TRABAJO FIN DE MÁSTER

TECNOLOGÍA ENERGÉTICA PARA DESARROLLO SOSTENIBLE

APLICACIÓN IOT PARA EL DIAGNÓSTICO DE MOTORES DE INDUCCIÓN

AUTOR: **CHÁFER FERRANDO, BENJAMÍN ADÁN**

TUTOR: PINEDA SÁNCHEZ, MANUEL

COTUTOR: SAPENA BAÑÓ, ÁNGEL

Curso Académico: 2020-21

"Fecha 12/2020"

Agradecimientos

Me gustaría agradecer a mi familia por hacerme la vida más fácil durante la cuarentena y a mis amigos de *la Armada Pío*, *los Caballeros de la Tabla Periódica*, *GIE es una Sorpres*, *Misión Imposible*, *Élite Torrent*, a Andrea, a Marian y a Fernando por todo el apoyo que he recibido por su parte y a mi amigo Sergio Sanz de Tudela por obligarme a aprender JavaScript que tanto me ha servido en la elaboración de este proyecto.

Además, tampoco quiero olvidarme del Departamento de Ingeniería Eléctrica de la UPV, principalmente de mis tutores, por haberme ayudado en todo lo posible, incluso en estos tiempos tan complicados para la docencia.

Espero no haberme dejado a nadie.

Resumen

Se propone un sistema de monitorización y detección de averías para motores de inducción, no intrusivo, consistente en el seguimiento de la intensidad del estátor y la velocidad de giro y basado en Internet of Things (IoT). La detección de posibles averías se llevará a cabo mediante un análisis del espectro de la frecuencia por FFT con el fin de localizar armónicos asignados a fallos en el motor debido a una rotura de barras en el rotor. El equipo estará formado por tecnologías de código abierto de bajo coste, como NodeMCU, Node-RED, Raspberry Pi y MQTT y permitirá garantizar acceso a la información a los operarios a través de una aplicación web compatible con los navegadores de internet más frecuentes.

Palabras clave: Monitorización, motor de inducción, MCSA, Internet of Things, ESP32, Arduino, MQTT, Node-RED.

Resum

Es proposa un sistema de monitorització i detecció d'averies per a motors d'inducció, no intrusiu, consistent en el seguiment de la intensitat de l'estator i la velocitat de gir i basat en Internet of Things (IoT). La detecció d'averies es portarà a càrrec mitjançant un anàlisi de l'espectre de la freqüència per FFT amb el fi de localitzar harmònics assignats a fallades en el motor per causa d'un trencament de barres al rotor. L'equip estarà format per tecnologies de codi obert de baix cost, com NodeMCU, Node-RED, Raspberry Pi i MQTT i permetrà garantir l'accés a la informació als operaris a través d'una aplicació web compatible amb els navegadors d'internet més freqüents.

Paraules clau: Monitorització, motor d'inducció, MCSA, Internet of Things, ESP32, Arduino, MQTT, Node-RED.

Abstract

It is being proposed a non-intruding system of monitoring and fail detection consistent in the tracing of the current of the stator and the spin speed based in Internet of Things (IoT). Fail detection is getting done by performing a frequency spectrum analysis through an FFT in order to locate the harmonics assigned to motor failures due to rotor bar break. The equipment is formed by low cost open source technologies, as NodeMCU, Node-RED, Raspberry Pi and MQTT and also will grant the workers access to the information through a web application compatible with the most used web navigators.

Keywords: Monitoring, induction motor, MCSA, Internet of Things, ESP32, Arduino, MQTT, Node-RED.

Índice

Índice de figuras

Índice de tablas

Índice de ecuaciones

APLICACIÓN IOT PARA EL DIAGNÓSTICO DE MOTORES DE INDUCCIÓN

MEMORIA

Autor:
Benjamín Cháfer Ferrando

Tutores:
Manuel Pineda Sánchez
Ángel Sapena Bañó

Capítulo 1. Introducción

1.1. Antecedentes

Los motores eléctricos de inducción son, en la actualidad, un recurso ampliamente utilizado en la industria. Tal es así que el 65% de la energía generada en la Unión Europea (y en cantidades similares en el resto del mundo) va destinada a abastecer dichos motores, como se puede observar en la Tabla 1.

Tabla 1. Porcentaje de energía dedicado a motores eléctricos en varios países [1].

País	Porcentaje
EEUU	75 %
Reino Unido	50 %
Unión Europea	65 %
Jordania	31 %
Malasia	48 %
Turquía	65 %
Eslovenia	52 %
Canadá	80 %
India	70 %
China	60 %
Corea	40 %
Brasil	49 %
Australia	30 %
Sudáfrica	60 %

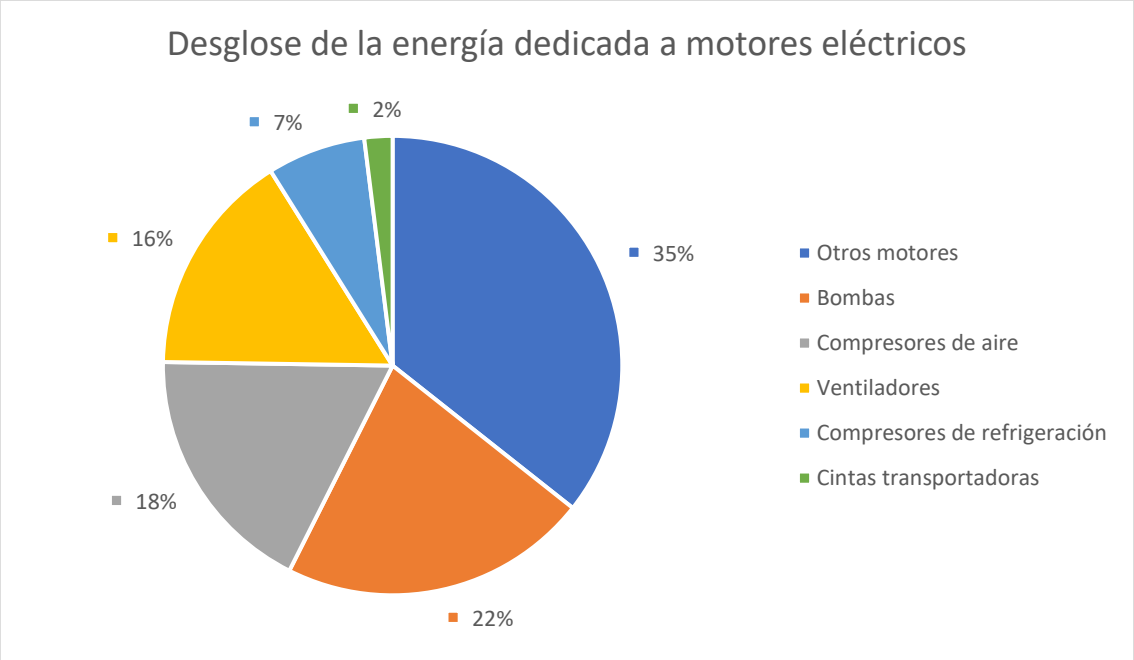


Figura 1. Desglose de la energía dedicada a motores eléctricos en el sector industrial en la Unión Europea [1].

Dados estos datos, es evidente que los motores de inducción son una parte elemental para la industria global, pues, tal como se muestra en la Figura 1, estos se emplean en aplicaciones muy variadas. No es extraño su uso tan expandido, ya que los motores de inducción presentan varias ventajas respecto a los demás tipos de motores como, por ejemplo, su bajo coste de adquisición y mantenimiento, su robustez, su simplicidad de construcción y la posibilidad de funcionar como generador de energía eléctrica, como sucede, por ejemplo, en los aerogeneradores o en las centrales termoeléctricas y nucleares.

No obstante, pese a su gran fiabilidad, resulta imposible anular la posibilidad de fallo. A causa de esto, cuando un motor falla de forma imprevista, es necesario realizar paros en el proceso industrial con el fin de realizar el mantenimiento del motor afectado o de sustituirlo en caso de que el fallo sea irreparable. Cabe destacar que cada parada puede reducir o incluso detener por completo la producción o actividad. Para evitar estos inconvenientes, en la actualidad se está implementando el uso de sistemas de monitorización, on-line y mediante comunicación sin cables, con capacidad de diagnosticar posibles fallos sin necesidad de detener la maquinaria [2], incluso a distancia.

En la actualidad muchos sistemas que utilizan estas tecnologías no invasivas, no son on-line, de modo que suelen estar conectados a un ordenador o a una pantalla y no permiten la monitorización a distancia, implicando desplazamientos (y costes derivados de estos) que podrían ser evitados con una aplicación on-line. Debido a la situación causada por la pandemia de coronavirus SARS-CoV-2 ha quedado patente la importancia de poder reducir la presencialidad en la industria, de modo que es otro punto importante que aplicar a estos sistemas. Si además se tiene en cuenta que los parques eólicos se suelen situar en zonas de difícil acceso, ya sea por la gran cantidad de accidentes geográficos [3] o por tratarse de un parque off-shore, el coste y la dificultad del desplazamiento es todavía mayor.

Precisamente, una de las tecnologías que está revolucionando la conectividad en la industria es el IoT (Internet of Things). IoT se basa en la interconexión entre varios dispositivos a través de una red con el fin de que interaccionen e intercambien información sin necesidad de la intervención humana [4]. De hecho, en el campo industrial se utiliza principalmente para conectar dispositivos y sensores consiguiendo así analizar datos y proporcionar alertas o mensajes a los usuarios que deban realizar las acciones humanas o iniciar protocolos de actuación de forma autónoma en reacción a dichas alertas [4].

1.2. Fundamentos teóricos para la construcción del proyecto

Como se ha explicado anteriormente, en la industria el uso de motores eléctricos de inducción es crucial. Por este motivo, la posibilidad de que tras un fallo imprevisto en un motor se tenga que parar la producción para el mantenimiento, reparación o posible sustitución resulta en una reducción de la productividad [5], que puede llegar a ser un problema grave si en el transcurso de recambio o reparación del motor averiado se dispone de menos recursos de producción.

Tradicionalmente los sistemas de monitorización han requerido que el motor a analizar se encuentre apartado de la producción durante el diagnóstico. Con el fin de prevenir estas paradas técnicas o reducir su duración, es necesaria la introducción de un sistema de detección de fallos, con el cual poder planificar las paradas técnicas. La necesidad detectar fallos en los motores antes de que se produzca una avería que los inutilice, es lo que marca la importancia de desarrollar sistemas de monitorización en tiempo real. Esto marca el terreno para la nueva

generación de técnicas de monitorización: dar el salto a sistemas no invasivos y que permitan mantener el motor activo [2].

Durante los últimos años, algunas las magnitudes que se han ido utilizando para la monitorización de motores eléctricos de inducción son: monitorización de la **temperatura**, mediante la cual se estiman las pérdidas de potencia; monitorización del **flujo magnético**, que refleja las variaciones causadas por los armónicos de la corriente del estátor; monitorización de la **frecuencia de vibración**, pues se ha demostrado que cada tipo de fallo tiene asociado una frecuencia de vibración; monitorización de la **tensión del estátor**, que puede ser usado para medir la potencia y par instantáneos; monitorización de la **corriente del estátor**, medida con sensores Hall, se basa en el análisis en frecuencia de las componentes de la intensidad ya que cada tipología de error amplifica ciertos componentes del espectro de la señal; y etc. [5]

Varios estudios se han enfocado en señalar las técnicas no invasivas más utilizadas en la actualidad como, por ejemplo, la localización de cortocircuitos en las espiras del estator mediante el análisis de la temperatura, el diagnóstico de los rodamientos mediante el análisis de las frecuencias de vibración o la determinación de fallos en el rotor mediante el análisis del espectro de frecuencias de la corriente del estátor ([2], [5] y [6]). En concreto, se han especializado en caracterizar las principales averías de los motores de inducción con el fin de detectarlas a partir de tomar medidas de características comunes como vibraciones, tensión o intensidad.

Las averías más comúnmente detectadas en un motor de inducción se producen en los cojinetes (40%), pues el continuo funcionamiento de los rodamientos de los cojinetes puede producir fallos por fatiga, como por ejemplo vibraciones y elevados niveles de ruido. La corrosión o la falta de lubricación pueden agravar el proceso. El desequilibrio del eje causado por tensiones y corrientes parásitas en el eje (causadas a su vez por perturbaciones en el flujo del motor) o el exceso de temperatura de funcionamiento. El fallo en los cojinetes repercute en el rendimiento del motor, causando variaciones en el par mediante la aparición de armónicos en la corriente del estátor [2].

Por otro lado, los fallos del bobinado del estátor (los cuales suponen un 38%), estos se producen principalmente por fallos en el aislante, creando así derivaciones fase-tierra, fase-fase e incluso cortocircuitos entre bobinas de la misma o diferente fase. Una forma de detectar estos fallos es, una vez más, el análisis de la corriente, pues dado el malfuncionamiento del campo magnético del entrehierro, aparecen armónicos fácilmente detectables en la corriente del estátor [2].

Con menos frecuencia se encuentran fallos debidos al rotor (10%), no obstante, los primeros indicios de poder realizar diagnósticos de la salud de un motor mediante el análisis de la corriente del estátor se lograron tras analizar fallos como la rotura de barras del rotor. Este tipo de avería incrementa el valor de la corriente alrededor de un 50% [2] en la zona del rotor donde se ha producido el fallo, lo que se traduce en un campo magnético rotatorio en dirección contraria, frenando así la marcha del rotor e incrementando el deslizamiento inherente a los motores asíncronos [7].

Los distintos ensayos realizados por investigadores han dado como resultado que para ciertos fallos de motor hay ciertas frecuencias de corriente asociadas, las cuales indican que existe esa avería por su presencia en el espectro de la señal de la intensidad del estátor [6].

De los más evidentes, por ejemplo, una rotura de las barras del rotor o un cortocircuito entre espiras (si el rotor está bobinado) genera una intensidad a lo largo del tiempo definida por la siguiente ecuación:

Ecuación 1. Función teórica de una señal ideal de un motor eléctrico de inducción

$$i(t) = A \cdot \cos(\omega \cdot t) \cdot [1 + \beta \cdot \cos(2 \cdot s \cdot \omega \cdot t)]$$

Donde:

- A = Amplitud (A)
- ω = 2 · π · f
- f = Frecuencia (Hz)
- t = Tiempo (s)
- s = Deslizamiento
- β = Coeficiente de fallo

Esta ecuación produce una señal compuesta como se puede observar tras la evaluación de su procedimiento:

$$i(t) = A \cdot \cos(\omega \cdot t) + [A \cdot \beta \cdot \cos(\omega \cdot t) \cdot \cos(2 \cdot s \cdot \omega \cdot t)]$$

A partir de la expresión anterior, se distingue claramente la onda fundamental, $A \cdot \cos(\omega \cdot t)$, a la que se suma otro factor más, el cual se analiza a continuación:

$$A \cdot \beta \cdot \cos(\omega \cdot t) \cdot \cos(2 \cdot s \cdot \omega \cdot t) = \\ = A \cdot \beta \cdot \frac{1}{2} \cdot [\cos(\omega \cdot t + 2 \cdot s \cdot \omega \cdot t) + \cos(\omega \cdot t - 2 \cdot s \cdot \omega \cdot t)]$$

De modo que, este segundo factor incluye en su interior dos ondas más, completando la Ecuación 1 como una suma de tres ondas distintas:

Ecuación 2. Expresión completa de la corriente en un motor con rotura de barras.

$$i(t) = A \cdot \cos(\omega \cdot t) + \frac{1}{2} \cdot A \cdot \beta \cdot \cos[\omega \cdot t \cdot (1 - 2 \cdot s)] + \frac{1}{2} \cdot A \cdot \beta \cdot \cos[\omega \cdot t \cdot (1 + 2 \cdot s)]$$

De esta forma, la señal tendría tres componentes: por un lado, la onda fundamental, $A \cdot \cos(\omega \cdot t)$, por otro lado, las dos ondas producidas por el fallo, $½ \cdot A \cdot \beta \cdot \cos[\omega \cdot t \cdot (1 \pm 2 \cdot s)]$ como se puede observar en la Figura 2.

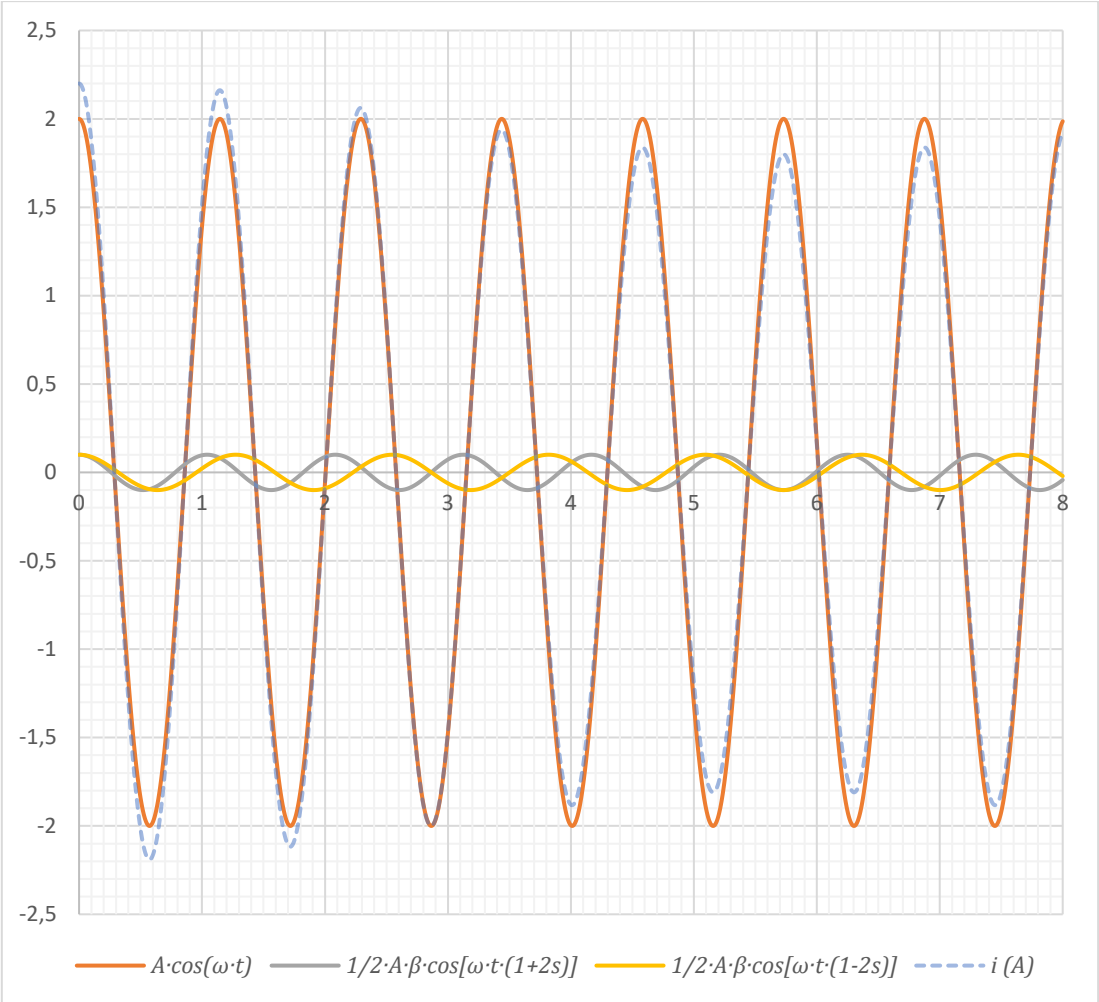


Figura 2. Desglose de las componentes de la intensidad en un motor con rotura de barras.

Como es de esperar, cada componente tiene su amplitud y su frecuencia, las cuales se pueden caracterizar tomando como base la expresión general de una onda de corriente:

Tabla 2. Caracterización de las componentes de la corriente en un motor con rotura de barras (siendo i_0 la onda fundamental e i_1 e i_2 las componentes de fallo).

Ecuación	Amplitud	Velocidad Angular	Frecuencia
$i(t) = A \cdot cos(\omega \cdot t)$	A	ω	$f = \frac{\omega}{2 \cdot \pi}$
$i_0(t) = A_0 \cdot cos(\omega_0 \cdot t)$	A_0	ω_0	$f_0 = \frac{\omega_0}{2 \cdot \pi}$
$i_1(t) = \frac{1}{2} \cdot A_0 \cdot \beta \cdot cos(\omega_0 \cdot t \cdot (1 - 2 \cdot s)$	$\frac{1}{2} \cdot A_0 \cdot \beta$	$\omega_1 = \omega_0 \cdot (1 - 2 \cdot s)$	$f_1 = \frac{\omega_1}{2 \cdot \pi}$
$i_2(t) = \frac{1}{2} \cdot A_0 \cdot \beta \cdot cos(\omega_0 \cdot t \cdot (1 + 2 \cdot s)$	$\frac{1}{2} \cdot A_0 \cdot \beta$	$\omega_2 = \omega_0 \cdot (1 + 2 \cdot s)$	$f_2 = \frac{\omega_2}{2 \cdot \pi}$

De lo que se puede extraer que:

- La amplitud de las corrientes de fallo se corresponde a la mitad de la amplitud de la onda fundamental multiplicada por el coeficiente β.

Ecuación 3. Amplitud de las ondas de fallo en un motor con rotura de barras.

$$A_b = \frac{1}{2} \cdot A_0 \cdot \beta$$

- La frecuencia de las corrientes de fallo corresponde al producto de la frecuencia fundamental y el factor (1 ± 2 · s).

Ecuación 4. Cálculo de las frecuencias de fallo.

$$f_1 = f_0 \cdot (1 - 2 \cdot s)$$
$$f_2 = f_0 \cdot (1 + 2 \cdot s)$$

Cabe destacar que para el análisis de estas ecuaciones solo se ha tenido en cuenta la frecuencia fundamental, ya que, aunque generalmente son despreciables, la corriente de un motor tiene armónicos en su aplicación real. Los armónicos encontrados en las frecuencias de fallo se ven claramente incrementados a causa del agravamiento de la avería, pero no dejan de encontrarse dentro del espectro de la corriente.

En resumen, estas averías son fácilmente detectables de una forma precoz mediante el análisis las componentes de la intensidad del estátor en contraposición a los actuales métodos de vigilancia del funcionamiento de los motores, los cuáles pueden implicar graves descensos en la producción industrial tras producirse la avería, pues suelen ser apartados del funcionamiento para ser analizados. Para introducir estas nuevas técnicas de monitorización on-line continua se propone el presente proyecto.

1.3. Objetivos

El objetivo del proyecto es construir un sistema de diagnóstico remoto de motores de inducción cuyo fin es poder detectar posibles averías y malfuncionamientos a través de una medición y procesamiento periódicos de la intensidad tratando que los datos puedan estar almacenados en la nube, estando así disponibles en todo momento y lugar.

En primer lugar, se deberá poder tomar medidas tanto de la corriente del estátor como de la velocidad angular del rotor. Para ello será necesaria la selección de sensores para tal cometido. Estas mediciones deberán ser previamente procesadas, luego, nuevamente, se deberá realizar una selección de sistemas para poder procesar las medidas anteriormente realizadas. Este sistema deberá tener capacidad de realizar cálculos complejos como la Transformada Rápida de Fourier (en adelante, FFT) y deberá ser compatible con los demás estándares que se utilicen en este proyecto, con mayor énfasis en la compatibilidad con los sensores escogidos para la medición tanto de la corriente del estátor como de la velocidad de giro del motor.

En tanto que el procesamiento de las medidas requiere la programación del sistema de procesamiento, será necesaria la realización de una selección de lenguajes y entornos de programación, con el fin de agilizar y optimizar el procesamiento de las mediciones. Este lenguaje (o lenguajes) deberá permitir la realización de la FFT sobre un tramo de la corriente medida. Tras ello, el sistema deberá ser capaz de detectar las frecuencias de fallo y, en consecuencia, comparar el nivel de los armónicos asociados a ellas con los valores considerados como fallo para poder determinar la existencia de una avería y la naturaleza de esta.

Una vez procesadas las medidas y localizados los posibles fallos, deberá poder almacenarse la información relevante para poder crear un histórico que muestre la evolución del comportamiento del motor analizado. Para esta función se deberá seleccionar un sistema de

almacenamiento de datos que permita la extracción de estos de forma ágil y sencilla, pues posteriormente se pondrán a disposición del usuario.

Finalmente, para la entrega de todos los datos procesados y almacenados, se deberá seleccionar un canal y sistema de comunicación, de tal forma que el usuario sea capaz de evaluar de forma remota las medidas actuales y pasadas (que se habrían almacenado como se establece en el anterior párrafo) desde cualquier zona del mundo, así como los resultados de los análisis de las frecuencias de fallo, de la manera más clara y evidente posible. Así, de un vistazo rápido, un operario podrá identificar la salud del motor. Este canal deberá ser accesible desde la mayor gama de dispositivos posible, pues lo que se busca en este apartado es la monitorización remota de las instalaciones, sobre todo de aquellas poco accesibles.

En resumen, se trabajará para la consecución de un sistema tan sencillo como sea posible que permita al personal identificar fallos (mediante el método teórico explicado en 1.2 Fundamentos teóricos para la construcción del proyecto) de una forma rápida y evidente desde cualquier zona del mundo y desde cualquier dispositivo compatible.

1.4. Requisitos y especificaciones

Para definir este proyecto se deberá trabajar en base a unos requisitos y especificaciones que se presentan a continuación.

1.4.1. Sensores

En cuanto al sensor de intensidad, los esfuerzos se centrarán en conseguir realizar mediciones de intensidad en el rango entre -6 A y 6 A, pues el motor del que se dispone en el Departamento de Ingeniería Eléctrica de la Universitat Politècnica de València se encuentra en ese rango. Por otro lado, la frecuencia de muestreo planeada será de 1 kHz, es decir, 1000 tomas de medida por segundo, de modo que el ancho de banda, por seguridad, será de al menos el doble, es decir 2 kHz.

Por otro lado, el sensor de velocidad o encoder deberá poder trabajar en el rango entre 0 y 3000 rpm por lo menos, pues este es el valor máximo de la velocidad de sincronismo alcanzada por un motor de inducción alimentado a 50 Hz, es decir, la frecuencia comúnmente utilizada en España y la UE.

1.4.2. Sistema de Control

Se debe considerar que el sistema de control sea autónomo, de forma que no dependa de otros dispositivos para funcionar y así se aligere y simplifique la construcción del sistema completo.

El sistema de control deberá contar fundamentalmente con al menos una entrada analógica conectada a un canal ADC para el sensor de intensidad y una entrada digital para el encoder así como de un sistema de temporización para que se ejecute la medición rutinariamente según la frecuencia de muestreo de 1 kHz anteriormente mencionada. Para que la temporización sea efectiva, el sistema deberá disponer de un mecanismo de interrupciones por software.

En cuanto a la comunicación, el sistema de control deberá ser compatible con al menos un protocolo estándar de comunicación de forma que sea posible el envío de información entre una gama lo más amplia posible de dispositivos. Para que la comunicación online sea posible, el sistema de control deberá ser capaz de conectarse a una red wifi protegida mediante el protocolo WPA2.

Se valorará positivamente que el sistema sea de bajo coste y de código abierto.

1.4.3. Comunicación

Se requiere de un protocolo de comunicación estándar para garantizar la compatibilidad entre el mayor número de dispositivos posible. Este estándar deberá ser lo más rápido y sencillo posible y los mensajes deberán tener el menor tamaño que sea posible.

1.4.4. Sistema de Monitorización

El sistema de monitorización debe consistir en una pantalla en la que encontrar fácilmente los datos proporcionados por el sistema de control, siendo estos la intensidad media, el análisis de los armónicos y el estado del motor la información fundamental que se deben mostrar. Puesto que esta información se puede comprender mejor a través de gráficos y representaciones de señales, será menester que este sistema de monitorización pueda realizar y mostrar dichas representaciones gráficas.

Se debe considerar la posibilidad de modificar las funciones básicas del sistema de control desde la pantalla proporcionada por el sistema de monitorización, por lo tanto, este deberá permitir el envío de configuración al sistema de control de la misma forma que se reciben las medidas desde este. Obviamente, para este cometido, el sistema de monitorización deberá ser capaz de conectarse a una red wifi protegida mediante el protocolo WPA2.

1.4.5. Almacenamiento de datos

El mayor requerimiento respecto al almacenamiento de datos es que este se produzca de una forma ágil y sin causar retardos en la ejecución de los demás pasos de la aplicación. Se requerirá, además, que tenga capacidad para almacenar los valores de las frecuencias sospechosas de causar fallos de los 30 días anteriores con motivo de poder realizar un estudio histórico del progreso de dichas frecuencias.

Será conveniente garantizar compatibilidad con todos los apartados anteriores, así como con los sistemas operativos más utilizados para que su contenido pueda ser visualizado por cualquier usuario.

1.5. Alternativas propuestas

Una vez establecidos los límites del proyecto, se valorarán diferentes maneras de llevarlo a cabo siguiendo los requerimientos indicados en el apartado anterior.

1.5.1. Sensores

Transformador de corriente

Estos componentes convierten una alta intensidad en una de menor valor (por convención se suelen aplicar relaciones de transformación X:5 o X:1), pues los transformadores de corriente se suelen utilizar en casos que precisen medir estando presente una tensión o intensidad muy elevada, aunque se puede regular la relación de transformación mediante el enrollamiento del conductor primario alrededor del núcleo. [8]

Cabe destacar que, en caso de que se abra el circuito secundario mientras en el primario sigue circulando corriente, el secundario tenderá a ofrecer una corriente de valor infinito, lo cual implica hacer peligrar la seguridad la instalación.

Sensor de efecto Hall

Otra opción que se propone es el uso de un sensor de efecto Hall, pues se trata de un componente sencillo y de tamaño reducido, a lo que acompaña un valor económico reducido. Funciona como un transductor de salida analógica, lo cual significa que convierte la intensidad medida en un valor dentro de un rango de la tensión de salida.

1.5.2. Sistema de Control

PC industrial

Por último, se contempla el uso de un PC industrial para realizar el proyecto. En este caso, el sistema de adquisición de señales se realizará a través de una tarjeta de adquisición de propósito general.

Dependiendo de la potencia de este equipo, se utilizará también como servidor o se utilizará uno externo.

A favor: suelen disponer de gran potencia de procesamiento y gran variedad de puertos para así poseer una buena compatibilidad con varios estándares.

En contra: ocupa mucho espacio, la complejidad de sus componentes dificulta el mantenimiento, los costes son muy elevados y, como en un PC habitual, su fragilidad mecánica y técnica es un punto débil a tener en cuenta.

Autómata programable (PLC)

Una de las opciones propuestas es la utilización de un autómata programable para este proyecto. En esta situación, se programaría un sistema de procesado tras la entrada analógica de la intensidad. En adición a esto, se programaría una pantalla de visualización con controles para monitorizar la intensidad medida, la frecuencia estimada y la forma de onda.

Esta pantalla o una similar deberá estar disponible online para su acceso remoto.

A favor: los lenguajes de programación (diagrama de contactores y BDF) son prácticamente estándares en la programación de autómatas; se puede acoplar una pantalla.

En contra: la programación solo se puede llevar a cabo mediante programadores de autómatas; el sistema habitualmente cerrado y no suele haber compatibilidad entre marcas; tienen un coste elevado.

Microcontrolador

Otra alternativa propuesta es la utilización de un microcontrolador para ser utilizado a modo de sensor de intensidad a partir de la instalación de sensores de intensidad eléctrica sobre su ADC, y, en caso de ser necesario, la introducción de un divisor de tensión intermedio para ajustar la tensión al rango de funcionamiento del ADC.

A favor: se programa mediante el lenguaje de programación C, el cual es muy extendido y similar a prácticamente todos los lenguajes actuales; ocupa poco espacio; los costes de mantenimiento y sustitución son muy bajos.

En contra: puede necesitar periféricos externos, la programación es a muy bajo nivel.

1.5.3. Comunicación

HTTP

El protocolo HTTP (Protocolo de Transmisión de Hipertexto) se utiliza para intercambiar datos en la web. Se trata de un sistema de peticiones de tipo Cliente-Servidor, lo que significa que para que se produzca el intercambio de información, el cliente debe crear una petición para iniciar la conexión. [9]

Se trata de un protocolo ampliable [9], lo que le ha permitido evolucionar añadiendo funciones a lo largo del tiempo y actualizarse a las nuevas tecnologías.

MQTT

El protocolo de comunicación MQTT es el más utilizado en aplicaciones IoT y tiene el aliciente de ser de código abierto, lo cual abre la posibilidad de que exista una gran documentación por parte de la comunidad.

El sistema MQTT es muy simple: Cada envío consiste en un tema y un mensaje. Los periféricos envían datos a un servidor y los que van a recibir la información se suscriben a los temas que sean necesarios, pues para que el receptor pueda leer el mensaje, es necesario que anteriormente este se haya suscrito al tema al que pertenece.

1.5.4. Sistema de Monitorización

Pantalla

Se estudia la instalación y configuración de una pantalla de monitorización para mostrar todos los resultados de la actuación del sistema utilizado para adquirir datos desde el motor.

Aplicación móvil

Una alternativa para realizar el seguimiento de los motores consiste en el desarrollo de una aplicación móvil compatible con tabletas y smartphones. En este caso, la monitorización podría ser realizada a distancia, pues cada usuario tendría la posibilidad de conectarse al sistema desde el lugar que se encuentre e incluso recibir notificaciones push en caso de avería.

Aplicación web

Al realizar una aplicación web se garantiza la compatibilidad con todos los dispositivos que dispongan de un navegador web, es decir, la inmensa mayoría de estos. Puesto que estas aplicaciones web son fácilmente encapsulables en aplicaciones móviles, se dispone de todas las ventajas de una aplicación móvil, incluyendo la posibilidad de recepción de notificaciones push y, obviamente, la posibilidad de ser utilizada en una pantalla fija en la instalación.

1.5.5. Almacenamiento de datos

Base de datos SQLite

Se utilizaría una base de datos SQL mediante el sistema SQLite, pues es un sistema de código abierto. Crea un fichero local para almacenar la información y, para interactuar con este, se utilizan funciones y librerías logrando así una integración con el código del programa sin necesitar un motor externo de funcionamiento.

Fichero JSON

Se trata de un formato ampliamente utilizado para almacenar datos en el desarrollo de software. Pese a basarse en texto, el formato JSON utiliza una estructura de campos y valores, como en una base de datos convencional.

Fichero de texto CSV

Un fichero CSV consiste en una sucesión de datos tabulados separados mediante líneas y comas, lo cual lo convierte en la opción más simple.

1.6. Descripción de la alternativa seleccionada

Una vez determinados los requerimientos básicos del proyecto y analizadas las diferentes alternativas para todos los componentes, es el momento de realizar una selección para realizar la construcción del proyecto.

1.6.1. Sensores

Puesto que no se espera tener que medir una corriente mayor de 6 A, se ha decidido por la inclusión de un sensor de efecto hall para realizar las mediciones, pues un transformador de corriente aumentaría el tamaño de la instalación. Otro punto a favor del sensor de efecto Hall es que la señal de salida viene determinada como un rango dentro de la tensión de suministro del sensor, lo cual es fácilmente captable e interpretable por cualquier ADC.

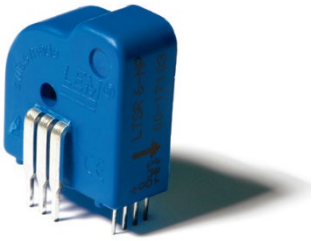


Figura 3. Sensor de efecto Hall LEM LTSR 6-NP [10]

El modelo elegido es el LTSR 6-NP del fabricante LEM USA Inc. Este puede medir entre -6 A y 6 A.

El voltaje de salida depende del valor medido y se puede interpretar mediante la siguiente expresión [10]:

Ecuación 5. Función de transferencia del sensor de intensidad.

$$V_{out} = 2{,}5 \pm \left(0{,}625 \cdot \frac{I_P}{I_{PN}}\right) [V]$$

Donde I_P es la intensidad del circuito primario, es decir, la que se va a medir, e I_{PM} es la intensidad nominal del primario, el cual, según el documento de características es 6 A. La incertidumbre en las mediciones se estima en un ± 0,2% a una temperatura de 25°C. Cabe destacar que cuando no exista corriente en el circuito primario, el secundario proporcionará siempre 2,5 A de base.

En cuanto al sensor de velocidad, el motor que será analizado ya dispone de un encoder integrado que envía **720 pulsos por vuelta** (es decir, que durante una vuelta el encoder emitirá 720 pulsos), por lo cual no se realizará ninguna selección en este sentido.

1.6.2. Sistema de Control

Tras analizar todas las alternativas candidatas para la implementación del sistema de control, se ha seleccionado la opción correspondiente al uso de un microcontrolador debido a su bajo coste de adquisición. Teniendo en cuenta que se trata de un sistema de código libre, las herramientas para su configuración son gratuitas y además se dispone del soporte de la comunidad. Además, su reducido tamaño permite una mayor versatilidad a la hora de su disposición en la instalación a analizar.

Para la preparación del código y la creación de un prototipo se ha utilizado un kit de desarrollo tipo Node-MCU v3 que contiene un procesador ESP8266. No obstante, a la hora de construir el proyecto final se ha utilizado una placa Node-MCU v3 con un procesador ESP32, el cual tiene dos núcleos, lo que permite realizar varias acciones en paralelo.

Para la programación de las placas se utilizará el IDE Arduino 1.8.12 en lenguaje Arduino, basado en su mayoría en el lenguaje C y C++ con las librerías correspondientes al procesador elegido.

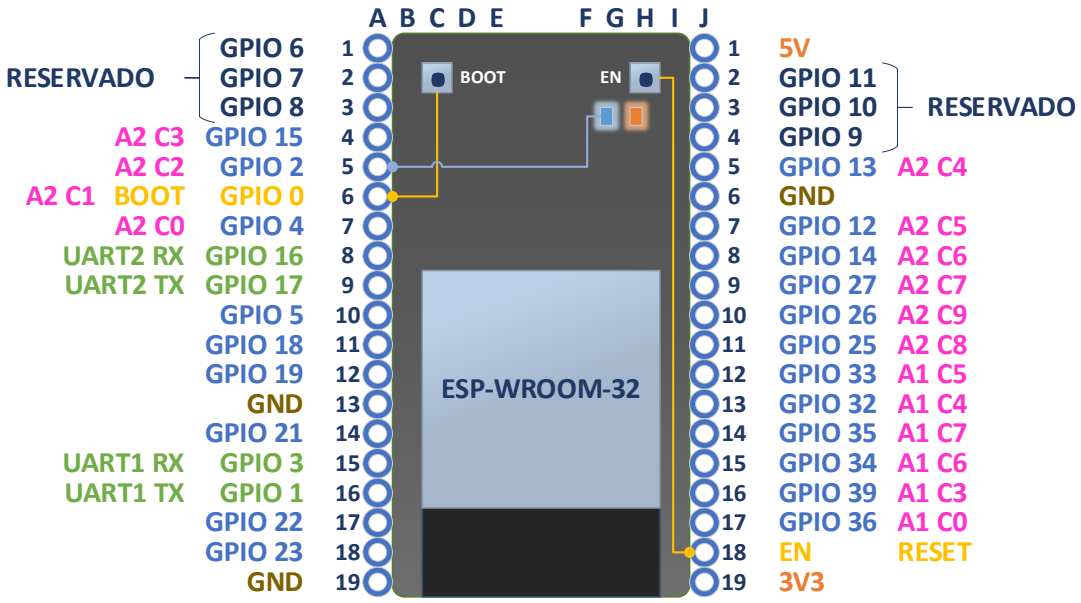


Figura 4. Disposición de los pines en un NodeMCU con ESP32

Como se puede observar en la Figura 4, el microcontrolador dispone de dos ADC, uno con 6 canales y otro con 10. Ambos son de 12 bits, de manera que el resultado que arrojarán tras recibir una medida será un número entero entre 0 y 4095 (2^{12} valores) con una resolución de 0,7326 mV según las explicaciones adjuntas en el Anexo 1.

Cabe decir que para transmitir el programa creado al ESP32 es necesario hacer que entre en modo descarga manteniendo pulsado el botón Boot mientras dure la conexión, de lo contrario se produce un error. Este hecho no ocurre en el modelo ESP8266 ya que entra en modo descarga automáticamente.

Para solucionar el error de conexión a la hora de transmitir el programa, se ha insertado el microprocesador en una placa de prototipos y se ha conectado un condensador de 10 µF entre los pines EN y GND [11] tal como se puede observar en la Figura 8. Esquema de montaje de la solución.

Según la documentación oficial del microprocesador ESP32, los pines destinados al uso del ADC 2 se comparten con la funcionalidad Wifi, de modo que, si este está activo, el ADC 2 no se podrá utilizar. Puesto que una de las utilidades más destacadas de este proyecto es la conexión a internet de forma inalámbrica, el uso de este ADC ha sido descartado, y únicamente se ha utilizado el ADC 1.

Figura 5. Generador de ondas Analog Discovery [12].

Durante los primeros meses del proyecto se utilizó un generador de ondas portátil controlado por USB modelo Analog Discovery de Digilent. No obstante, tras el confinamiento decretado por el Gobierno de España tras la entrada en vigor del Estado de Alarma a causa de la pandemia de SARS-CoV-2 se dejó de tener acceso a él.

1.6.3. Comunicación

Pese a que el ESP32 elegido anteriormente es compatible tanto con el protocolo HTTP y el MQTT, se ha decidido utilizar el MQTT debido a que el tamaño del mensaje de este es inferior debido a la diferencia de tamaño de las cabeceras. Otro punto a favor del MQTT es la diferencia en la velocidad, llegando en algunos casos a ser 20 más rápido y un 22% más eficiente en cuanto al consumo de energía [13].

MQTT se caracteriza por ser un protocolo basado en Publicación-Subscripción, un sistema de comunicación full-duplex, es decir, que existe intercambio de información en ambos sentidos. Mediante un bróker (una especie de dispositivo intermediario) se ponen a disposición de todos los dispositivos conectados a una red MQTT todos los mensajes emitidos en todos los temas. Se conoce como tema a una palabra clave que identifica un canal de comunicación por el cual se va a realizar un envío periódico de información. Por tanto, para recibir información sobre un tema un dispositivo deberá realizar la suscripción sobre dicho tema, para que este pueda recoger el mensaje desde el bróker.

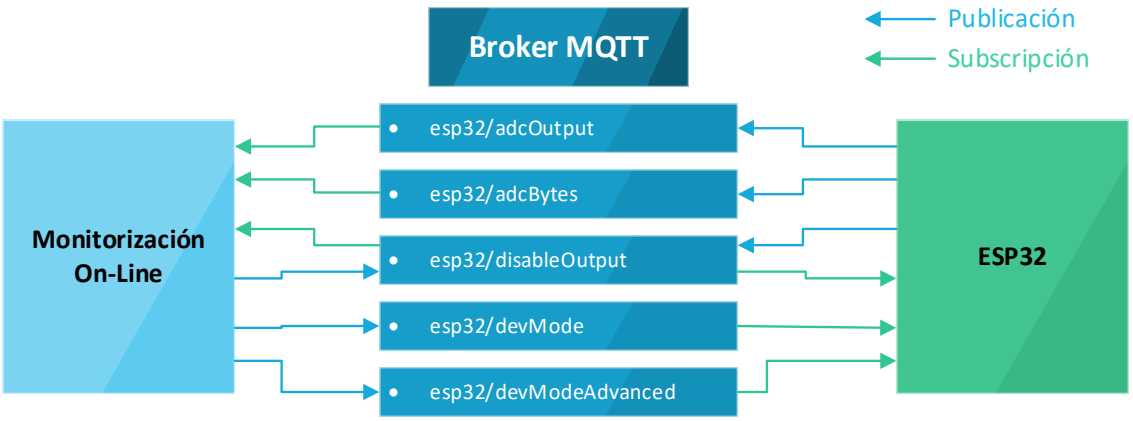


Figura 6. Esquema de transmisión de mensajes mediante MQTT

En la Figura 6 se muestran la mayoría de los temas que se van a ver involucrados en el proyecto y que se explicarán más adelante.

1.6.4. Sistema de Monitorización

En cuanto al sistema de monitorización se ha decantado por la construcción de una aplicación web frente a las demás alternativas porque esta engloba todas las características de las demás.

El proyecto requiere de un servidor con un software compatible con el protocolo MQTT, pero, ante el inconveniente de no disponer de uno a la hora de realizar el desarrollo, la alternativa de bajo precio a un servidor web dedicado es un PC de uso normal. Por lo tanto, para crear un servidor con MQTT, se ha optado por configurar una Raspberry Pi 4 Model B.

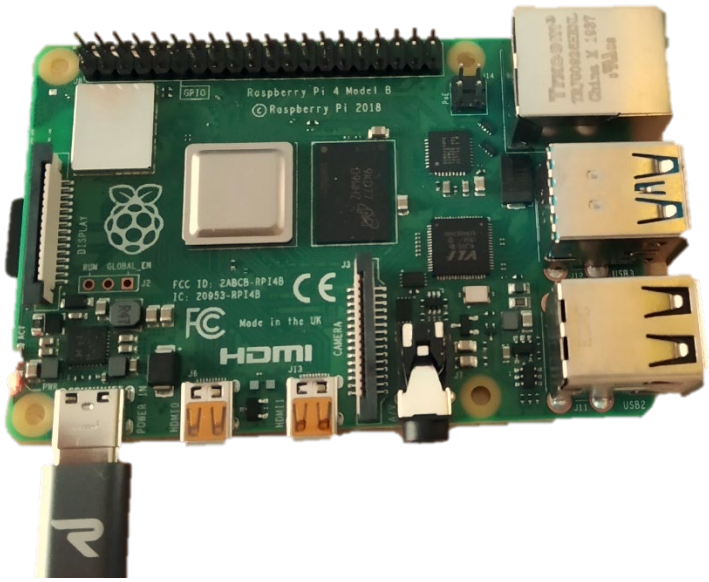


Figura 7. Raspberry Pi alimentada mediante un PC a través de un cable USB tipo C.

Raspberry Pi es un micro PC elaborado con elementos de código abierto y con los componentes básicos (véase la Figura 7), con el fin principal de ser destinado a la educación en países en vías de desarrollo a un módico precio. No obstante, la comunidad de desarrolladores le ha dado numerosos usos en sus proyectos caseros o incluso en sistemas más ambiciosos. Posteriormente se introducirá en una caja para mayor protección, y puesto que esta incorporará un ventilador, mayor longevidad del procesador.

Para diseñar la aplicación web de procesado de datos recibidos desde el ESP32 y monitorización se utilizará Node-RED, un entorno open-source [14] de programación con bloques comúnmente utilizado en proyectos IoT (ver **Capítulo 4. Desarrollo de la aplicación IoT**), que se instalará en la Raspberry Pi 4 Model B.

La Raspberry Pi 4 Model B también funcionará como Bróker MQTT mediante la instalación del software *mosquitto* [15]. De esta forma, toda la parte del proyecto exterior al ESP32 queda localizada en un mismo sitio, la Raspberry Pi.

1.6.5. Almacenamiento de datos

En este proyecto, el almacenamiento de datos se realizará a través de una base de datos en la que se escribirán los valores de las frecuencias de fallo correspondientes a una rotura de barras en el rotor periódicamente con el fin de realizar un análisis histórico de la progresión de los armónicos causada por la avería.

Con el fin de que este almacenamiento de datos suceda con la mayor fluidez posible, el sistema elegido es la escritura y lectura de un fichero JSON, pues se trata de una base de datos estructurada, pero en formato texto, de modo que no hace falta un software o una librería externos para trabajar con ella.En este sentido, cabe destacar que anteriormente se trató de trabajar con una base de datos SQLite, pero por problemas con el rendimiento del sistema se decidió cambiar el sistema de almacenado (véase 3.2. Uso de SQLite para almacenar valores en Node-RED).

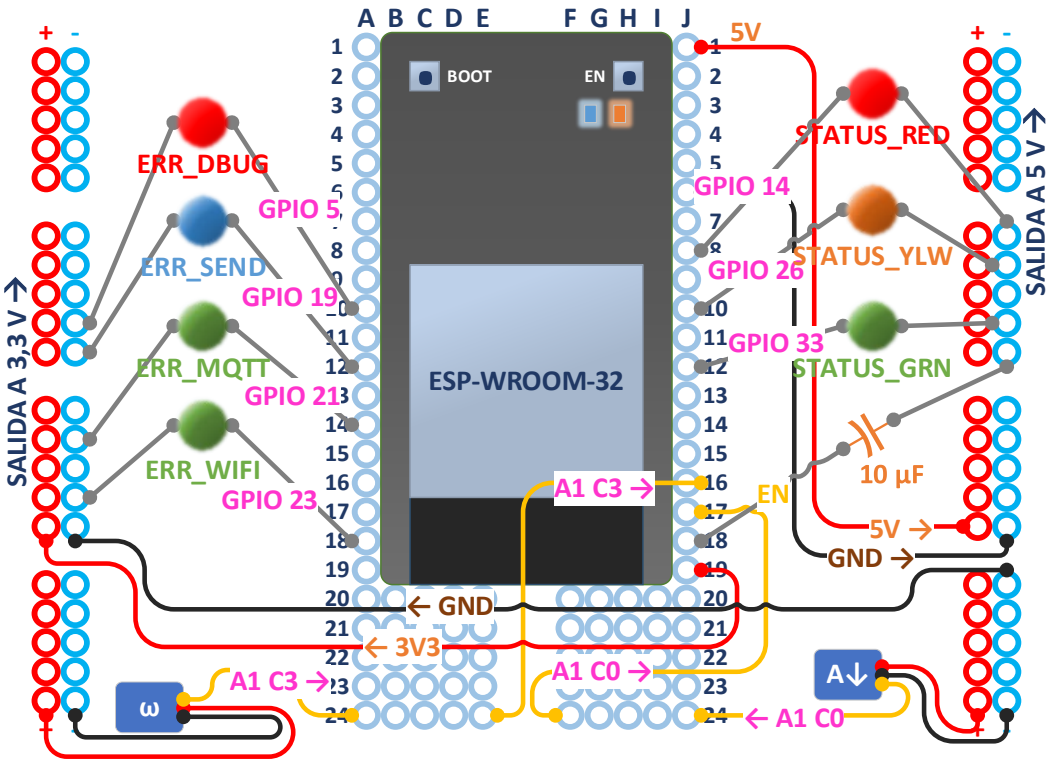


Figura 8. Esquema de montaje de la solución en una placa de prototipos.

1.6.6. Resumen del proyecto

Finalmente, la opción que ha resultado más económica y adecuada a los **Requisitos y especificaciones** es la compuesta por un microcontrolador Node-MCU v3, al que se conectará

un sensor hall para medir la intensidad del estator y un encoder para medir la velocidad del motor y una serie de LEDs para monitorizar el estado del sistema. El montaje de este sistema puede observarse en la **Figura 8**.

Una Raspberry Pi servirá como servidor IoT que contenga la aplicación que permitirá analizar las medidas tomadas por el Node-MCU v3 y la aplicación web que servirá para suministrar los datos de las mediciones y el análisis anterior.

La comunicación entre ambos elementos anteriores se llevará a cabo mediante el protocolo MQTT, un sistema liviano basado en Publicación-Suscripción utilizado en varias soluciones IoT del mercado.

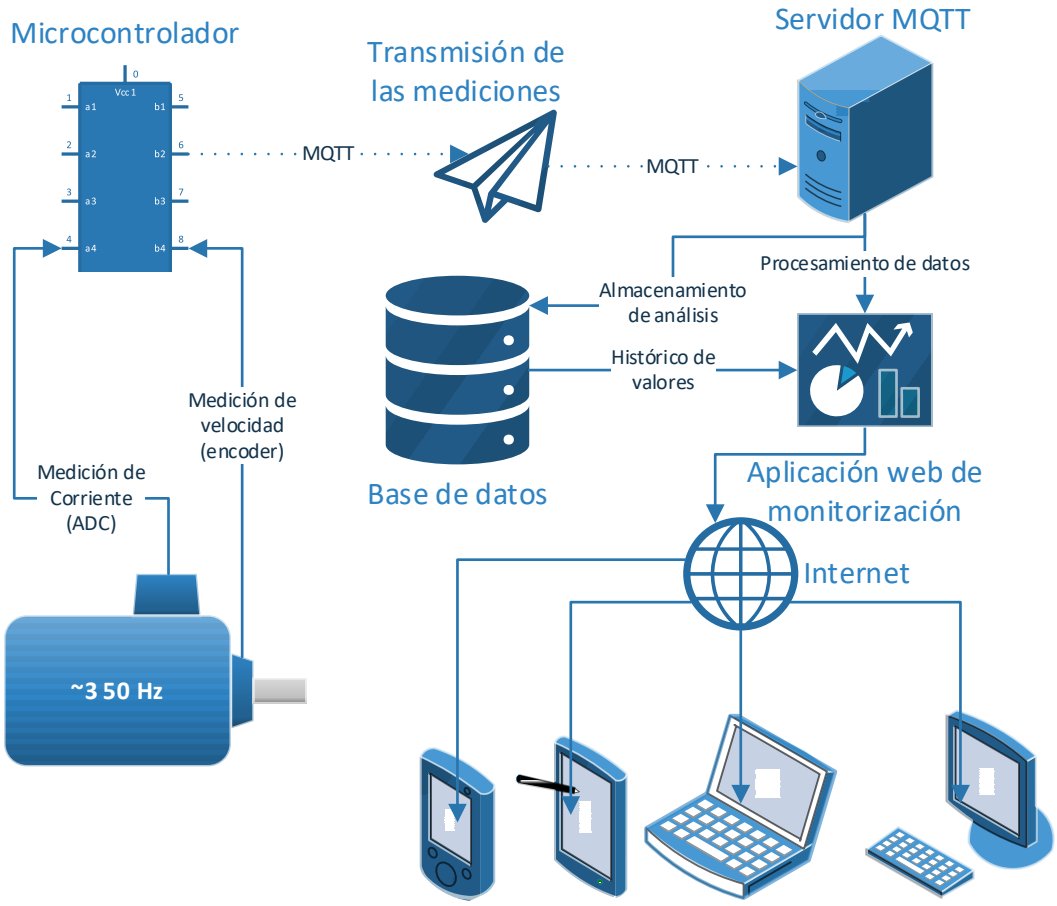


Figura 9. Diagrama general de la solución seleccionada.

Capítulo 2. Programación del microprocesador

A la hora de comenzar a trabajar con el microprocesador ESP32 ha sido conveniente instalar sus librerías, como anteriormente se hizo con el ESP8266. El procedimiento se ha encontrado en internet la página web de Prometec [16].

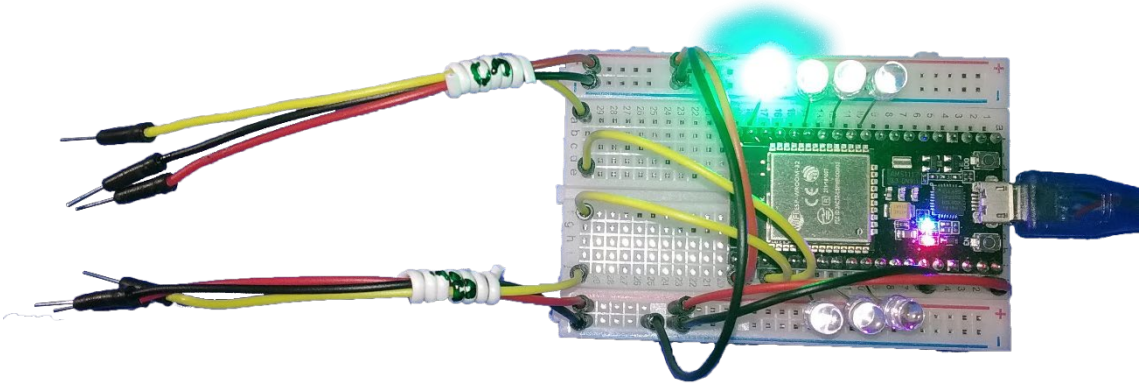


Figura 10. Montaje del ESP32 (sin condensador de 10 µF)

Este programa está enfocado al uso del ADC (Conversor del Analógico al Digital), pues, mediante este, debe recoger las medidas de la intensidad del estátor y almacenarlas en un buffer (conjunto de valores almacenado de forma temporal) para su posterior análisis según la Transformada Rápida de Fourier.

Por supuesto, otras funciones fundamentales son:

- La conexión a una red Wifi para permitir el envío de las medidas al servidor y, posteriormente, a la página web de monitorización.
- Generación de señales de prueba en caso de no disponer de un motor real.
- Envío y recepción de mensajes con medidas y configuraciones.

2.1. Estructura modular del programa

Puesto que se han implementado varias funciones relacionadas a diferentes partes del hardware del ESP32, se ha decidido establecer una estructura modular en el programa. Esto significa que se ha creado un fichero `.ino` para cada elemento de hardware de la siguiente forma:

- **main.ino:** en este fichero se han implementado las funciones `setup()` y `loop()`, tal como se explica en **Anexo 2. Introducción a Arduino.**
- **adc.ino:** en este fichero se incluyen las funciones para tomar e interpretar las mediciones del ADC.
- **debug_signal.ino:** contiene las funciones necesarias para generar valores de ondas senoidales sintéticas.
- **functions.ino:** está compuesto de los bucles que se ejecutarán con el protocolo MQTT, en los que se envían las mediciones.
- **led.ino:** este fichero contiene las funciones necesarias para el control del LED, tanto el incorporado como los externos.
- **mqtt.ino:** Contiene las funciones que inicializan y configuran la conexión MQTT, así como los temas suscritos.

- **mqtt_callback.ino:** está compuesto por la función que se ejecuta cada vez que un mensaje de un tema subscrito es publicado en el servidor MQTT.
- **wifi_connection.ino:** las funciones contenidas en este fichero permiten la conexión a una red Wi-Fi.

Los ficheros con código tienen, además, un fichero de cabecera cada uno con el fin de definir ciertas funciones como globales para que así puedan ser utilizadas por el resto de funciones de otros ficheros `.ino`.

Para evitar redundancias, estos ficheros de cabecera incorporan directivas de preprocesamiento que impiden ser incluidos más de una vez:

```
#ifndef _CONFIG_H
#define _CONFIG_H

// (…)

#endif
```

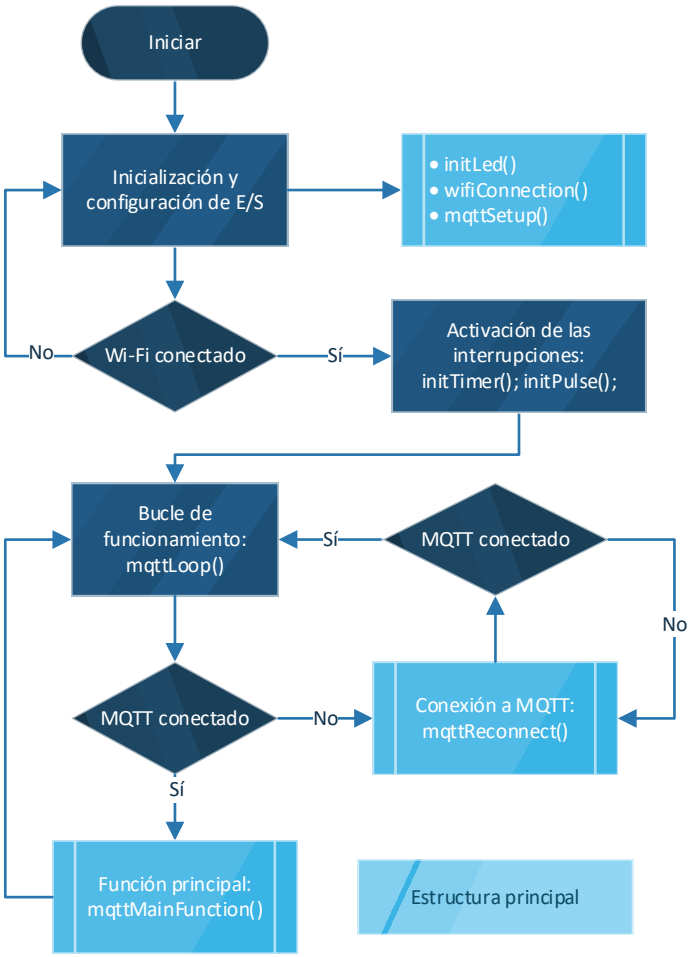


Figura 11. Diagrama de flujo general del programa implementado en el ESP32.

2.2. Inicialización y configuración

Como paso inicial, antes de empezar con el bucle de funcionamiento, la aplicación debe inicializar cada componente que se va a utilizar y configurar los eventos que causarán una

interrupción en el programa principal para ejecutar una medición, tanto de corriente como de velocidad angular (véase la Figura 11). La inicialización es imprescindible para indicar al microcontrolador qué modo de funcionamiento va a tener cada periférico.

Para modificar la configuración de forma fácil y rápida, se ha implementado un archivo de cabecera llamado `config.h`, el cual tiene varias definiciones de constantes correspondientes a valores de configuración de los distintos ficheros `.ino` que se usan en el proyecto. Este aspecto del proyecto permite que el programa diseñado pueda aplicarse en casos donde las especificaciones sean distintas del actual, facilitando la adaptación a situaciones diversas.

2.2.1. Inicialización de la comunicación vía puerto serie

El primer elemento a inicializar es la comunicación vía puerto serie, para lo cual se ha utilizado la función heredada `Serial.begin()`. Esta función recibe el *baudrate* (frecuencia de comunicación con la terminal) necesario para realizar la conexión. Para esta aplicación se ha escogido la velocidad de 115200 Bd, valor que se ha asignado a la variable `BAUDRATE` en el archivo de cabecera `config.h`.

Una vez activa la comunicación vía puerto serie, el programa es capaz de mostrar información en la pantalla del equipo de desarrollo gracias a la función *Monitor Serie* incluida en el Arduino IDE.

Los primeros datos que aparecerán son los correspondientes a la velocidad del procesador, extraídos mediante la función `getCpuFrequencyMhz()`, incluida en la librería propia del `ESP32 esp32-hal-cpu.h` y mostrada en la consola con la función `Serial.print()`.

2.2.2. Inicialización de los LED

A continuación, se inicializarán los LED mediante la función `initLed()`, la cual se ha implementado en el fichero `led.ino`. Esta función recibe como argumento el puerto a configurar como salida digital mediante la función heredada `pinMode` de manera análoga a como se ha explicado anteriormente en el apartado Funcionamiento del indicador LED del Anexo 2. En este caso, a diferencia del ESP8266, la lógica del LED no es invertida.

Para poder realizar un seguimiento del estado de la inicialización, se ha incorporado, mediante la función heredada Serial.print(), un mensaje de consola con el siguiente contenido:

```
Inicializado pin X como salida.
```

Donde X será sustituido por el puerto configurado. Acto seguido, el LED se activará durante un segundo con el fin de comprobar el buen funcionamiento de este mediante la función `ledSet()` y finalmente se apagará mediante la función `ledReset()`. Ambas reciben el puerto sobre el que van a realizar la acción como argumento.

Se realiza esta acción con todos los LED incluidos en el proyecto, que están identificados con los siguientes alias en el fichero de cabecera `esp32pinout.h`:

```
#define LED_BUILTIN 2  // LED interno

// (…)

#define LED_ERR_WIFI   23   // Encendido al conectar a red wifi
#define LED_ERR_MQTT   21   // Encendido al conectar a broker mqtt
#define LED_ERR_SEND   19   // Parpadea durante envío de medidas
```

```
#define LED_ERR_DBUG   5    // Encendido durante modo desarrollo

#define LED_STATUS_GRN   14 // LED verde del semáforo
#define LED_STATUS_YLW   26 // LED amarillo del semáforo
#define LED_STATUS_RED   33 // LED rojo del semáforo
```

2.2.3. Conexión a una señal Wifi

El siguiente paso en la inicialización del sistema es configurar la conexión Wifi. Para ello, se ha implementado la función `wifiConnection()` en el fichero `wifi_connection.ino`, la cual funciona exactamente igual que la función detallada en el apartado Establecimiento de la conexión a una red Wifi en el Anexo 2, salvo por una diferencia: en este caso, las definiciones del SSID y la contraseña se han realizado en el fichero de cabecera `config.h`.

```
#define USE_WIFI        true
#define WIFISSID        "SSID_WIFI"
#define WIFIPASSWORD    "PASSWORD"
```

Tras acabar con éxito la conexión a una red Wifi se encenderá `LED_ERR_WIFI` para dar conocimiento de ello al operador.

2.2.4. Configuración de la conexión MQTT

Para poder establecer la conexión MQTT más adelante, es necesario que previamente se especifique la IP y el puerto del servidor MQTT. Se ha implementado la función `mqttSetup()`, en el fichero `mqtt.ino`, que se encarga de esta tarea. Para ello se utiliza la función `client.setServer()`, integrada en la librería `PubSubClient.h`, que recibe como argumentos la IP y el puerto del servidor MQTT. Ambos datos están definidos en el fichero de cabecera `config.h`:

```
#define SERVER_IP       "192.168.1.X"
#define SERVER_PORT     1883
```

Finalmente, mediante la función `client.setCallback()` se ha especificado la función que se ejecutará tras recibir un mensaje dentro de un tema al que previamente se haya suscrito.

2.2.5. Configuración las interrupciones

Interrupción por temporizador

El cometido del temporizador es provocar la ejecución de una función cada cierto tiempo, es por esto que es de vital importancia su inclusión en este proyecto.

La inicialización del temporizador se lleva a cargo mediante la función `initTimer()`, la cual se encuentra en el fichero `isr.ino`, y recibe como argumento el número de temporizador a configurar.

En primer lugar, se ha definido un objeto de temporizador al cual se le ha llamado `timer`. Este objeto recibe su valor mediante la función `timerBegin()` incluida en la librería base del ESP32.

La función `timerBegin()` recibe como argumentos el número de temporizador (entre 0 y 3), la preescala del temporizador y un valor booleano para indicar si el temporizador va a funcionar contando hacia atrás (`true`) o hacia adelante (`false`).

La preescala del temporizador funciona como divisor de frecuencia, lo cual implica que la frecuencia base (80 MHz) será dividida entre la preescala y se obtendrá una frecuencia de reloj adaptable según el valor de la preescala.

En este proyecto se ha utilizado 80 como preescala para poder realizar mediciones con un periodo entre muestras de 1 ms.

A continuación, se ha asignado a este temporizador una interrupción, es decir, una función que se ejecutará cada vez que el temporizador salte, mediante la función `timerAttachInterrupt()`, la cual recibe como argumentos el objeto temporizador `timer` creado anteriormente, un puntero a la función `onTimer()` que ejecutará tras la interrupción, y un valor booleano para indicar si la interrupción la va a llevar a cabo un cambio de flanco (`true`) o un cambio de nivel (`false`).

El siguiente paso es configurar el disparo del temporizador con la función `timerAlarmWrite()`. Esta función recoge como argumentos el objeto `timer`, el periodo en microsegundos y un valor booleano que indica si el temporizador se reiniciará tras acabar el periodo (`true`) o no (`false`).

Por último, se usa la función `timerAlarmEnable()` para activar la cuenta del temporizador. Esta recibe `true` para empezar y `false` para parar.

La función `onTimer()` incrementará la variable global `interruptCounter` en caso de recibir una interrupción, por lo que, identificando este cambio, se podrá ejecutar una acción en consecuencia.

Interrupción por pulsos del encoder

Por otro lado, el encoder funciona enviando pulsos, concretamente 720 pulsos por vuelta, como ya se explicó en **1.6.1 Sensores**. De esta manera, se deberá implementar la función `initPulse()` para inicializar la interrupción por cada pulso. Esta función se encarga de asignar la interrupción al puerto que recibirá los pulsos mediante la función `attachInterrupt()` que recibe como argumentos: el puerto a utilizar como entrada (`pin`); la función que se ejecutará, que en este caso es `onPulse()`; y el tipo de pulsos que vigilará, el cual se ha decidido usar es `RISING`, es decir, un flanco de subida.

La función `onPulse()` incrementará la variable global `pulseCounter` en caso de recibir una interrupción, de modo que, esta servirá para contar el número de pulsos obtenido por el ESP32 desde el encoder anexo al motor analizado.

Tanto el periodo del temporizador (`USECONDS`), como el número de temporizador (`TIMER_ID`) y el valor de la preescala (`TIMER_PRESCALER`), así como los pulsos por vuelta que emite el encoder (`PULSES_PER_LAP`) y el pin de entrada de este (`PULSE_COUNT`), están definidos en el fichero de cabecera `config.h`:

```
#define SAMPLE_RATE     1000  // Hz
#define MSECONDS        1     // (1 / SAMPLE_RATE) * 1000
#define USECONDS        1000  // (1 / SAMPLE_RATE) * 1000000
// (…)
#define PULSES_PER_LAP  720

// (…)

#define TIMER_ID        0
```

```
#define TIMER_PRESCALER 80

// (…)

#define PULSE_COUNT     ADC1_CH1
```

2.3. Bucle de programa

Una vez se ha completado la configuración e inicialización de los periféricos, se enciende el `LED_BUILTIN` para dar conocimiento de ello al usuario. Entonces, se procede con el bucle de funcionamiento, el cual incluye las instrucciones rutinarias que permiten al ESP32 cumplir con su cometido, el cuál es recoger y almacenar medias para ser enviadas por MQTT a un servidor IoT para su posterior procesamiento.

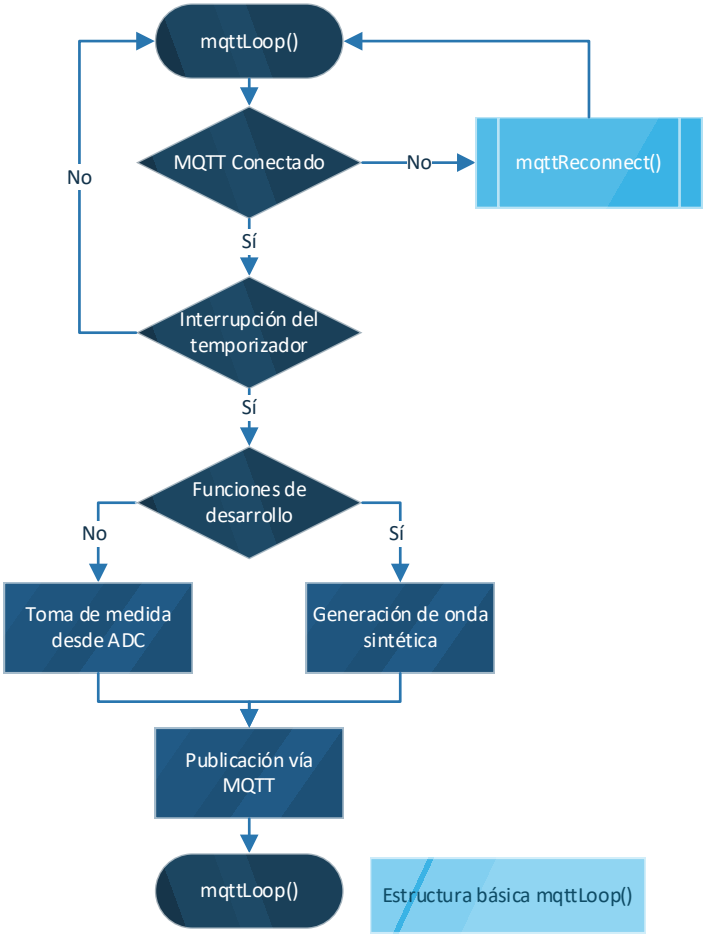


Figura 12. Diagrama de flujo general de mqttLoop()

Estas instrucciones se han implementado bajo la función `mqttLoop()` situada en el fichero `mqtt.ino`, que hace las veces de la función `loop` (véase Fundamentos de la programación con Arduino en el Anexo 2) y su esquema de trabajo se puede observar en la Figura 12.

```
void loop()
{
  mqttLoop();
}
```

El código implementado para realizar las funciones de MQTT está basado en un ejemplo online de libre uso [17].

Puesto que durante la implementación de la solución no se dispone de medidas reales, estas deben ser generadas de forma artificial mientras se está ejecutando el desarrollo de la aplicación. Por esta misma razón, se ha optado por crear una rama paralela al bucle de funcionamiento dedicada a extraer medidas de la onda generada, como se explicará con mayor detenimiento en el apartado 2.3.3 Origen de las medidas en modo de desarrollo.

2.3.1. Conexión al servidor IoT vía MQTT

El comportamiento habitual y el modo de desarrollo tienen como elemento común el establecimiento de la conexión a la red MQTT bajo la función `mqttReconnect()`.

Esta función crea un bucle de tipo `while`, mediante el cual trata de conectarse al servidor MQTT hasta que, por fin, la conexión sea exitosa.

Una vez la conexión se ha establecido, en primer lugar, se ilumina `LED_ERR_MQTT` mediante la función `ledSet()` para indicarlo. A continuación, comienza la suscripción a los siguientes temas, para poder recibir información, mediante la función propia de la biblioteca `PubSubClient.h`, `client.subscribe("tema")`:

```
client.subscribe("esp32/enableOutput");
client.subscribe("esp32/devMode");
client.subscribe("esp32/devModeAdvanced");
client.subscribe("esp32/motorStatus");
```

De no haberse superado la conexión, el sistema mostrará por el puerto serie un código de error y se realizará otro intento de conexión tras 5 segundos.

2.3.2. Recogida y envío de datos

Tras el establecimiento de la conexión MQTT, se ejecuta la función `mqttMainFunction()`, situada en el fichero `functions.ino`.

Para garantizar el posterior envío de datos, se han establecido las variables `adc_raw_value` y `adcBytes[]` que son el valor devuelto por el ADC tras realizar una medida y el buffer que almacena las medidas tomadas por el ADC en cada periodo del temporizador.

El primer paso en esta función es comprobar que ha habido una interrupción por parte del temporizador (véase 2.2.5 Configuración las interrupciones). En caso afirmativo, se discrimina si el ESP32 está en modo de desarrollo o en funcionamiento habitual. La diferencia entre ambos casos radica en el origen de las medidas.

En el modo habitual (ver Figura 13), el valor se recoge directamente del ADC mediante la función `AdcGetMeasure()`, la cual se encuentra implementada en el fichero `adc.ino`. Esta función devuelve la medición actual del ADC seleccionado (mediante la constante `ADC_CHANNEL` definida en `config.h`).

```
// ADC
#define ADC_CHANNEL      ADC1_CH0
```

Sin embargo, en el modo de desarrollo, las medidas se generan mediante la función `mqttDebugFunct()`, como ya se explicará más adelante.

Cada medida se almacena dividida en dos bytes en la variable de tipo *array* `adcBytes`, pues el protocolo MQTT solo envía *arrays* de bytes.

Para realizar la división en bytes se ha implementado la función `getBytesFromInt()`, la cual recibe como argumentos el valor entero y el orden del byte a recibir.

Posteriormente, tras adquirir el número de medidas establecido en `config.h` mediante la constante `SAMPLE_NUM`, se envía `adcBuffer` bajo el tema `esp32/adcBuffer`.

```
// Sampling
// (…)
#define SAMPLE_NUM      1024 * 8
```

Finalmente, se envía, dentro del tema `esp32/speedOutput`, el valor de la variable `rotating_speed` que contiene el valor de la velocidad calculado a partir de los pulsos enviados por el encoder mediante la función `getSpeedFromPulses()`, la cual utiliza el siguiente procedimiento matemático para realizar el cálculo:

Ecuación 6. Cálculo de la velocidad del motor a partir de la señal del encoder

$$n_m = \frac{pulseCounter}{spinTime} \cdot \frac{1\ revolución}{PULSES\_PER\_LAP} \cdot \frac{60\ s}{1\ min}$$

Donde n_m es la velocidad en rpm del motor, `pulseCounter` es la variable global donde se almacena el número de pulsos contados por el ESP32, `spinTime` es una variable local cuyo valor indica el tiempo en segundos que lleva el ESP32 recibiendo pulsos desde la anterior medición (por definición `SAMPLE_NUM / 1024`) y `PULSES_PER_LAP` es una constante definida en el archivo `config.h` (ver **Interrupción por pulsos del encoder**) que determina el número de pulsos equivalentes a una vuelta.

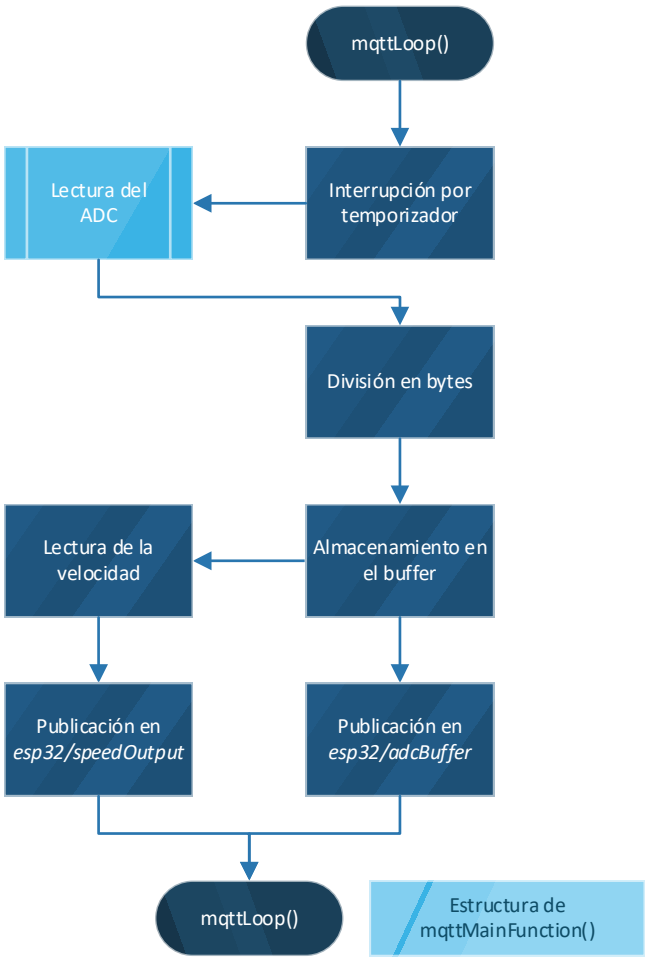


Figura 13.Diagrama de flujo del bucle de funcionamiento mqttLoop().

2.3.3. *Origen de las medidas en modo de desarrollo*

A diferencia del modo de funcionamiento habitual, las medidas enviadas en modo de desarrollo provienen de una onda sintética seleccionada desde la función `mqttDebugFunct()`, la cual recoge como argumento el tiempo desde que el ESP32 está en marcha en milisegundos. Además, se establece la velocidad `DEBUG_SPEED` como velocidad por defecto, ya que en modo de desarrollo no se va a contar con un encoder que proporcione la velocidad.

```
adc_raw_value = mqttDebugFunct(now);
```

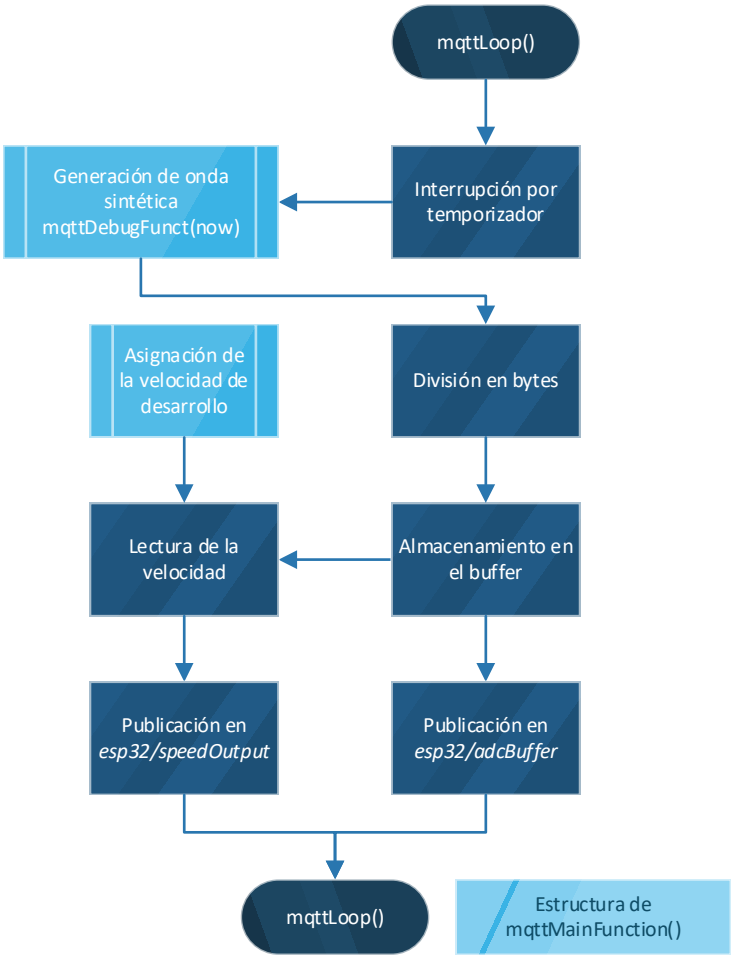


Figura 14. Diagrama de flujo del bucle de funcionamiento mqttLoop() con funciones de desarrollo activadas.

En este punto el programa comprueba el tipo de onda seleccionada y en función de esta genera un tipo de señal distinto:

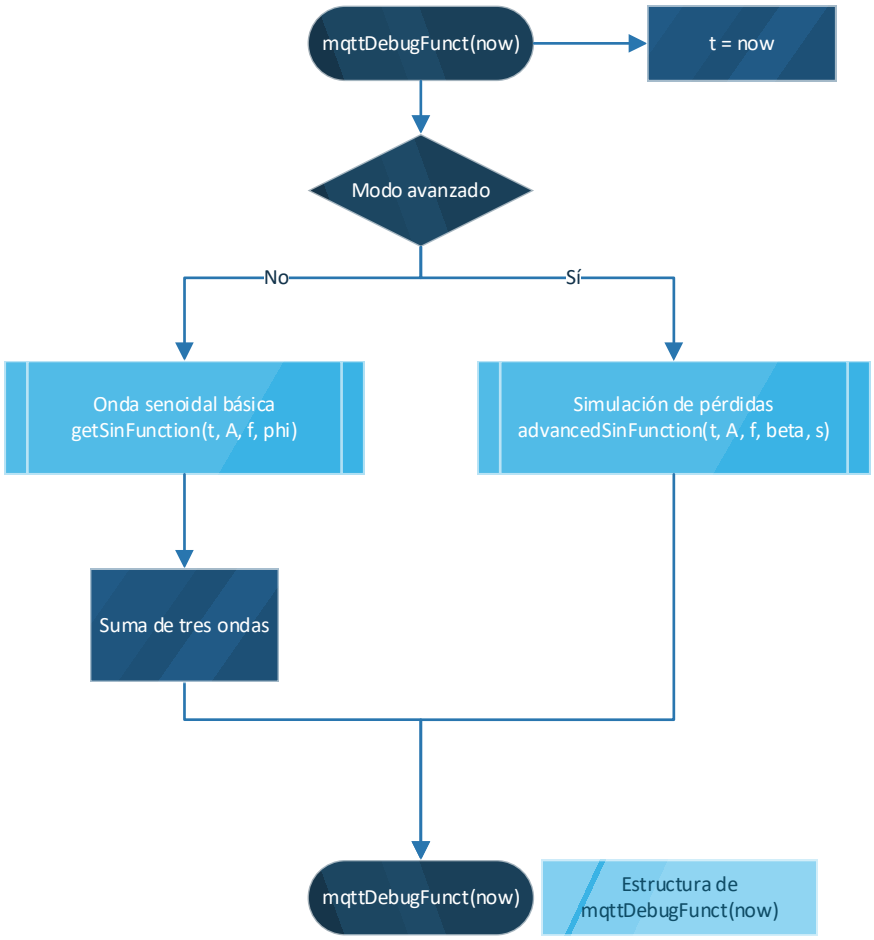


Figura 15. Diagrama de flujo de la función mqttDebugFunct().

Suma de hasta tres señales:

El modelo utilizado para asignar a cada señal su valor es la función habitual de una señal senoidal:

Ecuación 7. Función típica de una señal senoidal pura

$$f(t) = A \cdot \sin(\omega \cdot t + \varphi)$$

Donde:

- A = Amplitud (A)
- ω = 2 · π · f
- f = Frecuencia (Hz)
- t = Tiempo (s)
- φ = Desfase

La función `getSinFunction()` recibe como argumentos el tiempo, la frecuencia, la amplitud y el desfase de cada señal. Esta aplica la Ecuación 7 para obtener el valor de cada señal en el instante de tiempo introducido.

Una vez asignado cada valor, se suman todos para obtener el valor total de la señal en el instante actual.

```
debug_signal = debug_f_funct + debug_g_funct + debug_h_funct +
DEBUG_OFFSET;
```

Se ha añadido el término `DEBUG_OFFSET` para evitar valores negativos durante el análisis mediante FFT que se explicará en el 0.

Modelo de señal de un motor con pérdidas (Modo avanzado)

El modelo utilizado para asignar a cada señal su valor es la función teórica para un motor eléctrico de inducción:

Ecuación 8. Función teórica de una señal de un motor eléctrico de inducción

$$f(t) = A \cdot \cos(\omega \cdot t) \cdot [1 + \beta \cdot \cos(2 \cdot s \cdot \omega \cdot t)]$$

Donde:

- A = Amplitud (A)
- ω = 2 · π · f
- f = Frecuencia (Hz)
- t = Tiempo (s)
- s = Deslizamiento
- β = Coeficiente de fallo

Para crear esta señal se ha empleado el mismo método que con la función anterior. La función `advancedSinFunction()` recibe como argumentos el tiempo, la frecuencia, la amplitud, el deslizamiento y el coeficiente de fallo β de un supuesto motor. Esta aplica la Ecuación 7Ecuación 8 para obtener el valor de cada señal en el instante de tiempo introducido.

```
debug_signal = advancedSinFunction(t, DEBUG_F_A, DEBUG_F_FREQ,
DEBUG_BETA, DEBUG_S) + DEBUG_OFFSET;
```

Se ha añadido el término `DEBUG_OFFSET` para evitar valores negativos durante el análisis mediante FFT que se explicará en el Capítulo 4.

Para definir los valores usados como argumentos en las funciones anteriores se ha recurrido a definirlos como constantes globales en el fichero `debug_signal.h`:

```
#define DEBUG_F_A     1264
#define DEBUG_F_FREQ  50
#define DEBUG_F_PHI   0
#define DEBUG_G_A     0
#define DEBUG_G_FREQ  30
#define DEBUG_G_PHI   1
#define DEBUG_H_A     0
#define DEBUG_H_FREQ  20
#define DEBUG_H_PHI   1
#define DEBUG_OFFSET  1264
#define DEBUG_BETA    1/10
#define DEBUG_S       0.05

#define DEBUG_SPEED   2984
```

2.4. Interrupciones por entrada de datos vía MQTT

Hasta ahora se han explicado los procedimientos para enviar datos desde el ESP32 hacia el servidor IoT, no obstante, esta comunicación es bidireccional y, por tanto, se ha decidido añadir algunas funciones de desarrollo que pueden ser activadas y desactivadas desde un sistema externo vía MQTT.

Puesto que los mensajes MQTT pueden llegar en cualquier momento, el sistema debe poder detener toda ejecución mientras se procesa el mensaje. Esta función que determina qué mensaje ha llegado se denomina *callback*, que podría entenderse como una interrupción.

Esta función se encarga de leer el tema del mensaje y, si corresponde a uno de los temas a los que está suscrito (véase 2.3.1 Conexión al servidor IoT vía MQTT) realizará la acción asociada a este.

- **esp32/enableOutput:** Se espera un mensaje de tipo booleano, mediante el cual se activa o desactiva el envío de datos de forma que el generador de señales de la aplicación en el servidor IoT sea el encargado de suministrar las medidas.
- **esp32/devMode:** Se espera un mensaje de tipo booleano, mediante el cual se activa o desactiva el modo de desarrollo modificando el valor de la variable global `useDebugLoop`. Además, enciende el led externo verde para indicarlo.
- **esp32/devModeAdvanced:** Se espera un mensaje de tipo booleano, mediante el cual se activa o desactiva el modo de desarrollo avanzado (motor con pérdidas) modificando el valor de la variable global `useDebugAdvanced`.
- **esp32/motorStatus:** Se espera uno de estos mensajes provenientes de la aplicación web y que indicarán el estado del motor: `OK`, `ADVISE` o `FAIL`. Estos controlarán los LED del semáforo, pues encenderán las luces verde (`LED_STATUS_GRN`), naranja (`LED_STATUS_YLW`) y roja (`LED_STATUS_GRN`) respectivamente.

Capítulo 3. Estudio de alternativas descartadas

Durante el desarrollo de este proyecto se han realizado varios procedimientos que no llegaron a producir el efecto deseado o generaron errores y malfuncionamientos. Estos son: la inclusión del Sistema Operativo FreeRTOS, la implementación de bases de datos SQLite para almacenar datos de mediciones y por último el uso de SPIFFS, el sistema de archivos del microcontrolador ESP32 para reproducir una señal de un motor real.

3.1. FreeRTOS

Para aprovechar la arquitectura doble núcleo del ESP32, se ha diseñado una estructura de programa de forma que la adquisición de datos se ejecute en un núcleo y la interpretación y el envío de éstos en el otro núcleo como está representado en la Figura 16. Organización del programa.

Esta estructura se basa en FreeRTOS [18], un sistema operativo de código abierto instalado de serie en el ESP32, el cual está diseñado para trabajar con tareas. Para ello incluye una librería de código que soporta la creación de tareas, como la función `xTaskCreatePinnedToCore`, la cual permite definir la tarea y asignársela a uno de los dos núcleos. Esta función se usará siempre en la función `setup` y las tareas se deberán definir antes de iniciar `setup`.

Estas tareas se ejecutan en bucle simultáneamente gracias a la estructura de doble núcleo, ocupando un núcleo cada tarea.

Lamentablemente, este sistema no ha cumplido con las expectativas, ya que ha demostrado ser incapaz de procesar mediciones con el ADC de señales a más de 50 Hz, dando como resultado un constante bloqueo y posterior reinicio del equipo.

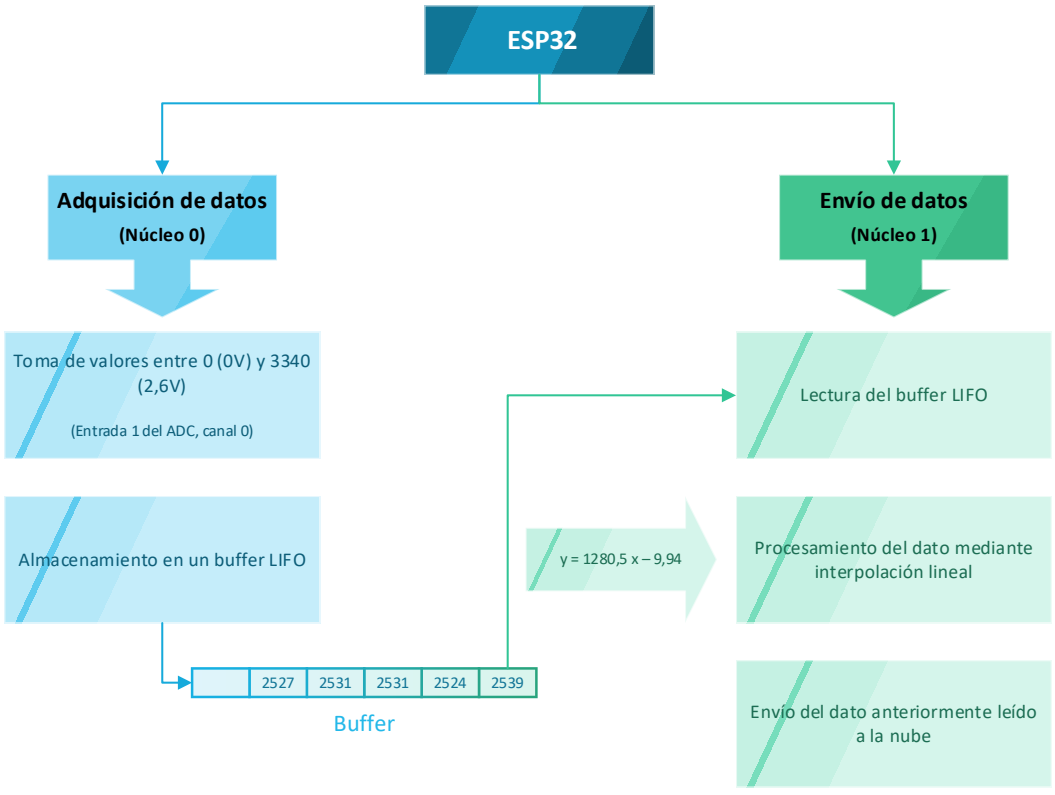


Figura 16. Organización del programa con FreeRTOS

3.2. Uso de SQLite para almacenar valores en Node-RED

Tras comprobar la dificultad para enviar un buffer de números enteros desde el ESP32 hasta el servidor Node-RED vía MQTT, se ha implementado una base de datos SQLite para almacenar los datos que van llegando con el tema `esp32/adcOutput` por lo que, a diferencia de la aplicación final, en esta alternativa el buffer de las medidas de corriente se construye a partir de las mediciones llegadas directamente del ADC.

SQLite es un sistema de gestión de bases de datos de código libre y perteneciente al dominio público. A diferencia de otros sistemas, SQLite utiliza un fichero local para almacenar la información y la interacción con este se realiza a partir de funciones y librerías integrándose así con el código del programa sin necesitar un motor de funcionamiento. Por esta razón SQLite resulta más eficiente que otros sistemas.

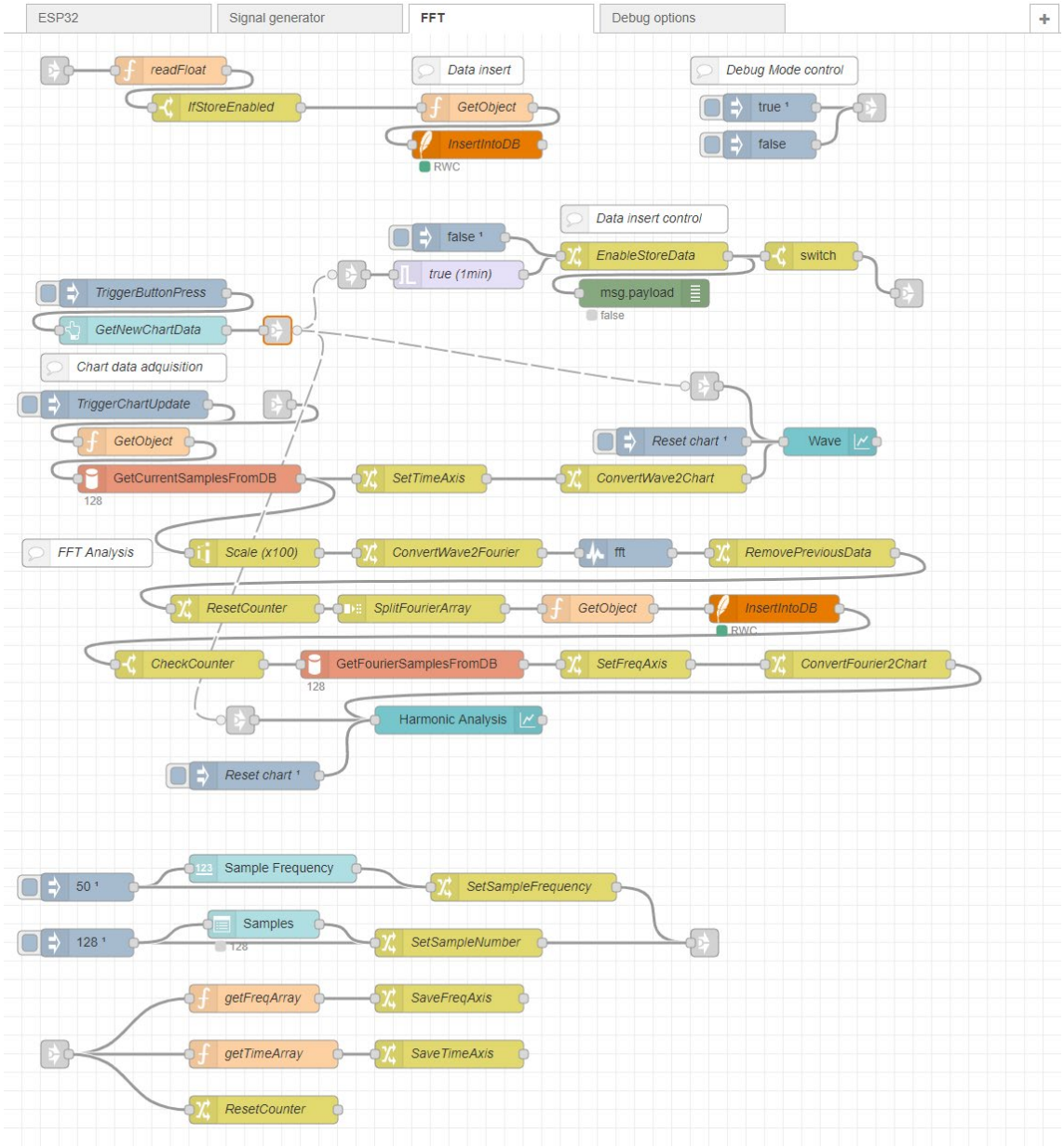


Figura 17. Flujo de análisis de Fourier con almacenamiento en bases de datos SQLite

A partir de una inyección desde el bloque *TriggerButtonPress*, o de una pulsación del botón *GetNewChartData* las gráficas *Wave* y *Harmonic Analysis* se reestablecen mediante el envío de un *array* vacío (`[ ]`). Además, el valor de la variable global `storeData` es establecido en `true` durante un minuto, transcurrido ese minuto, la variable `storeData` pasa a tener el valor `false`.

Mientras la variable global `storeData` tiene el valor `true`, el sistema implementado recoge valores individuales mientras `storeData` tenga el valor `true` y los almacena en una base de datos de SQLite. Puesto que los valores llegan como una cadena de texto, se ha implementado la función `ReadFloat` que interpreta estos valores como números decimales.

La tabla que se usa para los valores recibidos desde el ESP32 ha sido denominada ***current_values*** con los campos ***id***, que es el campo clave y se genera automáticamente incrementando en uno el valor del registro anterior; ***timestamp_value***, que es de tipo fecha y recoge en formato *timestamp* el momento en que se registra la medida y, por último, el campo ***currentvalue***, que es de tipo numérico y recoge el valor recibido de la corriente.

Las instrucciones SQL que permiten la creación de esta tabla han sido las siguientes:

```
CREATE TABLE `current_values` ( `id` INTEGER PRIMARY KEY
AUTOINCREMENT, `timestamp_value` DATE, `currentvalue` NUMERIC )
```

Por este motivo, las instrucciones SQL ejecutadas en el bloque *InsertIntoDB* serán las siguientes:

```
INSERT INTO current_values (timestamp_value, currentvalue)
VALUES ($timestamp, $current);
```

Para que estas instrucciones puedan ser utilizadas, se ha incluido la función JavaScript *getObject* antes del bloque *InsertIntoDB*, que le otorga al mensaje la propiedad *params*, la cual es el siguiente objeto JSON:

```
msg.params = {
    $current: msg.payload,
    $timestamp: Date.now(),
}

return msg;
```

Una vez el valor de la variable global `storeData` se ha reestablecido a `false`, se finaliza el almacenamiento de datos y comienza la extracción de los datos anteriormente introducidos.

En cuanto a la extracción de datos, se ha vuelto a recurrir a la función JavaScript *GetObject*, que, aunque no ha resultado ser estrictamente necesaria, el bloque de extracción devuelve error si no recibe un mensaje con la propiedad *params*.

El bloque que extrae los datos de la corriente almacenados es un subflujo que ha recibido el nombre ***GetCurrentSamplesFromDB***, el cual, como observamos en la Figura 18, se compone de un bloque *switch*, el cual selecciona una salida a partir del número de muestras especificado en la página de monitorización.

Cada alternativa conduce a un bloque *GetArrayFromDB*, que contiene las siguientes instrucciones SQL:

```
SELECT id,currentvalue FROM current_values ORDER by id DESC
LIMIT X;
```

Siendo X el número de muestras anteriormente indicado el cual puede ser 128, 256, 512, 1024, 2048 o 4096.

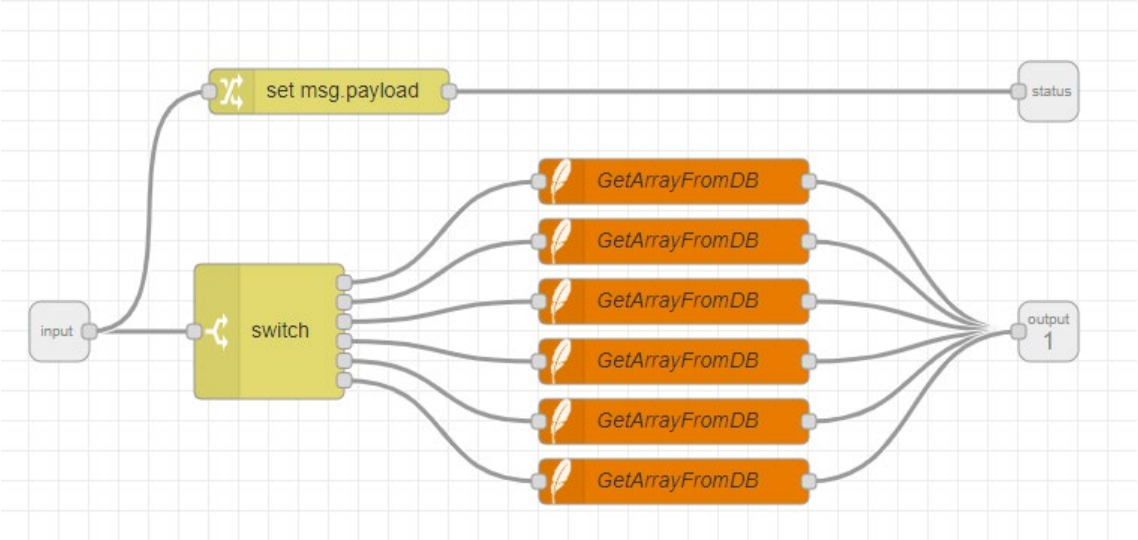


Figura 18. Interior del bloque del subflujo de extracción de datos, GetCurrentSamplesFromDB.

Finalmente, se devuelve un objeto consistente en un array con los últimos X valores de la corriente (campo ***currentvalue*** de la tabla *current_values*) y su número de identificación (campo ***id*** de la tabla *current_values*) almacenados en la base de datos hacia la salida (*output 1*) del subflujo. Adicionalmente, se puede observar el número de muestras escogido a través de la salida de estado (*status*) en la visión general del flujo principal FFT, debajo del subflujo ***GetCurrentSamplesFromDB***.

Una vez obtenido el array de las mediciones de la corriente, se envía hacia la gráfica de *node-red-dashboard*, realizando de antemano las modificaciones necesarias, tales como la adición del eje x mediante el bloque *SetTimeAxis*. En este caso, el bloque *ConvertWave2Chart* modifica la propiedad *payload* del mensaje de forma que sea un objeto JSON con la siguiente estructura:

```
[
   {
       "series": [ "Amplitude" ],
       "data": [ [payload.currentvalue] ],
       "labels": [ xAxis ]
   }
]
```

La propiedad *data* del objeto recoge la propiedad *current_value* del objeto extraído anteriormente.

En cuanto al análisis de Fourier de la onda, tras escalar los datos se ha modificado la propiedad *payload* mediante el bloque *ConvertWave2Fourier* para ajustarla al siguiente objeto:

```
{
   "data": [payload.currentvalue]
}
```

A continuación, se realiza el análisis de Fourier mediante el bloque *fft* como en la aplicación final.

Al contrario que en la aplicación final, antes de producir la división del objeto resultado de la FFT, se establece en cero una variable global llamada `countFreq`, que funciona a modo de contador para evitar el envío de menos valores de los que se han establecido según el número de muestras.

De forma análoga al procedimiento seguido para guardar los valores de la corriente en la base de datos, se ha creado una tabla para almacenar los resultados de la FFT.

Esta tabla ha sido denominada ***fft_values*** con los campos ***id***, que es el campo clave y se genera automáticamente incrementando en uno el valor del registro anterior; ***freq***, que es de tipo numérico y recoge el valor de la frecuencia asociada al resultado actual de la FFT y, por último, el campo ***fft_value***, que es de tipo numérico y recoge el valor del resultado de la FFT.

Las instrucciones SQL que permiten la creación de esta tabla han sido las siguientes:

```
CREATE TABLE `fft_values` ( `id` INTEGER PRIMARY KEY AUTOINCREMENT,
`freq` NUMERIC, `fft_value` NUMERIC )
```

Por este motivo, las instrucciones SQL ejecutadas en el bloque *InsertIntoDB* serán las siguientes:

```
INSERT INTO fft_values (freq, fft_value)
VALUES ($freq, $fft);
```

Para que estas instrucciones puedan ser utilizadas, se ha incluido la función JavaScript *getObject* antes del bloque *InsertIntoDB*, que le otorga al mensaje la propiedad *params* como objeto JSON:

```
let i = global.get("countFreq") || 0;
let array = global.get("tempFreq") || 0;
let f = Number(array[i]);

msg.params = {
    $fft: msg.payload / 100,
    $freq: f
}

i++;
global.set("countFreq", i);

return msg;
```

En esta función se puede observar que a cada medida que se configura mediante esta función, la variable global `countFreq` incrementa en uno su valor y que el valor de la FFT recupera su escala tras ser multiplicado por 100 para eliminar las cifras decimales.

Cuando el bloque *CheckCounter* detecta que ya se han insertado en la base de datos todas las muestras, se procede a la extracción de los datos anteriormente introducidos mediante el subflujo ***GetFourierSamplesFromDB*** el cual es análogo al subflujo ***GetCurrentSamplesFromDB***, obteniendo así un array de los resultados de la FFT.

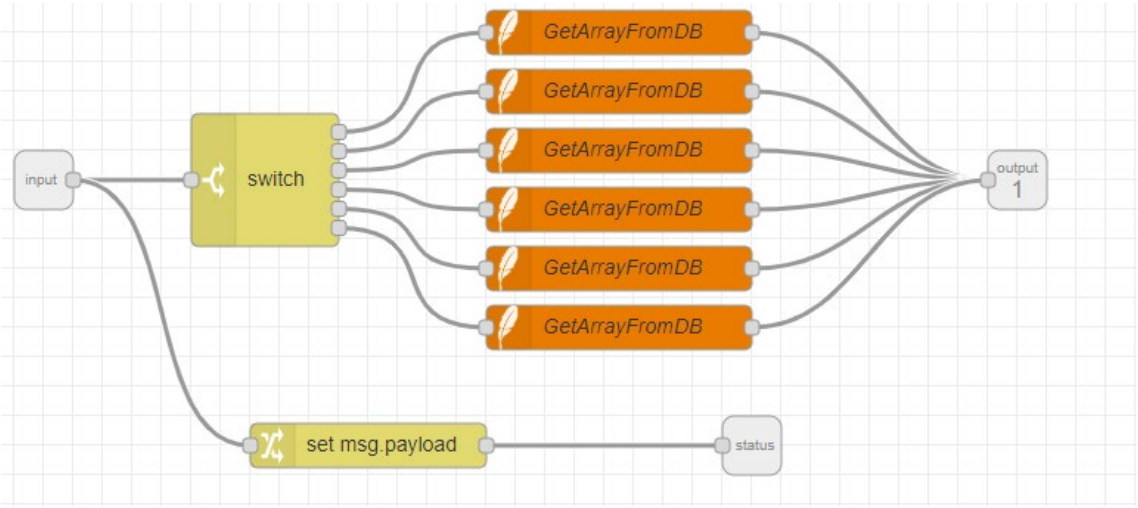


Figura 19. Interior del bloque del subflujo de extracción de los datos de la FFT, GetFourierSamplesFromDB. Su funcionamiento es idéntico al de GetCurrentSamplesFromDB.

Cada alternativa conduce a un bloque *GetArrayFromDB*, que contiene las siguientes instrucciones SQL:

```
SELECT id,fft_value FROM fft_values ORDER by id DESC
LIMIT X;
```

Siendo X el número de muestras anteriormente indicado el cual puede ser 128, 256, 512, 1024, 2048 o 4096.

Una vez obtenido el array de los resultados de la FFT, se envía hacia la gráfica de *node-red-dashboard*, realizando de antemano las modificaciones necesarias, tales como la adición del eje x mediante el bloque *SetTimeAxis*. En este caso, el bloque *ConvertWave2Chart* modifica la propiedad *payload* del mensaje de forma que sea un objeto JSON con la siguiente estructura:

```
[
   {
       "series": [ "FFT" ],
       "data": [ [payload.fft_value] ],
       "labels": [ xAxis ]
   }
]
```

Finalmente, tras todo este procedimiento, se ha conseguido representar tanto la forma de la onda como el análisis de Fourier.

No obstante, esta alternativa se ha descartado por dos razones de peso: La primera es que, debido a la gran cantidad de operaciones en bucle realizadas, el consumo de recursos en la Raspberry Pi es muy elevado, dando lugar a continuos fallos de conexión, así como una velocidad de procesamiento especialmente lenta y un requerimiento de la CPU de hasta 200%, llegando a bloquear por completo el dispositivo.

La segunda es que, al no poder saber con exactitud el periodo de la toma de medidas en la base de datos, se obtienen formas de onda incompletas y un análisis de Fourier de baja precisión.

No obstante, realizar esta alternativa ha sido esencial para asentar la base del funcionamiento de la aplicación final, ya que muchas de las funciones se han adaptado a partir de las creadas para esta alternativa.

3.3. Lectura de señal desde archivo mediante el microcontrolador

Tras conseguir acceso a una medición real de un motor eléctrico de inducción en formato de fichero de texto, se decidió implementar el código necesario para realizar lecturas sobre este fichero desde el ESP32.

Para ello se ha recurrido al sistema de ficheros SPIFFS (Serial Peripheral Interface Flash Fail System) que se puede activar en el ESP32 mediante la inclusión de la librería `SPIFFS.h`.

El formato elegido para este fichero se ha definido en 4.3.4 Envío de señales desde un archivo.

En primer lugar, se ha instalado el plugin para Arduino IDE [19] para poder transmitir los ficheros de texto al ESP32 desde el ordenador. Una vez instalado, se ha iniciado la programación.

La inicialización del sistema de archivos se lleva a cabo mediante la función `initFile()` invocada en la función `setup()` del fichero `main.ino` y declarada en el fichero `file_utils.ino`.

La función `initFile()` contiene, como es habitual, un bucle en el que avisa en caso de fallo durante la activación de SPIFFS y provoca un reintento cada 5 segundos hasta que SPIFFS esté finalmente activo.

Una vez activado correctamente el sistema de gestión de archivos SPIFFS, se ha procedido a la declaración del fichero a transmitir como variable de tipo `File` para que la función `SPIFFS.open(FILE_NAME)` le asigne como valor el contenido del fichero.

La constante `FILE_NAME` está definida en el fichero de cabecera `config.h`:

```
// SPIFFS
#define FILE_NAME             "/sim_ir.txt"
```

Si el fichero no ha sido encontrado o es ilegible, el programa entra en otro bucle hasta que la lectura sea posible.

Una vez abierto el fichero, se muestra en pantalla el contenido de este a través de un bucle de tipo `while` con la condición `file.available()` (disponible gracias a la librería `SPIFFS.h`) la cual es una función que devuelve true hasta llegar al último carácter del fichero, lo que permite guardar el contenido en una cadena de texto.

La lectura se lleva a cabo dentro de este bucle mediante la definición de la variable global `signalString` como un array de caracteres. Esta operación funciona copiando carácter a carácter desde el fichero a la variable `signalString`.

A continuación, se muestra el número de bytes leídos y comienza el procesamiento, que consiste en la conversión de cada carácter de `signalString` en un valor numérico para ser mostrado posteriormente vía puerto serie.

Este valor numérico está representado por la variable de enteros `newData`. Esta recibe cada valor mediante un bucle `for` que logra convertir, de cuatro en cuatro caracteres, la variable `signalString` en un número entero mediante la función `charToInt()`.

```
newData = (charToInt(signalString[i++]) << 12) +
(charToInt(signalString[i++]) << 8) + (charToInt(signalString[i++]) <<
4) + charToInt(signalString[i++]);
```

La función `charToInt()` ha sido implementada en base al código ASCII [20] de los caracteres. Se establecen tres casos:

- Si es un carácter numérico, estará comprendido entre el carácter `'0'` (con valor ASCII 48) y el carácter `'9'` (con valor ASCII 57). En este caso, el resultado devuelto será la diferencia entre el valor del carácter introducido y el correspondiente a `'0'`.
- Si es un carácter alfabético, debe estar comprendido entre el carácter `'A'` (con valor ASCII 65) y el carácter `'F'` (con valor ASCII 70). En este caso, el resultado devuelto será la diferencia entre el valor del carácter introducido y el correspondiente a `'A'` al que se le sumará 10 al ser este el valor de 0xA en decimal.
- En caso de no ser ninguno de los anteriores, el valor devuelto es `-1`, el cual se identificará como error.

```
int charToInt(char x)
{
  if(x >= 48 && x <= 57) return x - 48;
  if(x >= 65 && x <= 70) return x - 55;
  return -1;
}
```

Por último, cada iteración de `newData` es mostrada por pantalla vía puerto serie y es añadida a una variable global de tipo array llamada `signalFromFile` cuyo objetivo es ser enviada como buffer de medidas (véase 2.3.2 Recogida y envío de datos).

Tras la ejecución del programa principal del ESP32, el microcontrolador comenzó a entrar en fallo. En un comportamiento habitual, el sistema muestra una descripción del error vía puerto serie para inmediatamente reiniciarse.

En este caso, el error salta arbitrariamente en cualquier instante del programa y el ESP32 queda bloqueado en mitad de la transcripción del fallo.

```
Se ha abierto el archivo "/sim_ir.txt" correctamente.
Guru Meditation Error: Core  1 panic'ed (InstrFetchProhibited).
Exception was unhandled.
Core 1 register dump:
PC      : 0x15122500  PS      : 0x00060130  A0      : 0x800d4ba4  A1
: 0x3ffb1ef0
A2      : 0x3ffc1a04  A3      : 0x3f4003f8  A4      : 0x00000001  A5
: 0x3f4003a3
A6      : 0x3f400381  A7      : 0x00000001  A8      : 0x800d4b65  A9
: 0x3ffb1ee0
A10     : 0x3ffc1a04  A11     : 0x3f4003f8  A12     : 0x00000008  A13
: 0x0000ff00
A14     : 0x00ff0000  A15     : 0xff000000  SAR     : 0x00000008
EXCCAUSE: 0x00000014
EXCVADDR: 0x15122500  LBEG    : 0x400014fd  LEND    : 0x4000150d
LCOUNT  : 0xfffffffd

ELF file SHA256:
```

Por esta razón, se ha interpretado el fallo como una sobreescritura del programa en la memoria del ESP32 y se ha optado por realizar esta operación desde la aplicación en el servidor IoT (véase 4.3.4 Envío de señales desde un archivo).

Capítulo 4. Desarrollo de la aplicación IoT

La aplicación del servidor debe recibir las medidas recibidas para poder mostrarlas a través de la página web de monitorización. Para esto, deberá tomar cada medida realizada con el ADC e interpretarla para calcular el valor real, ya que la medida que se recibe es un número entero entre 0 y 4095.

Además, deberá ser capaz de enviar configuración al ESP32, como activar o desactivar las señales de prueba.

De la misma forma que el ESP32, se implementará también un generador de señales de prueba por si no se dispusiera de un motor real.

Para transmitir las medidas realizadas por el ESP32 se ha establecido un servidor IoT en la Raspberry Pi. Este servidor se encarga de enviar y recibir los mensajes enviados vía MQTT desde y hacia el ESP32.

Al no disponer de monitor, teclado ni demás elementos de comunicación Usuario-Máquina, el acceso a la Raspberry Pi se realiza mediante Secure Shell (SSH), la cual es una herramienta integrada en Windows 10 para acceder a la terminal de Linux (en este caso Raspbian) de forma remota.

Raspbian, además tiene funcionalidad VNC, el cual provee de un escritorio remoto virtual desde el que se puede acceder a la interfaz gráfica de Raspbian.

Puesto que la función del servidor es enviar las mediciones a la nube y que estas se puedan representar gráficamente en un sitio web, se ha instalado el entorno de programación Node-RED en la Raspberry Pi.

4.1. Introducción a Node-RED

Node-RED es un sistema de programación por bloques basado en Node.js, el cual es un entorno de programación para IoT, a su vez basado en JavaScript. Se ejecuta directamente en el navegador del dispositivo donde se encuentre instalado.

Una aplicación Node-RED puede componerse de varios flujos, los cuales son una especie de lienzos donde colocar bloques y relacionarlos. El orden de estas relaciones es crucial, ya que el mensaje, es decir, la información, se va trasmitiendo en cascada desde el origen en un desplazamiento de izquierda a derecha.

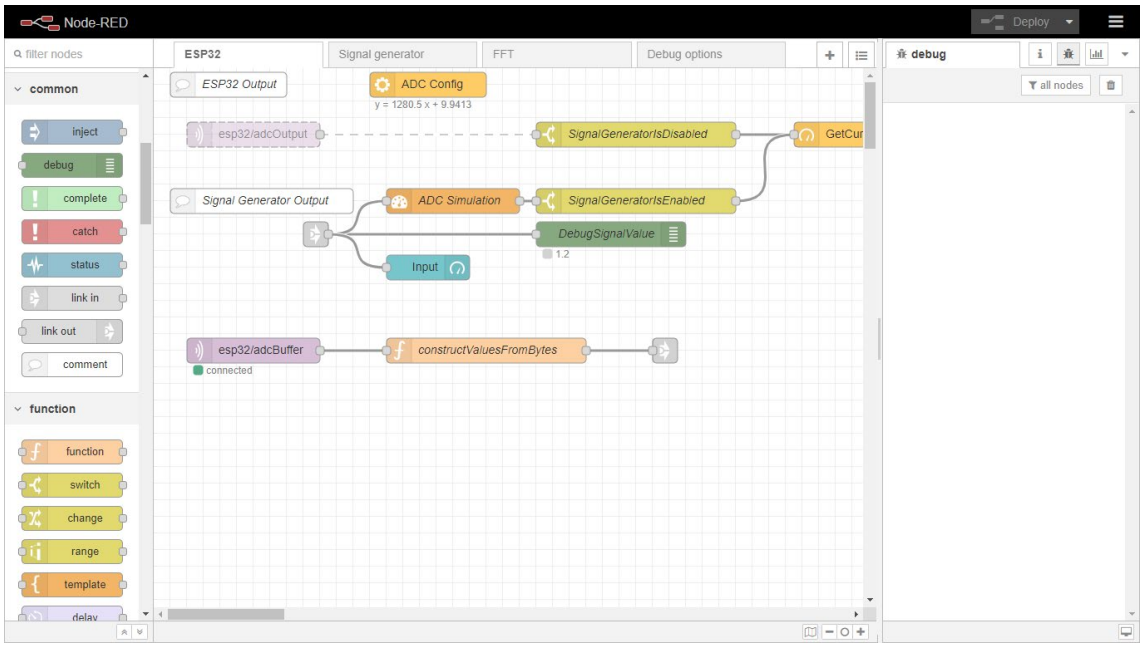


Figura 20. Entorno de programación Node-RED ejecutándose en Microsoft Edge.

Cada bloque tiene su funcionalidad, entre los que se destacan los bloques de *Inyección*, los cuales inyectan un mensaje en el sistema; los bloques de *Función*, que permiten la implementación de una función JavaScript que recoge el mensaje como argumento para ser modificado y posteriormente devuelto; los bloques de control o *Switch*, que crean una bifurcación en el flujo dependiendo de una o varias condiciones; los bloques de *modificación*, que permiten modificar el mensaje en curso o una variable global o local y, por último, los bloques de *depuración*, los cuales sirven para mostrar información de desarrollo en la consola.

Existen también los bloques de *vinculación*, que permiten transmitir mensajes entre flujos, así como varios más, que se usan en menor medida. Adicionalmente, se pueden instalar paquetes de bloques con más funcionalidades, como por ejemplo *node-red-dashboard* que permite la creación de una página de monitorización.

Los mensajes, al ser objetos JSON, pueden tener varias propiedades, que se manifiestan como campos del objeto *msg*. Por defecto todos los mensajes tienen la propiedad *payload*, la cual contiene la información del mensaje. No obstante, la versatilidad de JavaScript permite añadir tantas propiedades como se deseen con tan solo definirlas, por ejemplo, en un bloque de *modificación*.

Una vez terminada la programación, se puede presionar el botón *Deploy* (Desplegar) que lanza o actualiza la aplicación.

Una peculiaridad del entorno de programación Node-RED es la ausencia de un sistema de depuración paso a paso, de forma que al detectar resultados no deseados la única manera de encontrar la causa de estos es ensayo y error.

Lamentablemente la documentación oficial es escasa y dirigida a profesionales acostumbrados a trabajar con sistemas similares, lo cual dificulta también la solución de errores. Asimismo, la comunidad de usuarios de la plataforma es la encargada de resolver dudas en foros oficiales.

4.2. Descripción de la solución implementada

De manera análoga a la estructura modular establecida en el capítulo anterior, las diversas funcionalidades implementadas en Node-RED se han distribuido en varios flujos distintos, que conforman el *back-end*, es decir el código que solo puede ver un programador o administrador. Las funciones se proporcionan en el Anexo 3 Funciones y otros fragmentos de código a modo de documentación.

Para la realización de este sistema se han establecido los siguientes flujos:

- **Monitoring**: gestiona la entrada de datos que se reciben desde el ESP32, y carga el buffer de medidas en la página de monitorización. También se ha diseñado un emulador del ADC del ESP32 para probar ondas sintéticas.
- **Signal generator**: Genera ondas sintéticas.
- **Analysis**: Permite extraer todos los datos referentes al estado del motor mediante la realización de un análisis del espectro de la corriente por el método de la FFT.
- **Signal from file**: Permite abrir un fichero con mediciones anteriormente realizadas y simular una lectura de estas.
- **Initialize and configuration**: Contiene las instrucciones necesarias para que la aplicación IoT se prepare para iniciar su funcionamiento.

Además, se ha llevado a cabo la instalación del plugin para Node-RED *node-red-dashboard* el cual añade controles de interfaz de usuario y funciones para diseñar su disposición en una página de visualización, la cual conforma el *front-end*, es decir, lo que ve el usuario final.

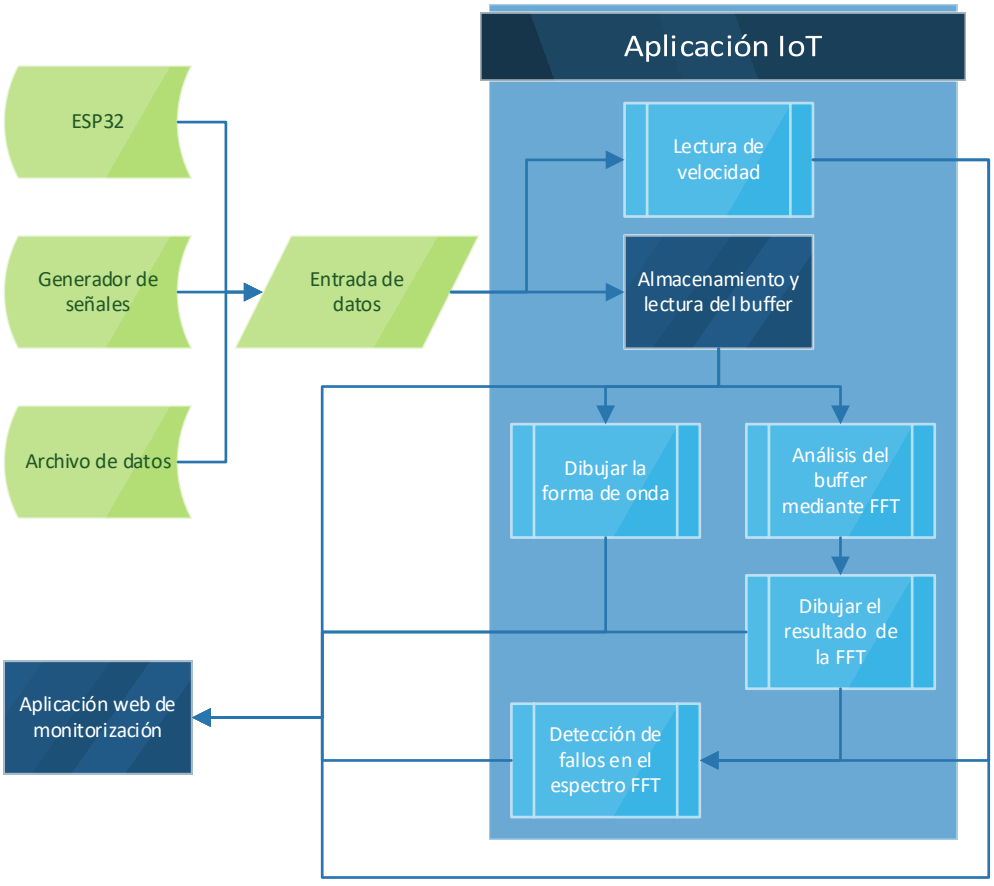


Figura 21. Diagrama de flujo general de la aplicación en el servidor IoT.

4.3. Flujos del programa (Back-end)

A continuación, se describen los flujos creados para construir el programa de la aplicación IoT, los cuales permiten la adquisición de medidas y su posterior análisis. Cabe destacar que durante este apartado se ha omitido la explicación de los controles pertenecientes a la aplicación web de monitorización, pues estos se explicarán posteriormente en 4.3.5 Generador de señales.

4.3.1. Inicialización

Puesto que el sistema Node-RED es monohilo, se deberá prestar atención a no sobrecargar el funcionamiento de la aplicación a base de enviar excesivos mensajes simultáneos.

Con el fin de poder controlar esta situación, se ha diseñado este flujo, el cual, mediante la definición inicial de las variables globales que configuran las funciones del programa, lo que hace que este flujo sea fundamental para el correcto funcionamiento del sistema.

Gran parte de las variables globales son accesibles y modificables desde la aplicación web de monitorización, como se demuestra en 4.4 Aplicación web de monitorización (Front-end).

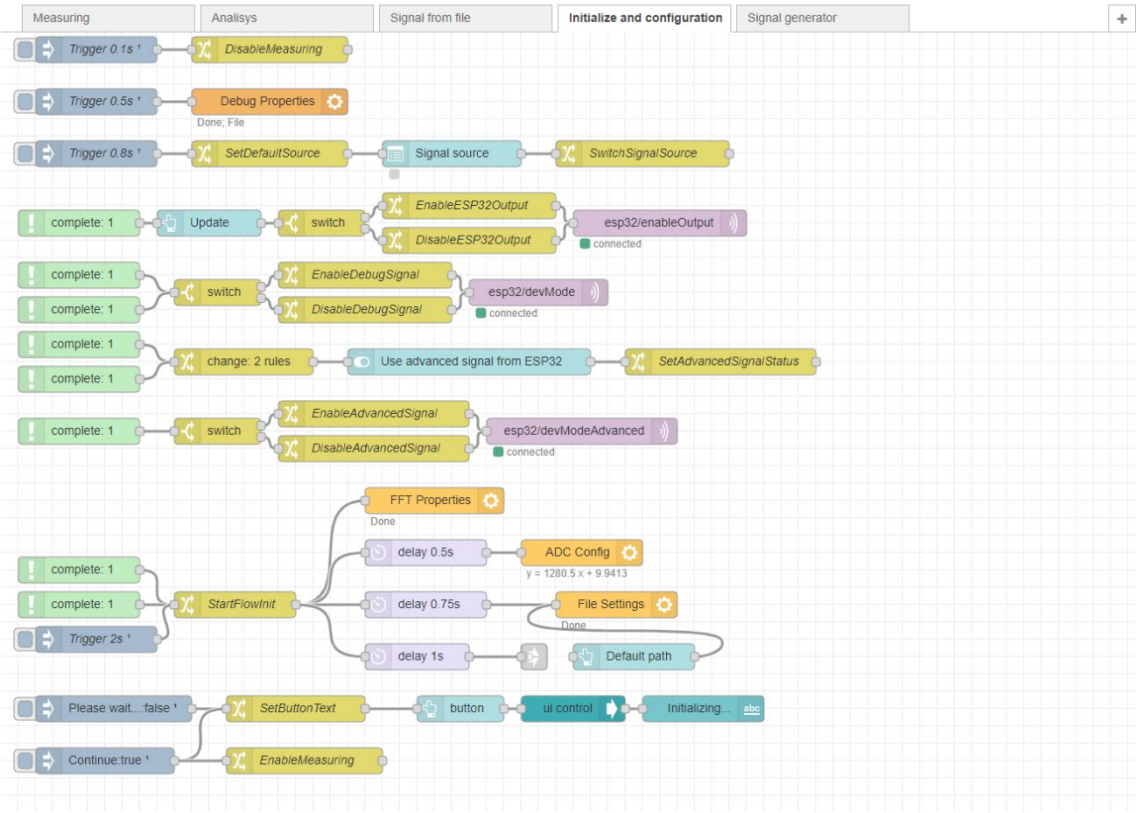


Figura 22. El flujo Initialize and configuration se encarga de la puesta a punto del sistema.

En este flujo se realizan numerosas operaciones de inicialización de variables y restablecimientos a los valores predeterminados de forma ordenada e individual, es decir que, cuando termina una comienza la siguiente hasta finalizar toda la configuración inicial, evitando así una posible sobrecarga de mensajes a procesar por el sistema. Para ello, cada operación de configuración comienza con, o bien un bloque de inyección automática con suficiente tiempo de retardo respecto de la anterior, o bien un bloque de "Compleción", el cual dará luz verde a la operación en cuanto la anterior dé señal de que ha finalizado.

La secuencia de inicialización que tiene lugar en este flujo es la siguiente:

- **0,1 segundos:** En primer lugar, se anula toda entrada de datos estableciendo la variable global *MeasuringIsOn* en `false`. De esta forma se evita que se procesen mensajes con datos provenientes del ESP32 (o de un archivo de datos) durante la ejecución de la inicialización.
- **0,5 segundos:** Se ejecuta el subflujo ***Debug Properties***, desde el cual se selecciona la fuente de datos predeterminada de lectura mediante una lista desplegable. La fuente elegida es configurada a través de la función `setDebugProperties` (Initialize… > Debug Properties) (Anexo 3, **¡Error! No se encuentra el origen de la referencia.**) estableciendo las variables globales *DebugSignalSource*, *DebugESP32Advanced* en los valores especificados para las configuraciones correspondientes y dándole el valor inicial de 0 a *MeasuringArrayIndex*.
- **0,8 segundos:** El valor anteriormente establecido en *DebugSignalSource* se copia al desplegable de la **Aplicación web de monitorización (Front-end)**.

. Este mismo paso se realizará al cambiar la fuente de medidas desde la aplicación web.

- Al terminar el proceso anterior, si el **Generador de señales** (4.3.5) o el **Envío de señales desde un archivo** (4.3.4) están activos se desactiva la salida del ESP32 mandando un mensaje con contenido "false" bajo el tema *esp32/enableOutput*. De haberse elegido recoger medidas desde el ADC o desde la señal de desarrollo, activa la salida del ESP32 al mandar un mensaje de contenido "true" bajo el tema *esp32/enableOutput*.
- Una vez acabado el proceso anterior, activa o desactiva el modo de desarrollo del ESP32 tras mandar un mensaje de contenido "true" o "false" respectivamente bajo el tema *esp32/devMode* dependiendo de si se ha elegido recoger medidas desde la señal de desarrollo o no al establecer el valor de *DebugSignalSource*.
- Al terminar el proceso anterior, si se ha establecido la señal avanzada del ESP32 como salida, se activa el modo de desarrollo avanzado del ESP32 tras mandar un mensaje de contenido "true" bajo el tema *esp32/devModeAdvanced*. En caso contrario, el mensaje enviado llevará como contenido "false" y desactivará el modo de desarrollo avanzado en el ESP32.

A continuación, la configuración de los flujos comienza:

- **2 segundos:** Se ejecuta el subflujo ***FFT Properties***, el cual permite establecer el número y frecuencia de muestras (*FourierSampleNum* y *FourierSampleFreq* respectivamente) e inicializa el array de la FFT (*FourierArray*) a una cadena vacía el nivel de zoom al 100% (*FourierWaveZoom*) y las unidades del análisis FFT a dB (*FourierUseDB*) mediante la función `SetFourierProperties` (Initialize… > Debug Properties) (Anexo 3, **¡Error! No se encuentra el origen de la referencia.**).
- **2,5 segundos:** Se ejecuta el subflujo ***ADC Config***, el cual recibe los parámetros de la ecuación de linealidad del ADC (Figura 23) y el número de decimales deseado y los almacena en variables globales (*MeasuringAdcA*, *MeasuringAdcB* y *MeasuringAdcFloat*, respectivamente) para que pueda ser usado por todos los elementos de la aplicación. Estos parámetros responden a la ecuación de la forma "$y = a \cdot x + b$" donde "y" se corresponde con la interpretación del ADC (ver **Anexo 1 Caracterización del ADC**) y "x" con el valor real de la medida. De esta forma, "a" y "b" son los términos lineal e independiente respectivamente, como se intuye a partir de la Ecuación 9. Se emplea la

función `SetAdcProperties` (Initialize… > Debug Properties) (Anexo 3, **¡Error! No se encuentra el origen de la referencia.**)

Ecuación 9. Relación entre el valor devuelto por el ADC y el valor real de la medida

$$y = 1280{,}5x - 9{,}9413$$

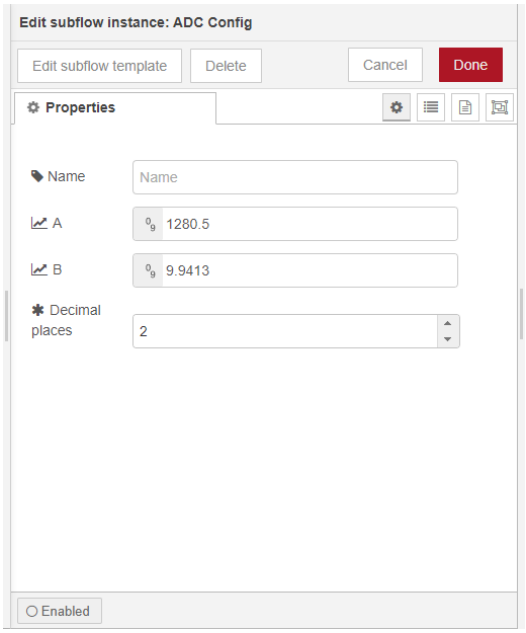


Figura 23. Parámetros del subflujo ADC Config correspondientes a la zona lineal del ADC

- **2,75 segundos:** Se ejecuta el subflujo ***File Settings***, el cual permite establecer el directorio de los archivos de las señales de las tres fases de un motor en una variable global en forma de objeto (*FileDirectories*), la duración en segundos de la simulación (*FIleSimLength*, el tamaño del buffer a analizar) e inicializa la fase a analizar como "R" por defecto (*FileCurrentPhase*) mediante la función `SetFileSettings` (Initialize… > Debug Properties)(Anexo 3, **¡Error! No se encuentra el origen de la referencia.**).
- **3 segundos:** Se actualiza la interfaz de la aplicación web con los datos anteriores.
- **4 segundos:** Finalmente se activa la entrada de datos estableciendo la variable global *MeasuringIsOn* en `true`.

Cabe destacar que los mensajes que se envían, pese a ser, en ocasiones, valores booleanos (es decir, verdadero o falso) se envían como cadenas de texto debido a las limitaciones impuestas por el sistema MQTT empleado para la comunicación sensor-servidor IoT.

4.3.2. Adquisición de medidas

Este flujo se compone de tres ramas: la primera se encarga de recoger los datos recibidos desde el ESP32 mediante los temas MQTT *esp32/adcBuffer* y *esp32/speedOutput*. Estos se corresponden al buffer de medidas de corriente y a la monitorización de la velocidad respectivamente. Puesto que los datos anteriores son emitidos en formatos que no son procesables, se deben convertir en datos que puedan ser interpretados por la aplicación IoT.

La adquisición de las medidas realizadas por el ESP32 se realiza mediante una subscripción a través del protocolo MQTT. El procesador ESP32 toma una medida cada milisegundo, lo que se

traduce en una frecuencia de muestreo de 1 kHz. En la Ecuación 10 se puede observar la relación entre la frecuencia de muestreo (*f*) y el periodo de toma de datos (T).

Ecuación 10. Cálculo de la frecuencia de muestreo

$$f = \frac{1}{T} = \frac{1}{10^{-3}s} = 1000\ Hz = 1\ kHz$$

Para evitar sobrecargar el sistema con 1000 mensajes cada segundo, se envía en su lugar un *buffer* (cadena de valores, también denominada *array*) cada 8 segundos con las medidas realizadas en ese lapso de tiempo. Finalmente, este *buffer* contendrá 8000 medidas tomadas en 8 segundos y permitirá analizar y representar la onda en 4.3.3. Análisis de señales

Una vez recibido el buffer de la corriente, este debe ser escalado correctamente a partir de la interpretación del ADC. Cabe recordar que el ADC interpreta los valores medidos como un número entero entre 0 y 4095, los cuales siguen una relación de transformación lineal tal como se indica en el Anexo 1 Caracterización del ADC.

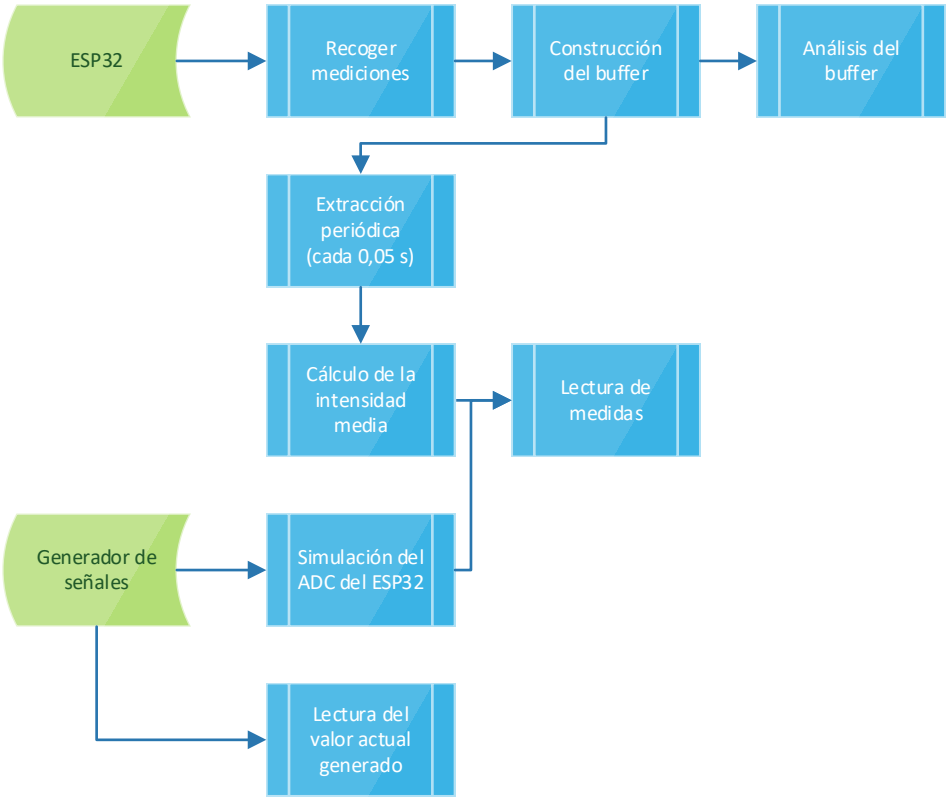


Figura 24. Diagrama de flujo del funcionamiento del flujo Measuring.

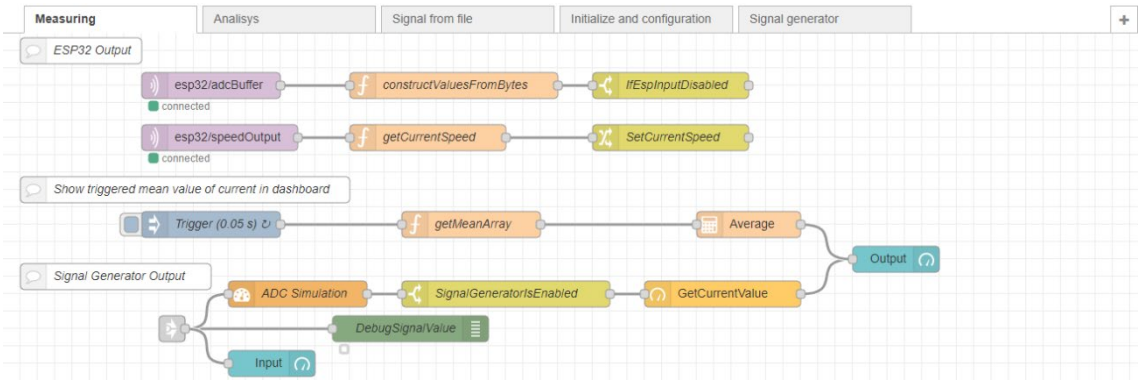


Figura 25. Flujo Measuring dedicado a recoger medidas.

Debido a las limitaciones encontradas al utilizar el protocolo MQTT para Arduino, el envío del *buffer* se ha realizado en forma de bytes, es decir, grupos de 8 bits. Tal como se ha explicado en 1.6.2 Sistema de Control, se cuenta con un ADC de 12 bits, lo que hace que cada medición se tenga que enviar como una pareja de dos bytes, lo que hace un total de 16000 bytes.

Estos bytes emparejados deben ser reagrupados para que se puedan interpretar, por lo cual se ha implementado la función **`constructValuesFromBytes`** (Measuring), la cual recorre todos los valores de dos en dos para agrupar los bytes mediante operaciones de desplazamiento de bits.

Finalmente, devuelve una cadena con los 8000 valores convertidos desde la interpretación del ADC a valores reales con dos decimales. Esta cadena se almacena en la variable global *MeasuringArray*, la cual se envía inmediatamente después al flujo **FFT** donde será utilizado para analizar la onda.

En cuanto a la velocidad de giro, es enviada como una cadena de texto, por lo que hay que convertirla a su valor numérico. De ello se encarga la función **`getCurrentSpeed`** (Measuring). A continuación, un bloque de modificación cambiará el valor de la variable global *MeasuringCurrentSpeed* al valor numérico de la velocidad de giro anteriormente extraído.

La segunda rama consiste en la visualización de la intensidad en la Generador de señales

Debido a que, por causas mayores, se ha dejado de disponer tanto del generador Analog Discovery como de su software WaveForm, se ha implementado un generador de señales básico. Este ha sido utilizado para diseñar la interfaz de usuario.

Este flujo básicamente genera, mediante funciones JavaScript, una señal que es enviada al flujo anterior para simular una medición del ADC del ESP32.

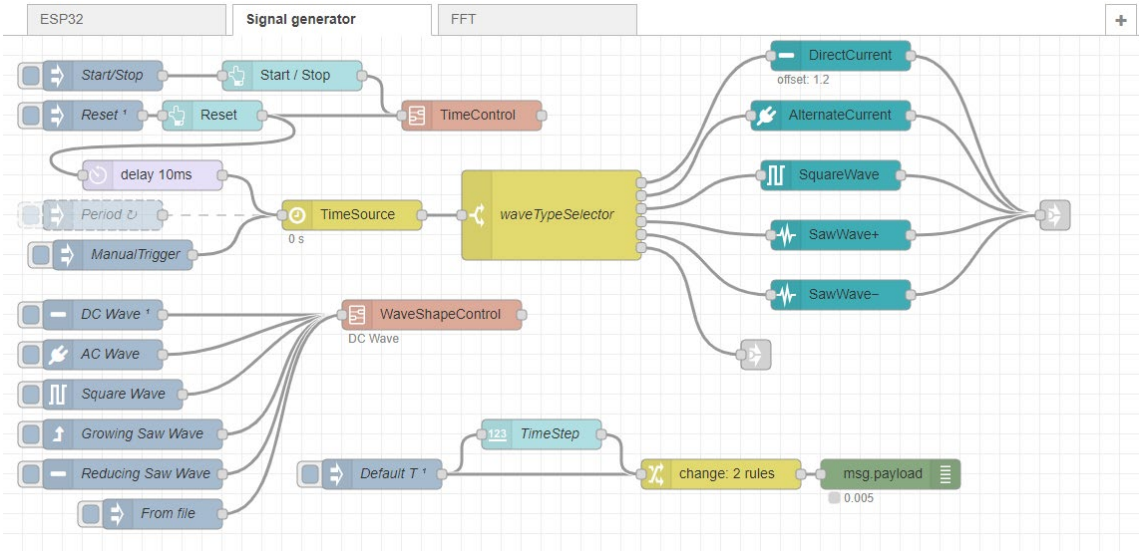


Figura 30. Flujo Signal generator dedicado a generar ondas sintéticas

Para tal efecto, se han creado cinco subflujos (***DirectCurrent***, ***AlternateCurrent***, ***SquareWave***, ***SawWave+*** y ***SawWave-***) con funciones de JavaScript, uno por cada tipo de onda. El modelo en el que se ha basado este sistema ha sido el software WaveForms del que se ha hablado en el Capítulo 1 (Sistema de Control).

El sistema dispone de un botón “Start / Stop”, el cual hace que comience a correr el tiempo, y de un botón “Reset”, el cual reinicia el reloj a cero.

El efecto de tiempo se ha conseguido mediante una inyección periódica a una función JavaScript (`TimeSource`) que incrementa una variable global llamada `time`. Esta se toma como entrada para las funciones que generarán las ondas.

La selección del tipo de onda se lleva a cabo mediante un bloque `switch`, el cual dirige el valor actual del tiempo hacia la función escogida, dependiendo del tipo de onda indicado en `WaveShapeControl`.

Los parámetros de cada tipo de onda se pueden configurar mediante variables locales de cada subflujo como se puede observar en la Figura 31.

Cabe destacar que, a medida que el tiempo corre, el sistema se ralentiza de manera notable, de modo que este generador de señales únicamente se ha utilizado durante el desarrollo de la aplicación.

Aplicación web de monitorización (Front-end). Para ello se establece un bloque de inyección programada cada 0,05 segundos que ejecuta la función `getMeanArray` (Measuring), la cual recoge, en orden, grupos de 50 medidas desde *MeasuringArray*, para posteriormente calcular la media del grupo y ser mostrada, gracias al bloque “gauge” *Output*, en la Aplicación web de monitorización (Front-end).

La tercera rama es el 4.3.5 Generador de señales en la cual, entre la recepción de datos y el subflujo ***GetCurrentValue***, se ha colocado un bloque de control que sirve a modo de semáforo, de forma que bloquea el paso de la información si la fuente definida en la variable global *DebugSignalSource* no es **Signal generator**.

Las señales son generadas en la propia aplicación y pasan anteriormente por un simulador del ADC del ESP32. Este subflujo llamado ***ADC Simulation*** consta de un bloque de control, en la cual dependiendo del rango de la medida tomará una zona del ADC y usará la relación que convenga.

Además, el valor sin convertir puede verse en la página de monitorización a través del bloque "gauge" *Input* (también desde el bloque de depuración *DebugSignalValue*).

Una vez tomado el valor del ADC (ya sea real o simulado), este se hace pasar por el subflujo ***GetCurrentValue***, en el cual se transformará según la ecuación correspondiente a la zona de comportamiento lineal del ADC.

Para poder comprobar de forma rápida el comportamiento del subflujo, se ha añadido una salida de estado en la cual se muestra el valor convertido en cada momento.

Finalmente, el valor ya convertido se envía al bloque *Output*, el cual representa este valor en la Aplicación web de monitorización (Front-end).

4.3.3. Análisis de señales

El desencadenante de un análisis de señales es el bloque de inyección programada *TriggerAnalysis* cada minuto, que provoca el establecimiento del nivel de zoom en la gráfica de forma de onda al valor de la variable global *FourierWaveZoom*.

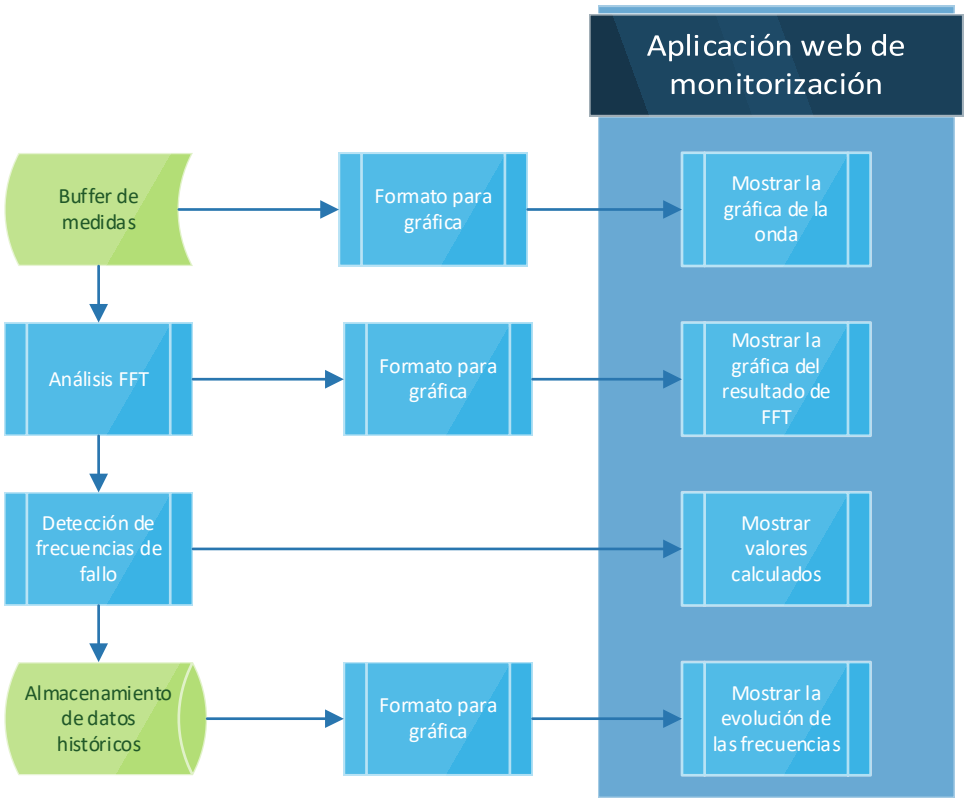


Figura 26. Diagrama de flujo del análisis de señales en la aplicación IoT.

Como se ha explicado en 4.3.2 Adquisición de medidas, el buffer de las mediciones llega desde el ESP32 en forma de array de bytes e inmediatamente se convierte en un array de números con

dos cifras decimales tras juntar los bytes y traducir el resultado desde el valor indicado por el ADC a un valor real de corriente. Este array se encuentra en la variable global *MeasuringArray*.

Una vez obtenido el array de las mediciones de la corriente, se ha implementado una gráfica llamada *Wave* para poder representar la forma de la onda. Esta gráfica se controla mediante un bloque incorporado en *node-red-dashboard* y está basada en Chart.js [21].

El formato que requiere la gráfica para poder interpretar los datos del array de mediciones es el siguiente:

```
[
   {
      “series”: [“nombreDeLaSerie”],
      “data”  : [[y1, y2, y3, …, yn]],
      “labels”: [“x1”, “x2”, “x3”, …, “xn”]
   }
]
```

En resumen, se requiere de un objeto JSONata [22] en el que se introducen los datos necesarios para crear una gráfica. Este objeto consta de tres campos: `“series”`, `“data”` y `“labels”`.

En verde se representan las cadenas de texto y en naranja los valores numéricos.

Para representar la forma de la onda se ha hecho pasar el array de mediciones por una serie de bloques encargados de adaptar los datos del array a la estructura del objeto JSONata especificada anteriormente.

El primer bloque es *SetTimeAxis*, el cual añade al mensaje un array con los valores pertenecientes al eje X en la propiedad *xAxis*. Este array es la variable global *FourierTimeAxis*, que se ha conseguido mediante la función **`getTimeArray`** (Analysis) que se ejecuta tras la ejecución de *TriggerAnalysis* cada minuto.

Esta función crea un array desde *0* sumándole el periodo al valor anterior, hasta conseguir completar un segundo de gráfica. Posteriormente, el array del eje X se almacena en la variable global *FourierTimeAxis* mediante el bloque *SaveTimeAxis*.

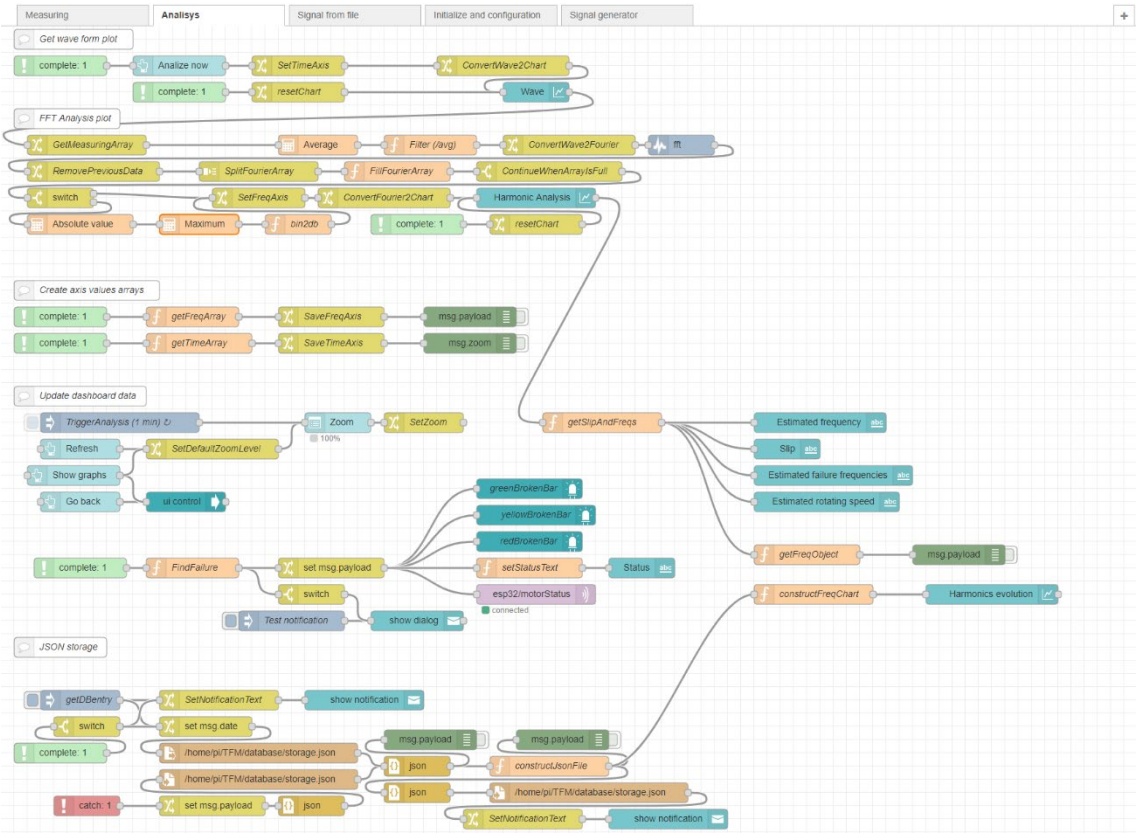


Figura 27. Flujo FFT destinado a realizar el análisis de Fourier y representar las gráficas.

Una vez se ha insertado el array del eje X en el mensaje con las mediciones, el siguiente bloque, *ConvertWave2Chart* se encarga de darle el formato necesario modificando el contenido de este para coincidir con el siguiente objeto JSONata:

```
[
   {
       "series": [ "Amplitude" ],
       "data"  : [ [payload] ],
       "labels": [ xAxis ]
   }
]
```

Finalmente, la gráfica se muestra en la página de monitorización.

A continuación, se recurre al análisis de Fourier mediante la Transformada Rápida de Fourier (FFT) para identificar posibles armónicos en la onda. Para ello, se debe adaptar el mensaje para que el array de las mediciones de la corriente pueda ser analizado por el bloque *fft*. Para ello se ha implementado el bloque *ConvertWave2Fourier* para modificar la propiedad *payload* y ajustarla al siguiente objeto:

```
{
   "data": [payload]
}
```

El bloque *fft* devuelve un objeto consistente en tantos campos como muestras se hayan establecido además del campo *data* que se ha utilizado como entrada. Todos los campos de los resultados tienen un nombre de la forma "*fft_x*", en el que x representa el orden en que se tomó

la medida. Esta forma de devolver los resultados impide su tratamiento para ser mostrado en una gráfica de igual forma que se ha procedido con la forma de la onda.

Por esta misma razón se ha optado por convertir este objeto en un array, en primer lugar, eliminando el campo *data* mediante el bloque *RemovePreviousData*. A continuación, se ha añadido el bloque *SplitFourierArray* que convierte cada resultado de la FFT en un mensaje individual. Para reunir todos los mensajes individuales en un array, el primer paso ha sido crear el array vacío *FourierArray* como variable global durante la **Inicialización** (4.3.1).

El siguiente paso consiste en rellenar *FourierArray* con todos los mensajes procedentes del análisis mediante FFT. Para ello se ha implementado la función `fillFourierArray` (Analysis), que se ejecutará tantas veces como mensajes lleguen para asignar el valor de estos mensajes a la posición correspondiente del array. Esta función crea una copia de la variable *FourierArray* llamada *FourierArrayBackup* ya que la primera se vacía tras cada envío. El siguiente bloque, *ContinueWhenArrayIsFull*, no permitirá que continúe la ejecución hasta que *FourierArray* no tenga todos los datos correspondientes.

Para representar la gráfica de la función de Fourier se ha hecho pasar el array de mediciones por una serie de bloques encargados de adaptar los datos del array a la estructura del objeto JSONata.

El primero es un bloque de control que comprobará el valor de *FourierUseDB* para determinar la unidad de la FFT, si se proporcionará en amperios o decibelios. En caso de haber elegido expresarlo en dB se ejecutará la función

bin2DB (Analysis), la cual convertirá los valores de *FourierArray* a decibelios mediante la siguiente ecuación:

Ecuación 11. Cálculo del valor en decibelios de los elementos del array de medidas

$$y(dB)_i = 20\log\frac{y(A)_i}{\max\left(y(A)\right)}$$

En la que y(dB) es el *array* en decibelios e y(A) es el array en amperios e *i* es el índice del elemento.

El siguiente bloque es *SetFreqAxis*, el cual añade al mensaje un array con los valores pertenecientes al eje X en la propiedad *xAxis*. Este array es la variable global *FourierFreqAxis*, que se ha conseguido tras la ejecución de *TriggerAnalysis* cada minuto.

Esta función crea un array a partir de la Ecuación 12, hasta conseguir tantos valores como medidas haya en el array de medidas.

Ecuación 12. Obtención del eje X de la función FFT.

$$f_n = \frac{n \cdot f_m}{N}$$

Posteriormente, el array del eje X se almacena en la variable global `tempFreq` mediante el bloque *SaveFreqAxis*.

Una vez se ha insertado el array del eje X en el mensaje con las mediciones, el siguiente bloque, *ConvertFourier2Chart* se encarga de darle el formato necesario modificando el contenido de este para coincidir con el siguiente objeto JSONata:

```
[
   {
       "series": [ "FFT" ],
       "data"  : [ [payload] ],
       "labels": [ xAxis ]
   }
]
```

Finalmente, la gráfica se muestra en la página de monitorización.

A continuación, se ejecuta la función **`getSlipAndFreqs`** (Analysis), la cual calcula, en primer lugar, la frecuencia fundamental, la cual es la que tendrá el valor más elevado dentro del espectro extraído en la FFT, el cuál se encuentra en forma de *array* en la variable global *FourierArrayBackup*. A continuación, se calcula el deslizamiento mediante la Ecuación 13, en la que *s* es el deslizamiento, n_s es la velocidad de sincronismo (habitualmente es 3000 rpm para 50 Hz) y n_m es la velocidad del motor medida en el ESP32 y almacenada en la variable global *MeasuringCurrentSpeed*.

Ecuación 13. Cálculo del deslizamiento

$$s = \frac{n_s - n_m}{n_s}$$

Una vez determinada la frecuencia fundamental y el deslizamiento, esta se utiliza para calcular las frecuencias de fallo asociadas a la rotura de barras de la forma que se ha deducido de la Tabla 2. Entonces, el deslizamiento, la frecuencia fundamental y las frecuencias de fallo se almacenan

en las variables globales *MeasuringSlip*, *FourierMainFreq* y *FourierFailFreqs* respectivamente y se envían incrustadas en el mensaje para ser mostradas en la **Aplicación web de monitorización** (Front-end).

A continuación, se crea un objeto JavaScript en la variable global *FourierFreqObject* para almacenar los resultados y mostrar la información de estos mediante la función **`getFreqObject`** (Analysis). Este objeto permite identificar la frecuencia, valor y posición en el espectro de cada frecuencia de fallo y tiene la siguiente estructura:

```
{
  "MAIN" : {
      index : freq0index,
      freq  : freq0,
      value : freq0Value
  },
  "BAR_1": {
      index : freq1index,
      freq  : freq1,
      value : freq1Value
  },
  "BAR_2": {
      index : freq2index,
      freq  : freq2,
      value : freq2Value
  }
}
```

Tras la creación de este objecto, se ejecuta la función **`FindFailure`** (Analysis) mediante un bloque de compleción. Esta función se encarga de detectar valores asociados a fallos dentro de la variable global *FourierFreqObject*, la cual se había definido en la función anterior como un objeto con los datos de las frecuencias de fallo. La detección de estos valores se realiza comparando el campo *value* en cada elemento del objeto con los umbrales de fallo. Si el valor es mayor o igual a -45 dB se considera que el motor está fallando, si por el contrario se encuentra entre -45 dB y -55 dB, se identifica como sospecha o alerta de posible futuro fallo.

Tras la identificación de los posibles fallos, la función construye un mensaje de aviso que constituye una ventana emergente e ilumina un piloto del semáforo que identifica el estado del motor con un código de colores:

Tabla 3. Código de colores del estado del motor

	Verde	El motor no presenta síntomas de fallo.
	Amarillo	Comienzan a aparecer amplitudes asociadas a averías en determinadas frecuencias.
	Rojo	Existen valores críticos en amplitudes asociadas a averías en determinadas frecuencias. El motor está fallando.

Este mismo código es enviado vía MQTT al ESP32 para encender el LED correspondiente bajo el tema *esp32/motorStatus*.

Tras haber procesado e interpretado el resultado del análisis, el siguiente paso es el almacenamiento de estos valores para poder disponer de una evolución de las frecuencias susceptibles de provocar fallos en el motor. Para ello, se emplea una base de datos JSON que almacena un array de 90 elementos. Cada uno de estos elementos contendrá con una instancia del objeto *FourierFreqObject* correspondiente a un análisis. Al monitorizar esta base de datos se consigue observar la evolución de las frecuencias de fallo asociadas a una rotura de barras del rotor.

Esta base de datos se completará a medida que se realicen más mediciones a través de la función **`constructJsonfile`** (Analysis), la cual lee el actual array del archivo JSON de la base de datos y le añade el último objeto medido y la fecha y hora en que fue tomado. Al completar los 90 valores del array, se elimina el primer valor recibido para hacer espacio al siguiente, a modo de buffer circular. Tras la adición del nuevo valor, los cambios en la base de datos son guardados en el servidor.

Por otro lado, la función anterior transmite el mensaje del nuevo array, de forma que posteriormente pueda ser adaptado a una gráfica por la función **`constructFreqChart`** (Analysis), transformado dicho array en un objeto JSONata con la siguiente estructura:

```
{
    “series": [["BAR 1"], ["BAR 2"]],
    “data”  : [[bar1_data],[bar2_data],
    “labels": [tiempoMedida]
}
```

De esta forma, se muestra la evolución de las frecuencias de fallo en la **Aplicación web de monitorización** (Front-end).

4.3.4. Envío de señales desde un archivo

Debido a que, durante el trascurso del año 2020, debido a la crisis sanitaria producida por la pandemia del coronavirus SARS-CoV-2, se decretó el confinamiento a través de la declaración de Estado de Alarma por parte del Gobierno de España, ha resultado imposible realizar ensayos en motores reales.

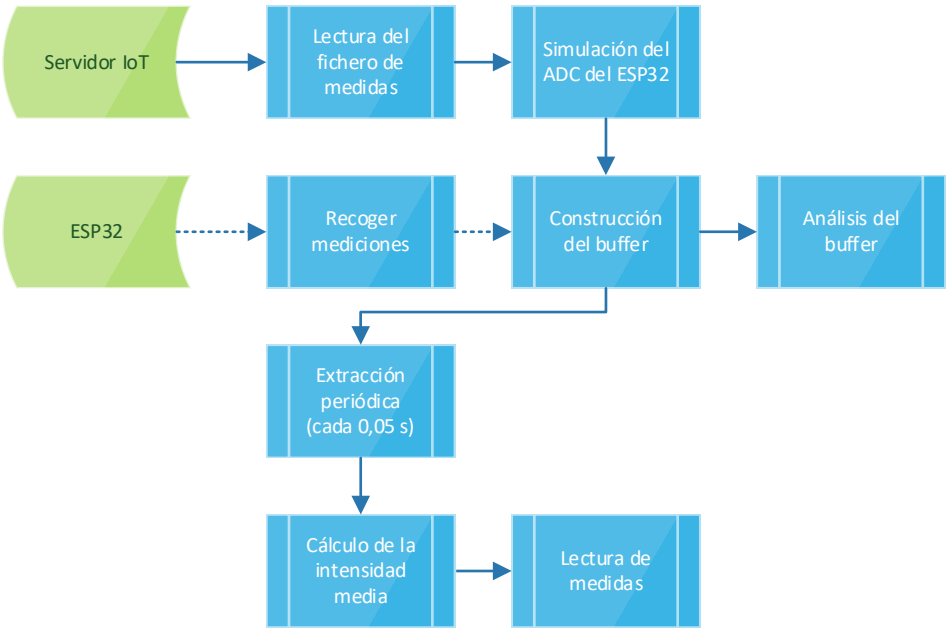


Figura 28. Diagrama de flujo de la extracción de medidas desde archivos de texto.

Por esta misma razón, se ha implementado un sistema de lectura de ficheros de texto con múltiples valores de medidas ya realizadas por el Departamento de Ingeniería Eléctrica de la Universitat Politècnica de València.

El formato elegido para el archivo de texto con las mediciones es el siguiente:

- Valores numéricos en formato hexadecimal de dos bytes (de 0 a FFFF).
- Cinco caracteres en total, siendo dos para cada byte y el salto de línea.
- Un valor por línea.

Este formato permite una mayor versatilidad y facilidad de interpretación, puesto que implica un número de caracteres fijo y un carácter de separación común.

En el flujo (Figura 29) podemos encontrar dos caminos distintos. El primero configura la aplicación web de visualización para definir los valores por defecto (la dirección de los ficheros de medidas y la fase a estudiar), conforme se explicará más adelante en.

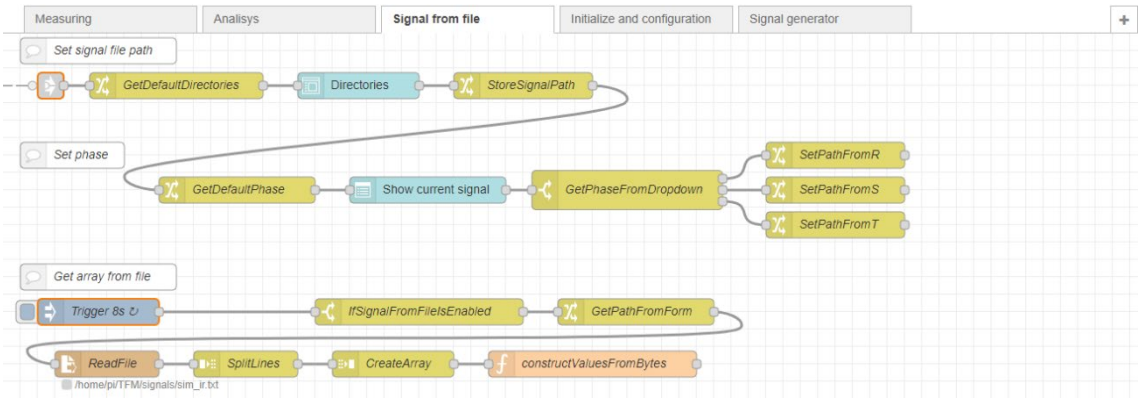


Figura 29. Flujo Signal from file, cuya función es extraer medidas desde un archivo de texto.

En cambio, el segundo funciona tal como se especifica en la Figura 28. Se ha especificado un bloque de inyección periódico, que envía un mensaje con la misma frecuencia que el ESP32 enviaría un buffer de bytes.

El bloque *IfSignalFromFileIsEnabled* comprueba que está activa la lectura desde fichero. Si es así, el bloque *GetPathFromForm* extrae el directorio del fichero a leer desde la aplicación web.

Este directorio es recogido por el bloque *ReadFile*, el cual convierte el fichero en un objeto JSON con tantos elementos como medidas. Para poder gestionar estos datos, es conveniente convertir este objeto en un array. Con este fin se han implementado los bloques *SpliteLines* y *CreateArray*, los cuales separan los elementos del objeto y los reagrupan en un *array* respectivamente.

Finalmente, se ha implementado el bloque función `constructValueFromBytes` (Signal from file), el cual es muy similar al implementado en 4.3.1 Inicialización y contiene la siguiente función:

En este caso también recorre todos los valores de dos en dos para agrupar los bytes mediante operaciones de desplazamiento de bits, pero en este caso, a cada segundo el buffer cambia, dependiendo del número de valores incluidos en el fichero de texto, en este caso 8 segundos.

Una vez creado el array es enviado a analizar (4.3.3 Análisis de señales) exactamente igual que si el buffer hubiera llegado desde el ESP32.

4.3.5. Generador de señales

Debido a que, por causas mayores, se ha dejado de disponer tanto del generador Analog Discovery como de su software WaveForm, se ha implementado un generador de señales básico. Este ha sido utilizado para diseñar la interfaz de usuario.

Este flujo básicamente genera, mediante funciones JavaScript, una señal que es enviada al flujo anterior para simular una medición del ADC del ESP32.

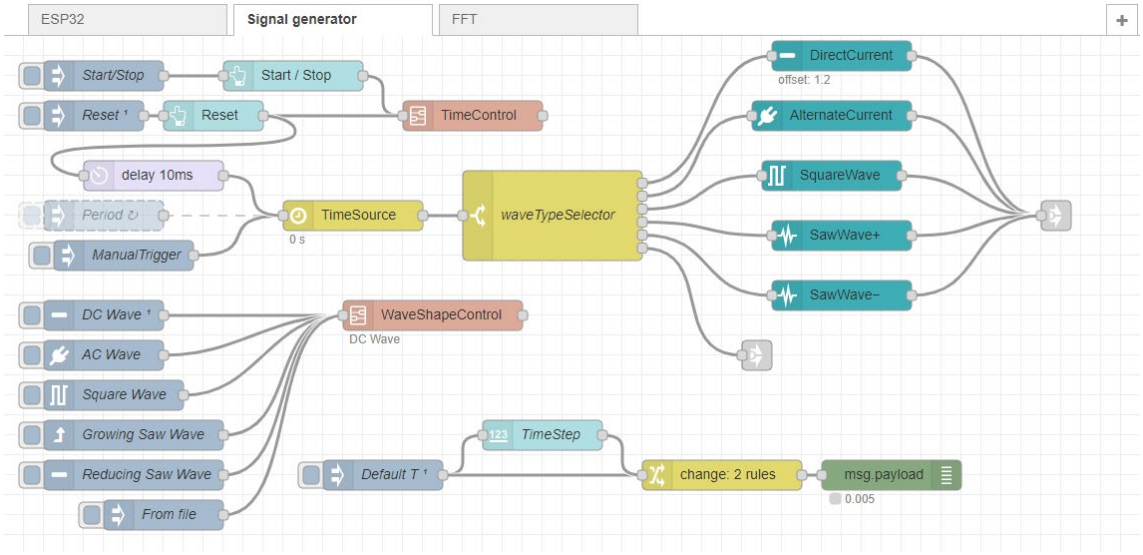


Figura 30. Flujo Signal generator dedicado a generar ondas sintéticas

Para tal efecto, se han creado cinco subflujos (***DirectCurrent***, ***AlternateCurrent***, ***SquareWave***, ***SawWave+*** y ***SawWave-***) con funciones de JavaScript, uno por cada tipo de onda. El modelo en

el que se ha basado este sistema ha sido el software WaveForms del que se ha hablado en el Capítulo 1 (Sistema de Control).

El sistema dispone de un botón “Start / Stop”, el cual hace que comience a correr el tiempo, y de un botón “Reset”, el cual reinicia el reloj a cero.

El efecto de tiempo se ha conseguido mediante una inyección periódica a una función JavaScript (`TimeSource`) que incrementa una variable global llamada `time`. Esta se toma como entrada para las funciones que generarán las ondas.

La selección del tipo de onda se lleva a cabo mediante un bloque `switch`, el cual dirige el valor actual del tiempo hacia la función escogida, dependiendo del tipo de onda indicado en `WaveShapeControl`.

Los parámetros de cada tipo de onda se pueden configurar mediante variables locales de cada subflujo como se puede observar en la Figura 31.

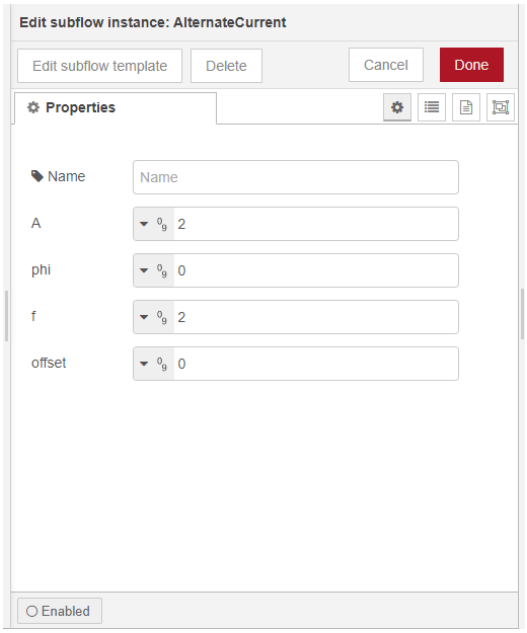


Figura 31. La modificación de la amplitud, desfase y frecuencia se efectúa mediante variables locales.

Cabe destacar que, a medida que el tiempo corre, el sistema se ralentiza de manera notable, de modo que este generador de señales únicamente se ha utilizado durante el desarrollo de la aplicación.

4.4. Aplicación web de monitorización (Front-end)

La aplicación web de monitorización debe ofrecer las mediciones del motor en tiempo real, así como la forma de onda y una gráfica con el análisis de Fourier realizado desde el servidor.

Además, proporciona varias opciones de desarrollo, como la elección de la fuente de la señal de entrada (ADC, ESP32 o servidor). También permite configurar las señales producidas por el servidor.

Se trata de una página HTML con controles basados en JavaScript en la que se muestran las medidas actuales y los resultados del análisis de la señal por FFT. Estos controles están proporcionados por Node-RED en su paquete *node-red-dashboard*. Los controles de *node-red-dashboard* se insertan en los flujos de igual manera que los bloques para poder interactuar con los datos calculados.

Esta aplicación web dispone de tres pestañas con visores y controles:

- **Monitoring:** Contiene todos los datos recogidos por el ESP32 y analizados por la aplicación IoT.
- **Signal Generator:** Contiene campos modificables para controlar el generador de señales (4.3.5 Generador de señales)
- **Debug:** Permite activar y desactivar las opciones de desarrollo.

En la Figura 32 se puede observar el flujo de información entre la aplicación web y los flujos de la aplicación IoT del servidor.

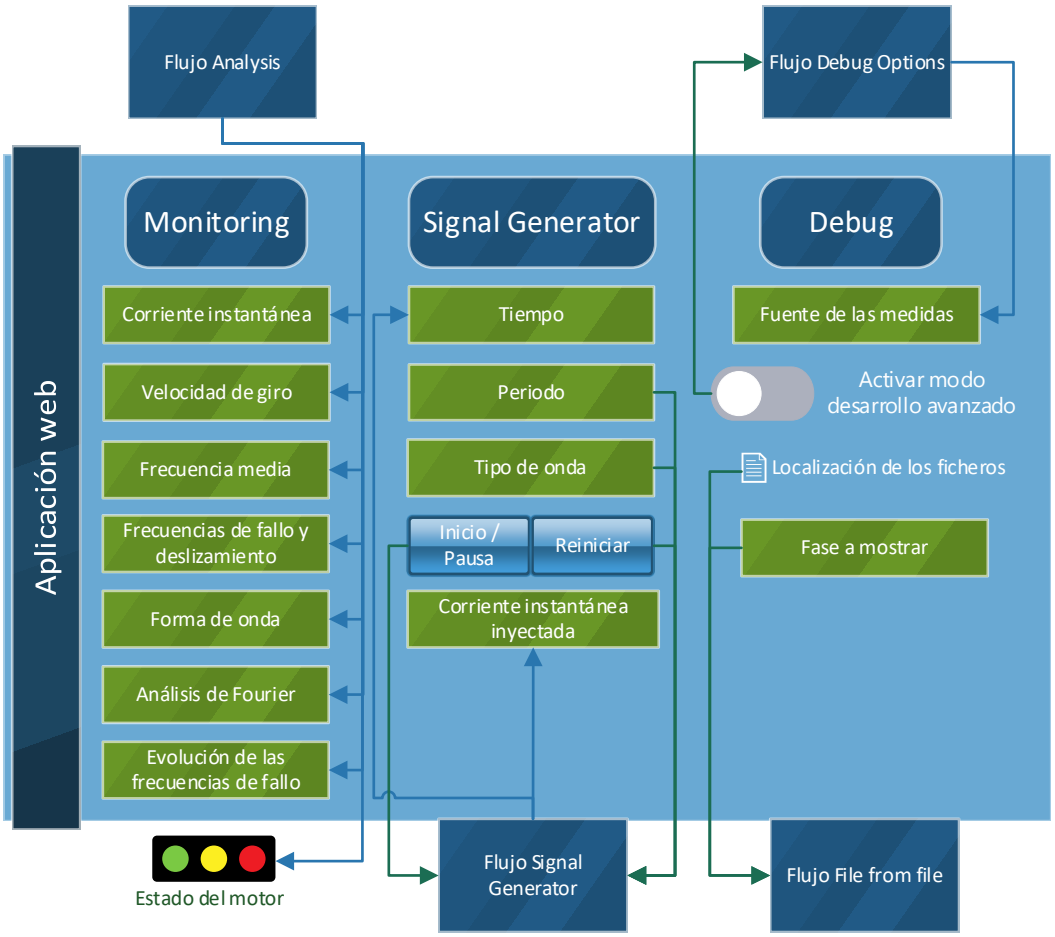


Figura 32. Diagrama de intercambio de información entre la aplicación web y los flujos de Node-RED

La aplicación realizada con *node-red-dashboard* tiene soporte nativo para dispositivos móviles, haciendo así su uso más sencillo y universal.

La aplicación web de monitorización se ha construido a partir de bloques en Node-RED (véase 4.1 Introducción a Node-RED), lo cual significa que los controles están repartidos entre todos los flujos de la aplicación IoT.

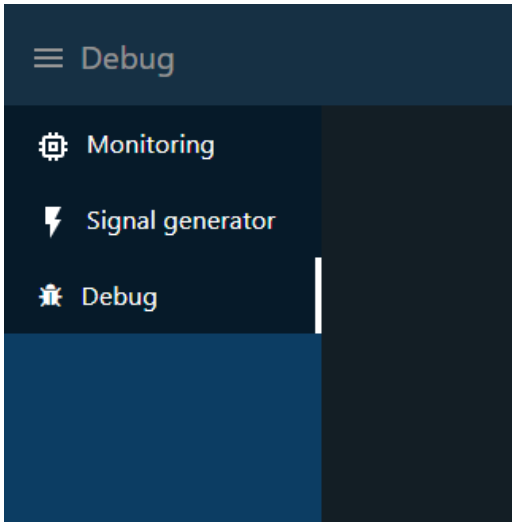


Figura 33. Barra lateral de navegación

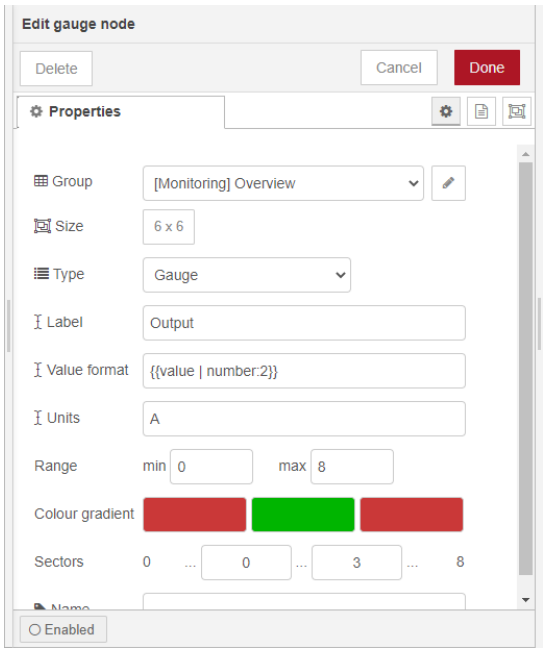


Figura 34. Construcción de la página de visualización

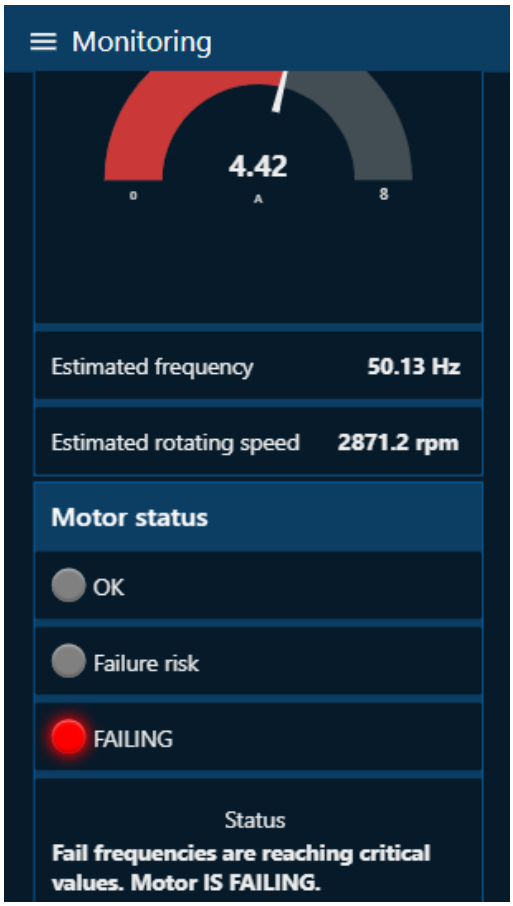


Figura 35. Aplicación web en la pestaña de monitorización en un dispositivo móvil

Estos bloques representan controles de interfaz de usuario tales como botones, cuadros de introducción de texto, etc. No obstante, también se dispone de elementos de visualización como, por ejemplo, el visor de intensidad con forma de semicírculo (configurado en la Figura 34), los campos de texto y los pilotos, los cuales se pueden observar en la Figura 35.

La navegación entre páginas se efectúa mediante la barra lateral principalmente, la cual se puede observar en la Figura 33.

Durante la etapa de 4.3.1 Inicialización, se debe evitar que el usuario ejecute ninguna acción, pues, al tratarse Node-RED de un sistema monohilo, el sistema puede colapsar si recibe varias órdenes al mismo tiempo ya que tratará de ejecutarlas a la vez aumentando así el consumo de recursos. Para ello, se ha implementado la pantalla de carga que se puede observar en la Figura 36. Esta pantalla evita que se ejecute ninguna acción hasta que el sistema no haya terminado de inicializarse. Finalmente, el botón “Continuar” dirige al usuario a la página de monitorización.

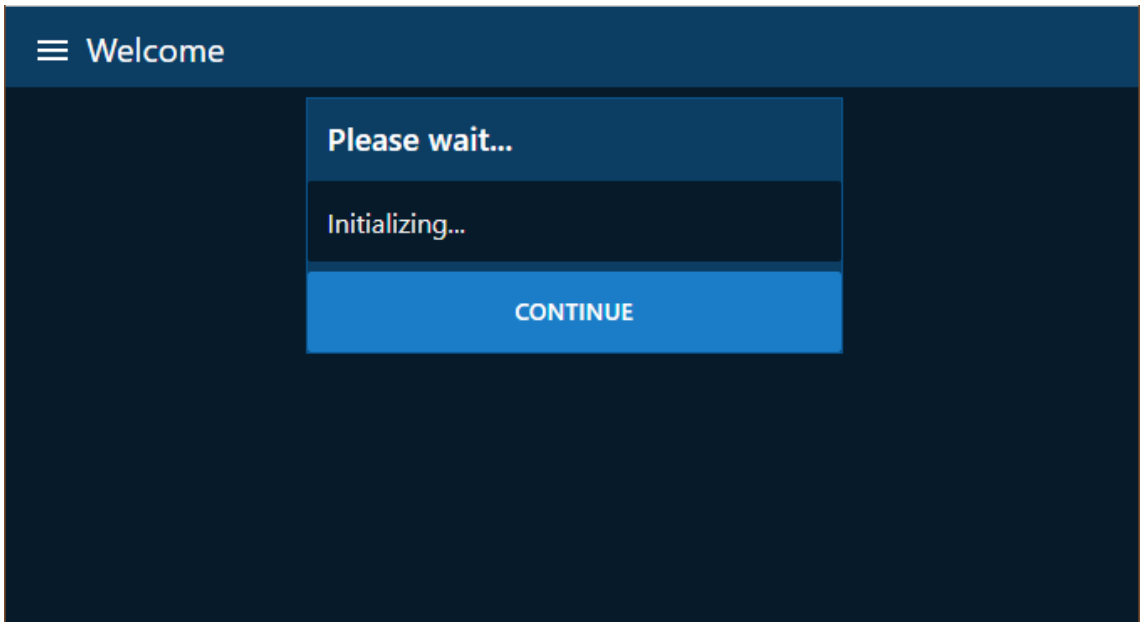


Figura 36. Pantalla de inicialización

En la página de monitorización (Figura 37), se encuentra, en la columna de la izquierda, la representación de la intensidad en tiempo real (con un margen de 8 segundos) en forma de semicírculo o *gauge*, así como la frecuencia y la velocidad de giro estimadas. La columna de la derecha, en cambio, se ha dedicado a mostrar el estado del motor, el cual se puede apreciar rápidamente observando el semáforo conformado por tres pilotos y una breve explicación del mismo. Adicionalmente, se localiza en esta misma columna el deslizamiento (*slip*) y las frecuencias de fallo calculadas a partir de este.

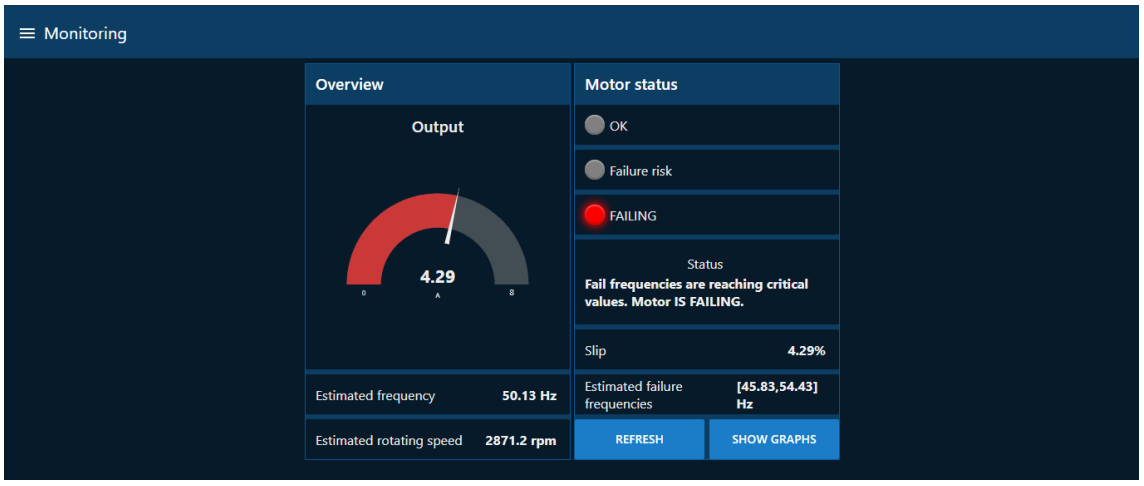


Figura 37. Página de monitorización

En la esquina inferior derecha de la página de monitorización se encuentran dos botones que permiten actualizar los cálculos (*Refresh*) y mostrar las gráficas (Figura 38) asociadas al análisis de la onda (*Show graphs*) respectivamente.

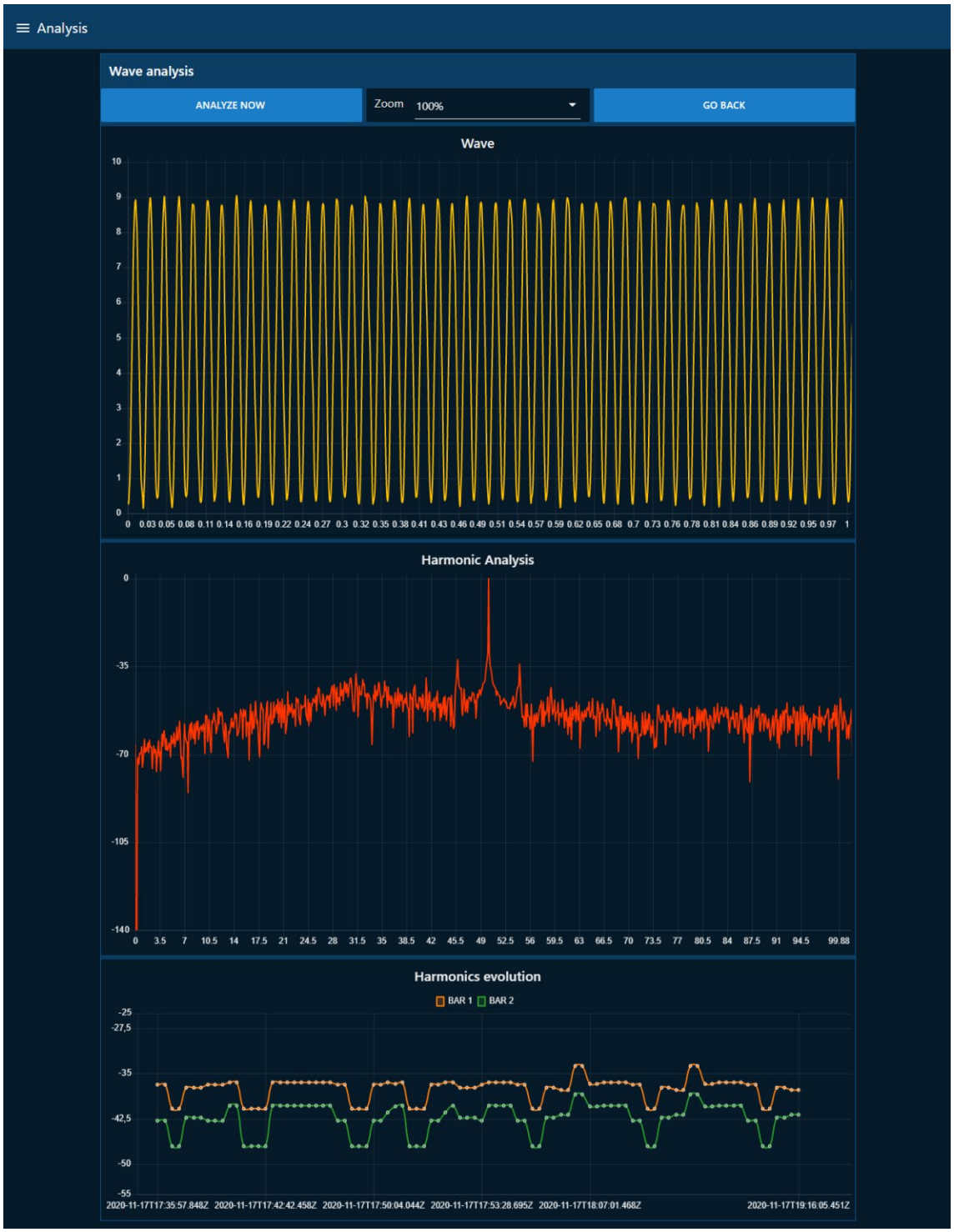


Figura 38. Resultado del análisis FFT en la aplicación web de monitorización

Al entrar en esta página se actualizan automáticamente las gráficas. No obstante, si se desea repetir el análisis, se puede optar por presionar el botón *Analyze now*, el cual desencadenará el análisis de nuevo. El botón *Go back*, devolverá al usuario a la página de monitorización.

En cuanto a las gráficas disponibles en esta página, se encuentran, en primer lugar, la forma de onda, la cual reproduce el valor de la intensidad durante un segundo, con posibilidad de ampliarla hasta un 800%. En segundo lugar, se encuentra el análisis del espectro mediante la

FFT, y, finalmente, una evolución histórica de las frecuencias de fallo localizadas, cuya duración es 90 puntos.

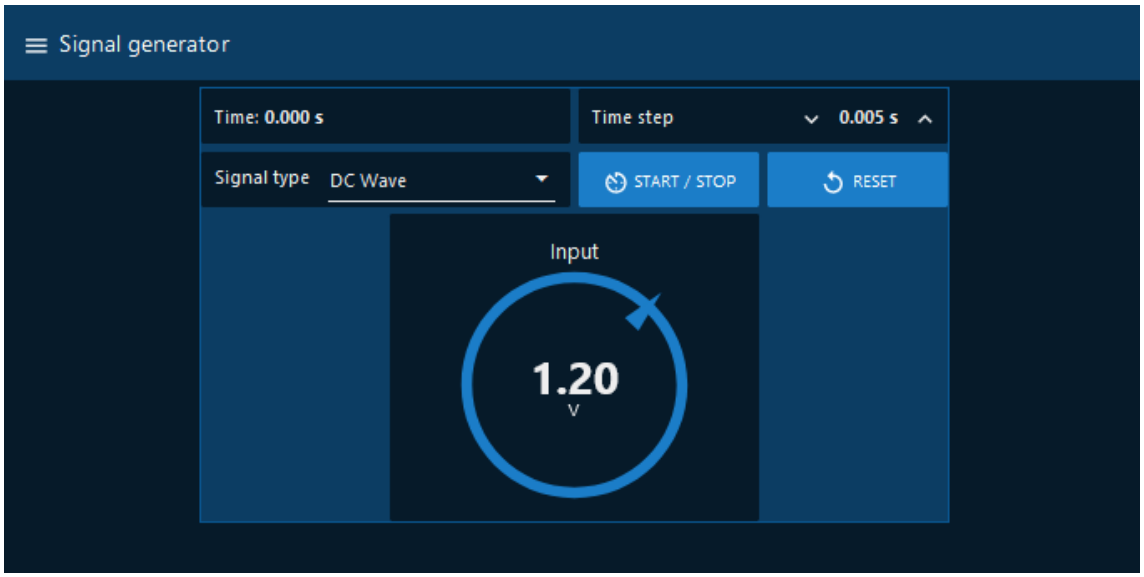


Figura 39. Aplicación web en la pestaña del generador de señales

Mediante la barra lateral, se puede acceder a otras páginas, las cuales en principio no estarán disponibles en un uso habitual, sino que solo podrán acceder los encargados de mantenimiento del sistema de monitorización para comprobar posibles averías de este.

Por ejemplo, la Figura 39 muestra el generador de señales descrito en 4.3.5 Generador de señales y en la Figura 40 se observan las opciones de desarrollo, como la fuente de las medidas, o los directorios de los ficheros de señales.

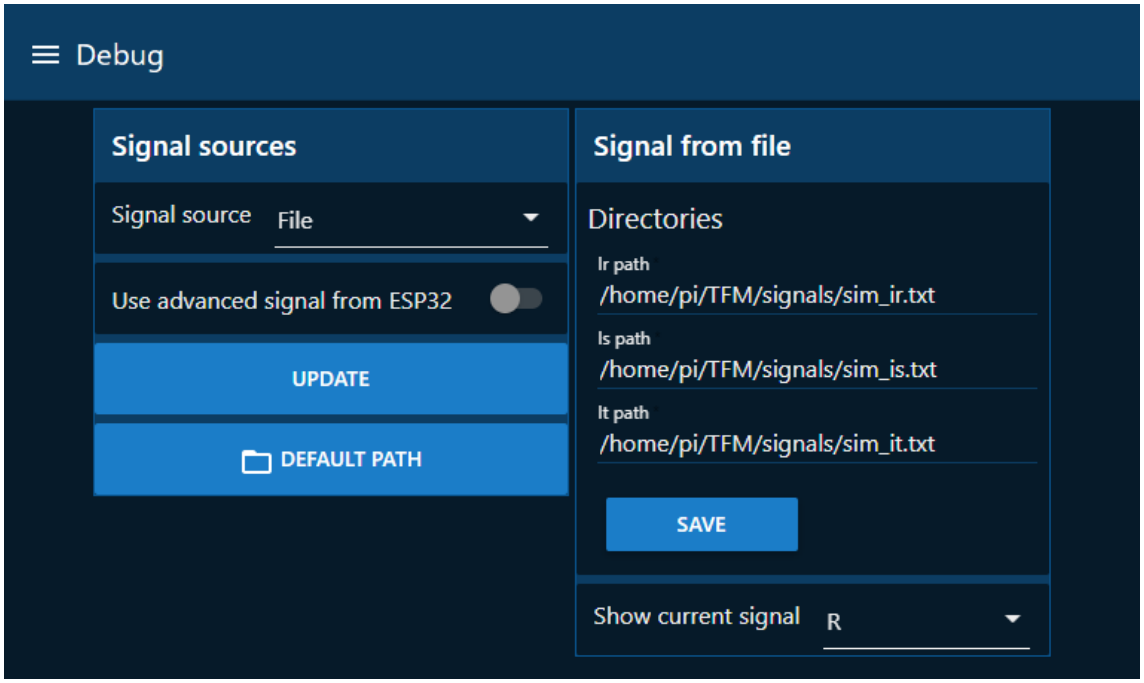


Figura 40. Aplicación web en la pestaña de opciones de desarrollo

Como valor adicional, se ha configurado la aplicación web para que lance un mensaje emergente en caso de fallo del motor, como puede verse en la Figura 41.

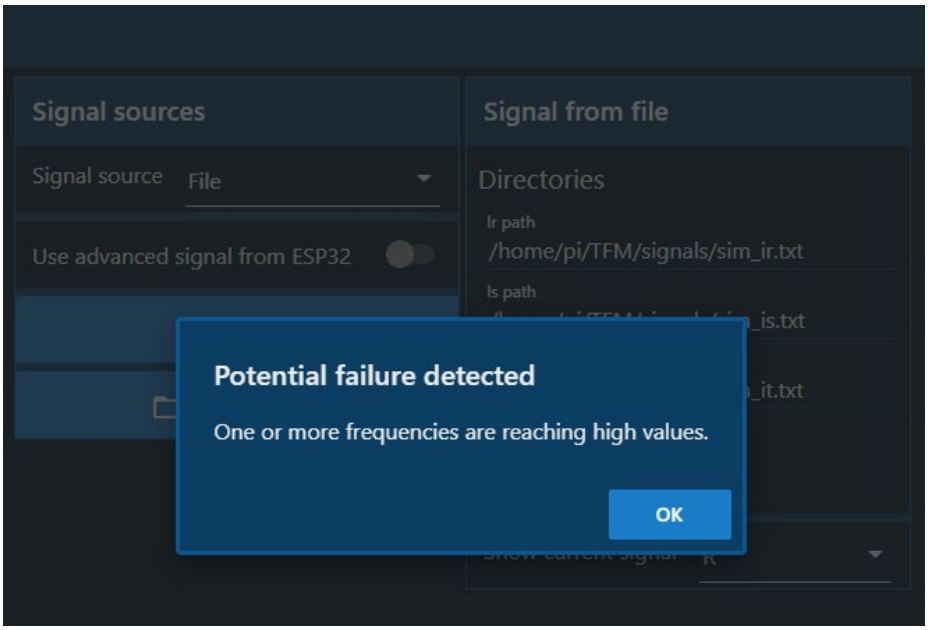


Figura 41. Mensaje de fallo del motor

Capítulo 5. Pruebas, ensayos y resultados

Debido a que, durante el trascurso del año 2020, debido a la crisis sanitaria producida por la pandemia del coronavirus SARS-CoV-2, se decretó el confinamiento a través de la declaración de Estado de Alarma por parte del Gobierno de España, ha resultado imposible realizar ensayos en motores reales.

Es por ello que en su lugar se han realizado diferentes pruebas y ensayos para tratar de sustituir la información que podría otorgar un ensayo en un motor real.

5.1. Prueba de la aplicación IoT mediante el generador de señales

Con el fin de demostrar el correcto funcionamiento de los componentes de la aplicación web se ha implementado un generador de señales, como se ha explicado en 4.3.5 Generador de señales.

El ensayo consiste en ejecutar una onda senoidal de amplitud 2 V y frecuencia 2 Hz. Se ha escogido una frecuencia muy baja ya que la alta carga que se genera en el procesador de la Raspberry Pi, que aloja la aplicación funcionando de servidor IoT, satura con facilidad el funcionamiento de la aplicación.

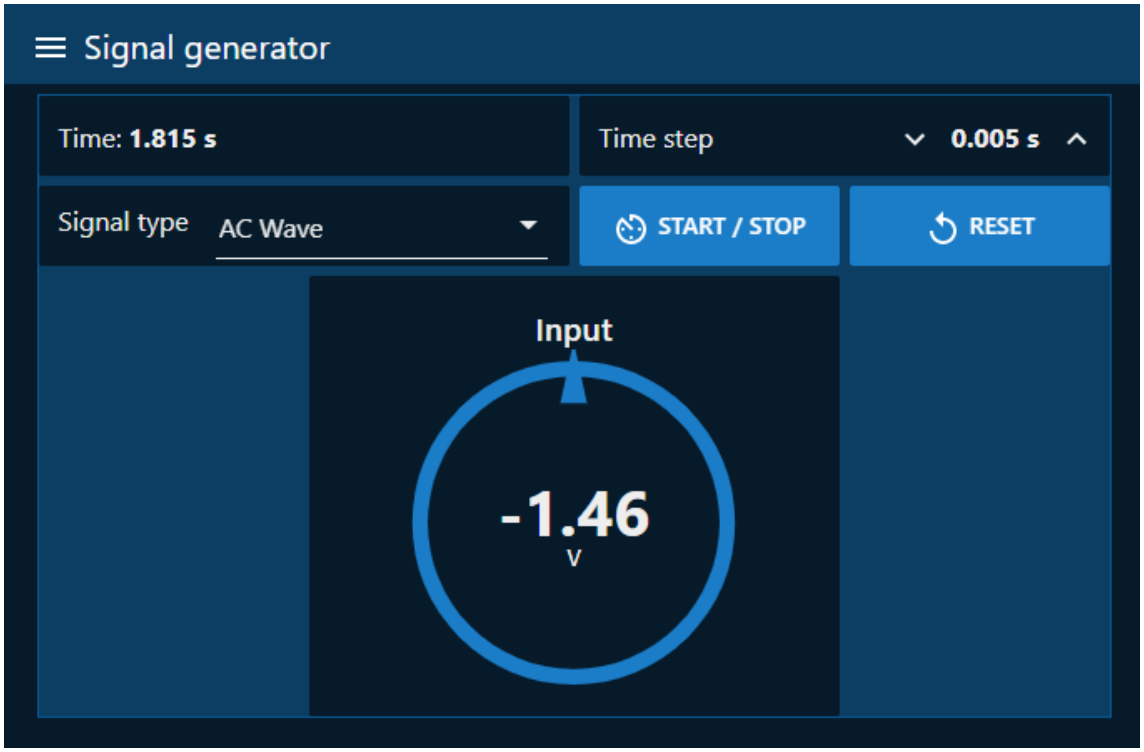


Figura 42. Captura de la aplicación web ejecutando la onda de prueba

En la Figura 42 se puede observar el avance del tiempo y la tensión instantánea aplicada desde el generador mientras que en la Figura 43 se muestra el valor de la tensión instantánea medida por la aplicación, simulando un mensaje desde el ESP32, en otro instante de tiempo posterior.

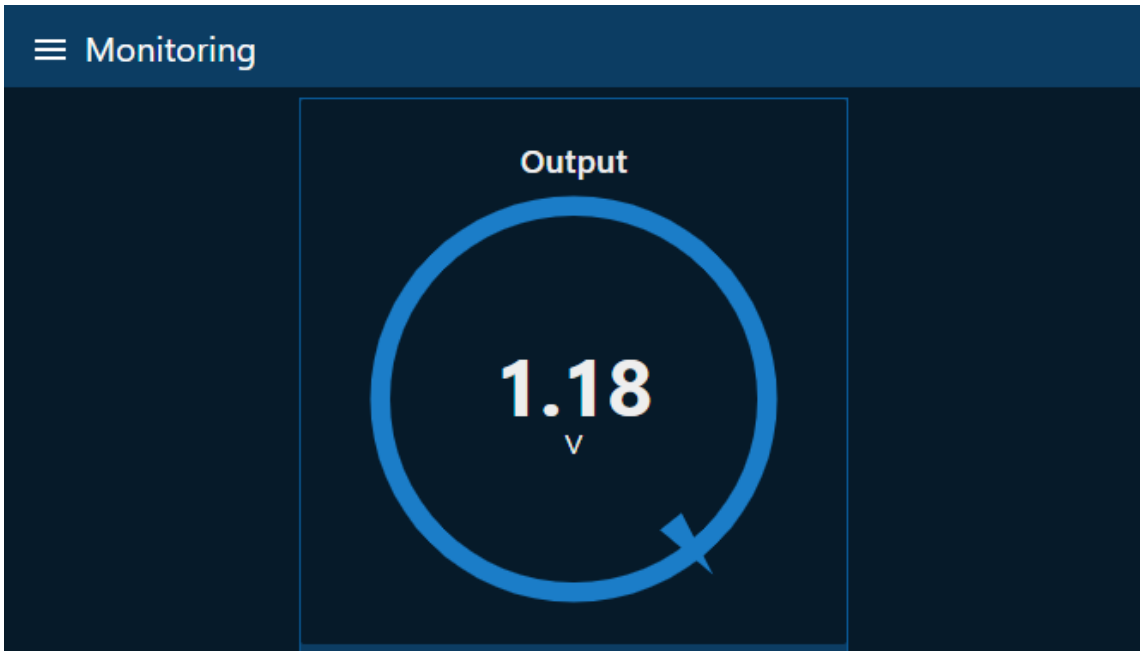


Figura 43. Captura de la aplicación web en su fase inicial durante la monitorización de la señal enviada desde el generador de señales

5.2. Prueba de la adquisición de datos mediante potenciómetro

Para comprobar el funcionamiento del envío de valores y del buffer desde el ADC del ESP32 se ha construido un potenciómetro y se ha conectado entre la tierra y la salida de corriente continua de 3,3 V.

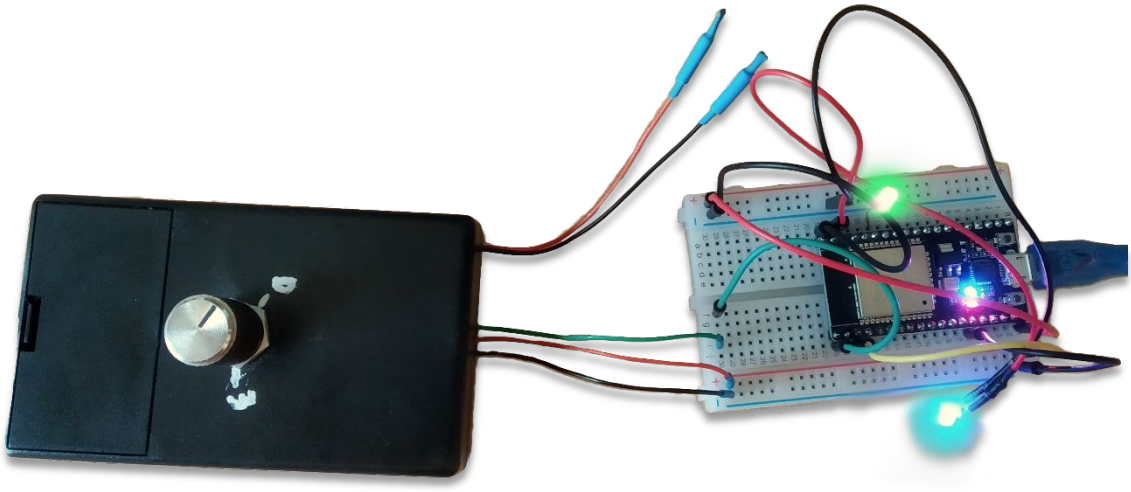


Figura 44. Montaje del ESP32 para probar la recepción de señales externas

El potenciómetro sirve para variar manualmente la tensión procedente de la salida de 3,3 V del ESP32 y comprobar que el ADC funciona, pues variando la tensión de entrada, se observa cómo cambia el valor mostrado por la aplicación web (Figura 45).

Cabe destacar que el LED azul parpadea por cada buffer enviado, tal como se estableció en **2.3.2 Recogida y envío de datos**.

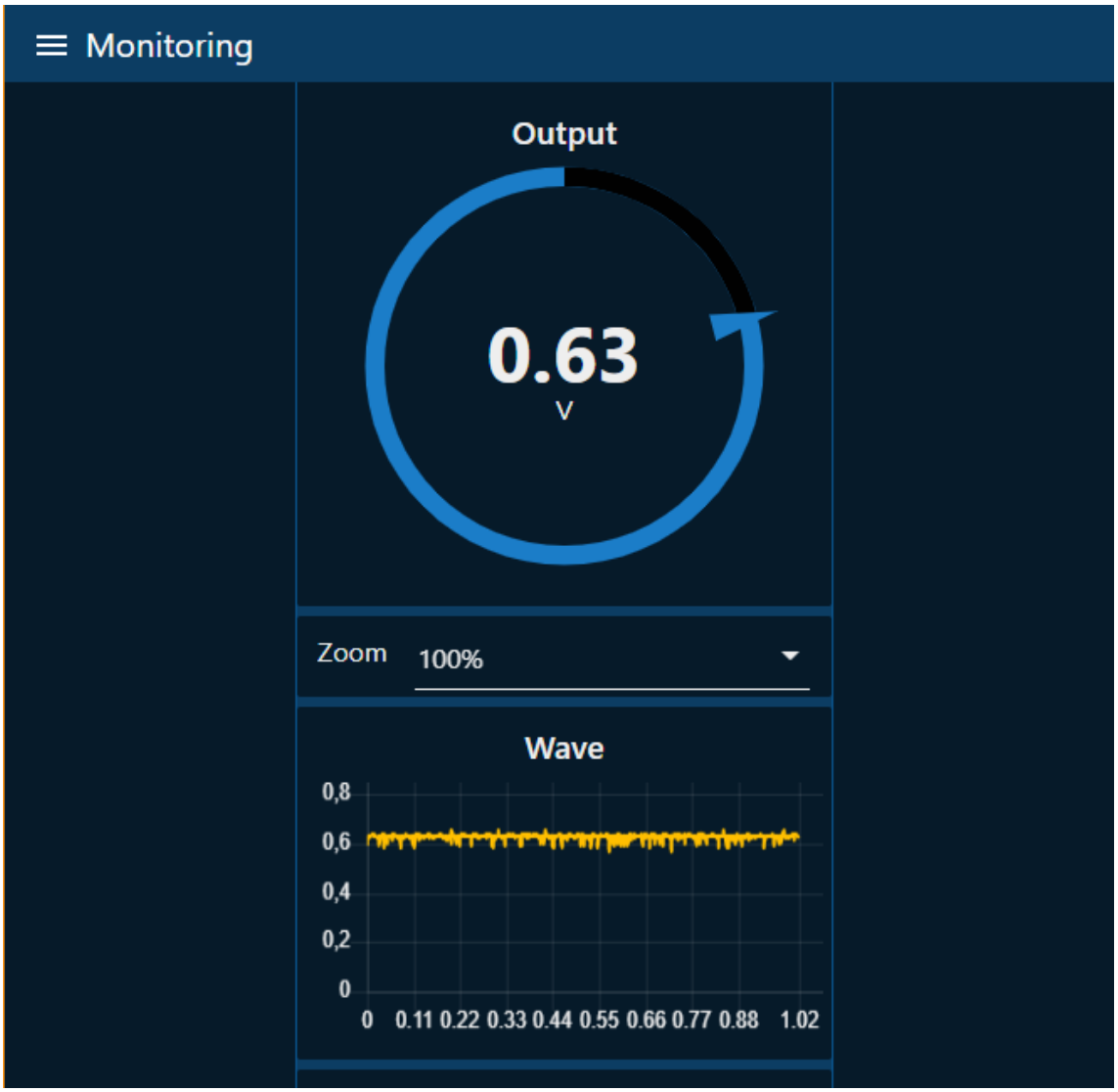


Figura 45. Captura de la prueba de la aplicación web en su fase inicial mediante el potenciómetro

5.3. Prueba de análisis mediante ondas generadas en el ESP32

Una vez demostrado el funcionamiento de la aplicación web, como del envío de señales desde el ESP32, es el momento de realizar la comprobación de la capacidad de análisis de la aplicación.

Para este fin, se ha implementado el código que permite generar ondas senoidales en el ESP32 y posteriormente enviarlas al servidor IoT para su análisis como si los datos se hubieran tomado mediante el ADC.

La función escogida para la prueba es una adaptación de la Ecuación 8, utilizada en el modo de desarrollo avanzado, como se puede observar en la Ecuación 14.

Ecuación 14. Función de la onda enviada al servidor IoT desde el ESP32.

$$f(t) = 1264 \cdot \cos(2 \cdot \pi \cdot 50 \cdot t) \cdot [1 + 0{,}01 \cdot \cos(2 \cdot 0{,}05 \cdot 2 \cdot \pi \cdot 50 \cdot t)]$$

Se ha utilizado una amplitud de 1264 ya que es el resultado de medir 2 V a través del ADC (véase Anexo 1).

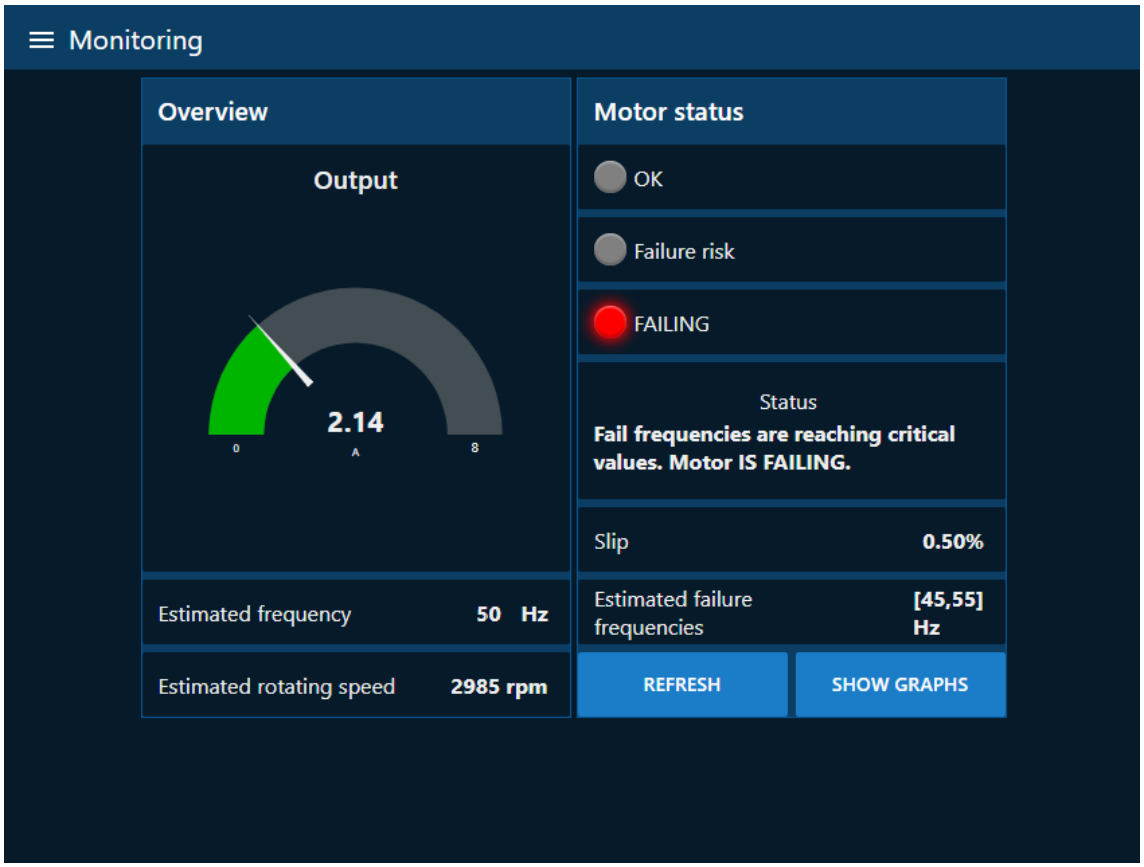


Figura 46. Resumen del estado del motor usando la onda generada por el ESP32

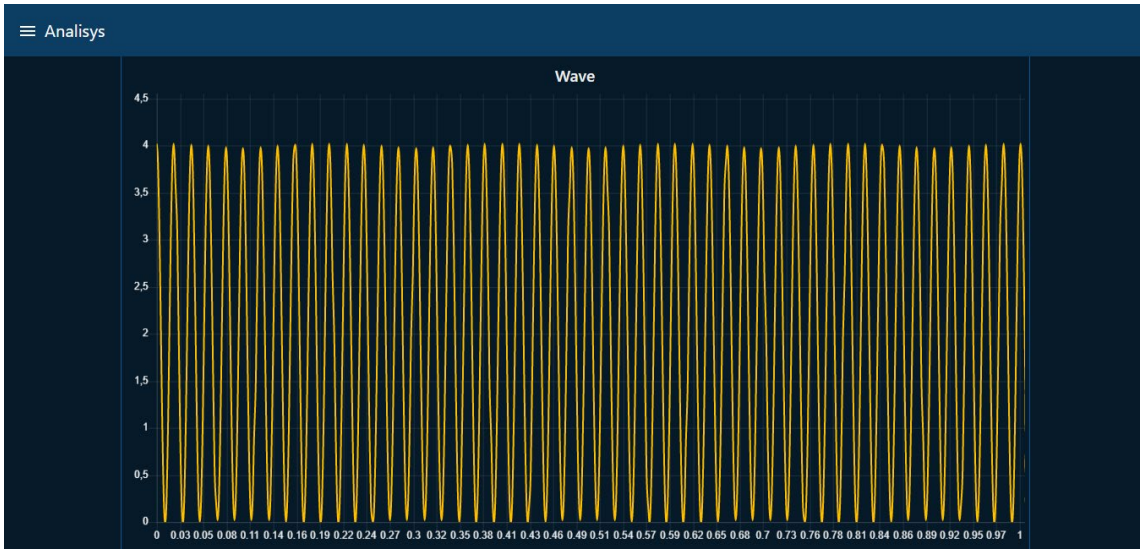


Figura 47. Captura del análisis de la onda generada por el ESP32

Tras recibirse y procesarse la señal enviada, se puede observar la forma de la onda en la Figura 47. A continuación, se presenta el análisis realizado mediante la FFT, como se puede ver en la Figura 48, el cual, como peculiaridad, presenta tres picos que serán analizados en 5.5 Interpretación de los resultados.

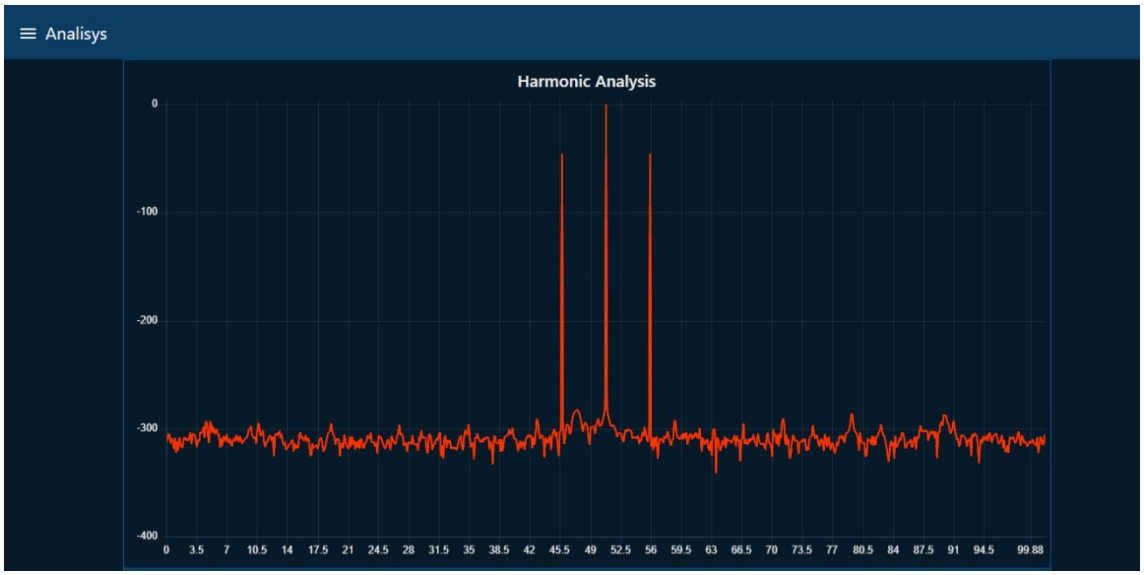


Figura 48. Análisis de FFT de la onda sintética generada por el ESP32

5.4. Ensayo de un motor real

Se tomó un ensayo realizado anteriormente por el Departamento de Ingeniería Eléctrica de la Universitat Politècnica de València registrado en un fichero de datos binarios de MATLAB con variables y matrices correspondientes a cada magnitud medida (ver **Figura 49.** Montaje del motor analizado en la UPV.).

Se cuenta con 100 segundos de datos tomados con un microsegundo de diferencia. Los datos se corresponden con la tensión e intensidad por fases y par mecánico respecto al tiempo. No obstante, únicamente se hará uso de la intensidad de la fase R (`ir`).

Como se ha explicado anteriormente en **4.3.4 Envío de señales desde un archivo**, a la hora de leer un archivo con medidas, se ha establecido un formato de valores hexadecimales de dos bytes separados por saltos de línea.

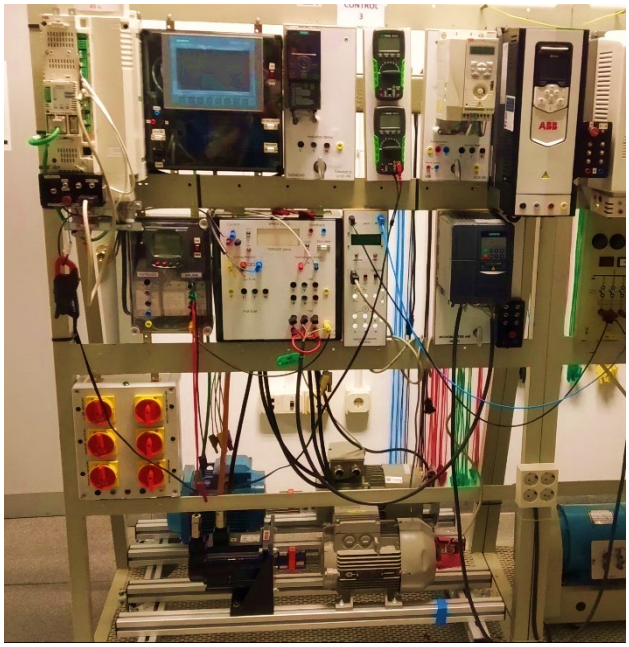


Figura 49. Montaje del motor analizado en la UPV.

Para poder aplicar este formato a los datos se ha creado la **Función de extracción de medidas desde archivo .m** (Anexo 3) en Matlab para exportar la información a un archivo de texto legible por la aplicación del servidor IoT.

Para poder usar la función anterior, se deberá volver a muestrear los valores a la frecuencia que utiliza el ESP32 para medir, es decir, 1 kHz.

A cada medida se le añade un offset para evitar valores negativos que puedan afectar al análisis por FFT. A continuación, es interpretada como si fuese recogida por el ADC del ESP32 mediante la función `signal2Adc(x)` (**Función para la simulación del ADC a partir de un valor medido**, Anexo 3).

Por último, escribe los valores en los ficheros correspondientes a cada fase en formato hexadecimal.

Cabe destacar que, puesto que la prueba se realizará en el servidor directamente, no es relevante que cumpla con los límites del ADC (16 bits en lugar de 12 bits).

Para acceder a los resultados de este ensayo, basta con iniciar la aplicación web de monitorización. Una vez en la página principal, la aplicación proporciona los siguientes datos:

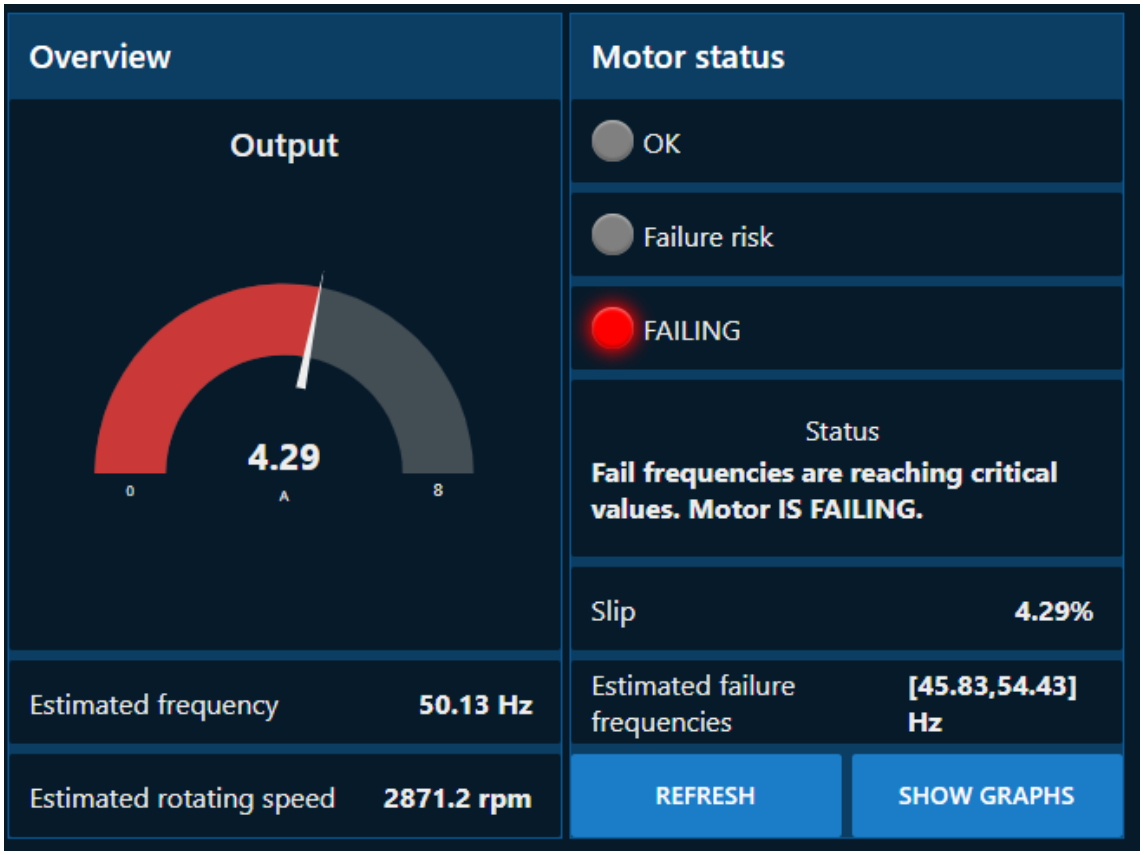


Figura 50. Vista principal de la aplicación web al introducir la señal del motor real.

Se puede observar cómo indica mediante un luminoso rojo que el sistema está fallando. Esta información también es posible recibirla mediante un vistazo rápido al ESP32, en el que, gracias a su programación, se ven replicados los luminosos de la Figura 50.

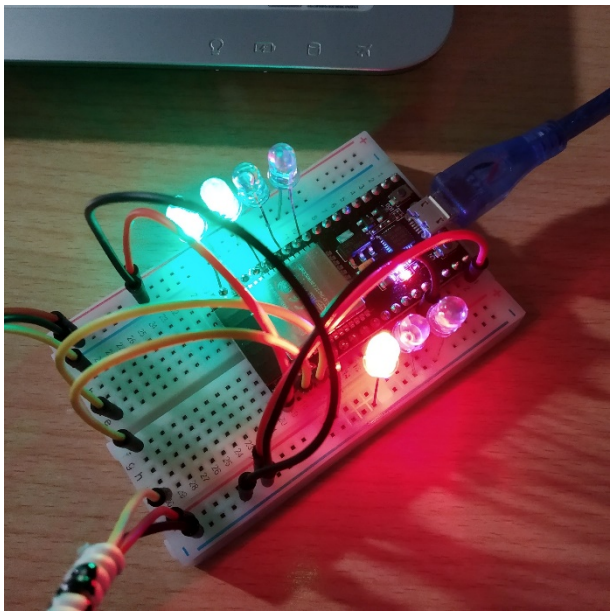


Figura 51. Indicación luminosa de "tipo semáforo" indicando el estado del motor.

Para adquirir mayor información acerca del estado, se ha pinchado sobre el botón “SHOW GRAPHS”, mediante el cual se abre la ventana “Analysis”.

Esta ventana contiene datos como la gráfica de forma de onda (Figura 52), el análisis de armónicos mediante la FFT (Figura 53) y el histórico de las frecuencias de fallo por rotura de las barras del rotor (Figura 54).

Figura 52. Ensayo de una señal de un motor real con forma de onda ampliada al 800%.

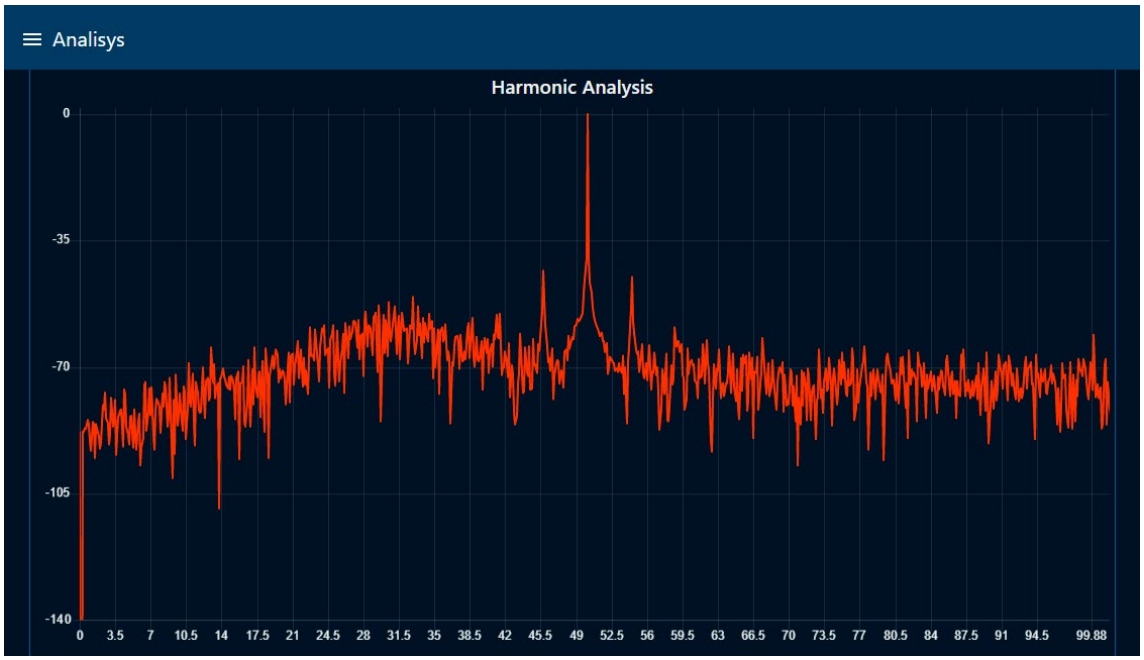


Figura 53. Análisis mediante FFT de una señal de un motor real.

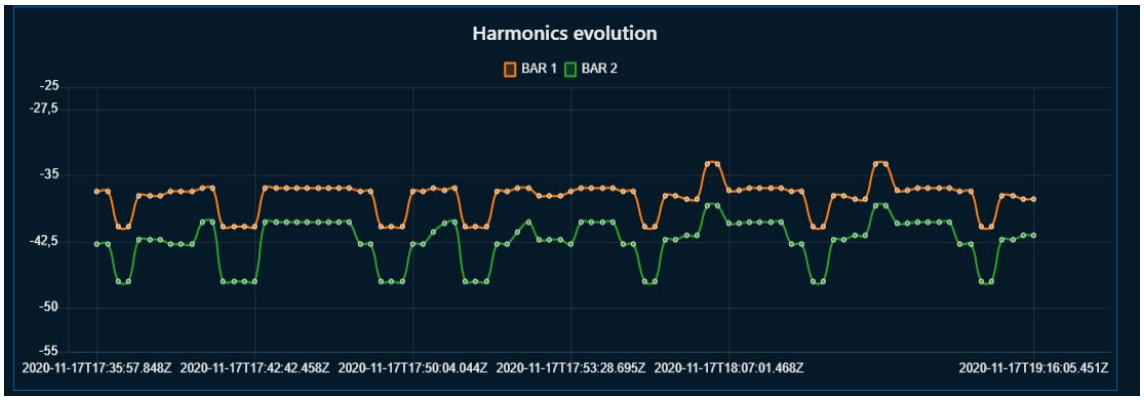


Figura 54. Evolución de las frecuencias de fallo en las últimas 90 medidas

5.5. Interpretación de los resultados

Durante el proceso de la prueba con el generador de señales, se ha tardado más de un minuto desde que se envía la acción de parado hasta que finalmente se detiene, mientras el generador sigue enviando valores de forma entrecortada.

Este aspecto es causado por la sobrecarga de información enviada, pues cabe recordar que Node-RED es un sistema monohilo, por lo que no puede ejecutar varias acciones simultáneas. Debido a las bajas prestaciones que ofrece una Raspberry Pi este efecto se agrava en gran medida. Con gran probabilidad, un servidor industrial dedicado a esta aplicación permitiría un funcionamiento más optimizado y preciso, no obstante, esto entra en la naturaleza de Node-RED.

Cabe destacar el buffer de medidas se recibe correctamente ya que, al tener una frecuencia de un envío por segundo, no se sobrecarga el limitado procesador de la Raspberry Pi.

En general, pese a las limitaciones causadas por el hardware y el software del servidor IoT, los resultados son correctos y cuadran con los resultados que se esperaban.

A continuación, se plantea un resumen de los resultados que se han obtenido.

- **Prueba de la aplicación IoT mediante el generador de señales:** El objetivo era comprobar el buen funcionamiento del generador de señales. Se ha comprobado que el funcionamiento es correcto, pues se genera una onda senoidal con la amplitud y frecuencia establecida. No obstante, como punto negativo, el rendimiento del sistema dificulta la experiencia, pues el sistema llega a colapsarse en varias ocasiones.
- **Prueba de la adquisición de datos mediante potenciómetro:** El objetivo era demostrar el correcto funcionamiento del sistema de envío y recepción de medidas. Se ha comprobado que la ejecución es correcta, ya que, al modificar la posición del potenciómetro, la lectura marca un valor de tensión entre 0 y 3 V.
- **Prueba de análisis mediante ondas generadas en el ESP32:** El objetivo era comprobar el correcto funcionamiento del sistema de análisis mediante FFT. El resultado es el esperado, pues, al recibir el buffer de las medidas, es capaz de mostrar una gráfica de la onda durante un segundo. Además, se realiza el análisis mediante FFT y se muestra en pantalla en su gráfica con tres picos: el central correspondiente a la frecuencia fundamental (50 Hz) y los laterales que corresponden a las frecuencias de fallo indicadas en la Figura 48, y, mediante este análisis, se indica también la frecuencia estimada (los 50 Hz estimados se corresponden con la frecuencia teórica).
- **Ensayo de un motor real:** El objetivo era demostrar la capacidad del sistema para analizar señales de motores reales. Pese a no disponer de motores reales, mediante ficheros de texto se ha creado una simulación. El resultado es el esperado, pues, al recibir el buffer de las medidas, es capaz de mostrar una gráfica de la onda durante un segundo. Además, se realiza el análisis mediante FFT y se muestra en pantalla en su gráfica, y, mediante este análisis, se indica también la frecuencia estimada (50,13 Hz estimados frente a los 50 Hz teóricos). Un detalle clave a tener en cuenta es que se consigue localizar las frecuencias de fallo (se calculan 45,83 Hz y 54,43 Hz frente a las frecuencias 45,75 Hz y 54,5 Hz que se detectan en los datos almacenados) y se advierte al usuario sobre el estado del motor tras localizarlas mediante el semáforo.

Capítulo 6. Conclusiones

Tras los ensayos realizados anteriormente, se pueden extraer varias conclusiones.

Se ha conseguido realizar correctamente medidas periódicas sobre la intensidad con una frecuencia de muestreo de 1 kHz y sobre la velocidad de giro de la máquina analizada a través de los sensores elegidos conectados al ESP32 dentro del rango especificado, adecuándose a así con los requisitos establecidos. Tras ello, el sistema proyectado ejecuta el análisis por FFT mediante el envío de la información recabada por los sensores con el fin de extraer las frecuencias de fallo asociadas a una rotura de barras y crear avisos al respecto mediante visualizadores numéricos, gráficas y luminosos en Node-RED. También se puede destacar la instalación de señales luminosas en forma de semáforo en el ESP32 a modo de alerta básica.

En cuanto a la comunicación entre el sistema de control (ESP32) y el sistema de monitorización (Node-RED) es funcional y universal. Esto permite que la aplicación web de monitorización esté disponible desde cualquier zona del mundo a través de Internet. Además, al estar basado en el lenguaje HTML es compatible con prácticamente la totalidad de navegadores web.

En lo relativo a la escritura de la base de datos JSON, el retardo causado por esta es prácticamente imperceptible y permite localizar los puntos de cada serie de datos para poder ser representado en una gráfica a posteriori.

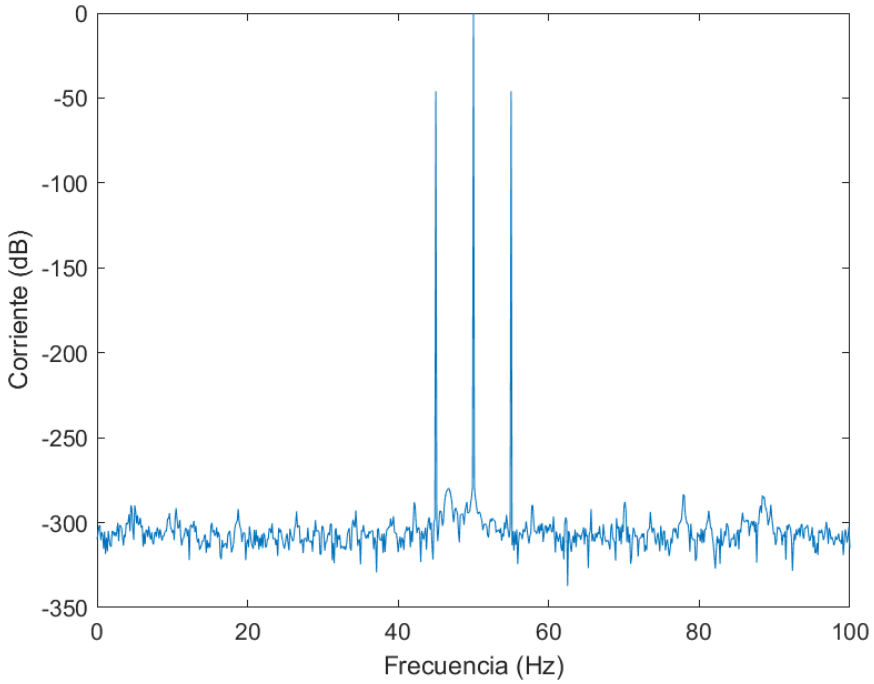


Figura 55. Análisis de la onda sintética mostrada en la Figura 48 utilizando MATLAB

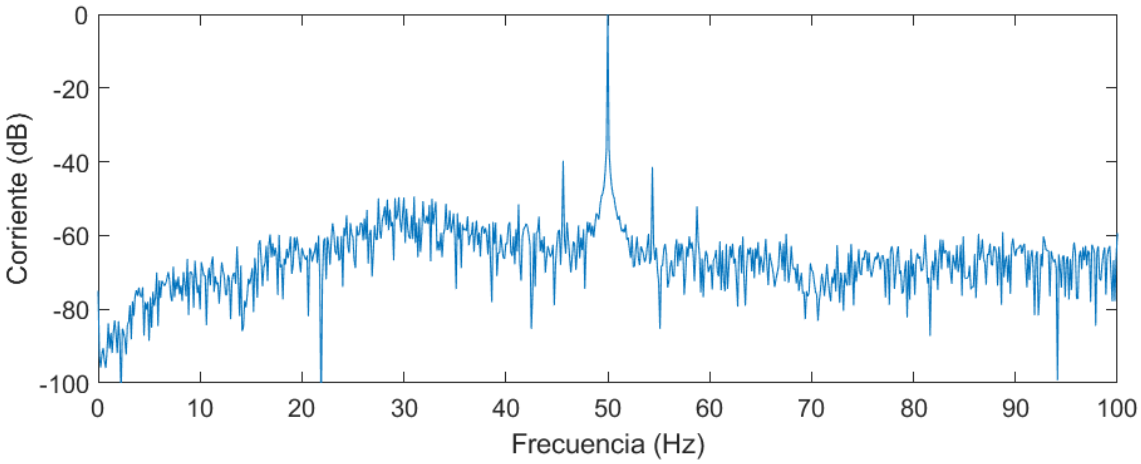


Figura 56. Análisis mediante FFT de la misma señal que la Figura 53 ejecutado en MATLAB.

Por otro lado, quedan patentes las capacidades del sistema para recoger y analizar medidas en un motor real, como se puede observar en el apartado anterior, pues al representar la FFT de la misma onda empleada tanto en **5.3 Prueba de análisis mediante ondas generadas en el ESP32** como en **5.4 Ensayo de un motor real** en MATLAB, se ha obtenido el mismo patrón de fallo (Figura 55 y Figura 56).

De esta manera, queda patente que se cumplen los **Objetivos** (1.3) y los **Requisitos y especificaciones** establecidos (1.4)

Bibliografía

[1] R.Saidur, “A review on electrical motors energy use and energy savings,” *Renewable and Sustainable Energy Reviews,* vol. 14, no. 3, p. 879, April 2010.

[2] M. A. Alsaedi, “Fault diagnosis of three-phase induction motor: A review,” *Optics,* vol. 4, no. 1-1, pp. 1-8, 2015.

[3] D. Page, «Así se construye en España un parque eólico en lo más alto de una montaña,» *El Independiente,* 10 10 2020.

[4] M. Gracia, «Deloitte,» [En línea]. Available: https://www2.deloitte.com/es/es/pages/technology/articles/IoT-internet-of-things.html. [Último acceso: 18 Agosto 2020].

[5] K. M. Siddiqui, K. Sahay and V. K. Giri, "Health Monitoring and Fault Diagnosis in Induction Motor - A Review," *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering,* vol. 3, no. 1, pp. 6549-6565, 2014.

[6] S. Nandi, H. A. Toliyat y X. Li, «Condition Monitoring and Fault Diagnosis of Electrical Motors - A Review,» *IEEE Transactions on Energy Conversion,* vol. 20, nº 4, pp. 719-729, 2005.

[7] N. Mehla and R. Dahiya, "An Approach of Condition Monitoring of Induction Motor using MCSA," *International Journal of Systems Applications, Enineering & Development,* vol. 1, no. 1, pp. 13-17, 2007.

[8] «Area Tecnología,» [En línea]. Available: https://areatecnologia.com/electricidad/transformador-de-corriente.html. [Último acceso: 6 Noviembre 2020].

[9] Mozilla Developer Network, «Generalidades del protocolo HTTP,» 7 Agosto 2020. [En línea]. Available: https://developer.mozilla.org/es/docs/Web/HTTP/Overview. [Último acceso: 29 Octubre 2020].

[10] LEM USA Inc., «Datasheet Current Transducer LTSR 6-NP,» [En línea]. Available: https://www.lem.com/sites/default/files/products_datasheets/ltsr_6-np.pdf. [Último acceso: 6 Noviembre 2020].

[11] Random Nerd Tutorials, «Random Nerd Tutorials,» [En línea]. Available: https://randomnerdtutorials.com/solved-failed-to-connect-to-esp32-timed-out-waiting-for-packet-header/.

[12] Digilent, "Digilent Store," [Online]. Available: https://store.digilentinc.com/analog-discovery-100msps-usb-oscilloscope-logic-analyzer-limited-time/. [Accessed 2020 Mayo 10].

[13] Á. H. Albarracín, «BorrowBits,» Abril 2020. [En línea]. [Último acceso: 9 Noviembre 2020].

[14] Node-RED, «GitHub Inc.,» [En línea]. Available: https://github.com/node-red/node-red. [Último acceso: 9 Noviembre 2020].

[15] L. (echel0n), «App Code Labs,» 7 Enero 2019. [En línea]. Available: https://appcodelabs.com/introduction-to-iot-build-an-mqtt-server-using-raspberry-pi. [Último acceso: 11 Noviembre 2020].

[16] Prometec, «Prometec,» [En línea]. Available: https://www.prometec.net/instalando-esp32/.

[17] R. Santos, "Random Nerd Tutorials," [Online]. Available: https://randomnerdtutorials.com/esp32-mqtt-publish-subscribe-arduino-ide/. [Accessed 20 Abril 2020].

[18] FreeRTOS, «FreeRTOS,» [En línea]. Available: https://www.freertos.org/. [Último acceso: 23 Marzo 2020].

[19] Me-No-Dev, "GitHub," 15 Jan 2019. [Online]. Available: https://github.com/me-no-dev/arduino-esp32fs-plugin. [Accessed 3 Ago 2020].

[20] V. Cerf, "IETF," 16 Octubre 1969. [Online]. Available: https://tools.ietf.org/html/rfc20. [Accessed 26 Agosto 2020].

[21] Chart.js, «Line - Chart.js documentation,» [En línea]. Available: https://www.chartjs.org/docs/latest/charts/line.html.

[22] "JSONata Documentation," [Online]. Available: http://docs.jsonata.org/overview.html.

[23] lrmoreno007, «esp8266-arduino-spanish,» [En línea]. Available: https://esp8266-arduino-spanish.readthedocs.io/es/latest/installing.html.

[24] Google, «Material Design,» [En línea]. Available: https://material.io/design. [Último acceso: 30 Agosto 2020].

APLICACIÓN IOT PARA EL DIAGNÓSTICO DE MOTORES DE INDUCCIÓN

PRESUPUESTO

Autor:
Benjamín Cháfer Ferrando

Tutores:
Manuel Pineda Sánchez
Ángel Sapena Bañó

Presupuesto

1. Introducción

El siguiente documento abarca el desglose y los detalles del coste de ejecución del presente proyecto.

Con el fin de localizar y diversificar los costes, se ha dividido la actuación en cinco etapas:

1. Montaje y construcción del sensor ESP32
2. Montaje e instalación del servidor IoT
3. Diseño y desarrollo del software para el ESP32
4. Diseño y desarrollo del software para el servidor IoT
5. Realización de ensayos y pruebas

2. Mano de obra

A continuación, se retratan los costes generados por la contratación de un ingeniero industrial para la realización del proyecto.

Para calcular estos costes, se ha partido del salario bruto mensual indicado en el convenio colectivo de trabajo para la Industria, la Tecnología y los Servicios del sector Metal de la provincia de Valencia para el año 2019.

Tabla 4. Detalle del coste del ingeniero industrial.

Ingeniero Industrial	
Concepto	**Coste**
Sueldo base anual	22.689,84 €/año
Pagas extra	5.037,20 €/año
Pluses	1.475,88 €/año
Seguridad social (aprox. 35%)	7.941,44 €/año
Sueldo total año	**37.144,37 €/año**
A facturar	
Sueldo por día	146,82 €/día
Sueldo por hora	18,35 €/hora

Tabla 5. Cuadro de precios de la mano de obra.

Etapa	**Horas**	**Coste**
Selección de equipos	10	18,35 €
Valoración de alternativas	10	18,35 €
Montaje y construcción del sensor ESP32	2	36,70 €
Montaje e instalación del servidor IoT	2	36,70 €
Diseño y desarrollo del software para el ESP32	360	6.606,71 €
Diseño y desarrollo del software para el servidor IoT	360	6.606,71 €
Realización de ensayos y pruebas	26	477,15 €
TOTAL	**750**	**13.763,97 €**

3. Materiales

Tabla 6. Cuadro de precios de los materiales necesarios.

Producto	Cantidad	Precio unitario	Precio total
Placa ESP32 Devkit Wifi + Bluetooth 4MB IoT	1	6,16 €	6,16 €
Pack Cables Dupont Para Puentear En Placa De Prototipo	5	0,03 €	0,16 €
Raspberry Pi 4 - Modelo B - 2gb	1	33,02 €	33,02 €
Diodo LED 3mm (3V~6V)	7	0,91 €	6,37 €
Mini Cable USB A Micro USB 50cm Para Arduino Nano Micro Node-MCU	1	1,57 €	1,57 €
Placa De Prototipos De 400 Puntos Blanca	1	2,44 €	2,44 €
Placa De Prototipos De 170 Puntos Naranja	1	0,91 €	0,91 €
Argon One - Caja Sobremesa Para Raspberry Pi 4	1	23,93 €	23,93 €
Raspberry Alimentación USB-C Negro	1	10,99 €	10,99 €
Rampow Cable USB Tipo C - Cable USB C A USB 3.0 Carga Rápida Y Sincronización	1	5,78 €	5,78 €
Sensor de efecto Hall LEM LTSR 6-NP	1	13,78 €	13,78 €
		TOTAL (SIN IVA)	**91,79 €**

4. Presupuesto descompuesto

Tabla 7. Presupuesto descompuesto del planteamiento del proyecto

Capítulo 1: Planteamiento del proyecto				
Unidad	**Descripción**	**Rendimiento**	**Precio**	**Importe**
uds.	**Selección de equipos**			
h	Ingeniero industrial	10	18,35	183,52 €
uds.	**Valoración de alternativas**			
h	Ingeniero industrial	10	18,35	183,52 €
%	Costes directos complementarios	0,01	183,52	1,84 €
	Costes directos			368,87 €
	Costes indirectos	0,02	368,87	7,38 €
	COSTE TOTAL			**376,25 €**

Tabla 8. Presupuesto descompuesto de la construcción del sensor.

Capítulo 2: Construcción del sensor				
Unidad	**Descripción**	**Rendimiento**	**Precio**	**Importe**
uds.	**Montaje y construcción del sensor ESP32**			
h	Ingeniero industrial	2	18,35	36,70 €
uds.	Placa Esp32 Devkit Wifi + Bluetooth 4mb IoT	1	6,16	6,16 €
uds.	Placa De Prototipos De 400 Puntos Blanca	1	2,44	2,44 €
uds.	Pack Cables Dupont Para Puentear En Placa De Prototipo	12	0,03	0,37 €
uds.	Diodo Led 3mm (3v~6v)	7	0,91	6,37 €
uds.	Mini Cable USB A Micro USB 50cm Para Arduino Nano Micro Node-MCU	1	1,57	1,57 €
uds.	Placa de prototipos de 170 puntos naranja	1	0,91	0,91 €
uds.	Sensor de efecto Hall LEM LTSR 6-NP	1	13,78	13,78 €
%	Costes directos complementarios	0,01	68,30	0,68 €
	Costes directos			68,99 €
	Costes indirectos	0,02	68,99	1,38 €
	COSTE TOTAL			**70,37 €**

Tabla 9. Presupuesto descompuesto de la configuración del servidor IoT.

Capítulo 3: Configuración del servidor IoT				
Unidad	**Descripción**	**Rendimiento**	**Precio**	**Importe**
uds.	**Montaje e instalación del servidor IoT**			
h	Ingeniero industrial	2	18,35	36,70 €
uds.	Raspberry Pi 4 - Modelo B - 2GB	1	33,02	33,02 €
uds.	Argon One - Caja Sobremesa Para Raspberry Pi 4	1	23,93	23,93 €
uds.	Rampow Cable USB Tipo C - Cable USB C A USB 3.0 Carga Rápida Y Sincronización	1	5,78	5,78 €
uds.	Raspberry Alimentación USB-C Negro	1	10,99	10,99 €
%	Costes directos complementarios	0,01	110,42	1,10 €
	Costes directos			111,53 €
	Costes indirectos	0,02	111,53	2,23 €
	COSTE TOTAL			**113,76 €**

Tabla 10. Presupuesto descompuesto del diseño y la programación del software del ESP32.

Capítulo 4: Diseño y programación del software del ESP32				
Unidad	**Descripción**	**Rendimiento**	**Precio**	**Importe**
uds.	**Diseño y desarrollo del software para el ESP32**			
h	Ingeniero industrial	360	18,35	6.606,71 €
%	Costes directos complementarios	0,01	6606,71	66,07 €
	Costes directos			6.672,77 €
	Costes indirectos	0,02	6672,77	133,46 €
	COSTE TOTAL			**6.806,23 €**

Tabla 11. Presupuesto descompuesto del diseño y la programación de los flujos del servidor IoT.

Capítulo 5: Diseño y programación de los flujos del servidor IoT				
Unidad	**Descripción**	**Rendimiento**	**Precio**	**Importe**
uds.	**Diseño y desarrollo del software para el servidor IoT**			
h	Ingeniero industrial	360	18,35	6.606,71 €
%	Costes directos complementarios	0,01	6606,71	66,07 €
	Costes directos			6.672,77 €
	Costes indirectos	0,02	6672,77	133,46 €
	COSTE TOTAL			**6.806,23 €**

Tabla 12. Presupuesto descompuesto de la realizaciónd ensayos y tests.

Capítulo 6: Ensayos y tests				
Unidad	**Descripción**	**Rendimiento**	**Precio**	**Importe**
uds.	**Realización de ensayos y tests**			
h	Ingeniero industrial	26	18,35	477,15 €
%	Costes directos complementarios	0,01	477,15	4,77 €
	Costes directos			481,92 €
	Costes indirectos	0,02	481,92	9,64 €
	COSTE TOTAL			**491,56 €**

5. Presupuesto de ejecución material, presupuesto de inversión y presupuesto base de licitación

Tabla 13. Presupuesto de ejecución material, presupuesto de inversión y presupuesto base de inversión.

RESUMEN	IMPORTE
Capítulo 1: Planteamiento del proyecto	376,25 €
Capítulo 2: Construcción del sensor	70,37 €
Capítulo 3: Configuración del servidor IoT	113,76 €
Capítulo 4: Diseño y programación del software del ESP32	6.806,23 €
Capítulo 5: Diseño y programación de los flujos del servidor IoT	6.806,23 €
Capítulo 6: Ensayos y tests	491,56 €
TOTAL PRESUPUESTO DE EJECUCIÓN MATERIAL	**14.288,14 €**
15% Gastos generales	2.143,22 €
6% Beneficio industrial	857,29 €
TOTAL PRESUPUESTO DE INVERSIÓN	**17.288,65 €**
21% IVA	3.630,62 €
TOTAL PRESUPUESTO BASE DE LICITACIÓN	**20.919,27 €**

El presupuesto de ejecución material asciende a la expresada cantidad de EUROS:
CATORCE MIL DOSCIENTOS OCHENTA Y OCHO CON CATORCE.

El presupuesto de inversión asciende a la expresada cantidad de EUROS:
DIECISIETE MIL DOSCIENTOS OCHENTA Y OCHO CON SESENTA Y CINCO.

El presupuesto base de licitación asciende a la expresada cantidad de EUROS:
VEINTE MIL NOVECIENTOS DIECINUEVE CON VEINTISIETE.

6. Previsiones para la producción en masa

Una vez realizado el prototipo del sistema propuesto, conviene realizar una estimación de costes en caso de iniciar su comercialización en masa. Teniendo en cuenta que el trabajo empleado en el planteamiento ya no se debe repetir a corto plazo una vez realizado y que al adquirir una cantidad elevada de componentes se suele realizar un descuento, este es el resumen para la previsión de un coste unitario en caso de una producción en masa:

Tabla 14. Cuadro de precios de los materiales necesarios (para producción en línea)

Producto	Cantidad	Precio unitario	Precio total
Placa ESP32 Devkit Wifi + Bluetooth 4MB IoT	1	2,48 €	2,48 €
Pack Cables Dupont Para Puentear En Placa De Prototipo	12	0,03 €	0,37 €
Diodo LED 3mm	7	0,91 €	6,37 €
Mini Cable USB A Micro USB 50cm Para Arduino Nano Micro Node-MCU	1	1,57 €	1,57 €
Sensor de efecto Hall LEM LTSR 6-NP	1	8,16 €	8,16 €
		TOTAL (SIN IVA)	**10,79 €**

Se entiende que, en este caso, el servidor lo aporta el cliente, por lo que queda fuera del presupuesto. Además, se reduce el material necesario, pues se pasa de un entorno variable (placa de prototipos) a un entorno fijo (cables soldados).

Tabla 15. Presupuesto descompuesto de la construcción del sensor (para producción en línea).

Construcción del sensor				
Unidad	**Descripción**	**Rendimiento**	**Precio**	**Importe**
uds.	**Montaje y construcción del sensor ESP32**			
h	Ingeniero industrial	2	18,35	36,70 €
uds.	Placa Esp32 Devkit Wifi + Bluetooth 4mb IoT	1	2,48	2,48 €
uds.	Pack Cables Dupont Para Puentear En Placa De Prototipo	12	0,03	0,37 €
uds.	Diodo Led 3mm (3v~6v)	7	0,91	6,37 €
uds.	Mini Cable USB A Micro USB 50cm Para Arduino Nano Micro Node-MCU	1	1,57	1,57 €
uds.	Sensor de efecto Hall LEM LTSR 6-NP	1	8,16	8,16 €
%	Costes directos complementarios	0,01	55,65	0,56 €
	Costes directos			56,21 €
	Costes indirectos	0,02	56,21	1,12 €
	COSTE TOTAL			**57,33 €**

Tabla 16. Presupuesto de ejecución material, presupuesto de inversión y presupuesto base de inversión (para producción en línea).

RESUMEN	**IMPORTE**
Construcción del sensor	57,33 €
TOTAL PRESUPUESTO DE EJECUCIÓN MATERIAL	**57,33 €**
15% Gastos generales	8,60 €
6% Beneficio industrial	3,44 €
TOTAL PRESUPUESTO DE INVERSIÓN	**69,37 €**
21% IVA	14,57 €
TOTAL PRESUPUESTO BASE DE LICITACIÓN	**83,94 €**

El presupuesto de ejecución material asciende a la expresada cantidad de EUROS:
CINCUENTA Y SIETE CON TREINTA Y TRES.

El presupuesto de inversión asciende a la expresada cantidad de EUROS:
SESENTA Y NUEVE CON TREINTA Y SIETE.

El presupuesto base de licitación asciende a la expresada cantidad de EUROS:
OCHENTA Y TRES CON NOVENTA Y CUATRO.

APLICACIÓN IOT PARA EL DIAGNÓSTICO DE MOTORES DE INDUCCIÓN

ANEXO 1: CARACTERIZACIÓN DEL ADC

Autor:
Benjamín Cháfer Ferrando

Tutores:
Manuel Pineda Sánchez
Ángel Sapena Bañó

Anexo 1.Caracterización del ADC

1.1. Introducción

Prácticamente todas las señales que se pueden medir son analógicas. Esto implica una medida de una señal continua sin perdidas de datos. No obstante, para poder gestionar señales analógicas desde un dispositivo digital, es necesario convertirlas a este sistema.

El ADC (Conversor de Analógico al Digital, por sus siglas en inglés) se encarga de realizar esta conversión mediante la discretización de la onda.

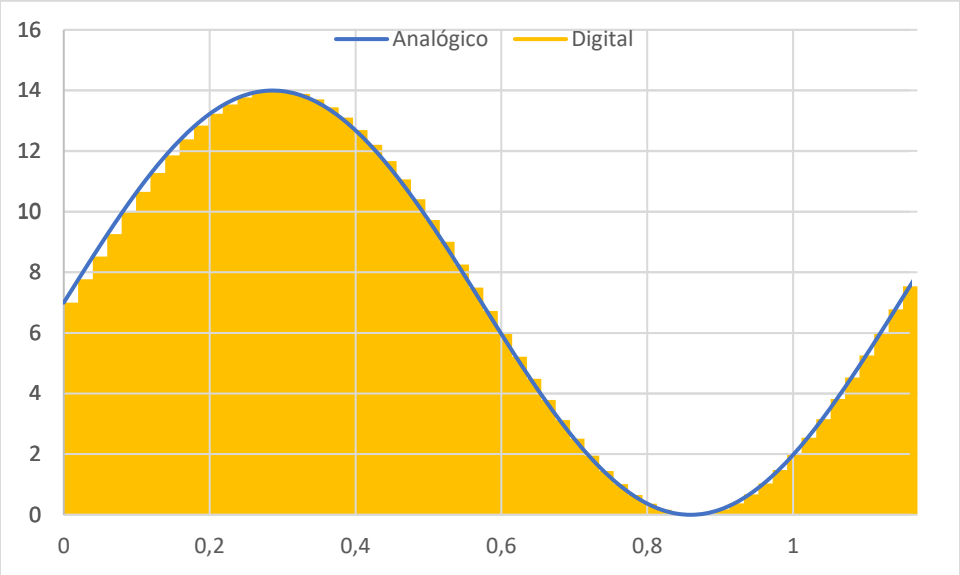


Figura 57. Comparación entre la onda analógica y la interpretación digital

Como se puede observar en la Figura 57, la discretización de la onda produce un efecto de escalera en la onda. De estos escalones que aparecen depende la precisión de las medidas. Cuantos más escalones haya, la precisión será mayor. Al incremento entre escalones se le llama resolución y se calcula con la siguiente ecuación:

Ecuación 15. Cálculo de la resolución del ADC

$$Resolución = \frac{Valor\ máximo}{2^{número\ de\ bits}}$$

Un ADC no podrá detectar un cambio en la señal menor que el que dicta la resolución.

Como resultado de la conversión de la señal analógica a digital, el ADC devuelve el número de escalón en el que se encuentra la medida actual, no el valor real. Es por esto que, tras adquirir la medición, será necesario aplicar un factor de conversión para hallar el valor real de la medida.

Generalmente, la relación entre el valor proporcionado por el ADC y el valor real al que corresponde este siguen una proporción lineal en un rango concreto de valores. De esta forma, el factor a utilizar tendrá una estructura $\boldsymbol{f(x) = a \cdot x + b}$ característica de las funciones lineales.

El objetivo de este anexo es la caracterización del ADC utilizado en este proyecto, definiendo la resolución y los parámetros correspondientes al factor de conversión lineal anteriormente mencionado.

1.2. Procedimiento

Para caracterizar el ADC correctamente, se ha inyectado a este una serie de señales de corriente continua con valores de 0 V, 0,1 V, 0,2 V, 0,5 V, 1 V, 1,5 V, 2 V, 2,5 V, 2,6 V, 2,7 V, 2,8 V, 2,9 V, 3 V y 3,3 V.

Para obtener las mediciones se ha implementado en la función `loop()` el siguiente código:

```
int adc_raw_value = analogRead(ADC_CHANNEL);
Serial.println(adc_raw_value);
```

El cual muestra mediante la función Monitor Serie, incluida en el IDE Arduino, todos los valores que recoge el ADC, como se puede observar en la Figura 58.

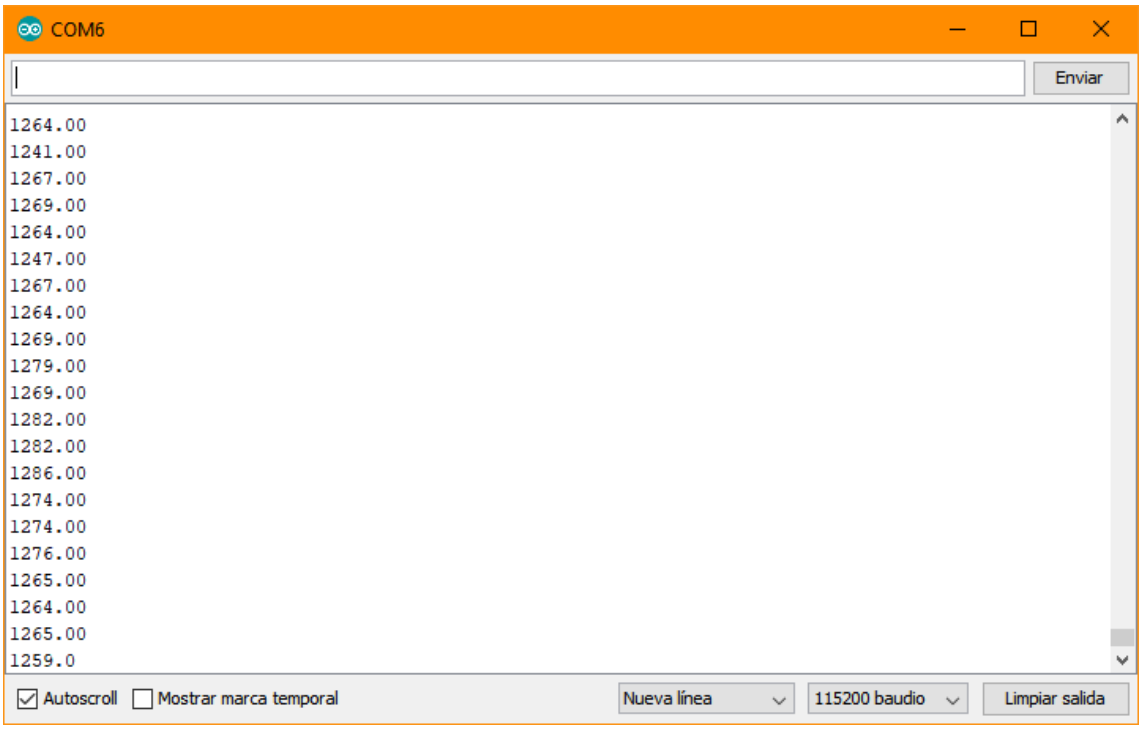
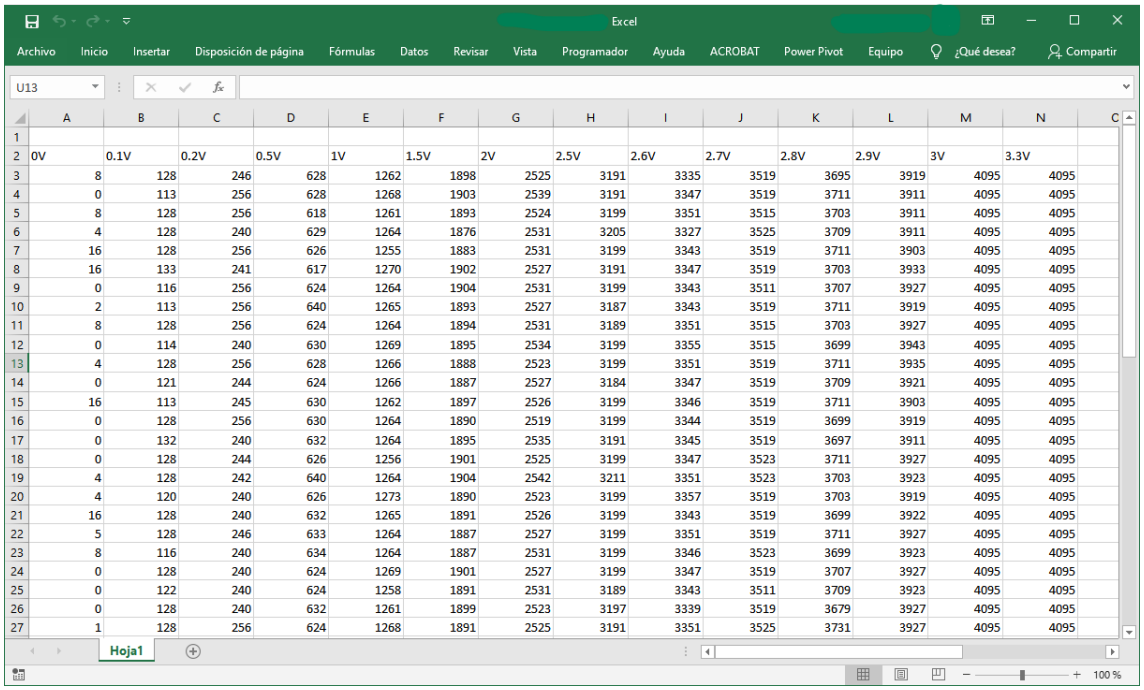


Figura 58. Monitor Serie durante la medición de 1 V.

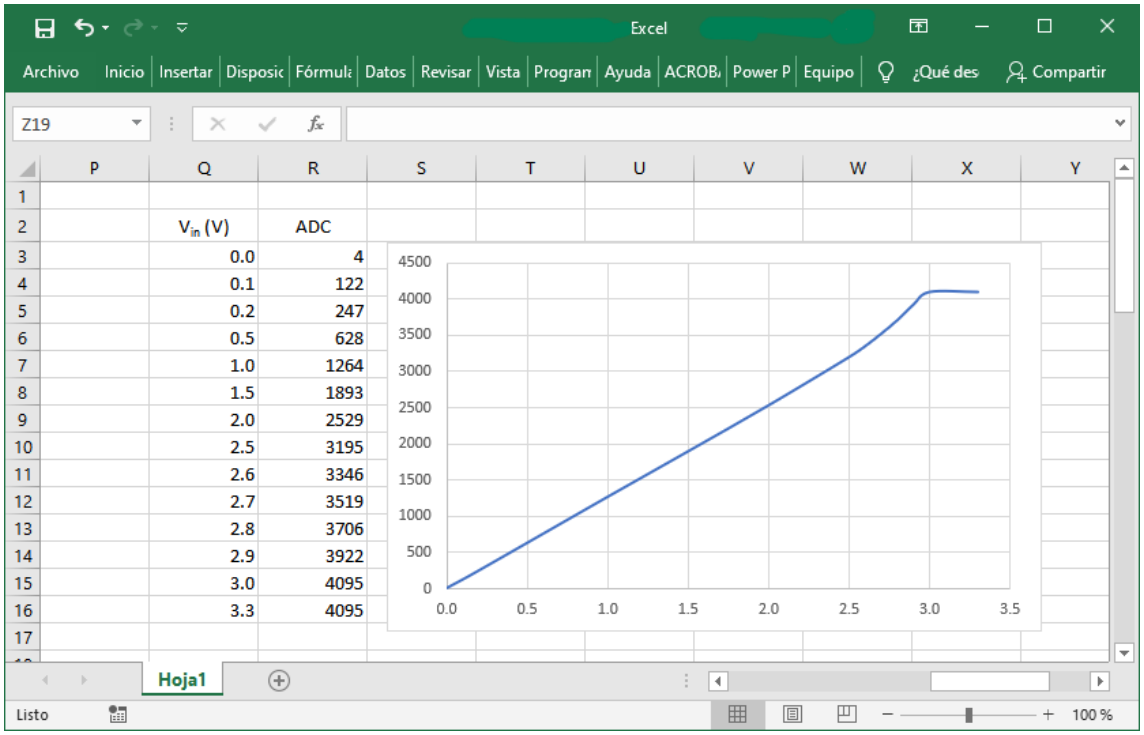
Posteriormente, se han introducido estos resultados en la hoja de Excel, representada en la Figura 59, con el fin de extraer el promedio de las medidas.

Una vez obtenidos los promedios de todas las mediciones, se ha procedido a representarlos en una gráfica (Figura 60) para estimar la relación entre los resultados arrojados por el ADC y el valor de la señal que se ha inyectado a este.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1														
2	0V	0.1V	0.2V	0.5V	1V	1.5V	2V	2.5V	2.6V	2.7V	2.8V	2.9V	3V	3.3V
3	8	128	246	628	1262	1898	2525	3191	3335	3519	3695	3919	4095	4095
4	0	113	256	628	1268	1903	2539	3191	3347	3519	3711	3911	4095	4095
5	8	128	256	618	1261	1893	2524	3199	3351	3515	3703	3911	4095	4095
6	4	128	240	629	1264	1876	2531	3205	3327	3525	3709	3911	4095	4095
7	16	128	256	626	1255	1883	2531	3199	3343	3519	3711	3903	4095	4095
8	16	133	241	617	1270	1902	2527	3191	3347	3519	3703	3933	4095	4095
9	0	116	256	624	1264	1904	2531	3199	3343	3511	3707	3927	4095	4095
10	2	113	256	640	1265	1893	2527	3187	3343	3519	3711	3919	4095	4095
11	8	128	256	624	1264	1894	2531	3189	3351	3515	3703	3927	4095	4095
12	0	114	240	630	1269	1895	2534	3199	3355	3515	3699	3943	4095	4095
13	4	128	256	628	1266	1888	2523	3199	3351	3519	3711	3935	4095	4095
14	0	121	244	624	1266	1887	2527	3184	3347	3519	3709	3921	4095	4095
15	16	113	245	630	1262	1897	2526	3199	3346	3519	3711	3903	4095	4095
16	0	128	256	630	1264	1890	2519	3199	3344	3519	3699	3919	4095	4095
17	0	132	240	632	1264	1895	2535	3191	3345	3519	3697	3911	4095	4095
18	0	128	244	626	1256	1901	2525	3199	3347	3523	3711	3927	4095	4095
19	4	128	242	640	1264	1904	2542	3211	3351	3523	3703	3923	4095	4095
20	4	120	240	626	1273	1890	2523	3199	3357	3519	3703	3919	4095	4095
21	16	128	240	632	1265	1891	2526	3199	3343	3519	3699	3922	4095	4095
22	5	128	246	633	1264	1887	2527	3199	3351	3519	3711	3927	4095	4095
23	8	116	240	634	1264	1887	2531	3199	3346	3523	3699	3923	4095	4095
24	0	128	240	624	1269	1901	2527	3199	3347	3519	3707	3927	4095	4095
25	0	122	240	624	1258	1891	2531	3189	3343	3511	3709	3923	4095	4095
26	0	128	240	632	1261	1899	2523	3197	3339	3519	3679	3927	4095	4095
27	1	128	256	624	1268	1891	2525	3191	3351	3525	3731	3927	4095	4095

Figura 59. Hoja de Excel con los resultados de las medidas.



	P	Q	R	S	T	U	V	W	X	Y
1										
2		V_{in} (V)	ADC							
3		0.0	4							
4		0.1	122							
5		0.2	247							
6		0.5	628							
7		1.0	1264							
8		1.5	1893							
9		2.0	2529							
10		2.5	3195							
11		2.6	3346							
12		2.7	3519							
13		2.8	3706							
14		2.9	3922							
15		3.0	4095							
16		3.3	4095							
17										

Figura 60. Representación de los resultados del ADC en función de la señal aplicada.

De esta gráfica se puede concluir que hay tres tramos en esta función. Un tramo donde se intuye claramente una linealidad entre 0,0 V y 2,6 V; otro tramo ligeramente curvo e irregular a partir de 2,6 V y hasta 3,0 V y finalmente una recta a partir de 3.0 V, que es el **valor máximo del** ADC y, por tanto, donde se desbordan las mediciones.

Finalmente se han aplicado líneas de tendencia a cada tramo de la función con el objetivo de fijar los parámetros de la función de transferencia para averiguar el valor real de los resultados del ADC.

1.3. Análisis de resultados y conclusiones

A raíz de los resultados obtenidos durante 1.2 Procedimiento, se ha elaborado la Tabla 17, que ha servido para crear la gráfica representada en la Figura 60 y posteriormente adaptada en la Figura 61.

Tabla 17. Promedio de las mediciones del ADC en relación con el valor de la señal aplicada al ADC.

V_{in} (V)	0	0,1	0,2	0,5	1	1,5	2
ADC	4	122	247	628	1264	1893	2529
V_{in} (V)	2,5	2,6	2,7	2,8	2,9	3	3,3
ADC	3195	3346	3519	3706	3922	4095	4095

Las líneas de tendencia insertadas mediante Excel también indican la ecuación de su aproximación a la linealidad, así como el valor R^2 del ajuste por mínimos cuadrados, el cual indica la calidad de la aproximación, siendo mayor cuanto más se acerca a la unidad.

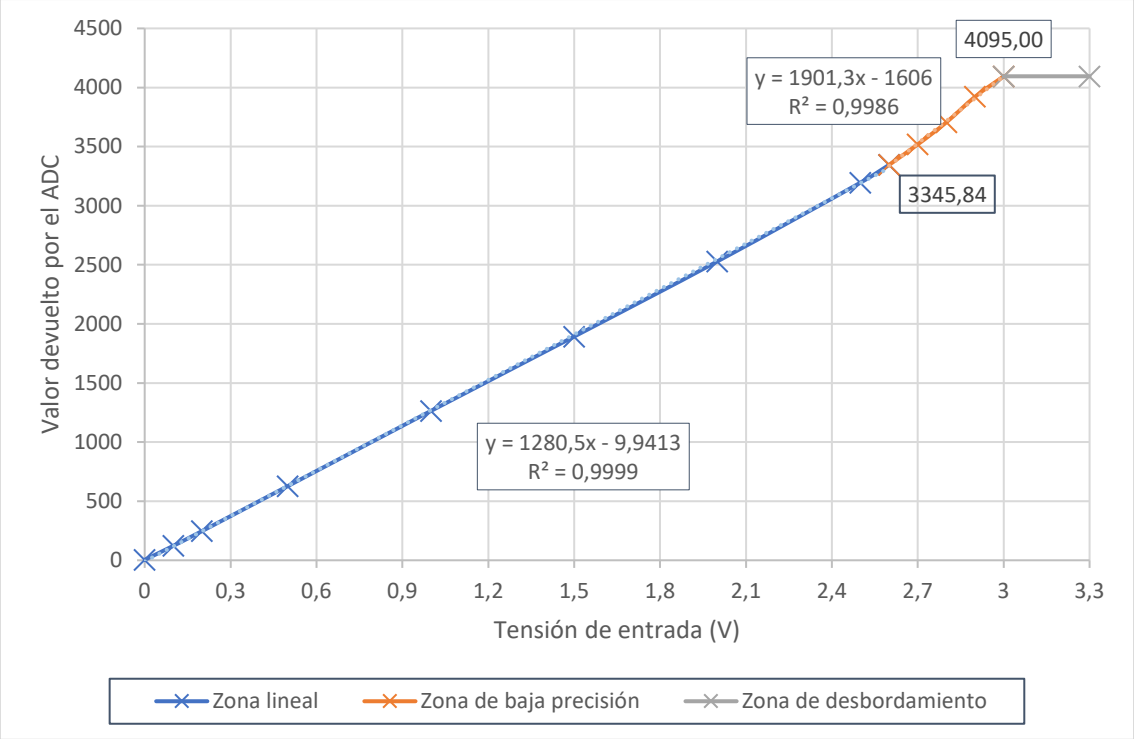


Figura 61. Función de transferencia entre el resultado del ADC y la tensión aplicada a la entrada del ADC.

En esta gráfica se han distinguido los tres tramos anteriormente mencionados: La zona lineal, donde el comportamiento de la función es prácticamente lineal ($V_{in} \in [0, 2{,}6]$); la zona de baja precisión, en la cual el comportamiento de la función deja de ser puramente lineal y tiene una ligera curva que impide una correcta interpretación ($(V_{in} \in (2{,}6, 3)$) y finalmente, la zona de desbordamiento, en la cual se ha alcanzado el valor máximo de medición y no puede aumentar más ($V_{in} \in [3, \infty)$).

Con todos estos datos, se pueden extraer las siguientes conclusiones:

1. Para obtener un resultado más preciso, el rango de aplicación de este ADC deberá ser la zona lineal, es decir, entre 0 V y 2,6 V. Para ello, será necesaria la introducción de un divisor de tensión que permita trabajar en este rango de tensiones.

2. Los parámetros de la función de transferencia serán los que corresponden a la zona lineal, de forma que:

Ecuación 16. Función de transferencia del ADC.

$$\forall\, V_{in} \in [0,2,6]$$
$$f(V_{in}) = 1280{,}5 \cdot V_{in} - 9{,}9413$$

3. La resolución, siguiendo la Ecuación 15, será igual a:

$$Resolución = \frac{Valor\ máximo}{2^{número\ de\ bits}} = \frac{3\ V}{2^{12}} = \frac{3\ V}{4095} = 0{,}7326\ mV$$

De forma que se deberá tener en cuenta que el ADC no detectará una variación menor a 0,7326 mV.

APLICACIÓN IOT PARA EL DIAGNÓSTICO DE MOTORES DE INDUCCIÓN

ANEXO 2: INTRODUCCIÓN A ARDUINO

Autor:
Benjamín Cháfer Ferrando

Tutores:
Manuel Pineda Sánchez
Ángel Sapena Bañó

Anexo 2.Introducción a Arduino

Arduino es un ecosistema de hardware y software de código abierto que pretende promocionar el uso de IoT a base de un coste bajo y una potente comunidad de usuarios.

Se caracteriza por la simplicidad de la programación de sus productos y la compatibilidad de su software en hardware de terceros, como, en este caso, el ESP8266 y el ESP32, a través de librerías de compatibilidad.

2.1. Fundamentos de la programación con Arduino

La programación en Arduino se lleva a cabo mediante dos funciones principales, las cuales controlan el funcionamiento de todo el programa. Estas funciones son `setup` y `loop`.

La función `setup` se ejecuta automáticamente al encender la placa y se utiliza para configurar los diferentes periféricos de la placa a programar, mientras que la función `loop` se ejecuta inmediatamente después de que `setup` termine. La función `loop` contiene todos los comandos que proporcionan funcionalidad al proyecto, es, en resumen, la finalidad del programa. Como su nombre indica, se ejecuta en bucle infinitamente.

2.2. Procedimiento

Antes de comenzar la programación se ha llevado a cabo el procedimiento de instalación [23] de las librerías y ejemplos para el microprocesador ESP8266.

En base a los ejemplos incluidos con las librerías del procesador ESP8266, se ha implementado, en varias fases, un código de prueba para conocer los distintos componentes de la placa que van a ser utilizados.

2.2.1. Funcionamiento del indicador LED

Para configurar el indicador LED, en primer lugar, se ha establecido el pin D4 como salida digital mediante la función heredada `pinMode` la cual recibe como parámetros el pin a configurar y el modo de configuración. En este caso, el pin D4 tiene el alias `LED_BUILTIN` y le es asignado el modo `OUTPUT`.

Una vez configurado el pin del LED, se ha añadido la función `ledSwitch`, la cual alterna el estado del LED. A través de la función heredada `DigitalRead` se obtiene el valor de la tensión en el pin del LED y, posteriormente, se le asigna al pin el valor contrario. Cabe destacar aquí que dicho pin tiene lógica inversa, por lo que al asignar el valor `HIGH` el LED se apaga y al asignar el valor `LOW` se enciende.

2.2.2. Utilización de la comunicación vía puerto serie

Puesto que el programa ha sido implementado en el IDE proporcionado por Arduino, es posible recibir comunicación a través del puerto serie y mostrarla en pantalla de forma nativa, es decir, no es necesario recurrir a software externo, como, por ejemplo, PuTTY.

Según viene en la documentación de la placa NodeMCU v3, e incluso viene serigrafiado en el reverso de esta, la tasa de bits empleada debe ser de 9600 bps. La función heredada encargada de iniciar la transmisión vía puerto serie es `Serial.begin`, la cual recibe como argumento la tasa de bits.

La importancia de la comunicación vía puerto serie radica en la necesidad de proporcionar retroalimentación al usuario, en este caso a quien va a realizar la programación. Para satisfacer dicha necesidad, existen las funciones `Serial.print` y `Serial.println`, las cuales permiten mostrar una información definida por el usuario, ya sea un texto predefinido en el código o valores leídos por el programa.

2.2.3. Adquisición de datos mediante el ADC

El producto final debe medir corrientes de máquinas eléctricas rotativas. El ADC (conversor analógico-digital) tiene como función la adquisición de medidas analógicas y convertirlas en datos digitales que puedan ser procesadas por el programa.

Este modelo de placa posee un ADC integrado, situado en el pin `A0`. La configuración del ADC como entrada de datos ha sido implementada en la función `initAdc`. Aquí se ha usado la función heredada de Arduino `pinMode`, que también se emplea en la configuración del LED, salvo que en este caso se establece el modo en `INPUT` (entrada).

Una vez configurado el ADC, se pueden leer los valores medidos mediante la función heredada `analogRead`, la cual obtiene como argumento el pin donde se va a realizar la lectura de datos.

Se ha implementado la función `getMeasure`, la cual realiza la lectura con la función `analogRead` y la muestra por la consola mediante comunicación vía puerto serie mediante la función `Serial.print`.

2.2.4. Establecimiento de la conexión a una red Wifi

El objetivo de utilizar una conexión Wifi es poder enviar los datos de la lectura del ADC a la nube de forma que puedan recibirse en todo tipo de dispositivos.

En esta fase del programa únicamente se ha establecido la conexión a una red Wifi generada por un smartphone con el sistema operativo Android.

El nombre de la red Wifi y su contraseña se definen como constantes en las primeras líneas del programa de la siguiente forma:

```
#define WIFISSID        "NombreDeLaRed"
#define WIFIPASSWORD    "Contraseña"
```

De esta manera se pueden realizar cambios en la red Wifi sin complicaciones.

Para realizar una conexión a una red Wifi, se ha implementado la función `wifiConnection`. En ella el receptor se ha configurado como estación Wifi (`WIFI_STA`) mediante la función heredada `WiFi.mode`, la cual recibe como argumento el tipo de conexión. De no haberse especificado ningún modo en concreto, trataría de actuar tanto estación Wifi tanto como punto de acceso, lo cual podría causar problemas de comunicación entre los dispositivos conectados a la misma red.

La conexión se lleva a cabo justo después, mediante la función heredada `WiFi.begin`, la cual toma como argumentos el SSID (nombre de la red) y la contraseña de acceso.

Una vez iniciado el proceso de conexión, se ha establecido un bucle mientras no exista conexión, el cual tras varios intentos mostrará un mensaje de error por consola y volverá a ejecutarse el bucle.

Finalmente, tras conseguir la conexión, la función `Serial.print` mostrará por la consola la IP del dispositivo.

2.2.5. Estructura del programa de prueba

La estructura de un programa en un microprocesador se compone básicamente de dos partes: la inicialización y el bucle del programa. En la inicialización se configuran los pines del microprocesador y en el bucle se implementa la funcionalidad que se le quiera dar al sistema.

Para este proyecto, la inicialización se compone de las funciones `initSerial`, `initLed`, `initAdc` y `wifiConnection` que se han descrito anteriormente.

El bucle ha sido diseñado inicialmente para que tome una medida cada segundo a la vez que el LED parpadea mediante las funciones nuevas `ledSwitch` y `getMeasure` que se han implementado en anteriores epígrafes y el periodo de tiempo es marcado con la función heredada de Arduino `delay`, la cual establece un tiempo de pausa que se introduce como argumento de la función.

Tras varias pruebas se ha determinado que la inexistencia de redes Wifi a las que conectar el microprocesador ESP8266 hace que el programa entre en un bucle infinito. Por esta razón se ha establecido la constante booleana `USE_WIFI` para que controle si se debe buscar redes si esta está establecida en `true`.

Cabe destacar que hasta ahora únicamente se ha establecido un bucle, ya que el microprocesador ESP8266 únicamente dispone de un núcleo, por lo tanto, solo admite una acción simultánea. Por ello, para probar la velocidad de muestreo del ESP8266, se ha eliminado el parpadeo del LED y la temporización de las medidas, quedando únicamente la función `getMeasure`.

APLICACIÓN IOT PARA EL DIAGNÓSTICO DE MOTORES DE INDUCCIÓN

ANEXO 3. FUNCIONES Y OTROS FRAGMENTOS DE CÓDIGO

Autor:
Benjamín Cháfer Ferrando

Tutores:
Manuel Pineda Sánchez
Ángel Sapena Bañó

Anexo 3. Funciones y otros fragmentos de código

3.1. Utilizado en Node-RED

3.1.1. setDebugProperties (Initialize... > Debug Properties)

```
let sourceText;
let defaultSource = env.get("DefaultSource");
// ESP32 ADC        0
// ESP32 DEBUG      1
// Signal Generator 2
// File             3

let esp32mode = env.get("Esp32Mode");

global.set("DebugSignalSource", defaultSource);
global.set("DebugESP32Advanced", esp32mode);
global.set("MeasuringArrayIndex", 0);

switch(defaultSource) {
    case "0": sourceText = "ESP32 ADC"; break;
    case "1": sourceText = "ESP32 DEBUG"; break;
    case "2": sourceText = "Signal Generator"; break;
    case "3": sourceText = "File"; break;
    default: msg.payload = "Error"; return msg;
}

msg.payload = "Done; " + sourceText;

return msg;
```

3.1.2. SetFourierProperties (Initialize... > Debug Properties)

```
let array = [ 0 ];
let useDb = env.get("UseDb");
let sampleFreq = env.get("SampleFreq");
let sampleNum = env.get("SampleNum");
let waveZoom = env.get("WaveZoom");

let status = "Done";

global.set("FourierArray", array);
global.set("FourierUseDB", useDb);
global.set("FourierSampleFreq", sampleFreq);
global.set("FourierSampleNum", sampleNum);
global.set("FourierWaveZoom", waveZoom);

// status = global.get("FourierWaveZoom");

msg.payload = status;

return msg;
```

3.1.3. SetAdcProperties (Initialize... > Debug Properties)

```
let a = env.get("a");
let b = env.get("b");
let float = env.get("float_dec");
let status;

global.set("MeasuringAdcA", a);
global.set("MeasuringAdcB", b);
global.set("MeasuringAdcFloat", float);

status = "y = " + a + " x + " + b;

msg.payload = status;

return msg;
```

3.1.4. SetFileSettings (Initialize... > Debug Properties)

```
let rDefaultPath = env.get("rDefaultPath");
let sDefaultPath = env.get("rDefaultPath");
let tDefaultPath = env.get("rDefaultPath");

let defaultPhase = env.get("DefaultPhase");

let simLength = env.get("SimLength");

let objectDirectories =
{
    "irPath": rDefaultPath,
    "isPath": sDefaultPath,
    "itPath": tDefaultPath
}

global.set("FileDirectories", objectDirectories);
global.set("FileCurrentPhase", defaultPhase);
global.set("FileSimLength", simLength);
msg.payload = "Done";

switch(defaultPhase) {
    case "R": global.set("FileCurrentPath", objectDirectories.irPath);
break;
    case "S": global.set("FileCurrentPath", objectDirectories.isPath);
break;
    case "T": global.set("FileCurrentPath", objectDirectories.itPath);
break;
    default: msg.payload = "Error: " + defaultPhase;
global.set("FileCurrentPath", null); break;
}

return msg;
```

3.1.5. constructValuesFromBytes (Measuring)

```
const a = global.get("MeasuringAdcA"), b =
global.get("MeasuringAdcB"), dec =  global.get("MeasuringAdcFloat");
let byteArray = msg.payload;
let valueArray = [ 0 ];
let i, j = 0;
let realMsr;

for (i = 384; i < byteArray.length; i = i + 2) {
    let value = 0;
    value = (byteArray[i] << 8) + byteArray[i + 1];
    realMsr = (value - b) / a;
    valueArray[j] = Math.round(realMsr * Math.pow(10, dec)) /
Math.pow(10, dec);
    j++;
}

global.set("MeasuringArray", valueArray);
msg.payload = valueArray;
msg.arrayLength = j;

return msg;
```

3.1.6. getCurrentSpeed (Measuring)

```
msg.payload = parseFloat(msg.payload);
return msg;
```

3.1.7. getMeanArray (Measuring)

```
let measuringArray = global.get("MeasuringArray") || [ 0 ];
let index = global.get("MeasuringArrayIndex") || 0;
let meanArrayLength = 50;

msg.payload = measuringArray.slice(index - meanArrayLength, index);

index += meanArrayLength;

if(index >= measuringArray.length) index = meanArrayLength;

global.set("MeasuringArrayIndex", index);

if(global.get("MeasuringIsOn")) return msg;
```

3.1.8. *getFreqArray (Analysis)*

```
let samples = global.get('FourierSampleNum') || 1024;
let freq = global.get('FourierSampleFreq') || 1000; // frecuencia de
muestreo
let axis = [ 0 ];
let axisString = [ "0" ];
let i;
for (i = 0; i < samples / 10; i++) {
    if (i === 0) axis[i] = 0;
    else axis[i] = i * freq / samples;
    axisString[i] = (Math.round(axis[i] * 100 ) / 100 ).toString();
}
msg.payload = axisString;
return msg;
```

3.1.9. *getTimeArray (Analysis)*

```
let zoom = global.get('FourierWaveZoom') || 1;
let freq = global.get('FourierSampleFreq') || 1000;
let samples = freq;
//let samples = global.get('FourierSampleNum') || 8000;
let axis = [ 0 ];
let axisString = [ "0" ];
let i;
for (i = 0; i < (samples / zoom); i++) {
    if (i === 0) axis[i] = 0;
    else axis[i] = axis[i - 1] + (1 / freq);
    axisString[i] = (Math.round(axis[i] * 100) / 100).toString();
}
msg.payload = axisString;
msg.zoom = zoom;
return msg;
```

3.1.10. *fillFourierArray (Analysis)*

```
let array = global.get('FourierArray');
let samples = global.get('FourierSampleNum');

array.push(msg.payload);
global.set('FourierArray', array);
msg.payload = null;

if(array.length == samples) {
    msg.payload = array;
    global.set('FourierArrayBackup', array);
    global.set('FourierArray', []);
}

return msg;
```

3.1.11. bin2DB (Analysis)

```
const dec =  global.get("adc_float");
// let fft_array = msg.payload;
let fft_array = msg.fftAbs;
let db_array = [ 0, 0 ];
//let fft_max = Math.abs(Math.max.apply(null, fft_array));
let fft_max = msg.fftmax;
let i, length = fft_array.length;

for (i = 0; i < length; i++) {
    //db_array[i] = 20 * Math.log((Math.abs(fft_array[i]) / fft_max),
10);
    db_array[i] = 20 * Math.log((fft_array[i] / fft_max), 10);
    if (isNaN(db_array[i])) {
        db_array[i] = 20 * Math.log((0.01 / fft_max), 10);
    }
}

global.set('FourierArrayBackup', db_array);
msg.payload = db_array;

return msg;
```

3.1.12. getSlipAndFreqs (Analysis)

```
let nm = global.get("MeasuringCurrentSpeed") || 2985;
let ns = 3000;

let freqArray    = global.get('FourierFreqAxis');
let fourierArray = global.get('FourierArrayBackup') || [ 0 ];
let array        = fourierArray.slice(0, freqArray.length)
let max          = Math.max(array) || 0;
let maxFreq      = freqArray[array.indexOf(max)];

let f0 = maxFreq;

let s = (ns - nm) / ns;
let sp = s * 100;

let f1 = Math.round(f0 * (1 - 2 * s) * 100) / 100;
let f2 = Math.round(f0 * (1 + 2 * s) * 100) / 100;

global.set("MeasuringSlip", sp);
global.set("FourierMainFreq", f0);
global.set("FourierFailFreqs", [ f1, f2 ]);

msg.speed     = nm;
msg.slip      = global.get("MeasuringSlip");
msg.max       = global.get("FourierMainFreq");
msg.failfreqs = global.get("FourierFailFreqs");

return msg;
```

3.1.13. getFreqObject (Analysis)

```
const tolerance = 0.5;

let freqAxis         = global.get("FourierFreqAxis");
let failFreqs        = global.get("FourierFailFreqs");
let fourierFullArray = global.get("FourierArrayBackup");
let fourierArray     = fourierFullArray.slice(0, freqAxis.length);

let freqLimits  = [ 0 ];

let freq1Values  = [ 0 ];
let freq1Indexes = [ 0 ];
let freq1Results = [ 0 ];

let freq2Values  = [ 0 ];
let freq2Indexes = [ 0 ];
let freq2Results = [ 0 ];

let newValue = 0;

freqLimits = [failFreqs[0] - tolerance, failFreqs[0] + tolerance,
failFreqs[1] - tolerance, failFreqs[1] + tolerance];

freq1Values.pop();
freq1Indexes.pop();
freq1Results.pop();

freq2Values.pop();
freq2Indexes.pop();
freq2Results.pop();

for(i = 0; i < freqAxis.length; i++)
{
    newValue = parseFloat(freqAxis[i]);
    if ((newValue > freqLimits[0] && newValue < freqLimits[1])) {
        freq1Values.push(newValue);
        freq1Indexes.push(i);
        freq1Results.push(fourierArray[i]);
    }
    else if ((newValue > freqLimits[2] && newValue < freqLimits[3])) {
        freq2Values.push(newValue);
        freq2Indexes.push(i);
        freq2Results.push(fourierArray[i]);
    }
}

let freq0index = freqAxis.indexOf(global.get("FourierMainFreq"));
let freq1index = freq1Results.indexOf(Math.max.apply(null,
freq1Results));
let freq2index = freq2Results.indexOf(Math.max.apply(null,
freq2Results));

let freqData  = {
```

```
  "MAIN" : {
      index : freq0index,
      freq  : parseFloat(global.get("FourierMainFreq")),
      value : fourierArray[freq0index]
  },
  "BAR_1": {
      index: freq1Indexes[freq1index],
      freq: freq1Values[freq1index],
      value: freq1Results[freq1index]
  },
  "BAR_2": {
      index : freq2Indexes[freq2index],
      freq  : freq2Values[freq2index],
      value : freq2Results[freq2index]
  }
}

global.set("FourierFreqObject", freqData);
msg.payload = global.get("FourierFreqObject");

return msg;
```

3.1.14. FindFailure (Analysis)

```
let freqObject = global.get("FourierFreqObject");
msg.show = false;
msg.ledStatus = "OK";

let messageFail = "One or more failure frequencies have been
detected.";
let topicFail = "Failure detected";

let messageSuspect = "One or more frequencies are reaching high
values.";
let topicSuspect = "Potential failure detected";

let failThreshold    = -45;
let suspectThreshold = -55;

let i, fail = 0, suspect = 0;

if (freqObject.BAR_1.value > failThreshold) fail++;
if (freqObject.BAR_2.value > failThreshold) fail++;

if (freqObject.BAR_1.value > suspectThreshold) suspect++;
if (freqObject.BAR_2.value > suspectThreshold) suspect++;

if (fail > 1 || suspect > 1)
{
    if (fail < suspect) {
        msg.payload = messageFail;
        msg.topic = topicFail;
        msg.ledStatus = "ADVISE";
    }
    else {
        msg.payload = messageSuspect;
        msg.topic = topicSuspect;
        msg.ledStatus = "FAIL";
    }
    msg.show = true;
}

return msg;
```

3.1.15. constructJsonfile (Analysis)

```
let data = global.get("FourierFreqObject");
let dbObject = msg.payload;
let d = new Date ();

let newEntry =
    {
        time: d.toJSON(),
        data: data
    }

if (dbObject.length >= 90) dbObject.pop();

dbObject.push(newEntry);
msg.payload = dbObject;

return msg;
```

3.1.16. constructFreqChart (Analysis)

```
let freqObj = msg.payload;
let i;

let objChart =
    {
        series: [["BAR 1"], ["BAR 2"]],
        data  : [ ],
        labels: [ ]
    }

let bar1array = [ ];
let bar2array = [ ];

for (i = 0; i < freqObj.length; i++) {
    objChart.labels.push(freqObj[i].time);
    if (isNaN(freqObj[i].data.BAR_1.value)) {
        bar1array.push(bar1array[i - 1]);
    } else {
        bar1array.push(freqObj[i].data.BAR_1.value);
    }
    if (isNaN(freqObj[i].data.BAR_2.value)) {
        bar2array.push(bar2array[i - 1]);
    } else {
        bar2array.push(freqObj[i].data.BAR_2.value);
    }
}

objChart.data.push(bar1array);
objChart.data.push(bar2array);

msg.payload = [objChart];

return msg;
```

3.1.17. constructValueFromBytes (Signal from file)

```
const a = global.get("MeasuringAdcA"), b =
global.get("MeasuringAdcB"), dec =  global.get("MeasuringAdcFloat");
let byteArray = msg.payload;
let samples = global.get('FourierSampleNum');
let valueArray = [ 0 ];
let numSecond = global.get('FileSecondCounter');
let simLength = global.get('FileSimLength');
let i, j = 0;
let realMsr;

for (i = (samples * (numSecond - 1)); i < (samples * numSecond); i++)
{
    let value = 0;
    value = parseInt(byteArray[i], 16);
    realMsr = (value - b) / a;
    valueArray[j] = Math.round(realMsr * Math.pow(10, dec)) /
Math.pow(10, dec);
    j++;
}

if (numSecond < simLength ) numSecond++;
else numSecond = 1;
global.set('FileSecondCounter', numSecond);

//msg.status = ({fill:"red",shape:"ring",text:"t = " + numSecond + "
s"});

global.set("MeasuringArray", valueArray);
msg.payload = valueArray;
msg.arrayLength = j;

return msg;
```

3.2. Utilizado en MATLAB

3.2.1. Función de extracción de medidas desde archivo .m

```
function x = extractData2(ir, t)

    ir_ok = signal2Adc(ir + max(ir));

    fileID = fopen('main/data/sim_ir.txt','w');
    fprintf(fileID,'%04X\n',ir_ok);
    fclose(fileID);

    plot(t, ir_ok)

    x = ir_ok;
end
```

3.2.2. Función para la simulación del ADC a partir de un valor medido

```
function y = signal2Adc(x)
    y = cast(1280.5 * x - 9.9413, 'uint32');
end
```

APLICACIÓN IOT PARA EL DIAGNÓSTICO DE MOTORES DE INDUCCIÓN

ANEXO 4. CÓDIGO DEL ESP32

Autor:
Benjamín Cháfer Ferrando

Tutores:
Manuel Pineda Sánchez
Ángel Sapena Bañó

Anexo 4.Código del ESP32

A continuación, se presenta el programa realizado en Arduino IDE para ser introducido en el microcontrolador ESP32.

Para que este fuera extraído se ha utilizado la aplicación web *Arduino Code Printer*[1].

Índice

[1] **Arduino Code Printer**, Pieter P. – https://tttapa.github.io/Tools/Arduino-Code-Printer.html

main.ino

Benjamín Cháfer

```ino
#include "config.h"
#include "esp32pin.h"
#include "led.h"
#include "wifi_connection.h"
#include "adc.h"
#include "mqtt.h"
#include "isr.h"

#include "esp32-hal-cpu.h"

// the setup function runs once when you press reset or power the board
void setup()
{
  // initialize serial communication at 115200 bits per second:
  Serial.begin(BAUDRATE);
  Serial.write(12);
  Serial.println("Inicializando...");
  Serial.print("Velocidad del reloj: ");
  Serial.print(String(getCpuFrequencyMhz()));
  Serial.println(" MHz");
  initLed(LED_BUILTIN);
  initLed(LED_ERR_WIFI);
  initLed(LED_ERR_MQTT);
  initLed(LED_ERR_SEND);
  initLed(LED_ERR_DBUG);
  initLed(LED_STATUS_GRN);
  initLed(LED_STATUS_YLW);
  initLed(LED_STATUS_RED);
  if(USE_WIFI)
  {
    wifiConnection();
  }
  mqttSetup();
  initTimer(TIMER_ID);
  initPulse(PULSE_COUNT);
  ledSet(LED_BUILTIN);
  Serial.println("Preparando tareas...");
}

void loop()
{
  mqttLoop();
}
```

esp32pin.h

Benjamín Cháfer

```
 1  #ifndef _ESP32PIN_H
 2  #define LED_BUILTIN 2
 3
 4  #define ADC1_CH0    36
 5  //#define ADC1_CH1    37
 6  //#define ADC1_CH2    38
 7  #define ADC1_CH3    39
 8  #define ADC1_CH4    32
 9  #define ADC1_CH5    33
10  #define ADC1_CH6    34
11  #define ADC1_CH7    35
12
13  #define LED_ERR_WIFI   23
14  #define LED_ERR_MQTT   21
15  #define LED_ERR_SEND   19
16  #define LED_ERR_DBUG   5
17
18  #define LED_STATUS_GRN   14
19  #define LED_STATUS_YLW   26
20  #define LED_STATUS_RED   33
21
22  #define _ESP32PIN_H
23  #endif
```

config.h

Benjamín Cháfer

```
#ifndef _CONFIG_H
#define _CONFIG_H

#define MOVISTAR 0
#define ANDROID  1

#define NETWORK MOVISTAR
// #define NETWORK ANDROID

#include "esp32pin.h"

// Sampling
#define SAMPLE_RATE     1000  // Hz
#define MSECONDS        1
#define USECONDS        1000
#define SAMPLE_NUM      1024 * 8
#define BUFFER_LENGTH   SAMPLE_NUM * 2  // Se envían valores de 2 bytes
#define PULSES_PER_LAP  720

// Timer
#define TIMER_ID        0
#define TIMER_PRESCALER 80

// Serial
#define BAUDRATE        115200

// Wifi

#if NETWORK == ANDROID
  #define USE_WIFI        true
  #define WIFISSID        "****"
  #define WIFIPASSWORD    "****"
#elif NETWORK == MOVISTAR
  #define USE_WIFI        true
  #define WIFISSID        "****"
  #define WIFIPASSWORD    "****"
#else
  #define USE_WIFI        false
#endif

// MQTT - mqtt.h
#if NETWORK == ANDROID
  #define LOCAL_IP        "0.0.0.0"
  #define SERVER_IP       "0.0.0.0"
#elif NETWORK == MOVISTAR
  #define LOCAL_IP        "0.0.0.0"
  #define SERVER_IP       "0.0.0.0"
#endif

#define SERVER_PORT     1883

// ADC
#define ADC_CHANNEL     ADC1_CH0
#define PULSE_COUNT     ADC1_CH1
/* Calibration  y = ax + b" */
#define ADC_PARAM_A     1280.5
#define ADC_PARAM_B     -9.9413
#define ADC_CONVERT     true

bool enableOutput = true;
bool useDebugAdvanced = false;
bool useDebugLoop = false;

#endif
```

led.ino

Benjamín Cháfer

```
1  #include "led.h"
2  #include "config.h"
3
4  void initLed(int pin)
5  {
6    pinMode(pin, OUTPUT);
7    Serial.print("Inicializado pin ");
8    Serial.print(pin);
9    Serial.println(" como salida.");
10   ledSet(pin);
11   delay(1000);
12   ledReset(pin);
13 }
14
15 void ledSwitch(int pin)
16 {
17   digitalWrite(pin, !digitalRead(pin));
18 }
19
20 void ledSet(int pin){
21   digitalWrite(pin, HIGH);
22 }
23
24 void ledReset(int pin){
25   digitalWrite(pin, LOW);
26 }
27
28 bool ledIsSet(int pin){
29   if(digitalRead(pin) == HIGH) return true;
30   else return false;
31 }
```

led.h

Benjamín Cháfer

```
1   #ifndef _LED_H
2   #define _LED_H
3
4   void initLed(int pin);
5   void ledSwitch(int pin);
6   void ledSet(int pin);
7   void ledReset(int pin);
8   bool ledIsSet(int pin);
9
10  #endif
```

adc.ino

Benjamín Cháfer

```
1   #include "adc.h"
2   #include "config.h"
3
4   float adcToReal(int adcValue)
5   {
6     float voltage = 0;
7     // y = ax + b
8     voltage = ((float)adcValue - ADC_PARAM_B) / ADC_PARAM_A;
9     return voltage;
10  }
11
12  int realToAdc(float voltage)
13  {
14    int adc_value = 0;
15    // y = ax + b
16    adc_value = (voltage * ADC_PARAM_A) + ADC_PARAM_B;
17    return adc_value;
18  }
19
20  int AdcGetMeasure()
21  {
22    int adc_raw_value = 0;
23    adc_raw_value = analogRead(ADC_CHANNEL);
24    return adc_raw_value;
25  }
```

adc.h

Benjamín Cháfer

```
1  #ifndef _ADC_H
2  #define _ADC_H
3
4  float adcToReal(int adcValue);
5  int realToAdc(float voltage);
6  int AdcGetMeasure();
7
8  #endif
```

wifi_connection.ino

Benjamín Cháfer

```
#include "wifi_connection.h"
#include "config.h"

#include <ETH.h>
#include <WiFi.h>
#include <WiFiAP.h>
#include <WiFiClient.h>
#include <WiFiGeneric.h>
#include <WiFiMulti.h>
#include <WiFiScan.h>
#include <WiFiServer.h>
#include <WiFiSTA.h>
#include <WiFiType.h>
#include <WiFiUdp.h>

void wifiConnection() {
  const char* ssid     = WIFISSID;
  const char* password = WIFIPASSWORD;

  int timeOutConnection = 0;

  Serial.print("Conectando a red WiFi: ");
  Serial.println(ssid);

  // Ported from ESP8266
  // Explicitly set the ESP8266 to be a WiFi-client, otherwise, it by default,
  // would try to act as both a client and an access-point and could cause
  // network-issues with your other WiFi-devices on your WiFi-network.

  WiFi.mode(WIFI_STA);
  Serial.println("Configurado como cliente WiFi.");
  WiFi.begin(ssid, password);
  Serial.println("Iniciando conexión a la red WiFi.");

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    timeOutConnection++;
    ledSwitch(LED_ERR_WIFI);
    Serial.print(".");
    if(timeOutConnection >= 40){
      Serial.println("");
      Serial.println("Error de conexión. Reintentando.");
      timeOutConnection = 0;
    }
  }
  digitalWrite(LED_ERR_WIFI, HIGH);
  Serial.println("");
  Serial.println("Conexión con éxito.");
  Serial.print("\tDirección IP: ");
  Serial.println(WiFi.localIP());
}

bool getWifiStatus(){
  if (WiFi.status() == WL_CONNECTED) return true;
  else return false;
}
```

wifi_connection.h

Benjamín Cháfer

```
1  #ifndef _WIFI_H
2  #define _WIFI_H
3
4  void wifiConnection();
5  bool getWifiStatus();
6
7  #endif
```

mqtt.ino

Benjamín Cháfer

```
#include "mqtt.h"
#include "mqtt_callback.h"
#include "config.h"
#include "led.h"
#include "debug_signal.h"
#include "functions.h"


/*********
  Rui Santos
  Complete project details at https://randomnerdtutorials.com
*********/

void mqttSetup()
{
  client.setServer(mqtt_server, SERVER_PORT);
  Serial.print("Conectando a ");
  Serial.print(SERVER_IP);
  Serial.print(":");
  Serial.println(SERVER_PORT);
  client.setCallback(callback);
}

void mqttReconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Conectando al servidor MQTT...");
    delay(200);
    ledSwitch(LED_ERR_MQTT);
    delay(200);
    ledSwitch(LED_ERR_MQTT);
    delay(200);
    ledSwitch(LED_ERR_MQTT);
    delay(200);
    ledSwitch(LED_ERR_MQTT);
    // Attempt to connect
    if (client.connect(mqtt_server)) {
      ledSet(LED_ERR_MQTT);
      Serial.println("Listo.");
      // Subscribe
      client.subscribe("esp32/enableOutput");
      client.subscribe("esp32/devMode");
      client.subscribe("esp32/devModeAdvanced");
      client.subscribe("esp32/motorStatus");
    } else {
      ledReset(LED_ERR_MQTT);
      Serial.print("Error, rc=");
      Serial.print(client.state());
      Serial.println(" volviendo a intentar en 5 segundos.");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void mqttLoop() {
  if (!client.connected())
  {
    mqttReconnect();
  }
  client.loop();
  mqttMainFunction();
}
```

mqtt.h

Benjamín Cháfer

```
#ifndef _MQTT_H
#define _MQTT_H

#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>

WiFiClient espClient;
PubSubClient client(espClient);

long lastMsg = 0;
char msg[50];
int value = 0;
const char* mqtt_server = SERVER_IP;

void mqttSetup();
void mqttLoop();
void mqttLoopDebug();

#endif
```

isr.ino

Benjamín Cháfer

```
#include "isr.h"
#include "config.h"

// https://techtutorialsx.com/2017/10/07/esp32-arduino-timer-interrupts/

void IRAM_ATTR onTimer()
{
  portENTER_CRITICAL_ISR(&timerMux);
  interruptCounter++;
  portEXIT_CRITICAL_ISR(&timerMux);
}

void initTimer(int timer_id)
{
  timer = timerBegin(timer_id, TIMER_PRESCALER, true);
  Serial.print("Inicializado temporizador ");
  Serial.print(timer_id);
  Serial.println(".");
  timerAttachInterrupt(timer, &onTimer, true);
  Serial.println("Interrupciones asignadas.");
  timerAlarmWrite(timer, USECONDS, true);
  Serial.println("Preescala\tFrecuencia de disparo");
  Serial.print(TIMER_PRESCALER);
  Serial.print("\t\t");
  Serial.print(USECONDS);
  Serial.println(" Hz");
  timerAlarmEnable(timer);
  delay(2000);
}

void IRAM_ATTR onPulse()
{
  //portENTER_CRITICAL_ISR(&timerMux);
  pulseCounter++;
  //portEXIT_CRITICAL_ISR(&timerMux);
}

void initPulse(int pin)
{
  Serial.print("Inicializado entrada ");
  Serial.print(pin);
  Serial.println(".");
  attachInterrupt(pin, onPulse, RISING);
  Serial.println("Interrupciones asignadas.");
  delay(2000);
}
```

isr.h

Benjamín Cháfer

```
1   #ifndef _TIMER_H
2   #define _TIMER_H
3
4   volatile int interruptCounter;
5   volatile int pulseCounter;
6   int totalInterruptCounter;
7
8   hw_timer_t * timer = NULL;
9   portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
10
11  #endif
```

functions.ino

Benjamín Cháfer

```
#include "adc.h"
#include "mqtt.h"
#include "mqtt_callback.h"
#include "config.h"
#include "led.h"
#include "debug_signal.h"
#include "isr.h"
#include "functions.h"

void mqttMainFunction()
{
  long now = millis();
  char adcString[8];
  char speedString[8];
  int adc_raw_value = 0;
  float rotating_speed = DEBUG_SPEED;

  if (interruptCounter > 0 && enableOutput) // interruptCounter vale 1 al producirse una interrupción
  {
    // lastMsg = now;

    portENTER_CRITICAL(&timerMux);
    interruptCounter--;
    portEXIT_CRITICAL(&timerMux);

    //totalInterruptCounter++;

    if (useDebugLoop)
    {
      // int adc_raw_value = 0;
      adc_raw_value = mqttDebugFunct(now);
      // Convert the value to a char array
      dtostrf(adc_raw_value, 1, 2, adcString);
      // Get sample bytes
      adcBytes[bytes++] = getBytesFromInt(adc_raw_value, 0);
      adcBytes[bytes++] = getBytesFromInt(adc_raw_value, 1);
    }
    else
    {
      // int adc_raw_value = 0;
      adc_raw_value = AdcGetMeasure();
      // Convert the value to a char array
      dtostrf(adc_raw_value, 4, 0, adcString);
      adcBytes[bytes++] = getBytesFromInt(adc_raw_value, 0);
      adcBytes[bytes++] = getBytesFromInt(adc_raw_value, 1);
    }
    // If byte buffer is full
    if(bytes == BUFFER_LENGTH)
    {
      client.beginPublish("esp32/adcBuffer",BUFFER_LENGTH, false);
      client.write(adcBytes, BUFFER_LENGTH);
//      Serial.write(12);
//      Serial.print(totalInterruptCounter);
//      Serial.print(": ");
      Serial.print("Buffer enviado: ");
      Serial.println(client.endPublish());
      if (useDebugLoop) rotating_speed = getSpeedFromPulses();
      dtostrf(rotating_speed, 4, 2, speedString);
      client.publish("esp32/speedOutput", speedString);
      bytes = 0;
      ledSwitch(LED_ERR_SEND);
    }
    // client.publish("esp32/adcOutput", adcString);
    // Serial.println(adcString);
  }
}

float getSpeedFromPulses()
{
  int spinTime = SAMPLE_NUM / 1024;
  float result = (pulseCounter * 60) / (spinTime * PULSES_PER_LAP);
  pulseCounter = 0;
  return result;
}

```

```
76  int mqttDebugFunct(long now)
77  {
78    int debug_f_funct = 0;
79    int debug_g_funct = 0;
80    int debug_h_funct = 0;
81    int debug_signal = 0;
82    float t = now/1000.0;
83
84    if(useDebugAdvanced)
85    {
86      debug_signal = advancedSinFunction(t, DEBUG_F_A, DEBUG_F_FREQ, DEBUG_BETA, DEBUG_S) +
    DEBUG_OFFSET;
87    }
88    else
89    {
90      debug_f_funct = getSinFunction(t, DEBUG_F_A, DEBUG_F_FREQ, DEBUG_F_PHI);
91      debug_g_funct = getSinFunction(t, DEBUG_G_A, DEBUG_G_FREQ, DEBUG_G_PHI);
92      debug_h_funct = getSinFunction(t, DEBUG_H_A, DEBUG_H_FREQ, DEBUG_H_PHI);
93      debug_signal = debug_f_funct + debug_g_funct + debug_h_funct + DEBUG_OFFSET;
94    }
95
96    return debug_signal;
97  }
98
99  byte getBytesFromInt(int value, int pos)
100 {
101   switch(pos)
102   {
103     case 0:
104       return value >> 8;
105       break;
106     default:
107       return value & 0xFF;
108       break;
109   }
110 }
```

functions.h

Benjamín Cháfer

```
#ifndef _FUNCTS_H
#define _FUNCTS_H

void mqttMainFunction();
void mqttDebugFunct();

int bytes = 0;
int sampleCount = 0;
byte adcBytes[BUFFER_LENGTH];
int sampleReal[SAMPLE_NUM];
int sampleImag[SAMPLE_NUM];

#endif
```

mqtt_callback.ino

Benjamín Cháfer

```
#include "mqtt.h"
#include "mqtt_callback.h"
#include "config.h"
#include "led.h"
#include "debug_signal.h"
#include "functions.h"

void callback(char* topic, byte* message, unsigned int msglength)
{
  Serial.print("Message arrived on topic: ");
  Serial.print(topic);
  Serial.print(". Message: ");
  String messageTemp;

  for (int i = 0; i < msglength; i++)
  {
    Serial.print((char)message[i]);
    messageTemp += (char)message[i];
  }
  Serial.println();

  if (String(topic) == "esp32/enableOutput")
  {
    Serial.print("Salida de datos ");
    if(messageTemp == "true")
    {
      enableOutput = true;
      // ledReset(LED_ERR_DBUG);
      Serial.println("activada.");
    }
    else if(messageTemp == "false")
    {
      enableOutput = false;
      Serial.println("desactivada.");
    }
  }
  if (String(topic) == "esp32/devMode")
  {
    Serial.print("Modo debug ");
    if(messageTemp == "true")
    {
      useDebugLoop = true;
      ledSet(LED_ERR_DBUG);
      Serial.println("activado.");
    }
    else if(messageTemp == "false")
    {
      useDebugLoop = false;
      ledReset(LED_ERR_DBUG);
      Serial.println("desactivado.");
    }
  }
  if (String(topic) == "esp32/devModeAdvanced")
  {
    Serial.print("Modo avanzado ");
    if(messageTemp == "true")
    {
      useDebugAdvanced = true;
      Serial.println("activado.");
    }
    else if(messageTemp == "false")
    {
      useDebugAdvanced = false;
      Serial.println("desactivado.");
    }
  }
  if (String(topic) == "esp32/motorStatus")
  {
    if (messageTemp == "OK")
    {
      ledSet(LED_STATUS_GRN);
      ledReset(LED_STATUS_YLW);
      ledReset(LED_STATUS_RED);
    }
    else if(messageTemp == "ADVISE")
```

```
76        {
77          ledReset(LED_STATUS_GRN);
78          ledSet(LED_STATUS_YLW);
79          ledReset(LED_STATUS_RED);
80        }
81        else
82        {
83          ledReset(LED_STATUS_GRN);
84          ledReset(LED_STATUS_YLW);
85          ledSet(LED_STATUS_RED);
86        }
87      }
88    }
```

mqtt_callback.h

Benjamín Cháfer

```
1  #ifndef _CALLBACK_H
2  #define _CALLBACK_H
3
4  void callback(char* topic, byte* message, unsigned int msglength);
5
6  #endif
```

debug_signal.ino

Benjamín Cháfer

```
1   #include "debug_signal.h"
2   #include "adc.h"
3   #include "config.h"
4
5   float degToRad(float deg)
6   {
7     return deg * PI / 180;
8   }
9
10  int getSinFunction(float t, float A, int f, float phi)
11  {
12    float w = 2.0 * PI * f;
13    int result = A * sin(w * t + phi);
14    return result;
15  }
16
17   int advancedSinFunction(float t, float A, int f, float beta, float s)
18   {
19    float w = 2.0 * PI * f;
20    float i = A * cos(w * t) * (1 + beta * cos(2 * s * w * t));
21    return realToAdc(i);
22   }
```

debug_signal.h

Benjamín Cháfer

```
 1  #ifndef _DEBUG_H
 2  #define _DEBUG_H
 3
 4  #define DEBUG_F_A     2
 5  #define DEBUG_F_FREQ  50
 6  #define DEBUG_F_PHI   0
 7  #define DEBUG_G_A     0
 8  #define DEBUG_G_FREQ  30
 9  #define DEBUG_G_PHI   1
10  #define DEBUG_H_A     0
11  #define DEBUG_H_FREQ  20
12  #define DEBUG_H_PHI   1
13  #define DEBUG_OFFSET  2
14  #define DEBUG_BETA    0.1
15  #define DEBUG_S       0.05
16
17  #define DEBUG_SPEED   2985
18
19  int getSinFunction(float t, float A, float f, float phi);
20  int advancedSinFunction(float t, float A, int f, float beta, float s);
21
22  #endif
```