ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

PROYECTO FIN DE CARRERA

DESARROLLO SOFTWARE PARA INTERFAZ USB DE ADQUISICIÓN DE AUDIO

INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN
SONIDO E IMAGEN

MÁLAGA, 2007 EMILIO MOLINA MARTÍNEZ

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

DESARROLLO SOFTWARE PARA INTERFAZ USB DE ADQUISICIÓN DE AUDIO

REALIZADO POR:

Emilio Molina Martínez

DIRIGIDO POR:

Enrique Márquez Segura

DEPARTAMENTO DE: Ingeniería de Comunicaciones

TITULACIÓN: Ingeniería Técnica de Telecomunicación
Sonido e Imagen

PALABRAS CLAVE: Software para audio, C#, MATLAB Builder .NET, USB, Interpolación.

RESUMEN: *Este proyecto consiste en la creación de un software capaz de comunicarse a través del USB con una interfaz multicanal de adquisición de audio. Este objetivo se ha logrado mediante una aplicación creada en C# y basada en un entorno de pistas, capaz de registrar y reproducir audio de forma simultánea. Además, se han incluido efectos de sonido y varios sistemas de interpolación de señales en tiempo real. Por último, se ha logrado ampliar la funcionalidad, y sobre todo el potencial del proyecto mediante la integración de MATLAB con C#.*

Málaga, Septiembre de 2007

ÍNDICE

"La ingeniería es el maravilloso puente
que une lo abstracto y lo cotidiano"

Niklaus Wirth

Capítulo 1 - Introducción

Este documento es la memoria del Proyecto Fin de Carrera (PFC) “Desarrollo software para interfaz USB de adquisición de audio”, realizado por el alumno D. Emilio Molina Martínez y dirigido por D. Enrique Márquez Segura, profesor del Departamento de Ingeniería de Comunicaciones de la Escuela Técnica Superior de Informática y Telecomunicaciones (ETSIT) de la Universidad de Málaga (UMA).

El Proyecto Fin de Carrera “Desarrollo hardware para interfaz USB de adquisición de audio”, dirigido por el mismo profesor, fue otorgado al alumno D. Alejandro Márquez Raposo de forma independiente. En acuerdo mutuo, y con el consentimiento del tutor, dicho alumno y el autor de esta memoria plantearon desarrollar un proyecto de mayor envergadura de forma conjunta para conseguir un resultado de mejor calidad.

En el apartado 1.2 se explica con más detalle en qué consistió esta división de tareas y qué objetivos se marcaron para los Proyectos Fin Carrera de cada alumno. Es necesario en consecuencia establecer una clara diferencia entre el *proyecto conjunto*, que hace referencia al proyecto global abarcando software y hardware, y el *proyecto individual* de cada alumno.

A lo largo de este capítulo de introducción se pretende realizar una descripción detallada de los objetivos del proyecto, explicar brevemente la evolución desde el planteamiento inicial hasta la solución finalmente adoptada, y exponer la estructura completa de este documento.

1.1.- Definición de objetivos

Proyecto individual

El objetivo de este Proyecto Fin de Carrera es el siguiente:

La creación de un software para PC capaz de comunicarse a través del USB con una interfaz de adquisición de audio multicanal. Dicho software debe almacenar los datos provenientes de la interfaz en ficheros de audio independientes, y debe disponer de un panel de control para configurar los parámetros asociados al hardware.

Este objetivo es el requisito mínimo para dar por finalizado el proyecto. Se pueden ir incluyendo mejoras variadas, siempre y cuando se cumpla el objetivo anterior rigurosamente.

Proyecto conjunto

Como ya se dijo anteriormente, este proyecto forma parte de un proyecto de mayor envergadura que abarca software y hardware. Por tanto, es necesario también definir claramente el objetivo del proyecto conjunto. Éste es el siguiente:

La creación de un dispositivo hardware basado en microcontrolador, con su correspondiente software para PC, capaz de capturar audio a través de varios canales simultáneamente, enviar los datos por USB al PC, y almacenarlos en distintos ficheros de audio.

A continuación se describe con más detalle cómo se realizó la división de tareas entre el desarrollador hardware y el desarrollador software.

1.2.- Desarrollo del proyecto conjunto mediante trabajo en equipo

El proyecto conjunto ha sido desarrollado por un equipo de dos personas, ambos estudiantes de Ingeniería Técnica de Telecomunicación por Sonido e Imagen, y guiados por el mismo tutor. Como suele hacerse en el trabajo en equipo, cada componente se encarga de una tarea distinta para luego unificar ambos trabajos en una etapa final. En este caso, hay dos desarrollos claramente diferenciadas: el hardware, y el software. Habiendo desarrollado cada cual su sección, se unificarán en un futuro para completar el objetivo global del proyecto.

El encargado del desarrollo hardware es el responsable, como mínimo de:

- Crear la placa de circuito impreso para el desarrollo con el PIC18F4550 con ayuda de algún software de diseño de PCB's (en este caso KiCad)

- Diseñar e implementar la parte analógica para la adaptación de señales de micrófono y de línea a la entrada del conversor analógico-digital (de 0v a 5v).
- Programar el PIC para que sea capaz de muestrear los distintos canales a sus velocidades de muestreo correspondientes.
- Encargarse de enviar todos los datos por el USB de acuerdo a un protocolo establecido.

El encargado del desarrollo software es el responsable, como mínimo de:

- Recibir los datos provenientes de la tarjeta de sonido de acuerdo al protocolo establecido.
- Almacenar cada canal o conjunto de canales en un fichero de audio distinto simultáneamente.
- Crear una interfaz gráfica para Windows que permita al usuario interaccionar con el programa.
- Implementar un panel de control que permita escoger los parámetros del dispositivo.

El punto de unión entre el estudiante encargado del hardware y el encargado del software es el protocolo de comunicación. Ambos han de decidir cómo se establecerá la comunicación por USB, por lo tanto esta es una tarea totalmente compartida.

El autor de esta memoria fue el encargado del desarrollo software. No obstante, siempre se mantendrá presente la integración que requieren ambos desarrollos para lograr el objetivo global.

1.3.- Planteamiento inicial

En este apartado, se expondrá el planteamiento inicial para el proyecto conjunto que se esbozó antes de comenzar la implementación. Se verá más adelante, que a lo largo del desarrollo esta idea inicial ha ido evolucionando en función de los conocimientos adquiridos, y siempre orientada a un resultado de más calidad.

En consecuencia, la solución finalmente adoptada es ciertamente distinta a la esbozada en un principio, ya que se incluyeron algunas mejoras[1] fundamentales que cambiaron el rumbo de todo el desarrollo. Se comenzará explicando la idea inicial acerca de cómo debía ser en un principio el hardware, para continuar comentando de forma más extensa el software.

Hardware

Se pensó que el hardware debería estar dividido en dos secciones: Una sección analógica y otra sección digital. El punto de unión entre ambas es el conversor analógico-digital. Ambas secciones estarían integradas en una misma placa de circuito impreso. Esta placa además debería diseñarse con algún software específico, como en este caso KiCad (software libre).

Subsistema analógico

El subsistema analógico sería el encargado de convertir el rango de amplitudes de una señal de micrófono o de línea al de la entrada del conversor analógico-digital. El circuito estaría compuesto por operacionales, pero con un ajuste en la alimentación para adecuarlo a la del USB. Esto es así porque un operacional normalmente suele llevar una alimentación simétrica, mientras que la alimentación que proporciona el USB es asimétrica (0-5v). Para conseguir esto sería necesario añadir un offset de 2.5v a la señal.

Además, debido al muestreo posterior que se va a realizar de la señal, resulta necesario incluir un filtrado antialiasing. Inicialmente se planteó utilizar filtros paso-bajo de 4º orden, aunque más adelante se vio que la relación coste-beneficio no era suficientemente alta, por lo que se redujo el orden de los filtros a 3.

Una vez qué se dispone de una señal analógica situada en el rango de amplitudes entre 0 y 5v, entraría en juego el subsistema digital, que se describe a continuación.

Subsistema digital

El núcleo del subsistema digital sería el microcontrolador. En este caso, el microcontrolador viene impuesto en las condiciones del proyecto y es el PIC18F4550. Incluye conversor analógico digital de 13 canales, e interfaz para ser comunicado con el USB

[1] Como ya se verá en el apartado 2.3, se escogió como protocolo de comunicación la especificación estándar para dispositivos USB Audio 1.0 [8]. La elección de este protocolo es la mejora que más ha influido en el transcurso del desarrollo.

directamente. El microcontrolador estaría integrado en la placa junto al programador, para que la constante programación del mismo fuera sencilla.

Por otro lado, el firmware debería ser capaz de procesar las muestras provenientes del conversor analógico-digital (AD), incluirlas en algún tipo de tramas siguiendo un protocolo específico, y enviarlas a través del puerto USB. Este protocolo sería un protocolo personalizado y creado expresamente para este hardware y este software.

En función de la velocidad de funcionamiento del microprocesador se establecerían más o menos canales de audio. En un principio se planteó la posibilidad de implementar 4 canales. Más tarde se verá que esto es imposible debido a las limitaciones intrínsecas del PIC18F4550.

Con respecto a la calidad del audio, se planteó en todo momento audio a 8 bits, con una frecuencia de muestreo de 32Khz o 22Khz, y en estéreo. Sería una calidad de sonido similar a la que ofrecía la *Sound Blaster Pro* de *Creative*, lanzada al mercado en Mayo de 1991. No obstante, por las mismas razones que la limitación en número de canales, la frecuencia de muestreo se tuvo que reducir a 16Khz.

Por último, todo el circuito, con los conectores y los potenciómetros se integraría en una caja de plástico para darle un aspecto más compacto.

Software

Una vez que se hubo decidido que C# sería el lenguaje de programación a utilizar, y tras investigar acerca de las posibilidades del mismo, se plantearon distintos aspectos del software para comenzar.

Entorno de pistas

El entorno de pistas debería representar gráficamente los archivos de onda y permitir al usuario interaccionar con los controles de reproducción y grabación, junto con los de configuración general.

Además, desde un principio se pensó en la necesidad del uso de *ficheros de picos* (apartado 3.7 de esta memoria). No es más que un sistema de ficheros intermedios para agilizar la representación de la forma de onda de enormes ficheros WAV, evitando sobrecargas de la memoria RAM y haciendo que la representación del fichero nunca sobrepase algunas décimas de segundo.

La idea inicial se mantuvo prácticamente sin cambios hasta el final del desarrollo. Simplemente se fueron incorporando poco a poco pequeños detalles que hicieron que la interfaz fuera más intuitiva, veloz y vistosa.

Programación multitarea

El software debería estar orientado a la multitarea, definiendo una serie de hilos[2] que permitirían ejecutar todas las operaciones necesarias de manera simultánea. Durante la grabación, por ejemplo, serían necesarios los siguientes hilos:

- Un hilo que se dedicase a captar muestras del USB y almacenarlas en búferes.
- Otro hilo que fuera almacenando adecuadamente los búferes en ficheros de audio independientes.
- Otro hilo encargado de la representación gráfica en pantalla y desplazamiento del cursor.
- Otro hilo que fuera reproduciendo las pistas que no se están grabando.

Todos ellos deberían estar correctamente sincronizados mediante *semáforos*[3]. Se trataría de una estructura típica *productor-consumidor* (según [3]), y por tanto es necesario proteger los datos comunes. Para más información sobre programación multihilo, recurrir a [1] y a [3].

Este sistema fue el planteado inicialmente. Permitía que se detuviese temporalmente la grabación en el disco duro sin que se perdieran muestras (gracias a los búferes intermedios). De hecho, siguiendo este esquema se consiguió capturar señales muestreadas a 800 Hz.

En el diseño final, el esquema no varió en exceso; simplemente se simplificó y se añadió el concepto de *listas circulares de búferes*.

Comunicación USB

Puesto que la idea inicial era utilizar un protocolo personalizado, se planteó la posibilidad de usar el puerto USB como si de un puerto serie se tratara. Como punto de partida para esta tarea se recurrió al artículo técnico publicado en *www.hobbypic.com* llamado PicUSB [6]. Este artículo

[2] Los sistemas operativos multiproceso permiten el uso de hilos, hebras o *threads*, que se utilizan para ejecutar de forma simultánea varias tareas.

[3] Los semáforos son herramientas para restringir o permitir el acceso de un hilo a un cierto recurso.

incluye un ejemplo para C# en el que comunica el PIC18F4550 con el PC por USB como si de un puerto serie se tratara. La idea sería utilizar los recursos que aporta este artículo para transmitir tramas de información siguiendo un protocolo propio.

Este sistema de hecho estuvo funcionando correctamente, pero tenía una limitación en velocidad bastante importante y por ello en el diseño final no se utilizó. En el apartado 2.3 se explica con más detalle cómo fue evolucionando el protocolo de comunicación y cual fue finalmente la solución adoptada.

1.4.- Descripción simplificada de la solución final

Debido a la gran carga de búsqueda y síntesis de información que conlleva este proyecto, resulta muy difícil prever todos los imprevistos que puedan surgir a lo largo del desarrollo antes de comenzarlo. Por ello, se fue buscando un compromiso constante entre trabajar según lo planificado e innovar para encontrar mejores soluciones.

La solución final, pues, es el resultado de una planificación inicial, y de una serie de investigaciones realizadas a lo largo del desarrollo. En este apartado se va a describir muy brevemente en qué consiste la solución final. El resto de detalles se irán desvelando a lo largo de todo el documento.

Hardware

Se desarrolló por tanto un sistema con una sección analógica y otra sección digital basada en el PIC18F4550. La sección analógica consta de 2 canales seleccionables entre entrada de línea o entrada de micrófono. Esta selección se realiza a través de un conmutador, que activa un selector distinto en función del tipo de entrada que sea.

Una vez que se ha realizado esta selección de canales, los datos se procesan para ser adaptados en ancho de banda y amplitud al conversor analógico-digital integrados en el PIC. Esta operación analógica se realiza mediante amplificadores operacionales, y consiste en un filtro antialiasing y un corrector de amplitud.

Los datos digitalizados se procesan en la sección digital para ser enviados posteriormente a través del USB. Tras una correcta organización de los mismos, se envían siguiendo el protocolo estándar USB Audio 1.0 [8] a través del USB.

Gracias al uso de este protocolo, cualquier software de procesado de sonido podrá controlar correctamente el software, ya que existe un driver estándar para este protocolo en cada plataforma.

Software

Se ha desarrollado un entorno de pistas basado en formularios como interfaz de usuario. Este entorno se ha desarrollado en Microsoft Visual C# Express, lenguaje perteneciente a la Plataforma .NET (ver apartado 3.3).

Cada pista se asocia con un fichero WAV temporal, que contiene datos de audio estéreo o mono. Esta pista contiene los componentes necesarios para representar gráficamente la forma de onda del fichero en distintos zooms. Estas pistas, además, están integradas en un *panel de pistas* que va a permitir agregarlas o eliminarlas fácilmente. Este panel no tiene un número de máximo de pistas, por lo que se podrán agregar pistas de forma indefinida hasta que el PC no disponga de recursos suficientes para gestionar a todas.

El panel de pistas está directamente relacionado con la reproducción y la grabación de audio, ya que contiene la clase[4] encargada de ambas tareas: `CMotorDeAudio`. Esta clase, a través de clases auxiliares más sencillas, permite la reproducción y la grabación de las pistas del panel. En la Figura 1.1 se muestra una captura de pantalla del programa funcionando.

[4] Una clase en programación es un contenedor de uno o más datos (variables o propiedades miembro), y de las operaciones necesarias para operar con dichos datos (funciones/métodos).

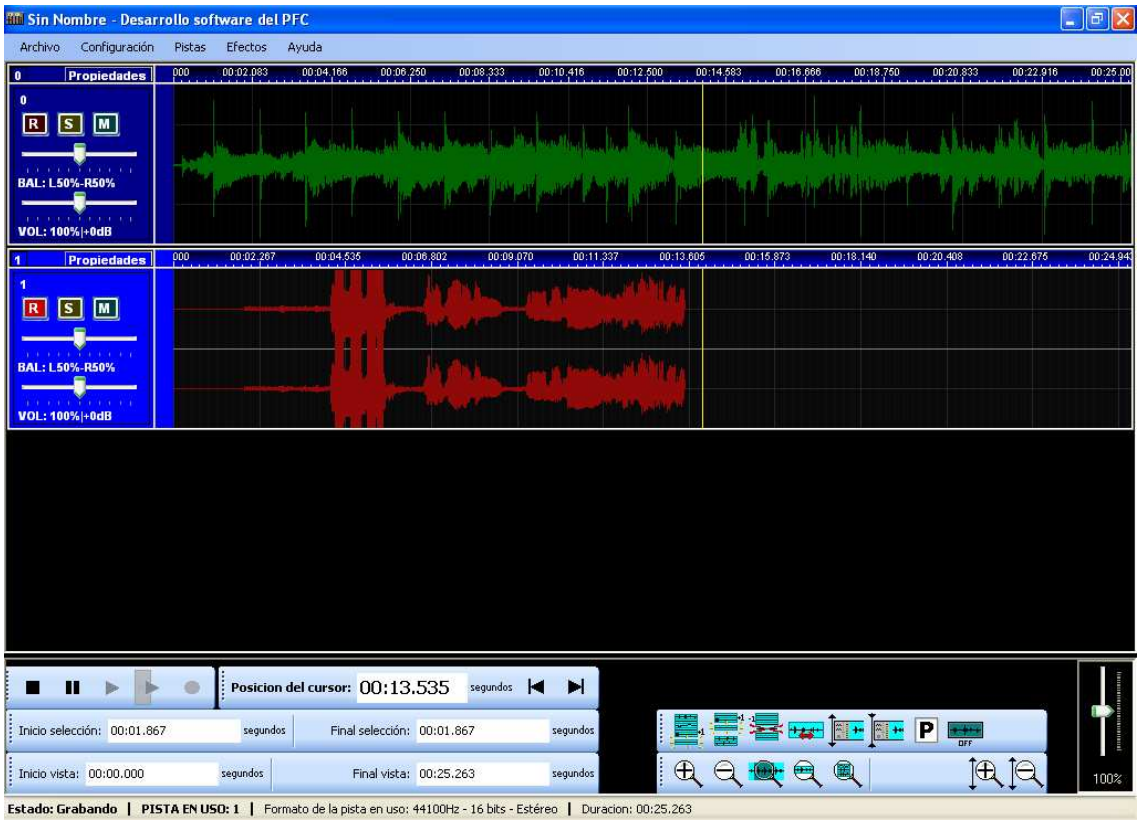

Figura 1.1.- Captura de pantalla de la aplicación en funcionamiento

Si se profundiza un poco más en el flujo de audio, se verá que para reproducir y grabar audio se ha hecho uso de la API[5] para gestión multimedia de Win32: winmm.dll. Con las funciones de esta API se puede controlar correctamente el hardware desarrollado, gracias al uso de la especificación estándar USB Audio 1.0.

Se ha desarrollado además un gestor de sesiones para permitir guardar y cargar el estado de las pistas y la configuración de las mismas en un único fichero. Este gestor almacena la sesión en un fichero con extensión .PFC, extensión que hace referencia al Proyecto Fin de Carrera.

Además, se han incluido varios sistemas de interpolación a tiempo real, que permiten la reproducción de pistas con distintas frecuencias de muestreo simultáneamente. Estos sistemas pueden ofrecer mayor o menor calidad de interpolación en función de la velocidad de procesamiento del PC en cuestión.

[5] Las siglas API significan Interfaz para Programación de Aplicaciones (*Application Programming Interface)*, y es un conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece una biblioteca para ser utilizado por otro software como capa de abstracción.

Por último, se ha integrado un sistema de efectos que permite aplicar efectos a pistas a tiempo real *(online)* o manipulando el fichero WAV asociado de forma independiente (*offline)*.

1.5.- Estructura del documento

En este apartado se van a resumir uno a uno los capítulos de la memoria para una dar una vista global de todo el documento.

Capitulo 1 – Introducción

En el capítulo de introducción se ha contextualizado el proyecto, establecido sus objetivos básicos y descrito brevemente la solución adoptada para cumplirlos.

Además se deja clara la diferencia entre los objetivos globales del proyecto común y los objetivos concretos de este PFC (desarrollo software).

Capítulo 2 – Desarrollo hardware

En este capítulo se pretende realizar un breve resumen del desarrollo hardware, siempre orientado, por supuesto, al desarrollo software. Además, en este capítulo se habla acerca del protocolo de comunicación y qué pasos fueron necesarios para llegar a adoptar la especificación USB estándar.

Capítulo 3 – Desarrollo software

Este capítulo se puede considerar el verdadero núcleo de toda la memoria. En él se describe con detalle todo el proceso de creación del software y el funcionamiento interno de la aplicación. Este capítulo dispone de los siguientes apartados:

Análisis de requisitos

Se realiza un análisis de las funcionalidades, opciones, y demás posibilidades que va a necesitar un usuario convencional para poder usar el software de una forma intuitiva y efectiva.

Elección del lenguaje de programación

Se hace un estudio de todos los lenguajes de programación que fueron susceptibles de utilizarse para el desarrollo. Finalmente se explica por qué C# fue el lenguaje utilizado.

Introducción a la Plataforma .NET

En este apartado se explican los conceptos básicos que componen la Plataforma .NET, y qué implicaciones tiene su uso en el funcionamiento de la aplicación.

Elección de la API adecuada para audio

En este capítulo describen las distintas APIs capaces de comunicarse con los dispositivos de audio, y por qué se escogió la API contenida en winmm.dll. Se explica además cómo usar estas funciones para conseguir una correcta comunicación con los dispositivos.

Integración de winmm.dll en C#

Se explica cómo integrar funciones nativas e importadas de cualquier DLL[6] en C#. Para ello se hace uso de `DLLImport`. Además, se explica el caso concreto de la importación de las funciones y tipos pertenecientes a winmm.dll.

Descripción del sistema de reproducción y grabación

Habiendo descrito todas las funciones que van a permitir la reproducción y grabación de audio, se describen las clases que van a permitir la sencilla utilización de todas ellas. Además en este apartado se explica con detenimiento el sistema de búferes que van a permitir una comunicación fluida con los dispositivos de audio.

Clases creadas para el uso de ficheros

Se describen las diferentes clases que permiten la interacción del software con el disco mediante ficheros de distintos formatos. Se explica cómo se accede a los 4 tipos de ficheros que controla este software: .WAV, .MP3, .PIK y .PFC.

Componentes y clases creadas para generar el entorno basado en pistas

Aquí se describe todo lo relacionado con la creación e interacción de los componentes de usuario que van a permitir un entorno basado en pistas.

[6] Los ficheros DLL (Dynamic Linking Library) son librerías de funciones, clases, tipos, etc. Tienen la característica de que se puede recurrir a ellos en tiempo de ejecución, diferenciándose en esto de las librerías estáticas (.LIB).

Mezclador y distribuidor

En este apartado se describe un punto muy importante del software. Se explica el funcionamiento de la clase encargada de mezclar todas las pistas para su reproducción y de distribuir el audio entrante de los dispositivos a cada pista.

Además, todo lo relacionado con la interpolación de la señal en tiempo real también se explica en este apartado de la memoria.

Efectos de sonido e inclusión de rutinas MATLAB en C#

Se explica cómo se implementan los efectos de sonido, para su aplicación en tiempo real o procesando el fichero WAV de forma independiente. Además, se hace un breve resumen acerca de MATLAB Builder .NET, y cómo se ha realizado su integración con C# para poder integrar rutinas MATLAB en la aplicación.

Diagrama esquemático de la aplicación

En este apartado se pretende dar una vista global en forma de diagrama esquemático de las distintas clases y componentes. Para ello se han representado casi todas las clases de la aplicación, y mediante conexiones se ha indicado la relación que existe entre ellas.

Descripción de la solución .NET en Microsoft Visual C# 2005 Express.

Se hace una descripción de la solución .NET sobre la que se ha desarrollado toda la aplicación. Se comentan los proyectos de los que consta la solución, las DLL externas necesarias y los requisitos para poder compilar correctamente la aplicación con Visual C# Express.

Capítulo 4 – Conclusiones

En este capítulo se realiza una valoración general del proyecto, enfatizando las características clave del mismo. Además, se comentan las dificultades encontradas a lo largo del desarrollo y se profundiza en los aspectos que han requerido una investigación detenida para su elaboración. Por último se proponen líneas futuras de trabajo por si se decidiera realizar una segunda versión mejorada de este Proyecto Fin de Carrera.

Capítulo 2 - Desarrollo hardware

Puesto que este Proyecto Fin de Carrera individual tiene como objetivo el desarrollo software del proyecto global, no es conveniente profundizar en exceso en la descripción del hardware. Para ello se debería recurrir al proyecto de D. Alejandro Márquez Raposo, que trata con mucha más profundidad el desarrollo hardware.

Por tanto, esta descripción del hardware siempre irá enfocada hacia la creación del software y la repercusión que tiene sobre el mismo.

2.1.- Utilidad de conocer el hardware para desarrollar el software

Dado que el software y el hardware realmente deben formar un proyecto conjunto, es necesario para su integración el conocimiento de ambas partes.

Dependiendo de la funcionalidad del software, será necesario conocer con un nivel de detalle mayor o menor el hardware. Como este software no necesitará controlar a “bajo nivel” los parámetros del hardware, basta con un conocimiento general de su estructura. Sería un caso distinto, por ejemplo, si fuera necesario modificar desde el software parámetros internos del hardware, tales como el tamaño del buffer de salida o la frecuencia de reloj de funcionamiento.

En el caso concreto de esta aplicación, es necesario conocer el número de canales de los que dispone la tarjeta de sonido y la frecuencia de muestreo máxima capaz de proporcionar. Como algo opcional sería interesante también conocer la funcionalidad de los canales (micrófono o entrada de línea), así como sus posibilidades de control (volumen, balance, etc.).

Por otro lado, aunque no se ha mencionado, resulta obvia la necesidad de conocer el protocolo de comunicación que se va a usar para poder establecer una comunicación adecuada con la tarjeta de sonido.

2.2.- Esquema general del hardware

El hardware va a subdividirse en dos subsistemas bien diferenciados: la sección analógica y la sección digital (ver Figura 2.1). Aunque realmente existe una gran relación entre ambas, el desarrollo de una es bastante independiente de la otra.

Para seguir en cierto modo el recorrido de la señal desde que se emite hasta que se registra en el PC, se comenzará hablando de la sección analógica, para continuar con el tratado digital de la señal en el hardware, y terminar en el procesado que realizará el PC.

Descripción del subsistema analógico

La sección analógica del hardware es la encargada de adaptar la amplitud y el ancho de banda de las señales analógicas provenientes de una fuente de audio (como un micrófono, o un preamplificador) a los niveles requeridos a la entrada del conversor analógico-digital. Este subsistema analógico consta de dos canales de entrada distintos, seleccionables entre entrada de micrófono y entrada de línea.

Descripción del subsistema digital

Como ya se comentó en el capítulo anterior, el núcleo de todo el subsistema digital va a ser el microcontrolador, en este caso el PIC18F4550. El objetivo principal de la parte digital va a ser la conversión a datos digitales de las señales analógicas, el procesado interno para la organización de los mismos, y la emisión hacia el PC por el puerto USB siguiendo un protocolo establecido (se comentará el uso de la especificación USB Audio 1.0 en el siguiente subapartado).

Debido a las limitaciones del PIC18F4550, a la hora de muestrear las señales analógicas, se trabajará con una frecuencia de muestreo de 16Khz, y una precisión de 8 bits por muestra. Conocer estos dos datos es imprescindible para poder diseñar el firmware y el software.

Tras procesar las señales, se organizarán las muestras según el protocolo establecido y se enviarán través del puerto USB en dirección al PC. Aquí acaba el camino de la señal a través del hardware. Realmente, no es necesario conocer con más detalle el funcionamiento interno del procesado digital, ya que lo único que importa es el protocolo con el que se vayan a organizar los datos a la salida. Por tanto, el siguiente paso lógico es explicar qué protocolo se ha usado para enviar los datos a través del USB.

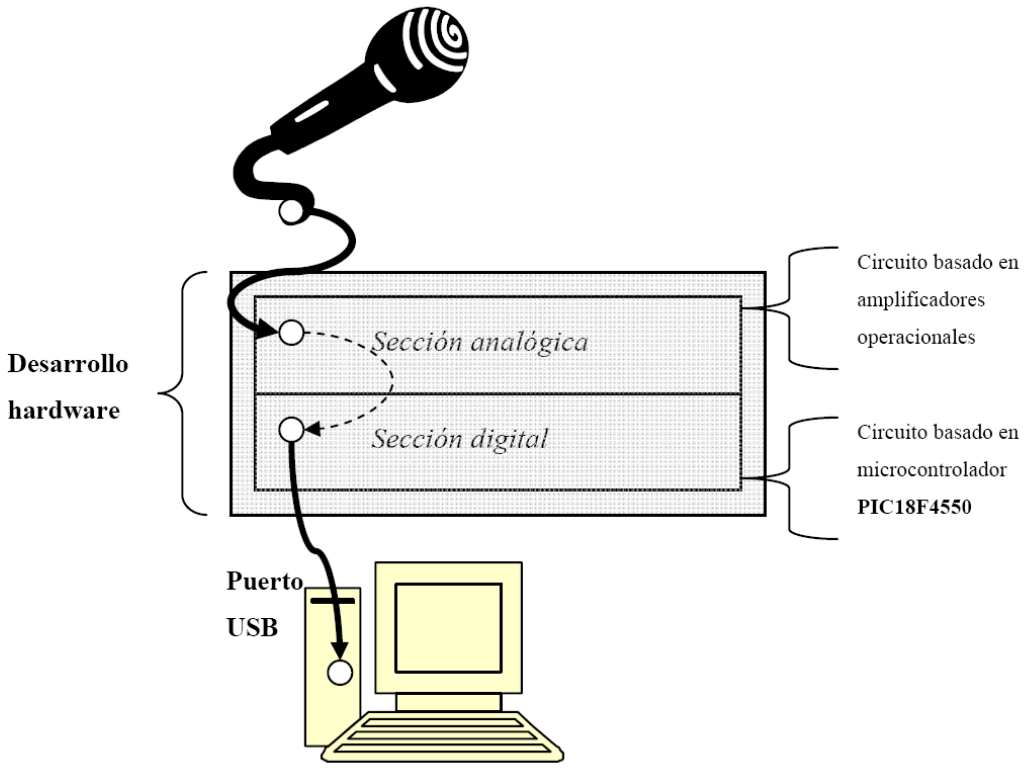

Figura 2.1. – Esquema general del hardware

2.3.- Interfaz proporcionada por el hardware al software

La elección del protocolo de comunicación es quizá el tema más complejo y que más ha evolucionado a lo largo de todo del desarrollo. Ésta es realmente la decisión más importante que se ha tomado, ya que en torno ha ella ha girado todo lo demás.

Evolución de la idea del protocolo de comunicación

Para comprender mejor el contexto en el que se tomó esta decisión tan importante para el desarrollo, conviene explicar cual fue el punto de partida, cual fue la elección final, y por qué se decidió que esta era la mejor opción posible.

Esbozo inicial del posible protocolo de comunicación

Inicialmente, no se tenía una idea clara sobre cómo debía ser el protocolo de comunicación. Por analogía con el resto de protocolos de comunicaciones digitales, se sabía que

los datos debían ir encapsulados en tramas, y que la tarea del PC debía ser desencapsular estos datos para poder procesarlos de forma independiente.

La primera comunicación que se logró entablar con éxito entre el PIC18F4550 y el PC fue gracias a un artículo encontrado en *www.hobbypic.com* llamado PicUSB [6]. En este artículo se describía una forma sencilla de comunicar el PC y el PIC. Para ello utilizaba un driver proporcionado por Microchip, junto con una API que facilitaba su utilización. Fue un excelente punto de partida, ya que abstraía al desarrollador de los mecanismos a bajo nivel que posibilitaban esta comunicación.

En un principio se pensó usar el sistema descrito en este artículo para realizar la comunicación de audio. Parecía buena idea debido a su sencillez en el firmware, a la capacidad para enviar datos binarios en tramas de hasta 1024 bytes, y a la facilidad para captar las muestras en el PC desde C# (ya que se usaban clases ya implementadas por Microchip). Sin embargo existía un problema que más tarde haría necesario descartarlo: la velocidad de transmisión no era suficiente para los requisitos del sistema. Se logró muestrear audio a 800 muestras por segundo y enviarlo con éxito usando este sistema, pero más tarde se llegó a la conclusión de que de que existía una limitación de velocidad implícita en este sistema de comunicación. Por tanto, comenzó una labor de investigación para intentar solucionar este problema. Pronto se llegó a otra posible solución: la implementación a bajo nivel de un driver propio.

Posibilidad de desarrollo de driver propio

Se planteó la posibilidad de desarrollar a bajo nivel un driver propio que permitiera una comunicación aprovechando toda la velocidad del USB 2.0. Sin embargo planteaba varios inconvenientes importantes.

El primer inconveniente es la complejidad que acarrea el desarrollo de un driver propio, ya que requiere una programación en C a muy bajo nivel y hacía prever que llevaría demasiado tiempo su desarrollo.

Además, como se pretendía que Windows fuera capaz de reconocer el dispositivo USB como dispositivo de sonido, se complicaba aún más la tarea, ya que para conseguir esto habría que adaptarse al modelo genérico de drivers de Windows WDM (Windows Driver Model, ver [10]). Si esto no se hacía, el hardware sólo podría ser utilizado desde el software destinado específicamente para ello, a través de una especie de API específica que sería necesario desarrollar.

Aunque esta posibilidad estuvo presente durante gran parte del desarrollo, pronto se encontró la solución que más se adaptaba a lo buscado: adoptar el protocolo estándar ya establecido para dispositivos USB de audio.

Especificación estándar para USB Audio Devices

Se obtuvo de *www.usb.org* [8] el documento clave para esta nueva posibilidad: *the Universal Serial Bus Device Class Definition for Audio Devices 1.0.*

Este documento describe cómo debe organizar y enviar los datos el dispositivo USB de audio que quiera acogerse al protocolo estándar; es decir, el protocolo de comunicación. En él se describe con todo detalle la organización mediante diagrama de bloques del dispositivo, el envío de las tramas, etc.

La gran ventaja de este sistema es que permite la utilización de drivers ya creados y probados por otros desarrolladores. Además se permite que cualquier software de audio capaz de controlar cualquier tarjeta de sonido tenga la posibilidad de utilizar el dispositivo USB desarrollado.

Esta opción es la más adecuada con diferencia para el desarrollo, ya que facilita la programación en el PC, permite el uso del hardware desde programas de audio distintos, y ofrece un protocolo de comunicación optimizado para audio por USB. Por tanto, esta fue finalmente la decisión escogida.

Uso desde Windows del dispositivo de audio USB

El driver que permitirá usar desde Windows cualquier dispositivo estándar de audio USB (como el desarrollado) es usbaudio.sys. Este driver va a ser realmente la interfaz que ofrece el dispositivo de audio al PC (en el caso de Windows).

Usbaudio.sys permite a Windows reconocer como dispositivo de audio el dispositivo USB. Una vez que Windows reconoce un dispositivo de audio como tal, usarlo es tan sencillo como usar cualquier tarjeta de sonido comercial. De hecho, en el Administrador de Dispositivos de Windows aparecerá el dispositivo USB propio como dispositivo USB de audio correctamente instalado En Linux se dispone del driver *ALSA snd-usb-audio,* que sería el que se tendría que utilizar si se estuviera trabajando en este sistema operativo.

Habiendo adoptado este sistema de comunicación, el software podrá controlar cualquier tarjeta de sonido (incluido el dispositivo de desarrollado), y el hardware podrá ser controlado desde cualquier software de audio (incluida la aplicación desarrollada).

Capítulo 3 - Desarrollo del software

Dado que este proyecto tiene como objetivo el desarrollo software, el capítulo presente será el más extenso y profundo de todos. Este capítulo abarca todo el proceso de análisis y diseño, desde el planteamiento inicial, hasta la solución final.

3.1.- Análisis de requisitos

A continuación se realiza un análisis breve acerca de qué características serían deseables en un software multicanal para su uso práctico.

En primer lugar, es necesario un entorno basado en pistas. Cada pista debe ser una unidad independiente que lleva asociada un fichero de audio y una serie de parámetros. Cada pista debe disponer de un control de volumen y otro de balance como mínimo, aunque sería deseable disponer además de controles de SOLO, MUTE y REC. Además, se debe permitir agregar o eliminar pistas de la forma que más interese.

Estos entornos basados en pistas se pueden encontrar en cualquier software de este tipo, como por ejemplo *Cubase 3.0, Adobe Audition 2.0, Audacity, Musix*… Permiten representar gráficamente los ficheros de onda, hacer zoom y desplazamientos, y además tener un control sobre los parámetros más importantes asociados a cada pista. En la Figura 3.1 se muestra una captura de pantalla del programa *Adobe Audition 1.5*, para comprender en qué consiste el entorno basado en pistas.

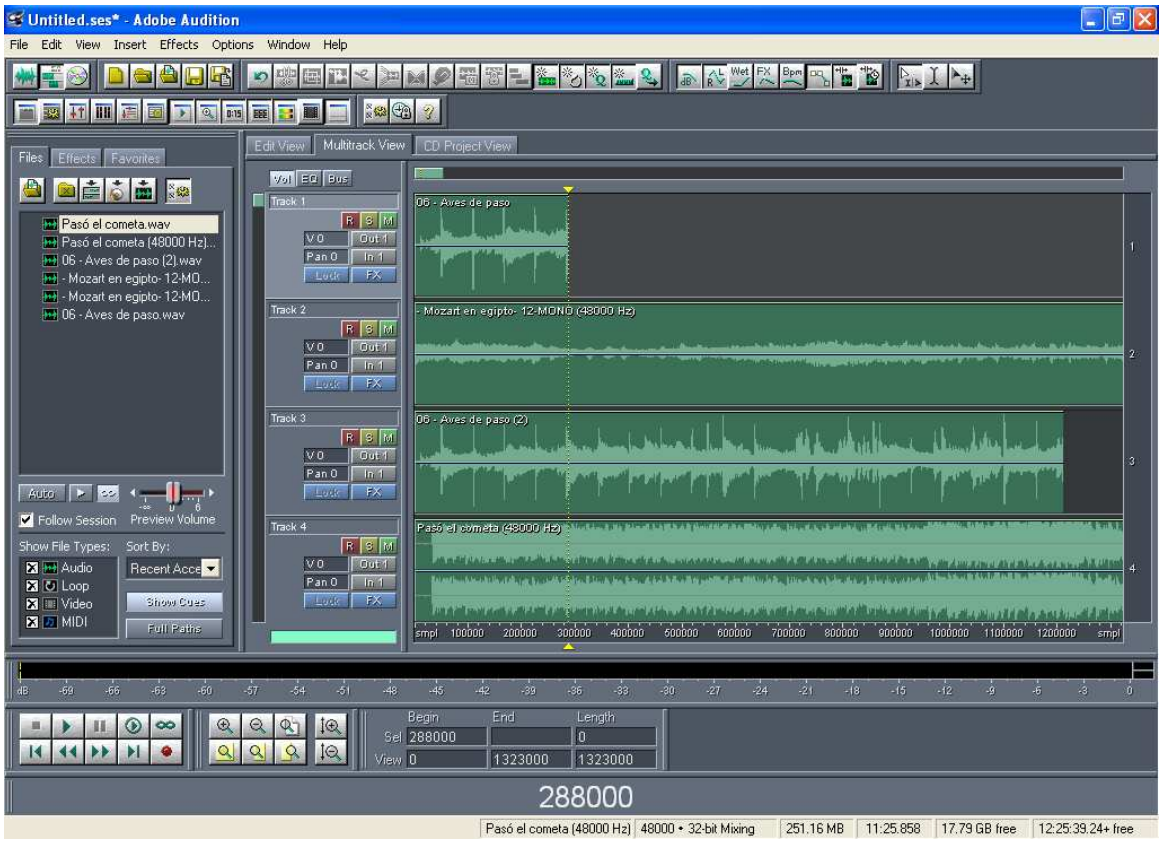

Figura 3.1.- Captura de pantalla de Adobe Audition 1.5

En todo entorno de pistas, resulta imprescindible además la capacidad de reproducir y registrar simultáneamente pistas distintas (comunicación *full-duplex*). Es decir, poder escuchar unas pistas mientras en otras se registra el audio entrante. Con este sistema se puede crear un tema musical polifónico fácilmente, ya que se pueden ir escuchando instrumentos ya grabados mientras se registran encima nuevos instrumentos.

Existe una característica que aunque no es imprescindible, sí muy recomendable, sobre todo en este caso: la capacidad de reproducir pistas con diferentes frecuencias de muestreo simultáneamente. Esto es algo que *Adobe Audition* por ejemplo no lo hace, y sin embargo *Audacity* (software libre) sí. Debido a que la frecuencia de muestreo del sistema hardware desarrollado no es igual al estándar de CD (44.1Khz), sería muy deseable poder reproducir ficheros de ambos formatos simultáneamente. De otra forma sería necesario submuestrear o sobremuestrar cada fichero por separado para adaptarlos a la frecuencia de muestreo establecida.

Es también muy práctica, para un software de audio, la capacidad de desplazar una onda con respecto a las otras. De esta forma, si por ejemplo se desea colocar un pequeño fragmento de audio en un instante determinado, simplemente hay que arrastrarlo hasta la posición deseada.

Por último, otra característica que resulta muy deseable es la capacidad de aplicar efectos de audio. Estos efectos de sonido pueden o no disponer de una interfaz gráfica para seleccionar sus parámetros (o para representar, por ejemplo, el espectro frecuencial). Además, estos efectos de sonido deberían poderse aplicar en tiempo real sobre el audio, o manipulando el fichero WAV.

Habiendo delimitado a grandes rasgos las características y posibilidades del software desde el punto de vista del usuario, se continuará explicando las cuestiones que se plantean como desarrollador a la hora de elaborar un software de estas características.

3.2.- Elección del lenguaje de programación

Un paso muy importante a la hora de desarrollar un software de cualquier tipo es la elección del lenguaje de programación a utilizar, ya que de éste dependerán numerosos factores de la aplicación: rendimiento, facilidad para su programación, portabilidad, etc.

Se planteó la posibilidad de usar alguno de estos lenguajes: *Labwindows, Java, C++ ó C# (Plataforma .NET).*

Labwindows

Inicialmente, este lenguaje pareció ser una buena solución, ya que está muy orientado a aplicaciones de control a través de los puertos del PC. Este proyecto consiste justamente en eso: un dispositivo conectado al PC por el puerto USB que necesita ser controlado. Además proporciona la posibilidad de realizar interfaces gráficas muy vistosas de forma sencilla.

Incluso se llegó a realizar una pequeña prueba de registro de una señal sinusoidal generada internamente en el PC con su correspondiente representación gráfica. Sin embargo ciertos detalles hicieron que se descartara este lenguaje.

En primer lugar este lenguaje está basado en C, no en C++. No se dispone de una programación orientada a objetos, y esto hace que las posibilidades empiezan a cerrarse enormemente. Además, el sistema de eventos que trae conlleva la utilización de excesivas variables globales, complicando enormemente la programación.

La conclusión es que este lenguaje es adecuado cuando se desea una aplicación sencilla con grandes capacidades de control, pero no para crear un software con una estructura compleja.

Java

Java es un lenguaje que tiene una ventaja importantísima: es portable de una plataforma a otra, algo que aportaría un valor añadido al software. Además es un lenguaje orientado a objetos, prescinde del uso de punteros y existe mucha documentación sobre él en Internet. Sin embargo, tiene el grave inconveniente de que no es un lenguaje compilado, sino semi-interpretado, por lo que el rendimiento termina siendo bajo. Esta aplicación requiere gran movimiento de datos y una buena capacidad de cálculo, así que este detalle descartó inmediatamente a Java.

C++

Quizá la opción que se barajó durante más tiempo fue programar en C++. Se trata de un lenguaje de muy buen rendimiento, orientado a objetos y con amplia documentación en Internet. Existe la posibilidad de programar aplicaciones multiplataforma en él, e incluso se puede utilizar un entorno de desarrollo visual, como Visual C++ 6.0. Existen además numerosos entornos de desarrollo para Linux en C++.

Sin embargo, programar en C++ aplicaciones complejas puede llegar a ser bastante laborioso. El tiempo dedicado a programar una aplicación de estas características en C++ puede ser excesivo, ya que trabajar con punteros hace mucho más complicada la depuración del programa. Por tanto, aunque no se descartó esta posibilidad de inmediato se intentó buscar la solución en la novedosa Plataforma .NET de Microsoft.

C#

Finalmente, C# pareció ser la solución más compensada y que mejor se adaptaba a las necesidades planteadas. Las ventajas del uso de este lenguaje son numerosas.

En primer lugar este lenguaje pertenece a la Plataforma .NET de Microsoft, lo que permite desarrollar aplicaciones usando toda la potencia de ésta. Al mismo tiempo se confía el desarrollo a una de las mayores firmas de software del mundo: Microsoft, garantizando la adaptación continua a futuras tecnologías.

La sintaxis y el concepto general de C# es muy similar a Java, ya que se trabaja a un nivel muy cercano al usuario y hace prescindible el uso de punteros. No obstante, si surge la necesidad, C# permite el uso controlado de gestión a bajo nivel de memoria (punteros, reservas de memoria, etc). Además, el entorno de desarrollo que ofrece .NET facilita la programación y la depuración enormemente.

C# dispone además de otras ventajas de las que carecen otros lenguajes de programación: sistema de tipos homogéneos, indexadores, capacidad de definición de propiedades, operadores sobrecargables, etc. Son pequeños detalles que facilitan la programación y favorecen la correcta estructuración del código.

Es importante mencionar que debido a la arquitectura de la Plataforma .NET, C# no es un lenguaje compilado propiamente dicho, sino que sigue una arquitectura similar a la de Java. Sin embargo, Microsoft Visual C# 2005 Express, a diferencia de Java, sí consigue un rendimiento suficientemente alto como para desarrollar una aplicación de estas características sin demasiados problemas. Cabe destacar, no obstante, que el rendimiento de C# siempre será más bajo que el de C++ debido a su naturaleza de lenguaje semi-compilado.

A continuación se comentarán con más detalle las características de la Plataforma .NET.

3.3.- Introducción a la Plataforma .NET de Microsoft

La Plataforma .NET es la nueva apuesta de Microsoft para crear una plataforma de desarrollo de software muy orientada a Internet y redes de computadores, multiplataforma y que además permita un rápido desarrollo de aplicaciones. El objetivo de Microsoft es ir adaptando todo el sistema operativo de forma conjunta con la Plataforma .NET. De hecho, el nuevo Windows Vista permite una mejor integración con .NET que su anterior XP.

.NET Framework

La base de la Plataforma .NET es el *framework*, o marco de trabajo. Este marco de trabajo consta esencialmente de:

- El conjunto de lenguajes de programación
- La Biblioteca de Clases Base o BCL (Base Class Library).
- El Entorno Común de Ejecución para Lenguajes, o CLR (Common Language Runtime)

Este marco de trabajo está contenido en el .NET Framework. La última versión de este .NET Framework es la 3.0, aunque esta aplicación está desarrollada con la versión 2.0.

Se dispone además del .NET Framework convencional, de un marco de trabajo especializado para el desarrollo en aplicaciones móviles: el .NET Compact Framework. Es una versión reducida y especializada del .NET Framework.

El marco de trabajo .NET soporta ya más de 20 lenguajes de programación (C#, Visual Basic, C++, Fortran, Cobol .NET, J#, etc.). Debido a que todos tienen el mismo marco de trabajo como raíz común, las diferencias entre lenguajes se traducen únicamente en detalles como la sintaxis, etc. Las bibliotecas de clases son comunes a todos los lenguajes, y todos se ejecutan sobre una especie de "máquina virtual": el CLR.

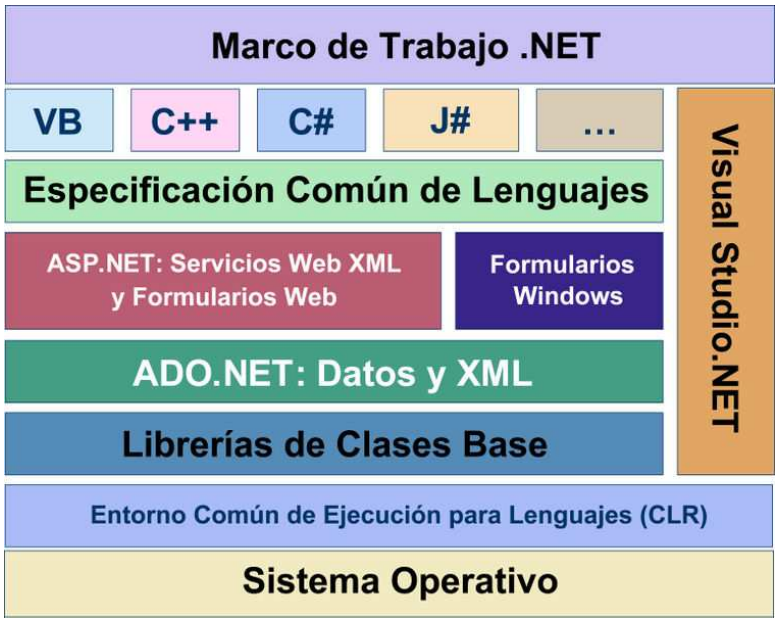

Figura 3.2.- Estructura de capas de la Plataforma .NET (Fuente de la imagen: [25])

Common Language Runtime (CLR)

El núcleo del Framework .NET es el entorno de ejecución (CLR), sobre el cual se cargan las aplicaciones desarrollada en cualquier lenguaje .NET. Algo similar a lo que hace Java con la famosa "máquina virtual de Java".

Gracias al concepto de CLR, permite en teoría la portabilidad de la Plataforma .NET. Bastaría con desarrollar el correspondiente entorno de ejecución para el sistema operativo deseado, pudiendo cargar sobre él cualquier aplicación .NET.

La herramienta de desarrollo, al "compilar" la aplicación, crea unos ficheros en un código intermedio llamado MSIL (Microsoft Intermediate Lenguaje), similar al código ensamblador. Posteriormente, el CLR realiza un compilado a tiempo real JIT (Just-in-time) que ya sí genera el código máquina real que ejecutará el hardware. Debido a que la aplicación se compila a nivel de máquina en algún punto del proceso, el rendimiento es mucho mayor que en el sistema de Java.

Actualmente existe un proyecto de software libre llamado Proyecto Mono. Este proyecto actualmente está desarrollando un Framework para GNU/Linux similar al de Windows. No

obstante queda aún mucho trabajo por desarrollar para poder compatibilizar todas las clases de Framework .NET con Mono.

Biblioteca de Clases Base (BCL)

Los Framework .NET (1.1, 2.0, y actualmente 3.0) incluyen potentes herramientas en forma de librerías de clases. Dependiendo del tipo de aplicación que se desee desarrollar, se disponen de clases clasificadas en tres grandes grupos:

- ASP.NET y Servicios Web XML (Orientadas a Internet)
- Windows Forms (Aplicaciones basadas en formularios)
- ADO.NET (Herramientas de bases de datos)

A su vez, cada uno de estos grupos contiene una enorme cantidad de clases que van a permitir un cómodo y potente desarrollo de aplicaciones.

3.4.- Elección de la API adecuada para audio

Tal y como se explicó en el apartado 2.3, se ha seguido la especificación estándar para dispositivos de audio USB como protocolo de comunicación. Esto implica que para poder trabajar con el dispositivo USB desarrollado, desde Windows se recurrirá al driver genérico: usbaudio.sys.

Sin embargo, si se intenta utilizar usbaudio.sys directamente desde el lenguaje de programación, se observará que la tarea no es tan sencilla. Existen parámetros del driver que se deberían programar en C a bajo nivel si se desean lograr resultados satisfactorios. Para ahorrarse esta laboriosa tarea aparecen las APIs (siglas explicadas en nota al pie [5]).

Las APIs interaccionan con elementos de un nivel más bajo, como los drivers, para proporcionar servicios al programador sin necesidad de conocer su funcionamiento interno completamente. Estas APIs casi siempre se presentan en uno o varios archivos .DLL. En la Figura 3.3 se muestra un esquema de la estructura de capas que existe entre la aplicación y el hardware.

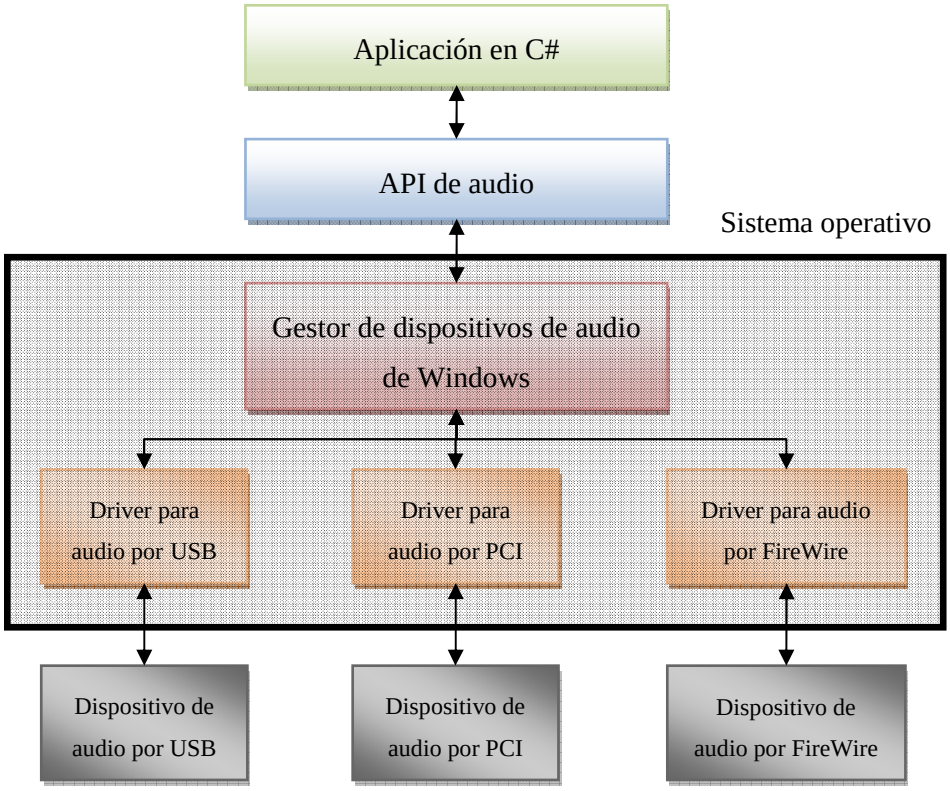

Figura 3.3.- Estructura de capas que existe entre aplicación y hardware

En este caso, es necesaria una API que permita usar dispositivos de audio con facilidad. Tras una laboriosa búsqueda de información se encontraron varias APIs que realizan esta tarea. Las APIs más utilizadas en Windows XP son DirectSound, PortMusic y WinMM.

Si el dispositivo de audio a controlar siguiera la especificación ASIO[7] existe una API ASIO para su control. El dispositivo USB desarrollado no es ASIO, por lo que esta API no es necesaria en este caso.

En el nuevo Windows Vista, existe el conjunto de APIs Core Audio (nombre heredado de la API para audio de Macintosh), que ofrece posibilidades nuevas para gestión multicanal de audio con baja latencia. Existe además una versión para .NET Compact Framework que permite

[7] Las siglas ASIO significan Flujo de Audio de Entrada y Salida (*Audio Stream Input Output)*, y es un estándar que pretende establecer un protocolo de comunicación entre los dispositivos de audio y el software con características novedosas, como baja latencia y gestión multicanal eficaz.

desarrollar aplicaciones de audio para dispositivos móviles. Sin embargo, puesto que se va a trabajar en Windows XP, esta API no resulta interesante.

A continuación se comentará acerca de DirectSound, PortMusic y WinMM para argumentar la elección final de la API a utilizar.

DirectSound

Esta API forma parte del paquete de librerías DirectX de Microsoft. Estas librerías pretenden ofrecer al desarrollador todo un juego de utilidades para el desarrollo de juegos. No obstante, son muy usadas para el desarrollo de cualquier tipo de aplicaciones multimedia.

DirectSound incluye un amplio potencial para la reproducción simultánea de sonidos (algo muy usado en juegos), pero descuida las funciones relacionadas con la grabación. Aunque las nuevas versiones de DirectX empiezan a incluir más y más funciones adecuadas para audio, se pensó que para el desarrollo de esta aplicación era un poco arriesgado aventurarse con estas librerías. Por ello se prefirió no usarlas, aunque no se descartó la necesidad de recurrir a ellas.

PortMusic

Esta API es multiplataforma, e incluye todas las funciones necesarias para realizar aplicaciones de audio. Sin embargo, el hecho de ser multiplataforma en este caso no es útil ya que la aplicación estará desarrollada solamente para Windows. En consecuencia, la complejidad que conlleva el uso de PortMusic no sale rentable. Además, no se encontró una documentación clara con ejemplos precisos en Internet para comenzar a trabajar.

Win MM

Esta API forma parte de la macro API nativa de Windows (API Win32) y se puede encontrar nombrada de distintas formas: Win MM (Windows multi-media), MMSystem, o simplemente la API de Win32 para manejo de sonido.

Esta es finalmente la API que se usó para el desarrollo. Aunque no proporciona demasiada potencia para la gestión multicanal de audio, es suficiente para crear el software propuesto. De hecho, numerosos programas comerciales utilizan esta API para el control de dispositivos de audio, por lo que resulta fiable trabajar con ella.

La API WinMM proporciona funcionalidad para la grabación y la reproducción de audio a través del dispositivo de sonido instalado que se especifique. Además, incluye funciones para

adquirir datos acerca de los dispositivos de audio instalados, como su nombre, su fabricante, sus características, etc.

La API se encuentra en el fichero winmm.dll, incluido en el directorio C:\WINDOWS\system32. Para sistemas de 16 bits antiguamente se usaba mmsystem.dll. Este fichero existe aún en Windows XP debido a que winmm.dll hace uso de él internamente para funcionar correctamente, aunque esto es algo que el programador no tiene por qué conocer.

La librería winmm.dll consta de funciones, tipos y constantes predefinidas. A las funciones de WinMM se las denominará habitualmente *funciones nativas*, ya que pertenecen a la API Win32 nativa de Windows. Se puede encontrar un listado con su correspondiente descripción de todas ellas en la ayuda online para desarrolladores de Microsoft [11].

En este caso, de las muchas funciones que incluye winmm.dll, se usarán solamente las necesarias para identificar los dispositivos, reproducir audio y registrar audio de la forma más simple posible. A continuación se describe brevemente la forma de reproducir y registrar audio con esta API.

Uso de WinMM para reproducir y registrar audio

Las funciones de reproducción y grabación están pensadas para enviar o recibir bloques de datos con un tamaño especificado de forma continuada. Los bloques de datos nuevos se van colocando en una cola para ser reproducidos o registrados cuando les toque. Si el tamaño del buffer es muy grande, se producirán latencias grandes en la grabación y en la reproducción, pero se logrará que el PC trabaje de una forma más liviana. El tamaño de buffer a escoger supone un compromiso entre latencia y recursos.

Para poder comenzar a enviar o recibir datos, primero se debe “iniciar” una comunicación con un dispositivo determinado. Para ello, es necesario crear una instancia del dispositivo en una variable a modo de identificador. Este identificador será el que se use para determinar a qué dispositivo (o desde qué dispositivo) van dirigidos los datos y en qué formato. Una vez que la comunicación ha terminado, hay que eliminar de la memoria este identificador y todos los recursos asociados.

Cuando se inicia una comunicación es necesario también especificar la rutina (o función) que se ejecutará al terminar la reproducción o la grabación de un buffer. Esta rutina debe tener una cabecera ya establecida para poder funcionar adecuadamente. Se utilizará para hacerle saber al resto del programa cuándo ha finalizado la ejecución de un determinado buffer, y lograr así el

sincronismo necesario. Si no existiera, el resto del programa no tendría ninguna forma de saber si está reproduciendo o ha acabado ya de hacerlo.

Los bloques de datos a enviar o recibir deben ir integrados en un tipo de estructuras concretas para poder ser enviados o recibidos. Por así decirlo, de la misma forma que los datos digitales van encapsulados en tramas, estos bloques de datos deben ir encapsulados en estructuras del tipo `WaveHdr`. Por lo tanto, antes de enviar o recibir datos hay que crear una estructura de tipo `WaveHdr`, inicializarla y asociarla al bloque de datos mediante un puntero a él. En este punto ya está todo a punto para enviar datos por las funciones anteriormente mencionadas.

Es importante destacar el hecho de que para enviar cada vez un bloque de datos nuevo no es necesario crear una nueva estructura del tipo `WaveHdr`, basta con cambiar el puntero a otro bloque de datos distinto.

Las funciones que van a permitir realizar este proceso son las siguientes:

- `waveOutOpen()` → Crea un identificador asociado al dispositivo de audio de salida especificado.
- `waveInOpen()` → Crea un identificador asociado al dispositivo de audio de entrada especificado.
- `waveOutWrite()` → Reproduce un buffer pasado por parámetro. Si ya había alguno en reproducción se almacena en una cola de reproducción.
- `waveInAddBuffer()` → Añade un buffer preparado a la cola de buffers de grabación.
- `waveOutReset()` → Detiene la reproducción y vacía la cola de buffers.
- `waveInReset()` → Detiene la grabación y vacía la cola de buffers.
- `waveInStart()` → Comienza el registro sobre la cola de buffers.
- `waveOutClose()` → Elimina de memoria el identificador de dispositivo de reproducción.
- `waveInClose()` → Elimina de memoria el identificador de dispositivo de grabación.
- `waveOutPrepareHeader()` → Prepara una instancia del tipo `WaveHdr` con su bloque de datos de salida.

- `waveInPrepareHeader()` → Prepara una instancia del tipo `WaveHdr` con su bloque de datos de entrada.
- `waveOutUnprepareHeader()` → Libera de memoria una instancia de `WaveHdr` para reproducción.
- `waveInUnprepareHeader()` → Libera de memoria una instancia de `WaveHdr` para grabación.
- `waveInStop()` → Detiene la grabación.
- `waveOutGetNumDevs()` → Devuelve el número de dispositivos de salida de audio instalados en Windows.
- `waveInGetNumDevs()` → Devuelve el número de dispositivos de entrada de audio instalados en Windows.

Estas funciones, en numerosas ocasiones, pasan por parámetros variables de tipos definidos en winmm.dll, como ciertas estructuras o algunos tipos enumerados. Las estructuras más importantes son:

- `WaveFormat` → Tipo de datos para definir el formato de un flujo de audio. En él se incluyen parámetros como la frecuencia de muestreo o el número de bits por muestra.
- `WaveHdr` → Tipo que deben tener las instancias que contendrán los bloques de datos.

Por otro lado, existen numerosas constantes definidas para designar estados de la reproducción o la grabación, o para asignar nombre a los posibles errores que se puedan producir. El resto de miembros de la API se pueden encontrar en [11].

3.5.- Integración de winmm.dll en C#

Una vez que se conocen las funciones de la API que se desean utilizar, es necesario poder recurrir a ellas desde el lenguaje de programación. Casi todos los lenguajes traen opciones de interoperabilidad para invocar funciones contenidas en ficheros DLLs en tiempo de ejecución. En C#, como es de esperar, esto también se puede hacer. Para ello es necesario recurrir a las clases pertenecientes al espacio de nombres[8] `System.Runtime.InteropServices`.

[8] Un espacio de nombres en C# sirve para organizar en grupos un conjunto de clases, y puede ser usado para diferenciar dos clases con el mismo nombre creadas con finalidades distintas.

Cuando se hablaba de las características de C#, se decía que aunque el uso de punteros es mínimo al desarrollar, se dispone de ellos para los casos en que fuera necesario. A la hora de utilizar funciones contenidas en DLLs desarrolladas en C o C++, se observa que muchos parámetros de funciones son punteros. Por ello el Framework de .NET incluye un tipo llamado `IntPtr` que se utiliza para representar un puntero o un identificador. Para los pasos de punteros por parámetro se utilizará este tipo.

En el espacio de nombres anteriormente mencionado se puede encontrar el método estático `DLLImport`, que va a permitir definir métodos en una clase cualquiera provenientes de una DLL. El Código 3.1 ilustra cómo importar de una DLL una función nativa para su uso en C#.

```
[DllImport("winmm.dll")]
public static extern int waveOutWrite(IntPtr hWaveOut,
                                      ref WaveHdr lpWaveOutHdr,
                                      int uSize);
```

Código 3.1.- Importación de función a partir de una DLL nativa

En cuanto a los tipos de datos nativos, simplemente es necesario crear las mismas estructuras que las descritas en la documentación de winmm.dll, y nombrarlas de la misma forma. De esta forma, los tipos `WaveFormat` y `WaveHdr` anteriormente mencionados están declarados sabiendo qué campos tienen las estructuras de datos y qué nombres deben llevar. Por ejemplo, el tipo `WaveHdr` se declararía cómo muestra el Código 3.2.

```
[StructLayout(LayoutKind.Sequential)]
public struct WaveHdr
{
      public IntPtr lpData; // puntero a un array de bytes(datos)
      public int dwBufferLength; // longitud de dicho array
      public int dwBytesRecorded; // se usa sólo para grabación
      public IntPtr dwUser; // para uso del cliente
      public int dwFlags; // Flags que no se usarán
      public int dwLoops; // contador de control de bucle
      public IntPtr lpNext; // PWaveHdr, reservado para el driver
      public int reserved; // reservado para el driver
}
```

Código 3.2.- Definición de tipo nativo mediante estructuras en C#

Quedando perfectamente definido el tipo con esta declaración, y pudiendo usarse en las funciones de la API. En este caso se han realizado todas las declaraciones de miembros nativos de winmm.dll en el mismo fichero: WaveNative.cs.

3.6.- Descripción del sistema de reproducción y grabación

Hasta aquí se ha descrito cómo se puede usar la API de winmm.dll desde C#. Sin embargo, su uso sigue siendo demasiado complejo como para que se use directamente desde el programa principal. Esto hace plantearse que sería conveniente integrar las funciones de la API en clases autosuficientes capaces de reproducir y registrar audio por sí solas. Para implementar dichas clases, es conveniente crear algunas otras clases auxiliares con el fin de modularizar al máximo el código.

Clase WaveOutBuffer

Es necesaria una clase que defina algún tipo de buffer ideado para salida de audio. Además, esta clase debe ser capaz de reproducir a través de distintos dispositivos de sonido simultáneamente. Para ello va a albergar un array de elementos de tipo `WaveHdr` (uno para cada dispositivo de sonido de salida), y permitirá la reproducción de los bloques de datos asociados a cada una de las estructuras simultáneamente. Conviene recordar que la estructura `WaveHdr` almacenaba un puntero a un bloque de datos, y tras una correcta preparación permitía a `waveOutWrite()` reproducir dichos datos. No se pueden reproducir datos sin haberlos preparado previamente en una instancia del tipo `WaveHdr`.

La clase `WaveOutBuffer` debe disponer al menos de un array de arrays (uno por cada dispositivo de sonido de salida instalado en el PC) llamados `Data[]`. Esta variable en realidad va a contener punteros a arrays, por eso sólo se colocan un par de corchetes en su declaración. Además será necesario un método llamado `Play()`. De esta forma, mediante el acceso a `Data[]` se cargan los datos a reproducir, y mediante `Play()` se inicia la reproducción (o se colocan a la cola de reproducción de cada dispositivo respectivamente).

Pero además, la clase `WaveOutBuffer` tiene una particularidad que será de gran utilidad: En sí misma es una lista enlazada circular de *n* buffers de tipo `WaveOutBuffer`. Esto se consigue incluyendo en `WaveOutBuffer` una variable tipo `WaveOutBuffer` llamada `SiguienteBuffer`. El Código 3.3 muestra cómo se implementa este sistema.

```
internal class WaveOutBuffer : IDisposable
{
  public WaveOutBuffer SiguienteBuffer;
  public IntPtr[] Data;
...
```

Código 3.3.- Implementación práctica de la lista circular

Este sistema permitirá recorrer el sistema de búferes mediante llamadas del tipo: `m_BufferActual = m_BufferActual.SiguienteBuffer.` Al cabo de *n* asignaciones como estas se retornaría a la cabecera de la lista. El algoritmo a grosso modo para reproducir audio almacenado en un fichero sería el siguiente:

Se leen del fichero (o ficheros) de audio los datos correspondientes, se cargan en el primer buffer de la lista enlazada y se reproducen. Mientras este búfer se reproduce, se va recorriendo la lista circular y se van cargando de datos el resto de búferes, de manera que estos búferes se colocarán en la cola de reproducción. Cuando se da la vuelta a la lista y se llega al búfer que se está reproduciendo en ese momento, hay que esperar hasta que termine antes de cambiar sus datos, ya que si no se estarían alterando los datos en uso.

Todo este proceso garantizará en todo momento que existen datos suficientes en la cola para que no se interrumpa la reproducción. Se explicará este algoritmo mediante un diagrama de flujo en la descripción de la siguiente clase.

Clase WaveReproductor

Aunque la clase `WaveOutBuffer` de forma independiente es capaz de reproducir bloques de datos sueltos en distintos dispositivos simultáneamente, es necesario organizar todo el entramado de búferes dentro de otra clase para una reproducción continua. La clase encargada de esta tarea se va a llamar `WaveReproductor`.

Desde el programa principal (el código cliente), la clase que se usará finalmente para la reproducción de audio es `WaveReproductor`. Ésta a su vez usará la clase `WaveOutBuffer` de forma interna e implementará el algoritmo descrito en el apartado anterior.

Interfaz de la clase WaveReproductor

Cada vez que se desee reproducir audio se creará una instancia de esta clase, y cada vez que se detenga la reproducción dicha instancia se eliminará de la memoria. Es decir, cuando se

pulsa Play se creará una instancia de `WaveReproductor`, y cuando se pulsa Stop esta instancia se eliminará.

A la hora de iniciar una reproducción hay que elegir parámetros tales como el formato de audio, o el conjunto de dispositivos de salida a utilizar. Además, es necesario algún tipo de sincronización para poder enviar los datos de audio de forma ordenada y controlada al reproductor. Esta sincronización se logrará mediante un método delegado[9] pasado por parámetro que será el que proporcione los datos a `WaveReproductor`. El sistema es similar al uso de eventos: un método asignado en tiempo de ejecución que se "dispara" en el momento adecuado para realizar una determinada acción. Será necesario además un método llamado `IniciarReproduccion()`, para poder escoger el momento en el que comenzará la reproducción. Para detener la reproducción simplemente se hará una llamada al destructor de la clase. Los miembros más importantes de `WaveReproductor` son los indicados en el Código 3.4 y en el Código 3.7.

```
public WaveOutPlayer(int Dispositivo, WaveFormat Formato);
public long PosicionEnBytes;
public int NumBuffers;
public double PosicionEnSegundos;
public long PlayingPunteroEnBytes;
public double PosicionPunteroEnSegundos;
public int DeviceCount;
public void Dispose();
public void IniciarReproduccion();
```

Código 3.4.- Miembros públicos más importantes de WaveOutPlayer

```
private DelegadoRellenaBufer m_FillProc;
```

Código 3.5.- Método delegado para adquisición de datos en búfer

[9] Un método delegado es, en C#, lo que en C++ un puntero a función. En este caso concreto se utiliza para generalizar el método de llenado de búferes, dando libertad al programador para su asignación.

Algoritmo interno para reproducción

Cuando se inicia la reproducción, comienza un proceso cíclico para ir cargando buffers y reproduciéndolos en orden. Este proceso se puede comprender bastante bien mediante un diagrama de flujo, como el de la Figura 3.4.

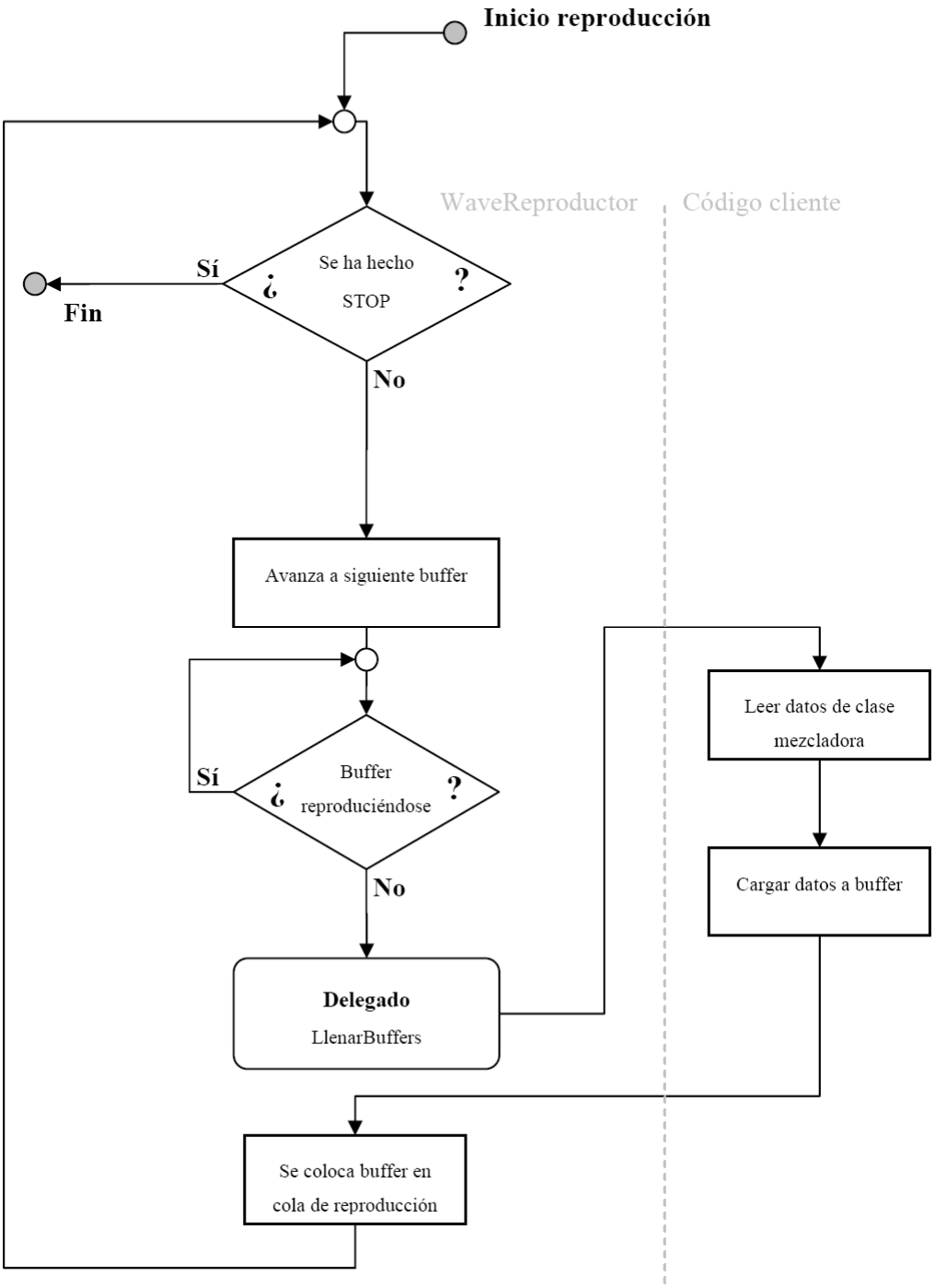

Figura 3.4.- Diagrama de flujo del algoritmo de reproducción

El algoritmo de reproducción se ejecutará en segundo plano mediante un hilo o hebra, para poder así seguir teniendo acceso a la interfaz gráfica del programa.

Cuando se llama al método `Dispose()` (o al destructor de la clase), se interrumpe este ciclo y se eliminan todos los recursos de memoria.

Qué no hace WaveReproductor

Es importante destacar un detalle importante. La clase `WaveReproductor` no es la encargada de mezclar todas las pistas, sino que presupone que los datos vienen ya correctamente organizados. Cuando `WaveReproductor` necesita datos, recurre al delegado que se le ha pasado por parámetro, y éste le devuelve un array de bytes listo para ser reproducido.

Se dispone de otra clase que se explicará más adelante encargada de realizar la mezcla de las pistas y de generar una serie de arrays de bytes que se entregará a `WaveReproductor` para que lo reproduzca por los dispositivos pertinentes.

Clase WaveInBuffer

Esta clase va a ser muy similar a `WaveOutBuffer`, por lo que se ahorraran los detalles comunes en su descripción. Solamente va a variar el sentido del flujo de datos. Si en `WaveOutBuffer`, los datos iban dirigidos desde un array de bytes hacia el dispositivo de salida, en `WaveInBuffer` los datos provienen del dispositivo de entrada en cuestión, para ser almacenados en arrays de bytes.

Al igual que en `WaveOutBuffer`, se podrá registrar audio de distintos dispositivos de audio simultáneamente y almacenarlo en distintos arrays de bytes. Además, también se disponen de las mismas estructuras de tipo `WaveHdr`, del miembro `Data[]`, y en general de los mismos miembros que `WaveOutBuffer`. Cuando se ejecuta `Record()`, los datos entrantes se van colocando en `Data[]` para que posteriormente `WaveGrabador` los recoja y se queden almacenados en el PC.

`WaveInBuffer` también consta de una lista enlazada de buffers que será controlada por `WaveGrabador`. Esto garantizará que siempre hay bufferes en cola para registrar datos del dispositivo, evitando así saltos y clicks.

Clase WaveGrabador

La analogía establecida entre `WaveOutBuffer` y `WaveInBuffer` es muy similar a la que se establece entre `WaveReproductor` y `WaveGrabador`. Ambos tienen la misma funcionalidad,

solamente variando el sentido del flujo. Ambas permiten trabajar con distintos dispositivos de sonido simultáneamente, y ambas trabajan con los mismos parámetros. Además `WaveGrabador` también trabaja mediante hilos en segundo plano, para evitar bloquear la interfaz gráfica.

Sólo hay una diferencia importante entre `WaveReproductor` y `WaveGrabador`. Mientras que `WaveReproductor` puede empezar a llenar los buffers directamente, `WaveGrabador` no puede almacenar en el disco un buffer que aún no ha sido registrado desde el dispositivo. Por ello, en la inicialización de `WaveGrabador` se realiza un llenado de todos los buffers de la lista enlazada para poder comenzar con el ciclo de la Figura 3.5.

Por lo demás, la clase reproductora y la clase grabadora son análogas y ofrecen funcionalidades muy parecidas. En la Figura 3.5 está representado un esquema en el que se ilustra el camino de los datos a través de las distintas clases explicadas y el sentido del flujo.

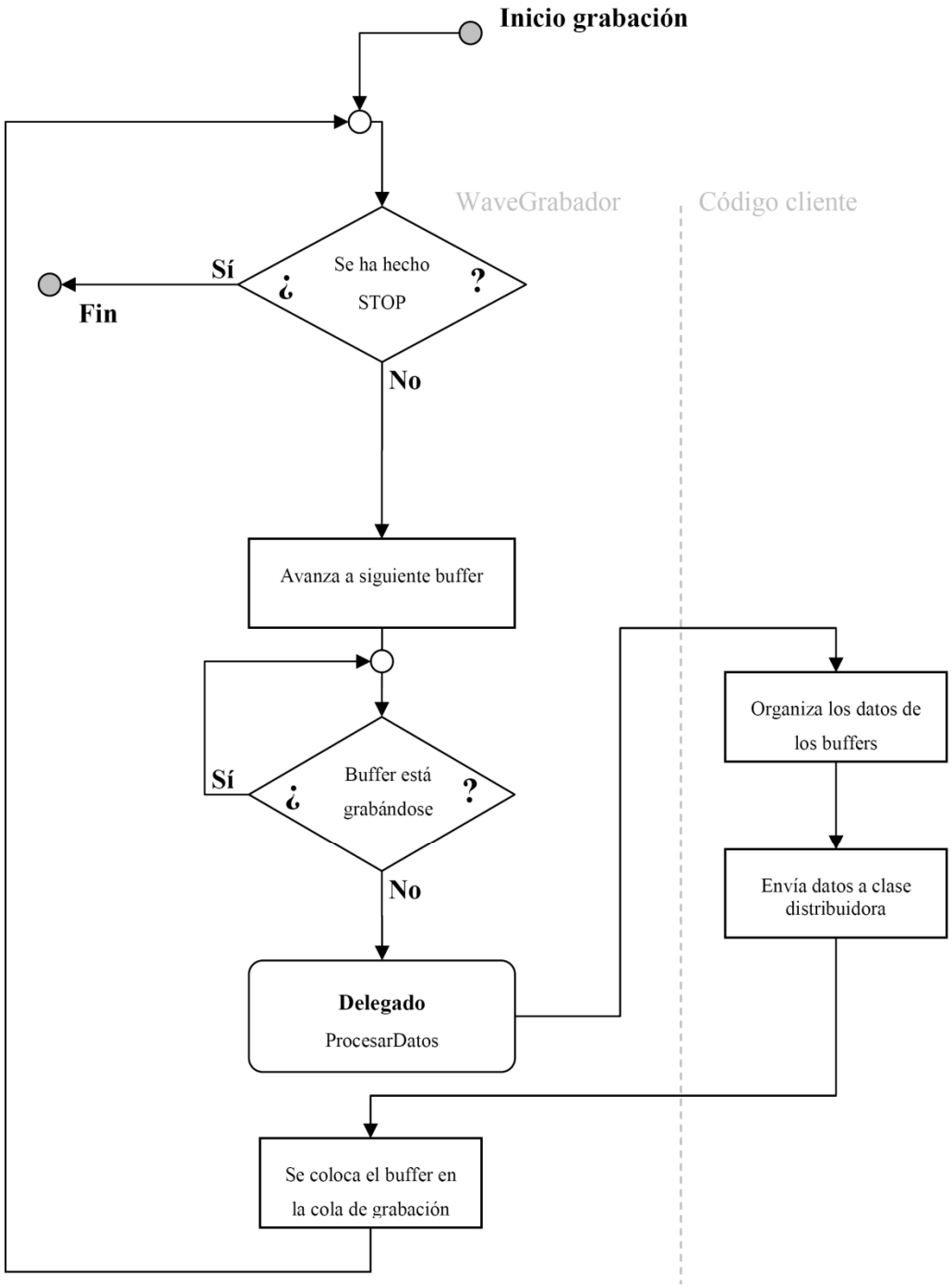

Figura 3.5.- Diagrama de flujo del algoritmo de grabación

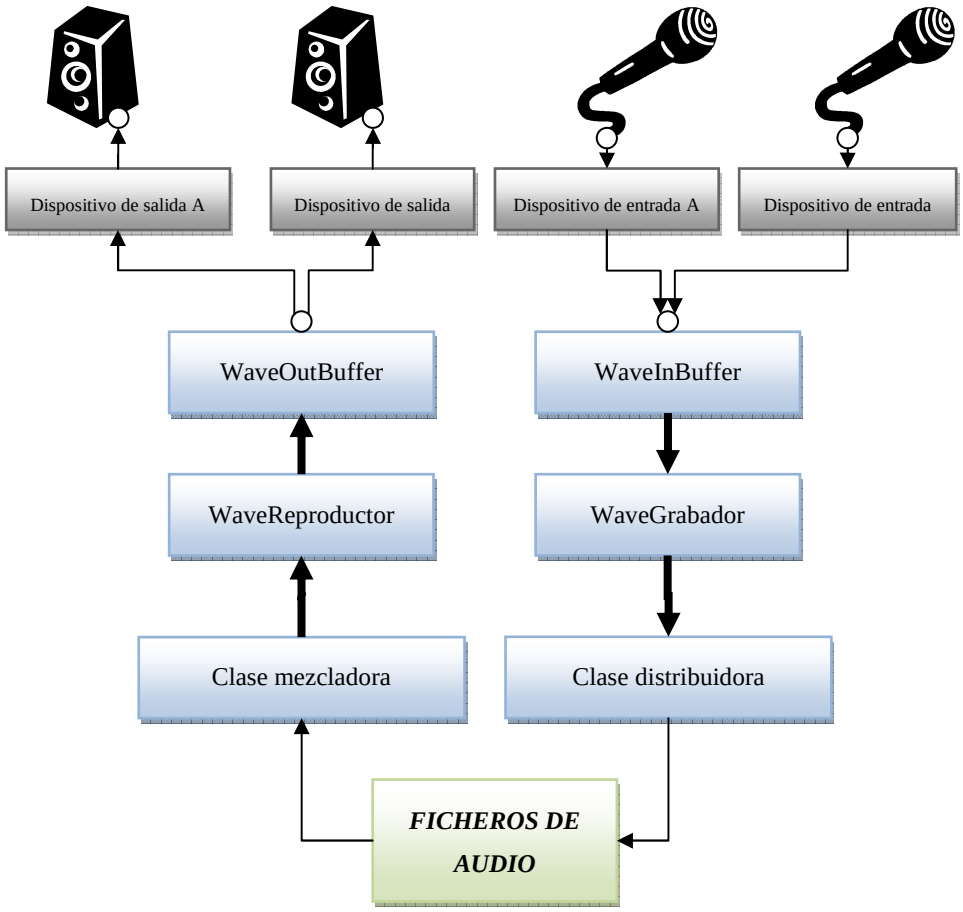

Figura 3.6.- Sentido del flujo de datos por las distintas clases y dispositivos

3.7.- Clases creadas para el uso de ficheros

En este capítulo se van describir las distintas formas que tiene la aplicación de interaccionar con el disco duro, bien sea mediante ficheros de audio o cualquier otro tipo de ficheros.

La aplicación va a trabajar con 4 tipos de ficheros:

- Ficheros .WAV
- Ficheros .MP3
- Ficheros .PIK
- Ficheros .PFC

Los ficheros WAV son ficheros de audio en formato PCM (sin comprimir). Es el formato más sencillo para trabajar, ya que cada muestra se codifica con un valor numérico directamente. Para controlar estos ficheros WAV se dispone de la clase `WaveStream`, que se explicará más adelante.

Los ficheros MP3 son ficheros de audio en formato comprimido. Es necesario un compresor y un descompresor específico para su uso. En la aplicación se recurre a las librerías que proporciona LAME[10].

Los ficheros PIK se explicarán más adelante cuando se describan las técnicas de optimización de `UCGraficaOnda`. Son ficheros auxiliares que se utilizan para hacer más fluida la representación gráfica de ficheros WAV.

Los ficheros PFC son ficheros que almacenan sesiones. Se utilizan para poder almacenar el estado actual de las pistas y los ficheros cargados en ellas en cualquier momento. Estos archivos PFC almacenan todas las características de las pistas y demás configuraciones que pueda haber escogido el usuario. Los ficheros WAV correspondientes a la sesión se almacenan en el mismo directorio que este fichero .PFC. Se dispone de una clase llamada `CSesion` para el gestión de sesiones (abrir sesión, guardar sesión, etc.).

Clase WaveStream

Esta clase facilita la lectura y escritura en ficheros WAV. Para ello se utiliza el concepto del "`WaveStream`". Se trata de un flujo de datos de audio que podrán ser fácilmente leídos o escritos en un fichero WAV del disco duro. Además, este `WaveStream` lleva asociados parámetros de formato tales como frecuencia de muestreo, número de bits por muestra o número de canales. Este concepto de flujo de datos lo introdujo C++ de una forma mucho más genérica para la lectura y escritura de ficheros.

Además, el `WaveStream` asociado a un WAV contiene directamente un `WaveStream` asociado a su fichero de picos (ver apartado 3.7). Cualquier escritura en el fichero WAV se ve reflejada de la forma correcta en el fichero de picos. De esta forma, cuando se registra audio no es necesario realizar un procesado aparte para el cálculo de los picos, sino que se hace simultáneamente.

[10] LAME es un proyecto de software libre para el desarrollo de librerías capaces de codificar y decodificar audio en distintos formatos, incluido el MP3.

`WaveStream` sólo será capaz de leer ficheros WAV en formato PCM (sin comprimir). En la Tabla 3.1 se describe la cabecera de un fichero WAV en formato PCM.

Posición en fichero	Tamaño	Concepto	Descripción
0x0000	4 bytes	ChunkID	Contiene los caracteres “RIFF” en formato ASCII
0x0004	4 bytes	Chunk Size	Tamaño total del fichero restándole 8 bytes. No se usará.
0x0008	4 bytes	Format	Contiene los caracteres “WAVE” en formato ASCII
0x000C	4 bytes	SubChunk1ID	Contiene los caracteres “ fmt” en formato ASCII
0x0010	4 bytes	SubChunk1Size	Contiene el valor 16 por tratarse de un fichero PCM
0x0014	2 bytes	AudioFormat	Contiene el valor 1 por tratarse de un fichero PCM
0x0016	2 bytes	NumChannels	Numero de canales: 1 para mono, 2 para estéreo.
0x0018	4 bytes	SampleRate	Frecuencia de muestreo en Hz.
0x001C	4 bytes	ByteRate	SampleRate x NumChannels x BitsPerSample / 8
0x0020	2 bytes	BlockAlign	NumChannels x BitsPerSample / 8
0x0022	2 bytes	BitsPerSample	Número de bits que ocupará cada muestra
0x0024	4 bytes	SubChunk2ID	Contiene los caracteres “data” en ASCII
0x0028	4 bytes	SubChunk2Size	Tamaño del fichero en bytes dedicado únicamente a datos de audio. Se calcula con la siguiente fórmula: Número de muestras x NumChannels x BitsPerSample/8
0x002C	Variable entre 0 y 4.294.967.304 bytes	Data	Datos de audio en formato PCM. En caso de haber dos canales se irán alternando las muestras de un canal y de otro.

Tabla 3.1.- Estructura de un fichero WAV

El objetivo primordial de `WaveStream` es aislar al usuario de estos detalles del fichero WAV. Por tanto debe disponer de métodos sencillos para la lectura y escritura. Puesto que esta clase hereda de `Stream` (perteneciente al .NET Framework), contiene ya los métodos y propiedades necesarios para el fácil acceso a datos.

Clases MadlldibWrapper y Aumpel

Para integrar en el software la gestión de MP3 se hizo uso de algunas DLL gratuitas (licencia GNU) para compresión y descompresión de MP3: lame_enc.dll, libsndfile.dll y madlldlib.dll.

Estas DLL contienen todo lo necesario para todo el procesado completo de MP3. No obstante, puesto que se trata de DLL creadas en C++, es necesario recurrir a `DLLImport` para importar sus métodos.

Para facilitar la tarea, se obtuvo de *www.thecodeproject.com* un artículo didáctico [7] en el que se utilizan estas tres DLL para crear clases de C# que realicen el procesado. Por tanto, el creador del software simplemente ha integrado en la aplicación este ejemplo, pero no ha sido necesario conocer los detalles del mismo.

En caso de pensar en comercializar el software, sería necesario estudiar los términos de la licencia, y en caso de duda utilizar alguna librería de pago para la gestión de MP3, como *mp3PRO.*

La clase `Aumpel` realiza la compresión y descompresión en MP3. La conversión también se realiza a través de su `método Convert()` pasando por parámetro los nombres de los ficheros a transformar, y el tipo de conversión.

Clase CSesion

Esta clase va a encargarse de gestionar lo que se entiende como *sesión.* Una sesión comprende el número de pistas actual, el estado y las propiedades de cada una, los ficheros de audio cargados en ellos y la configuración general establecida. La clase `CSesion` se encargará de ofrecer al usuario la posibilidad de guardar y cargar sesiones para poder continuar un trabajo después de haber cerrado el programa. Esta clase contiene una referencia al `UCPanelDePistas` en uso, y actúa sobre él para gestionar las sesiones.

La clase `CSesion` va a ofrecer las siguientes funcionalidades:

- Cerrar sesión
- Nueva sesión
- Guardar sesión
- Abrir sesión

Cerrar sesión

Esta funcionalidad simplemente cierra todas las pistas actuales y descarga todos los ficheros de audio implicados. También desliga al programa de cualquier archivo .PFC en uso.

Nueva sesión

Cuando se llama a Nueva Sesión, simplemente se cierra la sesión actual y se añade una pista de audio vacía. Este control y el anterior realizan funciones muy similares.

Guardar sesión

Esta función guarda en un fichero pasado por parámetro de extensión .PFC todos los datos relacionados con la sesión actual. Este fichero .PFC se puede abrir con cualquier procesador de textos para observar todos los datos que almacena.

Además, cuando se guarda la sesión, todos los ficheros temporales asociados a las pistas actuales se almacenan en el mismo directorio que el fichero .PFC con nombres específicos para poderse después encontrar.

Estos ficheros toman el nombre de la sesión, seguidos del número de pista y con extensión WAV. Por ejemplo, si la sesión se llama Prueba.pfc, los ficheros a cada pista de audio serán: Prueba0.WAV, Prueba1.WAV, Prueba2.WAV…

Abrir sesión

Esta función simplemente rescata una sesión guardada a partir de un fichero de extensión .PFC.

Lo primero que va a hacer la función es cerrar la sesión actual para comenzar a cargar del fichero la nueva sesión. Después, establece la configuración general de acuerdo al fichero y configura las características de cada pista. Por último carga los ficheros guardados en el directorio del fichero .PFC en cada pista. Si alguno de estos ficheros no se encuentra, el resto de la sesión puede seguir cargándose sin problemas debido a la gestión de excepciones.

3.7.- Componentes y clases creados para generar el entorno basado en pistas

Consideraciones generales

El desarrollo del entorno de pistas ha sido una de las tareas más complejas y laboriosas. Tanto la estructuración a nivel conceptual, como la implementación de los distintos algoritmos, han sufrido una constante evolución y perfeccionamiento a lo largo de todo el desarrollo.

A la hora de diseñar una interfaz gráfica de usuario (en inglés *Graphics User Interface* o GUI) para una aplicación de Windows en .NET, se recurre al espacio de nombres `System.Windows.Forms`. Este espacio ofrece todas las herramientas necesarias para desarrollar aplicaciones basadas en formularios. Los formularios son contenedores para todo tipo de elementos: botones, etiquetas, menús, barras de herramientas, barras de tareas, o incluso otros formularios. Todos estos elementos se pueden encontrar en el espacio de nombres mencionado.

En Visual Studio .NET, la creación de formularios y la colocación de controles está muy facilitada. Esto es así gracias a los diseñadores de aplicaciones visuales que incluye el Entorno de Desarrollo Integrado (siglas 'IDE' en inglés). Para colocar cualquier control simplemente basta con arrastrar y colocarlo en la posición deseada. El código asociado se genera automáticamente y no es necesario escribirlo manualmente.

Además, el Framework de .NET permite elaborar controles de usuario personalizados. Cualquier clase derivada de `System.Windows.Forms.UserControl` se convierte en un control personalizado al que se puede modificar cualquier parámetro. Este control puede contener además otros controles. Incluso es posible la modificación de su representación visual mediante el reemplazo del método protegido `OnPaint()`, o mediante el evento `Paint` de la clase `UserControl`.

Para poder dibujar líneas, rectángulos, círculos, colocar texto, etc. se dispone de la librería GDI+. GDI+ es la evolución de GDI, antigua librería gráfica de la API Win32. Se pueden localizar todas las funcionalidades gráficas necesarias en el espacio de nombres `System.Drawing`. El recurso [24] fue de gran ayuda para la tarea de la programación gráfica.

Diseño esquemático de la GUI

Para comenzar a desarrollar la interfaz gráfica, primero se hará un análisis esquemático de la misma. En la Figura 3.7 se ha realizado una primera aproximación de lo que debería ser la interfaz gráfica del programa.

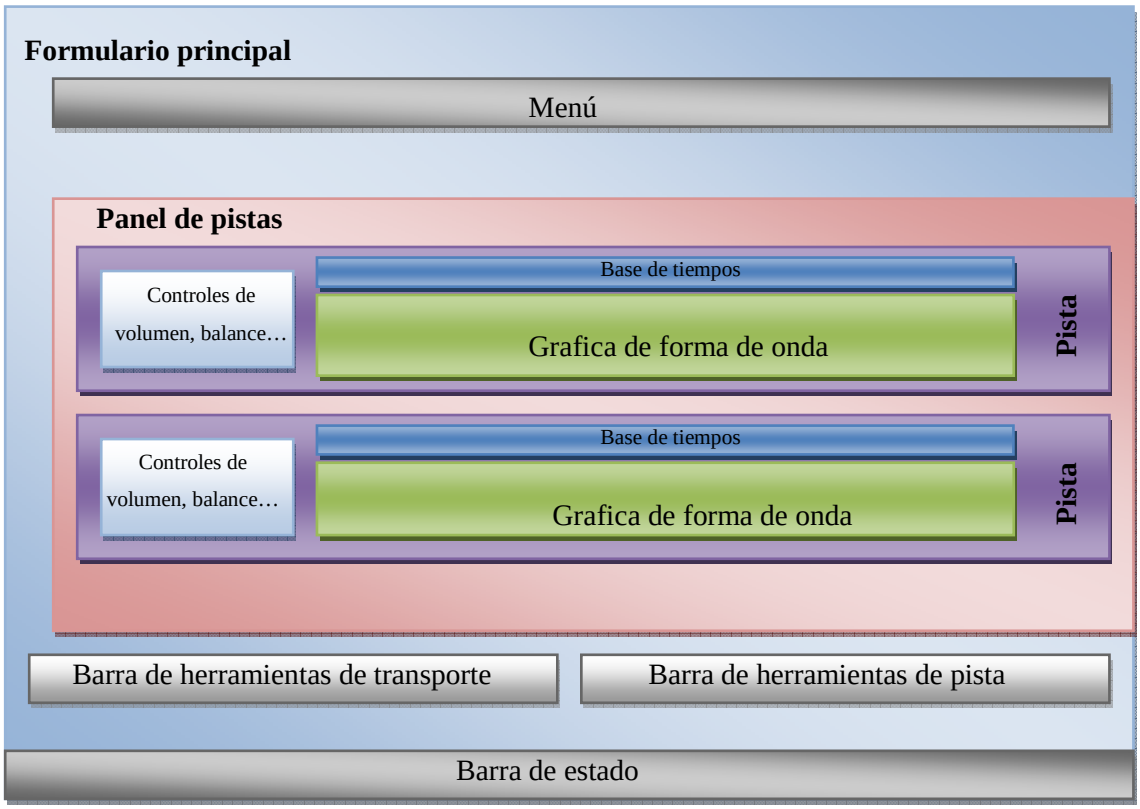

Figura 3.7.- Esquema de la interfaz de usuario de la aplicación

En el espacio de nombres `System.Windows.Forms` se encuentran algunos de los elementos que formarán parte de la interfaz. Por ejemplo, se dispone de herramientas para realizar todo tipo de barras de herramientas, menús y barras de tareas. Se pueden encontrar también botones, barras de desplazamiento, etiquetas, etc. para poder realizar los controles de volumen, balance, solo, mute, rec, etc.

Sin embargo, hay ciertos controles con funcionalidades muy específicas de los que no se dispone en ninguna librería. Estos componentes será necesario diseñarlos e implementarlos creando clases derivadas de `UserControl`. Se trata del control “Grafica de onda”, el control “Base de tiempos”, el control compuesto “Pista” y el control compuesto “Panel de pistas”. Cada

uno tendrá funcionalidades bien definidas, y estarán ideados para una correcta integración de todos ellos.

Para organizar el código de forma óptima, se ha seguido la siguiente regla de estilo: todas las clases que heredan de `UserControl` van a comenzar por `UC`. Por tanto, los controles mencionados anteriormente se implementarían en las siguientes clases:

- `UCBaseDeTiempos`: Esta clase implementa un control capaz de representar una base de tiempos, capaz de mostrar los valores en muestras o en segundos.
- `UCGraficaOnda`: En esta clase se implementará un control capaz de representar gráficamente archivos de onda. Además, tendrá funcionalidades de zoom, posicionamiento de cursor y selección.
- `UCPista`: Esta clase va a integrar las tres anteriores, y además recurrirá a diversos controles de Windows Forms para los controles de parámetros auxiliares.
- `UCPanelDePistas`: Esta control va a funcionar como un contenedor para objetos de tipo pista, con capacidades para registro y reproducción de audio.

Conceptos genéricos sobre controles de usuario

Antes de comenzar con la descripción de cada uno de los controles, se van a definir varios conceptos:

- *Interfaz superior de un control de usuario:* Es el conjunto de posibilidades que ofrece para que el usuario lo pueda controlar. Por ejemplo, el control que tiene el usuario sobre un botón forma parte de su interfaz superior. O el acceso mediante teclado de un usuario a un cuadro de texto también forma parte de su interfaz superior.
- *Interfaz inferior de un control de usuario:* Es el conjunto de métodos y eventos que permiten a un control contenedor usarlo. Por ejemplo, un botón ofrece a su contenedor el evento clic para poder darle uso. Este evento formaría parte de la interfaz inferior del control. O el método `Refresh()` para actualizar su representación también formaría parte de su interfaz inferior. Realmente, la parte pública de una clase configura la interfaz inferior de la misma.
- *Parámetros del control:* Los parámetros de un control es el conjunto de valores que tienen las variables internas de un control. Por ejemplo, en un botón, un posible parámetro del control podría ser un valor booleano que represente pulsado o no

pulsado. O también se podría considerar un parámetro del botón el color, o las dimensiones.

- *Implementación de un control:* Consiste en el conjunto de miembros privados que posibilitan el funcionamiento de un control.
- *Representación visual de un control de usuario:* Es la representación que ofrece por pantalla dicho control. Por ejemplo, el dibujo rectangular que ofrece un botón formaría parte de su representación visual.

Generalmente el esquema para los controles habituales de Windows Forms es el de la Figura 3.8. Las acciones que se realizan en su interfaz superior van a ser procesadas y van a variar los parámetros internos del control. Todo esto antes de que el contenedor del control pueda actuar.

Los controles personalizados que se han creado siguen más bien el esquema de la Figura 3.9. Las acciones sobre su interfaz superior no van a modificar sus parámetros internos directamente, sino que será el contenedor quien realice esa tarea a través de la interfaz inferior.

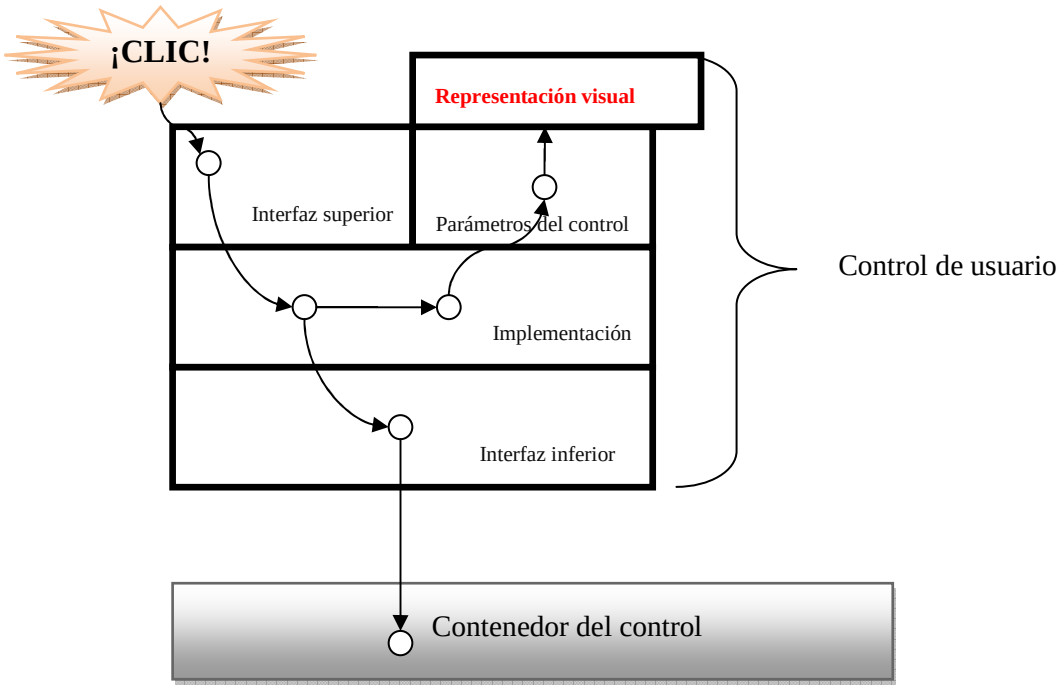

Figura 3.8.- Esquema habitual de controles de usuario

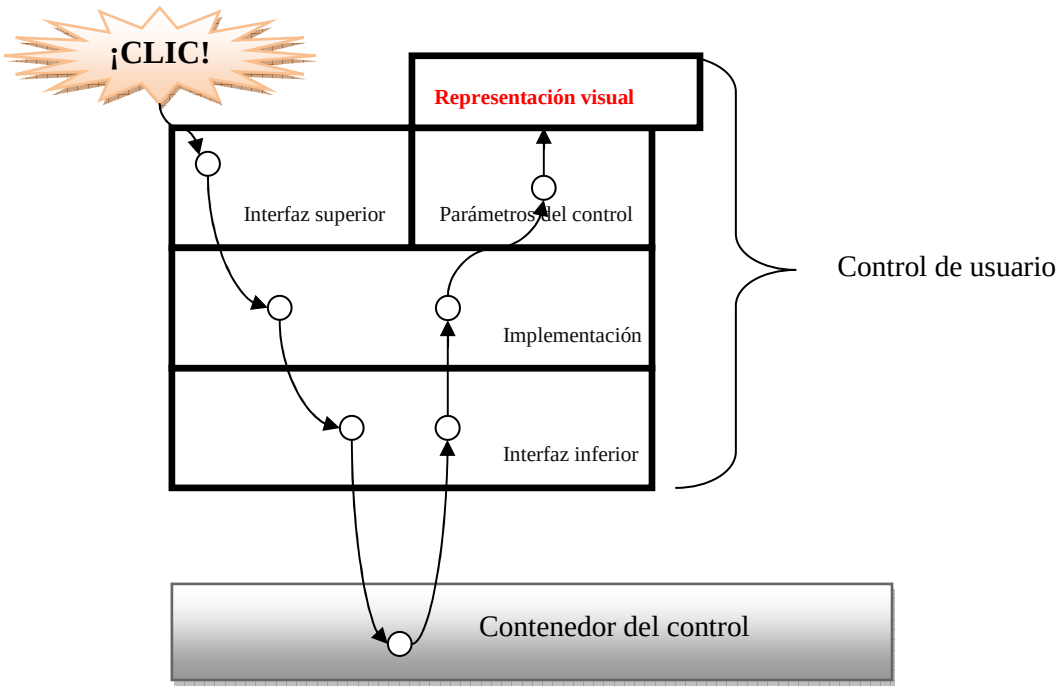

Figura 3.9.- Esquema de los controles de usuario personalizados desarrollados

Este sistema va a permitir que desde el contenedor se pueda controlar los parámetros internos de todos los controles. Esta centralización permite un flujo de programa más controlado y flexible.

A continuación se describe cómo funciona cada uno de los controles personalizados que ha sido necesario crear.

El control UCBaseDeTiempos

Representación gráfica

Este control va a servir como base de tiempos para la representación gráfica. Va a indicar en qué segundo se encuentra el gráfico de onda.

Los números indicadores estarán distribuidos uniformemente para que resulten legibles e intuitivos. Además, la barra está subdividida en partes más pequeñas para tener más precisión en la medida.

Interfaz superior

Por otro lado, el usuario tendrá dos formas de interactuar con la barra:

- Haciendo clic y arrastrando
- Haciendo clic sin arrastrar

Cada acción provoca un evento distinto, que el control contenedor usará para variar parámetros de la representación gráfica. En el primer caso desplazará la representación a derecha o a izquierda, mientras que en el segundo colocará el cursor en la posición deseada.

Este control sigue el esquema de la Figura 3.9, por lo que sus parámetros internos sólo serán modificados desde el control contenedor.

Implementación

La implementación de la barra es una tarea sencilla. Sin embargo, sería conveniente hacer mención del sencillo algoritmo incluido para distribuir los números siempre de forma legible. Para ello, se intenta que siempre haya aproximadamente 10 divisiones a lo largo de la barra. De esta forma, los números nunca quedarán demasiado apretados o demasiado distanciados.

Interfaz inferior

El control `UCBaseDeTiempos` ofrece diversos miembros públicos. Por un lado, propiedades públicas para configurar los colores, y las muestras de inicio y de fin que va a representar, etc. Por otro lado métodos públicos para actualizar el buffer de la base de tiempos, o para representarlo en pantalla. Y por último dispone de varios eventos públicos para indicar al control contenedor que se pretende cambiar el cursor de sitio o desplazar la gráfica, entre otros.

El control UCGraficaOnda

Representación gráfica

El control `UCGraficaOnda` se creó para representar formas de onda por pantalla. Permite representar ficheros de 8 o 16 bits, tanto mono como estéreo. Además, permite realizar funciones de acercamiento (zoom in) y alejamiento (zoom out). Se podrá además seleccionar regiones del fichero, o situar un cursor de referencia. Esto será de gran utilidad para el trabajo de audio multipista.

La representación gráfica se basa en una estructura de capas independientes superpuestas. A continuación se describen las distintas capas por orden de superposición:

- Capa 1: Lo primero que se pinta en pantalla es un rectángulo relleno del color de fondo (por defecto negro).
- Capa 2: Sobre la capa anterior se dibuja la rejilla en color gris. La pista contenedora se encargará de alinear la base de tiempos con esta rejilla.
- Capa 3: Sobre la rejilla se representa la forma de onda propiamente dicha. Por defecto la onda se dibuja de color verde. Puede representar un fichero mono, o uno estéreo.
- Capa 4: Sobre todas las capas anteriores, se dibuja un rectángulo azul semitransparente para indicar la zona seleccionada.
- Capa 5: Existe otra capa que consiste en un rectángulo blanco semitransparente que abarca toda la gráfica. Esta sólo se dibujará si la pista está *en uso*. Este concepto se explicará más adelante.
- Capa 6: Esta capa consiste únicamente en una línea vertical amarilla que representa al cursor.

Estas capas se van a almacenar en búferes independientes, y se van a actualizar de forma separada. Cuando se pinta en pantalla el control `UCGraficaOnda` lo que se hace es dibujar en orden los búferes superpuestos. Para poder conseguir esto se aprovecha la posibilidad de utilizar color semitransparente.

En la Figura 3.10 aparece la representación final con todas las capas superpuestas para un fichero de audio mono. En la Figura 3.11 sucede lo mismo pero para un fichero estéreo.

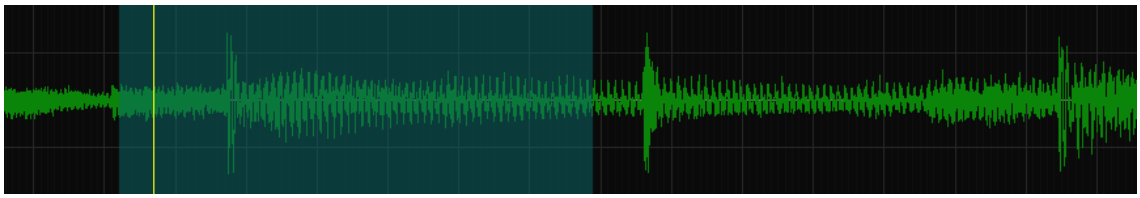

Figura 3.10.- UCGraficaOnda representando un fichero mono

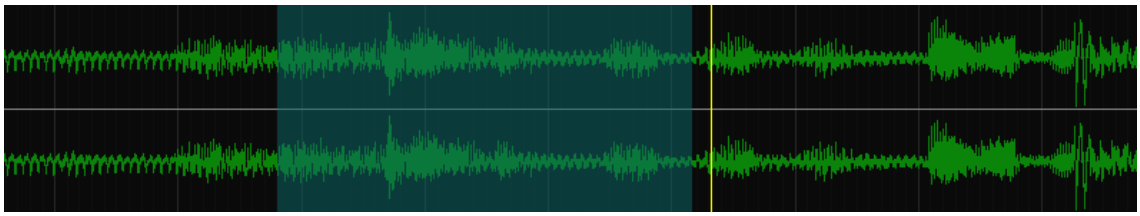

Figura 3.11.- UCGraficaOnda representando un fichero estéreo

El hecho de trabajar con capas independientes facilita todas las tareas de representación. Por ejemplo, representar un archivo mono o uno estéreo es algo que incumbe sólo a la capa 3 (forma de onda), dejando las demás intactas. Se puede cambiar el cursor de sitio sin necesidad de cargar de nuevo toda la forma de onda, o modificar la rejilla sin tocarle a ningún otro elemento.

Cuando se representan muy pocas muestras en pantalla, se dibuja un pequeño rectángulo encima de cada una para localizarlas más fácilmente. Esto se hace modificando la capa de forma de onda, por lo que no afecta en absoluto al resto de las capas.

Interfaz superior

El usuario podrá interactuar con `UCGraficaOnda` de las siguientes formas:

- Mediante un clic simple el usuario podrá colocar el cursor en la posición deseada.
- Si el usuario hace clic y arrastra, podrá seleccionar una región de audio. Una vez seleccionada se puede reproducirlo, hacer zoom, o aplicar efectos de sonido.
- Una vez seleccionado un trozo de audio, si se coloca el ratón sobre los límites de la selección, se puede hacer una modificación de la misma con precisión.
- Si el usuario hace clic con el botón derecho aparece un menú contextual en el que se disponen de diversas opciones. El menú está representado en la Figura 3.12.

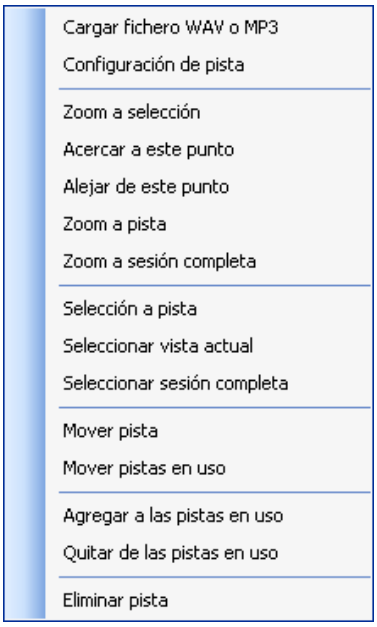

Figura 3.12.- Menú contextual de UCGraficaOnda

En el menú se pueden encontrar opciones para configuración de pista, control de zoom (zoom in, zoom out, etc), o bien se puede seleccionar de forma precisa la vista actual, la pista en cuestión o todas las pistas.

Es necesario explicar el concepto de "*pistas en uso*" antes de continuar con la descripción de las funcionalidades del menú. Cuando se trabaja con multitud de pistas, las pistas en uso son aquellas sobre las que se está ejerciendo alguna acción. Por ejemplo, cuando se hace clic sobre una pista para seleccionar una región, ésta pasa a ser una "pista en uso" directamente. Las pistas en uso van a tener un color más claro que las demás para poder diferenciarlas. Se puede además tener varias pistas en uso mediante las dos opciones del menú contextual. De esta forma ciertas acciones se pueden realizar sobre varias pistas simultáneamente.

Aparecen en el menú dos opciones interesantes: "Mover esta pista" y "Mover pistas en uso". En el subapartado anterior se estudió el control `UCBaseDeTiempos`. Se vio que haciendo clic y arrastrando sobre él se puede desplazar la representación gráfica a derecha o a izquierda. Sin embargo, no hay que confundir esta tarea con la que realizan las dos opciones del menú contextual mencionadas.

Con las opciones de desplazamiento del menú contextual se puede desplazar una pista con respecto a las otras. Esto puede ser de gran utilidad para colocar un determinado fichero de audio en la posición deseada sin mover los demás.

Es importante hacer hincapié en el hecho de que `UCGraficaOnda` no relaciona su interfaz superior con sus parámetros internos. Se necesita siempre de un objeto contenedor (`UCPista`) que realice estas funciones (ver Figura 3.9).

Implementación

La implementación de `UCGraficaOnda` ha sido una tarea laboriosa y compleja. Este control, debido a la cantidad de funcionalidades que tiene, y a la complejidad de su estructura, ha sido uno de los más dificultosos de todos.

`UCGraficaOnda` obtiene los datos de una variable de tipo `WaveStream` que se le pasa por parámetro. De esta forma, `UCGraficaOnda` se abstrae del fichero del que está leyendo los datos, simplemente usa la interfaz que proporciona `WaveStream`.

La implementación relativa a la representación gráfica está determinada por la estructura de capas explicada anteriormente. Cada capa es un buffer de tipo `System.Drawing.Bitmap`. Se ha asociado al evento `Paint` de `UCGraficaOnda` un método encargado de pintar en pantalla el control. Este método simplemente superpone en orden las distintas capas. Sin embargo, mucho más complejo es el mecanismo para actualizar las distintas capas en los momentos necesarios. De hecho, la actualización de las capas es una tarea repartida entre `UCGraficaOnda` y su contenedor: `UCPista`. Es importante comentar que cuando se actualiza un buffer relacionado a una determinada capa, no se está representando en pantalla. Simplemente, se queda actualizado para que en la próxima llamada al método de refresco aparezca correctamente.

A la hora de actualizar un buffer, se sigue casi siempre el mismo procedimiento: se actualizan los valores relativos a él, y después se crea el buffer correspondiente. Por ejemplo, si se modifica la región seleccionada, primero es necesario actualizar los valores relativos a la selección, y luego se crea el buffer gráfico asociado.

A continuación se describe cómo y cuando se actualiza cada capa.

- Color de fondo: El rectángulo relleno del color de fondo que servirá como sustrato se pinta justo antes de actualizar la rejilla. De hecho está dentro de la misma rutina para actualizar la rejilla.
- Rejilla gris: La actualización de la rejilla está a cargo de `UCPista`. Se aprovecha que `UCBaseDeTiempos` es capaz de recibir un buffer de tipo bitmap y pintar en él una rejilla acorde a sus divisiones. Siempre que se vaya a hacer algún cambio en la representación, `UCPista` actualizará antes la rejilla.

- Forma de onda: Este buffer corresponde actualizarlo a `UCGraficaOnda`. Siempre que hay un cambio en la representación, `UCGraficaOnda` actualiza este buffer. Se han hecho grandes esfuerzos por optimizar los métodos de actualización de esta capa. Más adelante se dedicará un apartado exclusivamente a explicar qué recursos se han utilizado para conseguir dicha optimización.
- Selección de regiones: El rectángulo semitransparente de color azul corresponde pintarlo a `UCGraficaOnda`. Se va a actualizar siempre que se realice un cambio sobre los límites de la selección.
- Indicador de pista en uso: Cuando la pista es una "pista en uso", con la intención de hacer más brillantes los colores se sitúa cuadro semitransparente blanco. Esto lo realiza `UCGraficaOnda` en el momento de pintar las capas si la variable booleana `m_PistaEnUso` está en trae.
- Cursor: El cursor es una línea vertical amarilla para indicar posiciones dentro del fichero. Para acelerar la actualización del mismo, el cursor no va a formar parte de la estructura de capas propiamente dicha. Cada vez que se actualizan las capas, siempre se coloca el cursor sobre todas ellas. Pero si solamente se ha variado el cursor de sitio, no se pintan de nuevo todas las capas. Simplemente se mueve la línea de sitio sustituyendo la antigua posición por el fondo que había anteriormente. Esto acelera los desplazamientos de cursor.

Antes se ha comentado que la actualización de la capa de forma de onda se realiza mediante métodos optimizados. A la hora de representar una forma de onda es necesario trabajar con gran cantidad de datos, por lo que será el lastre de todo el proceso de representación.

Se han usado dos recursos principalmente para optimizar la actualización del buffer de onda:

- Uso de "ficheros de picos"
- Desplazamientos horizontales optimizados

A continuación se describen ambos métodos.

Ficheros de picos

Para representar ficheros de onda en pantalla es necesario leer todas las muestras del mismo. El eje *Y* representa la amplitud de la muestra, y en el eje *X* la posición temporal de la misma. Hay que leer todas y cada una de ellas para poder conseguir la gráfica completa. Si las

muestras deben leerse del disco duro, el proceso sería demasiado lento. Por otro lado, si se almacena en la memoria RAM el fichero completo, existe el problema de que ésta se agotaría demasiado pronto.

Un fichero de 5 minutos en formato mono con una frecuencia de muestreo de 44100Hz contiene 13.230.000 muestras (unos 13.230.000 bytes si el fichero es de 8 bit - mono). Sin embargo los pixels de la pantalla son discretos, por lo que tanto el eje *X* como el *Y* habrá que discretizarlo. Imagínese una gráfica de 800 de largo por 150 pixels de alto. Si el fichero de 5 minutos se intenta representar en esta gráfica, en cada línea horizontal de 1x150 pixels entrarían 16537.5 muestras. El resultado de pintar estas 16537.5 con sus distintas amplitudes sobre el mismo píxel en el eje *X* va a ser una línea recta vertical. Si se observa de nuevo la Figura 3.13, se aprecia que la gráfica consiste en una serie de líneas verticales contiguas.

En la Figura 3.13 se ilustra la discretización de la gráfica al representarse en pantalla. Para representar la Figura 3.13.b, realmente sólo se necesitan 8 datos: la muestra mayor y menor de cada línea. Por tanto, no es necesario recurrir a todos los datos de la Figura 3.13.a.

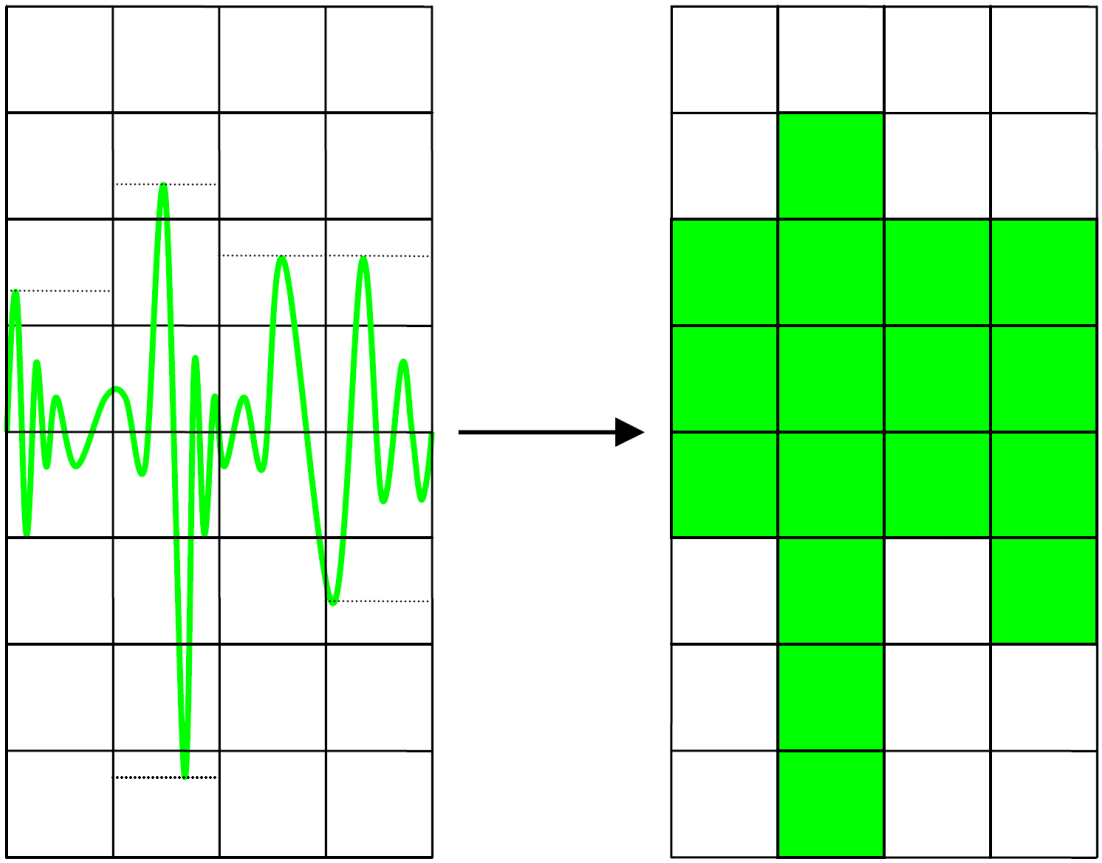

Figura 3.13.- Ejemplo de discretización de onda para representarla en pantalla

La conclusión es que para representar, por ejemplo, un fichero de 5 minutos sobre una gráfica de 800 píxeles de largo, no se necesitan tantos datos. Basta con tener la muestra mayor y la muestra menor de cada 16537.5 muestras, y utilizarlas para trazar una línea vertical. Por tanto, con 800 x 2 datos se podrían representar perfectamente la gráfica. Se ha reducido en un factor de 16537.5 / 2 el número datos.

Si se almacenasen en un fichero la muestra mayor y la muestra menor de cada *n* muestras, podría utilizarse para trazar la gráfica sin tener que recurrir a todos los datos. En esto consiste el fichero de picos. Por defecto se almacena la muestra mayor y menor de cada 256 muestras. Esto ofrece una reducción de datos en un factor de 128.

Por tanto, si en la representación hay más de 256 muestras en cada línea de grosor un píxel, se recurre al fichero de picos. Si por el contrario hay menos de 256 muestras por píxel se leerán los datos del fichero original.

Esta solución permite realizar acercamientos y alejamientos de manera rápida sin utilizar excesiva memoria RAM.

Desplazamientos horizontales optimizados

Cuando la gráfica se desplaza horizontalmente, gran parte de las muestras ya representadas se pueden reutilizar. La idea para optimizar los desplazamientos consiste en desplazar el buffer existente y leer del fichero sólo las muestras nuevas que aparecen. Por ejemplo, si el desplazamiento es hacia la derecha, por la izquierda aparecerán muestras nuevas. Si el desplazamiento es hacia la izquierda sucede al contrario.

Cuando se necesita actualizar el buffer de onda, y sólo ha habido un desplazamiento horizontal, se utiliza este método para ahorrar recursos. En la Figura 3.14 se ilustra este sistema.

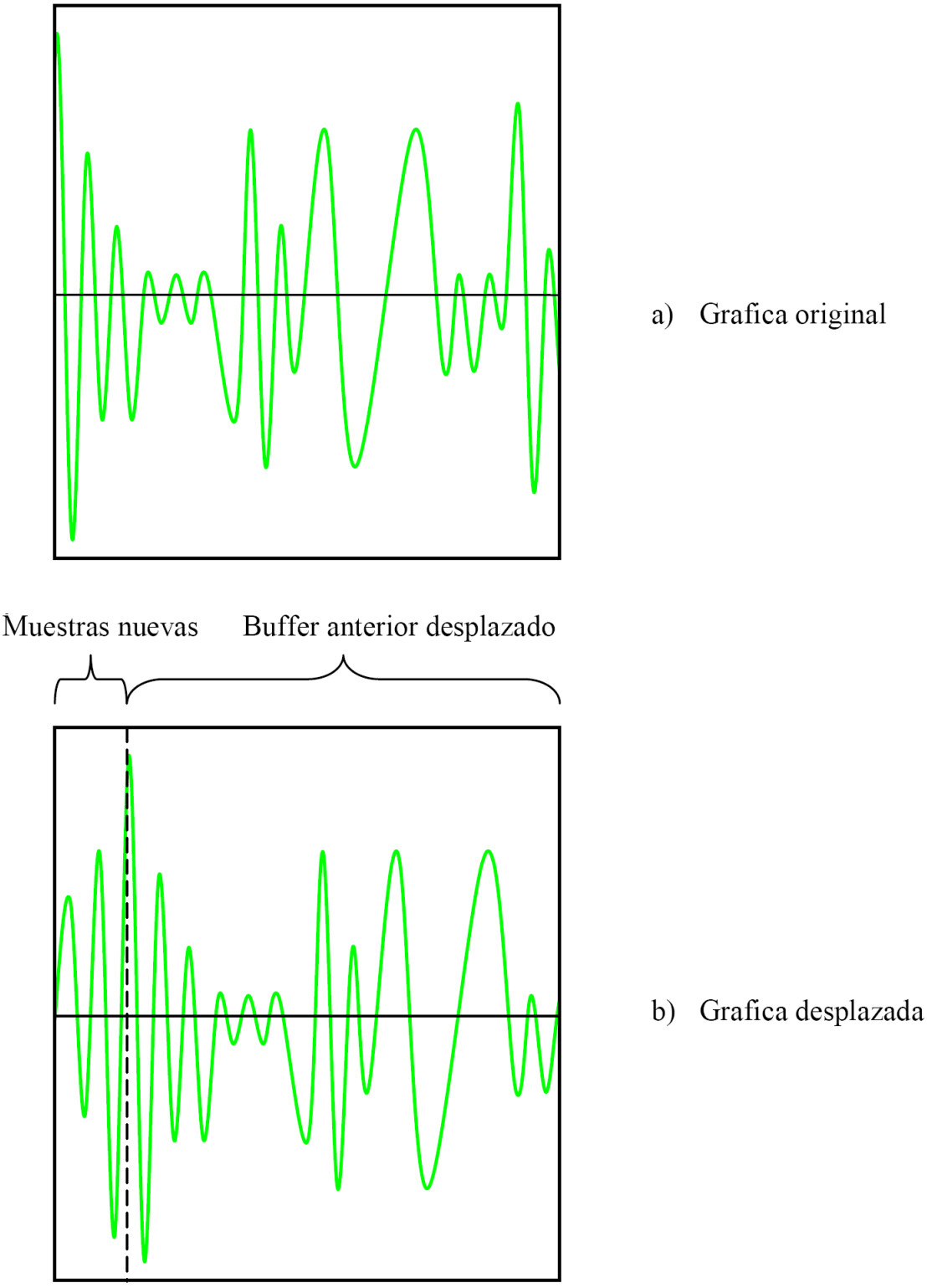

Figura 3.14.- Ejemplo de desplazamiento de gráfica

Interfaz inferior

El control `UCGraficaOnda` ofrece gran cantidad de métodos y propiedades públicas. Está diseñado para formar parte de un control mayor, por lo que muchos métodos y propiedades de forma aislada no tienen sentido.

Para poder representar un fichero de audio por pantalla, hay que pasarle por parámetro al control un `WaveStream`. Además es necesario indicarle el segundo de inicio y el segundo final de la representación. Hay un gran conjunto de métodos, propiedades y eventos asociados a las muestras que se representarán en pantalla.

Se dispone además de otro conjunto de miembros asociados al control de las regiones seleccionadas. Se puede obtener la selección actual, establecerla, recibir un evento si se produce algún cambio sobre ella, etc. Por otro lado, el control del cursor también dispone sus propios miembros. Se puede obtener o establecer su posición, determinar si se desea que no sea modificable, etc.

Por último, se disponen de miembros también para desplazar archivos a la posición deseada. En el código, todas las referencias a "desplazamientos absolutos" están relacionadas con esta acción.

El control UCPista

Este control va a servir como contenedor para los controles explicados anteriormente. Se encarga de sincronizarlos para que en todo momento estén correctamente actualizados.

El control `UCPista` está preparado para funcionar de forma independiente, o bien para estar contenido en un control contenedor. Por ello, `UCPista` está preparado para actuar según el esquema de la Figura 3.8, o el de la Figura 3.9. Si en el control contenedor se utilizan los eventos provenientes de `UCPista`, éste perderá su independencia. `UCPista` realiza tareas distintas si existe un contenedor de pistas que si no lo hay.

Con este sistema, si un panel contenedor contiene 8 pistas, un cambio sobre una pista puede afectar a las otras 7. Por ejemplo, al mover el cursor en una pista puede moverse en todas a la vez. Sin embargo, si se desea una aplicación monopista sin ningún tipo de contenedor, también puede usarse `UCPista` con algunos pequeños ajustes.

`UCPista` por lo tanto va a ser un control versátil, complejo, y con numerosas funcionalidades. Se comprenderá mejor su diseño cuando se explique `UCPanelDePistas`, pero antes se describirá de una forma más detallada `UCPista`.

Representación gráfica

Como unidad fundamental de trabajo, la pista debe ofrecer gran cantidad de funcionalidades al usuario. En la Figura 3.15 se puede observar el aspecto externo de la pista de audio. Cada pista incluye su propia gráfica de onda, su base de tiempos, su vúmetro y diversos controles de Windows Forms.

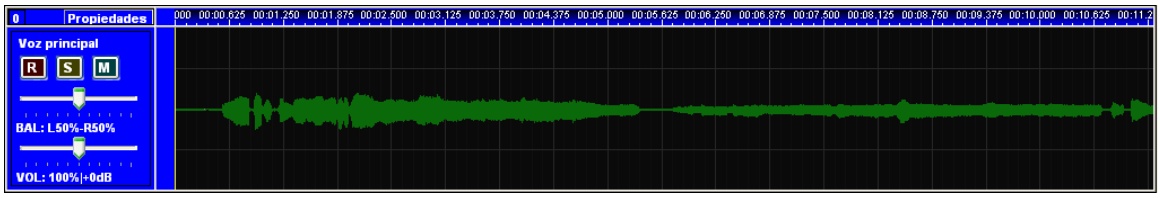

Figura 3.15.- Control UCPista

El volumen aparece indicado en tantos por ciento o en decibelios, para tener una doble referencia del mismo. El balance por otro lado simplemente indica qué tanto por ciento de señal va a cada canal. Los botones R, S y M significan Rec, Solo y Mute respectivamente. Tienen dos posibles estados: On y Off. Cuando un botón se encuentra en Off aparece en color oscuro, mientras que si se encuentra en On aparece con un color brillante.

Por otro lado, si se hace clic sobre el botón llamado “Propiedades”, se abre otra ventana desde la que se podrán controlar numerosos parámetros asociados a la pista de audio. Esta ventana se muestra en la Figura 3.17.

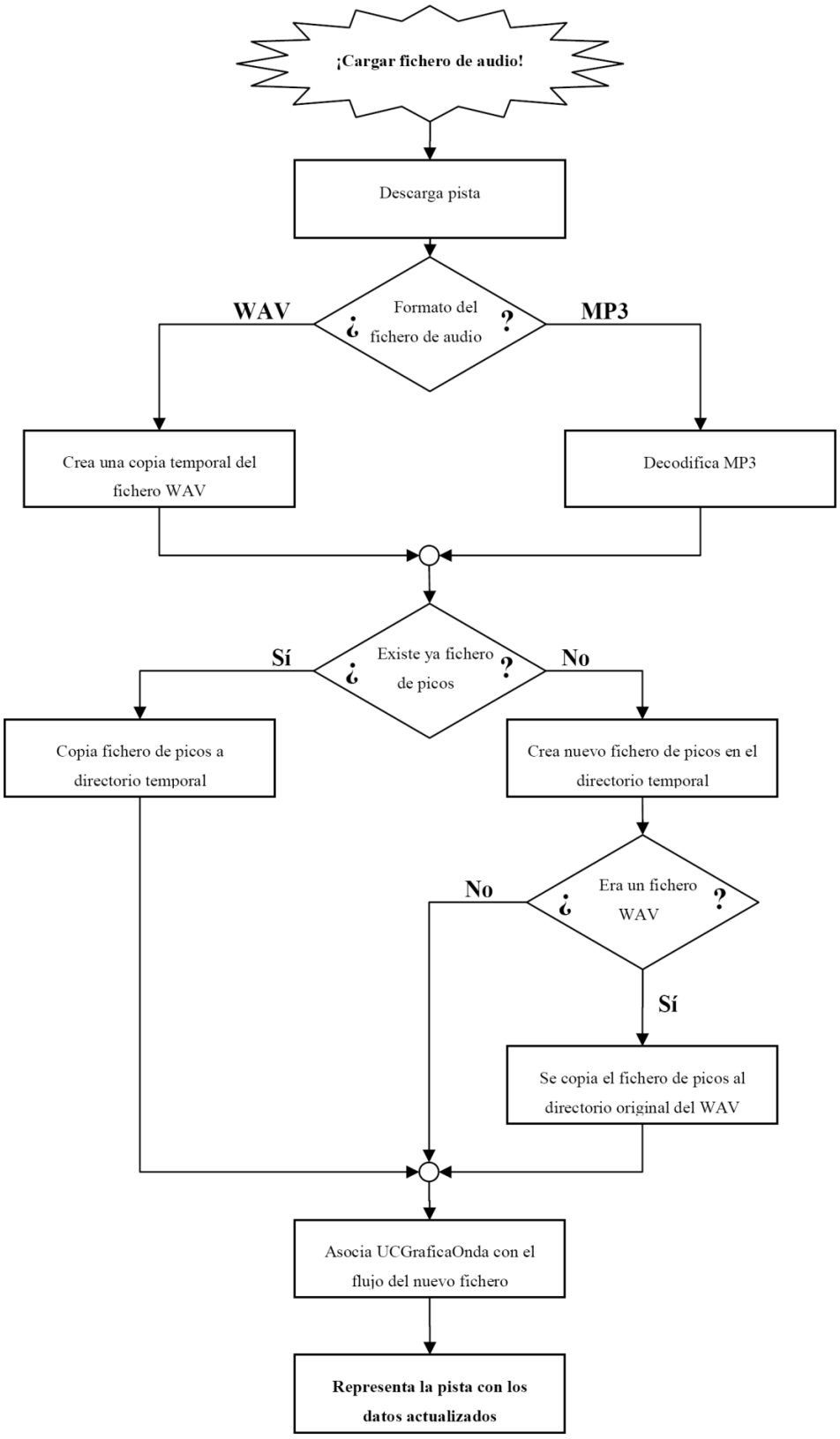

Figura 3.16.- Algoritmo de cargado de ficheros WAV ó MP3

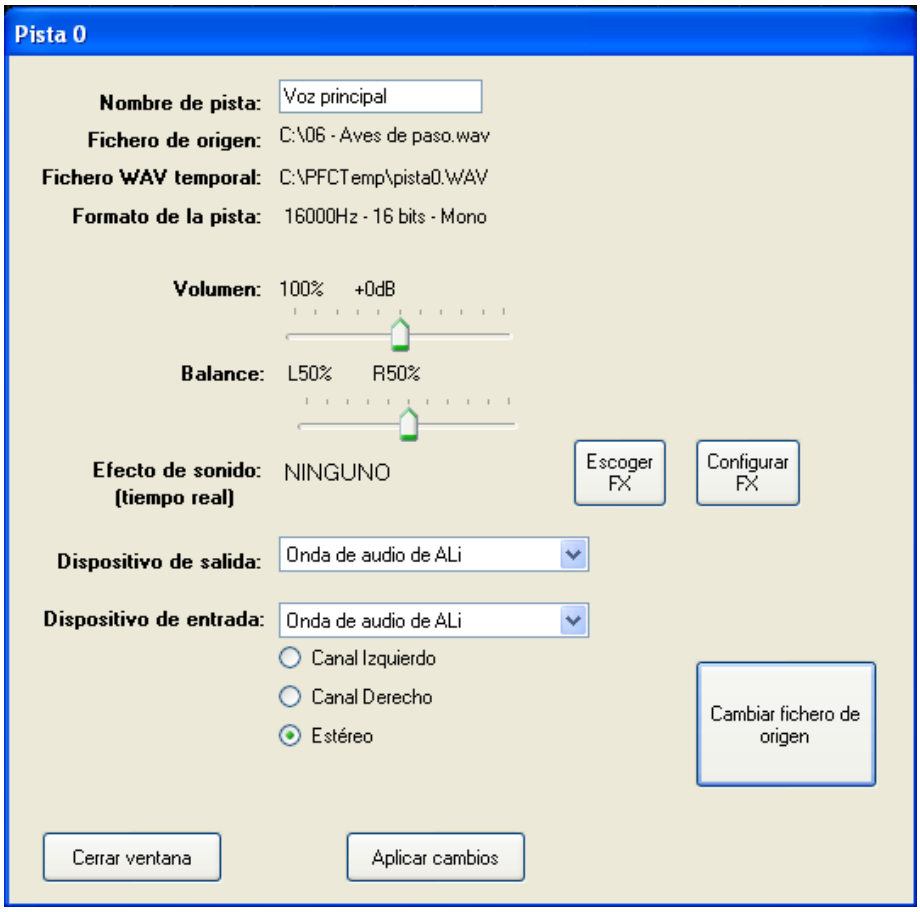

Figura 3.17.- Panel de propiedades de pista

Desde esta ventana se puede cargar un fichero de audio nuevo, cambiar los dispositivos de entrada y salida, escoger un efecto de sonido o manipular los controles de volumen y balance. Además, en esta ventana vienen resumidos todos los datos más significativos de la pista, e incluso algunos de ellos pueden modificarse. Cuando se pulsa sobre "Aplicar cambios", todas las propiedades se trasladan a la pista de audio directamente.

Interfaz superior

El usuario puede interaccionar con la pista de las siguientes formas:

- A través del control `UCGraficaOnda` (descrito anteriormente).
- A través del control `UCBaseDeTiempos` (descrito anteriormente).
- A través de los botones de SOLO, MUTE y REC. Cuando se pulsan pasan a estado ON hasta que se vuelvan a pulsar.

- A través de las barras de volumen y balance.
- Haciendo clic sobre cualquiera de sus componentes o sobre las zonas libres de controles para establecerla como “pista en uso”.
- Mediante la ventana de propiedades que aparece tras pulsar en el botón “Propiedades”.

Con todas estas funcionalidades, el usuario tiene el dominio de todos los parámetros modificables asociados a la pista.

Implementación

Cuando se crea la pista, se inicializan todos los parámetros de todos los componentes contenidos en ella. El usuario en ese momento obtiene una pista vacía, sin ningún archivo de audio asociado a ella.

Carga de un fichero WAV en la pista

Es entonces cuando el usuario debe cargar un fichero de audio a la pista. Al cargarse el fichero, si no existe un fichero de picos, `UCPista` quien lo genera. Además, `UCPista` se encarga de la gestión de archivos temporales. En el flujograma de la Figura 3.16 se observa qué hace `UCPista` cuando abre un fichero.

Para trabajar de forma más rápida, no se genera el fichero de picos cada vez que se abre el WAV. Se intenta almacenar el fichero de picos en el directorio original del WAV, para poder leerlo la siguiente vez que se use este fichero. Los ficheros van a tener una terminología estándar para que el programa los pueda encontrar de forma inmediata. La terminología es la siguiente:

- El fichero WAV que se almacenará en el directorio temporal tiene el nombre: Pista<ID>.WAV. Donde <ID> es el número entero identificador de pista. Por ejemplo, la pista 0 creará en el directorio temporal el fichero: Pista0.WAV.
- Los ficheros de picos siempre tienen el mismo nombre que los ficheros WAV, y están almacenados en el mismo directorio, solamente que tienen la extensión .PIK. De esta forma, en el directorio temporal se creará también una serie de ficheros del tipo: Pista<ID>.PIK.

Dado que los procesos de la Figura 3.16 son bastante largos, una barra de progreso indicará en todo momento qué tanto por cierto de operación se ha completado. Además, para no perder el contacto con la interfaz gráfica, estos procesos se realizarán en segundo plano mediante

un hilo (ver nota al pie número [2]). El cuadro de diálogo que incluye la barra de progreso dispondrá de un botón "Cancelar" que directamente terminará los procesos y dejará la pista sin ningún fichero cargado.

Componente versátil

Anteriormente se ha comentado que `UCPista` está diseñado para funcionar de forma autosuficiente, o integrado en un contenedor de pistas. Para conseguir esto se hace uso de los eventos. Si el evento es nulo, se entiende que no está contenido en un control contenedor de pistas. Por el contrario, si el evento no es nulo, `UCPista` va a entender que está contenido en un contenedor de pistas. Un ejemplo práctico puede ser el del Código 3.6, donde `ZoomHaCambiado` es un evento de `UCPista`.

```
if (ZoomHaCambiado != null)
{
    ZoomHaCambiado(this, ZoomIn(e.PosEnSegundos));
}
else
{
    ...
```

Código 3.6.-Ejemplo de utilización de evento sólo si éste está implementado en el contenedor

Por conseguir esto se usa una filosofía parecida al *polimorfismo*. En el polimorfismo, una clase heredada de otra puede reemplazar sus métodos. En este caso, un control que contiene a otro puede reemplazar sus métodos.

Cuando se explique `UCPanelDePistas`, se terminará de explicar cómo "reemplaza" los métodos de `UCPista`.

Sincronización de gráfica y base de tiempos

Se ha explicado que la base de tiempos y la gráfica de una misma pista están sincronizadas mediante `UCPista`. El esquema de la Figura 3.18 ilustra como consigue `UCPista` realizar esta tarea si no existe un panel de pistas que la albergue. Si la pista está integrada en `UCPanelDePistas` dejará que sea esté quien gestione los eventos (Figura 3.19).

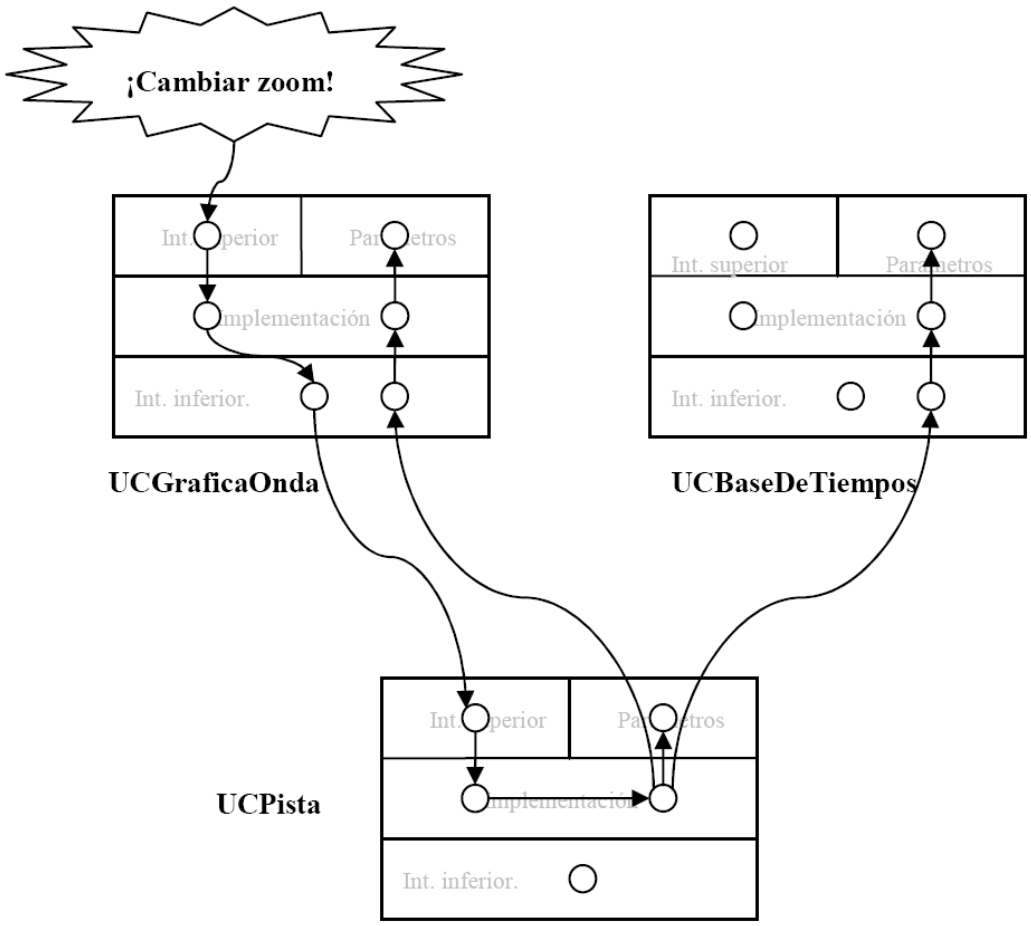

Figura 3.18.- Funcionamiento de UCPista si no existe un UCPanelDePistas que la contenga

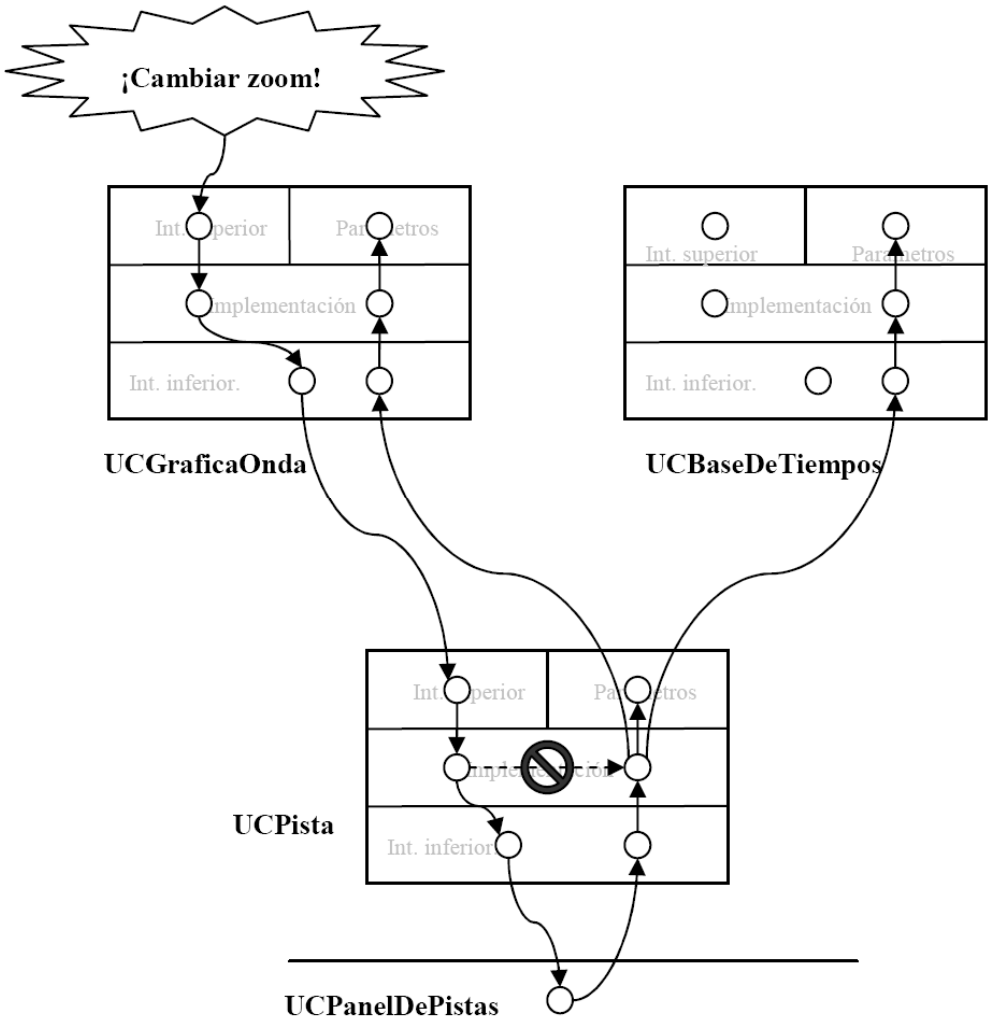

Figura 3.19.- Funcionamiento de UCPista cuando está contenida en un UCPanelDePistas

La interfaz superior es el conjunto de posibilidades que ofrece `UCPista` al usuario para interaccionar con él. Por tanto, las interfaces de `UCGraficaOnda` y `UCBaseDeTiempos` formarán parte de la interfaz superior de `UCPista`.

Cuando se realiza una acción sobre `UCGraficaOnda`, por ejemplo, `UCPista` se encarga de distribuir los cambios a los objetos necesarios. De esta forma, si se modifica el zoom de la representación, `UCGraficaOnda` y `UCBaseDeTiempos` sufrirán los cambios a la misma vez.

Estos son los tres aspectos más importantes de la implementación de `UCPista`, ya que el resto son detalles y ajustes para un funcionamiento correcto.

Interfaz inferior

`UCPista` ofrece métodos, propiedades y eventos públicos con distintas funcionalidades. En primer lugar, `UCPista` proporciona gran cantidad de miembros asociados al fichero WAV correspondiente a ella. Proporciona métodos y propiedades para cargar un fichero, leer del mismo, escribir en él, posicionar el índice de lectura y escritura, leer el formato del fichero… entre otros. Además se dispone de un evento para indicar que se ha cargado un nuevo fichero en la pista. Estos miembros ofrecen toda la funcionalidad necesaria para el uso de ficheros WAV en la pista.

Por otro lado, existen miembros asociados a la representación sobre `UCGraficaOnda` y `UCBaseDeTiempos`. Proporciona métodos para establecer el zoom, actualizar la representación, establecer la región seleccionada, posicionar el cursor, establecer los colores, etc. Por otro lado existen multitud de eventos para indicar cambios en la interfaz. Todos estos miembros van a permitir el control preciso de la representación gráfica en cada instante.

Es importante destacar que a la hora de definir la representación siempre se usan valores de tiempo, nunca muestras. Esto es así para abstraer la representación del formato, ya que independientemente de las muestras por segundo del fichero, siempre se representará el mismo intervalo de tiempo.

Relacionados con otros parámetros como el volumen, el balance, o los flags (SOLO, MUTE y REC) hay otros miembros públicos de interés. Se disponen de propiedades que permiten obtener o establecer todos estos valores. El volumen y el balance toman valores del 0 al 1, almacenados en variables de tipo `double`. Si la pista se encuentra en MUTE, o existe otra pista en SOLO, el volumen inmediatamente se torna al valor 0.

El control UCPanelDePistas

El control `UCPanelDePistas` es el contenedor que contendrá las pistas y proporcionará todas las funcionalidades para la reproducción y grabación en ellas. Se puede decir que `UCPanelDePistas` en sí ya cumple los objetivos propuestos para el software. Este `UCPanelDePistas` irá incluido en un formulario que servirá únicamente como interfaz. En el formulario no hay implementación más allá de llamadas a métodos de `UCPanelDePistas`.

Representación gráfica

Las pistas se colocarán en horizontal, una debajo de otra. Si el número de pistas es demasiado grande para abarcarlo en pantalla, se utilizará una barra de desplazamiento vertical. En la Figura 3.20 se muestra un ejemplo de `UCPanelDePistas` en pantalla.

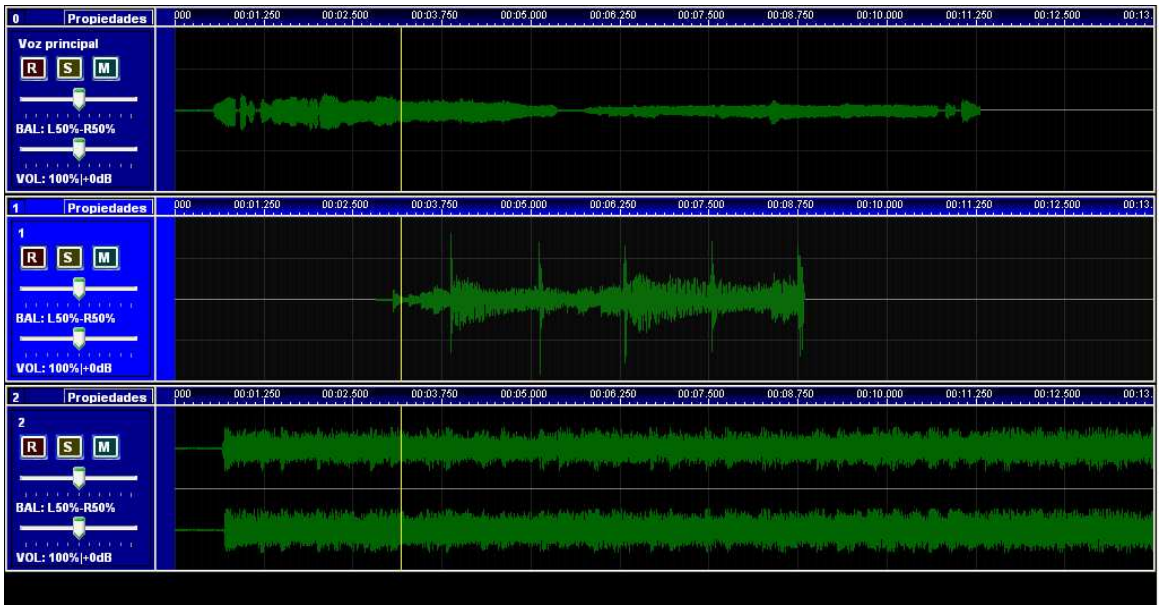

Figura 3.20.- Aspecto visual de UCPanelDePistas

El ancho de las pistas se ajustará en todo momento al de `UCPanelDePistas`. El alto de las pistas se puede controlar de forma individual. Los cambios sobre una pista se reflejarán en el resto de pistas para tenerlas alineadas en todo momento. Si el panel está vacío, aparecerá como un rectángulo negro únicamente.

Interfaz superior

El control `UCPanelDePistas` directamente no ofrece ningún tipo de posibilidad al usuario para controlarlo directamente. Se podría decir que la interfaz superior del panel es el conjunto de interfaces superiores de todas las pistas. Para controlar el `UCPanelDePistas`, siempre se recurren a barras de herramientas o menús externos.

Una vez que las pistas están integradas en el panel, los cambios en una pista se reflejan en todas las demás. Visto desde ese punto de vista, la interfaz de `UCPanelDePistas` está constituida por todos los elementos contenidos en él.

Externos al panel deben existir controles que permitan la utilización del mismo:

- Control de transporte: Control que permita iniciar reproducción o grabación, detener, pausar, rebobinar, etc.
- Control de zoom: Control que permita realizar distintos tipos de zoom sin necesidad de recurrir al menú contextual de cada pista.

- Control de cursor y de selección: Control que permita situar el cursor con precisión en la posición indicada. También debe ser capaz de situar los límites de la región seleccionada con precisión.
- Control para la gestión de pistas: Control que debe permitir agregar pista, insertar pista, eliminar pista, o cambiar el orden de las mismas.

Todos estos controles no están integrados en `UCPanelDePistas`, por lo que tendrán que implementarse en el formulario que lo contenga.

Implementación

Lo más interesante de `UCPanelDePistas` es la manera en la que gestiona todas las pistas y accede a ellas. Con la Figura 3.19 se explicó que dicho esquema para los controles de usuario permitía que el control contenedor gestione sus parámetros internos. `UCPanelDePistas` será en última instancia quien "gobierne" a todos los demás, es el auténtico administrador de todos los cambios.

`UCPanelDePistas` gestiona los eventos procedentes de las pistas indicando que se pretenden realizar cambios, y actúa en consecuencia. Por ejemplo, en una pista se decide cambiar el cursor de sitio, simplemente se hará clic simple sobre `UCGraficaOnda` en la posición deseada. El usuario percibe que inmediatamente el cursor ha cambiado de sitio, pero realmente ha sucedido lo siguiente:

1. `UCGraficaOnda` recibe un evento de clic.
2. `UCGraficaOnda` lo gestiona y provoca un evento de cambio de cursor.
3. `UCPista` recibe el evento de cambio de cursor y lo gestiona. Al estar contenida en un panel de pistas no realiza ningún cambio, sólo provoca un evento de cambio de cursor.
4. `UCPanelDePistas` recibe el evento de cambio de cursor de una determinada pista. En este momento, el panel ordena a cada pista situar el cursor en la posición deseada.
5. Cada `UCPista` recibe la orden necesaria para cambiar el cursor, y ordena a `UCGraficaOnda` posicionar y representar el cursor en su posición adecuada.

Se van recorriendo los controles hasta llegar a `UCPanelDePistas`, que distribuye las órdenes a todas las pistas. De esta forma se centraliza la gestión de eventos y llamadas a métodos,

facilitando la programación y ofreciendo versatilidad. Si no existiera `UCPanelDePistas`, sería `UCPista` quien cambiaría directamente el cursor. En eso consiste la doble funcionalidad de `UCPista`, en la capacidad para ser o no autosuficiente.

El esquema de la Figura 3.21 representa a grosso modo cómo funciona el sistema de "distribución de órdenes". Este sistema es el mismo para todo el resto de cambios que se pueden producir en una pista. Siempre baja el evento hasta `UCPanelDePistas`, y éste realiza las acciones necesarias sobre cada una de las pistas.

Pero además, `UCPanelDePistas` va a ser el encargado de la reproducción y grabación de audio. `UCPanelDePistas` contiene una instancia de la clase `CMotorDeAudio`, y través de ella se producen todas las comunicaciones con los dispositivos de audio. Esta clase se explicará con detalle en el siguiente capítulo. En `UCPanelDePistas` no hay implementación relacionada con la reproducción y la grabación, ya que únicamente usa los métodos que ofrece `CMotorDeAudio`.

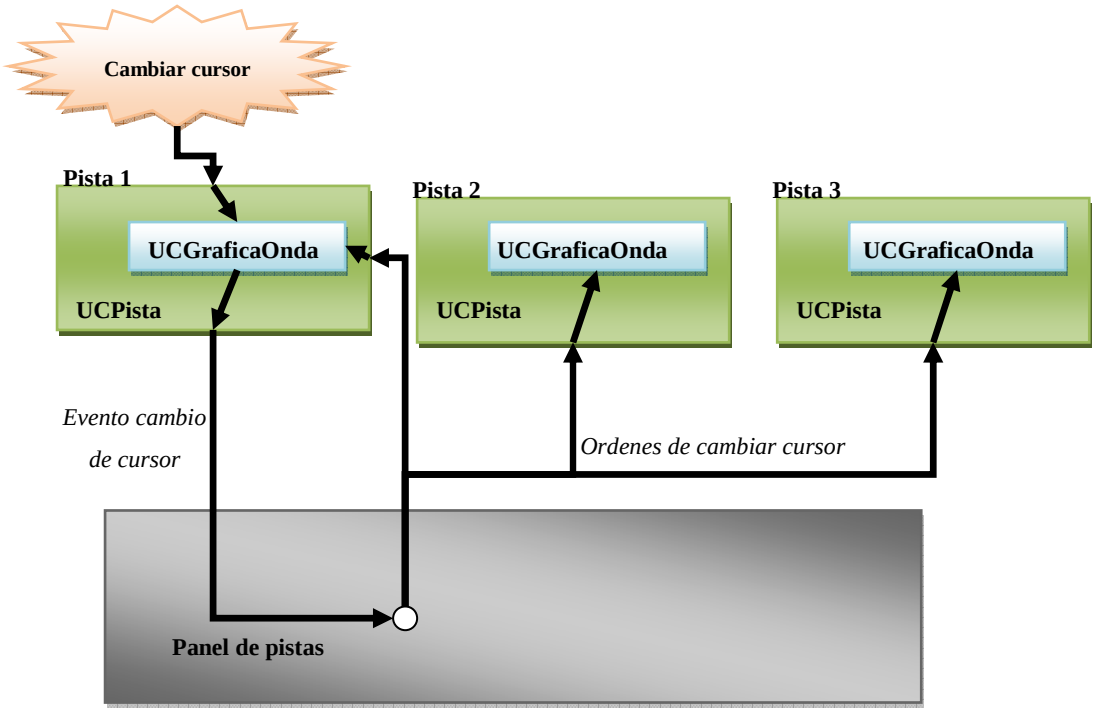

Figura 3.21.- Arquitectura basada en la centralización de órdenes sobre UCPanelDePistas

Si se elimina o se inserta una pista, los IDs de cada pista van a variar, y por tanto es necesario recurrir al fichero WAV adecuado del directorio temporal. Por ello, cuando se modifica el IDs de las pistas, se renombran los ficheros WAV para que cada una siga asociado al fichero correcto.

Interfaz inferior

`UCPanelDePistas` ofrece al formulario métodos y propiedades para la gestión de pistas, para la representación de todas las pistas y para la reproducción-grabación principalmente. Por otro lado, con la intención de que el formulario principal pueda mostrar por la barra de tareas el estado de `UCPanelDePistas`, se disponen de eventos para indicar cambios en el estado de éste. El uso de `UCPanelDePistas` desde el formulario es muy intuitivo, y por ello no se ha realizado una explicación detallada.

Conclusiones asociadas a la interfaz gráfica de la aplicación

El sistema expuesto anteriormente para crear la interfaz gráfica, aunque tiene ciertos inconvenientes, dispone sobre todo de numerosas ventajas.

En primer lugar, permite la representación de ficheros con distinta frecuencia de muestreo de forma alineada. Esto es así porque las órdenes para definir la representación se dan en segundos, no en muestras. Si se definiera el intervalo de representación en muestras, sucedería que dos ficheros con distinta frecuencia de muestreo estarían representando distinto intervalo de tiempo. Esto es importante para poder usar con facilidad ficheros con distintas frecuencias de muestreo en la misma sesión.

Por otro lado, puesto que `UCPanelDePistas` controla todos los eventos, el flujo del programa se simplifica enormemente. Para añadir un componente nuevo simplemente hay que tenerlo en cuenta en `UCPanelDePistas` y se integrará a la perfección con el resto de la interfaz.

Sin embargo, el uso de GDI+ como librería gráfica para el pintado de los controles implica una desventaja: es lenta. Si en vez de GDI+ se hubiese utilizado OpenGL, o DirectX la tarea de pintado se agilizaría. No obstante, esta lentitud no impide el correcto uso del software, y además es muy sencilla de programar, por lo que no se planteó usar otra librería.

3.8.- Mezclador y distribuidor

Una vez que se disponen de clases para la reproducción de audio y de un entorno basado en pistas es necesario conectar ambas cosas. Para ello se ha creado la clase `CMotorDeAudio`. Esta clase será la encargada de reproducir y grabar en `UCPanelDePistas`. Además, también se encargará de gestionar la monitorización: cambio de cursor a la posición de reproducción, desplazamiento de la representación, etc.

Esta clase va a disponer de dos partes bien diferenciadas: el mezclador y el distribuidor. El primero se encargará de la reproducción y la segunda de la grabación, y ambos realizan funciones completamente inversas.

Mezcla a tiempo real

En el apartado 3.6 se describió la clase `WaveReproductor` y las clases auxiliares de ésta. Esta clase recurría a un delegado cuando necesitaba datos para la reproducción, esperando que otra clase le entregara datos para reproducir. En este caso, los datos van a ser la suma de todas las pistas.

La clase `CMotorDeAudio` contiene la rutina que entregará los datos a `WaveReproductor` durante la reproducción. En esta rutina se leerán los datos de cada pista, se le aplicarán efectos, se sumarán, y se convertirán a un array de bytes listo para ser reproducido. De esta forma se estarán mezclando y reproduciéndose de forma simultánea, es decir, a tiempo real.

La reproducción de las pistas será independiente del formato de las mismas. Es decir, cada pista puede tener distinta frecuencia de muestreo, distinto número de bits por muestra y ser mono o estéreo. Más adelante se verá que la forma en la que realiza la mezcla hace que el número de bits por muestra y el número de canales no influya en absoluto.

Consideraciones generales de la mezcla

Formato de salida

Antes de comenzar a mezclar, es necesario establecer un formato final para la reproducción. Este formato será el que usará `WaveReproductor` en todos los dispositivos de salida para reproducir los datos. El objetivo de todos los procesos de mezcla es obtener un array de bytes en el formato de salida establecido.

Una característica de la que dispone este software y no disponen otros es de la capacidad de escoger totalmente el formato de salida. Por tanto se podrá modificar la frecuencia de muestreo, el número de bits por muestra y el número de canales. Este formato de salida, como ya se ha dicho, es completamente independiente al formato de las pistas individuales.

Lectura de datos de las pistas

Cuando se leen datos de una pista, estos se ofrecen como un array de bytes. Este array de bytes puede contener audio mono o estéreo, a 8 o a 16 bit, y con cualquier frecuencia de muestreo. Dependiendo del formato de la pista, este array tendrá una longitud distinta.

El primer paso es aislar las muestras de audio de este array de bytes. El resultado se almacenará en dos arrays de tipo `double[]`: Uno para el canal izquierdo y otro para el canal derecho. Si el fichero es mono ambos arrays contendrán la misma información.

En consecuencia, dependiendo del formato de la pista, estos arrays de doubles tendrán más o menos longitud.

Mezcla y conversión de formatos

Tras realizar este paso, se disponen de una serie de arrays de doubles de distintos tamaños. Estos arrays de tipo `double[]` se deben mezclar adecuadamente, para formar un par de arrays de doubles resultado de la mezcla total. El último paso de la mezcla será transformar este par de arrays de doubles a un array de bytes en el formato determinado, para entregarlo a `WaveReproductor` y reproducirlo.

El tamaño del array de bytes final es algo que viene establecido por `WaveReproductor`. En consecuencia también estará preestablecido el número de muestras del array de doubles resultante de la mezcla.

A la hora de mezclar, es necesario homogeneizar el tamaño del buffer en todas las pistas para poder sumar muestra a muestra. Para ello se aplicará un cambio en la frecuencia de muestreo en cada pista de forma independiente. El algoritmo para cambiar la frecuencia de muestreo se puede escoger para buscar un compromiso entre calidad y cómputo necesario. Más adelante se detallarán las opciones disponibles y se comentarán las ventajas e inconvenientes de cada una.

Este sistema permite que la frecuencia de muestreo de cada pista sea algo independiente del mezclado. En consecuencia, se pueden combinar datos provenientes de un CD (44100Hz) con datos grabados del hardware creado en este proyecto (16000Hz) sin problemas.

A continuación se describe el algoritmo de filtrado y todas las herramientas auxiliares que permiten realizar los pasos intermedios.

Algoritmo de mezclado

En la Figura 3.22 se muestra el flujograma de la rutina de mezclado. Esta rutina se ejecutará cada vez que `WaveReproductor` necesite cargar datos nuevos. Para llevar a cabo todas estas operaciones se han realizado diversas funciones. En el siguiente apartado se describen las más importantes de todas ellas.

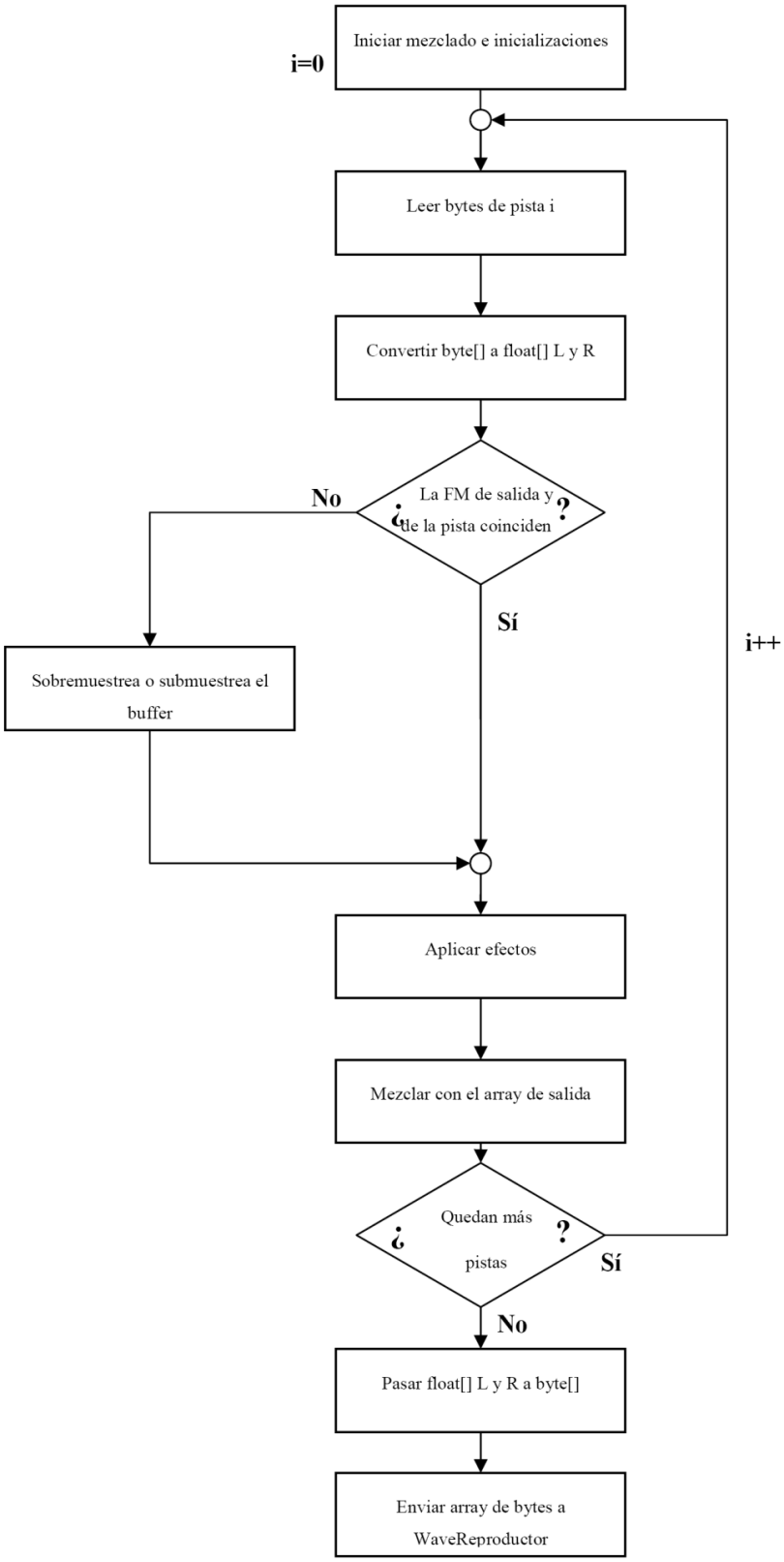

Figura 3.22.- Algoritmo de mezclado

Funciones y procedimientos auxiliares

Conversión byte[]→ double[] y double[] → byte[]

Los arrays de bytes van codificados en formato PCM (sin compresión). En este formato los datos de audio van organizados de la siguiente manera:

- Una muestra de 8 bits tiene el *00* (0) como pico mínimo y el *FF* (255) como pico máximo. El valor *10* (128) por tanto equivaldría al valor medio de amplitud 0.
- Una muestra de 16 bits se almacena en complemento a 2 en sistema little-endian. El valor máximo será *FF 7F* (32767), y el valor mínimo será *00 80* (32768). El valor *00 00* corresponde a una amplitud de 0.
- Si el fichero es mono, una muestra va colocada detrás de otra sin ningún tipo de separador.
- Si el fichero es estéreo, las muestras van alternándose. Así, las muestras impares corresponderán al canal izquierdo, y las muestras pares al canal derecho.

Los valores de tipo `double` estarán comprendidos entre 0 y 1. Teniendo en cuenta esto, convertir de bytes a double, o viceversa consiste simplemente en realizar sencillas operaciones. Solamente es necesario tener en cuenta que el byte alto y bajo están invertidos con respecto al orden natural de escritura (little-endian). La Figura 3.23 muestra un ejemplo de transformación de array de bytes a doubles en un formato de 16 bit – Estéreo. Si se desea convertir un par de arrays de doubles a un array de bytes, simplemente habría que realizar el proceso inverso.

Las conversiones para cada formato requieren un algoritmo distinto, por ello estas rutinas de conversión van a resultar bastante extensas. El array de bytes siempre va a tener un número de elementos mayor o igual que los doubles, como se muestra en la Figura 3.23.

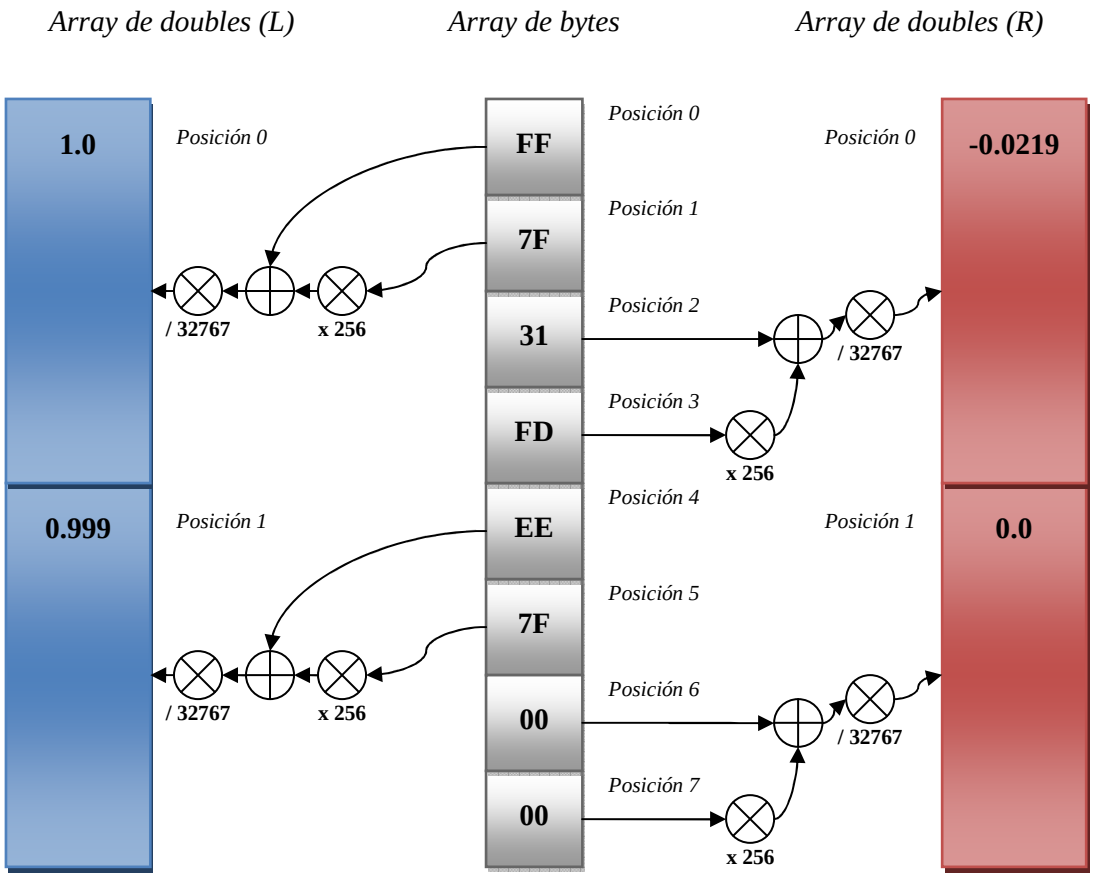

Figura 3.23.- Conversión de array de bytes en formato PCM a dos arrays de muestras

Filtrado y aplicación de efectos

En el apartado 3.9 se describe con detalle cómo se implementan los efectos de sonido y cuales han sido implementados ya en el desarrollo. Cada efecto de sonido va a estar implementado en una clase distinta y va a proporcionar una interfaz para su correcta aplicación. En la clase `CMotorDeAudio`, no se implementa ningún tipo de efecto, simplemente se utilizan los métodos públicos de los efectos ya creados.

Estos efectos se almacenan en una variable de tipo `CEfectoGenerico` dentro de cada pista de forma individual. La función que utiliza `CMotorDeAudio` simplemente aplica el efecto asociado a una determinada pista al par de arrays de doubles.

Mezclado de pistas

A la hora de mezclar, se irán sumando sobre un mismo par de arrays de doubles todos los arrays de doubles de cada pista, muestra a muestra. Estos datos además estarán afectados por el volumen y el balance de la pista en cuestión.

Los datos irán multiplicados directamente por el volumen. Esta magnitud puede variar entre 0 (sin sonido) y 10 (sonido al 1000%).

Con el balance el cálculo es un poco distinto. El balance toma valores entre 0 y 1, donde 0 significa balance total al canal izquierdo, y 1 balance total al canal derecho. Por tanto, el valor de balance podría representar el factor por el que multiplicar el canal derecho, y el valor (1 – balance) el factor por el que multiplicar el canal izquierdo. Sin embargo, si se hace así, la sensación sonora aumenta cuando se balancea totalmente a un canal. Para evitar esto, el valor nunca podrá superar la unidad. La operación para calcular el factor de multiplicación asociado al volumen y al balance sería la del Código 3.7.

```
double FactorL = (Pista.Volumen*Math.Min(1-Pista.Balance, 0.5))*2;
double FactorR = (Pista.Volumen*Math.Min(Pista.Balance, 0.5))*2;
```

Código 3.7.- Cálculo del factor de multiplicación a partir del volumen y el balance

Los arrays de doubles de cada pista, para mezclarlos, sencillamente hay que sumarlos. Por ello hay que cuidar que dos pistas no estén próximas al 100% de amplitud, ya que al mezclarse saturarán. En un principio, al mezclarse las pistas se realizaba la media de todas ellas, pero pronto se llegó a la conclusión de que este método de mezcla no era el adecuado.

El problema del cambio de frecuencia de muestreo a tiempo real

Como ya se ha comentado en puntos anteriores, el software es capaz de realizar una interpolación de pistas a tiempo real para reproducir y registrar pistas de diferente frecuencia de muestreo a tiempo real. Para conseguir esto es necesario implementar métodos que consigan interpolar o diezmar la señal a tiempo real.

Este conjunto de métodos encargados de cambiar la frecuencia de muestreo a tiempo real tiene especial importancia dentro de las funciones auxiliares del mezclado. Es un punto clave a la hora de obtener una buena calidad en la mezcla final, y además está relacionado con temas importantes de procesado de señal.

Durante la carrera, el sistema estudiado para la conversión de frecuencias de muestreo consistía en interpolar y diezmar con los factores adecuados. Por ejemplo, una señal muestreada a

12Khz que se desea transformar a 9Khz necesita ser interpolada idealmente en un factor de 3 y posteriormente diezmada en un factor de 4. De esta forma, la conversión se estará realizando correctamente sin pérdida de calidad.

Con el audio digital habitual, este sistema tiene un grave problema: Las relaciones entre frecuencias de muestreo no suelen ser sencillas. Por ejemplo, una conversión de 44100Hz a 48000Hz, bastante habitual, requiere una interpolación con un factor de 160 y posteriormente un diezmado con un factor de 147. De esta forma, 44100 · 160 / 147 = 48000Hz. Este problema causa que se requieran arrays intermedios de tamaños demasiado grandes (44100 · 160 = 7056000 muestras), y además la velocidad de cálculo se reduzca enormemente.

Por tanto, es necesario buscar métodos de interpolación más rápidos y eficaces, pero que no deterioren la señal y tengan una buena base teórica. Son tres los algoritmos que se han desarrollado con este propósito, dos de ellos aplicables a tiempo real:

- Interpolación basada en retenedor de orden cero (tiempo real)
- Interpolación basada en filtrado paso-bajo en el dominio del tiempo (tiempo real)
- Interpolación basada en función ‘resample’ de MATLAB

Interpolación a tiempo real basada en retenedor de orden cero

El sistema retenedor de orden cero

El retenedor de orden cero es un sistema que sirve para convertir una señal discreta en una señal continua, es decir, es un *reconstructor*. El sistema muestrea la señal x(t) en un instante determinado, y retiene el valor hasta el siguiente instante en el cual se toma una muestra. En la Figura 3.24 se muestra un ejemplo de reconstrucción con este sistema de una señal en el dominio del tiempo.

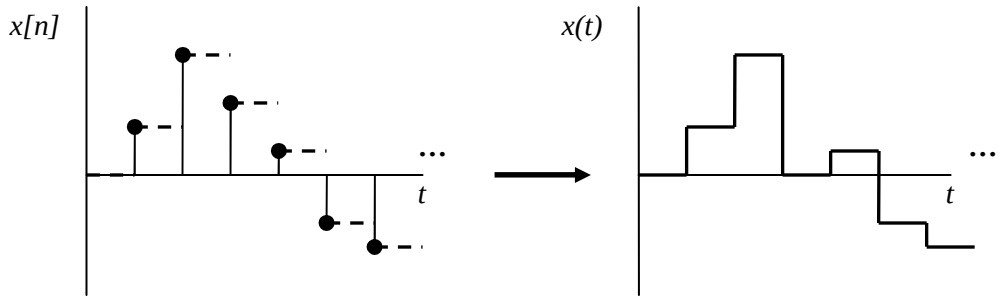

Figura 3.24.- Reconstrucción de señal mediante un retenedor de orden cero

Si se realiza un estudio del módulo la respuesta en frecuencia del retenedor de orden cero, se verá que se trata de una sinc con todos los lóbulos, es decir, susceptible de introducir infinitas componentes frecuenciales. En la Ecuación 3.1 se muestra la respuesta en frecuencia analíticamente.

$$H_0(j\omega) = e^{-j\omega T/2}\left[\frac{2\sin(\omega T/2)}{\omega}\right]$$

Ecuación 3.1

En la Figura 3.25 se ha incluido la respuesta en frecuencia de forma gráfica, superpuesta con el espectro de una sinusoide discreta a reconstruir, para entender la distorsión que se introduce. Se ha recuadrado en esta figura la banda de paso del filtro reconstructor ideal para que se realice una comparación.

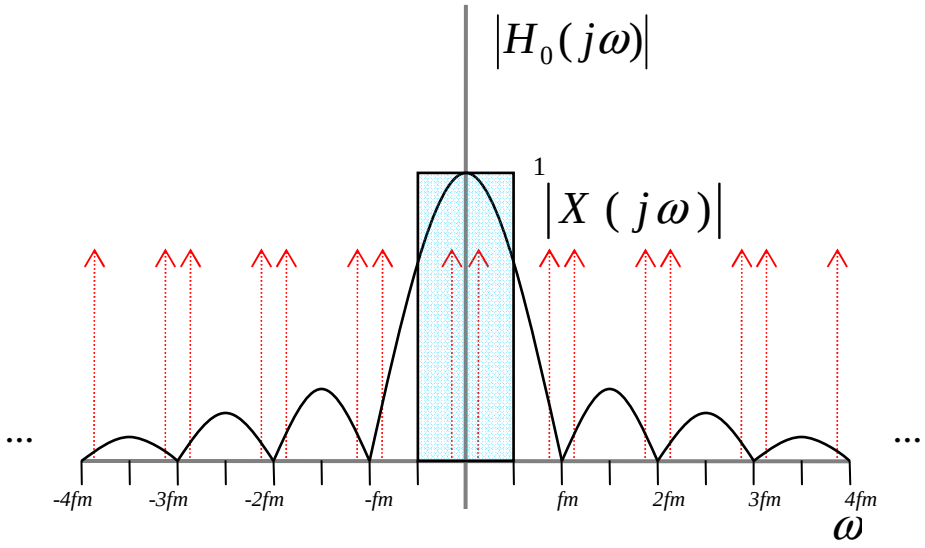

Figura 3.25.- Respuesta en frecuencia del retenedor de orden cero

Se puede entender por tanto, viendo las figuras descritas, que la señal continua resultante de la reconstrucción va a estar muy distorsionada. Las réplicas debido a la discretización de la señal no se van a filtrar correctamente, introduciéndose como distorsión.

Muestreo de la señal continua resultante con una frecuencia de muestreo distinta

La base del algoritmo consiste en realizar una conversión de *x[n]* a *x(t)*, para después muestrear esta señal continua con una frecuencia de muestreo distinta y generar *y[n]*. La Ecuación 3.2 describe el muestreo de una señal analíticamente.

$$y[n] = x\left(\frac{n}{f'_m}\right)$$

Ecuación 3.2

La Figura 3.26 ilustra en qué consiste este proceso gráficamente. Figura 3.27 se ilustra un ejemplo similar, pero con una frecuencia de muestreo mayor que la original.

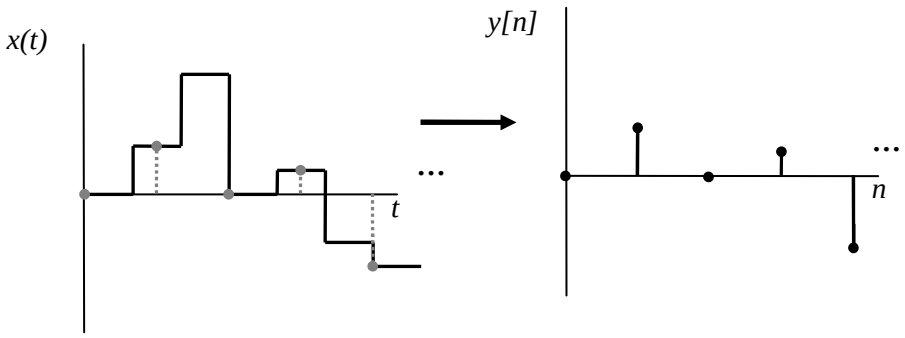

Figura 3.26.- Muestreo de la señal reconstruida con una frecuencia de muestreo menor que la original

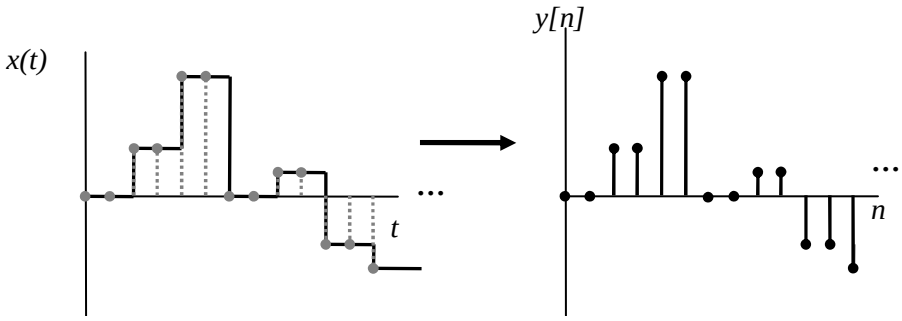

Figura 3.27.- Muestreo de la señal reconstruida con una frecuencia de muestreo mayor que la original

Si se analiza frecuencialmente el efecto de este muestreo, se entenderá rápidamente la cantidad de distorsión que se introduce debido al aliasing y a las réplicas anteriores no filtradas en la reconstrucción.

Debido a las infinitas réplicas no filtradas de la señal x(t), no existe una componente armónica máxima. Por tanto, cualquier frecuencia de muestreo que se escoja estará por debajo de la frecuencia de Nyquist, que es infinita, y en consecuencia siempre se producirá aliasing. Además, en el caso de la Figura 3.26, el aliasing será más pronunciado porque no sólo va a incluir réplicas anteriores atenuadas al espectro resultante, sino que también puede incluir réplicas nuevas de los armónicos principales. Esto es así debido a que la frecuencia de muestreo nueva no llega siquiera a la frecuencia de Nyquist necesaria para muestrear la señal original.

Pero sea mayor o menor la frecuencia de muestreo nueva, si se analiza el espectro de y[n], se observará que aparece muy distorsionada y con cantidad de componentes frecuenciales no deseadas en ambos casos. En la Figura 3.28 se muestra el espectro de un tono puro de 440Hz muestreado a 44100Hz. Tras reconstruir la señal con un retenedor de orden cero, y remuestrear la señal con una frecuencia de muestreo distinta, se han obtenido las señales cuyos espectros se muestran en las Figura 3.29, Figura 3.30 y Figura 3.31. Conviene hacer notar que el eje de frecuencias se ha normalizado para que la mitad de la frecuencia de muestreo coincida con π.

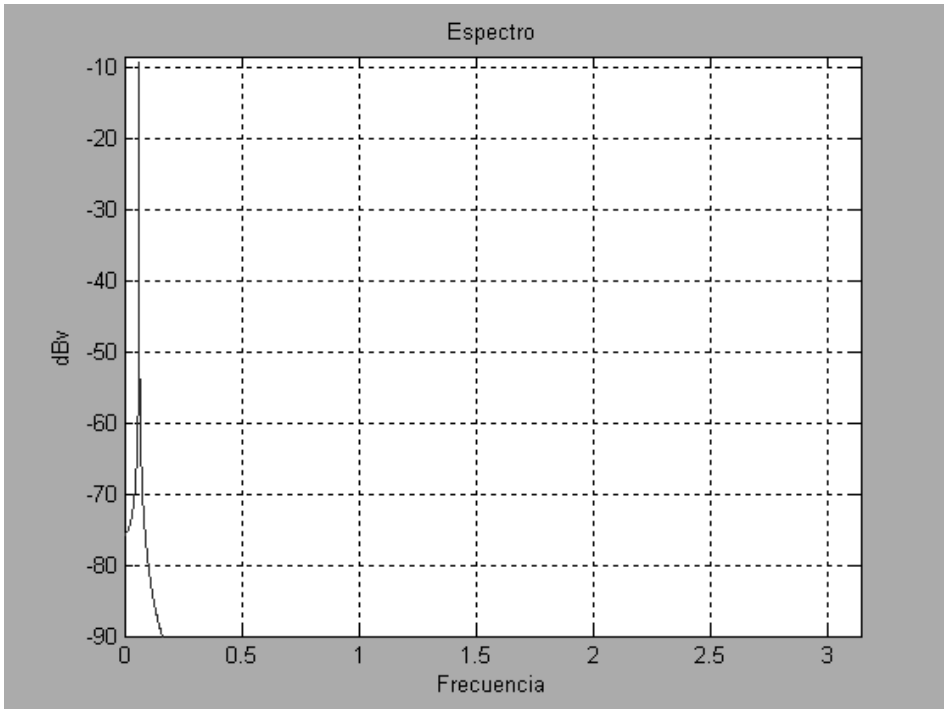

Figura 3.28.- Espectro de un tono puro de 440Hz muestreado a 44100Hz

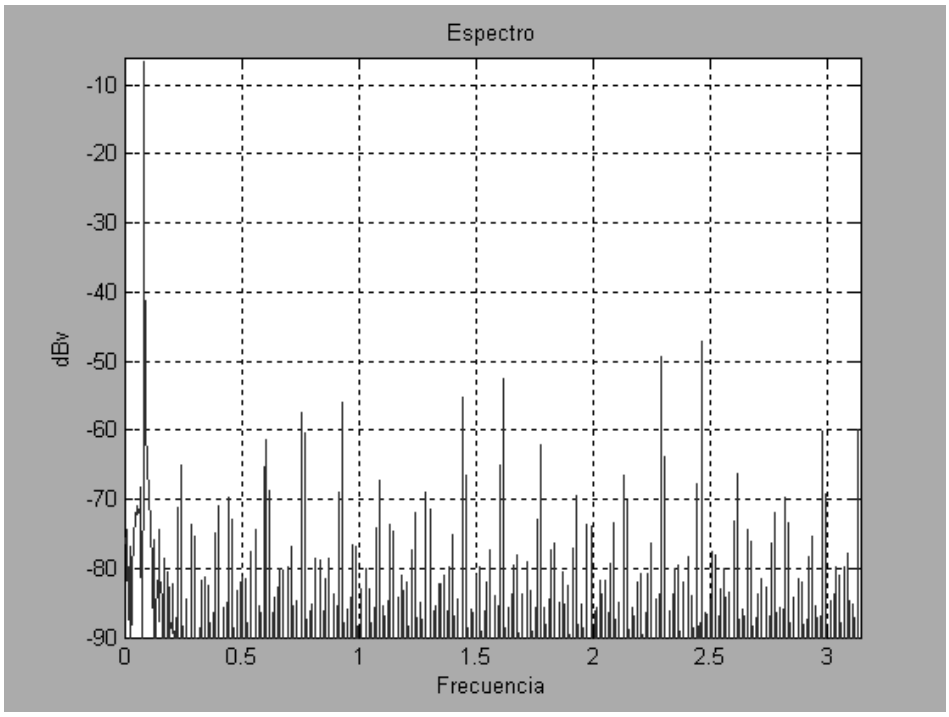

Figura 3.29.- Espectro de un tono puro de 440Hz tras un cambio de frecuencia de muestreo de 44100Hz a 32000Hz con retenedor de orden cero

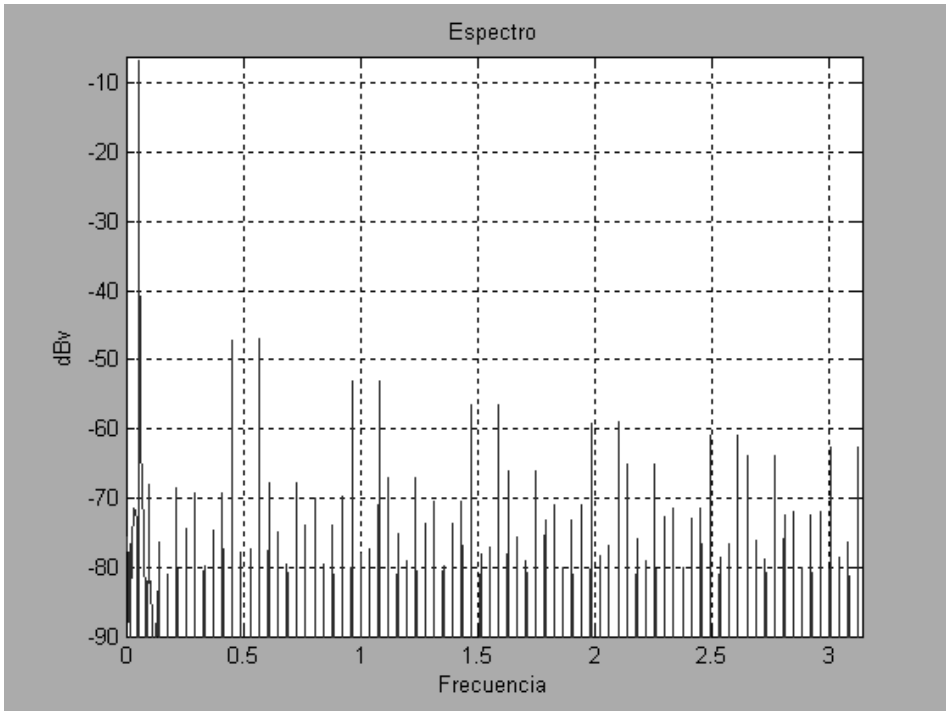

Figura 3.30.- Espectro de un tono puro de 440Hz tras un cambio de frecuencia de muestreo de 44100Hz a 48000Hz con retenedor de orden cero

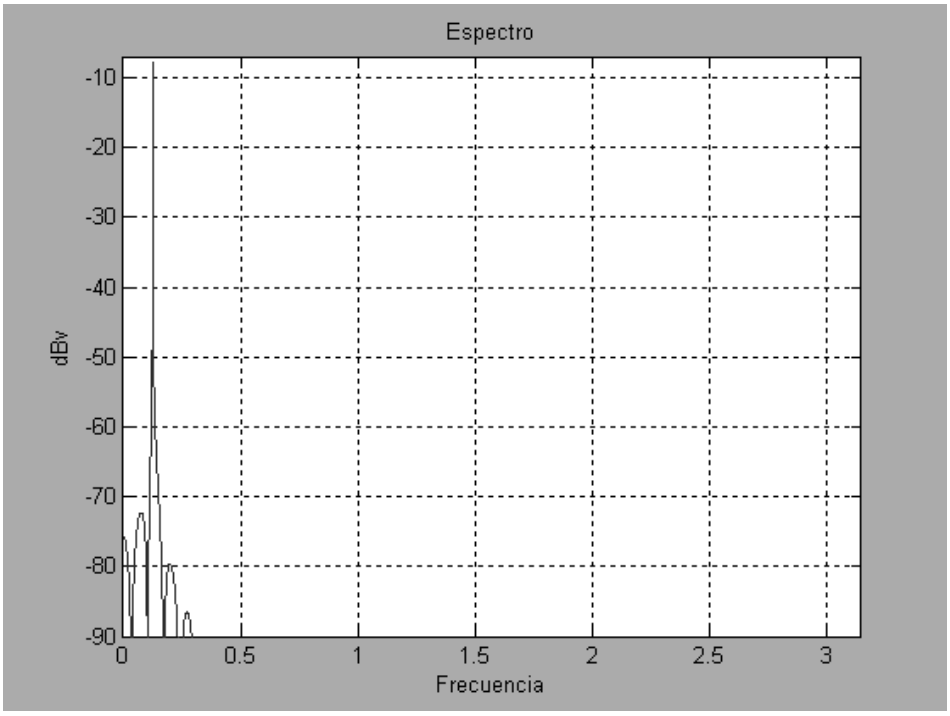

Figura 3.31.- Espectro de un tono puro de 440Hz tras un cambio de frecuencia de muestreo de 44100Hz a 22050Hz con retenedor de orden cero

En estas figuras se puede apreciar perfectamente en qué consiste la distorsión introducida por el aliasing de las réplicas mal filtradas durante la reconstrucción. Muy interesante resulta el caso de la conversión de 44100Hz a 22050Hz. Debido a la relación entera entre frecuencias, los armónicos causados por el aliasing van a coincidir siempre en las mismas componentes. Este caso concreto, es literalmente un diezmado de factor 2.

Pero además, observando el comportamiento de los armónicos en las conversiones no enteras, se llega a la conclusión de que a altas frecuencias la distorsión es aún mayor. Para corroborar esto se han incluido tres figuras que representan el mismo proceso que el anterior, pero con un tono de 20Khz.

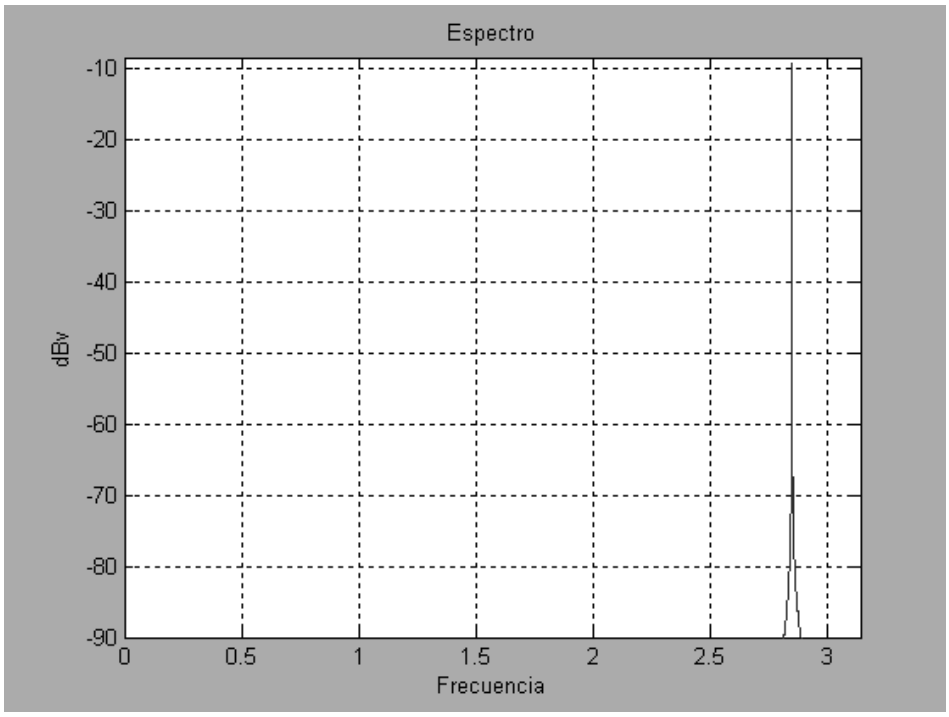

Figura 3.32.- Espectro de un tono puro de 20Khz muestreado a 44100Hz

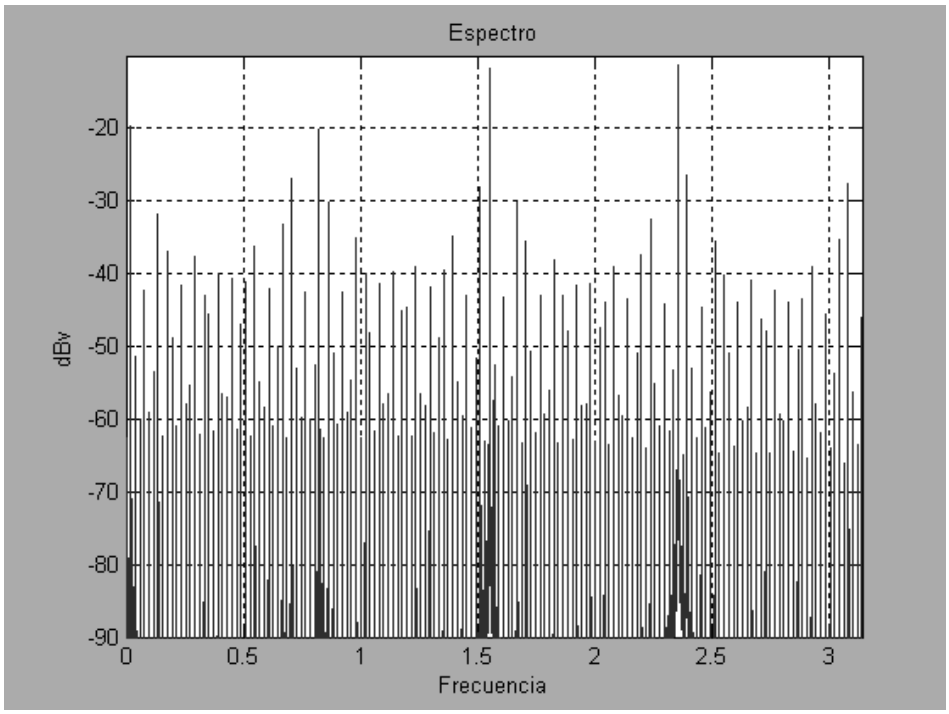

Figura 3.33.- Espectro de un tono puro de 20KHz tras un cambio de frecuencia de muestreo de 44100Hz a 32000Hz con retenedor de orden cero

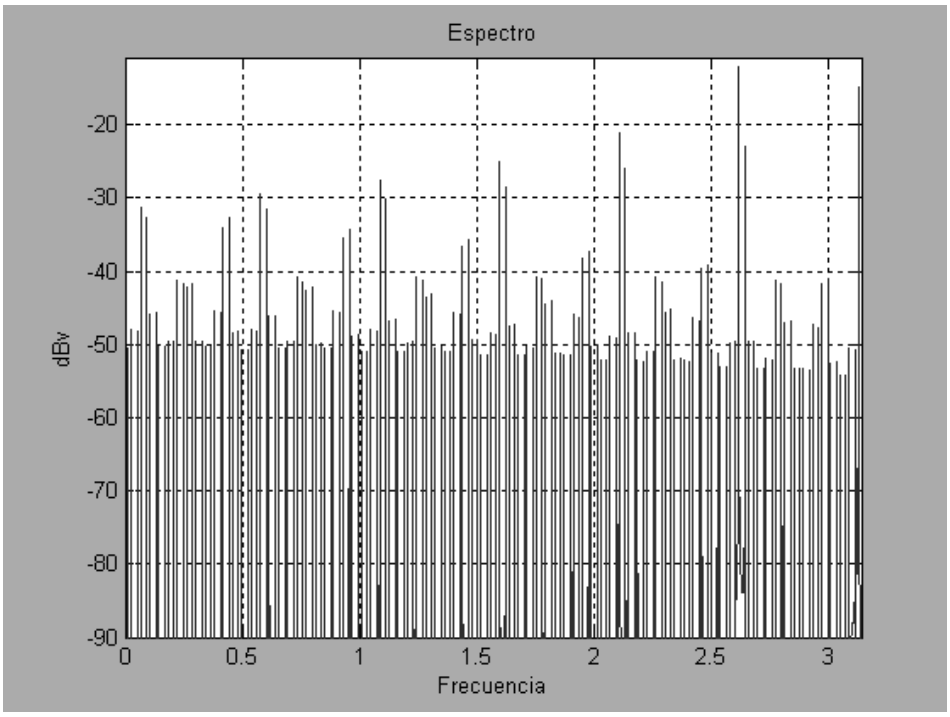

Figura 3.34.- Espectro de un tono puro de 20KHz tras un cambio de frecuencia de muestreo de 44100Hz a 48000Hz con retenedor de orden cero

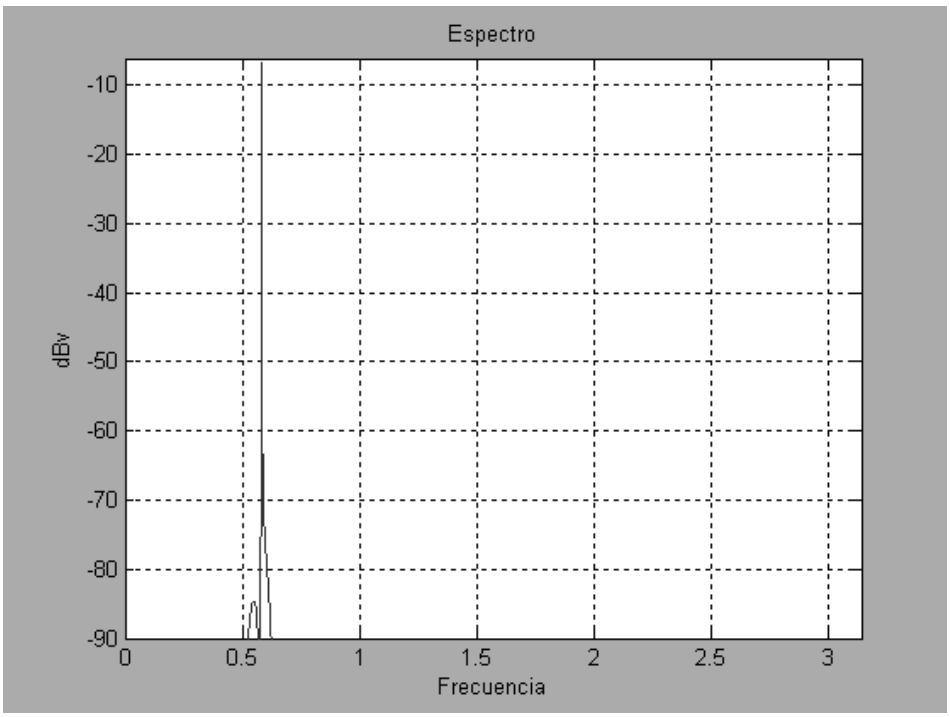

Figura 3.35.- Espectro de un tono puro de 20KHz tras un cambio de frecuencia de muestreo de 44100Hz a 22050Hz con retenedor de orden cero

Con el tono de 20Khz sí se aprecia la diferencia de aliasings que produce la conversión a una frecuencia mayor que la original, y la conversión a una frecuencia menor. En la Figura 3.33, que representa el espectro de una conversión a una frecuencia de muestreo menor, se puede ver que hay dos componentes de máxima amplitud. Estas dos componentes corresponden a las réplicas inmediatas por aliasing, ya que han sido muestreadas con una frecuencia inferior a la frecuencia de Nyquist necesaria.

Además, vuelve a mostrarse el caso en que la relación entre las frecuencias es entera y sencilla: de 44100Hz a 22050Hz. En este caso, la componente que aparece es fruto totalmente del aliasing. Además, debido a esta relación entera, sucede otra vez que todas las componentes frecuenciales mal filtradas durante la reconstrucción, se irán acumulando en el mismo punto del espectro, y no producirán una distorsión repartida por todo el eje de frecuencias. Esto no quiere decir que la señal tenga más calidad, ya que resulta igual de distorsionante una componente armónica incorrecta de semejante amplitud.

Implementación del algoritmo

Este tipo de conversión, de forma algorítmica, es la más sencilla que se puede realizar. Se realizará mediante procesado de búferes consecutivos, y consistirá simplemente en un 'muestreo' del búfer siguiendo el factor adecuado.

Si se realiza una conversión de búfer de 7 muestras, a un búfer de 14 muestras, por ejemplo, la Figura 3.36 ilustra cual sería el procedimiento a seguir. Únicamente es necesario recurrir a la posición $0.5 \cdot i$ del búfer original para generar la posición i del búfer nuevo. Si la posición original no es un número entero, se realizará un truncado para quedarse únicamente con la parte entera. De esta forma, se podría decir que la salida es: `Salida[i] = Entrada[(int)(0.5*i)]`.

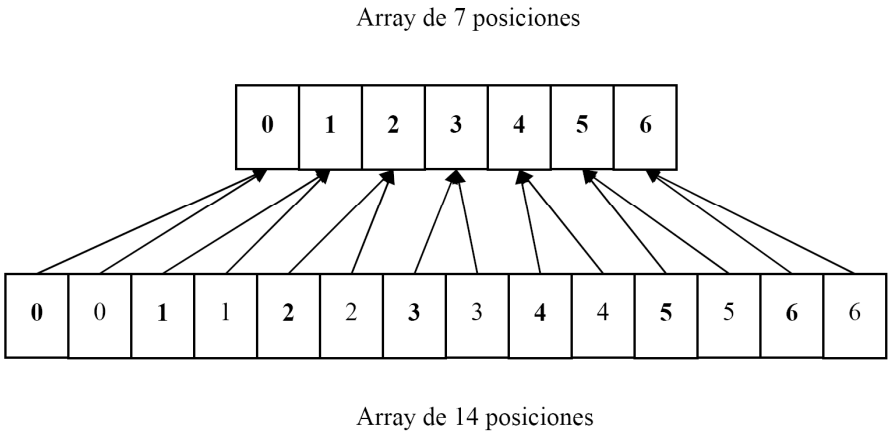

Figura 3.36.- Ejemplo de aplicación práctica del retenedor de orden cero cuando los tamaños de los búferes tienen una relación numérica entera

En caso de que la razón entre tamaño de búfer original y tamaño final no sea entera, la única diferencia que existe es que el factor de multiplicación no será 0.5, sino cualquier otro. En la Figura 3.37 se incluye un ejemplo en el que se pretende pasar de un búfer de 7 muestras a uno de 10. En este casi, la razón será 0.7 = 10/7. Por tanto, el código necesario para realizar esta conversión es: `Salida[i] = Entrada[(int)(0.7*i)]`.

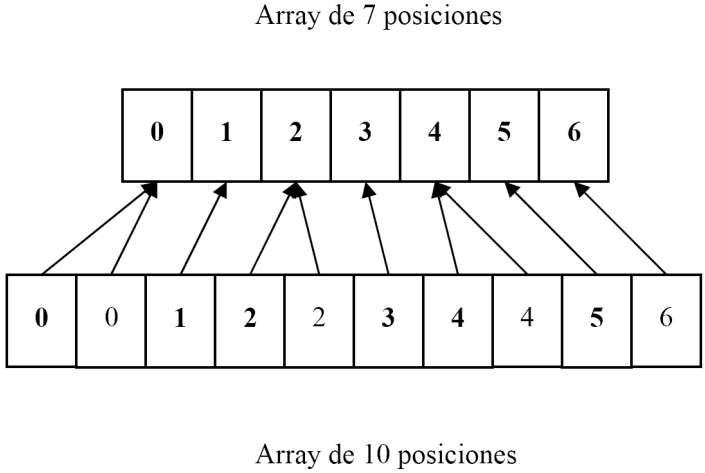

Figura 3.37.- Ejemplo de aplicación práctica del retenedor de orden cero cuando los tamaños de los búferes tienen una relación numérica no entera

Calidad subjetiva del audio resultante

A nivel subjetivo, la pérdida de calidad depende mucho del tipo de sonido que se esté convirtiendo. Tras realizar algunas pruebas, se llegó a la conclusión de que son dos los casos donde la pérdida de calidad se hace intolerable:

- Sonidos con pocas componentes armónicas o tonos puros.
- Sonidos con pocas componentes frecuenciales agudas aleatorias.

En estos dos casos, la distorsión que se introduce hace que sea muy perceptible la degradación del sonido. Sin embargo, para música normal, con componentes frecuenciales bien repartidas y cierta aleatoriedad de sonidos agudos, la calidad resultante es muy aceptable. Quizá no para la exportación de la mezcla, pero sí para trabajar de forma rápida haciéndose una buena idea de cómo va a sonar todo. Además, cuando la relación entre las frecuencias de muestreo a convertir es entera, la calidad de la conversión es bastante mayor.

En conclusión, es un método que viene bien para trabajar de forma liviana e ir escuchando más o menos como va quedando todo. Sin embargo, para la exportación de la mezcla este método no es el apropiado debido a la distorsión que introduce a la señal.

Para conseguir una mayor calidad resulta imprescindible realizar varios filtrados paso-bajo para ir eliminando componentes armónicas indeseadas. En esto se basarán los métodos explicados en posteriores subapartados.

Interpolación basada en convolución con sincs en el dominio del tiempo

Como se ha explicado ya, sería insuficiente dejar en la aplicación como único algoritmo interpolador el anteriormente expuesto. Por ello se ha implementado otro algoritmo que requiere un mayor procesado, pero ofrece una calidad mayor.

De esta forma se ha llegado a elaborar un algoritmo que consiste en una interpolación basada en la convolución con señales tipo sinc, capaz de realizar incluso el filtrado antialiasing necesario. Con él se simula el proceso ideal de reconstrucción y muestreo en el dominio del tiempo. A lo largo de la carrera se han estudiado conceptos de muestreo y reconstrucción con bastante profundidad, por tanto no se van a explicar de nuevo todos ellos. No obstante, sí conviene destacar las conclusiones más importantes que incumben al desarrollo de este algoritmo.

Reconstrucción teórica ideal de una señal discreta en el domino del tiempo

Dada una señal x(n) discreta, muestreada con una frecuencia de muestreo de f_m, entonces la señal continua reconstruida idealmente será la de la Ecuación 3.3.

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] \frac{\sin\left(\pi \cdot f_m \left(t - \frac{n}{f_m}\right)\right)}{\pi \cdot f_m \left(t - \frac{n}{f_m}\right)}$$

Ecuación 3.3

Desde el punto de vista temporal, se trata de la convolución de x[n] con sincs. Desde el punto de vista frecuencial se trata de un filtrado paso-bajo ideal con frecuencia de corte $f_m / 2$.

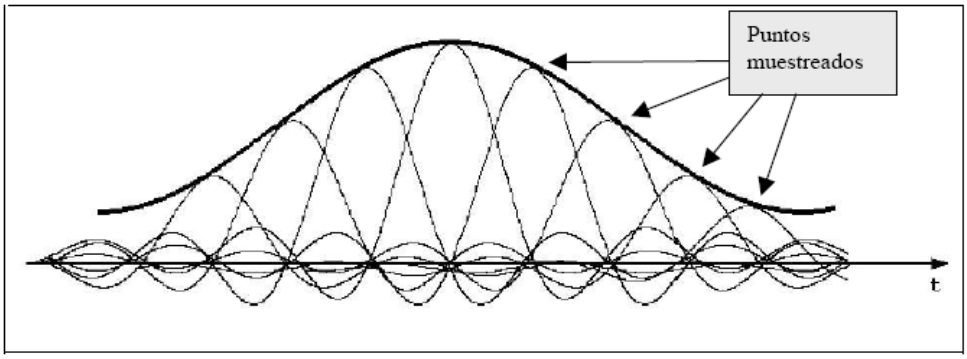

Figura 3.38.- Interpolación por superposición de sincs

En la Figura 3.38 se puede ver las sincs del sumatorio de forma individual superpuestas. La suma de todas ellas va a formar la señal continua reconstruida.

La Ecuación 3.3 representa una reconstrucción perfecta, pero sucede que en la práctica esto es imposible de realizar. El hecho de que el sumatorio tenga infinitos términos hace inviable su ejecución mediante un ordenador. Por tanto, es necesario realizarlo utilizando un número finito de términos.

Para el cálculo de x(t), las muestras que más van a afectar al resultado son las próximas a $n=(int)t \cdot f_m$. Siendo *(int)t* la conversión a entero de *t*. Por tanto, sería conveniente que para el cálculo de x(t) se tuvieran en cuenta los términos mencionados. Este proceso se denomina *enventanado.*

La forma de calcular x(t) con un número finitos de términos más apropiada será la de la Ecuación 3.4.

$$x(t, Orden) = \sum_{n=(\text{int})t\cdot f_m - Orden}^{(\text{int})t\cdot f_m + Orden} x[n] \frac{\sin\left(\pi \cdot f_m\left(t - \frac{n}{f_m}\right)\right)}{\pi \cdot f_m\left(t - \frac{n}{f_m}\right)}$$

Ecuación 3.4

De esta forma para el cálculo de x(t) se tienen en cuenta únicamente los términos de mayor relevancia. Además, la variable Orden da una idea del número de términos que se evalúan. Concretamente se evalúan $2 \cdot Orden + 1$ términos.

De esta forma, se puede obtener cualquier x(t) con un número finito de operaciones, a partir de x[n].

Muestreo de la señal continua resultante con una frecuencia de muestreo distinta

La fórmula que rige el muestreo en el dominio del tiempo es la Ecuación 3.5.

$$x[n] = x\left(\frac{n}{f_m}\right)$$

Ecuación 3.5

Donde x[n] con corchetes es la señal muestreada, y x(t) con paréntesis es la señal original. Lo único que hay que tener en cuenta, es que si f_m debe ser igual o mayor que el doble de la máxima componente frecuencial de x[n] (Teorema de Nyquist). Si esto no es así, x[n] está representando la discretización de x(t) con distorsión (por aliasing). Para evitar este problema, se suele aplicar lo que se llama un filtro anti-aliasing.

Este filtro debe estar situado antes del conversor analógico-digital y debe tener como frecuencia de corte la mitad de la frecuencia de muestreo que se vaya a utilizar.

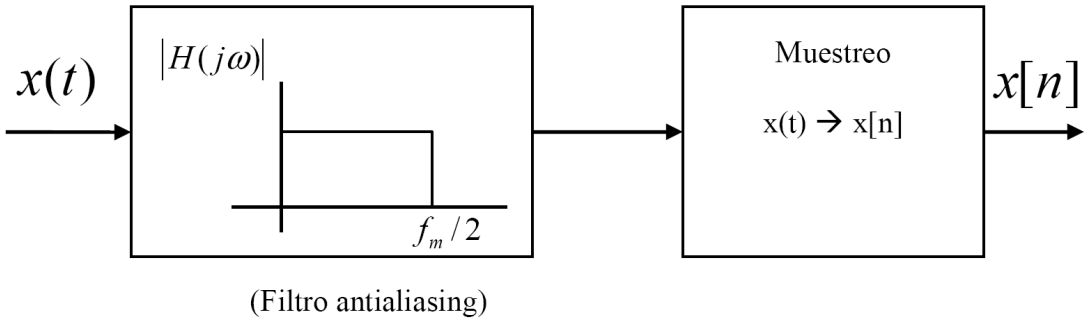

Figura 3.39.- Posición del filtro anti-aliasing en el proceso de muestreo

En la Figura 3.39 se puede ver la posición de un filtro antialiasing. Se usa para muestrear señales cuya frecuencia máxima está por encima de $f_m / 2$.

Implementación del algoritmo

Partiendo de los dos conceptos anteriores, se puede entender cómo funciona el método de interpolación desarrollado. El objetivo es pasar de una señal x[n] muestreada con f_{m1}, a otra señal y[n] que contiene los datos de x[n] muestreados a f_{m2}.

Para conseguir esto, primero se realizará una reconstrucción de x[n] en x(t), y posteriormente se muestreará esta señal con la nueva frecuencia de muestreo para generar y[n].

En la Figura 3.40 se muestra el proceso.

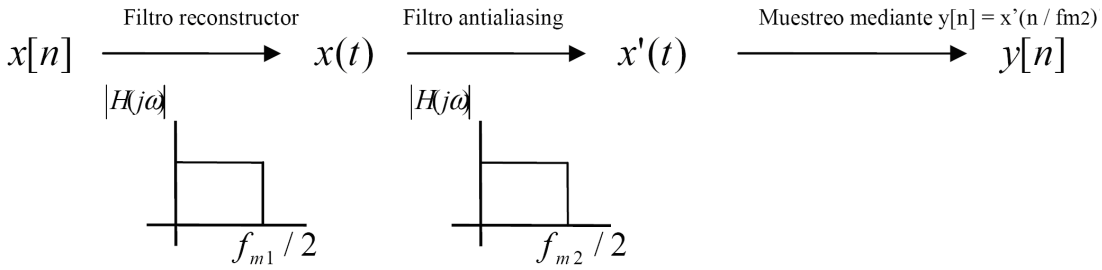

Figura 3.40.- Proceso completo de cambio de frecuencia de muestreo

Puesto que la señal continua intermedia es completamente auxiliar, no tiene por qué representar la señal x[n] reconstruida exactamente. Por tanto, se van a realizar algunos cambios que van a provocar que la señal x(t) propiamente dicha no sea la reconstrucción de x[n].

En primer lugar, dado que lo único importante es la relación entre f_{m1} y f_{m2}, se va a realizar la siguiente normalización: f_{m1} va a ser $f_{m1'}=1$, y f_{m2} va a ser $f_{m2'}= f_{m2}/ f_{m1}$. A estas nuevas frecuencias normalizadas se les añadirá la coma para diferenciarlas de las iniciales. Habiendo

realizado esta simplificación, resulta que x(0)=x[0], x(1)=x[1]… Algo que facilita el proceso posterior.

Por otro lado, el filtro reconstructor y el filtro antialiasing están en serie, por lo que se podrían agrupar en un solo filtro. Este filtro tendrá como frecuencia de corte la más restrictiva de $f_{m1'}$ y $f_{m2'}$. Aquella que sea más baja será la que determinará la frecuencia de corte del filtro.

De esta forma, el proceso queda como muestra la Figura 3.41.

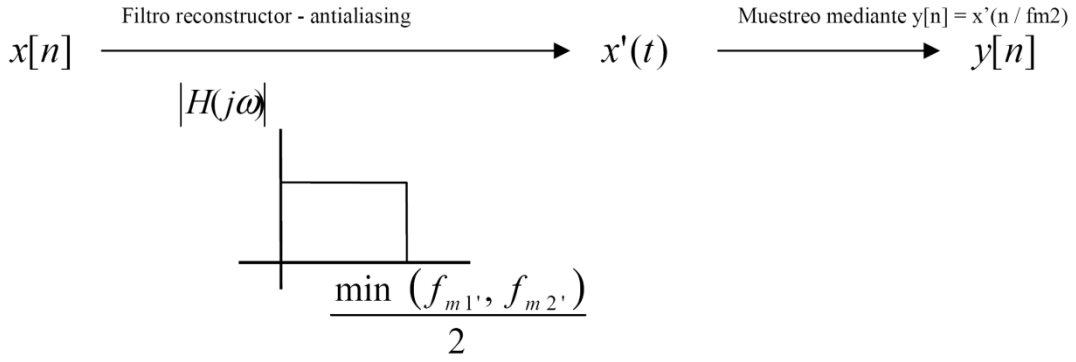

Figura 3.41.- Fusión de filtro reconstructor y antialiasing en uno solo

Habiendo esquematizado el proceso de cambio de frecuencia de muestreo, es necesario definirlo analíticamente mediante ecuaciones y fórmulas.

En primer lugar es necesario exponer cual es el resultado de filtrar una señal discreta para generar la continua con un filtro distinto al filtro ideal reconstructor (ver Ecuación 3.6).

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] \frac{2\pi \cdot f_c \cdot A}{\pi} \frac{\sin\left(2\pi \cdot f_c\left(t - \frac{n}{f_m}\right)\right)}{2\pi \cdot f_c\left(t - \frac{n}{f_m}\right)}$$

Ecuación 3.6

Donde A es la amplitud del filtro, f_c es la frecuencia de corte del filtro, y f_m es la frecuencia de muestreo de la señal digital.

Como es sabido, para reconstruir la señal discreta, la amplitud del filtro debe ser T. Esto es algo que se estudia en distintas asignaturas de la carrera.

Además, como f_m se ha normalizado a 1, T = 1 / f_m = 1, y la frecuencia de corte es el mínimo entre 2, de $f_{m1'}$=1, y de $f_{m2'}$= f_{m2}/ f_{m1}, la fórmula resultante se muestra en la Ecuación 3.7.

$$x'(t) = \sum_{n=-\infty}^{\infty} x[n] \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right) \cdot \frac{\sin\left(\pi \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right)(t-n)\right)}{\pi \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right)(t-n)}$$

Ecuación 3.7

Teniendo en cuenta que no se puede realizar jamás un número infinito de sumas, la expresión final quedaría como se muestra en la Ecuación 3.8.

$$x'(t, Orden) = \sum_{n=(\text{int})t-Orden}^{(\text{int})t+Orden} x[n] \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right) \cdot \frac{\sin\left(\pi \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right)(t-n)\right)}{\pi \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right)(t-n)}$$

Ecuación 3.8

Por último, si se tiene en cuenta la relación de la Ecuación 3.9.

$$y[m] = x'\left(\frac{m}{f'_{m2}}\right)$$

Ecuación 3.9

Entonces se tiene el proceso completo de reconversión, ya que se ha obtenido y[n] en función de x[n].

Sin embargo, existe otro detalle que no se ha tenido en cuenta, y que hace que la Ecuación 3.8 no se pueda implementar en un ordenador. Sucede que esta fórmula requiere en ciertas ocasiones muestras futuras, algo imposible de proporcionar en un sistema a tiempo real.

La solución a esto es aplicarle un retardo a la salida con respecto a la entrada, para no necesitar nunca recurrir a muestras futuras. Aplicando este retardo la fórmula queda como muestra la Ecuación 3.10.

$$x'(t, Orden) = \sum_{n=(\text{int})t-2 \cdot Orden}^{(\text{int})t} x[n] \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right) \cdot \frac{\sin\left(\pi \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right)(t-n-Orden)\right)}{\pi \cdot \min\left(1, {f_{m2}}/{f_{m1}}\right)(t-n-Orden)}$$

Ecuación 3.10

De esta forma, para el cálculo de x'(t) solamente es necesario recurrir como máximo a x[(int)t]. Este sistema de cálculo de x'(t) sí se puede implementar en un PC, y de hecho es el que se ha implementado en la aplicación.

A modo de síntesis, la Ecuación 3.11 expresa y[m] en función de x[n]. Las variables m y n se nombran de distinto modo porque durante el sumatorio no significan lo mismo.

$$y[m,Orden]=\sum_{n=(\text{int})\left(\frac{m\cdot f_{m1}}{f_{m2}}\right)/-2\cdot Orden}^{(\text{int})\left(\frac{m\cdot f_{m1}}{f_{m2}}\right)} x[n]\cdot \min\left(1,{f_{m2}}/{f_{m1}}\right)\cdot \frac{\sin\left(\pi\cdot\min\left(1,{f_{m2}}/{f_{m1}}\right)\left(\left(\frac{m\cdot f_{m1}}{f_{m2}}\right)-n-Orden\right)\right)}{\pi\cdot\min\left(1,{f_{m2}}/{f_{m1}}\right)\left(\left(\frac{m\cdot f_{m1}}{f_{m2}}\right)-n-Orden\right)}$$

Ecuación 3.11

Esta operación es la que hace el algoritmo de interpolación por sincs implementado en la aplicación, solamente que optimizado para minimizar el número de divisiones. Aplicando esta fórmula y[m] va a contener lo mismo que x[n], pero con distinta frecuencia de muestreo.

Análisis de calidad del algoritmo

Son dos los aspectos que hacen que la calidad de este algoritmo sea mejor que el basado en el retenedor de orden cero.

- Filtrado paso-bajo de mejor calidad durante la reconstrucción
- Filtrado anti-aliasing durante el muestreo.

El primer filtrado paso-bajo se encarga de eliminar en la medida de lo posible durante la reconstrucción todas las réplicas del espectro debido al muestreo de la señal. Como ya se ha explicado anteriormente, este filtro se implementa en el dominio del tiempo mediante una convolución de la señal con una señal tipo sinc.

Para comprender analíticamente cómo funciona un filtro ideal se han incluido algunas ecuaciones interesantes. La Ecuación 3.12 define un filtro paso-bajo ideal H(jω), del que se obtiene su respuesta al impulso. La Ecuación 3.13 define $x_p(t)$ como el muestreo en tiempo continuo de x(t). Esta definición se realiza únicamente para que sea coherente el desarrollo y todas las funciones estén en tiempo continuo. Finalmente, la Ecuación 3.14 y la Ecuación 3.15 representan el procesado en el dominio de la frecuencia y del tiempo necesarios para generar $x_r(t)$, que es la señal filtrada y reconstruida.

$$H(j\omega)=\begin{cases} T \xrightarrow{si} |\omega| < \omega_c \\ 0 \xrightarrow{si} |\omega| > \omega_c \end{cases} \longrightarrow h(t)=\frac{T\cdot\sin(\omega_c t)}{\pi\cdot t}$$

Ecuación 3.12

$$x_p(t)=\sum_{n=-\infty}^{\infty} x(nT)\delta(t-nT)$$

Ecuación 3.13

$$x_r(t)=x_p(t)*h(t)$$

Ecuación 3.14

$$x_r(t)=\sum_{n=-\infty}^{\infty} x(nT)h(t-nT)$$

Ecuación 3.15

$$X_r(j\omega)=X_p(j\omega)\cdot H(j\omega)$$

Ecuación 3.16

Sin embargo, debido al enventanado que es necesario realizar para utilizar un número finito de términos, estos procesos anteriormente descritos pasan de ser ideales a ser reales, con sus correspondientes pérdidas de calidad. En la Ecuación 3.18 se observa qué sucede con h(t) cuando se enventana y cómo afecta a la respuesta en frecuencia (Ecuación 3.19). En la Ecuación 3.17, el parámetro *O* va a representar el tamaño de la ventana. En cuanto mayor sea este parámetro, más próximo al filtrado ideal será el procesado, pero también será necesario más coste computacional para conseguirlo. La Figura 3.42 representa cómo evoluciona el espectro H(jω) conforme el parámetro *O* va creciendo. Se puede ver que la respuesta en frecuencia va siendo cada vez mejor, más plana y con un corte mucho más pronunciado.

$$v(t) = \begin{cases} 1 \xrightarrow{si} |t| < O \\ 0 \xrightarrow{si} |\omega| > 0 \end{cases}$$

Ecuación 3.17

$$h'(t) = h(t) \cdot v(t)$$

Ecuación 3.18

$$H'(j\omega) = \frac{1}{2\pi} H(j\omega) * V(j\omega)$$

Ecuación 3.19

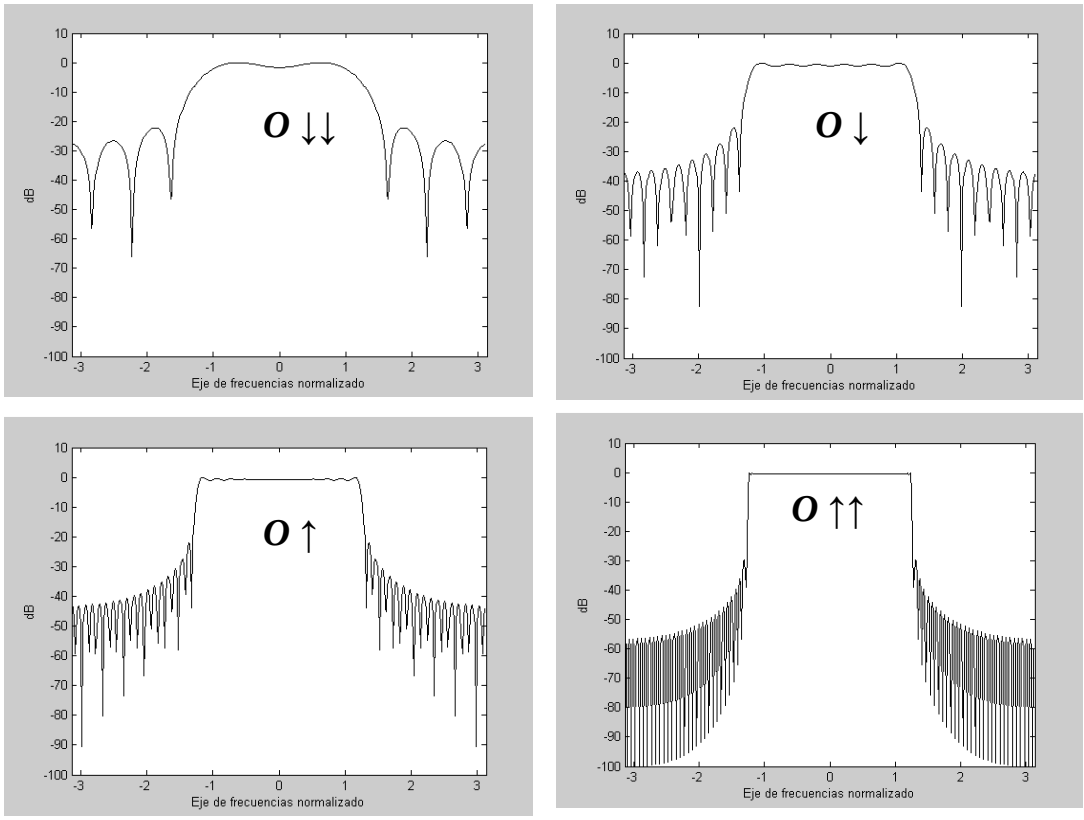

Figura 3.42.- Respuesta en frecuencia de un filtro paso-bajo ideal aplicando enventanado rectangular en el dominio del tiempo para distintos órdenes

Habiendo comprendido dónde está la no idealidad de éste proceso, es necesario observar qué calidad ofrece la aplicación para el procesado de audio. En la Figura 3.43 aparece el espectro del mismo tono de 440Hz de la Figura 3.28 procesado mediante este método de interpolación. Para la creación de estas señales procesadas se ha utilizado un orden de procesamiento de 64 muestras, que es el valor máximo que se ha implementado. De esta forma, se estudia el mejor de los casos con este algoritmo para delimitar la calidad del mismo.

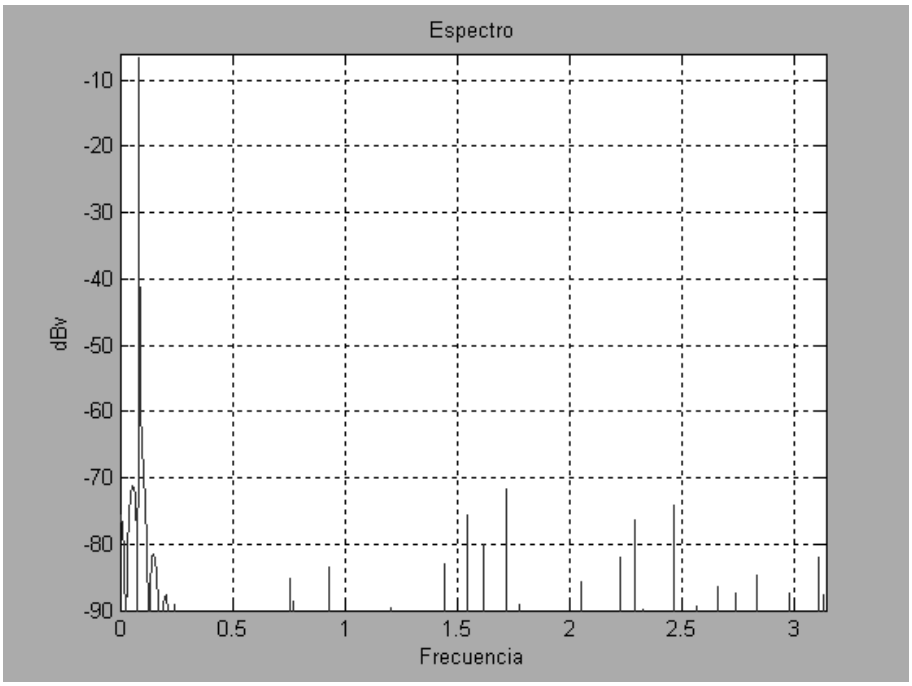

Figura 3.43.- Espectro de un tono puro de 440Hz tras un cambio de frecuencia de muestreo de 44100Hz a 32000Hz con interpolación por superposición de sincs enventanadas

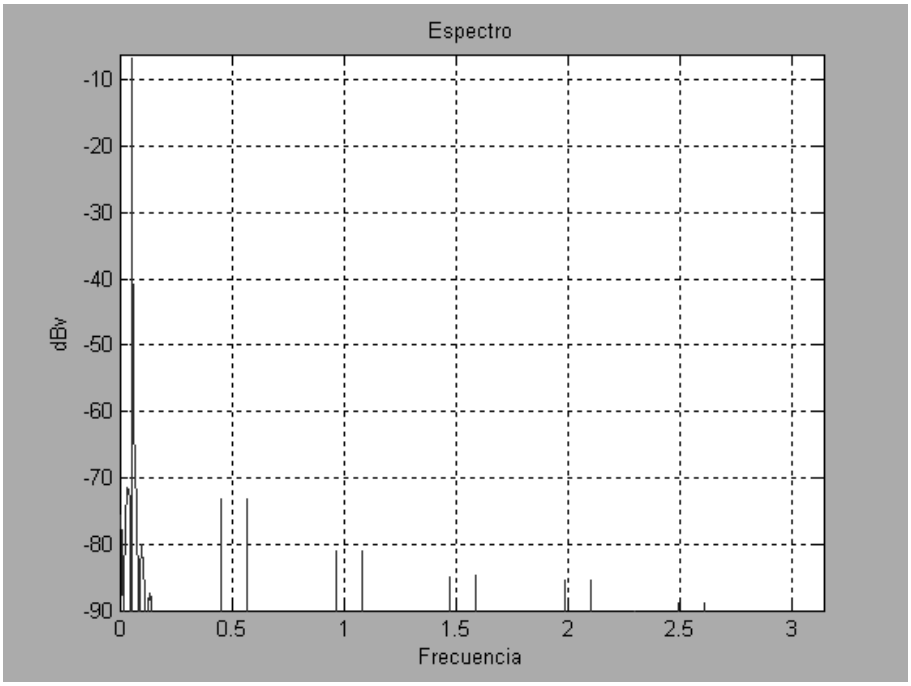

Figura 3.44.- Espectro de un tono puro de 440Hz tras un cambio de frecuencia de muestreo de 44100Hz a 48000Hz con interpolación por superposición de sincs enventanadas

Además, gracias al filtrado antialiasing, los problemas de armónicos que surgían con el tono de 20Khz desaparecen. En la Figura 3.45 se muestra el procesado de un tono puro de 20Khz para su conversión a 32000Hz y 48000Hz. Nótese, que dado que con 32000Hz no se puede muestrear un tono de 20Khz, la señal resultante tiene una muy pequeña amplitud (ver eje de ordenadas de la figura).

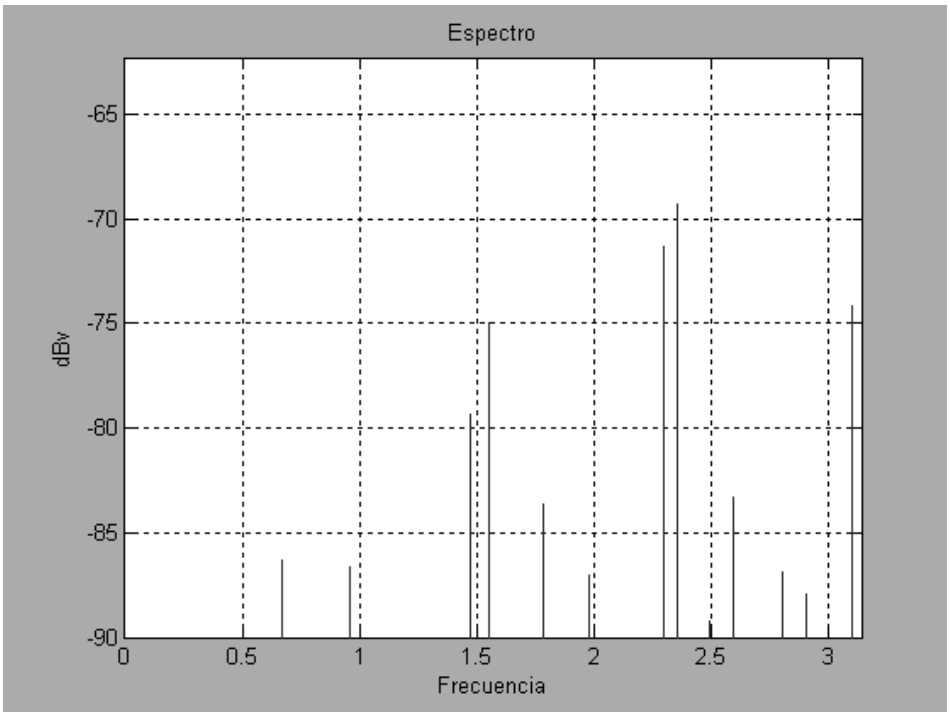

Figura 3.45.- Espectro de un tono puro de 20KHz tras un cambio de frecuencia de muestreo de 44100Hz a 32000Hz con interpolación por superposición de sincs enventanadas

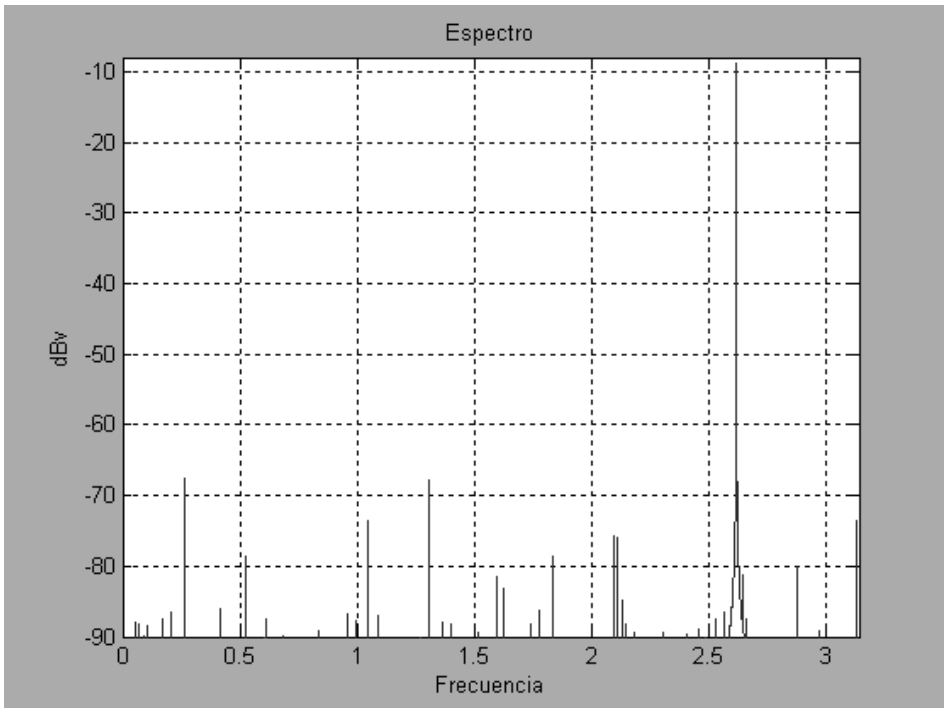

Figura 3.46.- Espectro de un tono puro de 20KHz tras un cambio de frecuencia de muestreo de 44100Hz a 48000Hz con interpolación por superposición de sincs enventanadas

Viendo todos estos resultados empíricos, se extraen varias conclusiones:

- Este método obtiene bastante mejor calidad que el método anterior, aunque sigue sin ser suficiente para aplicaciones de audio profesional (70dB de relación señal-ruido es insuficiente).
- La velocidad de procesamiento necesaria para interpolar con este sistema es bastante mayor que con el método anterior.
- Es útil para exportar la mezcla, aunque para trabajar a tiempo real puede llegar a requerir un PC demasiado potente si se elige un orden alto.

Inclusión de enventanado tipo Hanning

En el punto anterior se explicó el efecto del enventanado rectangular sobre el filtrado ideal paso-bajo. Debido a que el número de muestras a procesar es finito, la idealidad del sistema anterior se pierde y surge la necesidad de buscar soluciones que atenúen el efecto del enventanado.

La solución más sencilla y eficaz es utilizar una ventana distinta a la rectangular para multiplicar la respuesta al impulso del filtro ideal. De esta forma, V(t) en la Ecuación 3.17 variaría para convertirse en otra función. El objetivo de esto es mejorar la respuesta en frecuencia del filtro reconstructor-antialiasing, reduciendo la distorsión.

En la aplicación, a modo de ejemplo, se ha implementado una interpolación con enventanado Hanning. Esta ventana es una función que viene descrita en la Ecuación 3.20.

$$v[n] = 0.5 - 0.5 \cdot \cos\left(\frac{2\pi n}{N-1}\right)$$

Ecuación 3.20

Si en vez de la ventana rectangular, se utiliza esta ventana, la respuesta en frecuencia del filtro mejorará bastante. Si se observa la Ecuación 3.20, se verá que mientras v(t) en la Ecuación 3.17 era una señal continua, esta es una señal discreta. En realidad, la razón de esto es que en la práctica el enventanado se utiliza como si se tratase de una señal discreta (ver Ecuación 3.15, que calcula la convolución con un sumatorio aun siendo señales continuas).

La respuesta en frecuencia del filtro paso-bajo que resulta con Hanning, en comparación con la ventana rectangular, aparece en la Figura 3.47.

|H(jω)| Con ventana rectangular

|H(jω)| Con ventana Hanning

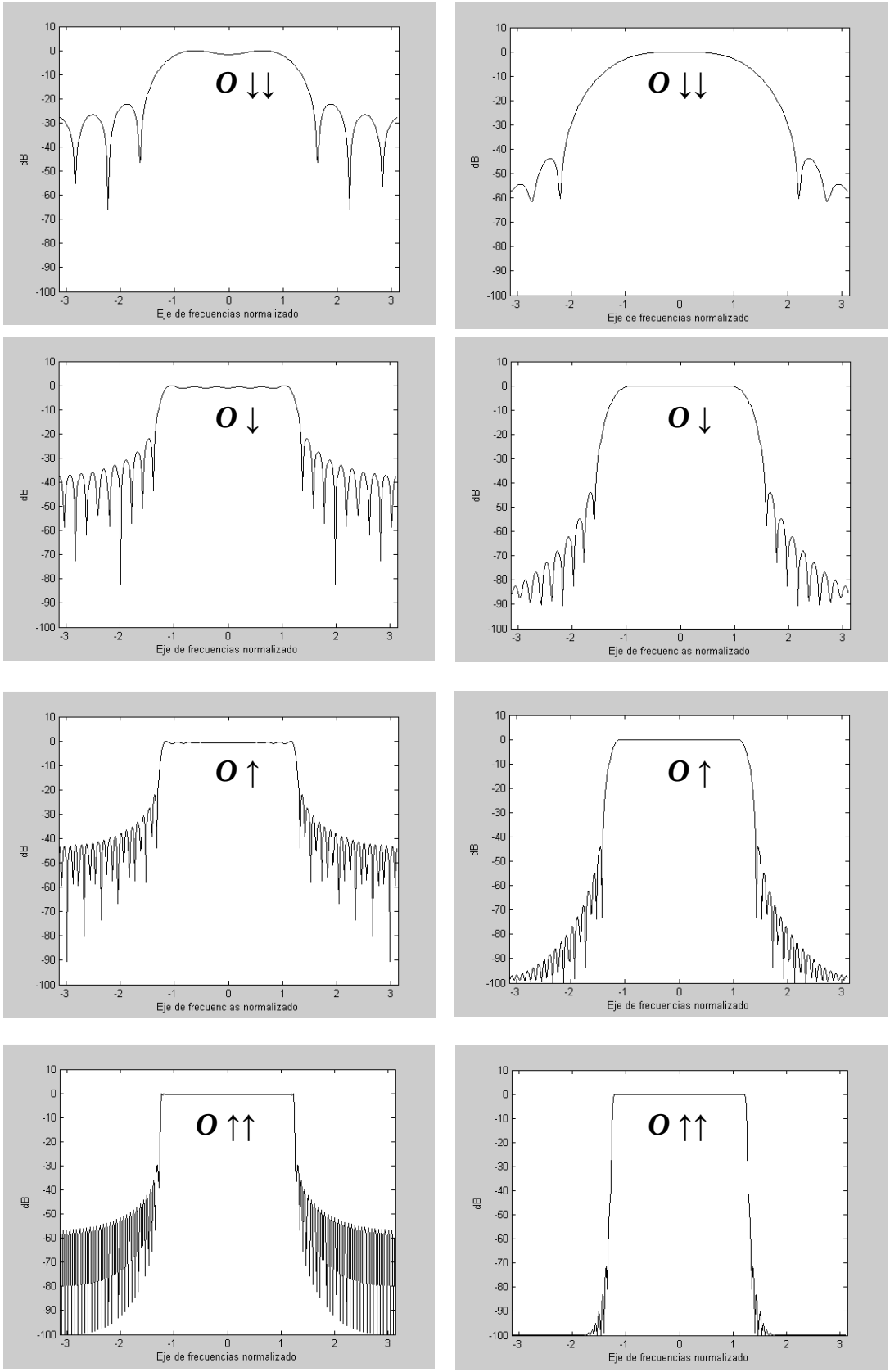

Figura 3.47.- Comparación de la respuesta en frecuencia de un filtro paso-bajo ideal con enventanado rectangular y con enventanado Hanning

El resultado de aplicar el enventanado Hanning, es que las componentes armónicas indeseadas se reducen, y por tanto la distorsión es menor.

En la aplicación se ha dejado un control para que el usuario escoja si desea aplicar enventanado Hanning o no. Realmente la única función de éste es didáctica, ya que la aplicación del enventanado no consume prácticamente recursos.

Interpolación basada en función 'resample' de MATLAB

Función 'resample'

MATLAB dispone de una función llamada 'resample', que recibe como parámetro un array, la frecuencia de muestreo de entrada y de salida, y devuelve el mismo array con otra frecuencia de muestreo distinta. Aprovechando esta función se ha implementado en la aplicación otro sistema de remuestreo, de mucha más calidad, pero mucho más lento.

Según la ayuda de MATLAB, esta función realiza la conversión aplicando un filtro *polifase*. Este tipo de filtrado es más efectivo que los anteriormente vistos, pero también es mucho más costoso computacionalmente hablando.

Esta función tiene la siguiente cabecera:

```
DatosNuevos = Resample(DatosViejos, Fm1,Fm2,OrdenFiltros);
```

Calidad y velocidad del algoritmo

Este algoritmo consigue una calidad mucho mayor que los anteriores, pero tarda también mucho más. Es inviable para su uso en tiempo real, aunque sí puede usarse con éxito para exportar la mezcla general con la máxima calidad.

La distorsión que introduce este algoritmo es tan baja, que su uso puede aplicarse en audio profesional sin problemas. En la Figura 3.48 se puede ver el resultado de transformar un tono puro de 440Hz a distintas frecuencias de muestreo con este algoritmo.

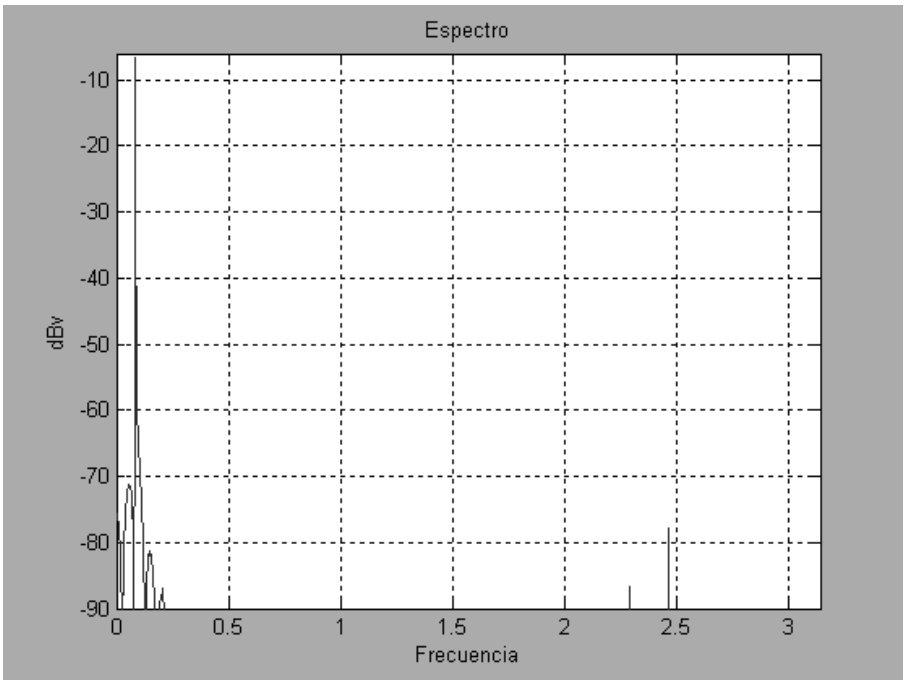

Figura 3.48.- Espectro de un tono puro de 440Hz tras un cambio de frecuencia de muestreo de 44100Hz a 32000Hz con función 'resample' de MATLAB

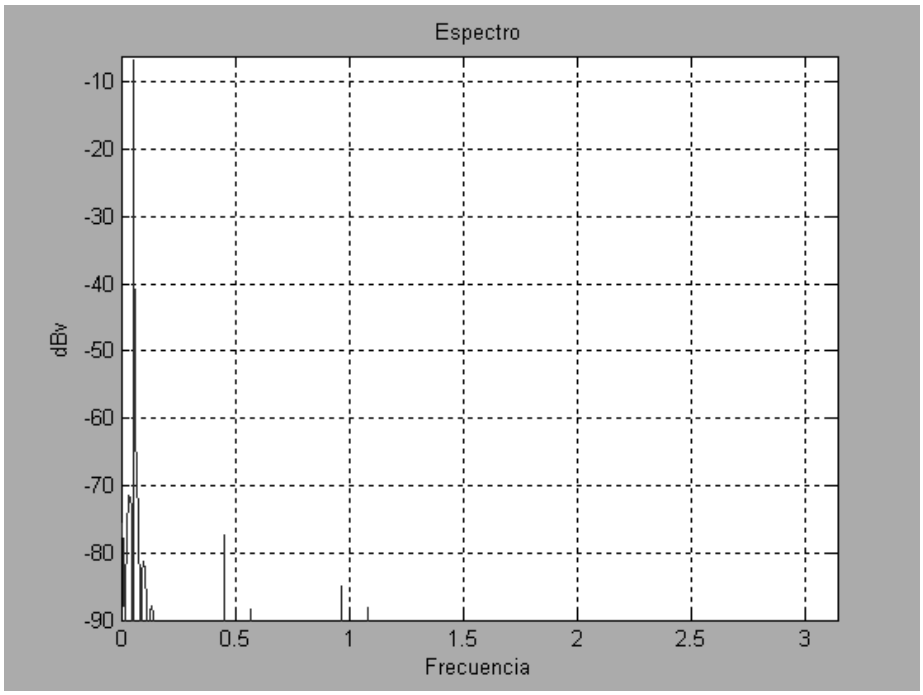

Figura 3.49.- Espectro de un tono puro de 440Hz tras un cambio de frecuencia de muestreo de 44100Hz a 48000Hz con función 'resample' de MATLAB

Si se compara la calidad que ofrece este algoritmo con los demás, se ve que es claramente superior. Sin embargo, su aplicación es tan lenta que en la aplicación se ha implementado únicamente para obtener la mezcla final con el máximo de calidad.

Mezclado para exportación a fichero WAV o MP3

Para el mezclado a tiempo real se explicó anteriormente que `WaveReproductor` recurría a la rutina de mezclado cada *X* segundos para obtener los datos a reproducir.

Esta misma rutina de mezclado se puede utilizar para mezclar todas las pistas y exportarlo a un fichero de audio externo. Sin embargo, en este caso no será `WaveReproductor` quien recurra a la rutina de mezclado, sino una función llamada `Mezclar()` incluida dentro de `CMotorDeAudio`.

Esta rutina tiene los siguientes parámetros:

```
public void Mezclar(double InicioMezcla,
                    double FinalMezcla,
                    string FicheroSalida)
```

Código 3.8.- Cabecera de la función Mezclar

Simula que se ha hecho Play, pero los bytes resultante de la mezcla los dirige a un fichero de salida en lugar de al reproductor de audio. Además, al no haber ninguna pausa entre la mezcla de un búfer y el siguiente, el proceso es mucho más veloz que la mezcla a tiempo real. El resultado final es un fichero WAV (con nombre `FicheroSalida`) que contiene la mezcla de todas las pistas, con efectos incluidos si procede.

Para que el usuario pueda seleccionar todos los parámetros necesarios para crear la mezcla se ha creado un formulario llamado `FormCreaMezcla`, cuyo aspecto se muestra en la Figura 3.50. Este formulario simplemente sirve de interfaz para utilizar este método `Mezclar` mencionado anteriormente.

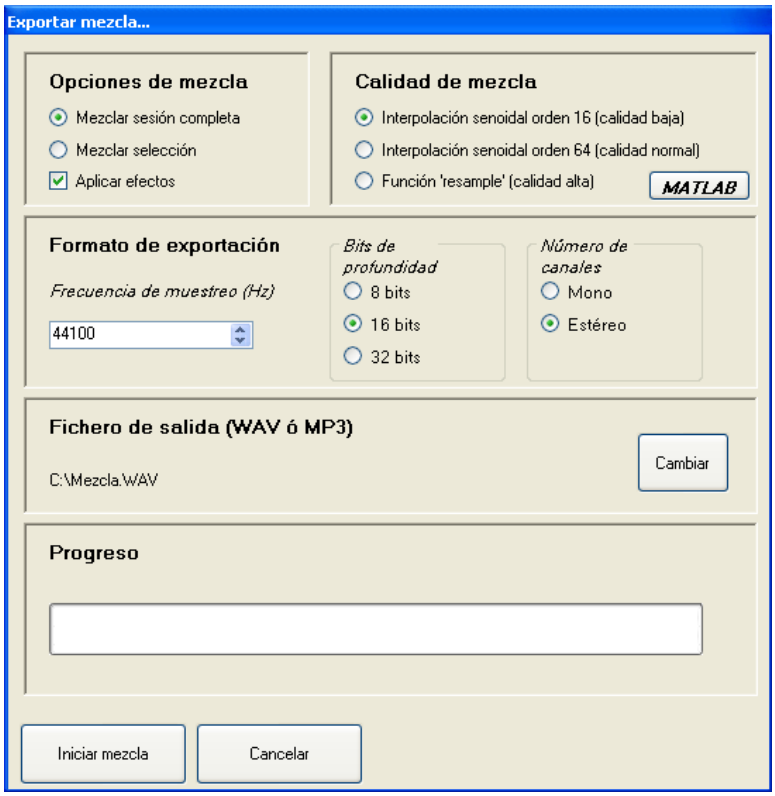

Figura 3.50.- Aspecto visual de FormCreaMezcla

Como se puede apreciar, el usuario puede escoger las opciones más importantes a la hora de realizar una mezcla: segundo inicial, segundo final, formato de exportación, nombre de fichero de salida, etc. La conversión a MP3 se realiza a través de la clase `Aumpel`, ya descrita en el apartado 3.7.

Además, se puede escoger si se quiere que los efectos se tengan en cuenta a la hora de exportar la mezcla o no. Esto en el código se traduce simplemente en una sentencia `if` antes de aplicar los efectos a los array de doubles L y R.

Distribución de datos a pistas durante la grabación

Durante la grabación, proveniente de cada dispositivo de entrada entra un array de bytes con el formato establecido. Este array de bytes contiene los datos del audio que se debe registrar en las pistas marcadas con una R durante la grabación. El proceso mediante el cual se distribuyen los datos provenientes de los dispositivos a cada pista se va a llamar “distribución”.

Si se observa cómo funciona la clase `WaveGrabador`, contendrá un delegado pasado por parámetro que será el encargado de darle uso a los datos provenientes de los distintos dispositivos de entrada. La rutina delegada que realizará esta función está contenida en `CMotorDeAudio` y se

llama `Distribuidor()`. Esta rutina se encargará de transformar el array de bytes entrante en un par de array de doubles, adaptar el formato de este array al de las pistas en cuestión, y registrarlo en las pistas.

El proceso de distribución se explicará con menos detalle que el de mezclado debido al gran número de analogías con el mismo, ya explicadas. Por tanto se empezará comentando el algoritmo de distribución. En la Figura 3.51 se expone un flujograma del mismo.

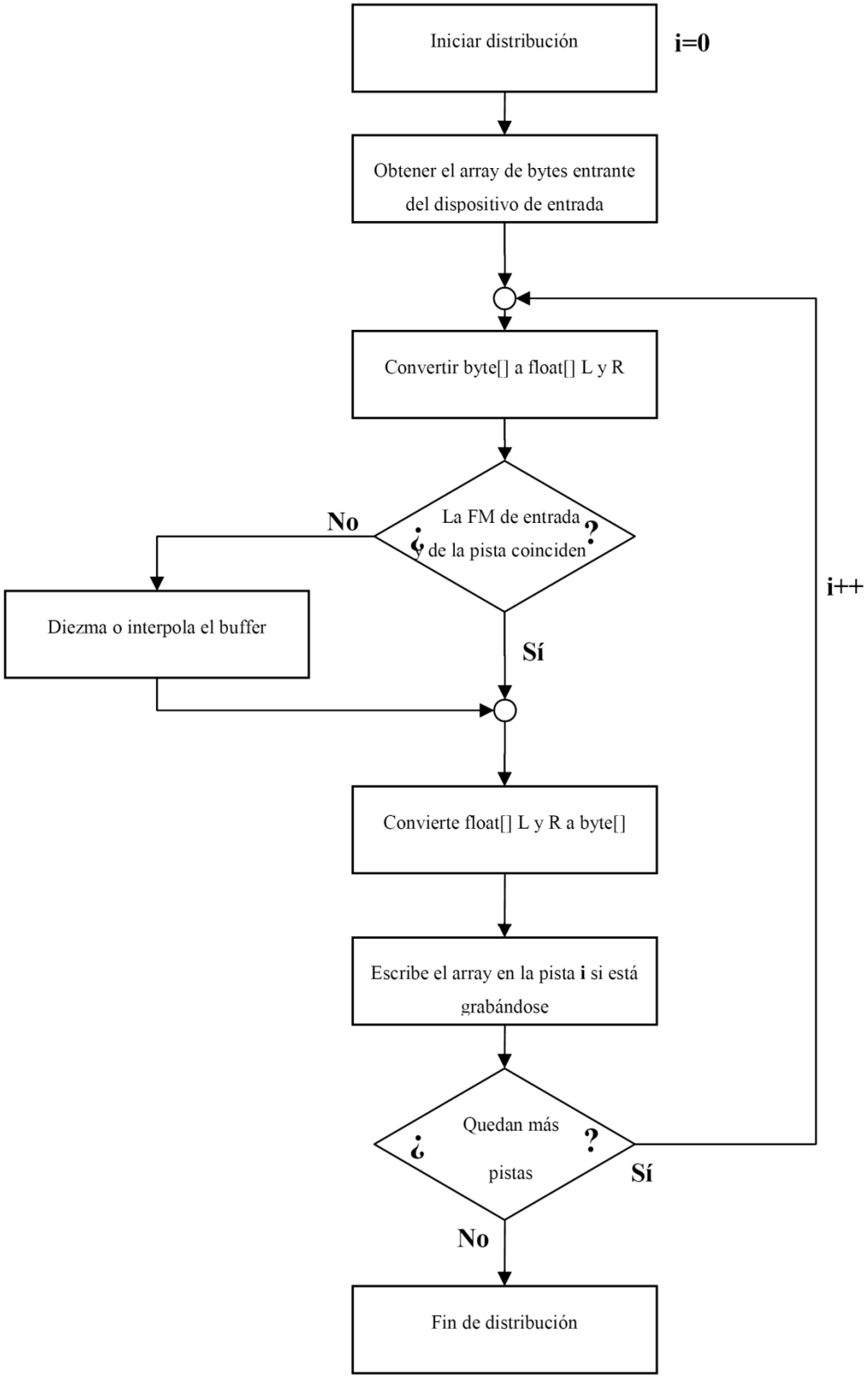

Figura 3.51.- Algoritmo de distribución de datos a las distintas pistas

Panel de control

Para la configuración de todos los parámetros asociados a la reproducción y la grabación se ha creado un panel de control. En este panel, el usuario puede configurar todos los parámetros de los que depende la reproducción y grabación de sonido (formato, tamaño de búferes, etc.).

Se ha creado un formulario con varias pestañas, en las que se puede configurar:

- El formato de reproducción y de grabación, variando frecuencia de muestreo, bits por muestra y número de canales.
- El número de búferes y el tamaño de estos para la reproducción y grabación.
- El algoritmo de interpolación a tiempo real que se utilizará.

El panel de control está directamente asociado a `CMotorDeAudio`, ya que todos los parámetros manipulables forman parte de esta clase, aunque se instancia en `FormPrincipal`.

3.9.- Efectos de sonido e inclusión de rutinas MATLAB en C#

Unas de las aportaciones más interesantes al software es la capacidad para utilizar efectos de sonido. Estos efectos de sonido se podrán aplicar a cada pista independientemente a tiempo real, o bien procesando un determinado fichero de audio.

En el mundo del software de audio, un efecto de sonido no es más que un procesado de señal. Desde el punto de vista musical, los procesados más útiles y comunes son: amplificación, normalización, compresión, silenciado, reverberación y ecualización. No obstante, existen una infinidad de efectos de sonido distintos, siendo algunos de ellos especialmente interesantes.

Los efectos de sonido, musicalmente hablando se pueden clasificar en cuatro categorías:

- Efectos basados en la amplitud y el volumen
- Efectos basados en el espacio y el estéreo
- Efectos basados en el timbre
- Efectos basados en el tiempo

Los efectos basados en la amplitud y el volumen van a modificar solamente la amplitud de la señal. Esta modificación puede ser variable con el tiempo, o puede depender de parámetros

propios de la señal. Ejemplos de este tipo de efectos: normalización, compresión, limitación, trémolo…

Los efectos basados en el espacio y el estéreo son aquellos que pretenden simular el sonido en el espacio. Generalmente los parámetros de los efectos suelen ser fijos en el tiempo, aunque no es algo imprescindible. Algunos ejemplos claros de de efectos basados en el espacio y el estéreo son: reverberación, eco, posicionamiento estéreo, posicionamiento 3D.

Los efectos basados en el timbre son aquellos que modifican la distribución armónica de un sonido. Las variaciones tímbricas pueden o no variar en función del tiempo dependiendo del tipo de efecto. Los ejemplos más claros son: ecualización, chorus, distorsión, flanger, etc.

Por último, los efectos basados en tiempo son aquellos que modifican la velocidad de reproducción o la duración de un sonido. Los casos más típicos son la ralentización y la aceleración.

Debido a la infinidad de efectos que hay, en ciertas ocasiones un determinado efecto no es fácilmente clasificable. Incluso, hay algunos procesados como los de reducción de ruido que se debería incluir casi en una categoría aparte.

Procesado de la señal digital

Para realizar el procesado de la señal, se manipularán sus muestras adecuadamente con el fin de conseguir un determinado efecto. Por tanto, si por ejemplo se multiplican los valores numéricos de todas las muestras por 0.5, el resultado es la misma señal con la mitad de amplitud.

Además, en multitud de ocasiones, para el cálculo de una determinada muestra es necesario disponer de muestras anteriores (sistemas con memoria), e incluso de muestras futuras (sistemas anticausales).

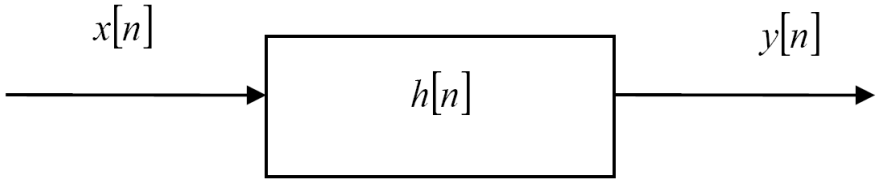

Figura 3.52.- Esquema genérico de sistema procesador de señal

En la Figura 3.52 se representa el esquema de un procesador, que recibe una señal de entrada x[n], y devuelve una señal de salida y[n] ya procesada.

Para realizar un procesado software de la señal hay que crear una estructura similar. Se trata de una clase que sea capaz de recibir un array de muestras (x[n]), procesarlo y devolver otro array de muestras (y[n]). Además, esta clase puede albergar muestras anteriores, parámetros, etc.

La idea para la aplicación de efectos por tanto va a ser esta: una clase que albergará todas las rutinas de procesado, los parámetros y datos auxiliares necesarios para realizar un procesado a un array de entrada.

En este desarrollo se ha creado un modelo genérico de efecto llamado `CEfectoGenerico`, que consta de los miembros que aparecen en el Código 3.9.

```
public class CEfectoGenerico
{
    public CEfectoGenerico()
    public string NombreEfecto;
    public virtual void FinalizadoProcesado()
    public virtual void Configurar()
    public virtual void Procesado(ref double[] CanalL,
                                  ref double[] CanalR)
}
```

Código 3.9.- Miembros de CEfectoGenerico

Para crear efectos, simplemente habrá que heredar clases de `CEfectoGenerico`, y sobrescribir los métodos. De esta forma todos los efectos tendrán la misma interfaz y será fácil su utilización homogénea.

Aplicación de efectos a tiempo real

Si se observa la Figura 3.4 mostrada en el apartado 3.8, se verá que existe un punto en el proceso de mezclado dedicado a la aplicación de efectos. Pues es ahí donde entra en juego todo lo que está explicando. Una vez se convierten los datos de una pista a dos arrays de `doubles` (L y R), antes de mezclarse con el resto, se pueden manipular estos arrays sin problemas.

Aprovechando esto, se intercala una llamada al procesado de cada efecto integrado en el control `UCPista` y se realiza dicho procesado a los arrays L y R. De esta forma, a la misma vez que se realiza todo el mezclado, se aplican los efectos pertinentes a cada pista. Sería algo similar a las inserciones en las mesas de mezcla habituales.

El único problema que supone esto es una reducción de la velocidad de procesamiento, y por tanto el aumento de la probabilidad de que aparezcan paradas y chasquidos durante la reproducción. Una posible solución es aumentar el número de búferes y el tamaño del búfer. No obstante, existe una limitación clara en el número de efectos a tiempo real por sesión.

Aplicación de efectos a ficheros de audio independientes

Para la aplicación de un efecto de audio a un fichero WAV de forma permanente, se ha creado una rutina dentro de la clase `CMotorDeAudio` llamada `AplicarEfecto()`. Esta rutina tiene como parámetros el segundo inicial, el segundo final, el número de pista y el efecto. Lo que hace es ir leyendo búferes del fichero y aplicándoles el efecto.

Además, es importante mencionar el hecho de que el procesado se realice en segundo plano mediante un subproceso. De esta forma, la GUI se mantiene libre y se dispone de un botón Cancelar para detener el proceso.

La aplicación de un efecto a una pista de forma separada sobrescribe el fichero WAV asociado e impide la recuperación de los datos originales. Por tanto, se recomienda que se pruebe el efecto antes a tiempo real. Si a tiempo real el efecto cumple las expectativas, se puede aplicar de forma permanente a la pista para aliviar recursos. A la hora de aplicar un procesado, el usuario tendrá la posibilidad de escoger el fichero completo, o por el contrario una porción del mismo (la porción seleccionada).

Análisis de señales

Otra posible utilización del procesado de ficheros de forma independiente es el análisis de señales. Puesto que cada efecto puede disponer de formularios propios y cuadros de diálogo de información, se pueden aprovechar para mostrar datos por pantallas relacionados con el análisis de la señal.

Para ello, la rutina de procesado del efecto no va a modificar los valores de las muestras, sólo va a utilizarla para generar otros datos de salida por pantalla.

Inclusión de rutinas MATLAB en C#

MATLAB es un programa de cálculo numérico, orientado a matrices y vectores. Desarrollado por MathWorks en 1984, se ha convertido en una de las mejores aplicaciones del mercado para la resolución de problemas de ingeniería que requieran un gran procesado numérico.

Gracias al añadido del toolbox (caja de herramientas) de procesado de señal, se disponen de multitud de funciones muy útiles a la hora de realizar procesados: cálculo de transformadas, diseño de filtros, filtrado a partir de coeficientes, etc.

Es por ello, que disponiendo de MATLAB con el ToolBox de procesado de señales, la creación de un efecto de sonido se simplifica enormemente. Una tarea que en C# posiblemente conllevaría varias horas de trabajo e investigación, puede resolverse en MATLAB en algunos minutos y con muy poca cantidad de código.

Además, MATLAB incluye un sistema de tipos especialmente útil para el procesado de señales. Trabaja generalmente con matrices de números complejos con precisión doble, así que cualquier número o variable, siempre va a ser un caso concreto de este tipo. Por ejemplo, un vector que contenga los valores [1, 2, 3, 4, 5], se traduce en MATLAB como una matriz de dimensión (1,5), que contiene números cuya componente imaginaria es nula en todos ellos.

Las funciones de MATLAB se almacenan en ficheros de extensión .m, y tienen la capacidad de recibir multitud de vectores o matrices, y devolver también una cantidad indefinida de las mismas. Estas funciones van a tener funcionalidades completas y en ellas se dispone de todas las potentes herramientas que MATLAB ofrece para procesado, representación gráfica, etc.

MATLAB Builder .NET

Mediante MATLAB Builder .NET, se pueden desarrollar componentes utilizables en cualquier solución desarrollada con la Plataforma .NET. Aprovechando esta posibilidad, se han desarrollado distintos efectos de ejemplo en MATLAB para integrarlos con la aplicación.

Para ejecutar MATLAB Builder .NET desde MATLAB hay que escribir en la línea de comandos la palabra “dotnettool”. Con esta herramienta se puede crear un componente .NET en DLL que contenga tantas clases como se desee con un número ilimitado de funciones .m desarrolladas. En la Figura 3.53 se puede ver una captura de pantalla de MATLAB Builder .NET.

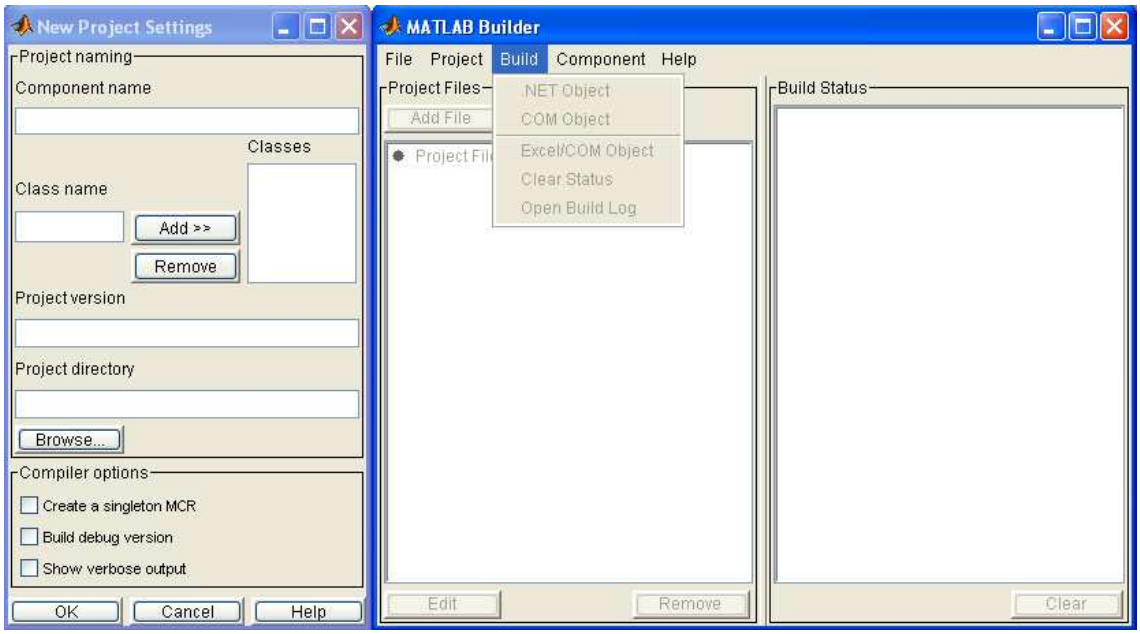

Figura 3.53.- Captura de pantalla de MATLAB Builder .NET

Esta aplicación genera finalmente una DLL que se puede agregar a las referencias del proyecto que se desee. Para ello hay que crear un proyecto nuevo, crear una clase, agregarle algunos ficheros .m y darle a Build→.NET Object.

Una vez agregada la DLL al proyecto .NET en cuestión, se podrán utilizar todas sus clases como si en C# estuvieran desarrolladas. Simplemente hay que tener en cuenta que el sistema de tipos de las funciones de MATLAB es propio, y hace falta realizar ciertas conversiones para lograr trabajar cómodamente con los valores numéricos de los vectores.

Conversión de tipos

La funciones importadas de MATLAB trabajan con un sistema de tipos propio distinto al de C#. Para poder disponer de estos tipos es necesario agregar a las referencias del proyecto el fichero MWArray.DLL.

El tipo del cual heredan todos los demás es `MWArray`, que representa un array de una o más dimensiones de cualquier tipo. Los parámetros de entrada y salida de las funciones que se importen deben ser siempre de tipo `MWArray`. Puesto que este tipo no define si los elementos del array van a ser caracteres, números, valores lógicos, etc. no se puede inicializar directamente, es necesario recurrir a los tipos heredados.

De `MWArray` heredan los siguientes tipos (directa o indirectamente):

- `MWNumericArray` – Representa un array de valores numéricos.

- `MWLogicalArray` – Representa un array de valores lógicos.
- `MWCellArray` – No se asocia a .NET.
- `MWStructArray` – No se asocia a .NET.
- `MWCharArray` – Representa un array de caracteres.

En el procesado de señales, el tipo heredado que más se usará es `MWNumericArray`. Para convertir un array numérico de tipo nativo de C# a un `MWNumericArray`, simplemente es necesario pasarlo por parámetro al constructor. El tipo `MWNumericArray` contiene una infinidad de sobrecargas en el constructor para poder pasar por parámetro arrays de muy distintos tipos.

Este `MWNumericArray` se utilizará para las funciones importadas de MATLAB, y el resultado será necesario convertirlo de nuevo en un tipo nativo de C#. Para ello las variables de tipo `MWNumericArray` disponen de dos métodos interesantes: `ToVector()` y `ToArray()`. El método `ToVector()` convierte el array a un System.Array de tipo `double` unidimensional. El método `ToArray()` convierte los datos a un System.Array de tipo `double` bidimensional (para matrices). Posteriormente se puede realizar una conversión explícita de System.Array a `double[]` para disponer de los datos en un tipo nativo de C#.

Sobrecarga de las funciones importadas

Las funciones importadas de MATLAB no van a tener una cabecera idéntica a la establecida en MATLAB. Se crean varias sobrecargas de cada función, en función del número de parámetros de entrada y salida que se usen.

La sobrecarga más sencilla es aquella en la que no hay parámetros de entrada ni de salida. La más compleja, es aquella en la que están todos los parámetros de entrada y todos los parámetros de salida. No obstante, conociendo este detalle, es muy intuitivo el funcionamiento de la función en C#, por lo que no se va a explicar aquí. Para más información recurrir a [23].

Efectos desarrollados en este software

A modo de demostración, se han desarrollado algunos efectos de sonido sencillos en la aplicación. Se pretende demostrar además lo sencillo que resulta incluir rutinas MATLAB para el desarrollo de un efecto de sonido.

Amplificación

Para amplificar una señal, simplemente es necesario multiplicar cada muestra por un factor numérico. Solamente es necesario tener en cuenta que los valores nunca deben superar el máximo y el mínimo, y por lo tanto deben verse recortados. Si esto no se tiene en cuenta, la máquina puede interpretar valores falsos que van a introducir aún más distorsión.

En la aplicación, la amplificación se ha desarrollado como ejemplo de efecto que tiene formulario de configuración. Cuando se crea el efecto, aparece una ventana donde se pueden modificar los parámetros más importantes. Estos parámetros se pueden modificar más tarde a través de la misma ventana.

Búsqueda de máximos

Como ejemplo de analizador de señales que muestra resultados por pantalla, se ha implementado este efecto de sonido. Realmente, no se trata de un efecto de sonido propiamente dicho por el hecho de que analiza la señal original pero no la modifica.

Este analizador, una vez que termina de procesar toda la señal, muestra en un formulario por pantalla los valores de amplitud máximos de ambos canales. Estos datos son útiles a la hora de normalizar una señal de audio.

Para normalizar una señal de audio simplemente es necesario combinar el algoritmo de búsqueda de máximos con el de amplificación. Cuando se analiza una señal, el valor máximo de amplitud tendrá un valor en decibelios negativo con respecto a la amplitud máxima. Si se desea normalizar solamente será necesario amplificar la señal el mismo valor en decibelios anterior pero con signo positivo.

Silenciado

Este efecto simplemente anula a 0 todas las muestras para generar silencio. Es útil para eliminar trozos de la señal que no interesan.

Eco simple IIR

Como ejemplo de efecto que implementa un sistema con memoria se ha implementado un sencillísimo sistema de eco. Este eco simplemente recurre a la salida generada en la iteración anterior y la suma atenuada por un valor α a la entrada actual. De esta forma se genera una señal resultante que simula un efecto de eco. En la Figura 3.54 se muestra lo que sería la estructura de este sistema:

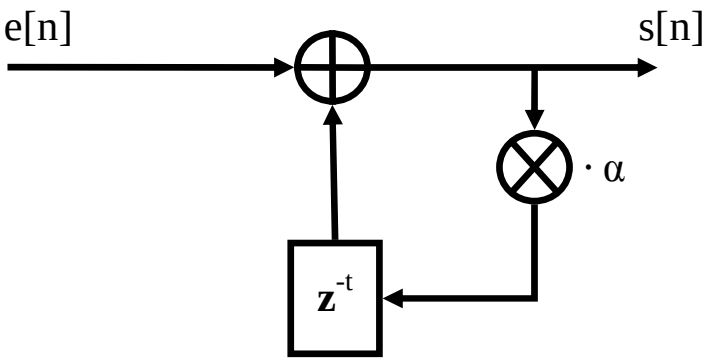

Figura 3.54.- Estructura del sencillo eco IIR implementado

Donde *t* es el tiempo que transcurre entre repeticiones en muestras, y α el factor atenuante de cada repetición. La respuesta libre en teoría es infinita, aunque en la práctica, al disminuir exponencialmente se ve limitada por la cuantización.

Filtrado mediante rutina MATLAB

Como ejemplo de rutina de más complejidad que combina formulario de configuración e inclusión de rutinas MATLAB, está este efecto. Consiste en un filtro configurable que se genera mediante funciones MATLAB para la generación de coeficientes y aplicación de filtros. Estas funciones son las mismas que se utilizaron en la asignatura de Laboratorio de Audio Digital.

Este efecto simplemente genera unos coeficientes para un filtro de tipo FIR en función de los parámetros establecidos en la ventana de configuración. Una vez que los coeficientes del filtro están establecidos, mediante una rutina de procesado se aplican sobre los búferes de audio consecutivos para procesar toda la señal.

Análisis espectral mediante rutina MATLAB

Por último, como ejemplo práctico de rutina de procesado basada en funciones MATLAB, se ha implementado un analizador de espectro. Este espectro se muestra en una gráfica tipo MATLAB, en escala vertical logarítmica, y utilizando un enventanado tipo Hanning.

Este analizador de espectro va calculando espectros de potencia de los distintos búferes consecutivos, y los promedia para mostrar el resultado final. Este método de promediado se llama método de Barlett, y es adecuado para el análisis espectral de señales aleatorias.

3.10.- Diagrama esquemático de la aplicación

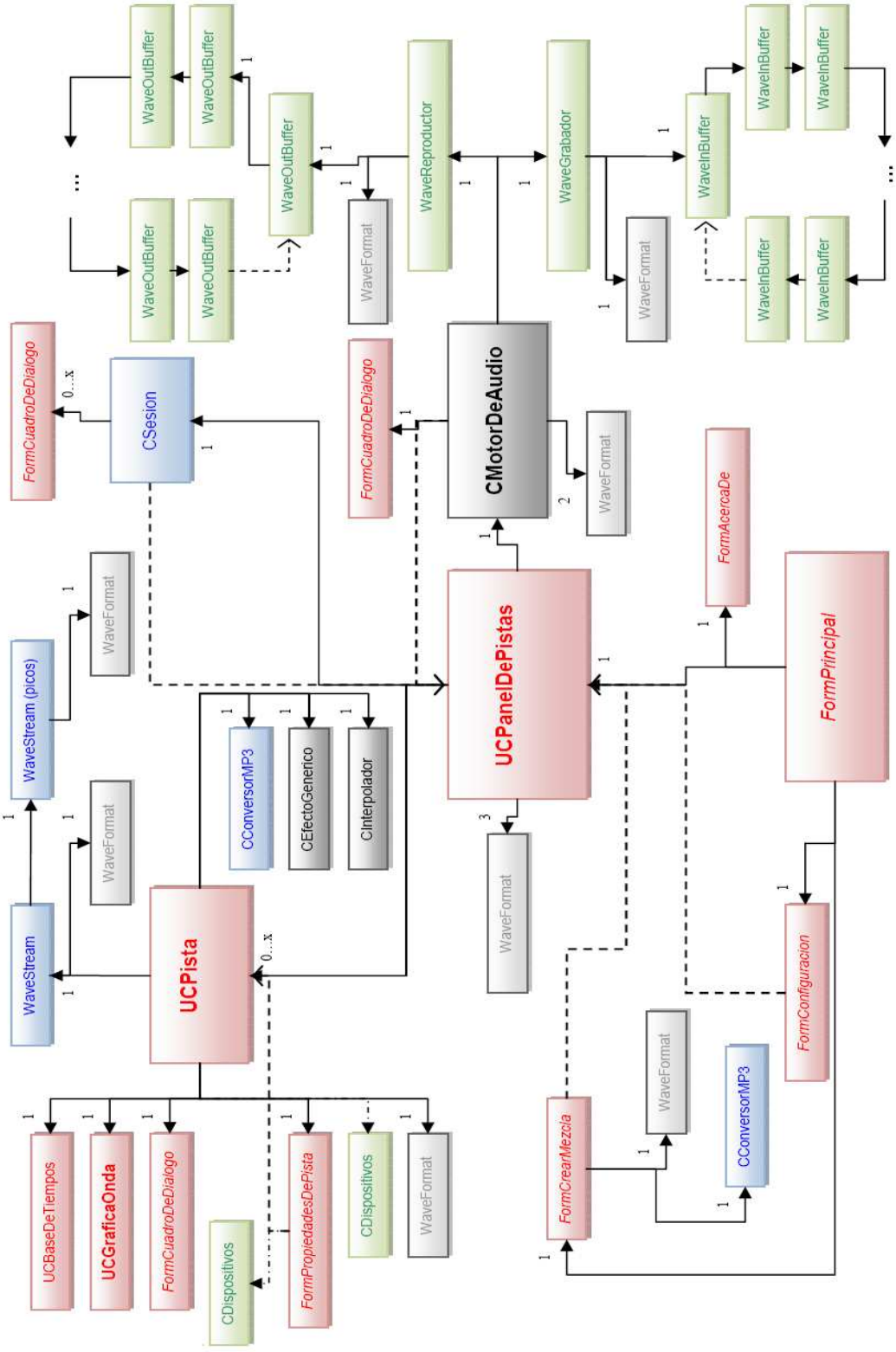

Figura 3.55.- Diagrama de bloques de las clases de la aplicación

Nomenclatura utilizada

En el diagrama de la Figura 3.55 se ha utilizado la siguiente notación:

Significado de las flechas

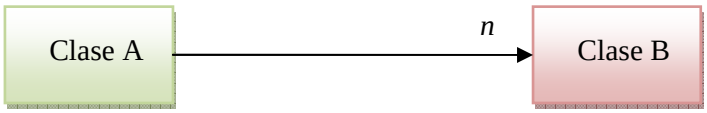

Figura 3.56.- Relación de contenencia

La Figura 3.56 representa una relación de contenencia. Quiere decir que la clase A contiene *n* instancias de la clase B. Por tanto, al eliminar de memoria la clase A, será necesario eliminar también las *n* instancias de clase B.

Si en *n* aparece "0…x", quiere decir que el número de instancias varía en tiempo de ejecución y pueden ser 0 o más hasta un límite indefinido.

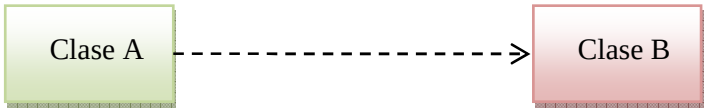

Figura 3.57.- Relación de referencia

La Figura 3.57 representa una relación de referencia. Es decir, la clase A contiene una referencia en forma de puntero a la clase B, aunque la clase B no esté instanciada dentro de la clase A. Esto se utiliza para recurrir a una determinada clase desde puntos distintos a donde se instanció.

Es importante comprender que la clase A sólo contiene un puntero a la clase B, no contiene una variable de tipo B.

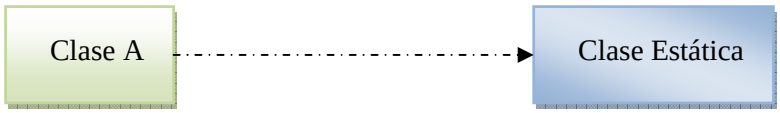

Figura 3.58.- Utilización de clase estática

La Figura 3.58 representa la utilización por parte de la clase A de una clase estática determinada. Esta clase estática no necesita ser instanciada, por lo se trata de un caso distinto a los anteriormente mencionados.

Colores de las clases

Las clases en rojo son aquellas que disponen de una representación por pantalla. Dicho de otro modo, son aquellas con las que el usuario va a poder interaccionar directamente. Se incluyen por supuesto los formularios, y además los controles de usuario.

Las clases en azul son aquellas que se comunican mediante ficheros. En el apartado 3.7 se explican qué clases se utilizan para acceder a los distintos tipos de ficheros.

Las clases en verde son aquellas que se comunican con los dispositivos de audio a través de la API winmm.dll. Pueden comunicarse para obtener información acerca de los dispositivos (como hace `CDispositivos`), para inicializar la comunicación con algún dispositivo, o para grabar y reproducir.

Las clases en gris son aquellas que a pesar de ser clases, se comportan como si fueran estructuras. En este caso, simplemente es de este tipo la clase `WaveFormat`, que como métodos sólo contiene el constructor.

El resto de clases que no cumplen ninguna de las características anteriores se colorean en negro.

Tipo de letra

Las clases en negrita son aquellas que tienen una mayor complejidad y forman núcleos importantes de un grupo concreto de clases menores.

Las clases en *cursiva* son las clases heredadas de System.Windows.Forms, es decir, los formularios.

Comentarios generales acerca del diagrama

FormPrincipal

En el diagrama se observa que la clase `FormPrincipal` no está instanciada en ninguna otra. Esto es así porque `FormPrincipal` es el formulario inicial del programa.

Además, dentro de `FormPrincipal` existen formularios secundarios, como `FormConfiguracion` o `FormCrearMezcla`. Ambos recurren mediante un puntero a la clase `UCPanelDePistas` instanciada en `FormPrincipal`.

UCPanelDePistas

La instancia de `UCPanelDePistas` va a ser el núcleo de todo el programa. Esta clase va a gestionar todas las pistas, las sesiones de usuario y además la reproducción y grabación de audio. Para realizar estas tareas, se recurre a clases intermedias como `CMotorDeAudio`, o `CSesion`. `UCPanelDePistas` contiene un número indefinido de pistas de tipo `UCPista`, y se encarga de controlarlas y hacerlas accesibles desde el resto del programa.

UCPista

La clase `UCPista` implementa una pista de audio, unidad fundamental de trabajo en la aplicación. Contiene elementos gráficos complejos como `UCGraficaOnda` y `UCBaseDeTiempos`. Pero además, `UCPista` recurre a formularios para mostrar sus propiedades, o para representar barras de progreso.

Por otro lado, `UCPista`, a través de `WaveStream`, ofrece una interfaz para el acceso completo a ficheros WAV abstrayendo al resto del programa de los detalles relacionados con el formato WAV.

Además, `UCPista` contiene una instancia de tipo `CEfectoGenerico`, y una instancia de tipo `CInterpolador`. Ambas serán utilizadas por `CMotorDeAudio` durante la reproducción para la aplicación de efectos a tiempo real y para la interpolación a tiempo real.

CMotorDeAudio

Esta clase va a ser la encargada de la reproducción y la grabación de las pistas contenidas en `UCPanelDePistas`. Para poder acceder al panel, contiene un puntero a la clase que contiene a ella misma.

Para la reproducción, `CMotorDeAudio` hace uso de `WaveReproductor`, y para la grabación de `WaveGrabador`. Conviene hacer notar la lista enlazada circular de clases de tipo `WaveOutBuffer` y `WaveInBuffer` que se ha creado para garantizar un buen rendimiento.

El número de instancias de búferes varía en función de los parámetros configurados por el usuario, y el tamaño de muestras que alberga cada uno también.

CSesion

La clase `CSesion`, mediante un puntero al panel que la contiene, puede agregar o eliminar pistas de la manera necesaria. Además, accede al disco para leer ficheros de tipo .PFC que contienen datos sobre la sesión actual.

Combinando ambas capacidades almacena o carga el estado actual de la aplicación en ficheros para poder continuar con la misma sesión una vez cerrada.

3.9.- Descripción de la solución .NET en Microsoft Visual C# 2005 Express

En el apartado 3.3 se ha descrito brevemente la arquitectura de la Plataforma .NET y los conceptos básicos asociados a ella. Sin embargo, no se comentó qué herramienta se utiliza para programar en C#: Microsoft Visual C# 2005 (Express). La versión Express es gratuita, es más limitada que la profesional, y su licencia permite crear cualquier tipo de programas, incluso comerciales.

El conjunto de archivos de código fuente, de ficheros auxiliares, etc. que conforma una aplicación en .NET se llama *solución.* Una solución consta de uno o más proyectos, siendo uno de ellos el proyecto de inicio. Un proyecto puede ser de muchos tipos, aunque los más comunes son los de tipo “Aplicación de Windows” y los de tipo “Biblioteca de clases”. Los primeros se compilan en un fichero EXE ejecutable, y los segundos en un fichero DLL que contiene clases reutilizables.

En concreto, la solución .NET que alberga el software desarrollado dispone de 5 proyectos:

- Proyecto ComponentesPrincipales (biblioteca de clases)
- Proyecto ConversionDeFormatos (biblioteca de clases)
- Proyecto Efectos (biblioteca de clases)

- Proyecto InterfacesDeUsuario (aplicación de Windows ejecutable)
- Proyecto InterfazES (biblioteca de clases)

Proyecto ComponentesPrincipales

Este proyecto alberga los controles de usuario y las clases asociadas que casi conforman toda la aplicación. Con su creación se ha pretendido separar los controles de usuario del formulario principal del programa, aislando en cierto modo la interfaz y la implementación. Las clases más importantes que conforman este proyecto son:

- `UCPanelDePistas`
- `UCPista`
- `UCGraficaOnda`
- `UCBaseDeTiempos`
- `CMotorDeAudio`
- `CSesion`
- `FormConfiguracion`
- `FormCrearMezcla`
- `FormCuadroDeDialogo`
- `FormPropiedadesDePista`

El espacio de nombres asociado a este proyecto es ComponentesPrincipales. Cuando se compila genera un fichero .DLL que contiene las clases anteriormente mencionadas.

Proyecto ConversionDeFormatos

Se ha aprovechado una de las clases para realizar conversiones MP3-WAV y WAV-MP3. Esta clase es `CConversorMP3`, que contiene varios métodos útiles para realizar dicha conversión. Puesto que la mayoría de las clases de este proyecto no han sido implementadas en la creación de esta aplicación, no se van a mencionar.

Proyecto Efectos

Este proyecto contiene todo lo asociado a los efectos de audio y el procesamiento de señal. Contiene todos los efectos creados y sirve como nexo de unión entre C# y MATLAB.

Es importante mencionar que entre sus referencias están el componente .NET creado por MATLAB Builder, y la librería de tipos MWArray.DLL, que contiene todos los tipos nativos de MATLAB.

Contiene las siguientes clases:

- `CEfectoGenerico`
- `CInterpolador`

Proyecto InterfacesDeUsuario

Este proyecto contiene la interfaz de usuario del programa, es decir, el formulario principal. La intención es hacer de este formulario una interfaz totalmente sustituible para la aplicación. Contiene dos formularios:

- `FormPrincipal`
- `FormAcercaDe`

Además tiene otra función secundaria, que es facilitar el flujo del programa. Dado que cualquier acción se hace a través de este formulario, se tiene totalmente controlado de donde parte la orden y hacia donde se dirige.

Este proyecto es el único que genera un ejecutable, ya que es el proyecto de inicio de toda la solución.

Proyecto InterfazES

En este proyecto se ha integrado la interfaz entrada salida con los dispositivos de audio y los ficheros WAV. Se trata de clases pensadas para el flujo de audio, bien sea entrante o saliente. Además, todas las referencias a la API winmm.dll se realizan en la clase `WaveNative` incluida en este proyecto.

El proyecto consta de las siguientes clases:

- `WaveGrabador`
- `WaveInBuffer`
- `WaveReproductor`
- `WaveOutBuffer`
- `WaveNative`

- `CDispositivos`

Compilación

Tras compilar toda la solución, se generarán varios ficheros DLL y EXE. Además, a estos hay que incluir las DLL externas de las que hace uso la aplicación. Por tanto, el conjunto de ficheros que conforman la aplicación van a ser los siguientes:

- PFC.EXE → Resultado de compilar el proyecto InterfacesDeUsuario.
- ComponentesPrincipales.DLL
- InterfazES.DLL
- Efectos.DLL
- ConversionDeFormatos.DLL
- MWArray.DLL → Librería de formatos MATLAB
- FuncionesMatlab.DLL → Librería de funciones .m.
- Lame_enc.DLL → Necesaria para la conversión MP3.
- Madlldlib.DLL → Necesaria para la conversión MP3.
- Libsndfile.DLL → Necesaria para la conversión MP3.

Estas librerías deben estar instaladas en el mismo directorio que el fichero PFC.EXE.

Capítulo 4 - Conclusiones

Con la intención de reunir en un solo capítulo los aspectos más importante de todo el desarrollo, se ha incluido en esta memoria el capítulo presente denominado “Conclusiones”.

4.1.- Conclusiones generales

El objetivo inicial del proyecto era crear un software capaz de comunicarse con una interfaz USB capturadora de audio (ver apartado 1.1). Este objetivo se ha cumplido con éxito, ya que se ha logrado establecer correctamente dicha comunicación. Pero además, se han incluido una serie de mejoras que dan al proyecto un importante valor añadido.

Uso de protocolo de comunicación estándar

La mejora más significativa es el uso de un protocolo de comunicación estándar ya existente: la especificación USB Audio 1.0 [8]. En Windows, el driver usbaudio.sys permite la comunicación con los dispositivos de audio que cumplen esta especificación. Además, existe un driver para cada plataforma realizando la misma función. Esto conlleva una ventaja importantísima tanto para el hardware como para el software:

- El hardware se ve beneficiado con el uso de este protocolo, ya que cualquier aplicación de audio en cualquier plataforma podrá controlarlo sin problemas. Es decir, el software desarrollado no se hace imprescindible para su funcionamiento. De hecho, el hardware tal y como ha sido creado actúa como una tarjeta de sonido más instalada en el PC.
- El software también ha salido beneficiado con esta mejora, ya que la estandarización permite al sistema operativo reconocer el hardware como si fuera un dispositivo de audio más. De esta forma, el sistema de comunicación desarrollado permite al software controlar no solo el hardware específico de este proyecto, sino cualquier tarjeta de sonido instalada en el PC.

Por otro lado, el uso de la especificación ha repercutido en una mayor independencia entre el desarrollador hardware y desarrollador software. La depuración del software se ha realizado con la tarjeta de sonido incluida en el PC, y la depuración del hardware se ha realizado mediante programas para edición de audio ya desarrollados. Gracias al uso del protocolo estándar, se tenía la garantía de que si ambos desarrollos funcionaban correctamente en esas condiciones, funcionarían de forma conjunta sin problemas.

Además, el tiempo que se ha ahorrado en diseñar e implementar el protocolo de comunicación, se ha podido invertir en otros aspectos de los desarrollos. De esta forma, tanto el hardware como el software han terminado por ser más robustos y completos.

Funcionalidades añadidas al software

Tal y como se estableció en el apartado 1.1, el objetivo principal del desarrollo software era la comunicación por USB con el hardware y el almacenamiento de los datos entrantes en ficheros de audio independientes. Este objetivo se ha cumplido en su totalidad, pero además se han incluido una serie de funcionalidades que aportan un valor añadido al proyecto. A continuación se enumeran la gran mayoría de estas mejoras:

- Posibilidad de reproducir y registrar audio de forma simultánea en un número ilimitado de pistas.
- Posibilidad de reproducir un fichero de audio en un segundo distinto al inicial, para combinar distintas pistas en distintos momentos de la forma que más interese.
- Representación gráfica de la forma de onda de cada fichero de audio, disponiendo controles de selección, zoom, etc.
- Posibilidad de utilizar todas las tarjetas de sonido instaladas en el PC simultáneamente, tanto para grabación como para reproducción.
- Posibilidad de reproducir ficheros de audio con frecuencias de muestreo distintas simultáneamente.
- Distintos métodos de interpolación a tiempo real para conseguir el objetivo anterior.
- Posibilidad de controlar el volumen, el balance, el dispositivo de entrada y salida, etc. para cada pista de forma independiente. Además

se dispone de un control de volumen general que afecta a todas las pistas.

- Posibilidad de usar tanto ficheros WAV como ficheros MP3 en cada pista.
- Gestión de sesiones para poder continuar con una determinada mezcla después de cerrar el programa. Además, se permite exportar toda la sesión mezclada en un WAV o en un MP3.
- Uso de efectos de sonido tanto a tiempo real como aplicados sobre un determinado fichero de audio de forma permanente.
- Inclusión de rutinas de MATLAB para el procesado de señales.

Desde un punto de vista musical, el software tiene muchas más posibilidades de lo que los objetivos iniciales del proyecto determinaban. Permite la elaboración de temas musicales con casi tanta facilidad como los programas comerciales, ya que permite grabar por separado distintos instrumentos para luego escucharlos todos de forma simultánea.

Pero además, desde el punto de vista de un ingeniero técnico de telecomunicaciones, la aplicación resulta llamativa por la combinación de un entorno agradable y funcional, y las técnicas de procesado de señal incluidas. La integración de MATLAB con la Plataforma .NET aporta además un amplio abanico de posibilidades para futuros proyectos. De hecho, uno de los mayores problemas que supone programar en MATLAB es la creación de un buen entorno gráfico, algo resuelto con esta integración.

Documentación para desarrollo software

Aunque a lo largo de toda la carrera son muchos los conocimientos adquiridos, y la base conseguida para realizar proyectos como este, resulta imposible abarcar todos los aspectos necesarios para construir, por ejemplo, un software como el que se ha desarrollado. Debido a esto, es necesaria una importante labor de documentación y de síntesis de la información encontrada. Son cuatro los aspectos que han requerido una mayor inversión de tiempo en búsqueda de información:

- La programación orientada a objetos en C#
- La comunicación a través de la API con los dispositivos de sonido
- La interpolación de señales a tiempo real

- La integración de MATLAB y C#

Programación Orientada a Objetos en C#

Durante la carrera, son varias las asignaturas donde se aprenden conceptos básicos de programación, e incluso se llegan a estudiar las *clases* (ver nota al pie número [4]). Sin embargo, es justo en ese punto donde comienza el siguiente paso en el estudio de la programación: la *Programación Orientada a Objetos (POO)*. Fue necesario documentarse acerca de este tipo de programación, sus ventajas, las herramientas que proporciona en el desarrollo de aplicaciones y su correcto uso para el desarrollo de este software.

La Programación Orientada a Objetos proporciona un nivel de abstracción mayor mediante unos entes llamados "objetos". Estos objetos se modelan con clases, y van a ser entidades independientes, con unos parámetros internos y unos determinados métodos de utilización, que interactúan entre sí. El lenguaje C# está orientado a objetos, y por tanto todas estas características van a estar ligadas directamente al estudio del lenguaje. Para comprender el funcionamiento de este tipo de programación, y aplicarla en C# se recurrió principalmente a los recursos bibliográficos [1], [5] y [26].

Comunicación de C# con la tarjeta de sonido

Por otro lado, para el desarrollo de este software, es necesario también conocer la forma de comunicar C# con la tarjeta de sonido. Puesto que se trata de algo muy específico, en ninguno de los recursos anteriores se encontró información para realizar esta tarea. En consecuencia fue necesario documentarse por distintas vías, y sintetizar en la aplicación los conocimientos adquiridos. En los artículos [14] y [15], publicados por Ianier Munoz en *www.thecodeproject.com*, se explicaba como usar la API winmm.dll desde Windows para grabación y reproducción de audio. Estos dos recursos, por tanto, fueron de especial ayuda para esta tarea. Pero además, los recursos bibliográficos [9], [11], [18] y [21] complementaron la información y posibilitaron la realización de un sistema más completo y eficiente.

Interpolación de señales

Otro punto que ha necesitado de cierta investigación es la interpolación de señales. En primer lugar se encontró en el famoso libro de teoría de señales *Oppenheim* [17] la teoría necesaria para comprender e implementar el *retenedor de orden cero* (apartado 3.8). Este interpolador es el más sencillo, y gracias a esto es el más apropiado para realizar interpolación a tiempo real si la calidad no es un parámetro crítico.

Posteriormente, en el mismo libro [17] se encontró la teoría necesaria para implementar un interpolador mediante convolución con sincs a tiempo real. En [20] se encontró una implementación en C orientada a DSP de un algoritmo similar, algo que también fue de gran ayuda. La mejora de la respuesta en frecuencia del filtro paso-bajo mediante enventanado Hanning, también fue una idea surgida de las distintas fuentes consultadas.

Por último, se implantó también un sistema de interpolación basado en una función propia de MATLAB. Esta función es 'resample', debidamente explicada en el apartado 3.8. Esta función de MATLAB se descubrió mediante el estudio de las funciones incluidas en el toolbox de procesado de señales.

Inclusión de rutinas MATLAB en C#

Para lograr con éxito la combinación con MATLAB y C# fue imprescindible recurrir a la ayuda de MATLAB [23], donde aparecen explicaciones detalladas y ejemplos diversos. Esta tarea no resultó especialmente difícil gracias a la cantidad de información encontrada en dicha fuente.

4.2.- Ampliando el alcance del proyecto

Aunque el proyecto en conjunto está terminado con numerosas mejoras y características novedosas, resulta inevitable plantearse en qué aspecto se podría haber trabajado en caso de haberse tratado de un proyecto de mayor envergadura. Debido a la limitación en tiempo y recursos del proyecto, no se pudo seguir ampliando indefinidamente.

El objetivo de este punto no es explicar qué ha faltado para terminar el proyecto, ya que este proyecto está acabado. Lo que se pretende es dejar abiertas unas interesantes líneas de trabajo por si se deseara en un futuro realizar una segunda versión mejorada de este proyecto.

Estas líneas de trabajo se dividen en tres tipos: las asociadas al hardware, las asociadas al software, y las asociadas a ambos.

Posibles líneas de trabajo para mejorar el hardware

- Utilizar un microcontrolador de mayores prestaciones que estuviese algo más orientado al desarrollo de sistemas de audio (DSP). Esto facilitaría la ampliación de canales. Además resultaría sencillo ampliar el formato de cada canal de 16 Khz-8 bits a 44.1 Khz-16 bits, obteniendo así una calidad próxima a la del CD.

- Implementar un dispositivo de audio *full-duplex* (flujo de datos bidireccional) capaz de registrar audio, pero también de reproducirlo. Se planteó durante bastante tiempo esta posibilidad con el PIC18F4550 utilizando la salida PWM de la que dispone. No obstante, debido a la limitación en velocidad del PIC, y a la mala calidad de audio que se hubiese obtenido, se descartó esta opción.

Posibles líneas de trabajo para mejorar el software

- Cada efecto de audio se podría almacenar en una DLL distinta para poder ampliar la lista de efectos con sólo añadir una DLL. Esta gestión de DLL's es posible utilizando las clases del espacio de nombre `System.Reflection`.
- Otra posible mejora es incluir opciones de edición. Gracias al diseño del software, que ya incluye opciones de selección, no sería nada difícil incluir opciones de edición como suprimir audio, copiar y pegar, o convertir en silencio.
- Una posible mejora es añadir la posibilidad de deshacer operaciones realizadas sobre las pistas (el típico Undo). No obstante, es necesario tener en cuenta que esta mejora implicaría un sistema de gestión de ficheros temporales mucho más complejo, y por lo tanto el esfuerzo invertido sería enorme.
- Una opción que mejoraría la versatilidad del software es añadir compatibilidad con plugins VST[11] para el procesado de la señal. Sin embargo, esta tarea no es nada fácil, ya que un host VST en C# requiere mucha tarea de búsqueda y síntesis de información si se parte de conocimientos básicos. Un host VST en C# podría haber sido de forma individual un auténtico Proyecto Fin de Carrera de informática.
- Por último, el rendimiento gráfico del programa hay que reconocer que es mejorable (aunque suficiente). Para mejorar esto se puede recurrir a librerías gráficas de alto rendimiento como DirectX o OpenGL. No obstante, la aplicación de estas librerías supone un gran esfuerzo y merecería la pena únicamente si el resto del software está muy mejorado.

[11] Las siglas VST significan Tecnología de Estudio Virtual (*Virtual Studio Technology)*, y representan el estándar lanzado por Steinberg para creación de efectos de sonido y sintetizadores.

Posibles líneas de trabajo para mejorar el proyecto conjunto

- En primer lugar, si el hardware se amplia con más entradas, resulta conveniente una gestión más eficaz que la que proporciona usbaudio.sys. Por tanto se podría desarrollar un driver propio para Windows (o incluso para Linux) más adaptado al hardware. No obstante, esta mejora tendría sentido sólo no en el caso de haber desarrollado un hardware mucho más complicado, ya que tal y como se decidió hacer, era mucho más sencillo y eficiente utilizar usbaudio.sys.
- Sería una buena idea incluir compatibiliad ASIO (ver nota al pie número [7]) al hardware y al software. Este estándar desarrollado por Steinberg aumenta la eficacia de la gestión de audio multicanal reduciendo la latencia. No obstante, esta mejora supone un enorme esfuerzo, y supone renunciar al uso de usbaudio.sys. Para comenzar a trabajar con ASIO, lo más conveniente es recurrir a [22].
- Por último, se podría añadir compatibilidad MIDI[12] al hardware y al software. Sin embargo, al igual que alguna de las mejoras anteriores, es algo laborioso de implementar y supondría una gran inversión de tiempo. Como recurso interesante se ha aportado [9].

[12] Las siglas MIDI significan Interfaz Digital para Instrumentos Musicales (Musical Instrument Digital Interface). Es un protocolo de comunicación estándar para la comunicación de instrumentos musicales.

Valoración de las líneas futuras de trabajo

La inclusión de todas estas mejoras podría llegar a convertir el conjunto hardware-software en un producto de gama alta capaz de competir en el mercado del audio profesional. No obstante, sería un proceso lento basado en el añadido de pequeñas mejorías para ir alcanzando un mayor nivel de calidad poco a poco.

Si se estudia la evolución de cualquier software o hardware comercial de gama media-alta, se observará que una idea no se desarrolla desde el principio con toda su complejidad. Uno de los ejemplos más significativos quizá sea el programa *Adobe Audition 2.0*. Este software ha sufrido una notable evolución desde 1996 hasta 2007, derivando inicialmente de *Cool Edit 96*, que no incluía edición multipista ni compatibilidad MP3. Posteriormente, fue evolucionando a través de *Cool Edit 2000, Cool Edit Pro, Adobe Audition 1.0*, y finalmente *Adobe Audition 2.0*. Esta última versión del software incluye compatibilidad ASIO, MIDI, gestión multipista, compatibilidad VST e infinidad de opciones útiles para la creación musical.

Pero además, en el aspecto hardware, basta con observar la evolución de la familia de tarjetas de sonido *Sound Blaster* de Creative. Desde la primera *Sound Blaster 1.0* lanzada en 1989, que reproducía audio a 22050Hz-8bit y grababa a 11025Hz-8bit, ha habido una larga evolución hasta hoy. Actualmente, el último modelo es la *Sound Blaster X-FI*, capaz de muestrear audio a velocidades de muestreo profesionales con 24 bit de resolución, además de incluir un procesado por DSP para aplicación de efectos en tiempo real.

Por tanto, este Proyecto Fin de Carrera, aun siendo completamente funcional, no deja de ser una idea inicial, un punto de partida para desarrollar un sistema más complejo y competente en el mercado. Hay que tener en cuenta que no por ello este trabajo pierde valor, sino al contrario, ya que como dijo un gran sabio griego, *«el comienzo es la parte más importante de cualquier acto»*.

BIBLIOGRAFÍA

[1] Albahari Joseph Threading in C# [Internet] // Web personal de Joseph Albahari. - 2007. - 16 de Junio de 2007. - *http://www.albahari.com/threading/*.

[2] Axelson Jan USB Complete - Lakeview Research, 2005. – Edición Tercera.

[3] Carver Richard H. y Tai Kuo-Chung Modern Multithreading - Wiley-Interscience, 2005.

[4] Charte Ojeda Francisco Visual C# .NET - Anaya Multimedia, 2002.

[5] Ferguson Jeff, Patterson Brian y Beres Jason La Biblia de C# - Anaya Multimedia, 2002.

[6] Fernández-Caro Belmonte Jaime PICUsb [Internet] // HobbyPic. - Diciembre de 2005. - 15 de Julio de 2007. - *http://www.hobbypic.com/*.

[7] Gilad-ap Aumplib: C# Namespace and Classes for Audio Conversion [Internet] // The Code Project. - Octubre de 2004. - *http://www.codeproject.com/cs/media/Aumplib.asp*.

[8] IBM Corporation, Microsoft Corporation, Logitech, y otros. Universal Serial Bus Device Class Definition for Audio Devices [Internet] // Universal Serial Bus. - Marzo de 1998. - *www.usb.org*.

[9] Jordá Puig Sergi Audio Digital y MIDI. Guías monográficas - Madrid : Anaya Multimedia, 1997.

[10] Microsoft Corporation Getting started with WDM audio drivers [Internet] // Windows Hardware and Driver Control. - Enero de 2006. - *www.microsoft.com/whdc/device/audio/wdmaud-drv.mspx*.

[11] Microsoft Corporation Multimedia Functions [Internet] //Microsoft Developer Network. - Junio de 2006. - *http://msdn2.microsoft.com/en-us/library/ms712636.aspx*.

[12] Microsoft Corporation Universal Audio Architecture Hardware Design Guidelines [Internet] // Windows Hardware and Driver Control. - Junio de 2006. - *www.microsoft.com/whdc/device/audio/UAA_HWdesign.mspx*.

[13] Molina Martínez Alberto y Piedra Fernández José Antonio Sistema de gestión y reconocimiento automático de imágenes de satélite. Aplicación de un clasificador neurodifuso. Proyecto Fin de Carrera - Almería : UAL, 2004.

[14] Munoz Ianier A full-duplex audio player in C# using the waveIn / waveOut APIs [Internet] // The Code Project. - Septiembre de 2003. - *http://www.codeproject.com/cs/media/cswavrec.asp*.

[15] Munoz Ianier A low-level audio player in C# [Internet] // The Code Project. - Agosto de 2003. - *http://www.codeproject.com/cs/media/cswavplay.asp*.

[16] National Instrument Manual de Usuario de LabWindows/CVI . - 2001.

[17] Oppenheim Alan V., Willsky Alan S. y Nawab S. Hamid Señales y Sistemas - Pearson Education, 1998. – Edición Segunda.

[18] Overton David Playing Audio In Windows Using waveOut Interface [Internet] // Insomnia Visions. - 2002. –

http://www.insomniavisions.com/documents/tutorials/wave.php

[20] Posadas Yogüe Juan Luis y Benet Gilbert Ginés Transformada Rápida de Fourier (FFT) e interpolación a tiempo real. Algoritmos y aplicaciones. [Internet]. - Julio de 1998. - *http://www14.brinkster.com/aleatoriedad/FFT.pdf*.

[21] Schwab Geoff Recording and playing sounds with the Waveform Audio Interface in .NET Compact Framework [Internet] // MSDN. - Enero de 2004. - *http://msdn2.microsoft.com/en-us/library/aa446573.aspx*.

[22] Steinberg ASIO. Audio Streaming Input Output Development Kit [Internet] . - 2005.- *http://www.steinberg.net/534+M52087573ab0.html*

[23] The MathWorks Inc. Ayuda de MATLAB R2006. - Enero 2006

[24] Villa Brieva Ricardo y Villatorio Machuca Francisco R. Programación en Visual Studio .NET bajo C# de Aplicaciones Gráficas [Internet]. - Febrero de 2002. - *www.lcc.uma.es/pfc/37.pdf*.

[25] Wikipedia. Artículo sobre .NET de Microsoft.- Agosto de 2007.- *http://es.wikipedia.org/wiki/.NET*

[26] Wille Christoph C# - Prentice Hall, 2001.