Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería en Tecnologías Industriales

Lectura de códigos 2D empleando una cámara de un Arduino

Autor:

Bustos Jiménez, Fernando

Tutor:

Pablo Gómez, Santiago de

Departamento de Tecnología Electrónica

Valladolid, septiembre de 2024.

RESUMEN Y PALABRAS CLAVE

En este proyecto se ha implementado un lector de códigos QR utilizando una placa ESP32. El microcontrolador utiliza una cámara digital para captar las imágenes, las analiza en busca de QRs y transmite el mensaje decodificado al puerto serie de un ordenador personal.

Para ello, se han empleado librerías ya existentes en los entornos de desarrollo Arduino IDE y PlatformIO IDE. En esta memoria, se describen el código desarrollado, las herramientas empleadas y los resultados del funcionamiento del dispositivo.

Palabras clave:

- Arduino
- ESP32
- Código QR
- Cámara OV2640
- PlatformIO IDE

ABSTRACT AND KEYWORDS

In this project, a QR code reader has been implemented using an ESP32 board. The microcontroller uses a digital camera to capture images, analyzes them for QR codes, and transmits the decoded message to the serial port of a personal computer.

For this purpose, existing libraries in the Arduino IDE and PlatformIO IDE development environments were used. This document describes the developed code, the employed tools, and the results of the device's operation.

Key words:

- Arduino
- ESP32
- QR code
- OV2640 Camera
- PlatformIO IDE

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN Y OBJETIVOS

1.1 Introducción

El mundo de los microcontroladores, especialmente los basados en Arduino, ha transformado la electrónica moderna, abriendo puertas a la automatización y la conectividad de dispositivos en formas que hace no tanto parecían inalcanzables. Estos dispositivos ofrecen a profesionales y aficionados una solución eficiente, de bajo coste y adaptable a innumerables aplicaciones que abarcan desde la robótica y el Internet de las Cosas, hasta la automatización industrial. Todo esto los posiciona como una parte crucial en el ecosistema tecnológico actual.

Por otro lado, los códigos QR se han convertido en una herramienta omnipresente en la vida cotidiana, desde el acceso a información en productos y servicios hasta la integración en sistemas de pago y autenticación. Estos códigos, que permiten almacenar gran cantidad de información en un espacio reducido, han sido fundamentales para mejorar la interacción entre el mundo físico y digital. Gracias a su sencillez y capacidad de adaptarse a diversas situaciones, los QR son ahora un estándar global en sectores como la logística, el marketing y la seguridad.

Dentro de esto, la intersección entre estos dos ámbitos es particularmente relevante. La capacidad de placas como las ESP32 para procesar imágenes de forma rápida y mediante instrucciones fáciles de programar es un avance significativo. Además, a medida que estos dispositivos se vuelven más pequeños y potentes, la capacidad de integrar estas dos tecnologías amplía las posibilidades para soluciones innovadoras.

Así, en el ámbito de la electrónica, microcontroladores y códigos QR representan la convergencia entre hardware y software de una manera accesible y poderosa. Para estudiantes, ingenieros y entusiastas, estas herramientas permiten desarrollar proyectos que responden a problemas reales y, a la vez, abren la puerta a la innovación.

En este marco, el objetivo de este proyecto será programar un microcontrolador ESP32 para detectar y decodificar códigos QR en las imágenes captadas por la cámara digital con la que está equipado.

A nivel práctico, un dispositivo como este podría tener muchas aplicaciones. En primer lugar, dado su bajo coste, versatilidad y robustez, podría ser integrado a nivel industrial facilitando la comunicación entre las estaciones de una cadena de montaje, o como parte del control de inventarios de una empresa. Y es que a día de hoy, son muchas las empresas que utilizan

códigos QR u otros símbolos bidimensionales para etiquetar sus productos, las materias primas y otros elementos del proceso de producción.

Incluso, teniendo en cuenta la seguridad de sus datos y la dificultad para la manipulación externa de las instrucciones y los resultados, este dispositivo podría formar parte de un sistema de recuento electoral digital y automatizado. Este lector tan solo sería el primer paso del proceso, leyendo las papeletas en forma de código bidimensional, pero aseguraría una conexión segura entre el votante y un ordenador en el que se pudiera almacenar y encriptar su voto.

En cuanto al transcurso de esta memoria, en primer lugar se tratarán de explicar los conceptos más relevantes del marco teórico que rodea al proyecto. Después, se explicará el desarrollo seguido hasta la consecución de un código final que cumpla con los objetivos propuestos en el siguiente apartado de esta introducción. Por último, se validará su funcionamiento y adecuación a diferentes aplicaciones, seguido de una reflexión final sobre el trabajo realizado, en forma de conclusión.

1.2 Objetivos

El objetivo final de este proyecto es programar un microcontrolador ESP32 para capturar imágenes, reconocer en ellas la presencia de códigos bidimensionales de tipo QR, leerlos corrigiendo sus errores, y enviar el resultado a un ordenador personal a través de conexión serie con un cable USB.

En cuanto a objetivos intermedios y específicos, el primero de ellos será establecer una conexión funcional entre la cámara, el ESP32 y el ordenador. Después de eso, se aprenderá a manipular la cámara y las imágenes capturadas, entendiendo cómo funcionan parámetros clave de la fotografía digital como son la resolución y el formato de un fotograma. Tras esto, se buscará y documentará la librería adecuada para trabajar con códigos QR en un entorno como el propuesto.

También, se pueden definir como objetivos específicos comprobar cómo ciertas variables de los códigos QR afectan a su reconocimiento. Así, se harán pruebas para medir la velocidad del programa ante diferentes formatos de código, se comprobará el tamaño mínimo de un QR legible a distintas distancias y se trabajará con los diferentes protocolos de corrección de errores.

Por último, quizás el más importante es el que se podría llamar objetivo intrínseco del proyecto, que sería familiarizarse con las herramientas, metodologías y recursos que en él se van a emplear. Así, se aprenderá a trabajar en el ámbito de las placas Arduino, siendo este un mundo de gran aplicabilidad e importancia presente y futura a nivel global y con un grandísimo valor didáctico en la rama de Electrónica. Todo esto aplicado al análisis de imágenes y al dominio de los códigos QR, siendo estas otras áreas en constante desarrollo y expansión.

Además, en concreto la rama de Electrónica no es algo que se trate en gran profundidad ni de forma muy práctica en el grado de Ingeniería en Tecnologías Industriales, por lo que casi todos los conocimientos aplicados en este proyecto serán adquiridos durante su desarrollo. Estas experiencias se irán mostrando en los sucesivos apartados de este documento.

2 ESTADO DE LA TÉCNICA

En este apartado, se revisará el estado de la técnica relativo al proyecto y se tratará de desarrollar los conocimientos teóricos necesarios para la comprensión y seguimiento del mismo. De esta forma, se podrá conocer y entender las herramientas empleadas para la consecución de los objetivos descritos en el apartado anterior.

Estas herramientas serán tanto el hardware utilizado, como el entorno de programación, lenguaje y demás elementos de software empleados. Se irá en un orden creciente de complejidad y concreción para poder llegar a entender las decisiones tomadas en cuanto a aspectos técnicos.

2.1 Introducción a Arduino

Arduino es una plataforma de desarrollo libre que facilita el uso y la expansión de la electrónica en sus dos caras: hardware y software. Fue creada en 2005 como parte de un proyecto para estudiantes del *Interaction Design Institute Ivrea* (IDII), de Ivrea (Italia), pero su impacto fue mucho más allá de eso. A día de hoy, la compañía está dirigida tanto a un público profesional como aficionado, teniendo una importante comunidad internacional que la rodea en los dos ámbitos.

Además, es importante reseñar que los productos de la compañía Arduino son considerados como hardware y software libre, bajo la Licencia Pública General de GNU (GPL) y la Licencia Pública General Reducida de GNU (LGPL). Esto permite la manufactura y distribución de productos Arduino por cualquier individuo.

Comúnmente, al referirse a Arduino, se puede hablar de dos partes principales. Por un lado, la placa Arduino es una pieza de hardware con diseño modular, cuyo componente fundamental es un microcontrolador. A este microcontrolador se le pueden añadir toda una gama de componentes electrónicos en función del propósito del proyecto, como por ejemplo sensores, displays o dispositivos de transmisión de datos.

Por otra parte, Arduino IDE (*Integrated Development Environment*) es el entorno de desarrollo de software oficial de la plataforma Arduino. En sí, se trata de una aplicación adaptada a múltiples sistemas operativos (Windows, macOS, Linux) que ofrece una serie de herramientas de software que permiten escribir, depurar, editar y grabar un programa (llamado *sketch*) a nuestra placa.

Arduino IDE no es el único entorno de desarrollo disponible para trabajar con microcontroladores. También existen otros que ofrecen características diferentes con las que se adaptan a todo tipo de proyectos. Uno de los IDE más conocidos es PlatformIO IDE, herramienta de gran potencia dada su robustez y flexibilidad. En sí, PlatformIO es una extensión que se integra con editores de código como el desarrollado por Microsoft: Visual Studio Code (VS Code). Este es un editor de código fuente ligero y altamente personalizable, que junto a PlatformIO, ofrece un entorno de desarrollo profesional con características avanzadas y multitud de herramientas a disposición del usuario.

En este proyecto, los entornos de desarrollo utilizados serán tanto Arduino IDE como PlatformIO. Mientras que el primero es ideal para principiantes debido a su simplicidad y facilidad de uso, según se avance con el desarrollo del código se harán necesarias las funcionalidades avanzadas del segundo.

En cualquier caso, al escribir un programa en cualquiera de estos IDE, se utilizará el mismo lenguaje de programación. Este es un lenguaje simple basado en C++, y expandible mediante librerías adicionales. Las librerías en Arduino son colecciones de código que proporcionan funcionalidades específicas, permitiendo a los usuarios reutilizar código sin tener que reescribirlo desde cero. Así, se simplifican tareas complejas y se permite a los desarrolladores centrarse en la lógica de alto nivel de sus proyectos. La utilización de librerías es una parte importante de este trabajo y se explicará con más detalle en apartados posteriores.

Una vez escrito el código, el entorno de desarrollo utiliza un compilador para verificar y transformar el código. Así es como el programa se traduce del lenguaje de alto nivel en el que se desarrolla, al código binario que el procesador entiende. Después, el código es subido al microprocesador de la placa, que lo almacena en memoria y ejecuta.

Como se puede ver, el IDE es una gran instrumento porque hace mucho más accesible la programación de microcontroladores, enmascarando las partes más complejas y tediosas de la programación en bajo nivel. Además, ofrece todo tipo de herramientas adicionales para hacer más fácil el desarrollo de código. [1]

Figura 1. Logos de Arduino [9] y PlatformIO [13]

2.2 Microcontrolador ESP32-S3 WROOM-1 con cámara

En primer lugar, la parte central de este proyecto es el microcontrolador utilizado, que es el ESP32-S3-WROOM-1. Como su nombre indica, pertenece a la familia ESP32, que son una serie de chips SoC (*System on a Chip*). Es decir, estos módulos contienen todos los componentes básicos de una computadora en un solo circuito integrado. Están fabricados por la empresa Espressif Systems, multinacional de origen chino que desarrolla y fabrica microcontroladores de bajo coste utilizados ampliamente en proyectos del Internet de las Cosas.

El ESP32-S3 es una versión moderna del chip ESP32, que incorpora mayores capacidades de almacenamiento y procesado de datos, tecnología Bluetooth 5.0 y se hace un énfasis en sus mejoras de seguridad. La mayor seguridad de datos de esta variante de chip proviene de su capacidad de admitir firmas digitales y la utilización de encriptado AES-XTS para la memoria flash. Esta característica evita el acceso no autorizado y la manipulación de su memoria y los programas que contiene, haciendo que este dispositivo sea utilizado en aplicaciones con requisitos de seguridad estrictos, como son sistemas de procesamiento de pagos y cámaras de seguridad.

Por otro lado, la palabra “WROOM” quiere decir que el dispositivo cuenta con una antena para recibir y enviar las señales de Bluetooth y Wifi. El hecho de que sea WROOM-1 quiere decir que se trata de una antena PCB (*Printed Circuit Board*) integrada en el propio módulo del procesador, en contraste con el modelo WROOM-1U, en el que la antena es de conexión externa.

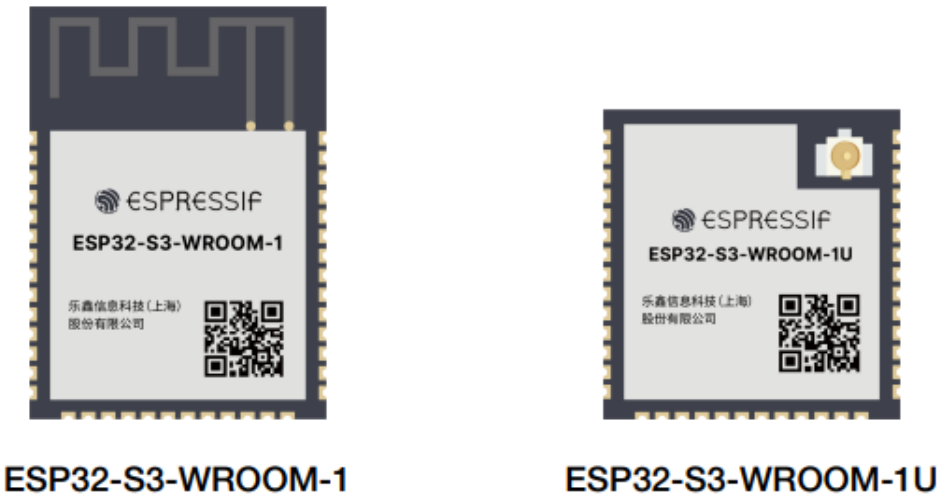


Figura 2. Comparación entre los dos paquetes de antenas para el ESP32-S3 [2]

Además de los elementos ya mencionados y que forman parte del conjunto ESP32-S3 WROOM-1, un elemento de vital importancia en este proyecto será la cámara OV2640 con la que se capturan las imágenes que posteriormente serán procesadas por el microcontrolador. Se hará especial hincapié en su explicación más adelante.

Con todo esto, en los siguientes apartados se explicarán más en detalle las características técnicas del hardware utilizado, hardware que puede verse en la Figura 3.

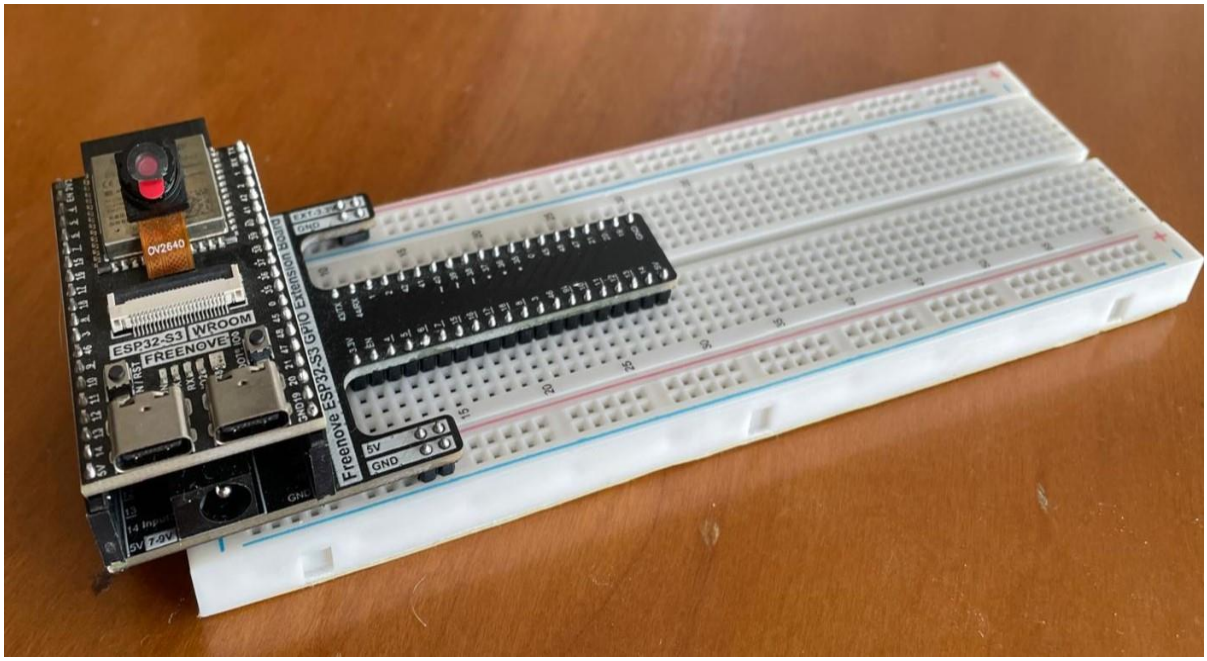


Figura 3. Hardware utilizado, con procesador, cámara y placa de expansión

2.2.1 Layout del ESP32-S3 WROOM-1

De forma esquemática, el aspecto del ESP32-S3 WROOM-1 es el de la Figura 4. En la tabla adyacente se nombran cada uno de los elementos recuadrados sobre el dibujo y su función se explicará a continuación:

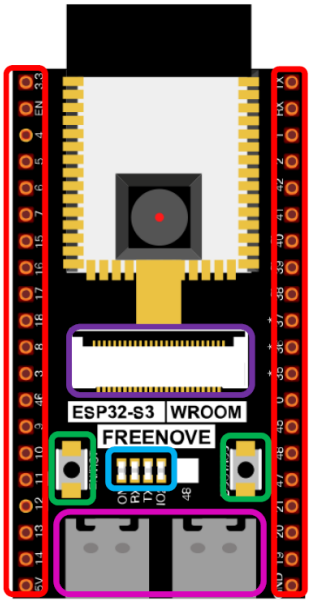


Figura 4. Esquema ESP32 WROOM-1 [12]

Color del recuadro	Componente
	Pines GPIO
	Indicadores LED
	Interfaz de la cámara
	Botones: *Reset*, *Boot mode selection*
	Puertos USB

Tabla 1. Componentes del ESP32 WROOM-1

Los pines GPIO (*General Purpose Input/Output*) se utilizan para conectar y controlar una amplia variedad de dispositivos externos como sensores, actuadores y otros componentes electrónicos. Pueden ser configurados como entradas para leer datos o como salidas para enviar señales. Para esto, en el IDE se utilizan comandos como *pinMode(pin, mode)* (seleccionar el propósito de un pin específico), por poner un ejemplo.

Los indicadores LED sirven para mostrar el estado de operación del dispositivo. Hay un LED configurable por el usuario, y otros tres que muestran cuándo el microcontrolador está recibiendo datos, enviándolos y otras cuestiones como la conexión a la red wifi, etc.

En morado, se visualiza el puerto al que se debe conectar la cámara, en este caso una pequeña cámara digital que se explicará más adelante.

Los botones permiten actuar sobre el estado del procesador. *Reset* reinicia el estado del ESP32, haciendo que el programa cargado vuelva a ejecutarse desde el inicio, mientras que *Boot Mode Selection* lo pone en modo arranque, permitiendo la carga de nuevo firmware.

Por último, en este proyecto se utilizarán los puertos USB-C para cargar programas y recibir datos, a la vez que sirven de fuente de alimentación para

el ESP32 mientras este esté conectado a nuestra computadora. Para ello, es de vital importancia utilizar un cable que permita transmisión de datos.

2.2.2 Placa de expansión

Como se puede ver en la Figura 3 de la página 8, el procesador ESP32-S3 WROOM-1 se encuentra unido a una placa de expansión diseñada específicamente para él (*ESP32S GPIO Extension Board*). Además, esta a su vez está anclada a un tablero de circuitos o *breadboard*. De forma general, estos componentes amplían las capacidades de cualquier ESP32, añadiendo espacio para nuevos módulos periféricos de todo tipo y facilitando la integración de los mismos en proyectos complejos.

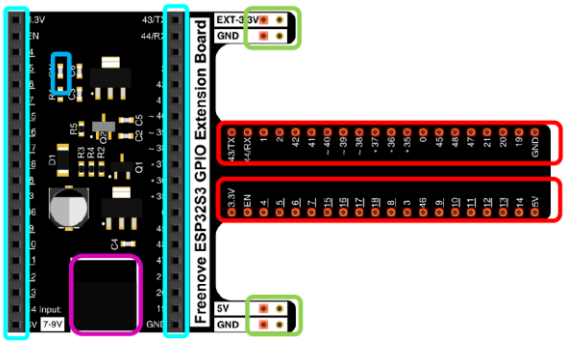


Figura 5. Placa de expansión de ESP32 [12]

Color del recuadro	Componente
	Pines GPIO
	Indicadores LED
	Anclaje para pines GPIO del ESP32
	Output de tensión para periféricos
	Fuente de alimentación externa

Tabla 2. Componentes de la placa de expansión para ESP32

En el caso de este proyecto, el único fin que se va a dar a la placa de expansión es utilizar su puerto para alimentación (Figura 5, en morado) para conectar una fuente de alimentación externa y permitir el funcionamiento del dispositivo sin estar conectado a un computador.

Esta fuente de alimentación externa debería proporcionar 5V ó 3,3V de tensión alterna o continua. En sí, el procesador trabaja con 3,3V corriente continua, pero está preparado para recibir cualquiera de estos dos voltajes y dispone de un convertidor AC/DC interno. La tensión podría ser aportada en la forma de unas baterías o por medio de un cable de alimentación conectado a la red eléctrica con un pequeño adaptador (Figura 6).

Figura 6. Fuentes de alimentación. Baterías (izq.) y adaptador (dcha.)

Con respecto al *breadboard*, en este proyecto no se utilizarán más componentes electrónicos que los ya descritos, por lo que no será utilizado como medio de interconexión. Así su función será más bien la de servir de punto de anclaje para la placa de extensión y el ESP32, facilitando su manipulación y protegiendo los pines.

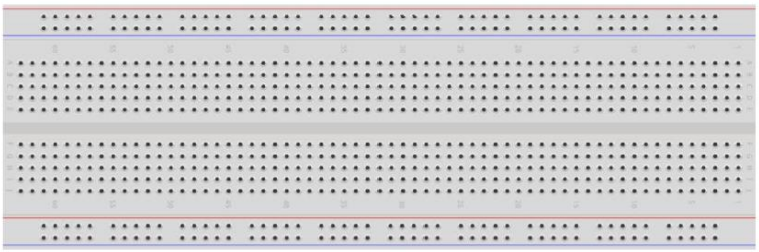


Figura 7. Tablero de circuitos o breadboard [12]

2.2.3 Procesador

El ESP32-S3 WROOM-1 cuenta con un procesador Xtensa LX7 de doble núcleo. Trabaja con 32 bits a una velocidad máxima de 240 MHz y utiliza coma flotante de precisión simple (*FPU*). [2]

Esto significa que puede realizar cálculos complejos y ejecutar múltiples tareas de manera simultánea y eficiente.

2.2.4 Memoria

El procesador cuenta con las siguientes características de memoria: [2]

- **384 KB de ROM** (*Read Only Memory*): Datos de solo lectura como el protocolo de bootloader.

- **8 MB de memoria flash**: Este será el espacio disponible para almacenar los programas subidos desde el IDE. Además, la comunicación entre el procesador y el almacenamiento es de tipo SPI (*Serial Peripheral Interface*), con cuatro líneas de conexión en paralelo: *Quad SPI*. Esto mejora el rendimiento y la velocidad de acceso a la memoria flash.

- **512 KB de SRAM** (*Static Random Acces Memory*): Memoria de alta velocidad para almacenamiento temporal de datos y variables. La hoja de especificaciones de la placa también habla de 16KB de SRAM in RTC (*Real-Time Clock*), es decir, datos que se conservan cuando el dispositivo está en modo de hibernación, aunque esta funcionalidad no será utilizada en este proyecto. Lo que sí resulta de vital importancia para el proyecto es comprender las subdivisiones internas de la SRAM:

· IRAM (*Instruction RAM*): Almacena instrucciones del programa que se copian desde la memoria flash a la IRAM para su ejecución directa desde SRAM, mejorando así la velocidad de ejecución. Puede ocupar hasta 366 KB si fuera necesario, aunque lo normal es utilizar unos pocos kilobytes.

· DRAM (*Data RAM*): Aquí es donde se almacenan los datos durante la ejecución del programa. Igual que la IRAM, puede tener un máximo de 366 KB, siempre que sea necesario (en este proyecto lo será) y la suma de ambas no supere los 512 KB totales de SRAM. Dentro de la DRAM, se distinguen diferentes segmentos:

· Pila de datos (*Stack*): Esta estructura, cuyo tamaño puede ser definido por el usuario, almacena variables locales, direcciones de retorno y los argumentos de las funciones, por lo que llega a ser muy utilizada en le ejecución de ciertas funciones.

· Montón (*Heap memory*): Este tipo de memoria se refiere a datos cuya ubicación se asigna dinámicamente durante la ejecución del programa. Es decir, es el propio usuario el que utiliza funciones (*malloc*, *new*) para solicitar espacios en la memoria a medida que van siendo necesarios, y luego los libera (*free*, *delete*) cuando ya no son utilizados. En el caso de este proyecto, lo que se busca con esto es evitar que ciertas variables de gran tamaño se ubiquen en la pila de datos y provoquen su desbordamiento.

· El resto de la memoria DRAM, que no forma parte de la pila de datos ni ha sido asignada dinámicamente como *heap*, se utiliza para almacenar variables globales y estáticas, y otros buffers de datos de gran tamaño.

Por otra parte, ciertos modelos de ESP32-S3 cuentan con lo que se conoce como PSRAM (*Pseudo Static Random Acces Memory*). Esto proporcionaría una mayor capacidad de almacenamiento temporal para aplicaciones que requieren grandes cantidades de memoria, como el procesamiento de imágenes, como es el caso de este proyecto.

Sin embargo, este microprocesador ESP32-S3 WROOM-1 no cuenta con memoria de este tipo. Así, todos los datos y variables serán almacenados en los 512 KB de SRAM, incluyendo las imágenes capturadas por la cámara. Esto será un factor importante a tener en cuenta a la hora del desarrollo del código del proyecto, como se verá más adelante.

Para hacer las comprobaciones pertinentes sobre la capacidad de almacenamiento de la placa, se han ejecutado funciones de la biblioteca de propósito general *Arduino.h*, que se instala como parte de Arduino IDE. Con

estas funciones se comprueba de primera mano y con exactitud el espacio asignado para cada una de las categorías de memoria.

Por último y para ser más exactos, si se presta atención a la hoja de especificaciones de la placa, [2] se puede ver que estas especificaciones de memoria (8 MB de flash y 0 de PSRAM) señalan a este modelo como la variante ESP32-S3-WROOM-1-N8. Esto lo separa de otros que disponen de más o menos flash, o de cierta PSRAM.

2.2.5 Comunicación

El ESP32-S3 WROOM-1 ofrece diversas opciones de conexión tanto a dispositivos periféricos, como al computador. Estas conexiones sirven para control, carga de programas y transmisión de datos. En este proyecto se utilizará principalmente la conexión por cable, aunque también se hablará brevemente de las opciones Wifi y Bluetooth que soporta.

En primer lugar, el dispositivo permite Wifi 802.11 b/g/n, permitiendo conexiones inalámbricas con velocidades de hasta 150 Mbps. Incluye también características avanzadas como la agregación de tramas (A-MPDU y A-MSDU) y un intervalo de guarda de 0.4 µs, que optimizan la eficiencia y velocidad de la comunicación inalámbrica. [2]

El módulo además soporta Bluetooth LE (Low Energy) 5, incluyendo características como Bluetooth mesh, velocidades de hasta 2 Mbps, y la capacidad de manejar múltiples conjuntos de anuncios. En relación con lo hablado en apartados anteriores, el ESP32-S3 WROOM-1 incorpora un mecanismo de coexistencia interna para compartir la misma antena PCB entre Wifi y Bluetooth, optimizando el rendimiento de ambas conexiones simultáneamente.

En cuanto a conexión por cable, el dispositivo incluye soporte para USB 2.0 OTG (*On The Go*). Esta funcionalidad permite al procesador actuar tanto como host (controlador) como dispositivo (periférico) en una conexión USB. Esta conexión se hace a una velocidad máxima de 12 Mbps, o lo que se conoce como velocidad *Full-Speed*. [2]

Además, la conexión con un computador para la carga de programas y transmisión de datos se realiza a través de lo que se conoce como *USB Serial/JTAG Controller*. La interfaz JTAG (*Joint Test Action Group*) permite a los desarrolladores utilizar herramientas avanzadas para controlar el ESP32, leer y escribir en la memoria, establecer puntos de interrupción y realizar otras operaciones de depuración complejas a través de la conexión USB, algo muy útil en el desarrollo de firmware.

Por otra parte, esta pieza de software también actúa como un puente entre la interfaz USB del ordenador y la UART del ESP32, facilitando la transmisión y recepción de datos seriales mediante USB, lo cual elimina la necesidad de un convertidor USB-UART externo.

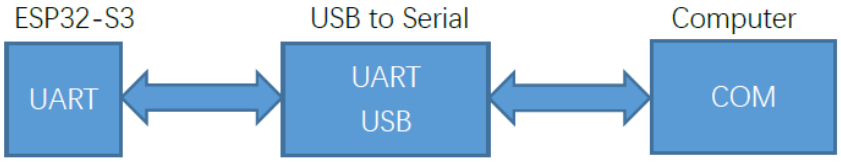


Figura 8. Esquema de la conexión a través del convertidor USB-UART [12]

Relacionando esto con Arduino IDE, significa que pueden cargarse los programas en el ESP32-S3 WROOM-1 y recibir los datos desde el mismo hasta el monitor serie del IDE tanto mediante conexión USB CDC (*Communication Device Class*), como UART (*Universal Asynchronous Receiver-Transmitter*).

En la comunicación UART o comunicación serie, la información se transmite en formato de bits asíncronos, donde los datos son enviados uno tras otro sin necesidad de una señal de reloj común. Este es el tipo de comunicación con la que trabaja el ESP32 internamente.

Por otro lado, con el protocolo USB CDC los datos se transfieren en formato de paquetes USB estándar. Si se elige este tipo, el ordenador recibirá y enviará información en este formato USB y el convertidor integrado en el ESP32-S3 WROOM-1 será el que la transforme en UART.

La conexión UART es más simple, gasta menos recursos y permite utilizar un mayor número de librerías, pero es más lenta. Por el contrario, la conexión USB es más rápida, aunque tiene más complejidad para su implementación.

En este proyecto se van a utilizar frecuencias de muestreo relativamente bajas, con al menos 2 segundos entre que se captura y analiza un fotograma y el siguiente. Así, la velocidad de transmisión de datos no será un factor limitante técnicamente y por lo tanto el tipo de conexión elegida será UART, valorando principalmente su robustez.

2.2.6 Cámara

La cámara es una parte fundamental de este proyecto, ya que será la que capte las imágenes en las que se tienen que detectar los códigos bidimensionales. El ESP32-S3 WROOM-1 viene con un puerto específico para acoplar una pequeña cámara digital, como se ha podido ver en la Figura 4. El modelo empleado es la OV2640, lanzado en 2005 por la empresa Omnivision. A pesar de que su fabricación terminara en 2009, todavía se comercializa como parte de packs de productos ESP32 y sigue teniendo uso a día de hoy.

Figura 9. Cámara OV2460

Esta es una cámara CMOS de bajo voltaje que proporciona una resolución máxima UXGA (*Ultra Extended Graphics Array*: 1632x1232) y puede operar a una velocidad de hasta 15 fotogramas por segundo. La cámara es controlada a través de la interfaz SCCB (*Serial Camera Control Bus*) y soporta un control completo sobre la calidad de la imagen, el formato y la transferencia de datos de salida.

El módulo cuenta con un pequeño microcontrolador integrado, lo que permite funciones de procesamiento de imagen como control de exposición, balance de blancos, saturación de color y cancelación de ruido, todas programables a través de SCCB. La cámara también es capaz de operar en condiciones de poca luz, lo que la hace ideal para aplicaciones portátiles integradas.

Algunas de estas rutinas de procesamiento y control de imagen están programadas en el propio microcontrolador de la OV2460 y se pueden activar para ejecutarse de forma automática. Este es el caso de funciones como el Control Automático de Exposición (AEC), Control Automático de Ganancia (AGC) y más medidas para mejorar la calidad de imagen, reduciendo fuentes comunes de contaminación como el ruido de patrón fijo.

Además, se pueden elegir múltiples formatos de salida de imagen, en concreto Raw RGB, RGB (RGB565/555), GRB422, YUV (422/420) y YCbCr (4:2:2). También, tamaños de imagen desde UXGA hasta resoluciones más bajas. Por último, soporta modos de operación en video y captura de instantáneas.

Todas estas características deben ser tenidas en cuenta para la captura y procesamiento preciso de imágenes, lo que será necesario para la posterior detección y lectura de códigos 2D en el proyecto. [3]

2.3 Software

Una vez descrito el hardware empleado, se puede comenzar a hablar de los aspectos de software más relevantes para el proyecto. Se tratarán de explicar los conceptos pertinentes para entender el desarrollo del código y su funcionalidad.

2.3.1 Estructura de programación

Como se ha mencionado anteriormente, para este proyecto se utilizarán dos entornos de desarrollo diferentes: Arduino IDE y PlatformIO IDE. Las características generales de ambos ya han sido descritas, teniendo muchas similitudes y alguna diferencia, siendo estas últimas el motivo de hacer uso de las dos plataformas. En general, en este apartado se tratará de ahondar en la arquitectura de un proyecto de software embebido.

Tanto Arduino IDE como PlatformIO IDE siguen un enfoque simplificado y accesible para el desarrollo de proyectos en microcontroladores. Aunque es posible utilizar otros lenguajes, se trabajará en C++.

En ambos entornos, el código principal se organiza en dos funciones básicas: *setup* y *loop*. La función *setup* se ejecuta una única vez al inicio del programa. En ella se suele incluir todo lo necesario para poner en marcha el programa: asignación de pines, inicialización de periféricos... Por su parte, la función *loop* contiene el código que se ejecuta repetidamente durante el ciclo de vida del programa.

Además de estas funciones, se pueden declarar variables globales y definir funciones personalizadas fuera de *setup* y *loop*. Las variables globales, declaradas fuera de cualquier función, son accesibles desde cualquier parte del programa, lo que las hace útiles para almacenar estados o valores que necesitan ser compartidos entre distintas funciones. Las funciones personalizadas permiten modularizar el código, dividiendo tareas complejas en partes más manejables que pueden ser llamadas desde *setup* o *loop*, mejorando así la legibilidad y el mantenimiento del código.

2.3.1.1 Arduino IDE

Arduino IDE es un entorno de desarrollo sencillo y directo, ideal para proyectos pequeños o para aquellos que se inician en el mundo de los microcontroladores. En el caso de este proyecto, será utilizado para dar los primeros pasos: familiarizarse con el ESP32-S3 WROOM-1 y ejecutar códigos sencillos que pongan a prueba el funcionamiento del hardware.

Así pues, la estructura de un programa desarrollado en Arduino IDE es extremadamente simple. Un proyecto típico se compone de un único archivo con la extensión ***.ino***, que contiene el código fuente del programa. Este es el archivo que se compila y sube directamente al microcontrolador.

Este archivo *.ino* se puede apoyar en otros archivos complementarios, que ayudan a segmentar el código fuente y hacerlo más comprensible y fácil de modificar. Estos son los archivos de cabecera (*.h*) y archivos fuente adicionales (*.cpp*). Como este tipo de archivos son, por así decirlo, los ladrillos que conforman las librerías, se explicará su funcionamiento más adelante cuando hablemos de ellas.

2.3.1.2 PlatformIO IDE

Frente a Arduino IDE, PlatformIO ofrece una estructura más organizada y profesional, adecuada para proyectos grandes y desarrolladores que necesitan un control más preciso sobre su entorno de desarrollo.

Al crear un proyecto en PlatformIO, se genera una serie de carpetas y archivos que facilitan la organización del código fuente, las bibliotecas, y las configuraciones específicas del proyecto. A continuación, se describe en detalle la función y propósito de cada uno de estos elementos

- **Archivo platformio.*ini***: Es el archivo de configuración principal de un proyecto en PlatformIO. Aquí se definen las opciones clave del proyecto, como la plataforma de hardware y el tipo concreto de placa que se está utilizando, las bibliotecas necesarias, y otras configuraciones específicas del entorno (frecuencia de la conexión con la placa, puerto utilizado…).

- **Carpeta *src***: Aquí se almacena el código fuente principal del proyecto. Por defecto, contiene un archivo *main.cpp* que incluye las funciones *setup* y *loop*, además de cualquier otro código necesario para el funcionamiento del programa. Dentro de esta carpeta, también se pueden crear subcarpetas y archivos *.cpp* adicionales para organizar el código en módulos, mejorando la legibilidad y mantenibilidad del proyecto.

- **Carpeta *lib***: Está destinada a albergar librerías personalizadas que son específicas del proyecto. A diferencia de las instaladas globalmente (en la propia interfaz de PlatformIO) o a través del archivo *platformio.ini*, las bibliotecas en esta carpeta están integradas directamente en el proyecto y no dependen de instalaciones externas. Esta carpeta permite que las dependencias del proyecto sean locales, lo que facilita la portabilidad del proyecto entre diferentes entornos de desarrollo.

- **Carpeta *include***: Se utiliza para almacenar archivos de encabezado (*.h*) que pueden ser compartidos entre diferentes archivos fuente (*.cpp*) dentro del

proyecto. Este tipo de archivos se explicarán en el siguiente apartado, pero en general se utilizan para declarar funciones y variables globales que se usarán en varias partes del proyecto. Mantener estos archivos en una carpeta separada facilita la organización y reutilización del código.

- **Otros archivos y carpetas**: No serán alterados por el usuario durante este proyecto, pero también forman parte de la arquitectura de un proyecto en PlatformIO, por lo que se explicarán brevemente:

  · Carpeta *.pio*: Almacena archivos de construcción generados automáticamente durante la compilación.

  · Carpeta *.vscode*: Contiene configuraciones específicas de Visual Studio Code, como ajustes del editor y tareas de depuración

  · Carpeta *test*: Guarda archivos para pruebas unitarias del proyecto, facilitando la verificación del código.

  · Archivo *.gitignore*: Especifica qué archivos deben ser ignorados por Git, que, a grandes rasgos, se podría definir como el historial de modificaciones realizadas en el proyecto.

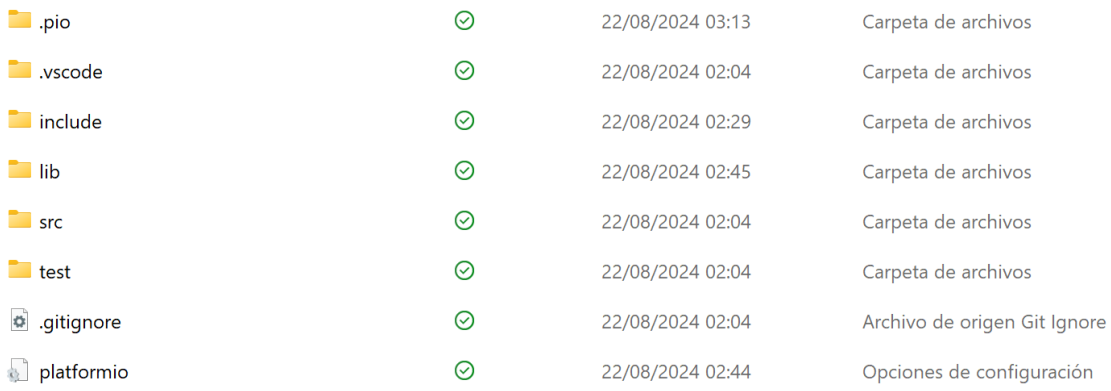


Figura 10. Arquitectura de un proyecto en PlatformIO IDE

2.3.2 Librerías

Una de las características más poderosas de los entornos de desarrollo empleados es la capacidad de gestionar e incluir librerías externas en el proyecto. Las librerías (también se pueden llamar bibliotecas) son colecciones de código preescrito que proporcionan funciones, estructuras de datos y en general métodos específicos para realizar tareas comunes. Varios ejemplos del propósito que pueden tener las librerías son: controlar sensores, manejar protocolos de comunicación, trabajar con pantallas... También, existen algunas cuya función es adaptar el entorno de programación al hardware concreto que se está empleando.

Las bibliotecas son desarrolladas por una comunidad global de programadores, desde fabricantes de hardware hasta desarrolladores independientes. El acceso a ellas generalmente es libre, a través de repositorios web como GitHub [4].

2.3.2.1 Estructura de una librería

La estructura de las librerías generalmente se divide en archivos *.h* (encabezados) y *.cpp* (implementaciones), junto con documentación que explica su uso.

En **archivo de encabezado *.h*** es donde se declaran las funciones, clases y variables globales que pueden ser utilizadas por otros archivos. Este archivo define la interfaz pública de una biblioteca o de un conjunto de funciones, proporcionando las declaraciones necesarias para que otros archivos puedan utilizar estas funcionalidades sin conocer los detalles de su implementación.

También, este tipo de archivo suele contener comentarios sobre la utilidad de las funciones, nociones importantes sobre su funcionamiento y el tipo de variables que se esperan como input y output de las mismas. Es por esto que en la búsqueda de librerías y funciones útiles para el proyecto, es a los archivos de encabezado de la biblioteca donde hay que acudir.

Por otro lado, el **archivo fuente *.cpp*** contiene la implementación de las funciones y métodos declarados en el archivo .h. En otras palabras, mientras el archivo *.h* dice "qué" hace una función, el archivo *.cpp* explica "cómo" lo hace. Esta separación permite una mejor organización del código, especialmente en proyectos grandes, y facilita la reutilización de código en diferentes proyectos, al encapsular funcionalidades en bibliotecas bien definidas.

2.3.2.2 *Integración en el proyecto*

Los dos entornos de desarrollo utilizados facilitan la inclusión de estas bibliotecas en los proyectos. En Arduino IDE, las bibliotecas pueden ser gestionadas a través del gestor de bibliotecas, donde los usuarios pueden buscar, instalar y actualizar bibliotecas de forma sencilla.

Por su parte, PlatformIO ofrece un sistema de gestión de dependencias más sofisticado. Por un lado, se ofrece la opción de incluir los archivos de la librería dentro de la carpeta *lib* del proyecto, permitiendo así realizar ajustes sobre el código de la biblioteca si se está familiarizado con su funcionamiento. Otra opción es declarar las bibliotecas necesarias en el archivo *platformio.ini*. En este caso, las librerías se instalarían automáticamente cuando se compilara el código, asegurando que las versiones correctas de las bibliotecas se instalen y se mantengan actualizadas según las necesidades del proyecto.

Dadas las herramientas más avanzadas que PlatformIO pone a disposición de los desarrolladores, existen más librerías disponibles para esta plataforma que para Arduino IDE. Este será el principal motivo de desarrollar el código final del proyecto en dicha plataforma, ya que no existe ninguna librería para Arduino IDE enfocada en la detección y decodificación de códigos bidimensionales.

Para terminar de concretar, en este proyecto será fundamental el uso de dos librerías:

- ***esp_camera*** permite configurar y controlar cámaras digitales como la OV2640 en placas ESP32, permitiendo capturar imágenes, ajustar parámetros del sensor y otras funciones. La librería es proporcionada por Espressif Systems y forma parte del conjunto de archivos descargados para la gestión de la tarjeta ESP32-S3 WROOM-1. Está disponible para Arduino IDE y PlatformIO IDE.

- ***quirc*** es una herramienta de código abierto diseñada para la detección y decodificación de códigos QR a partir de imágenes. Pese a que es una librería versátil que puede ser aplicada a dispositivos móviles además de sistemas embebidos, no ofrece soporte para Arduino IDE.

2.3.3 Resolución y formato de imagen

Al hilo de lo descrito anteriormente sobre la cámara en cuanto a hardware, a continuación se ahondará un poco más en las características de resolución y formato de la imagen capturada. La alternativa elegida en estos aspectos será muy relevante a la hora del posterior análisis del fotograma en busca de códigos QR.

En cuanto a aspectos prácticos, para hacer que la cámara actúe con los parámetros necesarios para cada aplicación, solo habrá que incluir en el código un par de líneas en las que se especifican la resolución y el tamaño de la imagen a utilizar.

2.3.3.1 Resolución de la imagen

En el ámbito de las cámaras, el término resolución se refiere a la cantidad de detalles que puede capturar en una imagen, generalmente medida en términos de píxeles. Cada píxel es un punto en la imagen que contiene información de color y luminosidad.

Así, cuando se combinan muchos píxeles, se forma la totalidad de una imagen. En una pantalla o en una imagen digital, los píxeles se organizan en una cuadrícula, y la resolución de la imagen se refiere a la cantidad de píxeles en las dimensiones horizontales y verticales.

De esta forma, cuantos más píxeles tenga una imagen, mayor será su resolución. Esto significa que cada fotograma podrá mostrar más detalles y será más nítido, lo que se notará especialmente en objetos que estén a cierta distancia de la cámara.

En este proyecto no van a representarse las imágenes en ningún monitor, por lo que los aspectos visuales no se tendrán en cuenta. Sin embargo, sí será relevante tener buena resolución para que las formas que componen los códigos QR queden mejor definidas al capturar la imagen. Además, cuanto mayor sea la resolución, mayor será la distancia respecto a la cámara a la que un código todavía resulte legible. Con el mismo razonamiento, también podrían llegar a leerse códigos más pequeños.

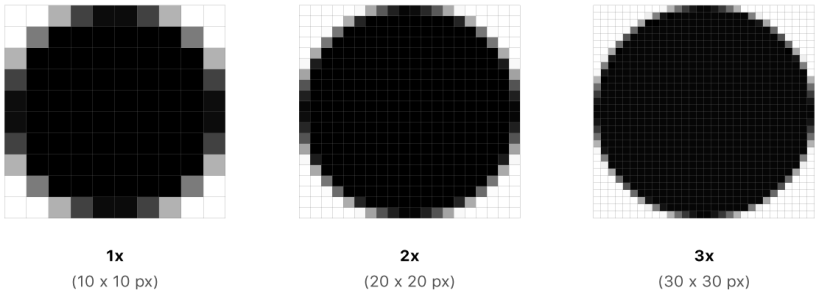


Figura 11. Cómo la resolución de una imagen afecta a las formas que contiene [11]

Como ejemplo, en la Figura 11 se diferencia cómo una única forma (un círculo negro) se vería representado con diferentes resoluciones. Mientras que en la primera imagen, que tiene 10 píxeles en cada eje, apenas se podría distinguir el círculo de un rombo u otra figura similar, con la resolución de 30x30 píxeles se hace mucho más clara la forma real del objeto.

Como ya se ha mencionado. la cámara OV2640 de la que se dispone en este proyecto permitiría una resolución máxima UXGA (1600x1200 píxeles). Además, el dispositivo permite una gran variedad de resoluciones menores a esta, llegando a un mínimo de 96x96 píxeles. En la Tabla 3 se pueden observar las 14 opciones posibles.

Image resolution	Sharpness	Image resolution	Sharpness
FRAMESIZE_96X96	96x96	FRAMESIZE_HVGA	480x320
FRAMESIZE_QQVGA	160x120	FRAMESIZE_VGA	640x480
FRAMESIZE_QCIF	176x144	FRAMESIZE_SVGA	800x600
FRAMESIZE_HQVGA	240x176	FRAMESIZE_XGA	1024x768
FRAMESIZE_240X240	240x240	FRAMESIZE_HD	1280x720
FRAMESIZE_QVGA	320x240	FRAMESIZE_SXGA	1280x1024
FRAMESIZE_CIF	400x296	FRAMESIZE_UXGA	1600x1200

Tabla 3. Resoluciones disponibles para la cámara OV2640 [12]

2.3.3.2 Formato de la imagen

Para hablar de formato de una imagen, primero hay que recordar la definición de píxel: un píxel es punto en la imagen que contiene información de color y luminosidad. Dicho esto, así como la resolución de una imagen indica cuántos píxeles contiene, su formato define cómo está expresada y almacenada la información de cada píxel.

Diferentes formatos de imagen utilizan diferentes métodos para representar los colores y la intensidad de luz, y estos métodos afectan la calidad de la imagen, el tamaño del archivo y la compatibilidad con dispositivos y software.

En primer lugar, se puede hablar de RGB. Este formato almacena los colores de una imagen usando tres canales: Rojo (Red), Verde (Green) y Azul (Blue). Cada canal representa la intensidad de uno de esos colores en un píxel.

El RGB es común en pantallas y dispositivos de visualización, ya que refleja cómo los humanos percibimos el color. Es por lo tanto el formato en el que mejor se conservan los colores originales de la instantánea. Sin embargo, debido a que requiere tres valores para cada píxel, y cada uno de estos valores ocupa 5 o 6 bits, los archivos en formato RGB suelen ser grandes.

Dentro de este formato, la cámara soporta RGB565 y RGB555, donde los números indican cómo se distribuyen los bits entre los canales de color. Por

ejemplo, en RGB565, se utilizan 5 bits para el rojo, 6 para el verde y 5 para el azul. RAW RGB es otro formato compatible con esta cámara, en el que se almacenan los datos sin procesamiento, lo que permite un control completo sobre la imagen, pero requiere aún más espacio y procesamiento.

Por otra parte, se tiene el formato YUV. En él, se separa la información de la imagen en un canal de luminancia (Y) y dos canales de crominancia (U y V). La luminancia (Y) representa el brillo o la intensidad de la luz, mientras que la crominancia (U y V) contiene la información de color. Más concretamente, U es la crominancia azul, que se calcula como la cantidad de azul que debe añadirse a la luminancia para obtener el color original. Básicamente, el valor U contiene la información necesaria para describir cuánto azul hay en el color final del píxel. El valor V es similar al U, solo que relacionándolo con el color rojo.

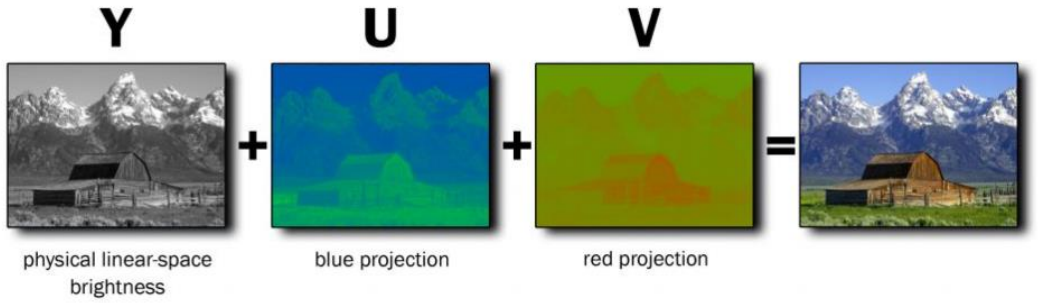


Figura 12. Separación de canales en una imagen en formato YUV [15]

El formato YUV es muy eficiente, porque que permite reducir la cantidad de datos de color (U y V) sin afectar significativamente la calidad percibida, ya que el ojo humano es más sensible a la luminancia que a los colores. En concreto, mientras que RGB obliga a asignar un número similar de bits para cada color para cada pixel, con YUV se puede tener un número menor de bits asignados a las crominancias que a la luminancia, ahorrando espacio sin que el resultado final se vea distorsionado.

Para el caso de la cámara OV2640, se tienen disponibles los formatos YUV422, YUV420 y YCbCr422. El número que contiene cada formato indica la forma de empaquetado de la información de los píxeles, en el que entra en juego el concepto de submuestreo.

Para explicarlo, se toma como ejemplo el formato YUV422. En él, se almacena en 8 bits un valor de luminancia (Y) para cada píxel. En cambio, los valores de crominancia (U y V) también se almacenan en 8 bits, pero son compartidos por cada dos píxeles adyacentes horizontalmente. Es decir, en vez de hacer un muestreo para cada punto, se ha “submuestreado” y asignado el mismo valor de crominancia para dos píxeles contiguos, despreciando la posible variación de color que pueda haber entre ellos.

Así, cada píxel es de 16 bits: 8 para su luminancia (Y) y 8 para solo una de sus crominancias (por ejemplo, U). La información de su otra crominancia (V en este ejemplo) estará contenida en 8 bits del siguiente píxel, que a su vez tendrá sus propios 8 bits de luminancia y leerá el valor de crominancia que le falta (U) del ya mencionado pixel anterior.

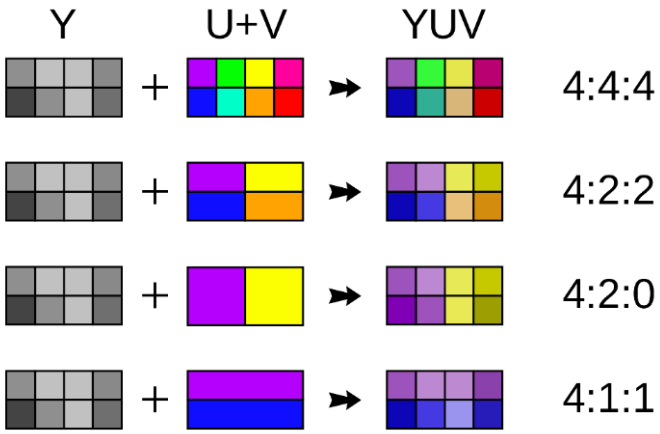


Figura 13. Variantes del formato de imagen YUV [16]

Visto este ejemplo y la Figura 12, se podrá entender también el formato YUV420, en el que el submuestreo se lleva a cabo en horizontal y en vertical, con lo que los 16 bits totales de crominancia se comparten entre 4 píxeles. Por lo tanto, la información de cada píxel ocupa tan solo 12 bits, conservando 8 para la luminancia (Y), Por otra parte, YCbCr422 hace referencia a una forma diferente de calcular la crominancia, más utilizada para sistemas de vídeo digitales.

También, es importante constatar que existen fórmulas para transformar cualquier imagen de formato RGB a YUV, incluso después de haber sido capturado el fotograma.

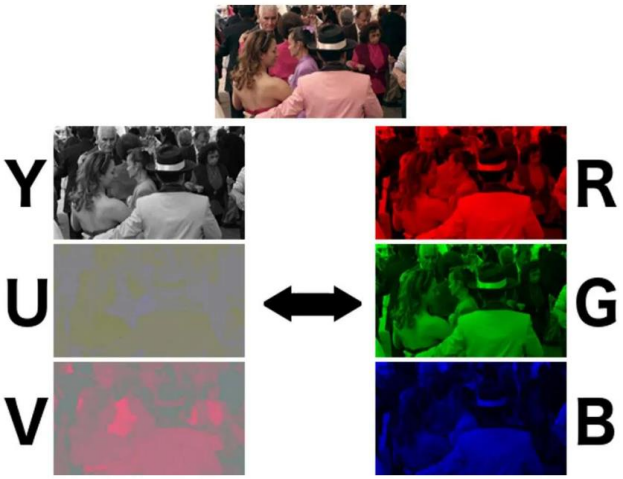


Figura 14. Formatos de imagen en color disponibles [15]

Finalmente, si se realiza la misma separación en los canales Y, U y V del formato YUV, pero solo se almacenan los datos de luminancia, el formato resultante será la escala de grises (grayscaling). De esta forma, se pierde completamente la información cromática de la imagen, pero se mantiene la precisión en cuanto a la luminosidad de cada píxel, que se ve reducido a tan solo 8 bits. Es decir, cada pixel está representado por un valor de 0 a 255 que indica su luminosidad.

2.3.3.3 Características finales de la imagen

Combinando lo explicado en los dos apartados anteriores, se llega a unas características finales de la imagen que definen su funcionalidad. Con recursos ilimitados, la resolución UXGA permitiría apreciar el máximo de detalle en la imagen en cuanto a formas, mientras que el formato RGB sería el más fidedigno conservando los colores.

Sin embargo, existe un factor limitante: el tamaño de la imagen. Concretamente, el espacio que ocupa una instantánea es el resultado de multiplicar el número total de pixeles en la imagen (resolución) por el número de bits necesarios para almacenar cada pixel (formato). Además, cuando hablamos de tamaño como obstáculo, estamos englobando dos problemas:

Por un lado, un tamaño grande de imagen implica un mayor tiempo para procesarla. Este tiempo será debido al propio procesador interno de la cámara, el guardado en la memoria del ESP32 y las operaciones que el programa del proyecto realice sobre la misma para detectar los códigos bidimensionales.

Aunque es posible verse lastrado por esto, no será un problema para este proyecto. Como se comprobará al final del proyecto, el procesador empleado es lo suficientemente rápido para que el intervalo de muestreo resultante sea más que adecuado para la aplicación.

Por otra parte, la consecuencia más obvia de una imagen de gran tamaño, es que se necesita un espacio de memoria acorde para albergarla. Aquí es donde entra en juego la anteriormente mencionada ausencia de memoria PSRAM. Con un módulo de este tipo, se dispondría de espacio extra para almacenar la imagen, que es una variable de gran tamaño.

Así, para el caso de este proyecto habrá que adaptar el código para los 366 KB de memoria para datos o DRAM de los que se disponen, dejando también espacio para el resto de variables del código. La decisión tomada finalmente y su justificación se verá más adelante en el apartado de desarrollo del proyecto.

2.4 Códigos QR

Los códigos QR (*Quick Response* ≈ Respuesta rápida) son un formato de código bidimensional de muy amplio uso en la actualidad, compatibles con multitud de dispositivos del día a día como teléfonos inteligentes y otros escáneres sencillos. A día de hoy, es muy común encontrarlos en etiquetas de productos, tickets de eventos, y campañas publicitarias, donde facilitan el acceso instantáneo a sitios web, promociones, y contenido multimedia. Además, han sido fundamentales en la digitalización de servicios como por ejemplo los pagos móviles.

En cuanto a su historia, los códigos QR fueron desarrollados en 1994 por la empresa japonesa Denso Wave, una subsidiaria de Toyota, con el propósito de mejorar la gestión de inventarios en la industria automotriz. Inicialmente, surgieron como una evolución de los códigos de barras bidimensionales, que ya se utilizaban para almacenar más información que los códigos de barras lineales. Sin embargo, los QR ofrecen capacidades de almacenamiento y velocidades de escaneo muy superiores, lo que permitió su adopción en un amplio rango de aplicaciones.

Figura 15. Evolución códigos de barras estándar, a códigos de barras 2D, a QRs [10]

Así, una de las características que los hace tan poderosos es que los QR permiten almacenar una gran cantidad de información en poco espacio. Para ello utilizan una matriz de **módulos** cuadrados en blanco y negro, en la que se codifican datos complejos como texto, enlaces a sitios web y otros tipos de información digital.

Su estructura también ofrece gran robustez ante la pérdida de información, lo que los dota de gran versatilidad. Para ello, a la hora de codificar los datos se utilizan protocolos de corrección de errores estandarizados en varios grados de resistencia al fallo.

En resumen, los códigos QR se han convertido en una herramienta esencial en la era digital debido a su rapidez, eficiencia y capacidad de manejo de grandes volúmenes de información en una sola imagen. Es por esto por lo que familiarizarse y trabajar con ellos programando un ESP32 para que los detecte y decodifique es el objetivo último de este proyecto. Para ello, a continuación se ahondará en sus tipos, estructura y funcionamiento.

2.4.1 Tipos de códigos bidimensionales

El mundo de los códigos bidimensionales para almacenamiento y transmisión de información es muy amplio, por lo que lo mejor será empezar por identificar las diferentes variantes. Se entrará más en detalle en los códigos QR más habituales o estándar, que es con los que se trabajará en este proyecto. También se mencionarán brevemente otras variantes de códigos QR y de otro tipo.

2.4.1.1 Códigos QR estándar

Se puede definir como códigos QR estándar a aquellos de más amplio uso en los diferentes sectores en los que se utilizan códigos bidimensionales a nivel global. Esto se debe a que es el tipo de código más robusto y con las características más estandarizadas. Dentro de este código QR "clásico", existen dos modelos:

- **Modelo 1**: Fue el primer tipo de código QR introducido al mercado y es reconocible por su estructura básica que carece de patrones de alineamiento. Esta ausencia limita la capacidad del modelo 1 para corregir distorsiones, especialmente en códigos de mayor tamaño. Además, su tamaño máximo y por lo tanto, capacidad de almacenamiento, es menor en comparación con el modelo 2.

- **Modelo 2**: Desarrollado para superar las limitaciones del modelo 1, el modelo 2 incluye patrones de alineamiento, lo que mejora significativamente la capacidad del código para corregir deformaciones y ser leído desde distintos ángulos. Además, esta versión admite códigos más grandes en los que se puede codificar una mayor cantidad de datos, haciendo posible almacenar información más compleja y diversa. Debido a estas mejoras, el modelo 2 es el estándar dominante en la actualidad.

En este proyecto, se trabajará únicamente con el modelo 2, que es la versión más depurada del código y por lo tanto el formato utilizado en la totalidad de las aplicaciones modernas de los QR estándar. De hecho, a día de hoy resulta muy difícil encontrar librerías, sitios web y otros software que todavía trabajen con el modelo 1.

2.4.1.2 Otros códigos QR y bidimensionales

Para completar la clasificación de los códigos bidimensionales más comunes, en este apartado se mencionarán otras variantes. Todos estos no serán utilizados en este proyecto, debido a que se usan en aplicaciones más especializadas y no hay a disposición del público tanta información sobre su funcionamiento ni herramientas para trabajar con ellos.

En cuanto a los códigos que se pueden englobar dentro de la familia de los QR, se tienen:

- Micro QR: Una versión compacta del código QR estándar, que puede almacenar menos datos, pero es útil en etiquetas pequeñas.

- Rectanguar Micro QR (rMQR): Permite un diseño más flexible en términos de proporción y orientación, sin ser tan limitado en almacenamiento, por lo que es igualmente ideal para entornos con restricciones de espacio.

- Security QR Code (SQRC): Utilizado para aplicaciones donde la seguridad es crucial, ya que permite ocultar parte de la información a escáneres no autorizados.

- Frame QR: Ofrece un área central personalizable para insertar logotipos o diseños, ofreciendo mayor flexibilidad estética para aplicaciones de marketing y branding.

Figura 16. Variantes de código QR [14]

Fuera de la denominación de QR, se pueden encontrar códigos bidimensionales como los siguientes:

- DataMatrix: Código bidimensional compacto y altamente eficiente, ideal para aplicaciones industriales y médicas por su capacidad de almacenar grandes cantidades de datos en espacios muy pequeños.

- Aztec Code: Código 2D con un patrón central que lo hace fácil de leer desde cualquier ángulo. Es utilizado en billetes electrónicos y documentos de viaje por su alta fiabilidad y capacidad de corrección de errores.

Figura 17. Otros códigos bidimensionales [14]

2.4.2 Estandarización de códigos QR

Desde su creación, los códigos QR han sido regulados y estandarizados por varias normas internacionales para asegurar su interoperabilidad y confiabilidad en diversas aplicaciones. La primera norma sobre el tema se publicó en 1999 en Japón, y la más actual data de 2015, en la que se definen los requerimientos para los tipos de código más recientes.

En lo que concierne a este proyecto, la versión utilizada a día de hoy de la norma para los códigos QR es la **ISO/IEC 18004:2015** [5]. Esta actualización unificó todas las especificaciones previas y consolidó las características técnicas de los códigos QR modelo 2, que se han convertido en el estándar global. Entre otras especificaciones, esta norma detalla cómo deben organizarse la estructura del código, así como los niveles de corrección de errores. También incluye pautas sobre el manejo de la información codificada y otros elementos clave. Todo ello se irá desarrollando en los siguientes apartados.

2.4.3 Estructura de un código QR

De acuerdo con los estándares de la norma, todos los códigos QR del mismo tipo deben tener la misma estructura, en la que cada parte de la matriz de píxeles (llamados **módulos**) contiene un tipo de información. A continuación, se explicará la organización del QR más completo y ampliamente utilizado, el modelo 2. El modelo 1 sería similar solo que sin marcas para alineamiento. Con la Figura 18 como apoyo, se puede explicar las partes de la matriz y su propósito:

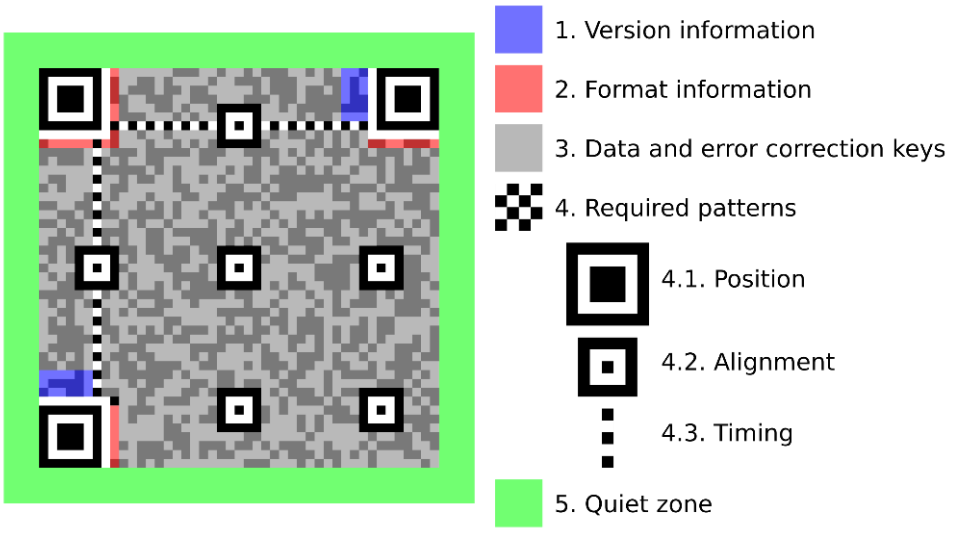


Figura 18. Estructura de un código QR modelo 2

- 1. **Versión**: Esta sección contiene los bits que indican la versión del código QR utilizada, dentro del modelo 2. Cada versión tiene un tamaño de cuadrícula específico y por lo tanto contiene más o menos información. Existen un total de 40 versiones, empezando por la versión 1 de 21x21 módulos, e incrementando de 4 en 4 módulos por eje hasta la 40 en la que cada código utiliza 177x177 píxeles diferentes.

- 2. **Formato**: Aquí se almacenan datos sobre el nivel de corrección de errores utilizado y el patrón de enmascaramiento aplicado al código. Hay 4 niveles de corrección de errores disponibles, cuyo funcionamiento se explicará más tarde al hablar de codificación, y 8 patrones de enmascaramiento posibles. Estos patrones invierten selectivamente ciertos módulos (cambiando de blanco a negro o viceversa) según una fórmula predefinida, con el objetivo de mejorar la distribución visual de los módulos negros y blancos Así, se evitan formas no deseadas como grandes espacios en blanco o negro.

- 3. **Datos y claves de corrección de errores**: Aquí se encuentran los datos codificados, que pueden incluir información en formato numérico, alfanumérico, binario o kanji (caracteres japoneses). Además, en esta área se implementan los bits de corrección de errores (según el nivel de corrección seleccionado), lo que permite la recuperación de la información incluso si el código está parcialmente dañado o sucio.

- 4. **Patrones de función**: Estas son las únicas partes de la cuadrícula que son iguales para todos los códigos QR. No contienen datos codificados, pero facilitan su detección, posicionamiento y la lectura de los módulos que sí contienen la información:

  · 4.1. **Posicionamiento**: Estos patrones permiten que el lector detecte la presencia y la orientación del código QR dentro de una imagen. Durante la lectura, el escáner utiliza estos patrones para alinear el marco de la imagen con el código QR y así poder procesar correctamente los datos almacenados.

  · 4.2. **Alineamiento**: Exclusivos del Modelo 2, estos patrones ayudan a corregir cualquier distorsión que pueda ocurrir durante la lectura, asegurando que los datos se interpreten correctamente incluso si el código está inclinado o deformado. Están dispersos dentro del área de datos, en particular en versiones más grandes del código QR. . Serán más numerosos cuanto mayor sea la cuadrícula, pero siempre tienen el mismo aspecto y se distribuyen uniforme y equidistantemente en la matriz.

  · 4.3. **Sincronización**: Las líneas de sincronización, formadas por bits alternos, permiten al escáner determinar el tamaño exacto de los módulos y la densidad del código. A medida que el escáner recorre el

código QR, utiliza estas líneas de sincronización para contar el número de módulos entre los patrones de posicionamiento, con lo que establece la estructura y tamaño de la cuadrícula

- 5. **Zona de silencio**: Este es el margen en blanco alrededor del código QR que actúa como un borde de seguridad. Su función es asegurar que el lector distinga claramente el código de su entorno, evitando interferencias durante la lectura. La norma ISO/IEC 18004 establece que esta zona debe tener un ancho mínimo de cuatro módulos.

2.4.4 Codificación de códigos QR. Corrección de errores

La codificación de un código QR sigue unas normas estrictas definidas con exactitud en la norma ISO/IEC 18004 [5]. En este apartado se explicarán cada uno de los pasos del proceso, pero primero será necesario entender algunos conceptos.

En primer lugar, se recordará que para el código QR estándar existen 40 versiones, que hacen referencia al tamaño de la cuadrícula, partiendo de 27x27 (Versión 1) hasta 177x177 (Versión 40). Lógicamente, a mayor tamaño de la matriz, mayor será la capacidad de almacenamiento de datos. También se recordará que esta información puede ser de tipo numérico, alfanumérico, binario o kanji (caracteres japoneses).

Estos datos se codifican dentro del código QR en bloques de 1 byte llamados **codewords**. Gráficamente, cada codeword se representa con 8 módulos, uno para cada bit, siguiendo un patrón que se verá más adelante.

La capacidad de almacenamiento del símbolo también dependerá del grado de corrección de posibles errores deseado para el código. Así, los códigos QR pueden manejar diferentes niveles de corrección de errores, que determinan qué porcentaje de módulos dañados o ilegibles puede corregirse sin perder información:

- L (Low): Hasta un 7% de los módulos pueden ser restaurados.

- M (Medium): Hasta un 15% de los módulos pueden ser restaurados.

- Q (Quartile): Hasta un 25% de los módulos pueden ser restaurados.

- H (High): Hasta un 30% de los módulos pueden ser restaurados.

Con todo esto, dentro de lo que comúnmente se conoce como código QR estándar se tienen hasta 160 variantes con 160 valores de memoria diferentes. Como ejemplo, para referirse a un código de cuadrícula 65x65 con corrección baja de errores, se dirá que es un QR 12-L, que cuenta con 2960 módulos para datos agrupados en 370 codewords. Esta y el resto de las

capacidades de almacenamiento de las versiones 10 a 15 se pueden observar en la Tabla 4 a continuación.

Version	Error correction level	Number of data codewords	Number of data bits	Data capacity			
				Numeric	Alphanumeric	Byte	Kanji
10	L	274	2 192	652	395	271	167
	M	216	1 728	513	311	213	131
	Q	154	1 232	364	221	151	93
	H	122	976	288	174	119	74
11	L	324	2 592	772	468	321	198
	M	254	2 032	604	366	251	155
	Q	180	1 440	427	259	177	109
	H	140	1 120	331	200	137	85
12	L	370	2 960	883	535	367	226
	M	290	2 320	691	419	287	177
	Q	206	1 648	489	296	203	125
	H	158	1 264	374	227	155	96
13	L	428	3 424	1 022	619	425	262
	M	334	2 672	796	483	331	204
	Q	244	1 952	580	352	241	149
	H	180	1 440	427	259	177	109
14	L	461	3 688	1 101	667	458	282
	M	365	2 920	871	528	362	223
	Q	261	2 088	621	376	258	159
	H	197	1 576	468	283	194	120
15	L	523	4 184	1 250	758	520	320
	M	415	3 320	991	600	412	254
	Q	295	2 360	703	426	292	180
	H	223	1 784	530	321	220	136

Tabla 4. Capacidad de almacenamiento para códigos QR versión 10-15 [5]

2.4.4.1 Conversión del mensaje

El primer paso al generar un código QR es convertir el mensaje original en una cadena de código binario. Dependiendo del tipo de datos que se desee almacenar (numérico, alfanumérico, binario o kanji), se utilizan métodos o **modos de conversión** diferentes, pero el proceso es esencialmente el mismo.

El primer paso es segmentar la información. Por ejemplo, en el modo numérico se agrupan dígitos de tres en tres. Cada uno de estos bloques se convierte a binario ocupando 4, 7 o 10 bits según sea necesario. Para ilustrar la explicación, se convertirá el mensaje "1234567":

1234567 → 123 456 7 →
→ 0001111011 (10 bits) 111001000 (10 bits) 0111 (4 bits) →
→ 1111011 111001000 0111

A continuación, se incluyen al comienzo de la cadena una serie de 4 bits que señalan el modo de codificación. Seguidamente, se añade un indicador del número de caracteres del mensaje original, siendo la longitud de este código variable en función de la versión. Siguiendo el ejemplo, si el mensaje anterior se quisiera guardar en un QR 1-H, se utilizarían 10 bits:

0001 (Modo: numérico) 0000000111 (7 caracteres) →
Cadena "123567" ≡ 0001 0000000111 0001111011 111001000 0111

Por último y solo para ejemplificar en qué consisten las diferencias entre los modos, en alfanumérico se agruparían los caracteres de dos en dos ocupando hasta 11 bits por bloque. El indicador de modo sería "0010" y en un código 1-H se utilizarían 9 bits para indicar el número de caracteres.

2.4.4.2 Aplicación del algoritmo de corrección de errores

Para preparar el mensaje ante posibles fallos, se utiliza el **algoritmo de Reed-Solomon.** Este es un método de corrección de errores utilizado en muchas aplicaciones, como CDs, DVDs, transmisión de datos, y por supuesto, códigos QR. Fue desarrollado en 1960 por Irving S. Reed y Gustave Solomon.

Así, este algoritmo se utiliza para detectar y corregir errores en datos transmitidos o almacenados. Funciona mediante la creación de bloques de información adicionales que se añaden a los datos originales. Si algunos de los bloques del mensaje inicial ya no están disponibles (se han dañado, ensuciado, no salen en la imagen...) los codewords de corrección de errores permiten recuperar la información perdida.

Además, cuanta más información adicional acompañe a un mensaje en forma de bits para corrección de errores, más fácil será que la información original pueda recuperarse aun estando dañada. Eso sí, si se trabaja con una cuadrícula fija, aumentar el número de codewords para corrección de errores supone un menor número de módulos destinado a los datos. Esto explica por qué, como se pudo observar en la Tabla 4, según se asciende en el nivel de corrección de errores, disminuye la capacidad de almacenamiento de una misma versión de código QR.

Version	Total number of codewords	Error correction level	Number of error correction codewords	Number of error correction blocks
1	26	L M Q H	7 10 13 17	1 1 1 1
2	44	L M Q H	10 16 22 28	1 1 1 1
3	70	L M Q H	15 26 36 44	1 1 2 2
4	100	L M Q H	20 36 52 64	1 2 2 4

Tabla 5. Codewords de error y bloques de datos para códigos QR versión 1-4 [5]

El primer paso del algoritmo es dividir la cadena binaria que contiene el mensaje original en bloques. Cada uno de estos bloques se trabajará de forma independiente de ahora en adelante. El número de divisiones de la información original depende de la versión de QR y el nivel de corrección de errores elegido. En la tabla 5 se puede ver el número total de codewords, cuántas se dedican a corregir errores y en cuántos bloques se organizan, para cada nivel de las versiones 1 a 4.

A continuación, los códigos de corrección de cada bloque se calculan utilizando la aritmética polinómica en un **campo de Galois *GF(2⁸)***. Un campo de Galois, o campo finito, es un conjunto matemático que contiene un número finito de elementos y permite realizar operaciones de suma, resta, multiplicación y división con ellos (excepto la división por cero). Son muy utilizados en la teoría de la información y la criptografía.

La elección de un campo $GF(2^8)$ quiere decir que todas las operaciones se realizan con elementos de 8 bits, que coincide con la longitud de cada codeword. Además, el campo viene definido por un polinomio irreducible, en este caso $x^8 + x^4 + x^3 + x^2 + 1$, que en binario se representa como 100011101 [5]. Su función se entenderá más adelante.

Para pasar la información de cada bloque de datos al campo de Galois, se utiliza un **polinomio de datos**, en el que cada coeficiente se corresponde con 8 bits (un codeword) de la cadena binaria de datos. Como ejemplo, si se tiene un bloque de datos constituido por los codewords $[c_0, c_1, \ldots, c_k]$, el polinomio de datos sería:

$$D(x) = c_0 \cdot x^k + c_1 \cdot x^{k-1} + \ldots + c_k$$

Por otro lado, se tiene un **polinomio generador**, que varía en función del número de codewords de corrección de errores que debe haber en el bloque. Los polinomios generadores vienen especificados en la norma ISO/IEC 18004, tal y como se puede ver en la Tabla 6.

Number of error correction code-words	Generator polynomials
2	$x^2 + \alpha^{25}x + \alpha$
5	$x^5 + \alpha^{113}x^4 + \alpha^{164}x^3 + \alpha^{166}x^2 + \alpha^{119}x + \alpha^{10}$
6	$x^6 + \alpha^{166}x^5 + x^4 + \alpha^{134}x^3 + \alpha^5x^2 + \alpha^{176}x + \alpha^{15}$
7	$x^7 + \alpha^{87}x^6 + \alpha^{229}x^5 + \alpha^{146}x^4 + \alpha^{149}x^3 + \alpha^{238}x^2 + \alpha^{102}x + \alpha^{21}$
8	$x^8 + \alpha^{175}x^7 + \alpha^{238}x^6 + \alpha^{208}x^5 + \alpha^{249}x^4 + \alpha^{215}x^3 + \alpha^{252}x^2 + \alpha^{196}x + \alpha^{28}$
10	$x^{10} + \alpha^{251}x^9 + \alpha^{67}x^8 + \alpha^{46}x^7 + \alpha^{61}x^6 + \alpha^{118}x^5 + \alpha^{70}x^4 + \alpha^{64}x^3 + \alpha^{94}x^2 + \alpha^{32}x + \alpha^{45}$
13	$x^{13} + \alpha^{74}x^{12} + \alpha^{152}x^{11} + \alpha^{176}x^{10} + \alpha^{100}x^9 + \alpha^{86}x^8 + \alpha^{100}x^7 + \alpha^{106}x^6 + \alpha^{104}x^5 + \alpha^{130}x^4 + \alpha^{218}x^3 + \alpha^{206}x^2 + \alpha^{140}x + \alpha^{78}$
14	$x^{14} + \alpha^{199}x^{13} + \alpha^{249}x^{12} + \alpha^{155}x^{11} + \alpha^{48}x^{10} + \alpha^{190}x^9 + \alpha^{124}x^8 + \alpha^{218}x^7 + \alpha^{137}x^6 + \alpha^{216}x^5 + \alpha^{87}x^4 + \alpha^{207}x^3 + \alpha^{59}x^2 + \alpha^{22}x + \alpha^{91}$
15	$x^{15} + \alpha^8x^{14} + \alpha^{183}x^{13} + \alpha^{61}x^{12} + \alpha^{91}x^{11} + \alpha^{202}x^{10} + \alpha^{37}x^9 + \alpha^{51}x^8 + \alpha^{58}x^7 + \alpha^{58}x^6 + \alpha^{237}x^5 + \alpha^{140}x^4 + \alpha^{124}x^3 + \alpha^5x^2 + \alpha^{99}x + \alpha^{105}$

Tabla 6. Polinomios generadores para bloques de 2 a 15 codewords [5]

Por simplicidad, se tomará como ejemplo un bloque con tan solo 2 codewords de corrección de errores. Este bloque tan pequeño corresponde a códigos Micro QR, de tan solo 5 codewords totales. Se tendrían:

Suponiendo como datos: *00001100 01010100 11001010*

$D(x) = 12x^2 + 84x + 202$

$G(x) = x^2 + \alpha^{25}x + \alpha$

α es el elemento primitivo del campo, en este caso α = 2. Por lo tanto, a priori, $\alpha^{25} = 2^{25}$ es mayor que 2^8, que es el valor máximo que puede tener un elemento dentro del campo de Galois *GF(2⁸)*. Aquí es donde entra en juego el polinomio irreducible mencionado anteriormente (100011101), que se utiliza para transformar cualquier número a un valor de 8 bits.

Para ello, se alinea el primero de sus 9 bits con el bit más significativo del número que se quiere reducir y entre ambos se aplica una operación XOR. EL resultado tendrá sí o sí su bit más significativo al menos una posición más a la derecha que antes y con él se repite la operación anterior. Después de unas pocas iteraciones, se llegará a un número de 8 bits.

100000000000000000000000000
XOR
100011101
=
000011101000000000000000000
XOR
100011101000000000000
=
...
=
000000000000000000010001111 → $\alpha^{25} = 10001111 = 143$

Teniendo ya el polinomio generador, se puede pasar a calcular el patrón de corrección de errores. Para ello, se divide el polinomio de datos entre el polinomio generador con aritmética de campo finito (módulo 256) y se observa el residuo:

$$D(x) / G(x) = C(x) + R(x) / G(x)$$

$12x^2 + 84x + 202 / x^2 + 143x + 2$ → $C(x) = 12$

R(x) se obtiene multiplicando G(x) por 12, aplicando módulo 256 y restándolo a D(x):

$$12 \cdot (x^2 + 143x + 2) = 12x^2 + 1716(mod\ 256)x + 24 = 12x^2 + 180x + 24$$

$$R(x) = (12x^2 + 84x + 202) - 12x^2 + 180x + 24 \rightarrow$$

$$R(x) = 160x + 178$$

Los codewords de corrección de errores serán los coeficientes del polinomio *R(x)*, que, como se ha visto, es el residuo de la división del polinomio que contiene los datos del mensaje, entre el polinomio generador que viene

establecido en la norma según el número de patrones de rectificación de fallos requeridos. Estos codewords después se distribuirán en el código junto con aquellos en los que están codificados los datos. Terminando este ejemplo:

Codewords de corrección de errores: 10100000 (160_{10}) , 10110010 (178_{10})

Bloque final con 5 codewords para representar:

[Datos: 00001100 01010100 11001010] [Errores: 10100000 10110010]

Como se puede observar en la Tabla 6, el grado del polinomio generador siempre es igual al número de codewords de corrección de errores buscado. Lógicamente, esto es así para que el residuo *R(x)* tenga el mismo número de coeficientes que los bytes que queremos generar, pero también tiene como resultado polinomios generadores muy complejos para bloques de datos grandes. Esta es la principal razón para que el primer paso del algoritmo de Reed- Solomon sea dividir el mensaje que se quiere codificar en cadenas más pequeñas.

En el ejemplo de este apartado se ha utilizado un bloque de datos muy pequeño, pero con él se puede entender el funcionamiento general del algoritmo de Reed-Solomon. Es fácil imaginar que el poder de computación del escáner empleado se pondrá a prueba para las versiones más grandes del QR.

2.4.4.3 Colocación de datos en la matriz del código

Una vez obtenida la cadena de código binario que se quiere representar, incluyendo tanto los datos del mensaje original como los patrones que servirán para la corrección de errores, se puede pensar en ubicarlos dentro de la estructura de un código QR. Nuevamente, este proceso viene perfectamente documentado en la norma ISO/IEC 18004 [5].

La matriz del código QR comienza como una cuadrícula en blanco, cuyo tamaño depende de la versión del código. Los patrones de función (posicionamiento, alineación y sincronización) se colocan lo primero. Siempre tienen la misma forma y se colocan en posiciones predefinidas de la matriz (Ver Figuras 18 y 21).

Las posiciones correspondientes al formato y la versión del código se dejan libres por el momento. La codificación de estos datos es sencilla y en ella también se utiliza un mecanismo de corrección de pérdida de información similar al explicado anteriormente. Concretamente, la información de formato se guarda en una secuencia de 15 bits: 5 bits representan el nivel L, M, Q o H, 3 bits el patrón de máscara y 7 son de corrección de errores. Por su parte, la información de la versión se almacena en una secuencia de hasta 18 bits, dependiendo del tamaño. Por ejemplo, en las versiones 1 a 6 ni siquiera están presentes estos datos.

La colocación de los codewords dentro de la región de codificación de la matriz es un proceso crucial. Cada codeword se representa por una agrupación de 8 bits denominada carácter de símbolo, que pueden ser regulares o irregulares. Los caracteres regulares representan la mayoría de los codewords y se organizan en bloques de 2 × 4 módulos dentro del símbolo. Los caracteres irregulares se emplean en situaciones especiales, como al cambiar de dirección o cerca de patrones de alineación y otras funciones.

El bit más significativo de cada codeword, conocido como bit 7, se coloca en la primera posición disponible en el carácter de símbolo. Dependiendo de la dirección de colocación, este bit ocupará diferentes posiciones en el bloque de módulos. Si la dirección es hacia arriba, el bit más significativo se colocará en el módulo inferior derecho del bloque regular; si la dirección es hacia abajo, se colocará en el módulo superior derecho. Para caracteres irregulares, el posicionamiento de los bits dependerá de la forma concreta del bloque.

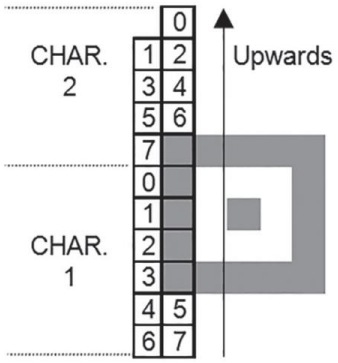


Figura 20. Caracteres de símbolo irregulares [5]

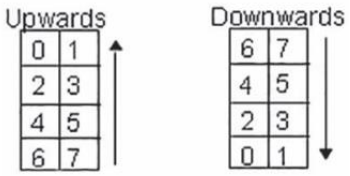


Figura 19. Ubicación de bits en un carácter de símbolo regular, según dirección [5]

Para ir colocando los caracteres, la cuadrícula del código se divide en columnas de dos módulos de ancho. Siguiendo estas columnas, las codewords se van ubicando en una trayectoria en zigzag, comenzando desde la esquina inferior derecha de la cuadrícula y moviéndose alternativamente hacia arriba y hacia abajo, de derecha a izquierda. Primero se sitúan todos los caracteres que contienen el mensaje original, y luego los generados como mecanismo de corrección de errores.

Además, si durante la aplicación del mecanismo de Reed-Solomon se ha dividido la cadena en bloques, los codewords de cada grupo se intercalan en el código para maximizar la capacidad de corrección de errores. De esta forma, se evita que una pequeña área dañada de la cuadrícula afecte a un bloque completo de datos.

Por último, cuando el número total de módulos destinados a la codificación de datos es mayor que los utilizados por el mensaje, se utilizan bits de relleno (*Remainder Bits*) con valor 0 para completar la capacidad del código.

En la Figura 21 que ocupa gran parte de esta página está representada la estructura de un QR versión 7-H. Contiene un total de 196 codewords (1568 bits) en 5 bloques, con 66 bytes destinados a almacenamiento de datos y 130 a la corrección de errores.

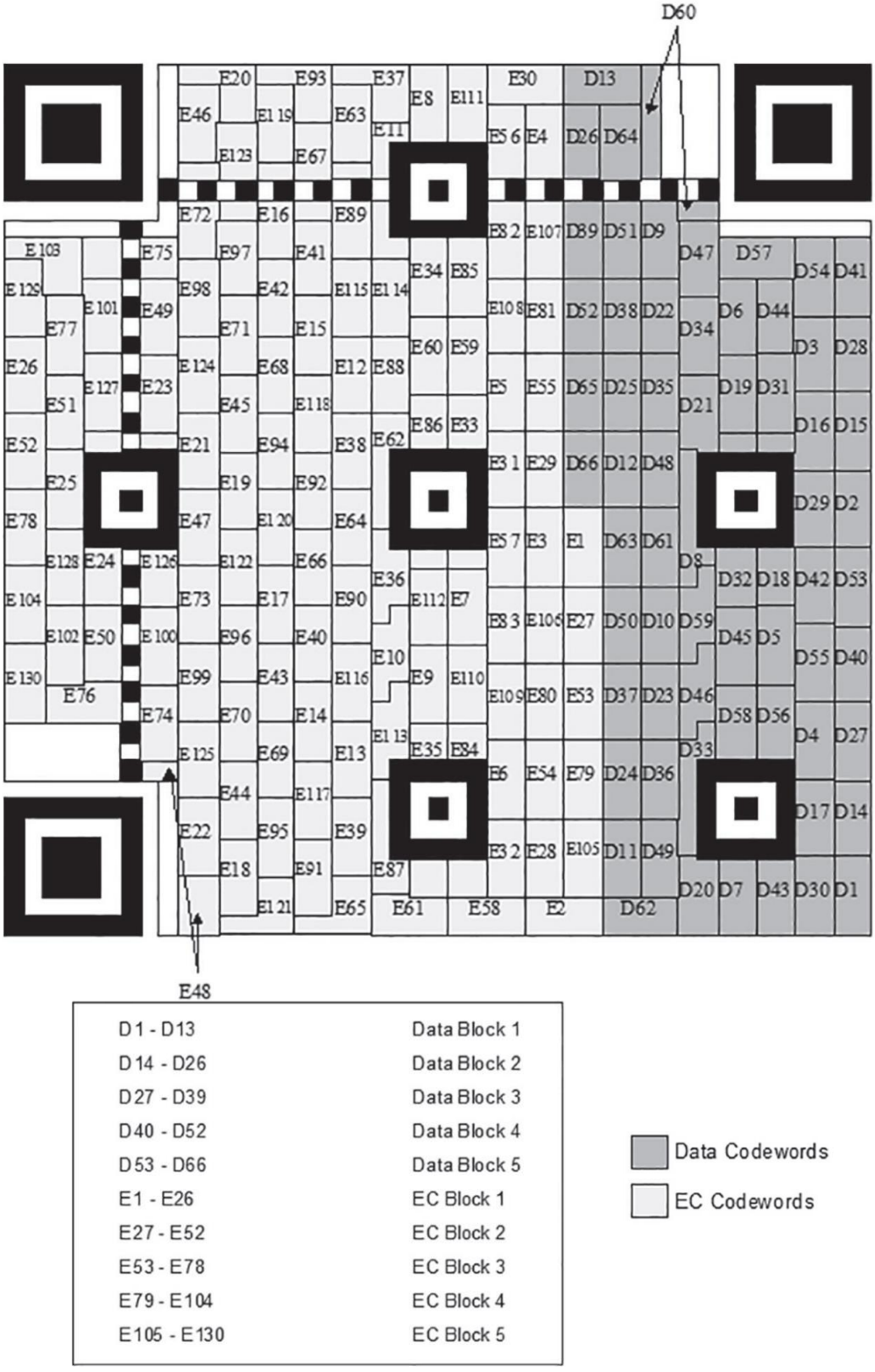


Figura 21. Estructura de un código QR versión 7-H [5]

2.4.4.4 Enmascaramiento o "data masking" de la cuadrícula

El último paso en el proceso de generación de un código QR es aplicar un patrón de enmascaramiento o *data masking*. Estos métodos son utilizados para mejorar la legibilidad y confiabilidad del código, evitando patrones problemáticos de módulos que podrían dificultar la lectura por parte de un escáner.

Existen un total de 8 máscaras aplicables a la matriz de datos de una etiqueta QR, como se puede apreciar en la Figura 22. Se aplican mediante una operación XOR entre la cuadrícula del código QR generada anteriormente (excluyendo las áreas reservadas formato, versión y patrones de lectura) y cada uno de los patrones de enmascaramiento disponibles.

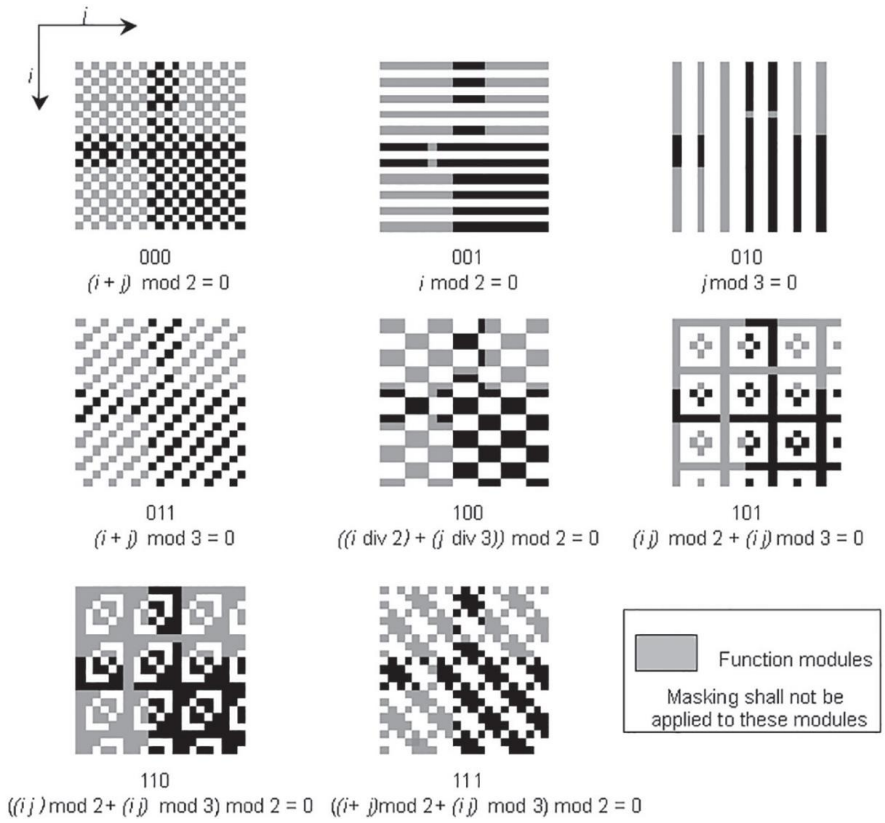


Figura 22. Patrones de enmascaramiento para códigos QR [5]

Después de aplicar de uno en uno todos los patrones de enmascaramiento, se evalúan los resultados para determinar cuál produce el código QR más equilibrado y fácil de leer. Esta evaluación se realiza mediante un sistema de penalización que asigna puntos por características indeseables en el patrón resultante.

Cada tipo de agrupación de módulos que puede inducir a error está especificado en la norma y ponderado. Entre estas características no deseadas se encuentran: secuencias repetitivas de un mismo módulo, patrones similares a los de posicionamiento o alineamiento, desbalance en la proporción de

módulos oscuros y claros o concentraciones locales altas de módulos iguales. Para cada uno de los 8 casos, se calcula un número que representa el número de defectos de la matriz y se escoge la opción con el menor valor.

Una vez aplicado el patrón de enmascaramiento elegido, se puede finalmente codificar el número que representa la máscara elegida, e incluir esta información dentro de la parte del código dedicada a la versión. Con esto, se completa la totalidad de la cuadrícula del código QR, que ya ha adoptado su forma final reconocible por cualquier escáner.

2.4.5 Decodificación de códigos QR

La decodificación de un código QR sigue pasos inversos al proceso de codificación. Dado que ya se ha explicado el proceso de codificación en detalle, se será menos específico en la explicación de este apartado. De esta forma, a continuación se describirá el flujo general del proceso de decodificación de un código QR:

- 1. Localización y captura de la imagen del símbolo: Se debe localizar y obtener una imagen del código QR. En ella, se identifican los módulos oscuros y claros como una matriz de bits "0" y "1".

- 2. Lectura de la información de formato: Se decodifica la información de formato, realizando ajuste de errores si es necesario. Con estos datos, son conocidos el patrón concreto de enmascaramiento y el nivel de corrección de errores utilizados.

- 3. Lectura de la información de la versión: Cuando es aplicable, se lee la información de la versión para confirmar el tamaño de la cuadrícula.

- 4. Eliminación del enmascaramiento de datos: Se elimina el enmascaramiento de datos aplicando una operación XOR entre el patrón de bits de la región de codificación y el patrón de enmascaramiento de datos cuya referencia se extrajo de la información de formato.

- 5. Lectura de los caracteres del símbolo: Los caracteres del símbolo se leen según las reglas de colocación para la variante correspondiente y se restauran los codewords de datos y corrección de errores del mensaje.

- 6. Detección y corrección de errores: Se utilizan los codewords de corrección de errores para validar la información extraída de la cadena de datos. Si se detecta algún error, se corrige con la información de los bytes de error.

- 7. Decodificación y salida de los caracteres de datos: Finalmente, se decodifican los caracteres de datos de acuerdo con el modo en uso y se obtiene el resultado final.

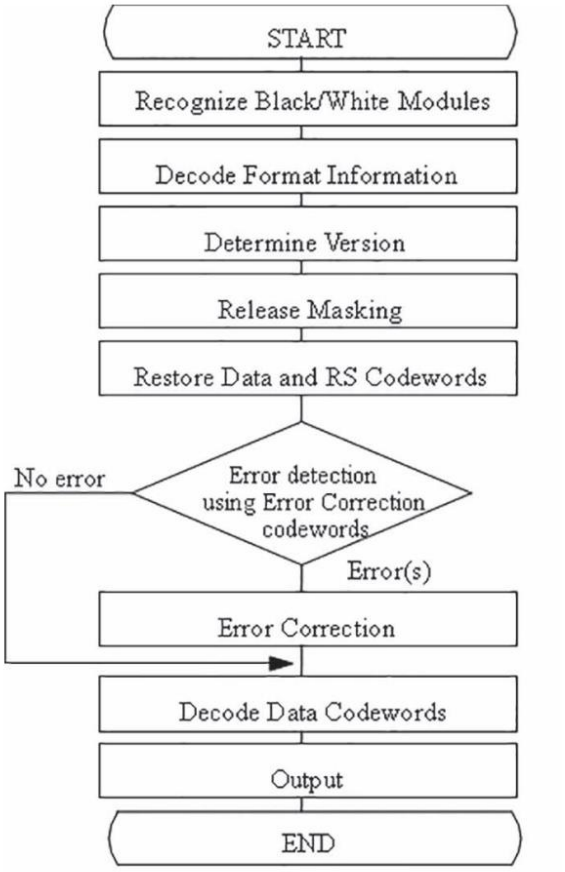


Figura 23. Diagrama de flujo de proceso de la decodificación de un código QR [5]

Para más detalle, se puede consultar el algoritmo de referencia propuesto en la norma ISO/IEC 18004 [5]. En él se explican a nivel más básico cada uno de los pasos descritos anteriormente. Todos los escáneres utilizan un método de decodificación de códigos QR como el descrito, incluyendo la biblioteca utilizada en este proyecto.

3 DESARROLLO DEL PROYECTO

En este apartado se hablará de los pasos seguidos hasta la consecución de los objetivos y se explicará en detalle la solución obtenida. Así, se comenzarán explicando las etapas previas al desarrollo de cualquier código, a continuación se comentarán los aspectos más relevantes de programas intermedios creados en el progreso del proyecto, y por último se detallará el código final creado para la detección y lectura de códigos QR utilizando el ESP32 ya descrito.

3.1 Establecer el entorno de desarrollo

El primer paso en un proyecto como este es instalar y configurar los programas necesarios para establecer una conexión con el ESP32-S3 WROOM-1, desarrollar un programa de código y comprobar que cumple su función adecuadamente.

Como ya se ha visto, el programa que se va a utilizar inicialmente para desarrollar, depurar y cargar el código es Arduino IDE. Se instala la última versión desde la página oficial de Arduino. En este caso, será Arduino IDE 2.3.2 para Windows.

A continuación siguiendo el manual que acompaña al ESP32-S3 WROOM-1, se instala el driver CH343. Este es un convertidor entre el protocolo de comunicación serie UART y el USB CDC, utilizados para la comunicación entre el módulo ESP32 y la computadora. Para obtener este programa, Freenove (el fabricante del chip) redirige al usuario a la página web de Nanjing Qinheng Microelectronics Co. [6]. Allí, se descarga la última versión de CH343 para Windows.

Relacionándolo con lo visto en el apartado 2.2.5 sobre comunicación con la placa, este driver permite que el ordenador suba códigos y reciba datos en formato UART, a pesar de que trabaje internamente con USB CDC. Solo como ejemplo para entender mejor el propósito del driver, con la otra opción no sería CH343 el que trabajara, sino que la información llegaría al ESP32 en formato USB CDC y sería el convertidor interno del módulo quien la transformara en el formato UART que el procesador entiende. Ver Figura 5, página 14.

Como último paso de la puesta a punto del IDE, se debe de configurar al programa para trabajar de forma específica con el microcontrolador ESP32-S3. Para ello, en el gestor de tarjetas adicionales del programa añadimos el enlace a un documento JSON creado por Espressif Systems [7]. Este archivo contiene

la información sobre las tarjetas compatibles, las versiones disponibles y los enlaces a los archivos necesarios para la instalación de las herramientas específicas del ESP32.

Tras hacer esto, se podrá instalar el Gestor de Tarjetas ESP32 en su última versión, a través del propio programa Arduino IDE. Al hacerlo, se incorporan al entorno de desarrollo todas las bibliotecas básicas, herramientas y configuraciones necesarias para programar las placas ESP32 en Arduino IDE. Dentro de estas librerías se incluye *esp_camera*, que será muy útil para trabajar con la cámara posteriormente.

Si todo esto se ha realizado correctamente, se podrá ver que en el menú de "Herramientas", "Gestor de Tarjetas", de Arduino IDE, ha aparecido una nueva carpeta: "esp32". Aquí, se debe seleccionar la opción "*ESP32-S3 Dev Module*", para trabajar de forma específica con el microcontrolador del cual se dispone para este proyecto.

Este ha sido el último paso de la puesta a punto del entorno de desarrollo. Ahora, Arduino IDE ha ajustado las configuraciones de compilación y carga de código para ser compatibles con las características y capacidades concretas del ESP32-S3 WROOM-1. Esto incluye la configuración de los parámetros de conexión, la velocidad del puerto serie y otras configuraciones específicas del hardware.

3.1.1 Comprobaciones. "CameraWebServer"

Una vez establecido el entorno de desarrollo, se debe comprobar su funcionamiento, así como la conexión con el ESP32-S3 WROOM-1 y el correcto funcionamiento de todas las características del dispositivo.

Primero, se utilizan programas con propósitos muy sencillos y concretos, como apagar y encender el LED incorporado al módulo ("Blink") y transmitir unos pocos caracteres desde el ESP32 al monitor serie del IDE. De esta forma, se comprueba con éxito la carga de programas a la tarjeta y la capacidad de transmisión de datos desde la misma a la computadora.

A continuación, se comprueba la funcionalidad de la cámara mediante el programa "CameraWebServer". Este programa es un ejemplo didáctico que se ha precargado a Arduino IDE al descargar el paquete de bibliotecas de ESP32, como parte de la librería *esp_camera*. Así, el programa viene ya completo y solo hay que ajustar un par de parámetros.

En sí, el código de "CameraWebServer" sirve para obtener la dirección IP del ESP32-S3 WROOM-1 a través del monitor serial. Esta dirección a continuación es utilizada para establecer una conexión mediante Wifi entre el

módulo y un servidor web en el que se puede visualizar la imagen de la cámara y alterar sus parámetros, como se puede ver en la Figura 24.

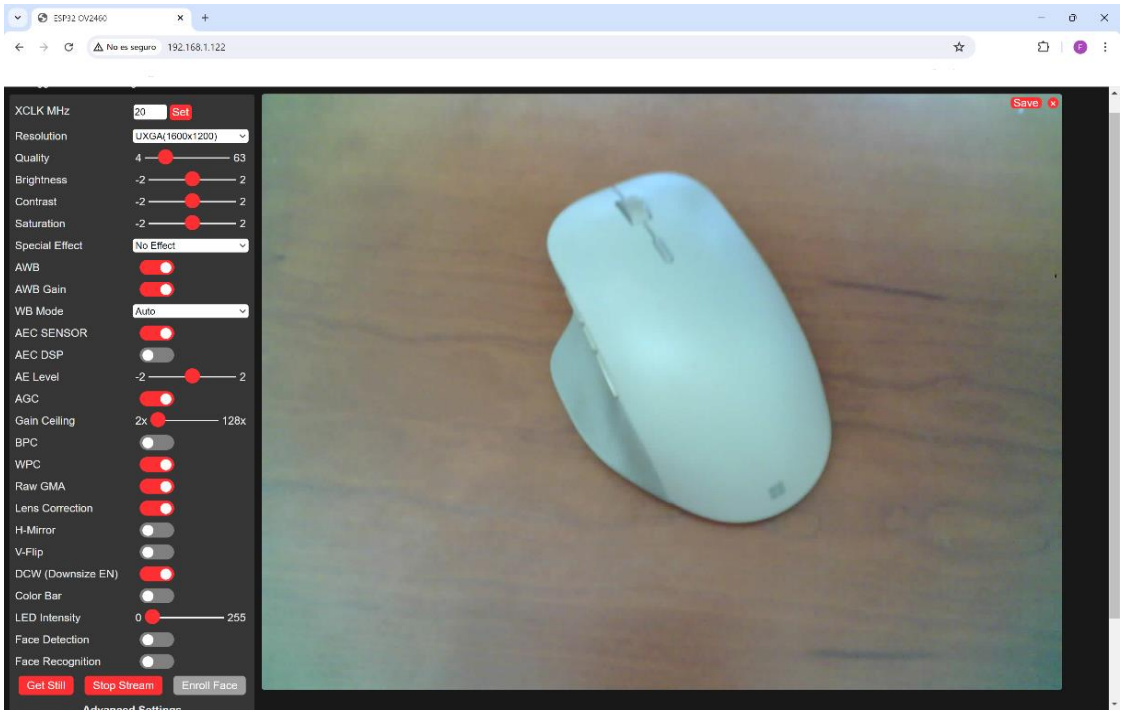


Figura 24. Ejemplo de la cámara en funcionamiento: "Camera Web Server"

En la parte superior de la imagen, se puede apreciar la dirección IP que ha devuelto el ESP32 a través del monitor serie y que nos lleva al servidor web. A la izquierda, se pueden ver los menús para alterar parámetros como la resolución de la imagen, brillo, contraste y otras características como el balance automático de blanco (AWB y AWB gain) y la orientación de la imagen.

También, en la parte baja del panel de control se pueden observar las opciones disponibles para transmitir vídeo en directo, tomar una instantánea e incluso incluye algunas opciones de reconocimiento facial que no serán utilizadas en este proyecto. La imagen de la derecha, de un ratón de ordenador, es una instantánea tomada durante una grabación de prueba.

Con esto, se ha comprobado que la cámara funciona y las posibilidades que ofrece en cuanto a alterar sus parámetros. Sin embargo, en este código se ha conseguido establecer una conexión con la cámara, pero la conexión únicamente se ha hecho entre el dispositivo y el servidor web. Es decir, ninguno de los datos de la imagen está realmente transmitiéndose a la computadora, sino que esta simplemente sirve como display para la información que recibe el servidor. Así, el siguiente paso en el proyecto será establecer una comunicación efectiva entre la cámara y el ordenador.

3.2 Recibir y manipular información de la cámara

Una vez hechas todas las comprobaciones sobre el funcionamiento del hardware, se continuará avanzando hasta el objetivo final del proyecto. La siguiente meta que se ha establecido es transmitir las imágenes capturadas por la cámara a la computadora, para su posterior manipulación.

Como forma de asegurar que la información se transmite y guarda sin errores, se pretende elaborar un programa que capture una instantánea cada dos segundos a través de la cámara OV2640, la almacene en el ESP32 y después la analice contando el número de píxeles con un valor de luminancia superior a un umbral establecido. Después, la placa enviará los datos al ordenador a través del cable USB, para que se vean reflejados en el monitor serie del IDE. También, se representará la información gráficamente.

El entorno de desarrollo utilizado continúa siendo Arduino IDE, haciéndose uso de la biblioteca *esp_camera*. Como ya es sabido, esta biblioteca ha sido instalada como parte del paquete de gestión de tarjetas ESP32.

La importancia de este código radica en que muchas de las funciones empleadas en él serán utilizadas posteriormente en el programa final. En estos comandos y variables será en lo que se centre la explicación de este apartado, para así hacer más liviana la comprensión del código final de lectura de códigos QR.

3.2.1 Explicación del código

El código empleado se puede consultar en su totalidad en los Anexos a este documento, con el nombre “CuentapixelesYUV”. Sus partes más relevantes para este proyecto se irán explicando a continuación.

3.2.1.1 Inicialización de la cámara

Antes de comenzar a redactar cualquier parte del código, se tienen que declarar las bibliotecas y otros archivos de apoyo utilizados:

```
#include "Arduino.h"              // Librería para funciones básicas: puerto serie...
#include "esp_camera.h"           // Librería para manejar la cámara


// Incluir definición de pines
#include "camera_pins.h"          // Archivo en el que se definen los valores para los pines de la cámara

// DECLARAR VARIABLES

// Configuración de la cámara
camera_config_t config;           // Variable que contendrá la configuración de la cámara
```

Figura 25. Declaraciones previas del programa “CuentapixelesYUV”

El documento *camera_pins.h* que se incluye es un archivo de cabecera en el que se definen valores para los pines que conectan a la cámara con el microprocesador. Estas declaraciones se incluyen en un archivo de apoyo separado del programa principal con el objetivo de compartimentar el código y de esta forma hacerlo más legible y fácil de utilizar aprovechar para otras aplicaciones. Para que Arduino IDE sea capaz de reconocer este archivo al compilar el código, deberá estar ubicado en la misma carpeta que el archivo *.ino* del código principal.

Los valores para los pines definidos en *camera_pins.h* se guardarán dentro de la variable *config*. Ha sido declarada como *camera_config_t*, siendo esta una estructura de datos propia de la librería *esp_camera*. Esta variable será utilizada para inicializar la cámara, como se puede ver en la Figura 26.

```
void setup() {
  // Inicializar la comunicación serie
  Serial.begin(115200);
  Serial.println("Inicializando...");

  // Configuración de la cámara
  config.ledc_channel = LEDC_CHANNEL_0;
  config.ledc_timer = LEDC_TIMER_0;
  config.pin_d0 = Y2_GPIO_NUM;
  config.pin_d1 = Y3_GPIO_NUM;
  config.pin_d2 = Y4_GPIO_NUM;
  config.pin_d3 = Y5_GPIO_NUM;
  config.pin_d4 = Y6_GPIO_NUM;
  config.pin_d5 = Y7_GPIO_NUM;
  config.pin_d6 = Y8_GPIO_NUM;
  config.pin_d7 = Y9_GPIO_NUM;
  config.pin_xclk = XCLK_GPIO_NUM;
  config.pin_pclk = PCLK_GPIO_NUM;
  config.pin_vsync = VSYNC_GPIO_NUM;
  config.pin_href = HREF_GPIO_NUM;
  config.pin_sccb_sda = SIOD_GPIO_NUM;
  config.pin_sccb_scl = SIOC_GPIO_NUM;
  config.pin_pwdn = PWDN_GPIO_NUM;
  config.pin_reset = RESET_GPIO_NUM;
  config.xclk_freq_hz = 20000000;

  config.pixel_format = PIXFORMAT_YUV422;                  // Formato de píxel
  config.frame_size = FRAMESIZE_CIF;                       // Tamaño de la imagen
  config.fb_location = CAMERA_FB_IN_DRAM;                  // Ubicación de la captura
  config.fb_count = 1;                                     // Número de frame buffers

  Serial.println("Cámara configurada");

  // Inicializar la cámara
  esp_err_t error = esp_camera_init(&config);       // error recoge el resultado de inicializar la cámara
  if (error != ESP_OK) {                            // Si inicializado adecuadamente, error = ESP_OK
    Serial.printf("Error al inicializar la cámara 0x%x", error);
    return;
  }
  Serial.println("Cámara inicializada correctamente");
}
```

Figura 26. Función setup() del programa "CuentapixelesYUV"

Ya dentro de la función *setup* y después de inicializar el puerto serie, se rellena la variable *config* con todos los datos que han sido definidos en *camera_pins.h*. Esta información se utiliza para determinar los canales de transmisión de datos entre la cámara y la placa, sintonizar los relojes internos

de ambos, elegir donde se guardan los fotogramas capturados y otras cuestiones básicas para la interconexión de los dispositivos. Para seleccionar estos valores es necesario consultar las hojas de especificaciones de la cámara OV2640 [3] y del ESP32-S3 WROOM-1 [2].

La variable *config* también se utiliza para elegir el formato y la resolución de la imagen. En este caso se optará por el formato YUV422, que permite aislar la información de luminosidad del pixel aun manteniendo información cromática. También se realizaron pruebas satisfactorias con los formatos RGB y en escala de grises. La resolución elegida será CIF (400x296), la misma que se utilizará posteriormente para reconocimiento de códigos QR.

Con sus datos ya rellenados, la variable *config* se utiliza para inicializar la cámara a través de la función *esp_camera_init*. Esta función recibe como input la ubicación de una variable de tipo *camera_config_t* y devuelve una de tipo *esp_err_t* que se utiliza para verificar el proceso. Tras esto, se puede pasar a la función *loop*.

```
void loop() {
  // Capturar una imagen
  camera_fb_t *fb = esp_camera_fb_get();                    // Captura una imagen y guardar el puntero *fb
  if (!fb) {
    Serial.println("Error al capturar la imagen");          // Comprobar la captura
    return;
  }

  // Contar píxeles según la luminancia (Y)
  int umbralLuminancia = 100;                                 // Umbral de luminancia
  int contadorPI = 0;
  uint8_t *buffer = fb->buf;
  for (int i = 0; i < fb->width * fb->height * 2; i = i + 2) {    // Recorrer cada píxel
    uint8_t y = buffer[i];                                    // Extraer componente Y (luminancia)
    if (y > umbralLuminancia) {                               // Comparar con el umbral
      contadorPI++;
    }
  }

  // FUNCIONES PARA EL TRATAMIENTO DE DATOS
  // [...]

  // Liberar el frame buffer
  esp_camera_fb_return(fb);

  delay(intMuestreo * 1000); // Espera antes de la siguiente captura
}
```

Figura 27. Función loop del programa "Cuentapixeles"

En esta parte del código es donde se capturan y analizan las imágenes. Para lo primero, se utiliza la función *esp_camera_fb_get*, que devuelve la estructura de datos *fb* (*frame buffer*). Como se ha especificado en la variable *config*, la imagen en sí se ubica en la DRAM, y el frame buffer únicamente indica las posiciones de memoria ocupadas (un puntero), además de contener otra información relevante como la resolución de la instantánea tomada.

A continuación, se extrae de *fb* el puntero a la imagen (*fb->buf*) y se pasa a otro puntero (*imagen*) . Este es utilizado en un bucle for para comparar cada píxel de la imagen con el umbral de luminancia establecido. Dado que cada píxel en formato YUV422 emplea un byte para la luminancia y otro para la crominancia, extrayendo de la ubicación de la imagen un byte cada dos posiciones de memoria se consigue aislar la información de luminosidad.

El resultado de cada comparación se va almacenando en un contador hasta que se llega al último píxel. Después, este contador con el número de píxeles que superan el umbral de luminancia será reflejado en el monitor serie. También, para poner en perspectiva los datos, se calculan otras variables como la media de píxeles iluminados en el último minuto y valores máximos y mínimos. Para ello, se utilizan funciones que no se han incluido en la Figura 27 por no ser relevantes para el proyecto.

El último paso es liberar el frame buffer en el que se ha almacenado la imagen, dejando espacio para la que va a ser capturada en la siguiente iteración de *loop*. Además, antes de terminar el bucle, se añade un retardo de dos segundos para facilitar la lectura de los resultados en el monitor serie.

3.2.2 Resultados obtenidos

Una vez finalizado, Arduino IDE compila el código del programa y lo traslada a la memoria flash del ESP32. Así, la placa ejecutará las instrucciones descritas y enviará los datos oportunos al ordenador. Esta información será recogida en el monitor serie del entorno de desarrollo, añadiéndose nuevas líneas con cada iteración:

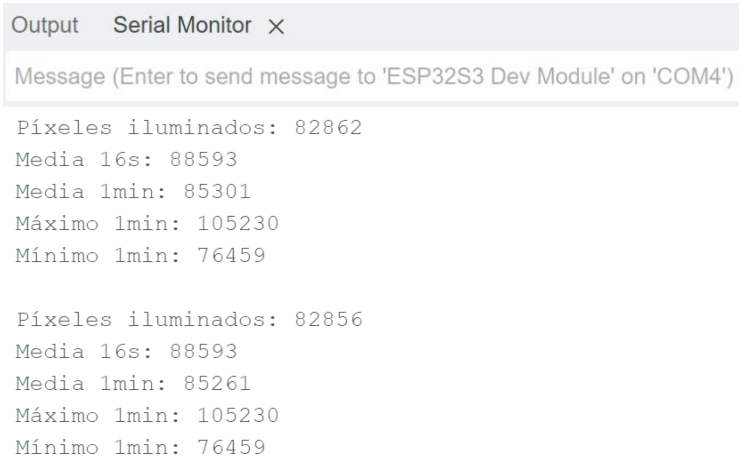


Figura 28. Monitor serie durante la ejecución del programa "CuentapixelesYUV"

Además, para visualizar los cambios en la imagen con el tiempo, se puede utilizar la herramienta para trazar gráficos que viene integrada en el propio Arduino IDE. En este caso, será necesario modificar ligeramente el código para

garantizar su funcionamiento, haciendo que la placa únicamente envíe al ordenador un valor numérico por cada iteración del programa. Con esta información, el trazador de gráficos generará una representación de la cantidad de píxeles iluminados en tiempo real.

Por desgracia, Arduino IDE no permite ninguna opción de personalización del gráfico generado automáticamente, haciéndolo algo engorroso y poco vistoso. Para un diagrama más depurado, se utilizará un código Python creado en PlatformIO. Este programa leerá la información que llega al puerto serie y la representará en un gráfico con una escala definida para los ejes y otras prestaciones.

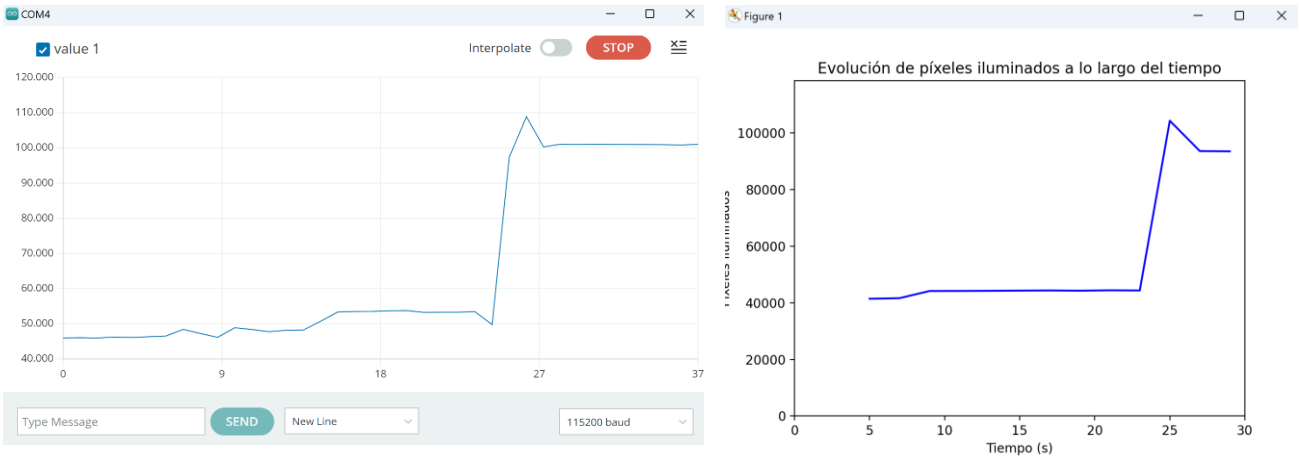


Figura 29. Respuesta de "CuentaPixeles" en Arduino IDE (izq.) y PlatformIO (dcha.)

Cambiando la orientación de la cámara hacia una fuente de luz, el número de píxeles que superan el umbral de luminancia debería subir, como así reflejan los gráficos. De hecho, se puede observar que durante un momento casi la totalidad de los píxeles de la imagen (400x296 = 118400 píxeles) superan dicho umbral. Solo para concretar, la información representada no es exactamente la misma en ambos diagramas, debido a que están tomados en ensayos diferentes. Esto es a razón de que los dos programas no pueden acceder a la vez al mismo puerto serie.

Aunque de forma algo rudimentaria, de esta forma queda demostrado el correcto funcionamiento de las funciones de la librería *esp_camera*. Aparentemente, la imagen es capturada, almacenada y procesada sin problemas.

Ahora sí, habiendo comprobado el hardware y software de la placa y después de familiarizarse con la gestión de la cámara, se puede pasar a desarrollar el código final del proyecto, el lector de códigos QR.

3.3 Reconocer y decodificar códigos QR

En este apartado se desarrollará la explicación del programa final del proyecto, destinado a la detección y lectura de códigos QR. El objetivo será entender la estructura del código, su funcionamiento y las decisiones tomadas durante su desarrollo. El contenido completo del mismo puede ser consultado en los Anexos con el nombre "Lector_QR". Los resultados obtenidos, así como varias pruebas sobre su desempeño, se relatarán en el apartado 4 de este proyecto.

3.3.1 Archivos del programa

En contraste con lo visto anteriormente, este programa se ha desarrollado utilizando el entorno de desarrollo PlatformIO, junto con el editor de código VS Code. Como ya se ha explicado antes, este cambio se debe a las herramientas más avanzadas que ofrece esta plataforma, así como la utilización de la librería *quirc* para detección de códigos QR, no disponible en Arduino IDE.

Ya se explicó la arquitectura de un proyecto de PlatformIO como parte del marco teórico, con lo que ahora se podrá entender su aplicación en este proyecto. Así, los archivos que componen el proyecto "Lector_QR" son los siguientes:

- ***platformio.ini***: Archivo de configuración general del programa, cuyo contenido se puede ver en la Figura 30. Las características que en él se definen sirven para que PlatformIO ponga a disposición del usuario las herramientas específicas para trabajar con el ESP32-S3 WROOM-1U. También, se especifican las características de conexión entre la placa y la computadora, y se establece la inclusión de la biblioteca *esp_camera*.

```
[env:esp32-s3-devkitc-1]                ; Entorno de desarrollo específico
platform = espressif32                  ; Plataforma de hardware utilizada
board = esp32-s3-devkitc-1              ; Placa específica en uso (ESP32-S3 WROOM-1U)
framework = arduino                     ; Disponibles bibliotecas generales de Arduino

upload_speed = 115200                   ;  Velocidad de transmisión de datos entre el
monitor_speed = 115200                  ; ordenador y la placa: 115200 bits / segundo
upload_port = COM4                      ;  Puerto a través del que se transmite la
monitor_port = COM4                     ; información

lib_deps = espressif/esp32-camera@^2.0.4    ; Librería esp_camera
```

Figura 30. Archivo de configuración platformio.ini

- **Librería *quirc***: La librería *quirc* incluye todas las funciones utilizadas para analizar las imágenes capturadas por la cámara, permitiendo detectar y decodificar códigos QR estándar del modelo 2. Los archivos de esta biblioteca se han descargado del repositorio web GitHub [4] e incluido en la carpeta *lib*

```
#define PWDN_GPIO_NUM  -1
#define RESET_GPIO_NUM -1
#define XCLK_GPIO_NUM  15
#define SIOD_GPIO_NUM  4
#define SIOC_GPIO_NUM  5

#define Y2_GPIO_NUM 11
#define Y3_GPIO_NUM 9
#define Y4_GPIO_NUM 8
#define Y5_GPIO_NUM 10
#define Y6_GPIO_NUM 12
#define Y7_GPIO_NUM 18
#define Y8_GPIO_NUM 17
#define Y9_GPIO_NUM 16

#define VSYNC_GPIO_NUM 6
#define HREF_GPIO_NUM  7
#define PCLK_GPIO_NUM  13
```

Figura 31. Archivo de apoyo camera_pins.h

del proyecto. De esta forma, el código principal tiene acceso a todo el contenido de la librería cada vez que es compilado con PlatformIO.

- **Archivo de apoyo *camera_pins.h*:** Se trata del mismo archivo de cabecera utilizado en el código anterior. En él se definen valores para los pines y otras variables que se utilizan para cuestiones básicas en la interconexión de la cámara y el microprocesador (especificar los canales de transmisión de datos, etc.). Para poder ser accesible para el programa principal, *camera_pins.h* se guarda en la carpeta *include.*

- ***main.cpp***: Este archivo se encuentra en la carpeta *src y* contiene el código fuente principal del proyecto. Su contenido se detallará en el siguiente apartado

3.3.2 Explicación del código

El código fuente principal del programa (*main.cpp*) contiene las instrucciones que el ESP32 deberá seguir para capturar imágenes, analizarlas en busca de códigos QR, decodificarlos y mostrar su contenido al usuario. Su contenido completo se puede encontrar en los Anexos, a continuación se irá explicando por partes.

3.3.2.1 Declaraciones previas

```
1   #include "esp_camera.h"            // Librería para cámara
2   #include "quirc.h"                 // Librería para códigos QR
3   #include "Arduino.h"               // Funciones básicas de Arduino
4
5   #include "camera_pins.h"
6
7   // Configuración de la cámara
8   camera_config_t config;
9
10  // Variables globales para evitar uso excesivo de la pila
11  camera_fb_t *fb;        // Frame buffer de la cámara
12  struct quirc *qr;       // Datos y buffers para procesar los códigos
13
14  // Declaración de la función loopPersonalizado
15  void loopPersonalizado(void *pvParametros);
16
17  // Definir el tamaño de pila deseado para la nueva tarea loopPersonalizado
18  #define tam_pila 28000  // Tamaño de la pila en bytes
```

Figura 32. Declaración de bibliotecas y otras variables

En primer lugar, se declaran las librerías *esp_camera.h* (configurar la cámara, sacar capturas…), *quirc.h* (leer y decodificar códigos QR) y *Arduino.h* (funciones básicas como controlar el puerto serie…). También, se incluye el ya mencionado archivo de apoyo *camera_pins.h* y se declara la variable *config*, que contendrá los parámetros para la inicialización de la cámara al igual que se hacía en el programa "CuentapixelesYUV".

A continuación, se declaran como variables globales los punteros que van a ser utilizados más tarde. Como ya se explicó, *fb* apunta al frame buffer de la cámara, que contendrá el fotograma tomado en el formato que se indique, así como otros datos de la imagen, por ejemplo su resolución.

Por otra parte, la estructura de datos a la que dirige el puntero ***qr*** es el objeto central en torno al que gira la librería *quirc*. Contiene varios datos y buffers necesarios para: alojar la imagen que se va a analizar; almacenar información sobre los códigos QR detectados, como sus posiciones y el contenido de la cuadrícula; decodificar estos códigos y guardar el resultado. La forma de trabajar con esta variable se verá más adelante.

El razonamiento detrás de declarar estos punteros al principio del código y no cuando realmente se crean los datos contenidos en ellos es ahorrar espacio en la pila de datos. Las variables globales se almacenan por defecto en la memorio *heap*, mientras que declarar una variable dentro de una función la almacena dentro de la pila asociada a dicha función.

En este programa, se utilizan funciones que hacen un uso extensivo de la pila de datos, especialmente las funciones que decodifican los códigos QR de las imágenes. Esto se comprobó al ejecutar el programa las primeras veces, ya que la lectura de un código QR siempre resultaba en un desbordamiento de la pila de datos que impedía al microprocesador funcionar adecuadamente.

Por este motivo, se tratará en todo momento de ahorrar espacio en la pila de datos, guardando las variables de cierto tamaño dentro de la memoria *heap*, y tomando otras disposiciones como la que se puede ver al final de la Figura 31. En vez de utilizar la función *loop* que viene predefinida en el programa, se declara una función ***loopPersonalizado*** en la que se podrá ajustar el tamaño de la pila. Así, se pasan de los 8 KB del bucle estándar, a los 28 KB que se asignarán con la variable *tam_pila*.

Las comprobaciones para decidir este tamaño de la pila fueron realizadas con una función que utilizaba comandos de la librería *esp_task_caps.h* para medir la memoria restante en la pila de datos y en el resto de la DRAM (memoria *heap*). Recurriendo a ella antes y después de realizar alguna tarea, se puede comprobar el uso que hace de cada tipo de memoria. Aunque esta función ya ha sido eliminada del programa tras cumplir su función durante el desarrollo

del código, en la Figura 33 a continuación, se tiene un ejemplo de su implementación y uso.

```
#include "esp_heap_caps.h"


void usoMemoria(const char* etiqueta) {
    // Memoria libre en el heap
    size_t heapLibre = heap_caps_get_free_size(MALLOC_CAP_DEFAULT);

    // Memoria mínima disponible en la pila desde el inicio de la tarea actual
    size_t pilaLibre = uxTaskGetStackHighWaterMark(NULL);

    Serial.printf("[%s] Heap libre: %u bytes, Pila disponible: %u bytes\n", etiqueta, heapLibre, pilaLibre * sizeof(StackType_t));
}

void setup() {
    // Imprimir el uso de memoria inicial
    usoMemoria("setup");
```

Figura 33. Función utilizada para medir el uso de memoria

3.3.2.2 Función setup()

Dada su longitud, para su estudio en este documento se dividirá la función *setup* en las Figuras 34 y 35.

```
21  void setup() {
22      Serial.begin(115200);
23      Serial.println("Inicializando...");
24
25      // Configuración de la cámara
26      config.ledc_channel = LEDC_CHANNEL_0;
27      config.ledc_timer = LEDC_TIMER_0;
28      config.pin_d0 = Y2_GPIO_NUM;
29      config.pin_d1 = Y3_GPIO_NUM;
30      config.pin_d2 = Y4_GPIO_NUM;
31      config.pin_d3 = Y5_GPIO_NUM;
32      config.pin_d4 = Y6_GPIO_NUM;
33      config.pin_d5 = Y7_GPIO_NUM;
34      config.pin_d6 = Y8_GPIO_NUM;
35      config.pin_d7 = Y9_GPIO_NUM;
36      config.pin_xclk = XCLK_GPIO_NUM;
37      config.pin_pclk = PCLK_GPIO_NUM;
38      config.pin_vsync = VSYNC_GPIO_NUM;
39      config.pin_href = HREF_GPIO_NUM;
40      config.pin_sccb_sda = SIOD_GPIO_NUM;
41      config.pin_sccb_scl = SIOC_GPIO_NUM;
42      config.pin_pwdn = PWDN_GPIO_NUM;
43      config.pin_reset = RESET_GPIO_NUM;
44      config.xclk_freq_hz = 20000000;
45
46      config.pixel_format = PIXFORMAT_GRAYSCALE;    // Formato de píxel en escala de grises
47      config.frame_size = FRAMESIZE_CIF;            // Tamaño de la imagen (400x296)
48      config.fb_location = CAMERA_FB_IN_DRAM;       // Ubicación de la imagen (DRAM)
49      config.fb_count = 1;                          // Número de frame buffers
50
51      Serial.println("Cámara configurada");
52
53      // Inicializar la cámara
54      esp_err_t error = esp_camera_init(&config);
55      if (error != ESP_OK) {
56          Serial.printf("Error al inicializar la cámara 0x%x\n", error);
57          return;
58      } else{
59          Serial.println("Cámara inicializada correctamente");
60      }
```

Figura 34. Inicialización de la cámara

En la Figura 34, se observan las líneas correspondientes a la inicialización del puerto serie y la cámara. Como se vio en "CuentapixelesYUV", los valores definidos en *camera_pins.h* se almacenan en la variable *config*, que después es utilizada para sintonizar la cámara OV2640.

En este caso, el formato elegido es escala de grises, dado que este es el único compatible con la librería *quirc*. El razonamiento es sencillo, como los códigos QR están compuestos por módulos oscuros o claros (normalmente blancos o negros, aunque puede haber otros colores), no es necesaria información cromática para analizarlos. Por lo tanto, el formato escala de grises conserva la información relevante (la luminancia) para el decodificador, utilizando el formato más compacto que permite la cámara OV2640, con solo 8 bits para cada píxel. Por otro lado, la elección de la resolución CIF (400x296) se justificará en el apartado 3.3.3.

```
62      // Inicializar el decodificador QR
63      qr = quirc_new();                           // Se crea el obeto qr, de tipo struct quirc
64      if (!qr) {
65          Serial.println("Error al crear el decodificador QR");
66          return;
67      } else {
68          Serial.println("Decodificador QR inicializado correctamente");
69      }
70
71      // Crear una nueva tarea para ejecutar el loop con mayor tamaño de pila
72      xTaskCreatePinnedToCore(
73          loopPersonalizado,       // Función que implementa la tarea
74          "loopPersonalizado",     // Nombre de la tarea
75          tam_pila,                // Tamaño de la pila para la tarea
76          NULL,                    // Parámetro que se pasa a la tarea (en este caso, ninguno)
77          1,                       // Prioridad de la tarea
78          NULL,                    // Puntero a la tarea creada (en este caso, no se necesita)
79          1                        // Núcleo en el que se ejecutará la tarea (core 1)
80      );
81  }
```

Figura 35. Inicialización del decodificador y creación de la tarea personalizada

Después de inicializar la cámara, se utiliza la función *quirc_new* para generar la estructura de datos del decodificador y guardar su ubicación en el puntero *qr* declarado anteriormente.

El último paso de la función *setup* es crear la función en la que se va a ejecutar el resto del código en bucle, con un tamaño de pila de datos mayor del habitual. Para ello, se utiliza la función *xTaskCreatePinnedToCore*, integrado de manera predeterminada en el framework de ESP-IDF (*Espressif IoT Development Framework*) . Este comando se utiliza comúnmente para manejar tareas concurrentes en placas Arduino.

Por lo tanto, esta función permite controlar en qué núcleo (core) se ejecuta una tarea, para lo que se elegirá el valor por defecto, el núcleo 1 de los dos disponibles en este microprocesador (0 y 1). También, se le asigna prioridad 1, aunque esto no será importante porque *loopPersonalizado* no va a competir

con otras tareas, y se desestima utilizar inputs o punteros para la función. Lo que sí es importante es especificar el tamaño de la pila de datos, que ha sido declarado en la variable *tam_pila*: 28 KB.

3.3.2.3 Bucle loopPersonalizado

Es en la función *loopPersonalizado* que se ha generado donde se incluyan las funciones destinadas a detectar y decodificar códigos QR. Nuevamente, la tarea se trasladará a este documento en varias Figuras: 36, 37 y 38.

```
83  // Definición de la función loopPersonalizado
84  void loopPersonalizado(void *pvParametros) {
85      while (true) {
86          // Capturar una imagen
87          fb = esp_camera_fb_get();  // Se obtiene un puntero a la memoria dinámica (heap)
88          if (!fb) {
89              Serial.println("Error al capturar la imagen");
90              return;
91          } else {
92              Serial.println("Captura tomada");
93          }
94
95          // Redimensionar el decodificador para ajustarse al tamaño de la imagen
96          static int ult_ancho = 0;
97          static int ult_altura = 0;
98
99          if (fb->width != ult_ancho || fb->height != ult_altura) {
100             if (quirc_resize(qr, fb->width, fb->height) < 0) {
101                 Serial.println("Error al redimensionar el decodificador QR");
102                 esp_camera_fb_return(fb);
103                 return;
104             }
105             ult_ancho = fb->width;
106             ult_altura = fb->height;
107         }
108
109         // Obtener el buffer de la imagen en el decodificador
110         uint8_t *imagen = quirc_begin(qr, NULL, NULL);
111         if (!imagen) {
112             Serial.println("Error al obtener el buffer de imagen de quirc");
113             esp_camera_fb_return(fb);
114             return;
115         }
116
117         memcpy(imagen, fb->buf, fb->width * fb->height);  // Copiar los datos de la imagen
118
119         // Finalizar la captura de la imagen
120         quirc_end(qr);
121
122         // Liberar recursos
123         esp_camera_fb_return(fb);  // Liberar la memoria del framebuffer
```

Figura 36. Captura y procesamiento de la imagen

Lo primero que se puede observar es que la totalidad del contenido de *loopPersonalizado* está en un bucle *while* cuya condición siempre es cierta (*true*). De esta forma, la tarea se ejecutará una y otra vez siempre que el ESP32

esté encendido y no se llegue a un comando *return*, que se utiliza en caso error para detener el bucle.

Se comienza tomando una instantánea con el comando *esp_camera_get*, que se guarda en el buffer de imagen *fb*. A continuación, se debe redimensionar el objeto *qr* para hacerlo coincidir con el tamaño de la imagen.

Para ello, se utiliza la función *quirc_resize*, utilizando como argumentos el ancho (*fb->width*) y alto (*fb->height*) de la imagen capturada. Esta acción consume tiempo y espacio, por lo que solo se querrá utilizar en el caso de que el tamaño de la imagen no coincida con el del decodificador. Esto es comprobado con un condicional *if*, que hace uso de unas variables estáticas (*ult_ancho* y *ult_altura*) en las que el tamaño de la imagen en una iteración permanece guardado para la siguiente.

Dado que no se va a cambiar la resolución de las capturas una vez subido el código, podría pensarse que este redimensionamiento del decodificador debería incluirse en la función *setup* y así aplicarse únicamente al inicio del programa. Sin embargo, en ese caso no se podría hacer uso de los datos guardados en la variable *fb*, y el tamaño de imagen debería ser especificado a mano cada vez que hubiera un cambio. Es para evitar este cambio manual que se optó por incluir *quirc_resize* dentro de *loopPersonalizado* utilizando el condicional *if*.

Una vez comprobado que el redimensionamiento del decodificador ha sido satisfactorio, se utiliza la función *quirc_begin* para obtener el puntero *imagen*, que apunta a la zona de memoria donde el objeto *qr* espera la imagen que se quiere analizar. Esta función también serviría para extraer el alto y ancho esperados para el fotograma, pero como son datos ya conocidos se incluye en sus posiciones la constante *NULL*. Así, se indica que esos punteros no deben dirigirse a ninguna posición de memoria.

Seguidamente, se utiliza la función *memcpy* para trasladar la imagen desde las posiciones de memoria del en las que se ha capturado (*fb*), hasta las que el decodificador *qr* tiene reservadas para llevar a cabo el análisis. Esta función forma parte de la librería estándar *Arduino.h* y, como se puede ver, sirve para copiar de una ubicación a otra bloques de memoria de un tamaño especificado (en este caso, *fb->width* fb->height*).

Con esto, el objeto *qr* ya tiene acceso a la captura y se utiliza *esp_camera_fb_return* para liberar los recursos empleados en almacenarla en primera instancia (*fb*). Además, con la función *quirc_end* se pone en marcha el procesamiento de la imagen dentro de la objeto *qr*. A partir de este momento, se podrán comenzar a extraer de la estructura de datos *qr* los resultados del análisis.

```
125     // Contar el número de códigos QR encontrados
126     int num_codigos = quirc_count(qr);  // Se extrae el número
127
128     if (num_codigos == 0) {
129         Serial.println("No se detectaron códigos QR \n");
130     } else {
131         for (int i = 0; i < num_codigos; i++) {
132             // Ubicar struct quirc_code y struct quirc_data en la memoria dinámica (heap)
133             struct quirc_code *code = (struct quirc_code *)malloc(sizeof(struct quirc_code));
134             struct quirc_data *data = (struct quirc_data *)malloc(sizeof(struct quirc_data));
135
136             if (!code || !data) {
137                 Serial.println("Error al asignar memoria para el código QR o los datos");
138                 free(code);
139                 free(data);
140                 return;
141             } else {
142                 Serial.println("Memoria asignada para el código QR y los datos");
143             }
144
145             quirc_extract(qr, i, code);  // Extraer el código QR
146             Serial.println("Código QR extraído");
147
148             quirc_decode_error_t err = quirc_decode(code, data);  // Decodificar el QR
149             Serial.println("QR decodificado");
```

Figura 37. Extracción de los resultados

Lo primero que se comprueba después del análisis de la imagen es el número de códigos QR que han sido detectados en ella (*quirc_count*). Si no se ha encontrado ninguno, se puede comenzar una nueva iteración y a así tomar y analizar un nuevo fotograma. En cambio, si sí se han hallado códigos en la instantánea, se entra en un bucle *for* que los va extrayendo de uno en uno.

En principio, la librería *quirc* no pone un límite para el número de códigos diferentes que se pueden decodificar en una sola imagen. En este caso, el techo se encontraría cuando el contenido de los códigos superara la capacidad de almacenamiento de la placa.

Para cada código QR detectado, se debe reservar un espacio en la memoria para ubicar sus resultados. El puntero *code* dirige a una estructura de datos de tipo *struct quirc_code* que contiene los datos brutos del código QR: su posicionamiento, orientación y los módulos de la matriz. Mientras, el puntero *data* de la estructura de datos tipo *struct quirc_data* será el que apunte a la información ya decodificada del QR.

Nuevamente, para ahorrar espacio en la pila de datos, se pretende guardar el contenido de *code* y *data* en la memoria *heap*. Utilizando el comando *malloc* (*memory allocation*), se asigna un bloque de memoria del tamaño requerido (*sizeof(struct quirc_code)* o *sizeof(struct quirc_data)*) para un puntero de nombre *code* o *data* (el asterisco * indica que son punteros). Además, *malloc* debe saber el tipo de estructura de datos a la que dirigirá el puntero, y es eso lo que se incluye entre paréntesis justo antes de escribir el comando (*(struct quirc_code *)* o *(struct quirc_data *)*).

Habiendo especificado el espacio en el que va a ser guardada, se puede extraer la información en bruto del código QR con la función *quirc_extract*. Posteriormente, estos datos son decodificados y trasladados desde *code* a *data* con la función *quirc_decode*, que además devolverá un código de error en caso de que el proceso de decodificación falle.

```
151             if (err) {
152                 Serial.printf("Error al decodificar el código QR: %s\n", quirc_strerror(err));
153             } else {
154                 if (num_codigos == 1){
155                     Serial.printf("Código QR detectado:\n\n");
156                     Serial.printf("%s\n\n", data->payload);
157                 } else {
158                     Serial.printf("Código QR %d detectado:\n\n", i + 1);
159                     Serial.printf("%s\n\n", data->payload);
160                 }
161             }
162 
163             // Liberar la memoria dinámica usada para code y data
164             free(code);
165             free(data);
166         }
167 
168         // Esperar a que el usuario presione un carácter para continuar iterando
169         Serial.println("¿Quiere continuar con la detección de códigos QR? Presione cualquier tecla para continuar. \n");
170         while (!Serial.available()) {
171             delay(10);                      // Esperar hasta que el usuario introduzca un carácter
172         }
173         // Limpiar el buffer serie
174         while (Serial.available()) {
175             Serial.read();
176         }
177     }
178 
179     delay(2000); // Esperar antes de la siguiente iteración
180 }
181 }
182 
183 void loop() {
184     // No se necesita nada aquí ya que el trabajo se realiza en loopPersonalizado
185 }
```

Figura 38. Presentación de resultados al usuario

Tras todo ese proceso, finalmente se puede visualizar en el monitor serie el mensaje original contenido en el código QR. En caso de fallo en la decodificación, lo que se incluirá es el código de error que informa sobre la causa del mismo.

La serie completa de caracteres del mensaje se encuentra en la variable *payload* (“carga útil”) de los datos decodificados en *data*. Podrá ser leído por el monitor serie incorporado en el entorno de desarrollo PlatformIO, u otras aplicaciones que permitan acceder al puerto serie de la computadora. Además, se enumerarán los códigos decodificados en el caso de que la imagen contenga más de uno.

Tras haber cumplido su propósito, los bloques de memoria de los punteros *code* y *data* pueden ser liberados. Como se han ubicado manualmente en la memoria dinámica del controlador con la función *malloc*, deben ser liberados también manualmente con la función *free*.

Por último, en el caso de que se haya detectado al menos un código QR, se detendrá la ejecución del programa hasta que el usuario introduzca cualquier carácter en el monitor serie. Con esta acción, se permite que el usuario copie el contenido del QR a otra aplicación, acceda a los servicios que en él se guardan (URL, email…), o simplemente pueda visualizar el mensaje el tiempo que crea necesario.

Para ello, se utiliza primero la función *Serial.available* para detectar si hay algún input en el monitor serie, hasta que finalmente lo hay. Después, la función *Serial.read* lee un byte cada vez, no lo guarda en ninguna variable, y en consecuencia lo borra. Ambas forman parte de la librería básica *Arduino.h*.

Después de esto y de un retardo de 2 segundos que asegura que la información no se acumula en el monitor serie con excesiva rapidez, se volverá al comienzo del bucle *while*. Así, se comenzará una nueva iteración del proceso contenido en la función *loopPersonalizado* tomándose y analizándose una nueva imagen.

3.3.3 Resolución de imagen empleada

En este apartado se discutirá la elección final tomada para una de las variables más importantes en todo el proyecto: la resolución con la que la cámara toma las imágenes. Además, su explicación servirá para entrar más en detalle en el funcionamiento real y las limitaciones del código.

Como ya se mencionó en el apartado 2.3.3, el factor limitante para escoger la resolución es la capacidad de almacenamiento del ESP32-S3 WROOM-1. Todos los datos utilizados en el código, incluyendo las imágenes (son sin duda la variable que más ocupa), deben ser almacenadas en los 366 KB de DRAM de los que se dispone. Cada imagen en resolución CIF (400x296) y escala de grises ocupa lo siguiente:

400x296 = 118400 píxeles/imagen x 1 byte/píxel (formato grayscaling) =
118.4 KB para cada imagen

Esto es ya en torno a un tercio del total disponible. Además, es muy importante tener en cuenta que, tal y como funcionan las librerías que se están utilizando, no solo se necesita dedicar para la imagen un hueco de su tamaño, sino dos. Así, en suma se debe dedicar a almacenarlas el doble del tamaño real de la imagen, dos tercios del total de memoria disponible.

La explicación se debe a la diferencia en las variables que las librerías *esp_camera* y *quirc* utilizan para almacenar la imagen. Por un lado, *esp_camera* guarda un espacio en la memoria del tamaño de la imagen al guardar la captura con la función *esp_camera_get*. Por otro, la librería *quirc* se hace con otro hueco de memoria diferente pero del mismo tamaño al dimensionar el decodificador *qr* con la función *quirc_resize*.

La solución más obvia y que más tiempo se empleó en intentar desarrollar es hacer que la captura de la cámara se guarde directamente en las posiciones de memoria reservadas por el decodificador. Es decir, hacer que la función *esp_camera_fb_get* almacene su contenido directamente en las posiciones del puntero *imagen*, en vez de hacerlo en las de la variable *fb*.

El problema radica en que la librería *esp_camera* utiliza una estructura de datos propia (*camera_fb_t*) para almacenar el puntero de la imagen captada por la función *esp_camera_fb_get*. De hecho, *fb* no solo almacena este puntero que dirige a las posiciones de memoria en las que se guarda la imagen (*fb->buf*, en este caso), sino que también contiene otra información sobre el fotograma como su tamaño (*fb->width*, *fb->height*).

En cambio, la función *quirc_begin* necesita trabajar directamente con un puntero en formato *uint8_t*, es decir, un entero sin signo de 8 bits. Así,

esp_camera_fb_get solo puede guardar la imagen en un formato que no es el mismo que el único que el decodificador puede leer.

Para solucionar esta incompatibilidad, se intentó modificar la función *esp_camera_fb_get* para que tuviera un puntero *uint8_t* como única salida. También se probó a alterar la función *quirc_begin* para que pudiera soportar como input una estructura del tipo de *camera_fb_t*. Sin embargo, en ambos casos las funciones resultaron complejas de manipular, debido a su interrelación con otras partes de sus respectivas librerías y a su cercanía a los niveles más bajos de programación.

Por eso y teniendo en cuenta que con la resolución elegida (400x296) se consigue un éxito adecuado en el reconocimiento de los códigos, finalmente se abandonó la idea de modificar las librerías ya existentes, algo que ciertamente se alejaba del objetivo inicial propuesto para este proyecto.

Recapitulando, tomando el espacio en el que se guarda la captura y el que se utiliza para analizarla y decodificarla, se utilizan 236,8 KB de memoria para la imagen (118,4 KB x 2). Sobre los 366 KB totales de DRAM disponibles inicialmente, esto deja 129,2 KB libres, que, teniendo en cuenta los 28KB de pila de datos, se quedan en 101,2 KB destinados a todos los demás datos, como por ejemplo las variables globales *config* y *tam_pila*, el resultado de procesar los códigos QR en *code* y *data*, o las propias *fb* y *qr*.

Como comparación, se puede calcular el excesivo uso de memoria que acarrearía elegir la resolución inmediatamente superior a la CIF (400x296). Comprobando en la Tabla 3, de la página 22, se puede ver que esta resolución sería la HVGA (480x320). El tamaño de la imagen en este caso sería:

480x320 = 153600 píxeles/imagen x 1 byte/píxel (formato grayscaling)
= 153.6 KB para cada imagen

Es decir, se tendrían que dedicar un espacio de 307,2 KB (153,6 KB x2) para el almacenamiento y procesamiento de las imágenes capturadas, dejando solo 58.8 KB de DRAM para el resto de variables globales y de la pila. Este es un margen muy ajustado que, como se puede comprobar ejecutando el código en dicha resolución HVGA, resulta en un desbordamiento de la memoria que hace imposible el funcionamiento del dispositivo.

Con todo esto en cuenta, se puede entender cómo 400x296 es la resolución máxima que el dispositivo soporta para este proyecto, y por lo tanto la que se ha elegido para el funcionamiento del ESP32. Como ya se ha mencionado y se demostrará en el apartado 4, esta resolución es suficiente para realizar con éxito la función que se pretende para el dispositivo: leer los códigos QR que se presenten ante la cámara en condiciones del día a día.

3.3.4 Ejemplo de resultados

A continuación, en la Figura 39 se incluye un ejemplo del funcionamiento del programa, detectando y decodificando adecuadamente un código QR versión 3-M con el mensaje "HOLA MUNDO". En él, el código de los archivos "Lector_QR" fue subido al ESP32 desde PlatformIO, e inmediatamente después se orientó la cámara hacia la pantalla de la computadora. En el monitor, se apreciaba el código que se puede ver en la izquierda de la figura, y la respuesta obtenida en el monitor serie se puede observar a su derecha.

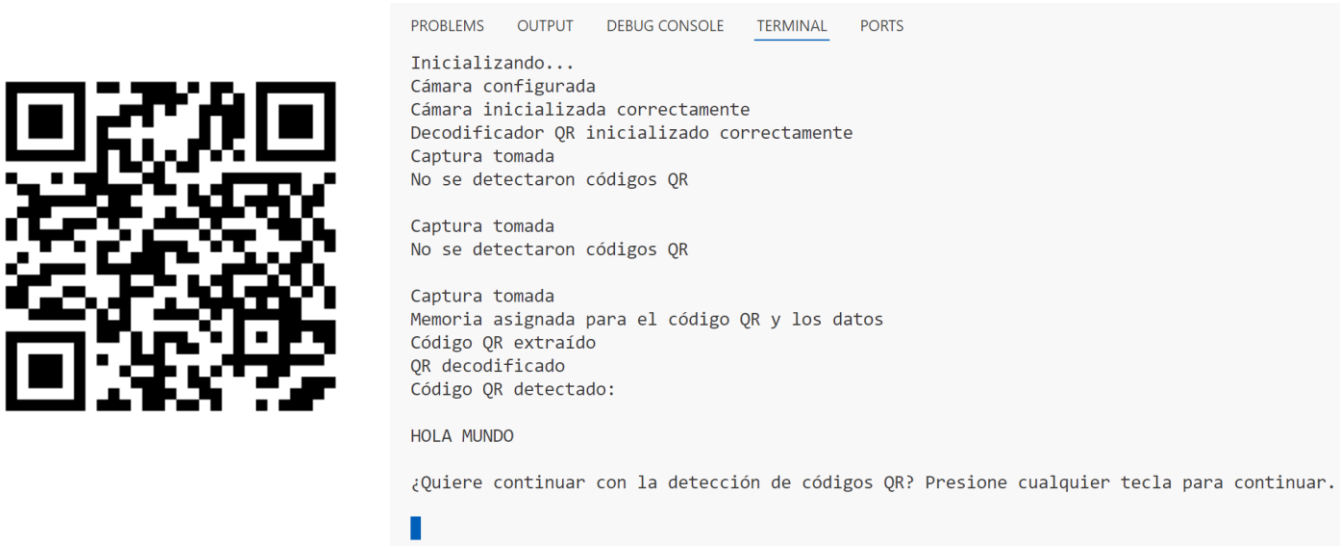


Figura 39. Ejemplo de funcionamiento del lector de códigos QR

Para más ejemplos y datos del desempeño del programa "Lector_QR", se debe acudir al apartado 4 de este proyecto.

4 RESULTADOS FINALES

Tras entender el marco teórico del proyecto, así como el desarrollo y funcionamiento del código final, en este apartado se pasará a analizar los resultados obtenidos. Así, se estudiará la respuesta del programa "Lector_QR" en diferentes situaciones, y se pondrá a prueba su desempeño ante diferentes variables.

Como ya es sabido, este programa utiliza un microcontrolador ESP32-S3 WROOM-1 con una cámara OV2640 para capturar imágenes, analizarlas en busca de códigos QR, decodificarlos y mostrar su contenido al usuario. Se ha desarrollado utilizando la librería *quirc* [4] en el entorno de desarrollo PlatformIO IDE, y es también en esta plataforma en la que se visualizarán los resultados transmitidos por la placa.

4.1 Resultados generales

En primer lugar, se puede hablar de cómo se comporta el proyecto en condiciones normales. Para ponerlo a prueba, se utilizan códigos QR generados en línea a través de la página web referenciada [8]. Existen multitud de webs con este propósito, pero la elegida fue esa porque permite al usuario elegir la versión y el nivel de corrección de errores deseado para el QR, además de soportar mensajes de diverso tipo: texto, enlaces a páginas web…

Para presentarlos ante la cámara, los códigos pueden ser impresos o directamente ser representados en el monitor del ordenador personal utilizado. Para estas primeras pruebas, se hará lo segundo por ser más sencillo y para poner a prueba una situación muy habitual en la que se puede encontrar un código QR del día a día: en una pantalla.

Una vez subido el programa a la placa, la cámara se orientará hacia la parte de la pantalla en la que se encuentra el QR. Ante esto, en las pruebas realizadas con el dispositivo se han documentado varios posibles resultados en el monitor serie. La frecuencia de éxito del programa se estudiará más adelante en el apartado de pruebas sobre el funcionamiento del dispositivo.

4.1.1 Resultados negativos

En primer lugar, se tienen varias opciones de resultados negativos. De esta forma se referirá a aquellas ocasiones en las que el dispositivo no tiene éxito en su objetivo final: transmitir al ordenador el mensaje original guardado en el código o códigos QR que se muestran ante la cámara.

En primer lugar, si el decodificador no consigue encontrar ningún código QR en las imágenes capturadas, se obtendrá la respuesta "Captura tomada. No se detectaron códigos QR" y el programa se continuará iterando hasta que un código sea encontrado. A priori, será común y nada problemático encontrarse con esta situación, ya que es fácil que la manipulación de la placa resulte en imágenes muy movidas o que directamente no muestren el código a decodificar. El problema se daría cuando, por el tamaño del código o la distancia al sensor, el decodificador no sea capaz de distinguir en la imagen ninguna matriz de píxeles con los patrones de posicionamiento propios de un QR, y se por lo tanto se quede entonces atrapado en un bucle infinito.

Por otro lado, el error también se puede dar a la hora de decodificar el código una vez localizado. Una posibilidad es que el código que se quiera detectar sea de un formato no soportado por la librería *quirc* (QR modelo 1, micro QR...), en cuyo caso se imprimirá el código de error: "*Format data ECC failure*".

En cambio, si el formato del código es correcto pero aun así el decodificador falla al traducir sus datos, se obtendrá el mensaje de error: "*ECC failure*" (*Error Correction Code failure*). Este error quiere decir que el decodificador no ha sido capaz de recomponer el mensaje con los módulos que ha reconocido. El resto de elementos de la cuadrícula no se habrán podido tener en cuenta por encontrarse ocultos en la imagen, o bien porque con el tamaño, la distancia o la distorsión del código, estos módulos se encuentran irreconocibles o se han guardado con el valor contrario al real. También puede pasar que, por estas mismas razones, los codewords de corrección de errores se contradigan entre sí, y/o no coincidan con la información de los módulos para el mensaje, no pudiéndose llegar a un consenso de lo que realmente contiene el código QR.

Fallo por no localización del QR

```
Captura tomada
No se detectaron códigos QR

Captura tomada
No se detectaron códigos QR

Captura tomada
No se detectaron códigos QR

Captura tomada
No se detectaron códigos QR

Captura tomada
No se detectaron códigos QR

Captura tomada
No se detectaron códigos QR

Captura tomada
No se detectaron códigos QR
```

Fallo por formato equivocado

```
Captura tomada
Memoria asignada para el código QR y los datos
Código QR extraído
QR decodificado
Error al decodificar el código QR: Format data ECC failure
¿Quiere continuar con la detección de códigos QR? Presione cualquier tecla para continuar.
```

Fallo en la decodificación

```
Captura tomada
Memoria asignada para el código QR y los datos
Código QR extraído
QR decodificado
Error al decodificar el código QR: ECC failure
¿Quiere continuar con la detección de códigos QR? Presione cualquier tecla para continuar.
```

Figura 40. Modalidades de fallo posibles para el lector de códigos QR

De todas estas posibilidades de error, la más común con diferencia es sin duda la última. La gran mayoría de veces que el programa falla es porque no consigue leer los suficientes módulos en la imagen para obtener la totalidad de la cadena de caracteres que contiene.

Obtener un error de formato por lo general avisa de que el código que se quiere leer no es adecuado. Sin embargo, hay ocasiones en las que puede saltar este fallo con códigos que se ha comprobado son correctos. Esto se tiene que deber a un fallo en la lectura de los módulos reservados a especificar la variante del código QR. Estos son lo primero que el decodificador busca en la cuadrícula, y un error en su reconocimiento puede hacer creer al decodificador que se está enfrentando a un código no compatible, del modelo 1 o micro QR.

Por último, nunca se suele dar la posibilidad de entrar en un bucle infinito en el que no se localiza el código en ningún fotograma. Si acaso, se pueden dar unas cuantas iteraciones en códigos QR que por su tamaño ni siquiera van a poder ser decodificados una vez encontrados y resultarán en "*ECC failure*".

4.1.2 Resultados positivos

Se consideran resultados positivos aquellos en los que se tiene éxito en la detección y decodificación del código QR. Así, el mensaje almacenado en cada símbolo encontrado será reflejado en el monitor serie. Ya se ha visto un ejemplo de esto en la Figura 39 del apartado 3.3.4 de este trabajo, pero en la siguiente figura se muestra otra demostración con la decodificación simultánea de dos códigos QR 3-M. Uno contiene de nuevo el mensaje "HOLA MUNDO", mientras que el otro almacena el link a la página web de la EII de Valladolid.

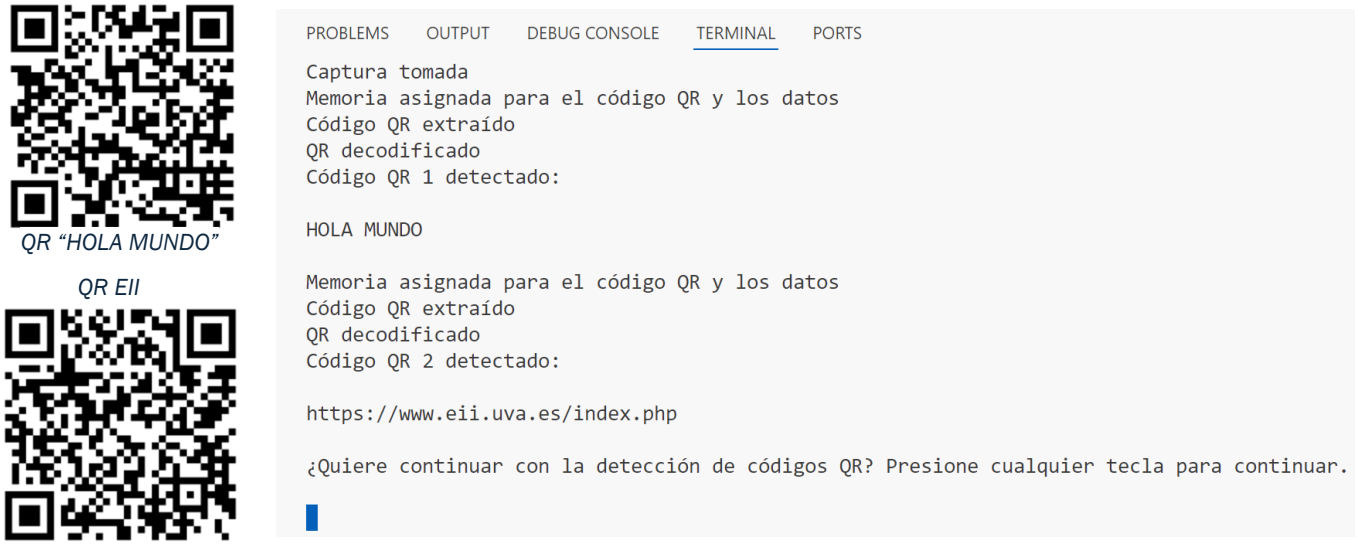


Figura 41. Respuesta del lector ante dos códigos QR simultáneos

4.2 Pruebas sobre funcionamiento del dispositivo

En este apartado, se detallarán las pruebas realizadas sobre el desempeño del ESP32-S3 WROOM-1 programado con el código "Lector_QR". Se analizará y evaluará su rendimiento y se tratarán de determinar sus límites y tasa de éxito en múltiples circunstancias.

4.2.1 Velocidad de procesamiento

Como primer test, se probará el rendimiento del código, es decir, la velocidad a la que se ejecuta el programa. Primero, se determinará qué variables afectan al tiempo de ejecución y en qué medida lo hacen. Después, se pondrá a prueba al código ante diferentes situaciones y se recogerán los resultados. Por último, se tratarán de determinar unas directrices generales para que el dispositivo trabaje de la forma más rápida posible.

Para hacer estas mediciones durante el funcionamiento del programa, se utilizará la función *millis*. Este comando permite almacenar en una variable el tiempo transcurrido desde que el código se pone en marcha, por lo que puede utilizarse para calcular el tiempo que emplea cualquier tarea. Por ejemplo, el tiempo que tarda en procesarse la imagen se mediría tal y como puede verse en las siguiente Figura 42:

```
t_inicio = millis();          // Tiempo inicial

quirc_end(qr);                // Procesamiento de la imagen

t_final = millis();           // Tiempo final
Serial.printf("Tiempo para procesar la imagen: %lu ms\n", t_final - t_inicio);
```

Figura 42. Medición de tiempo de ejecución

4.2.1.1 Duración por tipo de tarea

Primero, se determinarán cuáles son las tareas dentro del código que realmente afectan a su tiempo total de ejecución. Dentro de las que toman un tiempo sustancial en realizarse, habrá que distinguir entre aquellas que están presentes en todas las iteraciones y las que solo en caso de detectar un código QR.

Así, se realizaron múltiples ensayos. Despreciando las funciones con tiempos insignificantes, como inicializar el decodificador (0-1 ms) o copiar el puntero de imagen con *memcpy* (1-2 ms), los tiempos medidos están en la Tabla 7.

Tarea	Inicializar cámara	Sacar captura	Procesar la imagen	Extraer código	Decodificar código
Función	*esp_camera_init*	*esp_camera_get*	*quirc_end*	*quirc_extract*	*quirc_decode*
Duración (ms)	166	122	*Variable:* 2828-10445	*Variable:* 10-141	*Variable* 1-20

Tabla 7. Duración de las principales funciones del código final

Como era de esperar, la duración de tareas relacionadas con el funcionamiento de la cámara es independiente de la presencia de códigos que decodificar. En la tabla están representados los valores medios de todos los medidos.

Solo las tareas de la librería *quirc* muestran variabilidad en función del tipo de código QR mostrado ante la cámara. Los valores entre paréntesis son el mínimo y máximo medidos en los ensayos que se verán a continuación. Esta variación será estudiada en los siguientes apartados.

4.2.1.2 Variables a tener en cuenta

Ahondando en las variables que afectan al tiempo de procesamiento del código, lo primero que salta a la vista al hacer un par de ensayos es que el tamaño de la cuadrícula es muy importante. Así, la duración media aumenta significativamente para versiones grandes.

También, se puso a prueba si la longitud de la cadena de caracteres codificaba podía hacer variar el tiempo de ejecución. Se guardaron 100 y 50 caracteres en una misma versión (5-M) y se pudo constatar que esta variación no afectaba al tiempo de procesamiento del código con *quirc_end*, pero sí a las otras dos funciones.

Ciertamente es lógico, aunque la cuadrícula procesada es del mismo tamaño, la serie de codewords que contiene es menor si hay menos caracteres en el mensaje. Por lo tanto, menor es la cantidad de datos en bruto que trasladar a *code* con *quirc_extract* (de 3 a 2 ms) y más rápida será su traducción a *data* con *quirc_decode* (de 39 a 32 ms).

Por otra parte, se quiso comprobar el efecto del nivel de corrección errores elegido. Así, se probó a codificar una misma versión de QR (la 5) con el menor y el mayor nivel de seguridad ante el fallo, L y H respectivamente. Ante este ensayo, se pudo verificar una variación similar a la del caso anterior.

Nuevamente, el proceso de detección es el mismo en ambos códigos, solo que en este caso, para un nivel alto se tiene que realizar un mayor número de operaciones para sacar los codewords de corrección de errores (40 frente a 32

ms en *quirc_extract*) y con ellos decodificar el mensaje (7 frente a 4 ms en *quirc_decode*).

Sin embargo, como se pudo comprobar en la Tabla 7, la función que realmente hay que tener en cuenta para la velocidad del programa es *quirc_end*. Así, localizar el código en la imagen y procesar su cuadrícula ha demostrado ser sin duda lo más costoso computacionalmente. Por lo tanto, las variaciones documentadas para el nivel de corrección de errores y el número de caracteres no van a ser significativas sobre el tiempo de ejecución total del programa, comparadas con el tamaño del código, es decir, su versión.

4.2.1.3 Velocidad del programa para diferentes versiones

Sabiendo todo esto, se quiso poner a prueba al dispositivo ante diferentes versiones de QR, ya que esta es la variable con verdadero peso de las estudiadas. Así, se medirá el tiempo de ejecución la iteración completa del bucle y de cada tarea durante el reconocimiento de códigos QR de distintos tamaños.

Para evitar otros efectos indeseados aunque fueran leves, todos los símbolos tendrán el mismo nivel de corrección de errores (M) y almacenarán en torno a un 70% de su capacidad total de caracteres alfanuméricos. También se ha querido incluir el tiempo que tarda en procesarse una imagen que no contiene ningún código QR. Los resultados pueden visualizarse en la Tabla 8.

	SIN QR	QR 1	QR 3	QR 5	QR 8	QR 10	QR 12	QR 15
Procesar la imagen (ms)	86	2828	3559	3920	6005	6676	6980	10445
Extraer código (ms)	0	10	20	39	57	76	100	141
Decodificar código (ms)	0	1	1	3	6	12	13	20
Total iteración (ms)	208	2936	3741	4083	6237	6954	7312	11103

Tabla 8. Tiempo de ejecución del programa ante diferentes versiones QR

Se han tomado 7 muestras, llegando hasta códigos QR de versión 15-M (cuadrícula 77x77). Se han incluido como parte del experimento, pero realmente este tipo de códigos son ya demasiado grandes para casi cualquier aplicación, pudiendo contener hasta 600 caracteres alfanuméricos. A pesar de

todo, se puede ver que el programa no tarda un tiempo excesivo en analizarlos, apenas 11,1 segundos.

En cuanto a versiones de código de mayor uso a nivel multidisciplinar, las más comunes son las que están entre 3 y 8. Concretamente, para las versiones del ensayo, que son la 3-M, 5-M y 8-M, se pueden codificar hasta 61, 122 y 221 caracteres alfanuméricos respectivamente. Esto permite almacenar desde pequeños mensajes como direcciones de correo electrónico, hasta URLs y otros enlaces de gran tamaño. Vemos que el tiempo de ejecución del programa en estos casos es bastante pequeño, con unos 5 segundos de media.

Para visualizar la tendencia general, en la Figura 43 se ha representado gráficamente la duración media de una iteración para la detección de cada una de las diferentes versiones de QR:

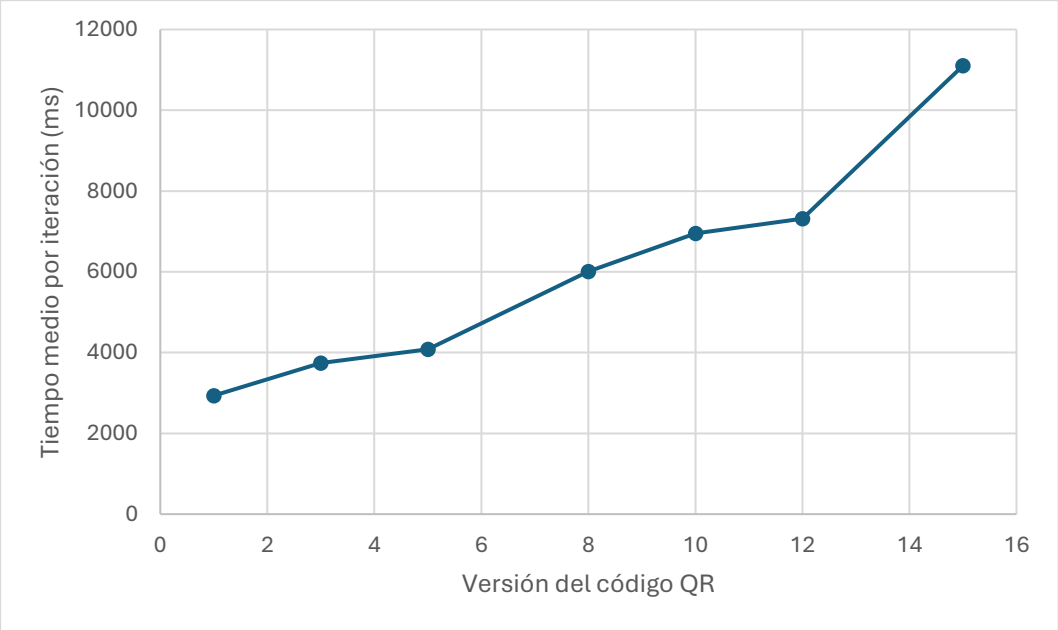


Figura 43. Tiempo de ejecución del programa ante diferentes versiones de QR

Todos los datos medidos en este apartado están disponibles en los Anexos, junto con el programa de prueba ejecutado y los códigos QR empleados para su testeo.

Como conclusión, el programa "Lector QR" presenta un rendimiento adecuado para sus posibles aplicaciones. Su velocidad es muy dependiente del tamaño de la cuadrícula empleada y por lo tanto funcionará mejor para mensajes cortos que pueden almacenarse en versiones pequeñas del símbolo. Con todo esto, para capturar, detectar y procesar los códigos más habituales a día de hoy, versiones 3-8, se tendrá un muy aceptable tiempo de espera máximo de unos 6 segundos.

4.2.2 Tamaño mínimo del código

El objetivo de este apartado es comprobar cómo la distancia y el tamaño del código QR afectan al éxito del programa. Para ello, se harán varios ensayos con el objetivo final de calcular y evaluar la relación entre el tamaño mínimo legible del código QR y la distancia entre este y la cámara.

4.2.2.1 Condiciones del ensayo

Las pruebas se llevarán a cabo de la siguiente forma: se colocará el dispositivo a diferentes distancias de un monitor, en el que habrá representado siempre el mismo código QR. Para cada distancia, se irá disminuyendo el tamaño del símbolo dentro de la pantalla hasta que este deje de ser legible. En este momento, se medirá el lado de la cuadrícula directamente en la pantalla. Concretamente, el QR utilizado será de la versión 4-Q, cuya cuadrícula 33x33 (1089 módulos) dota al código de una muy respetable capacidad para 67 caracteres alfanuméricos.

En la Figura 44 se pueden observar el aspecto externo del ensayo a 50 cm, así como la imagen captada por la cámara en ese momento. Tanto para poder capturar esa imagen como para orientar el dispositivo al monitor a cada distancia se ha utilizado el programa "CameraWebServer" explicado al comienzo del apartado 3 de este proyecto.

Figura 44. Ensayo para prueba de distancia

4.2.2.2 Resultados

Para todos estos ensayos, la causa de error del programa será un tamaño aparente muy pequeño del símbolo dentro de la imagen, haciendo que los límites entre los módulos queden difusos. En consecuencia, al procesar la imagen hay dos opciones de fallo. O el decodificador no reconoce ninguna cuadrícula (solo ve una mancha gris), o la detecta pero al procesarla no es capaz de distinguir el valor de cada uno de los bits. En esta última opción lo

que se ocasiona es un fallo en la decodificación, al no coincidir entre sí la información de los diferentes codewords de corrección de errores, ni tampoco con las de los codewords de datos.

Siguiendo este razonamiento y dado que cada versión del código QR tiene una densidad de módulos diferente, se infiere que el valor a tener en cuenta en estas pruebas será el tamaño que cada bit ocupa en la pantalla. Se podrá calcular como la longitud del lado del código entre el número de módulos que contiene. Representar los datos de esta forma permitirá aplicar los resultados a cualquier versión y no solo a la utilizada para los ensayos.

Con todo esto dicho, los resultados de las pruebas pueden apreciarse con exactitud en la Tabla 9 y gráficamente en la Figura 45.

Distancia (cm)	**10**	**30**	**50**	**80**	**100**
Tamaño del código (cm)	3,5	10	13	21	30
Tamaño del módulo (mm)	1,06	3,03	3,94	6,36	9,09

Tabla 9. Tamaño mínimo legible de código QR a ciertas distancias

En general, se puede comprobar que es necesario utilizar códigos de cierto tamaño en cuanto la distancia a la cámara aumenta. De hecho, se probó a situar el dispositivo a más de un metro de distancia, pero el símbolo no pudo ser reconocido ni siquiera ocupando la totalidad del monitor.

A pesar de eso, para la mayoría de aplicaciones, no será necesario trabajar con tales distancias. Por ejemplo, el pequeño tamaño del dispositivo y su bajo peso permitirían que fuese integrado dentro de un aparato similar a un lector de códigos de barras de los que se pueden encontrar en cualquier comercio. En este caso, se podría acercar el código a 10 centímetros de la cámara o

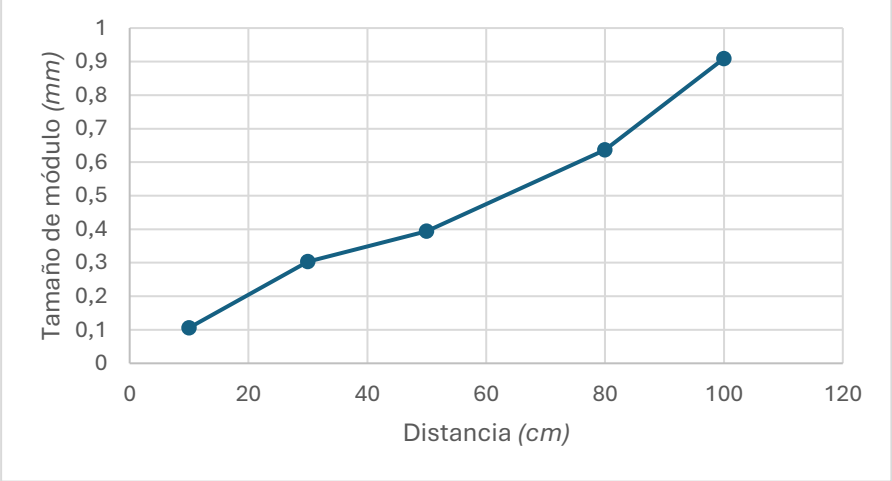


Figura 45. Tamaño del módulo de un QR legible frente a distancia

incluso menos, pudiéndose extraer hasta 67 caracteres de una cuadrícula de apenas 3,5 centímetros de lado.

Viendo la tendencia relativamente lineal del gráfico (al menos a simple vista), es interesante calcular la relación media entre el tamaño mínimo de un módulo legible y la distancia entre este y la cámara:

$$r = \frac{\frac{1{,}06}{10} + \frac{3{,}03}{30} + \frac{3{,}94}{50} + \frac{6{,}36}{80} + \frac{9{,}09}{100}}{5} = 0{,}072 \ \frac{mm\ módulo}{cm\ distancia}$$

Aunque harían falta más experimentos, esta relación *r* permite extrapolar el ensayo a otras distancias y también a otras versiones de QR. Por ejemplo, el tamaño que tendría que tener un código para que fuera posible escanearlo desde una punta a otra de una habitación, a unos 4 metros, sería:

0,072 mm/cm x 400 cm x 25 módulos/lado (QR 2) =
950,4 mm ≈ 95 cm

A esa distancia, el código tendría que tener casi un metro de lado. Sería viable para algo como un televisor, aunque muy poco práctico. Por seguir con el ejemplo, para una versión con menor densidad de módulos, por ejemplo una versión 2-Q:

0,072 mm/cm x 400 cm x 33 módulos/lado (QR 4) =
720,4 mm ≈ 70 cm

Sería algo más aceptable, pero todavía muy grande. Definitivamente, se ha visto que el lector QR diseñado en este proyecto podrá tener su aplicación en entornos en los que se pueda encontrar cerca del código que se quiere leer.

Por último, visto el experimento, es fácil darse cuenta de que los resultados son completamente dependientes de la resolución de imagen utilizada para la cámara. Sin duda, con una resolución mayor, es decir, aumentando el nivel de detalle del fotograma, se conseguiría leer códigos más pequeños a mayores distancias.

Sin embargo, la relación que se ha demostrado para este proyecto podría resultar aceptable para la mayoría de posibles aplicaciones del dispositivo, por lo que el resultado del ensayo puede considerarse como satisfactorio. Además, como se comentará en el apartado 5, aumentar la resolución con la que trabaja el programa es una línea futura de avance en el proyecto con muy fácil solución.

4.2.3 Comparación de niveles de corrección de errores

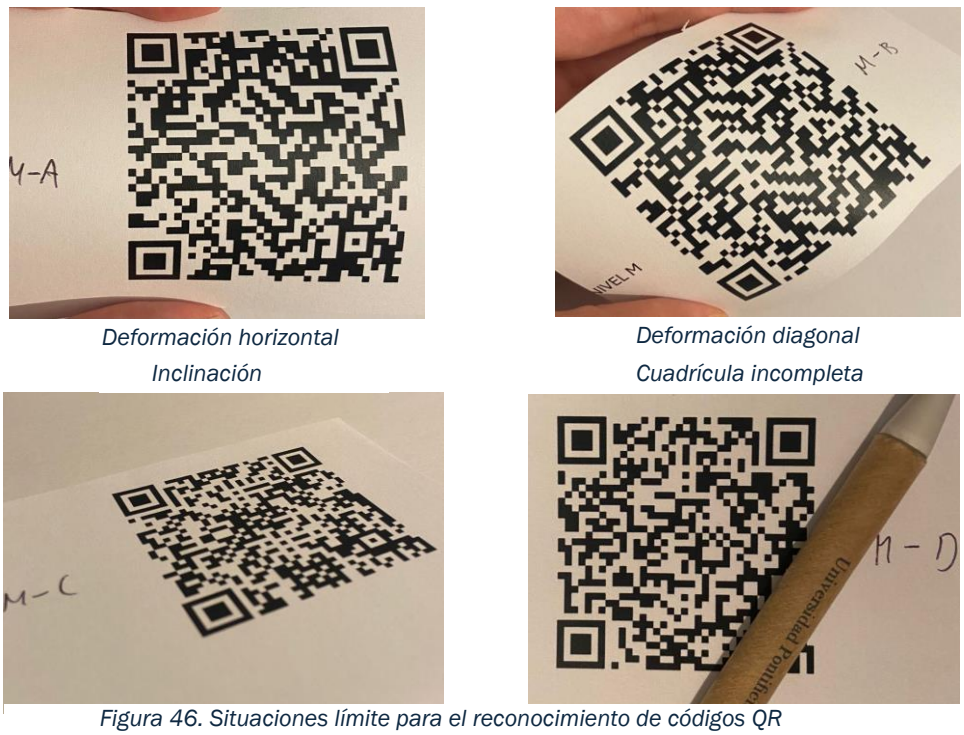
La última prueba a la que se someterá al programa pretende demostrar la eficacia de los diferentes niveles de corrección de errores disponibles para los códigos QR. Con este objetivo, se pondrá a prueba la detección y correcta decodificación de estas etiquetas en distintas situaciones límite.

En concreto, se trabajará con códigos QR 5, por ser esta una versión muy común y versátil. Para los ensayos, se imprimirá un total de 16 variantes de este código con mensajes similares, asignándose 4 para cada uno de los 4 niveles de corrección de errores. Niveles que, en orden ascendente de eficacia para subsanar fallos, son: L, M, Q y H.

4.2.3.1 Condiciones del ensayo

Todos los símbolos impresos tienen el mismo tamaño y se recortan por separado. A fin de hacer más completo el experimento, cada uno de los 4 códigos que se tienen para cada nivel se le someterá a una situación límite diferente, siendo todas circunstancias habituales en las que se pone a prueba la capacidad de los QRs para salvaguardar su mensaje.

Dos de los códigos serán físicamente deformados, uno de ellos a lo largo de su diagonal, y el otro en un eje paralelo a uno de sus lados. El tercer símbolo se colocará en un ángulo respecto al ángulo de visión de la cámara, haciendo que la perspectiva sea la que deforme su cuadrícula. El último de ellos verá parte de su matriz cubierta por un objeto (un bolígrafo), haciendo que un porcentaje de los módulos no sea visible para la cámara.



Deformación horizontal
Inclinación
Deformación diagonal
Cuadrícula incompleta

Figura 46. Situaciones límite para el reconocimiento de códigos QR

Para que las condiciones del ensayo sean lo mas equitativas posible para cada uno de los niveles, no se situará al dispositivo directamente delante de los códigos en las condiciones descritas. En cambio, imágenes como las de la Figura 46 han sido tomadas con una cámara externa y serán posteriormente mostradas al ESP32 a través de un monitor. De esta forma, se asegura que todas las imágenes tengan características similares y que no entran en juego variables como la posición relativa entre el lector y el símbolo o pequeñas diferencias entre la inclinación o la deformación sometidas a cada tipo de código. Todas las imágenes del ensayo están disponibles en los Anexos.

De esta forma, las imágenes fueron mostradas una a una a la cámara del lector QR y se fueron anotando los resultados. Fueron considerados como no legibles aquellos códigos que ni siquiera fueron reconocidos como tal en la imagen, y también los casos en los que su decodificación resultara en error ("*ECC failure*") al menos 3 veces seguidas.

Sabiendo todo esto, los resultados de la prueba se pueden ver en la Tabla 10, agrupados por nivel de corrección de errores y también por el tipo de situación en la que se fotografiaron. Lógicamente, "NO" y "SÍ" indican la no o sí detección y correcta decodificación del código en cuestión:

	L	M	Q	H	TOTAL SITUACIONES
Deformación horizontal	NO	NO	NO	SÍ	1/4
Deformación diagonal	NO	NO	NO	NO	0/4
Inclinación	NO	NO	NO	SÍ	1/4
Cuadrícula incompleta	NO	NO	SÍ	SÍ	2/4
TOTAL NIVELES	0/4	0/4	1/4	3/4	

Tabla 10. Éxito del lector ante diferente corrección de errores y situación límite

Lo primero que salta a la vista es que el éxito general de los ensayos ha sido bajo. Solo se ha conseguido detectar y decodificar la información del código para 4 de las 6 imágenes. A priori, esto se podría achacar a que las condiciones en las que se han fotografiado los QRs son demasiado desfavorables para su detección, al menos con un dispositivo y un código como los de este proyecto.

Nuevamente, algo clave para esta prueba es la resolución utilizada por la cámara. Al igual que antes, aumentar el número de píxeles en la imagen sin duda resultaría en un mayor éxito para la detección de códigos QR en situaciones como las mostradas. También, para un estudio más cuantitativo sería necesario aumentar notablemente las muestras, con varios códigos de cada tipo en la misma situación.

A pesar de eso, el ensayo sigue siendo útil para comprobar el efecto de los diferentes niveles de corrección de errores aplicables a la codificación de códigos QR. Como era de esperar, se observa claramente cómo los mejores resultados se obtienen para la categoría “L”, seguida de la “Q”. Sobre las “L” y “M” no se puede establecer ninguna diferencia , pero sin duda en condiciones más benignas se podría esperar un mejor rendimiento de los del nivel superior, el “M”.

Por otro lado, la mejor situación para el reconocimiento de los códigos ha sido esconder parte del contenido de la cuadrícula, siendo la deformación en diagonal la peor. Esto puede deberse a una dificultad para reconocer los límites de la cuadrícula cuando esta no tiene una forma exactamente cuadrada. Con esto, el lector parece que tendrá problemas para reconocer códigos deformados o mal orientados, pero será efectivo ante etiquetas manchadas o dañadas.

En general, los resultados de esta prueba pueden considerarse satisfactorios en cuanto a ejemplificar el efecto de los diferentes niveles de corrección de errores. Así, ha sido una demostración cualitativa de esta una de las características más relevantes de los códigos QR, que son el tema principal de este proyecto.

5 CONCLUSIONES

5.1 Conclusiones generales

Una vez explicado el marco teórico del proyecto, narrado su desarrollo, desglosado el contenido del programa final y comprobado su funcionamiento, se puede volver la vista atrás hacia los objetivos propuestos inicialmente.

En primer lugar, se ha logrado establecer una conexión funcional entre la cámara, el microcontrolador y el ordenador. Gracias a esto, se pudo utilizar el dispositivo para capturar y manipular imágenes, aprendiendo mientras tanto sobre parámetros clave como la resolución y el formato de los fotogramas.

A continuación, se ha hallado y documentado la librería adecuada para trabajar con códigos QR en el entorno de desarrollo propuesto. Se ha llegado a un programa final que cumple con el propósito deseado, y se han hecho comprobaciones sobre su desempeño. Con estas pruebas, se ha validado su funcionamiento y ahondado en su posible adecuación a distintas aplicaciones.

Como resultado final, se ha conseguido programar un lector de códigos QR a partir de un ESP32-S3 WROOM-1 con cámara OV2640. El dispositivo es capaz de capturar imágenes, reconocer en ellas la presencia de códigos bidimensionales de dicho tipo, leerlos corrigiendo sus errores, y enviar el mensaje decodificado a cualquier ordenador personal a través de un cable USB.

El lector es compatible con cualquier versión de código QR modelo 2, y permite trabajar con varios símbolos en una misma imagen. Es rápido en su ejecución, y apenas tarda unos segundos en realizar su cometido. A distancias cortas, es capaz de leer códigos de muy pequeño tamaño, siendo también posible utilizar cuadrículas más grandes para trabajar a cierta distancia. También, es efectivo reconociendo en condiciones adversas QRs con un nivel alto de corrección de errores, pudiéndose por lo tanto trabajar con códigos dañados o deformados.

Por último, en el proceso se ha aprendido a trabajar en el mundo de Arduino y se ha entendido el funcionamiento de los tan habituales códigos QR. Con esto, se han adquirido conocimientos, herramientas y recursos que si duda resultarán de gran utilidad en el futuro, tanto para quien ha desarrollado el proyecto, como para cualquier persona que acceda a él tras su publicación.

5.2 Líneas futuras

Mirando al futuro, la importancia de los microcontroladores y los códigos QR está destinada a crecer. Los avances en la miniaturización de componentes y la mejora de las capacidades de procesamiento hacen que los microcontroladores sean cada vez más potentes, mientras que la creciente digitalización demanda soluciones rápidas y efectivas para el manejo de datos.

Los códigos QR continúan siendo una herramienta flexible para transmitir información de manera rápida y segura, con la posibilidad de evolucionar hacia formas más avanzadas de codificación que integren nuevas tecnologías de corrección de errores y seguridad. Por lo tanto, es interesante discutir algunas de las posibles correcciones y ampliaciones futuras que se podrían hacer para avanzar en la dirección propuesta por este proyecto.

En primer lugar y como ya se ha mencionado varias veces, un avance que mejoraría sustancialmente las prestaciones del producto sería mejorar la resolución con la que trabaja la cámara. Capturar imágenes con un mayor nivel de detalle facilitaría mucho el correcto reconocimiento de códigos en condiciones adversas.

Además, habría varias formas de conseguir una mejor resolución, pudiéndose incluso llegar al máximo soportado de 1600x1200 píxeles. La más fácil sería ampliar la memoria del ESP32, conectando a él un módulo que lo dote de cierta capacidad de datos del tipo PSRAM. El microcontrolador está preparado para ello, y a cambio del par de euros que costaría se podría tener más espacio en el que alojar imágenes de mayor tamaño. Otra opción sería la discutida en el apartado sobre la resolución de imagen elegida finalmente, trabajar con otra librería o manipular las descritas para almacenar y decodificar las instantáneas de forma más eficiente.

Por otra parte, algo que indudablemente ayudaría a la relación entre el usuario y el lector de códigos QR sería implementar una interfaz más amigable para trabajar con él. Así, se podría crear una forma de manipular los parámetros del Arduino y recibir sus resultados más práctica, estética y que esconda las cuestiones más técnicas de trabajar en un entorno de desarrollo como PlatformIO IDE.

Aparte de eso, se podrá trabajar en diferentes direcciones según la aplicación concreta que se quiera dar a este dispositivo. Ahondando en a las opciones planteadas al inicio de este proyecto, si se desea incorporar este lector en un entorno industrial, se podría trabajar en un programa que almacenara y utilizara los datos recibidos del dispositivo para llevar un control de inventario u otras funciones. En ese sentido, también se podría diseñar algún tipo de cubierta que facilitara la manipulación del lector.

Por otro lado, si se le buscara dar una aplicación en un sistema de recuento electoral digital, se debería seguir trabajando en los sucesivos eslabones del sistema. Por ejemplo, se podría buscar una forma de encriptar los datos y transmitirlos con seguridad desde la computadora a la que el lector ha transmitido la información hasta un ordenador que lleve el recuento completo.

En general, este es un ámbito con infinitas posibilidades de desarrollo, así como infinitas son sus posibles aplicaciones. Con este trabajo, apenas se ha arañado el posible potencial de tecnologías y dispositivos como los utilizados. Aun con eso, este proyecto puede ser la base para sistemas mucho más complejos y así servir como punto de partida para multitud de propuestas innovadoras.

6 REFERENCIAS

Referencias principales, por orden de mención:

[1] **M. Banzi y M. Shiloh**, Make: Getting Started with Arduino, 3ª Edición ed., 2015.

[2] **Espressif Systems**, «ESP32-S3-WROOM-1 & ESP32-S3-WROOM-1U Datasheet,» 2023. [En línea]. Available: https://www.espressif.com.cn/sites/default/files/documentation/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf. [Último acceso: Agosto 2024].

[3] **Omnivision**, «OV2640 Camera Module Datasheet,» 28 febrero 2006. [En línea]. Available: https://www.uctronics.com/download/cam_module/OV2640DS.pdf. [Último acceso: Agosto 2024].

[4] **D. Beer**, «Librería "quirc",» Julio 2024. [En línea]. Available: https://github.com/dlbeer/quirc. [Último acceso: Agosto 2024].

[5] **ISO / IEC**, «ISO / IEC 18004:2015 Information technology — Automatic identification and data capture techniques — QR Code bar code symbology specification,» 1 febrero 2015. [En línea]. Available: https://gcore.jsdelivr.net/gh/tonycrane/tonycrane.github.io/p/409d352d/ISO_IEC18004-2015.pdf. [Último acceso: Agosto 2024].

[6] **Nanjing Qinheng Microelectronics Co.**, «CH343 Driver download,» [En línea]. Available: https://www.wch-ic.com/search?t=all&q=ch343. [Último acceso: Agosto 2024].

[7] **Espressif Systems**, «Documento JSON para gestión de tarjetas ESP32-S3 en Arduino IDE,» [En línea]. Available: https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json. [Último acceso: Agosto 2024].

[8] **Calm9**, «QR Code Generator,» 6 noviembre 2020. [En línea]. Available: https://qr.calm9.com/en/. [Último acceso: Agosto 2024].

Referencias de Figuras:

[9] **M. Banzi, D. Cuartielles, T. Igoe y D. Mellis**, «What is Arduino?,» 5 febrero 2018. [En línea]. Available: https://www.arduino.cc/en/Guide/Introduction. [Último acceso: Agosto 2024].

[10] **Denso Wave Inc.**, «About QR Code,» 29 enero 2013. [En línea]. Available: https://web.archive.org/web/20130129065726/http://www.qrcode.com/en/aboutqr.html. [Último acceso: Agosto 2024].

[11] **Pr1mer Tech**, «Image Size and Resolution,» 2024. [En línea]. Available: https://guidelines.pr1mer.tech/05-IconsImages/Image-Size-and-Resolution. [Último acceso: Agosto 2024].

[12] **Freenove**, «Manual para Freenove Ultimate Starter Kit for ESP32,» [En línea]. Available: https://freenove.com/fnk0047. [Último acceso: Agosto 2024].

[13] **PlatformIO**, «PlatformIO IDE,» 2024. [En línea]. Available: https://platformio.org/platformio-ide. [Último acceso: Agosto 2024].

[14] **Denso Wave Inc.**, «Types of QR Code,» [En línea]. Available: https://www.qrcode.com/en/codes/. [Último acceso: Agosto 2024].

[15] **Dexon Systems**, «What are RGB and YUV color spaces?,» 5 abril 2022. [En línea]. Available: https://dexonsystems.com/blog/rgb-yuv-color-spaces. [Último acceso: Agosto 2024].

[16] **F. Li**, «Why use YUV color space instead of RGB color space,» 16 diciembre 2023. [En línea]. Available: https://bachtech.medium.com/why-use-yuv-color-space-instead-of-rgb-color-space-df741a6a6888. [Último acceso: Agosto 2024].

ANEXOS

Dentro del ZIP “02577_Anexos”, se tienen las siguientes carpetas:

“Lector_QR”

Otros códigos desarrollados en el proyecto:

- Retransmisión de la cámara: “CameraWebServer”
- Manipulación de imágenes: “CuentapixelesYUV”

Pruebas sobre el programa final (códigos, QRs, y resultados):

- Velocidad
- Distancia
- Niveles de corrección de errores