ESCUELA POLITÉCNICA NACIONAL

ESCUELA DE FORMACIÓN DE TECNÓLOGOS

IMPLEMENTACIÓN DE UN PROTOTIPO DE MONITOREO DE ESPACIOS EN UN PARQUEADERO BASADO EN ESP32 Y PÁGINA WEB

TRABAJO DE INTEGRACIÓN CURRICULAR PRESENTADO COMO REQUISITO PARA LA OBTENCIÓN DEL TÍTULO DE TECNÓLOGO SUPERIOR EN REDES Y TELECOMUNICACIONES

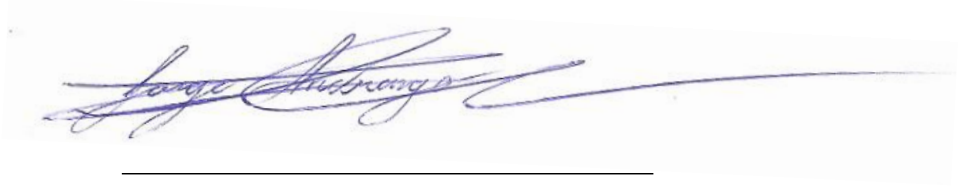
JORGE LUIS ANDRANGO QUILUMBA

DIRECTOR: ANDRES FERNANDO REYES CASTRO

DMQ, agosto 2024

CERTIFICACIONES

Yo, Jorge Luis Andrango Quilumba declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.



JORGE ANDRANGO

jorge.andrango@epn.edu.ec

aj_z-alferes95@hotmail.com

Certifico que el presente trabajo de integración curricular fue desarrollado por Jorge Luis Andrango Quilumba, bajo mi supervisión.

Andrés Fernando Reyes Castro
DIRECTOR

andres.reyes@epn.edu.ec

DECLARACIÓN DE AUTORÍA

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el (los) producto(s) resultante(s) del mismo, son públicos y estarán a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

Jorge Luis Andrango Quilumba

DEDICATORIA

Este proyecto de titulación está dedicado a todas las personas que me han ayudado a superar obstáculos y adversidades. En especial, quiero agradecer a mi familia. A mi madre y padre, por su amor incondicional, apoyo constante y por guiarme siempre por el camino correcto. A Sofí y mis hermanas, por su compañía, comprensión y por estar siempre a mi lado.

También quiero expresar mi profunda gratitud a mis abuelos, quienes formaron y criaron a mis padres con valores y principios sólidos, convirtiéndolos en las personas íntegras que son hoy. Gracias por ser el pilar sobre el cual se ha cimentado nuestra familia y por transmitirnos su sabiduría y fortaleza.

Finalmente, quiero agradecer a todas aquellas personas que han influido positivamente en mi vida, empujándome a realizar cosas inimaginables y a ser optimista sobre mi futuro, a pesar de mis inseguridades. Su confianza en mí y sus palabras de aliento han sido fundamentales para alcanzar esta meta.

A todos ustedes, gracias por ser mi pilar y fuente de inspiración.

Jorge

ÍNDICE DE CONTENIDOS

RESUMEN

Este proyecto de titulación se enfoca en el desarrollo e implementación de un prototipo de monitoreo de espacios de estacionamiento basado en el microcontrolador ESP32 y una página web. Los objetivos principales incluyen identificar los requerimientos del sistema, definir los componentes de hardware y software necesarios, diseñar el prototipo, implementar una maqueta funcional y realizar pruebas de funcionamiento.

Para la detección de los espacios disponibles, se utilizaron sensores ultrasónicos que miden la distancia entre el sensor y un objeto. Si la distancia es mayor a 10 (cm), el espacio se considera disponible; de lo contrario, está ocupado. Estos datos se transmiten a la base de datos en tiempo real de Firebase, donde se almacenan y se actualizan continuamente en la página web.

El diseño del sistema incluye una página web que muestra el estado de cada espacio de estacionamiento de manera textual y visual. El color verde indica disponibilidad y el color gris señala que el espacio está ocupado. La actualización continua de los datos proporciona una experiencia fluida al usuario.

En el documento, se presenta en la primera sección una descripción de los componentes empleados en el proyecto, junto con los objetivos generales y específicos. El marco teórico menciona temas relacionados con tecnologías como el ESP32 y los sensores ultrasónicos. En la sección de metodología, se describen las etapas clave del desarrollo del prototipo, desde la identificación de los requisitos hasta su implementación y pruebas. Luego, se presentan los resultados, los cuales incluyen la selección de componentes, el diseño del circuito, y la integración con la plataforma Firebase. Para finalizar, el documento concluye con un análisis de la efectividad del prototipo y se ofrecen recomendaciones para mejoras en el sistema.

PALABRAS CLAVE: ESP32, monitoreo de estacionamiento, sensores ultrasónicos, Firebase, base de datos en tiempo real, página web.

ABSTRACT

This thesis project focuses on the development and implementation of a parking space monitoring prototype based on the ESP32 microcontroller and a web page. The primary objectives include identifying the system requirements, defining the necessary hardware and software components, designing the prototype, implementing a functional model, and conducting performance tests.

Ultrasonic sensors were used to detect available spaces by measuring the distance between the sensor and an object. If the distance is greater than 10 cm, the space is considered available; otherwise, it is occupied. These data are transmitted to the Firebase database in real-time, where they are stored and continuously updated on the web page.

The system design includes a web page that displays the status of each parking space both textually and visually. Green indicates availability, while gray signals that the space is occupied. Continuous data updates provide a seamless user experience.

The document begins with a description of the components used in the project, along with the general and specific objectives. The theoretical framework covers topics related to technologies such as the ESP32 and ultrasonic sensors. The methodology section outlines the key stages in the development of the prototype, from requirement identification to implementation and testing. The results section presents the component selection, circuit design, and integration with the Firebase platform. Finally, the document concludes with an analysis of the prototype's effectiveness and offers recommendations for system improvements.

KEYWORDS: *ESP32, parking space monitoring, ultrasonic sensors, Firebase, real-time database, web interface.*

.

1 DESCRIPCIÓN DEL COMPONENTE DESARROLLADO

El componente desarrollado es un prototipo para el monitoreo de plazas en un parqueadero, mediante el uso de la placa de desarrollo ESP32. Los sensores de distancia conectados a la placa miden la proximidad a objetos para determinar si una plaza de parqueo está ocupada o libre.

El prototipo y su maqueta realizan el monitoreo de cuatro plazas; sin embargo, el código desarrollado para el prototipo permite manejar un multiplexor de 16 canales (CD74HC4067), incrementando la capacidad de medición de plazas del parqueadero. El sensor ultrasónico HC-SR04 emite pulsos que rebotan en los objetos presentes. El tiempo que tarda el pulso en regresar al sensor se usa para calcular la distancia mediante una fórmula sencilla [1].

Los valores de distancia obtenidos de los sensores HC-SR04 se utilizan para establecer la lógica de plaza disponible u ocupada, basada en una distancia mínima de 10 centímetros. Estos estados se envían en tiempo real a una base de datos de Firebase, eliminando la necesidad de almacenamiento en una base de datos basada en MySQL.

El diseño de la página web se centra en obtener los estados del estacionamiento desde la base de datos en tiempo real. Para ello, se configuran los parámetros básicos necesarios para inicializar la conexión a Firebase: ApiKey, AuthDomain, DatabaseURL, ProjectId, StorageBucket, MessagingSenderId y AppId.

Estos parámetros están configurados tanto en el código de la página web como en el código de la ESP32. Esto permite mostrar en la página web, alojada en un servicio de hosting, el estado y número de plazas disponibles, además de generar una alerta en caso de que no haya plazas disponibles en el parqueadero.

Finalmente, se llevaron a cabo pruebas para validar el funcionamiento del prototipo. Estas pruebas incluyeron la verificación de la lectura satisfactoria de los sensores, la estabilidad de la conexión con Firebase y la correcta visualización de los datos en la página web. Los resultados de estas pruebas demostraron que el sistema es capaz de monitorear eficazmente las plazas del parqueadero y proporcionar información en tiempo real de manera fiable.

1.1 Objetivo general

Implementar un prototipo de monitoreo de espacios en un parqueadero basado en ESP32 y página web.

1.2 Objetivos específicos

- Identificar los requerimientos del prototipo.
- Definir los componentes de hardware y software.
- Diseñar el prototipo de monitoreo.
- Implementar el prototipo mediante una maqueta.
- Realizar pruebas de funcionamiento del prototipo.

1.3 Alcance

Por medio del presente proyecto se busca implementar un prototipo de un sistema que permita realizar monitoreo de plazas en un parqueadero mediante el uso de la placa de desarrollo ESP32 y sensores que permitan determinar la cantidad de espacios disponibles en el parqueadero.

Una vez procesados los datos, estos serán enviados por medio del módulo WiFi de la placa hacia el servidor web, en donde se mostrará número de espacios disponibles en el parqueadero y su ubicación. Además, se mostrará en la página web una alerta en caso de que no existan espacios disponibles.

1.4 Marco Teórico

Internet de las cosas

El Internet de las Cosas (IoT, por sus siglas en inglés) representa un avance significativo hacia la conexión de un amplio rango de elementos físicos cotidianos a Internet. En otras palabras, se trata de llevar el mundo físico al mundo virtual, permitiendo la interconexión de objetos comunes como bombillas de luz, electrodomésticos y dispositivos médicos. El objetivo final de esta tecnología es consolidar el concepto de ciudades inteligentes, donde la infraestructura urbana se gestiona de manera más eficiente y sostenible.

Los dispositivos IoT se dividen comúnmente en dos categorías principales. La primera categoría comprende los actuadores, que son dispositivos diseñados para llevar a cabo acciones específicas. La segunda categoría incluye los sensores, cuyo propósito es recopilar y gestionar datos para su posterior envío y análisis. Estos sensores son

fundamentales para el funcionamiento del IoT, ya que proporcionan la información necesaria para la toma de decisiones automatizadas y la optimización de recursos [2].

Placa de desarrollo ESP32

La placa de desarrollo ESP32, también conocida como NodeMCU ESP32, es un dispositivo creado por Espressif Systems. Este dispositivo se basa en una serie de SoC (*System on a Chip*) y módulos que ofrecen un bajo costo y un consumo de energía reducido [3].

El ESP32 es un microcontrolador que integra conectividad WiFi, así como Bluetooth y *Bluetooth Low Energy* (BLE). El chip ESP32 está basado en un microprocesador de doble núcleo, con una frecuencia de funcionamiento de hasta 240 MHz. Para programar el dispositivo, es posible utilizar diversos entornos de desarrollo integrado (IDEs) como Arduino, CodeBlocks, Netbeans, Lua, entre otros.

Los puertos GPIO (*General Purpose Input/Output*) de la placa son capaces de entregar hasta 12 mA, permitiendo su uso tanto como entradas como salidas digitales. Además, la placa cuenta con 10 sensores táctiles capacitivos, que reaccionan al contacto y envían información al microcontrolador. También dispone de un sensor de temperatura incorporado, que permite monitorear la temperatura de funcionamiento de la placa [4].

TaskScheduler

La librería TaskScheduler es una herramienta utilizada para gestionar tareas concurrentes en microcontroladores como el ESP32. Esta librería permite crear, planificar y ejecutar múltiples tareas en paralelo de manera eficiente, lo cual es esencial en aplicaciones donde varios procesos deben ocurrir simultáneamente o en intervalos específicos. La TaskScheduler facilita la organización del código al separar las distintas funcionalidades en tareas independientes, mejorando así la legibilidad y el mantenimiento del código. Además, proporciona métodos para controlar la ejecución de las tareas, como iniciar, detener, pausar y reanudar, permitiendo una gestión flexible del flujo de trabajo [5].

TaskScheduler aprovecha al máximo las capacidades del ESP32 y el doble núcleo de este microcontrolador. Al dividir el trabajo en tareas más pequeñas y específicas, es posible ejecutar operaciones en paralelo, como leer sensores, enviar datos a un servidor, o actualizar una interfaz de usuario sin que una tarea bloquee a las demás. Esto es particularmente útil en proyectos de IoT donde se requiere la supervisión y control de múltiples dispositivos o sensores de manera simultánea.

Sensor Ultrasónico

Un sensor ultrasónico es un dispositivo de medición de distancia ampliamente utilizado en robótica. Funciona mediante dos transductores piezoeléctricos de ultrasonido: un emisor y un receptor. Su principio de su funcionamiento se basa en emitir pulsos de ultrasonido por encima de los 20(kHz), donde las ondas sonoras viajan a través del aire y rebotan al encontrar un objeto en un rango corto, como se india en la Figura 1.1. El receptor piezoeléctrico detecta las ondas reflejadas y envía una señal de (5V) durante un tiempo proporcional al intervalo que tardaron las ondas en viajar hasta el objeto y regresar [7].

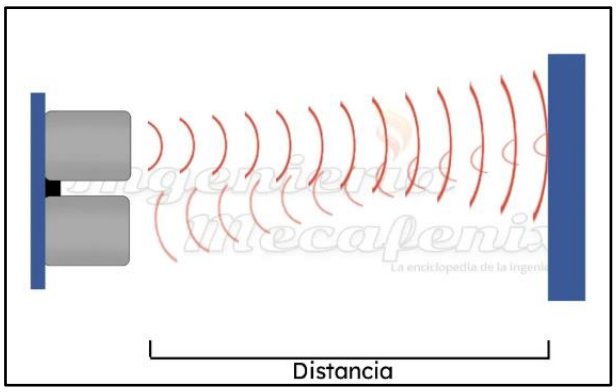


Figura 1.1 Sensor ultrasónico [6]

Multiplexor

El Multiplexor o Demultiplexor Digital/Análogo es un módulo que facilita el control de un mayor número de elementos utilizando menos entradas o salidas. Este dispositivo permite ampliar el número de entradas y salidas en placas de desarrollo como Arduino, ESP32 o PIC.

Un MUX (multiplexor) dirige múltiples señales hacia una sola entrada, mientras que su contraparte, el DEMUX (demultiplexor), distribuye una señal única a varias salidas. Este módulo permite gestionar hasta múltiples dispositivos utilizando solo un menor número pines. Estos pines de control permiten seleccionar el canal por el cual se desea escribir o leer, mientras que el pin de señal recibe o envía la señal del canal seleccionado [8].

Base de datos en tiempo real

Una base de datos en tiempo real es un sistema diseñado para capturar, procesar y enriquecer datos en el mismo momento en que se generan, lo que permite que estos datos estén disponibles y actualizados de inmediato. Este tipo de base de datos se utiliza para manejar flujos de datos continuos, a diferencia de los sistemas tradicionales que procesan datos en intervalos regulares.

Los tipos de bases de datos que pueden manejar datos en tiempo real son los siguientes:

Cuadrículas de datos en memoria: Almacenan datos en la memoria para acceder a ellos rápidamente.

Bases de datos en memoria: Similar a las cuadrículas de datos en memoria, pero diseñadas para aplicaciones específicas que requieren acceso rápido a los datos.

Bases de datos NewSQL: Combinan las características de las bases de datos SQL tradicionales con la capacidad de manejar grandes volúmenes de datos y alta velocidad de procesamiento.

Bases de datos NoSQL: Flexibles y escalables, adecuadas para manejar datos no estructurados y grandes volúmenes de información [9].

Hosting

El alojamiento web es un servicio que permite almacenar y hacer accesible un sitio web o aplicación web desde diversos dispositivos como computadoras, teléfonos inteligentes y tabletas. Un sitio o aplicación web generalmente está compuesto por múltiples archivos, incluyendo imágenes, videos, textos y código, que necesitan ser guardados en computadoras especiales llamadas servidores. Los proveedores de servicios de alojamiento web se encargan de mantener, configurar y operar estos servidores físicos, ofreciendo la infraestructura necesaria para alquilar y alojar los archivos de un sitio web. Además, estos servicios de alojamiento web ofrecen soporte adicional, como medidas de seguridad, copias de seguridad y optimización del rendimiento del sitio, lo que permite a los usuarios centrarse en las funciones principales de su página web [10].

Lenguaje HTML

El *Hypertext Markup Language* (HTML) es el lenguaje de marcado estándar para crear y estructurar documentos que se visualizan en un navegador web, es decir, para construir páginas web. Un hipertexto permite hacer referencia a otros fragmentos de texto, mientras que el lenguaje de marcado utiliza una serie de etiquetas para indicar a los navegadores web cómo deben mostrar y estructurar el contenido. Los documentos HTML pueden complementarse con tecnologías como las hojas de estilo en cascada y lenguajes de programación como JavaScript para definir su estilo y comportamiento.

Aunque HTML no es un lenguaje de programación, ya que no permite crear funciones dinámicas, es fundamental para construir y organizar secciones, párrafos y enlaces en una página web mediante el uso de elementos, etiquetas y atributos. Los navegadores

web reciben el texto HTML desde un servidor o desde almacenamiento local y lo transforman en una página web multimedia.

Entre los usos más comunes del HTML se encuentran:

Desarrollo web: Los desarrolladores utilizan HTML para diseñar cómo los navegadores muestran los elementos de una página web, como texto, hipervínculos y archivos multimedia.

Documentación web: HTML permite organizar y dar formato a documentos de manera similar a como lo hace Microsoft Word, facilitando su presentación y accesibilidad en la web. [11]

2 METODOLOGÍA

En primer lugar, se llevó a cabo un análisis de los requisitos tanto de hardware como de software necesarios para la implementación del prototipo. La selección del hardware se basó en la disponibilidad en el mercado, el funcionamiento, el número de plazas del parqueadero, y la comunicación de los sensores con la placa de desarrollo ESP32.

El diseño del prototipo incluyó la creación del diagrama de flujo del código de programación y su correspondiente implementación en Arduino IDE. El prototipo también se desarrolló en una placa de pruebas, en la cual se conectaron sensores ultrasónicos a un multiplexor de 16 canales (CD74HC4067) para aumentar la capacidad de monitoreo de la ESP32. Posteriormente, la placa ESP32 se programó para utilizar los sensores ultrasónicos y medir la distancia a los objetos, determinando así el estado de las plazas disponibles.

El desarrollo del código para la ESP32 se apoyó en la librería TaskScheduler, que permitió gestionar tareas concurrentes, como la lectura de los sensores y el envío de datos a la base de datos. Los datos de los sensores se enviaron en tiempo real a una base de datos en Firebase. La configuración de Firebase incluyó la inicialización de los parámetros ApiKey, AuthDomain, DatabaseURL, ProjectId, StorageBucket, MessagingSenderId y AppId en el código de la ESP32 y en la página web.

Para mostrar el estado de las plazas de estacionamiento en tiempo real, se desarrolló una página web utilizando el servicio de hosting de Firebase, configurada para obtener datos de la base de datos y generar alertas en caso de que todas las plazas estuvieran ocupadas.

Las pruebas realizadas demostraron que el prototipo podía monitorear eficazmente las plazas del parqueadero y proporcionar información en tiempo real sobre la disponibilidad de espacios. La página web mostró los datos correctamente y generó las alertas adecuadas cuando no había plazas disponibles, cumpliendo con el alcance planteado y demostrando la viabilidad de la solución para el monitoreo de plazas de estacionamiento.

3 RESULTADOS

Esta sección describe detalladamente el funcionamiento del prototipo, así como la selección de hardware y software utilizados para cumplir con los requerimientos establecidos. Además, se explica la elección específica de cada componente y la manera en que se llevó a cabo su implementación.

El prototipo monitorea las plazas disponibles en un estacionamiento y emite una alerta cuando no hay espacios libres. Para ello, mide la distancia entre un objeto y el sensor ultrasónico HC-SR04. Los datos obtenidos por la placa ESP32 son procesados y enviados a través de Wi-Fi a una base de datos en tiempo real, permitiendo su visualización en una página web.

3.1 Identificación de los requerimientos del prototipo

El prototipo de monitoreo tiene como objetivo evaluar la disponibilidad de plazas en un estacionamiento y mostrar esta información en una página web. Para su desarrollo, se consideran los siguientes requisitos:

Conexión Wi-Fi: El prototipo debe contar con conectividad Wi-Fi para acceder a Internet de forma inalámbrica, lo que permite la comunicación con la página web y la visualización de los datos de los sensores ultrasónicos.

Medición de Distancias: El prototipo debe ser capaz de obtener los valores de distancia de al menos cuatro sensores ultrasónicos para determinar el grado de ocupación de los estacionamientos.

Determinación de plazas: La cantidad de espacios disponibles deben ser mostrados correctamente mediante texto o en forma gráfica.

Actualización en Tiempo Real: Los datos recopilados por la placa de desarrollo deben actualizarse en tiempo real o enviarse a una base de datos en tiempo real.

Servicio de Hosting: Se requiere un servicio de hosting para que los usuarios puedan acceder a una página web y verificar la disponibilidad de espacios en el estacionamiento.

Alertas: Es necesario que en caso de no existir espacios disponibles se emita una alerta que informe la no disponibilidad de espacios en el parqueadero.

3.2 Selección de componentes de hardware y software

Selección de hardware

El prototipo necesita un microcontrolador con conexión Wi-Fi, por lo que se eligió la placa de desarrollo ESP32 en lugar de la ESP8266. La ESP32 ofrece una mejora significativa en el procesamiento de información gracias a sus dos núcleos de 32 bits y velocidades de reloj más altas, lo que permite la ejecución de tareas en paralelo, en contraste con la ESP8266 que solo cuenta con un núcleo de 32 bits [12].

El ESP32 cuenta con 520 (kBytes) de memoria, considerablemente más que los 80 (kBytes) del ESP8266. Esta diferencia justifica la elección del ESP32 a pesar de su mayor costo, ya que ofrece ventajas significativas en términos de capacidad y rendimiento, como se visualiza en la Tabla 3.1. Además, la tecnología 802.11n del ESP32 proporciona una velocidad de hasta 150 (Mbps), superando los 72,2 (Mbps) que ofrece la tecnología Wi-Fi del ESP8266.

Tabla 3.1. Comparación entre el ESP32 y ESP8266 [13]

Característica	**NODEMCU ESP32**	**NODEMCU ESP8266**
CPU	Xtensa Dual-Core 32-bit LX6 con 600 DMIPS	Xtensa Single-core 32-bit L106
Velocidad del WiFi	802.11n hasta 150 (Mbps)	Hasta 72,2 (Mbps)
Protocolo WiFi	802,11 b/g/n | 2,4 (Ghz)	802,11 b/g/n | 2,4 (Ghz)
GPIO	34	17
Bluetooth	SÍ	NO
Canales ADC	8 canales	Un solo canal
SPI/I2C/I2S/UART	4/2/2/3	2/1/2/2
Sensor táctil	SÍ (8-Canales)	NO
Sensor de efecto Hall	SÍ	NO

Tabla 3.1.

Sensor de temperatura	SÍ	NO
Costo	$12	$8
SRAM	520 (kB) | 8 (kB) de SRAM en RTC	Tamaño de la RAM < 50 (kB)
ROM	448 (kB) de ROM para el arranque y las funciones básicas	No hay ROM programable
Rango de temperatura de funcionamiento	-40°C ~ +85°C	-40°C ~ 125°C
Corriente operativa	Promedio: 80 (mA)	Valor medio: 80 (mA)

La selección de sensores se realizó mediante una comparación entre los módulos HC-SR04 y SRF08, como se muestra en la Tabla 3.2. Sin embargo, la elección también se basó en la cantidad de sensores necesarios, su disponibilidad en el mercado y el costo asociado.

El sensor HC-SR04 requiere dos pines para su comunicación, lo cual implica un mayor uso de pines en la placa ESP32 al hacer uso de varios sensores. En contraste, el módulo SRF08 utiliza una interfaz de comunicación I2C, que permite la conexión de múltiples módulos a través de un bus de datos. A pesar de estas ventajas, el costo elevado e poca oferta en el mercado del SRF08 justifican la elección del HC-SR04.

Aunque la ESP32 tiene un número limitado de 34 pines, es posible optimizar su uso mediante un multiplexor, reduciendo la cantidad de pines necesarios a 5. Esto permite un uso más eficiente del número de pines con el sensor HC-SR04, reforzando así la decisión de utilizar este módulo.

Tabla 3.2. Comparación entre módulo HC-SR04 y SRF08 modulo [14] [15]

Característica	**HC-SR04**	**SRF08**
Voltaje de operación	DC 5 (V)	DC 5 (V)
Consumo de corriente	15 (mA)	30 (mA)
Ángulo de detección	Aproximadamente 15 grados	Aproximadamente 45 grados

Tabla 3.2.

Interfaz de comunicación	Pines *Trigger y Echo*	I_2C
Rango de medición	2 (cm) hasta 400 (cm)	3 (cm) a 600 (cm)
Temperatura de operación	15°C a 70°C	0°C a 70°C
Tiempo de respuesta	20 (ms)	65 (ms)
Dimensiones	45*20*15 (mm)	20*20*16 (mm)
Características adicionales	N/A	Sensor de luz incorporado para medición de luz ambiente, ajustable en dirección I2C
Costo	$4	$30

Para seleccionar el multiplexor adecuado para el prototipo, se consideraron dos opciones: el módulo CD74HC4067 y el módulo Adafruit TCA9548A, cuya comparación se detalla en la Tabla 3.3.

El módulo CD74HC4067 permite la multiplexación de 16 canales digitales o analógicos en comparación con los 8 canales del módulo Adafruit TCA9548A, que utiliza la comunicación I2C. Sin embargo, algunos sensores que utilizan la interfaz I2C no permiten cambiar su dirección física, lo que impide tener dos dispositivos con la misma dirección en los mismos pines SDA/SCL, a menos que el dispositivo tenga una dirección física diferente o permita el cambio de dirección en el hardware. En teoría, el módulo Adafruit TCA9548A podría ser una opción viable para conectar varios sensores con la misma dirección física con comunicación I2C.

Por otro lado, el multiplexor CD74HC4067 es capaz de enviar los pulsos de activación a los sensores debido a que su interfaz puede ser digital o analógica y ofrece 16 canales, lo que lo convierte en una opción más favorable. Por ello, se justifica la elección del multiplexor CD74HC4067 para este prototipo, ya que se ajusta mejor a los requerimientos necesarios.

Tabla 3.3. Comparación entre módulo CD74HC4067 y TCA9548A [16] [17] [18]

Característica	CD74HC4067	Adafruit TCA9548A
Interfaz	Multiplexor Digital/Análogo (16 canales)	I_2C (bus de 8 canales)
Voltaje de operación	2 V a 6 V	1.65 V a 5.5 V
Corriente de operación	80 µA típico	Bajo consumo en reposo
Capacidad de canales	16 canales analógicos o digitales	8 canales bidireccionales
Comunicación	Control digital directo	Selección de canal a través de I2C
Latencia de selección de canal	Baja latencia en cambio de canales	Baja latencia en cambio de canales
Características adicionales	Multiplexado digital/analógico	• Protección contra interferencias en las señales de comunicación I2C • Capacidad de conectar o desconectar dispositivos al multiplexor I2C mientras el sistema está encendido
Aplicaciones típicas	Ampliar el número de entradas/salidas en microcontroladores	Conexión de múltiples dispositivos I2C en un solo bus

Selección del software

Para la elección del alojamiento web, se consideraron dos opciones: InfinityFree y Firebase, para lo cual se realizó una tabla comparativa con la finalidad de evaluar sus características, como se muestra en la Tabla 3.4.

InfinityFree ofrece alojamiento web gratuito, similar a Firebase, y proporciona una mayor capacidad de almacenamiento. Sin embargo, Firebase se destaca por su integración con bases de datos en tiempo real, una característica esencial para el funcionamiento del sistema implementado en el prototipo.

Firebase Hosting permite una integración directa con Firebase *Realtime Database*, facilitando la sincronización y actualización de datos en tiempo real sin necesidad de implementar soluciones adicionales. Esto es beneficioso para aplicaciones que requieren una actualización constante de datos, como el monitoreo de plazas de estacionamiento.

InfinityFree, aunque ofrece más almacenamiento para sitios web, no proporciona el mismo nivel de integración con bases de datos en tiempo real, por lo que la mejor opción para el prototipo es Firebase debido a su capacidad de manejo de bases de datos en tiempo real, lo que simplifica el desarrollo del prototipo.

Tabla 3.4. Comparación entre Firebase e InfinityFree [19] [20]

Característica	**Firebase**	**InfinityFree**
Costo	Gratuito	Gratuito
Almacenamiento	10 (GB)	5 (GB)
SSL gratuito	SÍ	SÍ
Soporte de bases de datos	MySQL	Base de datos en tiempo real

3.3 Diseño del prototipo de monitoreo

Esquema general del proyecto

El prototipo desarrollado se encarga de monitorizar continuamente el estado de ocupación de las plazas de estacionamiento utilizando sensores de distancia ultrasónicos. Estos sensores miden la distancia a los objetos (vehículos) en tiempo real. La señal de activación de los sensores se gestiona mediante un multiplexor, que organiza la secuencia correcta para disparar cada sensor, permitiendo así la escalabilidad del sistema para monitorizar un mayor número de espacios.

Una vez que los datos de los sensores son recogidos, son procesados por la placa ESP32. Esta placa, equipada con capacidad de conexión WiFi, se encarga de enviar los datos procesados a la base de datos en tiempo real proporcionada por Firebase. Este proceso de transmisión de datos se realiza de forma continua y sin interrupciones, garantizando que la información esté siempre actualizada.

La página web del sistema accede a esta base de datos en tiempo real para obtener el estado actual de las plazas de estacionamiento, una ilustración del proceso se puede observar en la Figura 3.1.

Esto permite a los usuarios consultar en cualquier momento la disponibilidad de espacios o a su vez mostrarle en la página web al usuario una alerta en caso de no existir ninguna disponibilidad en el estacionamiento.

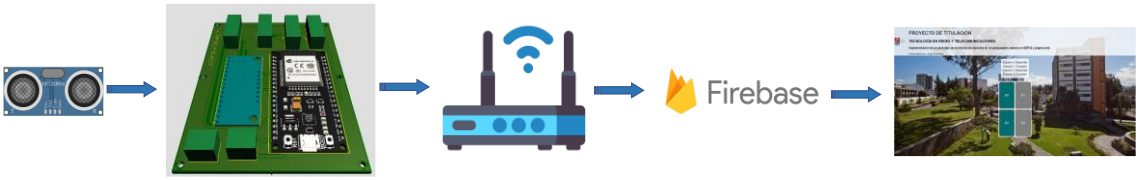


Figura 3.1 Esquema general del prototipo.

Circuito electrónico

El circuito electrónico del prototipo está compuesto por cuatro sensores ultrasónicos que se activan secuencialmente a través de un multiplexor, su conexión se visualiza en la Figura 3.2. El multiplexor se encarga de dirigir los pulsos de activación enviados por el microcontrolador ESP32, permitiendo la medición de distancias hacia objetos.

Sensores Ultrasónicos: Los sensores ultrasónicos están conectados al multiplexor. Cada sensor cuenta con un pin de activación, estos pines se conectan a los canales del multiplexor.

Multiplexor: El multiplexor está conectado a la ESP32 y actúa como intermediario entre los sensores y el microcontrolador. La ESP32 controla los canales del multiplexor mediante señales de selección de canal, lo que permite activar los sensores uno a uno de manera secuencial.

Placa ESP32*:* La ESP32 envía señales de control al multiplexor para seleccionar el canal correspondiente y activar el sensor deseado. Cada vez que se activa un sensor, este mide la distancia hasta un objeto y genera un pulso de salida.

Procesamiento de Datos: Los valores de duración del pulso de salida, que representan la distancia medida por cada sensor, son recolectados por la ESP32. Esta placa procesa los datos y los prepara para su transmisión.

Alimentación del circuito: Todos los componentes del circuito son alimentados con un voltaje de 5 (V) lo que simplifica la alimentación tanto de la ESP32, así como los sensores.

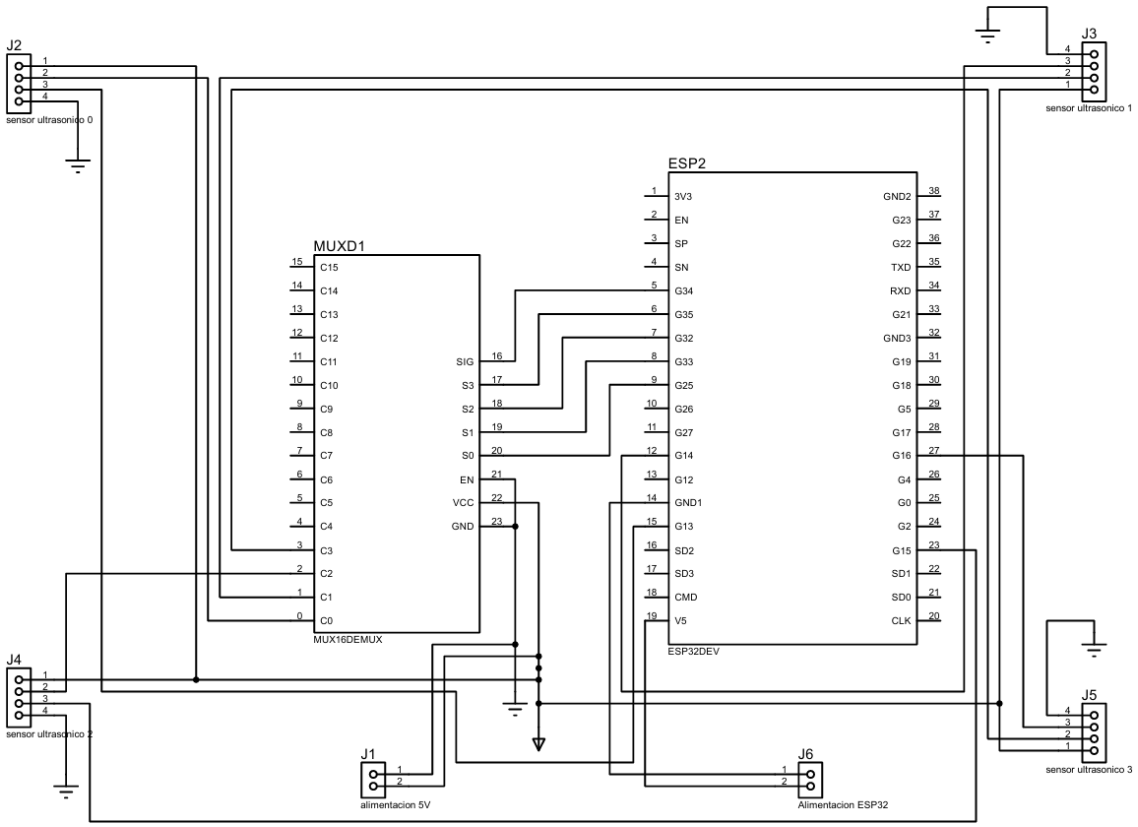


Figura 3.2 Circuito electrónico

Diseño de la placa

El diseño del circuito impreso se llevó a cabo utilizando el programa de simulación Proteus. Este software permite seleccionar la disposición de los componentes y definir las rutas que seguirán las pistas en el diseño de la placa PCB (*Printed Circuit Board*). Proteus ofrece la posibilidad de trazar las pistas de manera manual o automática; se optó por esta última opción para la impresión de la PCB, asegurando así la ausencia de cortocircuitos y optimizando el tamaño del circuito final.

Para minimizar el espacio ocupado, se diseñó una placa de doble cara. En el anverso, se dispusieron todas las pistas destinadas a la transmisión y recepción de información entre los componentes, como se muestra en la Figura 3.3. En el reverso, se ubicaron las pistas encargadas de la alimentación de los sensores y del multiplexor, tal como se ilustra en la Figura 3.4.

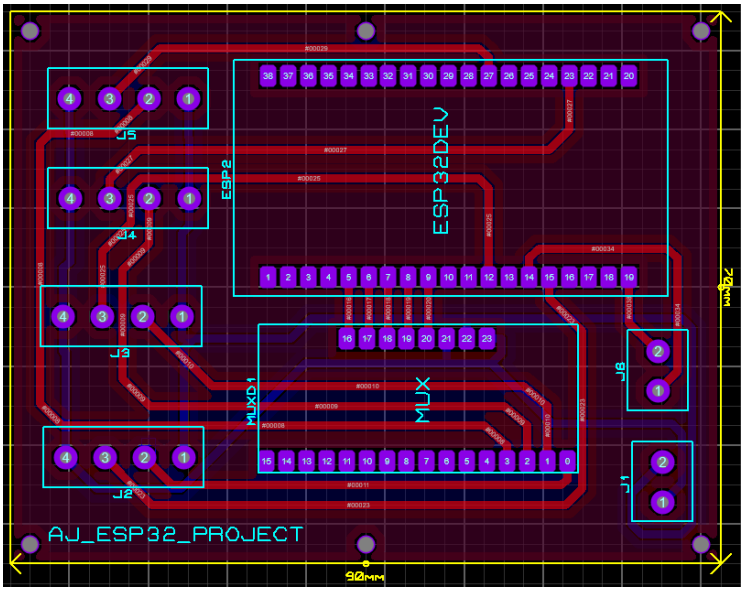


Figura 3.3 Anverso de la Placa de Circuito Impreso

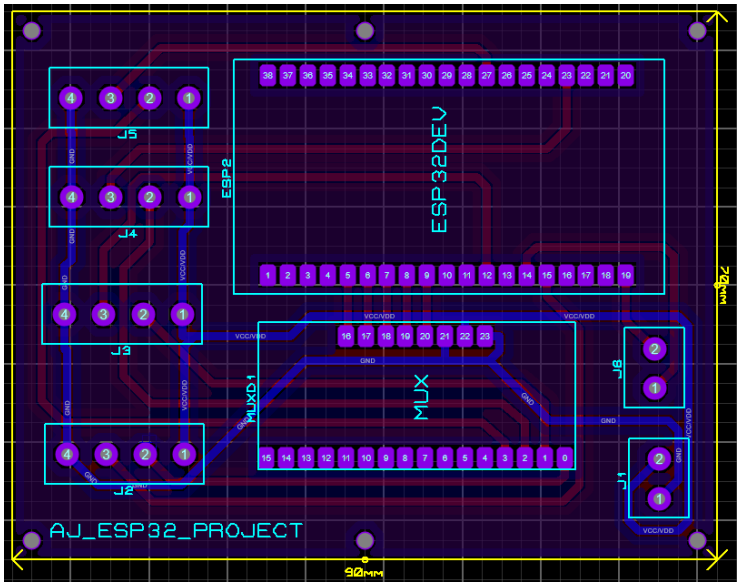


Figura 3.4 Reverso de la Placa de Circuito Impreso

Funcionamiento del sensor ultrasónico

El sensor ultrasónico HC-SR04, se utiliza ampliamente para la medición de distancias. Este sensor funciona emitiendo un pulso ultrasónico que viaja a través del aire y se refleja al encontrar un obstáculo. El sensor luego mide el tiempo que tarda el eco en regresar. El tiempo que el pin ECHO permanece en alto es medido por el microcontrolador, lo que permite calcular la distancia al objeto utilizando la fórmula de la Ecuación 3.1.

$$Distancia(m) = \frac{(Tiempo\ del\ pulso\ ECHO) \times (Velocidad\ del\ sonido = 340 m/s)}{2}$$

Ecuación 3.1. Medición de distancia en metros

Sin embargo, para simplificar su uso se requiere trabajar en centímetros, la fórmula se ajusta ligeramente usando la constante 0.034, que representa la velocidad del sonido en aire en cm/µs. El ajuste de la formula se visualiza en la Ecuación 3.2.

$$Distancia(cm) = \frac{(Tiempo\ del\ pulso\ ECHO) \times (0.034)}{2}$$

Ecuación 3.2. Medición de distancia en centímetros

La división por 2 se debe a que el tiempo medido incluye el viaje de ida y vuelta de la señal ultrasónica [21].

Diagrama de flujo del código fuente (configuración del sistema)

Este diagrama de flujo del prototipo muestra el proceso completo de inicialización y operación del sistema en la Figura 3.5, comenzando desde la configuración inicial hasta la habilitación de tareas específicas. Primero, se configuran los pines necesarios para el funcionamiento de los sensores ultrasónicos y el multiplexor, lo que incluye los pines de selección de canal (S0 y S1) y los pines de disparo y eco (TRIG_PIN y ECHO_PINS) de los sensores. Posteriormente, se inicializa el puerto serial para habilitar la comunicación y depuración a través de la consola.

El siguiente paso es conectar la placa ESP32 a una red WiFi. Si la conexión no se logra en el primer intento, el sistema espera un tiempo antes de intentar nuevamente, repitiendo este ciclo hasta que la conexión se establece correctamente. Una vez conectado a WiFi, se inicializa la conexión con Firebase, configurando así la base de datos en tiempo real que actualizará los datos de monitoreo.

Finalmente, se habilitan las tareas específicas del sistema: la tarea de medición de distancia (*measureDistanceTask*) y la tarea de envío de datos (*sendDataTask*). La tarea de medición se encarga de activar los sensores ultrasónicos secuencialmente a través del multiplexor, midiendo la distancia a los objetos y determinando la disponibilidad de las plazas de estacionamiento. La tarea de envío de datos procesa estas mediciones y las envía a Firebase a través de la conexión WiFi, asegurando que la información esté disponible en tiempo real para ser consultada en la página web.

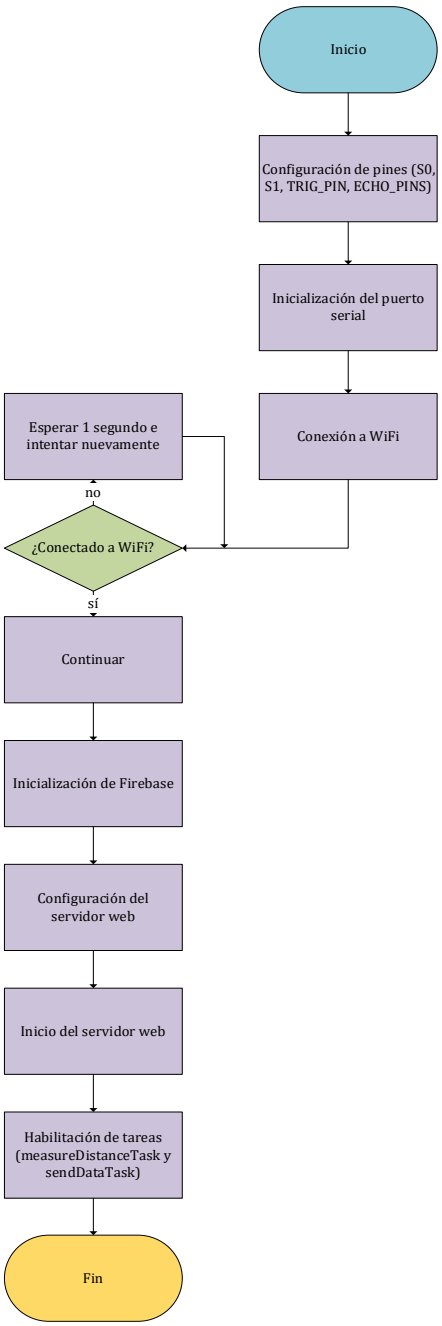


Figura 3.5 Diagrama de flujo de la configuración general del sistema

Diagrama de flujo del código fuente (Tarea medición de distancia)

El diagrama de flujo del ANEXO III para la tarea de medición de distancia detalla el proceso repetitivo mediante el cual se obtienen las distancias de los sensores ultrasónicos y se promedian para asegurar precisión. Al inicio de la tarea, se itera sobre cada sensor, utilizando un bucle que va de 0 a NUM_SENSORS-1. En cada iteración, se selecciona el canal correspondiente del multiplexor basado en el índice del sensor modulado por 4. Esto permite que el multiplexor active el sensor correcto.

A continuación, se envía un pulso de disparo al sensor correspondiente configurando el pin de disparo (TRIG_PIN) a nivel alto por 10 microsegundos y luego a nivel bajo. Este pulso activa el sensor ultrasónico para medir la distancia hasta un objeto, utilizando la función *pulseIn* para registrar el tiempo que tarda el pulso de eco en regresar al pin ECHO_PIN del sensor.

Si la distancia medida es mayor que cero, se suma esta distancia al acumulador (sumDistances[i]) correspondiente a ese sensor. Después de medir la distancia para todos los sensores en la iteración, se incrementa el contador (*counter*).

Cuando el contador alcanza el valor de 3, se procede a calcular la distancia promedio para cada sensor. Esto se realiza dividiendo el valor acumulado en sumDistances[i] entre 3. La distancia promedio calculada se almacena en distances[i], y luego se reinicia el acumulador de distancia (sumDistances[i]) y el contador (counter = 0). Este proceso se repite continuamente, asegurando que las mediciones de distancia sean admisibles.

Diagrama de flujo del código fuente (Tarea de envió de datos)

En el diagrama de flujo mostrado en la Figura 3.7 para la tarea de envío de datos (sendDataTask) se expone el proceso por el cual los datos recopilados de los sensores se envían a Firebase. Al inicio de la tarea, se itera sobre cada sensor, utilizando un bucle que va de 0 a NUM_SENSORS - 1.

Para cada sensor, se construye la ruta en Firebase donde se almacenarán los datos. Esta ruta se genera concatenando la cadena "/espacio" con el índice del sensor (i), formando así rutas como "/espacio0", "/espacio1", etc. Luego, se determina el estado del espacio de estacionamiento correspondiente a ese sensor. Si la distancia medida (distances[i]) es menor a 10, se considera que el espacio está "Ocupado", de lo contrario, se considera "Disponible".

Una vez determinado el estado del espacio, se envían estos datos a Firebase usando el método *Firebase.setString*. Si ocurre un error durante el envío de los datos, se muestra un mensaje de error en el puerto serial para alertar al usuario sobre el problema. Este proceso garantiza que la información sobre el estado de los espacios de estacionamiento se mantenga actualizada en tiempo real en la base de datos de Firebase.

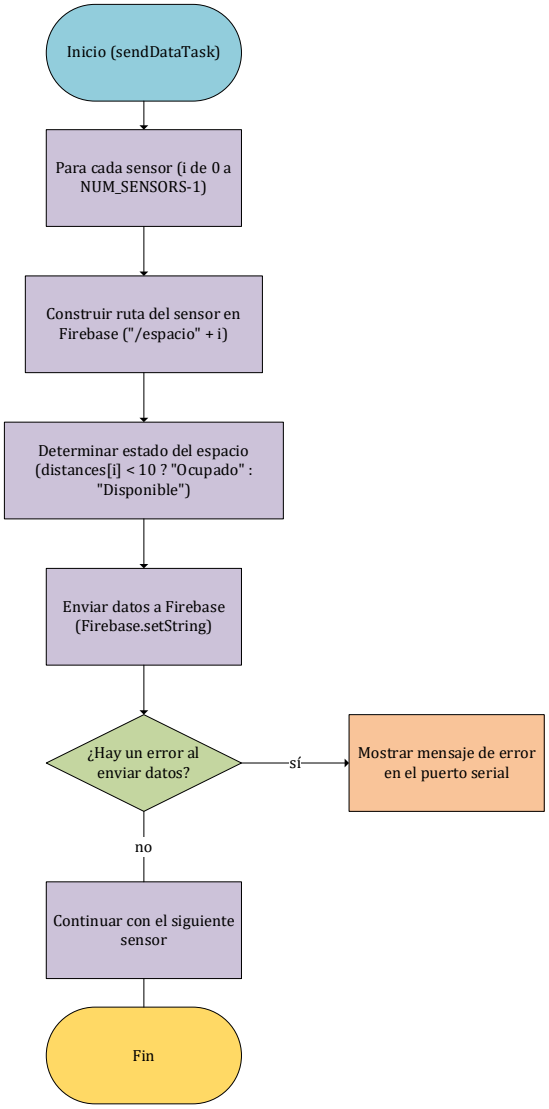


Figura 3.7 Diagrama de flujo del código para la tarea de envió de datos

Diagrama de flujo del código de la página web

En el diagrama de flujo del proceso de carga de la página HTML, mostrado en la Figura 3.8. Se describe la secuencia realizada en el lado del cliente para interactuar con Firebase y actualizar la interfaz de usuario en tiempo real. Al cargar la página HTML, primero se inicializa Firebase en el lado del cliente. A continuación, se define la función *actualizarDatos ()*, que es responsable de obtener una referencia a la base de datos de Firebase y configurar un *listener* para detectar cambios en los datos.

Cuando se producen cambios en la base de datos, el *listener* se activa, actualizando el texto de estado de los espacios de estacionamiento y sus colores correspondientes en la página web. Además, se verifica si todos los espacios están ocupados; si es así, se muestra una alerta al usuario, y si no, se oculta la alerta.

La función *actualizarDatos ()* se llama cuando se carga la ventana (*window.onload*), asegurando que los datos se sincronicen y se reflejen en la página web desde el inicio. Finalmente, se implementa un mecanismo para cambiar las imágenes de fondo cada 10 segundos, proporcionando una experiencia visual dinámica.

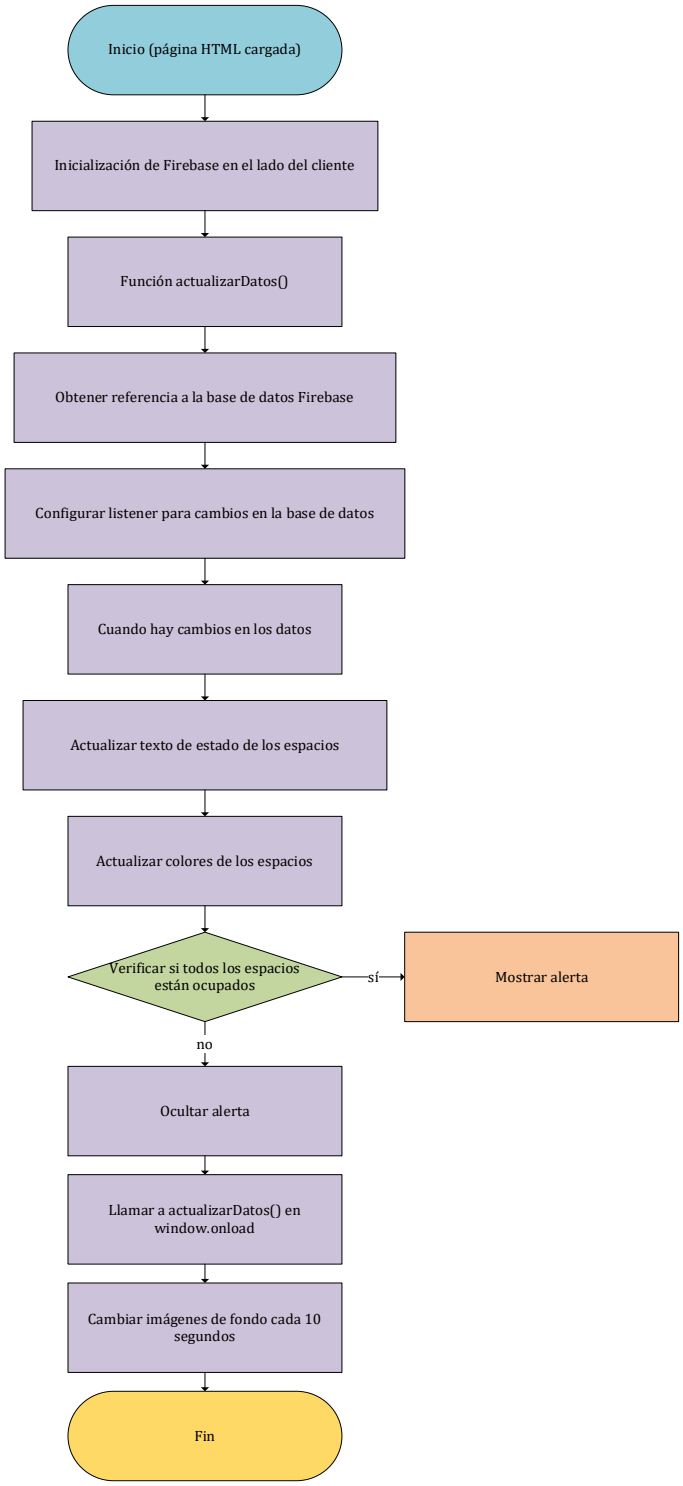


Figura 3.8 Diagrama de flujo del código de la página web

3.4 Implementación del prototipo

Configuración de la base de datos en tiempo real

Para configurar la base de datos en tiempo real, se creó un proyecto en Firebase utilizando una cuenta de Gmail existente, como se puede observar en la Figura 3.9. Después de completar este proceso, se obtuvieron los parámetros necesarios para inicializar la conexión a Firebase. Estos parámetros incluyen:

ApiKey: Esta clave API es utilizada para autenticar las solicitudes que se envían desde una aplicación a Firebase.

AuthDomain: Este dominio de autenticación se usa para gestionar la autenticación de usuarios, proporcionando un dominio específico a través del cual los usuarios pueden iniciar sesión.

DatabaseURL: Es la URL de la base de datos del en Firebase. Es para dirigir las solicitudes de la aplicación a la ubicación correcta en la nube.

ProjectId: Este es el identificador único del proyecto de Firebase. Se usa para identificar el proyecto en toda la plataforma Firebase.

StorageBucket: Este parámetro se refiere al espacio de almacenamiento en la nube proporcionado por Firebase para almacenar archivos.

MessagingSenderId: Este ID se utiliza en el servicio de mensajería en la nube de Firebase para identificar al remitente que envía mensajes a la aplicación.

AppId: Este es el identificador de la aplicación, único para cada aplicación en Firebase, y se utiliza para identificarla en toda la plataforma [22] [23].

Estos parámetros son esenciales y deben ser incorporados en el código de la página web para permitir el acceso y la interacción con la base de datos en tiempo real. De esta manera, la página web puede comunicarse eficazmente con Firebase, asegurando que los datos se mantengan actualizados y sincronizados en tiempo real.

Figura 3.9 Creación de la base de datos en tiempo real

Despliegue de la página web

Para la configuración del hosting, se instaló Node.js en el computador, que incluye el paquete npm (*Node Package Manager*), esencial para manejar los paquetes necesarios para este proceso. Tras la instalación de Node.js, se accedió a la consola de comandos con privilegios de administrador para ejecutar los comandos npm, los cuales son fundamentales para operar con Node.js.

Posteriormente, se creó un proyecto en la consola de Firebase, con lo cual se proporciona un conjunto de parámetros de inicialización que son vitales para establecer la conexión con la base de datos. Estos parámetros incluyen *ApiKey*, *AuthDomain*, *DatabaseURL*, *ProjectId, StorageBucket, MessagingSenderId* y *AppId*. Estos elementos son necesarios en el código de la página web para permitir el acceso y la interacción con la base de datos en tiempo real que ofrece Firebase.

Una vez configurado el proyecto en Firebase, como se puede visualizar en la Figura 3.10, se procedió a instalar las herramientas específicas de Firebase. Esto se hizo mediante la ejecución del comando *npm install -g firebase-tools* en la consola de comandos. Este paso es crucial ya que las herramientas de Firebase permiten gestionar el despliegue y la administración del sitio web. Con las herramientas instaladas, se creó una carpeta local en el sistema donde se almacenaron todos los archivos necesarios para la página web. En la consola de comandos, se navegó a esta carpeta y se inicializó Firebase con el comando *firebase init*.

Finalmente, para desplegar los archivos a Firebase, se utilizó el comando *firebase deploy*, el proceso de depliegue puede visualizarse en la Figura 3.11. Este comando sube todos los archivos desde la carpeta local al servidor de Firebase, haciendo que el sitio web esté disponible en línea. Al finalizar este proceso, Firebase proporciona una URL de hosting que se utiliza para acceder a la página web desplegada.

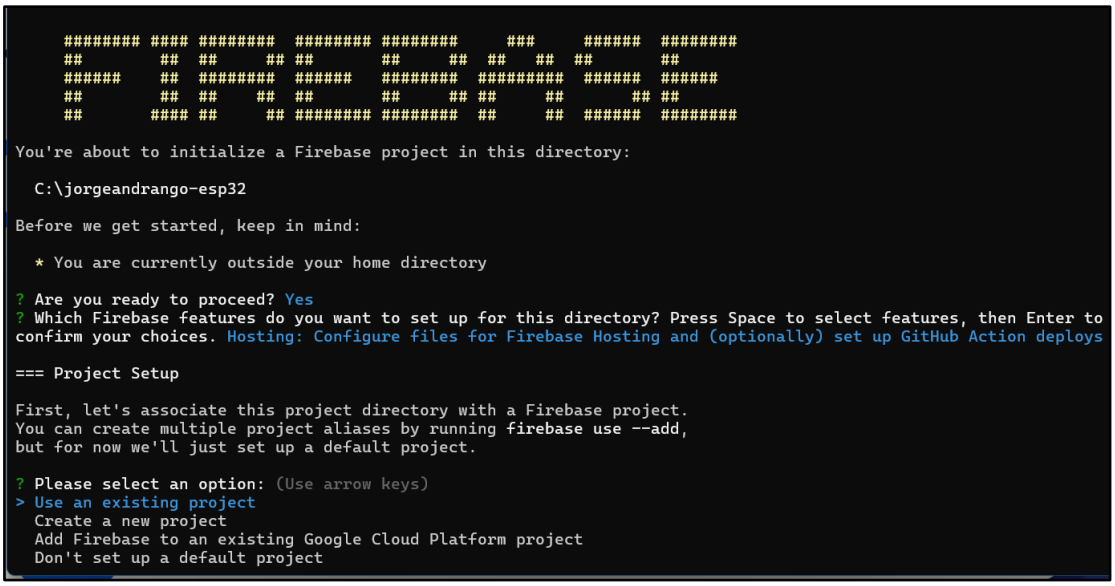


Figura 3.10 Inicialización del Proyecto de Firebase

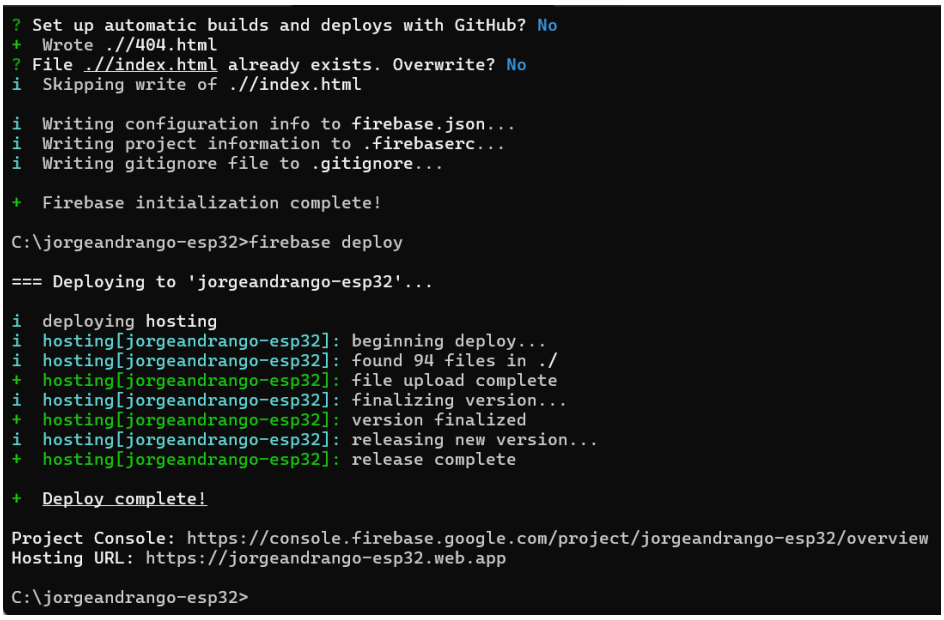


Figura 3.11 Despliegue de la página web en Firebase.

Código de programación en Arduino IDE

La porción de código mostrada en la Figura 3.12, incluye diversas librerías importantes para el funcionamiento del sistema de monitoreo de estacionamiento. La librería *WiFi.h* se utiliza para establecer la conexión WiFi en la placa ESP32, permitiendo la comunicación inalámbrica con otros dispositivos y servicios en la red. *FirebaseESP32.h* es necesaria para la integración con Firebase, lo que permite almacenar y recuperar datos en tiempo real desde la base de datos en la nube.

La librería *Wire.h* facilita la comunicación con los sensores ultrasónicos, mediante un protocolo de comunicación que permite la conexión de periféricos a la ESP32. *AsyncTCP.h* se emplea para manejar conexiones TCP asíncronas. Por otro lado, *ESPAsyncWebServer.h* permite la creación y manejo de un servidor web asíncrono.

Finalmente, *TaskScheduler.h* se utiliza para la gestión de tareas, permitiendo la programación y ejecución de múltiples tareas de forma concurrente, optimizando así el rendimiento del sistema.

```
#include <WiFi.h> // Librería para la conexión WiFi
#include <FirebaseESP32.h> // Librería para la conexión con Firebase en ESP32
#include <Wire.h> // Librería para la comunicación con sensores
#include <AsyncTCP.h> // Librería para el manejo de conexiones TCP asíncronas
#include <ESPAsyncWebServer.h> // Librería para el manejo de un servidor web asíncrono
#include <TaskScheduler.h> // Librería para la gestión de tareas
```

Figura 3.12 Inserción de librerías

La siguiente parte del código define los pines y variables necesarias para la operación del multiplexor CD74HC4067 y los sensores ultrasónicos, esto se puede visualizar en la Figura 3.13. Se asignan pines específicos de la ESP32 para controlar el multiplexor, con S0 y S1 como pines de selección, y TRIG_PIN como el pin de disparo del sensor ultrasónico.

El arreglo ECHO_PINS contiene los pines utilizados para recibir las señales de eco de los sensores ultrasónicos. Se define NUM_SENSORS como el número de sensores ultrasónicos conectados, en este caso, cuatro.

Para almacenar las distancias medidas por cada sensor, se utiliza un arreglo flotante *distances*. Adicionalmente, se declara un contador (*counter*) que ayuda en el cálculo del valor de las distancias medidas, y un arreglo *sumDistances* que acumula las distancias para cada sensor, esto facilita el procesamiento de los datos obtenidos.

```
// Definimos los pines del multiplexor CD74HC4067
  // Pin de selección S0
const int S0 = 25;
  // Pin de selección S1
const int S1 = 33;
  // Pin de disparo del sensor ultrasónico
const int TRIG_PIN = 34;
  // Pines de recepción de eco de los sensores ultrasónicos
const int ECHO_PINS[] = {13, 14, 15, 16};
  // Número de sensores
const int NUM_SENSORS = 4;
  // Arreglo para almacenar las distancias medidas por cada sensor
float distances[NUM_SENSORS];
  // Contador para la media de distancias
int counter = 0;
  // Suma acumulada de distancias para cada sensor
float sumDistances[NUM_SENSORS] = {0};
```

Figura 3.13 Declaración de pines

La siguiente sección del código mostrado en la Figura 3.14 define las credenciales necesarias para la conexión con el punto de acceso inalámbrico con la finalidad de comunicar la placa de desarrollo ESP32 a Internet.

```
// Credenciales de la red WiFi
const char* ssid = "SSID";
const char* password = "KEY";
```

Figura 3.14 Credenciales de la red WiFi

En este fragmento de código, mostrado en la Figura 3.16, se configura la conexión con Firebase, definiendo los parámetros esenciales para establecer la comunicación entre el ESP32 y la base de datos en tiempo real de Firebase. Se especifica la URL del host de Firebase mediante FIREBASE_HOST, y se proporciona la clave de autenticación con FIREBASE_AUTH. Estas constantes permiten que el dispositivo acceda y autentique su comunicación con la base de datos.

El objeto *firebaseData* de la clase *FirebaseData* se utiliza para manejar la transferencia de datos entre el ESP32 y Firebase, facilitando las operaciones de lectura y escritura en la base de datos. Posteriormente, *Scheduler taskScheduler* crea un planificador de tareas, el cual se utiliza para gestionar y ejecutar múltiples tareas de forma eficiente.

```
// Configuración de Firebase
  // URL del host de Firebase
#define FIREBASE_HOST "jorgeandrango-esp32-default-rtdb.firebaseio.com"
  // Clave de autenticación de Firebase
#define FIREBASE_AUTH "L8h5NsjUUjMprzxs84dsUzNk2JScsmNZUOzrmE2y"
  // Objeto para manejar los datos de Firebase
FirebaseData firebaseData;
  // Instancia del servidor web en el puerto 80
AsyncWebServer server(80);
  // Instancia del planificador de tareas
Scheduler taskScheduler;
```

Figura 3.16 Configuración de Firebase

En estas líneas de código se declaran dos tareas como se puede visualizar en la Figura 3.17, *measureDistanceTask* y *sendDataTask*, que serán utilizadas para medir distancias y enviar datos, respectivamente. Las tareas están programadas con intervalos específicos utilizando la biblioteca *TaskScheduler*. La primera tarea, t1, recurre a la función *measureDistanceTask* cada 250 milisegundos, lo que asegura que las distancias se midan a intervalos regulares y se mantenga la precisión en la monitorización de los espacios de estacionamiento.

La segunda tarea, t2, recurre a la función *sendDataTask* cada 1000 milisegundos, lo que permite enviar los datos a Firebase en intervalos de un segundo, asegurando que la información en la base de datos en tiempo real esté constantemente actualizada.

```
  // Declaración de la función de tarea para medir distancia
void measureDistanceTask();
  // Declaración de la función de tarea para enviar datos
void sendDataTask();

// Tareas programadas con intervalos específicos
  // Tarea para medir distancia cada 250 ms
Task t1(250, TASK_FOREVER, &measureDistanceTask, &taskScheduler);
  // Tarea para enviar datos cada 1000 ms
Task t2(1000, TASK_FOREVER, &sendDataTask, &taskScheduler);
```

Figura 3.17 Declaración de funciones para tareas

Este fragmento de código define la función *setup* (), como se puede observar en la Figura 3.18. La función se ejecuta una vez al iniciar el programa y es importante para configurar el entorno de operación del sistema. En primer lugar, los pines S0, S1 y TRIG_PIN se configuran como salidas utilizando la función *pinMode* (), mientras que los pines de eco (ECHO_PINS) se configuran como entradas. Esto permite que el microcontrolador controle el multiplexor y los sensores ultrasónicos adecuadamente.

A continuación, se inicializa la comunicación serial a 9600 baudios con *Serial.begin*(), lo cual es útil para la depuración y monitoreo de mensajes en la consola. Luego, se inicia la conexión WiFi con *WiFi.begin*() y se entra en un bucle que espera hasta que la conexión se establezca, imprimiendo mensajes de estado en el puerto serial durante el proceso.

Una vez establecida la conexión WiFi, se procede a inicializar Firebase con *Firebase.begin*(), permitiendo que el sistema se comunique con la base de datos en tiempo real. La función *Firebase.reconnectWiFi*() asegura la reconexión automática en caso de que la conexión WiFi se pierda.

Luego se habilitan las tareas de medición de distancia (*t1.enable*) y de envío de datos (*t2.enable*), lo que pone en marcha el ciclo de operación continua del sistema, asegurando que las mediciones y las actualizaciones a Firebase se realicen de manera periódica.

```
void setup() {
  // Configuración de pines como salida
  pinMode(S0, OUTPUT);
  pinMode(S1, OUTPUT);
  pinMode(TRIG_PIN, OUTPUT);
  for (int i = 0; i < NUM_SENSORS; i++) {
    pinMode(ECHO_PINS[i], INPUT); // Configuración de pines de eco como entrada
  }
  Serial.begin(9600); // Inicialización del puerto serial para depuración
  WiFi.begin(ssid, password); // Inicio de la conexión WiFi
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000); // Espera hasta que la conexión se establezca
    Serial.println("Conectando a la red WiFi...");
  }
  Serial.println("Conexión WiFi establecida"); // Mensaje de éxito de conexión WiFi
  Firebase.begin(FIREBASE_HOST, FIREBASE_AUTH); // Inicialización de Firebase
  Firebase.reconnectWiFi(true); // Reconexión automática de WiFi en caso de desconexión

  // Configuración del servidor web para servir la página HTML
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
    request->send_P(200, "text/html", index_html);
  });

  server.begin(); // Inicio del servidor web
  t1.enable(); // Habilitación de la tarea de medición de distancia
  t2.enable(); // Habilitación de la tarea de envío de datos
}
```

Figura 3.18 Configuración *void setup*

Las líneas de código de la Figura 3.19, definen el bucle principal del programa, identificado por la función *loop*(). Dentro de este bucle, se llama al método *execute*() del objeto *taskScheduler*, lo que permite la ejecución continua de las tareas programadas. Este método garantiza que las tareas configuradas previamente en el planificador de tareas se ejecuten en sus intervalos definidos, manteniendo el sistema funcionando sin interrupciones.

```
void loop() {
  taskScheduler.execute(); // Ejecución de las tareas programadas
}
```

Figura 3.19 Ejecución de las tareas programadas

Esta sección de código define la función *measureDistanceTask*(), como que se puede visualizar en la Figura 3.20, la cual es responsable de medir las distancias utilizando los sensores ultrasónicos. En primer lugar, se itera a través de cada sensor, seleccionando el canal adecuado en el multiplexor mediante la función *selectChannel*(). Luego, se envía un pulso de disparo al sensor activando el pin de disparo (TRIG_PIN) por 10 microsegundos. Posteriormente, se mide la distancia a través del pin de eco correspondiente (ECHO_PINS), y se almacena el valor en el arreglo *distances.*

Luego de obtener las distancias, estas se suman en el arreglo *sumDistances* siempre que sean mayores a cero. El contador counter se incrementa en cada iteración. Cuando el contador alcanza el valor de 3, se calcula el promedio de las distancias almacenadas dividiendo la suma acumulada por 3. Este promedio se almacena en el arreglo *distances* y se reinicia tanto el contador como las sumas acumuladas para las próximas mediciones. Este proceso asegura que los datos de distancia sean más precisos al promediar múltiples lecturas.

```
// Función para medir la distancia usando los sensores ultrasónicos
void measureDistanceTask() {
  for (int i = 0; i < NUM_SENSORS; i++) {
    selectChannel(i % 4); // Selección del canal del multiplexor
    digitalWrite(TRIG_PIN, HIGH); // Envío del pulso de disparo
    delayMicroseconds(10); // Espera de 10 microsegundos
    digitalWrite(TRIG_PIN, LOW); // Fin del pulso de disparo
    distances[i] = measureDistance(ECHO_PINS[i]); // Medición de la distancia
  }

  // Suma de distancias medidas para calcular el promedio
  for (int i = 0; i < NUM_SENSORS; i++) {
    if (distances[i] > 0.0) {
      sumDistances[i] += distances[i];
    }
  }
  counter++; // Incremento del contador
  if (counter == 3) { // Si se han tomado 3 mediciones
    for (int i = 0; i < NUM_SENSORS; i++) {
      float averageDistance = sumDistances[i] / 3.0; // Cálculo del promedio
      sumDistances[i] = 0; // Reinicio de la suma de distancias
      distances[i] = averageDistance; // Almacenamiento de la distancia promedio
    }
    counter = 0; // Reinicio del contador
  }
}
```

Figura 3.20 Función para medir la distancia

En la Figura 3.21 se observa la sección del código que implementa la función *sendDataTask*(), cuyo propósito es enviar los datos recopilados por los sensores ultrasónicos a la base de datos en tiempo real de Firebase. El proceso comienza iterando a través de cada sensor definido en el sistema (NUM_SENSORS). Para cada sensor, se construye la ruta de almacenamiento en Firebase utilizando el índice del sensor, resultando en una ruta como "/espacio0", "/espacio1", etc.

Luego, se determina el estado del espacio monitoreado por cada sensor, estableciendo lo siguiente; si la distancia medida es menor a 10 centímetros, el espacio se considera "Ocupado"; de lo contrario, se considera "Disponible". Esta información de estado se envía a Firebase usando la función *Firebase.setString*(), la cual toma como parámetros el objeto *firebaseData*, la ruta específica del sensor y el estado calculado.

Después de intentar enviar los datos, el código verifica si el código de respuesta HTTP es diferente de 200, lo que indicaría un error en el proceso de envío. Si se detecta un error, se imprime un mensaje en el monitor serial con los detalles del error utilizando *firebaseData.errorReason*(). Este mecanismo asegura que cualquier fallo en la comunicación con Firebase sea registrado para su posterior análisis y corrección.

```
// Función para enviar los datos de distancia a Firebase
void sendDataTask() {
  for (int i = 0; i < NUM_SENSORS; i++) {
    // Ruta en Firebase para cada sensor
    String sensorPath = "/espacio" + String(i);
    // Determinación del estado del espacio
    String status = (distances[i] < 10) ? "Ocupado" : "Disponible";
    // Envío de datos a Firebase
    Firebase.setString(firebaseData, sensorPath.c_str(), status);
    // Verificación de errores en el envío de datos
    if (firebaseData.httpCode() != 200) {
      // Mensaje de error
      Serial.println("Error al enviar datos: " + firebaseData.errorReason());
    }
  }
}
```

Figura 3.21 Función para enviar los datos a Firebase

La función *selectChannel*() está diseñada para configurar los pines de selección del multiplexor. Esta función toma *channel* como parámetro, el cual indica cuál de los canales del multiplexor debe activarse. Dentro de la función, se utiliza *digitalWrite*() para establecer el nivel lógico de los pines S0 y S1. La función *bitRead*() extrae el valor de un bit específico del número de canal, permitiendo así que los pines S0 y S1 se configuren de acuerdo al canal seleccionado, como se visualiza en la Figura 3.22.

```
// Función para seleccionar el canal del multiplexor
void selectChannel(int channel) {
  digitalWrite(S0, bitRead(channel, 0)); // Configuración del pin S0
  digitalWrite(S1, bitRead(channel, 1)); // Configuración del pin S1
}
```

Figura 3.22 Función para seleccionar el canal del multiplexor

La función *measureDistance*() está pensada para calcular la distancia a un objeto utilizando los sensores ultrasónicos. Primero, la función mide la duración del pulso de eco usando *pulseIn*(), que registra el tiempo que tarda la señal ultrasónica en regresar al sensor después de haber sido emitida. Luego, la distancia se calcula mediante la Ecuación 3.2., en donde 0.034 representa la velocidad del sonido en el aire en centímetros por microsegundo. El resultado se divide entre 2 porque el tiempo medido

incluye el viaje de ida y vuelta de la señal. Finalmente, la función retorna la distancia medida en centímetros, como se visualiza en Figura 3.23.

```
// Función para medir la distancia usando un sensor ultrasónico
float measureDistance(int echoPin) {
  // Medición de la duración del pulso de eco
  long duration = pulseIn(echoPin, HIGH);
  float distance = duration * 0.034 / 2;
  // Cálculo de la distancia en cm, 0.034
  //Es una constante que representa la velocidad del sonido
  return distance; // Retorno de la distancia medida
}
```

Figura 3.23 Función para medir las distancias con sensores

Código desarrollado para la página web

El código utilizado para el despliegue de la página web en Firebase está compuesto principalmente por HTML y también JavaScript. Sin embargo, JavaScript añade interactividad a la página web, permitiendo la manipulación dinámica del contenido y la comunicación en tiempo real con Firebase por lo que JavaScript permite actualizar la interfaz del usuario basándose en los datos recibidos de los sensores del estacionamiento, mostrando el estado de ocupación de los espacios en tiempo real.

El estilo de la página web se caracteriza por un diseño simple y moderno, la página utiliza fuente Arial. Utiliza una estructura flexible y centrada, con imágenes de fondo que cubren toda la pantalla y el fondo está configurado con fotografías del campus de la Escuela Politécnica Nacional. Los encabezados y textos están claramente definidos con colores y márgenes que mejoran la legibilidad. El encabezado de la página contiene el sello institucional y texto alineados horizontalmente en un fondo semitransparente, lo que facilita la identificación de la sección principal.

Las áreas que muestran el estado del estacionamiento están organizadas en una cuadrícula con dos columnas, donde cada espacio está estilizado para indicar si está ocupado o disponible mediante colores distintivos. Los textos de estado están centrados y envueltos en un contenedor con un fondo blanco semitransparente. Las alertas, se destacan con un fondo rojo y texto blanco, diseñadas para ser visibles solo cuando no hay espacios disponibles, como se muestra en la Figura 3.24.

```
        #alerta {
            display: none;
            margin-top: 20px;
            padding: 10px;
            background-color: red;
            color: white;
            border-radius: 10px;
            text-align: center;
            font-size: 18px;
        }
    </style>
```

Figura 3.24 Estilo de las alertas

El fragmento de código mostrado en la Figura 3.25, configura los parámetros necesarios para inicializar Firebase en la página web. Define un objeto *firebaseConfig* que contiene las claves y URL específicas del proyecto de Firebase, incluyendo la clave API, el dominio de autenticación, la URL de la base de datos en tiempo real, el ID del proyecto, el contenedor de almacenamiento, el ID del remitente de mensajes y el ID de la aplicación. Luego, mediante *firebase.initializeApp*(), se inicializa Firebase con la configuración proporcionada. Luego, se crea una instancia de la base de datos utilizando *firebase.database*(), permitiendo así que la página web interactúe con la base de datos en tiempo real para leer y escribir datos.

Este proceso es importante para establecer la comunicación entre el ESP32 y la base de datos de Firebase, asegurando que la información del estado del estacionamiento se actualice y refleje en la interfaz de usuario en tiempo real.

```
<script>
    var firebaseConfig = {
        apiKey: "AIzaSyB6VShuwPajNbdFv3TX1153A8v0q7P0Xwk",
        authDomain: "jorgeandrango-esp32.firebaseapp.com",
        databaseURL: "https://jorgeandrango-esp32-default-rtdb.firebaseio.com",
        projectId: "jorgeandrango-esp32",
        storageBucket: "jorgeandrango-esp32.appspot.com",
        messagingSenderId: "128663252811",
        appId: "1:128663252811:web:a79f1a2d0ed58d5fbc02ce"
    };
    firebase.initializeApp(firebaseConfig);
    var database = firebase.database();
```

Figura 3.25 Parámetros necesarios para inicializar Firebase

La función *actualizarDatos()* está diseñada para sincronizar y mostrar el estado de los espacios de estacionamiento en tiempo real utilizando Firebase, como se muestra en la Figura 3.26. Al referenciar la raíz de la base de datos con *database.ref*('/'), se establece un oyente que se activa cada vez que hay cambios en los datos. Dentro de la función de *callback*, los datos del *snapshot* se asignan a una variable *data*, y posteriormente se actualizan los elementos HTML correspondientes con los valores actuales de los espacios de estacionamiento.

Por ejemplo, los elementos con identificadores únicos (IDs) *text-espacio0, text-espacio1*, etc., se actualizan con los estados correspondientes (Ocupado o Disponible). Esto permite que la interfaz de usuario refleje en tiempo real el estado actual de cada espacio de estacionamiento, asegurando que la información esté siempre actualizada.

```
function actualizarDatos() {
    database.ref('/').on('value', function(snapshot) {
        var data = snapshot.val();
        // Actualizar el texto
        document.getElementById('text-espacio0').innerText = 'Espacio 0: ' + data.espacio0;
        document.getElementById('text-espacio1').innerText = 'Espacio 1: ' + data.espacio1;
        document.getElementById('text-espacio2').innerText = 'Espacio 2: ' + data.espacio2;
        document.getElementById('text-espacio3').innerText = 'Espacio 3: ' + data.espacio3;
```

Figura 3.26 Función para actualizarDatos

El fragmento de código de la Figura 3.27, actualiza los colores de los espacios de estacionamiento, su propósito es reflejar visualmente el estado de cada espacio en la interfaz de usuario. Utilizando el método *document.getElementById*, se seleccionan los elementos HTML correspondientes a cada espacio de estacionamiento por sus elementos con identificadores únicos (espacio0, espacio1, etc.). Luego, cada elemento se establece condicionalmente en función de los datos obtenidos de Firebase. Si el valor es "Ocupado", se asigna la clase *parking-spot occupied*, y si el valor es diferente, se asigna la clase *parking-spot empty*. Este mecanismo permite que cada espacio de estacionamiento cambie su color automáticamente, proporcionando una indicación visual clara del estado de disponibilidad de los espacios.

```
// Actualizar los colores de los espacios
document.getElementById('espacio0').className = data.espacio0 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';
document.getElementById('espacio1').className = data.espacio1 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';
document.getElementById('espacio2').className = data.espacio2 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';
document.getElementById('espacio3').className = data.espacio3 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';
```

Figura 3.27 Actualización de colores en la interfaz de usuario

En las siguientes líneas de código, como se puede visualizar en la Figura 3.28, se verifica el estado de todos los espacios de estacionamiento para determinar si están ocupados. Utilizando una condición lógica *if*, se comprueba si los valores de *data.espacio0*, *data.espacio1*, *data.espacio2* y *data.espacio3* son todos "Ocupado". Si esta condición se cumple, se muestra un mensaje de alerta cambiando el estilo del elemento HTML con el identificador 'alerta' a *display: block*. En caso contrario, se oculta el mensaje de alerta estableciendo *display: none.* Este mecanismo proporciona una forma dinámica de notificar a los usuarios cuando no hay espacios disponibles en el estacionamiento.

```
// Verificar si todos los espacios est n ocupados
if (data.espacio0 === "Ocupado" && data.espacio1 === "Ocupado" && data.espacio2 === "Ocupado" && data.espacio3 === "Ocupado") {
    document.getElementById('alerta').style.display = 'block';
} else {
    document.getElementById('alerta').style.display = 'none';
}
```

Figura 3.28 Verificación de los estados en la interfaz de usuario

El siguiente bloque de código, expuesto en la Figura 3.29, se ejecuta cuando la página web se carga completamente (*window.onload*). Primero, se llama a la función *actualizarDatos()* para obtener y mostrar los datos actuales del estacionamiento. Luego, se establece un intervalo que cambia la imagen de fondo de la página cada 10 segundos. La función *setInterval* alterna la imagen de fondo entre 'background1.jpg' y 'background2.jpeg' verificando cuál está actualmente aplicada y cambiando a la otra, esto añade una animación dinámica y atractiva al sitio web.

```
window.onload = function() {
    actualizarDatos();
    // Cambiar im genes de fondo din micamente
    setInterval(() => {
        document.body.style.backgroundImage = document.body.style.backgroundImage.includes('background1.jpg') ? "url('background2.jpeg')" :
"url('background1.jpg')";
    }, 10000); // Cambiar cada 10 segundos
};
```

Figura 3.29 Función para alternar la imagen de fondo

El siguiente fragmento de código, presentado en la Figura 3.30, se encarga de mostrar el estado de varios espacios en un estacionamiento mediante el uso de una estructura de divisores (div) y párrafos (p). El elemento div actúa como un contenedor genérico que agrupa otros elementos HTML, en este caso, párrafos identificados individualmente. El elemento div con la clase *status-text* agrupa varios párrafos, y cada uno de estos párrafos tiene un identificador único que corresponde a un espacio específico del estacionamiento (por ejemplo, text-espacio0 para el Espacio 0).

Estos identificadores permiten actualizar dinámicamente el contenido de cada párrafo a medida que se reciben datos en tiempo real desde Firebase, indicando si el espacio está ocupado o disponible. Inicialmente, cada espacio se muestra como "N/A", lo que indica que no hay información disponible hasta que se actualicen los datos.

```
<div class="status-text">
    <p id="text-espacio0">Espacio 0: N/A</p>
    <p id="text-espacio1">Espacio 1: N/A</p>
    <p id="text-espacio2">Espacio 2: N/A</p>
    <p id="text-espacio3">Espacio 3: N/A</p>
</div>
```

Figura 3.30 Actualización dinámica del texto mostrado

El fragmento de código presentado en la Figura 3.31, contiene dos divisores (div) que tienen propósitos específicos para la visualización del estado del estacionamiento. El primer div, con el identificador alerta, muestra un mensaje de advertencia ("No hay espacios disponibles en el parqueadero") cuando todos los espacios están ocupados. El segundo div, con la clase *parking-lot*, organiza visualmente los distintos espacios de estacionamiento, cada uno representado por un div con la clase *parking-spot* y un identificador único (espacio0, espacio1, espacio2, espacio3). Estos elementos permiten la actualización dinámica del estado de cada espacio, indicando si están ocupados o disponibles.

```
<div id="alerta">No hay espacios disponibles en el parqueadero.</div>
<div class="parking-lot">
    <div id="espacio0" class="parking-spot">E0</div>
    <div id="espacio1" class="parking-spot">E1</div>
    <div id="espacio2" class="parking-spot">E2</div>
    <div id="espacio3" class="parking-spot">E3</div>
</div>
```

Figura 3.31 Líneas de código para visualización del estado del estacionamiento

Montaje de componentes del Prototipo

Para la elaboración de la placa de circuito impreso, con dimensiones de 8 (cm) de ancho por 10 (cm) de largo, se utilizó el diseño desarrollado en Proteus. Este diseño fue exportado en formato PDF para comprobar el correcto dimensionamiento del espacio entre los pines, asegurando así una impresión precisa de la PCB, como se muestra en la Figura 3.32.

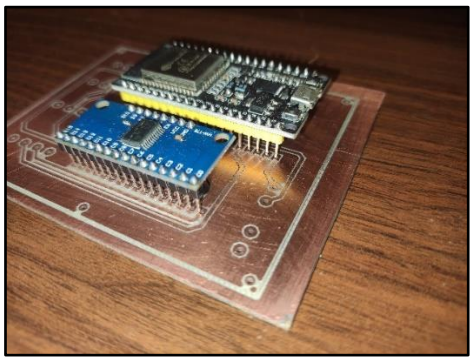


Figura 3.32 Placa de circuito impreso

Posteriormente, se procedió a soldar los conectores hembra para el multiplexor y el microcontrolador, así como los cables necesarios para los cuatro sensores ultrasónicos. Una vez situados los diferentes componentes en la placa, necesarios para el funcionamiento del prototipo, como se observa en la Figura 3.33, se verificó que no existieran errores en la soldadura ni impurezas en el circuito que pudieran causar

cortocircuitos y lecturas erróneas por parte de los sensores ultrasónicos. En la Figura 3.34 se presenta el montaje del circuito impreso junto con los componentes en la maqueta del estacionamiento.

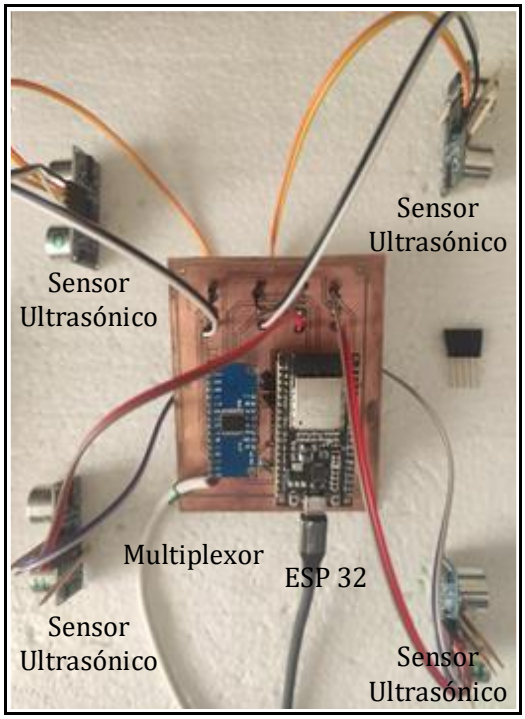


Figura 3.33 Montaje de componentes en la placa de circuito impreso

Figura 3.34 Ensamble de la placa de circuito impreso en la maqueta

3.5 Pruebas de funcionamiento

Conexión Wi-Fi

En la Figura3.35se presentan los mensajes del ESP32 en el monitor serial, los cuales indican una conexión inalámbrica exitosa. Estos mensajes incluyen el nombre de la red de área local inalámbrica (SSID) y la dirección IP local asignada al dispositivo. Estos detalles se muestran únicamente si la placa ESP32 logra conectarse correctamente al punto de acceso inalámbrico, proporcionando así información sobre el estado de la conexión.

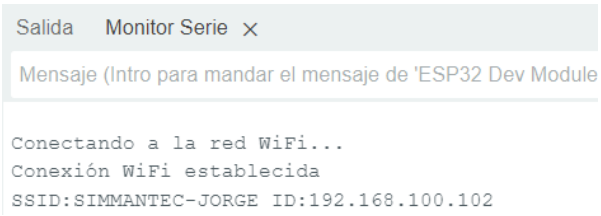


Figura3.35 Conexión inalámbrica ESP32

Determinación de plazas disponibles

La determinación de los espacios disponibles del prototipo se puede observar en la Figura 3.36. En el prototipo se implementaron sensores ultrasónicos para medir la distancia entre un objeto y el sensor. La lógica utilizada establece que si la distancia entre un objeto y el sensor es mayor a 10 (cm), el espacio se considera disponible; de lo contrario, se considera ocupado. Esto ofrece una solución efectiva para determinar el estado de los espacios en el estacionamiento.

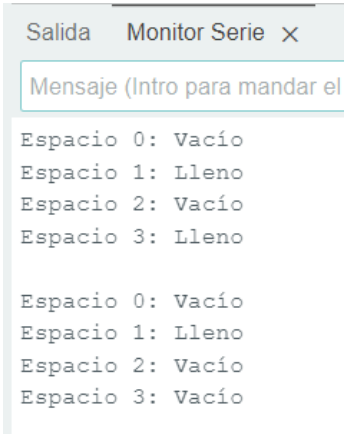


Figura 3.36 Determinación de espacios disponibles mediante texto.

Envió de datos a Firebase

En la página web de Firebase, se muestran continuamente las actualizaciones del estado del estacionamiento, tal como se ilustra en la Figura 3.37. Si ocurre un error en la transmisión de datos, la placa ESP32 está programada para mostrar un mensaje de error *'bad request'* en el monitor serial. Este mensaje de error es visible siempre que se tenga acceso a los datos proporcionados por el monitor serial.

Figura 3.37 Base de datos en tiempo real

Alojamiento de página web

En la Figura 3.38 se visualiza la disponibilidad de la página web y en la Figura 3.39 se muestra la página web la cual indica el estado de los espacios de estacionamiento de dos maneras: textual y visual. El color verde representa que el espacio está disponible, mientras que el color gris indica que el espacio está ocupado. Los datos se actualizan continuamente cada segundo, lo que permite que los cambios en los estados se reflejen de inmediato, proporcionando una experiencia fluida para el usuario.

```
PS C:\Users\aj_z-> ping jorgeandrango-esp32.web.app

Haciendo ping a jorgeandrango-esp32.web.app [199.36.158.100] con 32 bytes de datos:
Respuesta desde 199.36.158.100: bytes=32 tiempo=27ms TTL=51
Respuesta desde 199.36.158.100: bytes=32 tiempo=22ms TTL=51
Respuesta desde 199.36.158.100: bytes=32 tiempo=22ms TTL=51
Respuesta desde 199.36.158.100: bytes=32 tiempo=18ms TTL=51

Estadísticas de ping para 199.36.158.100:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
Tiempos aproximados de ida y vuelta en milisegundos:
    Mínimo = 18ms, Máximo = 27ms, Media = 22ms
PS C:\Users\aj_z->
```

Figura 3.38 Verificación de la Disponibilidad de la página web

Figura 3.39 Página web del monitoreo de espacios de estacionamiento

Alertas

La Figura 3.40 ilustra la funcionalidad dinámica del sistema de parqueo. Cuando todos los espacios están ocupados, se despliega un mensaje de advertencia que indica: 'No hay espacios disponibles en el parqueadero'. Esta notificación es temporal y se desactiva automáticamente en cuanto se libera alguna plaza de estacionamiento. De este modo, el sistema mantiene informados a los usuarios siempre sobre la disponibilidad de espacios.

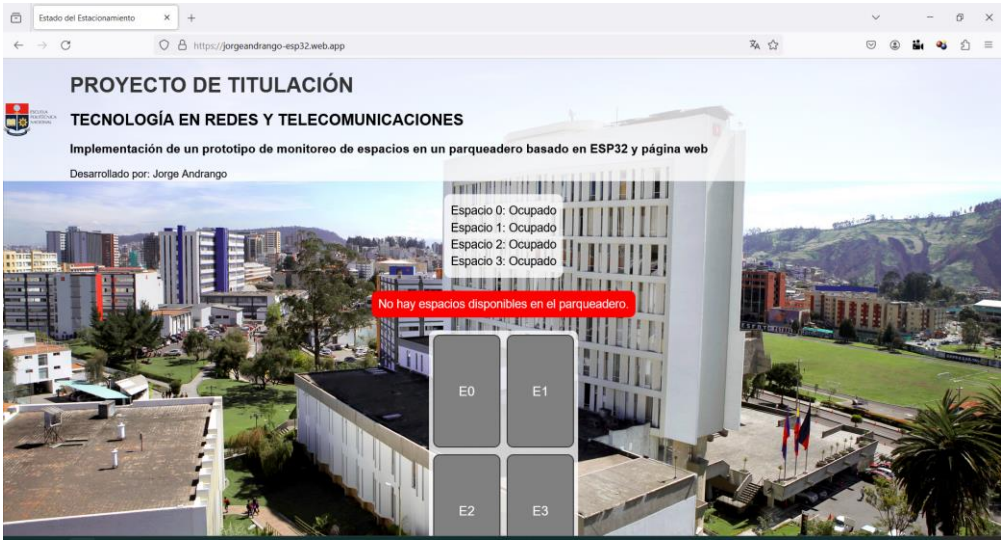


Figura 3.40 Visualización del mensaje de disponibilidad nula en Página web

4 CONCLUSIONES

- El análisis de los requerimientos como; conexión inalámbrica, determinación de plazas, actualización en tiempo real, servicio de hosting y alertas permitieron identificar las necesidades específicas del sistema de monitoreo de estacionamiento, tomando en consideración todos los aspectos técnicos y funcionales, como la capacidad de detectar y comunicar la disponibilidad de espacios. La identificación precisa de estos requerimientos fue crucial para el éxito del proyecto, asegurando que el prototipo respondiera adecuadamente a las necesidades planteadas.

- La selección de componentes de hardware, como los sensores ultrasónicos, el multiplexor, junto con las herramientas de software, como Firebase para la base de datos en tiempo real y HTML para la interfaz web, fueron elegidos por su compatibilidad y capacidad de satisfacer los requerimientos identificados.
- El diseño del prototipo involucró la planificación de diagramas de flujo detallados y la creación del circuito mediante el uso herramientas como Proteus para el diseño de la PCB. Esta aplicación permitió optimizar la disposición de los componentes, lo que resultó en una disminución de errores potenciales y en la reducción del tamaño del circuito impreso. El diseño bien estructurado fue un paso esencial que facilitó la posterior implementación y pruebas del prototipo.
- La implementación del prototipo en una maqueta permitió validar el diseño del prototipo en un entorno controlado. Este proceso incluyó la soldadura precisa de los componentes y la integración de todos los sistemas en una configuración funcional.
- Las pruebas confirmaron que el prototipo funciona según los requerimientos, las pruebas incluyeron la verificación de la conexión inalámbrica a Internet, la determinación de plazas en estacionamiento, la actualización de los datos en tiempo real, el acceso a una página web para la verificación de la ocupación de espacios y las alertas en la interfaz web. Estas pruebas validaron la implementación del prototipo y su capacidad para ser utilizado en aplicaciones reales.

5 RECOMENDACIONES

- Investigar sobre como cambiar la dirección del sensor para permitir la conexión de múltiples dispositivos o sensores en el mismo bus de datos.
- Se recomienda evaluar las potenciales aplicaciones de la biblioteca *TaskScheduler* ya que facilita la progracion de múltiples tareas en intervalos específicos, tambien permite estructurar el código de manera modular, separando diferentes funcionalidades en tareas distintas.
- Dado que el sistema utiliza un multiplexor, es importante evaluar la escalabilidad del prototipo para aplicaciones en estacionamientos más grandes y complejos.
- Considerar el potencial de este sistema para ser integrado en soluciones de conjuntos habitacionales, urbanizaciones y centros comerciales ya que podría mejorar la gestión de los espacios y reducir el tiempo de espera de los usuarios.
- Se recomienda revisar la documentación de la biblioteca FirebaseESP32 ya que las actualizaciones podrían no ser compatibles con el código anteriormente escrito en el contexto de realizar actualizaciones al sistema o incorporar nuevas funcionalidades y mejoras para los usuarios.

REFERENCIAS BIBLIOGRÁFICAS

[1] P. S. Cáceres, «ARDUINO: Sensor ultrasónico HC-SR04,» Blog de Tecnología - IES José Arencibia Gil - Telde, 6 febrero 2018. [En línea]. Available: https://www3.gobiernodecanarias.org/medusa/ecoblog/fsancac/2018/02/06/arduino-sensor-ultrasonico-hc-sr04/. [Último acceso: 6 julio 2024].

[2] «¿Qué es el Internet de las cosas (IoT) y cómo funciona?,» 7 julio 2024. [En línea]. Available: https://www.redhat.com/es/topics/internet-of-things/what-is-iot. [Último acceso: 7 julio 2024].

[3] C. Electricos, «ESP32 - Especificaciones y diseños,» Circuitos Eléctricos, 4 junio 2020. [En línea]. Available: https://www.circuitos-electricos.com/esp32-especificaciones-y-disenos/. [Último acceso: 7 julio 2024].

[4] D. Says, «What is ESP32, how it works and what you can do with ESP32?,» Circuit sCHOOLS, 4 abril 2023. [En línea]. Available: https://www.circuitschools.com/what-is-esp32-how-it-works-and-what-you-can-do-with-esp32/. [Último acceso: 7 julio 2024].

[5] D. Carrasco, «Arduino: TaskScheduler, no más millis ni delay,» ElectroSoftCloud, 20 marzo 2020. [En línea]. Available: https://www.electrosoftcloud.com/arduino-taskscheduler-no-mas-millis-ni-delay/. [Último acceso: 7 julio 2024].

[6] I. Mecafenix, «Que es un sensor ultrasónico y como funciona,» Ingeniería Mecafenix, 1 junio 2021. [En línea]. Available: https://www.ingmecafenix.com/automatizacion/sensores/ultrasonico/. [Último acceso: 7 julio 2024].

[7] «Sensor Ultrasonido HC-SR04,» Naylamp Mechatronics - Perú, [En línea]. Available: https://naylampmechatronics.com/sensores-proximidad/10-sensor-ultrasonido-hc-sr04.html. [Último acceso: 7 julio 2024].

[8] «Modulo Multiplexor Digital/Analógico 16 Canales CD74HC4067,» Electronilab, [En línea]. Available: https://electronilab.co/tienda/modulo-multiplexor-digital-analogico-16-canales-cd74hc4067/. [Último acceso: 7 julio 2024].

[9] «Real-Time Database,» Hazelcast, [En línea]. Available: https://hazelcast.com/glossary/real-time-database/. [Último acceso: 7 julio 2024].

[10] «¿Qué es el alojamiento web? - Explicación sobre el servicio de alojamiento web - AWS,» Amazon Web Services, Inc., [En línea]. Available: https://aws.amazon.com/es/what-is/web-hosting/. [Último acceso: 7 julio 2024].

[11] G. B, «¿Qué es HTML? Explicación de los fundamentos del Lenguaje de marcado de hipertexto,» Tutoriales Hostinger, 16 noviembre 2018. [En línea]. Available: https://www.hostinger.es/tutoriales/que-es-html. [Último acceso: 7 julio 2024].

[12] E. v. E. W. M. I. R. f. You?, «Last Minute Engineers,» 1 mayo 2024. [En línea]. Available: https://lastminuteengineers.com/esp32-vs-esp8266-comparison/. [Último acceso: 7 julio 2024].

[13] «ESP32 vs ESP8266 ¿Cuales son las diferencias entre ambos módulos?,» 17 junio 2020. [En línea]. Available: https://descubrearduino.com/esp32-vs-esp8266/. [Último acceso: 7 julio 2024].

[14] alldatasheet.com, «HC-SR04 Datasheet(PDF),» [En línea]. Available: http://www.alldatasheet.com/datasheet-pdf/pdf/1132203/ETC2/HC-SR04.html. [Último acceso: 7 julio 2024].

[15] alldatasheet.com, «SRF08 Datasheet(PDF),» [En línea]. Available: http://www.alldatasheet.com/datasheet-pdf/pdf/1222415/ETC2/SRF08.html. [Último acceso: 7 julio 2024].

[16] «Visión general | Adafruit TCA9548A Ruptura del multiplexor I2C de 1 a 8 | Sistema de aprendizaje Adafruit,» [En línea]. Available: https://learn.adafruit.com/adafruit-tca9548a-1-to-8-i2c-multiplexer-breakout/overview. [Último acceso: 8 julio 2024].

[17] alldatasheet.com, «TCA9548A Download,» [En línea]. Available: http://pdf1.alldatasheet.com/datasheet-pdf/download/463225/TI1/TCA9548A.html. [Último acceso: 8 julio 2024].

[18] «74HC4067 Multiplexor Análogo 16 Canales,» UNIT Electronics, [En línea]. Available: https://uelectronics.com/producto/74hc4067-multiplexor-analogico-digital-16-canales/. [Último acceso: 8 julio 2024].

[19] «Conozca los niveles de uso, cuotas y precios de Hosting | Firebase Hosting,» Firebase, [En línea]. Available: https://firebase.google.com/docs/hosting/usage-quotas-pricing?hl=es. [Último acceso: 8 julio 2024].

[20] «InfinityFree Reseña: ¿Ves lo que dicen más de 59 personas al respecto?,» GoogieHost, [En línea]. Available: https://googiehost.com/es/blog/infinityfree-una-estrategia-SEO-para-aparecer-en-las-b%C3%BAsquedas-de-Google./. [Último acceso: 8 julio 2024].

[21] P. S. Cáceres, «ARDUINO: Sensor ultrasónico HC-SR04,» 6 febrero 2018. [En línea]. Available: https://www3.gobiernodecanarias.org/medusa/ecoblog/fsancac/2018/02/06/arduino-sensor-ultrasonico-hc-sr04/. [Último acceso: 8 julio 2024].

[22] «¿Por qué pasamos apikey, databaseurl, storageBucket, domain, appId, messagingSenderId en un proyecto de Firebase con Node.js?,» 20 julio 2023. [En línea]. Available: https://es.davy.ai/por-que-pasamos-apikey-databaseurl-storagebucket-domain-appid-messagingsenderid-en-un-proyecto-de-firebase-con-node-js/. [Último acceso: 8 julio 2024].

[23] «Documentación de Firebase,» Firebase, [En línea]. Available: https://firebase.google.com/docs?hl=es-419. [Último acceso: 8 julio 2024].

6 ANEXOS

La lista de los **Anexos** se muestra a continuación:

ANEXO I. Certificado de originalidad

ANEXO II. Enlaces

ANEXO III. Diagrama de flujo de la Tarea medición de distancia

ANEXO IV. Código Fuente

ANEXO V. Código de la página web

ANEXO I: Certificado de Originalidad

CERTIFICADO DE ORIGINALIDAD

Quito, D.M. 29 de julio de 2024

De mi consideración:

Yo, ANDRÉS FERNANDO REYES CASTRO, en calidad de Director del Trabajo de Integración Curricular titulado IMPLEMENTACIÓN DE UN PROTOTIPO DE MONITOREO DE ESPACIOS EN UN PARQUEADERO BASADO EN ESP32 Y PÁGINA WEB elaborado por la estudiante JORGE LUIS ANDRANGO QUILUMBA de la carrera en Tecnología Superior en Redes y Telecomunicaciones, certifico que he empleado la herramienta Turnitin para la revisión de originalidad del documento escrito completo, producto del Trabajo de Integración Curricular indicado.

El documento escrito tiene un índice de similitud del 12%.

Es todo cuanto puedo certificar en honor a la verdad, pudiendo el interesado hacer uso del presente documento para los trámites de titulación.

NOTA: Se adjunta el link del informe generado por la herramienta Turnitin.

LINK

Atentamente,

ANDRES FERNANDO REYES CASTRO

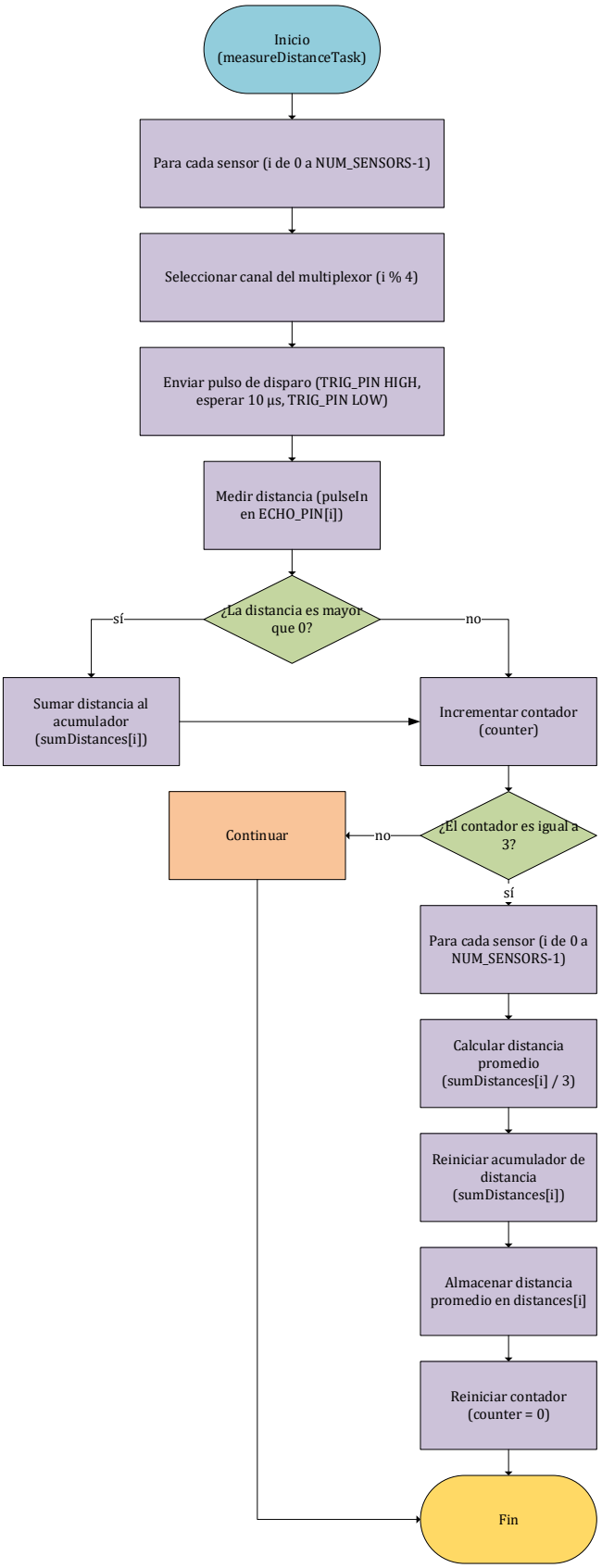
Docente

Escuela de Formación de Tecnólogos

ANEXO II: Enlaces

Anexo II.I Código QR de la implementación y pruebas de funcionamiento

ANEXO III: Diagrama de flujo de la tarea medición de distancia



ANEXO IV: Código Fuente

```
//Proyecto de grado

//Jorge Andrango

#include <WiFi.h> // Librería para la conexión WiFi

#include <FirebaseESP32.h> // Librería para la conexión con Firebase en ESP32

#include <Wire.h> // Librería para la comunicación I2C

#include <AsyncTCP.h> // Librería para el manejo de conexiones TCP asíncronas

#include <ESPAsyncWebServer.h> // Librería para el manejo de un servidor web asíncrono

#include <TaskScheduler.h> // Librería para la gestión de tareas


#include "data.h" // Incluye el archivo que contiene la página web en formato HTML


// Definimos los pines del multiplexor CD74HC4067

const int S0 = 25; // Pin de selección S0

const int S1 = 33; // Pin de selección S1

const int TRIG_PIN = 12; // Pin de disparo del sensor ultrasónico

const int ECHO_PINS[] = {13, 14, 15, 16}; // Pines de recepción de eco de los sensores ultrasónicos

const int NUM_SENSORS = 4; // Número de sensores

float distances[NUM_SENSORS]; // Arreglo para almacenar las distancias medidas por cada sensor

int counter = 0; // Contador para la media de distancias

float sumDistances[NUM_SENSORS] = {0}; // Suma acumulada de distancias para cada sensor


// Credenciales de la red WiFi

const char* ssid = "AABB";

const char* password = "XXYY";


// Configuración de Firebase

#define FIREBASE_HOST "jorgeandrango-esp32-default-rtdb.firebaseio.com" // URL del host de Firebase

#define FIREBASE_AUTH "L8h5NsjUUjMprzxs84dsUzNk2JScsmNZUOzrmE2y" // Clave de autenticación de Firebase
```

```
FirebaseData firebaseData; // Objeto para manejar los datos de Firebase
AsyncWebServer server(80); // Instancia del servidor web en el puerto 80
Scheduler taskScheduler; // Instancia del planificador de tareas

void measureDistanceTask(); // Declaración de la función de tarea para medir distancia
void sendDataTask(); // Declaración de la función de tarea para enviar datos

// Tareas programadas con intervalos específicos
Task t1(250, TASK_FOREVER, &measureDistanceTask, &taskScheduler); // Tarea para medir distancia cada 250 ms
Task t2(1000, TASK_FOREVER, &sendDataTask, &taskScheduler); // Tarea para enviar datos cada 1000 ms

void setup() {
  // Configuración de pines como salida
  pinMode(S0, OUTPUT);
  pinMode(S1, OUTPUT);
  pinMode(TRIG_PIN, OUTPUT);
  for (int i = 0; i < NUM_SENSORS; i++) {
    pinMode(ECHO_PINS[i], INPUT); // Configuración de pines de eco como entrada
  }
  Serial.begin(9600); // Inicialización del puerto serial para depuración
  WiFi.begin(ssid, password); // Inicio de la conexión WiFi
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000); // Espera hasta que la conexión se establezca
    Serial.println("Conectando a la red WiFi...");
  }
  Serial.println("Conexión WiFi establecida"); // Mensaje de éxito de conexión WiFi
  Firebase.begin(FIREBASE_HOST, FIREBASE_AUTH); // Inicialización de Firebase
  Firebase.reconnectWiFi(true); // Reconexión automática de WiFi en caso de desconexión

  // Configuración del servidor web para servir la página HTML
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
    request->send_P(200, "text/html", index_html);
  });
```

```
  server.begin(); // Inicio del servidor web
  t1.enable(); // Habilitación de la tarea de medición de distancia
  t2.enable(); // Habilitación de la tarea de envío de datos
}

void loop() {
  taskScheduler.execute(); // Ejecución de las tareas programadas
}

// Función para medir la distancia usando los sensores ultrasónicos
void measureDistanceTask() {
  for (int i = 0; i < NUM_SENSORS; i++) {
    selectChannel(i % 4); // Selección del canal del multiplexor
    digitalWrite(TRIG_PIN, HIGH); // Envío del pulso de disparo
    delayMicroseconds(10); // Espera de 10 microsegundos
    digitalWrite(TRIG_PIN, LOW); // Fin del pulso de disparo
    distances[i] = measureDistance(ECHO_PINS[i]); // Medición de la distancia
  }

  // Suma de distancias medidas para calcular el promedio
  for (int i = 0; i < NUM_SENSORS; i++) {
    if (distances[i] > 0.0) {
      sumDistances[i] += distances[i];
    }
  }

  counter++; // Incremento del contador
  if (counter == 3) { // Si se han tomado 3 mediciones
    for (int i = 0; i < NUM_SENSORS; i++) {
      float averageDistance = sumDistances[i] / 3.0; // Cálculo del promedio
      sumDistances[i] = 0; // Reinicio de la suma de distancias
      distances[i] = averageDistance; // Almacenamiento de la distancia promedio
    }
    counter = 0; // Reinicio del contador
```

```
  }
}

// Función para enviar los datos de distancia a Firebase
void sendDataTask() {
  for (int i = 0; i < NUM_SENSORS; i++) {
    String sensorPath = "/espacio" + String(i); // Ruta en Firebase para cada sensor
    String status = (distances[i] < 10) ? "Ocupado" : "Disponible"; // Determinación del estado del espacio
    Firebase.setString(firebaseData, sensorPath.c_str(), status); // Envío de datos a Firebase
    if (firebaseData.httpCode() != 200) { // Verificación de errores en el envío de datos
      Serial.println("Error al enviar datos: " + firebaseData.errorReason()); // Mensaje de error
    }
  }
}

// Función para seleccionar el canal del multiplexor
void selectChannel(int channel) {
  digitalWrite(S0, bitRead(channel, 0)); // Configuración del pin S0
  digitalWrite(S1, bitRead(channel, 1)); // Configuración del pin S1
}

// Función para medir la distancia usando un sensor ultrasónico
float measureDistance(int echoPin) {
  long duration = pulseIn(echoPin, HIGH); // Medición de la duración del pulso de eco
  float distance = duration * 0.034 / 2; // Cálculo de la distancia en cm, 0.034 Es una constante que representa la velocidad del sonido en el aire en centímetros por microsegundo, Se divide por 2 porque el tiempo medido es el tiempo total que tarda la señal ultrasónica en viajar hacia el objeto y regresar al sensor.
  return distance; // Retorno de la distancia medida
}
```

ANEXO V: Código de la página web

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Estado del Estacionamiento</title>
    <script src="https://www.gstatic.com/firebasejs/8.6.8/firebase-app.js"></script>
    <script src="https://www.gstatic.com/firebasejs/8.6.8/firebase-database.js"></script>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            flex-direction: column;
            align-items: center;
            background: url('background1.jpg') no-repeat center center fixed;
            background-size: cover;
            margin: 0;
            padding: 0;
        }
        h1 {
            margin-top: 20px;
            color: #333;
        }
        #header {
            display: flex;
            align-items: center;
            width: 100%;
            padding: 2px;
            background-color: rgba(255, 255, 255, 0.8);
        }
        #header img {
            height: 50px;
            margin-right: 20px;
        }
        .status-text {
            margin-top: 20px;
            text-align: center;
```

```
    background-color: rgba(255, 255, 255, 0.8);
    padding: 10px;
    border-radius: 10px;
}
.status-text p {
    font-size: 18px;
    margin: 5px 0;
}
.parking-lot {
    display: grid;
    grid-template-columns: repeat(2, 1fr);
    gap: 10px;
    margin-top: 20px;
    background-color: rgba(255, 255, 255, 0.8);
    padding: 6px;
    border-radius: 10px;
}
.parking-spot {
    width: 100px;
    height: 170px;
    border: 2px solid #333;
    display: flex;
    justify-content: center;
    align-items: center;
    font-size: 20px;
    color: white;
    border-radius: 10px;
}
.empty {
    background-color: #006D70;
}
.occupied {
    background-color: gray;
}
#alerta {
```

```
            display: none;
            margin-top: 20px;
            padding: 10px;
            background-color: red;
            color: white;
            border-radius: 10px;
            text-align: center;
            font-size: 18px;
        }
    </style>
    <script>
        var firebaseConfig = {
            apiKey: "AIzaSyB6VShuwPajNbdFv3TX1153A8v0q7P0Xwk",
            authDomain: "jorgeandrango-esp32.firebaseapp.com",
            databaseURL: "https://jorgeandrango-esp32-default-rtdb.firebaseio.com",
            projectId: "jorgeandrango-esp32",
            storageBucket: "jorgeandrango-esp32.appspot.com",
            messagingSenderId: "128663252811",
            appId: "1:128663252811:web:a79f1a2d0ed58d5fbc02ce"
        };
        firebase.initializeApp(firebaseConfig);
        var database = firebase.database();

        function actualizarDatos() {
            database.ref('/').on('value', function(snapshot) {
                var data = snapshot.val();
                // Actualizar el texto
                document.getElementById('text-espacio0').innerText = 'Espacio 0: ' +
data.espacio0;
                document.getElementById('text-espacio1').innerText = 'Espacio 1: ' +
data.espacio1;
                document.getElementById('text-espacio2').innerText = 'Espacio 2: ' +
data.espacio2;
                document.getElementById('text-espacio3').innerText = 'Espacio 3: ' +
data.espacio3;
                // Actualizar los colores de los espacios
```

```
            document.getElementById('espacio0').className = data.espacio0 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';
            document.getElementById('espacio1').className = data.espacio1 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';
            document.getElementById('espacio2').className = data.espacio2 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';
            document.getElementById('espacio3').className = data.espacio3 === "Ocupado" ? 'parking-spot occupied' : 'parking-spot empty';


            // Verificar si todos los espacios est n ocupados
            if (data.espacio0 === "Ocupado" && data.espacio1 === "Ocupado" && data.espacio2 === "Ocupado" && data.espacio3 === "Ocupado") {
                document.getElementById('alerta').style.display = 'block';
            } else {
                document.getElementById('alerta').style.display = 'none';
            }
        });
    }

    window.onload = function() {
        actualizarDatos();
        // Cambiar im genes de fondo din micamente
        setInterval(() => {
            document.body.style.backgroundImage = document.body.style.backgroundImage.includes('background1.jpg') ? "url('background2.jpeg')" : "url('background1.jpg')";
        }, 10000); // Cambiar cada 10 segundos
    };
  </script>
</head>
<body>
  <div id="header">
    <img src="logo.svg" alt="Logo Universidad">
    <div>
      <h1>PROYECTO DE TITULACI&Oacute;N</h1>
      <h2>TECNOLOG&Iacute;A EN REDES Y TELECOMUNICACIONES</h2>
      <h3>Implementaci&oacute;n de un prototipo de monitoreo de espacios en un parqueadero basado en ESP32 y p&aacute;gina web</h3>
```

```
            <div id="name-line">Desarrollado por: Jorge Andrango</div>
        </div>
    </div>
    <div class="status-text">
        <p id="text-espacio0">Espacio 0: N/A</p>
        <p id="text-espacio1">Espacio 1: N/A</p>
        <p id="text-espacio2">Espacio 2: N/A</p>
        <p id="text-espacio3">Espacio 3: N/A</p>
    </div>
    <div id="alerta">No hay espacios disponibles en el parqueadero.</div>
    <div class="parking-lot">
        <div id="espacio0" class="parking-spot">E0</div>
        <div id="espacio1" class="parking-spot">E1</div>
        <div id="espacio2" class="parking-spot">E2</div>
        <div id="espacio3" class="parking-spot">E3</div>
    </div>
</body>
</html>
```