Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Diseño e implementación de la adquisición y el procesado en un nodo de un sistema distribuido de localización por audio.

Autor: Álvaro Macián Martínez

Tutores: Daniel Pizarro Pérez y José Francisco Velasco Cerpa

2017

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Diseño e implementación de la adquisición y el procesado en un nodo de un sistema distribuido de localización por audio.

Autor: Álvaro Macián Martínez

Tutores: Daniel Pizarro Pérez y José Francisco Velasco Cerpa

Tribunal:

Presidente: Francisco Javier Meca Meca

Vocal 1º: Jesús Ureña Ureña

Vocal 2º: Daniel Pizarro Pérez

Calificación: .....

Fecha: .....

Agradecimientos

En primer lugar, a mis tutores José Velasco y Daniel Pizarro por darme la oportunidad de realizar este proyecto y sin los que hubiera sido imposible acabar esta bonita etapa. También a todas esas personas que me han aportado algo. A mis amigos y a todos los compañeros de clase que al final terminaron convirtiéndose en amigos.

Pero sobre todo, a mi familia, la que nunca me ha fallado y, tanto en los buenos como en los malos momentos, me han dado ese empujón que me hacía falta. A mi padre y a mi hermano, pero sobre todo a mi madre, mi ejemplo.

Resumen

El objetivo de este proyecto es el de llegar a la detección de la dirección del emisor de sonido, lo que se conoce como localización acústica. En este proyecto se implementa un sistema que permite llegar a tal objetivo realizando para ello una adquisición de sonido y el posterior procesado de la señal adquirida en tiempo real. Se va a comenzar hablando de la parte hardware para, posteriormente, pasar a hablar de todo el procesado al que las señales que recogen los micrófonos son sometidas para llegar a la localización. Tras esto, se hablará de la implementación que se ha llevado a cabo y cómo se ha realizado la misma, además de los resultados que se han obtenido y las conclusiones extraídas. La implementación se ha realizado haciendo uso de la herramienta *Keil* y, para contrastar los resultados obtenidos se han empleado, tanto *Matlab*, como *Audacity*.
Palabras clave: **audio, localización, TDOA, GCC, STM32F746G-DISCOVERY**.

Abstract

The objective of this project is to detect the direction of the sound emitter. This problem is known as acoustic localization. To achieve this goal, this project propose a system that acquires and process signals in real time, obtained from a pair of microphones. This document first describes the hardware architecture followed by a description of the signal processing algorithm that computes the target direction with respect to the pair of microphones. It then describes the implementation of the algorithm and shows experimental results and conclusions. This project is based on *Keil*, *Matlab* and *Audacity* software tools.
Keywords: **audio, location, TDOA, GCC, STM32F746G-DISCOVERY**.

Resumen extendido

En este proyecto se realiza el diseño de un sistema que permite al usuario adquirir señales de audio procedentes de los dos micrófonos que integra la placa **STM32F746G-DISCOVERY** para posteriormente ser procesadas y obtener la localización del emisor del sonido.

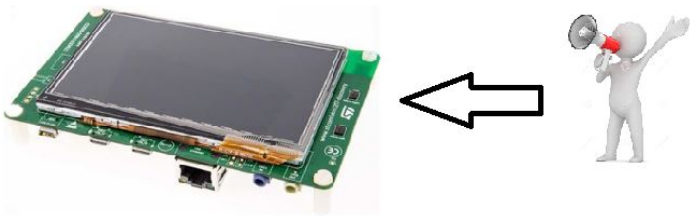


Figura 2: Adquisición de audio.

Para obtener la localización hay que realizar varias operaciones antes, y éstas pueden dividirse en dos etapas:

- La primera de ellas consiste en la **adquisición** de audio y el posterior **acondicionamiento** de la señal acústica que se ha recibido.
  El acondicionamiento que se realiza de la señal consiste en la aplicación de un filtro FIR que atenúa las altas frecuencias de la señal acústica. La aplicación de dicho filtro mejora los resultados que se obtienen posteriormente en localización debido a que la información relativa a la localización se encuentra en las frecuencias bajas. Por tanto, eliminando las altas frecuencias se elimina información irrelevante para el objetivo.

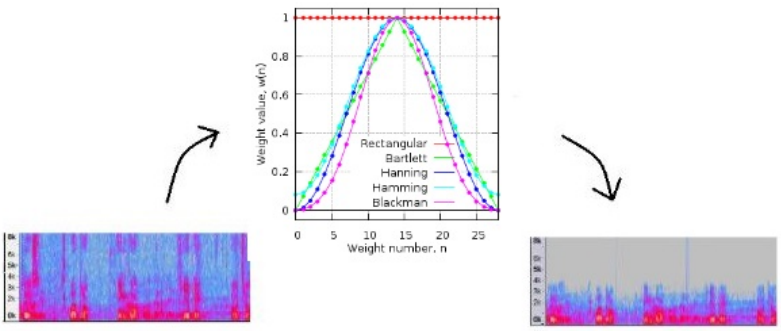


Figura 3: Acondicionamiento de la señal acústica.

- La segunda etapa se centra en la **localización** de la fuente sonora. Aquí se realiza un pequeño estudio de las técnicas existentes como SRP (Steered-Response Power), técnicas basadas en alta resolución espectral y GCC (Generalized Cross-Correlation), siendo esta última la técnica elegida. La obtención de la localización se realiza a través de TDOA (Time Delay of Arrival), basándonos en GCC-PHAT. La aplicación del filtro PHAT elimina la influencia de la amplitud de la señal recibida. Además, se ha demostrado que PHAT en entornos reverberantes produce mejores resultados que otros filtros.

Por último, se ha optado por realizar la grabación de las señales acústicas que reciben ambos micrófonos ya que resulta muy interesante y, a su vez, de gran utilidad para posteriores estudios. Con la grabación de las señales se pueden contrastar los resultados obtenidos mediante programas como *Matlab*.

El funcionamiento del programa, que se explicará en los siguientes capítulos del documento, se aprecia en el diagrama de bloques de la figura 4.

Figura 4: Diagrama de bloques del funcionamiento del programa.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Los arrays de micrófonos y los métodos de procesado de señal asociados a ellos, permiten utilizar un grupo de micrófonos de manera conjunta. Es decir, las señales captadas por los diferentes elementos del array son combinadas siguiendo diferentes objetivos como la mejora de la calidad de la señal obtenida o la localización de la fuente, tal y como se ha tratado en este proyecto. Este tipo de técnicas tienen mucho en común con aquellas utilizadas para otro tipo de sensores (por ejemplo antenas, ultrasonidos, etc.) siendo todas ellas objeto de profundo estudio durante las últimas décadas.[1, 2].

Habitualmente los arrays de micrófonos han usado disposiciones geométricas conocidas e incluso diseñadas específicamente para satisfacer algún objetivo. Además, debido al sincronismo necesario en este tipo de sistemas, los sistemas de adquisición utilizados solían ser cableados y centralizados. Sin embargo, la reciente aparición de técnicas calibración [3–6], que permiten descubrir la geometría del array ad-hoc, así como el abaratamiento y la mejora de los dispositivos móviles, que incorporan amplias capacidades de sensado y comunicaciones, han hecho que cada vez exista un mayor interés sobre los sistemas de adquisición distribuidos. Estos últimos ofrecen ciertas características que los hacen especialmente interesantes:

- Coste nulo de instalación
- Interoperabilidad de sistemas
- Gran versatilidad

Este proyecto se enmarca dentro del grupo de investigación *GEINTRA* que, actualmente, tiene una línea de investigación basada en la interacción hombre-máquina a través de señales de audio. El proyecto en sí forma parte de un sistema distribuido capaz de adquirir y procesar audio de manera sincronizada con el objetivo de localizar hablantes en entornos reverberantes. En este proyecto en concreto, se ha diseñado e implementado un prototipo de nodo.

Para la realización de este proyecto, se ha elegido la placa *STM32F7G-DISCOVERY* (1.1) que, a pesar de su reducido coste (49 dólares), dispone de un microcontrolador *ARM CORTEX-M7* que tiene una unidad de coma flotante y un conjunto de instrucciones de procesado de señal que facilitan mucho el trabajo en la manipulación de las señales. Además, la placa incorpora dos micrófonos digitales tipo *MEMS*, los cuales son los encargados de captar las dos señales sobre las que se va a trabajar.

Para llegar a la localización de la fuente emisora, es necesario realizar una serie de operaciones con las señales adquiridas mediante los dos micrófonos. En primer lugar se aplica un filtro *FIR* a ambas señales,

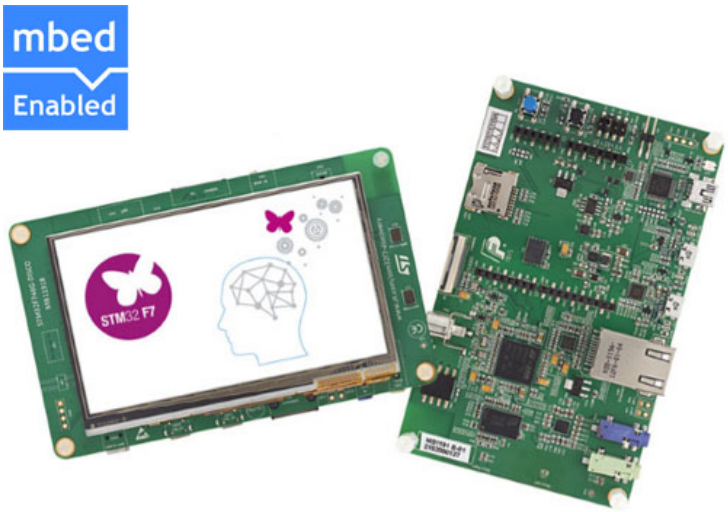


Figura 1.1: Tarjeta de desarrollo STM32F746NG-DISCOVERY

que nos va a proporcionar unos resultados más exactos ya que se ha estudiado que las componentes de alta frecuencia añaden aleatoriedad y, la información de localización se encuentra en las bajas frecuencias.[7]. Tras la aplicación del filtro, se realiza una estimación de la diferencia de tiempos de llegada de la señal a cada uno de los dos micrófonos de la tarjeta. Esto es denominado *TDOA* (*Time Delay of Arrival*) y, el método más comúnmente utilizado para realizar dicha estimación es *GCC* (*Correlación cruzada generalizada*)[7]. También es importante destacar, que dentro de dicho método existen distintos tipos de filtrados, que se comentarán más adelante, siendo *GCC-PHAT* el empleado ya que es el método más efectivo en condiciones reales, es decir, en entornos reverberantes con ruido ambiente.[8]. Por último, tras resolver unas ecuaciones trigonómetricas se llega al objetivo del proyecto, determinar la dirección del emisor del sonido. En los capítulos 3 y 4, se detalla la forma en la que se reconoce la dirección del sonido, hablando tanto de *TDOA*, como de toda la manipulación que se hace de las señales para obtener *GCC*.

Esta memoria se organiza en diez capítulos. El capítulo 1 consiste en una introducción al proyecto, donde se comenta que es lo que se ha realizado para llegar al objetivo final. El capítulo 2 describe la tarjeta empleada, abordando la parte hardware del proyecto. Después, en los capítulos 3 y 4 se realiza el estudio teórico del proyecto, en el que se va a comentar básicamente de qué forma se realiza la adquisición de audio y el posterior procesado que se le realiza a la señal para llegar al objetivo final de localización. Posteriormente, se pasa al desarrollo de la implementación del proyecto la cual se realiza en el capítulo 5, para después hablar de los resultados obtenidos y las consecuentes conclusiones en los capítulos 6 y 7. Para terminar, se realiza el pliego de condiciones en el capítulo 8, un análisis del presupuesto del trabajo (9) y un manual de usuario en el que se detalla el funcionamiento del programa (10).

Capítulo 2

Plataforma STM32F746G-DISCOVERY

2.1 Introducción

En este capítulo se hace un análisis en profundidad tanto de la tarjeta de desarrollo como de toda la parte teórica del proyecto, que luego se va a contrastar con los resultados obtenidos.
La tarjeta *STM32F746G-DISCO* está diseñada por *STMicroelectronics ARM*. Hay tres familias de procesadores basados en la arquitectura *ARMv7*, las cuales se diferencian en las aplicaciones a las que dan uso. Esta tarjeta está encuadrada en el grupo *Cortex-M*, el cual es desarrollado en aplicaciones para microcontroladores. Además dentro de *Cortex-M*, hay distintos tipos de procesadores, siendo este, el *M7*, el más nuevo y de más alto rendimiento. Las principales características de este procesador son:

- Gran abanico de instrucciones DSP debido al módulo *CMSIS-DSP*
- Mínimo consumo de energía y gran eficiencia energética
- Gran seguridad con características como recuperación de errores a través de la *ECC* (Memoria de recuperación de errores)
- *MPU* (Memory protection unit) opcional de 8 a 16 regiones que se encarga de separar las tareas en ejecución y también de la protección de los datos
- El procesador *Cortex-M7* soporta todas las instrucciones de sus antecesores *M3* y *M4*

La tarjeta está basada en el microcontrolador *STM32F746NGH6* y está provisto de 4 módulos de *I2C*, 6 *SPIs*, entrada para tarjeta *microSD*, pantalla LCD-TFT a color de 4.3 pulgadas, cuatro *USARTs*, otras cuatro *UARTs*, tres *ADCs* y dos *DACs* de 12 bits cada uno de ellos, dos *SAIs*, el módulo de la cámara digital, MAC Ethernet, 64Mbit de memoria *SDRAM* y 128Mbit de *QuadSPI* Flash. Además, cuenta con periféricos como *USB OTG(HS/FS)*, puerto ethernet de 10/100Mbit Ethernet, entrada y salida para audio *Jack*, un conector de entrada de audio *SPDI RCA*, dos micrófonos digitales *MEMS*, un conector para tarjetas de memoria *microSD* y conectores para Arduino.

Cabe destacar la importancia del *DSP* de la tarjeta para este proyecto. Debido a que el *DSP* posee un conjunto de instrucciones que realizan operaciones numéricas a gran velocidad, es muy útil para

aplicaciones en tiempo real como es este caso, en el que se procesan señales en entornos reales. Por este motivo, se ha elegido la tarjeta *STM32F746G-DISCO*. Debido a la alta velocidad de las operaciones del *DSP* se suele emplear en audio, tema en el que se encuadra este proyecto, así como reconocimiento de voz, vídeo...

Los *DSPs* poseen una arquitectura que les permite ejecutar a gran velocidad operaciones como las que se van a ver a continuación. Para poder realizar esto, suelen tener una arquitectura de memoria que permiten un acceso múltiple y, así poder realizar la carga de varios operandos simultáneamente. Las operaciones básicas que realizan los *DSPs* son la convolución, correlación, aplicación de filtros digitales, modulación de señales y transformadas discretas. Además de estas operaciones realizan más y se pueden clasificar de la siguiente forma:

Funciones matemáticas básicas : funciones matemáticas que realizan las operaciones básicas con vectores.

Funciones matemáticas rápidas : realizan las operaciones seno, coseno y raíz.

Funciones matemáticas complejas : realiza funciones con vectores complejos como obtener el complejo conjugado, el módulo, o realizar operaciones entre vectores complejos.

Funciones con filtro : son funciones con las que se pueden aplicar los filtro *FIR* e *IIR* a señales.

Funciones matriciales : con este tipo de funciones se puede realizar cualquier operación con matrices.

Funciones de transformadas : se aplican estas funciones para obtener por ejemplo la *FFT* de una señal.

Funciones de controlador : son funciones empleadas en controladores como los *PIDs*.

Funciones estadísticas : funciones que calculan máximo, mínimo o media de un vector.

Funciones de soporte : funciones que permiten realizar conversiones entre distintos formatos o copiar vectores.

Funciones de interpolación : realizan interpolaciones lineales o bilineales.

2.2 Hardware layout

En este apartado se puede observar el diseño de la placa, en primer lugar, mediante un diagrama de bloques de toda la parte hardware (figura 2.1) y, a continuación, con las dos partes de la placa, tanto la superior (*top layout*) como la inferior (*bottom layout*), cada una de ellas con sus respectivos componentes y que se aprecian en las figuras 2.2a y 2.2b.

2.3 ST-LINK/V2-1

El *ST-LINK/V2-1* es el módulo de depuración de la tarjeta que va integrado en la misma y, mediante su conector *mini-USB* se conecta al ordenador. Es el encargado de llevar a cabo la carga de programas en la tarjeta de desarrollo y también permite depurar los programas realizados.
Presenta algunas novedades con respecto a la versión anterior de *ST-LINK* que son:

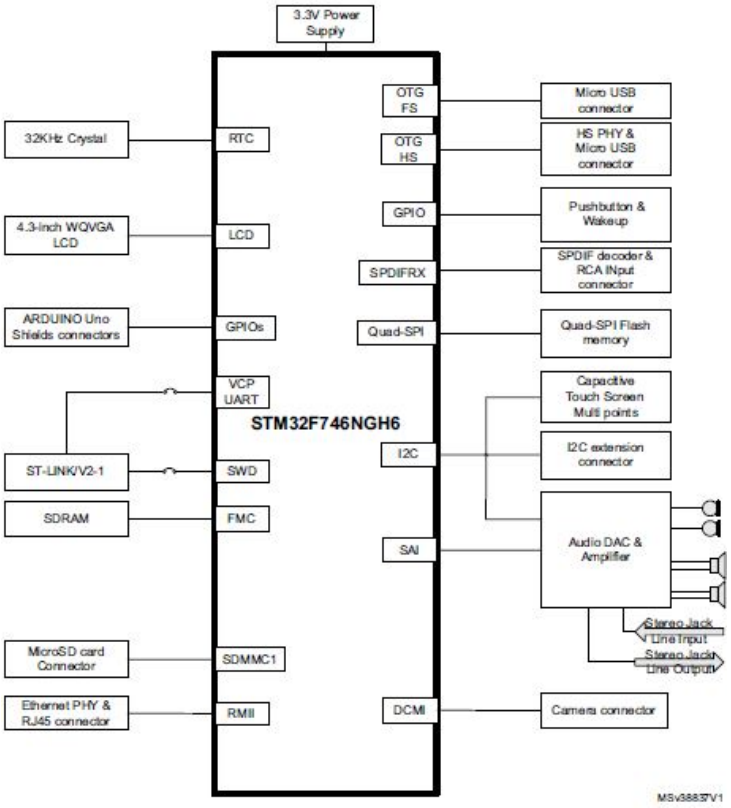


Figura 2.1: Diagrama de bloques del hardware de la tarjeta de desarrollo STM32F746G

- Reenumeración del software del USB
- Interfaz del puerto Virtual COM en el USB
- Interfaz de almacenamiento masivo en el USB
- Posibilidad de alimentar la tarjeta a través del USB

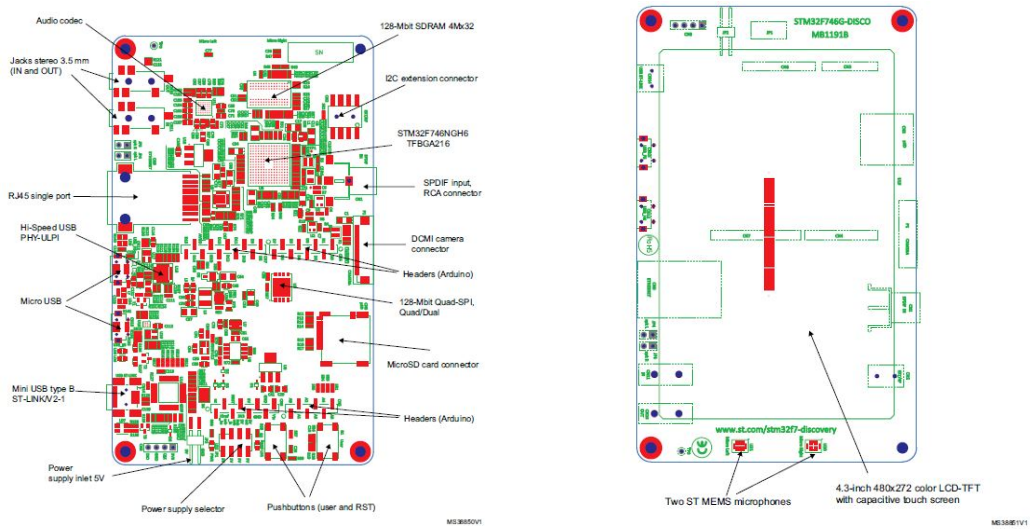
2.4 Alimentación

Hay cuatro posibles métodos de alimentación de la placa y se selecciona uno u otro en función de la configuración del *JP1*. En función del conector que se vaya a emplear para alimentar la placa, se debe colocar el *JP1* en una posición u otra.

- El primero de los tipos de alimentación es la alimentación externa.

  Poniendo el *JP1* en la primera posición, como se ve en la figura 2.3, se tiene que alimentar a la placa externamente y hay dos tipos de alimentación externa. Con la primera de ellas es necesario proporcionar, mediante una fuente de alimentación externa, 5V a la placa a través del conector *JP2*. Con el segundo tipo de alimentación externa, se proporcionan 7-12V en el pin *Vin* del conector *CN6*. Los módulos *JP2* y *CN6* se aprecian en las figuras 2.4 y 2.5 respectivamente.

- El segundo tipo de alimentación es el que se ha empleado en el proyecto y, en este caso, la alimentación se realiza a través del conector *ST-LINK*. Para ello, el conector *JP1* se ha conectado como se ve en la figura 2.6.



(a) Disposición de componentes de la capa *top*.

(b) Disposición de componentes de la capa *bottom*.

Figura 2.2: Disposición de componentes de ambas capas.

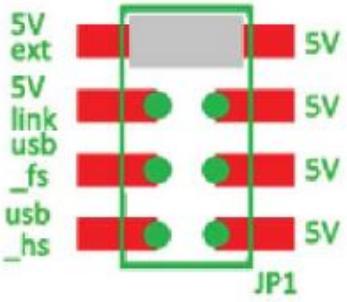


Figura 2.3: Primer tipo de conexión de *JP1*.

Realizando esta configuración en *JP1* hay que alimentar a la placa con un cable USB conectado en *CN14* y, de ahí al PC. Cabe destacar que esta conexión debe realizarse con la resistencia *R109* soldada ya que si no se puede dañar el PC y la placa no estará correctamente alimentada.

- La tercera y la cuarta opción para alimentar la placa es a través de los dos conectores *USBmicroAB* que vienen provistos en ella. En función de la configuración seleccionada en *JP1*, se alimentará a través del conector *CN12* o del *CN13*. Ambas configuraciones se pueden observar en las figuras 2.7a y 2.7b respectivamente.

2.5 Fuentes de reloj

Es un elemento clave de la tarjeta y se encarga de generar una señal de reloj periódica que proporciona sincronismo a todos los elementos. Errores en la generación de esta señal periódica pueden ser fatales para el funcionamiento del programa.
Esta tarjeta dispone de tres relojes:

- *X1*, oscilador de 24 MHz para el *USB OTG HS PHY* y el módulo de la cámara.
- *X2*, oscilador de 25 MHz para el microcontrolador *STM32F746NGH6* y el *Ethernet PHY*.

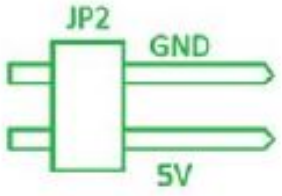


Figura 2.4: Conector *JP2*.

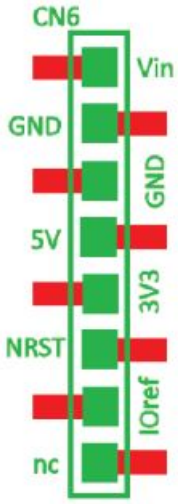


Figura 2.5: Conector *CN6*.

- *X3*, cristal de 32 kHz para el RTC (Real ime Controller) del *STM32F746NGH6*.

2.6 Fuentes de reset

La señal de reset de la tarjeta es activa a nivel bajo y hay tres fuentes que pueden provocar dicho reset:

- El botón de reset *B2* que está integrado en la placa.
- Mediante el conector *CN6* que esta previsto para Arduino.
- Mediante software, a través del anteriormente mencionado depurador que nos proporciona el *ST-LINK*.

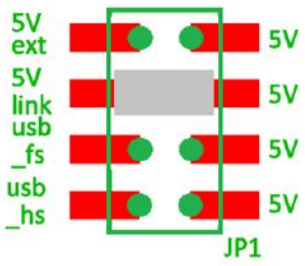
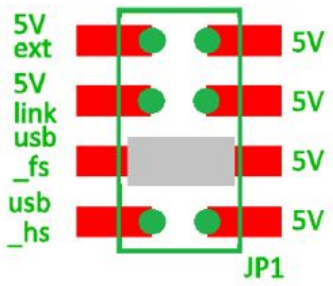
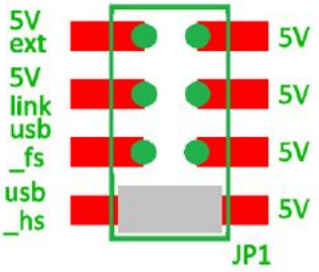


Figura 2.6: Configuración de *JP1* elegida para la alimentación de la tarjeta de desarrollo.



(a) Configuración de *JP1* para alimentar a través del conector *CN12*.



(b) Configuración de *JP1* para alimentar a través del conector *CN13*.

Figura 2.7: Tipos de conexión de alimentación tres y cuatro, seleccionados mediante el conector *JP1*.

2.7 Audio

Dentro de la importancia de todos y cada uno de los elementos de la tarjeta, este es uno de los más importantes en el desarrollo del proyecto.
El módulo de audio de la tarjeta está basado en el códec *WM8994* de *CIRRUS*, el cual dispone de 4 *DACs* y 2 *ADCs* que se conectan mediante la interfaz *SAI*(Serial audio interface), el cual se estudiará posteriormente, al microcontrolador. La comunicación se lleva a cabo a través de *I2C*. Las partes que componen este módulo de audio son las siguientes:

- Entrada analógica conectada al *ADC* del códec mediante el conector *CN11* (jack azul).
- Salida analógica conectada al *DAC* del códec mediante el conector *CN10* (jack verde).
- Dos altavoces que, mediante los conectores *JP3* y *JP4*, se pueden conectar al códec *WM8994*.
- Dos micrófonos digitales *MEMS*, que tienen la forma que se aprecia en la figura 2.8, incluidos en la tarjeta y que van conectados directamente al códec para adquirir audio.

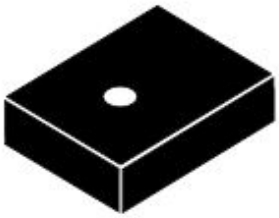


Figura 2.8: Micrófono digital MEM.

- El conector *CN1* coaxial que está implementado para recibir audio compatible con las especificaciones *SPDIF*.

Esta parte de audio se tratará con profundidad en el capítulo 3.

2.8 USB OTG

La tarjeta de desarrollo *STM32F746G-DISCO* está provista de un módulo USB para el intercambio de datos. La comunicación se lleva a cabo a través de los conectores *CN12* y *CN13*, que son conectores

USB micro-AB, en función de si se utiliza *USB OTG FS (full speed)* o *USB OTG HS (high speed)* respectivamente. En ambos casos, la placa es alimentada a 5V con un consumo máximo de corriente de 500 mA.

2.8.1 USB OTG FS

Cuando está conectado el *USB OTG FS* correctamente, es decir, cuando el chip *U6*, el de *USB power switch* está activo y hace que este USB trabaje como host, el led *LD6* se ilumina en verde. Sin embargo, si se produce un exceso de corriente se enciende en rojo El led *LD6*. Cabe destacar la importancia de que se trabaje como host para que así elementos externos se puedan conectar a la placa y, de esa manera, interactuar.

2.8.2 USB OTG HS

Al igual que en el caso de *full speed*, un chip está conectado al *VBUS*, en este caso el *U7*. Cuando se conecta este USB, el chip se activa y hace que el USB trabaje como host. En este caso el led que se enciende en verde es el *LD4* mientras que, si se produce un exceso de corriente se enciende en rojo el *LD3*.

2.9 Tarjeta microSD

Como se comentó anteriormente, la tarjeta permite la conexión de una tarjeta microSD. Esta tarjeta, que puede ser de 2-Gbytes o más, se conecta a la placa a través del puerto *SDMMC1*.
La conexión de una tarjeta nos permitirá guardar datos en ella, mediante un sistema de archivos, que posteriormente puedan ser empleados para el estudio.

2.10 Ethernet

La tarjeta $STM32F746G - DISCO$ permite realizar conexiones Ethernet de 10/100-Mbit a través de dos elementos.
El primero de ellos el conector *RJ-45* empleado en cualquier aplicación habitual para conexiones a internet y, mediante el microchip *LAN8742A-CZ-TR* que está comunicado con el microcontrolador mediante la interfaz *RMII*. Cabe destacar que de generar la señal de 25MHz, que es necesaria para el microchip, se encarga el oscilador *X2* del que se habló en la sección 2.5.

2.11 Memoria SDRAM

La función de la memoria SDRAM es la de almacenar datos durante la ejecución del programa, es decir el valor de todas las variables se almacena en esta memoria que, cuando se finalice el programa, borra todo lo que ha almacenado. Este tipo de memorias se denominan memorias volátiles.
La tarjeta cuenta con una memoria SDRAM de 128-Mbit (chip MT48LC4M32B2B5-6A) y está conectada al microcontrolador mediante la interfaz FMC. Únicamente se utilizan los 16 bits de menor peso, lo que hacen que tengamos una memoria accesible de solo 64-Mbit.

2.12 Memoria Quad-SPI Nor Flash

La memoria *Flash* es un tipo de memoria no volátil, es decir, cuando el dispositivo se apaga, los datos no se borran. Este tipo de memoria es la encargada del almacenamiento del programa principal.
La tarjeta dispone de 128-Mbit de memoria flash(chip N25Q128A13EF840E) y está conectada mediante la interfaz *Quad-SPI* al microcontrolador.

2.13 Módulo de la cámara

El conector *P1* de la placa permite conectar el módulo de la cámara (*STM32F4DIS-CAM*) que está diseñado por *STM*. La conexión del conector con la cámara se realiza a través de señales *DCMI*. El *STM32F4DIS-CAM* se puede apreciar en la figura 2.9.

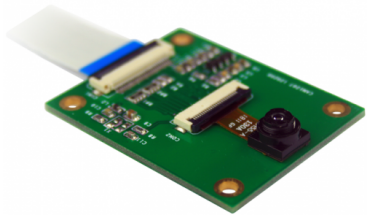


Figura 2.9: Módulo de la cámara.

2.14 Display LCD-TFT

Otro de los elementos más importantes de la tarjeta es la pantalla. Como ya se indicó en la introducción, se trata de un display LCD-TFT táctil de 4.3 pulgadas y tiene una resolución de 480x272. Se conecta al microcontrolador a través del interfaz RGB LCD.
La función que tiene el display es la de permitir que el usuario pueda moverse entre los distintos modos, de un programa, además de representar gráficas, cálculos, etc, es decir, permite la interacción hombre-máquina de la que se habló en la introducción.

Figura 2.10: Pantalla LCD-TFT de 4.3 pulgadas integradaen la placa.

Capítulo 3

Adquisición y acondicionamiento de la señal de audio

3.1 Introducción

En este capítulo se va a hablar de la parte de audio, donde entran tanto los micrófonos *MEMS*, como el códec *WM8994*, así como, la transferencia de datos entre el mismo y el microcontrolador, que se lleva a cabo a través de la interfaz *SAI*. También aquí se habla de la parte de acondicionamiento de la señal, es decir, del filtro FIR que se aplica a las dos señales acústicas y es quien se encarga de eliminar las componentes de alta frecuencia que añaden aleatoriedad a la señal.

3.2 Adquisición de la señal

La adquisición de la señal de audio consiste en un transductor que se encarga de convertir la presión acústica a la que es sometido en una señal eléctrica. La señal eléctrica resultante es la que se emplea para llevar a cabo la localización. Esta etapa de adquisición de la señal se divide a su vez en dos subsecciones.

En la primera de ellas se habla de los micrófonos *MEMS* integrados en la placa. Estos micrófonos emplean la tecnología *MEMS* (Micro-Electrical-Mechanical Systems), que es un tipo de tecnología empleada en diversas aplicaciones, tales como micrófonos, acelerómetros, giroscopios, sensores. En la segunda parte, se trata el códec *WM8994* y sus características.

3.2.1 MEMS MP34DT01

Los micrófonos son unos dispositivos de doble inyección que constan de dos componentes, el sensor y el circuito integrado, como puede verse en la figura 3.1. Estos micrófonos emplean la tecnología *MEMS*, como ya se citó antes, y tienen un precio similar al de los micrófonos de condensador tradicionales, proporcionando además una mayor exactitud y robustez. Los micrófonos consisten en un condensador de silicio que está compuesto por dos placas de silicio, una de ellas fija y la otra móvil. La superficie fija está cubierta por un electrodo, lo que la hace conductiva, y tiene numerosos agujeros que permiten el paso de la señal acústica. La superficie móvil únicamente está unida en uno de los lados de la estructura, quedando un agujero denominado agujero de ventilación en la otra parte de la estructura. Por este agujero es por

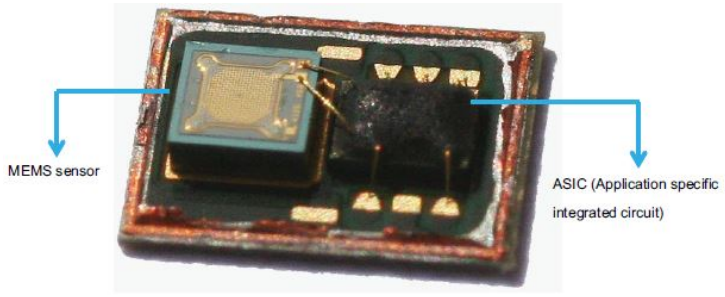


Figura 3.1: Interior de un dispositivo MEM.

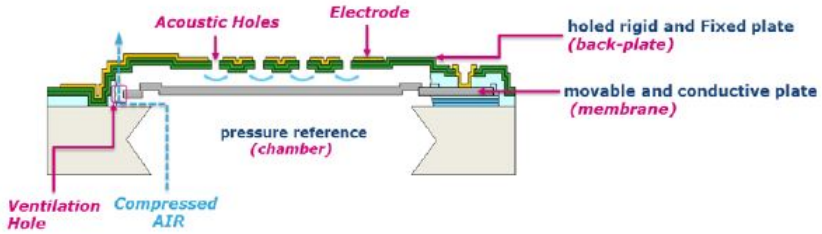


Figura 3.2: Funcionamiento de la tecnología MEMS.

donde sale el aire comprimido, lo que permite que se mueva la membrana hacia atrás como se ve en la figura 3.2.
El circuito integrado se encarga de transformar la capacitancia obtenida en una señal digital o analógica de acuerdo al tipo de micrófono que se esté empleando, en este caso una señal digital.

Los dos micrófonos integrados en la tarjeta son los ***MEMS MP34DT01***, dos sensores digitales de audio omnidireccionales cuya salida es una señal digital proporcionada en **modulación *PDM***. Cabe destacar también el bajo consumo de potencia que tienen estos micrófonos.

Estos micrófonos no son ideales y algunas de sus características van a comentarse a continuación:

Sensibilidad : consiste en la relación entre la señal eléctrica de salida del micrófono y la presión acústica que llega a la entrada del mismo y se mide en dBFS en micrófonos digitales. La presión acústica de referencia es 1Pa o, lo que es lo mismo, 94 dBSPL, a 1kHz.
Para relacionar la presión acústica con el sonido, se emplea la siguiente ecuación:

$$Sensibilidad_{dBSPL} = 20log(P/P_0) = 20log(1Pa/20uPa) = 94dBSPL \tag{3.1}$$

siendo P_0 lo mínimo audible por el oído humano.
En micrófonos digitales la sensibilidad es medida como porcentaje de la salida full-escala que es generada mediante esos 94 dBSPL de entrada.
Un punto clave en cuanto a la sensibilidad es la máxima potencia acústica permitida o AOP (*Acoustic overload point*), que tiene un valor de 120 dB. Por tanto, la sensibilidad se calcula de la siguiente manera:

$$Sensibilidad_{dBFS} = 94dB - 120dB = -26dB \tag{3.2}$$

Todo lo anteriormente comentado, se resume perfectamente en la figura 3.3, extraída de [9].

Directividad : la directividad indica la variación de la sensibilidad en función de la procedencia del sonido. Al ser estos micrófonos omnidireccionales, la sensibilidad no cambia independientemente

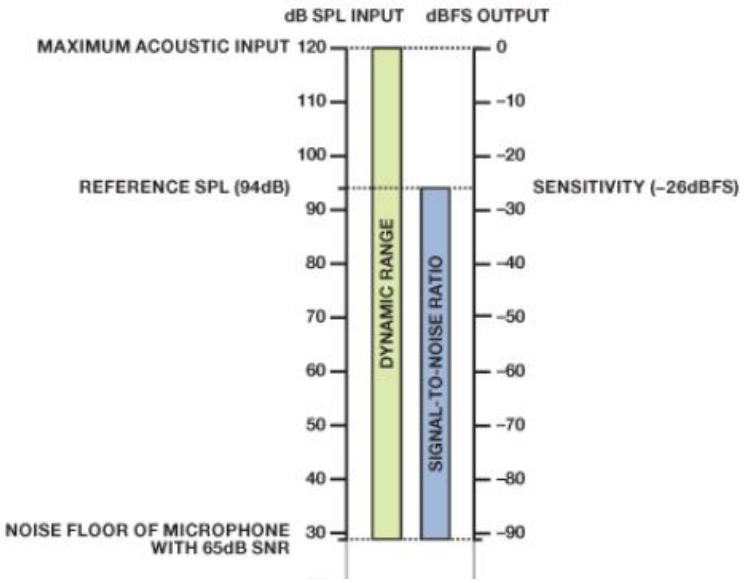


Figura 3.3: Relación entre los dominios acústico y digital.

de la procedencia del sonido, lo cual puede verse en la figura 3.4. Esto es de bastante utilidad en el tema de localización ya que, si no se recogiese sonido de alguna dirección en concreto, ésta no podría representarse.

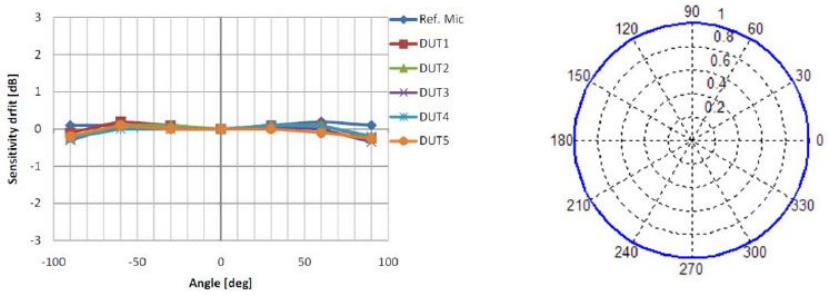


Figura 3.4: Directividad de los micrófonos MEMS MP34DT01.

SNR : la relación señal-ruido especifica la relación entre una señal de referencia dada y el ruido que se ha generado a la salida del micrófono. La señal de referencia es la señal a la salida del micrófono cuando la presión es de 1 Pa. El valor de SNR para este micro es de 63 dB. En la figura 3.5 se aprecia como varía *SNR* en función de la frecuencia.

Ruido equivalente de entrada : Cuando se produce la transformación de la presión acústica de la entrada en una señal eléctrica a la salida en un micrófono, se genera un ruido. Ese ruido expresado en dBSPL se denomina ruido equivalente de entrada y se calcula de la siguiente manera:

$$Ruido\_generado = Sensibilidad - SNR = -26 - 63 = -89dBFS \tag{3.3}$$

y en el dominio acústico:

$$Ruido\_equivalente\_entrada = 94 - 63 = 31dBSPL \tag{3.4}$$

como puede apreciarse en la figura 3.6 esto equivale a una habitación en silencio.

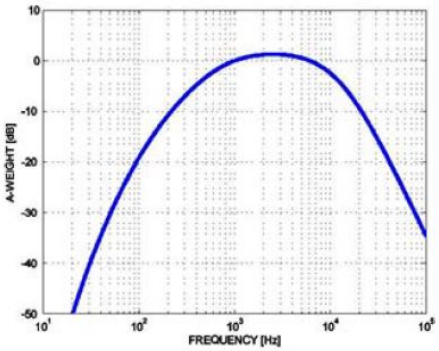


Figura 3.5: SNR en función de la frecuencia.

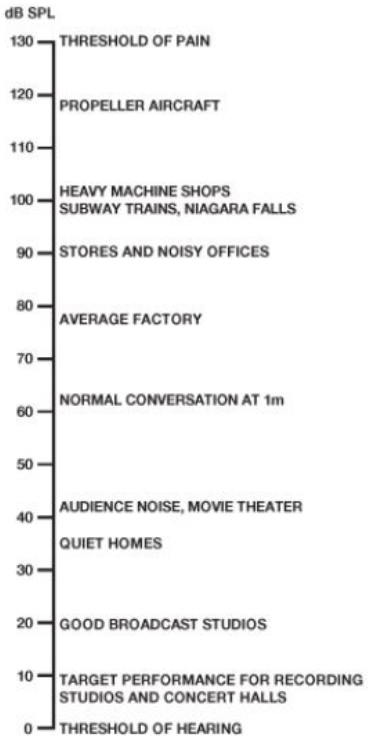


Figura 3.6: Niveles de presión en varios escenarios.

Respuesta en frecuencia : en función del rango de frecuencias en el que se esté trabajando, en estos micrófonos se produce una variación en su sensibilidad. Como puede apreciarse en la figura 3.7 sacada directamente del *datasheet* del fabricante.

La respuesta en frecuencia de un micrófono en términos de fase indica la distorsión de fase introducida por el micrófono.

En cuanto a la modulación *PDM* (Pulse Density Modulation), comentar que es una modulación que permite convertir una señal analógica a digital. No es la modulación más empleada, pero suele emplearse mucho en smartphones para transferir el sonido recogido por los micrófonos hacia el procesador de señales del propio dispositivo. Esta modulación es más sencilla que la *PCM* (Pulse Code Modulation), en la cual se utilizan varios bits para representar la señal, ya que, únicamente se emplea un bit y, la amplitud de la señal se determina mediante la densidad de impulsos en función del tiempo como se aprecia en la figura 3.8.

Muchos de los micrófonos cuya salida es *PDM* soportan audio estéreo y, en estos caso, la información

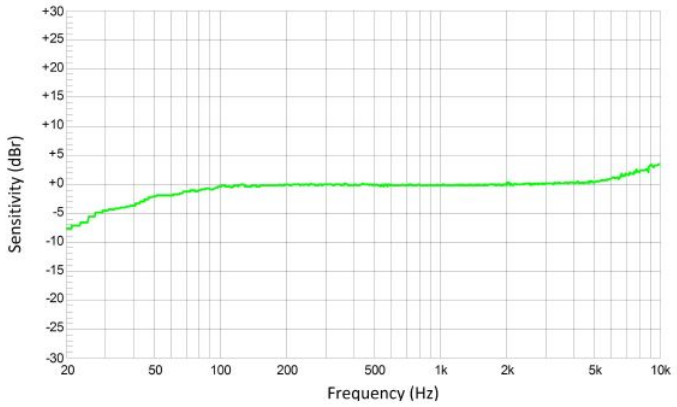


Figura 3.7: Respuesta en frecuencia de los micrófonos.

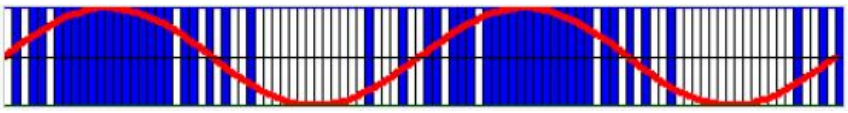


Figura 3.8: Dos períodos de una señal seno modulada con PDM y de 100 muestras.

de uno de los micrófonos se proporciona en el flanco de subida de la señal de reloj mientras que, la del otro micrófono se proporciona en el flanco de bajada como se puede ver en la figura 3.9 sacada del propio *datasheet* de los micrófonos. Esto es de vital importancia para el desarrollo del proyecto ya que, así, se puede separar la información de cada uno de los micrófonos y, de esta manera, manipular cada una de las señales por separado para llegar al objetivo final de la obtención de la dirección del sonido a través de GCC, la cual se explicará en el capítulo 4.

3.2.2 Códec de audio WM8994

El **códec de audio** que integra la placa es el *WM8994* de *CIRRUS*. Se trata de un códec de audio de baja potencia y alta calidad diseñado para interactuar con una amplia gama de procesadores y componentes analógicos, y se suele emplear sobretodo en smartphones, tablets, y dispositivos de audio y navegación portátiles. Una característica que tiene es su inmunidad al ruido, provocada por su arquitectura interna diferencial y por los filtros de ruido que incorpora.
El códec dispone de tres pines de interfaz de audio que proporcionan independencia y la capacidad de

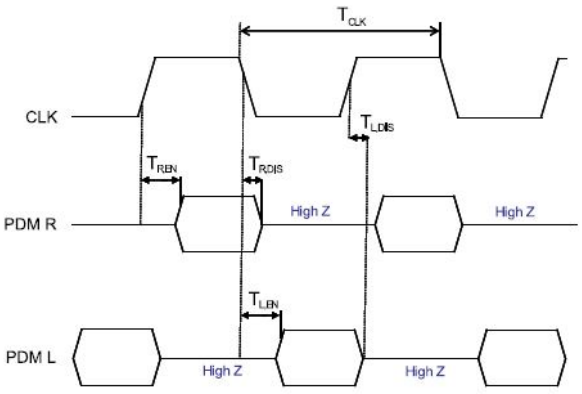


Figura 3.9: Cronograma de las señales de los micrófonos.

poder realizar conexiones asíncronas a múltiples dispositivos. Estas interfaces de audio suelen ser un procesador de aplicaciones, un procesador de banda base y un transmisor inalámbrico.
Además dispone de cuatro canales de entrada para micrófonos digitales, cuatro canales para DAC y ocho entradas analógicas, las cuales permiten la conexión de hasta cuatro micrófonos.
La configuración de amplificación del códec ha sido diseñada para proporcionar la mayor eficiencia y el mínimo consumo en reproducción de música y llamadas de voz.

Los micrófonos *MEMS* están interconectados con el códec de la manera que se ve en la figura

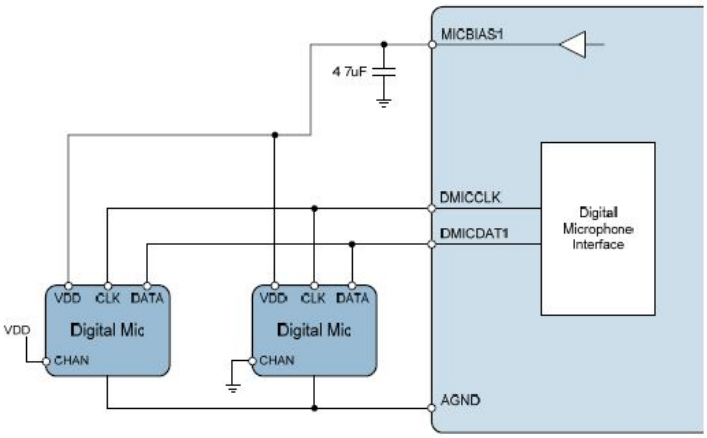
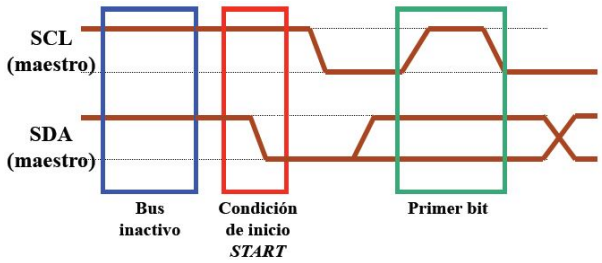


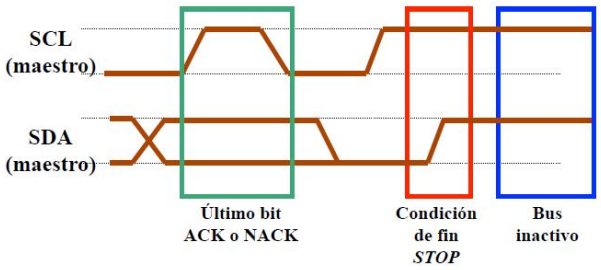
Figura 3.10: Diagrama de bloques de la conexión entre los micrófonos y el códec.

3.10. Se puede apreciar que es el códec quien proporciona la señal de reloj a los micrófonos y éstos le transfieren la información.
Uno de los puntos más importantes en el estudio del códec es la forma de transmisión de audio. Esta es realizada a través de la **interfaz SAI** y se realiza mediante *I2C*. El códec dispone de tres modos de interacción, *I2C*(2 cables), *SPI*(3 cables) y *SPI*(4 cables). El pin que indica cuál de los tres se emplea es el pin *CIFMODE*, el cual, la tarjeta lo integra configurado a nivel bajo, por lo que es por *I2C*.
La **comunicación I2C** se realiza entre dos dispositivos en los que uno actúa como maestro y el otro como esclavo. El dispositivo que actúa como maestro, que en este caso es la interfaz SAI, es el encargado de generar la señal de reloj y de empezar y terminar la transferencia. El códec WM8994 es el encargado de realizar las funciones de esclavo. Los datos, que se transmiten en formato *byte*, son transferidos por el bus de datos bidireccional, denominado *SDA* y, todos y cada uno de los *bytes* necesitan confirmación al llegar al destino mediante un *ACK*. Antes de que se produzca el inicio de la transmisión de datos, el maestro se encarga de iniciar la comunicación, mediante un flanco de bajada en el bus de datos como se ve en la figura 3.11a. Tras esto, se envía la dirección de memoria en la que se quiere escribir o leer y el bit que determina dicha operación.

El fin de la transmisión se realiza de la manera que se ve en las figura 3.11b, mediante un flanco de subida en el bus de datos SDA.
Para que se entienda mejor como se realiza el envío de datos del maestro al esclavo y viceversa, se expone en las figuras 3.12a y 3.12b. Por último, para terminar con la transferencia I2C, se muestra una imagen como la de la figura 3.13 en la que se realiza una escritura y lectura del maestro sobre el esclavo cambiando de dirección de acceso.



(a) Cronograma de la señal de inicio de la transferencia I2C.



(b) Cronograma de la señal de fin de la transferencia I2C.

Figura 3.11: Cronogramas de las señales de inicio y fin de transferencia I2C.

3.3 Acondicionamiento de la señal

En esta sección se va a hablar del filtro FIR que se le aplica a las señales que recogen los dos micrófonos de la placa.

3.3.1 Filtro FIR (Finite Impulse Response)

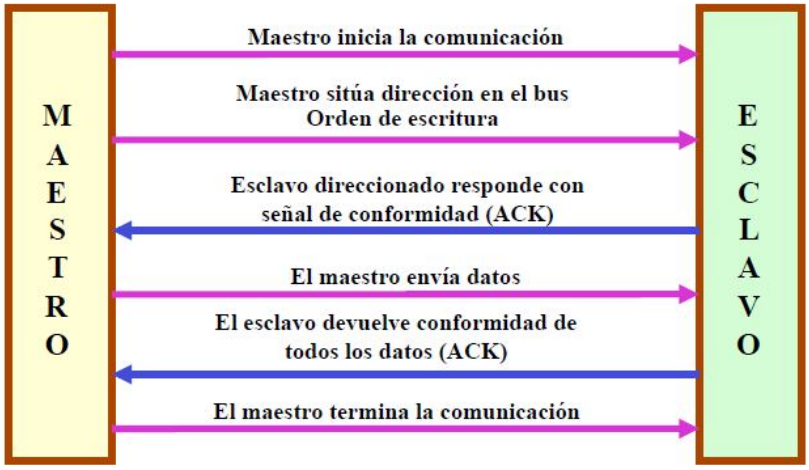
El **filtro FIR** es lo primero que se aplica a las dos señales obtenidas y consiste en un filtro digital que tiene una respuesta finita al impulso, de ahí las siglas FIR. Se usa dicho filtro ya que se trata de un filtro de fase lineal, lo cual quiere decir que el desfase introducido tras su aplicación es lineal para cada componente y, por tanto, no va a existir distorsión de fase. Esto es de vital importancia ya que se está trabajando con señales acústicas, cuyas fases no deben verse alteradas si se quieren obtener posteriormente unos resultados exactos [7], debido a que la información de localización se encuentra en la fase de la señal.

Los filtros FIR son filtros que se basan en obtener la salida a partir de las entradas actuales y anteriores y la ecuación que los define es la 3.5:

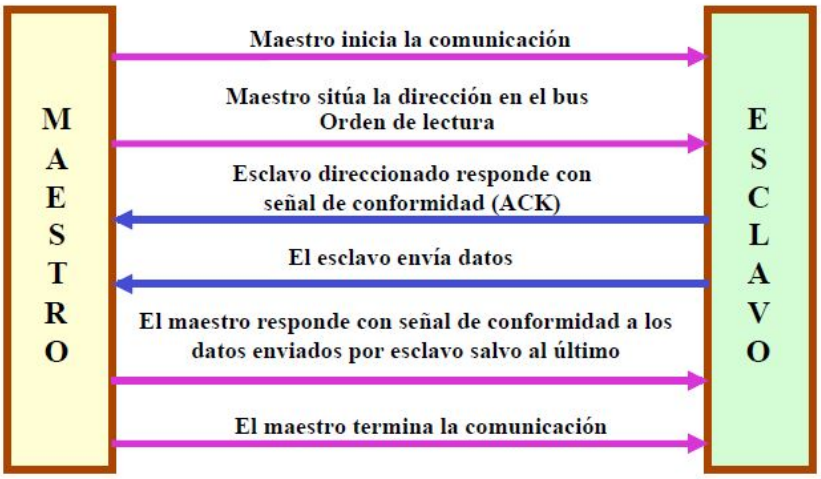
$$y(n) = b_0 x(n) + b_1 x(n-1) + ... + b_{N-1} x(n-N+1) = \sum_{k=0}^{N-1} b_k x_{n-k} \tag{3.5}$$

donde:

- **y(n)** es la señal de salida.
- **x(n)** es la señal de entrada.
- **b** es el valor de la respuesta al impulso.
- **N** determina el orden del filtro. Cabe destacar que el filtro tendrá N+1 coeficientes.



(a) Envío de datos del maestro al esclavo.



(b) Envío de datos del esclavo al maestro.

Figura 3.12: Esquema de la transmisión de datos por I2C.

Figura 3.13: Maestro lee y luego escribe en el esclavo previo cambio de dirección.

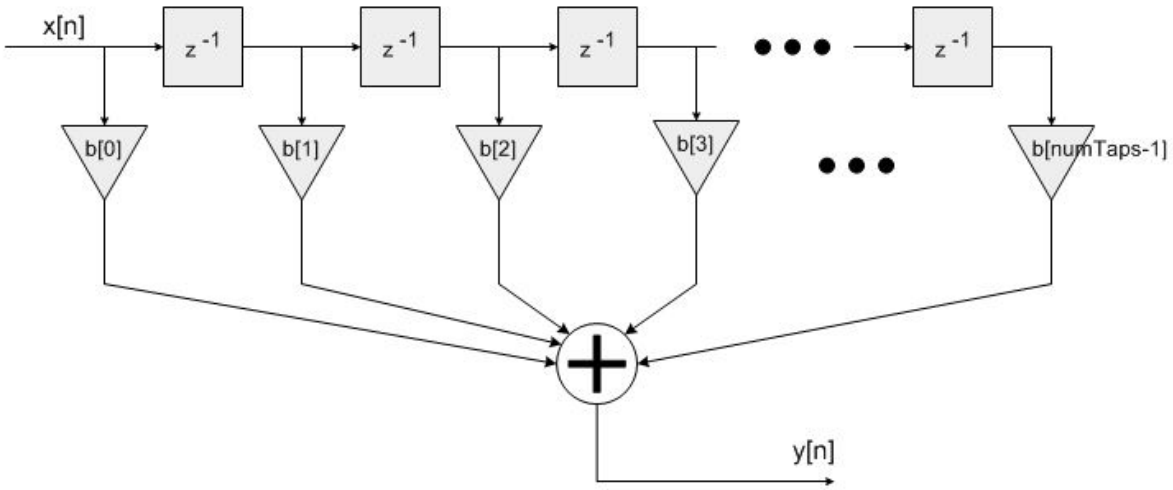


Figura 3.14: Filtro FIR.

Tabla 3.1: Tipos de filtros FIR de fase lineal.

Tipo	**Simetría**	**Longitud**	**Respuesta al impulso**
Tipo I	Simétrico	Impar	h(k) = h(N − 1 − k)
Tipo II	Simétrico	Par	h(k) = h(N − 1 − k)
Tipo III	Asimétrico	Impar	h(k) = −h(N − 1 − k)
Tipo IV	Asimétrico	Par	h(k) = −h(N − 1 − k)

Ante un impulso, la respuesta es finita, de ahí el nombre del filtro, siendo la salida la convolución de la entrada con la respuesta al impulso.

$$y(n) = \sum_{k=-\infty}^{\infty} h_k x_{n-k} \tag{3.6}$$

siendo h=0 para k<0 y k>=N.

Se puede demostrar que para que un filtro FIR sea lineal h_k tiene que ser simétrico, y hay cuatro tipos de filtro FIR de fase lineal según se ve en la tabla 3.1. El filtro FIR que se emplea en el proyecto es de tipo 1.

Hay varios métodos para el diseño del filtro FIR pero, el que se ha utilizado y uno de los más empleados es el método de ventanas. En este método se determina la respuesta al impulso unidad a partir de las especificaciones de la respuesta en frecuencia que se desea. Esto se aprecia en las ecuaciones de la frecuencia deseada (3.7) y de la respuesta al impulso unidad (3.8):

$$H_d(\omega) = \sum_{n=0}^{\infty} h_d(n) e^{-j\omega n} \tag{3.7}$$

$$h_d(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H_d(\omega) e^{j\omega n} d\omega \tag{3.8}$$

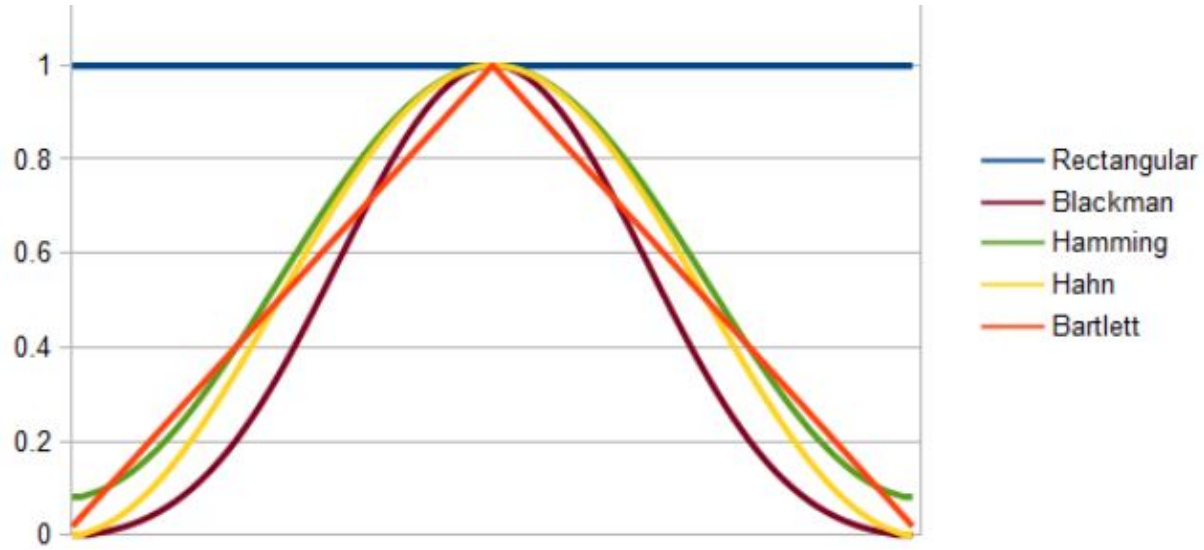


Figura 3.15: Distintos tipos de ventanas empleadas para realización del filtro FIR.

Debido a que la respuesta al impulso de la ecuación 3.8 es infinita tiene que poder truncarse y, es ahí donde entra en juego la ventana que da nombre al método. Para poder truncar la respuesta al impulso, lo que se hace es multiplicar la respuesta al impulso por una ventana [10]. Tras hacer esto, se tiene únicamente la respuesta al impulso multiplicada por el valor de la ventana en el ancho de la misma, siendo 0 en todo el rango fuera de ella.
Para continuar con la explicación se asume que se ha empleado la ventana rectangular, cuyo valor es 1, aunque, en el proyecto se ha empleado la ventana *Hamming*, cuya secuencia en el dominio del tiempo se ve en la tabla 3.2. Por tanto, la representación en el dominio de la frecuencia de la función de ventana es:

$$W(\omega) = \sum_{n=0}^{M-1} w_n(n) e^{-j\omega n} \tag{3.9}$$

donde **M** es la longitud del filtro. Sabiendo esto, se puede obtener la respuesta en frecuencia del filtro FIR ya truncado que se ve en la siguiente ecuación:

$$H(\omega) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H_d(v) W(\omega - v) dv \tag{3.10}$$

Tras esta presentación del filtro FIR, se muestra la tabla 3.2 en la que aparecen distintos tipos de ventana y su secuencia temporal destacando de nuevo que la ventana empleada ha sido la ventana de *Hamming*. En la figura 3.15 pueden observarse también dichos tipos de ventanas.

Tabla 3.2: Distintos tipos de ventana empleados para el diseño de un filtro FIR.

Nombre ventana	**Secuencia en el dominio del tiempo**
Rectangular	$\mathrm{w(n)} = 1$
Bartlett o triangular	$\mathrm{w(n)} = 1 - \frac{2\lvert \mathrm{n} - \frac{\mathrm{M}}{2} \rvert}{\mathrm{M}}$
Blackman	$\mathrm{w(n)} = 0{,}42 - 0{,}5\cos\frac{2\pi\mathrm{n}}{\mathrm{M}} + 0{,}08\cos\frac{4\pi\mathrm{n}}{\mathrm{M}}$
Hamming	$\mathrm{w(n)} = 0{,}54 - 0{,}46\cos\frac{2\pi\mathrm{n}}{\mathrm{M}}$
Hanning	$\mathrm{w(n)} = 0{,}5 - 0{,}5\cos\frac{2\pi\mathrm{n}}{\mathrm{M}}$

Capítulo 4

Localización de fuentes sonoras

4.1 Introducción

En este capítulo se realiza un estudio teórico de los elementos que se encargan de realizar el procesado de la señal para llegar al objetivo de la localización del emisor. Se van a explicar conceptos como *TDOA* y el método de *GCC-PHAT*, que son los dos elementos básicos que sustentan el proyecto, entrando a tratar también *FFT*.

Para llegar a la obtención de la dirección del sonido, se hace, como se citó en la introducción, una estimación del tiempo de llegada de la señal del emisor a cada uno de los micrófonos y un cálculo del retardo entre ambas señales. Con esto, podrá emplearse el método de GCC para llegar al objetivo pero, antes de obtener la correlación y de poder aplicar los conocimientos trigonométricos, es necesario realizar una serie de modificaciones a las señales. Primero se va a hablar de los distintos métodos de localización de fuentes sonoras existentes para, después centrar la explicación sobre el método empleado, TDOA. Luego se realiza un análisis de los efectos que provoca la discretización y también se comentan los conceptos trigonométricos empleados, así como la explicación de FFT. También en la parte de FFT, se comenta la técnica de *zero padding* que se realiza antes del cálculo de la FFT. Por último van a tratarse, el método de GCC-PHAT, del que se habló también en el capítulo 1, empleado para evaluar la diferencia entre ambas señales obtenidas, y del método *Welch*.
Todo este proceso por el que se pasa a las señales para llegar al objetivo puede verse claramente en el diagrama de bloques de la figura 4.1.

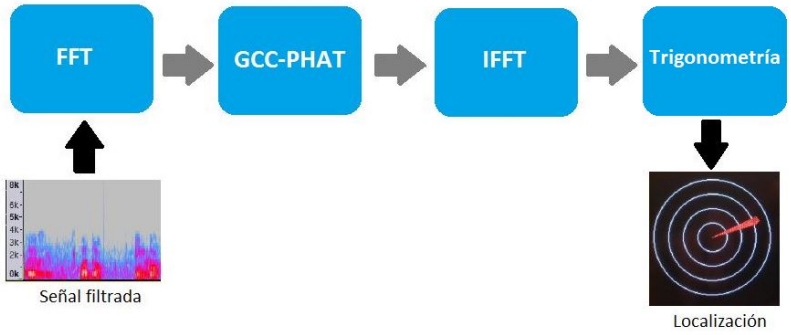


Figura 4.1: Diagrama de bloques de la obtención de la localización.

4.2 Estrategias de localización de fuentes sonoras

Como se comentó en la introducción, el objetivo final es conocer la localización del emisor de sonido. Hay tres grandes procedimientos que nos llevan a ese objetivo:

- Basados en maximizar la respuesta de potencia (SRP).
- Técnicas basadas en los conocimientos de alta resolución espectral para llegar a la localización.
- Técnicas que emplean la diferencia del tiempo de llegada de la señal del emisor al receptor entre dos receptores (**TDOA**).

En el primero de los métodos, SRP, se llega a la estimación de la localización mediante una señal recibida de los micrófonos que ya ha sido filtrada, ponderada y sumada [1]. Este método emplea la técnica de *beamforming* que, básicamente, realiza una maximización o minimización de las estadísticas espaciales asociadas con cada posición para llegar a la obtención de la localización [11]. SRP se basa en la potencia de la señal recibida, en concreto, en los datos que se obtienen cuando los micrófonos están dirigidos en la dirección de una localización específica, la dirección del emisor que es objeto de estudio. Cuando se emplea esta técnica se usan distintos filtros, siendo el más empleado el filtro *PHAT* que también se usa en este proyecto y se comentará a continuación.

En cuanto a las técnicas de alta resolución espectral, incluyen los métodos modernos basados en *beamforming* adaptados al campo del análisis de la alta resolución espectral [1]. Aprovechan la descomposición espectral de la matriz de covarianza de las señales de entrada para mejorar la resolución espacial del algoritmo [11]. Cabe destacar que estas técnicas son menos complejas y más sensibles a los pequeños errores de modelado que las técnicas basadas en *beamforming* como SRP. Por el contrario, las técnicas basadas en alta resolución espectral asumen que las fuentes de sonido son ideales y que se conoce la posición exacta de los micrófonos, condiciones que en entornos reales no son posibles [1].

El método que se ha empleado para llevar a cabo la localización es *TDOA* y se va a desarrollar en la siguiente sección.

4.3 TDOA (Time Difference of arrival)

El método empleado TDOA realiza la localización en dos pasos. Un primer paso, en el que se realiza la estimación de la diferencia del tiempo de llegada de la señal recibida entre los dos micrófonos de los que dispone la placa y, un segundo paso, en el que se aplican las ecuaciones trigonométricas y se resuelven las ecuaciones resultantes [7].
En la figura 4.2 se puede apreciar un par de micrófonos que simularían a los existentes en la placa y una fuente emisora de sonido. Esta fuente emite una señal que, en general, emplea más tiempo en llegar a un micrófono que al otro. Esto es debido a que no están ubicados en la misma posición. En la derecha de la misma figura, se observa la diferencia entre ambas señales, y se puede ver como una de ellas está retardada τ con respecto a la otra.
Teniendo en cuenta esto, la señal recibida por el micrófono 1, que es el que está en la imagen más lejano al emisor, será la siguiente:

$$x_{micrófono1} = At_{m1}(t)s(t-\tau) + No_{m1}(t) \tag{4.1}$$

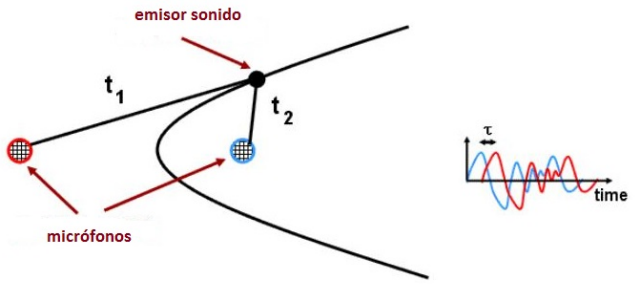


Figura 4.2: Imagen en la que se explican los fundamentos de TDOA.

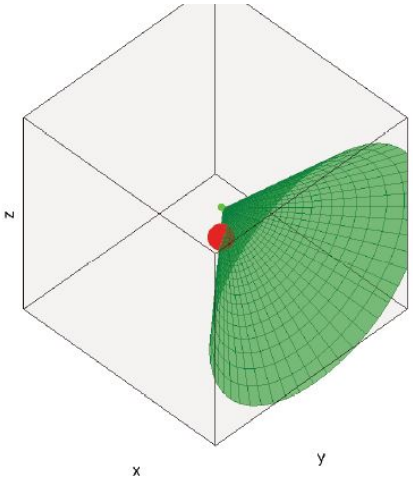


Figura 4.3: Hiperboloide de revolución que representa los puntos geométricos del espacio que comparten la misma diferencia de tiempos de llegada.

donde los parámetros At_{m1} y No_{m1} son la atenuación de la señal al micrófono 1 y el ruido captado por dicho micrófono respectivamente. Como los micrófonos están separados únicamente 2 cm, distancia relativamente pequeña en comparación con la distancia a la que se encuentra la fuente emisora del sonido, se asume que la atenuación que reciben ambos micrófonos es la misma por lo que la señal recibida en ambos micrófonos sería prácticamente igual pero retardada como se puede apreciar.

En la figura 4.2 se pueden apreciar los dos micrófonos y una hipérbola que está formada por los puntos geométricos del espacio que comparten la misma diferencia de tiempos de llegada, o lo que es lo mismo, τ.
Cuando nos encontramos en el caso tridimensional, esta hipérbola se convierte en una hiperboloide de revolución como la de la figura 4.3. Por tanto, para obtener una buena localización, es necesario disponer de varias de estas hiperboloides, lo cual se consigue con varios pares de micrófonos. Esto se encuentra lejos del objetivo de este proyecto pero, a su vez, forma parte del sistema completo que se está desarrollando. Para simplificar los cálculos, se asume que el locutor está lo suficientemente lejos (el doble de los 2 cm que separan a ambos micrófonos) y, de esta manera, el ángulo de llegada de la señal se puede aproximar al ángulo de la asíntota de la hipérbola comentada, como se ve en la figura 4.4.

Habiendo llegado ya a esta simplificación representada por la figura 4.4, se sabe que el valor máximo de τ será el de la separación entre los micrófonos, que se producirá cuando el emisor se encuentre alineado con ambos micrófonos.

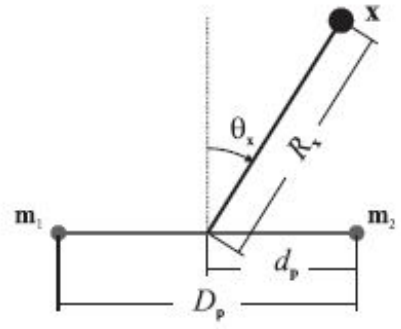


Figura 4.4: Conceptos trigonométricos aplicados para la localización.

Además, se sabe que el ángulo que forma la señal emitida con la normal es de θ_x, que es el mismo ángulo que forma una línea imaginaria perpendicular desde el micrófono 2 con la señal emitida(R_x). Por tanto, tenemos las ecuaciones

$$\sin(\theta_x) = \frac{\tau}{\frac{D_p}{C}} \tag{4.2}$$

$$\theta_x = \arcsin \frac{\tau C}{D_p} \tag{4.3}$$

4.4 Efectos de la discretización

En esta sección se continua con la explicación de cómo obtener la localización a través de la técnica TDOA y se comentan los efectos que la discretización produce en dicha localización.

Sabiendo que c es la velocidad del sonido, 343 m/s y que $\tau_{MAX} = \frac{D_p}{C}$. τ en la figura sería la distancia que hay desde el corte de la línea imaginaria trazada desde el micrófono 2 y perpendicular a la señal que se recibe con otra línea imaginaria paralela a la señal recibida que saldría desde el micrófono 1, hasta el propio micrófono 1.
Para llegar a la localización de la fuente emisora del sonido, únicamente es necesario conocer el valor de la frecuencia de muestreo, que es de 48 kHz, y el valor máximo que puede tener τ con la separación existente de los micrófonos de 2 cm, así como las ecuaciones (4.2) y (4.3) que se han descrito anteriormente. Entonces:

$$N_{muestras} = \tau_{MAX} \cdot f_{muestreo} \tag{4.4}$$

se obtiene un número de muestras relevantes igual a 3, lo cual quiere decir que se tienen tres muestras positivas, tres negativas y la muestra 0, siete muestras tienen relevancia, por lo que se discretiza la dirección de llegada del sonido a siete posibles ángulos. Si se dispusiese de una distancia mayor entre los micrófonos, la discretización sería menor y el cálculo más exacto incluso para una frecuencia de muestreo inferior. Para entender mejor lo comentado, en la figura 4.5 puede apreciarse un ejemplo donde se detecta la correlación. El valor del eje X que coincide con el máximo valor del eje Y sería el valor de τ que buscamos.
Para terminar con la localización, es necesario comentar la incertidumbre que se genera provocada por la hiperboloide de revolución que se ha desarrollado en la sección 4.3. Como ya se comentó, esta hiperboloide representa los puntos geométricos del espacio que comparten la misma diferencia de tiempos de llegada, lo que produce que, empleando únicamente dos micrófonos haya una confusión en si el emisor se encuentra delante o detrás de los micrófonos. Como no se dispone de más micrófonos, se ha optado

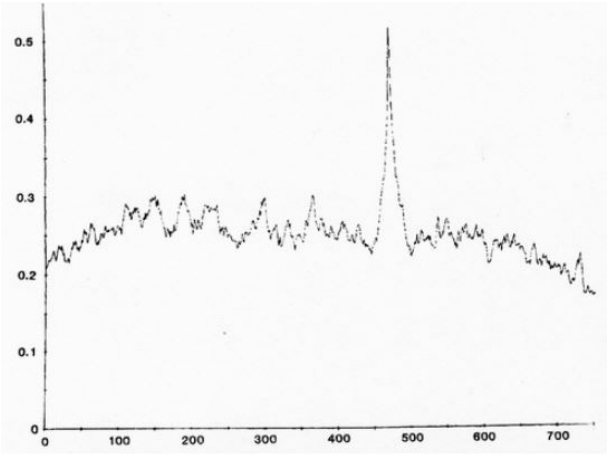


Figura 4.5: Imagen en la que se aprecia que τ es el valor del eje X donde se produce el máximo del eje Y.

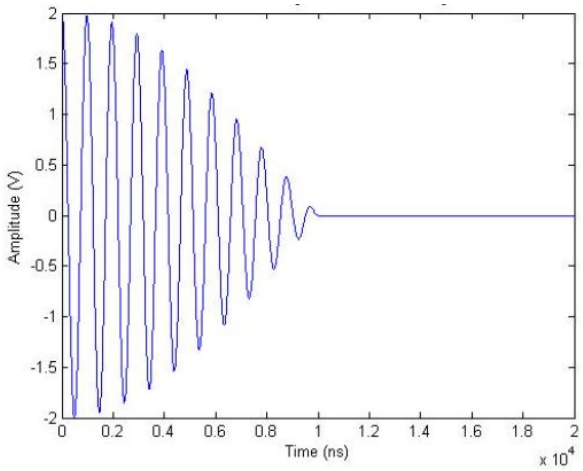


Figura 4.6: Señal a la que se le ha aplicado la técnica de zero padding.

por solucionar esto teniendo en cuenta únicamente 180º para llevar a cabo la localización.

A continuación se va a hablar de FFT, elemento clave para la obtención de la localización.

4.4.1 FFT (Fast Fourier Transform)

Antes de entrar en la explicación de la **FFT**, se va a introducir el concepto de **zero padding**, necesario debido al efecto que provoca el fenómeno de la convolución circular [7]. Como el resultado de la convolución de las señales es más grande en longitud que cada una de las señales, los resultados tras convolucionar se solaparían, por lo que para evitar esto, se amplía el tamaño del buffer que contiene la señal de entrada al doble de su tamaño original y se rellena con ceros. Esta técnica de zero padding puede observarse en la figura 4.6.

Para llevar a cabo el cálculo de GCC, método empleado para llegar a TDOA, es necesario recurrir a la utilización de la FFT, la cual es de amplia utilización en el procesado de señales. La FFT es un algoritmo eficiente que sirve para realizar de forma eficiente la transformada discreta de Fourier o DFT. La DFT lo que realiza es la conversión de una señal en el dominio del tiempo a una representación en

frecuencia y se basa en la siguiente ecuación:

$$X[k] = \sum_{k=0}^{N-1} x[n] e^{(-j2\pi kn)/N} \tag{4.5}$$

donde $e^{\frac{-j2\pi}{N}} = W_N$.

Hay varios métodos que permiten obtener la FFT a través de la DFT, a continuación se explica el algoritmo mediante diezmado en el tiempo. Este método consiste en la descomposición de los cálculos en sucesivas DFT más pequeñas. Está sustentado por las propiedades de simetría y periodicidad [12]. Se suele utilizar un valor de N igual a una potencia de dos, así se tiene una N de un número entero y par y se puede separar cada secuencia en un par de secuencias de la mitad de tamaño que la original. En las siguientes ecuaciones se aprecia lo comentado:

$$X[k] = \sum_{npar} x[n] W_N^{nk} + \sum_{nimpar} x[n] W_N^{nk} \tag{4.6}$$

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r] W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1] W_N^{(2r+1)k} \tag{4.7}$$

conociendo las propiedades comentadas en el párrafo anterior y que se ven en las siguientes dos ecuaciones:

$$W_N^2 = W_{N/2} \tag{4.8}$$

$$W_N^{n+N} = W_N^n \tag{4.9}$$

queda la siguiente ecuación:

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r] W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1] W_{N/2}^{rk} \tag{4.10}$$

En la ecuación (4.10) se aprecia que la primera suma es la de los elementos pares y, la segunda, la de los elementos impares. Posteriormente se combinan como se indica en la ecuación (4.11) para poder obtener la X[k] final del número de puntos total N.

$$X[k] = Y[k] + W_N^k Z[k] \tag{4.11}$$

donde k irá desde 0 hasta N-1.

4.5 GCC (Generalized cross correlation)

Para realizar el cálculo de TDOA, como ya se comentó, se ha empleado la correlación cruzada generalizada (**GCC**). Este método fue propuesto en 1976 por Knapp y Carter y se basa en la ecuación (4.1) pero, en este caso, ya se ha quitado el término de la atenuación al no ser relevante.

$$x_{micrófono1} = S_{m1}(t-\tau) + No_{m1}(t) \tag{4.12}$$

$$x_{micrófono2} = S_{m2}(t) + No_{m2}(t) \tag{4.13}$$

Hay que tener en cuenta que, según la figura 4.4, el micrófono más alejado es el número 1, por eso es el que tiene el retardo τ. Tras pasar al dominio de la frecuencia se tienen las siguientes ecuaciones:

$$X_{micrófono1}(\omega) = S_{m1}(\omega)e^{-j\omega\tau} + No_{m1}(\omega) \tag{4.14}$$

$$X_{micrófono2}(\omega) = S_{m2}(\omega) + No_{m2}(\omega) \tag{4.15}$$

Estas dos señales son las recibidas tanto en uno como en otro micrófono, tras haberse aplicado en ellas la FFT(Fast Fourier Transform), explicada en la subsección 4.4.1. **GCC** consiste en evaluar la medida del desfase que existe entre dos señales. Esto se consigue realizando la convolución de una de las señales con el complejo conjugado de la otra. Como estamos en el dominio de la frecuencia, antes de realizar la convolución de ambas señales será necesario haber obtenido el espectro en frecuencia.
GCC proporciona la ecuación siguiente:

$$R_{x1x2}(\omega) = \sum_{\omega} = \psi_{x1x2}(\omega)X_1(\omega)X_2^*(\omega)e^{-j\omega\tau} \tag{4.16}$$

En la ecuación anterior se tiene:

- $\psi_{\mathrm{x1x2}}(\omega)$ que es la función del filtro que se aplica a la GCC. En este caso **PHAT**.
- $\mathrm{X}_1(\omega)$ señal recibida en el micrófono 1 tras la aplicación de la FFT.
- $\mathrm{X}_2^*(\omega)\mathrm{e}^{-\mathrm{j}\omega\tau}$ complejo conjugado de la señal recibida en el micrófono 2 tras la aplicación de la FFT.

El filtro que se utiliza en este proyecto es el filtro PHAT, que es el que se ha demostrado que en condiciones reales produce mejores resultados [8]. Lo que realmente realiza este filtro es, eliminar la influencia de la amplitud de la señal que se recibe, para que la diferencia y lo que va a determinar la dirección del sonido la marque la fase. El filtro PHAT está definido por la ecuación:

$$\psi_{x1x2}(\omega) = \frac{1}{X_1(\omega)X_2(\omega)^*} \tag{4.17}$$

Por último, comentar brevemente que, además del filtro PHAT utilizado, que hace que **GCC-PHAT** nos lleve a la obtención de la localización del emisor mediante TDOA, existen otra serie de filtros que pueden apreciarse en la tabla 4.1:

4.6 Coherencia

En este último subapartado de la localización, se va a hablar de la coherencia, de por qué se ha empleado y en qué consiste.
Puesto que en el presente proyecto, es necesaria la adquisición continua de sonido, hay que ir adquiriendo bloques de sonido y haciendo el procesado de cada bloque. Mientras se procesan unas muestras, las siguientes se están adquiriendo, por lo que se tiene que determinar un tamaño para ese bloque, el cual se ha determinado que es de 1024 muestras debido a las limitaciones que impone el microprocesador con respecto a la función de la FFT. Esta función está limitada a un máximo de 4096 muestras, las cuales se dividen en 2048 elementos reales y 2048 elementos complejos. Como el sonido tiene solo parte real, los 2048 elementos reales vienen de, 1024 elementos que no son nulos y contiene sonido y otros 1024, con valor 0, que se emplean para la realización de la técnica de *zero padding* que se comentó anteriormente.

Tabla 4.1: Distintos tipos de ventana empleados para el diseño de un filtro FIR.

Nombre filtro	**Función**
Cross-correlation	1
ROTH	$1/\mathrm{G_{xy}(f)}$
SCOT	$1/\sqrt{\mathrm{G_{xx}(f)G_{xy}(f)}}$
PHAT	$1/\lvert\mathrm{G_{xy}(f)}\rvert$
HT	$\frac{\lvert\gamma_{\mathrm{xy}}(\mathrm{f})\rvert^2}{\lvert\mathrm{G_{xy}(f)}\rvert(1-\lvert\gamma_{\mathrm{xy}}(\mathrm{f})\rvert^2)}$

A una frecuencia de 48 kHz, estos 1024 elementos, se convierten en aproximadamente 21 ms de sonido, lo cual no es muy fiable para llegar a la localización del emisor. Para poder solucionar esto, se ha optado por realizar una media de 25 adquisiciones, con esto se tienen unos 500ms aproximadamente de audio, lo cual aporta más información, para llegar al objetivo de localización. Esta técnica es conocida en el procesamiento de señales como **coherencia** y se ha llegado a ella a través del **método de *Welch***. El método de *Welch* es empleado en estimación espectral y ayuda a reducir el ruido a costa de empeorar la resolución espectral, lo cual es deseable en localización. Lo que realiza es una media de las diferentes adquisiciones que se han realizado como se ve en la ecuación siguiente.

$$r_{X1X2} = \frac{\sum_{n=1}^{N} X_1^n(\omega) X_2^n(\omega)^*}{\sqrt{\sum_{n=1}^{N}(X_1^n(\omega))^2}\sqrt{\sum_{n=1}^{N}(X_2^n(\omega)^*)^2}} \tag{4.18}$$

donde **N** es el número de iteraciones que se realizan, 15 en nuestro caso, y **n** es la iteración actual.

Capítulo 5

Implementación

5.1 Introducción

En este capítulo se muestra de qué manera se ha realizado la implementación, explicando todas y cada una de las partes qué componen el programa que se ha realizado.
Para llevar a cabo esta implementación se ha empleado la herramienta de desarrollo *Keil µvision5* y, sobre todo, se ha hecho uso del módulo *CMSIS-DSP*, que es el procesador digital de señales que integra la placa, utilizando muchas de las funciones que proporciona este módulo para obtener una mayor eficiencia.

5.1.1 Entorno de desarrollo

Se ha elegido la herramienta *Keil µvision5* debido a su sencillez en la interfaz, que se mostrará a continuación, al conocimiento de la misma, ya que ha sido empleada en versiones anteriores, lo cual facilita el trabajo y, también, debido a la cantidad de ejemplos de programas que proporciona el fabricante. Un dato importante es que hay que tener en cuenta las actualizaciones que se realizan del programa. Si estas no se realizan los archivos se quedan desfasados y provocan errores a la hora de realizar la compilación. Esta herramienta también dispone del modo *debug*, el cual resulta muy útil ya que, permite simular proyectos e ir viendo los resultados de las variables, así como la ejecución paso por paso, donde se pueden detectar errores que son imperceptibles cuando se carga el programa en la placa.

En la figura 5.1 se aprecia, a la izquierda, en la ventana *project*, todas y cada una de las carpetas que organizan los archivos fuente y los ficheros de configuración. En el centro de la imagen, en la venta *Manage Run Time Environment* se observan todos los componentes software que pueden emplearse, es decir, cada uno de ellos tiene sus propias funciones, las cuales se pueden utilizar para la composición de un proyecto. En este proyecto en concreto, se han empleado básicamente las de *CMSIS-DSP* como se dijo anteriormente.

En la figura 5.2 se puede observar la ventana en la que se realiza la configuración de la tarjeta y de la interacción de la misma con la herramienta *Keil*. La pestaña seleccionada, *Device* nos permite seleccionar el modelo de tarjeta que se va a emplear. Además, en la pestaña *Target* puede seleccionarse la reserva de memoria ROM y RAM deseada, y, en la pestaña *Debug* se puede elegir el elemento encargado de realizar la depuración y carga del programa, en este caso el *ST-Link* o, si por el contrario se opta por emplear el simulador.

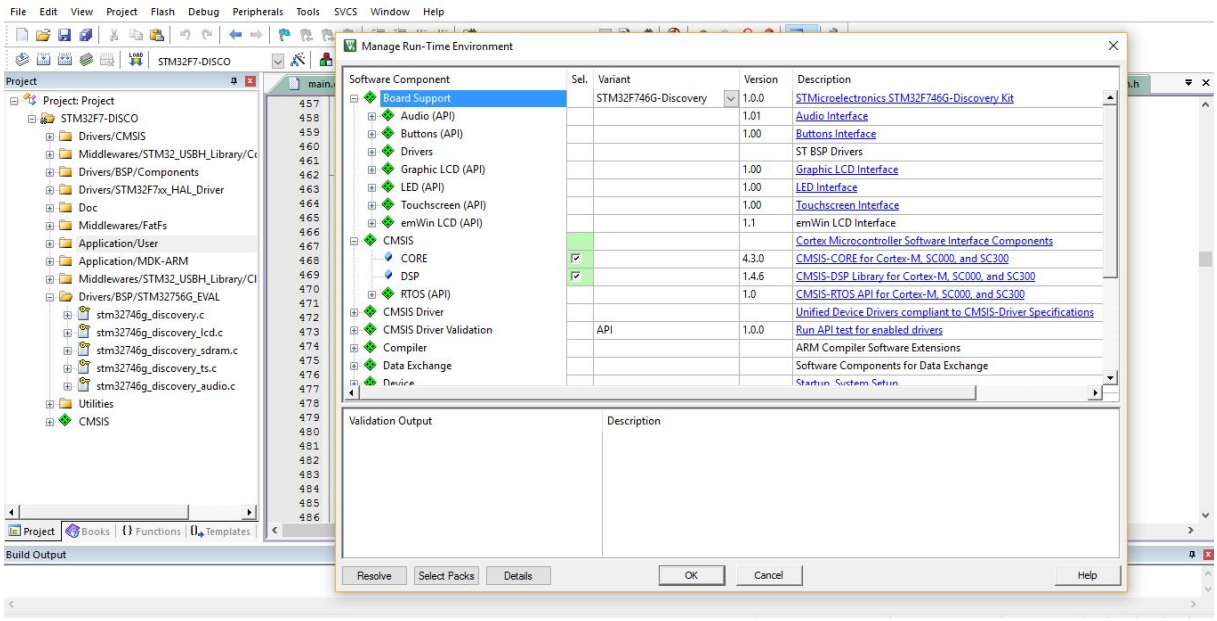


Figura 5.1: Entorno de desarrollo Keil.

Otro de los elementos importantes de esta herramienta es la ejecución paso a paso, que ya se citó antes. En la figura 5.3 puede apreciarse el botón de entrada y salida del depurador y, también un *breakpoint*. Los *breakpoints* tienen la función de parar la ejecución del programa en un punto de interés para el usuario y, de esta manera, poder evaluar cómo se está comportando el programa. Se colocan situándose con el ratón en la parte izquierda de los números que cuentan las líneas de código y seleccionando con el botón izquierdo del ratón la línea de código en la que se desea situar. En este caso se para el programa tras la realización del cálculo del ángulo de llegada del sonido y el cálculo del coseno, viéndose estos resultados abajo a la izquierda de la imagen, en la ventana *Watch* que será donde se muestren todos los resultados. Para ver los resultados de otra variable o un array, basta con arrastrar dicho elemento desde el programa principal hacia la ventana *Watch*.

5.1.2 Presentación del programa

El programa que se ha llevado a cabo es una adaptación de uno de los ejemplos que nos proporciona el propio fabricante de la tarjeta. Se trata del ejemplo *Audio playback and record* en el que se realiza la grabación y reproducción de audio. Este programa reproduce sonido introducido mediante un USB y, también, es capaz de grabar y almacenar el audio grabado en el USB. Se han realizado modificaciones de este programa para llegar al objetivo.
El programa consta de doce carpetas, como se ve en la figura 5.4, que organizan los archivos que componen el proyecto. Cabe destacar que todos los archivos son proporcionados por el fabricante, excepto los que se ubican en la carpeta de *Application/User*. Se van a comentar los que se han modificado además de los archivos que son de suma importancia en el proyecto.

- Drivers/CMSIS: se encuentra el archivo de configuración de los periféricos de la placa, que es proporcionado por el fabricante.
- *Drivers/BSP/Components*: contiene los drivers tanto del códec de audio como de la pantalla LCD.

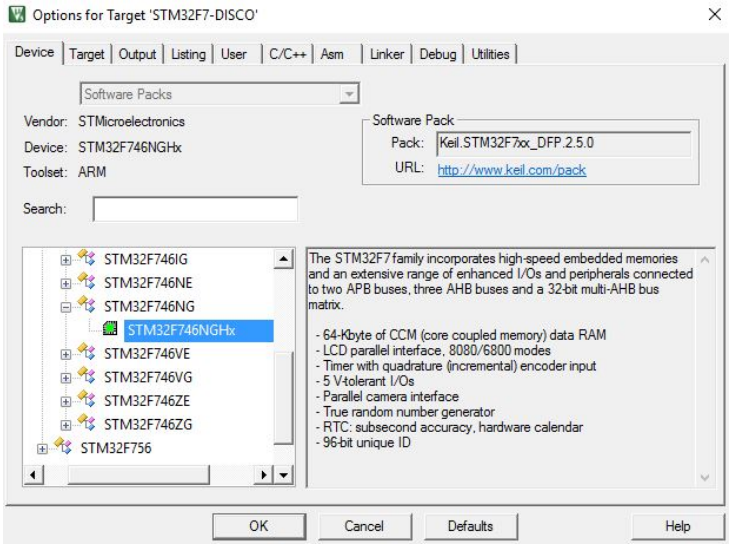


Figura 5.2: Ventana de configuración de Keil.

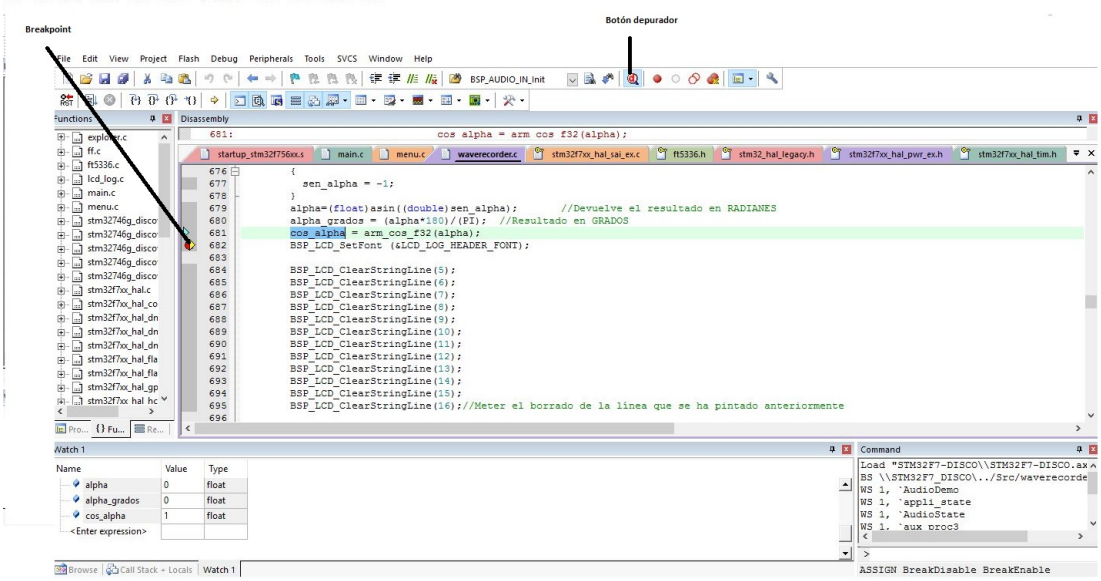


Figura 5.3: Ventana de configuración de Keil.

- *Drivers/STM32F7xxHALDriver*: aquí están las funciones que nos proporciona la inicialización y activación de elementos importantes del proyecto, como por ejemplo, el *SAI*, el I2C, las memorias, para que, se puedan emplear en los ficheros que están en la siguiente carpeta.
- *Drivers/BSP/STM32746G*: en esta carpeta se encuentran los ficheros que contienen todas las funciones que proporciona el fabricante que están basadas en las definidas en *HALDriver*, para llevar a cabo funciones de audio, de la pantalla etc. Aquí se encuentran muchas de las funciones empleadas en el proyecto.
- *Middlewares*: estas carpetas proporcionan funciones de incialización, activación y soporte del USB, y de los ficheros.
- *Application/MDKARM*: contiene el fichero *startupstm32ft56xx* en el que se encuentra el mapeo de memoria y aparecen todas las interrupciones.
- *Application/User*: como se dijo anteriormente, estos son los ficheros que contienen el grueso del proyecto y, por tanto, los que se van a comentar más a fondo.

Dentro de esta última carpeta *User* están, como se ha dicho, los archivos .c que contienen las líneas de código del proyecto y se van a presentar de manera rápida:

- *stm32f7xx-it*. Este archivo contiene el listado de las interrupciones, tanto las del propio microprocesador, como las de los periféricos, DMA por ejemplo. Este archivo no ha sido modificado.
- *usbh*. Los dos archivos de *USB* contienen todas las funciones que permiten llevar a cabo el control del USB, la inicialización, la configuración. Estos dos archivos no han sido modificados.
- *explorer*. Este archivo lo que realiza es una lectura del contenido del USB introducido para después representarlo en pantalla.
- *waverecorder*. Este archivo contiene todo el procesado realizado a la señal, así como la localización del emisor del sonido. También en él se realiza la grabación de los archivos .wav. En este archivo se encuentra la máquina de estados que se presentará a continuación y que puede apreciarse en la figura 5.6. Además de la máquina de estados, en este archivo hay varias funciones necesarias para la creación del archivo .wav, así como las *callbacks* que producen la interrupción y deciden cuando se inicia el procesado de la señal y se graba en el archivo el sonido guardado en el buffer.
- *menu*. Aquí se encuentra la primera de las máquinas de estados, la de la figura 5.5.
- *main*. Este archivo realiza todas las inicializaciones y contiene el cuerpo del programa.

Después de esta idea general y, para entender todo el funcionamiento del proyecto mejor, se van a presentar las dos máquinas de estados que se han realizado y en las siguientes subsecciones, se va a hablar de cada una de las partes que conforman el proyecto.

En primer lugar se puede apreciar en la figura 5.5 la máquina de estados principal que representa el funcionamiento de este programa. En ella se aprecian cuatro grandes estados:

- **REPOSO**. En este estado se encuentra el programa cuando se empieza ejecutar y la función que tiene es la de representar por pantalla el menú de inicio. Pasa al siguiente estado, tras representar en pantalla, siempre que no se produzca un error.

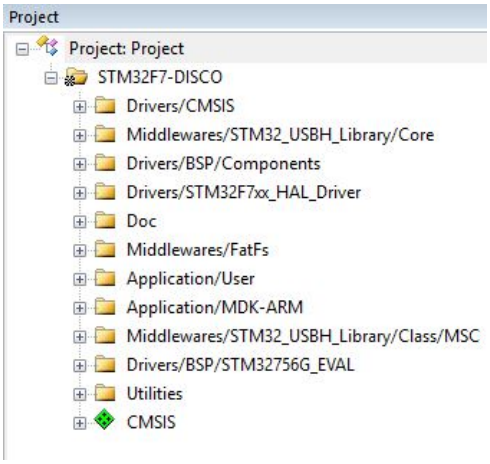


Figura 5.4: Imagen en la que se aprecian todas y cada una de las carpetas en las que está organizado el proyecto de Keil.

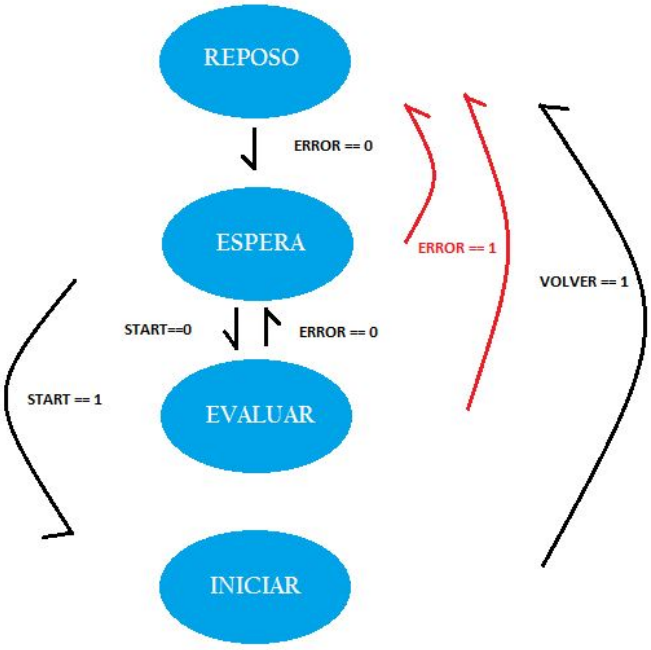


Figura 5.5: Grafo de la máquina de estados del programa.

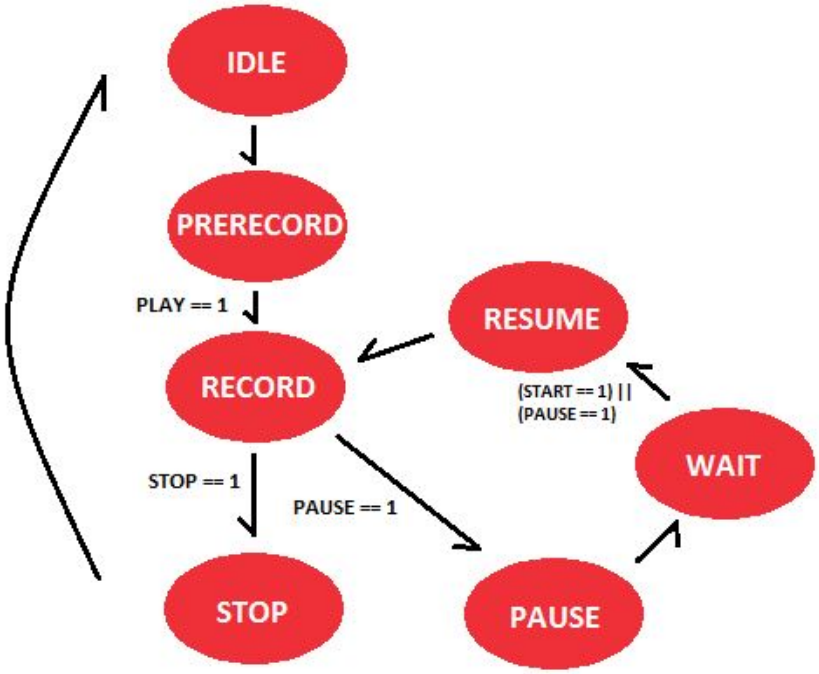


Figura 5.6: Grafo de la máquina de estados del estado INICIAR.

- **ESPERA**. Este estado es el encargado de evaluar la condición de que se pulse el botón de START que introduce al usuario en el estado INICIAR, el cual como se explicará ahora tiene también su propia máquina de estados y es donde se lleva a cabo toda la ejecución del programa. Si el botón START no es accionado, se pasa al siguiente estado, el estado EVALUAR. Siempre que se produzca un error, se vuelve al estado REPOSO.
- **EVALUAR**. Aquí lo único que se realiza es la evaluación de si se ha producido un error y, en ese caso, se vuelve al estado de REPOSO.
- **INICIAR**. Cuando se entra en este estado, se entra en otra fase, en la que se empezará a adquirir audio y a realizar el procesado, se entra en lo que realmente representa el programa.

Para terminar con esta pequeña introducción al programa, se expone la segunda máquina de estados del programa que se ve en la figura 5.6. Se entra en ella cuando se ha llegado al estado INICIAR de la máquina de estados de la figura 5.5. Se comentan brevemente los estados a continuación:

- **IDLE**. Es el estado de reposo. Al iniciarse el programa, se encuentra en este estado y pasa al siguiente estado tras accionar el botón de start.
- **PRERECORD**. Cuando se entra en este estado, cambia lo representado en pantalla y ya aparece el radar que mostrará la posición del emisor de sonido, así como los botones de start, pause y volver. Aquí además se realiza la inicialización de las funciones de audio que se comentarán en posteriores capítulos.
- **RECORD**. Siendo todos los estados importantes, este es el que destaca por encima de todos ya que, en él se realiza toda la adquisición de audio y el procesado, así como la representación de la localización.
- **STOP**. Cuando se está en el estado *RECORD* y se acciona el botón de stop, se entra en este estado, en el que se finaliza la adquisición y devuelve al usuario al estado de reposo de la figura 5.5.

- **PAUSE**. Al accionar el botón de pause, estando en el estado *RECORD*, se accede a este estado.
- **WAIT**. Tras pasar por el estado *PAUSE*, el estado siguiente es este, en el que únicamente se espera a que el usuario acciones o el botón de pause nuevamente o el botón de start, para que vuelva a reanudarse la adquisición, previo paso por el siguiente estado.
- **RESUME**. En este estado se vuelve a reanudar la adquisición de audio y se pasa al estado *RECORD* nuevamente.

5.2 Elementos empleados

Como aclaración del funcionamiento del programa, se realiza esta presentación de los elementos del lenguaje C que se emplean en el programa y son relevantes para el funcionamiento del mismo.

5.2.1 Estructuras

Aquí se presentan las estructuras presentes en el programa. Destacar que existen varias estructuras que se comentan a continuación.

5.2.1.1 AudioDemo

Forman parte de esta estructura los cuatro principales estados de la máquina de estados(*REPOSO*, *ESPERA*, *EVALUAR* e *INICIAR*), los cuales están definidos en una enumeración denominada *state*. A continuación se observan las dos líneas de código:

```
typedef struct _DemoStateMachine {
  __IO AUDIO_Demo_State state;
  __IO uint8_t select;
}AUDIO_DEMO_StateMachine;
```

5.2.1.2 BufferCtl

Esta estructura se encarga de llevar a cabo el control de buffer. En ella se han definido los siguientes elementos:

- **Pcm buff**: es el buffer que se ha definido para que se almacene en él el sonido adquirido con la función provista para ello.
- **Pcm ptr**: puntero cuyo valor es actualizado en las callbacks para indicar si el buffer está lleno a la mitad o por completo y, de esta manera, procesar la primera mitad o la segunda del mismo.
- **Wr state**: se trata de una enumeración que indica si el buffer de adquisición pcm buff se encuentra lleno o vacío. La funcionalidad que tiene es la siguiente. Cuando el buffer se encuentra lleno, se permite que se pase al procesado de la señal y, cuando se ha terminado, se vuelve a declarar el buffer vacío. Las encargadas de activar el estado lleno son las callbacks.
- **Offset**: la función que desempeña en el proyecto es de vital importancia. Se activa o desactiva en las callbakcs y, lo que realiza es decidir si se procesa la primera o la segunda mitad del buffer.

- **Fptr**: puntero que apunta a lo último que se ha grabado en el archivo .wav, para indicar por dónde debe continuarse dicha escritura. Actualiza su valor al final de cada iteración de adquisición con el valor de *byteswritten*.
- **Fptr 2**: puntero igual que el anterior pero empleado para la señal del otro micrófono.
- **Buffers**: se han definido dos buffers para el almacenamiento de las señales de ambos micrófonos por separado.

5.2.1.3 WaveFormat y FileList

Estas estructura, como su propio nombre indica, contienen todos los campos que se van a emplear para la construcción del archivo .wav.

5.2.2 Enumerados

En esta subsección se presentan los tipos enumerados, que no se han comentado y que forman parte del programa. Se empieza comentando el más importante de todos, el que contiene la segunda máquina de estados.

5.2.2.1 AudioState

En Audio State se tienen los estados de la segunda máquina. La máquina de estados que se aloja dentro del estado *INICIAR*. Como ya se vio anteriormente, aquí están los estados: *IDLE*, *PRERECORD*, *RECORD*, *STOP*, *PAUSE*, *WAIT*, y *RESUME*.

5.2.2.2 Appli State

Contiene los cuatro estados que se pueden producir en el USB, en función de si está o no conectado. Se emplean en la función USBH User Process y el programa principal podrá ejecutarse siempre que esté un dispositivo USB conectado a la tarjeta.

5.2.3 Buffers auxiliares

Para llevar a cabo todo el procesado de las señales, además de las instrucciones DSP que proporciona CMSIS-DSP, es necesaria la creación de buffers para poder operar, ya que estas funciones, como se verá más adelante, por norma general cogen los datos de un buffer y los almacenan en otro.
Las limitaciones de la tarjeta en cuanto a memoria, no permite la creación de muchos buffers y, entonces, se ha optado por la reutilización de los mismos. Se han definido los mínimos buffers necesarios con un tamaño estándar de 8192 muestras, que es el máximo de muestras que se va a emplear para realizar operaciones. Existe la excepción de los buffers *aux2* y *aux3*, que, al emplearse únicamente para realizar el método de *Welch* anteriormente comentado, se han definido con un tamaño de 4096 muestras.
También es importante destacar que, algunos buffers son float de 32 bits. mientras que otros tienen formato *q15*. El formato *q15*, emplea 16 bits, siendo el priemero el bit de signo, es decir, es lo mismo que unsigned int, a diferencia de este bit de signo. Como la adquisición de audio se hace con un buffer unsigned int, y el DSP no proporciona una función de conversión de este tipo a float, se han definido los buffers en formato *q15* y, mediante la utilización de punteros, se realiza esa conversión a *q15*, formato desde el cual el DSP si proporciona la conversión a float, con lo que se obtiene una mayor precisión al

emplear más bits.
Los buffers con formato *q15* y, que no son reutilizables son los siguiente.

- **Buff salida FIR** y **Buff salida FIR der**. Como su propio nombre indica, en estos buffers se van almacenado las muestras a las cuáles ya se les ha aplicado el filtro FIR. El resultado del filtro FIR se encuentra en estos dos buffers.
- **Buff ceros izq** y **Buff ceros der**. Estos buffers son los dos sobre los que se aplica la técnica de *zero padding*. Se crean y se rellenan de ceros, para, posteriormente, introducir en ellos el resultado del filtro FIR. Destacar que el número de muestras que tienen es el doble que el del filtro FIR.

Ahora se presentan los buffer float de 32 bits y que, a diferencia de los anteriores, sí que son reutilizables. Estos pueden reutilizarse porque en cuanto se realiza la conversión a float, ya se trabaja con este formato, por tanto, no podría emplearse un buffer que tuviese otro.

- **Aux proc 1**, **Aux proc 2** y **Aux proc 3**. Entres estos tres buffers se realiza prácticamente todo el procesado de la señal haciendo uso de las funciones proporcionadas por el DSP.
- **Aux 2** y **Aux 3**. Como ya se citó antes, estos dos buffers han sido creados única y exclusivamente para la realización del método *Welch*.

5.2.4 Punteros

Los punteros están presentes en la realización del filtro FIR, así como en el cálculo de la localización. Con los punteros **p maximo** y **p indice** se determinan el valor de la muestra de mayor valor dentro de las 7 que se tienen relevantes y el índice de dicha muestra respectivamente.

5.2.5 Variables

Son muchas las variables que se han utilizado en el programa, pero se van a comentar las más relevantes para el procesado de la señal. Estas son las siguientes:

- **Filtro**: esta variable es empleada para llevar a cabo la inicialización de la función del filtro FIR proporcionada por *STM*. Actúa como una bandera que se activa cuando se entra por primera vez en el estado *RECORD*, es decir, tras pasar por el menú principal.
- **Maximo**: determina el valor de la muestra más grande de las 7 que se tienen que son de relevancia.
- **Indice**: determina el número del índice de la muestra anterior.
- **Coherencia**: se utiliza para llevar a cabo el método Welch. Cada vez que se realiza una iteración se incrementa en una unidad, cuando su valor es de 14, se ha realizado el promediado y, por tanto, se pasa a realizar el cálculo de GCC y la representación por pantalla.
- **Tau** y **Tau max**: indican los valores de τ y de τ_{MAX}
- **Alpha**: es el ángulo (en radianes) que forma el emisor del sonido con la normal trazada entre ambos micrófonos. También se tienen como variables el sin y el cos de este ángulo, que se utilizarán para realizar la representación por pantalla.
- **Alpha grados**: almacena el valor del ángulo tras ser transformado de radianes a grados.

5.3 Inicialización del programa

En esta sección se comentan lo que se ha realizado para llevar a cabo la inicialización del programa, todos los elementos que se han inicializado y cómo para obtener un correcto funcionamiento.
Toda esta inicialización se lleva a cabo en el fichero *main.c* y, debido a que, como ya se comentó, se está reutilizando el programa para la construcción del nuevo, hay funciones que también son reutilizadas. Todas estas funciones que componen la inicialización se exponen a continuación:

- *Caché*: para realizar la inicialización de la caché, se emplea la función *CPU-CACHE-Enable*, que consta de dos funciones proporcionadas por el fabricante; una encargada de inicializar la parte de instrucciones; y la otra se encarga de los datos de la caché.

- *HAL Library*: a través de la función *HAL-Init* se realiza esta inicialización. Es una función proporcionada también por el fabricante y realiza la configuración del *Systick* para generar interrupciones, fija la prioridad para dichas interrupciones en 4, y realiza la inicialización del hardware de nivel bajo, para así poder hacer uso de las *callbacks*.

- *Reloj del sistema*: se configura a 216 MHz mediante la función *SystemClock-Config*.

- *Pantalla LCD*: también es necesario realizar la inicialización de la pantalla, para llevarla a cabo, se han seguido los pasos proporcionados en el fichero .c de la pantalla LCD. Se inicializa la pantalla mediante la función *BSP-LCD-Init*. Se aplica la configuración a la capa empleada mediante *BSP-LCD-LayerDefaultInit*, siendo en este caso ARGB8888, lo que significa RGB más la componente *alpha* que denota el grado de transparencia. Los cuatro 'ochos' indican que la resolución es de 32 bits. Se selecciona dicha capa mediante *BSP-LCDSelectLayer* y, por último se habilita el display LCD. También se emplea la función *BSP-TS-Init*, encargada de la inicialización de los drivers que habilitan la detección de posición donde se pulsa en pantalla (*touch screen*) y de comprobar si la pantalla está lista para usarse.

- *USB*: la inicialización de los drivers del *USB* son importantes porque al tratarse de un sistema que recoge audio, es importante la grabación para la realización de pruebas posteriores. Hay 3 funciones que son partícipes de dicha incialización y las tres son funciones internas del micro. En primer lugar, la función *USBH-Init* inicia las librerías para que se pueda utilizar el USB. La función *USBH-Start* se encarga de empezar el proceso del USB.

5.4 Desarrollo del programa

En esta sección se va a explicar paso a paso todo lo que realiza el programa entrando en detalles con las funciones más relevantes y, dejando para la siguiente sección la parte de la creación de la interfaz de usuario. Los archivos que abarcan todo lo que se va a tratar a continuación son "main.c", "menu.c ƶ "waverecorder.c".

Antes de entrar en profundidad, se van a mencionar dos funciones incluidas en el *main*:

- *USBH-Process*: esta función lleva a cabo todos los procesos del USB. Se encarga de la detección del USB al introducirlo, de leer todo lo que este contiene, detección de la extracción, etc. Es importante en el proyecto, ya que, si el usuario extrae el USB introducido, automáticamente, se paraliza todo el proceso del programa.

- *BSP-LED-Toggle*: esta función lo único que realiza es provocar la intermitencia del LED1 de la placa mientras el programa esté activo. Se trata de un sistema de detección de cuando el programa no está funcionando de manera correcta.

La función encargada del funcionamiento del programa es *AUDIO-MenuProcess*. En esta función se lleva a cabo la implementación de las dos máquina de estados del programa, figuras 5.5 y 5.6. La explicación va a ir centrada en el estado *INICIAR* que es cuando se realiza, tanto la adquisición de audio, como el procesado de la señal y la posterior representación en pantalla. Se puede hablar de dos funciones que acaparan todo el funcionamiento del programa de las que se va a hablar a continuación. Cuando se accede al estado *INICIAR* se entra en la función *AUDIO-REC-Start*, la cual tiene dos partes bien diferenciadas:

- Por un lado se encarga de la configuración para la grabación de audio, en este caso, se elige una frecuencia de 48 kHz como ya se comentó y se indica el uso de los dos micrófonos mediante la función *BSP-Audio-IN-Init*. Además, esta función también lleva a cabo la inicialización del códec WM8994 y la configuración del SAI, configuración en modo maestro RX, inicialización de registros necesario para el funcionamiento.
  La función *BSP-AUDIO-IN-Record* inicia la adquisición de audio y determina, el *buffer* en el que se va a guardar el audio adquirido y el tamaño de ese buffer en número de muestras. Dentro de esta función únicamente se emplea la función *HAL-SAI-Receive-DMA* como se ve a continuación, que comienza el proceso de transferencia del audio del SAI a memoria y, está también proporcionada por *STM*:

```
uint8_t  BSP_AUDIO_IN_Record(uint16_t* pbuf, uint32_t size)
{
  uint32_t ret = AUDIO_ERROR;

  /* Start the process receive DMA */
  HAL_SAI_Receive_DMA(&haudio_in_sai, (uint8_t*)pbuf, size);

  /* Return AUDIO_OK when all operations are correctly done */
  ret = AUDIO_OK;

  return ret;
}
```

  Las *callbacks* se encargan de llevar a cabo las operaciones que están definidas dentro de ellas cuando se produzca un evento. Pueden ser dos, que el buffer, esté lleno a la mitad o que esté lleno al completo, es decir, se hayan almacenado 2048 muestras. Cuando se entre en una de estas *callbacks*, se declarará el buffer como lleno y se iniciará el procesado.

- Por otro lado están los ficheros creados para el almacenamiento del audio. Se crean dos, uno para cada uno de los micrófonos con las funciones que nos proporciona el "ff.c". Además se realiza la configuración de las cabeceras y se escriben dichas cabeceras en el fichero creado. En la configuración se elige la frecuencia, así como, el número de bits por muestra y si es monocanal o multicanal. De estas funciones encargadas de la creación de los archivos .wav se hablará posteriormente en su propia subsección (4.4.6).

El código del estado *INICIAR* puede observarse a continuación:

```
case AUDIO_DEMO_INICIAR:
    if(appli_state == APPLICATION_READY)
```

```
{
  if(AudioState == AUDIO_STATE_IDLE)
  {
    /* Limpieza de pantalla */
    LCD_ClearTextZone();

    if(AUDIO_REC_Start() == AUDIO_ERROR_IO)
    {
      AUDIO_ChangeSelectMode(AUDIO_SELECT_MENU);
      AudioDemo.state = AUDIO_DEMO_REPOSO;
    }
  }
  else /* Si el estado no es el de reposo */
  {
    status = AUDIO_REC_Process();
    if((status == AUDIO_ERROR_IO) || (status == AUDIO_ERROR_EOF))
    {
      LCD_ClearTextZone();

      AUDIO_ChangeSelectMode(AUDIO_SELECT_MENU);
      AudioDemo.state = AUDIO_DEMO_REPOSO;
    }
  }
}
else
{
  AudioDemo.state = AUDIO_DEMO_ESPERA;
}
break;
```

Se aprecia que el estado *IDLE* de la máquina de estados 5.6 sirve únicamente para determinar si se va a empezar a adquirir y, de esta manera poder crear los ficheros e iniciar la adquisición, previa configuración. En la función *Audio-REC-Start* comentada antes, se fuerza al paso al estado *PRERECORD* para que ya se inicie la grabación y procesado mediante la función *AUDIO-REC-Process*.
En *Process*, es donde se encuentra la segunda máquina de estados, figura 5.6, la cual consta de siete estados, como ya se vio, y es donde se realiza prácticamente toda la ejecución del programa.

Tras el paso por la función *Start* ya se fuerza a la variable *AudioState* a ser igual al estado *PRE-RECORD*. En este estado únicamente se representa por pantalla un menú de adquisición que se explicará detalladamente cómo se ha realizado en la parte de representación por pantalla.
Cuando se acciona el botón *start* el sistema entra en el estado *RECORD*, el cual va a explicarse detenidamente debido a que es el que contiene toda la adquisición y procesado, es decir, la parte central del proyecto.
Debido a que se trata de un sistema en tiempo real, es necesario adquirir sonido mientras se está procesando la señal y detectando la localización del emisor. Para poder realizar esto de manera óptima, se ha optado por utilizar las callbacks, que van a detectar cuando se tiene información disponible para procesar en el buffer. Se tienen dos:

- *BSP-Audio-IN-TransferComplete-Callback*

- *BSP-Audio-IN-HalfTransfer-Callback*

Se definió la función *BSP-AUDIO-IN-Record* con un tamaño de buffer igual a 2048 muestras, por lo que se entrará en estas funciones cuando se tengan o 1024 o 2048 muestras. Estas funciones son funciones *weak* proporcionadas por *STM* y lo que se ha implementado en ellas es un indicador de si se tienen datos para poder empezar a procesar (*BUFFER FULL*). Otra variable implementada es *OFFSET* que indica si se tiene el buffer completo o a la mitad y será quien dictamine si se procesa la primera o la segunda parte del buffer. Este *offset* se emplea en la inicialización de los punteros que apuntan a los buffers izquierdo y derecho para esto mismo, para determinar si es la primera o la segunda mitad la que se va a procesar.

Se ha optado también por realizar la separación de las señales de ambos micrófonos dentro de estas dos funciones debido a que tener dos señales es necesario para poder obtener GCC, necesaria a su vez para llegar a la localización del emisor. A continuación, se observa una de las *callback* con el bucle *for* mencionado en su interior:

```
void BSP_AUDIO_IN_TransferComplete_CallBack(void)
{
  BufferCtl.pcm_ptr+= AUDIO_IN_PCM_BUFFER_SIZE/8;
  if(BufferCtl.pcm_ptr == AUDIO_IN_PCM_BUFFER_SIZE/8)
  {

    for(i=0;i<AUDIO_IN_PCM_BUFFER_SIZE/16;i++)
    {
      BufferCtl.buff_izq[i]=BufferCtl.pcm_buff[2*i];
      BufferCtl.buff_der[i]=BufferCtl.pcm_buff[2*i+1];
    }
    BufferCtl.wr_state    =  BUFFER_FULL;
    BufferCtl.offset   = 0;
    i=0;
  }

  if(BufferCtl.pcm_ptr >= AUDIO_IN_PCM_BUFFER_SIZE/8)
  {
    for(i=AUDIO_IN_PCM_BUFFER_SIZE/16;i<AUDIO_IN_PCM_BUFFER_SIZE/8;i++)
    {
      BufferCtl.buff_izq[i]=BufferCtl.pcm_buff[2*i];
      BufferCtl.buff_der[i]=BufferCtl.pcm_buff[2*i+1];
    }
    BufferCtl.wr_state    =  BUFFER_FULL;
    BufferCtl.offset   = AUDIO_IN_PCM_BUFFER_SIZE/16;
    BufferCtl.pcm_ptr = 0;
    i=0;
  }
}
```

La manera de separar las señales, es con el uso de un bucle *for* y, se van cogiendo muestras pares por un lado e impares por el otro. Se almacenarán por separado en distintos buffers, obteniendo, por un lado, el buffer del micrófono izquierdo y, por otro, el buffer del derecho.
Pcm.ptr determina si se tiene que llevar a cabo la separación en dos canales de las primeras 1024 muestras del buffer o de las segundas en función de si estamos en *Half Transfer* o *Transfer Complete*.

En este caso la función que se aprecia es la de transferencia completa, sin embargo, la de media transferencia es exactamente igual. No sería necesario el primer *if* en la de transferencia completa ni, el segundo en la de media transferencia pero, se han implementado ambas condiciones en las funciones por si ocurriese cualquier fallo.

También es importante el detalle de que, cada vez que se entra en una *callback* se indica que el buffer está lleno para que se proceda al procesado de la señal. De esta manera, se consigue una adquisición y procesado continuos.

Para una explicación más clara de la implementación del procesado de la señal, se separa en distintos apartados.

5.4.1 Filtro FIR

Para la aplicación del filtro FIR sobre cada una de las dos señales adquiridas por los micrófonos, se emplean dos funciones que proporciona *STM*. Destacar el uso de una bandera, denominada *filtro*, que se activa cada vez que se entre en el estado *RECORD*, previo paso por cualquier otro estado. Es importante esto porque una de las funciones es la inicialización del filtro FIR, la cual solo se emplea una vez. También citar otro dato importante, que es el uso de estas dos funciones en formato *q15*, ya que, la señal adquirida es *unsigned int* de 16 bits.
Esta función de inicialización se ve a continuación y consta de los siguientes campos:

```
arm_fir_init_q15(&S, num_coef, (q15_t *)&coeficientes_fir[0], &estados_fir[0],
 block_size);
```

- **Estructura S**: se trata de una estructura que contiene el número de coeficiente, un puntero que apunta al array que contiene los coeficientes y un puntero que apunta al array que contiene los estados, sabiendo que el número de estados debe ser igual al número de coeficientes más las muestras procesadas por llamada menos uno.
- **Número de coeficientes del filtro**.
- **Puntero** que apunta a los **coeficientes del filtro**.
- **Puntero** que apunta a los **estados del filtro**.
- **Número de muestras procesadas por llamada**.

Para el cálculo de los coeficientes del filtro FIR, se ha hecho uso de la herramienta *MATLAB*. Empleando una frecuencia de corte de 1.5 kHz, porque la señal de audio tiene la información por debajo de este frecuencia[7], y una frecuencia de muestreo de 48 kHz, que es la empleada en el programa, se obtienen los coeficientes que se observan en la figura 5.7.
Con el último término de la función de inicialización del filtro, el número de muestras procesadas por llamada, queda claro, que la implementación del filtro FIR debe llevarse a cabo en varias iteraciones, de ahí que se haya empleado el bucle *for* que se va a ver a continuación. Antes de esto, destacar que dentro del estado de *RECORD*, cada vez que se accede a el, se realiza la inicialización de todos los punteros que se emplean en el procesado al principio de sus respectivos buffers.
Ahora sí, a continuación se muestra el código bucle *for* necesario para la implementación del filtro:

```
>> Coef = round((2^15)*fir1(orden_filtro,2*Fo/fs))

Coef =

  Columns 1 through 10

        16          50          90         119          95         -29        -264        -551        -748        -666

  Columns 11 through 20

      -138         896        2332        3904        5251        6028        6028        5251        3904        2332

  Columns 21 through 30

       896        -138        -666        -748        -551        -264         -29          95         119          90

  Columns 31 through 32

        50          16
```

Figura 5.7: Imagen de Matlab en la que se aprecia cómo se ha realizado el cálculo de los coeficientes del filtro FIR.

```
for(j=0 ; j < ((AUDIO_IN_PCM_BUFFER_SIZE/8)/block_size); j++)
{
  arm_fir_q15(&S, p1+(j*block_size), buff_salida_FIR+(j*block_size), block_size);
  arm_fir_q15(&R,p2+(j*block_size),buff_salida_FIR_der+(j*block_size),block_size);
}
```

Aquí se aprecia que las iteraciones son de 32 en 32 muestras por el valor de *block-size* y que se van a realizar 32 iteraciones, que sale de dividir las 1024 muestras entre el propio *block-size*. Se utilizan 1024 muestras, por las limitaciones de la función que proporciona *STM* para la realización de la FFT, como ya se comentó en el apartado del método Welch, ubicado en la sección de localización.
En cuanto a lo que es la propia función, cuatro son los elementos que la componen:

- La **estructura** comentada anteriormente.
- Un **puntero** al **buffer de origen** de los datos.
- Un **puntero** al **buffer en el que se quieren almacenar las muestras** tras el filtrado.
- **Número de muestras procesadas por llamada**.

5.4.2 FFT

Antes de llevar a cabo la implementación de la FFT, es necesario realizar una serie de modificaciones en las señales, como ya se comentó en la sección de localización.
Una de ellas es necesaria, como es el uso de la técnica de *zero padding*. Para llevar a cabo esto, se crean dos buffers y se rellenan de ceros con las funciones que nos proporciona el fabricante para ello. Destacar que ocuparán el doble de tamaño que la señal de cada micrófono. Posteriormente, se copian cada una de las señales captadas por los micrófonos también empleando las funciones proporcionadas por *STM*.
Tras esto, para obtener una mayor resolución en los resultados, se ha optado por convertir los datos, los cuáles están en formato q15, a formato float de 32 bits.
Por último y, debido a que la función de la FFT proporcionada así lo requiere, se ha pasado del buffer de datos simple que tenemos con las muestras del audio adquirido, a un buffer de datos complejo, en el que la parte imaginaria de todas las muestras va a ser nula. Esto se ha realizado de la manera que se ve a continuación:

```
for(k=0;k<TAMANO_FFT/2;k++)      //Siendo TAMANO_FFT 4096 muestras
        {
          aux_proc3[2*k]=aux_proc1[k];
          aux_proc3[2*k+1]=0;
        }
```

El código anterior es de uno de los canales únicamente. Se aprecia que ya el tamaño es el doble, 2048 muestras, debido al *zero padding* aplicado justo antes. Tras el bucle, se va a tener un buffer de tamaño 4096 muestras, 2048 muestras reales, las pares, y 2048 imaginarias, las impares, todas ellas de valor cero como se puede observar.

Tras estas modificaciones realizadas en ambas señales, ya se puede proceder a la aplicación de la FFT, cuyas líneas de código se pueden ver aquí:

```
arm_cfft_f32(&arm_cfft_sR_f32_len2048, aux_proc3, 0, 1);
arm_cfft_f32(&arm_cfft_sR_f32_len2048, aux_proc1, 0, 1);
```

Se puede apreciar que la función empleada para ambas señales es la misma y consta de los siguientes campos:

- Un **puntero a una estructura FFT que determina el número de muestras de las que se va a realizar la transformada**. El valor es de 2048 porque la FFT toma cada muestra con parte real e imaginaria, es decir, se tienen buffers con muestras pares reales, e impares imaginarias, de 4096 muestras, pero al emplear la función de FFT, se convierten en 2048 muestras por este motivo.
- **Buffer sobre el que se realiza la transformada**.
- **Flag** que indica si la transformada es directa o inversa.
- **Bit reversal**, siempre de valor 1.

5.4.3 Promediado de la señal

Como ya se comentó en la parte de localización de este proyecto, se ha realizado un promediado de la señal empleando el método de *Welch* debido a que los resultados obtenidos son más exactos. Para implementar este método, ya que al final y al cabo se trata de un promedio de la señal resultante, se ha optado por realizar el promedio de 15 iteraciones.

$$Tiempo(ms) = \frac{8192 muestras}{48kHz} = 21{,}3ms \tag{5.1}$$

$$Tiempo(ms) = 21{,}3ms\Delta 15 = 320ms \tag{5.2}$$

Para realizar esto, se utiliza un bucle *for* como el siguiente:

```
for(p=0;p<TAMANO_FFT;p++)      //Sumatorio para realizar método Welch
        {
          aux2[p] = aux2[p] + aux_proc3[p];
          aux3[p] = aux3[p] + aux_proc1[p];
        }
```

```
for(n=0; n<numSamples; n++) {
    pDst[(2*n)+0)] = pSrc[(2*n)+0];      // real part
    pDst[(2*n)+1)] = -pSrc[(2*n)+1];     // imag part
}
```

Figura 5.8: Bucle que realiza la función del complejo conjugado proporcionada por el DSP.

Se emplean dos buffers adicionales en los que se va a ir almacenando la suma de las muestras a las que ya se les ha realizado la FFT. Además, se emplea una variable denominada coherencia y un *if* en el que la condición de entrada es que el valor de coherencia sea igual a 14. Al entrar aquí, se realiza la división entre 15, de cada uno de los dos buffers adicionales empleados para el almacenamiento de las muestras tras la FFT. Posteriormente, se realiza el cálculo de GCC, además de la representación por pantalla, como se verá a continuación. Sin embargo, cuando no se entra, porque el valor de coherencia no es el comentado, se aumenta en una unidad el valor de esta variable y se realiza la escritura de las partes correspondientes de las señales de los dos micrófonos en los ficheros que se crearon anteriormente. Además se declara el buffer de nuevo como vacío a la espera de que se llene y se vuelva a repetir el proceso. De esta manera, se consigue realizar el promedio que se ha comentado para la obtención de unos resultados más exactos.

5.4.4 GCC

Para llegar a la obtención de GCC, también se han empleado funciones exclusivas del DSP que nos proporciona la tarjeta (CMSIS-DSP).
Se sabe que la fórmula de la GCC es la de la ecuación siguiente:

$$\psi_{x1x2}(\omega) = X_1(\omega)X_2(\omega)^* \tag{5.3}$$

Por tanto, es necesario, lo primero de todo, obtener el complejo conjugado de una de las señales de los micrófonos antes de proceder a la multiplicación de ambas. Para obtener el complejo conjugado se emplea la función:

```
arm_cmplx_conj_f32(aux_proc1, aux_proc2, TAMANO_FFT/2);
```

Siendo el primer elemento un puntero que apunta al buffer que contiene los datos que se quieren convertir, el segundo elemento un puntero que apunta al buffer donde se quiere almacenar el resultado y, el tercer elemento, el número de muestras complejas en cada vector. Destacar el valor de este tercer elemento, que es igual a 1024 muestras, porque como se habla de muestras complejas, son 1024 muestras con parte real e imaginaria cada una de ellas, es decir, en realidad 2048 muestras. La definición de la función que nos proporciona CMSIS-DSP es la que se ve en la figura 5.8. Tras la obtención del complejo conjugado, es necesario realizar la multiplicación y, para ello se emplea la función siguiente:

```
arm_cmplx_mult_cmplx_f32(aux_proc3, aux_proc2, aux_proc1, TAMANO_FFT/2);
```

La función anterior realiza una multiplicación entre vectores complejos y está compuesta por cuatro elementos. Los dos primeros elementos son punteros que apuntan a los buffers que contienen los elementos a multiplicar, mientras que el tercero es un puntero que apunta al buffer en el que se quiere almacenar los datos de la muultiplicación. Por último se tiene el número de muestras complejas en cada vector.

Ya se indicó en la sección de localización que se había empleado el filtro *PHAT* y, para llevar a cabo su implementación, se han realizado dos pasos:

```
for(n=0; n<numSamples; n++) {
    pDst[n] = sqrt(pSrc[(2*n)+0]^2 + pSrc[(2*n)+1]^2);
}
```

Figura 5.9: Bucle que realiza la función del cálculo del módulo.

- El primero de ellos consiste en calcular el módulo de cada muestra compleja y almacenarla en un buffer simple. La función encargada de llevar a cabo dicho cálculo es la siguiente:

```
arm_cmplx_mag_f32(aux_proc1, aux_proc2, TAMANO_FFT/2);
```

  Donde los dos primeros elementos son punteros que apuntan a los buffers origen y destino respectivamente y, el tercer campo es el número de muestras complejas. Para entender mejor la operación realizada, se muestra en la figura 5.9.

- En el segundo paso, se realiza lo que falta para completar el filtro *PHAT*, es decir, se realiza la división de cada muestra entre el módulo que se ha calculado. Para llevar a cabo esto, se implementa un bucle *for* como el que se ve, que recorre elemento a elemento las 4096 muestras. Se realiza también una suposición en la división para que no se tome ninguna muestra como infinito, y se toman como cero los números del divisor inferiores a $1\Delta 10^{-7}$.

```
for(h=0; h<TAMANO_FFT/2; h++)
        {
          aux_proc3[(2*h)+0] = aux_proc1[(2*h)+0] / aux_proc2[h];
          aux_proc3[(2*h)+1] = aux_proc1[(2*h)+1] / aux_proc2[h];
          if((-(1e-7)) < aux_proc2[h] < 1e-7)
          {
            aux_proc3[(2*h)+0] = 0;
            aux_proc3[(2*h)+1] = 0;
          }
        }
```

Con esto que se ha comentado, se llega a obtener un único buffer de muestras complejas que es el resultado de la multiplicación de ambas señales tras aplicarles la FFT, una de ellas tras aplicarle el complejo conjugado, y tras dividir cada muestra compleja del buffer por su módulo. Para volver a tener una señal en el dominio del tiempo, es necesario aplicar la FFT inversa, con la función que ya se comentó. Cuando se aplica la FFT inversa, se tiene de nuevo un buffer simple, por tanto, la componente imaginaria, que son las muestras impares, se tienen que suprimir porque no aportan información alguna. Esto se ha conseguido implementar mediante un nuevo bucle *for* que va almacenando en un buffer únicamente las muestras pares.
Ahora ya, se tiene la información necesaria de GCC y se puede proceder al cálculo del valor τ que nos va a llevar a la obtención de la localización.

En la sección 4.4, se comentó el motivo por el cuál se tienen solo siete muestras relevantes. Como las muestras relevantes son, la del origen, tres por delante y, tres por detrás y, además, al realizar el cálculo de GCC se lleva al origen el centro, se tienen que reordenar las siete muestras necesarias.
Las que son de utilidad son, las cuatro primeras y las tres últimas, pero no de cualquier manera, sino que las tres últimas muestras se reordenarán en un nuevo buffer como las tres primeras, y, las cuatro primeras, se colocarán a continuación.

```
for(m=0;m<4;m++)
```

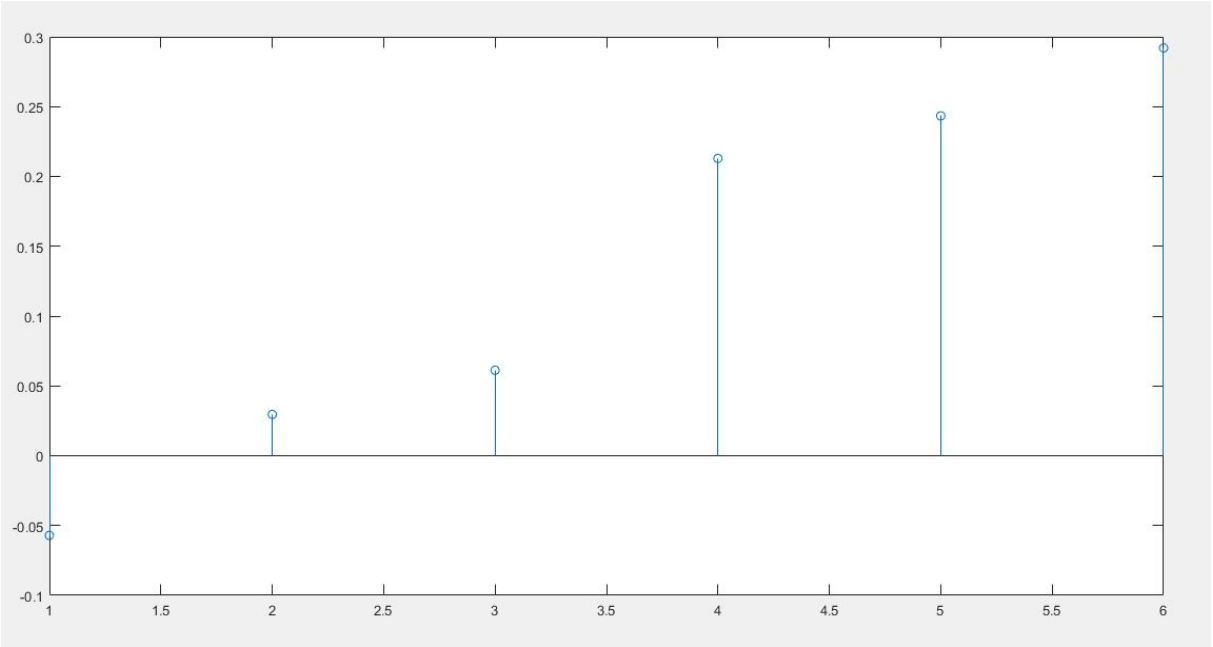


Figura 5.10: Captura de las siete muestras relevantes para determinar la localización.

```
{
  aux_proc3[m+3]=aux_proc1[m];
}
for(m=0;m<3;m++)
{
  aux_proc3[m]=aux_proc1[m+2045];
}
```

En la figura 5.10 se tiene un ejemplo real de la aplicación en el que aparecen las siete muestras relevantes que se acaban de comentar, siendo la mayor la última. Cabe destacar que en el eje X aparecen valores de 0 a 6 y, el valor de τ es el número de muestra dividido entre la frecuencia de muestreo habiéndole restado 3 antes, para determinar si la correlación es positiva o negativa., lo que se traducirá en que la localización del emisor forme un ángulo positivo o negativo con la normal trazada. Para finalizar con la implementación de GCC, es necesario determinar qué muestra es la más grande y realizar el correspondiente cálculo de τ. Esto se implementa como se aprecia en las dos siguientes líneas de código:

```
arm_max_f32(aux_proc3, 7, p_maximo, p_indice);

tau = ((float)(indice) - 3);
```

- La primera de ellas es una función que proporciona *STM* y que lo único que realiza es devolver el valor de la muestra mayor de un array y su correspondiente número de muestra dentro de ese array.
- La segunda línea de código, resta tres al número de índice en el array de la muestra de mayor valor para centrar las muestras.

5.4.5 TDOA

Para terminar con la implementación del procesado de la señal, tan solo falta obtener la localización final del emisor del sonido. La explicación de esta implementación, se basa en lo comentado en la parte

teórica de la localización.

Para obtener el sin del ángulo θ que forma el punto que se encuentra entre medias de los dos micrófonos con la normal, se realiza la operación siguiente:

$$\theta_x = \arcsin \frac{\tau}{\tau_{MAX}} \tag{5.4}$$

siendo τ_{MAX} el valor de τ cuando el emisor se encuentra formando 90º con el eje.
Como τ_{MAX} tiene un valor de 2.94 y, tenemos muestras 3 y -3, y no se pueden tener senos mas grandes en valor absoluto que la unidad, se realiza la siguiente suposición:

```
if(sen_alpha > 1)
    {
      sen_alpha = 1;
    }
    if(sen_alpha < (-1))
    {
      sen_alpha = -1;
    }
```

Con las siguientes tres líneas de código se calcula el valor del ángulo. Se realiza primero el arcsin del valor obtenido antes, cuya función devuelve el resultado en radianes. Se pasa ese resultado a grados y se calcula el valor del cos de ese ángulo, lo que se utilizará para realizar la representación por pantalla como se verá a continuación.

```
alpha=(float)asin((double)sen_alpha);
alpha_grados = (alpha*180)/(PI);
cos_alpha = arm_cos_f32(alpha);
```

Saliendo ya de la parte de procesado de la señal, hay otro estados que forman parte del programa que se deben comentar.
En cualquier momento, puede accionarse el botón *volver*, el cual nos dirige al estado *STOP*. Este es un estado de paso, es decir, el programa no se queda ahí hasta que se produzca otro evento, sino que realiza una serie de funciones e inmediatamente se vuelve a iniciar el proceso de nuevo, llevando al usuario al estado *IDLE* de la figura 5.6.
La primera función empleada en este estado es la encargada de detener la adquisición de audio y su implementación se ve en la siguiente línea:

```
BSP_AUDIO_IN_Stop(CODEC_PDWN_SW);
```

En cuanto a los dos ficheros .wav, al entrar en este estado, se actualizan las cabeceras, se escriben en ambos ficheros y se cierran. Para realizar todo esto, se emplea el archivo "ff.c"que proporciona todas las funciones.
También se inicializan con valor cero todas las variables que se emplean en los bucles *for* del estado *RECORD*.

Por último, del estado *PAUSE*, únicamente comentar que se emplea la función proporcionada por el fabricante para parar momentáneamente la adquisición y, se accede al estado *RESUME* cuando se vuelve a accionar el botón *pause* o el *play*, que reanudan la adquisición mediante la función proporcionada por *STM*.

5.4.6 Grabación de archivo .wav

Para terminar con la implementación se comenta cómo se realiza la grabación del archivo .wav. Esto es un añadido al proyecto y se ha decidido realizar porque, aprovechando que el proyecto del que se parte realiza esto, resulta interesante que el usuario pueda utilizar esta función posteriormente para realizar todo tipo de pruebas.

En primer lugar, se comentan las funciones que se emplean para llevar a cabo el control de las cabeceras de dicho archivo. Estas funciones son tres y se describen a continuación:

- *WavProcess Enc Init*. En esta función se lleva a cabo la inicialización del encoder. Se inicializan todos los campos de la estructura *WAVE FormatTypeDef* que es la encargada de almacenar información del archivo .wav. En esta función se determina la frecuencia, el número de canales, el número de bits por muestra, así como el número de bytes procesados por segundo.
- *WavProcess Header Init*. Esta función se encarga de asignar el formato correcto al fichero, es decir, de rellenar todos los campos de las cabeceras necesarios para el correcto funcionamiento del fichero. Aquí se define la frecuencia de muestreo, los bits que se procesan por muestra.
- *WavProcess Header Update*. Es la función que se encarga de ir actualizando constantemente estas cabeceras.

En cuanto a la grabación del archivo, se emplean funciones que nos proporciona el fichero "ff.c". Son funciones como *fopen*, *fwrite*, *fclose*, que llevan a cabo el control de los ficheros.

Se han creado dos archivos .wav, cada uno almacena el audio recogido por un micrófono. A continuación se detallan los pasos para la creación de dichos archivos.

1. *fopen*. Esta función se encarga de la creación de un fichero.
2. Tras la inicialización del encoder con la función detallada anteriormente *WavProcess Enc Init* y, sabiendo que se utiliza el puntero *pHeaderBuff*, se escriben estos datos de inicialización apuntados por el puntero en el archivo que se ha creado. Para llevar a cabo la escritura se utiliza la función *fwrite*.
3. Se escribe la información recogida por el micrófono en el archivo creado. Tras finalizar esta exposición de cómo crear el archivo se va a comentar detalladamente este punto.
4. Se lleva a cabo la actualización de las cabeceras mediante las funciones *fl-seek* y *WavProcess Header Update*.
5. Se escriben las cabeceras de nuevo con *fwrite*.
6. Se cierra el fichero mediante la función *fclose*.

Se va a comentar la parte en la que se utiliza la función *fwrite* para llevar a cabo la escritura del audio recogido por el micrófono, que puede verse a continuación:

```
f_write(&WavFile_der, (uint8_t*)(BufferCtl.buff_der),
                      AUDIO_IN_PCM_BUFFER_SIZE/4,
                      (void*)&byteswritten)
```

Se puede apreciar que se escribe en *WavFile der* el contenido del buffer *buff der* y que tiene un tamaño de 2048 muestras. El audio adquirido son 2048 muestras pero, como se separa en dos buffers independientes, quedan 1024 muestras para cada canal. Como los buffers están definidos como unsigned int de 16 bits, mientras, la escritura, como puede verse se realiza en tamaño byte, la función fwrite se emplea con 2048 muestras y no con 1024.

5.5 Representación y TouchScreen

En esta sección se van a comentar las funciones que realizan la representación por pantalla y las que permiten al usuario interaccionar con la tarjeta a través de la pantalla.

En primer lugar, se van a presentar las funciones que se han utilizado para realizar dicha implementación y, posteriormente, se van a presentar la representación por pantalla de cada uno de los estados que son el resultado de estas funciones.

5.5.1 Presentación de las funciones de representación por pantalla

Las funciones que se emplean en esta parte del proyecto, están todas proporcionadas por *STM* y se encuentran el el archivo .c del LCD. Solo se han creado dos funciones, las cuales se presentarán a continuación, y estas, emplean las funciones que nos proporciona el fabricante.
Primero se presentan las funciones de *STM*.

- *SetFont*. Esta función lo que realiza es determinar el tamaño de la letra. Se dispone de tres tipos de tamaños predefinidos y, en función de si lo que se va a escribir va a ser la cabecera o texto, se elige uno u otro.
  Cuando se emplea esta función, todo lo que se escriba a continuación se escribirá en el tamaño seleccionado, por lo que, cada vez que se vaya a producir una modificación en el tamaño de la letra, debe emplearse dicha función.

- *SetTextColor*. Sirve para determinar el color de lo que se va a escribir o a dibujar, ya que es empleada, tanto para texto, como para líneas o polígonos. Los colores vienen definido en el fichero .h del LCD.

- *SetBackColor*. Se emplea para determinar el fondo de lo que se va a dibujar. Se suele emplear por ejemplo para la creación de botones. Al utilizar esta función antes de la función de rellenar un polígono, se determina el color de ese relleno.

- *ClearStringLine*. Función que sirva para borrar líneas horizontales en la pantalla. El grosor de la línea lo determinará el *SetFont* actual. Se emplea en el paso de un modo a otro, para borrar lo que se tenía y dibujar el menú del nuevo modo.

- *DisplayStringAt*. Esta función es la que se emplea para representar texto por pantalla. Consta de cuatro campos, los dos primeros, las coordenadas X e Y respectivamente. El tercero es la cadena de caracteres que se quiere escribir y, el cuarto, consiste en el modo de alineamiento, es decir, si se quiere escribir centrado o en uno de los dos lados de la pantalla.

- *DrawLine*. La función de dibujar una línea realiza lo que su mismo nombre indica. Los cuatro campos de los que consta son, las coordenadas de inicio de X e Y repectivamente y, las coordenadas de X e Y finales.

- *DrawRect*. Se utiliza para dibujar rectángulos. Se introduce en los dos primeros campos las coordenadas de inicio del rectángulo y, los dos siguientes son la anchura y la altura respectivamente. La función *FillRect*, se encarga de dibujar un rectángulo relleno, por tanto, antes se tendrá que haber definido el color que se desea.

- Las funciones *FillCircle*, *DrawCircle*, *DrawEllipse* y *FillEllipse* se encargan de la realización de círculos y elipses respectivamente. Los dos primeros campos son los de las coordenadas del centro, mientras que, en el caso del círculo solo tiene un campo más, el del radio, y en el de la elipse, dispone de dos campos más, radio de X y radio de Y.

- *DrawPolygon*. Esta función sirve para dibujar polígonos. El primer campo determina un array que contiene las coordenadas de los puntos empleados, mientras que el segundo campo contiene el número de puntos del propio polígono.
  Cabe destacar que esta función en el proyecto se ha empleado únicamente para dibujar triángulos. El array comentada con las coordenadas de los tres puntos se ha denominado *PlaybackLogoPoints*.

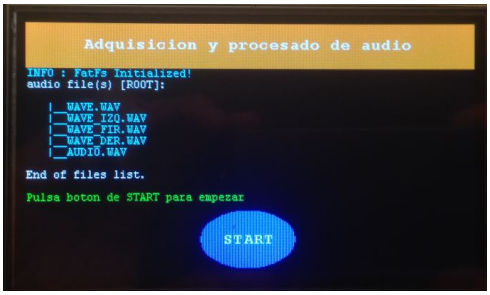
Tras esta presentación de las funciones de la representación, se va a comentar la utilidad de las dos funciones que se han creado.
En primer lugar, se presenta la función *LCD-ClearTextZone*, la cual, haciendo uso de la función que borra líneas horizontales, se encarga de realizar el borrado de toda la pantalla mediante un bucle *for*.
Por otro lado, se tiene la función *AUDIO-REC-DisplayButtons*, encargada de representar en pantalla el modo *PRERECORD*, cuando se acciona el botón de *start* pero todavía no se ha accionado el *play* y, por tanto, no se ha comenzado la adquisición.

Tras esta presentación, únicamente queda comentar cómo se ha realizado la parte de la representación de la dirección pero ésta va a explicarse junto con las imágenes de todos los estados a continuación.

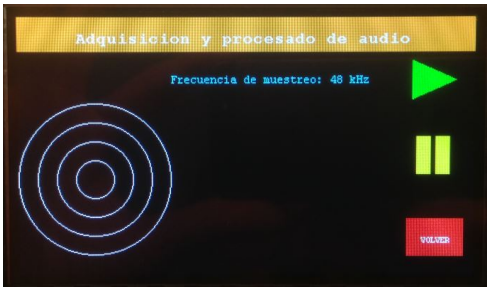
5.5.2 Representación por pantalla de los estados

En esta subsección se presentan en cuatro imágenes los estados por los que pasa el programa en la figura 5.11. En las figuras 5.11a y 5.11b se aprecia lo que se comentó anteriormente, los estados de *IDLE* y *PRERECORD*.
Cuando se acciona el botón *play* se entra en el estado, *RECORD*, el cual se aprecia en la figura 5.11c. De aquí destacar que, para destacar que se ha accionado, se dibuja sobre él una línea de color blanco. También destacar cómo se ha realizado la representación de la dirección, la cual puede verse también en la figura 5.11d. Para la representación de la localización del emisor se han empleado las siguientes líneas de código:

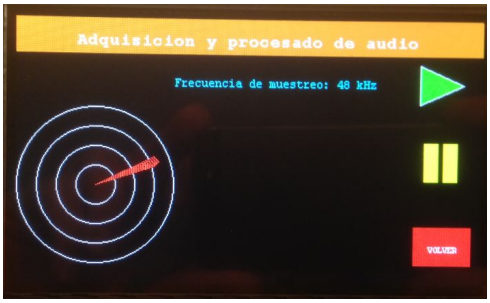
```
BSP_LCD_SetTextColor(LCD_COLOR_RED);
        BSP_LCD_DrawLine(80, 170, 75+(70*cos_alpha), 165+(70*sen_alpha));
        BSP_LCD_DrawLine(80, 170, 76+(70*cos_alpha), 166+(70*sen_alpha));
        BSP_LCD_DrawLine(80, 170, 77+(70*cos_alpha), 167+(70*sen_alpha));
        BSP_LCD_DrawLine(80, 170, 78+(70*cos_alpha), 168+(70*sen_alpha));
        BSP_LCD_DrawLine(80, 170, 79+(70*cos_alpha), 169+(70*sen_alpha));
        BSP_LCD_DrawLine(80, 170, 80+(70*cos_alpha), 170+(70*sen_alpha));
```



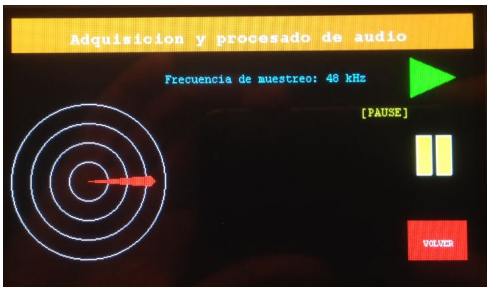
(a) Pantalla de inicio.



(b) Pantalla del estado *PRERECORD*.



(c) Pantalla del estado *RECORD*.



(d) Pantalla del estado *PAUSE*.

Figura 5.11: Comparación de los estados por los que pasa la pantalla al ir de un modo a otro.

```
BSP_LCD_DrawLine(80, 170, 79+(70*cos_alpha), 171+(70*sen_alpha));
BSP_LCD_DrawLine(80, 170, 78+(70*cos_alpha), 172+(70*sen_alpha));
BSP_LCD_DrawLine(80, 170, 77+(70*cos_alpha), 173+(70*sen_alpha));
BSP_LCD_DrawLine(80, 170, 76+(70*cos_alpha), 174+(70*sen_alpha));
BSP_LCD_DrawLine(80, 170, 75+(70*cos_alpha), 175+(70*sen_alpha));
```

Con la acumulación de 11 líneas se consigue representar una flecha que apunta al emisor del sonido. Se hace uso del sin y el cos del ángulo θ del que se habló en la parte del desarrollo del programa para realizar correctamente esta representación.

5.5.3 TouchScreen

En esta subsección se comenta cómo se realiza la detección de si un botón ha sido o no accionado para que el programa pase a otro estado. Esta detección se realiza cuando se entra en los estados *RECORD*, *PRERECORD* o *WAIT*, que son aquellos estados en los que el usuario puede interactuar.
En las siguientes líneas de código, que son del estado *RECORD* se aprecia cómo se realiza lo anteriormente comentado:

```
BSP_TS_GetState(&TS_State);
  if(TS_State.touchDetected == 1)
  {
    if ((TS_State.touchX[0] > TOUCH_STOP_XMIN) &&
     (TS_State.touchX[0] < TOUCH_STOP_XMAX) &&
        (TS_State.touchY[0] > TOUCH_STOP_YMIN) &&
         (TS_State.touchY[0] < TOUCH_STOP_YMAX))
    {
      AudioState = AUDIO_STATE_STOP;
    }
    else if ((TS_State.touchX[0] > TOUCH_PAUSE_XMIN) &&
     (TS_State.touchX[0] < TOUCH_PAUSE_XMAX) &&
              (TS_State.touchY[0] > TOUCH_PAUSE_YMIN) &&
               (TS_State.touchY[0] < TOUCH_PAUSE_YMAX))
    {
      AudioState = AUDIO_STATE_PAUSE;
    }
  }
}
```

En la primera línea de código se puede observar una función proporcionada por el fabricante en la que se devuelve la posición que se ha tocado del LCD en coordenadas. Por tanto, para detectar si se ha accionado un botón, lo único que se realiza es comprobar si ese valor devuelto por al función anterior está comprendido entre los límites de alguno de los botones y, si es así, se pasa al siguiente estado del programa.

Con esto se termina con la explicación de la implementación del proyecto.

Capítulo 6

Resultados

En este capítulo se va a hablar de los resultados que se han obtenido tras la implementación, de la que hemos hablado antes, y de cómo se ha demostrado que estos resultados son correctos.

Los resultados que se han obtenido en este proyecto son los que cabía esperar. No son unos resultados perfectos, debido a que se trata de un primer proyecto pero pueden considerarse bastante buenos. Se ha partido de un proyecto inicial que se encargaba de reproducir audio y se ha llegado a la localización de un emisor de sonido. Para llegar hasta aquí, se han realizado varias cosas con las señales obtenidas por los micrófonos y, para demostrar que estos resultados son buenos, se ha llevado a cabo la comparativa que se expone a continuación. Para llevar a cabo esta comparativa, se ha hecho uso de las herramienta *Matlab*, *Audacity*, así como de la ejecución paso a paso que proporciona Keil, la cual se explicó con detenimiento en el capítulo 5. Con la ejecución paso a paso, se han ido comparando por separados todas y cada una de las partes que componen el procesado de la señal.

Lo primero que se realiza sobre la señal es la aplicación del filtro FIR y, por ese motivo, lo primero que se muestra en este capítulo son los resultados del filtrado. En la figura 6.1 puede observarse que el filtro FIR (de paso bajo) se está aplicando perfectamente. En la parte de arriba de la imagen se aprecia el espectrograma de la señal de audio recogida por el micrófono derecho y, en la parte de abajo, se puede observar como han desaparecido todas las frecuencias superiores a la frecuencia de corte del filtro. Se ha realizado la comparación de los resultados del filtro FIR mediante la herramienta *Audacity* porque así puede apreciarse de una manera más visual que la que proporciona *Matlab*.

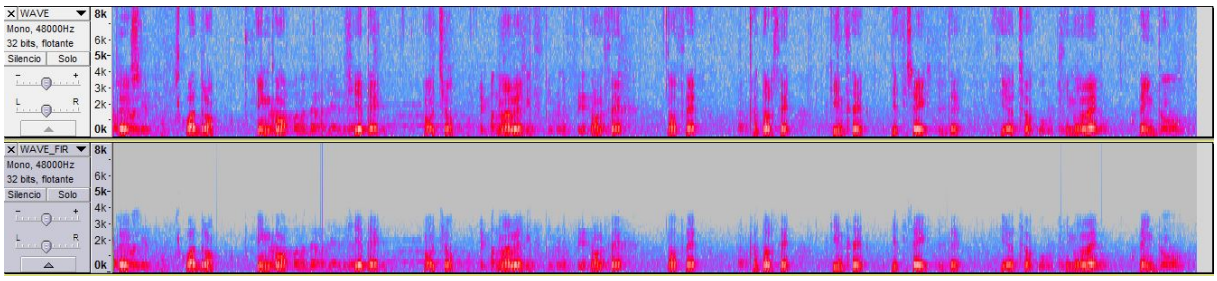


Figura 6.1: Espectrograma de la señal de audio captada por el micrófono antes y después de la aplicación del filtro FIR respectivamente.

Como se comenta en el primer párrafo, se han comparado los resultados que se obtienen en el procesado de la señal con respecto a los resultados que nos proporciona la herramienta *Matlab*, resultados cuya fiabilidad está más que demostrada. Con esto se demuestra de antemano que el procesado de la señal se está realizando de manera correcta.

Para llevar a cabo este estudio, se ha hecho uso del *Function Editor* de Keil, el cual se aprecia en la

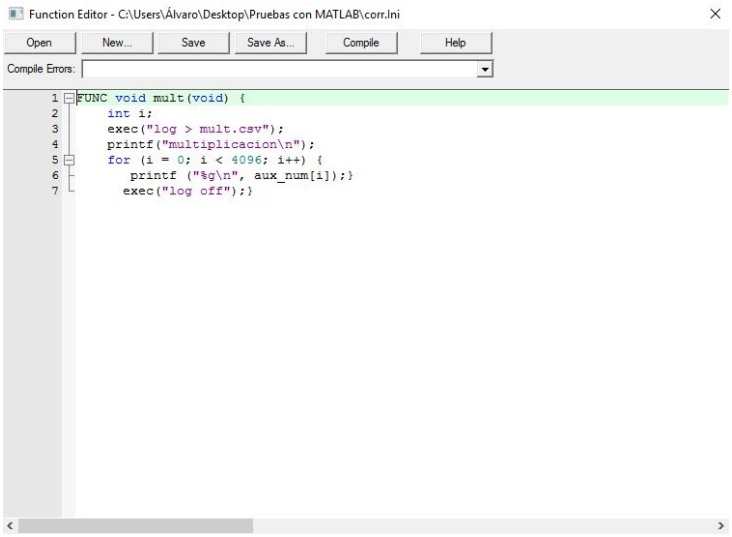


Figura 6.2: Function Editor de Keil.

figura 6.2, y que es necesario para la creación de un archivo .csv que pueda importarse a Matlab. Lo que se realiza es, llevar a cabo la comparativa desde que se realiza el filtro FIR, es decir, los datos obtenidos tras el FIR, son los que se importan a Matlab y, paralelamente, se realiza el procesado tanto en Keil como en Matlab. Tras crear la función que importa los datos, se ejecuta dicha función en la ventana *Command*, la cual puede verse en la figura 6.3.

Con los datos ya importados, pasa a realizarse el procesado por ambos lados. En primer lugar, se comparan los resultados al realizar la FFT en Keil, datos que se ven en la figura 6.4a, con los datos que se han obtenido realizándole a través del programa de Keil, los cuales se aprecian en la figura 6.4b. El código que se emplea para realizar la FFT en Matlab es el siguiente:

```
FFTizqmat=fft(bufferizq);
```

Destacar, que al realizar la FFT en Matlab, el resultado aparece en muestras complejas, sin embargo, cuando se realiza la FFT mediante Keil, se genera una muestra real y una imaginaria sucesivamente, tal y como se explica en la función interna del micro. Por tanto, para comprobar los resultados, debe compararse la muestra número 1 de Keil con la parte real de la muestra 1 de Matlab, la muestra número 2 de Keil con la parte imaginaria de la muestra 1 de Matlab y así sucesivamente. Si se comparan ambas figuras (6.4) se observa que los resultados son completamente iguales, con lo cual se verifica que

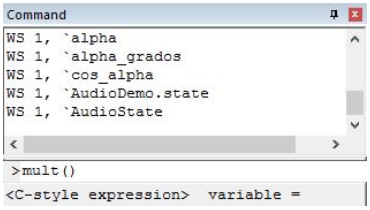
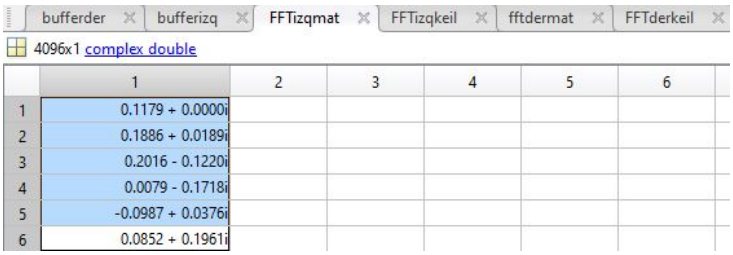
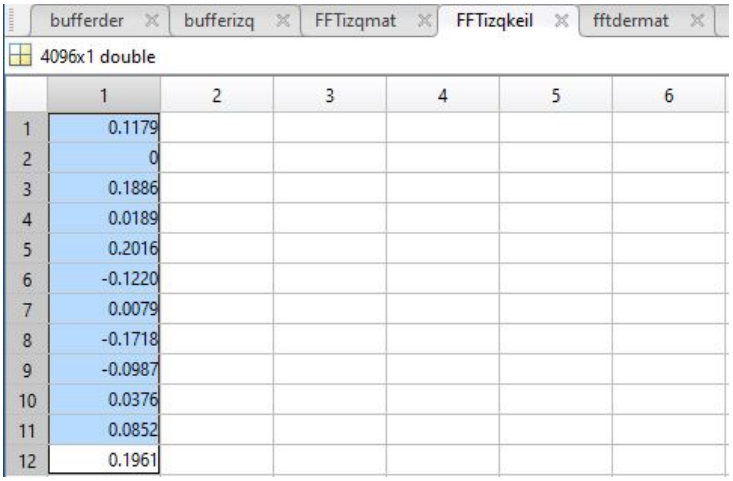


Figura 6.3: Ventana *Command* de Keil.



(a) Datos tras realizar FFT en Matlab.



(b) Datos tras realizar FFT en Keil.

Figura 6.4: Comparativa de los resultados obtenidos al realizar la FFT por vías distintas.

la FFT que realiza la función de Keil que proporciona *STM* devuelve unos resultados totalmente correctos.

Debido al promediado que se realiza para mejorar los resultados obtenidos, haciendo uso del método de *Welch* del que ya se ha hablado, para poder demostrar la veracidad de los resultados obtenidos con la herramienta Keil, se ha realizado una media ponderada de 15 adquisiciones también en Matlab y, los resultados que se han obtenidos son los de la figura 6.5.

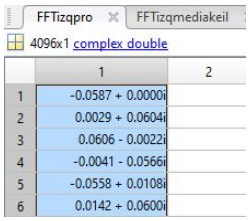
Tras realizar el promediado, se ha pasado a realizar el complejo conjugado de la señal que se obtiene por el micrófono derecho, también con ambas herramientas y, de nuevo, los resultados son iguales. Estos resultados se ven en la figura 6.6 y el código que se ha utilizado para realizar el complejo conjugado mediante Matlab es:

```
compl_conj_mat=conj(FFTderpro);
```

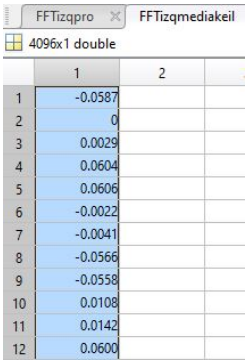
donde *FFTderpro* es el promedio de las 15 adquisiciones del micrófono derecho. El siguiente paso es demostrar que, tanto la multiplicación como la aplicación del filtro PHAT, se realizan en Keil de manera correcta. Para ello se emplea el código siguiente y los resultados se ven en las figuras 6.7 y 6.8.

```
multiplicacion_matlab = (FFTizqpro).*(compl_conj_mat);
modulo=abs(multiplicacion_matlab);
phat_matlab=(multiplicacion_matlab)./(modulo);
```

El último punto de comparación y, por supuesto, el definitivo, ya que es cuando se ha procesado toda la señal, es el de la IFFT. Si al realizar la misma, los resultados son iguales tanto en Keil como en Matlab, indica que el procesado que se realiza en Keil es bueno y nos devuelve unos resultados exactos. Para realizar la IFFT en Matlab se emplea la línea de código:

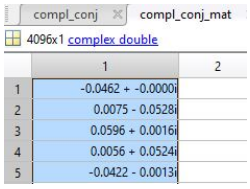


(a) Datos tras realizar promedio de 15 muestras en Matlab.

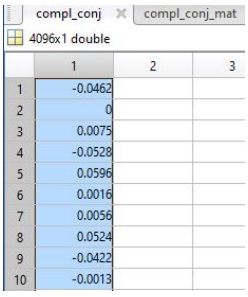


(b) Datos tras realizar promedio de 15 muestras en Keil.

Figura 6.5: Comparativa de los resultados obtenidos tras aplicar el método *Welch*.

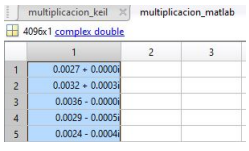


(a) Datos tras realizar complejo conjugado en Matlab.

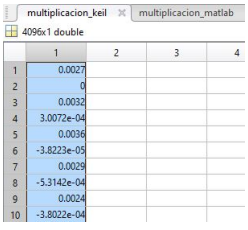


(b) Datos tras realizar complejo conjugado en Keil.

Figura 6.6: Comparativa de los resultados obtenidos tras realizar el complejo conjugado de la señal obtenida por el micrófono derecho.

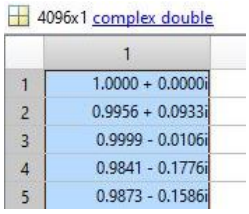


(a) Datos tras realizar multiplicación en Matlab.



(b) Datos tras realizar multiplicación en Keil.

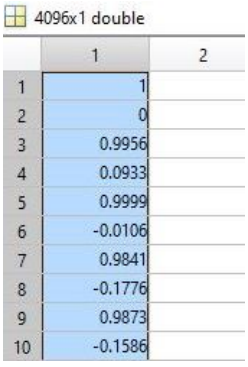
Figura 6.7: Comparativa de los resultados obtenidos tras realizar la multiplicación del espectro de la señal del micrófono izquierdo por el espectro del complejo conjugado de la señal del micrófono derecho.



4096x1 complex double

	1
1	1.0000 + 0.0000i
2	0.9956 + 0.0933i
3	0.9999 - 0.0106i
4	0.9841 - 0.1776i
5	0.9873 - 0.1586i

(a) Datos tras realizar filtro PHAT en Matlab.

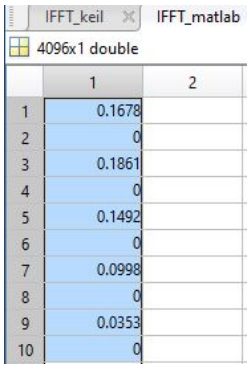


4096x1 double

	1	2
1	1	
2	0	
3	0.9956	
4	0.0933	
5	0.9999	
6	-0.0106	
7	0.9841	
8	-0.1776	
9	0.9873	
10	-0.1586	

(b) Datos tras realizar filtro PHAT en Keil.

Figura 6.8: Comparativa de los resultados obtenidos tras la aplicación del filtro PHAT.

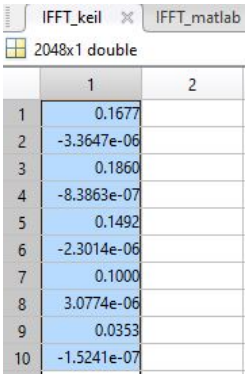


IFFT_keil | IFFT_matlab

4096x1 double

	1	2
1	0.1678	
2	0	
3	0.1861	
4	0	
5	0.1492	
6	0	
7	0.0998	
8	0	
9	0.0353	
10	0	

(a) Datos tras realizar filtro IFFT en Matlab.



IFFT_keil | IFFT_matlab

2048x1 double

	1	2
1	0.1677	
2	-3.3647e-06	
3	0.1860	
4	-8.3863e-07	
5	0.1492	
6	-2.3014e-06	
7	0.1000	
8	3.0774e-06	
9	0.0353	
10	-1.5241e-07	

(b) Datos tras realizar filtro IFFT en Keil.

Figura 6.9: Comparativa de los resultados obtenidos tras la aplicación de la IFFT.

```
IFFT_matlab=ifft(fat_matlab)
```

En la figura 6.9 se puede observar que los resultados obtenidos en ambas herramientas son prácticamente iguales.

Destacar que la función FFT de Keil ya coloca el centro de la correlación en el origen, por lo que en matlab no hay que realizar *fftshift*, función de desplazamiento del centro de la correlación, para comparar los resultados.

Para terminar con la comparativa, comentar que, aunque ya se dijo en el capítulo de la implementación, la siguiente línea del programa elimina las muestras impares para dar relevancia únicamente a las pares, las reales, que son las que llevan la información relativa a la localización.

Capítulo 7

Conclusiones y líneas futuras

En este apartado se resumen las conclusiones obtenidas y se proponen futuras líneas de investigación que se deriven del trabajo.

Tras llevar a cabo la realización del proyecto, se puede proceder a extraer las conclusiones pertinentes.
Se ha llegado al fin principal del proyecto que es la localización del emisor de sonido. Durante el desarrollo del mismo han surgido una serie de inconvenientes que se van a comentar y que se han conseguido solventar, sin embargo, hay otros que se van a proponer para que se solucionen en un futuro ya que se continuará trabajando sobre él.

Se ha conseguido realizar una buena localización a pesar de algunas de las limitaciones de la tarjeta. En primer lugar, comentar la cercanía existente entre ambos micrófonos, que ha llevado a aumentar la frecuencia que se propuso inicialmente de 16 kHz, a 48 kHz para obtener un mayor número de muestras relevantes para la detección. Esto conlleva que haya que discretizar la dirección del emisor en siete posibles ángulos y, de ahí, surge la incertidumbre cuando el emisor se encuentra entre dos de estos ángulos. Se optó por adoptar esta solución del aumento de la frecuencia ya que cuando se realizaba con 16 kHz los ángulos en los que podía estar el emisor eran únicamente tres, lo cual deslucía bastante la localización. Debido a que no se podía aumentar la distancia entre los micrófonos, se optó por este aumento de frecuencia.
También con respecto a la dirección en la que se encuentra el emisor del sonido, como ya se comentó en la sección 4.3, es destacable que existen 180º de incertidumbre, lo cual se debe a que únicamente se utilizan dos micrófonos. En la sección 4.3 se explicó el motivo de esta incertidumbre. Cuando llega el sonido, realmente llega una hiperboloide de revolución como la de la figura 4.3 en la que se aprecian los puntos del espacio que comparten la misma diferencia de tiempos de llegada. Si se dispone de varias de estas hiperboloides, se solventa dicho problema. Esto se ha llevado a cabo en otro proyecto en el que se ha implementado el protocolo de localización precisa PTP para sincronizar varias placas y, de esta manera, disponer de varios pares de micrófono y así mejorar la localización. Con este añadido de varios pares de micrófonos, se solventaría el problema de que haya que reproducir el sonido cerca de los mismos para que se aprecien los resultados claramente. En el caso de encontrarse el emisor en una habitación con varias placas dispuestas por la misma, se podría estar más separado de los micrófonos y aún así obtener la localización. Esto también es debido a que aunque la habitación esté en silencio siempre hay algo de ruido que puede distorsionar algo los resultados.
Que se haya tenido que aumentar la frecuencia de 16 a 48kHz lleva consigo un inconveniente, derivado

del propio micro, ya que la grabación en el archivo .wav no es del todo buena. Si se reduce la frecuencia a 16kHz se puede escuchar el audio grabado y es perfecto, sin embargo, al aumentar dicha frecuencia la calidad de la grabación disminuye. Además de esto, hay fragmentos del audio que no se almacenan en el archivo, esto es debido a que la función que se emplea para escribir en el archivo es bastante lenta comparada con todas las instrucciones DSP que se emplean en el proyecto. Como el proyecto consiste en la detección de la localización del emisor, queda como línea futura la mejora del audio grabado ya que es muy interesante que el audio grabado por ambos micrófonos quede almacenado y pueda ser objeto de estudio por parte del usuario. Como posible solución, puede emplearse la memoria SDRAM de la cual dispone la tarjeta para ir almacenando todos los datos del procesado de la señal y, cuando sea necesario escribir en el fichero, "volcar"toda la información y escribirla, mientras sigue procesándose la señal.
Debido a que se trata de un programa que realiza una adquisición de audio continua, se deben realizar adquisiciones y simultáneamente se debe procesar la señal. En un principio, se eligió un número de muestras igual a 8192, lo que se traduce en iteraciones de aproximadamente 170 ms. La función proporcionada por *STM* que realiza la FFT tiene la limitación de que solo puede procesar 4096 muestras. Teniendo en cuenta que se debe realizar la técnica de *zero padding* para la obtención de unos resultados óptimos y que, para realizar esta función se debe tener un buffer con parte real, se llegan a estas 4096 muestras habiendo adquirido 1024 muestras de sonido, esto se traduce en aproximadamente 21 ms.
Como con este tiempo tan pequeño los resultados que se obtienen no son fiables, se ha solucionado aplicando el método de *Welch* que se comentó en la sección 4.6.

Por último, al inicio de la realización del proyecto se esperaba que la detección de la posición se pudiese realizar y tras analizar todo el proyecto puede concluirse que los resultados son más que satisfactorios ya que se ha obtenido una notable localización.

Para finalizar, recopilando estos últimos párrafos, se proponen las siguientes mejoras:

- **Mejora de la función de FFT** que proporciona el fabricante para que el microcontrolador no limite el número de muestras a procesar.
- **Mejora de la grabación del sonido adquirido en un archivo**.
- **Evaluar de manera sistemática la localización**.

Capítulo 8

Pliego de condiciones

Es necesario disponer de una serie de elementos para el correcto funcionamiento del proyecto y, de esta manera, poder obtener la localización de la fuente emisora del sonido.

8.1 Condiciones hardware

- PC compatible con procesador de 32 o 64 bits
- 200Mb de espacio disponible en el disco duro para la ejecución del programa y guardado de datos.
- Tarjeta de desarrollo STM32F7 DISCOVERY
- Dispositivo USB con espacio libre para poder almacenar archivos de audio.
- Cable mini-USB para la conexión de la placa.
- Cable adaptador USB a microUSB para poder conectar USB a la placa.

8.2 Condiciones software

- Sistema operativo Windows 7 o superior
- Programa *Keil uVision 5* necesario para poder cargar el programa en la placa y, así poder ejecutarlo.
- Programa *Audacity* para poder evaluar las señales adquiridas.

Capítulo 9

Presupuesto

En este capítulo se realiza un estudio de los costes que conlleva la elaboración de este proyecto.

9.1 Presupuesto de equipamiento

En cuanto a este presupuesto se tienen dos partes diferenciadas, la parte hardware y la parte software. En las tablas 9.1 y 9.2 pueden apreciarse los costes del hardware y software respectivamente.

Tabla 9.1: Presupuesto de equipamiento hardware.

Concepto	**Unidades**	**Precio unidad**	**Importe total**
PC Acer Aspire V Nitro VN7-571G	1	780 €	780 €
Tarjeta de desarrollo STM32F746G-DISCOVERY	1	45.95 €	45.95 €
Coste hardware total			**825.95 €**

Tabla 9.2: Presupuesto de equipamiento software.

Concepto	**Unidades**	**Precio unidad**	**Importe total**
Windows 10	1	0 €	0 €
Licencia de Keil uVision 5	1	8260 €	8260 €
Licencia Matlab 2016 b	1	0 €	0 €
Audacity	1	0 €	0 €
Software Latex	1	0 €	0 €
Coste software total			**8260 €**

9.2 Presupuesto de mano de obra

En el presupuesto de la mano de obra se detallan las horas que se han empleado en el desarrollo del software del proyecto así como en la redacción de la memoria del TFG. Todo esto puede apreciarse en la tabla 9.3

Tabla 9.3: Presupuesto de mano de obra.

Concepto	**Unidades**	**Precio unidad**	**Importe total**
Horas desarrollo software	260	65 €	16900 €
Horas redacción memoria TFG	40	15 €	600 €
Coste mano de obra total			**17500 €**

9.3 Presupuesto total

Por lo deducido en las dos secciones anteriores e incluyendole el I.V.A. se obtiene un presupuesto total de **32169 €**.

Tabla 9.4: Presupuesto total del proyecto.

Concepto	**Importe total**
Coste equipamiento total	9085.95 €
Coste mano de obra total	17500 €
Total sin impuestos	26585.95 €
I.V.A. (21 %)	5583.05 €
Total	**32169 €**

Capítulo 10

Manual de usuario

10.1 Introducción

En este capítulo se da una idea general del funcionamiento del programa. De esta manera, el usuario puede interactuar con la tarjeta sin ningún tipo de problema.

10.2 Carga del programa

Para llevar a cabo la carga del programa en la tarjeta STM32F7, es necesario disponer del programa *Keil uVision 5*. Lo primero que debe realizarse es llevar a cabo la compilación de dicho programa seleccionando el botón que se ve en la figura 10.1. En esta figura también puede apreciarse el siguiente paso que se debe realizar, que se trata de la carga del programa en la tarjeta.

10.3 Funcionamiento del programa

Cuando el programa ya está cargado en la placa, se puede proceder a la ejecución. Importante el detalle de que hay que introducir un dispositivo USB para que se pueda ejecutar. En primer lugar, debe

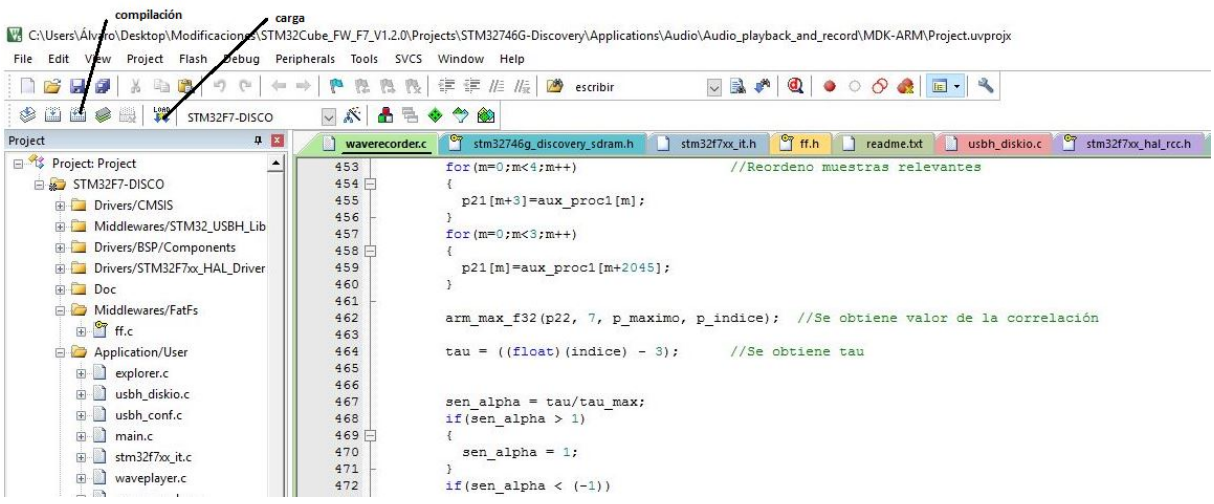


Figura 10.1: Captura de *Keil* donde se aprecia como compilar y cargar un programa en la placa.

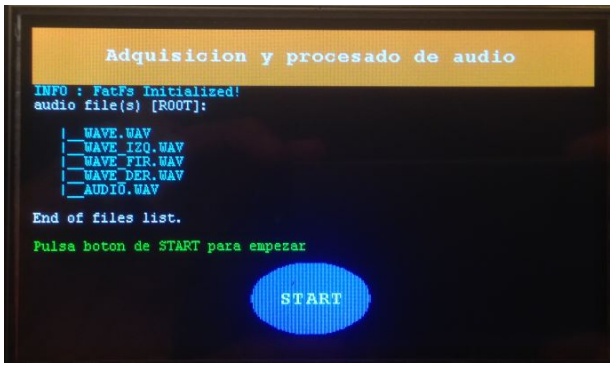


Figura 10.2: Pantalla de inicio.

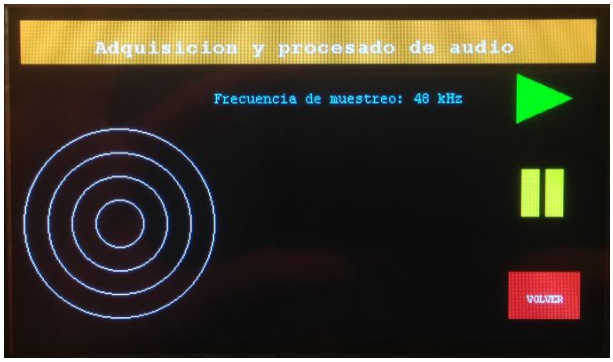


Figura 10.3: Pantalla que aparece tras accionar el botón *START*.

resetearse la tarjeta para que termine de realizarse la carga. Esto se realiza accionando el botón de reset *B2* como ya se comentó en la sección 2.6.

Tras el reset, el usuario se encuentra en la pantalla de inicio del programa. En esta pantalla (10.2) se aprecian los archivos de audio que se encuentran en el USB y el botón *START*, mediante el cual se va a la siguiente pantalla, que puede verse en la figura 10.3. En esta pantalla se observa el radar sobre el que se va a detectar al emisor del sonido, y tres botones. El botón *PLAY* que conducirá al programa a la siguiente pantalla. El botón *PAUSE*, que solo tendrá validez cuando el programa esté adquiriendo, es decir, se haya accionado el *PLAY*. Por último se tiene el botón *VOLVER* que conduce al usuario a la pantalla de inicio. El programa se mantiene en este estado hasta que se acciona *PLAY*.
Cuando esto sucede, se empieza la adquisición y la grabación del archivo y, además se muestra la dirección del emisor del sonido como puede verse en la figura 10.4.

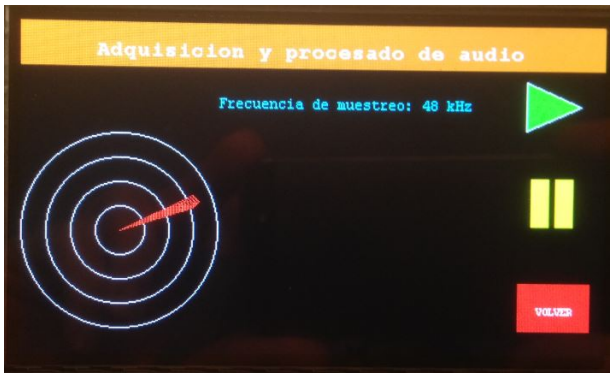


Figura 10.4: Pantalla en la que se realiza la representación de la localización.

El usuario puede detener en cualquier momento el programa y retornar a la pantalla de inicio mediante el botón *VOLVER*. Además, también existe la posibilidad de pausar y reanudar el programa empleando para ello el botón *PAUSE*.

Bibliografía

[1] M. Brandstein and D. Ward, *Microphone arrays: signal processing techniques and applications*. Springer Science & Business Media, 2001.

[2] J. Benesty, J. Chen, and Y. Huang, *Microphone array signal processing*. Springer Science & Business Media, 2008, vol. 1.

[3] M. Pollefeys and D. Nister, "Direct computation of sound and microphone locations from time-difference-of-arrival data." in *ICASSP*, 2008, pp. 2445–2448.

[4] Y. Kuang and K. Astrom, "Stratified sensor network self-calibration from tdoa measurements," in *Signal Processing Conference (EUSIPCO), 2013 Proceedings of the 21st European*. IEEE, 2013, pp. 1–5.

[5] M. J. Taghizadeh, R. Parhizkar, P. N. Garner, H. Bourlard, and A. Asaei, "Ad hoc microphone array calibration: Euclidean distance matrix completion algorithm and theoretical guarantees," *Signal Processing*, vol. 107, pp. 123–140, 2015.

[6] J. Velasco, M. Taghizadeh, A. Asaei, H. Bourlard, C. Martin-Arguedas, J. Macias-Guarasa, and D. Pizarro, "Novel GCC-PHAT model in diffuse sound field for microphone array pairwise distance based calibration," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, April 2015, pp. 2669–2673.

[7] J. Velasco, C. J. Martín-Arguedas, J. Macías-Guarasa, D. Pizarro, and M. Mazo, "Proposal and validation of an analytical generative model of SRP-PHAT power maps in reverberant scenarios," *Signal Processing*, vol. 119, pp. 209 – 228, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165168415002650

[8] C. Zhang, D. Florêncio, and Z. Zhang, "Why does phat work well in lownoise, reverberative environments?" in *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2008, pp. 2565–2568.

[9] J. Lewis, "Understanding microphone sensitivity," *Analog Dialogue*, vol. 46, no. 2, pp. 14–16, 2012.

[10] J. G. M. Proakis, S. G. Dimitris, V. Santalla del Río, A. Castro, J. Luis *et al.*, *Tratamiento digital de señales*, 1998.

[11] J. Velasco, D. Pizarro, and J. Macias-Guarasa, "Source localization with acoustic sensor arrays using generative model based fitting with sparse constraints," *Sensors*, vol. 12, no. 10, pp. 13 781–13 812, 2012.

[12] A. V. S. OPPENHEIM, W. Ronald, J. R. RONALD W SCHAFER, and T. J. P. GARCIA, *Tratamiento de señales en tiempo discreto*, 2000.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá