Proyecto Fin de Grado
Ingeniería de Tecnologías de Telecomunicación

ANALIZADOR DE FUNCIONES DE TRANSFERENCIA CON ARDUINO Y LABVIEW

Autor: Pedro Jesús Reina Varo
Tutor: Alfredo Pérez-Vega Leal

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

ANALIZADOR DE FUNCIONES DE TRANSFERENCIA CON ARDUINO Y LABVIEW

Autor:
Pedro Jesús Reina Varo

Tutor:
Alfredo Pérez-Vega Leal
Profesor titular

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Proyecto Fin de Carrera: ANALIZADOR DE FUNCIONES DE TRANSFERENCIA CON ARDUINO Y LABVIEW

Autor: Pedro Jesús Reina Varo

Tutor: Alfredo Pérez-Vega Leal

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A todos los que me habéis acompañado en esta etapa.

Agradecimientos

Escribir estas líneas, supone el cierre de una etapa fundamental en mi desarrollo personal e intelectual.

Quiero agradecer el apoyo incondicional de mis padres. Gracias por enseñarme estos años a valorar el esfuerzo, aunque los resultados no sean los esperados.

A mi hermana Chelo, por acompañarme estos últimos años de carrera y hacerlos mejores. Eres una referente. Sigue esforzándote porque al final de la escuela se sale.

A Belén por estar a mi lado siempre, ser mi confidente y mi compañera.

A mis abuelos, porque estos años en Sevilla me han permitido disfrutar de ellos en mi día a día.

A mis hermanitos del Perdón, por permitirme disfrutar de ellos para afrontar nuevos retos cada día.

A todos esos profesores que disfrutan de su profesión, se preocupan por la docencia y son el único motivo que nos ayuda a seguir a tantos y tantos estudiantes.

Y no puedo acabar sin acordarme de mis amigos del Telelimonar y mis Tabernícolas, y especial mención a mis machos, por tantas tardes de biblioteca y por enseñarme que el estudio no está reñido con la diversión y la amistad.

Pedro Jesús Reina Varo

Sevilla, 2019

Resumen

El propósito principal de este trabajo es el desarrollo de una aplicación que realice un análisis en frecuencia de la respuesta de cualquier sistema. Para ello con la ayuda de un microprocesador de bajo coste con interfaz Wi-Fi, se realizará un barrido en frecuencia para luego representar el diagrama de Bode del sistema.

Además del cálculo, se implementará un protocolo de comunicación basado en SLIP adaptado para la aplicación.

El sistema dispondrá de una jerarquía Cliente-Servidor donde son independientes cada uno de los extremos. Estos serán implementados en Labview y Arduino respectivamente.

Abstract

The main purpose of this work is the development of an application that makes a measurement of the frequency response of any system. To do this with the help of a low cost microprocessor with Wi-Fi interface, a frequency analysis will be performed to then represent the Bode diagram of the system.

In addition to the calculation, a communication protocol based on SLIP adapted for the application will be implemented.

The system will have a Client-Server hierarchy where each end is independent. These will be implemented in Labview and Arduino respectively.

Índice

Índice de tablas

Índice de ilustraciones

Acrónimos

IDE	Entorno de desarrollo integrado
TCP.	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
IoT	Internet of Things
USB	Universal Serial Bus
RAM	Random Access Memory
ROM	Read Only Memory
sen	Función seno
AP	Access Point
ADC	Analog-Digital Conversor
DAC	Digital-Analog Conversor
HTTP	Hypertext Transfer Protocol
TX	Transmisión
RX	Recepción
BAUD	Baudios
SLIP	Serial Line Internet Protocol

1 INTRODUCCIÓN

Desde los orígenes de la humanidad, los científicos e ingenieros se han basado en las matemáticas para modelar todos los fenómenos naturales, así como el funcionamiento de todo lo que ocurre a nuestro alrededor, con el objetivo de poder controlar o predecir esos sucesos desconocidos.

Ese afán por controlar el funcionamiento de las cosas, del conocimiento pleno de sus propios descubrimientos origina la aparición de una rama de la ingeniería que es la ingeniería del control.

Esta rama pretende conocer y modelar el funcionamiento de un sistema como una función matemática, para, gracias a ella, interpretar y predecir, las posibles respuestas que ofrece el mismo. Esta función se denomina ***Función de Transferencia,*** la cual se define como la relación entre la entrada y la salida en el dominio de Laplace de un sistema.

$$\boldsymbol{H(s) = \frac{Y(s)}{X(s)}}$$

En el campo de la electrónica, los ingenieros quieren saber cual va a ser la respuesta de su circuito ante determinados estímulos, es por ellos que es fundamental el conocimiento de esa Función para experimentar sobre el mismo.

En el siguiente documento, se expondrá el desarrollo de un analizador de funciones de transferencia a partir de una señal de entrada y de salida. Para ello con la ayuda de un microprocesador y el entorno de Ingeniería *LabVIEW* se implementará una aplicación Cliente-Servidor que permita analizar cualquier circuito y obtener el diagrama de Bode de la respuesta en frecuencia del sistema.

2 DESCRIPCIÓN DEL HARDWARE

2.1 ESP32-Devkit-v4

El ESP32 es un microprocesador de bajo consumo con un módulo wifi, fabricado por *Espressif*, fabricante de origen chino. Es el sucesor del ESP8266, ambos muy usados para aplicaciones de IoT y Wifi.

Se comercializa bien solo, o montado en un kit de desarrollo, que es el que se usará durante este trabajo. En concreto el kit *Devkit-v4,* que incorpora botones de Reset, interfaz micro USB, traductor USB a UART, pines de entrada y salida, entre otros componentes.

Este kit permite desarrollar fácilmente prototipos de las aplicaciones antes de fabricar el circuito específico. Es por lo que durante desarrollo de una aplicación, es común el uso de estos kits antes de tener un producto definitivo.

Las principales características de este microprocesador se resumen en la siguiente tabla:

Característica	***ESP32***
Procesador	***Tensilica Xtensa LX6 32 bit Dual-Core a 160MHz***
Memoria RAM	***520kB***
Memoria Flash	***Hasta 16MB***
ROM	***448kB***
Alimentación	***2.2V a 3.6V***
Consumo	***80 mA (promedio). 225mA (max)***
Coprocesador de bajo consumo	***Si. Consumo inferior a 150uA***
WIFI	***801.11 b/g/n (hasta +20 dBm) WEP, WPA***
Soft-AP	***Si***
Bluetooth	***V4.2 BR/EDR y BLE***
UART	***3***
I2C	***2***
SPI	***4***
GPIO	***11***
PWM	***16***
ADC	***18 (12bits)***

DAC	***2***
Timers	***3***
Ethernet	***10/100 Mbps MAC***

Tabla 1-1: Tabla de características del ESP32.

Además, el *pinout* de la placa es el siguiente:

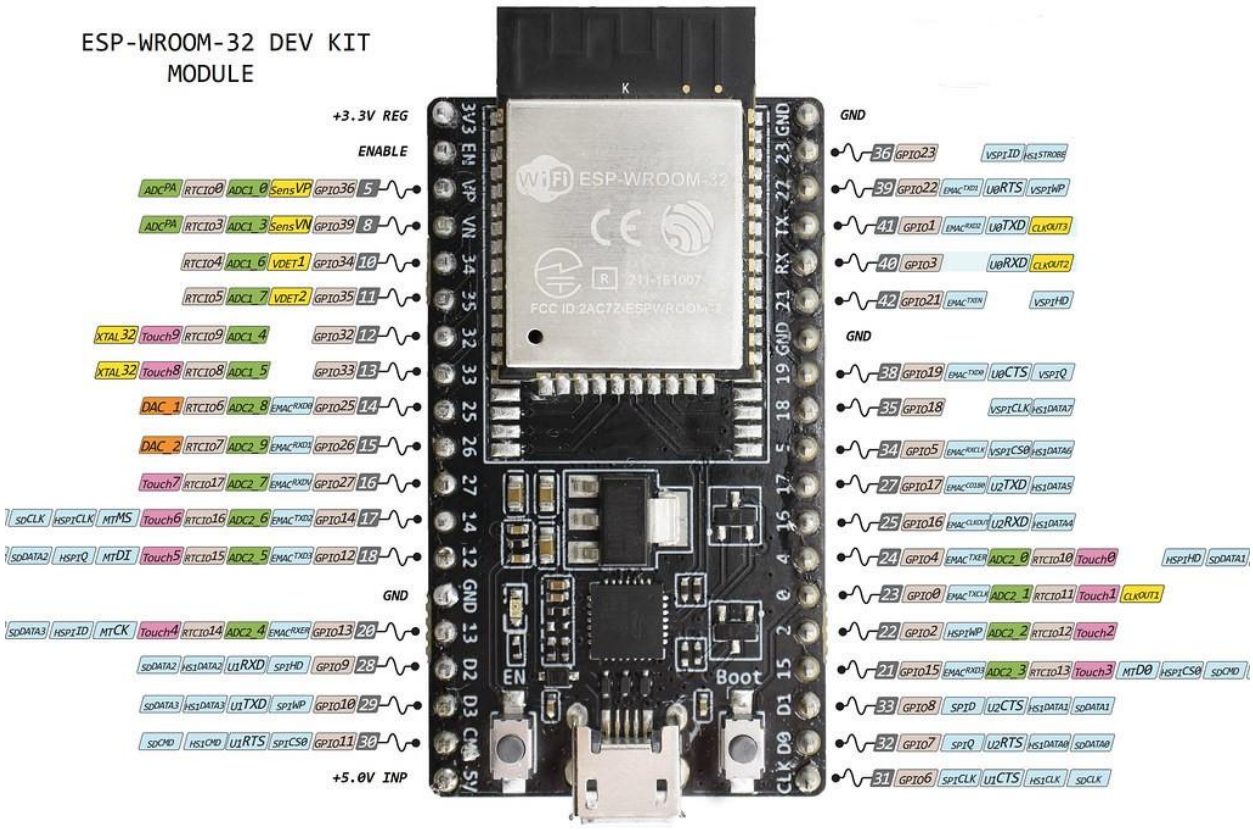

Ilustración 2-1. Pinout del ESP32 devkit v4.

2.2 Otros dispositivos

Para la implementación del cliente y el servidor, es necesario un ordenador que soporte el software específico para su diseño.

También se ha usado una protoboard para conectar los distintos pines que interaccionan del microprocesador.

Además, se han realizado las distintas simulaciones usando filtros RC de diferentes frecuencias, cables y diodos led.

También se ha usado un osciloscopio y un generador de ondas para comprobar las señales que genera el ESP32.

3 DESCRIPCIÓN DEL SOFTWARE

3.1 IDE de Arduino

Arduino es una compañía de software libre que se dedica al diseño y la manufactura de placas de desarrollo hardware. Esta compañía ofrece un IDE gratuito que facilita la programación de las placas de la compañía.

Aunque ESP32 es fabricada por Espressiff, Arduino[1] permite añadir librerías de placas de otros fabricantes para utilizar su entorno.

La comodidad de Arduino es que integra en una misma interfaz:

- Una ventana para escribir código:

  Esta contiene dos funciones, una función "*setup ()"*. Las instrucciones escritas en su interior son las primeras en ejecutarse. Está reservado para la configuración interna del micro (Configuración de registros, timers, pines, etcétera). Además, dispone de una función "*loop()",* que consiste en un bucle infinito en el cual se escribirán las instrucciones que debe de ejecutar el micro.

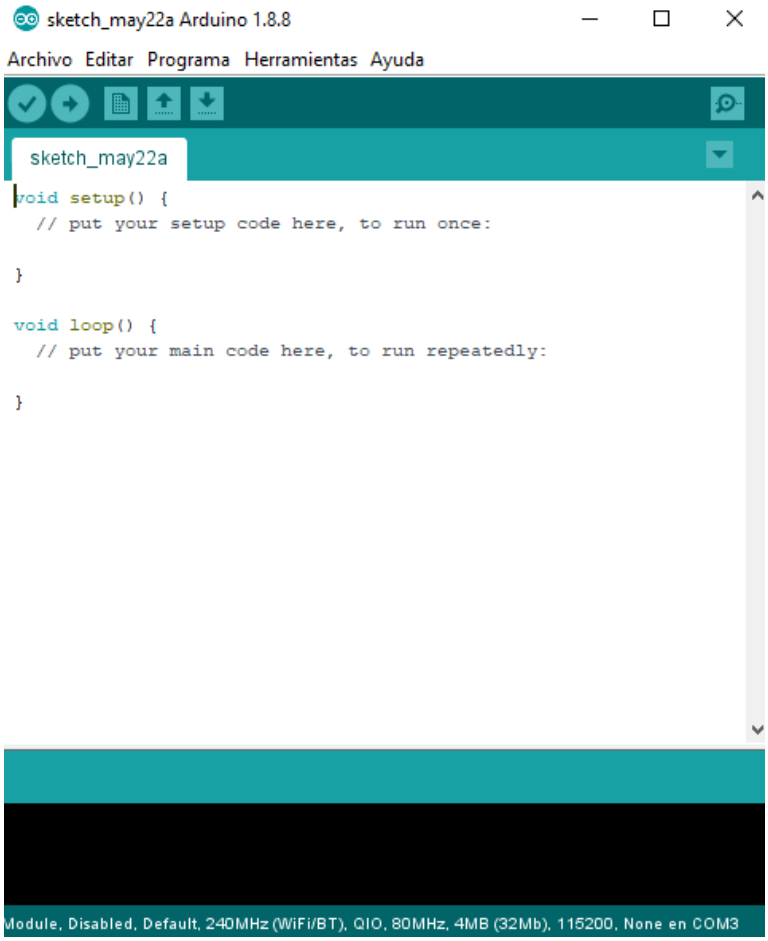

Ilustración 3-1. Ventana principal de Arduino.

[1] Durante el resto del texto, se hará referencia al IDE de Arduino indeferentemente como Arduino, servidor o micro.

- Un compilador: Interpreta el lenguaje de alto nivel de Arduino (Implementación de C++) y lo traduce a código máquina.
- Monitor Serie: Dispone de una consola para leer los mensajes transmitidos desde el Servidor por comunicación serie.

3.2 Entorno de ingeniería LabVIEW 2018

LabVIEW es un software de desarrollo y análisis enfocado a la ingeniería. Este entorno se basa en un lenguaje de programación gráfico tanto para aplicaciones hardware como software. Dispone de multitud de librerías que permiten el desarrollo de diversidad de aplicaciones.

Gracias a este software, es posible entre otras muchas cosas:

- La generación y representación de señales.
- El procesado de datos (Arrays, matrices, Strings) de todo tipo de variables (Float, Integer, caracteres, etcétera).
- Realizar comunicaciones tipo Serie, TCP/IP, UDP/IP, Bluetooth, HTTP, etcétera.
- Funciones para realizar cálculos complejos y análisis en frecuencia.
- Análisis, control y simulación de sistemas.
- Interacción con microprocesadores.

Además, hay toda una comunidad entorno a LabVIEW la cual constantemente desarrolla nuevos proyectos y aplicaciones, aumentando esas librerías.

El programa dispone de dos ventanas principales, las cuales son:

3.2.1 Ventana de display

En esta Ventana, se colocan todos los actuadores e indicadores que componen la aplicación (botones, gráficas, variables de entrada y salida, etcétera).

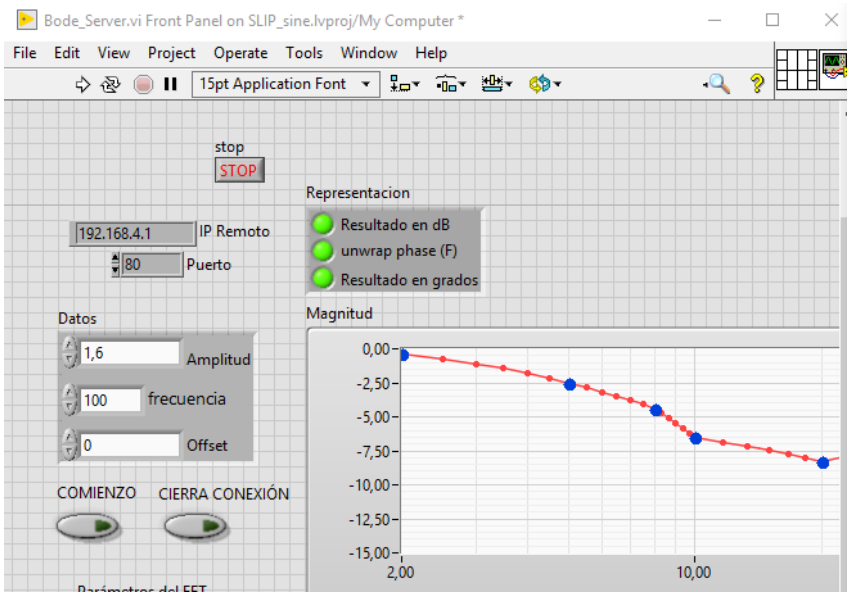

Ilustración 3-2: Vista de la ventana frontal de LabVIEW.

3.2.2 Diagrama de Bloques

Aquí es donde se programa la aplicación usando un entorno gráfico de diagrama de bloques.

La potencia de LabVIEW es, además de la multitud de librerías oficiales, una gran comunidad que aporta constantemente nuevas funciones y aplicaciones.

La programación es secuencial (de derecha a izquierda y de arriba a abajo).

En esta ventana, además, se encuentran las funciones de *debug*, en caso de hacer uso de *Breakpoints*.

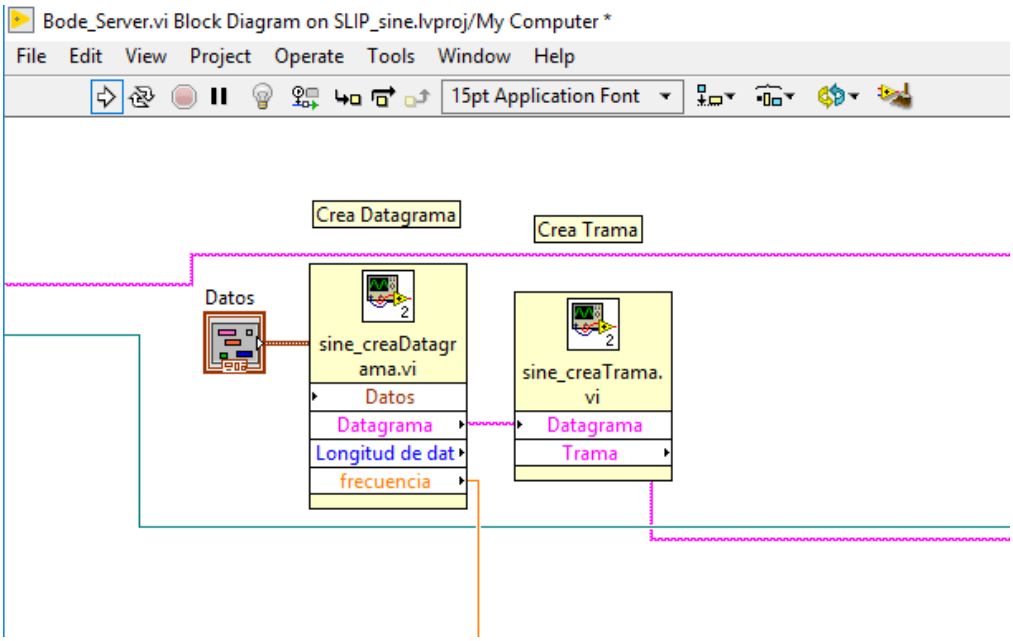

Ilustración 3-3: Vista de la ventana del Diagrama de bloques de LabVIEW.

Las dos últimas ilustraciones pertenecen son una parte de la aplicación que se describe en este texto.

4 PROTOCOLOS DE COMUNICACIÓN

Algún día todas las personas del mundo, tendrán un dispositivo de comunicación.

Graham Bell, 1922

Una parte fundamental de una aplicación cliente-servidor, es la comunicación entre ambos extremos. Esta comunicación debe ser fiable, fluida y eficiente.

En el siguiente capítulo se va a hacer una introducción teórica de los protocolos de comunicación usados en la aplicación.

4.1 Protocolo Serie

La comunicación serie se basa en el envío bit a bit de la información, de forma secuencial a través de un canal de comunicación o un bus.

Es muy usado en la comunicación entre dispositivos digitales por su sencillez.

Son necesarias dos líneas RX y TX respectivamente por la cual se recibe y transmite la información respectivamente.

Una configuración típica de las transmisiones UART, por ejemplo, en Arduino:

- 8 bits de datos.
- 1 bit de paridad.
- Sin bit de parada
- 1 bit de inicio.
- Velocidad a 112500 BAUD.

Es indispensable para la sincronización de los dos dispositivos que se comunican, que tengan la misma configuración del puerto Serie.

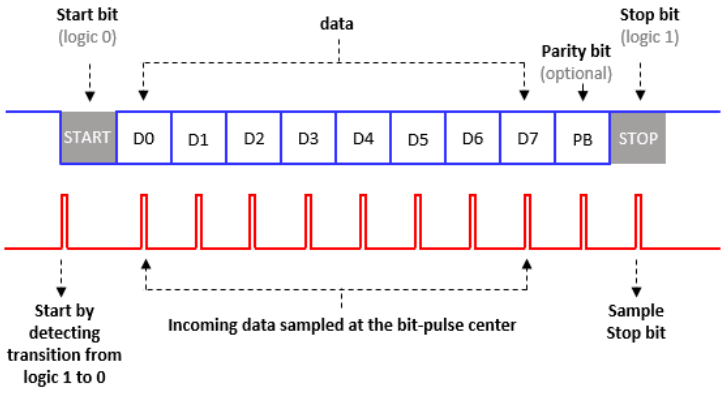

Ilustración 4-1: Trama del protocolo Serie.

Para nuestra aplicación, este protocolo se ha usado para para cargar los ficheros del compilador en el microprocesador, así como para generar mensajes de información al usuario desde el servidor, a través de la consola de Arduino.

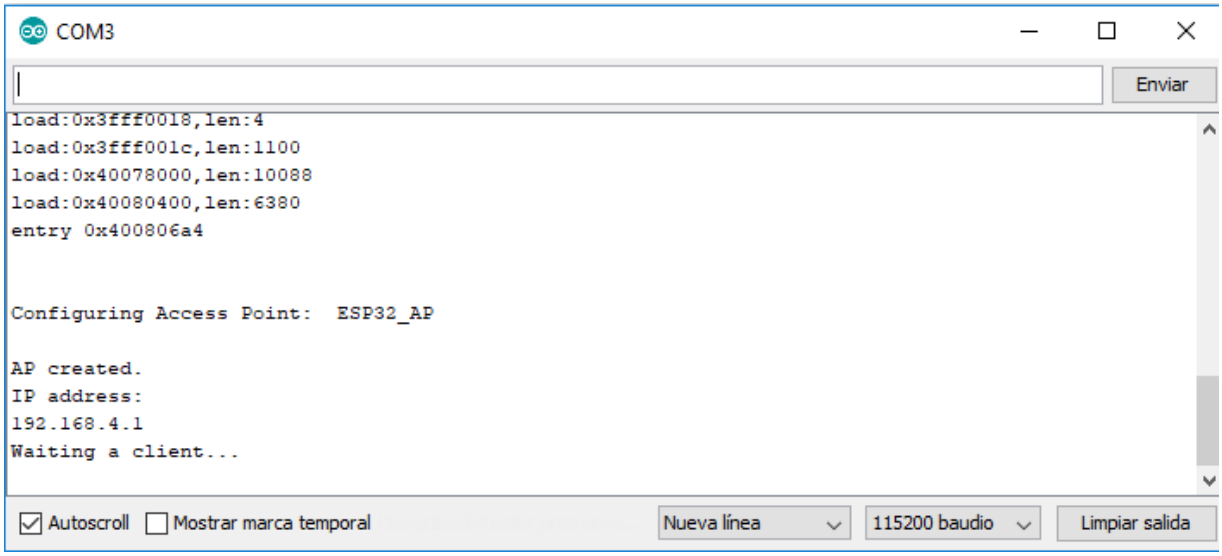

Ilustración 4-2: Consola del puerto serie del Servidor.

4.2 Wi-Fi

Wi-Fi es un estándar de comunicaciones, extendido a nivel mundial, el cual define la comunicación inalámbrica entre dispositivos a través de una red en las bandas de 2.4 o 5.2 GHz.

Estas redes se compondrán de Puntos de Acceso (AP) y estaciones (STA). Los AP, son dispositivos que generan una red WiFi a la que se conectarán las STA.

En esta aplicación, el ESP32, es el encargado de generar esa red, a la cual el ordenador se conectará para iniciar la comunicación.

En concreto, en Arduino, gracias a los métodos *softAP* y *softAPIP*, se puede configurar y conocer la IP local asociada. En la ilustración 4-2 se puede ver que se ha creado el AP.

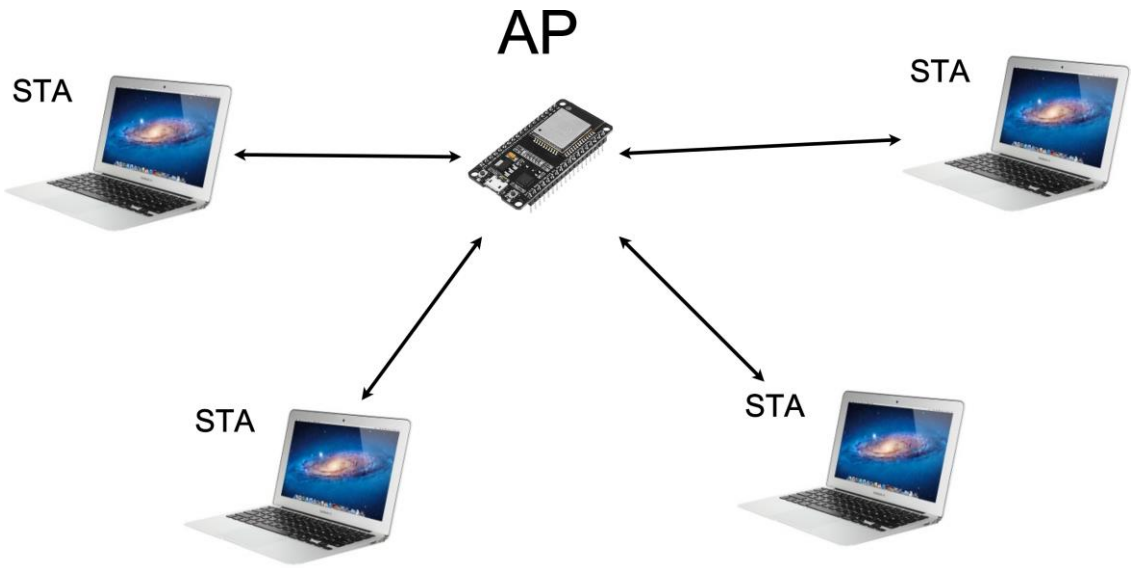

Ilustración 4-3: Esquema de AP y STA's.

La idea es que varios clientes (STA's) puedan conectarse simultáneamente al servidor (AP) por orden de llegada.

La comunicación entre el cliente y el servidor es bidireccional, pues el servidor se encuentra a la espera de una petición del cliente, en cuanto se realiza esa conexión, comienza el intercambio de paquetes de datos, cuyas tramas siguen el estándar SLIP, el cual se va a explicar a continuación.

4.3 Protocolo SLIP

El protocolo SLIP (*Serial Line Internet Protocol)*, es un protocolo de comunicación, a nivel de Enlace, extendido en el uso de microcontroladores que utilicen encapsulación IP. Es

El motivo por el que es muy utilizado en microcontroladores es por sus cortas cabeceras y poco procesamiento de los paquetes IP originales.

La transmisión tiene una configuración de envío serie de 8 bits de datos sin paridad. Se simula una transmisión serie para los paquetes IP. Para ello, los extremos se valen de varios caracteres especiales para diferenciar los paquetes.

El carácter "C0" se define como el fin de paquete o "SLIP END". Sirve para diferenciar el inicio de un paquete del siguiente. En la aplicación se ha añadido un "C0" al inicio y al final de cada trama de datos para detectar posibles fallos en la transmisión.

Otro carácter utilizado es el "DB" o "Carácter de escape".

Para el caso en el que la trama de datos presente el carácter "C0", este se sustituye por los caracteres "DB DD".

En el caso de contener el carácter "DB" en el datagrama, se sustituyen por los caracteres "DB DC".

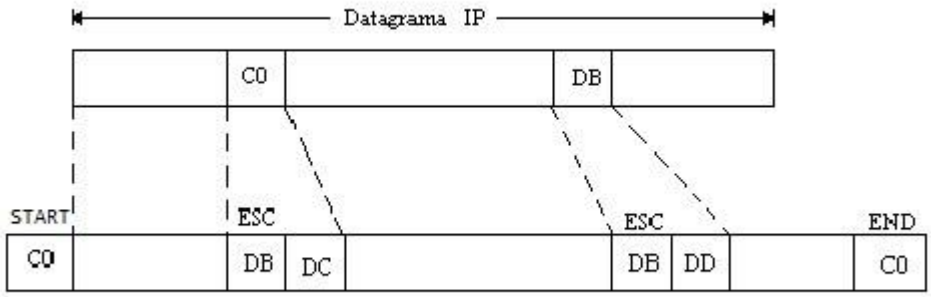

Ilustración 4-4: Formato trama SLIP.

Esta forma de transmitir los datos permite tener una transmisión de datos rápida y fácil de implementar. Posteriormente se explicará cómo se ha implementado el envío de datos la aplicación.

5 DISEÑO DE LA APLICACIÓN

Como anteriormente se ha explicado, la aplicación es una aplicación cliente-servidor donde el cliente está implementado en el ordenador usando LabVIEW y el servidor se implemente en el ESP32 con Arduino.

El cliente enviará los datos de las señales a generar al servidor, y este generará las señales de la petición. Se pasarán por un filtro Paso Bajo[2] y leerá las señales. Estos datos son enviados al cliente.

La comunicación entre ambos extremos se realiza a través de Wi-fi como se ha explicado anteriormente. Es necesario que el ordenador esté conectado a la red Wifi creada por el microprocesador.

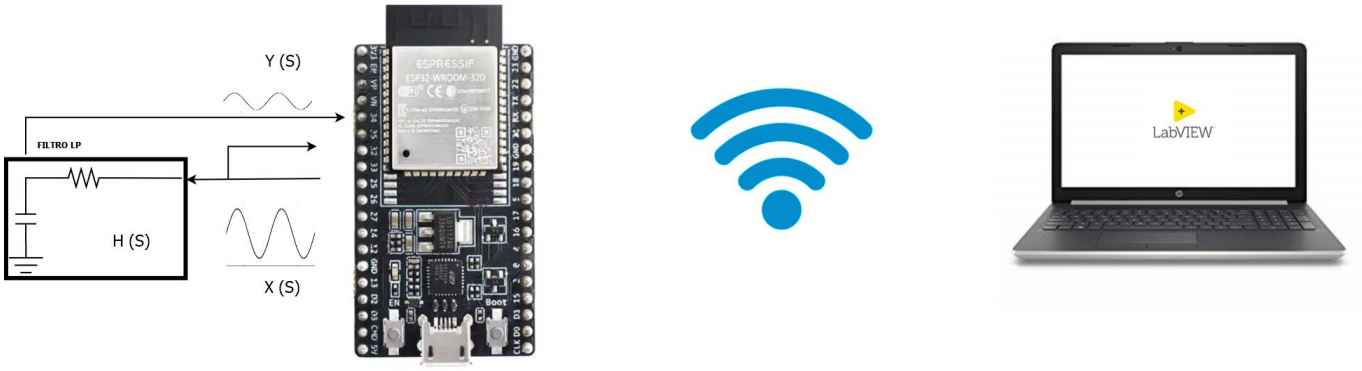

Ilustración 5-1: Diagrama de bloques de la aplicación.

5.1 Diseño del cliente

Para el diseño del cliente, se ha implementado una máquina de estados, haciendo uso de las estructuras "*While" y "Case"* del entorno LabVIEW.

Además, se han utilizado bloques (funciones), ya existentes en las librerías del programa, así como se han implementado bloques propios que ejecutasen tareas específicas.

El diagrama de flujo de la máquina de estados es el siguiente:

[2] En este ejemplo se ha realizado la prueba con un filtro paso bajo, pero la aplicación permite cualquier sistema.

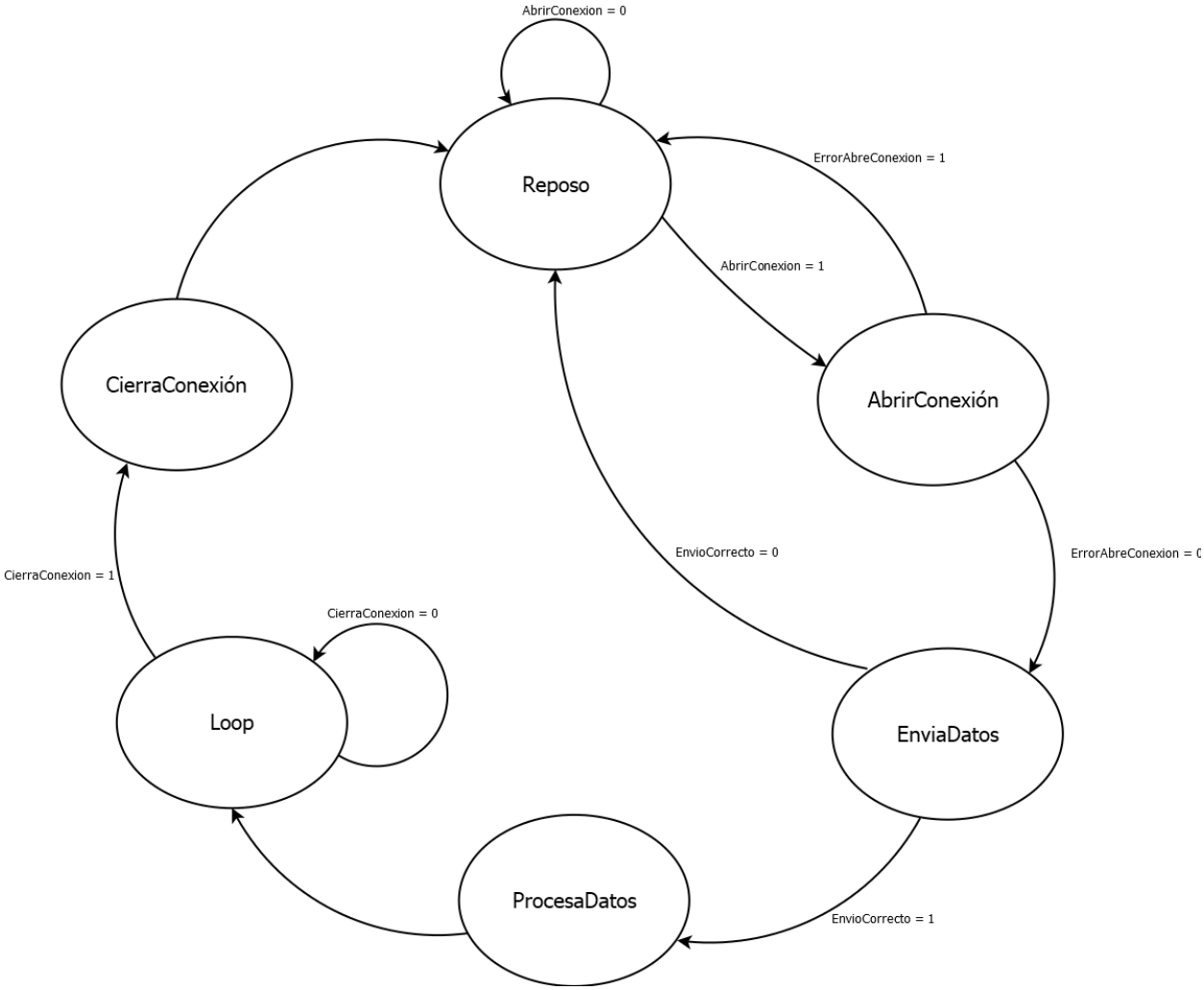

Ilustración 5-2: Máquina de estados del Cliente.

5.1.1 Reposo

Este es el estado inicial del cliente siempre y cuando no esté realizando ninguna función. El cliente se encuentra en espera activa, a que el usuario pulse el botón “Iniciar Conexión” del entorno gráfico del LabVIEW.

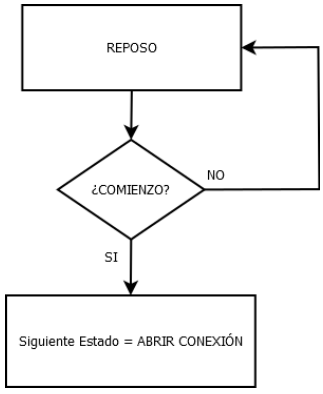

Ilustración 5-3: Diagrama de flujo del estado Reposo.

5.1.2 Abrir Conexión

En este estado, el cliente tratará de establecer conexión TCP con el servidor, haciendo uso del bloque funcional "TCP Open Conection". Este bloque recibe como parámetros la IP remota del servidor, el puerto remoto del servidor, así como un *timeout* para establecer la conexión. Este devuelve un *id* de la conexión establecida si ha sido satisfactoria, y un código de error en caso de fallo. El diagrama de bloques del estado es:

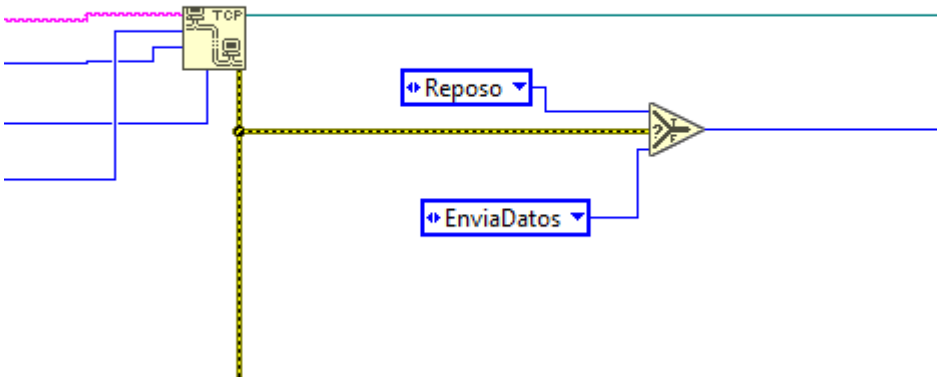

Ilustración 5-4: Diagrama de Bloques de AbrirConexión.

Tanto la IP como el puerto son introducidos por el usuario desde la interfaz gráfica mediante dos ventanas de texto.

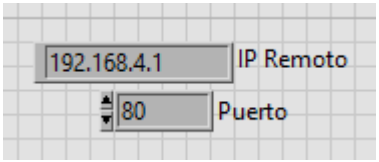

Ilustración 5-5: Configuración IP y puerto.

Una vez ejecutado el bloque de abrir conexión, en el caso de que no se genere código de error, se pasará al estado EnviaDatos. Si hubiese error, se vuelve al estado Reposo a la espera de otra petición de conexión. El diagrama de flujo del estado es el siguiente:

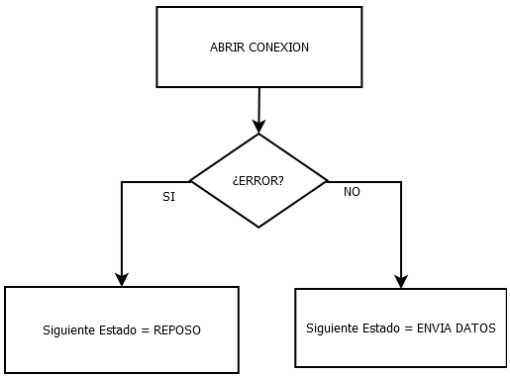

Ilustración 5-6: Diagrama de flujo estado Abrir Conexión.

5.1.3 Envia Datos

En este estado, el cliente creará un datagrama con la información que quiere mandar al servidor, encapsulará ese datagrama es una trama en formato SLIP y la mandará al servidor. El diagrama de bloques del estado es el siguiente:

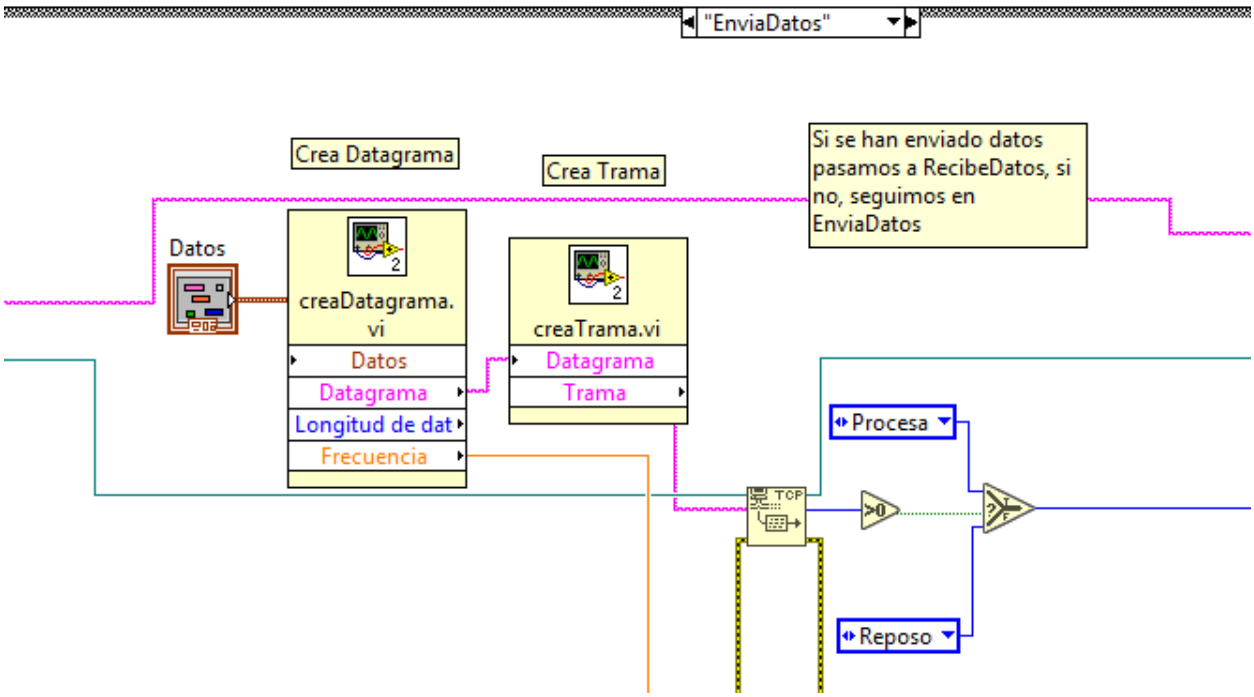

Ilustración 5-7: Diagrama de bloques EnviaDatos.

Los datos que se envían desde el cliente son: La amplitud y *offset* de la señal que tiene que generar, así como la frecuencia final del barrido que se debe realizar. Esta frecuencia del barrido no puede superar los 200Hz pues el microprocesador no admite un tiempo de muestreo superior manteniendo una relativa calidad de la señal.

Estos datos, se introducen en un cuadro de texto:

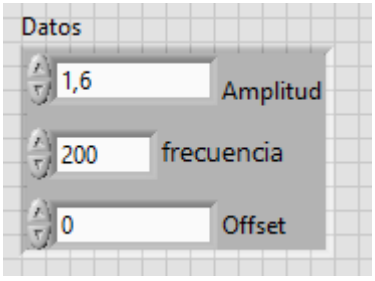

Ilustración 5-8: Cuadro de entrada de datos.

Esta información, representada en tipo flotante (4 bytes por dato), se pasa a un subbloque denominado "creaDatagrama" el cual recibe los datos en tipo flotante, y los pasa a una cadena de caracteres en Hexadecimal.

Esta cadena de caracteres en hexadecimal es parámetro de entrada de otro subbloque llamado "creaTrama". Este recibe el datagrama y lo procesa creando una trama SLIP, añadiendo los caracteres

"C0" y haciendo las sustituciones pertinentes. Devuelve otra cadena de caracteres.

Esta cadena de caracteres es pasada a la función "TCP Write" de LabVIEW, que transmite vía Wi-fi el datagrama hacia el servidor. Esta función *write* devuelve, además de un código de error, el número de bytes escritos. En caso de que el número de bytes escritos no coincida con el tamaño del datagrama, el programa cierra la conexión, y automáticamente vuelve al reposo.

En la ilustración 3-3, se observa la conexión de los dos subbloques explicados.

El diagrama de flujo seguido en el estado es el siguiente:

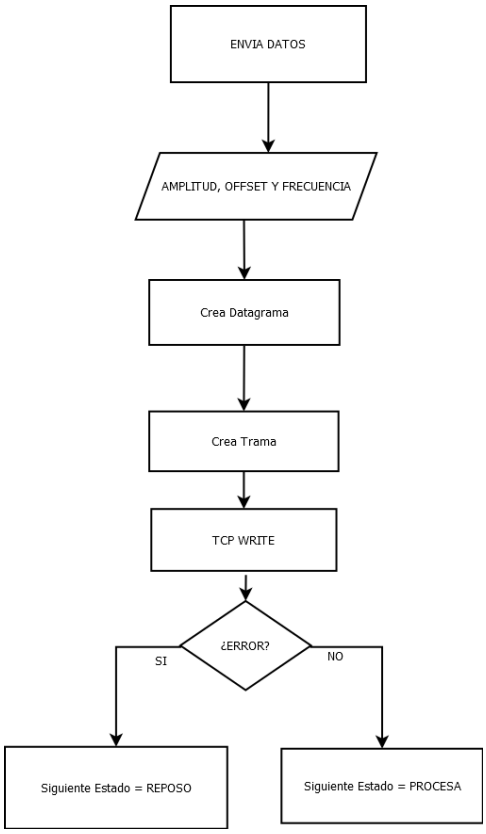

Ilustración 5-9: Diagrama de flujo de Envía datos.

5.1.4 Procesa Datos

En este estado es donde se ejecuta la mayoría de la aplicación. Este empieza con una escucha TCP mediante la función de LabVIEW "TCP Read", bloque que, configurado con un *timeout*, un *id* de conexión, y un tamaño de paquete recibido, devuelve un String con la información devuelta por el servidor.

La idea original para el envío de los resultados era la de realizar una comunicación fiable con el uso de un mensaje de asentimiento del cliente al servidor por cada trama recibida.

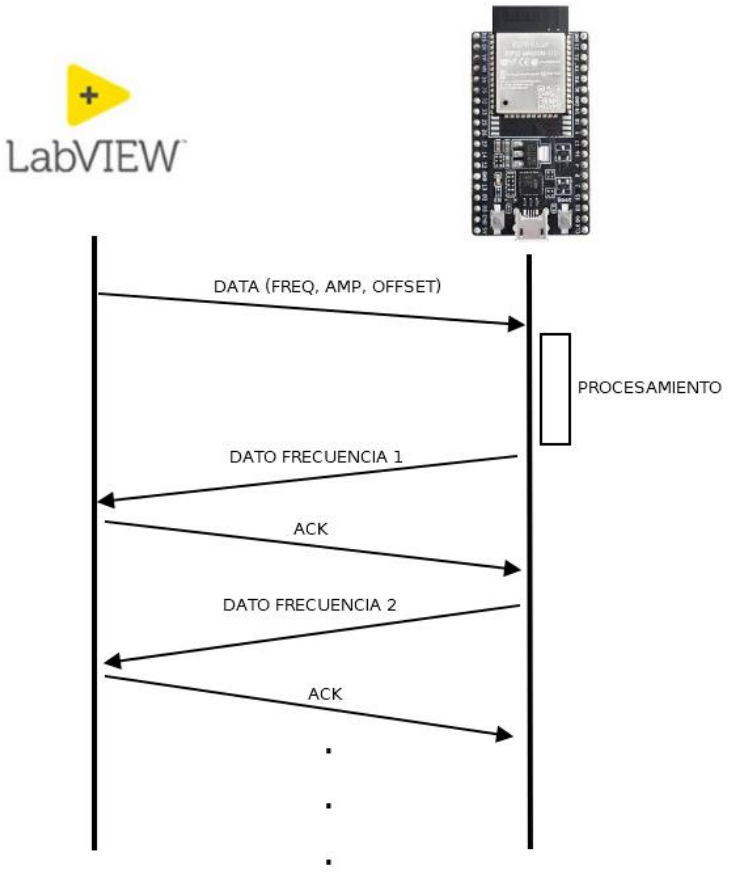

Ilustración 5-6: Paso de mensajes de la aplicación.

Esta solución parecía la más eficiente y elegante, pero problemas con la capacidad de la memoria del microprocesador y las funciones de lectura y escritura de LabVIEW, han hecho imposible que se implementara, por lo que finalmente, todos los datos desde el servidor se envían en una única trama sin asentimiento.

Esta trama recibida tiene un formato específico diseñado para la aplicación, de la siguiente forma:

AAAA	C0	Señal 1 entrada	C0
C0	Señal 2 entrada		C0
C0	Señal 3 entrada		C0
C0	Señal 4 entrada		C0
C0	...		C0
C0	Señal n entrada		C0
FFFF	C0	Señal 1 salida	C0
C0	Señal 2 salida		C0
C0	Señal 3 salida		C0
C0	Señal 4 salida		C0
C0	...		C0
C0	Señal n salida		C0

Ilustración 5-10: Formato de la trama recibida en el cliente.

Además de los caracteres “C0”, se han añadido dos indicadores al principio de cada bloque de datos para diferenciar los datos que corresponden a las señales de entrada con los datos que corresponden a las señales de salida. En concreto, la cabecera “AAAA” antes de las señales de entrada, y la cabecera “FFFF” antes de las señales de salida.

Estos datos, contienen las muestras discretas leídas por el microprocesador de la señal que genera, antes y después de pasar el filtro. La representación de las muestras son tipo *Float*, pero son recibidas en formato String.

A partir de aquí, secuencialmente, se irá procesando esta trama usando funciones implementadas explícitamente para la aplicación hasta llegar al diagrama de Bode final.

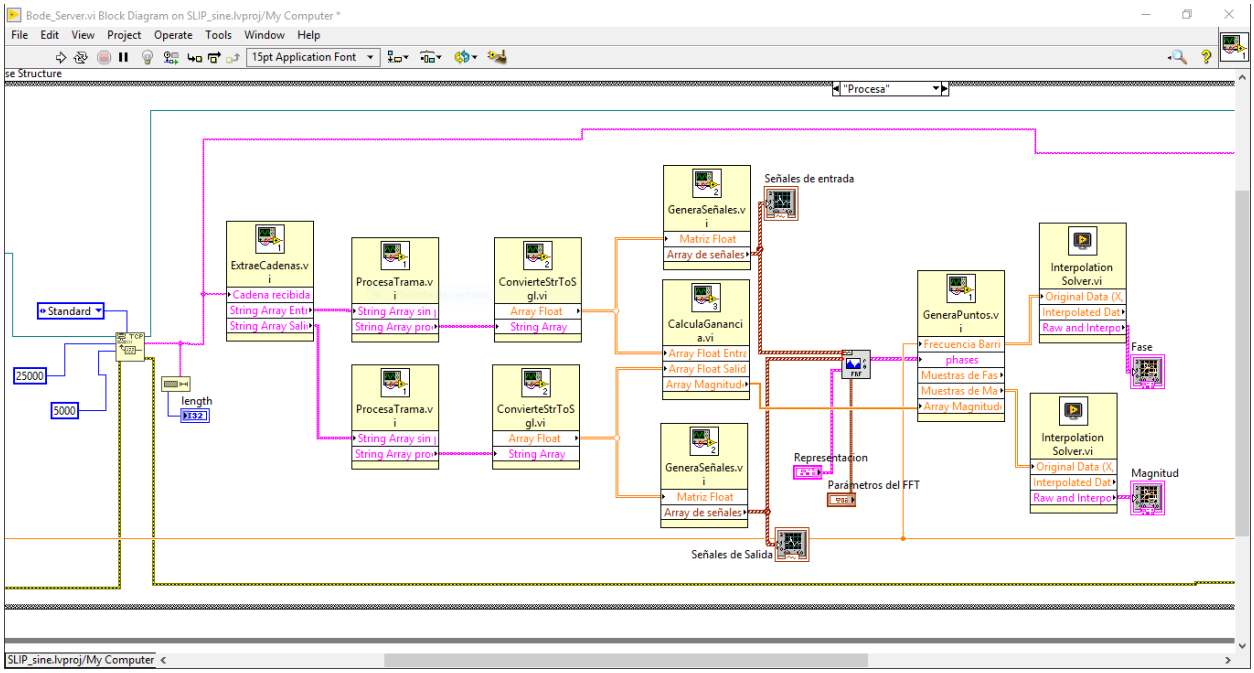

Ilustración 5-11: Diagrama de Bloques del estado Procesa.

En primer lugar, esta cadena de caracteres es pasada a la función llamada “*ExtraeCadenas”*, la cual devuelve dos arrays de cadenas de caracteres ayudándose de las marcas indicativas del comienzo de cada bloque. Un array contiene las cadenas de entrada y el otro las cadenas de salida.

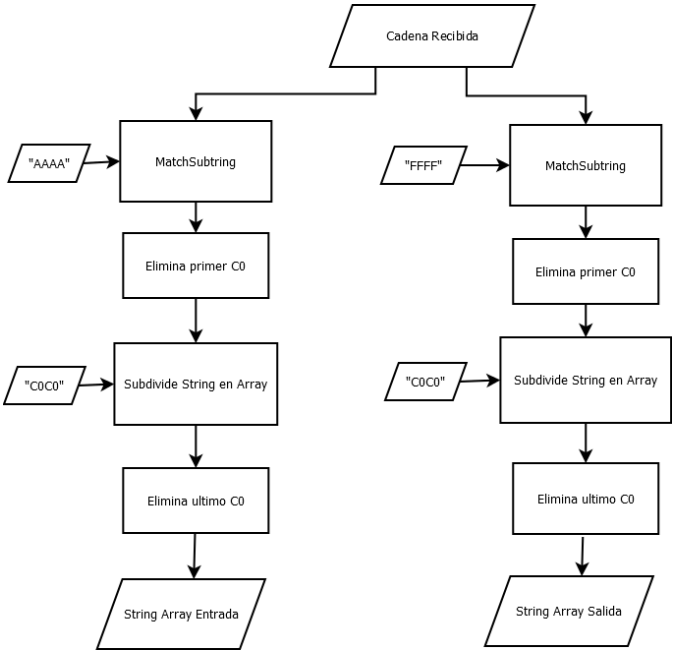

Ilustración 5-12: Diagrama de Flujo ExtraeCadenas.

Posteriormente, cada una de estos arrays, pasan por una función llamada "*ProcesaTrama"*. Este bloque, recibe un array de cadenas SLIP y devuelve un array de trama de datos, es decir, sustituye los caracteres "DB DC" y "DB DC" por "C0" y "DB" respectivamente en cada uno de los String que recibe. En este instante, la información ya está codificada tal y como se generó en el servidor.

El diagrama de flujo de la función es el siguiente:

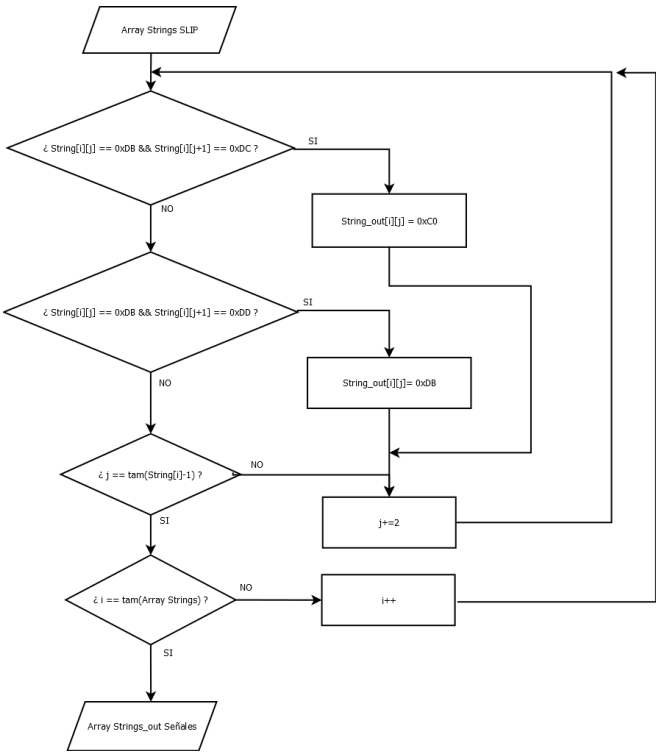

Ilustración 5-13: Diagrama de flujo de ProcesaTrama.

En este punto, hay que representar estos octetos de datos de manera que el programa los agrupe de cuatro en cuatro y los represente en formato *float,* en concreto de simple precisión. Para ello se introducen los arrays en otro bloque llamado "*ConvierteStrToSgl",* bloque, que, en su interior, mediante el uso de bucles y las funciones de LabVIEW *"StringSubset",* que extrae los datos en grupos de cuatro bytes, y la función "*TypeCast"*, que realiza el casteo necesario, se obtiene a la salida una matriz de valores *floats*, donde cada fila[3] corresponde a los valores discretos de una señal.

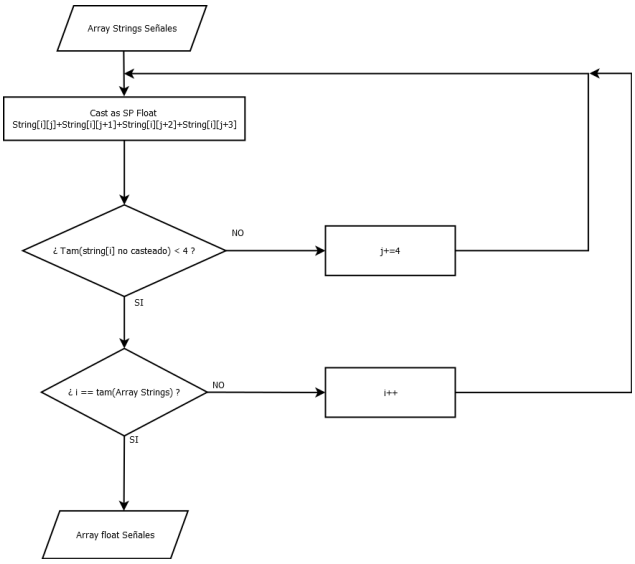

Ilustración 5-14: Diagrama de flujo ConvierteStrToSgl.

Estas matrices generadas (una con las señales de entrada y otra con los de salida), son argumento de dos bloques distintos.

Uno se llama "*genera_señales"*, el cual genera un array de matrices temporales mediante el bloque *"Build Waveform"*. Este recibe un array se valores discretos y el tiempo de muestreo, y devuelve una señal continua en el tiempo, la cual puede ser procesada por bloques específicos para el análisis de señales de LabVIEW.

[3] Destacar que el número de datos de cada señal varía, pues la frecuencia de muestreo del micro es fija, y conforme aumenta la frecuencia de la señal, la trama de datos es menor

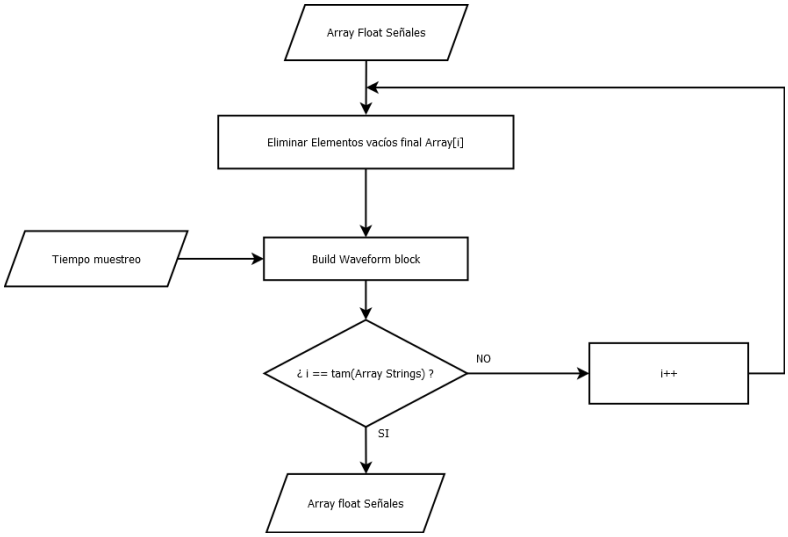

Ilustración 5-15: Diagrama de flujo GeneraSeñales

En concreto haciendo uso del bloque *"Frequency Response Function",* se obtiene el desfase entre las señales de entrada y salida para cada frecuencia. Este bloque tiene varios controladores para decidir las unidades de la salida.

Este bloque de análisis en frecuencia también devuelve las magnitudes de las frecuencias recibidas, pero el resultado obtenido era poco preciso, por lo que se ha optado por calcular la ganancia en un bloque independiente.

Para el cálculo de la ganancia, las dos matrices son pasadas como argumento a una función llamada "*Calcula Ganancia",* la realiza la siguiente operación, siendo Y la señal de salida, y X la señal de entrada.

$$\boldsymbol{Gain(dBV) = 20 * Log(\frac{Ymax - Ymin}{Xmax - Xmin})}$$

Una vez que se tienen los valores discretos de la magnitud y la fase, estas son pasadas como argumentos a la función "*GeneraPuntos"*, la cual asocia estos valores a la frecuencia a la que pertenecen, generando los puntos de la gráfica que van a ser representados.

Una vez se tienen los puntos discretos tanto de la magnitud como de la fase, se pasan al último bloque del estado. Este bloque ha sido una modificación de un de las librerías de LabVIEW y es el bloque *"Interpolation Solver",* el cual recibe una serie de puntos discretos, y representa las gráficas interpolando entre los puntos recibidos.

5.1.5 Loop

El estado *loop* es un estado muy simple. El cliente entra en un bucle infinito a la espera de la pulsación del botón "CierraConexión".

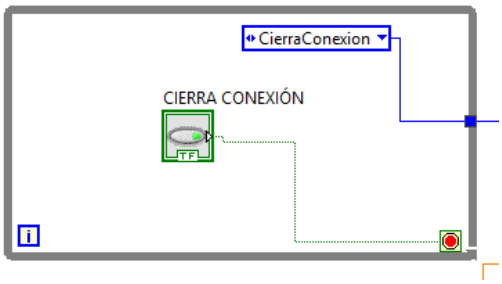

Ilustración 5-16: Estado Loop.

Mientras estamos en este estado, el cliente tendrá representado el diagrama de Bode en la ventana principal. Una vez se pulse, el programa pasa al estado de CierraConexión.

5.1.6 CierraConexión

Este estado hace uso del bloque "*TCP Close Connection"*, el cual recibe como parámetro la *id* de la conexión y la cierra. Una vez cerrada, se pasa al estado de Reposo a la espera de nuevas peticiones.

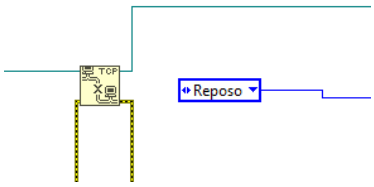

Ilustración 5-17: Estado CierraConexión.

5.2 Diseño del servidor

El servidor de la aplicación, a diferencia del cliente, programado por bloques, ha sido programado en el entorno Arduino, el cual es una variación de C++.

Para el diseño, al igual que el cliente, se ha implementado una máquina de estados con el objetivo de dividir la aplicación en tareas independientes.

La máquina de estados que se ha implementado es la de la siguiente ilustración:

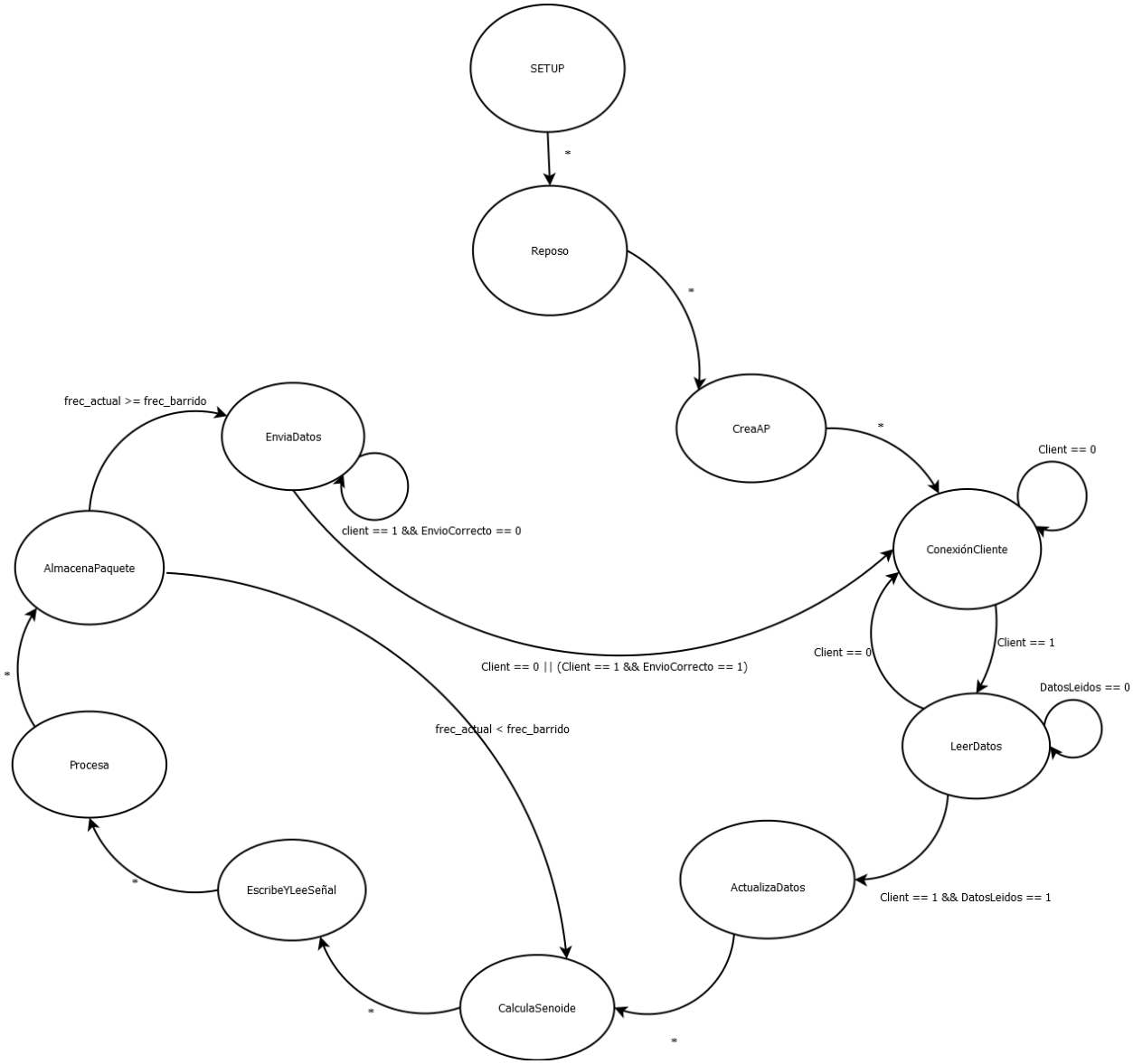

Ilustración 5-18: Máquina de estados del Servidor

Se ha hecho uso de librerías desarrolladas en la nube, ya que las librerías de Arduino son software libre, accesible para toda la comunidad. En concreto se han usado las librerías:

- WiFi.h: Métodos relacionados con la conexión WiFi.
- WiFiClient.h: Métodos relacionados con la gestión de clientes TCP o UDP.
- WiFiAP.h: Métodos relacionados con la configuración del ESP32 como un AP.
- Arduino.h: Métodos generales para el manejo de un microprocesador (Configuración de pines, interrupciones, escritura/lectura de pines, etc., DAC/ADC, …).

Además, se han implementado tres funciones propias:

- cambiaEstado: Configura la transición de estado.
- extraeTramaSLIP: Extrae los datos recibidos SLIP.
- creaTramaSLIP: Empaqueta como una trama SLIP los datos que se le devuelven al cliente.

Estas funciones se explicarán con detalle más adelante.

Todo el programa se va a ejecutar una vez para cada frecuencia que se va a realizar el barrido.

Se han tomado cinco muestras por década, las cuales son suficientes para después interpolar las gráficas. Estas frecuencias se guardan en la variable global *frecuencias_barrido* y se accede a ellas con el índice global *k*.

5.2.1 SETUP

El estado Setup, se ejecuta automáticamente al iniciar un programa Arduino. En él se configuran los registros del microprocesador, interrupciones y variables globales que se vayan a usar durante la ejecución del programa.

En esta aplicación, el Setup se ha usado para configurar el *Timer* 1 del ESP32 para generar las señales senoidales con la frecuencia deseada.

En concreto el *Timer* 1 tiene una frecuencia de 80MHz, por lo que se configura con el escalador 80 con la función *timerBegin (),* para que cada cuenta corresponda a 1 microsegundo.

Con esta configuración, se ajusta el tiempo de muestreo como:

$$Tm = \frac{1.000.000}{FM\_PLACA}$$

Esta FM_PLACA, es la frecuencia máxima de muestreo de la placa, configurada a 1.2kHz. Esta frecuencia de muestreo acota la frecuencia final del barrido a 200Hz (6 muestras por periodo).

El *Timer* es activado y desactivado en tiempo de ejecución con la frecuencia de la señal que se desea generar.

5.2.2 Reposo

El estado reposo es un estado transitorio entre el estado SETUP y el estado CreaAP.

5.2.3 CreaAP

En este estado, el microprocesador se configura como punto de acceso WiFi, con ello se consigue tener una aplicación que no necesita de redes externas para poder funcionar.

Esta red generada es local, exigiendo que el ESP32 y el ordenador se encuentren relativamente cerca, y sin tener acceso a internet.

También se configura la velocidad del puerto serie a 115200 baudios.

Inicialmente, la aplicación estaba configurada usando una red Wifi generada por otro dispositivo (Router), consiguiendo tener una aplicación que hacían al cliente y al servidor totalmente independientes.

Esta versión se descartó, pues, aunque se consiguió implementar, la robustez de la comunicación era mala, por lo que provocaba constantes fallos de conexión así, teniendo un índice de éxito muy bajo en cada petición y obligando a reiniciar los extremos de conexión constantemente.

Además, trabajar con el ESP32 como AP permitía trabajar en entornos en los que no se tenía acceso a internet.

En este estado, el servidor muestra por la interfaz de la consola el estado del AP, así como su dirección IP local. En la ilustración 4-2, se puede observar la salida generada por puerto Serie.

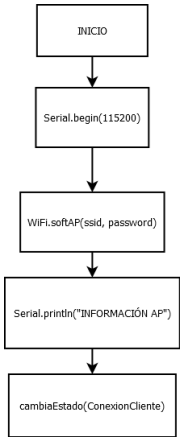

Ilustración 5-19: Diagrama de flujo del estado CreaAP.

5.2.4 ConexiónCliente

En estado el programa se queda en espera activa hasta que recibe una petición de conexión. Muestra por consola el mensaje *"new client"*.

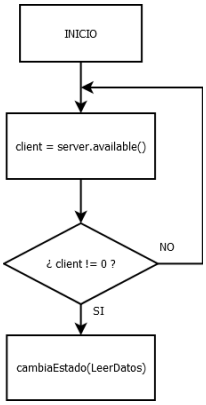

Ilustración 5-20: Diagrama de flujo del estado ConexionCliente.

5.2.5 LeerDatos

Una vez que un cliente contacta con el servidor, el cliente envía los datos relativos al tipo de análisis que quiere, mediante tres datos: Amplitud de la señal de entrada, *offset* de la señal y la frecuencia final del barrido. El servidor, en este estado, espera recibir estos datos.

Estos datos se leen byte a byte mediante la función *client.read()* y son almacenados en un buffer llamado *datosLeidos*. Una vez se llena el buffer, se comprueba si los datos recibidos siguen el formato de una trama SLIP. El tamaño de esta trama no es fijo, pues se esperan tres variables de tipo *float*, y dos cabeceras, pero no se saben cuántas sustituciones se han realizado en el cliente. Es por ello por lo que se ha reservado un buffer de 28 *bytes.*

Si la transmisión es satisfactoria, este buffer *datosLeidos,* es pasado como argumento a una función implementada para la aplicación llamada *ExtraeTramaSLIP()*. Esta función recibe una cadena SLIP, y la referencia a una cadena de caracteres vacía, así como el tamaño de ambas cadenas. La función actualiza el valor de la cadena *trama* con el valor del buffer después de extraer la trama SLIP (elimina "C0" y "DB's").

En el caso de que estos datos vengan alterados por un fallo en la transmisión, esta trama es desechada, cerrada la conexión con el cliente, y se vuelve al estado ConexiónCliente.

El diagrama de flujo del estado es el siguiente:

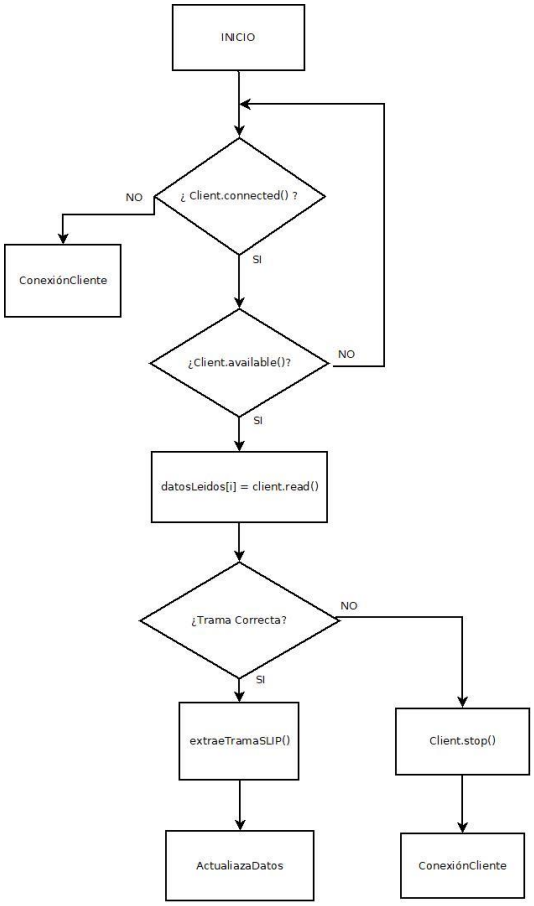

Ilustración 5-21: Diagrama de flujo del estado LeerDatos.

5.2.6 ActualizaDatos

En este estado, la trama de datos es interpretada como un *float single precision* (cuatro bytes), a través del uso de una unión que contiene una tabla de 4 bytes y una variable float.

La trama recibida tiene el siguiente formato:

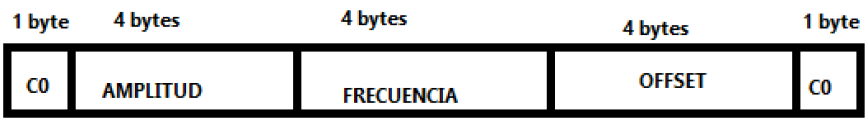

Ilustración 5-22: Formato de trama recibida en el servidor.

Esta trama sí tiene un tamaño predefinido, por lo que se ha hecho el casteo por asignación directa. Una vez se tienen tres variables globales con los valores recibidos, pasamos al estado para calcular la

senoide.

5.2.7 CalculaSenoide

En este estado, se van a calcular todos los valores discretos que compondrán la senoide, valores discretos que variarán en función de la frecuencia que estemos analizando, pues a mayor frecuencia, menos muestras tendrá la señal.

Para ello, inicialmente se calculan el número de muestras que tendrá la senoide para la frecuencia en cuestión. El número de muestras, L, viene determinado por:

$$L = \frac{FM_{PLACA}}{frec\_actual}$$

Para guardar los valores discretos, se crea un array a nivel global de *floats*, con el tamaño del número de muestras del periodo menor que se va a analizar.

Esta senoide que se va a generar, estará formada por *n* valores discretos. Arduino, representa los valores de salida en tensión como un entero entre 0 y 255, siendo 255 la máxima tensión que genera la placa que es 3.3V.

Para evitar que la señal saturase en cualquier momento, se ha dejado 0.1V de margen de seguridad, por lo que nuestra señal se encontrará centrada en 1.6 (Los micros no pueden generar tensiones negativas) y tendrá una V_{pp}= 3.2V

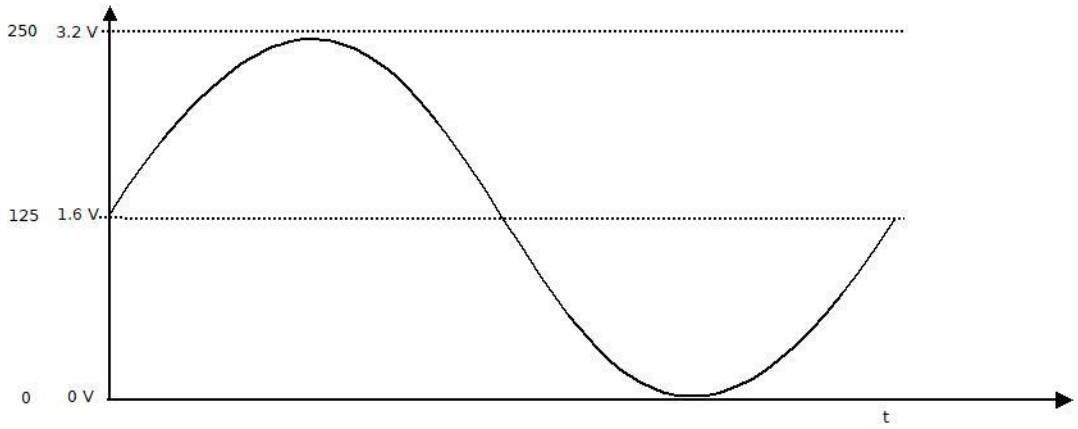

Ilustración 5-23: Senoide calculada.

Cada muestra sigue la función:

$$senoide_n = \frac{125 + A * \frac{125}{1.60} * Sen\left(2\pi * \frac{n}{L}\right)}{250}\ [V]$$

Este valor es calculado para cada valor de n en el periodo y se consiguen las L muestras de la senoide, la cual será la señal que pasemos por el filtro. La escritura de estas muestras con una frecuencia concreta se realizará gracias al *timer* configurado en el setup en el siguiente estado.

Inicialmente, se pretendía realizar el cálculo del valor discreto, así como la escritura de manera simultánea, pero problemas de capacidad de procesamiento del ESP32, hacía inviable generar las

señales de alta frecuencia con una calidad razonable.

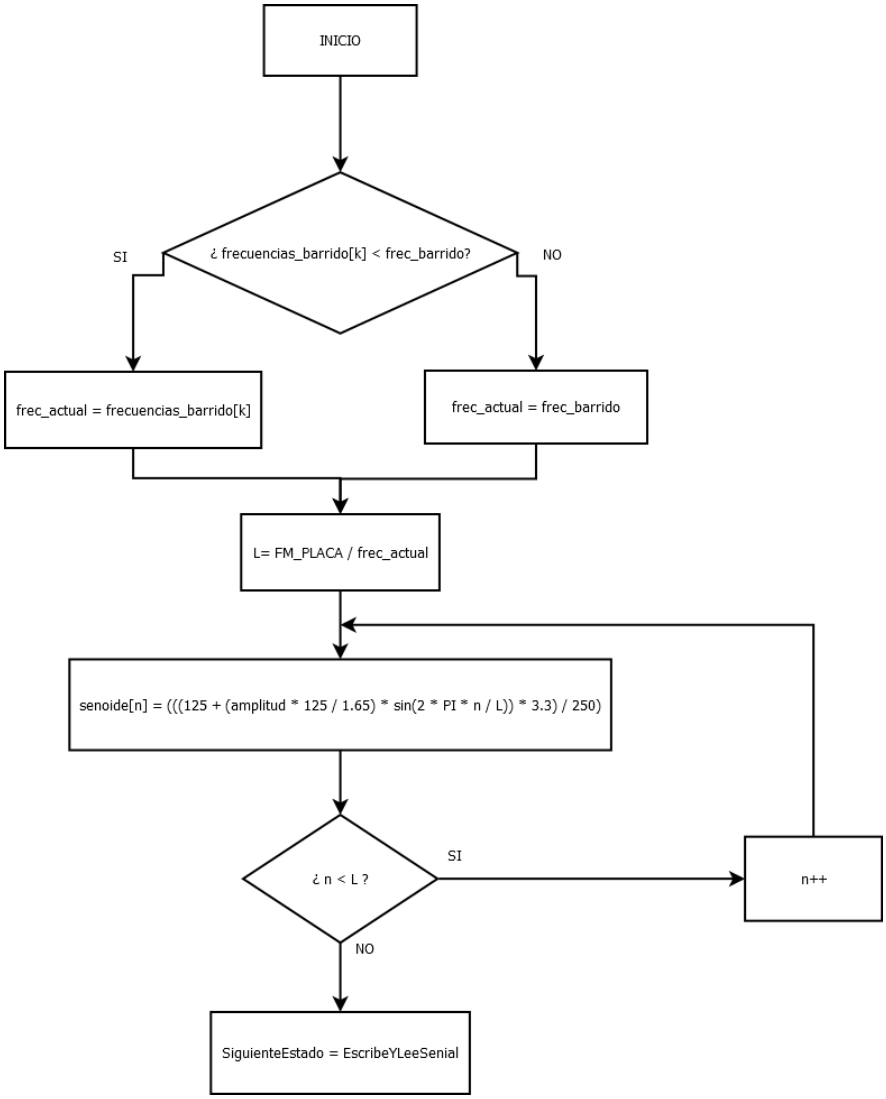

Ilustración 5-24: Diagrama de flujo estado GeneraSenoide.

5.2.8 EscribeYLeeSenial

Una vez calculados los valores de amplitud, toca generar la senoide y hacerla pasar por el filtro para captar la entrada y la salida para el cálculo del diagrama de Bode que se busca.

Para ello, en primer lugar, se activa el timer configurado previamente con el periodo de muestreo de la frecuencia deseada, y el programa se queda en espera activa dentro del estado. También se configura el pin 26 como pin de salida y los pines 32 y 35 como pines de entrada.

Como ya ha sido configurado, cada Tm segundos, la interrupción del timer se ejecuta, activando la variable *flag*. En ese instante, el programa ejecuta una vez las funciones *dacWrite(),* para escribir el valor discreto en el pin configurado, y la función *AnalogRead()*, la cual lee un valor entre 0 y 3.3 de los pines 32 (misma señal escrita) y 35 (señal de salida del sistema).

El motivo por el que se ha decidido leer la señal de entrada en vez de usar la ya calculada previamente es para que ambas señales (entrada y salida) tuviesen el mismo ruido de cuantización y se contrarrestase.

Otra decisión que se ha tomado para obtener la medida más fiable posible ha sido la de generar varios periodos de la señal. El número de periodos que se generan se configuran en la macro *NUM_PERIODOS.* En el caso de ejemplo, se han tomado cinco periodos por cada señal.

De estos cinco periodos, solo son leídos y almacenados los periodos tercero y cuarto con el objetivo de evitar transitorios de la señal y que esta estuviera estabilizada.

Todos estos datos son almacenados en los arrays de floats *sen_entrada* y *sen_salida.*

Es fundamental el índice usado para recorrer estos arrays, *j,* pues nos permite al final del estado saber el tamaño real de la señal generada, pues como se ha dicho antes, a mayor frecuencia, las señales tienen menos muestras.

Una vez las señales han sido muestreadas en las variables, se detiene el timer para que, en la próxima iteración, se vuelva a activar con la nueva frecuencia y se pasa al estado PROCESA.

Todo lo explicado anteriormente, se resume en el diagrama de flujo siguiente:

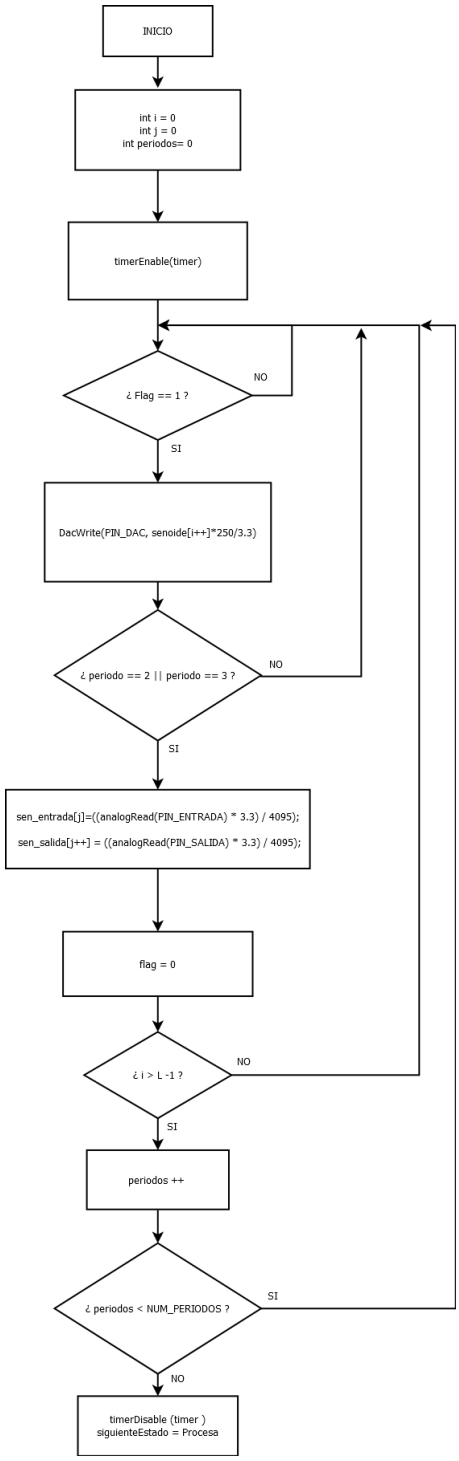

Ilustración 5-25: Diagrama de flujo del estado EscribeYLeeSenial.

5.2.9 Procesa

El estado procesa es el encargado de hacer el casteo de estos floats a bytes, en codificación Hexadecimal, con el objetivo de generar una trama SLIP tal y como la descrita en el apartado del Cliente.

Mediante el uso de la misma unión del estado ActualizaDatos, se van almacenando estos datos en dos *bytes arrays* llamados *trama_entrada* y *trama_salida.* El tamaño de estos arrays como es lógico es cuatro veces mayor que los *arrays* de *floats.*

Una vez terminado esta operación, se pasa al estado AlmacenaPaquete.

5.2.10 AlmacenaPaquete

El objetivo fundamental de este estado es generar la trama SLIP con todas las señales de entrada y salida, con sus respectivas sustituciones.

Como se comentó en la descripción del cliente, la idea original era la de enviar cada señal de manera independiente a la espera de un ACK por parte del cliente. Esto se implementó y desde la parte del servidor funcionaba de manera correcta. Problemas en el cliente, hicieron descartar esta solución.

Es por ello por lo que finalmente se ha utilizado la solución de hacer un único envío de datos con un solo paquete.

El problema que conlleva esto es que la memoria del microprocesador es limitada, y arrays de datos de gran tamaño saturan con facilidad la memoria.

Lo lógico sería pensar (y se ha intentado) el uso de memoria dinámica para un uso más eficiente de los recursos del micro. Funciones como *malloc(), o calloc()*¸ eliminarían este problema.

El problema que encontramos es que Arduino es un lenguaje de muy alto nivel, por lo que no permite el uso de estas funciones, generando problemas de ejecución.

Finalmente, en unos *arrays* llamados *paquete_salida_pro* y *paquete_entrada_pro*, con la ayuda de los índices que permiten saber el tamaño exacto de datos, así como haciendo uso de la función propia *creaTramaSLIP()*, se ha conseguido un programa relativamente óptimo, haciendo uso de memoria estática. Esta función, va almacenando las señales en una trama que recibe por referencia, quedando al final de la última iteración un solo paquete para todas las señales de entrada y otro para todas las señales de salida. También, añade los caracteres de escape “C0”, así como hace las sustituciones pertinentes.

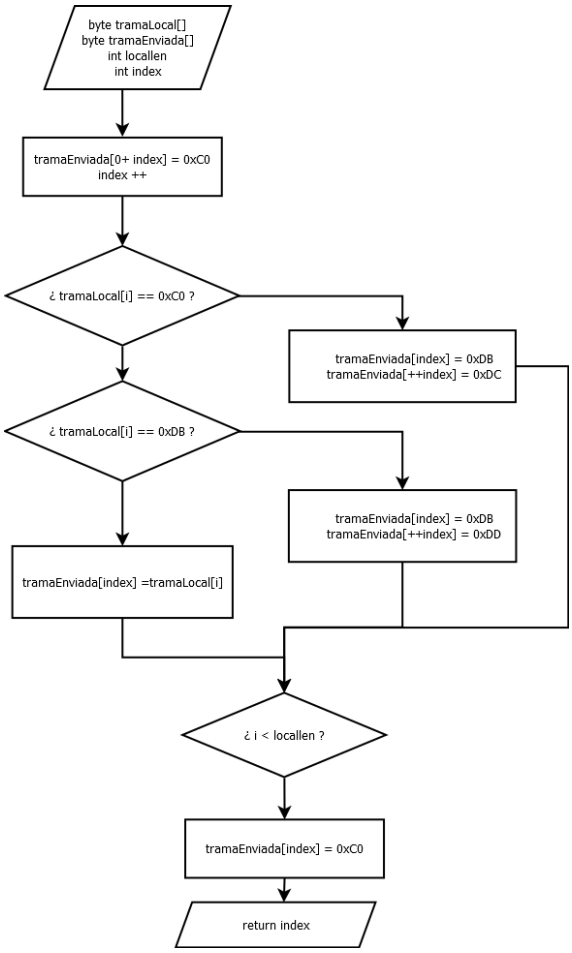

Ilustración 5-26: Diagrama de bloques función creaTramaSLIP.

El diagrama de flujo del estado AlmacenaPaquete es el siguiente:

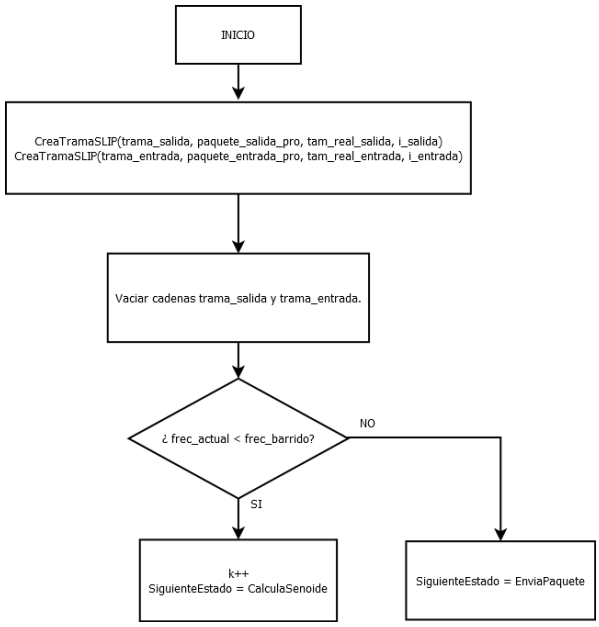

Ilustración 5-27: Diagrama de flujo del estado AlmacenaPaquete.

5.2.11 EnviaDatos

Cuando estos *arrays* contiene los paquetes que van a ser enviados, con el formato descrito en la Ilustración 5-10, están listos para ser enviados haciendo uso de la función *client.write()*. Esta recibe una cadena de datos, y el tamaño de los datos (en bytes) que debe enviar al cliente que se encuentra conectado.

En concreto, se harán cuatro escrituras: las dos cabeceras de los paquetes (“AAAA” y “FFFF”) y las

Devuelve el número de *bytes* que ha conseguido enviar de manera satisfactoria.

Si el número de *bytes* enviado no coincide con tamaño de los paquetes, se cierra la conexión y se vuelve al reposo advirtiendo del error. En caso contrario, si coinciden los datos enviados con el tamaño de los paquetes, el servidor lo notificará por consola, así como una estadística del tamaño de los paquetes enviados.

6 RESULTADOS OBTENIDOS

En el siguiente capítulo se va a mostrar el resultado de ejecutar la aplicación, así como contrastar estos resultados con los valores teóricos esperados.

Para ello, recordar que la aplicación es capaz de calcular el diagrama de Bode de cualquier sistema. En este caso, se va a usar como sistema un filtro RC paso bajo de frecuencia central 10 Hz.

El diagrama de bode de un filtro paso bajo ideal es el siguiente:

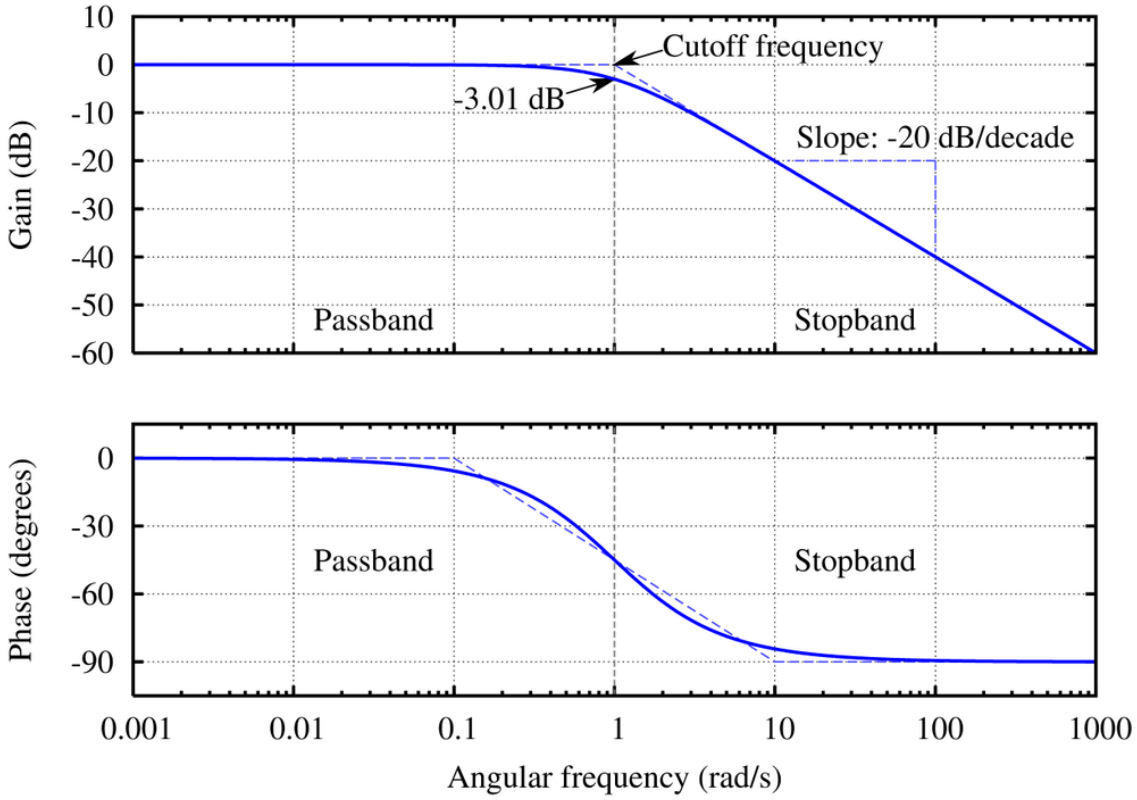

Ilustración 6-1: Diagrama de bode filtro RC paso bajo ideal.

En él se observa una caída de la magnitud a razón de 20 dB/dec a partir de la frecuencia central del filtro, así como una caída de la fase de 90º.

6.1 Ejecución de la Aplicación.

En primer lugar, conectamos el ESP32 mediante USB al ordenador para alimentarlo y tener conexión por puerto Serie.

Hay que destacar que el micro puede funcionar con una batería externa, por lo que podría funcionar con una *power bank*.

En el instante en el que el micro tiene alimentación, generará la red "*ESP32_AP"*, con la que el ordenador establecerá la conexión.

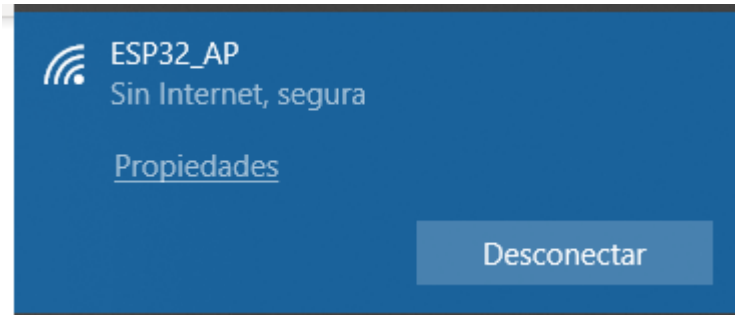

Ilustración 6-2: Red Wifi generada por el ESP32.

Una vez el ordenador está conectado a la red y abrimos el LabVIEW, esta es la interfaz gráfica que tiene el usuario:

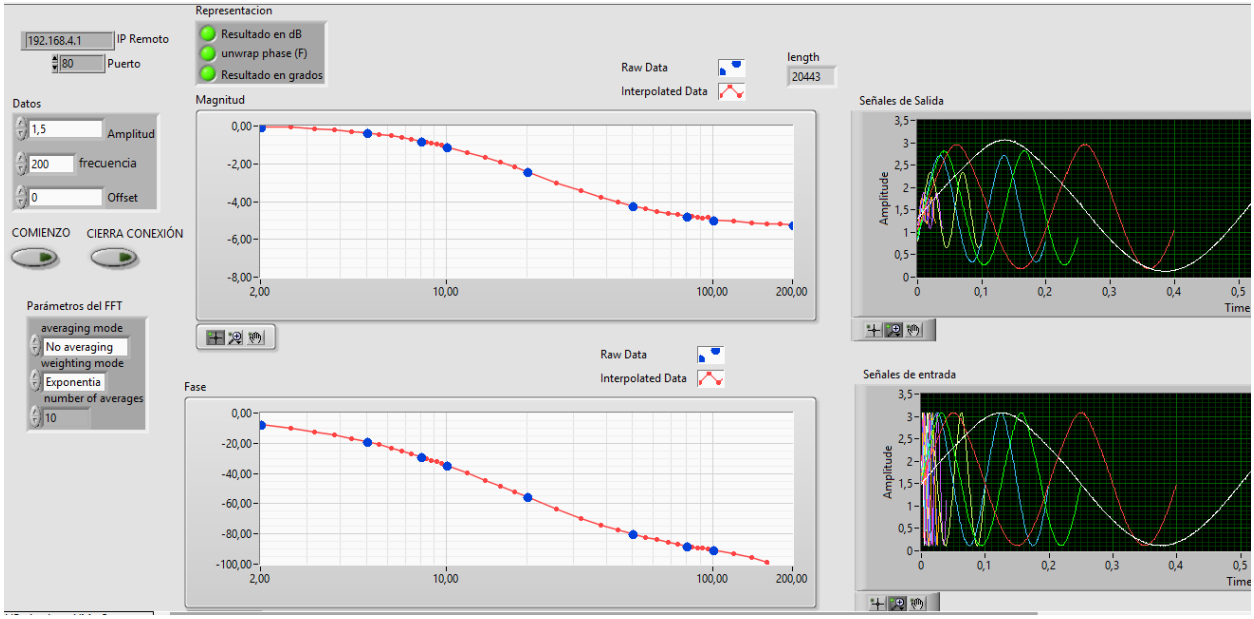

Ilustración 6-3: Interfaz principal de la aplicación.

En la ventana se pueden observar:

- Entrada de IP y Puerto remoto.
- Modo de representación del diagrama: Magnitud en dB o unidades naturales, fase en grados o radianes y el modo *unwrap phase*.
- Configuración del análisis a realizar.
- Botones de comienzo y fin de la aplicación.
- Displays para magnitud y fase.
- Displays de las señales de entrada y salida.

A la vez, leemos el puerto serie del micro con el programa *Putty*. La consola presenta que el servidor está esperando una conexión.

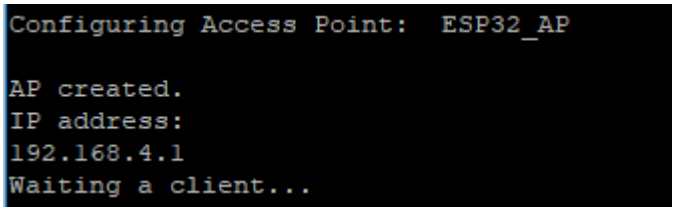

Ilustración 6-4: Salida de consola.

Desde la parte del cliente, ejecutamos el programa.

Introducimos los parámetros del análisis. Para el ejemplo, se va a configurar una frecuencia final del diagrama de 200Hz y una amplitud de señal de 1.5V.

Configuramos el modo de representación deseado, en este caso con la magnitud en dB y la fase en grados.

Pulsamos el botón “COMIENZO”, y empieza a funcionar la aplicación.

En ese instante el servidor muestra por puerto serie la siguiente información:

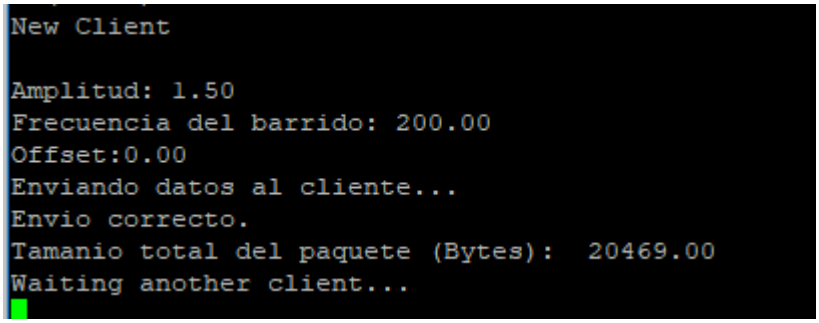

Ilustración 6-5: Mensaje por puerto serie.

En la consola muestra el estado de ejecución, así como el tamaño del paquete enviado al cliente. Después se queda a la espera de otra petición.

En ese instante recibimos en el entorno LabVIEW los datos correspondientes al análisis realizado.

El diagrama de Bode recibido es el siguiente:

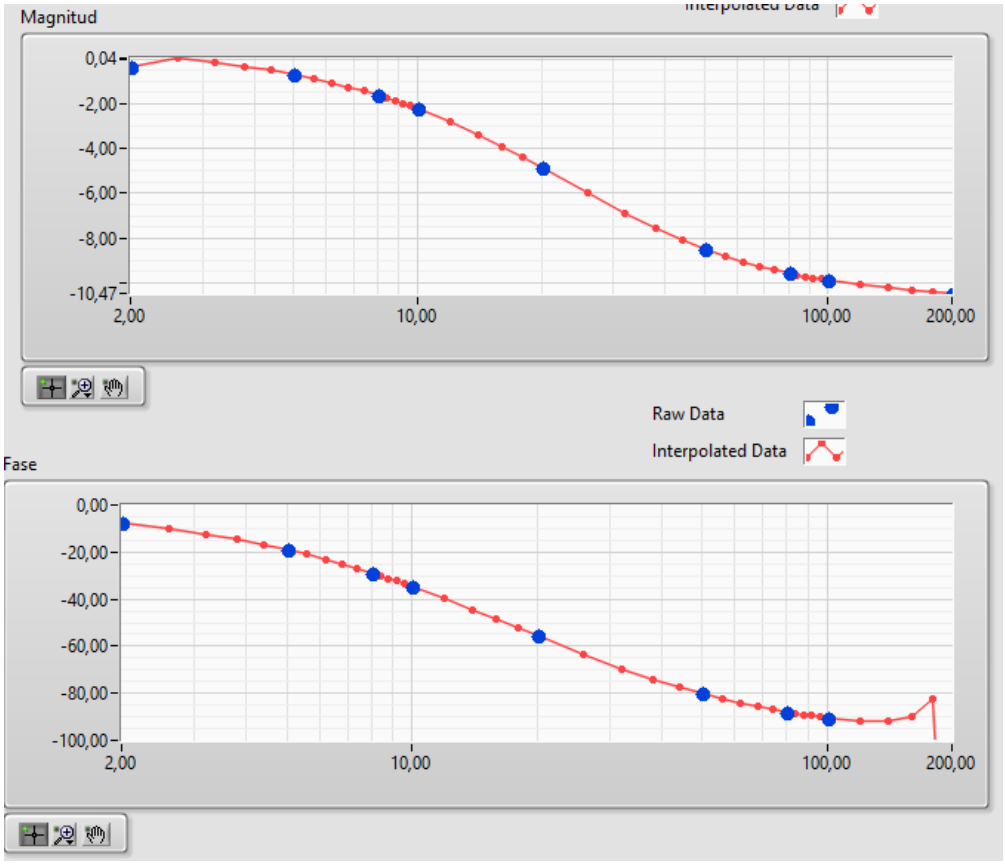

Ilustración 6-6: Diagrama de Bode devuelto.

En el diagrama se observa que la atenuación del filtro comienza a partir de los 10 Hz, cumpliendo lo esperado por la respuesta teórica, aunque se observa que la caída del filtro es menor de 20 dB/dec, que sería la respuesta ideal.

Con respecto a la fase, se observa cómo esta cae a los -90º correspondientes a un filtro de primer orden.

Además, en el display de la parte derecha que se muestran las señales generadas y leídas en una gráfica:

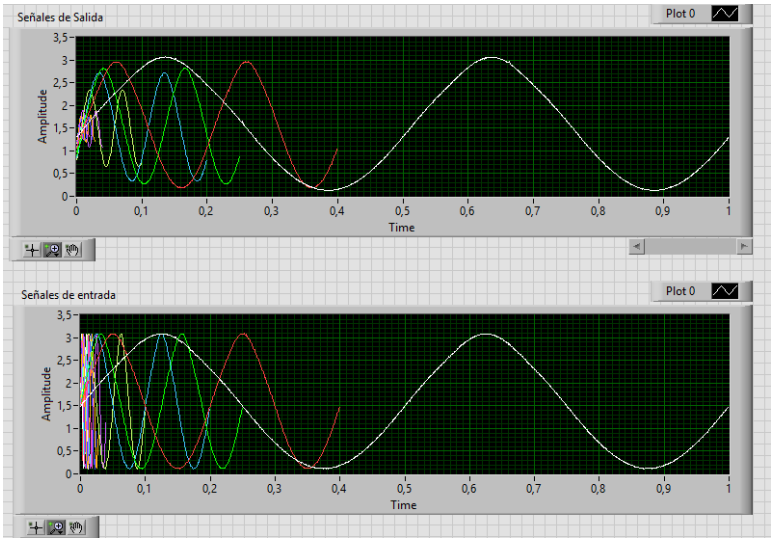

Ilustración 6-7: Señales de Entrada y salida del filtro.

Como se observa, las señales de entrada mantienen la misma amplitud, variando su frecuencia en función de la muestra que se esté tomando.

A cada una de las señales de entrada, le corresponde una señal de salida, las cuales, como se puede observar, tienen un nivel de amplitud menor conforme aumenta la frecuencia, debido al filtro.

Estas mismas señales, se han medido a través de un osciloscopio con mayor detalle.

6.2 Pruebas con osciloscopio.

En primer lugar, se va a hacer una captura de las señales generadas en cada análisis, tanto de entrada como de salida.

La captura generada es la siguiente:

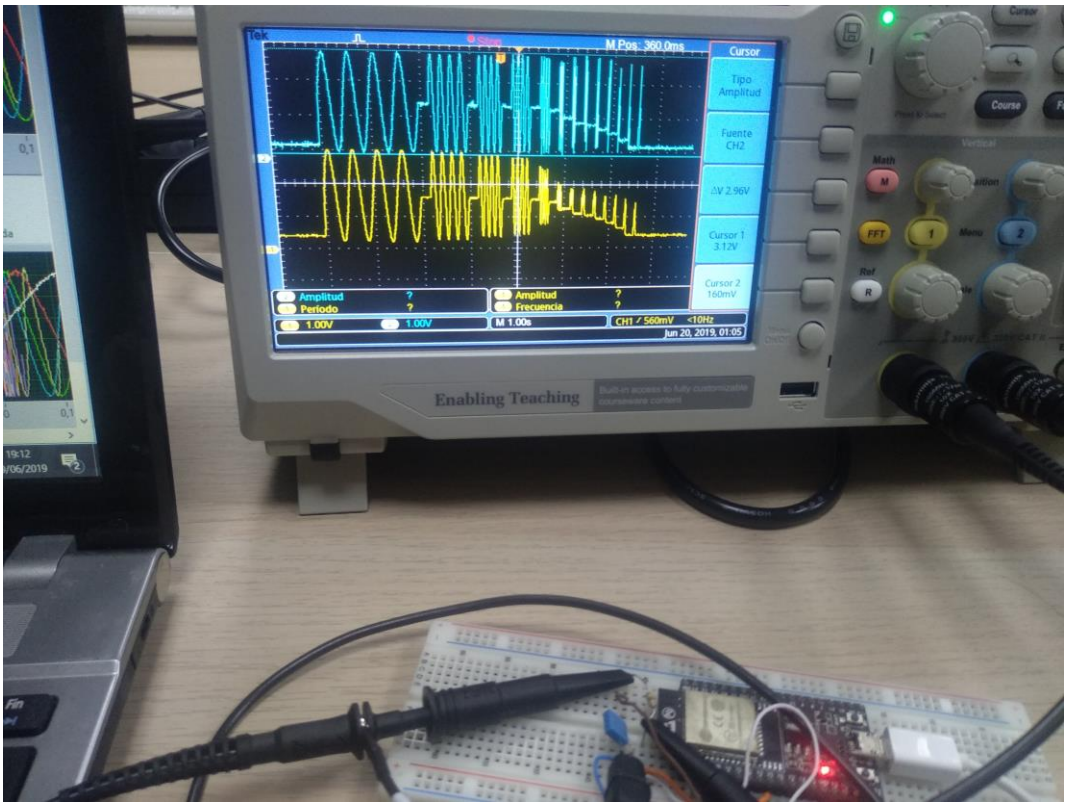

Ilustración 6-8: Captura con el osciloscopio de las señales de entrada y salida.

En la imagen se pueden observar por el canal 1 (azul), las señales que va generando el micro, cada vez con una frecuencia mayor. En total son 13 las señales generadas.

Al estar en la entrada del filtro, estas señales mantienen la amplitud de 1.5V, como se ha medido con los cursores.

Sin embargo, por el canal 2 (amarillo), las frecuencias leídas a la salida del filtro mantienen la frecuencia, pero se observa una clara atenuación de estas.

Si ampliamos en alguna de ellas, por ejemplo, en los 20 Hz, se obtiene:

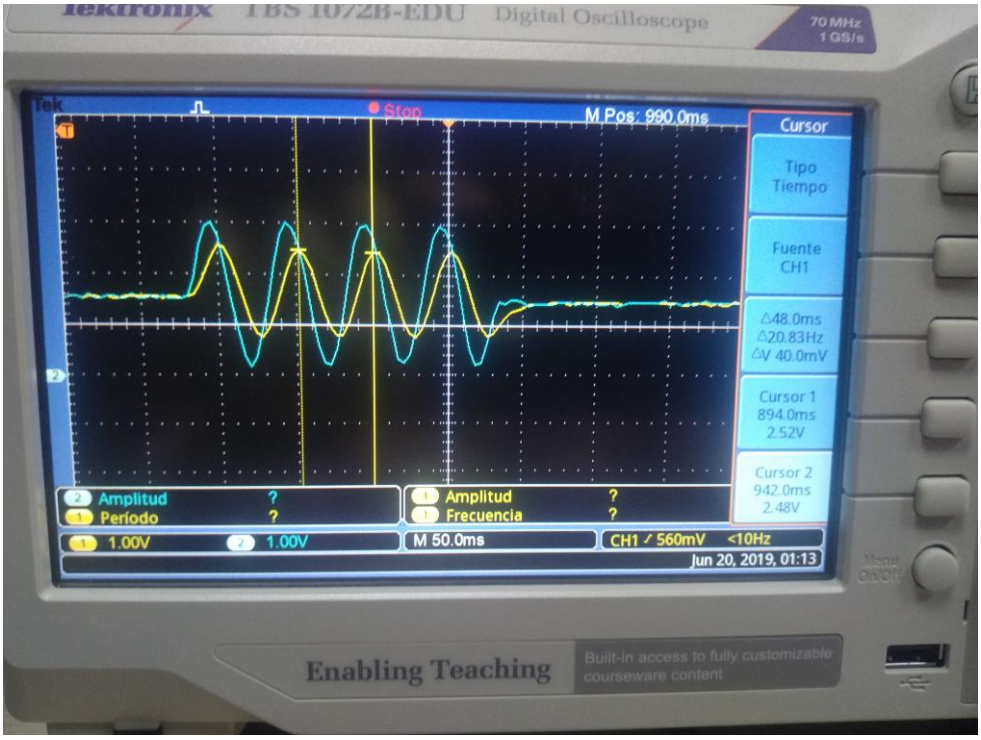

Ilustración 6-9: Detalle de la señal de 20 Hz.

Gracias a los cursores, comprobamos que la señal generada es correcta (20 Hz), por lo que el generador de senoides del micro es preciso en cuanto a frecuencia.

Si medimos la amplitud de la señal:

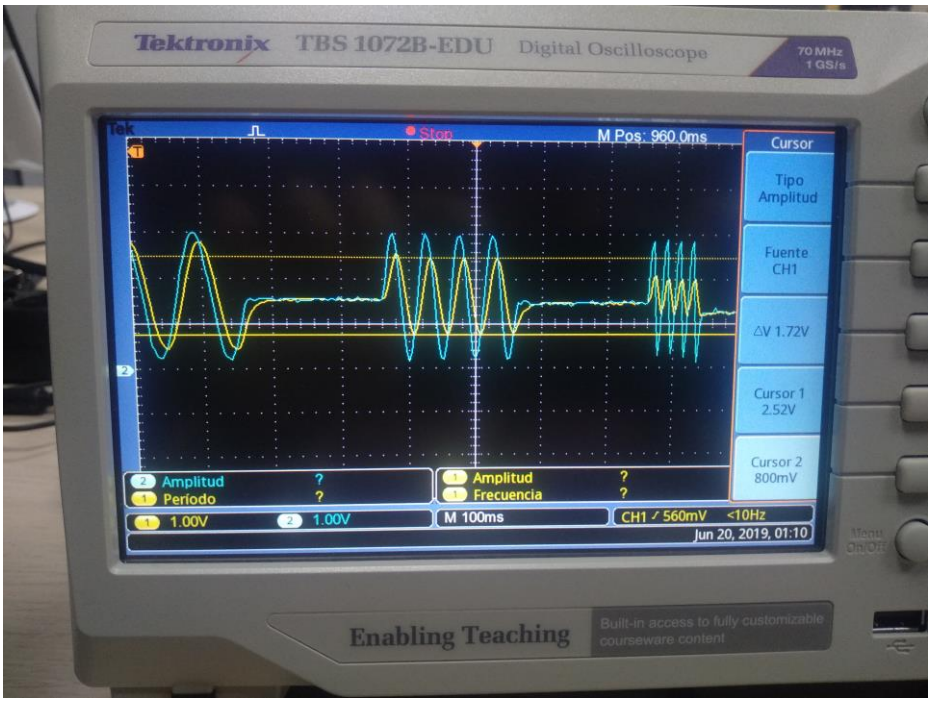

Ilustración 6-10: Detalle de la amplitud de la señal de 20 Hz.

Se observa una atenuación real de 4.83 dB.

Los puntos muestreados se representan en la siguiente tabla:

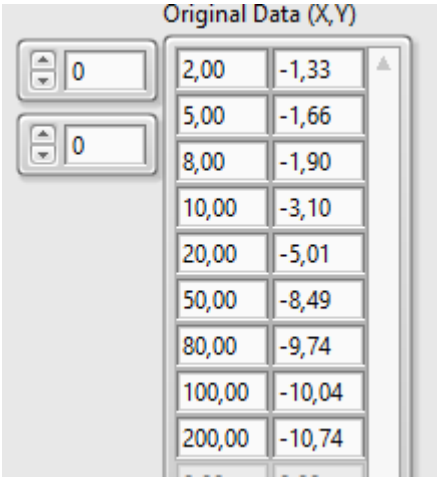

Ilustración 6-11: Puntos de la magnitud muestreados.

A los 20 Hz, la aplicación ha calculado una atenuación de 5.01 dB, por lo que el resultado es aceptable.

Si observamos las señales de alta frecuencia, se observa que las señales de salida son cada vez más atenuadas, pero debido al bajo número de muestras por señal, estas tienen muy mala calidad. Aun así, el filtro las rechaza, dando a la salida señales muy atenuadas.

Por ejemplo, la señal de 200Hz:

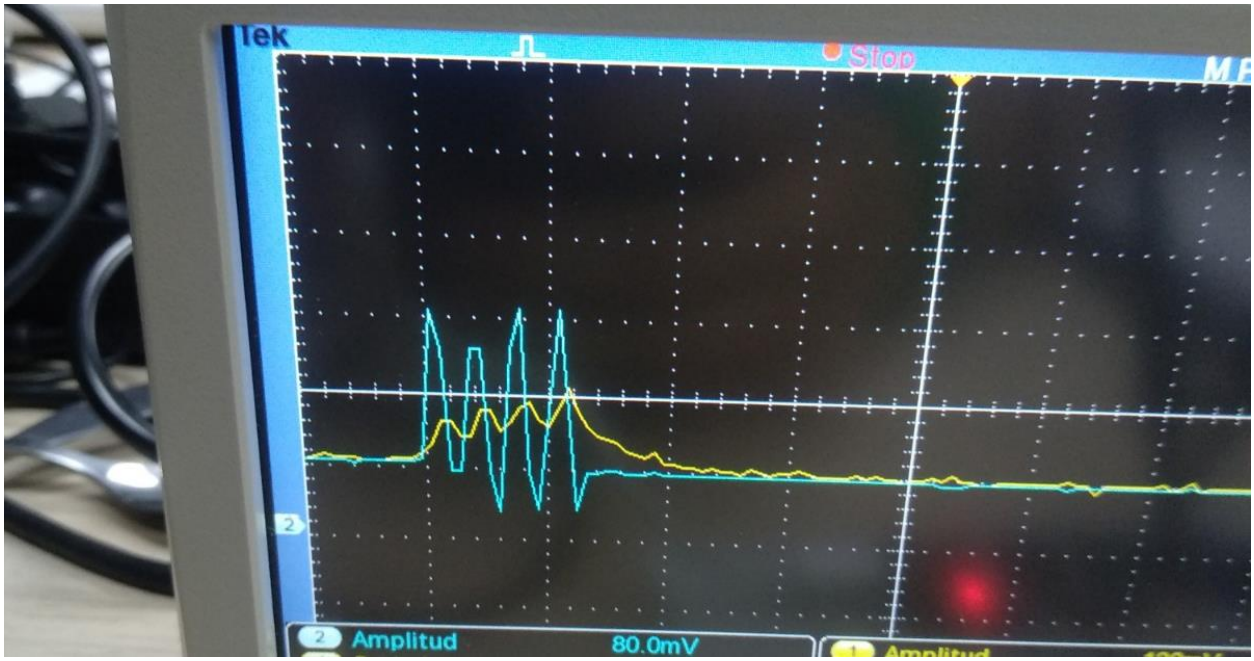

Ilustración 6-12: Detalle de la respuesta a 200Hz.

Se observa que, por la velocidad de escritura, al micro no le da tiempo a generar los valores deseados de las senoide, obteniéndose una señal con muchos armónicos. Esto produce una salida atenuada.

7 CONCLUSIONES Y LÍNEAS FUTURAS

En este trabajo hemos conseguido los siguientes objetivos:

- Implementar una aplicación cliente-servidor mediante comunicación wifi robusta.
- Implementar un generador de ondas con ESP32 hasta la frecuencia de los 200 Hz.
- Implementar un programa en LabVIEW capaz de calcular la magnitud y la fase de dos señales de entrada y salida respectivamente, así como interpolar los resultados para representar un diagrama de Bode del sistema.

Entre los contratiempos que se han encontrado durante el desarrollo de la aplicación se encuentran:

- Una baja frecuencia de muestreo por parte de ESP32, lo que ha provocado no poder obtener señales de una frecuencia más elevada.
- Problemas con el entorno Arduino, al ser un lenguaje de muy alto nivel, no permite sacar el rendimiento suficiente al microprocesador.
- Problemas para el envío secuencial de tramas entre el cliente y el servidor.

Como posible desarrollos futuros al proyecto:

- Programación del firmware usando librerías de más bajo nivel para el aprovechamiento de recursos.
- Aumento de la frecuencia de muestreo, así como generación de distintos tipos de onda.

Apéndice A
Códigos C++ para *Arduino*®

Código A.1 Código C++: Programa principal del servidor.

```
#include <WiFi.h>
#include <WiFiClient.h>
#include <WiFiAP.h>
#include <Arduino.h>

#define PIN_DAC 25
#define PIN_ENTRADA 32
#define PIN_SALIDA 34
#define TAM_BUFFER 28
#define TAM_DATA 14
#define NUM_PERIODOS 2
#define F_MAX 500
#define F_MIN 2
#define FM_PLACA 3000 //Frecuencia de muestreo de la placa.
#define TAM_TRAMA_VUELTA 34

// You can customize the SSID name and change the password
const char * ssid = "ESP32_AP";
const char * password = "123456789";

/* create a hardware timer */
hw_timer_t * timer = NULL;

/* Calculo de senoide */
const int Tm = 1000000 / FM_PLACA; //Un punto cada Tm micro segundos);
const int v_ampli = 248 / 2; //DAC 8 bits
int L; //Numero de muestras por periodo.


// Union para pasar los datos a float
union u_tag {
  byte b[4];
  float dato;
}u;

//TRAMAS SLIP RETURN
float senoide[NUM_PERIODOS * (FM_PLACA / F_MIN)];

float sen_salida[2 * (FM_PLACA / F_MIN)]; //2 periodos de la senoide
float sen_entrada[2 * (FM_PLACA / F_MIN)]; //2 periodos de la senoide

byte trama_salida[8 * (FM_PLACA / F_MIN)]; // 1 float = 4 bytes.
byte trama_entrada[8 * (FM_PLACA / F_MIN)];

byte paquete_salida_pro[4 * 5600]; //Aquí se guarda toda la informacion que se va a mandar.
byte paquete_entrada_pro[4 * 5600];

//Tramas de datos reibidos
byte datosLeidos[TAM_BUFFER];
byte trama[TAM_DATA];

//Variables usadas
int periodos = 0;
int i, j;
int flagenvio;
int k = 0;
```

```
int tam_real_entrada, tam_real_salida, tam_senoide;
int i_entrada = 0;
int i_salida = 0; //Indice global.


float frecuencias_barrido[] = {
  2,
  5,
  8,
  10,
  20,
  50,
  70,
  80,
  100,
  120,
  150,
  180,
  200
};

float amplitud;
float frec_barrido; //frecuencia final del barrido.
float offset;
float frec_actual; //frecuencia que se está calculando.
double size_total; //Tamanio total del paquete

int n = 0; //Contador del timer

short flag = 0;

// Puerto del servidor
WiFiServer server(80);
WiFiClient client;

//Estado mÃ¡quina de estados
enum State {
  Reposo,
  ConectarWifi,
  ConexionCliente,
  LeerDatos,
  ActualizaDatos,
  GeneraSenoide,
  AlmacenaPaquete,
  EnviaDatos,
  EscribeSenial,
  Procesa
};

//Variables de cambio de estado
State estadoActual;
State siguienteEstado;

int x = 0;

// Funcion del TIMER
void IRAM_ATTR onTimer() {
  flag = 1;
}

void setup() {

  /* 1 tick take 1/(80MHZ/80) = 1us so we set divider 80 and count up */
  timer = timerBegin(0, 80, true);

  /* Attach onTimer function to our timer */
  timerAttachInterrupt(timer, & onTimer, true);

  /* Set alarm to call onTimer function every second 1 tick is 1us
  => 1 second is 1000000us */
  /* Repeat the alarm (third parameter) */
  timerAlarmWrite(timer, Tm, true);

  siguienteEstado = Reposo;
}
```

```
void loop() {
  delay(50);
  switch (estadoActual) {
  case Reposo: //Reposo
    {
      siguienteEstado = ConectarWifi;
      cambiaEstado(siguienteEstado);
    }
    break;

  case ConectarWifi: //Conectar el wifi
    {
      Serial.begin(115200);
      delay(10);

      // We start by connecting to a WiFi network

      Serial.println();
      Serial.println();
      Serial.print("Configuring Access Point:  ");
      Serial.println(ssid);

      WiFi.softAP(ssid, password);

      Serial.println("");
      Serial.println("AP created.");
      Serial.println("IP address: ");
      Serial.println(WiFi.softAPIP());
      server.begin();

      Serial.println("Waiting a client...");
      siguienteEstado = ConexionCliente;
      cambiaEstado(siguienteEstado);
    }
    break;

  case ConexionCliente:
    {
      size_total = 0;
      delay(50);
      client = server.available(); // listen for incoming clients
      if (client) {
        flagenvio = 0;
        Serial.println("New Client"); // Se ha conectado un cliente.
        siguienteEstado = LeerDatos;
      }
      cambiaEstado(siguienteEstado);
    }
    break;
  case LeerDatos: //Leer Datos
    {
      if (client && client.connected()) {
        if (client.available()) {
          //Recibimos datos del Labview.
          for (int i = 0, cont = 0; i < TAM_BUFFER && cont != 2; i++) {
            datosLeidos[i] = client.read();
            delay(10);
            if (datosLeidos[i] == 0xC0) //Salimos cuando aparezcan dos 0xC0
            {
              cont++;
            }
          }
          extraeTramaSLIP(trama, datosLeidos, TAM_DATA, TAM_BUFFER);
          siguienteEstado = ActualizaDatos;
          cambiaEstado(siguienteEstado);
        }
      } else {
        client.stop();
        Serial.println("");
        Serial.println("Fallo en el envio desde el cliente");
        delay(5000);
        Serial.println("Waiting another client: ");
        siguienteEstado = ConexionCliente;
```

```
      }
      cambiaEstado(siguienteEstado);
    }

    break;

  case ActualizaDatos:
    {

      u.b[0] = trama[4];
      u.b[1] = trama[3];
      u.b[2] = trama[2];
      u.b[3] = trama[1];

      amplitud = u.dato;

      u.b[0] = trama[8];
      u.b[1] = trama[7];
      u.b[2] = trama[6];
      u.b[3] = trama[5];

      frec_barrido = u.dato;

      u.b[0] = trama[12];
      u.b[1] = trama[11];
      u.b[2] = trama[10];
      u.b[3] = trama[9];

      offset = u.dato;

      Serial.println("");
      Serial.print("Amplitud: ");
      Serial.print(amplitud);
      Serial.println("");
      Serial.print("Frecuencia del barrido: ");
      Serial.print(frec_barrido);
      Serial.println("");
      Serial.print("Offset:");
      Serial.println(offset);

      siguienteEstado = GeneraSenoide;
      cambiaEstado(GeneraSenoide);

    }
    break;

  case GeneraSenoide: //Generar la senoide
    {
      if (frecuencias_barrido[k] < frec_barrido)
        frec_actual = frecuencias_barrido[k];
      else
        frec_actual = frec_barrido;
      L = (FM_PLACA / frec_actual); // Numero de muestras de la senoide

      for (n = 0; n < L; n++) // Guardo un periodo de la senoide en una variable senoide.
        senoide[n] = (((125 + (amplitud * 125 / 1.65) * sin(2 * PI * n / L)) * 3.3) / 250); // Genera
Onda Senoidal. Entre 0 y 3.3 V

      siguienteEstado = EscribeSenial;
      cambiaEstado(siguienteEstado);
    }
    break;

  case EscribeSenial:
    {
      // Serial.println("EscribeSeñal");
      int i = 0;
      int j = 0;
      periodos = 0;
      timerAlarmEnable(timer);

      while (periodos < NUM_PERIODOS - 1) {

        Serial.print("");
        if (flag) //flag del timer
        {
          dacWrite(PIN_DAC, senoide[i++] * 250 / 3.3); //Escribe un valor entero [0-250]
```

```
        if (periodos == 2 || periodos == 3) //Solo guardo los periodos 2 y 3 de la seÃ±al.
        {
          sen_entrada[j] = ((analogRead(PIN_ENTRADA) * 3.3) / 4095);
          sen_salida[j++] = ((analogRead(PIN_SALIDA) * 3.3) / 4095);
        }
        flag = 0;
        if (i > L - 1) {
          i = 0;
          periodos++;
        }
      }

    }

    timerAlarmDisable(timer);
    tam_senoide = j;
    siguienteEstado = Procesa;
    cambiaEstado(siguienteEstado);
  }
  break;

case Procesa: // Generamos las tramas SLIP
  {

    for (int i = 0; i < tam_senoide; i++) //Recorrer todos los floats de sen_entrada y sen_salida
    {
      u.dato = sen_salida[i];
      trama_salida[(i * 4)] = u.b[3];
      trama_salida[(i * 4 + 1)] = u.b[2];
      trama salida[(i * 4 + 2)] = u.b[1];
      trama_salida[(i * 4 + 3)] = u.b[0];
      u.dato = sen_entrada[i];
      trama_entrada[(i * 4)] = u.b[3];
      trama_entrada[(i * 4 + 1)] = u.b[2];
      trama_entrada[(i * 4 + 2)] = u.b[1];
      trama_entrada[(i * 4 + 3)] = u.b[0];
    }

    tam_real_salida = 4 * tam_senoide;
    tam_real_entrada = 4 * tam_senoide;
    siguienteEstado = AlmacenaPaquete;
    cambiaEstado(siguienteEstado);
  }
  break;

case AlmacenaPaquete:
  {

    i_salida = creaTramaSLIP(trama_salida, paquete_salida_pro, tam_real_salida, i_salida);
    i_entrada = creaTramaSLIP(trama_entrada, paquete_entrada_pro, tam_real_entrada, i_entrada);

    for (int i = 0; i < (8 * (FM_PLACA / F_MIN)); i++) //Limpiamos las tramas
    {
      trama_salida[i] = 0x00;
      trama_entrada[i] = 0x00;
    }

    if (frec_actual < frec_barrido) {
      k++;
      frec_actual = frecuencias_barrido[k];
      flagenvio = 1;
      siguienteEstado = GeneraSenoide;
    } else {
      siguienteEstado = EnviaDatos;
    }
    cambiaEstado(siguienteEstado);
  }
  break;

case EnviaDatos:
  {
    Serial.println("Enviando datos al cliente...");

    int x_entrada = 0;
    int x_salida = 0;
    delay(50);
    if (client.connected()) {
      byte cabecera_entrada[2] = {
```

```
          0xAA,
          0xAA
        };
        byte cabecera_salida[2] = {
          0xFF,
          0xFF
        };
        x_entrada = client.write(cabecera_entrada, 2);
        x_entrada += client.write(paquete_entrada_pro, i_entrada);
        x_salida = client.write(cabecera_salida, 2);
        x_salida += client.write(paquete_salida_pro, i_salida);

        size_total = size_total + x_entrada + x_salida;
        Serial.println("Envio correcto.");
        delay(100);

        siguienteEstado = ConexionCliente;
        k = 0;
        i_entrada = 0;
        i_salida = 0;
        Serial.print("Tamanio total del paquete (Bytes):  ");
        Serial.println(size_total);
        delay(2000);
        Serial.println("Waiting another client...");
        flagenvio = 1;
        client.stop();
      } else {
        client.stop();
        Serial.println("Waiting another client...");
        siguienteEstado = ConexionCliente;
      }

      delay(10);

      cambiaEstado(siguienteEstado);
    }
    break;
  }
}
```

Código A.2: Función cambiaEstado

```
void cambiaEstado(State siguienteEstado) {
  estadoActual = siguienteEstado;
}
```

Código A.3: Función extraeTramaSLIP

```
void extraeTramaSLIP(byte trama[], byte datosLeidos[], int datalen, int bufflen) {
  for (int i = 0, j = 0; i < bufflen; i++, j++) {
    if (datosLeidos[i] == 0xDB && datosLeidos[i + 1] == 0xDD) //Almacenamos un 0xDB
    {
      trama[j] = datosLeidos[i];
      i++;
    } else if (datosLeidos[i] == 0xDB && datosLeidos[i + 1] == 0xDC) //Almacenamos un 0xC0
    {
      trama[j] = 0xC0;
      i++;
    } else //Almacenamos el dato
    {
      trama[j] = datosLeidos[i];
    }
  }
}
```

Código A.4: Función creaTramaSLIP

```
/* Genera una trama SLIP a partir de una trama en HEX*/
/* param:  tramaLocal: Cadena HEX sin procesar
           tramaEnviada: Cadena HEX procesada
           locallen: Tamanio en bytes de la cadena Local
           index: Tamanio en bytes de la cadena a enviar
           */

int creaTramaSLIP(byte tramaLocal[], byte tramaEnviada[], int locallen, int index) {

  tramaEnviada[0 + index] = 0xC0;
  index++;
  for (int i = 0; i < locallen; i++, index++) {
    if (tramaLocal[i] == 0xC0) {
      tramaEnviada[index] = 0xDB;
      tramaEnviada[++index] = 0xDC;
    } else if (tramaLocal[i] == 0xDB) {
      tramaEnviada[index] = 0xDB;
      tramaEnviada[++index] = 0xDD;

    } else
      tramaEnviada[index] = tramaLocal[i];
  }
  tramaEnviada[index] = 0xC0;
  index++;

  return index;

}
```

Apéndice B

Planos del dispositivo

Plano A.1: Esquemático

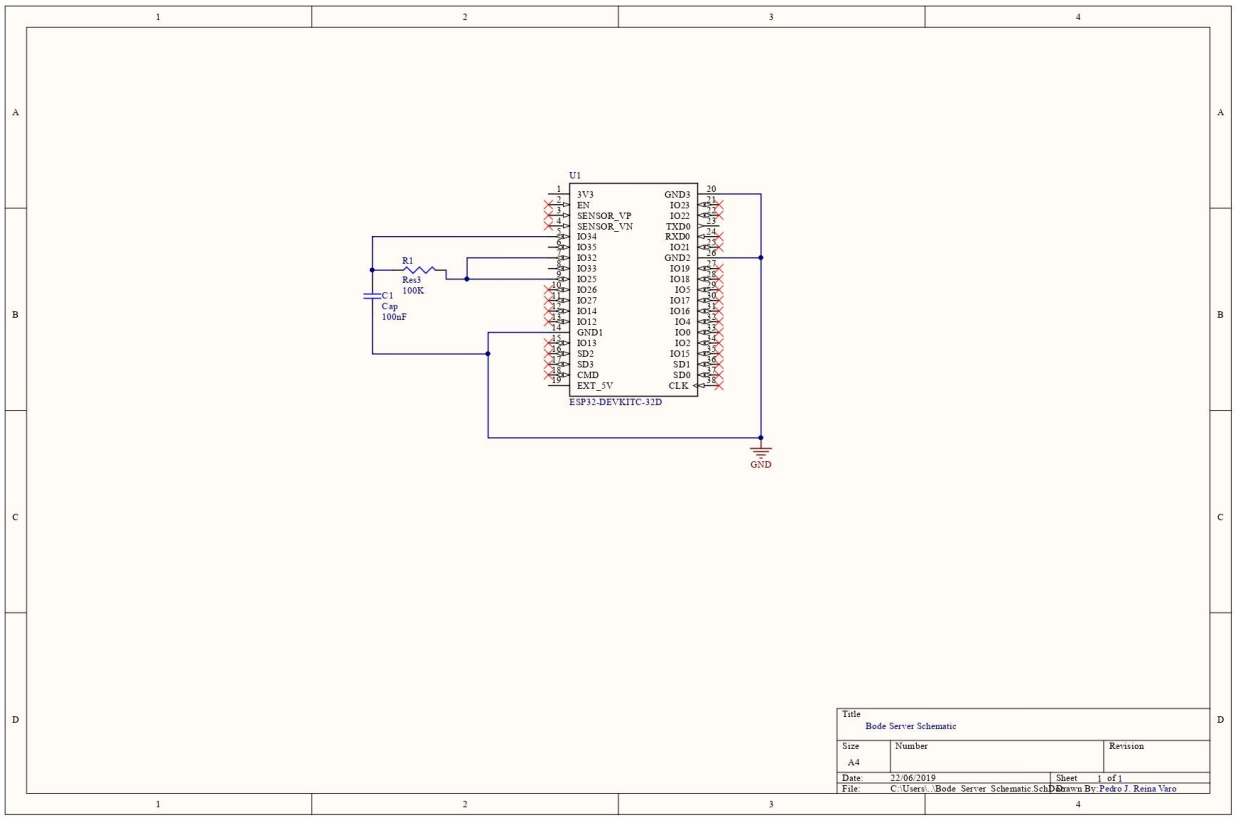

Ilustración B-0-1: Diseño esquemático Altium Designer.

Plano A.2: PCB Layout

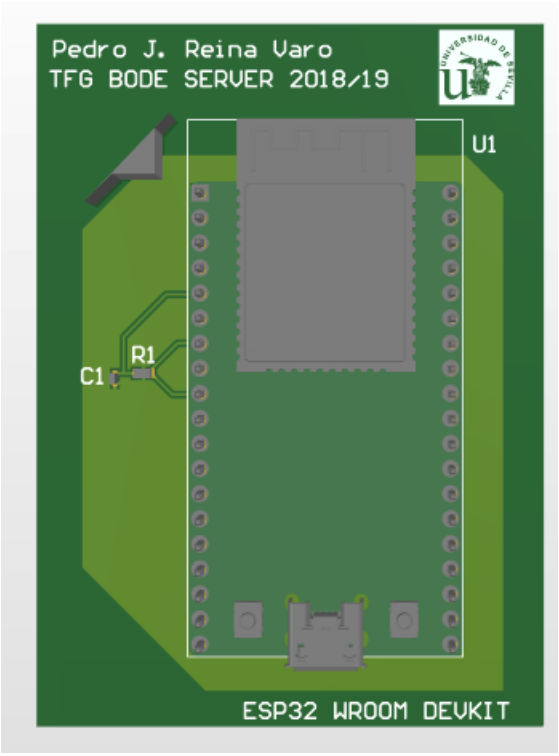

Ilustración B-2: Vista 3D del PCB con Altium Designer

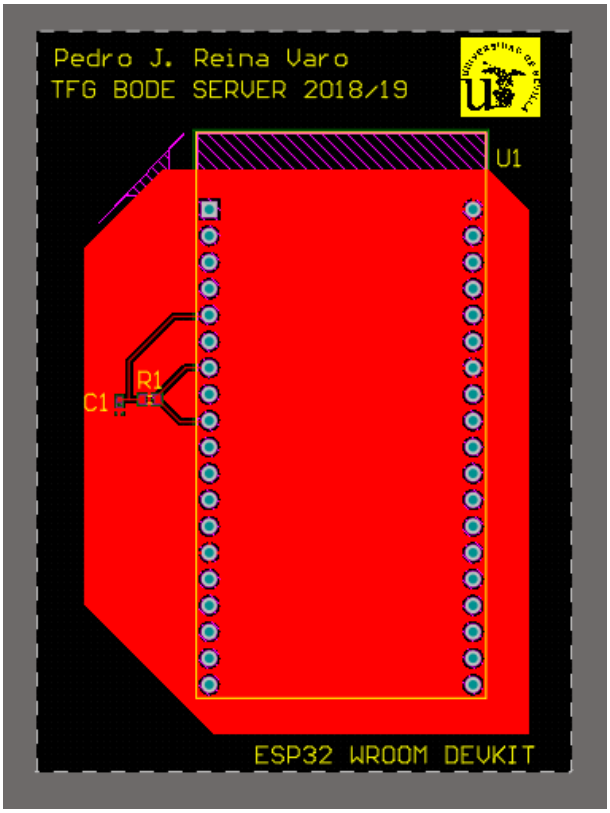

Ilustración B-3: Vista del PCB con Altium Designer.

Referencias

[1]Comparativa y análisis completo de los módulos wifi ESP8266 y ESP32:

https://blog.bricogeek.com/noticias/electronica/comparativa-y-analisis-completo-de-los-modulos-wifi-esp8266-y-esp32

[2] Serial Line Internet Protocol: https://es.wikipedia.org/wiki/Serial_Line_Internet_Protocol

[3] Foro de la comunidad Labview: https://forums.ni.com/t5/LabVIEW/bd-p/170

[4] Directorio de librerías de Arduino: https://www.arduino.cc/en/Reference/Libraries

[5] Labview Spectral Tutorial: ftp://ftp.ni.com/pub/devzone/LabVIEW_Spectral_Tutorial.pdf

[6] Altium designer: https://www.altium.com/es