5.2.

Código	Nombre del Evento	Descripción
0	WIFI_READY	El módulo WiFi está inicializado y listo.
1	SCAN_DONE	Se ha completado un escaneo de redes WiFi.
2	STA_START	Se inicia el modo estación (cliente WiFi).
3	STA_STOP	Se detiene el modo estación.
4	STA_CONNECTED	Se ha establecido conexión con el punto de acceso.
5	STA_DISCONNECTED	Se ha perdido o fallado la conexión con el punto de acceso.
6	STA_AUTHMODE_CHANGE	Se ha cambiado el modo de autenticación del punto de acceso.
7	STA_GOT_IP	El dispositivo ha recibido una dirección IP.
8	STA_LOST_IP	El dispositivo ha perdido la dirección IP.
9	STA_WPS_ER_SUCCESS	El proceso WPS se ha completado exitosamente.
10	AP_START	Se ha iniciado el modo punto de acceso.

Tabla 5.2: Eventos *WiFi* del ESP32 definidos por `wifi_event_t`.

Dado que el módulo ESP32Cam necesita conectarse a un punto de acceso en modo STA (*Station* o estación), la conexión se considerará establecida cuando en el log configurado se muestre el mensaje correspondiente al código '10', que indica que la configuración se ha establecido completamente.

El mensaje recibido al final de la secuencia de conexión es el esperado en la tabla 5.3. El mensaje aparece invocado por el *framework* Arduino debido a que este coexiste en el proyecto junto al de ESP-IDF, permitiendo el uso de funcionalidades de ambos en el ESP32Cam.

La tabla 5.3 resume los resultados esperables.

Test	Resultado esperado	Resultado obtenido
Conexión realizada completamente	10 Arduino Event: - AP_START	10 Arduino Event: - AP_START

Tabla 5.3: Tabla de test de telemetría de red y resultados obtenidos.

En la figura 5.2 se muestran los mensajes entregados por el ESP32Cam, que confirman que la conexión a la red con SSID *TTBRed* se ha realizado satisfactoriamente. Esta red ha sido preparada previamente, de forma específica, para probar la conectividad del dispositivo.

```
[  6166][D][WiFiGeneric.cpp:1040] _eventCallback(): Arduino Event: 0 - WIFI_READY

[  6256][D][WiFiGeneric.cpp:1040] _eventCallback(): Arduino Event: 2 - STA_START

[  6264][D][WiFiGeneric.cpp:1040] _eventCallback(): Arduino Event: 10 - AP_START

W (8432) Network: Connecting to SSID TTBred

{"net_state":{"ssid":"TTBred","state":1},"type":3}
D (8434) Commander: 51 bytes sent
```

Figura 5.2: Captura de los mensajes de telemetría de conexión en el log de VSCode

Tanto el identificador como contraseña de la red se configuran en el directorio /include/config/Types.h, junto al resto de parámetros de configuración como puertos, Uniform Resource Locator (URL) de conexión o *topics* de suscripción en *mqtt*. Estas variables, accesibles en tiempo de ejecución, son las usadas para llevar a cabo la conexión de red por defecto.

Adicionalmente, es posible saber si la conexión se ha realizado correctamente observando los mensajes devueltos en la User Interface (UI) de la propia UC. Se muestra un ejemplo en la figura 5.3

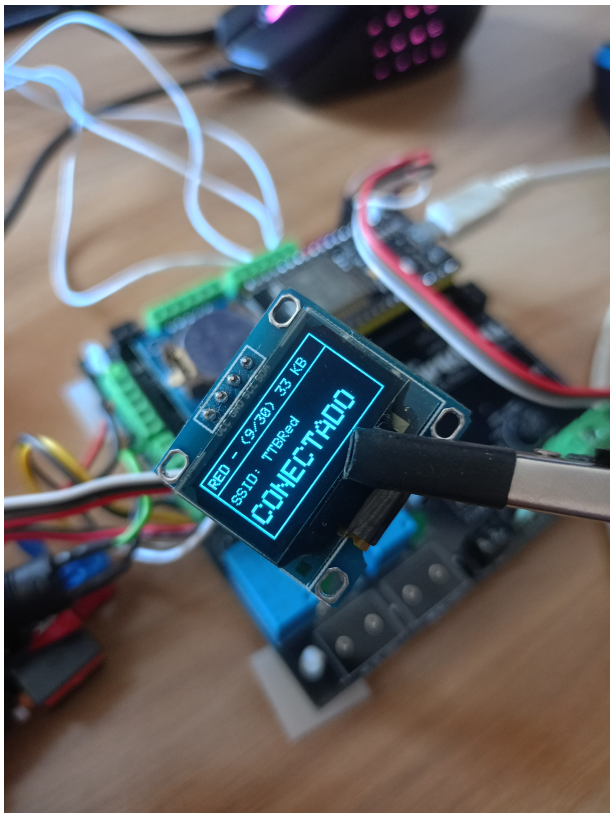


Figura 5.3: UI de la Unidad Central con la conexión satisfactoriamente abierta

5.0.3. Sincronía entre los módulos de UC y ESP32Cam por puerto serie RS232

Una vez verificada la correcta conexión a la red, el siguiente paso consiste en comprobar que la comunicación con la UC funciona adecuadamente. Gracias a la funcionalidad del hilo *Commander*, es posible leer directamente los paquetes de datos transmitidos a través del monitor de puerto serie. Para ello, basta con monitorizar la recepción de mensajes tipo "ack" tras el envío de telemetría desde el ESP32Cam, por ejemplo, durante el arranque del sistema.

También se comprueba si el ESP32Cam es capaz de entrar en modo configuración al recibir un comando tipo *cmd* directamente desde la UC por interacción directa de el usuario, solicitando el reinicio en modo "configuración".

Para este test, es imprescindible realizar la conexión física entre ambos dispositivos, siguiendo el esquema mostrado en la figura 3.1.

Dado que la comunicación se realiza a través del puerto serie, los mensajes enviados por el ESP32Cam deberían ser visibles en la consola. La correcta visualización de estos mensajes permitirá confirmar que la conexión física se ha establecido adecuadamente.

En la tabla 5.4 se plantean aquellos test que verifican que la conexión por puerto serie se está realizando correctamente entre el ESP32Cam y la UC. Al recibirse los comandos compuestos con un campo `type = 3`, queda comprobado que se trata de mensajes de tipo *tlm* (telemetría), emitidos desde otra unidad.

Test	**Resultado esperado**	**Resultado obtenido**
Telemetría de configuración de seguridad	sec_settings:{https:true,ssl:true, vpn:false, type: 3}	sec_settings:{https:true,ssl:true, vpn:false, type: 3}
Telemetría de configuración de cámara	camera:{mode:2, type:3}	camera:{mode:2, type:3}
Telemetría de configuración de mantenimiento	maintenance:{ota_enabled :true,type: 3}	maintenance:{ota_enabled :true,type: 3}

Tabla 5.4: Tabla de tests de telemetría de red y resultados obtenidos.

En la figura 5.4 se muestra la captura de pantalla con los logs recibidos confirmando que la conexión se ha llevado a cabo satisfactoriamente.

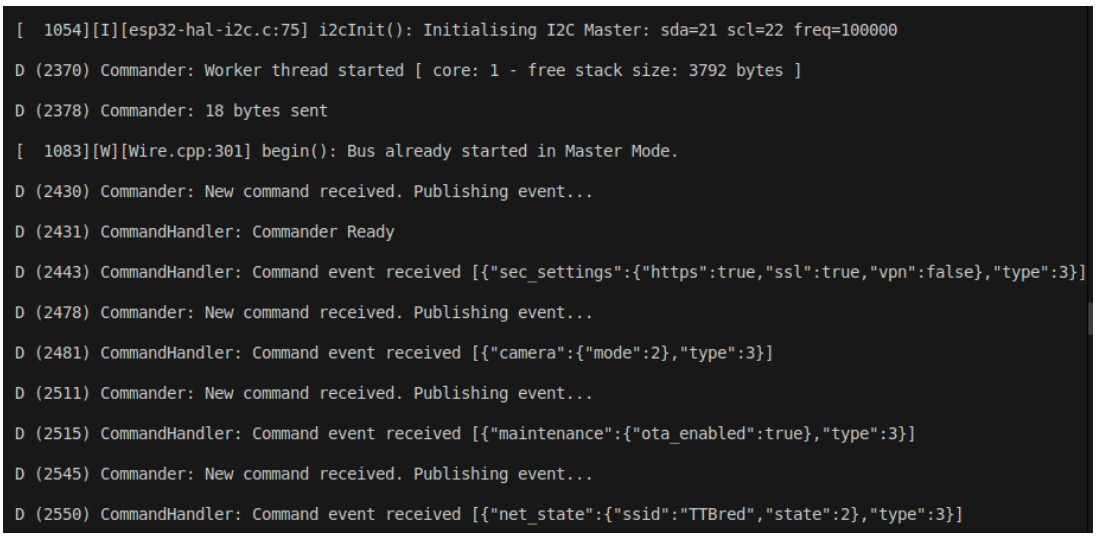


Figura 5.4: Captura de pantalla de los logs recibidos a través de puerto serie

5.0.4. Sincronía entre los módulos de UC y Esp32Cam por protocolo MQTT

Se comprueba a continuación que la conexión a través de MQTT se realiza correctamente. Para que la conexión sea comprobada, los mensajes enviados por el hilo *Network* deben poder ser visualizados por cualquier cliente suscrito a los *topics* correctos y conectado al *broker* MQTT.

Dado que Node-RED cuenta con herramientas para ello, se hará la prueba de conexión con MQTT en el propio cliente de la plataforma. Si los mensajes son recibidos, significa que la conexión se ha establecido correctamente, y que el dispositivo está publicando los mensajes adecuadamente dentro de los *topics* establecidos.

Se resume el comportamiento esperado en la tabla 5.5. El dispositivo publica sus mensajes en un *topic* con la forma `ttb/aparka/{ID del dispositivo}/{tópico de subscripción}`.

Test	**Resultado esperado**	**Resultado obtenido**
Telemetría de configuración de seguridad	sec_settings:{https:true, ssl:true, vpn:false}, type: 3	sec_settings:{https:true, ssl:true, vpn:false}, type: 3
Telemetría de configuración de cámara	camera:{mode:2,type: 3}	camera:{mode:2,type: 3}
Telemetría de configuración de mantenimiento	maintenance:{ota_enabled :true, type:3}	maintenance:{ota_enabled :true, type:3}

Tabla 5.5: Tabla de tests para conexión MQTT desde ESP32Cam

En la figura 5.5 se muestra el esquema de Node-RED desplegado para habilitar la captura de los mensajes trasmitidos a través de MQTT entre los componentes del proyecto.

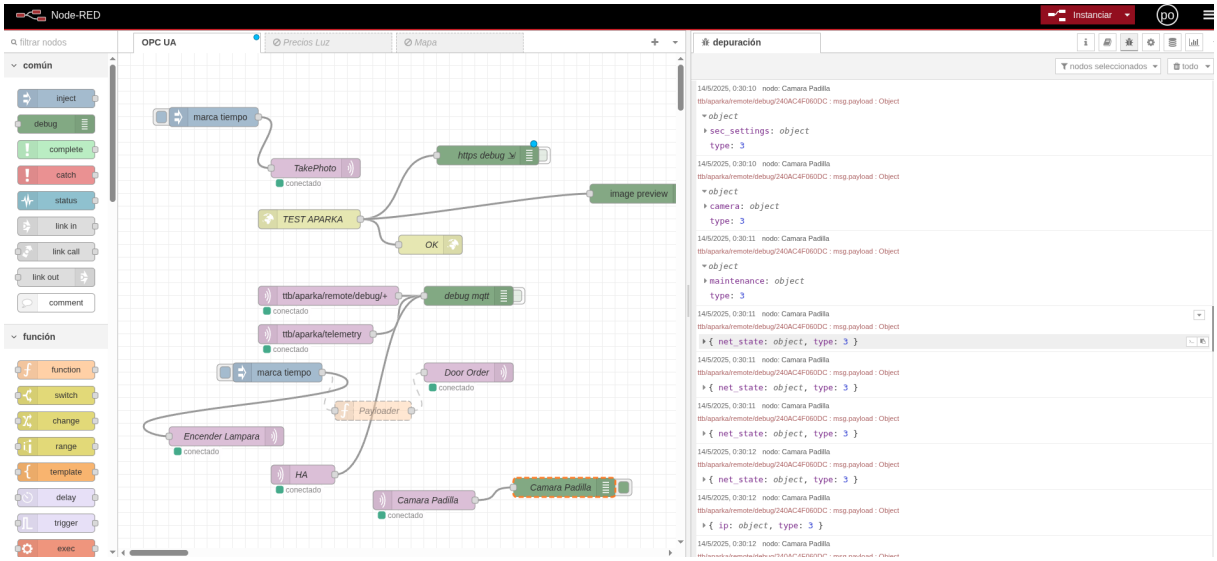


Figura 5.5: Captura de estructura desplegada en Node-RED

La figura 5.6 es una ampliación de la 5.5 que muestra el detalle de los mensajes recibidos en Node-RED.

Se trata de los mismos mensajes enviados a la UC, y publicados en el *topic* correspondiente al ESP32Cam, el cual se encuentra monitorizado. En la misma imagen se puede observar el *topic* de publicación de los mensajes. De esta manera, queda comprobado que las comunicaciones a través del protocolo MQTT funcionan correctamente.

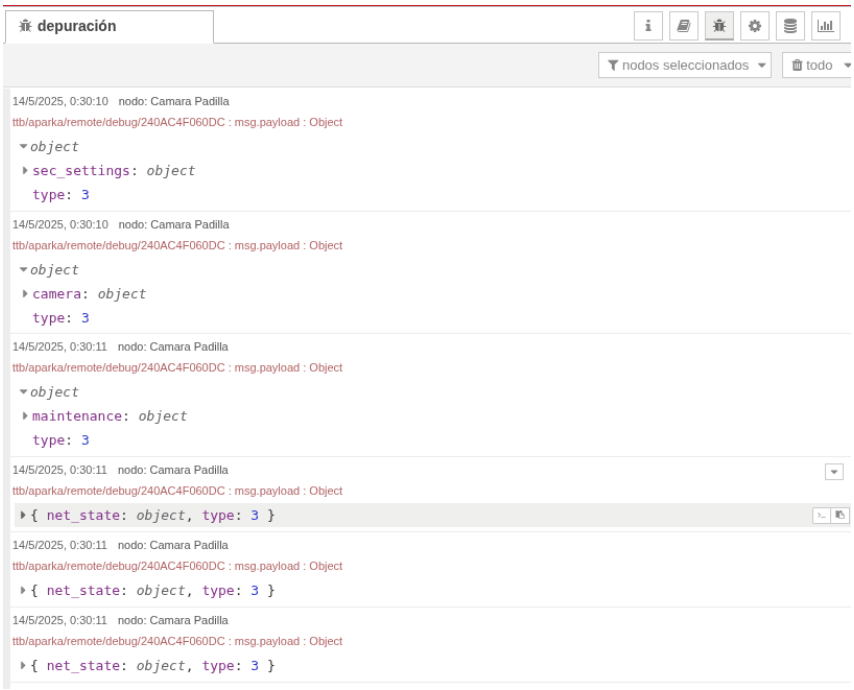


Figura 5.6: Detalle de los mensajes capturados en Node-RED a través de MQTT

5.0.5. Funcionamiento correcto de lector QR

Esta funcionalidad permite reconfigurar el dispositivo mediante la lectura de un código QR que contiene los datos necesarios para la conexión a la red. Para validar este proceso, se ha habilitado una red de pruebas específica destinada a facilitar la configuración y verificación de la conexión.

Cuando el dispositivo recibe el comando de configuración, el hilo *Device* se reinicia en modo *Configuration*.

Como se puede observar en la figura 3.17, cuando el ESP32Cam se inicia en modo configuración, es cuando se produce el lanzamiento del hilo `QRFinder`. Mientras este hilo se encuentra en ejecución, el dispositivo intenta localizar durante un período de 120 segundos un código correcto para conectar el dispositivo a la red usando las credenciales contenidas en el mismo.

Si transcurre el tiempo límite sin una lectura correcta, el hilo *QRFinder* devuelve un mensaje de error con el texto *Invalid QR*. Después, envía un mensaje informativo que será retransmitido por el canal correspondiente, y solicitará la actualización de los parámetros de red.

El dispositivo se reinicia cargando el último modo de funcionamiento previo al modo de configuración, que permanece almacenado en la memoria *flash*.

En la tabla 5.6 se enumeran los tests llevados a cabo para comprobar la funcionalidad de detección de código QR.

Test	**Resultado esperado**	**Resultado obtenido**
QR incorrecto: Mensaje Invalid QR obtenido	Invalid QR	Invalid QR
QR correcto: Envío de telemetría con nuevos datos	id:1, pass:“ ”, ssid:“ ”, type: 0	id:1, pass:“ ”, ssid:“ ”, type: 0

Tabla 5.6: Tabla de tests para conexión reconfiguración a través de código QR

Para comprobar que la comunicación se realiza de forma efectiva, se comprueba que el código QR es leído tras enviarse el comando. El código ha sido generado con una aplicación de terceros, llamada *QRCode Monkey* (`https://www.qrcode-monkey.com/`).

Se hacen dos comprobaciones distintas: mostrando a la cámara un código correcto, y otro incorrecto. En el caso del QR correcto, el dispositivo se reiniciará y enviará un mensaje de telemetría de red con el nuevo SSID y contraseña. En el caso incorrecto, publicará un mensaje informativo con el texto *Incorrect QR*.

En la figura 5.7 se muestra el mensaje recibido a través de protocolo MQTT, comprobando que el funcionamiento es correcto. Efectivamente, se retransmite el mensaje *Invalid QR*, tal y como se esperaba.

Figura 5.7: Captura de mensaje *Invalid QR* transmitido en modo remoto

Por otro lado, en la figura 5.8 se observa el mensaje devuelto por el ESP32Cam cuando la lectura es correcta. Tras comprobar que el código es correcto, y que el tipo es '0', la UC también reconfigura automáticamente sus credenciales para conectarse a la nueva red.

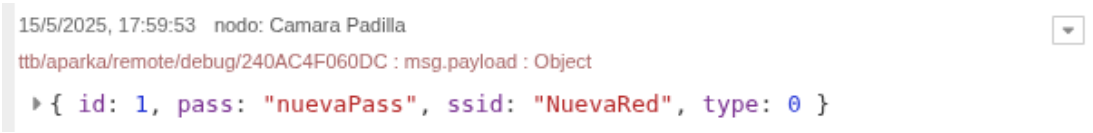


Figura 5.8: Captura de mensaje tras detección correcta del código QR

5.0.6. Solicitud de fotografía remota

Otra de las funcionalidades implementadas en el dispositivo es la capacidad para tomar fotografías a demanda de la UC de forma remota.

Para llevar a cabo la captura, el dispositivo ESP32Cam se encuentra suscrito desde su encendido a un *topic* llamado *takephoto*. En cuanto se publica un mensaje en este *topic*, que será enviado desde la UC, se encola la petición para tomar la fotografía desde el dispositivo de cámara.

La imagen se puede ver en Node-RED, donde se ha generado un nodo suscrito al mismo canal.

Para simular este comportamiento, se han configurado dos nodos en Node-RED, visibles en la esquina superior izquierda de la figura 5.9. Esta configuración publica directamente un carácter vacío en el *topic* `ttb/aparka/9C9C1FEAEDD0/takephoto`.

En la figura 5.9 se muestra una captura de la imagen tomada por la cámara tras recibir el correspondiente comando a través del topic `ttb/aparka/9C9C1FEAEDD0/takephoto`.´

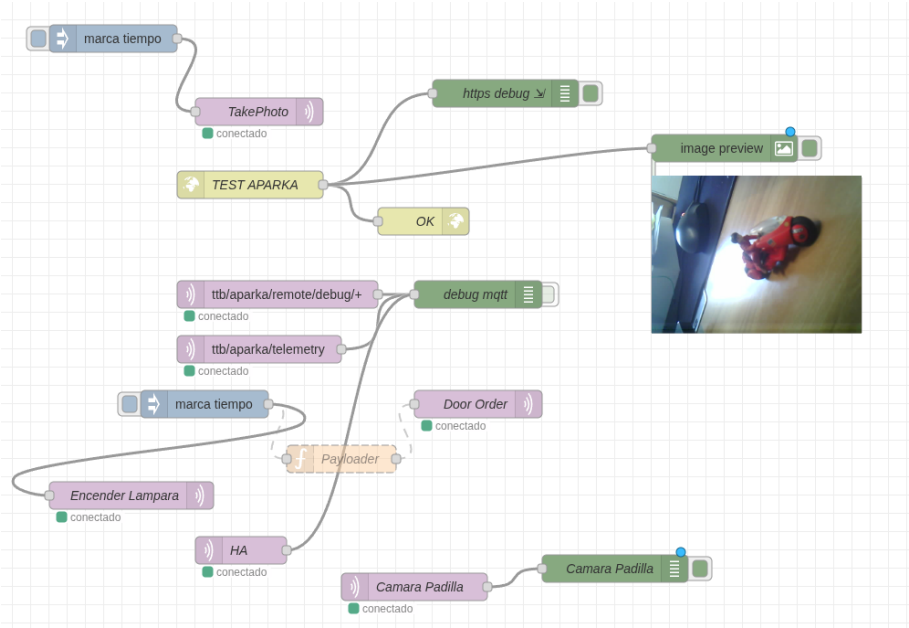


Figura 5.9: Captura de previsualización de imagen en Node-RED

Por tanto, se puede concluir que se cumplen todas las condiciones listadas en la tabla 5.7, y por tanto la funcionalidad ha sido implementada correctamente.

Test	**Resultado esperado**	**Resultado obtenido**
Inicializacion correcta del dispositivo	Previsualización de la imagen capturada y transmitida	Previsualización de la imagen capturada y transmitida

Tabla 5.7: Tabla de test para conexión MQTT desde ESP32Cam

5.0.7. Modo de depuración remota

Como se refiere en la figura 3.17, cuando se inicia el modo de depuración remota, tanto UC como ESP32Cam recibirán los mensajes mutuamente a través de protocolo MQTT, sin necesidad de que exista una conexión física entre ellas.

Esto permite la conexión y configuración de nuevos dispositivos, ya que una vez configurados cada nuevo Esp32Cam publica sus mensajes en su propio topic.

Para verificar el correcto funcionamiento de esta característica, ambos dispositivos deben recibir los mensajes enviados por su contraparte a través de protocolo MQTT y reflejarlos en el monitor del puerto serie como si se tratase de una comunicación por puerto serie.

Así, se comprueba que el modo de depuración remota ha sido implementado correctamente, ejecutando todos los tests de los apartados 5.0.1 a 5.0.6 con el modo remoto activado.

Se considera que si la respuesta de los tests ocurre de forma idéntica a la ya explicada, la funcionalidad ha sido implementada correctamente.

En la tabla 5.8 se condensan los resultados obtenidos. Como todos los tests se comportan exactamente de la misma forma en ambos modos, se puede concluir que esta funcionalidad ha sido implementada correctamente.

Test	**Funcionalidad testada en modo remoto**
Inicialización correcta del dispositivo	✓
Conexión establecida con la red	✓
Sincronía entre los módulos de CU y ESP32Cam por procolo serie RS232	✓
Sincronía entre los módulos de CU y ESP32Cam por procolo serie MQTT	✓
Funcionamiento correcto de lector QR	✓
Solicitud de fotografía remota	✓

Tabla 5.8: Tabla de comprobaciones necesarias para la correcta implementación del modo de depuración remota

Para culminar este apartado, es pertinente hacer referencia a la utilidad de la propia UI integrada en la Unidad Central, que permite de una forma accesible visualizar los parámetros de configuración del dispositivo.

Se incluyen ejemplos de esta funcionalidad en las figuras 5.10, 5.11 y 5.12.

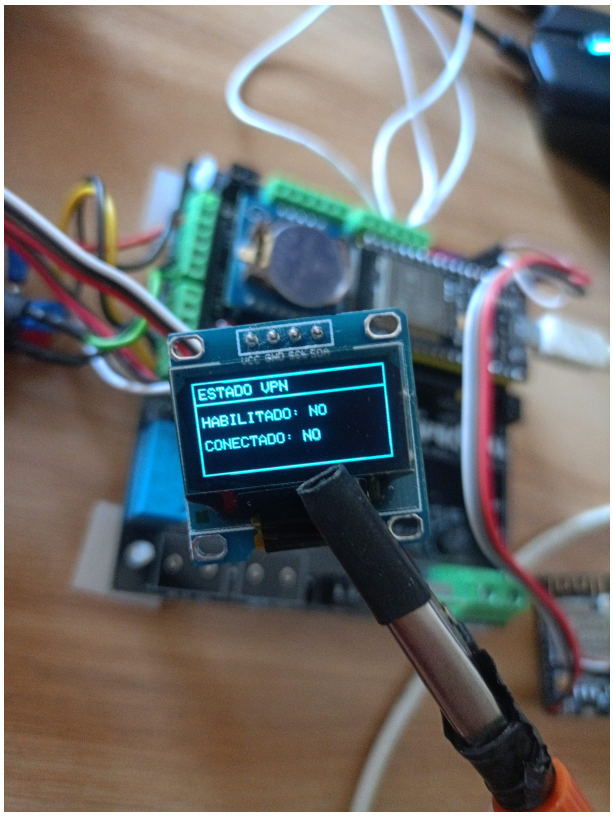


Figura 5.10: Imagen de UI mostrando estado de conexión VPN

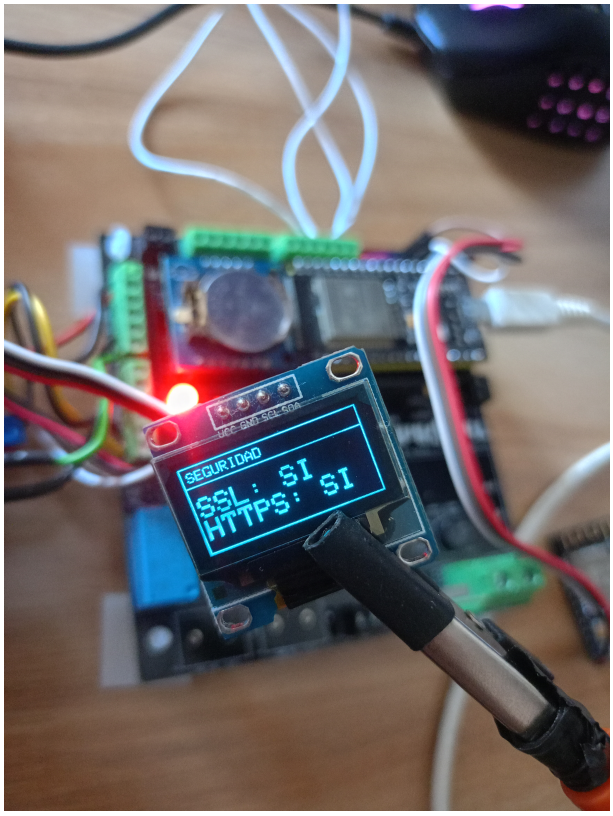


Figura 5.11: Imagen de UI mostrando estado de configuración de seguridad

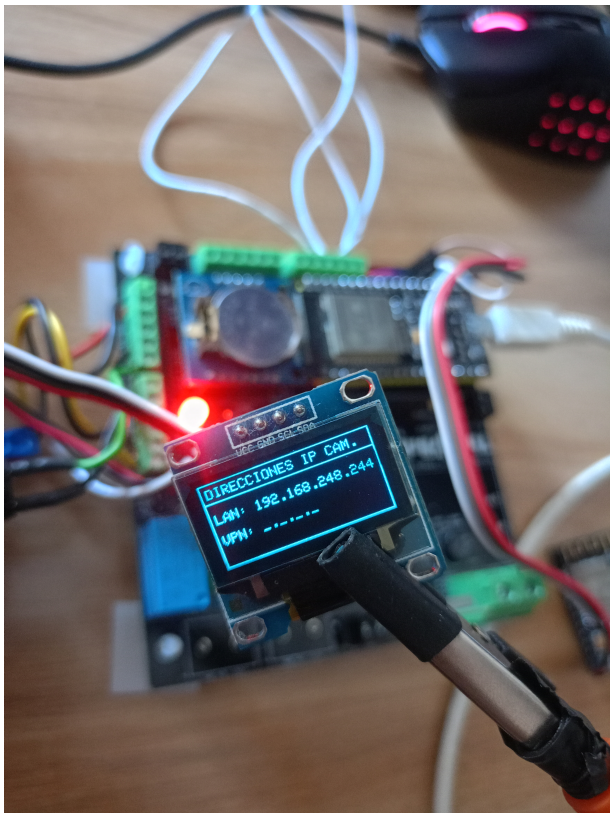


Figura 5.12: Imagen de UI mostrando IP asignada al ESP32Cam

5.0.8. Diario de desarrollo (Problemas encontrados y soluciones aportadas)

Durante el desarrollo del proyecto han aparecido diversos imprevistos sobre los que se ha trabajado para darles una solución eficiente. En este apartado, se condensan distintos contratiempos y la solución aportada al respecto.

En la tabla 5.9 se recogen los más relevantes encontrados durante el desarrollo del prototipo, y la solución que se concluyó más efectiva para solucionar el problema.

Problema encontrado	**Solución aportada**
Reinicio por rebosamiento de memoria en el hardware	Inclusión de mapa de memoria para ajustar la cantidad de memoria disponible y habilitar Pseudo Static Random Access Memory (PSRAM)
El modo de configuración inicia en un modo por defecto, no el escogido por el susuario	Habilitado el almacenamiento en la memoria flash.
No hay escalabilidad al añadir nuevos dispositivos ESP32Cam	El topic de mqtt será la dirección Media Access Control (MAC) del UC.
Los mensajes de telemetría sólo se recogen en los logs. Si se desactivan en modo de producción, no serán accesibles	Creación de una nueva categoría de mensajes exclusivamente para telemetría.

Tabla 5.9: Tabla resumen de soluciones aportadas

CAPÍTULO 6

Conclusiones

Ha sido muy enriquecedor comprobar cómo la aplicación de patrones de diseño específicos en un proyecto real impacta directamente en la eficiencia, la seguridad y el valor que se puede ofrecer a un usuario final. Me ha sorprendido gratamente la capacidad de cómputo del ESP32Cam, ya que ha permitido desarrollar un proyecto escalable utilizando relativamente pocas herramientas externas. De cara a futuras mejoras, resulta interesante considerar el potencial de expansión del proyecto si se migrara a plataformas con mayor capacidad. Esto abriría la puerta a la implementación de algoritmos de Machine Learning en el propio dispositivo, técnicas avanzadas de visión por computador o funcionalidades autónomas.

Explorar cómo se conciben y estructuran este tipo de proyectos desde la perspectiva de la arquitectura de software ha sido especialmente satisfactorio, y me motiva a seguir profundizando en esta área de especialización.

La naturaleza del proyecto también me ha brindado la oportunidad de fortalecer mis conocimientos en programación en C++ y en técnicas de prototipado, competencias que considero esenciales para el perfil profesional que deseo desarrollar, enfocado en la programación de sistemas embebidos y productos tecnológicos.

CAPÍTULO 7

Futuras líneas de trabajo

En este apartado se desarrollan posibles líneas de trabajo futuras para el proyecto, que podrán dotarle de mayores funcionalidades.

7.1. Desarrollo de una aplicación de usuario

Una posibilidad es el desarrollo de una Graphic User Interface (GUI) que permita la gestión de los distintos usuarios a través de una *webapp* de una forma sencilla y accesible, con niveles de permisos de usuario, capacidad para interactuar en remoto con los accesos y otras funcionalidades (Modo de seguimiento de imagen, iluminación auxiliar...) y sistema de notificaciones personalizables.

Esta rama de desarrollo podría también comprender mejoras en el propio display del dispositivo, con el objetivo de hacerlo más cómodo y legible. En tal caso, se tendría que abordar la implementación de un display (táctil o no) más grande que permita integrar una interfaz gráfica más completa diseñada en otros *frameworks* como por ejemplo *QT*.

7.2. Ampliación de funcionalidad

Dada la capacidad de escalabilidad que se ha incorporado al prototipo, se contempla la posibilidad de integrarle nuevos conjuntos de sensores que amplíen la información útil que puede ofrecer al usuario, más allá de los datos actualmente recogidos por el propio dispositivo. Considerando que la aplicación principal de este sistema está orientada a garajes y aparcamientos, resultaría especialmente valioso dotarlo de sensores capaces de medir la calidad del aire, monitorizar el funcionamiento de los sistemas de ventilación o detectar la presencia de personas o vehículos.

La incorporación de estos sensores no solo incrementaría la funcionalidad del prototipo, sino que también podría contribuir a mejorar la seguridad y el confort en los espacios donde se instale. Por ejemplo, la monitorización de la calidad del aire permitiría alertar sobre niveles peligrosos de gases como monóxido de carbono, mientras que la detección de presencia podría facilitar la gestión eficiente de la iluminación o la activación de sistemas de ventilación solo cuando sea necesario. La detección de vehículos también podría

complementarse, con el *hardware* adecuado, con un sistema de detección de matrículas Optical Character Recognition (OCR) que activara otras funcionalidades.

En definitiva, la integración de estos nuevos módulos abriría la puerta a un sistema más completo y adaptable a las necesidades específicas de cada entorno, reforzando el valor añadido que el prototipo puede aportar a sus usuarios finales.

Bibliografía

Amazon Web Services, Inc. (2024). *Guía del usuario de FreeRTOS*. Manual. Último acceso: 5 de mayo de 2025. URL: https://docs.aws.amazon.com/es_es/freertos/latest/userguide/freertos-ug.pdf.

Aprimatic (n.d.). *Imagen del dispositivo RX BTACCES*. Imagen extraída de la web de Aprimatic. URL: https://www.aprimatic.es/producto/seguridad-en-accesos/control-de-acceso/accesorios-control-de-acceso/rx-btacces/.

Ashton, Kevin (2009). «That 'Internet of Things' Thing». En: *RFID Journal*. URL: https://www.rfidjournal.com/articles/view?4986 (visitado 24-03-2025).

Balali, F. et al. (2020). «Internet of Things (IoT): Principles and Framework». En: *Data Intensive Industrial Asset Management*. Cham: Springer. DOI: 10.1007/978-3-030-35930-0_1. URL: https://doi.org/10.1007/978-3-030-35930-0_1.

Espressif Systems (2020). *ESP32-CAM Development Board Datasheet*. Accedido el 16 de Abril de 2025. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

Galiano, Josep Lluís Monte (2016). *Implantar SCRUM con éxito*. ProQuest Ebook Central. Barcelona: Editorial UOC. URL: https://ebookcentral.proquest.com/lib/bibliotecauma-ebooks/detail.action?docID=7051300.

Gershenfeld, Neil, Raffi Krikorian y Danny Cohen (2004). «The Internet of Things». En: *Scientific American* 291.4, págs. 76-81. DOI: 10.1038/scientificamerican1004-76.

Hillar, Gastón C. (2017). *MQTT Essentials – A Lightweight IoT Protocol: The Preferred IoT Publish-Subscribe Lightweight Messaging Protocol*. 1st. Packt Publishing. ISBN: 9781787287815.

Jabraeil Jamali, M.A. et al. (2020). «IoT Architecture». En: *Towards the Internet of Things*. EAI/Springer Innovations in Communication and Computing. Cham: Springer. DOI: 10.1007/978-3-030-18468-1_2. URL: https://doi.org/10.1007/978-3-030-18468-1_2.

Lazzari, L. y K. Farias (2023). «Event-driven architecture and REST architectural style: An exploratory study on modularity». En: *Journal of Applied Research and Technology* 21.3, págs. 338-351. DOI: 0.22201/icat.24486736e.2023.21.3.1764. URL: https://jart.icat.unam.mx/index.php/jart/article/download/1764/995.

Marcillo Parrales, Katherine Gabriela, Evelyn Andreina Mero Lino y María Mercedes Ortíz Hernández (2021). «Impresión 3D como eje de desarrollo en la industria 4.0». En: *Serie Científica de la Universidad de las Ciencias Informáticas* 14.4, págs. 151-160. URL: https://dialnet.unirioja.es/servlet/articulo?codigo=8590504.

Shanmugapriya, D. et al. (2021). «MQTT Protocol Use Cases in the Internet of Things». En: *Big Data Analytics. BDA 2021. Lecture Notes in Computer Science*. Ed. por S.N. Srirama et al. Vol. 13147. Springer, Cham, págs. 160-172. ISBN: 978-3-030-93619-8.

DOI: 10.1007/978-3-030-93620-4_12. URL: https://doi.org/10.1007/978-3-030-93620-4_12.

Villada Romero, José Luis (2023). *Desarrollo y optimización de componentes software para tareas administrativas de sistemas. IFCT0609*. Accessed May 19, 2025. ProQuest Ebook Central. Antequera: Bookwire GmbH. URL: https://ebookcentral.proquest.com/lib/bibliotecauma-ebooks/detail.action?docID=7051300.

Apéndices

APÉNDICE A

Anexo

A.1. Figuras

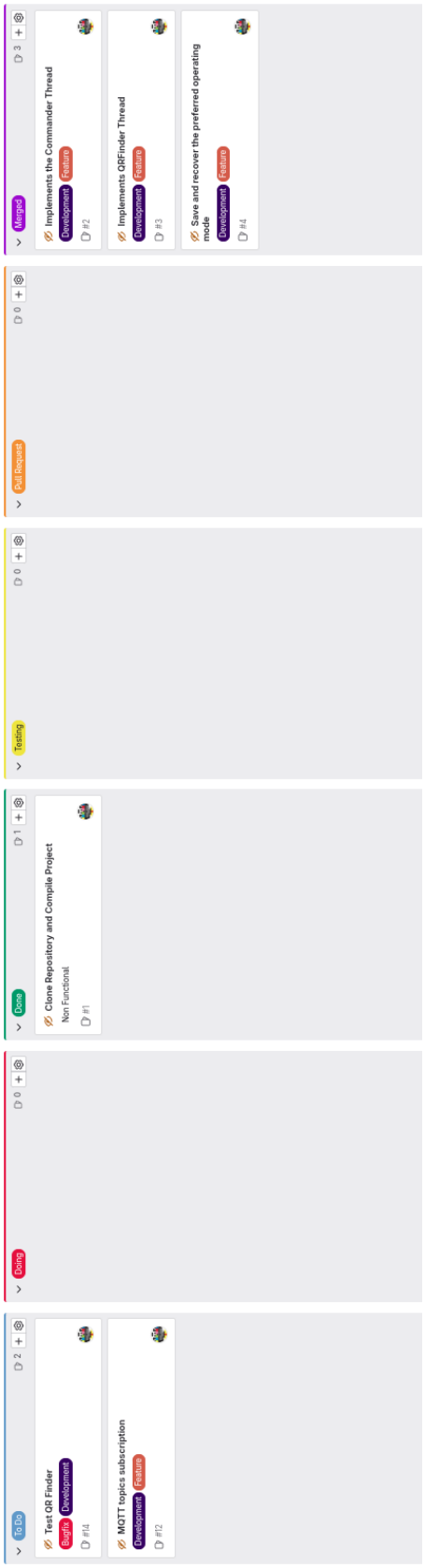


Figura A.1: Captura de tablero *Kanban* integrado en GitLab.

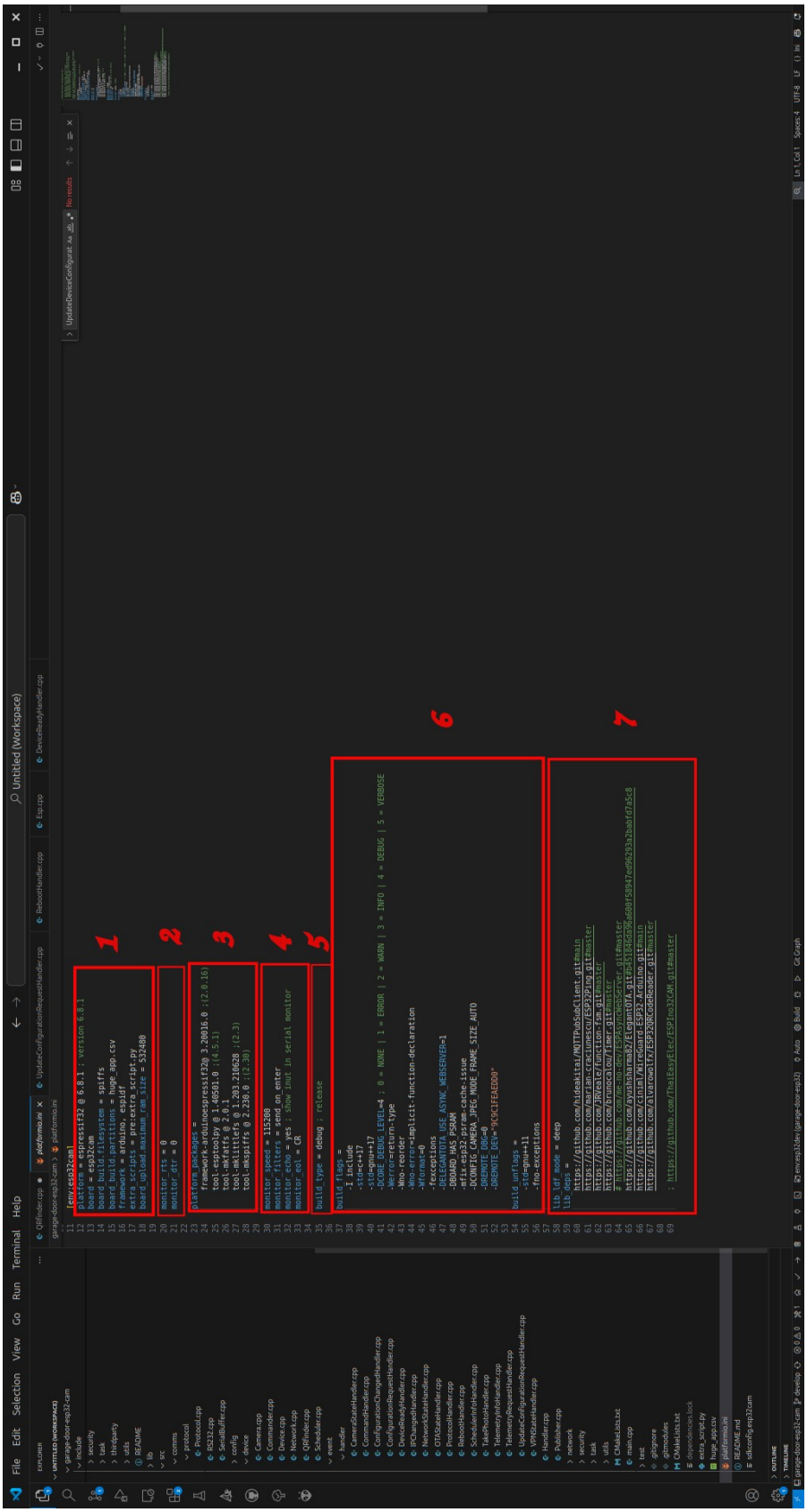


Figura A.2: Captura del contenido del archivo `platformio.ini`

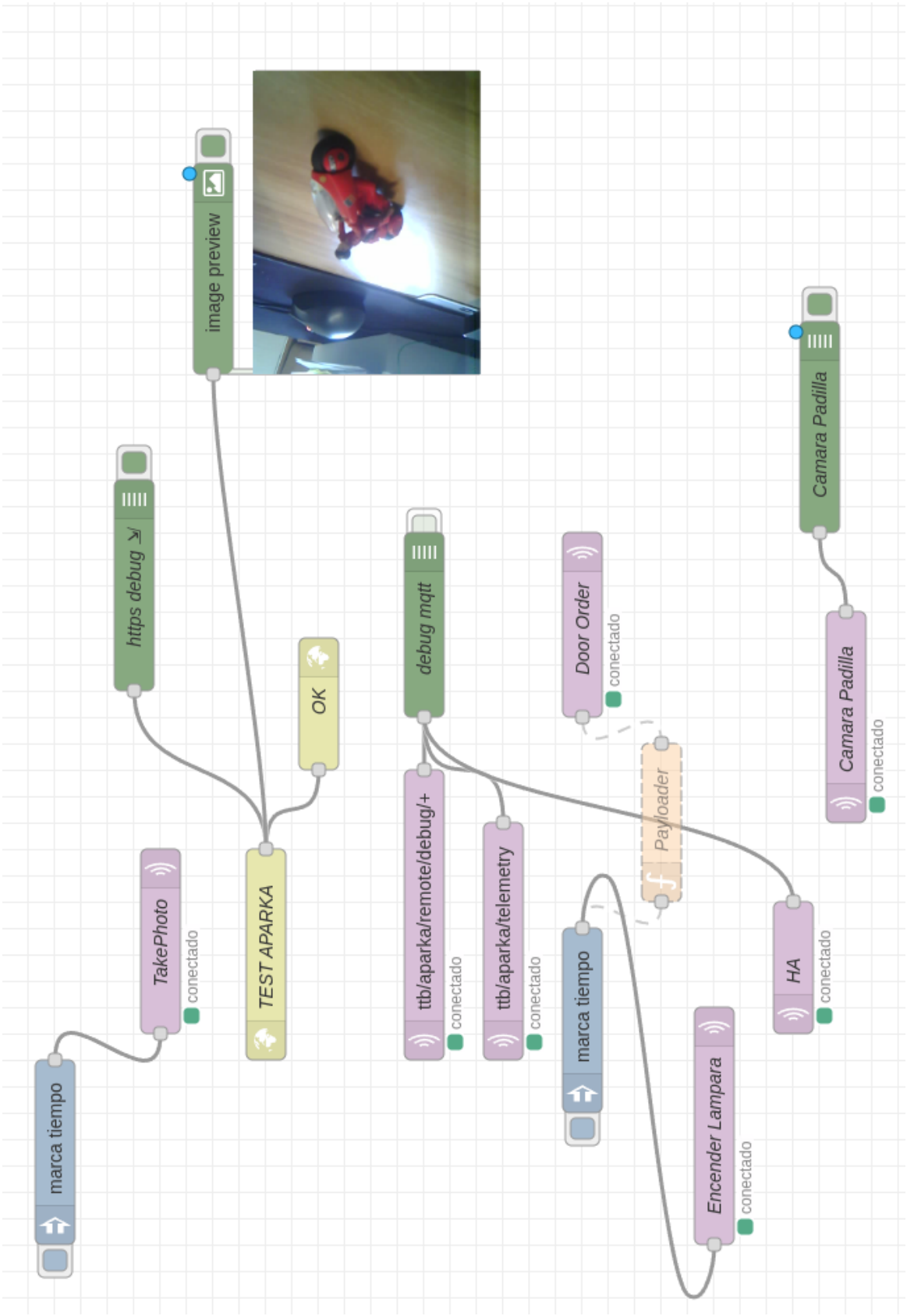


Figura A.3: Captura de panel principal de Node-RED mostrando la imagen capturada por el ESP32Cam

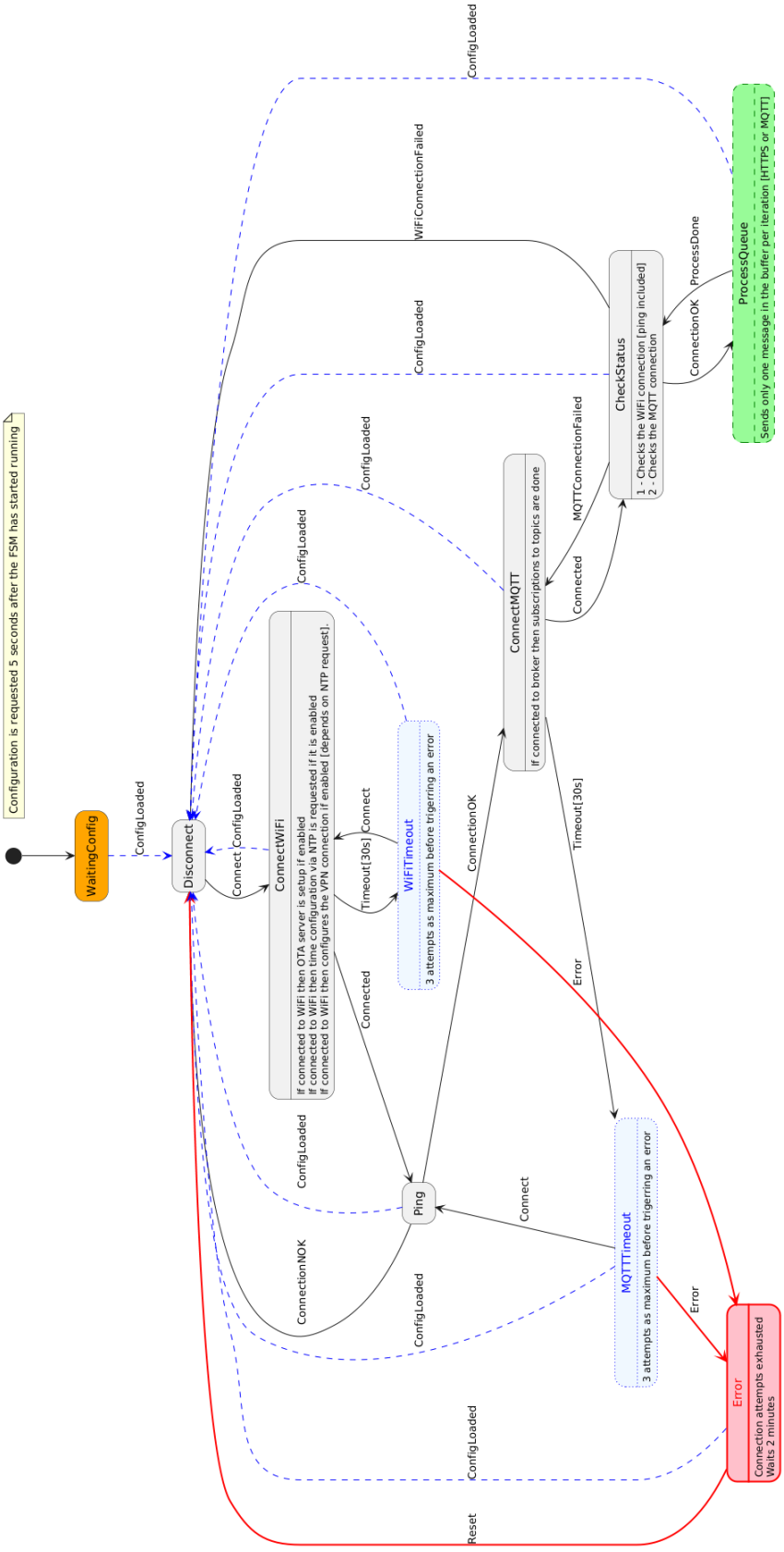


Figura A.4: Diagrama de estados dentro de la tarea *Network*

A.2. Diagramas de secuencia

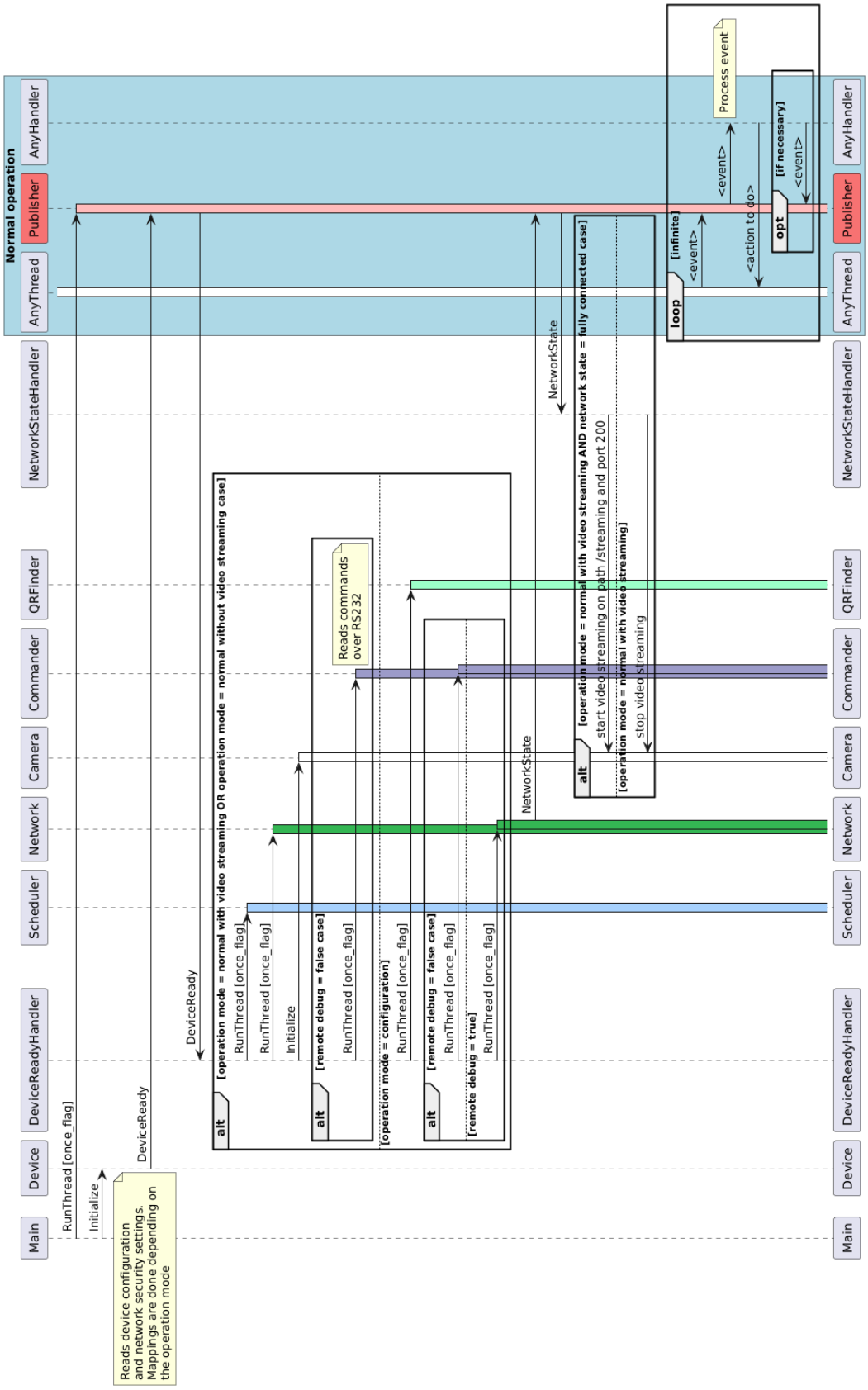


Figura A.5: Diagrama de puesta en marcha del dispositivo

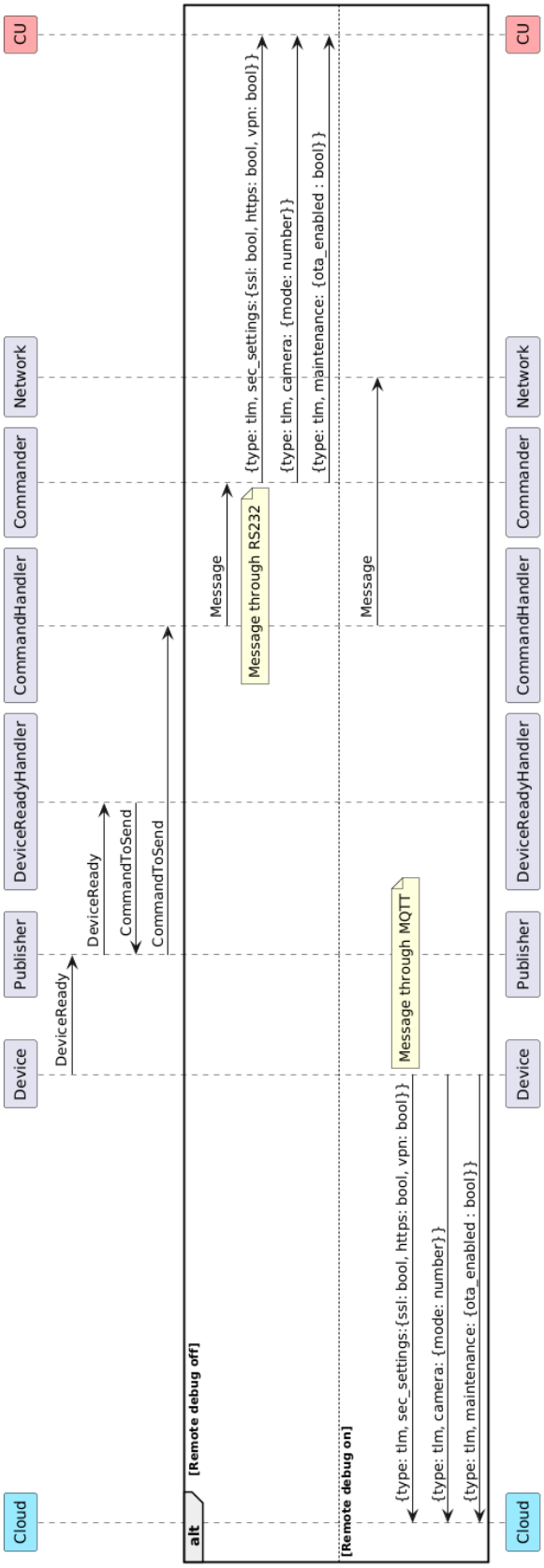


Figura A.6: Diagrama de secuencia de telemetría tras la notificación DeviceReady

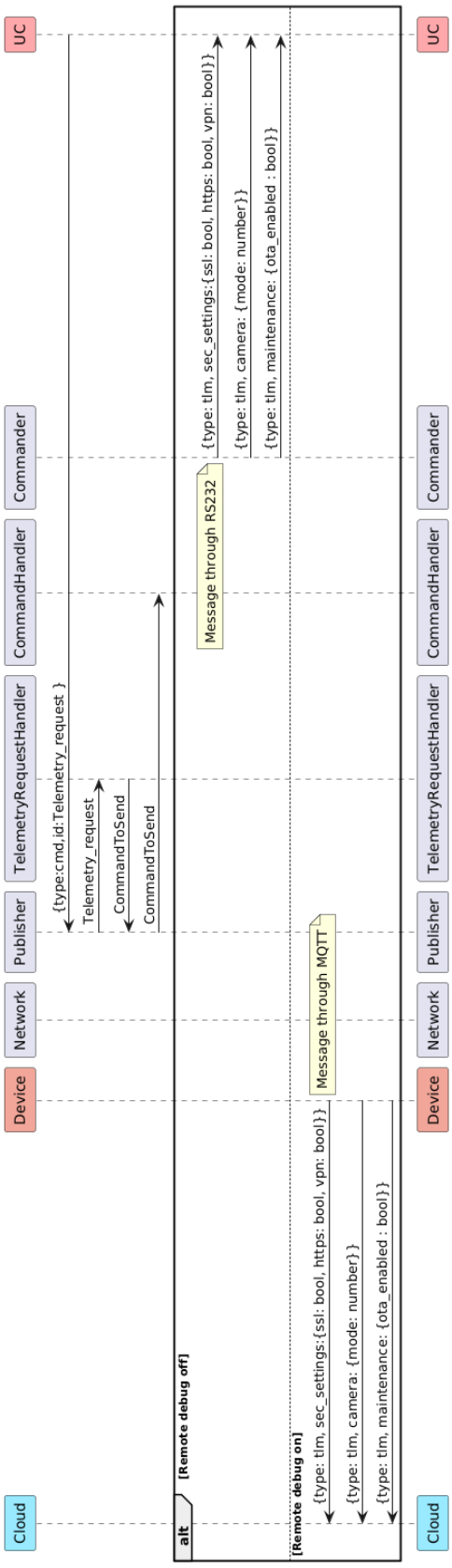


Figura A.7: Diagrama de secuencia de telemetría a demanda

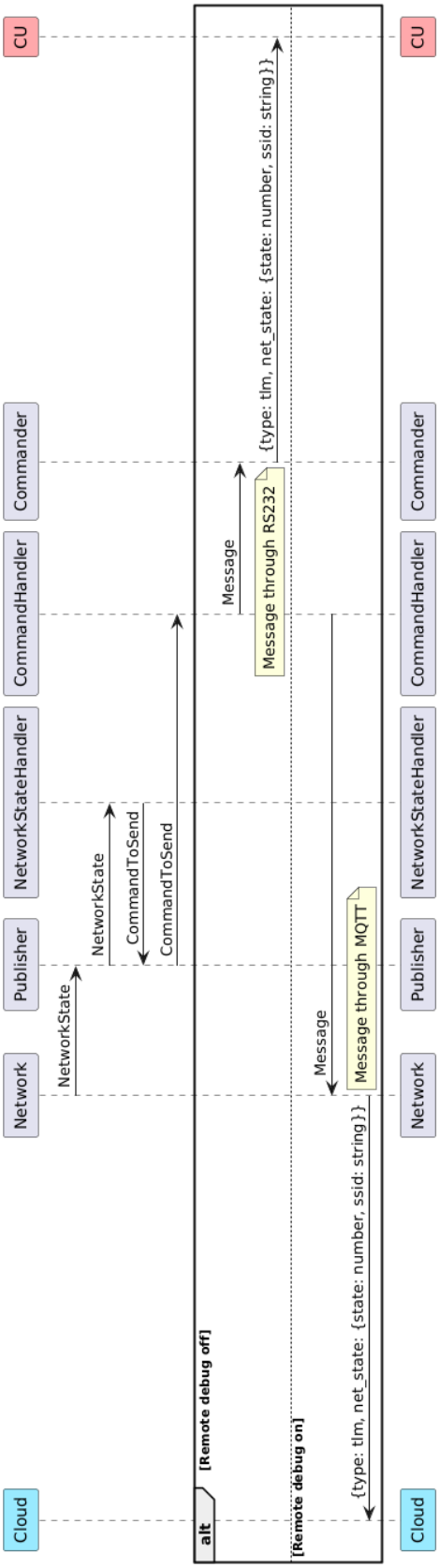


Figura A.8: Diagrama de secuencia de telemetría para cambios en el estado de la red

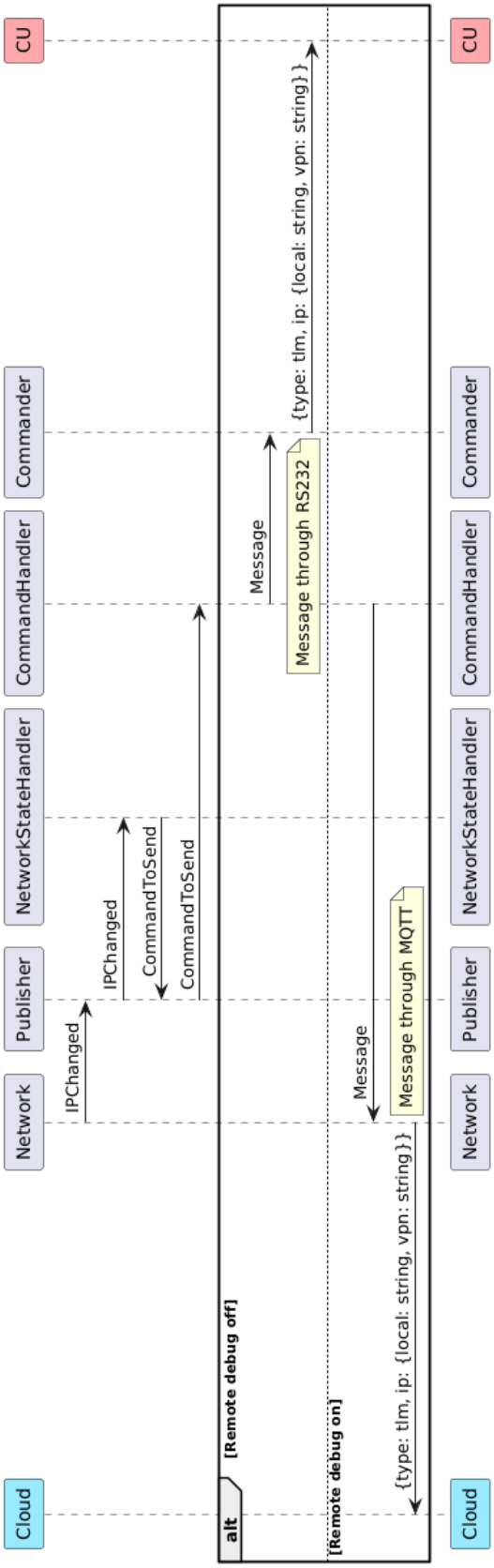


Figura A.9: Diagrama de secuencia de telemetría para cambios en dirección asignada de la red

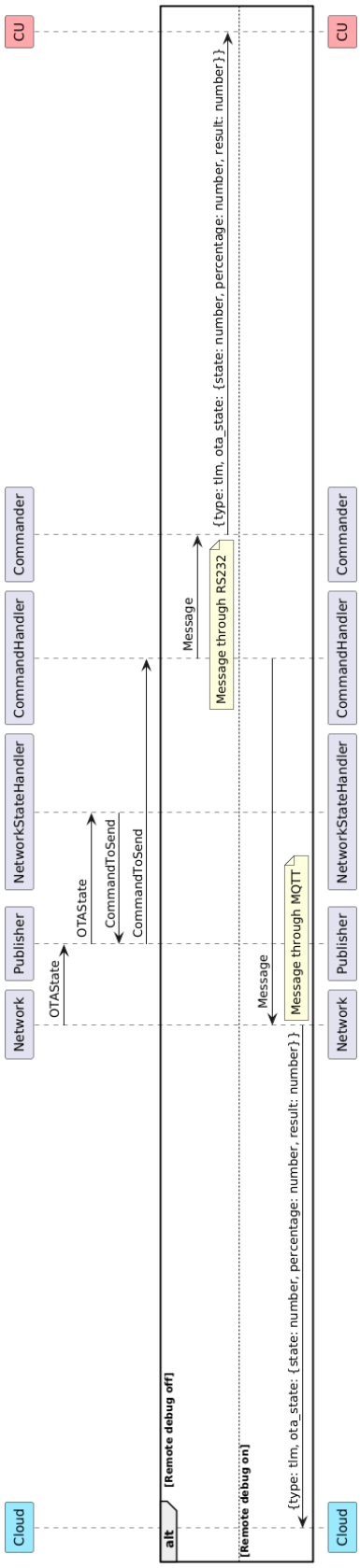


Figura A.10: Diagrama de secuencia de telemetría para monitorización de actualizaciones OTA

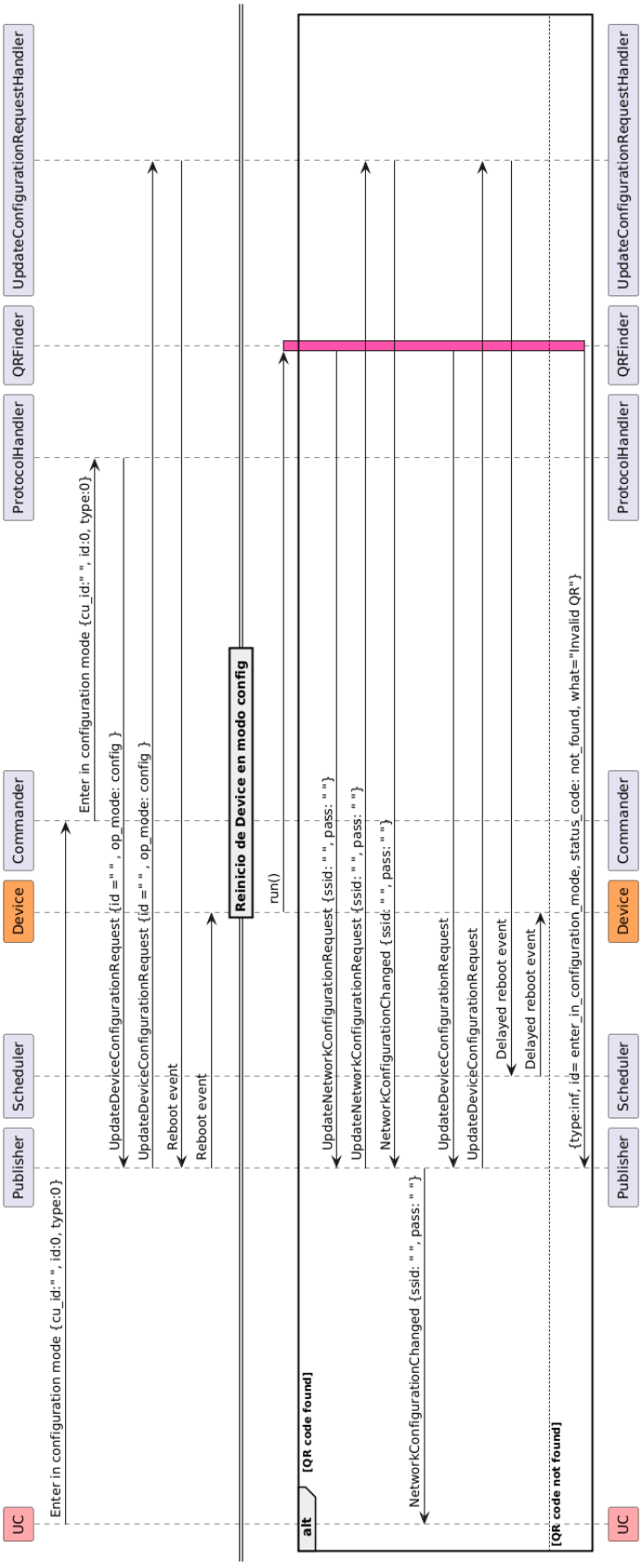


Figura A.11: Diagrama de secuencia para reinicio del dispositivo en modo configuración

A.3. Modelos 3D

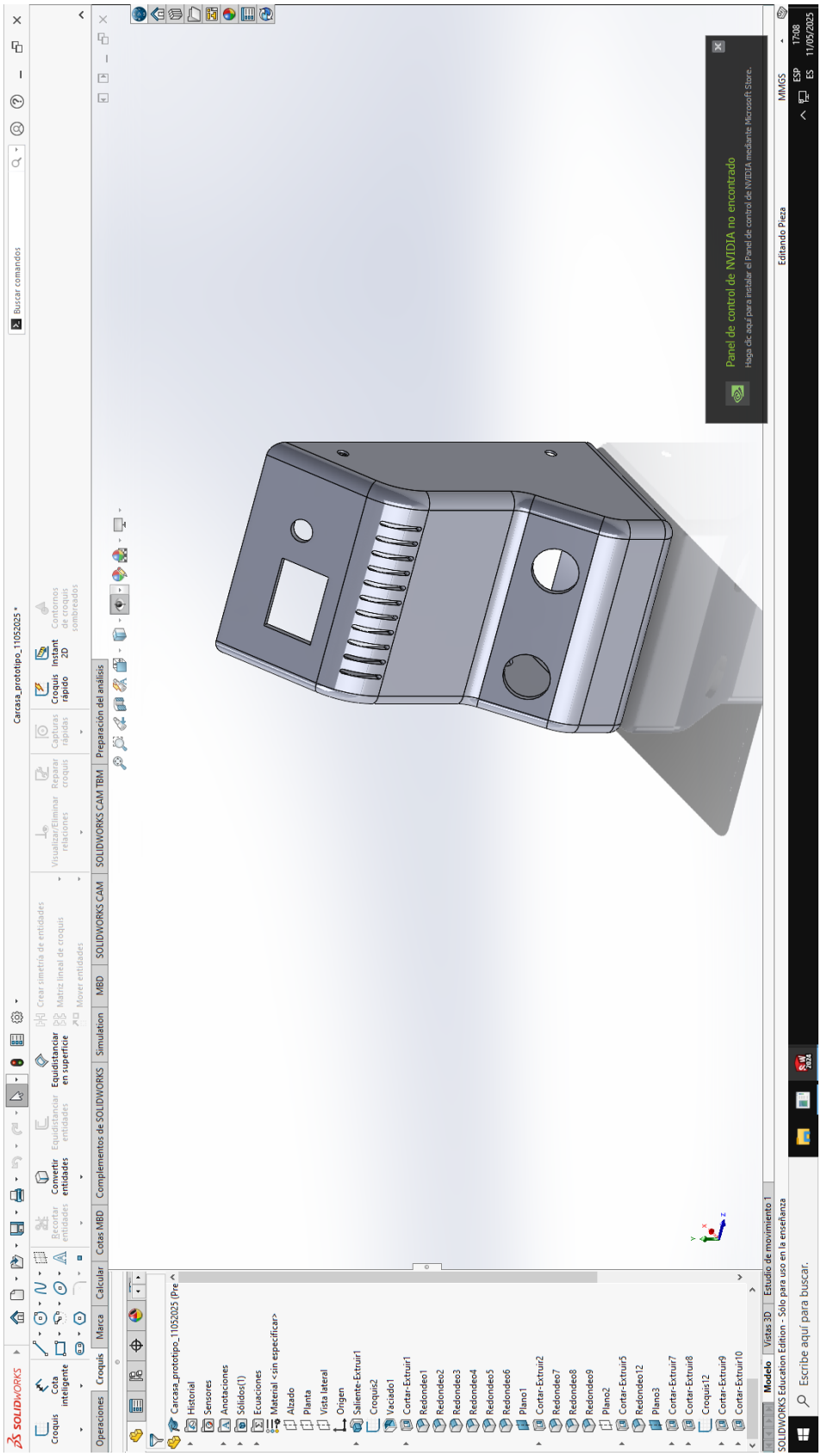


Figura A.12: Carcasa frontal del prototipo

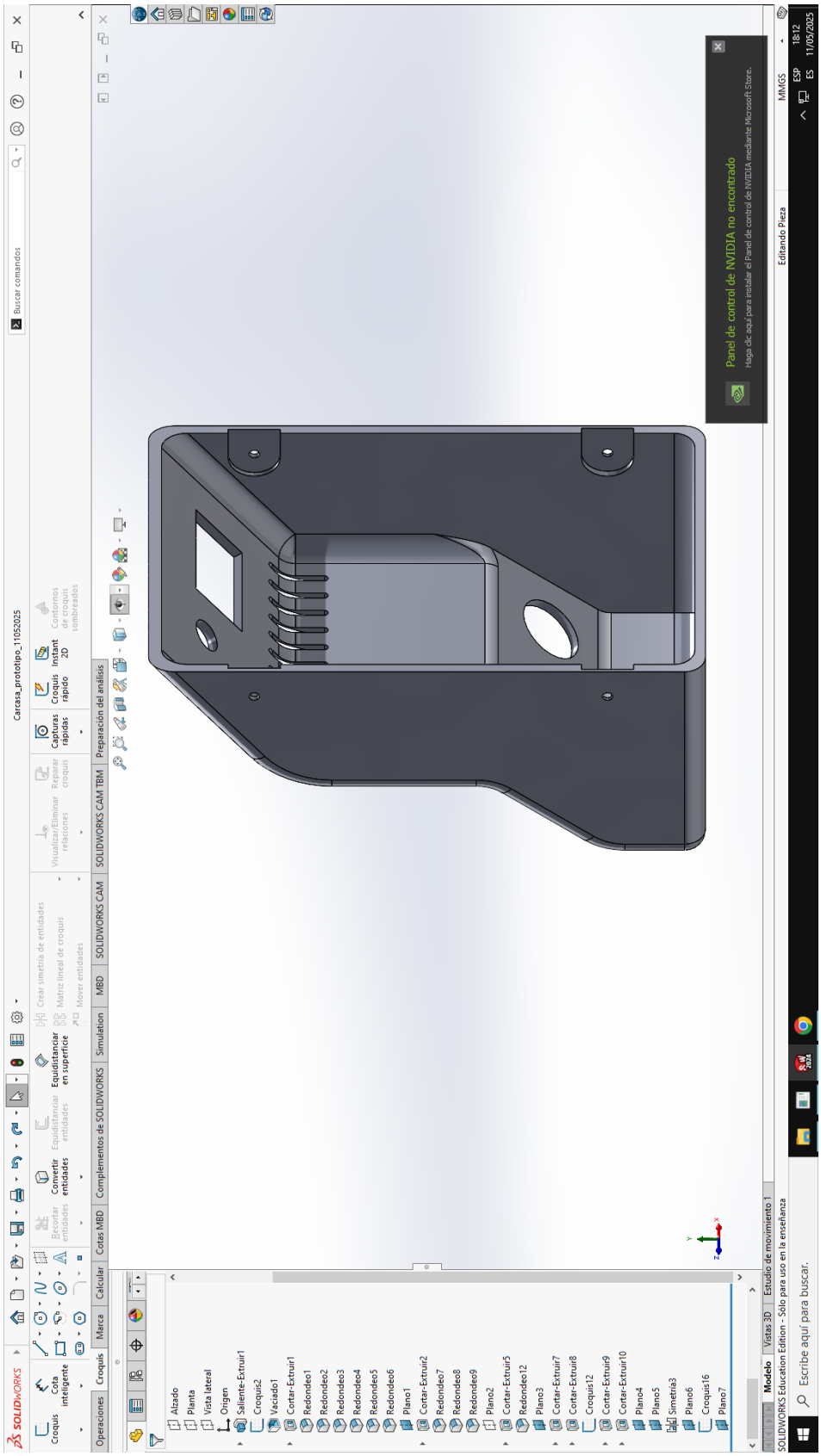


Figura A.13: Vista trasera de carcasa frontal del prototipo

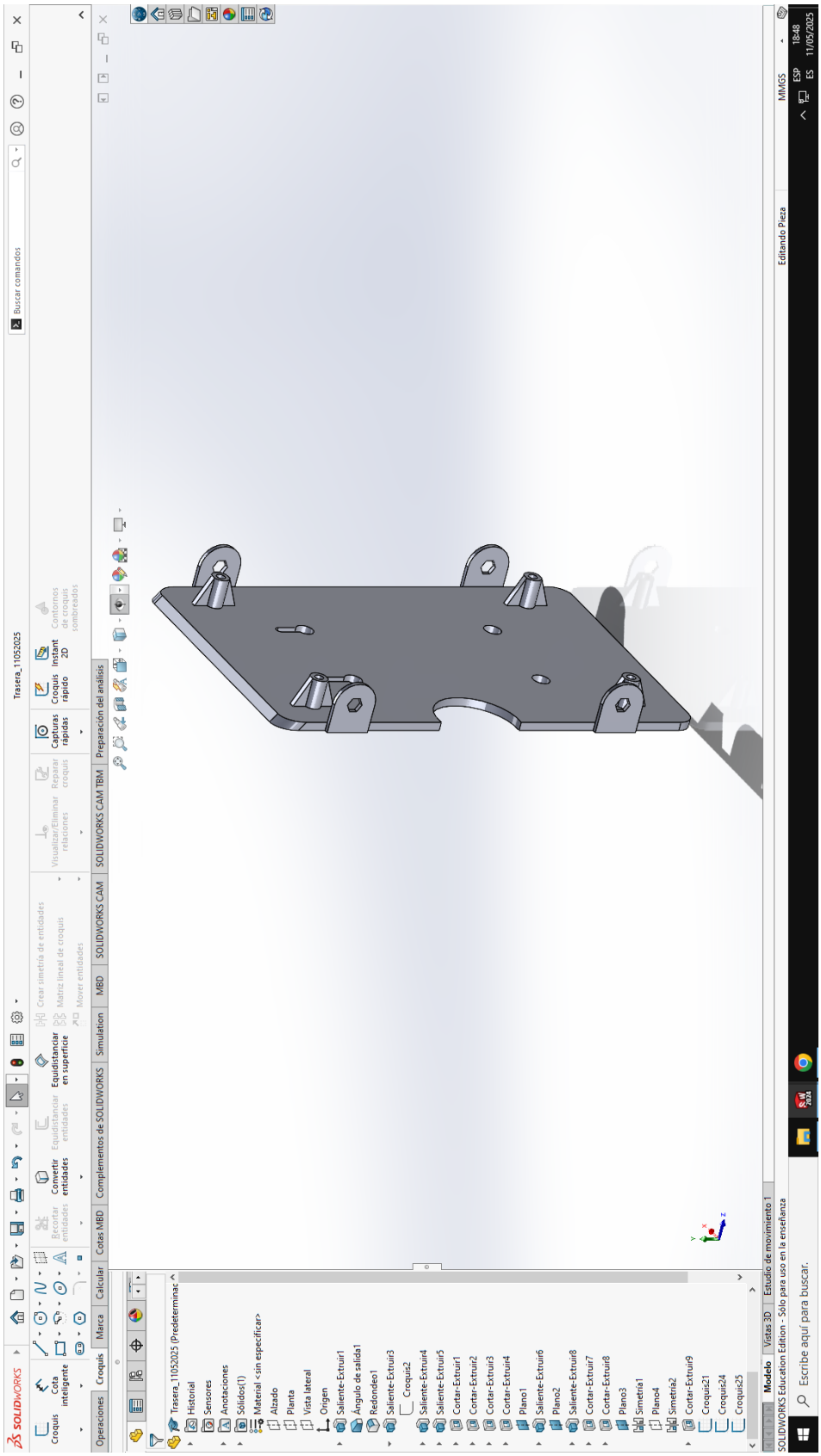


Figura A.14: Vista frontal de la tapa trasera del prototipo

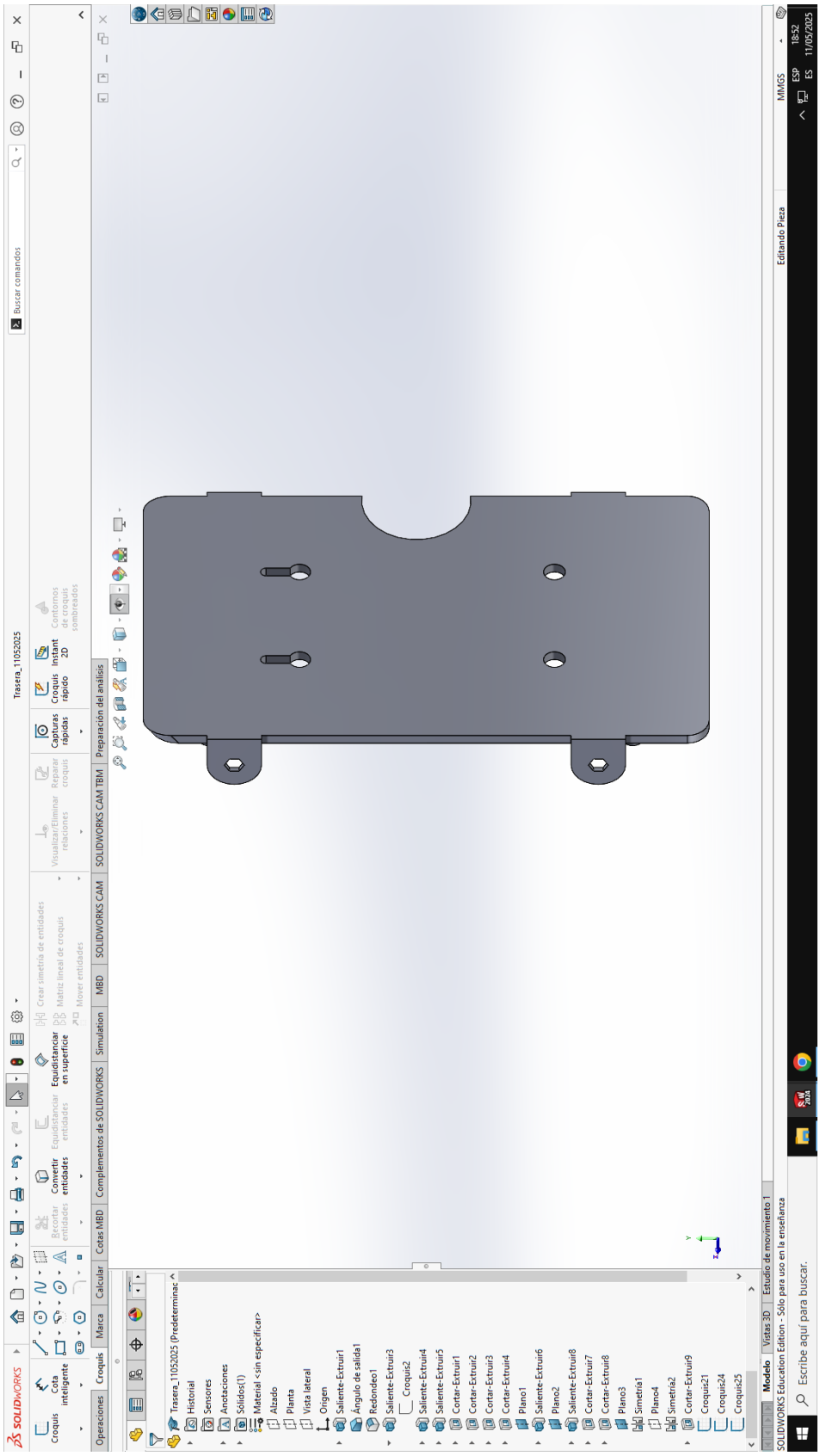


Figura A.15: Vista trasera de la tapa del prototipo

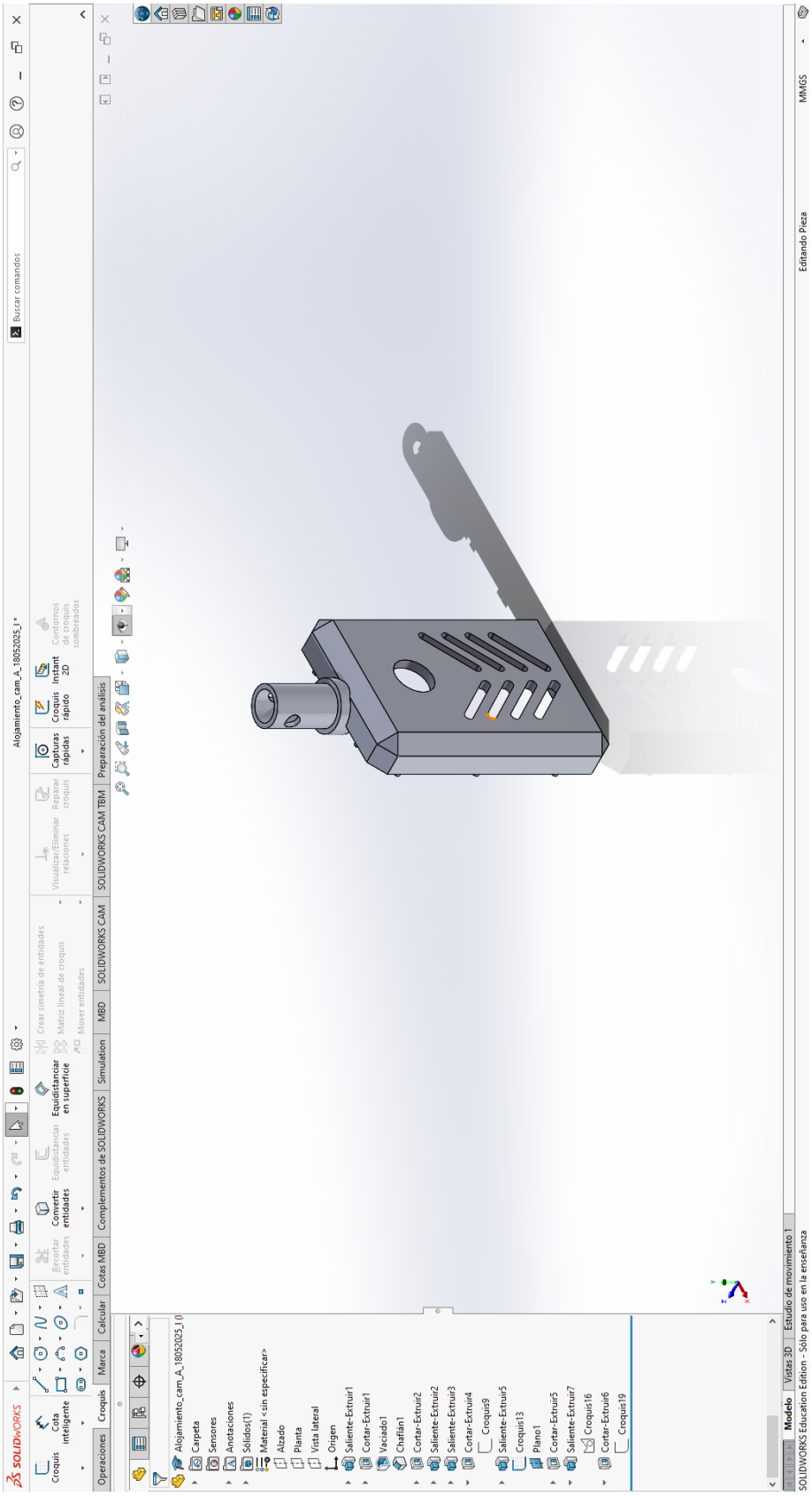


Figura A.16: Parte A del alojamiento para Esp32Cam

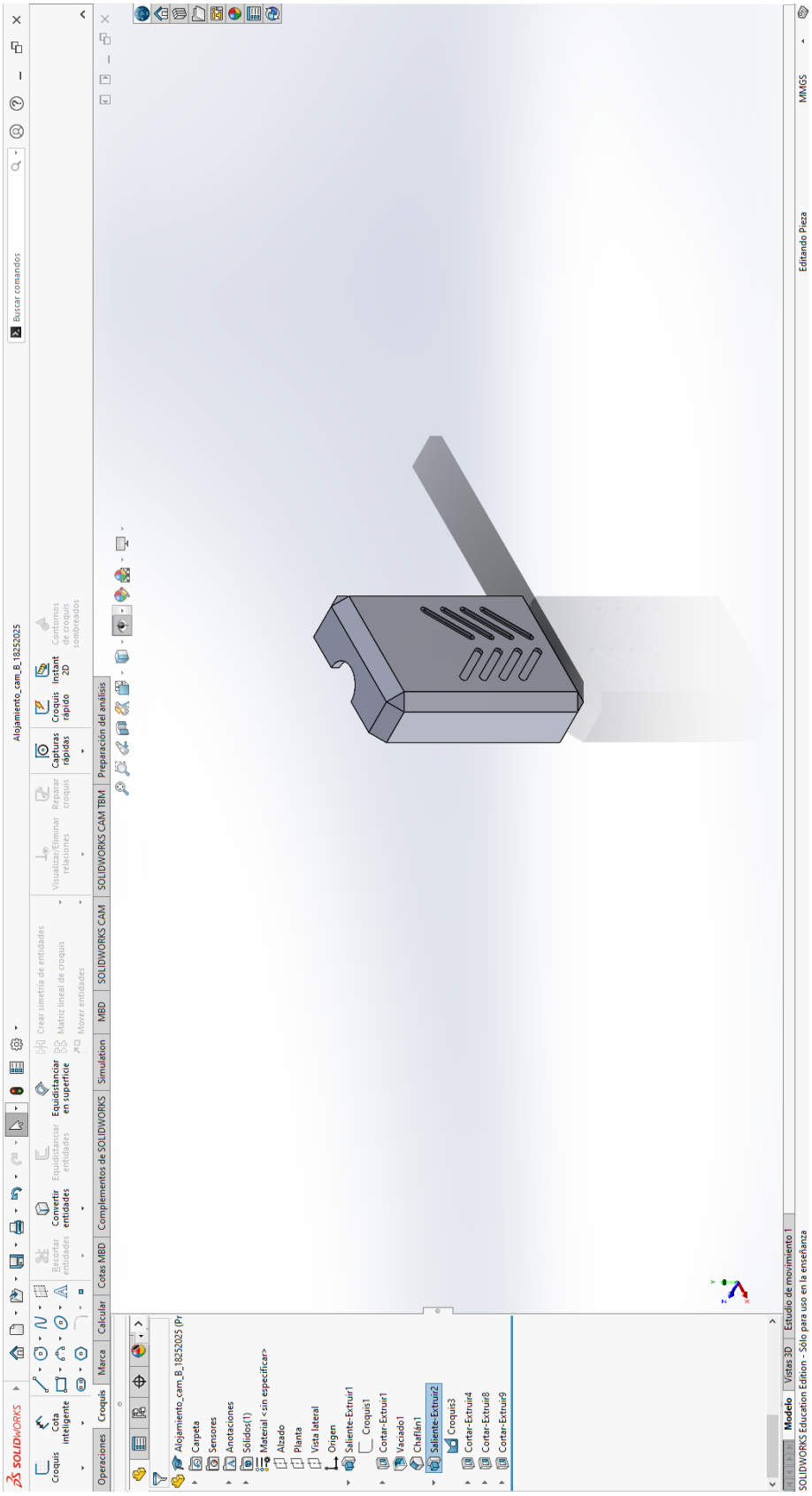


Figura A.17: Parte B del alojamiento para Esp32Cam

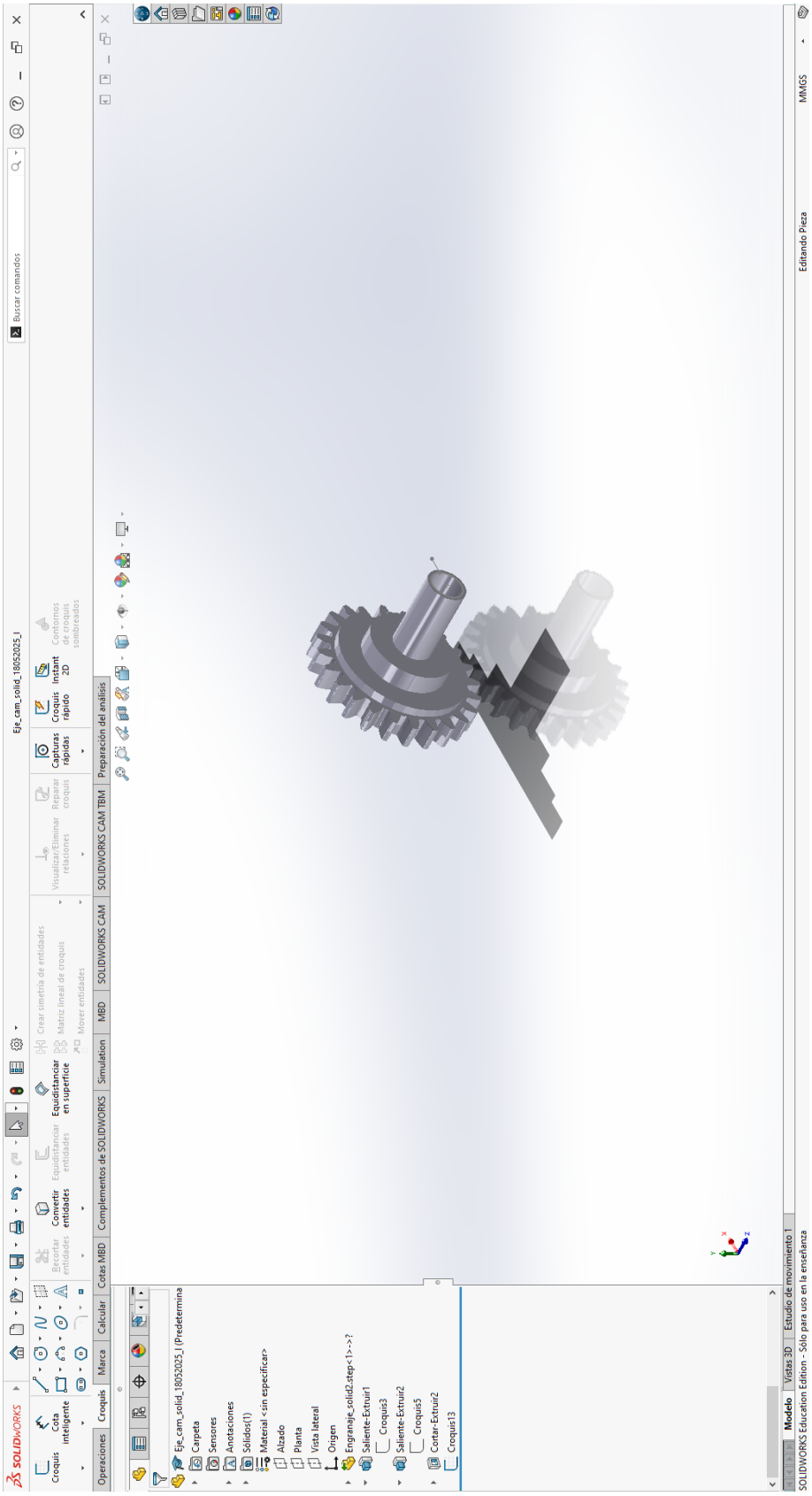


Figura A.18: Engranaje conector para compartimento de Esp32Cam

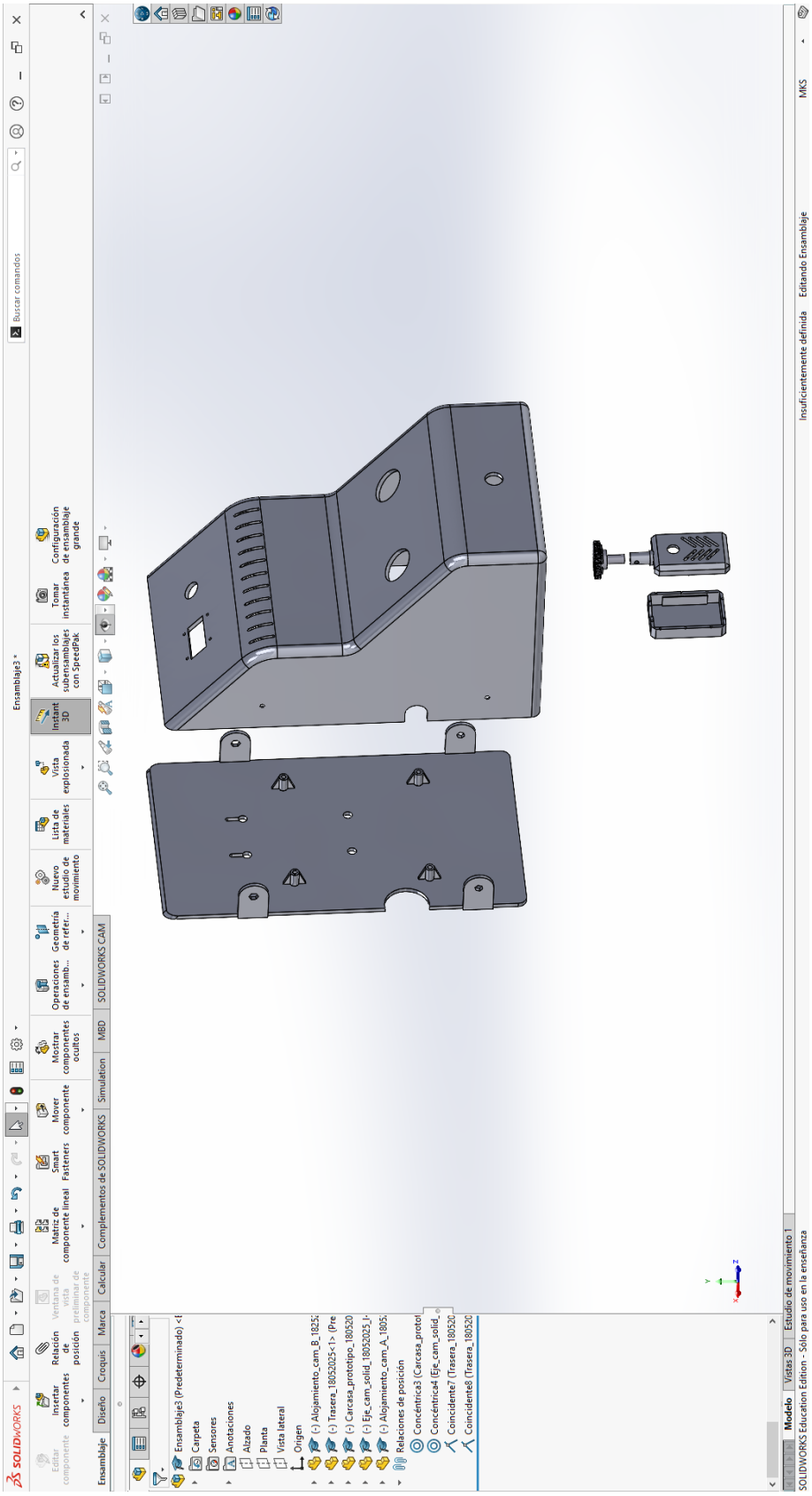
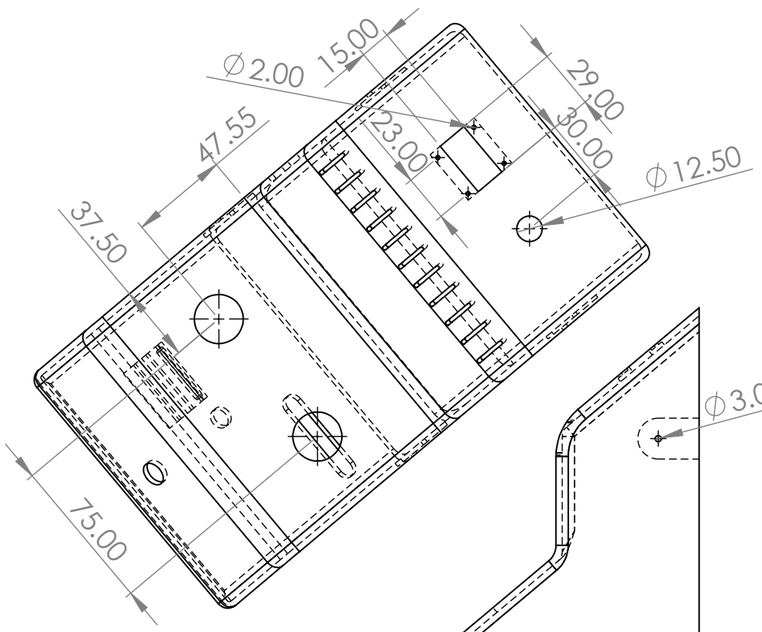
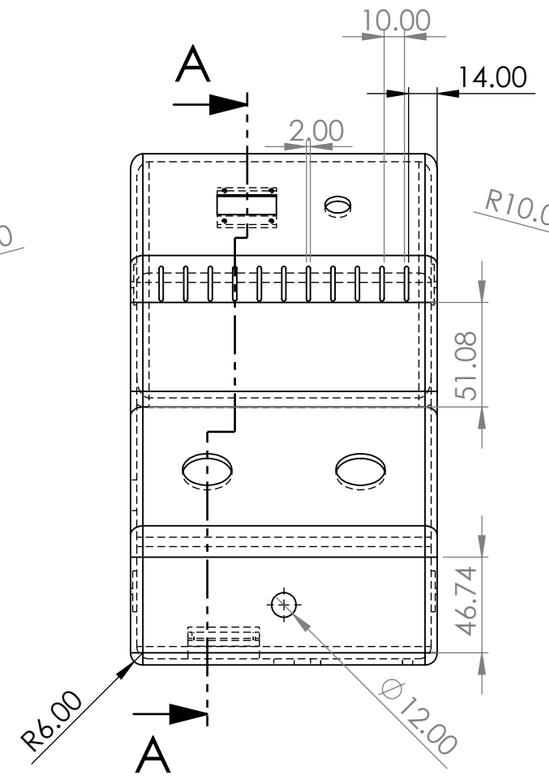


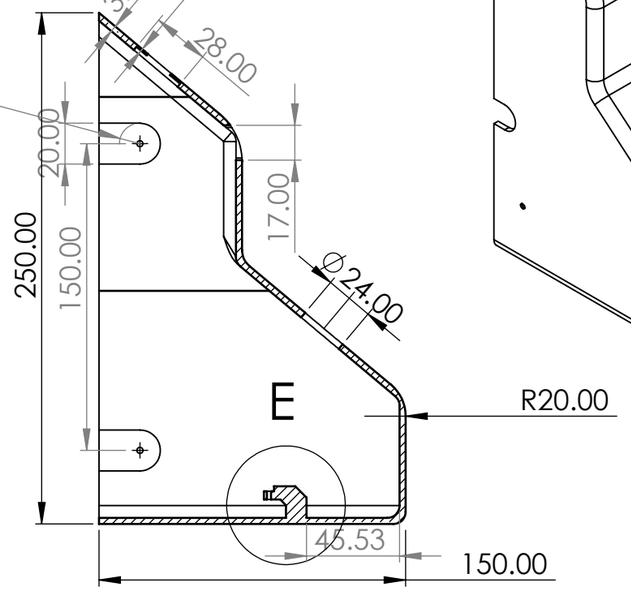
Figura A.19: Vista explosionada de todo el conjunto de modelos que forman parte de la carcasa

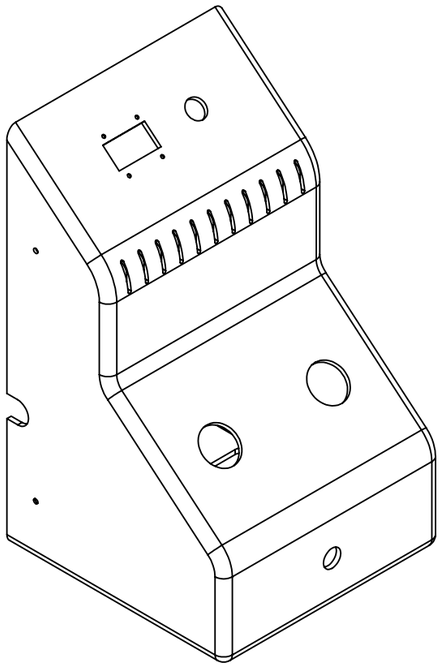
A.4. Planos normalizados de la carcasa



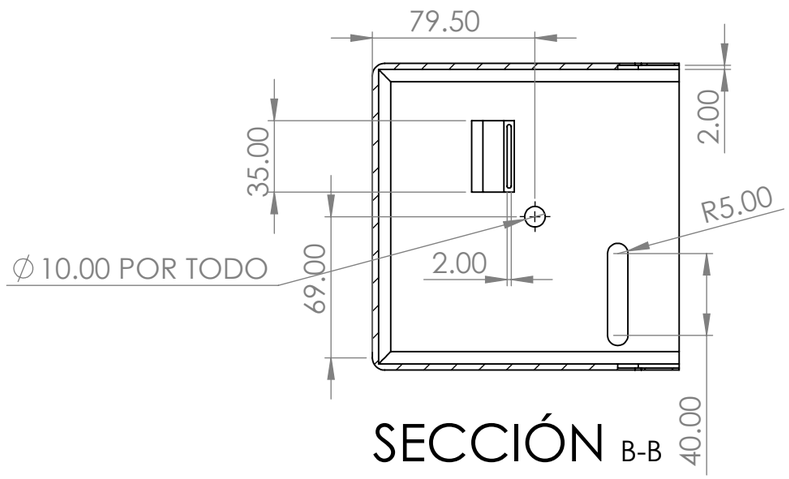
VISTA I
ESCALA 1 : 3



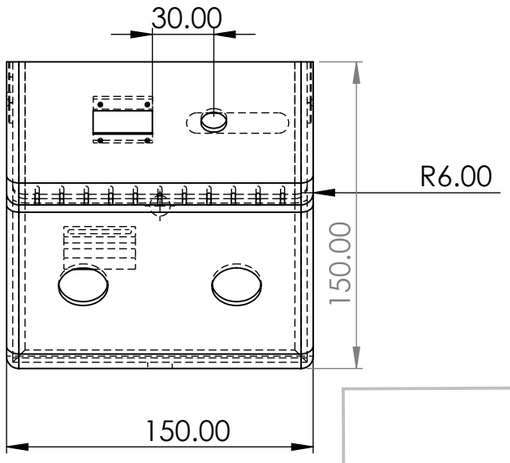


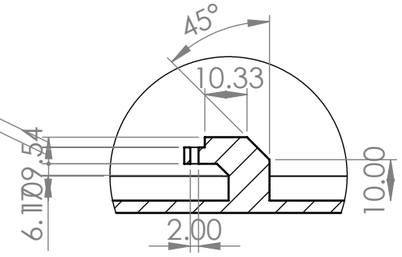


SECCIÓN A-A
ESCALA 1 : 3



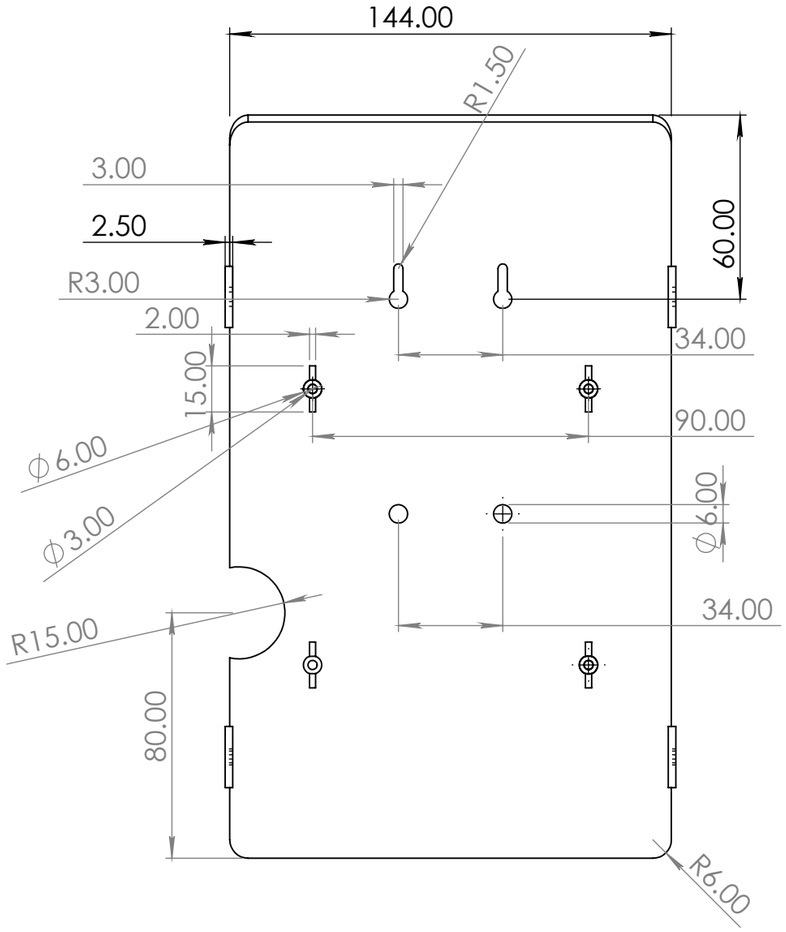
SECCIÓN B-B
ESCALA 1 : 3

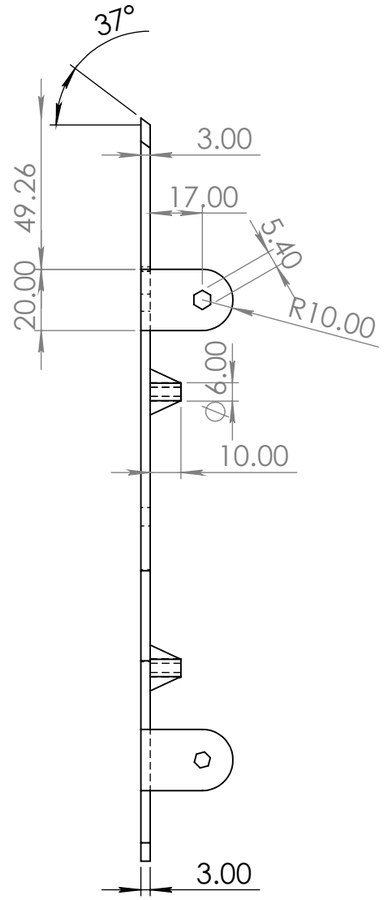


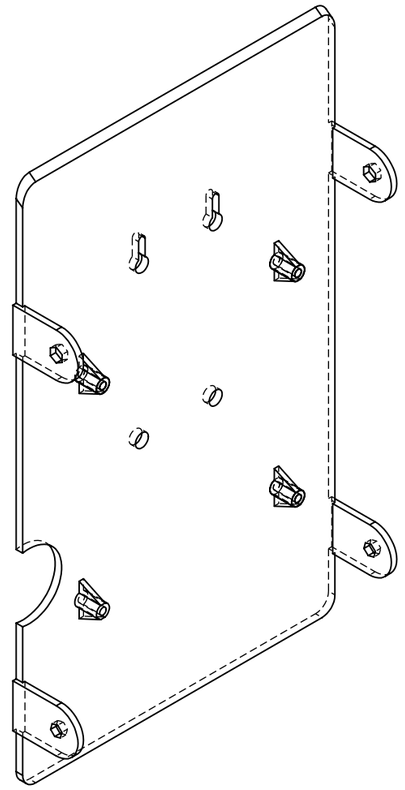


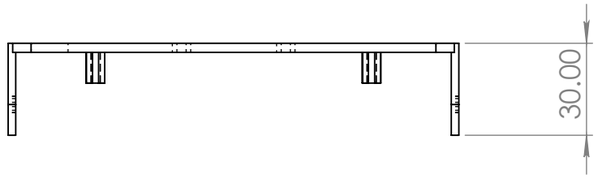
DETALLE E
ESCALA 2 : 3

		SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM				NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO:	
DIBUJ.	Alejandro Padilla		15/05/2025			Diseño de carcasa frontal	
VERIF.							
APROB.							
FABR.							
				MATERIAL: PLA		N.° DE DIBUJO : 1	A3
				PESO:		ESCALA:1:5	HOJA 1 DE 1

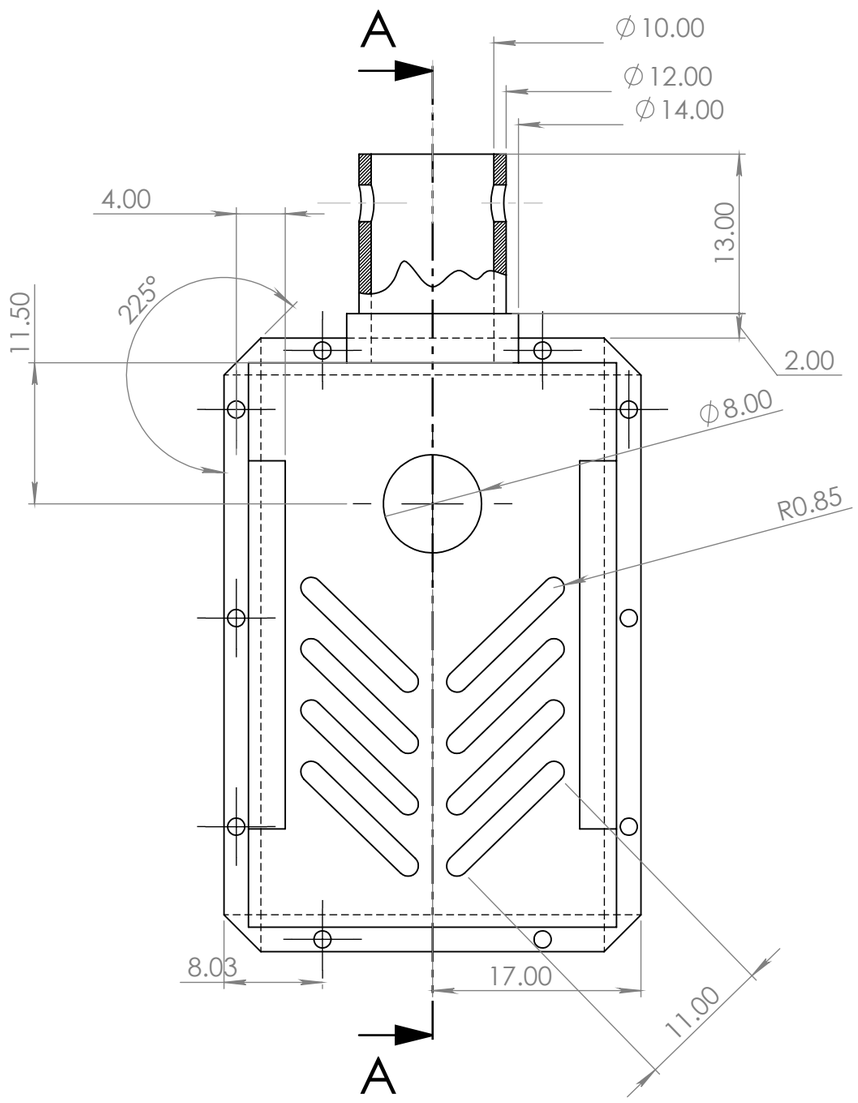


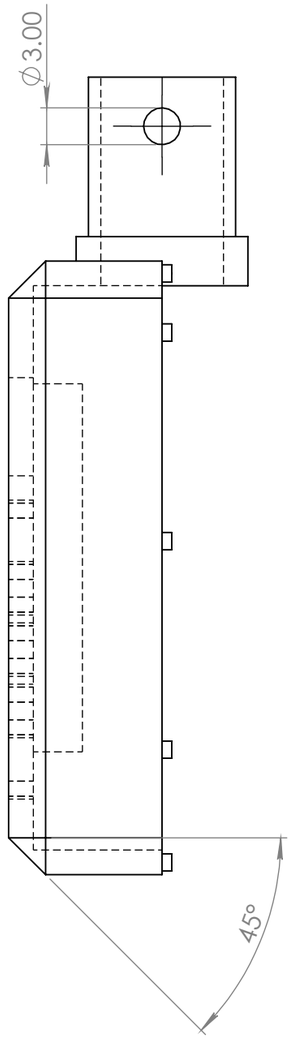


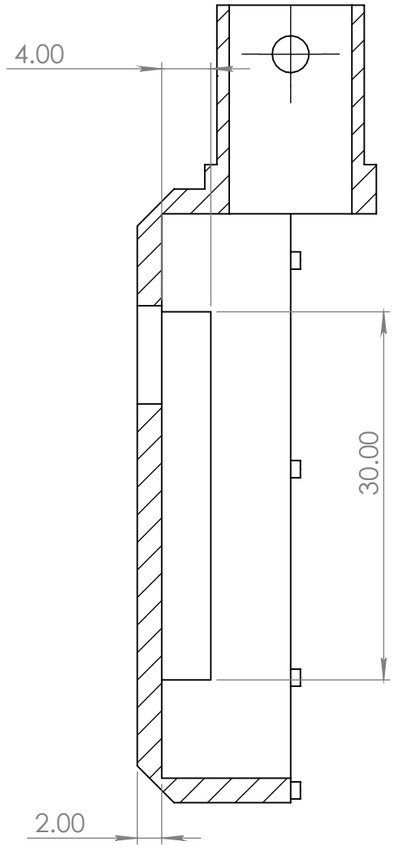




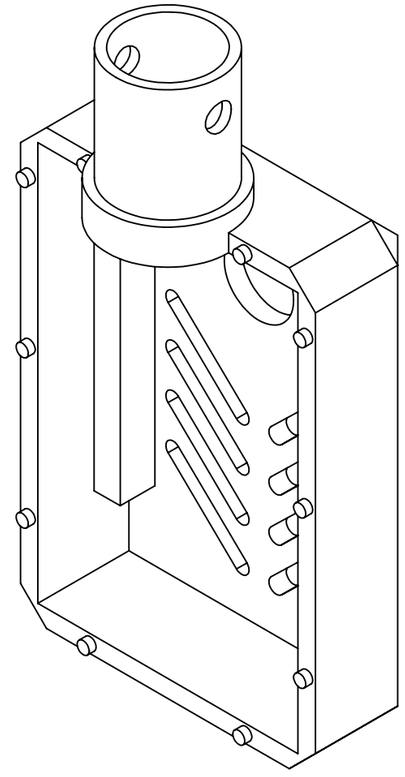
SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM	ACABADO:					NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO: Diseño de tapa trasera para carcasa	
DIBUJ.	Alejandro Padila		15/05/2025				
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: PLA		N.º DE DIBUJO 2	A3
				PESO:		ESCALA:1:2	HOJA 1 DE 1

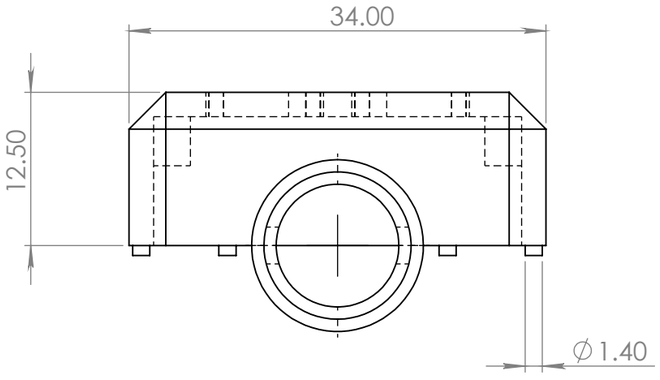




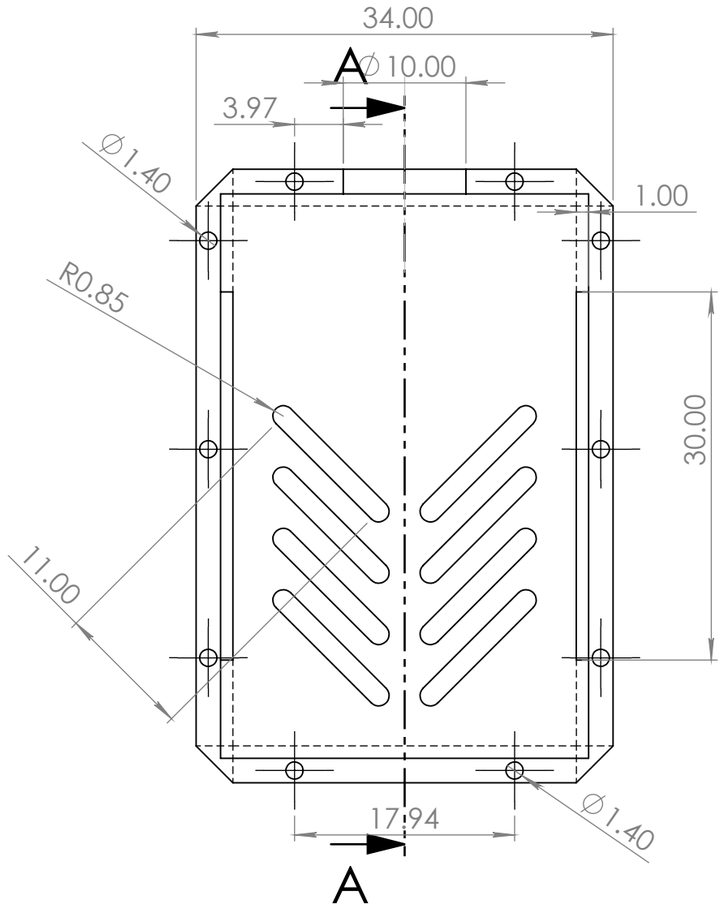


SECCIÓN A-A

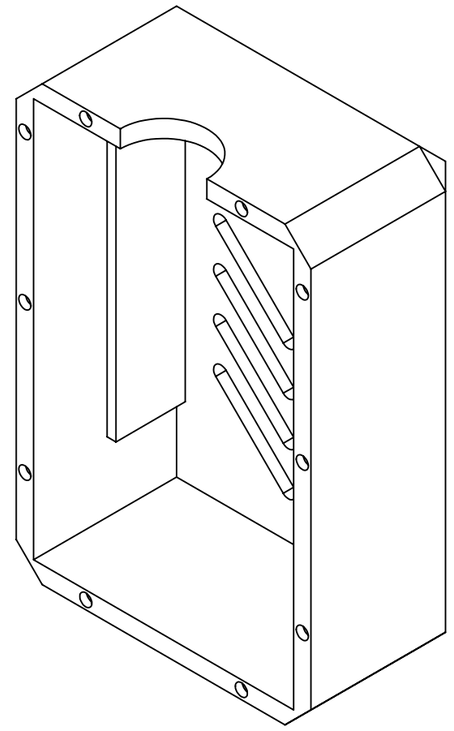


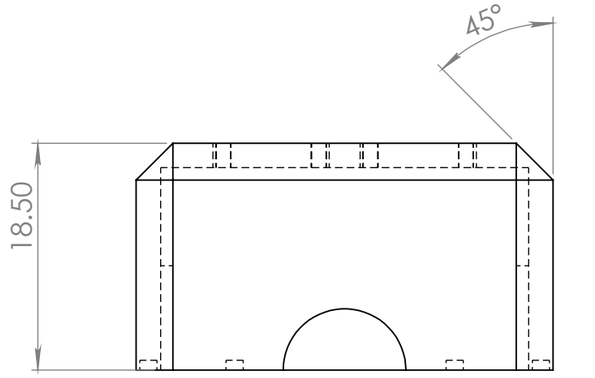


SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM	ACABADO:					NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO: Alojamiento para ESP32Cam - Pieza A	
DIBUJ.	Alejandro Padilla		12/05/2025				
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: PLA		N.° DE DIBUJO 3	A3
				PESO:		ESCALA:2:1	HOJA 1 DE 1



SECCIÓN A-A





SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM	ACABADO:					NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO: Alojamiento para ESP32Cam - Pieza B	
DIBUJ.	Alejandro Padilla		12/05/2025				
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: PLA		N.º DE DIBUJO 4	A3
				PESO:		ESCALA:2:1	HOJA 1 DE 1