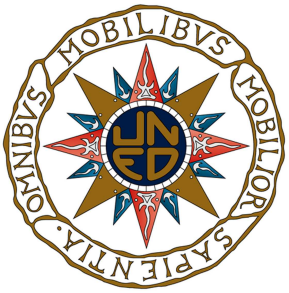


UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

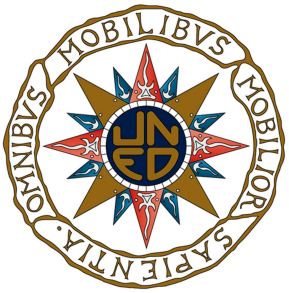
Proyecto de Fin de Grado en Ingeniería Informática

DESARROLLO DE UN PROTOTIPO DE MULTIEFECTOS DIGITAL

FIDEL ALONSO HERRERA CASTRO

Dirigido por: Dr. ALFONSO URQUÍA MORALEDA

Curso: 2017/2018 (Junio)



DESARROLLO DE UN PROTOTIPO DE MULTIEFECTOS DIGITAL

Proyecto de Fin de Grado en Ingeniería Informática
de modalidad *específica*

Realizado por: FIDEL ALONSO HERRERA CASTRO

Dirigido por: Dr. ALFONSO URQUÍA MORALEDA

Fecha de lectura y defensa.....

Me gustaría agradecer a varias personas la colaboración prestada en este trabajo

A mi padre y mi abuela, por los recursos económicos aportados
A mi hermano, por el apoyo moral
A mi amigo Carlos, músico, quien tuvo la idea original de realizar este proyecto

Resumen

En este trabajo de fin de Grado se hace uso de la librería JUCE y del kit de desarrollo de bajo coste para el procesador OMAP-L138 de Texas Instruments para elaborar un software destinado a procesamiento digital de señales de audio.

El trabajo se divide en dos partes. En la primera de ellas se diseña e implementa un retardo modulado, primero en forma de plugin VST y posteriormente como un programa ejecutable en el núcleo C674x del OMAP-L138. En la segunda parte se hace lo mismo con un circuito analógico de saturación, simulando el funcionamiento del circuito en tiempo real mediante un filtro digital de ondas.

Se incluye además código en C para configurar el codec de audio de la placa de desarrollo y varios módulos del procesador OMAP-L138, incluyendo un módulo de transferencia directa a memoria mediante el cual se envían las muestras capturadas usando el conversor A/D del codec a la memoria DDR2 incluida en la placa de desarrollo.

Para comprobar la funcionalidad de los plugins VST se usa el software Reaper en conjunto con una interfaz de audio, la MOTU Ultralite-mk3 hybrid, que proporciona los conversores A/D y D/A necesarios. Ambas versiones de cada uno de los programas tienen la capacidad de procesar bloques de muestras en tiempo real a medida que estos llegan del conversor A/D y devolverlos procesados al conversor D/A. Además, en el caso de las versiones en forma de plugin VST, se incluye una interfaz gráfica sencilla que permite modificar los parámetros de funcionamiento en tiempo real. El código y la documentación resultante pueden usarse como base para la implementación de otros efectos de audio o para mejorar el software presentado en funcionalidad y/o calidad.

En el caso del retardo modulado, se realiza en primer lugar el diseño de un retardo fraccionario para posteriormente integrarlo en una estructura más compleja que permita efectuar la modulación. Este enfoque permite un diseño en el dominio de la frecuencia, lo cual resulta ventajoso cuando se conoce la banda de frecuencias de interés. En el caso del circuito analógico, se realiza en primer lugar un análisis teórico del mismo. Posteriormente, el circuito se convierte en un filtro digital de ondas mediante el cual se simula su funcionamiento.

El resultado de ambos diseños se usa como guía para implementar el software. Para implementar los plugins VST se usan C++, la librería JUCE y Visual Studio 2017. Para implementar software que pueda ejecutarse en el núcleo C674x se usan C y Code Composer Studio.

El trabajo incluye pruebas de los cuatro programas obtenidos que muestran que la funcionalidad es aceptable y en concordancia con los objetivos iniciales.

Palabras clave

OMAP-L138, C674x, EDMA3, McASP, JUCE, VST, Filtros Digitales de Ondas, Plugins de Audio, Diferenciadores Digitales, DSP, Retardo Modulado, Retardo Fraccionario, Vibrato, Saturación, Distorsión.

Abstract

In this final Degree project I use the library JUCE and the low cost development kit for the Texas Instruments OMAP-L138 processor to elaborate a digital audio processing software.

The work is divided in two main parts. In the first one a modulated delay is designed and implemented, first in the form of a VST plugin and subsequently as a program executable in the C674x core of the OMAP-L138. In the second part the same is done with an analog overdrive circuit, simulating the operation of the circuit in real time by means of a wave digital filter.

C code is also included to configure the audio codec of the development board and various modules of the OMAP-L138, including a direct memory access module by means of which the samples captured using the A/D converter of the codec are sent to the DDR2 memory included in the development board.

In order to test the functionality of VST plugins the software Reaper is used in conjunction with an audio interface, the MOTU Ultralite-mk3 hybrid, which provides the necessary A/D and D/A converters.

Both versions of each program can process sample blocks in real time as they arrive from the A/D converter and send them back processed to the D/A converter. Also, in the case of VST plugin versions, a simple graphical interface is included which allows the modification of plugin parameters in real time. The resulting code and documentation can be used as a base for the implementation of other audio effects or to improve the presented software in functionality and/or quality.

In the case of the modulated delay, a fractional delay is first designed to subsequently integrate it into a more complex structure in order to make the modulation possible. This approach allows a frequency domain design, which is advantageous when the frequency band of interest is known.

In the case of the analog circuit, a theoretical analysis is first made. Subsequently, the circuit is converted into a wave digital filter by means of which the simulation is carried out.

The result of both designs are used as a guide to implement the software. In order to implement the VST plugins, C++, the JUCE library and Visual Studio 2017 are used. In order to implement software which can run in the C674x, C and Code Composer Studio are used.

The project includes tests of the four resulting programs which show that the obtained functionality is acceptable and in accordance with the initial objectives.

Keywords

OMAP-L138, C674x, EDMA3, McASP, JUCE, VST, Wave Digital Filters, Audio Plugins, Digital Differentiators, DSP, Modulated Delay, Fractional Delay, Vibrato, Overdrive, Distortion.

Índice general

Índice de figuras

CAPÍTULO 1

Introducción, objetivos y estructura

En este primer capítulo se establecen una serie de definiciones y conceptos que serán de utilidad a lo largo del resto del documento. En concreto, se definen las señales digitales y el tratamiento que se les aplica en este trabajo y se introducen las herramientas a utilizar, que serán concretadas en el capítulo siguiente.

1.1 Introducción

1.1.1 Señales continuas y señales discretas

Si una señal puede tomar todos los valores posibles en un rango finito o infinito, se dice que es una señal continua. En cambio, si la señal solo puede tomar un número determinado de valores, se dice que es discreta.

Si una señal toma infinitos valores en un intervalo de tiempo dado, se dice que es continua en el dominio del tiempo. Si por el contrario solo está definida en ciertos instantes de tiempo, se dice que es discreta en el dominio del tiempo.

Para que una señal pueda ser procesada digitalmente debe ser ser discreta en el tiempo y sus valores deben ser discretos. Este tipo de señales se llaman señales digitales.

Para convertir una señal analógica en una señal digital se sigue un proceso de muestreo y cuantificación. El muestreo consiste en tomar el valor de la señal analógica en determinados instantes. La cuantificación es un proceso de aproximación de los valores obtenidos y puede realizarse de forma simple mediante redondeo o truncamiento [1, pp. 8-9], [2, pp. 1-6].

En la figura 1.1 se muestra un ejemplo de señal analógica y su equivalente digital, después del muestreo y la cuantificación. Normalmente, los momentos en los que se realiza el muestreo son equidistantes un tiempo T y se dice que la señal ha sido muestreada con frecuencia $Fs = 1/T$. Nótese que en el caso de la figura mostrada la señal es muestreada 40 veces por segundo, por lo que cada muestra está separada $1/40 = 0.025$ segundos.

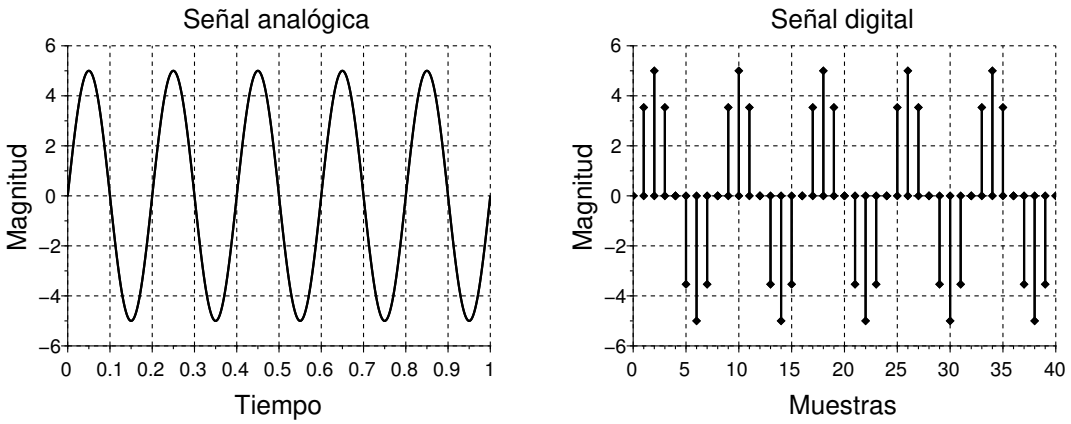


Figura 1.1: Señal analógica y su equivalente digital tras el proceso de muestreo.

1.1.2 El tratamiento digital de señales de audio

Una señal de audio analógica es normalmente una representación eléctrica de la velocidad del diafragma de un micrófono, aunque dicha representación también puede ser generada por otro tipo de transductor. Por ejemplo, las pastillas de guitarra u otros instrumentos similares utilizan la variación del campo magnético producida por la vibración de las cuerdas para generar señales eléctricas que pueden ser posteriormente convertidas en sonido.

El tratamiento digital de una señal de audio ofrece varias ventajas respecto a su tratamiento analógico. Pueden destacarse dos:

- La calidad de reproducción de un sistema de sonido digital bien diseñado solo depende de la calidad de los procesos de conversión.
- La conversión de una señal de audio al dominio digital permite procesos que no son posibles con una señal analógica.

Tras realizar el tratamiento digital de una señal esta puede ser convertida de nuevo en una señal analógica mediante un proceso de reconstrucción llevado a cabo por un conversor D/A. Este proceso será explicado desde el punto de vista del tratamiento digital de señales en el capítulo 3.

No es objetivo de este proyecto profundizar en los conversores A/D D/A desde el punto de vista electrónico ni tampoco en otros conceptos relacionados con la ingeniería de sonido en general. Una buena referencia para estos temas es [3]. Una introducción en castellano a los conversores A/D puede encontrarse en [4, pp. 207-242].

1.1.3 Efectos de audio

La expresión “efecto de audio” se usa para referirse a un determinado procesamiento aplicado a una señal de audio. En el curso de las últimas décadas han surgido gran cantidad de estos efectos, primero en su versión analógica y posteriormente en su versión digital. El objeto de este procesamiento es mejorar o modificar de algún modo la señal con la finalidad de crear un sonido diferente que sea más agradable u original.

El objetivo de este proyecto es crear un modelo original basado en técnicas conocidas de dos efectos de audio: retardo y saturación. Estos efectos se introducen brevemente en las siguientes secciones y serán estudiados en profundidad en capítulos posteriores.

1.1.4 Retardo

Una onda de sonido reflejada en una pared se superpondrá a la onda original. Si la pared está lejos, el oyente percibirá un efecto de eco. Si la pared está cerca, el oyente percibirá el retardo mediante un cambio en el “color” del sonido, es decir, una atenuación de ciertas frecuencias provocada por la superposición de ambas ondas [2, pp. 63].

Cuando la pared u obstáculo esté en movimiento, la duración del retardo variará, produciendo una oscilación del tono. Este efecto se conoce como “efecto Doppler” y sera estudiado en el capítulo 4.

1.1.5 Saturación

La mayoría de los efectos de audio son lineales, una propiedad que será estudiada en el capítulo tres. Basta decir aquí que si dos entradas son sumadas y procesadas por un sistema lineal el resultado será el mismo que el obtenido al procesar ambas entradas por separado y después sumarlas.

Los efectos de saturación están basados en una distorsión de amplitud no lineal, por lo que no comparten esta propiedad. Es precisamente esta no linealidad la que les da su sonido característico. Mientras que los efectos lineales no crean nuevas componentes de frecuencia, las funciones no lineales utilizadas en la saturación si lo hacen [5, pp. 167-188].

Zendrive es un efecto de saturación diseñado y vendido por Hermida Audio. A lo largo del tiempo han surgido diferentes clones hechos por aficionados que intentan imitar al original. El presentado en la figura 1.2 ha sido dibujado por mí a partir de los esquemas encontrados en foros y de los comentarios de la gente sobre el modelo de los componentes.

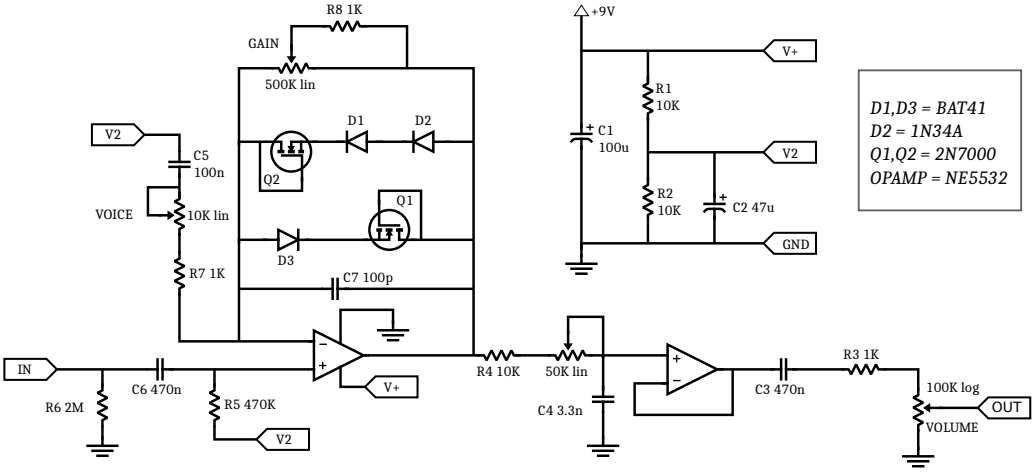


Figura 1.2: Clon del efecto zendrive.

1.1.6 Simulación de circuitos en tiempo real

El circuito planteado en la sección anterior puede describirse en la banda de frecuencias audible mediante ecuaciones diferenciales ordinarias no lineales. Un simulador de circuitos como SPICE puede resolver estos sistemas de ecuaciones diferenciales para predecir su comportamiento de forma muy precisa.

El problema que surge es que la simulación con SPICE es computacionalmente costosa, por lo que una solución apta para tiempo real requiere de alguna aproximación simplificada [6, pp. 3-4]. Diferentes métodos han sido propuestos, por ejemplo en [7], [8]. En este trabajo se utilizan filtros digitales de ondas para simular el comportamiento del circuito en tiempo real.

1.1.7 La interfaz de audio

Una interfaz de audio es un dispositivo ampliamente utilizado en la producción musical que permite realizar los procesos de conversión A/D y D/A en tiempo real con señales de audio de procedencia diversa, ocultando al usuario los detalles de la conversión.

Existen en el mercado una gran cantidad de modelos de interfaz. Algunos de ellos incluyen funcionalidades adicionales, como fuentes de alimentación para micrófonos de condensador o filtros digitales implementados en su software interno.

1.1.8 Los DAW

DAW (Digital Audio Workstation) es un término genérico que se usa para referirse al software usado en la producción musical para llevar a cabo el procesamiento de las señales

digitales obtenidas a través de la interfaz de audio. La gran mayoría de los DAW tienen además la capacidad de servir como *hosts* para plugins de audio, elementos que serán definidos en la siguiente sección.

En el contexto de este proyecto, el DAW elegido se usa para comprobar el funcionamiento de los sistemas diseñados en tiempo real; almacenar audio digital procesado y sin procesar; y controlar la interfaz de audio, sus conversores A/D D/A y las rutas de la señal.

Los DAW en conjunto con los *plugins* de audio ofrecen una gran cantidad de funcionalidades útiles en la ingeniería de sonido y en la mezcla de grabaciones profesionales. Una introducción a este campo puede encontrarse en [9].

En general, un DAW funciona mediante varias pistas donde pueden insertarse archivos de sonido o capturar grabaciones directamente desde la interfaz de audio. En estas pistas es posible insertar plugins de audio que procesarán la señal contenida en la pista cuando ésta se reproduzca.

1.1.9 Los plugins de audio y la librería JUCE

Los plugins de audio son programas diseñados para contener efectos de procesamiento digital que puedan ser usados de forma modular en diferentes DAW. Para asegurar la compatibilidad se han desarrollado en la industria diferentes estándares, como por ejemplo Virtual Studio Technology, de Steinberg; Apple's Audio Unit, de Apple; y Avid Audio Extension, de Avid.

JUCE es una librería de código abierto utilizada para el desarrollo de aplicaciones móviles y de escritorio. La librería JUCE permite construir proyectos en lenguaje C++ y utilizar el mismo código para producir plugins de audio con cualquiera de los formatos estándar mencionados anteriormente [5, pp. 307-308].

1.1.10 La placa de desarrollo

Una placa de desarrollo es una herramienta que las empresas ponen a disposición de los ingenieros para que estos puedan familiarizarse con el funcionamiento y la programación de un determinado producto, como por ejemplo, un microprocesador.

Estas placas suelen disponer del hardware mínimo necesario para hacer posible el funcionamiento del dispositivo, en el caso de un procesador memoria RAM y espacio de almacenamiento no volátil.

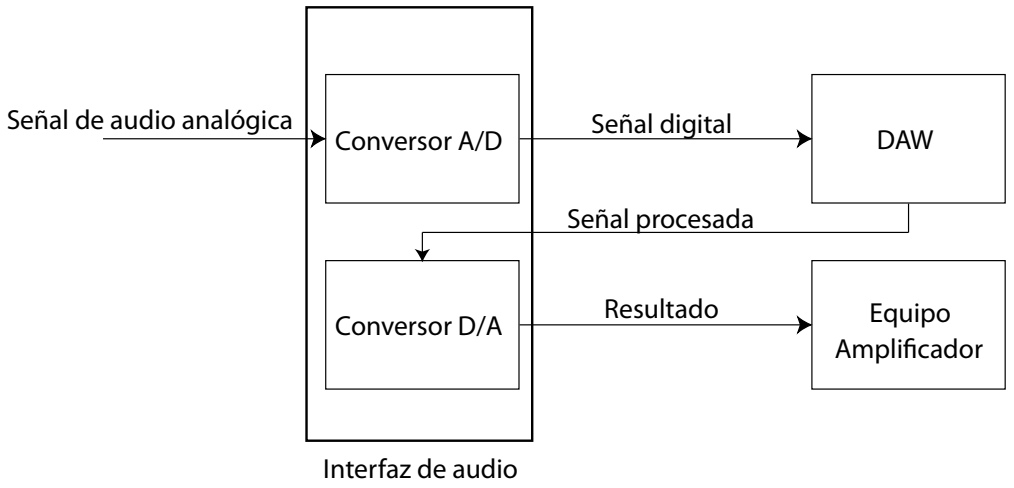


Figura 1.3: Esquema del camino de la señal usando el PC.

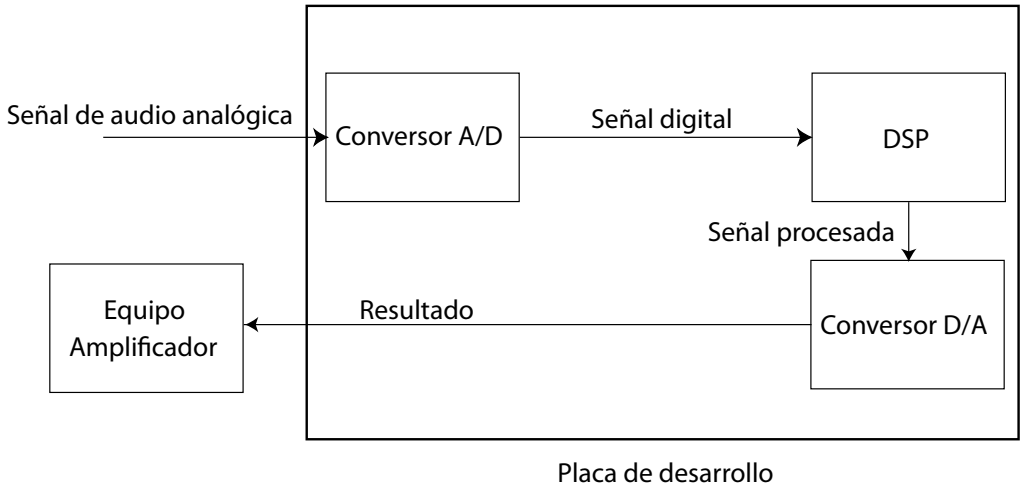


Figura 1.4: Esquema del camino de la señal usando la placa de desarrollo.

1.1.11 El camino de la señal

En la figura 1.3 se muestra un posible camino de una señal de audio a través de los elementos definidos en las secciones anteriores. En la primera fase del diseño de cada efecto, este será el esquema utilizando para comprobar los resultados que produce de forma práctica.

En la figura 1.4 se muestra otro camino de la señal. En este caso, el procesador integrado en la placa de desarrollo cumple la función de realizar el tratamiento digital de la señal. Este esquema será el utilizado cuando se implemente el efecto sobre la arquitectura objetivo.

Nótese que en este último caso, los conversores A/D y D/A están integrados en la placa de desarrollo.

1.2 Objetivos

En el apartado 1.1.11 se plantearon dos posibles caminos de la señal. Ambos serán usados durante el desarrollo del proyecto para aplicar dos tipos de efecto a la señal de entrada: retardo y saturación.

El objetivo de este proyecto es crear un prototipo de multiefectos digital autónomo. El producto del trabajo puede ser la base para mejorar los efectos obtenidos o añadir otros nuevos. Aunque la calidad de los efectos de audio es, por su naturaleza, algo subjetivo, se pretende que los sistemas basados en circuitos físicos simulen lo más fielmente posible los sistemas originales.

Además, el equipo debe tener la capacidad de funcionar en tiempo real, es decir, debe poder procesar las señales de entrada con suficiente rapidez, de tal modo que la latencia sea prácticamente imperceptible para el oído humano.

Dado que los requisitos específicos dependen del sistema a modelar, estos se describirán con detalle en el capítulo de análisis de cada efecto. Este método de trabajo será descrito en el siguiente capítulo.

A grandes rasgos, se plantean los siguientes objetivos:

- Desarrollar un efecto de retardo desde la perspectiva del tratamiento digital de señales.
- Desarrollar un efecto de saturación basándose en el circuito electrónico planteado en la sección 1.1.5.
- Implementar los anteriores sistemas en forma de plugin de audio.
- Implementar los anteriores sistemas sobre una placa de desarrollo de algún DSP activo en el mercado.
- Evaluar la calidad de los resultados obtenidos.
- Evaluar el coste computacional de los sistemas obtenidos y su viabilidad en tiempo real.
- Plantear posibles mejoras y añadidos para un prototipo posterior.

1.3 Justificación de la elección del trabajo

Mi interés en este campo empezó hace ya tres años y desde entonces quise aplicar los conocimientos obtenidos en el Grado al diseño de efectos de audio digitales. Al margen del interés personal, este proyecto ofrece la oportunidad de profundizar de forma práctica en muchos conceptos estudiados durante el grado:

- Se aplican contenidos estudiados en la asignatura “Tratamiento digital de señales” y se añaden otros nuevos relacionados con la materia.
- Se hace uso de conocimientos adquiridos en las asignaturas de ingeniería de computadores y en la asignatura “Electrónica Digital”.
- Se hace uso de conceptos aprendidos en la asignatura “Diseño de Sistemas Operativos”.
- En algunas partes de este trabajo es importante entender como los compiladores traducen los lenguajes de alto nivel. Los trabajos de implementación de un compilador llevados a cabo en la asignatura “Procesadores de Lenguaje II” fueron de gran ayuda para ello.
- Se ejercitan herramientas matemáticas útiles para cualquier labor de ingeniería.
- Se profundiza en los circuitos electrónicos y su simulación.
- Las labores de programación en C y C++ permiten adquirir una base en estos lenguajes, dado que estos no fueron usados en absoluto durante el resto del programa de estudios.
- El campo de los efectos de audio digitales es muy amplio y hay espacio para la aplicación de muchos conocimientos. Por ejemplo, es posible implementar efectos que simulen sistemas físicos haciendo uso de filtros adaptativos en conjunto con algoritmos de inteligencia artificial y/o aprendizaje automático [10], aunque esto queda fuera de los límites de este trabajo.

1.4 Estructura

Debido a que el proyecto consiste en la implementación de dos sistemas separados es posible dividir esta memoria en dos partes, una por efecto de audio. La primera parte del trabajo se centra en construir un retardo modulado que funcione sobre el procesador embebido. Posteriormente, en la segunda parte, se usa el mismo procesador para simular un circuito analógico en tiempo real.

En algunos capítulos del trabajo, especialmente en la primera parte, se hace una revisión de ideas y conceptos relacionados con el tratamiento digital de señales que serán igualmente útiles en la segunda parte. En la segunda parte, los capítulos de revisión se centran en la simulación de circuitos electrónicos mediante filtros digitales de ondas. Previamente a la implementación del software en la placa de desarrollo se hace también una breve revisión del procesador y la placa de desarrollo utilizados así como del IDE de Texas Instruments, *Code Composer Studio*.

A continuación se da una breve descripción de cada uno de los capítulos de esta memoria.

Se especifican claramente que capítulos y secciones son revisión bibliográfica y cuales son explicación del trabajo realizado o desarrollos propios:

- En el presente capítulo se dan algunas definiciones útiles, se explica lo que se pretende alcanzar y se expone la estructura del proyecto y la memoria.
- En el segundo capítulo, llamado "Metodología y herramientas", se define la metodología a seguir y se detallan las herramientas utilizadas para desarrollar el trabajo.
- En el tercer capítulo, llamado "Introducción al tratamiento digital de señales", se presentan herramientas matemáticas y conceptos usados en capítulos posteriores. Este es un capítulo de revisión bibliográfica.
- El cuarto capítulo, llamado "Retardo fraccionario en los efectos de audio", pertenece al proceso de análisis del efecto de retardo. En primer lugar, se hace una aproximación a la simulación de los retardos de sonido mediante tratamiento digital. Posteriormente, se hace una descripción de los retardos fraccionarios, así como una revisión del método usado en este trabajo para su diseño. Aunque este es un capítulo de revisión bibliográfica incluye código propio en Scilab. Además, el desarrollo hecho en la sección 4.6.3 usando cálculo diferencial es propio, ya que el documento original en el que se presenta el método no incluye ningún desarrollo ni se menciona el método de minimización utilizado.
- El quinto capítulo, llamado "Diseño de un retardo modulado", pertenece al proceso de diseño del efecto de retardo. En él se diseña un retardo modulado mediante las técnicas expuestas en el capítulo anterior y se usan ideas originales para implementar las estructuras del sistema. Este capítulo es trabajo propio.
- En el sexto capítulo, llamado "Introducción a la librería JUCE", se explican algunos conceptos necesarios para trabajar con la librería JUCE. Este capítulo es revisión bibliográfica.
- El séptimo capítulo, llamado "Implementación de un plugin de audio para el retardo modulado", pertenece al proceso de codificación del efecto de retardo. En él se explica la implementación del plugin haciendo uso de la librería JUCE y de las ideas planteadas y analizadas en los capítulos previos. Este capítulo es trabajo propio.
- El octavo capítulo, llamado "Pruebas del plugin de audio para el retardo modulado", pertenece al proceso de prueba del efecto de retardo. En él se llevan a cabo pruebas de caja negra con plugin implementado en el capítulo anterior. Este capítulo es trabajo propio.
- En el noveno capítulo, llamado "Introducción al OMAP-L138", se hace una introducción al procesador embebido usado en este trabajo y al codec de audio de la placa de desarrollo. Este capítulo es revisión bibliográfica.
- El décimo capítulo, llamado "Controladores para el OMAP-L138", es un capítulo

independiente de trabajo propio. En él se implementan controladores para configurar los módulos del procesador y crear un sistema de procesamiento en tiempo real que no necesite sistema operativo. A pesar de que el código se ha escrito desde cero está basado en gran medida en el código que T.B. Welch y otros distribuyen con la última edición de su libro en el momento de realizar este trabajo [11].

- El décimo primer capítulo, llamado "Implementación del retardo modulado en el C674x", pertenece al proceso de codificación del efecto de retardo. En él se desarrolla una implementación del algoritmo de retardo modulado, esta vez usando C en el IDE CCS. El resultado de este proceso será software ejecutable en el núcleo C674x. Este capítulo es trabajo propio.
- El décimo segundo capítulo, llamado "Pruebas del retardo modulado en el C674x", pertenece al proceso de pruebas del efecto de retardo. En él se hacen pruebas de caja negra usando la placa de desarrollo para comprobar la funcionalidad del software creado en el capítulo anterior. Este capítulo es trabajo propio.
- El décimo tercer capítulo, llamado "Introducción a los filtros WDF" es un capítulo independiente de revisión bibliográfica. En él se hace una introducción a los filtros WDF y al método de interpolación de Fritsch-Carlson.
- El décimo cuarto capítulo, llamado "Análisis del efecto zendrive" pertenece al proceso de análisis del efecto zendrive. En él se hace un análisis teórico del funcionamiento del circuito que otorga una idea general de su funcionalidad. Este capítulo es trabajo propio.
- El décimo quinto capítulo, llamado "Diseño del plugin oakdrive" pertenece al proceso de diseño del efecto zendrive, cuya versión virtual he decidido llamar "Oakdrive". En él se diseñan mediante filtros WDF estructuras que modelen el circuito analizado en el capítulo anterior. Este capítulo es trabajo propio.
- El décimo sexto capítulo, llamado "Implementación del plugin oakdrive" pertenece al proceso de codificación del efecto zendrive. En él se implementan haciendo uso de la librería JUCE las estructuras WDF diseñadas en el capítulo anterior con el objetivo de crear un plugin de audio que simule el circuito original. Este capítulo es trabajo propio.
- El decimo séptimo capítulo, llamado "Pruebas del plugin oakdrive", pertenece al proceso de pruebas del efecto zendrive. En él se llevan a cabo pruebas de caja negra con el plugin implementado en el capítulo anterior. Este capítulo es trabajo propio.
- El décimo octavo capítulo, llamado "Implementación del efecto oakdrive en el C674x", pertenece al proceso de codificación del efecto zendrive. En él se desarrolla una implementación del efecto zendrive, esta vez usando C en el IDE CCS. El resultado de este proceso será software ejecutable en el núcleo C674x. Este capítulo

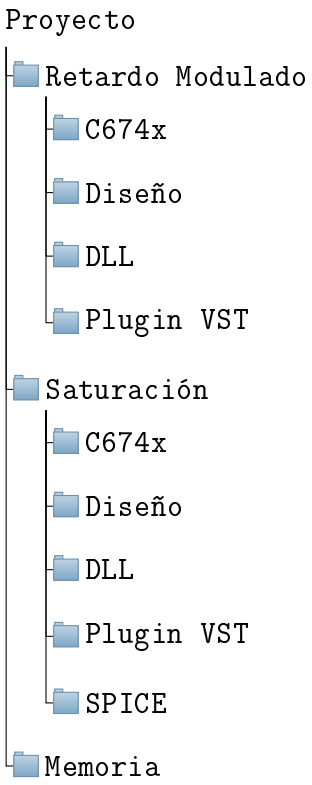


Figura 1.5: Estructura de directorios.

es trabajo propio.

- El décimo noveno capítulo, llamado "Pruebas del efecto oakdrive en el C674x" pertenece al proceso de pruebas del efecto zendrive. En él se hacen pruebas de caja negra usando la placa de desarrollo para comprobar la funcionalidad del software creado en el capítulo anterior. Este capítulo es trabajo propio.
- El décimo noveno capítulo, llamado "Pruebas del efecto oakdrive en el C674x" pertenece al proceso de pruebas del efecto zendrive. En él se hacen pruebas de caja negra usando la placa de desarrollo para comprobar la funcionalidad del software implementado en el capítulo anterior. Este capítulo es trabajo propio.
- En el vigésimo capítulo, llamado "Planificación y costes del proyecto" se exponen las tareas realizadas y el tiempo empleado y se estiman los costes totales.
- En el vigésimo primer capítulo, llamado "Conclusiones y futuros trabajos" se extraen conclusiones del trabajo realizado y se plantean posibles trabajos futuros.

1.5 Estructura de directorios

En la figura 1.5 se muestra la estructura de directorios de este trabajo. El proyecto se divide en dos carpetas principales, una para cada efecto. Además, una tercera carpeta contiene

esta memoria. A continuación se describe el contenido de cada subcarpeta:

C674x. Estas carpetas contienen el código del software para el procesador OMAP-L138 y una lista de instrucciones para su importación a Code Composer Studio.

Diseño. Estas carpetas contienen el código Scilab del diseño de cada efecto separado en archivos con nombres descriptivos.

DLL. Estas carpetas contienen el plugin DLL resultante de mi compilación, así como instrucciones para probarlo.

Plugin VST. Estas carpetas contienen el código del plugin VST de cada efecto y una lista de instrucciones para su compilación.

SPICE Esta carpeta contiene los archivos LTSpice usados durante el desarrollo.

En el momento de entregar este trabajo las versiones del software utilizado han cambiado y las licencias impiden redistribuir versiones anteriores. No obstante, las instrucciones aportadas permiten compilar el código con las últimas versiones disponibles el 27 de Mayo de 2018.

1.6 Conclusiones

En este capítulo se han introducido varios conceptos que serán usados a lo largo del trabajo. Además, se han descrito los objetivos del trabajo y se ha dado una definición general del contenido de cada uno de los capítulos. Finalmente, se ha descrito la estructura de directorios adjunta al trabajo y el contenido de cada una de sus carpetas.

CAPÍTULO 2

Metodología y Herramientas

En este capítulo se introduce el método de desarrollo y las herramientas elegidas para realizar el proyecto, así como la justificación de su elección y un breve listado de sus características, cuando proceda. Además, al final del apartado de metodología se exponen las limitaciones y ventajas del método elegido.

2.1 Metodología

El primer factor a tener en cuenta para entender la metodología utilizada es que las actividades a realizar en este trabajo pueden dividirse en dos tipos. Uno de ellos abarca los procesos relacionados con la electrónica y el tratamiento digital de señales; el otro cae dentro del ámbito de la ingeniería de software. Teniendo en cuenta este hecho, he elaborado una metodología en dos niveles que permita realizar el trabajo siguiendo un esquema secuencial.

En el primer nivel, la metodología es similar al modelo en cascada para desarrollo de software [12, pp. 35-36], [13, pp. 33-35]. En la figura 2.1 puede verse el esquema utilizado. Esta secuencia de procesos se repetirá para cada uno de los efectos a diseñar e implementar.

En el contexto de este trabajo, los procesos de análisis y diseño no consisten en encontrar las necesidades del usuario, sino en describir de forma teórica el procesamiento que el software debe aplicar sobre las señales de entrada. El objetivo es modelar el problema y obtener productos del trabajo que puedan ser usados en los procesos de codificación.

Al comienzo de este proyecto no es posible plantear los requisitos del software de manera formal, al carecer de modelos de los sistemas planteados. Los procesos de análisis y diseño producirán estos modelos, que servirán en si mismos como requisitos y guía para la elaboración del software.

En el segundo nivel, que se encuentra contenido dentro de los procesos de codificación, se plantean dos subprocesos:

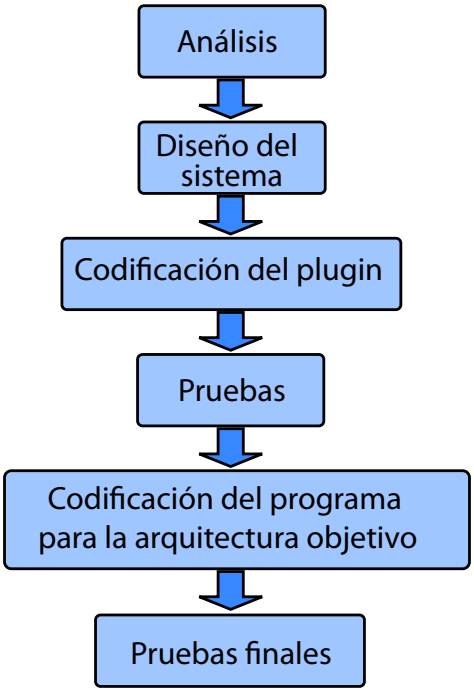


Figura 2.1: Metodología de desarrollo del proyecto.

- Codificación: Se escribirá código que realice el procesamiento de la señal e implemente la interfaz de usuario, si procede, basándose en los modelos obtenidos en los procesos anteriores.
- Pruebas: Utilizando el entorno de Visual Studio se escribirán pruebas unitarias que comprueben las funciones principales implementadas dentro de cada clase. Aunque estas pruebas no serán exhaustivas tienen como propósito ayudar en la escritura de código y reducir los fallos en cada incremento.

Esta metodología presenta diferencias con los procesos de desarrollo de software tradicionales:

- Los requisitos no están claramente definidos al empezar el proyecto.
- El mayor esfuerzo se hace en los procesos de análisis y diseño de los sistemas digitales. La codificación consiste, sobre todo, en implementar los modelos obtenidos mediante operaciones matemáticas y estructuras de datos adecuadas.
- Por esta razón, se pone énfasis en revisar el modelo teórico en el campo del tratamiento digital de señales antes de pasar a su implementación.

A lo largo de todo el trabajo mantengo una mentalidad flexible y busco los mejores recursos disponibles para expresar con claridad los procesos y los resultados obtenidos.

En las siguientes secciones se describen de forma general los procesos para cada uno de los efectos de audio. Cada uno de los procesos abarcará un capítulo de este trabajo.

2.1.1 Análisis

En el caso del retardo, el proceso de análisis consiste en plantear y entender el problema desde el punto de vista del tratamiento digital de señales.

En el caso de la saturación es necesario analizar y entender el circuito planteado.

2.1.2 Diseño del sistema

Una vez analizado el problema el siguiente paso consiste en modelar el sistema.

En el caso del retardo, se trata de construir un modelo de un sistema digital que resuelva de forma satisfactoria el problema planteado previamente, aplicando algún método conocido.

En el caso de la saturación se trata de diseñar un modelo que aproxime el comportamiento real del circuito mediante técnicas conocidas, o dicho de otra forma, de trasladar el tratamiento de la señal analógica al dominio digital. Como se mencionó anteriormente, la simulación mediante un software SPICE no es una solución apta para tiempo real.

2.1.3 Codificación del plugin

El objetivo de este proceso es codificar un plugin de audio ejecutable sobre un DAW que permita aplicar el sistema diseñado a una entrada de audio utilizando el camino de la señal expuesto en la sección 1.1.11. Aunque no es objetivo del proyecto el diseño de una GUI se implementará una interfaz gráfica simple para poder interactuar con el plugin.

2.1.4 Pruebas

En el caso del retardo las pruebas prácticas consisten en la comparación de varios plugins similares en el dominio de la frecuencia y en una evaluación subjetiva del sonido resultante.

En el caso de la saturación es posible evaluar los resultados comparando la respuesta teórica del circuito mediante simulación SPICE con la respuesta del sistema implementado. También es posible hacer pruebas subjetivas pidiendo a varios usuarios que distingan entre una señal procesada con el circuito original y otra procesada con el sistema diseñado.

Nótese que los métodos de prueba explicados son igualmente aplicables en las pruebas finales, con la única diferencia de que todo el procesamiento se produce en la placa de

desarrollo. Al tratarse de una señal digital el resultado será idéntico si se desprecia el efecto de los conversores A/D D/A y del ruido en sus entradas.

Estas pruebas son pruebas de caja negra, orientadas sobre todo a comprobar la funcionalidad del software. Como ya se ha dicho, el grado en que la funcionalidad es satisfactoria dependerá sobre todo de la calidad del modelo realizado durante los procesos de análisis y diseño. No obstante, un requisito fundamental es que la latencia se encuentre dentro de un limite aceptable apenas perceptible para el oído humano.

Aunque la calidad de los efectos de audio es un factor subjetivo, las pruebas pretenden garantizar que el software creado es funcional y tiene un mínimo de calidad de acuerdo al criterio de varias personas acostumbradas a usar este tipo de efectos.

2.1.5 Codificación del programa sobre la arquitectura objetivo

En este punto es posible tener la certeza de que el sistema diseñado es aceptable y pasar a su implementación sobre la placa de desarrollo.

2.1.6 Ventajas y limitaciones de la metodología elegida

Utilizando el método de desarrollo planteado se producen dos programas diferentes por cada efecto. Esto permite verificar que el sistema diseñado es satisfactorio antes de pasar a su implementación sobre la arquitectura objetivo. Además, utilizando este método aumenta la portabilidad. Cualquier usuario puede probar el sistema diseñado utilizando el plugin implementado sin más que disponer de un computador personal y un software DAW.

2.2 Herramientas

En esta sección se introducen las herramientas concretas que van a utilizarse y se da una breve justificación del uso de las mismas.

2.2.1 DAW

El DAW elegido es Reaper 5.35. Podría haberse usado cualquier otro DAW pero elegí Reaper debido a mi familiaridad con él. Además, puede evaluarse durante 60 días y el precio de una licencia para uso no comercial es de 60 USD, siendo el más barato de entre todos los DAW de pago [14].

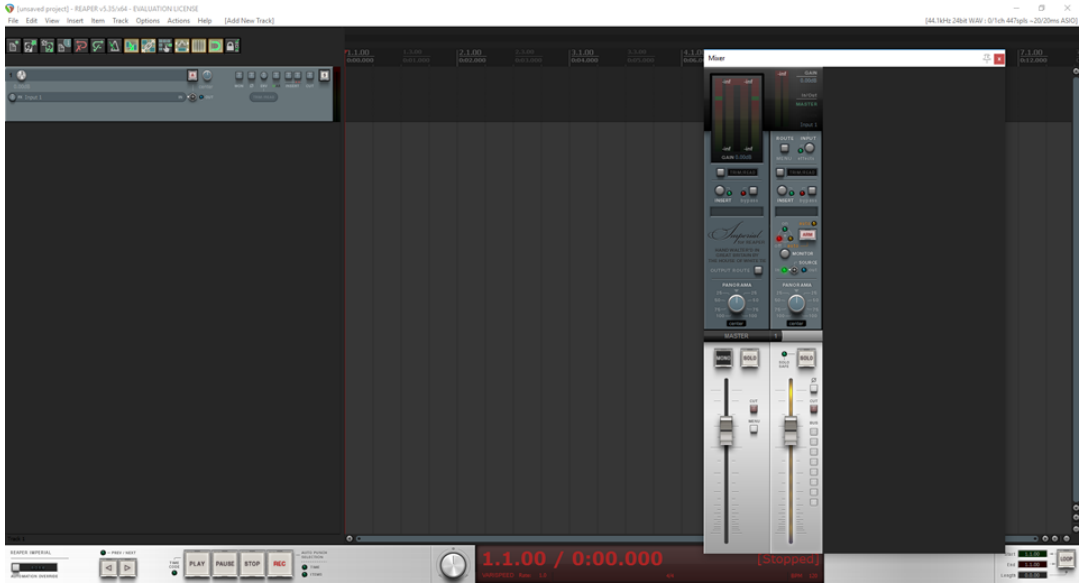


Figura 2.2: Pantalla principal de Reaper.

En la figura 2.2 puede verse una captura de la pantalla principal de Reaper. La interfaz del programa será introducida brevemente en capítulos posteriores.

2.2.2 JUCE

Los plugins de audio son programas diseñados para contener efectos de procesamiento digital que puedan ser usados de forma modular en diferentes DAW. Para asegurar la compatibilidad, se han desarrollado en la industria diferentes estándares. Estos incluyen Virtual Studio Technology, de Steinberg; Apple's Audio Unit, de Apple; y Avid Audio Extension, de Avid.

JUCE es una librería de código abierto utilizada para el desarrollo de aplicaciones móviles y de escritorio. La librería JUCE permite construir proyectos en lenguaje C++ y utilizar el mismo código para producir cualquiera de los formatos estándar de plugin mencionados anteriormente [5, pp. 307-308]. Esta librería facilita las cuestiones de comunicación con el DAW, permitiendo al programador centrarse en el tratamiento de la señal. Se está convirtiendo rápidamente en la librería más utilizada para este tipo de aplicaciones.

2.2.3 IDE para C++

El IDE utilizado en este proyecto es Visual Studio 2017. En el capítulo correspondiente se indicarán los pasos a seguir para usarlo en conjunto con las herramientas proporcionadas por la librería JUCE.

2.2.4 Herramientas de soporte para análisis y diseño

LTspice es un simulador SPICE con capturador de esquemas y visualizador de formas de onda desarrollado por Linear Technology. Es el software SPICE elegido para este proyecto.

Scilab es un software de análisis numérico con un lenguaje de programación de alto nivel para cálculo científico. Será utilizado en este proyecto para el análisis y diseño de los sistemas digitales y la generación de gráficos que aporten información sobre su respuesta en el dominio del tiempo o de la frecuencia.

2.2.5 Herramientas para la realización de la memoria

Texmaker es el editor de LaTeX elegido para escribir esta memoria. Además, se usará Inkscape como editor de gráficos vectoriales e imágenes. Como editor de textos complementario se usará Sublime Text 3. Todos estos programas pueden obtenerse de forma gratuita.

2.2.6 Osciloscopio Rigol DS1052E

En el contexto de este trabajo, este sencillo osciloscopio digital será usado para comprobar las forma de onda de los relojes en la placa de desarrollo y medir el tiempo de procesamiento de los efectos implementados.

2.2.7 Interfaz de audio

La interfaz de audio elegida es la UltraLite-mk3 Hybrid. Cualquier otra interfaz podría servir, pero ya disponía previamente de este modelo. En el contexto del proyecto esta interfaz tiene las siguientes características interesantes:

- Conversores A/D y D/A configurables que permiten hasta 192 Khz de frecuencia de muestreo.
- Interfaz USB que permite controlar el dispositivo haciendo uso de un software y transmitir las señales de audio en tiempo real a un computador personal.
- Varias salidas y entradas de audio analógicas.

En la figura 2.3 se muestran los paneles frontal y trasero de la interfaz elegida [15].

2.2.8 Placa de desarrollo

La placa de desarrollo elegida es la TMDSLCDK138 de Texas Instruments, que sirve como placa de evaluación para el procesador orientado a tratamiento digital de señales

Figura 2.3: Paneles frontal y trasero del modelo de interfaz elegido. (Imagen de www.motu.com)

OMAP-L138.

El OMAP-L138 es un SoC de bajo consumo con una arquitectura doble, que integra un procesador ARM926EJ-S y un nucleo C674x y puede funcionar a una velocidad de hasta 456 Mhz [16].

ARM926EJ-S es un procesador RISC con arquitectura ARM. Un procesador RISC (Reduced Instruction Set Computer) funciona con un conjunto de instrucciones simplificado y debido a ello no es necesario disponer de un hardware de control extremadamente complejo [17, pp. 3-6].

ARM es una arquitectura RISC que además ofrece lo siguiente [18, pp. 1-2]:

- Control sobre la ALU y el desplazador en cada instrucción de procesamiento de datos para maximizar su uso.
- Modos de direccionamiento auto-incrementales y auto-decrementales para optimizar los bucles del programa.
- Carga y almacenamiento de varias instrucciones para maximizar el rendimiento de datos.
- Versión condicional de todas las instrucciones para maximizar el rendimiento de ejecución.

El DSP C674x consiste en 64 registros de propósito general de 32 bits y ocho unidades funcionales, con una capacidad de procesamiento de hasta 2746 MFLOPS.

La generación C6000 tiene un set completo de herramientas optimizadas, incluyendo un compilador de C eficiente. Estos procesadores tienen una arquitectura de tipo VLIW [19, pp. 20].

En un procesador VLIW, el hardware no tiene que intervenir para descubrir el paralelismo entre instrucciones tal y como ocurre en los procesadores superescalares, sino que es responsabilidad del compilador [17, pp. 214-221].

La placa incluye, además del procesador, los siguientes elementos interesantes para el proyecto [20]:

- 128 MBytes de memoria SDRAM DDR2
- Puertos USB
- Entrada y salida de audio analógico
- Conversores A/D y D/A

El OMAP-L138 y su placa de evaluación tienen características ideales para este trabajo.

2.2.9 Code Composer Studio

Code Composer Studio (CCS) es el entorno integrado de desarrollo de Texas Instruments compuesto por un conjunto de herramientas que permiten el desarrollo y depuración de aplicaciones embebidas, incluyendo un compilador y un depurador.

2.2.10 Sonda de depuración XDS100V2

La sonda XDS100V2 permite depurar código en el procesador OMAP-L138 mediante el puerto JTAG de la placa de desarrollo. Esta sonda se conecta mediante USB al PC donde se esté ejecutando CCS.

Cuando se crea un nuevo proyecto en CCS automáticamente el programa solicitará que se seleccione el procesador para el que se pretende compilar el código y una sonda de depuración. Si se selecciona el modelo de placa de desarrollo CCS ejecutará automáticamente un script cada vez que comience una sesión de depuración. Este script realiza configuraciones básicas en los módulos del sistema y carga el programa en memoria

Como se mencionará posteriormente, en una versión definitiva del software esto debería ser tomado en consideración para habilitar una ejecución autónoma. Existen formas automatizadas de pasar de un entorno de desarrollo a uno de producción, si bien estas quedan fuera del marco de este trabajo.

2.3 Conclusiones

En este capítulo se ha definido y justificado la metodología a seguir durante el desarrollo del trabajo, dando una corta explicación del contenido de cada subproceso. Además, se han presentado las herramientas elegidas para el desarrollo del trabajo y la escritura de la memoria.

CAPÍTULO 3

Introducción al Tratamiento Digital de Señales

En este capítulo se presenta una introducción al tratamiento digital de señales breve e intuitiva, aunque poco formal matemáticamente. A lo largo de este capítulo se cita [1] como referencia, pero los conceptos mencionados pueden encontrarse en cualquier libro de texto sobre la materia explicados en profundidad. Recomiendo al lector poco familiarizado que consulte referencias adicionales. Lo aquí expuesto es solo una escueta explicación de algunas de las herramientas importantes en el tratamiento digital de señales que serán usadas en posteriores capítulos.

3.1 Señales sinusoidales continuas y discretas

Una señal sinusoidal continua puede representarse matemáticamente mediante la siguiente función:

$$x_a(t) = A\cos(\Omega t + \theta) \tag{3.1}$$

Donde A es la amplitud, Ω es la frecuencia en radianes/segundo y θ es la fase en radianes. La siguiente función representa una señal exponencial compleja:

$$x_a(t) = Ae^{j(\Omega t+\theta)} \tag{3.2}$$

Aplicando la identidad de Euler se obtiene otra forma de representar las señales sinusoidales [1, pp. 10-17]:

$$x_a(t) = A\cos(\Omega t + \theta) = \frac{A}{2}e^{j(\Omega t+\theta)} + \frac{A}{2}e^{-j(\Omega t+\theta)} \tag{3.3}$$

Si siguiendo el proceso descrito en el apartado 1.1.1 muestreamos la señal sinusoidal continua con un periodo T, es decir, tomamos valores discretos en instantes de tiempo equidistantes, resulta:

$$x(nT) = Acos(\Omega nT + \theta) \tag{3.4}$$

Donde n es una variable que toma cualquier valor entero, llamada número de muestra. Si definimos $\omega = \Omega T$ resulta:

$$x(n) = Acos(\omega n + \theta) \tag{3.5}$$

Siendo esta la representación de la señal sinusoidal en el dominio discreto. La variable ω es la frecuencia en radianes/muestra.

La frecuencia Ω puede representarse en ciclos/segundo o Hercios(Hz):

$$F = \frac{\Omega}{2\pi} \tag{3.6}$$

Por tanto, podemos calcular una variable f en ciclos/muestra para tiempo discreto:

$$f = \frac{\Omega}{2\pi}T \tag{3.7}$$

A esta frecuencia se le llama frecuencia normalizada. Es fácil darse cuenta de que:

$$f = \frac{\Omega}{2\pi}T = FT = \frac{F}{F_s} \tag{3.8}$$

Donde $F_s = 1/T$ es la frecuencia de muestreo, o dicho de otro modo, la cantidad de muestras que se toman en un segundo de la señal continua.

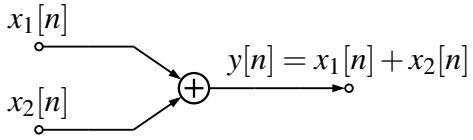


Figura 3.1: Sumador.

Consideremos ahora una señal sinusoidal discreta $Acos(\omega n + \theta)$, es evidente que:

$$x(n) = Acos(\omega n + \theta) = Acos((\omega + 2\pi)n + \theta) = Acos((\omega n + 2\pi n) + \theta) \qquad (3.9)$$

De donde se deduce que las señales sinusoidales discretas en el tiempo cuyas frecuencias están separadas un múltiplo entero de 2π son idénticas. A las señales discretas con frecuencias fuera del rango $[-\pi, \pi]$ se las denomina *alias* de la señal equivalente dentro de ese intervalo.

Utilizando esta conclusión y la expresión (3.8) se deduce que la frecuencia de la sinusoide continua en el tiempo cuando se muestrea a una frecuencia F_s debe encontrarse dentro del rango:

$$[\frac{-F_s}{2}, \frac{F_s}{2}]$$

De lo contrario se introduce una ambigüedad, ya que existen frecuencias en la señal original que no hay forma de determinar una vez esta se ha muestreado. Además, estas frecuencias se superponen en la señal muestreada, dando lugar a una distorsión de *aliasing*.

Un desarrollo completo de estas cuestiones puede encontrarse en [1, pp. 1-33].

3.2 Sistemas discretos en el tiempo

Un sistema discreto en el tiempo es un dispositivo o algoritmo que actúa sobre una señal discreta en el tiempo, que es la entrada o excitación, de acuerdo a una determinada regla bien definida, para producir una señal discreta en el tiempo, que es la salida del sistema.

Hay varias formas de representar los sistemas discretos en el tiempo. Una de ellas es el diagrama de bloques. La figura 3.1 es un ejemplo de diagrama de bloques de un sistema que simplemente suma dos señales de entrada.

3.2.1 Clasificación de los sistemas discretos en el tiempo

Existen varias formas de clasificar los sistemas discretos en el tiempo:

- Sistemas estáticos y dinámicos: Se dice que un sistema es estático si su salida en un instante n depende solo de la muestra de entrada en dicho instante. En cualquier otro caso se dice que el sistema es dinámico o con memoria.
- Sistemas invariantes y variantes en el tiempo: Se dice que un sistema es invariante en el tiempo si el conjunto de operaciones que realiza sobre la señal o señales de entrada no varía con el tiempo. En caso contrario, se trata de un sistema variante en el tiempo.
- Sistemas lineales y no lineales: Un sistema es lineal si satisface el principio de superposición. Este principio exige que la respuesta de un sistema a la suma ponderada de varias señales sea igual a la suma ponderada de las respuestas del sistema a cada una de las entradas individuales.
- Sistemas causales y no casuales: Un sistema es causal si su salida en un instante n depende solo de las entradas actuales y pasadas.

Una clasificación más completa de los sistemas discretos en el tiempo puede encontrarse en [1, pp. 53-60].

3.2.2 Respuesta de los sistemas lineales invariantes en el tiempo (LTI)

Cualquier señal discreta puede descomponerse en una suma de impulsos mediante la siguiente expresión:

$$\sum_{k=-\infty}^{\infty} x(k)\delta(n-k) \tag{3.10}$$

Donde δ representa la función impulso unitario, que toma valor 1 en el punto 0 y valor 0 en todos los demás.

Si un sistema es lineal e invariante en el tiempo y denotamos la transformación que hace sobre una señal de entrada como $\mathscr{T}$, entonces:

$$y(n) = \mathscr{T}[x(n)] = \mathscr{T}[\sum_{k=-\infty}^{\infty} x(k)\delta(n-k)] \tag{3.11}$$

Como el sistema es lineal, aplicando el principio de superposición resulta que:

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)\mathscr{T}[\delta(n-k)] \tag{3.12}$$

Si se denota la respuesta del sistema al impulso unitario como $h(n)$, teniendo en cuenta que el sistema es invariante en el tiempo:

$$h(n-k) = \mathscr{T}[\delta(n-k)] \tag{3.13}$$

Y por lo tanto:

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) \tag{3.14}$$

Esta expresión se denomina suma de convolución. Se dice que la entrada $x(n)$ se convoluciona con la respuesta al impulso $h(n)$ para proporcionar la salida $y(n)$.

Queda por tanto demostrado que un sistema LTI está totalmente caracterizado por su respuesta al impulso unitario h(n). Dicha respuesta puede ser finita (Sistemas FIR, *finite impulse response*) o infinita (Sistemas IIR, *infinite impulse response*).

3.2.3 Sistemas recursivos y no recursivos

Existen sistemas en los que es necesario o deseable expresar la salida no solo mediante los valores actual y pasados de la entrada, sino también en función de los valores ya disponibles de la salida. Esta es una implementación recursiva del sistema.

En muchos libros de texto se identifican los sistemas FIR con las implementaciones no recursivas y los sistemas IIR con las implementaciones recursivas. Sin embargo, esto es un error.

Tomemos por ejemplo el siguiente sistema FIR:

$$y(n) = x(n) + x(n-1) \tag{3.15}$$

Se trata obviamente de una implementación no recursiva. Sin embargo, utilizando la propiedad de invarianza en el tiempo, el anterior sistema también puede escribirse como:

$$y(n-1) = x(n-1) + x(n-2) \tag{3.16}$$

Restando las dos expresiones anteriores se obtiene:

$$y(n) - y(n-1) = x(n) - x(n-2) \tag{3.17}$$

Y por lo tanto:

$$y(n) = y(n-1) + x(n) - x(n-2) \tag{3.18}$$

Que es una implementación recursiva del sistema FIR tomado como ejemplo.

Tanto los sistemas FIR como los sistemas IIR pueden implementarse en su versión recursiva o no recursiva, aunque normalmente es preferible la implementación recursiva en los sistemas IIR [21].

Teniendo en cuenta lo expuesto, una clasificación de los sistemas en recursivos y no recursivos y una explicación de las ecuaciones en diferencias usadas para representarlos, así como de los métodos usados para su resolución, puede encontrarse en [1, pp. 79-94].

3.3 La transformada Z

La transformada Z de una señal discreta en el tiempo se define mediante la siguiente expresión:

$$X(z) = \sum_{n=-\infty}^{\infty} x(n) z^{-n} \tag{3.19}$$

Esta transformada solo existe para los valores de z que hacen converger la serie de potencias. La región de convergencia o ROC, *Region Of Convergence*, delimita el conjunto de valores

de z para los que $X(z)$ converge.

La transformada Z puede verse como una representación alternativa de una señal, muy útil en el análisis de sistemas LTI. En ocasiones es posible escribir la serie de potencias de forma compacta, obteniendo una representación de toda la señal en una sola expresión. Una introducción completa a la transformada Z puede encontrarse en [1, pp. 131-202].

3.3.1 Algunas propiedades de la transformada Z

Tomemos la siguiente suma ponderada de dos señales:

$$y(n) = a_1 x_1(n) + a_2 x_2(n) \tag{3.20}$$

Su transformada z es:

$$Y(z) = \sum_{n=-\infty}^{\infty} [a_1 x_1(n) + a_2 x_2(n)] z^{-n} = \sum_{n=-\infty}^{\infty} [a_1 x_1(n) z^{-n} + a_2 x_2(n) z^{-n}] \tag{3.21}$$

Separando ambos términos resulta:

$$Y(z) = a_1 \sum_{n=-\infty}^{\infty} x_1(n) z^{-n} + a_2 \sum_{n=-\infty}^{\infty} x_2(n) z^{-n} \tag{3.22}$$

Y utilizando la definición de transformada Z:

$$Y(z) = a_1 X_1(z) + a_2 X_2(z) \tag{3.23}$$

Esta propiedad de linealidad puede generalizarse fácilmente para un numero arbitrario de señales ponderadas.

Tomemos ahora la señal:

$$x(n-k) \tag{3.24}$$

Su transformada z es:

$$X(z) = \sum_{n=-\infty}^{\infty} x(n-k)z^{-n} \tag{3.25}$$

Tomando $g = n-k$ y sustituyendo:

$$X(z) = \sum_{g=-\infty}^{\infty} x(g)z^{-(g+k)} = \sum_{g=-\infty}^{\infty} x(g)z^{-g}z^{-k} = z^{-k}X(z) \tag{3.26}$$

Esta propiedad se conoce como propiedad de desplazamiento temporal.

Recordemos ahora que la suma de convolución se define como:

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) \tag{3.27}$$

La transformada z del resultado de la convolución es:

$$Y(z) = \sum_{n=-\infty}^{\infty} [\sum_{k=-\infty}^{\infty} x(k)h(n-k)]z^{-n} \tag{3.28}$$

Intercambiando el orden de los sumatorios y aplicando la propiedad de desplazamiento temporal:

$$Y(z) = \sum_{k=-\infty}^{\infty} x(k)[\sum_{n=-\infty}^{\infty} [h(n-k)z^{-n}] = H(z)\sum_{k=-\infty}^{\infty} x(k)z^{-k} = H(z)X(z) \tag{3.29}$$

De donde se extrae la importante conclusión de que la convolución en el dominio del tiempo equivale a una multiplicación en el dominio z. En un sistema LTI, H(z) es la transformada z de la respuesta al impulso unitario del sistema y se la conoce como función

de transferencia. Como se deduce de la expresión anterior:

$$H(z) = \frac{Y(z)}{X(z)} \tag{3.30}$$

La transformada z puede invertirse mediante diferentes métodos. El más sencillo, aunque no siempre posible, es la descomposición de una transformada z racional en fracciones parciales y la búsqueda en tablas de la transformada inversa de cada fracción.

3.4 Análisis de las señales en el dominio de la frecuencia

3.4.1 La serie de Fourier

La idea básica de la serie de Fourier es que toda señal periódica puede descomponerse en una combinación lineal de exponenciales complejas armónicamente relacionadas:

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{j\omega_0 kt} \tag{3.31}$$

Donde $\omega_0 = 2\pi(1/T)$ y T es el periodo fundamental de la señal. Multiplicando ambos lados de la expresión (3.31) por $e^{-jn\omega_0 t}$, donde n es un entero arbitrario, resulta:

$$x(t)e^{-jn\omega_0 t} = \sum_{k=-\infty}^{\infty} a_k e^{j\omega_0 (k-n)t} \tag{3.32}$$

Integrando ambas partes de la expresión anterior en un intervalo igual al periodo T:

$$\int_0^T x(t)e^{-jn\omega_0 t}dt = \int_0^T \sum_{k=-\infty}^{\infty} a_k e^{j\omega_0 (k-n)t}dt \tag{3.33}$$

Los coeficientes a_k del lado derecho de la expresión anterior no dependen de la variable de

integración, por lo que puede convertirse en una suma de integrales ponderadas:

$$\sum_{k=-\infty}^{\infty} a_k \int_0^T e^{j\omega_0(k-n)t} dt \tag{3.34}$$

Aplicando la fórmula de Euler puede descomponerse la integral de la expresión anterior en dos partes:

$$\int_0^T e^{j\omega_0(k-n)t} dt = \int_0^T cos[(k-n)\omega_0 t]dt + j\int_0^T sen[(k-n)\omega_0 t]dt \tag{3.35}$$

Para resolver la primera de las integrales vamos a considerar dos supuestos. El primero, que $n = k$. En este caso, como $cos(0) = 1$ el valor de la integral es T. Si $n \neq k$, entonces $k-n$ es un entero arbitrario multiplicado por la frecuencia fundamental ω_0. Es fácil ver que en este caso el periodo de oscilación del coseno en el interior la integral es una fracción del periodo T, por lo que la integral toma valor 0.

Para la segunda de las integrales, en el caso en el que $n = k$, $sen(0) = 0$ y por lo tanto la integral toma valor 0. Si $n \neq k$, siguiendo un razonamiento análogo al utilizado en la primera integral, el valor es igualmente 0.

Por lo tanto, la integral en la expresión (3.34) toma valor T cuando $k = n$ y valor 0 en caso contrario:

$$\sum_{k=-\infty}^{\infty} a_k \int_0^T e^{j\omega_0(k-n)t} dt = a_n T \tag{3.36}$$

De los dos párrafos anteriores y las expresiones (3.33), (3.34) y (3.36) se deduce que:

$$a_n = \frac{1}{T}\int_0^T x(t)e^{-jn\omega_0 t} dt \tag{3.37}$$

Esta expresión permite calcular los coeficientes que hacen cierta la igualdad en (3.31).

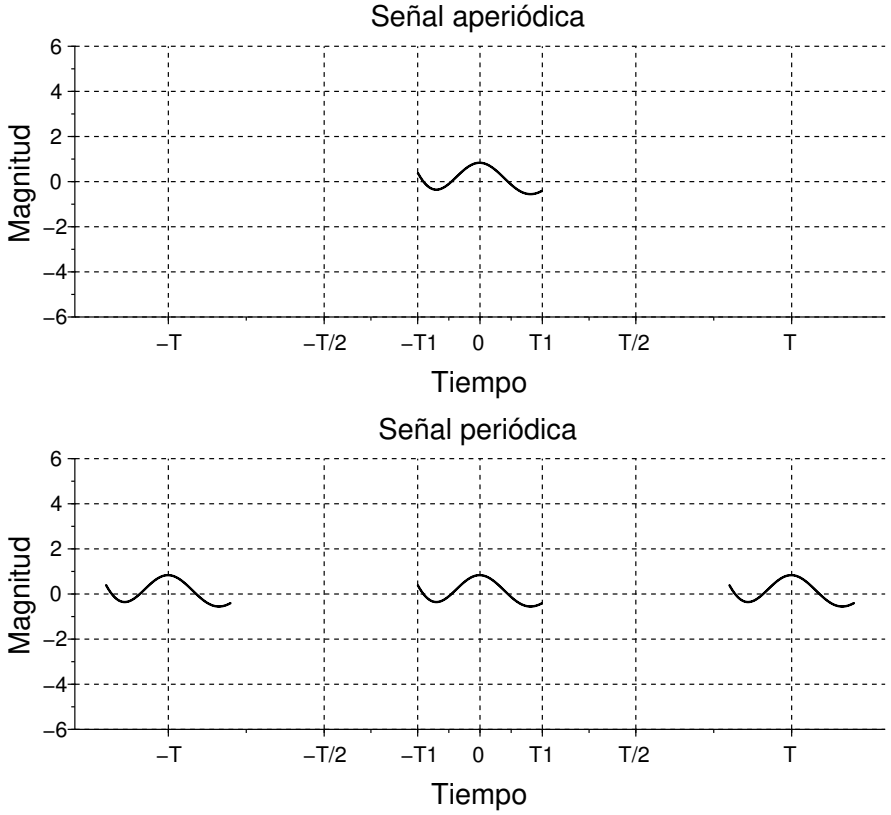


Figura 3.2: Señal aperiódica y señal periódica con periodo T.

3.4.2 La transformada de Fourier

La figura 3.2 muestra dos señales. La primera de ellas es una señal aperiódica. Si esta señal se repite en el tiempo con un periodo T se obtiene una señal periódica, tal y como se muestra en la segunda figura. Denotemos la señal aperiódica como $x(t)$ y la periódica como $\tilde{x}(t)$.

De acuerdo a lo expuesto en la sección anterior, la serie de Fourier de la señal periódica es:

$$\tilde{x}(t) = \sum_{k=-\infty}^{\infty} a_k e^{j\omega_0 kt} \tag{3.38}$$

Y sus coeficientes:

$$a_k = \frac{1}{T}\int_{-T/2}^{T/2} \tilde{x}(t) e^{-jk\omega_0 t} dt \tag{3.39}$$

Es fácil ver que en el intervalo de integración, la señal periódica puede sustituirse por la aperiódica:

$$\frac{1}{T}\int_{-T/2}^{T/2} \tilde{x}(t) e^{-jk\omega_0 t} dt = \frac{1}{T}\int_{-T/2}^{T/2} x(t) e^{-jk\omega_0 t} dt \tag{3.40}$$

Y en este caso, cualquier ampliación del intervalo de integración dará el mismo resultado, siempre que abarque la señal $x(t)$:

$$a_k = \frac{1}{T}\int_{-T/2}^{T/2} x(t) e^{-jk\omega_0 t} dt = \frac{1}{T}\int_{-\infty}^{\infty} x(t) e^{-jk\omega_0 t} dt \tag{3.41}$$

Ahora denotamos $X(\omega)$ como:

$$X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt \tag{3.42}$$

De (3.38), (3.41), (3.42) se tiene que:

$$\tilde{x}(t) = \sum_{k=-\infty}^{\infty} \frac{1}{T} X(k\omega_0) e^{jk\omega_0 t} \tag{3.43}$$

Para obtener la señal aperiódica $x(t)$ de esta expresión, hacemos que el periodo T tienda a infinito, de tal forma que las réplicas de la señal estén infinitamente distantes en el tiempo. Recordemos que:

$$\frac{\omega_0}{2\pi} = \frac{1}{T} \tag{3.44}$$

Sustituyendo:

$$\tilde{x}(t) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} [X(k\omega_0) e^{jk\omega_0 t}] \omega_0 \tag{3.45}$$

Cuando T tiende a infinito, ω_0 tiende a 0. Por lo tanto:

$$x(t) = \lim_{\omega_0 \to 0} (\frac{1}{2\pi} \sum_{k=-\infty}^{\infty} [X(k\omega_0) e^{jk\omega_0 t}] \omega_0) \tag{3.46}$$

Para resolver este límite, es importante recordar la integración de Riemann [22, pp. 191-218]. Será entonces fácil deducir que la suma se convierte en una integral cuando la función que aparece entre corchetes toma valores para el sumatorio en intervalos infinitamente cercanos:

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} [X(\omega) e^{j\omega t}] d\omega \tag{3.47}$$

La expresión (3.42) se denomina transformada de Fourier, y permite obtener el espectro de frecuencias de una señal aperiódica. La expresión (3.47) realiza el proceso inverso, obteniendo una señal en el dominio del tiempo a partir de su espectro de frecuencias.

3.4.3 La transformada discreta de Fourier

La transformada de Fourier para señales discretas o DTFT, *Discrete Time Fourier Transform*, se define como:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n} \tag{3.48}$$

Vemos por tanto que el espectro sigue siendo una función continua de la frecuencia, aunque la señal sea discreta. Sin embargo, es importante resaltar una diferencia con la transformada de Fourier. Utilizando la expresión anterior, se tiene:

$$X(\omega+2\pi) = \sum_{n=-\infty}^{\infty} x(n)e^{-j(\omega+2\pi)n} = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n}e^{-j2\pi n} = X(\omega) \tag{3.49}$$

Es fácil deducir que el espectro obtenido se repite centrado en los múltiplos de 2π. Esto es coherente con lo explicado en la sección 3.1, donde se demostró que las señales sinusoidales discretas cuyas frecuencias angulares está separadas un múltiplo entero de 2π son idénticas.

Debido a la anterior, en la transformada inversa de Fourier para el espectro de frecuencias de una señal discreta solo se integra un intervalo de anchura 2π, es decir:

$$x(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} [X(\omega)e^{j\omega n}]d\omega \tag{3.50}$$

Es importante mencionar que la transformada discreta de Fourier es la transformada z evaluada en $z = e^{j\omega}$, es decir, en la circunferencia de radio 1 con centro en el origen del plano complejo. Cuando se evalúa la función de transferencia de un sistema en $z = e^{j\omega}$ la función resultante se denomina respuesta en frecuencia del sistema. A su módulo y argumento se los denomina módulo y fase de la respuesta en frecuencia, respectivamente.

3.4.4 Serie de Fourier discreta en el tiempo

La serie de Fourier discreta en el tiempo o DTFS(Discrete Time Fourier Series) es el equivalente de la serie de Fourier para señales discretas. Se define como:

$$x(n) = \sum_{n=0}^{N-1} c_k e^{jk\frac{2\pi}{N}n} \tag{3.51}$$

Donde N es el periodo de la señal. Nótese que ahora la señal se representa como una suma de N funciones exponenciales armónicamente relacionadas. Los coeficientes c_k se calculan mediante la siguiente expresión:

$$c_k = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-jk\frac{2\pi}{N}n} \tag{3.52}$$

3.4.5 Algunas transformadas de Fourier

Exponencial compleja

El problema de encontrar la transformada de Fourier de una señal exponencial compleja puede plantearse informalmente como la búsqueda de una función $X(\omega)$ tal que:

$$e^{j\omega_0 t} = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega) e^{j\omega t} d\omega \tag{3.53}$$

Una de las formas de definir la función delta mediante integrales es [23]:

$$\int_{\omega-}^{\omega+} d\omega \delta(\omega - \omega_0) f(\omega) = \begin{cases} f(\omega_0) & \text{si } \omega- < \omega_0 < \omega+ \\ 0 & \text{en otro caso} \end{cases} \tag{3.54}$$

Por tanto es fácil ver que la función $X(\omega)$ buscada es:

$$X(\omega) = 2\pi\delta(\omega - \omega_0) \tag{3.55}$$

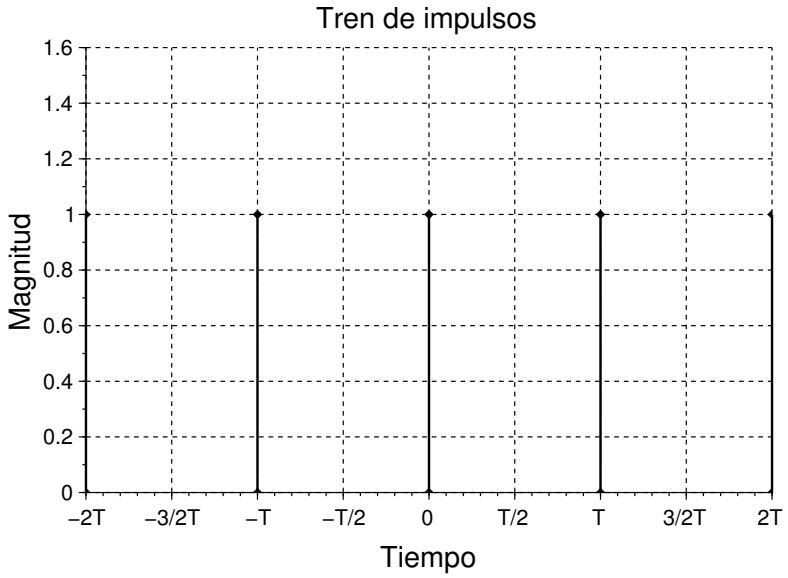


Figura 3.3: Tren de impulsos de periodo T.

Señal periódica

La serie de Fourier de una señal periódica es:

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{j\omega_0 kt} \tag{3.56}$$

Y por lo tanto su transformada de Fourier puede expresarse como:

$$X(\omega) = \int_{-\infty}^{\infty} \sum_{k=-\infty}^{\infty} a_k e^{j\omega_0 kt} e^{-j\omega t} dt = \sum_{k=-\infty}^{\infty} a_k \int_{-\infty}^{\infty} e^{j\omega_0 kt} e^{-j\omega t} dt \tag{3.57}$$

La integral en la última de las expresiones anteriores es la transformada de Fourier de la señal exponencial compleja $e^{j\omega_0 kt}$, y por lo tanto:

$$X(\omega) = 2\pi \sum_{k=-\infty}^{\infty} a_k \delta(\omega - \omega_0) \tag{3.58}$$

Tren de impulsos

En la figura 3.3 se muestra un tren de impulsos de periodo T. Esta señal toma valor 1 en todos los instantes múltiplos de su periodo y valor 0 en el resto de instantes.

Calculando los coeficientes de su serie de Fourier se tiene:

$$a_k = \frac{1}{T}\int_{-T/2}^{T/2} x(t)e^{-jk\frac{2\pi}{T}t}dt = \frac{1}{T}\int_{-\infty}^{\infty} \delta(t)e^{-jk\frac{2\pi}{T}t}dt = \frac{1}{T} \tag{3.59}$$

Por lo tanto, usando la transformada de Fourier de una señal periódica:

$$X(\omega) = 2\pi \sum_{k=-\infty}^{\infty} \frac{1}{T}\delta(\omega - k\omega_0) = \omega_0 \sum_{k=-\infty}^{\infty} \delta(\omega - k\omega_0) \tag{3.60}$$

3.4.6 Convolución continua y teorema de convolución continua

Resulta intuitivo que la suma de convolución de la expresión (3.14) puede representarse en tiempo continuo mediante la siguiente integral:

$$y(t) = \int_{-\infty}^{\infty} x(k)h(t-k)dk \tag{3.61}$$

El llamado Teorema de convolución [24, pp. 95-98], [25], [26], demuestra que una multiplicación en el dominio del tiempo equivale a una convolución en el dominio de la frecuencia, de acuerdo a la siguiente expresión:

$$\frac{1}{2\pi}(X * Y)(\omega) = \mathscr{F}(x \cdot y) \tag{3.62}$$

Cuando en la integral de convolución:

$$(X * Y)(\omega) = \int_{-\infty}^{\infty} X(\lambda)Y(\omega - \lambda)d\lambda \tag{3.63}$$

λ se encuentra en radianes por segundo. Además, el citado teorema también demuestra

que una convolución en el dominio del tiempo equivale a una multiplicación en el domino de la frecuencia.

3.5 Filtros digitales

Sería muy largo hacer aquí una introducción al diseño de filtros digitales y a la teoría de polos y ceros. En su lugar, voy a hacer una simple mención de los tipos de filtros y de algunas de las técnicas utilizadas para su diseño. Un capítulo introductorio dedicado a este tema puede encontrarse en [1, pp. 584-668].

A grandes rasgos, el término *filtro digital* es otra forma de referirse a un sistema digital que aplica un determinado procesamiento a una señal. En el diseño de filtros selectivos en frecuencia las características del filtro se especifican en el dominio de la frecuencia en función de su fase y magnitud.

Los filtros ideales son no causales y por lo tanto irrealizables para aplicaciones en tiempo real. La magnitud de los filtros físicamente realizables se caracteriza por una serie de parámetros de tolerancia:

- Frecuencia de corte de la banda de paso, ω_p.
- Frecuencia de corte de la banda eliminada, ω_s
- Rizado en la banda de paso, δ_p
- Rizado en la banda eliminada, δ_s
- Banda de transición

En cuanto a la fase, un filtro FIR tiene fase lineal si su respuesta al impulso es simétrica o antisimétrica. Aplicando esta condición, es posible derivar fórmulas generales para la respuesta en frecuencia. Considerando las condiciones de simetría y el tamaño del filtro, puede hacerse la siguiente clasificación de los filtros FIR de fase lineal:

- Tipo 1: Respuesta al impulso simétrica, tamaño del filtro impar
- Tipo 2: Respuesta al impulso simétrica, tamaño del filtro par
- Tipo 3: Respuesta al impulso antisimétrica, tamaño del filtro impar
- Tipo 4: Respuesta al impulso antisimétrica, tamaño del filtro par

Lo más común hoy en día para el diseño de filtros FIR es usar software CAD basado en el algoritmo de Remez [1, pp. 613].

Un tipo de filtro FIR de especial interés para este trabajo es el filtro de media banda. El nombre deriva de su banda de paso, que abarca la mitad del espectro de frecuencias. Casi la

mitad de sus coeficientes son cero, lo que unido a su simetría da la posibilidad de realizar implementaciones muy eficientes [27, pp. 120].

Los filtros IIR no pueden tener fase lineal, aunque son en general menos costosos computacionalmente. Una de las técnicas para su diseño consiste en partir de un filtro analógico y usar una transformación para convertir el filtro al dominio digital, por ejemplo, la transformación bilineal.

En general, el diseño de un filtro digital consiste en los siguientes pasos [28]:

- Determinar la respuesta deseada en fase o magnitud.
- Seleccionar un tipo de filtro adecuado para aproximar la respuesta deseada.
- Establecer un criterio para evaluar lo buena que es la aproximación.
- Diseñar un filtro de la clase elegida y evaluar si es aceptable.
- Sintetizar el filtro usando una estructura e implementación adecuadas.
- Analizar el desempeño del filtro.

3.6 Muestreo y reconstrucción de una señal analógica

El proceso de muestreo de una señal analógica explicado en la sección 3.1 puede expresarse en términos de un sistema como la multiplicación en el dominio del tiempo de una señal continua por el tren de impulsos mostrado en la figura 3.3, lo que equivale a una convolución en el dominio de la frecuencia con el espectro de la señal original. Denotemos la transformada de Fourier del tren de impulsos como:

$$H(\omega) = 2\pi \sum_{k=-\infty}^{\infty} \frac{1}{T} \delta(\omega - k\omega_0) \tag{3.64}$$

Y el espectro de la señal de entrada analógica como $X_c(\omega)$. El espectro de la señal resultante será:

$$Y(\omega) = \frac{1}{2\pi} H(\omega) * X_c(\omega) = \frac{1}{2\pi} [2\pi \sum_{k=-\infty}^{\infty} \frac{1}{T} \delta(\omega - k\omega_0)] * X_c(\omega) \tag{3.65}$$

Simplificando y desarrollando la convolución:

$$Y(\omega) = \frac{1}{T} \int_{-\infty}^{\infty} \sum_{k=-\infty}^{\infty} \delta(\lambda - k\omega_0) X_c(\omega - \lambda) d\lambda \tag{3.66}$$

Por definición, $\delta(\lambda - k\omega_0)$ solo toma valor no nulo cuando $\lambda = k\omega_0$. Por lo tanto:

$$Y(\omega) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c(\omega - k\omega_0) \tag{3.67}$$

Es decir, tras el proceso de muestreo existen réplicas del espectro de la señal centradas en todos los múltiplos de la frecuencia de muestreo ω_0. Nótese que si la frecuencia de muestreo no es por lo menos el doble del ancho de banda de la señal se producirá la ya mencionada distorsión de aliasing, al solaparse las réplicas de los espectros.

Si definimos un espectro $R(\omega)$ como:

$$R(\omega) = \begin{cases} T & \text{si } -\omega_0/2 \leq \omega \leq \omega_0/2 \\ 0 & \text{en otro caso} \end{cases} \tag{3.68}$$

Es fácil ver que suponiendo que la frecuencia de muestreo es al menos el doble del ancho de banda de la señal, la siguiente multiplicación en el dominio de la frecuencia da como resultado la señal original:

$$Y(\omega) \cdot R(\omega) = X_c(\omega) \tag{3.69}$$

$R(\omega)$ hace nulas todas las réplicas del espectro, por eso suele llamarse *filtro anti-imagen*. En la figura 3.4 se muestra una representación gráfica de este filtro. Si realizamos la transformada inversa de Fourier resulta:

$$r(t) = \frac{1}{2\pi} \int_{-\omega_0/2}^{\omega_0/2} T e^{j\omega t} d\omega = \frac{T}{2\pi jt} \int_{-\omega_0/2}^{\omega_0/2} e^{j\omega t} jt d\omega = \frac{T}{2\pi jt} [e^{j(\omega_0/2)t} - e^{-j(\omega_0/2)t}] \tag{3.70}$$

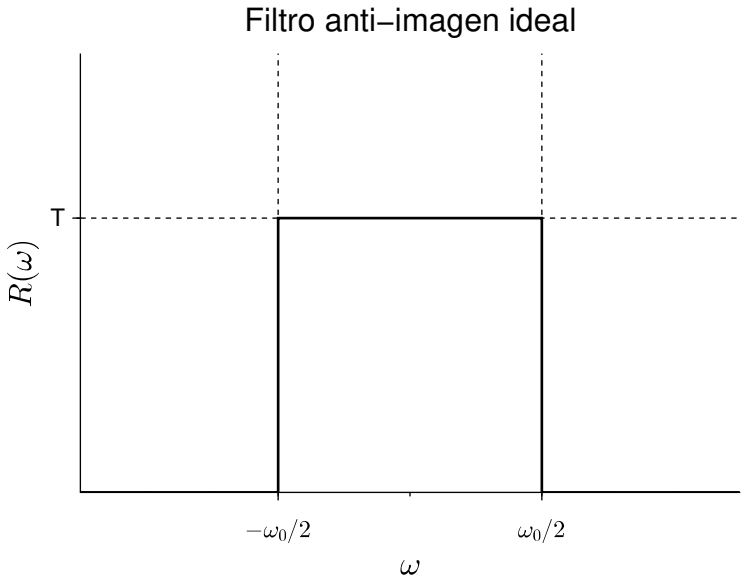


Figura 3.4: Filtro anti-imagen ideal.

Utilizando la fórmula de Euler y operando:

$$r(t) = \frac{T}{\pi t} sin \frac{\pi t}{T} = sinc \frac{\pi t}{T} \tag{3.71}$$

Donde $sinc(x)$ es la función seno cardinal. Por tanto, la convolución de la señal discreta con $r(t)$ reconstruye la señal original. Es por ello que el filtro *anti-imagen* suele denominarse *filtro de reconstrucción* cuando se encuentra en el dominio del tiempo.

Es importante ver que esta reconstrucción no es una aproximación de la señal original. Si se cumplen las condiciones anteriormente mencionadas para que no se produzca *aliasing*, la reconstrucción es en teoría perfecta. En la práctica, no es posible llevar a cabo la convolución infinita sugerida por la integral 3.61.

Por otro lado, tanto la cuantificación como los componentes electrónicos introducen errores en los procesos reales de muestreo y reconstrucción. Lo importante es que el aumento de la frecuencia de muestreo más allá del del doble del ancho de banda de la señal no aporta información relevante al proceso de reconstrucción.

Estas conclusiones se demuestran en el **Teorema de muestreo de Nyquist-Shannon**, que puede enunciarse como sigue:

■ Teorema 3.6.1 — **Teorema de muestreo.** Una señal continua en el tiempo con un ancho de banda de F_b Hercios solo puede recuperarse a partir de sus muestras si la

frecuencia de muestreo utilizada es al menos de $2F_b$ muestras por segundo.

3.7 Procesos de diezmado e interpolación

En esta sección se presenta una breve introducción a los procesos de diezmado e interpolación. Una guía completa para estas cuestiones puede encontrarse en [29, pp. 13-57].

3.7.1 Diezmado

Imaginemos que de una señal muestreada eliminamos dos de cada cuatro muestras. Esto es equivalente a multiplicar el periodo de muestreo por 2. Este proceso se conoce como diezmado y el factor que multiplica al periodo de muestreo se denota como M.

Tomando la expresión (3.67) y teniendo en cuenta la relación entre frecuencia y frecuencia normalizada, podemos definir el espectro muestreado como:

$$Y_d(\omega) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X_c\left(\frac{\omega}{T} - \frac{2\pi k}{T}\right) \tag{3.72}$$

Donde ahora ω es la frecuencia normalizada en radianes/muestra.El nuevo espectro tras un proceso de diezmado por un factor M puede expresarse como:

$$Y_{down}(\omega) = \frac{1}{MT}\sum_{k=-\infty}^{\infty} X_c\left(\frac{\omega - 2\pi k}{MT}\right) \tag{3.73}$$

Donde se ha aumentado el periodo de muestreo por un factor M, proceso equivalente a descartar muestras. Este nuevo espectro puede relacionarse con el original mediante el siguiente proceso: en primer lugar, se separa el sumatorio convirtiendo k en $i+kM$ y haciendo tomar a i valores desde 0 hasta $M-1$, de tal modo que cuando $i=0$ se suman los múltiplos de M, cuando $i=1$ se suman los múltiplos de M más la unidad, y así sucesivamente. Este proceso puede verse en forma de tabla en la figura 3.5. La suma pasa a expresarse como sigue:

$$Y_{down}(\omega) = \frac{1}{M}\sum_{i=0}^{M-1}\left[\frac{1}{T}\sum_{k=-\infty}^{\infty} X_c\left(\frac{\omega}{MT} - \frac{2\pi(i+kM)}{MT}\right)\right] \tag{3.74}$$

k \ i	0	1	2	…	M-1
⋮	⋮	⋮	⋮	⋮	⋮
-1	-M	-M+1	…	-2	-1
0	0	1	2	…	M-1
1	M	M+1	M+2	…	2M-1
2	2M	2M+1	2M+2	…	3M-1
⋮	⋮	⋮	⋮	⋮	⋮

Figura 3.5: Suma bidimensional.

Haciendo operaciones con la expresión dentro del paréntesis:

$$Y_{down}(\omega) = \frac{1}{M}\sum_{i=0}^{M-1}\left[\frac{1}{T}\sum_{k=-\infty}^{\infty} X_c\left(\frac{\omega - 2\pi i}{MT} - \frac{2\pi k}{T}\right)\right] \tag{3.75}$$

Por lo tanto:

$$Y_{down}(\omega) = \frac{1}{M}\sum_{i=0}^{M-1} Y_d\left(\frac{\omega}{M} - \frac{2\pi i}{M}\right) \tag{3.76}$$

Tras este proceso, el espectro original se achata y expande. En la práctica, como se infiere de la expresión (3.73), una señal debe ser muestreada por lo menos con una frecuencia M veces superior a la sugerida por el teorema de muestreo para poder submuestrearla un factor M sin que se produzca aliasing. Es por ello que antes de realizar este proceso se aplica a la señal un filtro paso bajo con una banda de paso hasta π/M, atenuando intensamente las frecuencias susceptibles de producir aliasing.

Estrictamente hablando, es el proceso completo de aplicar un filtro paso bajo a la señal y posteriormente submuestrearla lo que se conoce como diezmado. Sin embargo, es normal usar este término para referirse al proceso de descartar muestras.

3.7.2 Interpolación

El proceso de interpolación es opuesto al proceso de diezmado e introduce nuevas muestras en la señal, es decir, aumenta la frecuencia de muestreo. Para ello, en primer lugar se introducen ceros usando un factor L. Definamos la nueva señal obtenida, $x_{up}(n)$, como sigue:

$$x_{up}(n) = \begin{cases} x(n/L) & \text{si } n = 0, \pm L, \pm 2L... \\ 0 & \text{en otro caso} \end{cases} \tag{3.77}$$

La transformada discreta de Fourier de esta señal es:

$$X_{up}(\omega) = \sum_{n=-\infty}^{\infty} x_{up}(n)e^{-j\omega n} = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega nL} = X(\omega L) \tag{3.78}$$

Donde ω es una variable de frecuencia normalizada de acuerdo al nuevo periodo de muestreo. Nótese que si hacemos un diezmado por un factor $M = L$, recuperamos la señal original sin los ceros.

Tras este procesamiento el espectro de la señal se comprime un factor L, produciéndose ahora las copias o imágenes de la señal muestreada en múltiplos de $2\pi/L$. Es por ello que tras realizar este procesamiento se aplica a la señal un filtro paso bajo con una banda de paso hasta π/L, atenuando intensamente las nuevas réplicas del espectro.

Estrictamente hablando, es el proceso completo de introducir ceros en una señal y posteriormente aplicarle un filtro paso bajo lo que se conoce como interpolación. Sin embargo, es normal usar este término para referirse a la introducción de ceros.

3.8 Remuestreo

En el procesamiento de audio digital el proceso de remuestreo tiene dos usos principales. El primero, hacer compatibles dos sistemas con una frecuencia de muestreo diferente. El segundo, asegurar la ausencia de aliasing tras un procesamiento digital que produzca nuevas frecuencias fuera de la banda de Nyquist. El proceso completo consiste en interpolar la señal, realizar el procesamiento y finalmente diezmar la señal por el mismo factor por el que se realizo la interpolación, devolviendo la señal a su frecuencia de muestreo original. Nótese que implementar estas operaciones de forma directa es altamente ineficiente, ya que

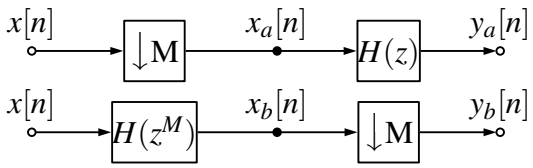


Figura 3.6: Primera identidad noble.

requiere generar muestras que serán posteriormente descartadas, en el caso del diezmado, o hacer operaciones con ceros, en el caso de la interpolación. Una forma eficiente de realizar este proceso es usar una realización de la interpolación y el diezmado en forma de filtros polifásicos, que será explicada en las siguientes secciones. Estas implementaciones permiten introducir los ceros tras aplicar el filtro paso bajo, en el caso de la interpolación, o descartar muestras antes de aplicar el filtro paso bajo, en el caso del diezmado. Más detalles sobre el tratamiento polifásico de señales digitales pueden encontrarse en [29].

3.8.1 Identidades nobles

En esta sección se van a presentar dos identidades que son de utilidad para la implementación eficiente del proceso de remuestreo.

La primera de ellas se muestra en la figura 3.6. Ambas estructuras son equivalentes. Para demostrarlo, tomemos primero $X_b(\omega)$:

$$X_b(\omega) = X(\omega)H(\omega M) \tag{3.79}$$

Por lo tanto, usando la expresión (3.76), se tiene que:

$$Y_b(\omega) = \frac{1}{M}\sum_{i=0}^{M-1} X_b(\frac{\omega}{M} - \frac{2\pi i}{M}) = \frac{1}{M}\sum_{i=0}^{M-1} X(\frac{\omega}{M} - \frac{2\pi i}{M})H(\omega - 2\pi i) \tag{3.80}$$

Es decir:

$$Y_b(\omega) = H(\omega)\frac{1}{M}\sum_{i=0}^{M-1} X(\frac{\omega}{M} - \frac{2\pi i}{M}) \tag{3.81}$$

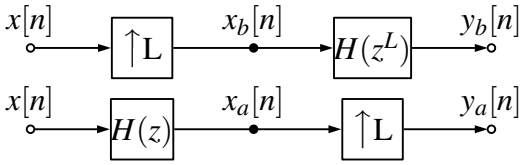


Figura 3.7: Segunda identidad noble.

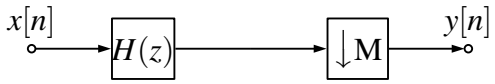


Figura 3.8: Esquema del proceso de diezmado.

Es fácil ver que este es el mismo resultado que obtenemos en la primera estructura, aplicando la expresión (3.76) a la señal $x[n]$

La segunda identidad noble se muestra en la figura 3.7. Tomando la expresión (3.78), se puede ver que:

$$Y_a(\omega) = X_a(\omega L) = X(\omega L)H(\omega L) \tag{3.82}$$

y que:

$$X_b(\omega) = X(\omega L) \tag{3.83}$$

$$Y_b(\omega) = X(\omega L)H(\omega L) \tag{3.84}$$

3.8.2 Representación polifásica del diezmado

El esquema del proceso de diezmado se muestra en la figura 3.8. La señal de salida viene dada por la suma de convolución:

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]x[nM-k] \tag{3.85}$$

Esta convolución puede convertirse en una suma bidimensional siguiendo un proceso análogo al seguido en la sección 3.7.1.

Se separa el sumatorio convirtiendo k en $i+kM$ y haciendo tomar a i valores desde 0 hasta $M-1$, de tal modo que cuando $i=0$ se suman los múltiplos de M, cuando $i=1$ se suman los múltiplos de M más la unidad, y así sucesivamente. Este proceso se muestra en forma de tabla en la figura 3.5.

Por tanto, la suma de convolución pasa a expresarse como sigue:

$$y[n] = \sum_{i=0}^{M-1} \sum_{k=-\infty}^{\infty} h[i+kM]x[(n-k)M-i] \tag{3.86}$$

Definimos:

$$p_i[k] = h[i+kM] \tag{3.87}$$

Como el componente polifásico i de $h[n]$. Nótese que esta es la respuesta al impulso desplazada i muestras y diezmada por un factor M.

De forma similar, definimos:

$$u_i[n] = x[nM-i] \tag{3.88}$$

Esta es la señal original retrasada i muestras y diezmada por un factor M. Usando estas

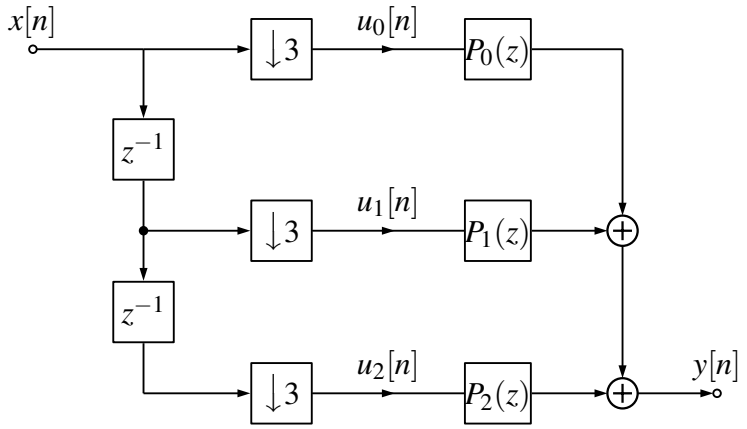


Figura 3.9: Estructura polifásica de diezmado.

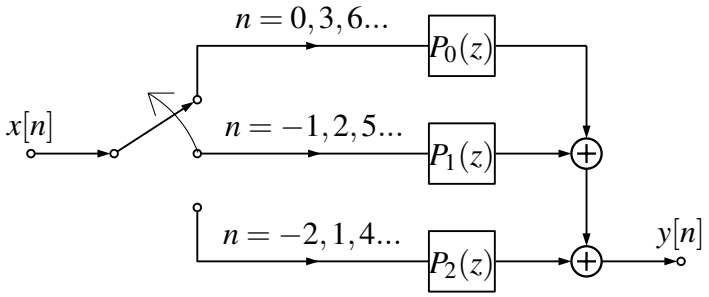


Figura 3.10: Estructura polifásica de diezmado alternativa.

dos ultimas igualdades, la suma en la expresión (3.86) puede representarse como sigue:

$$y[n] = \sum_{i=0}^{M-1} \sum_{k=-\infty}^{\infty} p_i[k] u_i[n-k] \tag{3.89}$$

Es decir:

$$y[n] = \sum_{i=0}^{M-1} p_i * u_i \tag{3.90}$$

En la figura 3.9 se representa el esquema de esta suma para $M = 3$. La figura 3.10 muestra una estructura alternativa, en la que se usa un conmutador que gira en sentido antihorario

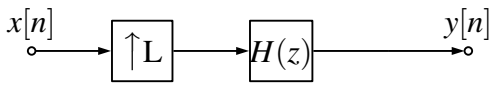


Figura 3.11: Esquema del proceso de interpolación.

cada vez que entra una nueva muestra. Ahora el diezmado se realiza antes del filtro, evitando producir muestras que serían descartadas si el diezmado se aplicara con posterioridad.

3.8.3 Representación polifásica de la interpolación

El esquema del proceso de interpolación se muestra en la figura 3.11. La señal de salida viene dada por la suma de convolución:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-Lk] \tag{3.91}$$

Nótese que ahora el factor L multiplica al parámetro de convolución en la respuesta al impulso, lo que es equivalente a introducir valores nulos en la señal.

Cambiemos los índices de esta convolución usando:

$$n = n'L + (L-1) - i \tag{3.92}$$

Donde n' es un entero y $0 \leq i \leq L-1$, quedando la suma de convolución como:

$$y[n'L+(L-1)-i] = \sum_{k=-\infty}^{\infty} x[k]h[n'L+(L-1)-i-Lk] \tag{3.93}$$

En la figura 3.12 se muestra una tabla con el valor del nuevo indexado para diferentes valores de n' e i. Sacando factor común en el indice de la respuesta al impulso:

$$y[n'L+(L-1)-i] = \sum_{k=-\infty}^{\infty} x[k]h[(n'-k)L+(L-1)-i] \tag{3.94}$$

n' \ i	0	1	2	…	L-1
⋮	⋮	⋮	⋮	⋮	⋮
-1	-1	-2	-3	…	-L
0	L-1	L-2	L-3	…	0
1	2L-1	2L-2	2L-3	…	L
2	3L-1	3L-2	3L-3	…	2L
⋮	⋮	⋮	⋮	⋮	⋮

Figura 3.12: Reindexado de la convolución de interpolación.

Ahora denotamos:

$$\begin{aligned} v_i[n'] &= y[n'L+(L-1)-i] \\ q_i[n'] &= h[n'L+(L-1)-i] \end{aligned} \tag{3.95}$$

y

$$v_i[n'] = x * q_i \tag{3.96}$$

Para recuperar la señal $y[n]$, todas las secuencias $v_i[n']$ se interpolan por un factor L, es decir, se introducen $L-1$ ceros entre muestras. Posteriormente, cada una de ellas se retrasa $i-(L-1)$ muestras. Finalmente, se suman todas ellas. Puede verse en la figura 3.12 que este procedimiento da como resultado la secuencia completa $y[n]$, al reconstruir el indexado inicial. Por tanto:

$$y[n] = \sum_{i=0}^{L-1} v_{i(\uparrow L)}[n+i-(L-1)] \tag{3.97}$$

La realización de esta expresión para $L=3$ se muestra en la figura 3.13. Ahora la interpolación se realiza después del filtro, evitando introducir operaciones con ceros.

En la figura 3.14 se muestra una estructura alternativa que hace uso de un conmutador que

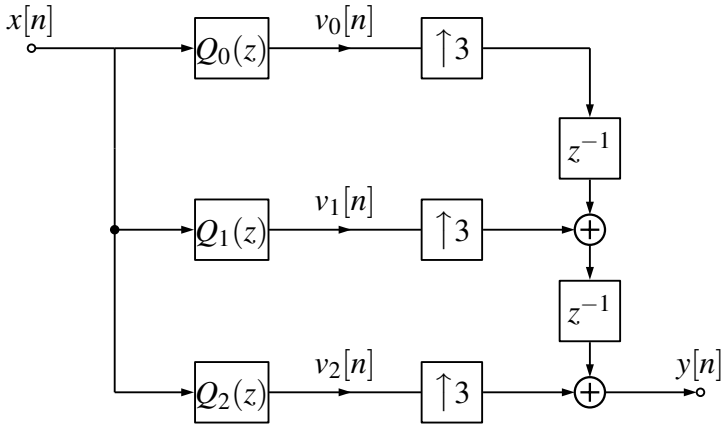


Figura 3.13: Estructura polifásica de interpolación.

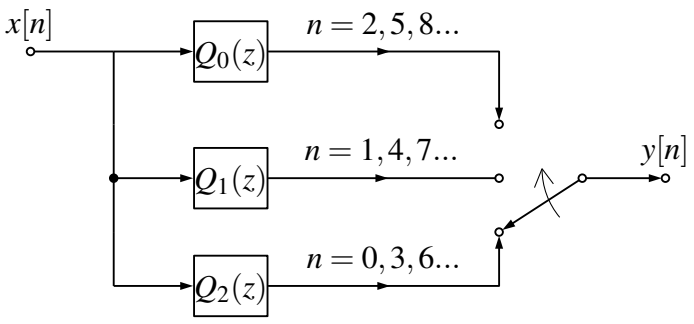


Figura 3.14: Estructura polifásica de interpolación alternativa.

gira L veces por cada muestra de entrada en sentido horario.

3.9 Consideraciones sobre la frecuencia de muestreo

Existe un debate poco formal respecto a algunas cuestiones relacionadas con la frecuencia de muestreo de las señales de audio. El consenso es que el espectro de frecuencias audible se encuentra entre los 20Hz y los 20 KHz.

Sin embargo, hay estudios que reportan percepciones de hasta 150 Khz a través de la vibración de los huesos del cráneo [30]. En cualquier caso, los equipos de audio de consumo son incapaces de reproducir frecuencias tan altas.

A efectos prácticos, en este trabajo se considera que una frecuencia de muestreo de 40000 Hz es suficiente para capturar toda la información de una señal de audio, y por lo tanto se escoge una frecuencia de muestreo de 44100 Hz, que proporciona un cierto margen para aplicar los efectos de procesado que serán usados en posteriores capítulos. En el caso del procesador embebido la frecuencia de muestreo sera de 48000 Hz, al ser la frecuencia de muestreo más cercana disponible en el codec de la placa de desarrollo.

3.10 Conclusiones

En este capítulo se han definido las señales discretas para después introducir los sistemas LTI y la transformada z. Posteriormente, se han presentado herramientas para el análisis de las señales en el dominio de la frecuencia. También se han analizado desde el punto de vista del tratamiento digital de señales los procesos de muestreo y reconstrucción, así como los de diezmado e interpolación. Finalmente, se ha hecho mención de los filtros FIR e IIR y se ha justificado la elección de la frecuencia de muestreo usada en este proyecto.

CAPÍTULO 4

Retardo fraccionario en los efectos de audio

Este capítulo pertenece al proceso de análisis del efecto de retardo. En él se explica en que consiste a grandes rasgos el retardo básico para posteriormente introducir el retardo fraccionario. Como se verá, este retardo fraccionario es la base de muchos efectos de audio interesantes. Finalmente, se hará una revisión del método usado en este trabajo para implementarlo. Todo el código Scilab de este capítulo puede encontrarse en la ruta *./Proyecto/Retardo Modulado/Diseño*.

4.1 Introducción

Los retardos en el sonido pueden ser experimentados por el oído humano en gran cantidad de circunstancias. Por ejemplo, una onda sonora reflejada en la ladera de una montaña será percibida por el oyente con un cierto retardo respecto a la señal original y una amplitud por lo general inferior. La distancia entre la fuente de sonido, el objeto donde se produce la reflexión y el receptor determinará el tiempo de retardo de cada onda sonora reflejada. En los siguientes apartados se intentará recrear este fenómeno mediante un sistema digital.

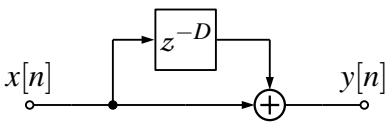


Figura 4.1: Retardo básico.

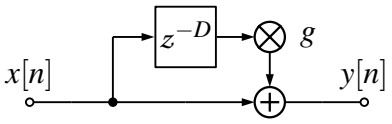


Figura 4.2: Retardo básico con amplitud relativa.

4.2 El retardo básico

En la figura 4.1 se muestra el esquema de un retardo básico. La ecuación en diferencias de este sistema es:

$$y(n) = x(n) + x(n-D) \tag{4.1}$$

A la muestra de entrada $x(n)$ se le suma otra muestra $x(n-D)$, donde D indica un retardo medido en número de muestras. Por tanto:

$$D = \tau f_s \tag{4.2}$$

Donde τ representa el tiempo de retardo en segundos y f_s es la frecuencia de muestreo utilizada.

4.2.1 Amplitud relativa

En la figura 4.2 se muestra el añadido de un parámetro g que representa la amplitud relativa de la señal retrasada respecto a la señal de referencia. Este parámetro se usa para atenuar la señal retrasada, que en los fenómenos naturales de reflexión llega normalmente con menos amplitud que la señal original. La ecuación en diferencias de este sistema es:

$$y(n) = x(n) + gx(n-D) \tag{4.3}$$

```
1  //Vector de valores de g
2  g = linspace(0.1,0.9,3);
3  //Creación de sistemas lineales
4  H = 1 + g * %z^-10;
5  H = syslin('d',H);
6  //Vector de valores de frecuencia normalizada
7  frq=0.001:0.001:0.5;
8  //Organizar los sistemas lineales en una matriz de una sola
       columna
9  v= matrix(H(1,:),3,1);
10 //Cálculo de la respuesta en frecuencia
11 rep =repfreq(v,frq);
12 //Cálculo del argumento en grados
13 arg = atan(imag(rep),real(rep));
14 phi = arg*180 ./%pi;
15 //Conversión de la amplitud a dB
16 db = 20*log(abs(rep))/log(10);
```

Listado 4.1: Análisis de la respuesta en frecuencia del retardo básico

4.2.2 Análisis de la respuesta en frecuencia

En este apartado se analizará la respuesta en frecuencia del sistema utilizando Scilab. La función de transferencia del retardo con amplitud relativa es:

$$H(z) = 1 + gz^{-D} \tag{4.4}$$

Se usarán 3 valores del parámetro g, entre 0.1 y 0.9, y un retardo D de 10 muestras. El código del listado 4.1 calcula la respuesta en frecuencia para estos valores, siendo *phi* la fase en grados y *dB* la magnitud en decibelios. Los resultados del análisis se muestran en las figuras 4.3 y 4.4. La magnitud de la respuesta en frecuencia presenta crestas y valles, que son más pronunciados cuanto más aumenta la amplitud relativa de la señal retrasada. Estas crestas y valles provienen de la anulación y la suma de fase de dos señales, la original y la retrasada.

Las crestas se corresponden con las frecuencias en las que la señal retrasada está desfasada un múltiplo de 360° respecto a la original, es decir, uno o varios periodos. Para que esto suceda, el periodo de la señal original debe ser múltiplo del retraso, D. En el ejemplo anterior, el primer máximo se da cuando el periodo $T = 10$, y por lo tanto la frecuencia normalizada $F = 1/10 = 0.1$.

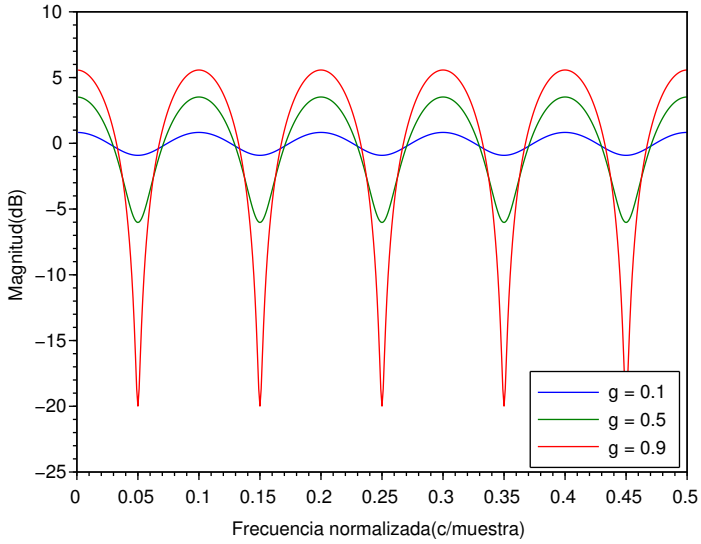


Figura 4.3: Magnitud de la respuesta en frecuencia del retardo con amplitud relativa

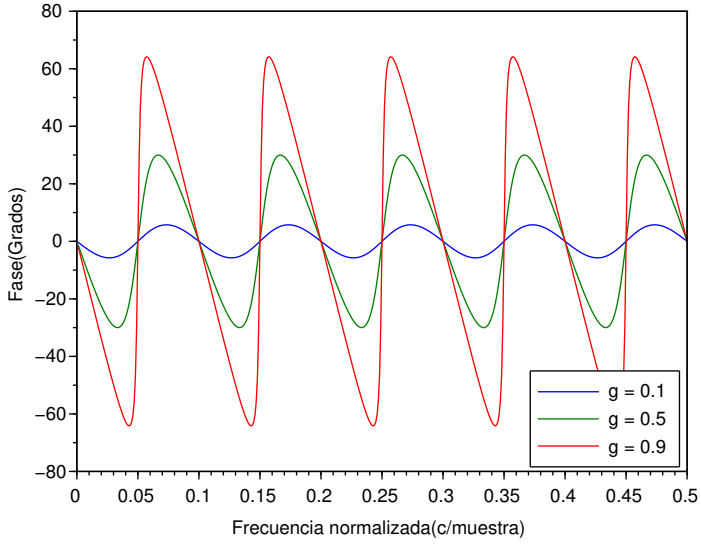


Figura 4.4: Fase de la respuesta en frecuencia del retardo con amplitud relativa

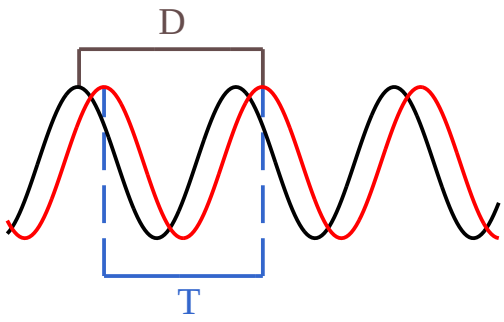


Figura 4.5: Suma de fase de una señal retrasada

En la figura 4.5 se muestra una señal sinusoidal en rojo y la correspondiente señal retrasada un número de muestras D en negro. Puede verse que si el periodo T de la señal fuera igual al número de muestras de retraso, la suma de ambas señales tendría amplitud máxima.

Cuanto mayor sea el valor de retardo D, menor será la separación entre las crestas. Para grandes valores de D el oído humano no puede identificar el efecto espectral de este filtro, y solo percibe una sensación de reverberación. Para valores pequeños de D el retardo temporal entre las dos señales es imperceptible y dado que las crestas y los valles se encuentran más separados se aprecia el efecto espectral.

Por la forma de su respuesta en frecuencia, este tipo de filtro y otros similares suelen ser llamados "filtros peine"[2, pp. 64], [1, pp. 304].

4.2.3 Retardo de fase

La fase de la respuesta en frecuencia representa el desplazamiento de fase en grados añadido a cada componente sinusoidal en la señal de entrada. En ocasiones es más intuitivo considerar el retardo de fase, definido como:

$$\hat{\tau} = -\frac{\Theta(\omega)}{\omega} \tag{4.5}$$

Donde $\Theta(\omega)$ se encuentra en radianes.

El retardo de fase representa en número de muestras el retardo experimentado por cada componente sinusoidal de la señal de entrada. Para calcularlo, se añade el código en el listado 4.2 al del listado 4.1.

El resultado se muestra en la figura 4.6. Hay un mayor retardo en las frecuencias inferiores

```
1 //Valor de la frecuencia en radianes por muestra
2 w=frq*2*%pi;
3 //Cálculo del retardo de fase para cada valor del argumento
4 for j = 1:3;
5 phasedelay(j,:) = -(arg(j,:)./w);
6 end;
```

Listado 4.2: Análisis de la fase de la respuesta en frecuencia del retardo básico

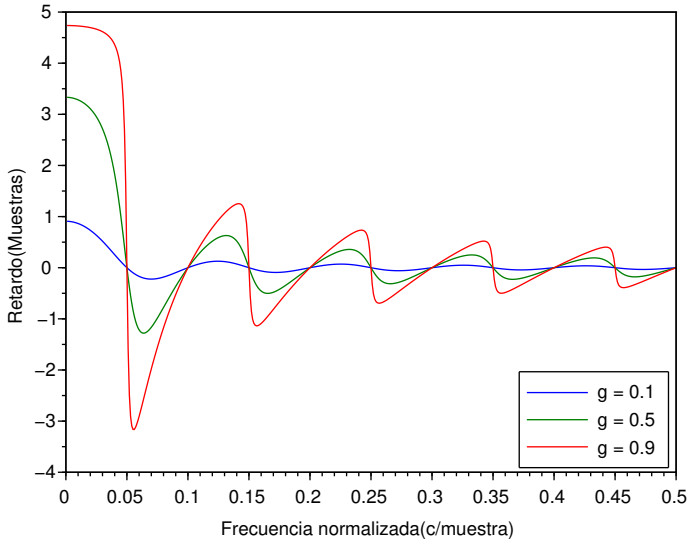


Figura 4.6: Retardo de fase.

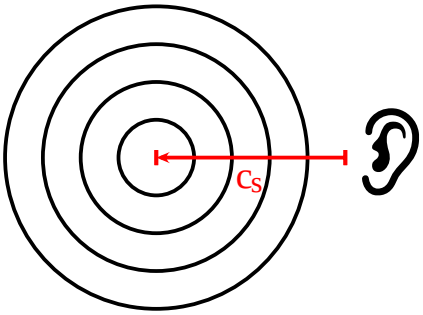


Figura 4.7: Efecto Doppler.

al punto donde se produce el primer valle, que va decreciendo a medida que aumenta la frecuencia.

4.3 Efecto Doppler

El principio del efecto Doppler se ilustra en la figura 4.7. Sin perder generalidad, podemos suponer que en el centro de los círculos existe una fuente de sonido que emite una onda sinusoidal. Si el oyente recibe en reposo f_s crestas de la onda por segundo, representadas en la figura como círculos concéntricos, al moverse con una velocidad c_s hacia la fuente de sonido las recibirá con una mayor frecuencia [2, pp. 146]:

$$f_d = f_s \left(1 + \frac{c_s}{c}\right) \tag{4.6}$$

Donde c es la velocidad del sonido en el medio. Una derivación completa de fórmulas que modelen este efecto en espacios bidimensionales y con diferentes velocidades puede encontrarse en [5, pp. 239-252].

4.4 Modulación del retardo

Como se vio en la sección anterior, una variación de la distancia a una fuente de sonido produce una variación del tono percibido por el oyente. Si tomamos el sistema de retardo examinado en los apartados anteriores y variamos el parámetro D produciremos el mismo efecto, ya que variar la distancia equivale a variar el tiempo de retardo.

La modulación del retardo es la base de muchos efectos de audio, como el chorus, el flanger o la simulación de altavoces Leslie.

En el chorus y el flanger, la señal de salida es una suma de la señal original de entrada y la señal retrasada dinámicamente. Cuando la salida está compuesta únicamente del retardo modulado con una señal sinusoidal, el unico efecto percibido es una variación periódica del tono. Este efecto se conoce como vibrato [31, pp. 1].

Este trabajo se centra en la implementación de un retardo fraccionario desde el punto de vista del tratamiento digital de señales que permita la modulación continua, lo que hace inmediata la implementación de un vibrato. Aunque no se contemple la implementación de otros efectos, esta sería muy fácil a partir del sistema básico.

4.5 El retardo fraccionario

Un retardo fraccionario permite una modulación continua con números fraccionarios. La dificultad estriba en que como se vio en capítulos anteriores, el espacio entre dos muestras no está definido en una señal digital.

Recordemos la propiedad de desplazamiento temporal de la transformada z para escribir una función de transferencia del retardo fraccionario ideal $H(z)$:

$$H(z) = z^{-(D+p)} \tag{4.7}$$

Donde D es la parte entera del retardo en número de muestras y p la parte fraccionaria. Evaluando esta expresión en $z = e^{j\omega}$ obtenemos la respuesta en frecuencia del sistema:

$$H(\omega) = e^{-j\omega(D+p)} \tag{4.8}$$

Evidentemente, la respuesta ideal en el dominio de la frecuencia de este sistema es aquella cuya magnitud es 1, es decir, no altera la amplitud de la señal:

$$|H(\omega)| = 1 \tag{4.9}$$

y cuyo retardo de fase es constante e igual a $(D+p)$:

$$\hat{\tau}(\omega) = (D+p) \tag{4.10}$$

por lo tanto la fase Ω será:

$$\Omega(\omega) = -(D+p)\omega \tag{4.11}$$

Realizando la transformada inversa de Fourier con la respuesta en frecuencia del sistema obtenemos la respuesta al impulso:

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi}[e^{-j\omega(D+p)}e^{j\omega n}]d\omega = \frac{1}{2\pi}\int_{-\pi}^{\pi}[e^{j\omega[n-(D+p)]}]d\omega \tag{4.12}$$

Resolviendo esta integral y aplicando la fórmula de Euler se llega a que:

$$h(n) = sinc(n-(D+p)) \tag{4.13}$$

Debido a que esta respuesta al impulso es infinita, es evidente que cualquier sistema FIR que implemente un retardo fraccionario será una aproximación del caso ideal. Se han propuesto numerosas formas de realizar esta aproximación [32], [33]. En las siguientes secciones se hace una revisión del método utilizado en este trabajo.

Una solución fácil para resolver el problema consiste en usar alguna forma de interpolación para inferir las muestras fraccionarias en función de las muestras existentes. Sin embargo, este es un método puramente matemático que no permite un diseño en el dominio de la frecuencia ni arroja luz sobre los aspectos relativos al tratamiento digital de señales del proceso.

4.6 Un diseño eficiente usando un diferenciador de primer orden

En esta sección se presenta una revisión del método en [34], que será el usado en este trabajo. Para ello, primero se introducen los diferenciadores, su respuesta en frecuencia y su

aproximación. Posteriormente, se presenta una revisión del método en cuestión. Finalmente, se exponen las fórmulas para diseñar diferenciadores de primer orden propuestas en [35].

4.6.1 Diferenciadores digitales y filtros FIR antisimétricos

Supongamos sin pérdida de generalidad una señal $x(t) = sin(\omega t)$. Su derivada es:

$$\frac{\partial x(t)}{\partial t} = \omega cos(\omega t) \tag{4.14}$$

Es decir, al derivar se produce un cambio de amplitud directamente proporcional a ω y un cambio de fase $\pi/2$. Podemos afirmar entonces que la respuesta en frecuencia del diferenciador ideal es:

$$D(\omega) = \omega e^{j\pi/2} = j\omega \tag{4.15}$$

Haciendo la transformada inversa de Fourier en tiempo discreto obtenemos la respuesta al impulso:

$$d(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} j\omega e^{j\omega n} d\omega = \frac{cos\pi n}{n} \tag{4.16}$$

Esta respuesta al impulso es antisimétrica, sugiriendo una aproximación por medio de un filtro FIR de fase lineal tipo 3 o tipo 4. La respuesta en frecuencia de estos filtros es:

$$F(\omega) = R(\omega)e^{j(\pi/2 - \omega(N-1)/2)} \tag{4.17}$$

donde:

$$R(\omega) = \begin{cases} \sum_{n=1}^{(N-1)/2} b(n) sin[n\omega] & \text{N impar (tipo 3)} \\ \sum_{n=1}^{N/2} b(n) sin[(n-1/2)\omega] & \text{N par (tipo 4)} \end{cases} \tag{4.18}$$

y

$$b(n) = \begin{cases} 2h\left(\frac{N-1}{2} - n\right) & \text{N impar} \quad 1 \leq n \leq \frac{N-1}{2} \\ \\ 2h\left(\frac{N}{2} - n\right) & \text{N par} \quad 1 \leq n \leq \frac{N}{2} \end{cases} \tag{4.19}$$

Haciendo $n_0 = (N-1)/2$, $F(\omega)$ puede escribirse como:

$$F(\omega) = R(\omega)e^{j(-\omega n_0 + \pi/2)} = R(\omega)e^{j\pi/2}e^{-j\omega n_0} \tag{4.20}$$

Para diseñar un diferenciador, se trata de hacer que $R(\omega)$ se aproxime lo máximo posible a ω. Si consideramos el caso ideal en el que $R(\omega) = \omega$, tenemos el siguiente diferenciador en forma de filtro FIR antisimétrico:

$$F_D(\omega) = \omega e^{j\pi/2}e^{-j\omega n_0} = (j\omega)e^{-j\omega n_0} \tag{4.21}$$

4.6.2 Diseño de un retardo fraccionario usando un diferenciador de primer orden

Pasemos ahora a explicar el método propuesto en [34]. La respuesta en frecuencia de un retardo fraccionario en el que la parte fraccionaria es variable puede representarse de la siguiente forma:

$$H(\omega, p) = e^{-j\omega(D+p)} = e^{-j\omega p}e^{-j\omega D} \tag{4.22}$$

Donde p es un retardo fraccionario en el rango $[-0.5, 0.5]$. Usando la serie de Taylor [22, pp. 177], el término e^{-jwp} puede ser expresado como un polinomio de p:

$$e^{-j\omega p} = \sum_{k=0}^{\infty} \frac{(-p)^k}{k!}(j\omega)^k \tag{4.23}$$

Este sumatorio puede separarse en dos partes, desde cero hasta M y desde $M+1$ hasta

infinito:

$$e^{-j\omega p} = \sum_{k=0}^{M} \frac{(-1)^k}{k!}(j\omega)^k p^k + p^{M+1}\left[\sum_{k=M+1}^{\infty} \frac{(-1)^k}{k!}(j\omega)^k p^{k-(M+1)}\right] \tag{4.24}$$

Multiplicamos ambos lados de la igualdad por $e^{-j\omega D}$ y denotamos el segundo término como $O(p^{M+1})$:

$$e^{-j\omega(D+p)} = \sum_{k=0}^{M} \frac{(-1)^k}{k!}(j\omega)^k e^{-j\omega D} p^k + O(p^{M+1}) \tag{4.25}$$

Si ahora hacemos $D = Mn_0$:

$$e^{-j\omega(D+p)} = \sum_{k=0}^{M} \frac{(-1)^k}{k!}(j\omega)^k e^{-j\omega k n_0} e^{-j\omega(M-k)n_0} p^k + O(p^{M+1}) \tag{4.26}$$

Usando el diferenciador de la expresión (4.21) se tiene que:

$$e^{-j\omega(D+p)} = \sum_{k=0}^{M} \frac{(-1)^k}{k!}[F_D(\omega)]^k (e^{-j\omega n_0})^{M-k} p^k + O(p^{M+1}) \tag{4.27}$$

Dado el rango de p, $O(p^{M+1})$ se aproxima a cero cuando M es grande. Por lo tanto, la anterior respuesta en frecuencia puede aproximarse mediante la siguiente expresión:

$$H_a(\omega, p) = \sum_{k=0}^{M} \frac{(-1)^k}{k!}[F_D(\omega)]^k (e^{-j\omega n_0})^{M-k} p^k \tag{4.28}$$

En [34] se establece un criterio de error y se llega a la conclusión de que la aproximación es muy buena para $M \geq 5$.

Si diseñamos un filtro $G(z)$ cuya respuesta en frecuencia aproxime la del diferenciador

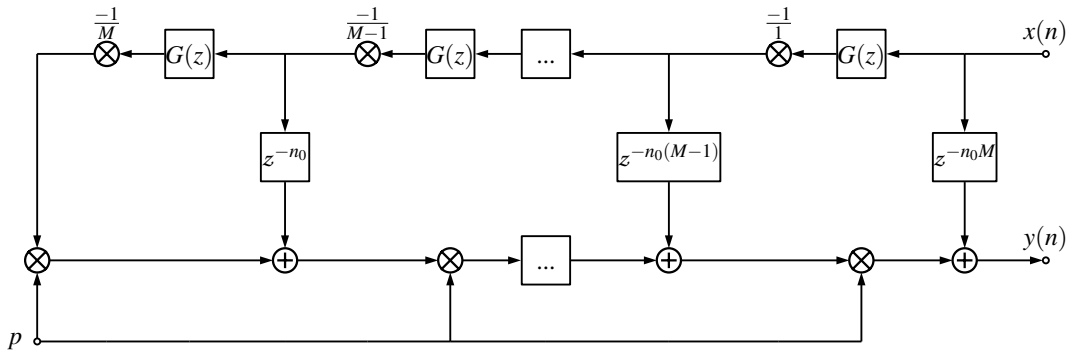


Figura 4.8: Estructura para el retardo fraccionario con diferenciador.

$F_D(\omega)$, entonces la respuesta en frecuencia del siguiente filtro aproxima $H_a(\omega, p)$:

$$\tilde{H}_a(z,p) = \sum_{k=0}^{M} \frac{(-1)^k}{k!} G(z)^k z^{-n_0(M-k)} p^k \tag{4.29}$$

Usando el método de Horner [36], este filtro puede implementarse mediante la estructura mostrada en 4.8. El problema se reduce, por tanto, a la aproximación de un diferenciador de primer orden. En [34] se encuentran ejemplos de diseño así como un análisis de la eficiencia en comparación con otras estructuras.

4.6.3 Fórmulas para el diseño de diferenciadores de primer orden

En esta sección se presenta una revisión del método propuesto en [35] para el diseño de diferenciadores de primer orden. Recordemos que la respuesta en frecuencia de un filtro FIR antisimétrico es:

$$F(\omega) = R(\omega) e^{j(\pi/2 - \omega(N-1)/2)} \tag{4.30}$$

donde:

$$R(\omega) = \begin{cases} \sum_{n=1}^{(N-1)/2} b(n) sin[n\omega] & \text{N impar (tipo 3)} \\ \\ \sum_{n=1}^{N/2} b(n) sin[(n-1/2)\omega] & \text{N par (tipo 4)} \end{cases} \tag{4.31}$$

y

$$b(n) = \begin{cases} 2h\left(\frac{N-1}{2} - n\right) & \text{N impar} \quad 1 \leq n \leq \frac{N-1}{2} \\ \\ 2h\left(\frac{N}{2} - n\right) & \text{N par} \qquad 1 \leq n \leq \frac{N}{2} \end{cases} \tag{4.32}$$

Y que para aproximar el diferenciador usado en la sección anterior, necesitamos hacer que $R(\omega)$ se aproxime a ω. El método propuesto en [35] consiste en minimizar la siguiente función de error:

$$E_{LS} = \frac{1}{\pi} \int_0^{\omega_p} [I(\omega) - R(\omega)]^2 \, d\omega \tag{4.33}$$

Donde ω_p es la frecuencia máxima de la banda de paso e $I(\omega) = \omega$. A continuación se presenta una minimización mediante cálculo diferencial para llegar a las mismas fórmulas que en [35]. $R(\omega)$ puede escribirse en forma matricial como:

$$R(\omega) = \mathbf{b}^T \mathbf{c}(\omega) \tag{4.34}$$

Donde:

$$\mathbf{b} = \begin{cases} [b(1), b(2), ..., b((N-1)/2)]^T & \text{N impar} \\ \\ [b(1), b(2), ..., b(N/2)]^T & \text{N par} \end{cases} \tag{4.35}$$

y

$$\mathbf{c}(\omega) = \begin{cases} [sin\omega, sin2\omega, ..., sin(\frac{N-1}{2}\omega)]^T & \text{N impar} \\ \\ [sin\frac{1}{2}\omega, sin\frac{3}{2}\omega, ..., sin(\frac{N-1}{2}\omega)]^T & \text{N par} \end{cases} \tag{4.36}$$

Se trata de encontrar $\mathbf{b}$ tal que E_{LS} sea mínima. Usando la condición necesaria de extremo

relativo [22, pp. 257], tenemos que:

$$\frac{dE_{LS}}{d\mathbf{b}} = \frac{d}{d\mathbf{b}}\frac{1}{\pi}\int_0^{\omega_p}[I(\omega)-R(\omega)]^2\,d\omega = 0 \tag{4.37}$$

Donde la notación $d/d\mathbf{b}$ se usa para expresar la derivada de la función respecto a cada uno de los elementos de $\mathbf{b}$, dando esta operación como resultado una matriz de derivadas. Usando la regla de derivación bajo la integral [37], [38]:

$$\frac{d}{d\mathbf{b}}\frac{1}{\pi}\int_0^{\omega_p}[I(\omega)-R(\omega)]^2\,d\omega = \frac{1}{\pi}\int_0^{\omega_p}\frac{\partial}{\partial\mathbf{b}}[I(\omega)-R(\omega)]^2\,d\omega \tag{4.38}$$

Desarrollando el cuadrado y usando la regla de la cadena:

$$\frac{1}{\pi}\int_0^{\omega_p}\frac{\partial}{\partial\mathbf{b}}[I(\omega)-R(\omega)]^2\,d\omega = \frac{2}{\pi}\int_0^{\omega_p}\frac{\partial R(\omega)}{\partial\mathbf{b}}[R(\omega)-I(\omega)]\,d\omega \tag{4.39}$$

Es fácil ver que:

$$\frac{\partial R(\omega)}{\partial\mathbf{b}} = \mathbf{c}(\omega) \tag{4.40}$$

Por lo que teniendo en cuenta la condición (4.37):

$$\int_0^{\omega_p}\mathbf{c}(\omega)\mathbf{b}^T\mathbf{c}(\omega)d\omega = \int_0^{\omega_p}I(\omega)\mathbf{c}(\omega)d\omega \tag{4.41}$$

Teniendo en cuenta que $\mathbf{b}^T\mathbf{c}(\omega) = \mathbf{c}^T(\omega)\mathbf{b}$ se llega a que:

$$\int_0^{\omega_p}\mathbf{c}(\omega)\mathbf{c}^T(\omega)d\omega\mathbf{b} = \int_0^{\omega_p}I(\omega)\mathbf{c}(\omega)d\omega \tag{4.42}$$

Denotando estas integrales como:

$$\mathbf{Q} = \int_0^{\omega_p} \mathbf{c}(\omega)\mathbf{c}^T(\omega)d\omega \tag{4.43}$$

y

$$\mathbf{i} = \int_0^{\omega_p} I(\omega)\mathbf{c}(\omega)d\omega \tag{4.44}$$

Se llega al siguiente sistema de ecuaciones lineales:

$$\mathbf{Qb} = \mathbf{i} \tag{4.45}$$

Ambas integrales pueden resolverse fácilmente usando integración por partes, dando como resultado las siguientes fórmulas:

$$i(n) = \begin{cases} \frac{\omega_p}{n}[sinc\, n\omega_p - cos\, n\omega_p] & \text{N impar} \quad 1 \leq n \leq \frac{N-1}{2} \\ \frac{\omega_p}{n-1/2}[sinc\,(n-1/2)\omega_p - cos\,(n-1/2)\omega_p] & \text{N par} \quad 1 \leq n \leq \frac{N}{2} \end{cases} \tag{4.46}$$

Donde $sinc\, x = sin\, x/x$, y:

$$q(n,m) = \begin{cases} \frac{\omega_p}{n}[sinc\,(n-m)\omega_p - sinc\,(n+m-l)\omega_p] & n \neq m \\ \frac{\omega_p}{2}[1 - sinc(2n-l)\omega_p] & n = m \\ \text{y} \quad l = 0 \quad \text{para} \quad 1 \leq n \leq \frac{N-1}{2} & \text{N impar} \\ l = 1 \quad \text{para} \quad 1 \leq n \leq \frac{N}{2} & \text{N par} \end{cases} \tag{4.47}$$

Un diferenciador de banda completa, es decir, $w_p = \pi$, solo puede simularse mediante un filtro FIR tipo 4. Se puede encontrar en este caso una fórmula directa para el vector **b**,

evitando resolver el sistema de ecuaciones lineales:

$$b(n) = \frac{8(-1)^{n+1}}{\pi(2n-1)^2} \qquad 1 \leq n \leq \frac{N}{2} \tag{4.48}$$

Finalmente, comentar que en [35] existe una errata en la expresión (5), donde hay que sustituir $n(N/2)$ por $b(N/2)$.

En el siguiente capítulo se muestra un programa en Scilab que permite el diseño de diferenciadores mediante este método, tanto de banda completa como de banda limitada.

4.7 Conclusiones

En este capítulo se ha analizado un sistema que retrasa una señal de audio con la ayuda de Scilab y se ha presentado el concepto de *retardo de fase*. Posteriormente, se ha explicado en que consiste a grandes rasgos el efecto Doppler y la modulación del retardo, haciendo énfasis en su importancia para la realización de muchos efectos de audio. Finalmente, se ha introducido el concepto de retardo fraccionario y se ha hecho una revisión de los métodos usados en este trabajo para implementarlo.

CAPÍTULO 5

Diseño de un retardo modulado

Este capítulo pertenece al proceso de diseño del efecto de retardo. En él se hace uso de los métodos explicados en el capítulo anterior para diseñar un retardo fraccionario. Posteriormente, se hacen algunas consideraciones sobre la implementación del retardo modulado que son de importancia para la elección de los parámetros de diseño del retardo fraccionario. Todo el código Scilab de este capítulo puede encontrarse en la ruta *./Proyecto/Retardo Modulado/Diseño*.

5.1 Diseño y análisis del retardo fraccionario

En esta sección se explica el contenido del archivo *"./Proyecto/RetardoModulado/Diseño/-Diferenciador y retardo fraccionario.txt"*. El archivo contiene código en Scilab para diseñar y analizar el diferenciador y el retardo fraccionario siguiendo los métodos expuestos en el capítulo anterior. Una explicación completa de todas las funciones nativas utilizadas puede encontrarse en el manual online de Scilab [39]. El código expuesto no constituye un programa, sino una herramienta de soporte para el diseño y análisis del sistema antes de pasar a su implementación.

5.1.1 Diseño de diferenciador

Para usar la misma notación que en [35], la matriz **i** usada en la sección 4.6.3 se renombra como **d**.

Función *even*

La función mostrada en el listado 5.1 sirve para comprobar si un número es par o no.

```
function[x] = even(n)
        if(pmodulo(n,2) == 1) then
         x=%f;
        else
         x=%t;
        end;
endfunction;
```

Listado 5.1: Función *even*.

```
function[x] = q(n,m,wp,N)
        l = even(N);
        if(n~=m) then
         x= (wp/2)*(sinc((n-m)*wp)-sinc((n+m-l)*wp));
        else
         x= (wp/2)*(1-sinc((2*n-l)*wp));
        end;
endfunction;
```

Listado 5.2: Función *q*.

Función *q*

La función mostrada en el listado 5.2 calcula valores para la matriz **Q** mediante su posición (m,n), la longitud del filtro N y la frecuencia límite de la banda de paso ω_p, de acuerdo a la expresión (4.47).

Función *d*

La función mostrada en el listado 5.3 calcula valores para la matriz **d** mediante su posición (n), la longitud del filtro N y la frecuencia límite de la banda de paso ω_p, de acuerdo a la

```
function[x] = d(n,wp,N)
        l = even(N);
        if(l) then
         x =  (wp/(n-1/2)) * ( sinc( (n-1/2)*wp) - cos
            ((n-1/2)*wp) );
        else
         x = (wp/n) * (sinc(n*wp) - cos(n*wp));
        end;
endfunction;
```

Listado 5.3: Función *d*.

```
function[x] = matrixq(wp,N)
          l = even(N);
          mlen = 0;
          if(l) then
           mlen = N/2;
          else
           mlen = (N-1)/2
          end;
          Q = zeros(mlen,mlen);
          for n = 1:mlen
                    for m = 1:mlen
                              Q(n,m) = q(n,m,wp,N);
                    end;
          end;
          x = Q;
endfunction;
```

Listado 5.4: Función *matrixq*.

expresión (4.46).

Función *matrixq*

La función mostrada en el listado 5.4 construye y devuelve la matriz $\mathbf{Q}$ de acuerdo al tamaño del filtro (N) y a la frecuencia límite de la banda de paso ω_p, haciendo uso de la función q.

Función *matrixd*

La función mostrada en el listado 5.5 construye y devuelve la matriz $\mathbf{d}$ de acuerdo al tamañó del filtro (N) y a la frecuencia límite de la banda de paso ω_p, haciendo uso de la función d.

Función *differentiator*

La función mostrada en el listado 5.6 construye un diferenciador haciendo uso de las funciones anteriores, tomando como parámetros el tamaño del filtro y un valor a entre 0 y 1, que representa el límite de la banda de paso. Si el diferenciador es de banda completa, se hace uso de la fórmula reducida (4.48).

La función $y = linsolve(x,z)$ computa todas las soluciones que satisfacen la expresión $xy+z=0$. Estas soluciones se dividen entre 2 para encontrar los coeficientes del filtro antisimétrico $h(n)$, de acuerdo a la expresión (4.32).

```
function[x] = matrixd(wp,N)
        l = even(N);
        mlen = 0;
        if(l) then
         mlen = N/2;
        else
         mlen = (N-1)/2
        end;
        D = zeros(mlen,1);
        for n = 1:mlen
                D(n,1) = -1*d(n,wp,N);
        end;
        x = D;
endfunction;
```

Listado 5.5: Función *matrixd*.

```
function[x] = differentiator(N,a)
        if(a==1) then
                b = zeros(N/2,1);
                for n = 1:N/2
                        b(n,1) = (8*(-1)^(n+1))/(%pi
                           *(2*n-1)^2);
                end;
        else
                Q = matrixq(a*%pi,N);
                D = matrixd(a*%pi,N);
                b = linsolve(Q,D);
        end;
        x = b./2;
endfunction;
```

Listado 5.6: Función *differentiator*.

```
//*************Diferenciador de banda limitada
N=63
a=0.92;
H = differentiator(N,a);
sol2=flipdim(H,1)
sol3 = zeros(1,1);
sol = -1*H
H = [sol2; sol3; sol]
H = H'
```

Listado 5.7: Código para diseñar un diferenciador de banda limitada con parámetros $a = 0.92$ y $N = 63$.

```
difpoly=poly(H,'z','coeff')
denomcoeff=[zeros(1,size(H,'c')-1),1]
denompoly=poly(denomcoeff,'z','coeff')
dif = difpoly/denompoly
```

Listado 5.8: Código para obtener la función de transferencia del diferenciador diseñado.

Diferenciador para el retardo fraccionario

El código mostrado en el listado 5.7 obtiene el vector **b** para un diferenciador con parámetros $N = 63$ y $a = 0.92$. Es necesario reflejar los coeficientes obtenidos mediante la función *flipdim*, de acuerdo a la expresión (4.32), introducir un cero en el punto medio y finalmente introducir el resto de coeficientes, que teniendo en cuenta que el filtro es antisimétrico se corresponden con el opuesto de los valores de H.

Posteriormente, el código del listado 5.8 crea un polinomio de z con los coeficientes del filtro. Para invertir el grado de las potencias de z se crea otro polinomio del mismo grado cuyo único coeficiente no nulo es el de la máxima potencia.

Por último, se crea un vector de frecuencias normalizadas, se declara un sistema lineal

```
//Respuesta en frecuencia del diferenciador de banda limitada
frq=0.001:0.001:0.5;
difsl = syslin('d',dif);
difrep = repfreq(difsl,frq);
frqr = frq*2*%pi;
plot(frqr,abs(difrep));
```

Listado 5.9: Código para obtener la respuesta en frecuencia del diferenciador diseñado.

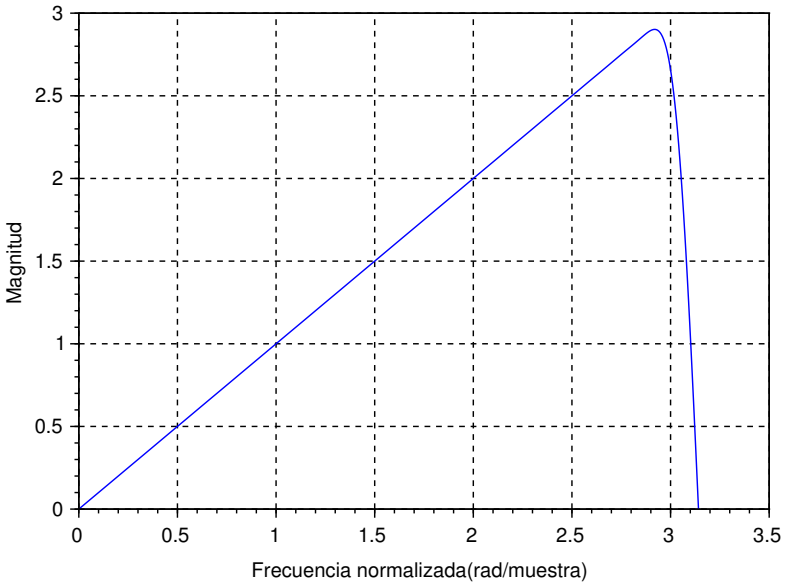


Figura 5.1: Magnitud de la respuesta en frecuencia del diferenciador de banda limitada diseñado.

en tiempo discreto con el polinomio construido anteriormente, se calcula la respuesta en frecuencia y se grafica su magnitud. Todo ello se hace mediante el código mostrado en el listado 5.9

En la figura 5.1 se muestra la magnitud de la respuesta en frecuencia del diferenciador de banda limitada diseñado. Puede verse que la aproximación es excelente dentro de la banda de frecuencias elegida.

5.1.2 Diseño del retardo fraccionario

Para analizar el sistema dado por la función de transferencia (4.29) se crea una matriz de filtros usando el diferenciador diseñado anteriormente mediante el código del listado 5.10, cada uno de ellos usando un valor diferente del parámetro p desde -0.5 hasta 0.5 a intervalos de 0.025, es decir, un total de 41 valores.

Posteriormente, se calcula la respuesta en frecuencia de cada uno de ellos mediante el código mostrado en el listado 5.11. La magnitud se convierte a decibelios y se multiplica por 10^4 para apreciar las desviaciones en una escala ampliada. Finalmente, se grafica el resultado.

Como puede apreciarse en la figura 5.2, la magnitud de la respuesta en frecuencia del filtro es excelente. La respuesta no es ideal a altas frecuencias cuando el parámetro p de retardo

```
//*************Construccion de la matriz de retardos fraccionarios
M=7;
n=31
//Matriz de filtros de retardo fraccionario
fdelay = 0;
p=-0.5:0.025:0.5
for k=0:M
        fdelay = fdelay + (((-1)^k)/factorial(k)) * dif
           ^k * %z^(-n*(M-k))*p^k
end;
fracdelay=syslin('d',fdelay)
```

Listado 5.10: Código para construir una matriz de retardos fraccionarios.

```
//Respuesta en frecuencia y conversion a dB
frq=0.01:0.01:0.5;
fracdelays = matrix(fracdelay(1,:),41,1)
rep =repfreq(fracdelays,frq);
arg = atan(   imag(   rep   ),real(     rep   )  )
phi = arg*180./%pi
db = 20*log(abs(rep))/log(10)
dbP = db*10^4
Sgrayplot(p,frq,dbP)
zm = min(dbP); zM = max(dbP);
xset("colormap",pinkcolormap(20))
colorbar(zm,zM)
```

Listado 5.11: Código para calcular la respuesta en frecuencia de los retardos fraccionarios.

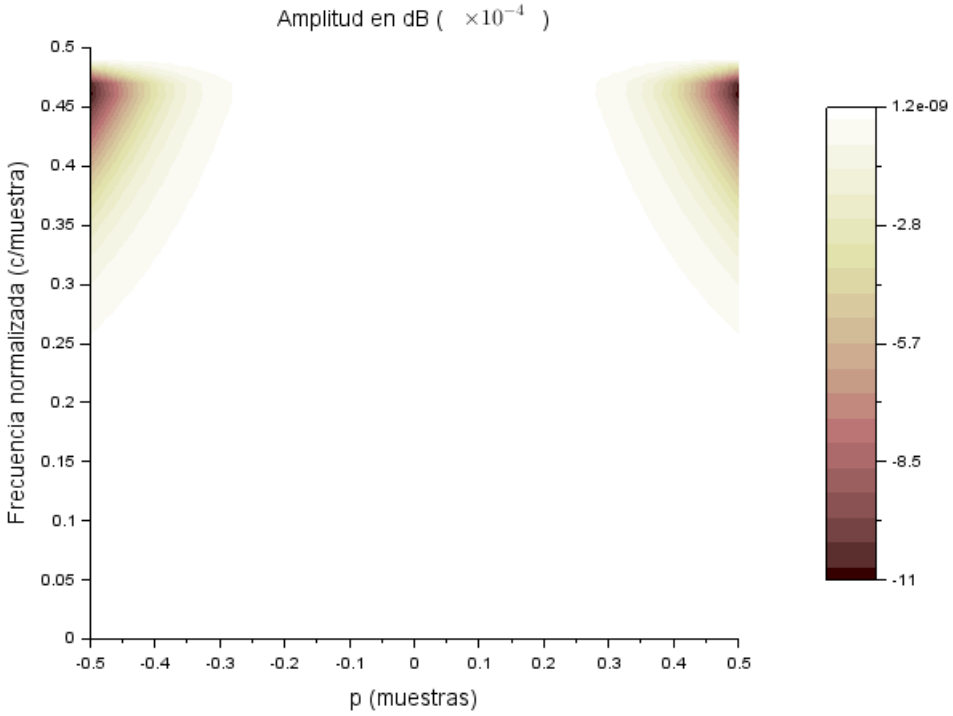


Figura 5.2: Magnitud de la respuesta en frecuencia del retardo fraccionario diseñado.

fraccionario se encuentra en los bordes de su intervalo de posibles valores. Aún en este caso, el error es despreciable para aplicaciones de audio.

5.1.3 Retardo de grupo

El retardo de grupo se define como:

$$\tau_g = -\frac{d\Theta(\omega)}{d\omega} \tag{5.1}$$

Idealmente, la fase es lineal y el retardo de grupo es una constante. Cuando la fase no es lineal, la relación entre fases de las componentes sinusoidales de entrada es alterada por el filtro. Es por eso que este tipo de distorsión de fase es conocido como *dispersión de fase* [40].

En general, la conservación de la envolvente de una señal de audio donde existen diversas frecuencias solo está garantizada cuando el retardo de grupo es una constante. Su retardo temporal en número de muestras será en ese caso igual al retardo de grupo. Si el retardo de grupo no es constante puede ser problemático para un sonido en el que el carácter dependa

```
//Delay de grupo
for k = 1:41
[tg(k,:),fr]=group(200,fracdelay(1,k));
end;
plot3d1(p,fr,tg, flag = [-1 1 4])
```

Listado 5.12: Código para calcular el retardo de grupo de los retardos fraccionarios.

de la preservación de los picos de intensidad.

5.1.4 Cálculo del retardo de grupo en el retardo fraccionario diseñado

Para calcular el retardo de grupo del filtro se utiliza la función *group* de Scilab con 200 puntos. El código puede verse en el listado 5.12 . Como puede apreciarse en la figura 5.3, los resultados son los esperados. Recordemos que en el método utilizado, el retardo $D = n_0 M$, y con $n_0 = 31$ y $M = 7$, $D = 217$. El retardo de grupo oscila, de acuerdo al valor del parámetro p, entre 216.5 y 217.5. En altas frecuencias, más allá de la banda de paso elegida para el diferenciador, el retardo de grupo no es ideal.

5.1.5 Conclusión

Los resultados son los esperados para la banda de paso elegida. Recomiendo al lector comparar los resultados obtenidos con la respuesta ideal del retardo fraccionario, expuesta en la sección 4.5. La máxima frecuencia en la banda de paso elegida, teniendo en cuenta la frecuencia de muestreo que va a utilizarse, es:

$$\frac{0.92\pi}{2\pi} 44100 = 20286\,Hz \tag{5.2}$$

Suficiente para abarcar el rango de frecuencias audible.

5.2 Retardo modulado

En esta sección se va a introducir de forma conceptual una estructura para implementar el retardo modulado. Para ello, se hará uso del esquema explicado en [31], que se reproduce en la figura 5.4.

Existe un array, *Voice[i]*, en el que se almacena una nueva muestra por la izquierda cada

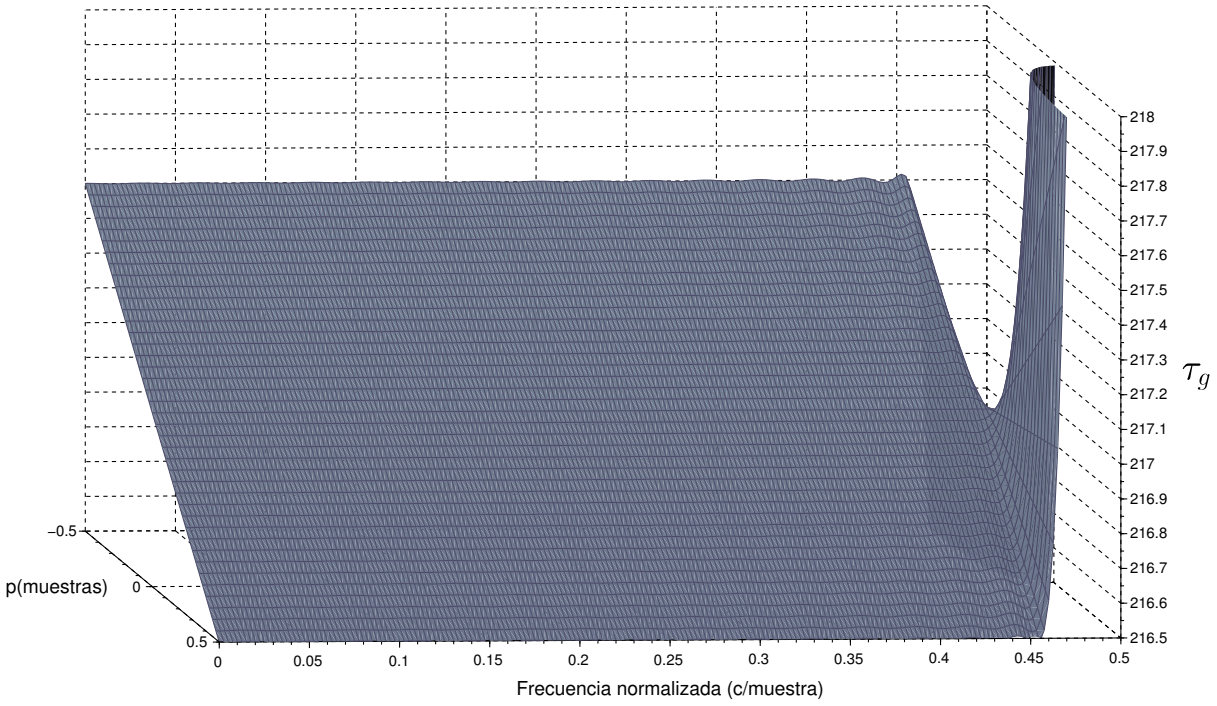


Figura 5.3: Retardo de grupo del retardo fraccionario diseñado.

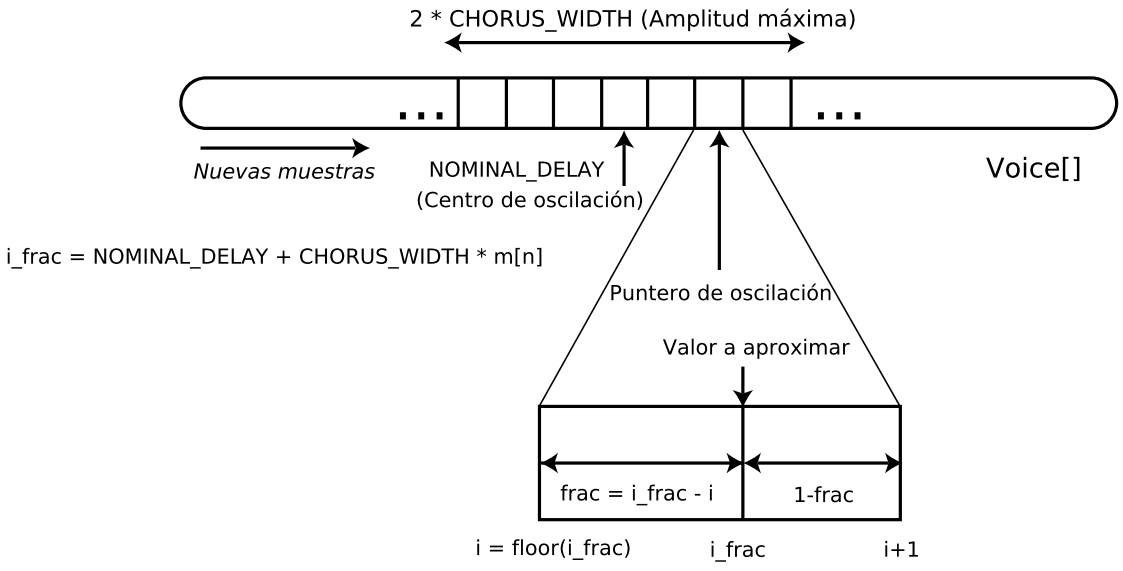


Figura 5.4: Estructura del retardo modulado.

vez que transcurre un intervalo temporal igual al periodo de muestreo, desplazando el resto de muestras hacia la derecha. Obviamente, *Voice[0]* se corresponde siempre con la última muestra y *Voice[i]* con un retraso de i muestras.

Existe un puntero que se mueve sobre este array, llamado i_frac. En cada instante temporal, la posición de este puntero es:

$$i\_frac[n] = NOMINAL\_DELAY + CHORUS\_WIDTH \cdot m[n] \tag{5.3}$$

Donde $NOMINAL\_DELAY$ es un retardo fijo que marca el centro de la oscilación, *CHORUS_WIDTH* es un parámetro que indica la amplitud y $m[n]$ es la señal de modulación. En este trabajo, dado que pretende implementarse un *vibrato*, la señal de modulación es:

$$m[n] = sin\left(2\pi f_m \frac{1}{F_s} n\right) \tag{5.4}$$

Donde f_m es la frecuencia de modulación y F_s es la frecuencia de muestreo del sistema. Nótese que esta función completa un periodo cada vez que $n = floor(F_s/f_m)$.

La mayoría de las veces, i_frac será un número fraccionario. En la figura 5.4 se usa la función *floor* (suelo), que obtiene el entero inferior más cercano a i_frac. A partir del valor de i puede obtenerse $frac$, la parte fraccionaria de i_frac, tal y como se muestra en la figura. El problema consiste, por tanto, en aproximar $Voice[i\_frac]$.

5.2.1 Ejemplo de funcionamiento

Para mejorar la comprensión de la estructura propuesta se va a escribir un código en Scilab que arroje luz sobre algunos aspectos de interés para su implementación mediante el retardo fraccionario diseñado en la sección 5.1. Tomamos los siguientes valores de prueba, aunque lo expuesto es válido para cualquier otro conjunto de valores:

- $NOMINAL\_DELAY = 4$
- $CHORUS\_WIDTH = 3$
- $f_m = 10$
- $F_s = 100$

A continuación se explica el código, mostrado en el listado 5.13.

```
1  fm = 10
2  fs = 100
3  ND = 4
4  CH = 3
5  n = 0:floor(fs/fm)
6  i_frac = ND+CH*sin(2*%pi*(fm/fs)*n)
7  i = floor(i_frac)
8  i(size(i,'c')) = i(1)
9  frac = clean(i_frac - i)
10 p = frac.*(frac<=0.5)+((-1 + frac).*(frac>0.5))
11 //Comprobación
12 max(abs(clean(p.*(p>=0)+i.*(p>=0) + p.*(p<0)+(i+1).*(p
       <0) - i_frac)))
```

Listado 5.13: Código para cálculos relacionados con un retardo modulado.

- Las cuatro primeras lineas son una asignación de los valores elegidos a las variables usadas para calcular i_frac.
- La Linea 5 crea un vector de valores de n numerados en orden creciente. Este vector tiene un tamaño igual al número de muestras que tarda en completar un periodo la señal de oscilación, es decir, $floor(F_s/f_m)$.
- La linea 6 calcula i_frac de acuerdo a la expresión (5.3).
- La linea 7 calcula los valores de i.
- La linea 8 corrige un error existente en la versión 6.0.0 de Scilab que hace que el último valor del vector no se redondee correctamente.
- La linea 9 calcula los valores de $frac$. La función *clean* se usa para hacer 0 en el vector que almacena los valores de *frac* las posiciones que contengan valores muy próximos a 0 .
- En la linea 10 se calcula un parámetro p entre -0.5 y 0.5 a partir de los valores de $frac$. Para ello, si el valor de $frac$ es menor o igual que 0.5, entonces se toma $frac$. En caso contrario, se toma $-(1-frac)$, es decir, $-1+frac$. Para entender el porqué puede observarse la figura 5.4: el valor de i_frac puede obtenerse sumando $frac$ a i o restando $1-frac$ a $i+1$.
- Esto lleva a la última linea de código, donde se suman los valores de p mayores o iguales que 0 a i y los valores de p menores que 0 a $i+1$. Si se suman ambos vectores obtenidos el resultado será obviamente el vector i_frac.
- Por tanto el resultado de esta última linea será 0, ya que la función *max* toma el valor máximo del vector.

Así, se tiene una forma de representar los valores del puntero i_frac en función de un

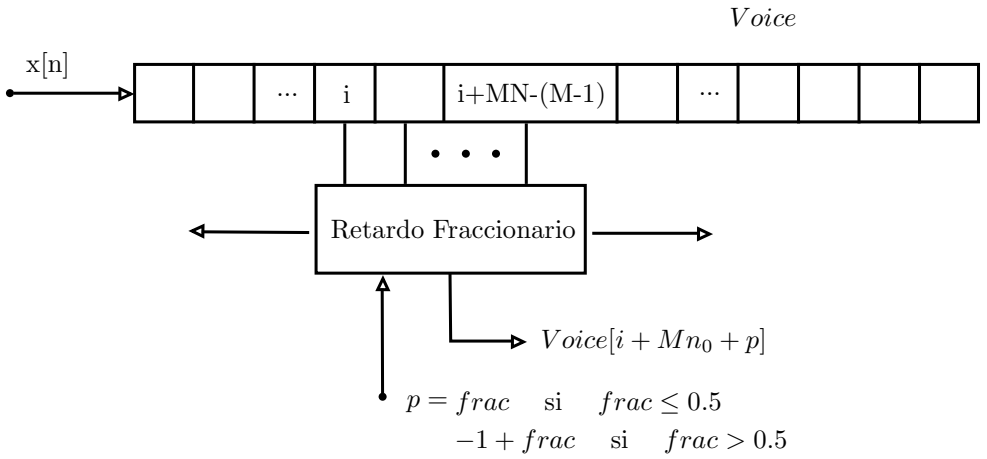


Figura 5.5: Estructura de retardo modulado con interpolador.

número fraccionario p en el intervalo $[-0.5, 0.5]$ y un entero i. Usando esta representación es posible introducir los valores del array *Voice* en el retardo fraccionario diseñado en la sección 5.1 y aproximar $Voice[i\_frac]$ usando el parámetro p.

Si observamos la estructura de la figura 4.8, puede deducirse que son suficientes $MN-(M-1)$ muestras para el cálculo del retardo en cada instante. Si al array *Voice* se le añaden $MN-(M-1)$ muestras adicionales a la derecha podemos desplazar el retardo fraccionario a lo largo del array y usarlo para obtener los valores de $Voice[i\_frac]$. Esta estructura se muestra en la figura 5.5. El retardo fraccionario puede verse como un filtro de interpolación que obtiene los valores necesarios y que oscila de izquierda a derecha.

Para entenderlo mejor, supongamos que $N = 5$, $n_0 = 2$ y $M = 3$. En la figura 5.6 se muestra una forma de realizar la estructura propuesta con un número mínimo de operaciones. Necesitamos un número de buffers igual a M, en este caso, tres. El tamaño del primer buffer empezando por la izquierda debe ser de N muestras, el segundo de $2N-1$, el tercero de $3N-2$ y así sucesivamente hasta $MN-(M-1)$ muestras en el último buffer. Este último buffer puede extenderse hacia arriba o hacia abajo hasta abarcar todo el array *Voice*, pero la estructura de retardo fraccionario solo utilizará las $MN-(M-1)$ muestras a partir de la posición en la que esté situada.

Empezando por la derecha, el segundo buffer debe contener las muestras que resultan de la convolución del diferenciador con las muestras del primer buffer. Empezando desde arriba, la linea verde indica las muestras utilizadas para obtener el primer resultado que se almacena en el segundo buffer. Puede imaginarse que esta linea se va desplazando hacia abajo, abarcando diferentes muestras, hasta que llega a la linea azul, cuyo resultado será la última muestra del segundo buffer. Una vez completada esta operación se hace exactamente lo mismo con el tercer buffer, excepto que ahora solo es necesario obtener 5

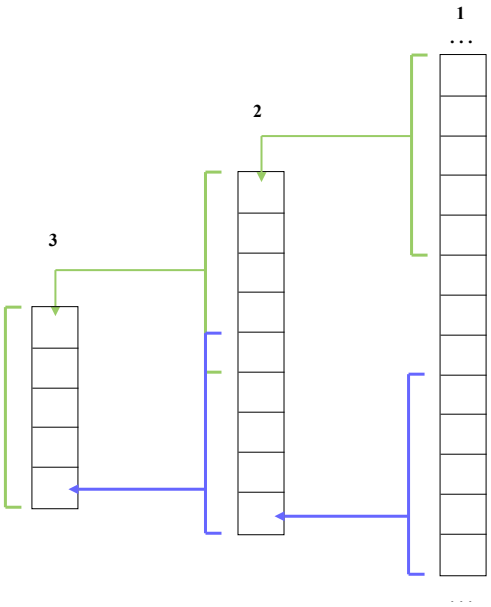


Figura 5.6: Estructura alternativa de retardo modulado con interpolador.

resultados en el ejemplo que nos ocupa.

Si una nueva muestra entra en el primer buffer por la parte superior, nótese que para actualizar la estructura es suficiente desplazar las muestras de todos los buffers hacia abajo y calcular de nuevo las muestras en la primera posición, señaladas con lineas verdes. Si por el contrario una nueva muestra entra en el primer buffer por la parte inferior, es suficiente con desplazar las muestras de todos los buffers hacia arriba y calcular las muestras en la última posición, señaladas con lineas azules. Aunque puede no resultar obvio, esto no es más que otra forma de ver la estructura de la figura 4.8. Teniendo todas las muestras actualizadas en los buffers de la estructura es siempre posible aproximar un punto cercano a la muestra Mn_0 del primer buffer dando al parámetro p un valor entre -0.5 y 0.5 y usando la estructura de la figura 4.8.

Si además de lo expuesto asumimos que el retardo fraccionario nunca se desplaza más de una muestra hacia la derecha en la figura 5.5, entonces solo es necesario realizar operaciones si se desplaza hacia la izquierda. Para entender esto, nótese que si el retardo fraccionario se desplaza hacia la derecha a la vez que una nueva muestra entra por la izquierda en la figura 5.5 las muestras de las que hace uso son idénticas. Dicho de otro modo, el primer buffer de la figura 5.6 contiene exactamente los mismos valores. Además, el tamaño necesario de los buffers pasa a ser Mn_0 para el primero, $(M-1)n_0$ para el segundo y así sucesivamente, siempre y cuando el tamaño del buffer sea mayor o igual que N, el número de coeficientes del diferenciador. Esto es así porque al establecer esta condición ya no necesitamos calcular las convoluciones señaladas con lineas azules en

la figura 5.6, pero aún necesitamos muestras suficientes para obtener la aproximación mediante la estructura de la figura 4.8, en la que el resultado de cada aplicación del filtro $G(z)$ debe ser retrasado un número de muestras determinado en cada etapa.

Si a pesar de lo expuesto la estructura no resulta obvia, recomiendo al lector examinar directamente el código en el capítulo de implementación.

5.2.2 El aliasing en el retardo modulado

En esta sección se analiza el problema del aliasing implícito que existe en el esquema de funcionamiento planteado anteriormente. Es posible decir que en la práctica existen dos desplazamientos en el array *Voice*, uno de las muestras de izquierda a derecha, provocado por la entrada de nuevas muestras, y otro del puntero i_frac, que será hacia la izquierda o hacia la derecha según el momento.

Por lo tanto, se está modificando el periodo de muestreo de la señal, T, como sigue:

$$T_d = T(1 - \Delta i\_frac) \tag{5.5}$$

Donde T_d es el nuevo periodo de muestreo. Si denotamos como F una cierta frecuencia antes de su muestreo y como f dicha frecuencia tras su normalización, el proceso de aumentar el periodo de muestreo produce una nueva frecuencia normalizada, f_d:

$$f_d = F \cdot T_d = F \cdot T(1 - \Delta i\_frac) = f(1 - \Delta i\_frac) \tag{5.6}$$

Nótese la similitud con la variación de frecuencia del efecto Doppler formulada en la sección 4.3.

En caso de que la nueva frecuencia normalizada dé como resultado un número negativo, cosa posible cuando el puntero i_frac se desplaza hacia la derecha, se está produciendo un cambio de fase. Esto sucede porque los valores empiezan a leerse hacia atrás en el tiempo.

Si el resultado es 0, el valor de salida se convierte en una constante, ya que el puntero i_frac permanece en la misma muestra.

Lo realmente preocupante es que existe la posibilidad de que la nueva frecuencia se encuentre fuera de la banda de Nyquist. Es por ello que este procesamiento se realizará tras una interpolación polifásica, que se diseñará en las siguientes secciones.

```
desp = clean(i_frac - [i_frac(size(i_frac,'c')), i_frac
    (1:size(i_frac,'c')-1)])
r_desp = 1-desp;
m_desp = max(abs(r_desp))
```

Listado 5.14: Código para cálculo del desplazamiento máximo.

Aún así, es necesario establecer un cierto límite para el valor de Δi_frac. En este trabajo se asume que que el valor de Δi_frac siempre se encuentra en el intervalo $[-0.5, 0.5]$, lo que garantiza que el factor $(1 - \Delta i\_frac)$ se encuentra dentro del intervalo $[0.5, 1.5]$. Esto significa que necesitamos interpolar por un factor 2 para realizar el procesamiento sin producir aliasing. Nótese además que se cumple la condición mencionada en la sección anterior de que el máximo desplazamiento hacia la derecha del retardo fraccionario en la figura 5.5 sea de una muestra.

Para garantizar esta condición, se toma una frecuencia máxima para la señal de oscilación de 10 Hz y un máximo para el parámetro $CHORUS\_WIDTH$ de 294 muestras. Para calcular el máximo desplazamiento dados un conjunto de valores para los parámetros del retardo modulado pueden añadirse al código del listado 5.13 las lineas del listado 5.14. Estas lineas calculan el desplazamiento del vector i_frac, le sustraen a 1 esta cantidad y finalmente obtienen el máximo factor multiplicador de frecuencia del vector resultante.

5.3 Diseño de un interpolador polifásico

En esta sección se va a diseñar un interpolador que incremente la frecuencia de muestreo del sistema a 88200 Hz, es decir, el doble de la original. Para ello, primero se va a diseñar un filtro FIR de media banda y posteriormente se va a descomponer en dos filtros polifásicos para realizar una implementación eficiente.

5.3.1 Diseño de filtro FIR de media banda

Para este diseño se utiliza la función $eqfir$ de Scilab. Esta función toma como argumentos el número de puntos del filtro; una matriz dando los extremos de la banda de paso y la banda eliminada; un vector dando la magnitud deseada para cada banda; y un último vector que indica el peso o la importancia del error en cada una de las bandas. El código completo se muestra en el listado 5.15. Las lineas adicionales construyen la función de transferencia y calculan la respuesta en frecuencia como en capítulos anteriores así que no insistiré en ello. La respuesta en frecuencia graficada puede verse en la figura 5.7. El

```
M=101
hn = eqfir(M, [0 .23; .27 .5], [1 0], [1 1]);
hn = hn .* (abs(hn)>0.00001)
hn = [hn,0]
M=M+1
hnpoly=poly(hn,'z','coeff')
denomcoeff=[zeros(1,size(hn,'r')-1),1]
denompoly=poly(denomcoeff,'z','coeff')
hnpoly = hnpoly/denompoly
hnsl = syslin('d',hnpoly);

frq=0.001:0.001:0.5;
hnrep = repfreq(hnsl,frq);
db = 20*log(abs(hnrep))/log(10)
plot(frq,db);
a = gca();
a.data_bounds=[0 -100;0.5 10];
```

Listado 5.15: Código para diseñar un filtro FIR de media banda.

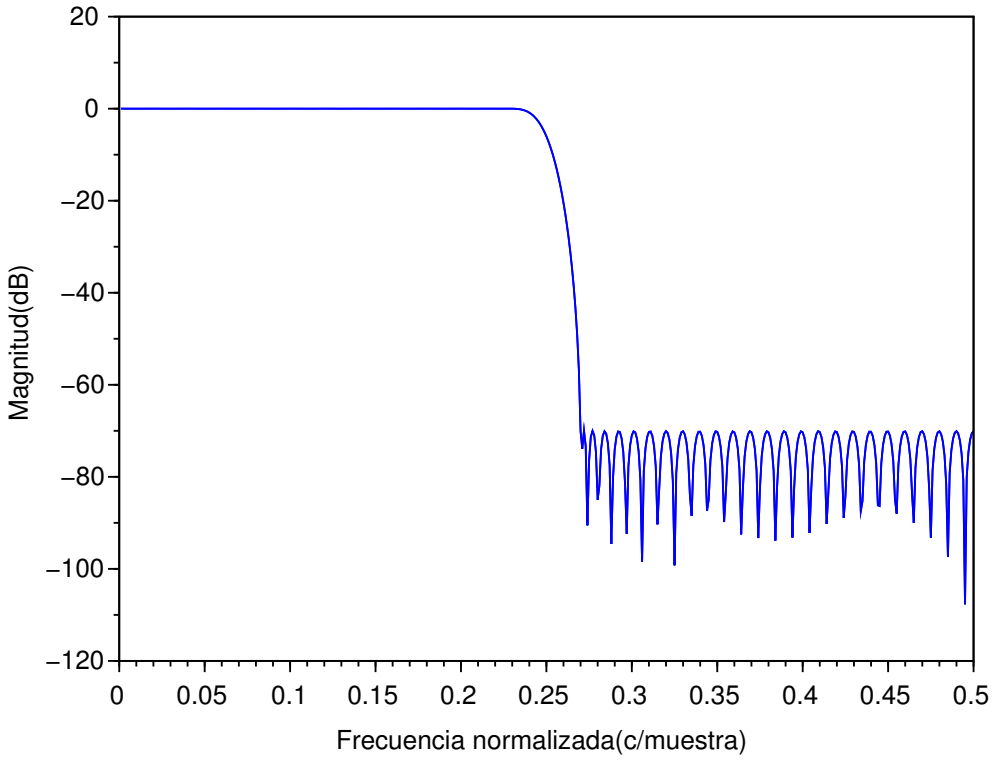


Figura 5.7: Magnitud de la respuesta en frecuencia del filtro FIR de media banda diseñado.

```
1  L=2
2  for  i=1:L
3            for  np=1:M/L
4                     q(np,i)  =  hn(np*L+(L-1)-i);
5            end;
6  end;
7  for  i=1:L
8            for  np=1:M/L
9                     p(np,i)  =  hn(i+(np-1)*L);
10           end;
11 end;
```

Listado 5.16: Descomposición polifásica del filtro de media banda.

filtro atenua un mínimo de 70 dB la banda de corte con un rizado constante. Como se mencionó anteriormente, la mitad de los coeficientes de este tipo de filtro resultan ser cero o muy cercanos a cero. La linea 3 del listado 5.15 sirve para redondear a cero todos los coeficientes menores que 0.00001 y mayores que -0.00001.

5.3.2 Estructura polifásica

Basándome en las expresiones derivadas en las secciones 3.8.2 y 3.8.3 he escrito el código del listado 5.16. La expresión $q(np,i) = hn(npL+(L-1)-i)$ se corresponde con la expresión (3.95), donde $np = n'$ y las diferentes componentes polifásicas se guardan como columnas en la matriz q. De forma análoga, la expresión $p(np,i) = hn(i+(np-1)L)$ se corresponde con la expresion (3.87), donde $np-1 = k$.

Este código se añade al del listado 5.15 y descompone el filtro en sus componentes polifásicas. Al ejecutarse, puede comprobarse que es posible realizar una implementación muy eficiente tanto para la interpolación como para el diezmado usando un filtro FIR simétrico y un simple retardo temporal como componentes polifásicas del filtro. Esto es así porque una de las componentes resulta tener un solo coeficiente distinto de cero e igual a 0.5 y la otra tiene coeficientes simétricos.

Si este desarrollo no resulta lo suficientemente clarificador puede consultarse [41], especialmente la figura 6 cerca del final del documento.

5.4 Conclusiones

En este capítulo se ha construido un modelo de retardo modulado a partir de un retardo fraccionario. Este modelo será usado como guía para las implementaciones realizadas en capítulos posteriores.

CAPÍTULO 6

Introducción a la librería JUCE

En este capítulo se hace una introducción a la librería JUCE y se explican los primeros pasos que es necesario dar para crear y configurar un nuevo proyecto. El proyecto creado será usado en el siguiente capítulo para implementar el retardo modulado diseñado.

6.1 Funcionamiento teórico de la librería JUCE

Los plugins de audio implementados usando la librería JUCE están divididos en dos componentes en forma de clases. Uno es el procesador, que se encarga de los cálculos necesarios para el procesamiento de la señal, y otro es el editor, que se corresponde con la GUI o interfaz de usuario. Dentro del editor es posible crear instancias de clases de controles gráficos que permitan al usuario interactuar con el plugin. De entre ellas, la clase *Slider* será la mas usada en este trabajo. El procesador contiene funciones para leer y modificar los parámetros del plugin, así como una rutina de *callback* que el DAW llama cada vez que necesita procesar un nuevo bloque de datos. En las siguientes secciones se describen con más detalle algunos de las funciones más importantes del procesador.

6.1.1 Rutina de *callback*

En los plugins de audio el procesamiento se hace por bloques o *buffers* que contienen un número determinado de muestras. La rutina de *callback* es llamada por el DAW cada vez que necesita procesar uno de estos bloques. De este modo, el plugin no necesita conocer en que momentos se requiere el procesamiento sino que es el DAW el que determina cuando es necesario procesar el siguiente bloque.

Cuando el DAW llama a la rutina de *callback* proporciona varios parámetros que contienen información para realizar el procesamiento. Estos incluyen el tamaño del bloque de datos, la frecuencia de muestreo configurada en el DAW, el número de canales de entrada y salida (izquierda y derecha, en equipos de audio clásicos) y un puntero a una región de

memoria que contiene el bloque de muestras a procesar. En JUCE, las muestras procesadas se almacenan en el mismo bloque de memoria en el que se recibieron.

Cuando el procesamiento termina el control vuelve al DAW, que decide que hacer con las muestras procesadas. Estas muestras suelen pasar directamente al canal principal o al siguiente plugin de una cadena de plugins.

Normalmente, el tamaño del bloque de datos es configurado en el DAW. Un tamaño de bloque grande es susceptible de provocar latencias demasiado altas, ya que se procesan muchas muestras en una sola llamada a la rutina. Un tamaño de bloque de datos inferior reduce la latencia, pero provoca que la función de *callback* sea llamada más veces por segundo.

6.1.2 La clase AudioProcessorValueTreeState

En las últimas versiones de JUCE esta clase es usada para almacenar el estado completo del plugin, incluidos sus parámetros. La clase proporciona además mecanismos para conectar los parámetros con la GUI de usuario. No he profundizado en este trabajo en la implementación interna de esta clase, pero el modo de usarla es muy específico y puede encontrarse un tutorial en la página de JUCE [42].

A grandes rasgos, el procesador del plugin contiene un miembro de la clase *AudioProcessorValueTreeState*. A este miembro se le añaden los parámetros del plugin en el constructor mediante la función *createAndAddParameter*, junto con datos adicionales como el rango del parámetro y su identificador.

El procesador contiene una función *createEditor* que es llamada por el DAW para crear una interfaz de usuario, por lo que el miembro de la clase *AudioProcessorValueTreeState* que contiene los parámetros puede ser fácilmente pasado al constructor del editor. Una vez en el editor, es posible asociar los parámetros a diferentes controles de usuario usando clases diseñadas para este propósito. En este trabajo se hará uso sobre todo de la clase *SliderAttachment*, que asocia un parámetro con un control de usuario de tipo *Slider* durante todo el tiempo de vida de una de sus instancias, actualizando automáticamente el valor tanto del parámetro cuando cambia el *Slider* como del *Slider* cuando cambia el parámetro. Todas estas cuestiones quedarán más claras cuando se explique el trabajo práctico realizado.

6.1.3 Inicialización y limpieza

Antes de que un plugin pueda empezar a funcionar es posible que sea necesario realizar una serie de acciones, como por ejemplo, reservar memoria o calcular previamente parámetros

necesarios para la ejecución.

En JUCE es posible realizar estas acciones en el constructor del procesador, que se ejecutará una vez tras la carga del plugin. Sin embargo, hay otra función más conveniente para ciertos casos llamada *prepareToPlay()*, que se ejecuta inmediatamente antes de que empiece el procesamiento de audio. Esta función es útil si, por ejemplo, se cambia la frecuencia de muestreo en el DAW y la instancia del plugin sigue siendo la misma. Es posible que en tal caso sea necesario reconfigurar el plugin o las estructuras de datos asociadas.

Por otro lado, existe una función en JUCE llamada *releaseResoures()* que se ejecuta inmediatamente después de que el host detenga el procesamiento de audio. Esta función puede usarse, por ejemplo, para vaciar bloques de almacenamiento interno del plugin.

6.2 Pasos iniciales con JUCE

El primer paso para trabajar con JUCE es descargar la librería. Esta puede descargarse de forma gratuita en `http://www.juce.com`. En el momento de realizar este proyecto, la última versión disponible es la 5.0.1.

Además, es necesario instalar los kits de desarrollo de los formatos de plugin que se pretenda generar tras la compilación. En el caso de este proyecto, el plugin se compilará solo con el formato VST de Steinberg. El kit de desarrollo puede descargarse de forma gratuita desde `http://www.steinberg.net/en/company/developers.html`. En el momento de realizar este proyecto la última versión disponible del kit de desarrollo es la 3.6.7. Sin embargo, esta versión aún no es compatible con la versión 5.0.1 de JUCE, por lo que se usará la versión 3.6.5, aún disponible en el momento de realizar el proyecto.

En el directorio raiz de JUCE se encontrará el programa *projucer*. Este programa sirve para crear nuevos proyectos de forma automatizada. El programa solicitará que nos registremos en la primera ejecución. Una vez finalizado el registro, el programa está listo para crear nuevos proyectos.

Lo primero que hay que hacer es indicarle a *projucer* la ruta de los kits de desarrollo que se vayan a utilizar, a través de File->Preferences. En el caso de este proyecto, se usa solamente el formato VST. La ruta por defecto del kit de desarrollo es `C:\SDKs\VST3_SDK`.

Posteriormente, se crea un nuevo proyecto a través de File->New Project... seleccionando el tipo de proyecto “Audio Plug-In”. Trás darle un nombre al proyecto y seleccionar un directorio de trabajo y un entorno de desarrollo se llega a la pantalla de la figura 6.1.

Haciendo clic en el icono de configuración (el primero arriba a la izquierda) pueden

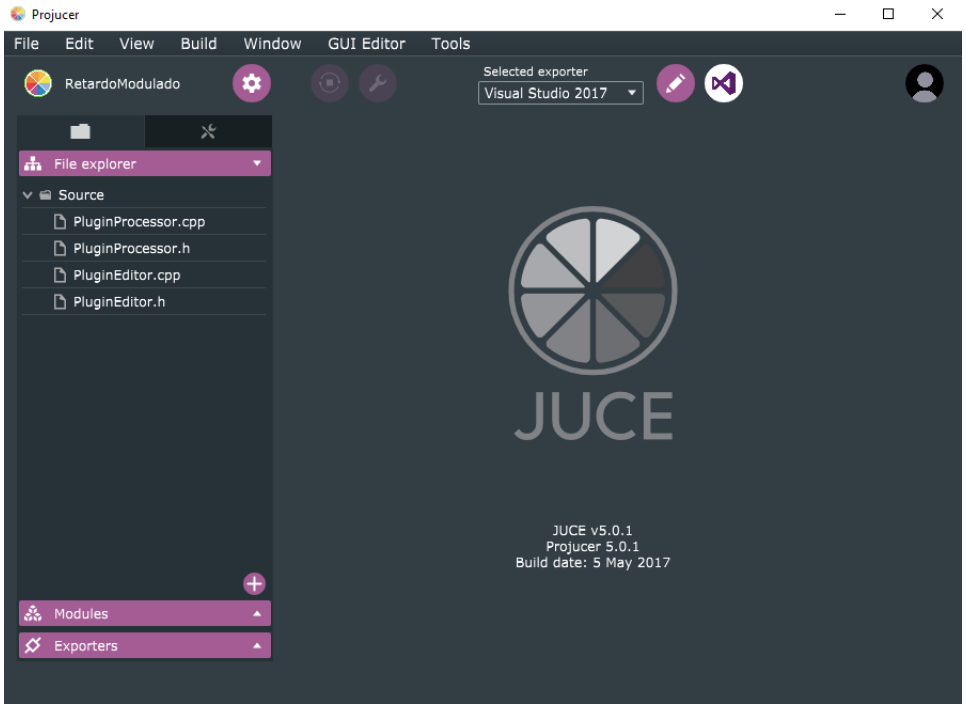


Figura 6.1: Pantalla de proyecto en *projucer* 4.2.4.

cambiarse varios campos importantes. A continuación se detallan los cambios hechos en la configuración por defecto:

- Project Version: 0.0.1
- Bundle Identifier: com.UNED.RetardoModulado
- Build VST: Verdadero
- Build VST3: Verdadero
- Build AudioUnit: Falso
- Plugin Description: Un plugin que crea un efecto de retardo modulado.
- Plugin Manufacturer: UNED
- Plugin Channel Configurations: 2,2

El último campo especifica el número de canales de audio de entrada y salida del plugin, en este caso, dos. En este punto ya se dispone de un proyecto configurado con los archivos de fuente y cabecera básicos y puede empezarse a trabajar sobre ellos.

6.3 Breve nota sobre concurrencia

Tanto los DAW como JUCE utilizan diferentes hilos para realizar el procesamiento, actualizar los parámetros y controlar los eventos de la interfaz gráfica. Aunque puede no resultar problemático para el funcionamiento en tiempo real si el diseño es bueno, en general es

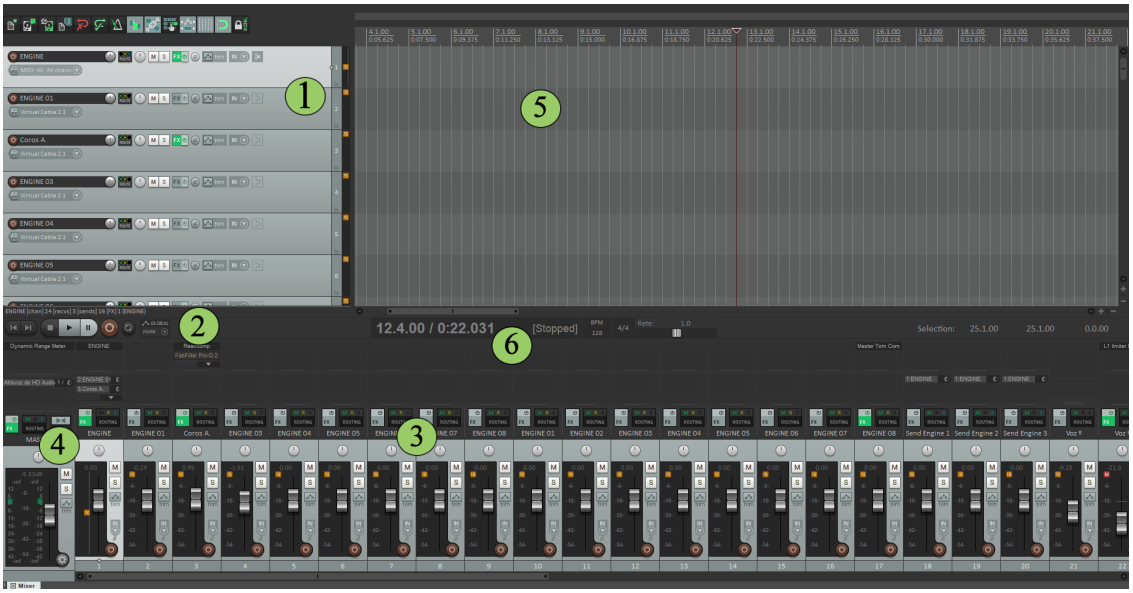


Figura 6.2: Pantalla principal de Reaper.

una mala idea bloquear de ningún modo el hilo de procesamiento.

A lo largo de este trabajo se notará que no se usa ningún mecanismo de sincronización. A veces, cuando se actualiza el parámetro de un plugin, es necesario realizar una serie de operaciones para completar los cambios y nada garantiza que estas sean atómicas. Es totalmente posible que el hilo de procesamiento continue la ejecución antes de que las acciones de actualización se hayan completado en otro hilo. Sin embargo, en el peor de los casos, esto producirá muestras incorrectas durante un muy breve periodo de tiempo hasta que el hilo donde se está ejecutando la actualización pueda completar la tarea.

Por otro lado, en general es posible asumir que las operaciones de lectura/escritura sobre cualquier elemento de tamaño igual o menor que 64 bits es atómica en arquitecturas x86, aunque esto no es cierto en todos los casos.

En resumen, lo más importante es tener presente el hecho de que los parámetros son actualizados desde un hilo de diferente y no hacer depender la estabilidad de la atomicidad de las operaciones.

6.4 Interfaz de un DAW

En esta sección intento dar una visión general del funcionamiento de un DAW a través de una breve explicación de la interfaz de Reaper.

En la figura 6.2 se muestra la pantalla de un archivo de Reaper con las principales secciones

numeradas:

1. Las pistas del archivo: En cada una de estas filas pueden introducirse sonidos, secuencias de notas MIDI o grabaciones realizadas desde la interfaz. Todas ellas se reproducirán en conjunto si se presiona el boton de *play* o su tecla abreviada, espacio.
2. Las inserciones: El panel inferior contiene una columna para cada una de las pistas. En la primera sección de esta columna es posible introducir plugins de audio. Cuando el contenido de la pista se reproduzca no pasará directamente a la pista principal, sino que será primero procesado por los plugins introducidos en esta lista en orden descendente.
3. Los *faders*: Estos controles permiten ajustar el volumen o intensidad de cada una de las pistas asi como su balance entre el canal izquierdo y el derecho.
4. El *master*: Aunque es una opción ajustable, la salida de todas las pistas se dirige por defecto a esta pista maestra en la que es posible igualmente ajustar el *fader* o introducir inserciones. Normalmente, se configura esta pista maestra para que el resultado de la mezcla pueda reproducirse directamente en un equipo externo.
5. El contenido de las pistas: Aquí se muestran en tomas o secciones desplazables los contenidos de la pista.
6. La barra de tiempo: Esta barra muestra la posición del cursor en el archivo, la velocidad de reproducción y el estado actual.

Los DAW son programas muy útiles en la producción de efectos sonoros y música. El resultado del trabajo puede exportarse a un archivo de audio con cualquier formato y características. Para el lector interesado, existen numerosos tutoriales online relacionados con el tema. Normalmente, el DAW puede configurarse para trabajar directamente con los drivers de la interfaz de audio, haciendo uso directo de sus entradas y salidas.

6.5 Conclusiones

En este capítulo se ha dado una visión general del funcionamiento de la librería JUCE y se ha expuesto el procedimiento para crear un nuevo proyecto que haga uso de la librería. Además, se ha mencionado la concurrencia presente en el software creado haciendo uso de esta librería. Finalmente, se ha incluido una breve explicación de la interfaz del DAW Reaper.

CAPÍTULO 7

Implementación de un plugin de audio para el retardo modulado

Este capítulo pertenece al proceso de codificación del efecto de retardo. En él se hace uso de las estructuras diseñadas en capítulos anteriores para implementar un plugin de audio que aplique un retardo modulado a una señal y pueda servir como base para la simulación de un vibrato u otros efectos. Todo el código del proyecto puede encontrarse en la ruta *./Proyecto/Retardo Modulado/Plugin VST/Source*.

7.1 Visión general

Al crear un nuevo proyecto Projucer extiende automáticamente la clase AudioProcessor con otra clase que contiene el identificador del proyecto en su nombre, en el caso de este trabajo el nombre final de la clase es RetardoModuladoV2AudioProcessor. En esta clase se sobrescribe la función *processBlock*(...), que sera llamada por el DAW cuando necesite procesar un bloque de muestras. En el listado 7.1 se muestra el código de esta función al principio del proyecto. Como puede verse, recibe como parámetros una referencia a un buffer que contiene las muestras a procesar y una referencia a una estructura de datos que contiene mensajes MIDI. Este segundo parámetro no será usado en este trabajo.

En principio el código no hace nada con las muestras recibidas. Los objetivos son:

1. Usar el interpolador diseñado para duplicar la frecuencia de muestreo.
2. Procesar el buffer de muestras usando el retardo fraccionario modulado diseñado de acuerdo a los parámetros establecidos por el usuario en el editor.
3. Usar el diezmador diseñado para devolver la frecuencia de muestreo a su original.

Normalmente, por cuestiones de eficiencia, los procesamientos complejos se hacen en bloque, dentro de un solo bucle a ser posible. Sin embargo, dado el carácter del proyecto y el hecho de que el plugin de audio se usa como apoyo previo a la implementación final en el procesador embebido, se diseñarán las clases de modo que el procesamiento

```
1  void RetardoModuladoV2AudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
2  {
3      const int totalNumInputChannels  = getTotalNumInputChannels();
4      const int totalNumOutputChannels = getTotalNumOutputChannels();
5          const float currentGain = *parameters.getRawParameterValue("4");
6
7      // In case we have more outputs than inputs, this code clears any output
8      // channels that didn't contain input data, (because these aren't
9      // guaranteed to be empty - they may contain garbage).
10     // This is here to avoid people getting screaming feedback
11     // when they first compile a plugin, but obviously you don't need to keep
12     // this code if your algorithm always overwrites all the output channels.
13     for (int i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
14         buffer.clear (i, 0, buffer.getNumSamples());
15     // This is the place where you'd normally do the guts of your plugin's
16     // audio processing...
17     for (int channel = 0; channel < totalNumInputChannels; ++channel)
18     {
19         float* channelData = buffer.getWritePointer (channel);
20
21         // ..do something to the data...
22     }
23 }
```

Listado 7.1: Código básico de la función processBlock.

se realice muestra a muestra. El procesador embebido tiene mucha menos capacidad de cálculo que un procesador de PC moderno, por lo que en la implementación final se hará el procesamiento del bloque completo en un solo bucle. La contrapartida es un nivel de abstracción mucho menor.

El camino a tomar es diseñar primero clases básicas que sirvan para implementar las estructuras derivadas en capítulos anteriores. Estas serán un buffer circular, un filtro FIR y un filtro FIR simétrico. Una vez hecho esto se diseñarán otras dos clases para los procesos de interpolación y diezmado. Finalmente, se diseñarán clases para llevar a cabo el retardo fraccionario y el retardo modulado. Como ayuda a las explicaciones del código se incluye un diagrama de clases parcial en la figura 7.1.

Paralelamente se diseñará un pequeño conjunto de pruebas unitarias. Para usar el motor de pruebas de Visual Studio basta con crear un proyecto de pruebas unitarias dentro de la propia solución creada por Projucer.

7.2 Buffer Circular

Un buffer circular es una estructura de datos muy usada en el tratamiento digital de señales. En la figura 7.2 se muestra el funcionamiento de un buffer circular. Las muestras nuevas se introducen por la izquierda y las viejas se descartan por la derecha, desplazando todas las muestras intermedias una posición hacia la derecha. El coste de esta operación es dependiente del tamaño del buffer.

Sin embargo, hay una forma de implementar un buffer circular sin realizar este desplazamiento. En la figura 7.3 se muestra un array y un puntero de escritura. Cuando una

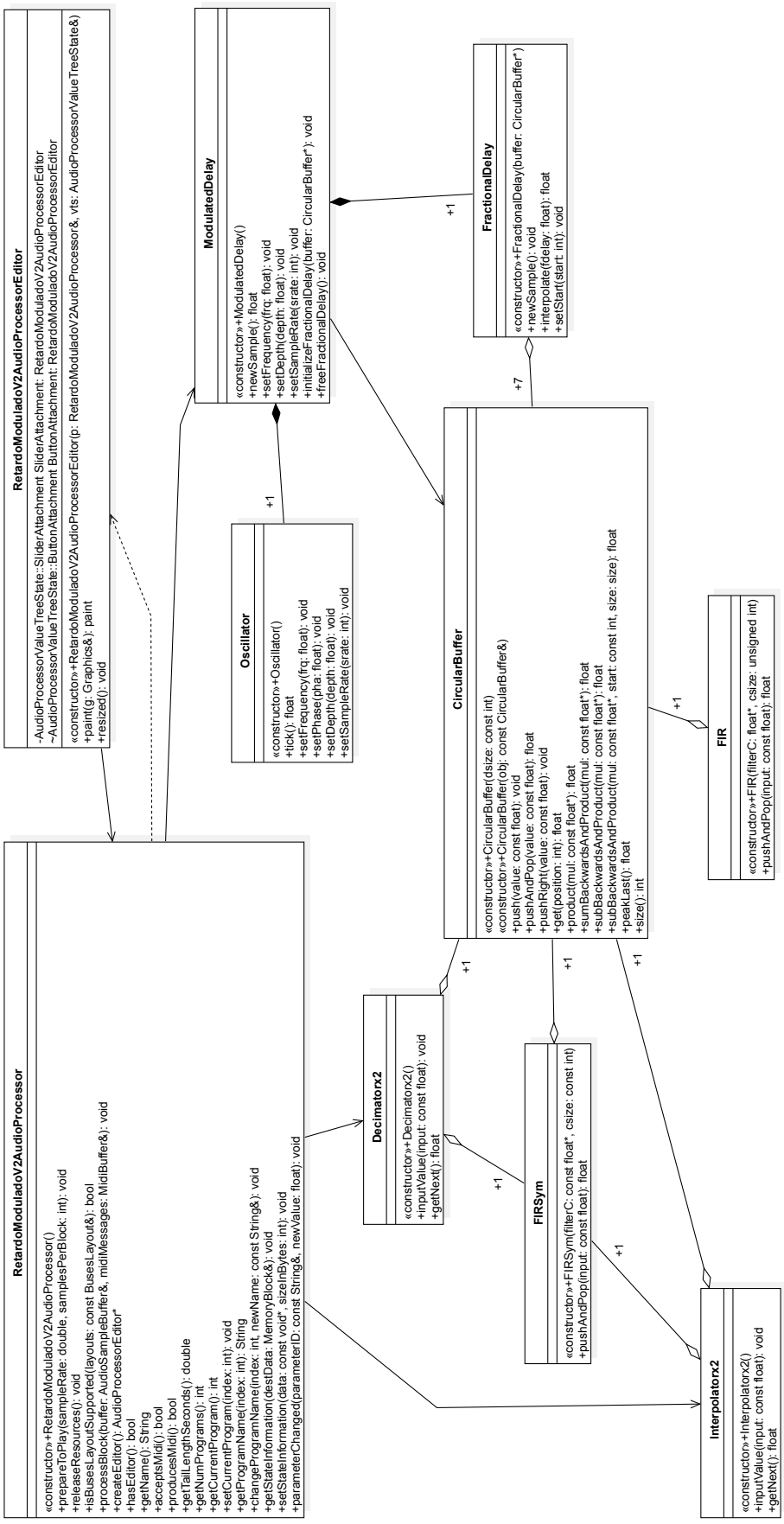


Figura 7.1: Diagrama de clases parcial del proyecto RetardoModuladoV2.

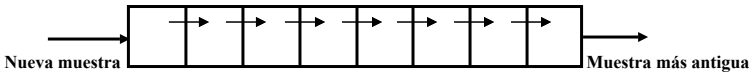


Figura 7.2: Buffer circular.

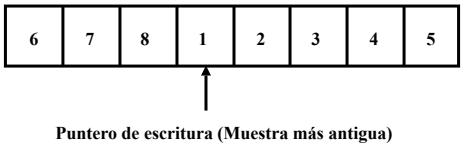


Figura 7.3: Buffer circular implementado con array y puntero de escritura.

nueva muestra es introducida se escribe en la posición del puntero y este se incrementa una posición hacia la derecha. Cuando el puntero llega al final, pasa a la primera posición. De este modo, el puntero siempre indica la última posición del buffer circular mientras que la posición anterior al puntero se corresponde con la primera. En el caso de la figura 7.3, la interpretación es que se han introducido números enteros sucesivos del 1 al 8 empezando en la cuarta posición del array. El siguiente entero (9) reemplazaría al más antiguo (1).

En este trabajo la clase que implementa el buffer circular contiene varias funciones para realizar operaciones con las muestras. Una de estas funciones puede, por ejemplo, recibir un puntero a los coeficientes de un filtro FIR y calcular el resultado de la convolución con el conjunto de muestras actual almacenado en el buffer.

En el listado 7.2 se muestra el código de la cabecera de la clase *CircularBuffer*. Los miembros privados son *float* buffer*, un puntero a la posición de memoria que contiene la primera posición del buffer circular; *int pointer*, un índice que indica el desplazamiento hasta el puntero de escritura; y *int mDsize*, un entero que indica el tamaño del buffer.

```
1  class CircularBuffer
2  {
3  public:
4          CircularBuffer(const int dsize);
5          CircularBuffer(const CircularBuffer &obj);
6          ~CircularBuffer();
7          //Use
8          void push(const float value);
9          float pushAndPop(const float value);
10         void pushRight(const float value);
11         float get(int position);
12         float product(const float* mul);
13         float sumBackwardsAndProduct(const float* mul);
14         float subBackwardsAndProduct(const float* mul);
15         float subBackwardsAndProduct(const float* mul, const int start, const int size);
16         float peakLast();
17         int size();
18 private:
19         float* buffer;
20         int pointer;
21         int mDsize;
22 };
```

Listado 7.2: Código de la cabecera de la case *CircularBuffer*.

```
#include "CircularBuffer.h"
CircularBuffer::CircularBuffer(const int dsize)
{
        buffer = new float[dsize]();
        mDsize = dsize;
        pointer = 0;
}
CircularBuffer::CircularBuffer(const CircularBuffer &obj) {
        buffer = new float[obj.mDsize]();
        mDsize = obj.mDsize;
        memcpy(buffer, obj.buffer, sizeof(float) * mDsize);
        pointer = obj.pointer;
}
CircularBuffer::~CircularBuffer()
{
        delete[] buffer;
}
void CircularBuffer::push(const float value) {
        buffer[pointer++] = value;
        if (pointer > mDsize - 1) { pointer = 0; }
}
float CircularBuffer::pushAndPop(const float value) {
        float aux = buffer[pointer];
        buffer[pointer++] = value;
        if (pointer > mDsize - 1) { pointer = 0; }
        return aux;
}
void CircularBuffer::pushRight(const float value) {
        pointer--;
        if (pointer < 0) { pointer = mDsize - 1; }
        buffer[pointer] = value;
}
float CircularBuffer::get(int position) {
        int rpos = pointer-(position+1);
        if (rpos < 0) rpos = mDsize + rpos;
        return buffer[rpos];
}
float CircularBuffer::peakLast() {
        return buffer[pointer];
}
float CircularBuffer::product(const float* mul)
{
        int bpointer = 0;
        float acc = 0;
        int apointer = pointer;
        for (unsigned int i = 0; i < mDsize; i++) {
                apointer--; if (apointer < 0) apointer = mDsize - 1;
                acc += buffer[apointer] * mul[bpointer++];
        }
        return acc;
}
float CircularBuffer::sumBackwardsAndProduct(const float* mul)
{
        int apointer = pointer;
        int bpointer = pointer;
        int cpointer = 0;
        float acc = 0;
        int top = (int)std::floor(mDsize / 2);
        for (unsigned int i = 0; i < top; i++) {
                apointer--; if (apointer < 0) apointer = mDsize - 1;
                acc += (buffer[apointer] + buffer[bpointer])*mul[cpointer];
                cpointer++;
                bpointer++; if (bpointer > mDsize - 1) bpointer = 0;
        }
        return acc;
}
float CircularBuffer::subBackwardsAndProduct(const float* mul)
{
        int apointer = pointer;
        int bpointer = pointer;
        int cpointer = 0;
        float acc = 0;
        int top = (int)std::floor(mDsize / 2);
        for (unsigned int i = 0; i < top; i++) {
                apointer--; if (apointer < 0) apointer = mDsize - 1;
                acc += (buffer[apointer] - buffer[bpointer])*mul[cpointer];
                cpointer++;
                bpointer++; if (bpointer > mDsize - 1) bpointer = 0;
        }
        return acc;
}
```

Listado 7.3: Código de la clase *CircularBuffer*, primera parte.

```
82  float CircularBuffer::subBackwardsAndProduct(const float* mul, const int start, const int size)
83  {
84          int apointer = pointer-(start+1);
85          if (apointer < 0) apointer = mDsize + apointer;
86          int bpointer = apointer - (size-1);
87          if (bpointer < 0) bpointer = mDsize + bpointer;
88          int cpointer = 0;
89          float acc = 0;
90          int top = (int)std::floor(size / 2);
91          for (unsigned int i = 0; i < top; i++) {
92                  acc += (buffer[apointer] - buffer[bpointer])*mul[cpointer];
93                  apointer--; if (apointer < 0) apointer = mDsize - 1;
94                  bpointer++; if (bpointer > mDsize - 1) bpointer = 0;
95                  cpointer++;
96          }
97          return acc;
98  }
99
100 int CircularBuffer::size() {
101         return mDsize;
102 }
```

Listado 7.4: Código de la clase *CircularBuffer*, segunda parte.

En los listados 7.3 y 7.4 se muestra la implementación de la clase *CircularBuffer*. Para representar el puntero de escritura se han usado índices en lugar de punteros en sentido estricto. A continuación se realizan algunos comentarios sobre la implementación de cada función y su propósito:

- *CircularBuffer(const int dsize)*: El constructor reserva memoria para un array de tipo float cuyo tamaño se especifica dinámicamente mediante el parámetro dsize. Este espacio es liberado en el destructor de la clase mediante el operador *delete[]*.
- *CircularBuffer(const CircularBuffer &obj)*: Este constructor recibe una referencia a un objeto de tipo *CircularBuffer* y copia el contenido de su array interno a una nueva instancia de la clase *CircularBuffer*.
- *void push(const float value)*: Esta función introduce la muestra en el array e incrementa el índice de escritura. Si el índice sobrepasa el tamaño del array debe volver a la posición inicial.
- *float pushAndPop(const float value)*: Esta función es análoga a la anterior pero además devuelve el valor de la posición del array en la que está situado el índice de escritura.
- *void pushRight(const float value)*: Esta función introduce una nueva muestra en el buffer por la derecha. Para ello, primero se desplaza el índice a la primera posición del buffer circular, es decir, se le resta la unidad, y posteriormente se introduce la muestra. Siempre que decremente el índice hay que comprobar que este no se hace menor que cero. De suceder esto, el índice debe pasar a la posición final del array. Algo análogo sucede si se incrementa el índice, este nunca debe exceder el tamaño del array donde se almacena el buffer circular.
- *float get(int position)*: Esta función devuelve el valor de una posición del buffer. 0 se corresponde con la muestra más reciente, 1 con la anterior y así sucesivamente. Por

tanto, el proceso a seguir es el siguiente: al índice de escritura se le resta la unidad para que este quede sobre la primera posición del buffer circular. Posteriormente, se resta al valor resultante el parámetro de posición. Si el resultado es negativo la posición buscada del buffer circular se encuentra comenzando a contar las posiciones negativas desde el final del array. Esta función asume que el valor de posición recibido es menor que el tamaño del buffer circular.

- *float product(const float* mul)*: Esta función asume que el parámetro es un puntero a un array del mismo tamaño que el buffer circular y multiplica la primera posición por la muestra más reciente del buffer, la segunda por la anterior y así sucesivamente. Un índice se va decrementando desde la posición marcada por el índice de escritura para recorrer el array. Es necesario comprobar en cada decremento que el valor del índice no se hace menor que cero. El valor devuelto es la suma de todos los productos.
- *float sumBackwardsAndProduct(const float* mul)*: Esta función asume que el parámetro es un puntero a un array con la mitad de tamaño que el buffer circular, suma la primera y la ultima posicion del buffer y multiplica el resultado por la primera posicion del array pasado como parámetro. El proceso continua tomando muestras del buffer desde ambos extremos hacia el centro, de tal forma que en la segunda iteración se suman la segunda y la penúltima posición del buffer y se multiplica el resultado por la segunda posición del array. El resultado devuelto es la suma de todos los productos. Esta función es de utilidad para realizar la convolución con los coeficientes de filtros FIR simétricos, reduciendo el número de productos que es necesario realizar a la mitad.
- *float subBackwardsAndProduct(const float* mul)*: Esta función es análoga a la anterior pero resta las muestras simétricas en el buffer en lugar de sumarlas. Es de utilidad para realizar filtros FIR antisimétricos.
- *float subBackwardsAndProduct(const float* mul, const int start, const int size)*: Esta función es análoga a la anterior pero toma dos parámetros adicionales: la posición de inicio y el tamaño del array pasado como parámetro. Esto permite aplicar filtros FIR antisimétricos a un buffer circular que tenga un tamaño diferente al número de coeficientes del filtro. Para ello, en primer lugar se sitúa el índice de inicio tal y como se hace en la función *float get(int position)*. Posteriormente, a este índice se le resta el número de coeficientes de entrada para encontrar la posición final. Al situar estos índices es necesario, como siempre, comprobar que las posiciones no excedan los límites del array.
- *int size()*: Esta función devuelve el tamaño del buffer circular.
- *float peakLast()*: Esta función devuelve el valor de la muestra más antigua en el buffer.

7.2.1 Pruebas

En el listado 7.5 se muestra el código de las pruebas unitarias diseñadas durante la implementación del buffer circular. Todas las pruebas consisten en introducir un conjunto de muestras en un buffer circular mediante la función *push* y posteriormente llamar a diferentes funciones de la clase. En cada prueba se incluye una aserción que comprueba que el resultado obtenido coincide con el esperado, calculado manualmente. Además, si el compilador está en la configuración de depuración el resultado obtenido se imprime en la consola.

7.3 Filtro FIR

En el listado 7.6 se muestra la cabecera de la clase *FIR*. Los miembros privados de la clase son un buffer circular y un puntero a un array de tamaño definido en tiempo de ejecución que almacena los coeficientes del filtro.

En el listado 7.7 se muestra el código de la clase. Como puede verse, el constructor recibe un puntero a los coeficientes del filtro y el tamaño del filtro. Los coeficientes son copiados al miembro de la clase mediante la función *memcpy*. El funcionamiento del filtro es sencillo: la función *pushAndPop(const float input)* recibe una muestra, la introduce en el buffer circular y calcula la convolución con los coeficientes usando la función *product*.

7.3.1 Pruebas

En el listado 7.8 se muestra el código de pruebas de la clase FIR. La prueba contiene un filtro paso bajo diseñado para producir una atenuación de 3 dB en la frecuencia normalizada 0.05 c/muestra. Introduciendo en el filtro una onda sinusoidal de amplitud 1 y frecuencia 0.05 c/muestra puede verificarse mediante aserciones que en las muestras donde se producen los picos de amplitud de la onda esta se atenúa realmente 3 dB, es decir, la amplitud de la onda resultante debería ser aproximadamente igual a la amplitud de la onda de entrada dividida entre 2.

7.4 Filtro FIR simétrico

En el listado 7.10 se muestra el código de la clase que implementa un filtro FIR simétrico. Como puede verse, el constructor recibe un puntero a un array que contiene la primera mitad de los coeficientes del filtro y el tamaño de este array. Los coeficientes son copiados al miembro de la clase que los almacena mediante la función *memcpy*. El

```
TEST_CLASS(CircularBufferTest)
	{
	public:

		TEST_METHOD(ProductTest)
		{
			float input[12] = { 2,1,1,2,1,1,2,1,1,2,1,1 };
			float coef[3] = { 2,4,1 };
			CircularBuffer de(3);
			for (unsigned int i = 0; i < 12; i++) {
				de.push(input[i]);
				float product = de.product(coef);
			#ifdef _DEBUG
				string msg = "delay input:" + std::to_string(input[i]);
				Logger::WriteMessage(msg.c_str());
				msg = "product result:" + std::to_string(product);
				Logger::WriteMessage(msg.c_str());
			#endif
				if (i + 1 % 3 == 0 && i>2)
					Assert::AreEqual(product, 9.0f);
			}
		}

		TEST_METHOD(SubBackwardsAndProductTest)
		{
			float input[12] = { 2,1,1,2,1,1,2,1,1,2,1,1 };
			float coef[3] = { 2,4,1 };
			CircularBuffer de(3);
			float product;
			for (unsigned int i = 0; i < 12; i++) {
				de.push(input[i]);
				product = 0;
				product = de.subBackwardsAndProduct(coef);
			#ifdef _DEBUG
				string msg = "delay input:" + std::to_string(input[i]);
				Logger::WriteMessage(msg.c_str());
				msg = "product result:" + std::to_string(product);
				Logger::WriteMessage(msg.c_str());
			#endif
				if (i + 1 % 3 == 0 && i>2)
					Assert::AreEqual(product, 2.0f);
			}
		}

		TEST_METHOD(SubBackwardsAndProductSETest)
		{
			float input[12] = { 2,6,4,2,3,8,1,2,4,6,8,1 };
			float coef[4] = { 2,4 };
			CircularBuffer de(12);
			for (int i = 0; i < 12; i++) {
				de.push(input[i]);
			}
			float aux = de.subBackwardsAndProduct(coef, 0, 5);
			#ifdef _DEBUG
				string msg = "product result:" + std::to_string(aux);
				Logger::WriteMessage(msg.c_str());
			#endif
			Assert::AreEqual(aux, 14.0f);
			aux = de.subBackwardsAndProduct(coef, 3, 5);
			#ifdef _DEBUG
				msg = "product result:" + std::to_string(aux);
				Logger::WriteMessage(msg.c_str());
			#endif
			Assert::AreEqual(aux, -22.0f);
		}

	};
```

Listado 7.5: Pruebas unitarias para la clase *CircularBuffer*.

```
#include "CircularBuffer.h"
class FIR
{
public:
	FIR(float* filterC, unsigned int csize);
	~FIR();
	float pushAndPop(const float input);
private:
	CircularBuffer buffer;
	float* mFilterC;
};
```

Listado 7.6: Cabecera de la clase *FIR*.

```
#include "FIR.h"
FIR::FIR(float *filterC, unsigned int csize):
buffer(csize)
{
        mFilterC = new float[csize];
        memcpy(mFilterC, filterC, sizeof(float) * csize);
}
FIR::~FIR()
{
        delete[] mFilterC;
}
float FIR::pushAndPop(const float input) {
        buffer.push(input);
        return (buffer.product(mFilterC));
}
```

Listado 7.7: Código de la clase *FIR*.

```
TEST_CLASS(FIRTest)
        {
        public:

                TEST_METHOD(FIRAttenuationTest)
                {
                        float LPfilterC[33] = { -0.00151365345728131 44f,-0.0018852172726891872f
                              ,-0.0024870497231888437f,
                                 -0.0031204059055759247f,-0.0033479739625081706f,-0.0025434719557685219f
                                       ,1.4188005322244471e-18f,
                                 0.0049209828980849414f,0.012629105331146631f,0.023167121660724481f
                                       ,0.036127597104533692f,
                                 0.050647069490689992f,0.065485879698827312f,0.079184759739924768f
                                       ,0.090273273824680 95f,
                                 0.097493754822762 11f,0.10000000000000001f,0.097493754822762 11f
                                       ,0.090273273824680 95f,
                                 0.079184759739924768f,0.065485879698827312f,0.050647069490689992f
                                       ,0.036127597104533692f,
                                 0.023167121660724481f,0.012629105331146631f,0.0049209828980849414f
                                       ,1.4188005322244471e-18f,
                                 -0.0025434719557685219f,-0.0033479739625081706f,-0.0031204059055759247f
                                       ,-0.0024870497231888437f,
                                 -0.0018852172726891872f,-0.0015136534572813144f
                        };
                        FIR LPFir(LPfilterC, 33);
                        float signal[66];
                        float buffer[66];
                        for (unsigned int i = 0; i < 66; i++) {
                                signal[i] = sin(2 * PI * 0.05 * i);
                        }
                        for (unsigned int i = 0; i < 66; i++) {
                                buffer[i] = LPFir.pushAndPop(signal[i]);
                        #ifdef _DEBUG
                                        string msg = std::to_string(buffer[i]) + ",";
                                        Logger::WriteMessage(msg.c_str());
                        #endif
                        }
                        Assert::AreEqual(0.5f, trunc(buffer[41] * 100) / 100);
                        Assert::AreEqual(-0.5f, trunc(buffer[51] * 100) / 100);
                        Assert::AreEqual(0.5f, trunc(buffer[61] * 100) / 100);
                }
        };
```

Listado 7.8: Pruebas de la clase *FIR*.

```
#include "FIR.h"
class FIRSym
{
        public:
                FIRSym(const float* filterC, const int csize);
                ~FIRSym();
                float pushAndPop(const float input);
        private:
                CircularBuffer buffer;
                float* mFilterC;
};
```

Listado 7.9: Cabecera de la clase *FIRSym*.

```
#include "FIRSym.h"
FIRSym::FIRSym(const float *filterC, const int csize) :
        buffer(csize * 2)
{
        mFilterC = new float[csize];
        memcpy(mFilterC, filterC, sizeof(float) * csize);
}

FIRSym::~FIRSym()
{
        delete[] mFilterC;
}
float FIRSym::pushAndPop(const float input) {
        buffer.push(input);
        return buffer.sumBackwardsAndProduct(mFilterC);
}
```

Listado 7.10: Código de la clase *FIRSym*.

```
TEST_CLASS(FIRSymTest)
        {
        public:

                TEST_METHOD(FIRSymAttenuationTest)
                {
                        float LPfilterC[16] = { -0.0016226630362833315f,-0.00193873474116433316f
                              ,-0.0024638768837075567f,
                                 -0.0029251064105144321f,-0.0028031760656520598f,-0.0014069429813817079f,
                                 0.0019930412581593924f,0.0079963695491252849f,0.016904897878442349f
                                       ,0.028592881996975239f,
                                 0.042447427868258281f,0.057398620296247767f,0.072040839672529736f
                                       ,0.084827753199102992f,
                                 0.094307397864001871f,0.09935423176055494f
                        };
                        FIRSym LPFir(LPfilterC, 16);
                        float signal[64];
                        float buffer[64];
                        for (unsigned int i = 0; i < 64; i++) {
                                signal[i] = sin(2 * PI * 0.05 * i);
                        }
                        for (unsigned int i = 0; i < 64; i++) {
                                buffer[i] = LPFir.pushAndPop(signal[i]);
                        #ifdef _DEBUG
                                string msg = std::to_string(buffer[i]) + ",";
                                Logger::WriteMessage(msg.c_str());
                        #endif
                        }
                        Assert::AreEqual(0.49f, trunc(buffer[41] * 100) / 100);
                        Assert::AreEqual(-0.49f, trunc(buffer[51] * 100) / 100);
                        Assert::AreEqual(0.49f, trunc(buffer[61] * 100) / 100);
                }

        };
```

Listado 7.11: Pruebas de la clase *FIRSym*.

funcionamiento del filtro es análogo al del filtro FIR, exceptuando que se hace uso de la función *sumBackwardsAndProduct(const float* mul)* para calcular la convolución con los coeficientes.

7.4.1 Pruebas

En el listado 7.11 se muestra el código de prueba para la clase *FIRSym*. El funcionamiento es análogo al del código de prueba de la clase *FIR*, exceptuando que en este caso se usan solo la mitad de los coeficientes del filtro. El error acumulado hace que el resultado no sea exactamente el mismo, por eso en las aserciones se usa el valor 0.49 en lugar de 0.5.

```
#include "../FIRSym.h"
class Interpolatorx2
{
        public:
                Interpolatorx2();
                void inputValue(const float input);
                float getNext();
        private:
                static const float int_filterC_1[25];
                FIRSym fira;
                CircularBuffer firb;
                int s;
                float a;
                float b;
};
```

Listado 7.12: Cabecera de la clase *Interpolatorx2*.

```
#include "Interpolatorx2.h"
const float Interpolatorx2::int_filterC_1[25] = { 0.000535f,  -0.0005685f,   0.0008606f,  -0.001243f,
0.001733f, -0.0023495f,   0.0031137f,  -0.0040499f,   0.0051839f,  -0.006548f,   0.0081774f,
-0.0101163f, 0.01242f,  -0.0151585f,   0.0184269f,  -0.0223577f,   0.0271442f,  -0.0330818f,
0.0406496f,-0.0506774f,   0.0647381f,  -0.086212f,   0.1239052f,  -0.21014f,   0.6359283f };
Interpolatorx2::Interpolatorx2():
fira(int_filterC_1, 25), firb(24)
{
        s = 0;
}
void Interpolatorx2::inputValue(const float input) {
        a = fira.pushAndPop(input);
        b = firb.pushAndPop(input);
}
float Interpolatorx2::getNext() {
                switch (s) {
                case 0:
                        s++;
                        return a;
                        break;
                case 1:
                        s = 0;
                        return b;
                        break;
                }
                return 0;
}
```

Listado 7.13: Código de la clase *Interpolatorx2*.

7.5 Interpolador

La implementación del interpolador es directa a partir de lo expuesto en la sección 5.3. Recomiendo al lector tener presente también la figura 3.14. En el listado 7.12 se muestra la cabecera de la clase *Interpolatorx2*. La clase contiene un array estático que almacena la mitad de coeficientes de una de las componentes polifásicas del filtro de media banda. Dado que la componente es simétrica no es necesario almacenar el resto de coeficientes.

En el listado 7.13 se muestra el código de la clase *Interpolatorx2*. La clase contiene dos funciones. La función *void inputValue(const float input)* se llama cada vez que se quiere introducir una muestra en el interpolador. La nueva muestra se introduce en ambas componentes polifásicas del filtro. La función *float getNext()* se usa para extraer muestras del interpolador y debe ser llamada dos veces cada vez que se introduce una nueva muestra. El entero s se usa como conmutador para alternar entre las componentes polifásicas. Una vez más, recomiendo al lector revisar la figura 3.14 y la sección 5.3 si lo expuesto no

```
#include "../FIRSym.h"
class Decimatorx2
{
public:
        Decimatorx2();
        void inputValue(const float input);
        float getNext();

private:
        static const float int_filterC_1[25];
        FIRSym fira;
        CircularBuffer firb;
        int s;
        float a;
        float b;
};
```

Listado 7.14: Cabecera de la clase *Decimatorx2*.

```
#include "Decimatorx2.h"
const float Decimatorx2::int_filterC_1[25] = { 0.0002675f,  -0.0002843f,   0.0004303f,
-0.0006215f,   0.0008665f,  -0.0011748f,   0.0015569f,  -0.0020249f,   0.0025919f,  -0.003274f,
 0.0040887f,  -0.0050582f,   0.00621f,  -0.0075792f,   0.0092134f,  -0.0111788f,   0.0135721f,
-0.0165409f,   0.0203248f,  -0.0253387f,   0.032369f,  -0.043106f,   0.0619526f,  -0.10507f,
 0.3179641f };

Decimatorx2::Decimatorx2() :
        fira(int_filterC_1, 25), firb(25)
{
        s = 0;
}
void Decimatorx2::inputValue(const float input) {
        switch (s) {
        case 0:
                s++;
                a = fira.pushAndPop(input);
                break;
        case 1:
                s = 0;
                b = firb.pushAndPop(input);
                break;
        }
}
float Decimatorx2::getNext() {
                return a + b*0.5f;
}
```

Listado 7.15: Código de la clase *Decimatorx2*.

resulta claro.

7.6 Diezmador

La implementación del diezmador es también directa a partir de lo expuesto en la sección 5.3. Recomiendo al lector tener presente también la figura 3.10. En el listado 7.14 se muestra la cabecera de la clase *Decimatorx2*. La clase contiene una vez más un array estático que almacena la mitad de los coeficientes de la componente polifásica del filtro de media banda. Dado que la componente es simétrica no es necesario almacenar el resto de coeficientes.

En el listado 7.15 se muestra el código de la clase *Decimatorx2*. La clase contiene dos funciones. La función *void inputValue(const float input)* se llama cada vez que se quiere introducir una muestra en el diezmador. La nueva muestra se introduce alternativamente

```
1   TEST_CLASS(IDTest)
2           {
3           public:
4
5                   TEST_METHOD(InterpolationTest)
6                   {
7                           Interpolatorx2 Ix;
8                           float signal[200];
9                           float buffer[400];
10                          for (unsigned int i = 0; i < 200; i++) {
11                                  signal[i] = sin(2 * PI * 0.1 * i);
12                          }
13                          ofstream outputFile("InterpolationTest.csv");
14                          int j;
15                          for (unsigned int i = 0; i < 200; i++) {
16                                  Ix.inputValue(signal[i]);
17                                  j = 2 * i;
18                                  buffer[j] = Ix.getNext();
19                                  string msg = std::to_string(buffer[j]) + ",";
20                                  outputFile << msg;
21                                  buffer[j + 1] = Ix.getNext();
22                                  msg = std::to_string(buffer[j + 1]) + ",";
23                                  outputFile << msg;
24                          }
25                          outputFile.close();
26                  }
27
28                  TEST_METHOD(InterpolationAndDecimationTest)
29                  {
30                          Interpolatorx2 Ix;
31                          Decimatorx2 Dx;
32                          float signal[500];
33                          for (unsigned int i = 0; i < 500; i++) {
34                                  signal[i] = sin(2 * PI * 0.1 * i);
35                          }
36                          ofstream outputFile("InterpolationAndDecimationTest.csv");
37                          for (unsigned int i = 0; i < 500; i++) {
38                                  Ix.inputValue(signal[i]);
39                                  Dx.inputValue(Ix.getNext());
40                                  Dx.inputValue(Ix.getNext());
41                                  string msg = std::to_string(Dx.getNext()) + ",";
42                                  outputFile << msg;
43                          }
44                          outputFile.close();
45                  }
46
47
48          };
```

Listado 7.16: Pruebas de la clase *Interpolatorx2*.

en cada componente polifásica del filtro. La función *float getNext()* se usa para extraer muestras del diezmador y debe ser llamada una vez cada dos veces que se introduzca una nueva muestra. El entero s se usa como conmutador para alternar entre las componentes polifásicas del filtro. Una vez más, recomiendo al lector revisar la figura 3.10 y la sección 5.3 si lo expuesto no resulta claro.

7.6.1 Pruebas

Las pruebas para el diezmador y el interpolador se implementaron de forma conjunta. El código completo se muestra en el listado 7.16. La función *InterpolationTest* interpola una señal sinusoidal con una frecuencia de 0.1 c/muestra e introduce el resultado en un archivo csv separado por comas. La función *InterpolationAndDecimationTest* es análoga pero en lugar de escribir el resultado de la interpolación en el archivo csv introduce las muestras resultantes en el diezmador y escribe el resultado final. Utilizando la función *csvread* de scilab es posible leer estos archivos y analizar el resultado en el dominio de la frecuencia.

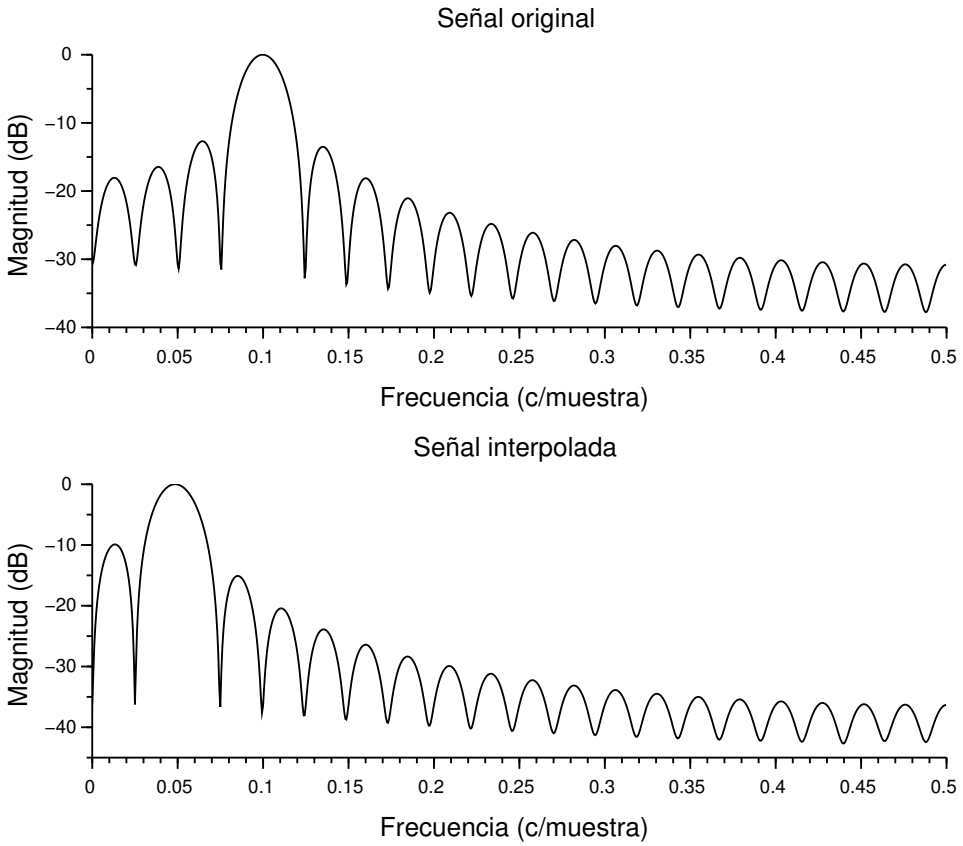


Figura 7.4: FFT del resultado de la prueba de interpolación.

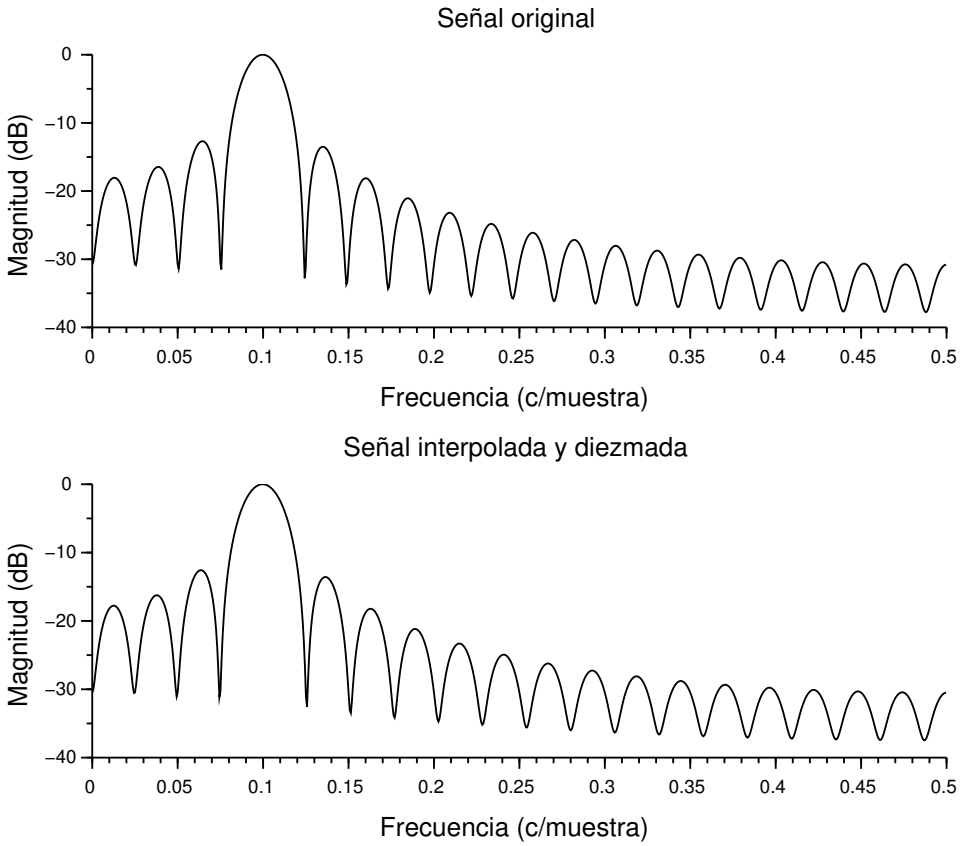


Figura 7.5: FFT del resultado de la prueba de interpolación y diezmado.

```
#define _USE_MATH_DEFINES
#include <math.h>
class Oscillator
{
public:
        Oscillator();
        float tick();
        void setFrequency(float frq);
        void setPhase(float pha);
        void setDepth(float depth);
        void setSampleRate(int srate);
private:
        float mFrq;
        float phstep;
        float mPha;
        float mDepth;
        int mSrate;
        bool cDepth;
        float nDepth;
};
```

Listado 7.17: Cabecera de la clase *Oscillator*.

Durante el desarrollo del trabajo analicé el resultado de ambas funciones de prueba en el dominio de la frecuencia utilizando la función *fft* de Scilab y comparando los resultados con la señal original. El resultado de la interpolación puede verse en la figura 7.4. El pico del lóbulo principal de la FFT se desplaza a 0.05 c/muestra, representando exactamente la misma frecuencia que antes de la interpolación. En la figura 7.5 se muestra el resultado de interpolar y diezmar la señal. Como cabía esperar, la señal resultante es idéntica a la original, despreciando el error intrínseco de las operaciones matemáticas realizadas. El código Scilab utilizado para realizar el análisis se encuentra en la ruta *./Proyecto/Retardo Modulado/Diseño/Prueba de interpolación y diezmado.txt*.

7.7 Oscilador

El objetivo de esta clase es crear una señal sinusoidal que sirva para hacer oscilar el retardo modulado y que permita modificar sus características mientras es generada.

En el listado 7.17 se muestra la cabecera de la clase *Oscillator*. El miembro *mFrq* determina la frecuencia de oscilación. A partir de esta frecuencia se calcula el miembro *phstep*, que determina el incremento del miembro *mPha* al solicitarse una muestra al oscilador mediante la función *float tick()*. El miembro *mPha* se incrementa partiendo de 0 y la señal se genera calculando el seno de *mPha*.

El miembro *mDepth* permite modificar la amplitud de la señal, multiplicando por él el resultado del seno. Para evitar alteraciones bruscas en la señal el cambio de este miembro solo sucede realmente cuando *mPha* es igual a 0 o mayor que 2π. Hasta entonces, se guarda el nuevo valor en el miembro *nDepth* y se señala que es necesario realizar el cambio usando el miembro *cDepth*.

En el listado 7.18 se muestra el código de esta clase. La función *float tick()* genera una

```
#include "Oscillator.h"
Oscillator::Oscillator()
{
        mSrate = 44100;
        mFrq = 2;
        mPha = 0;
        mDepth = 1;
        cDepth = false;
}
void Oscillator::setFrequency(float frq) {
        mFrq = frq;
        phstep = (frq / mSrate) * 2 * M_PI;
}
void Oscillator::setSampleRate(int srate) {
        mSrate = srate;
        setFrequency(mFrq);
}
void Oscillator::setPhase(float pha) {
        mPha = pha;
}
void Oscillator::setDepth(float depth) {
        nDepth = depth;
        cDepth = true;
}
float Oscillator::tick() {
        float sample = sin(mPha);
        sample *= mDepth;
        mPha += phstep;
        if (cDepth == true && (mPha > 2 * M_PI || mPha == 0)) {
                mDepth = nDepth;
                cDepth = false;
        }
        while (mPha > 2 * M_PI)
                mPha -= 2 * M_PI;
        return sample;
}
```

Listado 7.18: Código de la clase *Oscillator*.

nueva muestra, incrementa *mPha* y de ser necesario y posible de acuerdo a las condiciones establecidas modifica el miembro *mDepth*.

7.7.1 Pruebas

En el listado 7.19 se muestra el código de prueba de esta clase. En la linea de las pruebas de interpolación y diezmado, se crea una instancia de la clase *Oscillator* y se genera con ella una señal sinusoidal de 10 Hz. El resultado se guarda en un archivo csv que puede examinarse en Scilab. En este punto considero que la prueba es trivial, así que no insistiré

```
TEST_CLASS(OscillatorTest)
        {
        public:

                TEST_METHOD(OscillatorLFOTest)
                {
                        Oscillator LFO;
                        LFO.setFrequency(10);
                        LFO.setSampleRate(44100);
                        float buffer[44101];
                        ofstream outputFile("LFOtest.csv");
                        for (int i = 0; i < 44100; i++) {
                                buffer[i] = LFO.tick();
                                string msg = std::to_string(buffer[i]) + ",";
                                outputFile << msg;
                        }
                        outputFile.close();
                }

        };
```

Listado 7.19: Prueba de la clase *Oscillator*.

```
#include "../CircularBuffer.h"
#include <memory>
class FractionalDelay
{
public:
        FractionalDelay(std::shared_ptr<CircularBuffer> buffer);
        void newSample();
        float interpolate(float fdelay);
        void setStart(int start);
private:
        int mStart;
        float mRes;
        static const float mDifC[31];
        static const int difsize;
        static const int difsizeH;
        std::shared_ptr<CircularBuffer> mBuffer;
        CircularBuffer stage_A;
        CircularBuffer stage_B;
        CircularBuffer stage_C;
        CircularBuffer stage_D;
        CircularBuffer stage_E;
        CircularBuffer stage_F;
};
```

Listado 7.20: Cabecera de la clase *FractionalDelay*.

en ella.

7.8 Retardo fraccionario

El objetivo de esta clase es implementar el retardo fraccionario diseñado en la sección 5.1.

En el listado 7.20 se muestra la cabecera de la clase *FractionalDelay*. El miembro estático *mDifC* almacena los coeficientes del diferenciador. Los miembros estáticos difsize y difsizeH almacenan el tamaño total del filtro (en este caso $N = 63$ y $n_0 = 31$).

A la luz de lo expuesto en la sección 5.2.1, existen 6 buffers circulares a los que se asignará el tamaño adecuado en el constructor de la clase. Estos buffers son nombrados con letras en orden alfabético: *stage_A*, *stage_B*, etc.

Existe además un puntero a un buffer circular externo, $mBuffer$, que se corresponde con el array *Voice* mencionado en capítulos anteriores. Hacer este buffer un elemento externo a la clase permite utilizarlo para otro tipo de procesamientos en paralelo en caso de que algún día se decidiera implementar tal cosa, o incluso ejecutar varios retardos modulados a la vez con diferentes parámetros usando el mismo buffer principal. El puntero al buffer es un puntero inteligente compartido. El objeto al que apuntan este tipo de punteros solo es eliminado cuando desaparecen todas las referencias a él.

El miembro mStart indica la posición en la que se encuentra el retardo fraccionario dentro del array *Voice* y por tanto la posición sobre la que realizará las operaciones. Revisar las figuras 5.5 y 4.8 en caso de que esto no resulte claro.

En el listado 7.21 se muestra el código de esta clase. La función *void newSample()* actualiza

```
#include "FractionalDelay.h"
const float FractionalDelay::mDifC[31] = { 0.0001011f,  -0.0002149f,   0.0003893f,
-0.0006429f,   0.0009978f,  -0.0014794f,   0.0021168f,  -0.0029428f,   0.0039944f,
-0.0053127f,   0.0069436f,  -0.008938f,   0.0113528f,  -0.0142518f,   0.0177076f,
-0.0218039f,  0.0266393f,  -0.0323332f,   0.0390337f,  -0.0469311f,   0.0562781f,
-0.0674231f,  0.0808649f,  -0.0973506f,   0.1180552f,  -0.1449429f,   0.1815612f,
-0.2350249f,  0.3219692f,  -0.4923599f,   0.9961606f };

const int FractionalDelay::difsize = 63;
const int FractionalDelay::difsizeH = 31;

FractionalDelay::FractionalDelay(std::shared_ptr<CircularBuffer> buffer) :
        stage_A(difsizeH * 6), stage_B(difsizeH * 5),
        stage_C(difsizeH * 4),stage_D(difsizeH * 3),stage_E(difsize),
        stage_F(difsize) {
        mStart = 0;
        mBuffer = buffer;
}

void FractionalDelay::newSample() {
        float sta = mBuffer->subBackwardsAndProduct(mDifC,mStart, difsize) * -1;
        stage_A.push(sta);
        float stb = stage_A.subBackwardsAndProduct(mDifC, 0, difsize) * -0.5f;
        stage_B.push(stb);
        float stc = stage_B.subBackwardsAndProduct(mDifC, 0, difsize) * -0.3333333f;
        stage_C.push(stc);
        float std = stage_C.subBackwardsAndProduct(mDifC, 0, difsize) * -0.25f;
        stage_D.push(std);
        float ste = stage_D.subBackwardsAndProduct(mDifC, 0, difsize) * -0.2f;
        stage_E.push(ste);
        float stf = stage_E.subBackwardsAndProduct(mDifC, 0, difsize) * -0.1666666f;
        stage_F.push(stf);
        mRes = stage_F.subBackwardsAndProduct(mDifC, 0, difsize) * -0.1428571f;
}

float FractionalDelay::interpolate(float fdelay) {

        float aux = mRes * fdelay;
        aux += stage_F.get(difsizeH-1); aux *= fdelay;
        aux += stage_E.get(difsizeH*2-1); aux *= fdelay;
        aux += stage_D.peakLast(); aux *= fdelay;
        aux += stage_C.peakLast(); aux *= fdelay;
        aux += stage_B.peakLast(); aux *= fdelay;
        aux += stage_A.peakLast(); aux *= fdelay;
        aux += mBuffer->get(mStart + difsizeH * 7 - 1);
        return aux;
}

void FractionalDelay::setStart(int start) {
        mStart = start;
}
```

Listado 7.21: Código de la clase *FractionalDelay*.

```
#pragma once
#include "Oscillator.h"
#include "FractionalDelay.h"
#include "../CircularBuffer.h"
#include <memory>

class ModulatedDelay
{
public:
        ModulatedDelay();
        float newSample();
        void setFrequency(float frq);
        void setDepth(float depth);
        void setSampleRate(int srate);
        void initializeFractionalDelay(std::shared_ptr<CircularBuffer> buffer);
        void freeFractionalDelay();
private:
        Oscillator LFO;
        std::unique_ptr<FractionalDelay> fd;
        int oldi;
        int CHORUS_WIDTH;
        int NOMINAL_DELAY;
};
```

Listado 7.22: Cabecera de la clase *ModulatedDelay*.

los valores en los buffers circulares realizando las operaciones mostradas en la estructura 4.8. La función *float interpolate(float fdelay)* toma como parámetro un valor entre -0.5 y 0.5, llamado *p* en la figura 4.8 y *fdelay* en el código, y calcula la aproximación de la muestra fraccionaria.

7.9 Retardo modulado

El objetivo de esta clase es implementar el retardo modulado siguiendo el modelo expuesto en la sección 5.2.

En el listado 7.22 se muestra la cabecera de la clase *ModulatedDelay*. La clase tiene un miembro llamado LFO que es el encargado de generar la señal de baja frecuencia que hará oscilar el retardo fraccionario. También tiene dos miembros de tipo int, *CHORUS_WIDTH* y *NOMINAL_DELAY*, cuyo propósito fue explicado en la sección 5.2.

Dado que la clase *FractionalDelay* almacena varios buffers circulares cuya memoria puede ser liberada durante la ejecución cuando no se estén procesando muestras he implementado funciones para eliminar y crear una instancia de esta clase cuando sea oportuno usando punteros inteligentes.

El miembro *oldi* almacena la posición del retardo fraccionario en el array *Voice* y por tanto la posición sobre la que realizará las operaciones. Revisar las figuras 5.5 y 4.8 en caso de que esto no resulte claro.

En el listado 7.23 se muestra el código de esta clase. En las primeras lineas de la función *initializeFractionalDelay*, que será necesario llamar antes de realizar cualquier otra acción, se calcula la posición inicial del retardo fraccionario. Esta coincide con la amplitud máxima

```
#include "ModulatedDelay.h"

ModulatedDelay::ModulatedDelay() :
        LFO()
{
}

void ModulatedDelay::initializeFractionalDelay(std::shared_ptr<CircularBuffer> buffer) {
        CHORUS_WIDTH = floor((buffer->size() - 217) / 2); //294 at 44100 Hz sample rate
        NOMINAL_DELAY = CHORUS_WIDTH;
        fd = std::make_unique <FractionalDelay>(buffer);
        fd->setStart(NOMINAL_DELAY);
        LFO.setPhase(0);
        oldi = NOMINAL_DELAY;
}

void ModulatedDelay::freeFractionalDelay() {
        if(fd)
                fd.reset();
}

void ModulatedDelay::setFrequency(float frq) {
        LFO.setFrequency(frq);
}

void ModulatedDelay::setSampleRate(int srate) {
        LFO.setSampleRate(srate);
}

void ModulatedDelay::setDepth(float depth) {
        LFO.setDepth(depth);
}

float ModulatedDelay::newSample() {
        float displacement = LFO.tick() * CHORUS_WIDTH;
        float i_frac = NOMINAL_DELAY + displacement;
        int i = roundf(i_frac);
        float frac = i_frac - i;
        if (oldi < i) {
                oldi++;
                fd->setStart(oldi);
        }
        else {
                fd->setStart(oldi);
                fd->newSample();
                while (oldi > i) {
                        oldi--;
                        fd->setStart(oldi);
                        fd->newSample();
                }
        }
        return fd->interpolate(frac);
}
```

Listado 7.23: Código de la clase *ModulatedDelay*.

de oscilación, *CHORUS_WIDTH*, y con el centro de oscilación, *NOMINAL_DELAY*. En caso de que esto no resulte claro, consultar las figuras 5.4 y 5.5.

Como puede verse, es necesario pasar como parámetro a la función un puntero a un buffer circular. Este puntero será pasado a su vez al constructor del retardo fraccionario, que lo usará como array *Voice*.

Nótese que el puntero *fd* es un puntero único, ya que no habrá ningún otro puntero apuntando al objeto. En cuanto este puntero desaparezca, desaparecerá la instancia a la que apunta.

Las funciones *set* tan solo encapsulan llamadas a las funciones del miembro LFO, encargado de generar la señal sinusoidal de modulación.

La función *freeFractionalDelay* sirve para liberar la memoria ocupada por el retardo fraccionario, de ser necesario. Lo único que hace es llamar a la función *reset* del puntero inteligente, lo cual elimina el objeto interno. Obviamente habrá que llamar otra vez a la función *initializeFractionalDelay* antes de reanudar la ejecución.

La función *float newSample()* deberá ser llamada cada vez que se introduzca una muestra en el buffer circular externo y funciona como se describe a continuación: Primero se toma una muestra del oscilador. El valor de esta muestra oscilará con amplitud y frecuencia definidas por sus parámeros internos. El valor obtenido se multiplicará por *CHORUS_WIDTH*, la amplitud máxima de oscilación, dando como resultado una desplazamiento en el array *Voice* respecto al centro de oscilación. Una vez calculado este desplazamiento se calculará *i_frac* sumándolo a *NOMINAL_DELAY*, el centro de oscilación. Redondeando *i_frac* al entero más cercano se obtendrá la nueva posición del retardo fraccionario, llamada i en el código. El parámetro p en el rango $[-0.5, 0.5]$ se obtendrá restando de *i_frac* este entero. Este parámetro p es llamado $frac$ en el código mostrado.

Si la posición anterior del retardo fraccionario, $oldi$, es menor que i, a la luz de lo expuesto en la sección 5.2.1 solo es necesario emplazar el retardo fraccionario en la nueva posición.

Si por el contrario $oldi$ es mayor que i, lo primero que debemos hacer es señalar al retardo fraccionario que una nueva muestra ha entrado en el buffer circular principal. A continuación, vamos desplazando el retardo fraccionario una posición cada vez e indicando que actualice los buffers internos en cada movimiento. Dadas las condiciones impuestas sobre el rango de variación de i_frac el desplazamiento será como máximo de dos muestras, por lo que el coste total por muestra de entrada es menor de lo que pudiera parecer.

7.10 El procesador

Entro ahora en las clases propias de JUCE, en las que se harán las modificaciones necesarias para el funcionamiento del plugin.

En el listado 7.24 se muestra la cabecera de la clase *RetardoModuladoV2AudioProcessor*. Como puede verse, esta clase deriva de la clase *AudioProcessor* y a la vez de la clase *AudioProcessorValueTreeState::Listener*. Esta última clase contiene la función virtual *parameterChanged*, que será llamada por todas las instancias de la clase *AudioProcessorValueTreeState* asociadas cada vez que un parámetro cambie su valor.

Las funciones *void prepareToPlay* y *void releaseResources* fueron descritas en el capítulo anterior por lo que no insistiré en ellas. Nótese que la primera proporciona en cada llamada la frecuencia de muestreo actual, así como el tamaño configurado del bloque a procesar.

La función *createEditor* devuelve un editor para el procesador, usado como interfaz gráfica para interactuar con el plugin.

La función más importante es *void processBlock*. Dentro de ella se realizará todo el procesamiento cada vez que el DAW requiera procesar un bloque de muestras.

En cuanto a los miembros de la clase, *parameters* es una instancia de *AudioProcessorValueTreeState* en la que se almacenan los parámetros y el estado del procesador. El miembro *cb* es un puntero al buffer circular usado para el retardo fraccionario. Una vez más, nótese que es un puntero inteligente compartido, lo que implica que varios punteros pueden apuntar a la misma instancia. El miembro *md* es una instancia de la clase *ModulatedDelay* implementada anteriormente. Se incluyen además punteros a las clases de interpolación y diezmado. El hecho de usar punteros resultará claro posteriormente, pero se debe fundamentalmente a que de este modo puede hacerse un uso más eficiente de la memoria cuando el plugin no está activo.

La utilidad de los restantes miembros quedará clara cuando se explique la implementación. Por economía de espacio solo se mostrarán las secciones de código que han sido modificadas. En cualquier caso, el código completo puede encontrarse en la carpeta correspondiente adjunta a esta memoria.

En el listado 7.25 se muestra el constructor de esta clase. En la lista de inicialización se le pasa al miembro *parameters* una referencia al procesador asociado, en este caso **this*, así como un manejador de vuelta atrás, en caso de que se quisiera añadir la funcionalidad de deshacer acciones y volver a un estado anterior. Si no se incluye esta funcionalidad se le pasa un puntero nulo.

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "./Utils/ModulatedDelay/ModulatedDelay.h"
#include "./Utils/Resampling/Interpolatorx2.h"
#include "./Utils/Resampling/Decimatorx2.h"
//==============================================================================
/**
*/
class RetardoModuladoV2AudioProcessor  : public AudioProcessor,
                         public AudioProcessorValueTreeState::Listener
{
public:
    //==============================================================================
    RetardoModuladoV2AudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioSampleBuffer&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    double getTailLengthSeconds() const override;

    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;

    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;

        virtual void parameterChanged(const String &parameterID, float newValue) override;

private:
    //==============================================================================
        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(RetardoModuladoV2AudioProcessor)
        AudioProcessorValueTreeState parameters;

        std::shared_ptr<CircularBuffer> cb;
        ModulatedDelay md;
        std::unique_ptr<Interpolatorx2> ix;
        std::unique_ptr<Decimatorx2> dx;

        double currentSampleRate;
        float previousGain;

        bool interpolatex2;
};
```

Listado 7.24: Cabecera de la clase *RetardoModuladoV2AudioProcessor*.

```
#include "PluginProcessor.h"
#include "PluginEditor.h"
//==============================================================================
RetardoModuladoV2AudioProcessor::RetardoModuladoV2AudioProcessor():
#ifndef JucePlugin_PreferredChannelConfigurations
      AudioProcessor (BusesProperties()
                      #if ! JucePlugin_IsMidiEffect
                       #if ! JucePlugin_IsSynth
                        .withInput  ("Input",  AudioChannelSet::stereo(), true)
                       #endif
                        .withOutput ("Output", AudioChannelSet::stereo(), true)
                      #endif
                        ),
#endif
        md(), parameters(*this,nullptr)
{
        interpolatex2 = false;
        currentSampleRate = 0;

        parameters.createAndAddParameter("1",        // parameter ID
                "Modulation Frequency",        // parameter name
                String(" Hz"),      // parameter label (suffix)
                NormalisableRange<float>(0.0f, 10.0f, 0.1f),     // range
                4.0f,          // default value
                [](float value)
                {
                        // value to text function
                        return String(value);
                },
                [](const String& text)
                {
                        // text to value function
                        return text.getFloatValue();
                });
        parameters.addParameterListener("1", this);

        parameters.createAndAddParameter("2",        // parameter ID
                "Depth",        // parameter name
                String(" %"),      // parameter label (suffix)
                NormalisableRange<float>(0.0f, 100.0f, 1.0f),     // range
                50.0f,          // default value
                [](float value)
        {
                // value to text function
                return String(value);
        },
                [](const String& text)
        {
                // text to value function
                return text.getFloatValue();
        });
        parameters.addParameterListener("2", this);

        parameters.createAndAddParameter("3", "Interpolation Mode", String(),
                NormalisableRange<float>(0.0f, 1.0f, 1.0f), 0.0f,
                [](float value)
        {
                // value to text function
                return value < 0.5 ? "No Interpolation" : "x2 Interpolation";
        },
                [](const String& text)
        {
                // text to value function
                if (text == "No Interpolation")    return 0.0f;
                if (text == "x2 Interpolation")  return 1.0f;
                return 0.0f;
        });
        parameters.addParameterListener("3", this);

        parameters.createAndAddParameter("4",        // parameter ID
                "Original Signal",        // parameter name
                String(""),      // parameter label (suffix)
                NormalisableRange<float>(0.0f, 1.0f, 0.01f),     // range
                0.0f,          // default value
                [](float value)
        {
                // value to text function
                return String(value);
        },
                [](const String& text)
        {
                // text to value function
                return text.getFloatValue();
        });

        parameters.state = ValueTree(Identifier("RM2ValueTree"));
        md.setFrequency(4.0f);
        md.setDepth(0.5f);
}
```

Listado 7.25: Constructor de la clase *:RetardoModuladoV2AudioProcessor*.

```
1
2  void RetardoModuladoV2AudioProcessor::parameterChanged(const String &parameterID, float newValue) {
3          switch (parameterID.getTrailingIntValue()) {
4          case 1:
5                  md.setFrequency(newValue);
6                  break;
7          case 2:
8                  md.setDepth(newValue/100);
9                  break;
10         case 3:
11                 if (newValue == 0.0f) {
12                         interpolatex2 = false;
13                         md.setSampleRate(currentSampleRate);
14                 }
15                 else {
16                         interpolatex2 = true;
17                         md.setSampleRate(currentSampleRate * 2);
18                 }
19                 break;
20         }
21 }
```

Listado 7.26: Función *parameterChanged*.

El booleano *interpolatex2* sirve para indicar si debe realizarse el proceso de interpolación y diezmado y podrá ser modificado por el usuario mediante un parámetro.

El proceso para añadir a la instancia de la clase *AudioProcessorValueTreeState* cada uno de los parámetros necesarios se repite para cada uno de ellos y está comentado en el código. Consiste en proporcionar un identificador, un nombre, un sufijo,un rango, un valor por defecto, una función que permita convertir el valor del parámetro en texto y una función que permita convertir texto en un valor del parámetro. Finalmente, se añade al parámetro un escuchador asociándolo a su identificador, en este caso la propia instancia que está siendo creada, mediante la función *addParameterListener*. Nótese que las funciones de conversión de texto a float y viceversa pueden ser pasadas en forma de expresiones lambda de C++11 [43]. Nótese también que se ha añadido un parámetro *Original Signal* que será usado para mezclar la señal original con la resultante del procesamiento con una ganancia definida por el usuario.

Tras añadir los parámetros se asigna un identificador a la instancia de la clase *AudioProcessorValueTreeState*, en este caso *RM2ValueTree*, y se configuran los parámetros del retardo modulado con el valor inicial elegido.

En el listado 7.26 se muestra la función llamada cuando se produce un cambio en el valor de un parámetro. Como puede verse, el proceso consiste simplemente en seleccionar una acción de acuerdo al id del parámetro y llamar a la función correspondiente. Si el parámetro que ha cambiado es el que permite seleccionar la opción de interpolación es necesario modificar la frecuencia de muestreo del retardo modulado. Este es el propósito del miembro *currentSampleRate*, que almacenará la frecuencia de muestreo utilizada por el DAW. La frecuencia de muestreo utilizada por el retardo modulado será el doble solo en el caso de que el usuario haya decidido realizar la interpolación antes del procesamiento.

```
1  void RetardoModuladoV2AudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
2  {
3      // Use this method as the place to do any pre-playback
4      // initialisation that you need..
5          currentSampleRate = sampleRate;
6          double LFOSampleRate = sampleRate;
7          if (interpolatex2) LFOSampleRate *= 2;
8          if (!cb) {
9                  cb = std::make_shared<CircularBuffer>(805*floor(sampleRate/44100));
10                 md.initializeFractionalDelay(cb);
11                 md.setSampleRate(LFOSampleRate);
12         }
13         if (!ix) {
14                 ix = std::make_unique<Interpolatorx2>();
15                 dx = std::make_unique<Decimatorx2>();
16         }
17 }
18
19 void RetardoModuladoV2AudioProcessor::releaseResources()
20 {
21         // When playback stops, you can use this as an opportunity to free up any
22         // spare memory, etc.
23         md.freeFractionalDelay();
24         if (cb) {
25                 cb.reset();
26         }
27         if (ix) {
28                 ix.reset();
29                 dx.reset();
30         }
31 }
```

Listado 7.27: Funciones *prepareToPlay* y *releaseResources*.

En el listado 7.27 se muestran las funciones *prepareToPlay* y *releaseResources*. Lo primero que se hace cuando el DAW va a empezar la reproducción es actualizar el valor del miembro *currentSampleRate* con el valor proporcionado por el DAW de la tasa de muestreo actual. Si se está realizando el proceso de interpolación la tasa de muestreo del oscilador de baja frecuencia deberá ser el doble de la tasa de muestreo utilizada.

Si el buffer no existe es necesario crearlo usando el número de muestras establecido en capítulos anteriores. Se añade adicionalmente un multiplicador que permite ajustar proporcionalmente el tamaño del buffer si aumenta la tasa de muestreo. Con este buffer creado y la tasa de muestreo calculada es posible inicializar el retardo modulado. También es necesario crear el interpolador y el diezmador, en caso de que estos no existan.

Cuando el DAW termina el procesamiento es posible realizar el proceso inverso, es decir, liberar parte de la memoria usada por el retardo modulado y eliminar tanto el buffer como las instancias dedicadas a interpolación y diezmado, si estas existen.

Lo cierto es que he comprobado de forma práctica que el comportamiento de los DAW es a veces caótico al llamar a estas dos funciones (por ejemplo, la función *releaseResources* es a veces llamada dos veces seguidas). Por tanto, no es una mala práctica añadir medidas de seguridad adicionales en todas las operaciones que se realice dentro de ellos.

En el listado 7.28 se muestra la función *processBlock*. En primer lugar se limpian las muestras del canal derecho, ya que no serán procesadas en esta versión. Posteriormente, se toma el valor actual del parámetro de ganancia de la señal original. Para evitar cambios

```
1  void RetardoModuladoV2AudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
2  {
3          //Clean right channel
4          float* channelData = buffer.getWritePointer(1);
5          for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
6          {
7                  channelData[sample] = 0;
8          }
9
10         //Process left channel
11         const float currentGain = *parameters.getRawParameterValue("4");
12         float gainDelta = (currentGain - previousGain) / buffer.getNumSamples();
13         channelData = buffer.getWritePointer(0);
14         for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
15         {
16                 if (interpolatex2) {
17                         ix->inputValue(channelData[sample]);
18                         cb->push(ix->getNext());
19                         dx->inputValue(md.newSample());
20                         cb->push(ix->getNext());
21                         dx->inputValue(md.newSample());
22                         channelData[sample] = dx->getNext() + previousGain * channelData[sample];
23                         previousGain += gainDelta;
24                 }
25                 else {
26                         cb->push(channelData[sample]);
27                         channelData[sample] = md.newSample() + previousGain * channelData[sample];
28                         previousGain += gainDelta;
29                 }
30         }
31 }
32
33 AudioProcessorEditor* RetardoModuladoV2AudioProcessor::createEditor()
34 {
35         return new RetardoModuladoV2AudioProcessorEditor(*this, parameters);
36 }
```

Listado 7.28: Función *processBlock*.

bruscos de volumen se establece una variación de ganancia por muestra repartiendo la diferencia total de ganancia entre el número total de muestras del buffer.

En caso de que el proceso de interpolación esté seleccionado se introduce una muestra en el interpolador y se sacan dos muestras que son introducidas en el buffer, señalando al retardo modulado cada vez que una nueva muestra se ha añadido al buffer circular e introduciendo los resultados devueltos en el diezmador. Finalmente, se toma una sola muestra del diezmador y se añade a este resultado el producto de la señal original y la ganancia actual. En caso de que no sea necesario realizar el proceso de interpolación el procesamiento es trivial.

Nótese que las muestras a procesar están contenidas en una instancia de la clase *AudioSampleBuffer* y que para obtener un puntero de escritura se llama a la función *getWritePointer* pasando como argumento el número de canal. En este caso solo se procesa el canal izquierdo.

En el mismo listado también se muestra la implementación de la función *createEditor*. La única modificación realizada es que se pasa al constructor del editor una referencia a la instancia de la clase *AudioProcessorValueTreeState* asociada con el procesador para que una vez en el editor puedan asociarse los parámetros a los controles gráficos.

```
#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"
//==============================================================================
class RetardoModuladoV2AudioProcessorEditor  : public AudioProcessorEditor
{
public:
    RetardoModuladoV2AudioProcessorEditor (RetardoModuladoV2AudioProcessor& p, AudioProcessorValueTreeState
        & vts);
    ~RetardoModuladoV2AudioProcessorEditor();

    //==============================================================================
    void paint (Graphics&) override;
    void resized() override;
private:
        typedef AudioProcessorValueTreeState::SliderAttachment SliderAttachment;
        typedef AudioProcessorValueTreeState::ButtonAttachment ButtonAttachment;
        AudioProcessorValueTreeState& valueTreeState;
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    RetardoModuladoV2AudioProcessor& processor;

        Slider frqModSlider;
        ScopedPointer<SliderAttachment> frqModAttachment;
        Slider depthSlider;
        ScopedPointer<SliderAttachment> depthAttachment;
        Slider oSignalSlider;
        ScopedPointer<SliderAttachment> oSignalAttachment;

        ToggleButton x2InterpolationButton;
        ScopedPointer<ButtonAttachment> x2InterpolationAttachment;

        Label frqModLabel;
        Label depthLabel;
        Label oSignalLabel;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RetardoModuladoV2AudioProcessorEditor)
};
```

Listado 7.29: Cabecera de la clase *RetardoModuladoV2AudioProcessorEditor*.

7.11 El editor

El objetivo de esta clase es aportar una interfaz gráfica mediante la cual el usuario pueda manejar los parámetros del plugin.

En el listado 7.29 se muestra la cabecera. Las clases *Slider*, *ToggleButton* y *Label* son controles gráficos que aparecerán en el panel del plugin y se crea uno por cada parámetro. Para mantener estos controles vinculados a los parámetros contenidos en la instancia de la clase *AudioProcessorValueTreeState* que almacena el estado del procesador se utilizan las clases *SliderAttachment* y *ButtonAttachment*.

En el listado 7.30 se muestra el constructor de esta clase. Para cada control gráfico se hacen una serie de ajustes extraídos directamente del tutorial de JUCE para la clase *Slider* [44]. La parte más importante es la creación de las instancias de *SliderAttachment*, a las que se pasa en el constructor el id del parámetro con el que se quiere vincular el *Slider*. El *Slider* quedará vinculado al parámetro durante todo el tiempo de vida de la instancia de *SliderAttachment* creada.

En el listado 7.31 se muestran dos funciones de esta clase que son usadas para dar un estilo a la ventana y posicionar los elementos. Su funcionamiento es trivial, pero en caso de duda puede consultarse la documentación de JUCE.

```
RetardoModuladoV2AudioProcessorEditor::RetardoModuladoV2AudioProcessorEditor (
      RetardoModuladoV2AudioProcessor& p, AudioProcessorValueTreeState& vts)
    : AudioProcessorEditor (&p), processor (p),valueTreeState(vts)
{
    // Make sure that before the constructor has finished, you've set the
    // editor's size to whatever you need it to be.
    setSize (400, 300);

        frqModSlider.setSliderStyle(Slider::Rotary);
        frqModSlider.setTextBoxStyle(Slider::NoTextBox, false, 90, 0);
        frqModSlider.setPopupDisplayEnabled(true, this);
        frqModSlider.setTextValueSuffix(" Hz");
        addAndMakeVisible(frqModSlider);
        frqModLabel.setText("Modulation Frequency", dontSendNotification);
        frqModLabel.attachToComponent(&frqModSlider, true);
        frqModAttachment = new SliderAttachment(valueTreeState, "1", frqModSlider);

        depthSlider.setSliderStyle(Slider::Rotary);
        depthSlider.setTextBoxStyle(Slider::NoTextBox, false, 90, 0);
        depthSlider.setPopupDisplayEnabled(true, this);
        depthSlider.setTextValueSuffix(" %");
        addAndMakeVisible(depthSlider);
        depthLabel.setText("Depth", dontSendNotification);
        depthLabel.attachToComponent(&depthSlider, true);
        depthAttachment = new SliderAttachment(valueTreeState, "2", depthSlider);

        x2InterpolationButton.setButtonText("x2 Interpolation");
        //x2InterpolationButton.setToggleState(valueTreeState.getParameter("3")->getValue() >0.0f, dontSendNotification);
        addAndMakeVisible(x2InterpolationButton);
        x2InterpolationAttachment = new ButtonAttachment(valueTreeState, "3", x2InterpolationButton);

        oSignalSlider.setSliderStyle(Slider::Rotary);
        oSignalSlider.setTextBoxStyle(Slider::NoTextBox, false, 90, 0);
        oSignalSlider.setPopupDisplayEnabled(true, this);
        oSignalSlider.setTextValueSuffix("");
        addAndMakeVisible(oSignalSlider);
        oSignalLabel.setText("Original Signal", dontSendNotification);
        oSignalLabel.attachToComponent(&oSignalSlider, true);
        oSignalAttachment = new SliderAttachment(valueTreeState, "4", oSignalSlider);

}
```

Listado 7.30: Constructor de la clase *RetardoModuladoV2AudioProcessorEditor*.

```
void RetardoModuladoV2AudioProcessorEditor::paint (Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour (ResizableWindow::backgroundColourId));
    g.setColour (Colours::white);
    g.setFont (15.0f);
}

void RetardoModuladoV2AudioProcessorEditor::resized()
{
        const int sliderLeft = 120;
        frqModSlider.setBounds(sliderLeft, 20, 70, 70);
        depthSlider.setBounds(sliderLeft, 120, 70,70);
        oSignalSlider.setBounds(sliderLeft, 220, 70, 70);
        x2InterpolationButton.setBounds(210, 45, 150, 100);
}
```

Listado 7.31: Funciones *paint* y *resized* de la clase *RetardoModuladoV2AudioProcessorEditor*.

7.12 Conclusiones

Siguiendo los modelos creados en capítulos anteriores se ha implementado en este capítulo un plugin de audio para aplicar un retardo modulado a una señal. Se han implementado además clases para aplicar un proceso de interpolación y diezmado a una señal. Estas clases permiten multiplicar por dos la frecuencia de muestreo para realizar el procesamiento sin producir aliasing.

CAPÍTULO 8

Pruebas del plugin de audio para el retardo modulado

Este capítulo pertenece al proceso de pruebas del plugin de audio que implementa el efecto de retardo modulado diseñado. En él se comprueba de forma práctica el funcionamiento de la implementación realizada en el capítulo anterior, teniendo en cuenta tanto el desempeño individual como en comparación con otros plugins similares. El dll resultante de la compilación del código del capítulo anterior puede encontrarse en la ruta *./Proyecto/Retardo Modulado/DLL*, junto con instrucciones para su instalación.

8.1 Pruebas

Para realizar las pruebas, en primer lugar ubiqué el archivo dll resultante de compilar el código del capítulo anterior en la carpeta donde Reaper almacena los plugins de audio. Reaper escanea los nuevos plugins al iniciarse, por lo que estos estarán disponibles para introducirse como inserción en una pista tras el inicio del programa sin necesidad de realizar ninguna otra acción. En la figura 8.1 se muestra el panel del plugin tras ser añadido a las inserciones de una pista.

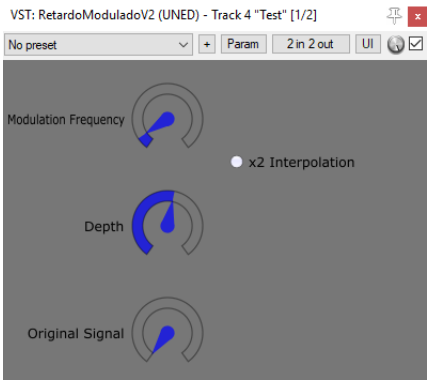


Figura 8.1: Panel del plugin de audio.

La carpeta en la que Reaper busca los plugins es configurable e incluso se puede crear una lista con varias carpetas. Para ello, basta con seguir la ruta Options->Preferences en el menú superior y seleccionar VST en la lista de la derecha. En la parte superior de la ventana aparecerá un campo llamado *VST plug-in paths* en el que se pueden introducir las rutas de las carpetas.

Para añadir un plugin como inserción basta con hacer clic en la lista de inserciones, marcada en la figura 6.2 con el número 2, y seleccionar el plugin correspondiente.

Tras comprobar el correcto funcionamiento de los sliders establecí un entorno de prueba como se describe a continuación. En primer lugar añadí en una pista el plugin GSinth2, disponible de forma gratuita en la página `https://www.gvst.co.uk/gsinth2.htm`. Este plugin es un sintetizador virtual que genera ondas sinusoidales. A continuación hice que la pista de la inserción contuviera una sola nota MIDI, en concreto la nota C6 (Do de la sexta octava), correspondiente con una frecuencia de 1046.502 Hz. Posteriormente, envié el sonido de esta pista a tres pistas diferentes y en cada una de ellas introduje un plugin distinto. En una de las pistas añadí como inserción el plugin creado en este trabajo, mientras que en las otras dos añadí diferentes plugins de vibrato con iguales parámetros de funcionamiento. Con esta configuración la onda sinusoidal pasa inalterada a cada una de las tres pistas, permitiendo comparar los resultados del procesamiento. Para visualizar los resultados en el dominio de la frecuencia usé el plugin *Frequency Spectrum Analyzer Meter*, que se instala por defecto con Reaper. Este plugin realiza una FFT de la señal. En este caso, lo configuré para utilizar una ventana de Blackman-Harris de 8192 puntos.

En la figura 8.2 se muestra el resultado de la comparación. El resultado del plugin implementado se muestra en el último gráfico. Obviamente, lo deseable es que la señal generada sea una modulación de las frecuencias de entrada y no presente artefactos adicionales.

Como puede verse, el primero de los plugins presenta una elevada distorsión en la que predominan armónicos de orden impar. El segundo y el tercero están más igualados en cuanto a distorsión, pero he comprobado de forma práctica que a frecuencias más altas de oscilación y/o frecuencias más altas de la señal de entrada el plugin implementado se comporta mejor en cuanto a distorsión.

El funcionamiento del plugin es el esperado, el pico presente en la frecuencia de entrada oscila de izquierda a derecha y es posible variar adecuadamente tanto su amplitud como su frecuencia de oscilación usando los sliders, sin errores ni ruidos.

He llevado a cabo pruebas adicionales con archivos de audio y la diferencia entre interpolar o no interpolar las muestras antes del procesamiento no es apreciable para mi oído, lo cual me hace pensar que el aliasing es mínimo y solo se produce en las frecuencias más altas

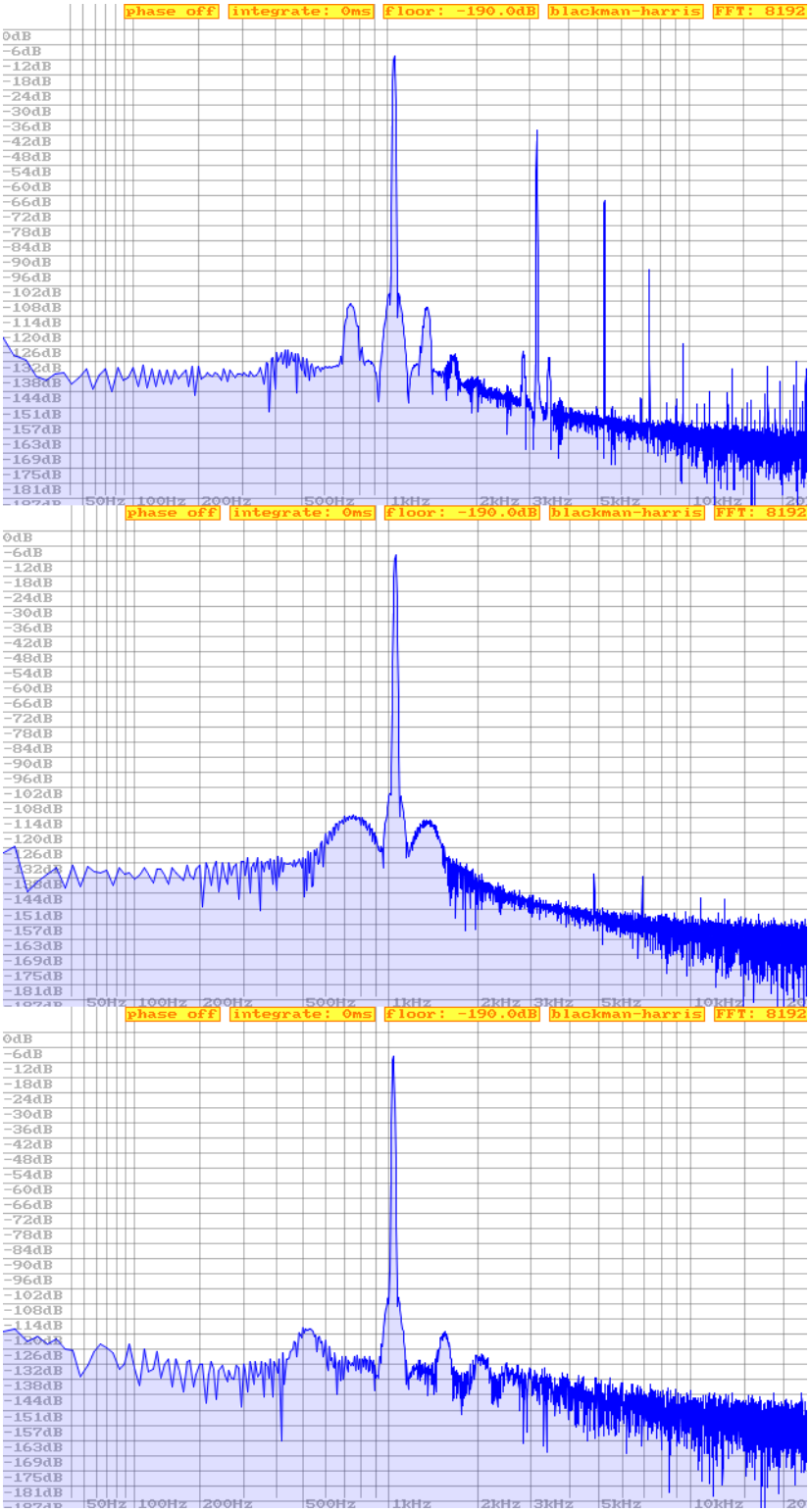


Figura 8.2: Respuesta de diferentes plugins de vibrato a una señal de 1046.502 Hz. La respuesta del plugin implementado se muestra en el último gráfico.

del rango audible.

Es posible obtener resultados interesantes, aunque la configuración de los parámetros depende del tipo de señal de entrada y del resultado que se quiera conseguir. Para simular el vibrato natural de un instrumento de cuerda es mejor usar frecuencias de oscilación intermedias y amplitudes muy cortas, mientras que para efectos creativos con sintetizadores, por ejemplo, aumentar la amplitud de oscilación produce resultados interesantes.

8.2 Conclusiones

El plugin implementado modula las frecuencias de entrada con gran exactitud, si bien en la gran mayoría de ocasiones en las que es deseable aplicar este tipo de efecto a un sonido es posible que no esté justificado el gasto de procesamiento. En futuros trabajos sería interesante usar el mismo método para implementar otros efectos que requieran un retardo o un retardo modulado y evaluar los resultados.

CAPÍTULO 9

Introducción al OMAP-L138

En este capítulo se hace una revisión del SoC integrado en la placa de desarrollo y sus módulos antes de pasar a la implementación final del plugin. En este trabajo se usa tan solo el núcleo C674x, aunque con vistas a incorporar posteriormente el procesador ARM9 incluido en el SoC. Es de gran utilidad el esquema de la placa de desarrollo disponible en [45] para saber como están conectados los elementos entre ellos. La documentación completa está disponible en la sección de referencias. Este capítulo en conjunto con consultas a las figuras de la documentación otorga una visión general de la estructura, funcionamiento y configuración del SoC necesaria para entender el código fuente de capítulos posteriores.

9.1 Estructura del OMAP-L138

El OMAP-L138 es un SoC de doble nucleo que integra un procesador ARM9 y un nucleo C674x. Contiene además un sistema PRU (Programmable Realtime Unit) de dos nucleos especializado en tareas en tiempo real que no será usado en este proyecto. El nucleo C674x contiene dos memorias internas de 32 kB, llamadas L1P y L1D, y una memoria de 256 kB, llamada L2. El uso que se hace de estas memorias es configurable, aunque normalmente las memorias L1P y L1D se usan como caché. La memoria L2 puede incluso ser dividida en dos bloques para crear regiones independientes con diferente propósito. Mas detalles sobre la configuración de la caché del C674x pueden encontrarse en el documento de referencia [46]. En este trabajo se utilizará la configuración por defecto, es decir, las memorias L1P y L1D se usarán como caché y la memoria L2 será de uso libre. Por otro lado, ambos procesadores tienen acceso a una región de memoria compartida dentro del chip de 128 kB que suele ser usada para comunicaciones inter-procesador. Un diagrama de bloques completo del SoC puede encontrarse en la hoja de datos [47].

La configuración de los diferentes módulos se realiza mediante la escritura de registros cuyas direcciones forman parte del mapa de memoria. Las direcciones concretas de los

registros están disponibles en la hoja de datos del SoC [47].

Algunos registros no son accesibles mediante direcciones del mapa de memoria sino que se modifican mediante una instrucción especial de lenguaje ensamblador, la instrucción MVC. Si se está programando en C incluir el archivo *c6x.h* proporciona una definición de estos registros para poder escribir datos en ellos tal y como se haría con una variable normal de tipo int.

9.2 Estructura del núcleo C674x

A grandes rasgos, el nucleo consiste en 8 unidades funcionales llamadas .L1, .S1, .M1, .D1, .L2, .S2, .M2 y .D2. Aunque cada una de ellas puede realizar operaciones de varios tipos, las unidades D están especializadas en operaciones de carga y almacenamiento en memoria; las unidades M en multiplicaciones en coma fija o flotante; las unidades L en operaciones lógicas y de desplazamiento, y las unidades S en sumas y restas. El núcleo tiene además 64 registros de 32 bits divididos en dos bancos de 32 registros cada uno. Las unidades tienen acceso directo solo a uno de los bancos de registros. Por ejemplo, .S1 solo tiene acceso directo al primer banco registros mientras que .S2 solo tiene acceso directo al segundo. Sin embargo, el procesador tiene un mecanismo llamado *crosspath* que permite el acceso de una unidad funcional al otro banco de registros con una salvedad: El registro al que se accede no puede haber sido escrito en el ciclo anterior. Si ha sido escrito en el ciclo anterior, el procesador automáticamente se detiene durante un ciclo antes de ejecutar la instrucción.

Para entender mejor el funcionamiento del procesador es una buena idea tomar fragmentos de código en ensamblador. Una guía completa de instrucciones en ensamblador para este procesador esta disponible en la documentación de referencia [48]. Las instrucciones en ensamblador se agrupan en bloques de hasta 8 instrucciones y cada una de ellas tiene asignada en el propio código una unidad funcional. Además, es posible añadir una guarda en cualquier instrucción indicando que solo se ejecute si el valor del registro indicado en la guarda es distinto de 0.

El encauzamiento de instrucciones se divide en tres etapas: Busqueda o fetch; dividida a su vez en PG, PS, PW y PR; decodificación, dividida a su vez en DP y DC; y ejecución, dividida a su vez en un número de subetapas que depende del tipo de operación: E1,E2,E3, etc. Mientras que las etapas de búsqueda y decodificación tienen una duración fija, la duración de la etapa de ejecución dependerá del tipo de operación a realizar. Por ejemplo, una multiplicación en coma flotante tiene una latencia de tres ciclos y por lo tanto tres subetapas de ejecución: E1, E2 y E3.

Hay que tener en cuenta que aunque es posible tomar bloques de 8 instrucciones en paralelo, las etapas de búsqueda y decodificación no pueden solaparse. La etapa de ejecución puede solaparse si la instrucción se ejecuta en otra unidad funcional o si se ejecuta en la misma unidad funcional pero las subetapas de ejecución no se solapan. Por ejemplo, una unidad M puede tomar los datos para realizar una multiplicación en coma flotante e inmediatamente tomar datos nuevos en el siguiente ciclo para realizar otra multiplicación, dado que la primera multiplicación ya se encontrará en la subetapa de ejecución E2 cuando la segunda se encuentre en la subetapa E1. La realidad es más complicada ya que las subetapas de ejecución no pueden solaparse siempre unas con otras, aún cuando no se trate de la misma subetapa. Si llamamos, por ejemplo, E1,E2,E3,E4 y E5 a las etapas de ejecución de una instrucción de 5 ciclos tal vez E2 pueda solaparse con E1 pero E4 no pueda solaparse con E2. Todas estas cuestiones se encuentran explicadas en la documentación de referencia [48].

Todo esto hace que encontrar una distribución óptima de las instrucciones que aproveche al máximo el paralelismo programando en ensamblador sea una tarea muy difícil de realizar de forma manual. Afortunadamente, el compilador incluido en el IDE de Texas Instruments (Code Composer Studio) hace extraordinariamente bien este trabajo de forma automatizada.

9.2.1 Desenrollamiento de bucles en el C674x

El C674x tiene un sistema de desenrollamiento de bucles por hardware que permite cargar un conjunto de instrucciones en un buffer y ejecutarlas repetidas veces desenrollando el bucle y aprovechando al máximo el paralelismo. Una explicación completa sobre el sistema de desenrollamiento de bucles puede encontrarse en [19], capítulo 7.

El compilador de Texas Instruments permite indicar mediante la directiva $MUST\_ITERATE$ el número de veces mínimo y máximo que se ejecutará un bucle. Esto permite traducir el código a ensamblador haciendo uso de instrucciones específicas del sistema de desenrollamiento de bucles y reducir así en muchos casos el tiempo de ejecución.

Hay que tener en cuenta que esto obviamente no funciona si existe una llamada a una función dentro del bucle, a no ser que la función sea *inline*, es decir, tenga su código insertado en la sección de código donde se encuentra la llamada.

9.3 Módulos

En esta sección describo los módulos más importantes usados en este trabajo y doy una visión general de sus características y opciones de configuración.

9.3.1 Módulo EDMA3

EDMA3 (Enhanced Direct Memory Access) es un protocolo de acceso directo a memoria que sirve para transferir directamente datos a la memoria desde dispositivos externos sin hacer un uso directo del procesador. Los detalles sobre su funcionamiento pueden encontrarse en la documentación del módulo [49].

El módulo se divide en dos bloques principales, EDMA3 CC (Channel Controller) y EDMA3 TC (Transfer Controller). El primero es el encargado de gestionar las transferencias mientras que el segundo las ejecuta y genera los eventos necesarios cuando estas se completan.

EDMA3 permite encadenar transferencias de forma que una transferencia directa a memoria con una configuración diferente pueda empezar automáticamente tras haber finalizado la anterior. Esto permite incluso enlazar diferentes configuraciones y hacer que la última transferencia de la cadena apunte a la primera, reiniciando el ciclo constantemente.

El EDMA3 CC sincroniza las transferencias mediante eventos recibidos de periféricos externos. Estos eventos son capturados en un registro llamado ER (Event Register). Si el evento tiene consecuencias o no dependerá del valor del registro EER (Event Enable Register) y del método de sincronización que se haya elegido.

Cada vez que una transferencia es completada el EDMA3 CC puede generar una interrupción. El EDMA CC tiene varios registros dedicados a configurar las interrupciones: El IPR (Interrupt Pending Register) recoge las solicitudes de interrupción. El IER (Interrupt Enable Register) indica que interrupciones están activadas. Por último, existen varios registros DRAEn (DMA Region Access Enable n) que permiten enmascarar las interrupciones para un determinado canal. Un diagrama lógico de las interrupciones con todos estos registros está disponible en la documentación de referencia [49], figura 12.

Las transferencias de EDMA3 se dividen en bloques llamados A, B y C. Un bloque A consiste en un número de bytes determinado. Un bloque B está formado por un número determinado de bloques A. Un bloque C esta formado por un número determinado de bloques B. Esta transferencia dividida en bloques permite diferentes tipos de sincronización que pueden consultarse en la sección 2.2 de la documentación [49]. En este trabajo se usan transferencias A-sincronizadas y todas las explicaciones dadas presuponen el uso de este

tipo de transferencia.

El EDMA3 CC tiene unas estructuras de datos llamadas PaRAM a traves de las cuales se realiza la mayor parte de la configuración. Entre otras cosas, se configura en ellas el tamaño de los bloques A, B y C. La configuración dada por un PaRAM se usa hasta finalizar la transferencia de todos los bloques. Los PaRAM se almacenan en registros accesibles mediante direcciones de memoria. Un PaRAM está organizado en 8 palabras de 32 bits divididas en diferentes campos. Estos campos son:

- **Channel options (OPT):** Ocupa los primeros 4 bytes. La función de cada uno de los bits puede encontrarse en el documento de referencia [49]. Cabe resaltar el bit TCCMODE, que indica en qué momento se considera una transferencia completa. Un 0 en este bit indica que la transferencia esta completa cuando los datos terminan de transmitirse, mientras que un 1 indica que la transferencia se considera completa una vez ha sido despachada al EDMA3 TC. El campo TCC indica el bit que se activará en el IPR cuando se complete la transferencia. Esto esta explicado en la sección 4.1.1 de la documentación [49]. También se explica en esta misma sección los valores que deben tomar los bits TCINTEN y ITCINTEN. En este trabajo se usarán los valores 1 y 0, respectivamente, lo cual indica en todos los posibles casos que las interrupciones de transferencia completa solo se generan cuando se ha completado la transferencia de todos los bloques de un PaRAM.

- **Source (SRC):** Ocupa los 4 bytes siguientes. Es una palabra de 32 bits que indica la dirección a partir de la cual empiezan a transferirse los bloques.

- **Count for 2nd dimension (BCNT):** Ocupa los siguientes 2 bytes. Indica el tamaño en bloques A de un bloque B. Este número se irá decrementando cada vez que se complete la transferencia de un bloque A hasta llegar a 0.

- **Count for 1st dimension (ACNT):** Ocupa los siguientes 2 bytes. Indica el tamaño en bytes de un bloque A.

- **Channel destination address (DST):** Ocupa los siguientes 8 bytes. Es una palabra de 32 bits que indica la dirección de memoria a la que son transferidos los bloques.

- **Destination BCNT index (DSTBIDX):** Ocupa los siguientes 4 bytes. Especifica un desplazamiento en bytes que se añade a la dirección de destino cada vez que termina la transferencia de un bloque A.

- **Source BCNT index (SRCBIDX):** Ocupa los siguientes 4 bytes. Especifica un desplazamiento en bytes que se añade a la dirección a partir de la cual empiezan a transmitirse los bloques cada vez que termina la transferencia de un bloque A.
- **BCNT reload (BCNTRLD):** Ocupa los siguientes 4 bytes. Una vez el contador BCNT llega a cero, el valor de este campo se recarga en BCNT.
- **Link address (LINK):** Ocupa los siguientes 4 bytes. Los datos del PaRAM al que apunte esta dirección se copiarán al PaRAM actual cuando se complete la transferencia de bloques. Este es el mecanismo que permite enlazar diferentes configuraciones. Un valor FFFFh especifica un vínculo nulo.
- **Destination CCNT index (DSTCIDX):** Ocupa los siguientes 4 bytes. Especifica un desplazamiento en bytes que se añade a la dirección de destino cada vez que termina la transferencia de un bloque B. No se usa en este trabajo.
- **Source CCNT index (SRCCIDX):** Ocupa los siguientes 4 bytes. Especifica un desplazamiento en bytes que se añade a la dirección de origen cada vez que termina la transferencia de un bloque B. No se usa en este trabajo.
- **Reservado (RSVD):** Ocupa los siguientes 4 bytes. No tiene función definida.
- **Count for 3rd dimension (CCNT):** Ocupa los últimos 4 bytes. Indica el tamaño en bloques B de un bloque C.

Si lo expuesto resulta confuso, recomiendo al lector que sobre todo observe la figura 5 de la documentación [49]. En ella se ve claramente como una transferencia A-Sincronizada está dividida en diferentes bloques y como existe un desplazamiento entre bloques dependiente de los campos BIDX y CIDX del PaRAM. En el caso de este trabajo el módulo EDMA3 inicia la transferencia de un bloque A cada vez que recibe un evento externo. Se dice que este evento sincroniza la transferencia de bloques. Cuando se han transmitido todos los bloques el módulo EDMA3 genera una interrupción de transferencia completa que será tratada por una rutina de interrupción, como se verá posteriormente.

9.3.2 Módulo McASP

El McASP (Multichannel Audio Serial Port) es un puerto en serie optimizado para recibir y/o transmitir datos a dispositivos externos o módulos internos en aplicaciones de audio multicanal.

A grandes rasgos, el proceso de recepción sucede como se explica a continuación: los datos son recibidos desde uno de los pines del puerto y pasados a un serializador. De este

serializador pasan a una unidad de formato, que permite organizar los bits de acuerdo a una cierta configuración definida por el programador. El proceso de transmisión sucede a la inversa, primero los datos pasan por una unidad de formato y posteriormente al serializador, desde donde son enviados en serie al pin correspondiente. Un diagrama de bloques completo puede encontrarse en la documentación de referencia, [50], Figura 1.

Cada uno de los pines de transmisión o recepción de datos del McASP se corresponde con un serializador. En la placa de desarrollo el pin AXR13 está conectado al pin DIN del codec, lo cual significa que para transmitir datos debe usarse el serializador 13. El pin AXR14 está conectado al pin DOUT del codec, lo cual significa que para recibir datos debe usarse el serializador 14.

En las transmisiones y recepciones del McASP una palabra se define como un conjunto de bits con significado. Un *slot* contiene una palabra, pero puede contener además bits adicionales sin significado. Un *frame* es un conjunto de *slots*.

El McASP necesita varias señales de reloj para sincronizarse. Estas pueden ser generadas por el propio módulo o por un dispositivo externo cuando el módulo trabaja en modo esclavo. Pueden usarse relojes independientes para emisión y recepción, aunque no es el caso en este trabajo. El *Frame Sync* sincroniza la recepción/transmisión de los *frames* y se recibe o emite en los pines AFSR/AFSX. El *Bit Clock* sincroniza la recepción/transmisión de bits individuales y se recibe o emite en los pines ACLKR/ACLKX. Finalmente, es necesario un reloj de alta frecuencia que se recibe o emite en los pines AHCLKR/AHCLKX. En la placa de desarrollo el pin AFSX está conectado al reloj de palabra del codec, el pin ACLKX está conectado al reloj de bit del codex y el pin AHCLKX está conectado al mismo oscilador modelo CB3LV a 24.576 Mhz que el codec. Por tanto, el McASP tiene que configurarse para trabajar como esclavo y recibir las señales de reloj desde el codec. Además, debe configurarse en los registros que las mismas señales de reloj en los pines de transmisión serán usadas para la recepción, ya que los pines ACLKR, AHCLKR y AFSR usados para las señales de reloj de recepción no están conectados a ningún reloj externo en la placa utilizada.

El McASP es compatible con un gran número de interfaces y formatos. En este trabajo se usará el formato TDM (Time-Division Multiplexing). En este formato, el flanco de subida del *Frame Sync* indica el comienzo de la transmisión de un *frame*. El número de *slots* en un frame y de bits en un *slot* se configura previamente, por lo que la transmisión/recepción puede continuar hasta el siguiente flanco de subida del *Frame Sync*, que indica un nuevo *frame*. En este trabajo un *frame* contiene dos *slots*, uno para el canal izquierdo y otro para el canal derecho, por lo que toda la información recogida por el codec en un periodo de muestreo se envía en un solo *frame*. Una figura con las formas de onda del formato TDM

puede encontrarse en la documentación de referencia [50], figura 6.

El módulo de McASP requiere un gran número de configuraciones para funcionar adecuadamente. En un intento de simplificar las explicaciones incluiré aquí los pasos para la configuración extraídos de la documentación [50], sección 2.4.1.2, adaptados al trabajo, resumidos y añadiendo en algunos casos explicaciones adicionales:

1. El GBLCTL (Global Control Register) es puesto a cero. Este registro controla las secciones de transmisión y recepción. Escribir en él un cero puede verse como el equivalente a mantener en reset el funcionamiento del puerto. Es importante volver a leer este registro siempre que se modifique para comprobar que el cambio se ha hecho efectivo, ya que los bits del registro son conmutados usando los relojes ACLKR y ACLKX. Normalmente, estos son cientos de veces más lentos que el reloj del DSP. Esto quiere decir que la escritura del registro no se hace efectiva inmediatamente desde el punto de vista del DSP y por lo tanto omitir la comprobación puede dar como resultado una mala inicialización.
2. Cada serializador del McASP tiene un registro de control, SRCTLn. Los registros de control de los serializadores que van a ser usados, SRCTL13 y SRCTL14 en el caso de este trabajo, deben ser puestos a cero. Esto es equivalente a mantener los serializadores en reset.
3. Configurar el registro de recepción RMASK. Este registro enmascara los bits leidos del buffer del serializador de recepción en todas las posiciones en las que su bit correspondiente es 0. También es necesario configurar el registro XMASK, su análogo para transmisión.
4. Configurar el formato de recepción de datos mediante el registro RFMT. También es necesario configurar el registro XFMT, su análogo para transmisión. Estos registros se componen de varios campos. A continuación se describen los campos del registro RFTM. Los campos del registro XFMT tienen exactamente las mismas funciones pero actúan sobre la unidad de formato de transmisión en lugar de sobre la de recepción:
   a) RDATDLY configura el valor del retardo en bits del primer bit efectivo respecto al *frame sync*. Por ejemplo, con un valor de 1 el primer bit efectivo se recibe un ciclo del reloj ACLKR después de la llegada de un *frame sync*.
   b) RRVRS define el orden de la cadena de bits. Si se pone a 0 el primer bit recibido es el menos significativo. Si se pone a 1 el primer bit recibido es el más significativo.
   c) RPAD define el valor usado para rellenar los bits enmascarados por RMASK.
   d) RPBIT define el valor usado para rellenar los bits enmascarados por RMASK si RPAD es 0x2.

e) Los bits RSSZ determinan el tamaño del *slot* de recepción. 0xB indica un tamaño de *slot* de 24 bits.

f) El bit RBUSEL selecciona si las lecturas del buffer del serializador (XRBUF) se originan en el bus de configuración o en el bus de datos.

g) Los bits RROT definen una rotación hacia la derecha en número de bits.

5. Configurar el registro AFSRCTL (Receive Frame Sync Control Register), que define las características del *frame sync*. Como el RFMT, este registro tiene su análogo para la sección de transmisión que también debe ser configurado, llamado ACLKXCTL. Este último registro contiene un campo adicional llamado ASYNC que habilita el uso de los relojes de transmisión para la sincronización de la recepción. Para ello, el bit 6 debe tener valor 0. Para más detalles, consultar el diagrama de generación del *frame sync* en la figura 17 de la documentación [50].
   a) Los bits RMOD determinan el modo de trabajo del *frame sync*. Como ya se mencionó, en este trabajo se usa el formato TDM con 2 *slots*.
   b) El bit FRWID indica la anchura del *frame sync* durante su periodo activo. Las opciones son una anchura de un bit o de una palabra completa.
   c) El bit FSRM indica la fuente del *frame sync*. Un 0 indica una fuente externa.
   d) El bit FSRP indica la polaridad del *frame sync*. Un 0 indica que el flanco de subida marca el inicio de un *frame*.
6. Configurar el reloj de bits del receptor mediante el registro ACLKRCTL. Es necesario configurar también el reloj de bits de transmisión mediante el registro ACLKXCTL.
   a) El bit CLKRP determina la polaridad del stream de bits. Un cero determina que el receptor muestrea los datos en el flanco de bajada del reloj en serie y que por lo tanto el dispositivo externo debería transmitir los datos en el flanco de subida.
   b) El bit CLKRM determina la fuente del CLK. Un cero determina que la fuente es externa. Como se mencionó anteriormente, es posible habilitar el uso de los relojes de transmisión para la sincronización de la recepción mediante el bit ASYNC del ACLKXCTL. Para más detalles, consultar los diagramas de generación de relojes en las figuras 15 y 16 de la documentación [50].
   c) Los bits CLKRDIV determinan el número por el que se divide la frecuencia del reloj de alta frecuencia si este es usado como fuente para el ACLKR. Esto es sencillo de entender observando las mismas figuras mencionadas anteriormente.
7. Configurar el registro AHCLKRCTL (Receive High-Frequency Clock Control Register) y el AHCLKXCTL, su análogo para transmisión. En el caso de este trabajo solo hay que indicar que la fuente de reloj es externa.
8. Configurar el RTDM (Receive TDM time slot register) Este registro indica que *slots*

TDM están activos, o dicho de otro modo, cuantos *slots* contiene un *frame*.

9. Configurar el registro RINTCTL, que controla la generación de interrupciones. El bit ROVRN con valor 1 indica que debe generarse una interrupción en caso de *overrun*. Un *overrun* sucede cuando un serializador de recepción intenta escribir en su buffer pero los datos antiguos aún no han sido leídos por el DMA o el DSP.
10. El registro RCLKCHK configura el sistema de detección de errores de reloj. Este puede verse en forma de diagrama en las figuras 32 y 33 de la documentación [50]. En este trabajo, el reloj del sistema se divide por 256. Cada 32 ticks del reloj de alta frecuencia (AHCLK) el reloj del sistema escalado debe dar entre 0 y 255 ticks. El reloj del sistema en el caso del OMAP-L138 proviene del reloj SYSCLK2, que funciona a la mitad de frecuencia que el DSP. Para más información sobre los relojes del sistema consultar la hoja de datos. He configurado así este registro simplemente porque es el mayor margen que permite el sistema de detección de errores de reloj.
11. Configurar los serializadores usando los registros SRCTLn. En el caso de este trabajo se configuran los serializadores 13 y 14 usando los registros SRCTL13 y SRCTL14 para funcionar como transmisor y receptor, respectivamente. Además, es necesario indicar si funcionan con lógica positiva o negativa. En el caso de este trabajo se usa lógica positiva.
12. Los pines del McASP pueden usarse para McASP o GPIO (General Purpose Input-Output). En el caso de este trabajo todos los pines son configurados para McASP usando el registro PFUNC.
13. Configurar los pines como entradas o salidas usando el registro PDIR. En el caso de este trabajo se programan todos los pines como entrada excepto AXR13.
14. Activar los divisores de reloj cambiando en el GBCTL los bits RCLKRST y XCLKRST. Este paso es necesario incluso si los relojes usados son externos.
15. Limpiar el estado del transmisor y el receptor mediante los registros XSTAT y RSTAT.
16. Sacar los serializadores del reset mediante los bits RSRCLR y XSRCLR del GBCTL.
17. Sacar las máquinas de estado de transmisión y recepción del reset usando los bits RSMRST y XSMRST del GBCTL. Las máquinas de estado controlan la interacción entre las diferentes unidades. En cuanto se sacan del reset, el sistema puede empezar a recibir *frame syncs*.
18. Un *buffer underrun* puede ocurrir cuando un serializador programado para transmitir intenta transmitir datos al pin de salida pero encuentra que el bufer todavía no ha sido escrito desde la última transmisión. Cuando esto ocurre, la máquina de estado del transmisor activa la bandera XUNDRN. El McASP puede configurarse para que la salida sea silenciada cuando esto sucede mediante el registro AMUTE. Para evitar

un *underrun* puede escribirse un cero en el buffer de salida antes de comenzar el funcionamiento.

19. Sacar los generadores de *frame sync* del reset. Este paso es necesario incluso si la fuente de los *frame sync* es externa.

Si lo aquí expuesto no es suficiente para adquirir una idea general del funcionamiento del módulo McASP recomiendo al lector revisar la documentación [50], fijándose sobre todo en las figuras que se han citado en este texto y leyendo en las secciones correspondientes la descripción de los registros de configuración que en ellas aparecen. Es de gran importancia el registro GBLCTL, cuya descripción se encuentra en la sección 3.8 de la documentación.

9.3.3 Módulo I2C

El módulo I2C provee una interfaz entre el procesador C674x y otros dispositivos compatibles con la especificación I2C. Los componentes externos conectados al bus de dos lineas pueden transmitir y recibir palabras de hasta 8 bits.

9.3.4 Módulo GPIO

El puerto GPIO (General Purpose Input/Output) proporciona una forma de interactuar con componentes externos usando una interfaz de baja velocidad.

Las señales GPIO están agrupadas en bancos de 16 señales por banco. Cada dos bancos tienen un grupo de registros de 32 bits para su control. Para configurar un pin GPIO como entrada o salida se usa el registro DIRn correspondiente, donde n es un entero. Para escribir o limpiar un pin se usan los registros *SET_DATAn* o *CLR_DATAn*, respectivamente.

Por ejemplo, para llevar el pin 8 del banco 0 a estado alto se llevarían a cabo los siguientes pasos usando el registro 01, dedicado a los bancos 0 y 1:

1. Escribir un 1 en el bit 8 de *CLR_DATA*01. Con esto se pone el pin en estado bajo.
2. Escribir un 1 en el bit 8 de *DIR*01. Con esto se indica que el pin se usa como salida.
3. Escribir un 1 en el bit 8 de *SET_DATA*01. Con esto se pone el pin en estado alto.

Una vez configurado el pin GPIO como entrada o salida puede omitirse la configuración en usos sucesivos. Una descripción completa de las funcionalidades del puerto GPIO puede encontrarse en la documentación [51].

9.3.5 Otros módulos

El SoC contiene otros módulos necesarios para el funcionamiento de cualquier software en cualquier placa, ya sea la de desarrollo o una personalizada. Algunos ejemplos son el módulo encargado de administrar la energía o el módulo encargado de la comunicación con la memoria RAM.

En un entorno de desarrollo estos módulos son configurados directamente por el depurador mediante un script cuando se inicia una sesión de depuración. Aunque no se tienen en cuenta en este trabajo, la configuración realizada por el script debería incluirse en el punto de entrada del código de una aplicación real. Entre otras cosas, el script configura la velocidad de los nucleos, los relojes, el módulo de energía y la memoria. Existe software para pasar de un entorno de desarrollo a un entorno de producción de forma automatizada, aunque este proceso queda fuera de los límites de este trabajo.

9.4 Interrupciones

Las interrupciones del DSP funcionan mediante una tabla de interrupciones. Esta suele implementarse totalmente en ensamblador y tiene un tamaño fijo, conteniendo rutinas de igual tamaño para las 15 interrupciones posibles, asociada cada una de ellas a un evento determinado.

La configuración de las interrupciones se completa mediante los siguientes registros no accesibles mediante el mapa de memoria:

- El ISTP (Interrupt Service Table Pointer Register) es usado para localizar la rutina de interrupción. El campo ISTB contiene la dirección base y el campo HPINT el desplazamiento hasta la rutina de interrupción. Al inicio de un programa hay que configurar el ISTB para que apunte a la dirección donde se ha situado la tabla de interrupciones.
- El IER (Interrupt Enable Register) sirve para activar las interrupciones del DSP.
- El IFR (Interrupt Flag Register) muestra un 1 en el bit correspondiente cuando ocurre la interrupción.
- El ICR (Interrupt Clear Register) permite limpiar manualmente las interrupciones en el IFR, algo que es necesario hacer cada vez que sucede una interrupción.
- El CSR (Control Status Register) contiene bits de control y estado. El bit menos significativo es el GIE (Global Interrupt Enabled) y es necesario ponerlo a 1 para habilitar las interrupciones.

Una guía completa sobre las interrupciones del DSP puede encontrarse en la documentación [52].

9.5 El codec de audio

La placa de desarrollo incluye un codec de audio, el TLV320AIC3106. Este codec contiene conversores A/D y D/A estéreo que pueden funcionar a una frecuencia de muestreo de hasta 96 kHz. Ofrece además otras funciones, como efectos de audio programables, ganancia programable y entrada de micrófono, que no serán usadas en este trabajo.

La configuración del codec se realiza escribiendo registros mediante el protocolo I2C. Cuando quiere escribirse un cierto registro es necesario enviar dos bytes, el primero contendrá la dirección del registro y el segundo el valor a escribir.

En la placa de desarrollo el codec tiene el pin MCLK (master clock) conectado a un oscilador modelo CB3LV a 24.576 Mhz. A partir de esta frecuencia es posible generar la salida en los pines BCLK (bit clock) y WCLK (word clock) mediante un divisor de frecuencia programable, cuyos detalles de funcionamiento se encuentran en la sección 11.3.3.1 de la hoja de datos [53]. Esto permite al codec funcionar como master en la transmisión de datos.

La transmisión de datos puede realizarse de varias formas. La utilizada en este trabajo es la llamada *DSP Mode*. En este modo el bit más significativo del canal derecho es valido en el flanco de subida del WCLK y el resto de bits de ambos canales se transmiten consecutivamente a continuación, siendo válidos en cada flanco de bajada del BCLK. Un esquema con las formas de onda está disponible en la sección 11.3.2.4 de la hoja de datos [53].

Se omiten aquí los detalles de la configuración del codec en este trabajo, que serán descritos en el capítulo siguiente.

9.6 Archivo CMD

Este archivo es usado en conjunto con el vinculador para generar un programa de salida. En él se describen las regiones de memoria y su posición en el mapa de memoria. Cada sección de código quedará vinculada a una región en la que será almacenada cuando se ejecute el programa.

Además, es posible incluir comandos en el archivo para especificar el tamaño de la pila y

el montículo o vincular librerías externas. El uso que se hace de este archivo quedará claro en el capítulo de implementación.

9.7 Conclusiones

Se han descrito de forma breve varios módulos del SoC así como el codec de audio disponible en la placa de desarrollo, haciendo especial énfasis en el módulo EDMA3 y en el puerto McASP. La configuración de estos dos módulos es la parte más importante del trabajo realizado en el siguiente capítulo.

CAPÍTULO 10

Controladores para el OMAP-L138

En este capítulo se implementan varios controladores para configurar los módulos del OMAP-L138 y crear un sistema de procesamiento en tiempo real que no necesite sistema operativo. La implementación está basada en el código escrito por T.B. Welch y otros incluido en la última edición de su libro en el momento de realizar este trabajo [11], aunque se ha escrito en su mayoría desde cero consultando la documentación del dispositivo.

10.1 Visión general

En este trabajo se usará una transferencia EDMA3 para manejar 3 buffers implementados usando arrays de floats. En un instante determinado, uno de los buffers recibirá muestras mediante DMA del buffer del serializador de recepción del McASP, el otro estará procesándose y el último estará siendo transferido mediante DMA al buffer del serializador de transmisión del McASP. Cada vez que un buffer termine el procesamiento se producirá una rotación y pasará a ser el buffer del cual se toman muestras para ser enviadas al McASP. De este modo, el DSP está permanentemente ocupado procesando uno de los buffers y se minimiza el tiempo dedicado a interrupciones, que solo suceden cuando un buffer está listo para su procesamiento. El módulo de EDMA3, mediante los PaRAM, hace posible rotar automáticamente los buffers y asegurar así que el DSP siempre tenga datos que procesar.

El McASP recibirá constantemente muestras del codec de audio y estas serán transferidas mediante EDMA3 al buffer destinado para ello en ese momento. El codec enviará alternativamente muestras del canal derecho y el canal izquierdo, por lo que las muestras del canal derecho se encontrarán en las posiciones pares del array y las del canal izquierdo en las impares.

Mi intención es usar en una versión posterior el nucleo ARM9 para controlar una interfaz de usuario con botones, potenciómetros digitales y una pantalla LCD. El nucleo ARM9 usará la memoria compartida de 128 kB e interrupciones para comunicarse con el DSP y

enviarle los mensajes del usuario para cambiar el valor de los parámetros de procesamiento. Aunque no he incluido esta parte en el trabajo, el código presentado deja abierta la puerta para implementarlo posteriormente.

El programa completo se ejecutará desde la memoria L2, en la que será cargado mediante la sonda de depuración. En una versión posterior habría que implementar un sistema de caché o contemplar el uso del sistema operativo en tiempo real de Texas Instruments, TI-RTOS. Este sistema operativo podría ejecutarse en el DSP a la vez que se ejecuta un Linux sobre el ARM9. En este caso habría que dividir la memoria RAM en dos regiones y hacer las modificaciones necesarias en la versión del sistema operativo Linux para que no hubiera un conflicto de memoria o controladores. Existe mucha documentación online al respecto, pero contemplar en profundidad estas cuestiones queda fuera de los límites de este trabajo.

Cada grupo de código se dividirá en tres partes, dos cabeceras y un archivo de implementación. Una de las cabeceras contendrá macros para acceder a los registros. La otra contendrá los prototipos de las funciones y la definición de las estructuras de datos. El código completo de este capítulo puede encontrarse en la ruta *./Proyecto/Retardo Modulado/C674x/ModulatedDelay*.

10.2 Archivo CMD

En el listado 10.1 se muestra el archivo CMD creado para este proyecto a partir del que se genera por defecto.

La definición de regiones de memoria dentro del bloque 'MEMORY' es sencilla. Cada linea define una región. El valor de 'o' indica en que dirección empieza la región. El valor de 'l' indica el tamaño en bytes.

Dentro del bloque 'SECTIONS' se asocia cada sección a una de las regiones de memoria definidas. Una determinada sección de código será almacenada en la región de memoria a la que haya sido asociada.

Los comentarios en las lineas que definen las regiones de memoria indican en que dispositivo de memoria se encuentran situadas. He comentado también las secciones para que resulte más claro el contenido de cada una de ellas.

En la memoria L2 he reservado una región llamada 'VECTORS' de 512 bytes para almacenar la tabla de rutinas de interrupción a partir de la dirección 0x11800000.

También he creado dos secciones personalizadas. La llamada 'vectors' contendrá las rutinas

```
1  // Run time support library
2  -l rts6740_elf.lib
3  // Optimized floating point operations
4  -l ti/mathlib/lib/mathlib_rts.lib
5
6  -heap              0x00002400      // Size of the heap in bytes = 9216
7  -stack                 0x00000C00          // Size of the stack in bytes = 3072
8
9  MEMORY
10 {
11 #ifdef DSP_CORE       /* DSP exclusive memory regions */
12
13      DSPL2ROM      o = 0x00700000  l = 0x00100000  /* 1MB L2 DSP local ROM */
14      DSPL2RAM      o = 0x00800000  l = 0x00040000  /* 256kB L2 DSP local RAM */
15      DSPL1PRAM     o = 0x00E00000  l = 0x00008000  /* 32kB L1 DSP local Program RAM */
16      DSPL1DRAM     o = 0x00F00000  l = 0x00008000  /* 32kB L1 DSP local Data RAM */
17
18 #endif
19
20      SHDSPL2ROM    o = 0x11700000  l = 0x00100000  /* 1MB L2 Shared Internal ROM */
21      VECTORS       o = 0x11800000  l = 0x00000200  /* 512 bytes of L2 are used for interruption table */
22      SHDSPL2RAM    o = 0x11800200  l = 0x0003FE00  /* 255.5kB L2 Shared Internal RAM */
23      SHDSPL1PRAM   o = 0x11E00000  l = 0x00008000  /* 32kB L1 Shared Internal Program RAM */
24      SHDSPL1DRAM   o = 0x11F00000  l = 0x00008000  /* 32kB L1 Shared Internal Data RAM */
25      EMIFACS0      o = 0x40000000  l = 0x20000000  /* 512MB SDRAM Data (CS0) */
26      EMIFACS2      o = 0x60000000  l = 0x02000000  /* 32MB Async Data (CS2) */
27      EMIFACS3      o = 0x62000000  l = 0x02000000  /* 32MB Async Data (CS3) */
28      EMIFACS4      o = 0x64000000  l = 0x02000000  /* 32MB Async Data (CS4) */
29      EMIFACS5      o = 0x66000000  l = 0x02000000  /* 32MB Async Data (CS5) */
30      SHRAM         o = 0x80000000  l = 0x00020000  /* 128kB Shared RAM */
31      DDR2          o = 0xC0000000  l = 0x08000000  /* 128MB DDR2 Data */
32
33 #ifndef DSP_CORE      /* ARM exclusive memory regions */
34
35      ARMROM        o = 0xFFFD0000  l = 0x00010000  /* 64kB ARM local ROM */
36      ARMRAM        o = 0xFFFF0000  l = 0x00002000  /* 8kB ARM local RAM */
37
38 #endif
39 }
40
41 SECTIONS
42 {
43      .text           >  SHDSPL2RAM                /*Executable code*/
44      .stack          >  SHDSPL2RAM                /*Stack*/
45      .bss            >  SHDSPL2RAM                /*Global and static variables*/
46      .cio            >  SHDSPL2RAM                /*Buffer for stdio functions*/
47      .const          >  SHDSPL2RAM                /*Initialized global variables*/
48      .switch         >  SHDSPL2RAM                /*Jump tables for large switch statements*/
49      .sysmem         >  SHDSPL2RAM                /*Memory allocation heap*/
50      .far            >  SHDSPL2RAM                /*Global and static variables declared far*/
51
52      /* TI-ABI or COFF sections */
53      .cinit          >  SHDSPL2RAM                /*Table to autoinitialize global variables when using the
            --rom_model option*/
54
55      /* EABI sections */
56      .neardata       >  SHDSPL2RAM                /*Near non-const global and static variables that are
            explicitily initialized*/
57      .fardata        >  SHDSPL2RAM                /*Far non-const global and static variables that are
            explicitily initialized. All pointers are far by default.*/
58      .rodata         >  SHDSPL2RAM                /*Global and static variables that have near and const
            qualifiers*/
59
60      /* Custom sections */
61      "vectors"       >   VECTORS                  /*Interruption SRs*/
62      "SDRAM"         >   DDR2                             /*External DDR2 memory*/
63 }
```

Listado 10.1: Archivo de comandos del vinculador *OMAPL138.cmd*

```
1  #ifndef SYSCFG_H_
2  #define SYSCFG_H_
3  #define SYS_BASE            0x01C14000
4  #define PINMUX0             *(unsigned int*)(SYS_BASE + 0x120)
5  #define PINMUX1             *(unsigned int*)(SYS_BASE + 0x124)
6  #define PINMUX2             *(unsigned int*)(SYS_BASE + 0x128)
7  #define PINMUX5             *(unsigned int*)(SYS_BASE + 0x134)
8  #define PINMUX13            *(unsigned int*)(SYS_BASE + 0x154)
9  #define PINMUX0_VALUE 0x08111111
10 #define PINMUX1_VALUE 0x11111111
11 #define PINMUX2_VALUE 0x11111111
12 #define PINMUX5_VALUE 0x00008000
13 #define PINMUX13_VALUE 0x00008800
14 #endif /* SYSCFG_H_ */
```

Listado 10.2: Contenido del archivo *SYSCFG.h*

de interrupción. La llamada 'SDRAM' será usada para almacenar variables en la memoria RAM mediante una directiva del compilador.

Además, al comienzo del archivo se vinculan dos librerías mediante el comando 'l'. La librería *rts6740_elf.lib* contiene soporte básico para la arquitectura y rutinas de bajo nivel para realizar operaciones básicas. La librería *mathlib_rts.lib* contiene rutinas optimizadas en ensamblador para realizar operaciones matemáticas como senos o logaritmos de forma eficiente. La documentación de esta librería está disponible en [54].

También se establecen los tamaños del montículo y la pila mediante los comandos 'heap' y 'stack', respectivamente.

Ni el compilador ni el depurador muestran advertencia alguna sobre posibles desbordamientos de pila y he comprobado de forma práctica que esto puede dar lugar a errores difíciles de depurar. Si el programa se comporta de forma extraña durante la depuración sin causa aparente, por ejemplo, el contador de programa salta a direcciones de memoria fuera de la región de código, seguramente se haya producido un desbordamiento de la pila.

10.3 Multiplexores de pines

El OMAP-L138 tiene varios multiplexores que controlan a que módulo está conectado cada pin. Antes de la ejecución de un programa es necesario configurar estos multiplexores mediante registros para tal fin. En la página de Texas Instruments puede encontrarse una aplicación que genera automáticamente valores para los registros de control de los multiplexores en función de los módulos que se quiera usar [55].

En el listado 10.2 puede verse el contenido del archivo *SYSCFG.h*. En el se definen las macros de acceso a los registros de control de multiplexores que es necesario usar y los valores para cada uno de ellos. Como se verá posteriormente, estos valores son usados dentro de la función *main* antes de la ejecución del programa.

```
1  #ifndef I2C_DEFINES_H_
2  #define I2C_DEFINES_H_
3
4  #include <ti/csl/tistdtypes.h>
5
6  typedef struct  {
7      volatile Uint32 ICOAR;
8      volatile Uint32 ICIMR;
9      volatile Uint32 ICSTR;
10     volatile Uint32 ICCLKL;
11     volatile Uint32 ICCLKH;
12     volatile Uint32 ICCNT;
13     volatile Uint32 ICDRR;
14     volatile Uint32 ICSAR;
15     volatile Uint32 ICDXR;
16     volatile Uint32 ICMDR;
17     volatile Uint32 ICIVR;
18     volatile Uint32 ICEMDR;
19     volatile Uint32 ICPSC;
20     volatile Uint32 ICPID1;
21     volatile Uint32 ICPID2;
22 } I2CRegs;
23
24 #define I2C0     ((I2CRegs *)0x01c22000)
25
26 #define ICMDR_NACKMOD                   0x8000
27 #define ICMDR_FREE                      0x4000
28 #define ICMDR_STT                       0x2000
29 #define ICMDR_IDLEEN                    0x1000
30 #define ICMDR_STP                       0x0800
31 #define ICMDR_MST                       0x0400
32 #define ICMDR_TRX                       0x0200
33 #define ICMDR_XA                        0x0100
34 #define ICMDR_RM                        0x0080
35 #define ICMDR_DLB                       0x0040
36 #define ICMDR_IRS                       0x0020
37 #define ICMDR_STB                       0x0010
38 #define ICMDR_FDF                       0x0008
39 #define ICMDR_BC_MASK                   0x0007
40
41 #endif /* I2C_DEFINES_H_ */
```

Listado 10.3: Macros y estructura para los registros del puerto I2C.

10.4 Módulo I2C

En el listado 10.3 se muestra el contenido del archivo *I2C_defines.h*. Este archivo define en primer lugar una estructura de datos con todos los registros de 32 bits que permiten configurar el puerto I2C. Posteriormente se define una macro que consiste en una conversión de la dirección de memoria del primer registro de configuración del puerto a un puntero a la estructura de datos. Este puntero permite un fácil acceso a todos los registros ya que sus direcciones se encuentran situadas consecutivamente en el mapa de memoria. Este es un patrón que se repetirá en todos los archivos usados para definir macros de acceso a registros.

Nótese también que en las declaraciones se añade la palabra reservada *volatile*. Esto indica al compilador que es posible que la variable cambie entre accesos aunque aparentemente no lo haga de acuerdo al código. Imaginemos por ejemplo que una variable es modificada por una rutina de DMA. Si el compilador optimiza la traducción de código asumiendo que la variable no cambia de valor es posible que el programa resultante no funcione de forma correcta. Muchos registros pueden presentar este comportamiento, cambiando su valor incluso en respuesta a un determinado estado del procesador.

```
#ifndef I2C_H_
#define I2C_H_

#include "I2C_defines.h"
#include <ti/csl/tistdtypes.h>

void Init_I2C();
int I2C_WriteBytes(Uint16 slaveaddr, Uint8* write_data, Uint16 write_len);

#endif /* I2C_H_ */
```

Listado 10.4: Cabecera I2C.h.

```
#include "I2C.h"

void Init_I2C()
{
    I2CRegs *i2c = I2C0;

    i2c->ICMDR    = 0;
    i2c->ICPSC    = 7;
    i2c->ICCLKL   = 37;
    i2c->ICCLKH   = 37;
    i2c->ICMDR   |= 0x0020;
}

void wait(Uint32 delay)
{
    volatile Uint32 i;
    for (i = 0;i < delay;i++);
}

int I2C_WriteBytes(Uint16 slaveaddr, Uint8* write_data, Uint16 write_len) {
    I2CRegs *i2c = I2C0;
    volatile Int32 timeout = 100000, i;

    /* Set the I2C controller to write len bytes */
    i2c->ICCNT = write_len;
    i2c->ICSAR = slaveaddr;
    i2c->ICMDR = ICMDR_STT | ICMDR_TRX | ICMDR_MST | ICMDR_IRS | ICMDR_FREE;
    /*Software timer*/
    wait(100);
    /* Transmit data */
    for (i = 0; i < write_len; i++) {
        i2c->ICDXR = write_data[i];
        // Wait for "XRDY" flag to transmit data
        while ( !(i2c->ICSTR & 0x0010) ){
            if(timeout-- < 0) {
                i2c->ICMDR = 0;
                return 0;
            }
        }
    }
    i2c->ICMDR |= 0x0800;
    return 1;
}
```

Listado 10.5: Contenido del archivo I2C.c.

Las macros definidas posteriormente representan los bits del registro ICMDR. Mediante un OR lógico es posible seleccionar que bits deben estar activados utilizando las macros asignadas a cada uno de ellos.

En el listado 10.4 se muestra el contenido del archivo *I2C.h*. La función *Init_I2C()* sirve para inicializar el puerto mientras que la función *I2C_WriteBytes()* se usa para escribir un array de enteros de 8 bits. Ambas funciones se describen en las siguientes secciones.

Los detalles completos sobre el módulo I2C pueden encontrarse en la documentación de referencia [56]. Se omite aquí una descripción de los detalles del protocolo.

10.4.1 Función de inicialización

En el listado 10.5 se muestra el contenido del archivo *I2C.c*. A continuación se explica la función de inicialización, *Init_I2C()*:

- Cuando el bit 5 (IRS) del registro ICMDR (I2C Moder Register) es puesto a 0 el puerto I2C permanece deshabilitado.
- El registro ICPSC (I2C Prescaler Register) divide el reloj base para obtener la frecuencia de trabajo deseada del puerto I2C. La frecuencia del reloj I2C será la frecuencia del reloj base dividida entre el número entero sin signo presente en los primeros 8 bits de este registro mas la unidad. El reloj base del módulo I2C es alimentado con una señal de 75 MHz cuando el DSP funciona a 300 Mhz. Así, en el caso de este trabajo, el resultado final es de 9.375 MHz. Un gráfico que muestra la generación de relojes en el módulo I2C puede encontrarse en la documentación [56], Figura 3.
- Cuando el módulo I2C trabaja como master el periodo del reloj de entrada es multiplicado por el resultado de sumar 5 al entero sin signo contenido en los primeros 16 bits del registro ICCLKL (I2C Clock Low-Time Divider Register) para determinar la duración del estado bajo en el pin SCL (Serial Clock Line) del I2C. La cantidad sumada varía si el valor del IPSC es menor que 2. Para más detalles, consultar la figura 3 de la documentación [56].
- El registro ICCLKH (I2C Clock High-Time Divider Register) es análogo al anterior, pero determina el tiempo que el reloj SCL permanece en estado alto. En el caso de este trabajo ambos tiempos son iguales. La fórmula para calcular la frecuencia resultante del SCL se encuentra en la figura 3 mencionada anteriormente. En el caso de este trabajo la frecuencia final del reloj SCL es de 100 Khz.
- Finalmente, escribiendo un 1 en el bit IRS del ICMDR se habilita el puerto I2C.

10.4.2 Función de escritura

A continuación se explica la función de escritura, *I2C_WriteBytes(...)*, que toma como parámetros la dirección del esclavo, un array de enteros sin signo de 8 bits y el tamaño de dicho array:

- El registro ICCNT (I2C Data Count Register) indica el número de palabras que serán enviadas. En este caso, el número de palabras a enviar es uno de los parámetros de la función.
- El registro ICSAR (I2C Slave Address Register) contiene una dirección de 7 bits que será usada para iniciar transferencias con el esclavo. Esta dirección es uno de

los parámetros de la función.

- El registro ICMDR es configurado en la linea 27 para iniciar una transferencia. El bit STT genera una condición de comienzo en el bus, es decir, el pin SDA (Serial Data) pasa de estado alto a estado bajo. El bit TRX pone el puerto I2C en modo de transmisión. El bit MST indica que el módulo I2C esta funcionando en modo maestro. El bit IRS activa el puerto I2C sacándolo del reset. El bit FREE se usa para depuración. Cuando este último bit está activado el puerto I2C seguirá funcionando si ocurre un punto de ruptura en el depurador de alto nivel.
- He comprobado experimentalmente que es necesario añadir un pequeño delay en este punto para que la configuración de los registros surta efecto, aunque la documentación no dice nada al respecto.
- La palabra de 8 bits a enviar se escribe en el registro ICDXR (I2C Data Transmit Register). Este paso y el siguiente se repiten para cada palabra de 8 bits.
- El bit ICXRDY del registro ICSTR (I2C Interrupt Status Register) indica que el registro ICDXR está preparado para admitir nuevos datos. El código comprueba este bit un máximo de 1000000 veces. Si su valor continua siendo 0 se asume que ha sucedido un error y se devuelve un 0.
- Finalmente, se escribe un 1 en el bit STP del ICMDR. Esto genera una condición de finalización en el bus, es decir, el pin SDA pasa de estado bajo a estado alto. La finalización correcta de la transferencia devuelve un 1.

10.5 Codec AIC3106

En el listado 10.6 se muestra el contenido del archivo *aic*31_*if*.*h*, extraido del ejemplo disponible en el SDK de Texas Instruments para el uso del codec. Este archivo contiene la definición de las macros para el acceso a los registros de configuración del codec. Se ha cortado el archivo por motivos de espacio. Las direcciones de los registros así como todos los detalles de funcionamiento y configuración del codec pueden encontrarse en la hoja de datos [53]. Nótese que este archivo no es propio sino código libre distribuido por Texas Instruments.

En el listado 10.7 se muestra el contenido del archivo *AIC*3106.*h*. Una de las funciones es la encargada de escribir los registros de configuración del codec mediante I2C, la otra inicializa el codec. Ambas se describen en las siguientes secciones.

```
#ifndef _AIC31_IF_H_
#define _AIC31_IF_H_
/*****************************************************************************
**                        INTERNAL MACRO DEFINITIONS
*****************************************************************************/
/*
** Register Address for AIC31 Codec
*/
#define AIC31_P0_REG0               (0)  /* Page Select  */
#define AIC31_P0_REG1               (1)  /* Software Reset */
#define AIC31_P0_REG2               (2)  /* Codec Sample Rate Select */
#define AIC31_P0_REG3               (3)  /* PLL Programming A */
#define AIC31_P0_REG4               (4)  /* PLL Programming B */
#define AIC31_P0_REG5               (5)  /* PLL Programming C */
#define AIC31_P0_REG6               (6)  /* PLL Programming D */
#define AIC31_P0_REG7               (7)  /* Codec Datapath Setup */
#define AIC31_P0_REG8               (8)  /* Audio Serial Data I/f Control A */
#define AIC31_P0_REG9               (9)  /* Audio Serial Data I/f Control B */
#define AIC31_P0_REG10              (10) /* Audio Serial Data I/f Control C */
#define AIC31_P0_REG11              (11) /* Audio Codec Overflow Flag */
#define AIC31_P0_REG12              (12) /* Audio Codec Digital Filter Ctrl */
#define AIC31_P0_REG13              (13) /* Headset / Button Press Detect A */
#define AIC31_P0_REG14              (14) /* Headset / Button Press Detect B */
#define AIC31_P0_REG15              (15) /* Left ADC PGA Gain Control */
#define AIC31_P0_REG16              (16) /* Right ADC PGA Gain Control */
#define AIC31_P0_REG17              (17) /* MIC3L/R to Left ADC Control */
#define AIC31_P0_REG18              (18) /* MIC3L/R to Right ADC Control */
#define AIC31_P0_REG19              (19) /* LINE1L to Left ADC Control */
#define AIC31_P0_REG20              (20) /* LINE2L to Left ADC Control */
#define AIC31_P0_REG21              (21) /* LINE1R to Left ADC Control */
#define AIC31_P0_REG22              (22) /* LINE1R to Right ADC Control */
#define AIC31_P0_REG23              (23) /* LINE2R to Right ADC Control */
#define AIC31_P0_REG24              (24) /* LINE1L to Right ADC Control */
#define AIC31_P0_REG25              (25) /* MICBIAS Control */
#define AIC31_P0_REG26              (26) /* Left AGC Control A */
#define AIC31_P0_REG27              (27) /* Left AGC Control B */
#define AIC31_P0_REG28              (28) /* Left AGC Control C */
#define AIC31_P0_REG29              (29) /* Right AGC Control A */
#define AIC31_P0_REG30              (30) /* Right AGC Control B */
#define AIC31_P0_REG31              (31) /* Right AGC Control C */
#define AIC31_P0_REG32              (32) /* Left AGC Gain */
#define AIC31_P0_REG33              (33) /* Right AGC Gain */
#define AIC31_P0_REG34              (34) /* Left AGC Noise Gate Debounce */
#define AIC31_P0_REG35              (35) /* Right AGC Noise Gate Debounce */
#define AIC31_P0_REG36              (36) /* ADC Flag */
#define AIC31_P0_REG37              (37) /* DAC Power and Output Driver Control */
#define AIC31_P0_REG38              (38) /* High Power Output Driver Control*/
[...]
```

Listado 10.6: Macros para los registros del codec AIC3106.

```
#ifndef AIC3106_H_
#define AIC3106_H_

Uint32 AIC3106_Write_Reg(Uint8 address, Uint8 data);
Uint32 Init_AIC3106();

#endif /* AIC3106_H_ */
```

Listado 10.7: Cabecera AIC3106.h.

```
Uint32 AIC3106_Write_Reg(Uint8 address, Uint8 data)
{
    Uint8 transfer_bytes[2];
    transfer_bytes[0] = address;
    transfer_bytes[1] = data;
    return I2C_WriteBytes(AIC31_I2C_ADDR,transfer_bytes, 2);
}
```

Listado 10.8: Definición de la función de escritura de registros en el AIC3106.

10.5.1 Función de escritura en registros

En el listado 10.8 se muestra la definición de la función *AIC3106_Write_Reg(...)*. Esta toma como parámetros dos enteros sin signo de 8 bits, la dirección del registro y los datos, los introduce en un array y los pasa la función de escritura implementada para el módulo I2C junto con la dirección de esclavo del codec, definida en el archivo *aic31_if.h*.

10.5.2 Función de inicialización

En el listado 10.9 se muestra la definición de la función *Init_AIC3106(...)*. En esta función se hace uso de la función implementada en la sección anterior para escribir en los registros de configuración del codec mediante el módulo I2C. A continuación se explica todo el procedimiento de configuración:

- Los registros del AIC3106 están divididos en dos páginas. Para seleccionar la primera de ellas se usa el primer registro. Escribir un 0 equivale a seleccionar la primera página de registros.
- El codec se resetea escribiendo un 1 en el bit 7 del segundo registro.
- Para seleccionar la frecuencia de muestreo del codec se desactiva el PLL escribiendo un 0 en el bit 7 del registro 3 y un 1 en el bit 5. Este último bit configura el valor de un parámetro que se denota como *Q*, en este caso el valor es 4. El registro 2 se divide en dos bloques de cuatro bits que configuran dos parámetros usados para definir las frecuencias de muestreo de los conversores analógico-digital y digital-analógico. Estos parámetros se denotan como *NADC* y *NDAC*. En este caso, ambos son iguales a 1 al escribirse un 0 en todos los bits del registro. A la luz de lo que se muestra en la figura 20 de la hoja de datos [53], esta configuración resulta en una frecuencia de muestreo para ambos conversores de 48 KHz. Los multiplexores mostrados en la figura serán configurados en registros posteriores.
- El registro 7 puede invertir los canales o silenciarlos. En este caso, se configura para que los conversores digital-analógico reconstruyan los datos de sus respectivos canales.
- El registro 8 sirve para configurar varios aspectos del codec (consultar la tabla 16 de la hoja de datos). Con la configuración usada en este proyecto, los relojes BCLK y WCLK (bit clock y world clock) se generan en el codec (el codec trabaja como maestro).
- El registro 9 se configura para usar un tamaño de palabra de 24 bits y el modo de transmisión-recepción *DSP mode*. Además, se configura el BCLK para generarse en modo de transferencia continua. La figura 18 de la hoja de datos representa las formas de onda del modo *DSP*.

```
Uint32 Init_AIC3106()
{
    Uint8   CSR = 0x00, PLL_A = 0x20;
    //Reset
    if(!AIC3106_Write_Reg(AIC31_P0_REG0, 0 ))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG1, 0x80 ))
        return 0;
    //Fs = 48000 Hz
    if(!AIC3106_Write_Reg(AIC31_P0_REG2, CSR))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG3, PLL_A))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG7, 0x0A))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG8, 0xF0))
        return 0;
    //DSP mode, 24 bits, continuous transfer
    if(!AIC3106_Write_Reg(AIC31_P0_REG9, 0x60))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG10, 0x00))
        return 0;
    //PGAs = 0 dB
    if(!AIC3106_Write_Reg(AIC31_P0_REG15, 0x00))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG16, 0x00))
        return 0;
    //ADCs
    if(!AIC3106_Write_Reg(AIC31_P0_REG19, 0x04))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG22, 0x04))
        return 0;
    //Mic
    if(!AIC3106_Write_Reg(AIC31_P0_REG17, 0xFF))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG18, 0xFF))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG25, 0x00))
        return 0;
    //Disable AGC
    if(!AIC3106_Write_Reg(AIC31_P0_REG26, 0x00))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG29, 0x00))
        return 0;
    //Select the DAC L1 R1 Paths
    if(!AIC3106_Write_Reg(AIC31_P0_REG41, 0x02))
            return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG42, 0x6C))
           return 0;
    //DAC's power up
    if(!AIC3106_Write_Reg(AIC31_P0_REG37, 0xE0))
            return 0;
    //HPR/L configuration
    if(!AIC3106_Write_Reg(AIC31_P0_REG38, 0x10))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG43, 0x00))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG44, 0x00))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG47, 0x80))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG51, 0x09))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG58, 0x00))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG64, 0x80))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG65, 0x09))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG72, 0x00))
        return 0;
    //Route DAC_L1 to LEFT+
    if(!AIC3106_Write_Reg(AIC31_P0_REG82, 0x80))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG86, 0x09))
        return 0;

    //Route DAC_R1  to RIGHT+
    if(!AIC3106_Write_Reg(AIC31_P0_REG92, 0x80))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG93, 0x09))
        return 0;

    //CLK configuration
    if(!AIC3106_Write_Reg(AIC31_P0_REG101, 0x01))
        return 0;
    if(!AIC3106_Write_Reg(AIC31_P0_REG102, 0x02))
        return 0;
    return 1;
}
```

Listado 10.9: Definición de la función de inicialización del codec AIC3106.

- El registro 10 sirve para configurar el desplazamiento del reloj de bits (BCLK) respecto al reloj de palabra (WCLK). En este caso el desplazamiento es 0, es decir, los datos empiezan a transmitirse con el flanco de subida del reloj de palabra.
- Los registros 15 y 16 controlan la ganancia de los conversores analógico-digital y digital-analógico. Con la configuración mostrada en el código se establece la ganancia en 0 dB.
- Los registros 19 y 22 se configuran para que los conversores analógico-digital trabajen con entrada no balanceada. La ganancia de entrada se establece en 0 dB y se activa el conversor de cada canal. El PGA (Programmable Gain Amplifier) se programa para que los cambios de ganancia se produzcan en pasos de 0.5 dB cada periodo de muestreo. Consultar la hoja de datos para más detalles.
- Los registros 17, 18 y 25 se usan para desactivar la entrada de micrófono, ya que no será usada en este trabajo.
- El codec incorpora un amplificador de ganancia automática (AGC) con parámetros configurables, incluyendo el nivel de amplitud y el tiempo de ataque y liberación. Esta función es útil para otras aplicaciones pero no será usada en este proyecto, por lo que se desactiva usando los registros 26 y 29.
- Cada salida de los conversores digital-analógico tiene tres posibles caminos. Estos se configuran usando el registro 41. Para más detalles consultar el diagrama de bloques en la sección 11.2 de la hoja de datos [53].
- Los conversores digital-analógico se encienden usando el registro 37 y sus ganancias se establecen en 0 dB mediante los registros 43 y 44.
- A continuación se establecen las rutas de salida de las señales. Para más detalles consultar el bloque funcional de la sección 11.2 de la hoja de datos.
- El conversor digital-analógico izquierdo se dirige hacia HPLOUT mediante el registro 47.
- La salida analógica HPLOUT se pone a 0 dB de ganancia y se saca del silencio mediante el registro 51.
- La salida analógica HPLCOM se pone en silencio mediante el registro 58.
- El conversor digital-analógico derecho se dirige hacia HPROUT mediante el registro 64.
- La salida analógica HPROUT se pone a 0 dB de ganancia y se saca del silencio mediante el registro 65.
- La salida analógica HPRCOM se pone en silencio mediante el registro 72.
- Mediante los registros 82, 86, 92 y 93 se dirigen las salidas de los conversores digital-analógico izquierdo y derecho hacia los puertos $LEFT\_LOP$ y $RIGHT\_LOP$.
- Los registros 101 y 102 se usan para configurar los multiplexores de la figura 20 de la

```
#ifndef GPIO_DEFINES_H_
#define GPIO_DEFINES_H_
#define GPIO_BASE               0x01E26000
#define GPIO_PCR                *( volatile Uint32* )( GPIO_BASE + 0x04 )
#define GPIO_BINTEN             *( volatile Uint32* )( GPIO_BASE + 0x08 )
#define GPIO_DIR(n)             *( volatile Uint32* )(GPIO_BASE + 0x10 + ((n) * 0x28))
#define GPIO_OUT_DATA(n)        *( volatile Uint32* )(GPIO_BASE + 0x14 + ((n) * 0x28))
#define GPIO_SET_DATA(n)        *( volatile Uint32* )(GPIO_BASE + 0x18 + ((n) * 0x28))
#define GPIO_CLR_DATA(n)        *( volatile Uint32* )(GPIO_BASE + 0x1C + ((n) * 0x28))
#define GPIO_IN_DATA(n)         *( volatile Uint32* )(GPIO_BASE + 0x20 + ((n) * 0x28))
#define GPIO_SET_RIS_TRIG(n)    *( volatile Uint32* )(GPIO_BASE + 0x24 + ((n) * 0x28))
#define GPIO_CLR_RIS_TRIG(n)    *( volatile Uint32* )(GPIO_BASE + 0x28 + ((n) * 0x28))
#define GPIO_SET_FAL_TRIG(n)    *( volatile Uint32* )(GPIO_BASE + 0x2C + ((n) * 0x28))
#define GPIO_CLR_FAL_TRIG(n)    *( volatile Uint32* )(GPIO_BASE + 0x30 + ((n) * 0x28))
#define GPIO_INTSTAT(n)         *( volatile Uint32* )(GPIO_BASE + 0x34 + ((n) * 0x28))
enum {USER_LED1=1, USER_LED2 = 2, USER_LED3=4, USER_LED4 = 8};
#endif /* GPIO_DEFINES_H_ */
```

Listado 10.10: Macros para los registros del módulo GPIO.

```
#ifndef GPIO_H_
#define GPIO_H_
#include "GPIO_defines.h"
#include <ti/csl/tistdtypes.h>
void Init_GPIO_LEDS();
void Write_LEDS(Uint8 led_bits);
#endif /* GPIO_H_ */
```

Listado 10.11: Cabecera GPIO.h.

hoja de datos. En concreto, se hace que *CODEC_CLK* sea igual a *CLKDIV_OUT* y que *CLKDIV_IN* sea igual a MCLK. Recordemos que el pin MCLK está conectado a un oscilador externo de 24.576 Mhz.

10.6 Módulo GPIO

La placa de desarrollo tiene cuatro LEDs que pueden ser usados mediante el puerto GPIO. El pin 13 del banco 6 controla el LED D4. El pin 12 del banco 6 controla el LED D5. El pin 12 del banco 2 controla el LED D6. El pin 9 del banco 0 controla el LED D7.

En el listado 10.10 se muestra el contenido del archivo *GPIO_defines.h*. Cada grupo de registros GPIO controla dos bancos, comos se dijo anteriormente, y tiene un tamaño total de 40 bytes. Por tanto, para acceder a un cierto grupo de registros n solo hay que multiplicar el tamaño del grupo de registros por n y sumar el resultado a la dirección de memoria de los registros del primer grupo.

Por otro lado, se define una enumeración en la que el LED D4 se denota como *USER_LED*1, el LED D5 se denota como *USER_LED*2, el LED D6 se denota como *USER_LED*3 y el LED D7 se denota como *USER_LED*4. A cada led se hace corresponder uno de los 4 bits más significativos de un entero.

En el listado 10.11 se muestra el contenido de la cabecera *GPIO.h*. La función *Init_GPIO_-LEDS()* se usa para inicializar los bancos GPIO. La función *Write_LEDS(...)* toma un

```
#include "GPIO.h"
void Init_GPIO_LEDS()
{
    GPIO_CLR_DATA(0) = 0x00000200;
    GPIO_DIR(0) &= ~0x00000200;
    GPIO_CLR_DATA(1) = 0x00001000;
    GPIO_DIR(1) &= ~0x00001000;
    GPIO_CLR_DATA(3) = 0x00003000;
    GPIO_DIR(3) &= ~0x00003000;
}
void Write_LEDS(Uint8 led_bits)
{
    if(led_bits & USER_LED1)
        GPIO_SET_DATA(3) = 0x00002000;
    else
        GPIO_CLR_DATA(3) = 0x00002000;
    if(led_bits & USER_LED2)
        GPIO_SET_DATA(3) = 0x00001000;
    else
        GPIO_CLR_DATA(3) = 0x00001000;
    if(led_bits & USER_LED3)
        GPIO_SET_DATA(1) = 0x00001000;
    else
        GPIO_CLR_DATA(1) = 0x00001000;
    if(led_bits & USER_LED4)
        GPIO_SET_DATA(0) = 0x00000200;
    else
        GPIO_CLR_DATA(0) = 0x00000200;
}
```

Listado 10.12: Contenido del archivo GPIO.c.

entero sin signo de 8 bits. Cada uno de los 4 bits menos significativos se corresponde con el estado de uno de los LED.

En el listado 10.12 se muestra el contenido del archivo *GPIO.c*. La función de inicialización configura los pines GPIO usados para controlar los LED como salida. La función de escritura comprueba cada uno de los bits del parámetro tomado y pone el pin correspondiente en estado alto o bajo usando lógica positiva.

10.7 Módulo McASP

En el listado 10.13 se muestra el contenido del archivo *McASP_defines.h*, en el que se encuentra la definición de una estructura de datos y de varias macros para el acceso a los registros de configuración del módulo McASP. En el listado se han omitido algunos registros de la estructura por cuestiones de espacio. Una vez más, los registros están situados de forma consecutiva en el mapa de memoria por lo que un puntero a la estructura definida situado en la dirección base dará un fácil acceso a todos los registros.

En el listado 10.14 se muestra el contenido del archivo *McASP.c*. Como puede verse el archivo contiene una sola función, por lo que se ha omitido el archivo de cabecera *McASP.h*. La función de inicialización es una aplicación de lo explicado en la sección 9.3.2. El módulo McASP trabaja en modo TDM con un tamaño de *slot* de 24 bits y un tamaño de *frame* de 2 *slots*. Esto hace el formato compatible con el modo DSP del codec cuando el tamaño de palabra es de 24 bits. Adicionalmente, he comentado el código para ayudar a

```
#ifndef MCASP_DEFINES_H_
#define MCASP_DEFINES_H_
#include <ti/csl/tistdtypes.h>
typedef struct {
    volatile Uint32 REVID;
    volatile Uint32 RSVD0[3];
    volatile Uint32 PFUNC;
    volatile Uint32 PDIR;
    volatile Uint32 PDOUT;
    volatile Uint32 PDIN;
    volatile Uint32 PDCLR;
    volatile Uint32 RSVD1[8];
    volatile Uint32 GBLCTL;
    volatile Uint32 AMUTE;
    volatile Uint32 DLBCTL;
    volatile Uint32 DITCTL;
    volatile Uint32 RSVD2[3];
    volatile Uint32 RGBLCTL;
    volatile Uint32 RMASK;
    volatile Uint32 RFMT;
    volatile Uint32 AFSRCTL;
    volatile Uint32 ACLKRCTL;
    volatile Uint32 AHCLKRCTL;
    volatile Uint32 RTDM;
    volatile Uint32 RINTCTL;
    volatile Uint32 RSTAT;
    volatile Uint32 RSLOT;
    volatile Uint32 RCLKCHK;
    volatile Uint32 REVTCTL;
    volatile Uint32 RSVD3[4];
    volatile Uint32 XGBLCTL;
    volatile Uint32 XMASK;
    volatile Uint32 XFMT;
    volatile Uint32 AFSXCTL;
    volatile Uint32 ACLKXCTL;
    volatile Uint32 AHCLKXCTL;
    volatile Uint32 XTDM;
    volatile Uint32 XINTCTL;
    volatile Uint32 XSTAT;
    volatile Uint32 XSLOT;
    volatile Uint32 XCLKCHK;
    volatile Uint32 XEVTCTL;
    volatile Uint32 RSVD4[12];
    //Se han omitido registros
} McaspRegs;
#define MCASP_0     ((McaspRegs *)0x01D00000)
#define MCASP_1     ((McaspRegs *)0x01D04000)
#define GBLCTL_XFRST            0x1000
#define GBLCTL_XSMRST           0x0800
#define GBLCTL_XSRCLR           0x0400
#define GBLCTL_XHCLKRST         0x0200
#define GBLCTL_XCLKRST          0x0100
#define GBLCTL_RFRST            0x0010
#define GBLCTL_RSMRST           0x0008
#define GBLCTL_RSRCLR           0x0004
#define GBLCTL_RHCLKRST         0x0002
#define GBLCTL_RCLKRST          0x0001
#endif /* MCASP_DEFINES_H_ */
```

Listado 10.13: Macros y estructura para los registros del módulo McASP.

```
#include "McASP.h"
void Init_McASP0()
{
    McaspRegs* mcasp = MCASP_0;
    //Reset
    mcasp->GBLCTL     = 0;
    while(!(mcasp->GBLCTL == 0));
    //receiver
    mcasp->RMASK      = 0x00FFFFFF;
    mcasp->RFMT       = 0x000080B8;     //24 bits slot size, bit reverse, no rotation, (mask after)
    mcasp->AFSRCTL    = 0x00000100;     //rising edge, external, 1 bit width, 2-slot TDM
    mcasp->ACLKRCTL   = 0x00000000;     //tx side
    mcasp->AHCLKRCTL  = 0x00000000;     //tx side
    mcasp->RTDM       = 0x00000003;     //serializer shifts in data during TDM slots 1 and 2
    mcasp->RINTCTL    = 0x00000001;     //enable overrun interrupt
    mcasp->RCLKCHK    = 0x00FF0008;     //SYSCLK2/256 ticks within 0-255 for 32 AHCLKR ticks
    //transmitter
    mcasp->XMASK      = 0x00FFFFFF;
    mcasp->XFMT       = 0x000080BE;     //24 bits slot size, (mask before), 24 bit rotation, bit reverse, 1
          bit delay
    mcasp->AFSXCTL    = 0x00000100;     //rising edge, external, 1 bit width, 2-slot TDM
    mcasp->ACLKXCTL   = 0x00000000;     //Transmitter clocks provides the source for both transmitter and
         receiver
    mcasp->AHCLKXCTL  = 0x00000000;     //external
    mcasp->XTDM       = 0x00000003;     //serializer shifts out data during TDM slots 1 and 2
    mcasp->XINTCTL    = 0x00000020;     //data interrupts enabled (irrelevant for DMA)
    mcasp->XCLKCHK    = 0x00FF0008;     //SYSCLK2/256 ticks within 0-255 for 32 AHCLKR ticks
    mcasp->SRCTL13    = 0x0000000D;     //transmitter, high logic
    mcasp->SRCTL14    = 0x0000000E;     //receiver, high logic
    mcasp->PFUNC      = 0x00000000;     //all pins used for mcasp
    mcasp->PDIR       = 0x00002000;     //all pins are inputs except axr13
    mcasp->DITCTL     = 0x00000000;     //digital mode control not used
    mcasp->DLBCTL     = 0x00000000;
    mcasp->AMUTE      = 0x00000000;     //amute pin disabled
    //start clocks
    mcasp->GBLCTL |= 0x0202;
    while(!(mcasp->GBLCTL & 0x0202));
    mcasp->GBLCTL |= 0x0101;
    while(!(mcasp->GBLCTL & 0x0101));
    //clear errors
    mcasp->XSTAT = 0x0000FFFF;
    mcasp->RSTAT = 0x0000FFFF;
    //clear serializers
    mcasp->GBLCTL |= 0x0404;
    while(!(mcasp->GBLCTL & 0x0404));
    //start state machines
    mcasp->GBLCTL |= 0x0808;
    while(!(mcasp->GBLCTL & 0x0808));
    //prevent underrrun
    mcasp->XBUF13 = 0;
    //start frame syncs
    mcasp->GBLCTL |= 0x1010;
    while(!(mcasp->GBLCTL & 0x1010));
}
```

Listado 10.14: Contenido del archivo McASP.c.

```
1  #ifndef EDMA_DEFINES_H_
2  #define EDMA_DEFINES_H_
3  #include <ti/csl/tistdtypes.h>
4
5  #define EDMA3_EVENT_MCASP0_RX    0
6  #define EDMA3_EVENT_MCASP0_TX    1
7
8  typedef struct {
9      volatile unsigned int          option;
10     volatile Uint32          srcAddr;
11     volatile Uint32          aCntbCnt;
12     volatile Uint32          dstAddr;
13     volatile Uint32          srcDstBidx;
14     volatile Uint32          linkBcntrld;
15     volatile Uint32          srcDstCidx;
16     volatile Uint32          cCnt;
17 } EDMA3_Param;
18
19 #define EDMA3_0_PARAM_BASE      0x01C04000
20 #define EDMA3_0_PARAM_OFFSET    0x20
21 #define EDMA3_0_PARAM(x)        (EDMA3_0_PARAM_BASE + (x * EDMA3_0_PARAM_OFFSET))
22
23 #define EDMA3_0_CC_BASE         0x01C00000
24 #define EDMA3_0_CC_ECR          (EDMA3_0_CC_BASE + 0x1008)
25 #define EDMA3_0_CC_EESR         (EDMA3_0_CC_BASE + 0x1030)
26 #define EDMA3_0_CC_DRAE1        (EDMA3_0_CC_BASE + 0x0348)
27 #define EDMA3_0_CC_IESR         (EDMA3_0_CC_BASE + 0x1060)
28 #define EDMA3_0_CC_ICR          (EDMA3_0_CC_BASE + 0x1070)
29 #define EDMA3_0_CC_IPR          (EDMA3_0_CC_BASE + 0x1068)
30
31 #endif /* EDMA_DEFINES_H_ */
```

Listado 10.15: Contenido del archivo *EDMA_defines.h*.

entender lo que sucede en cada linea.

La parte más complicada de entender es el formateo de los bits que entran o salen de los buffers de los serializadores. Recomiendo al lector observar las figuras 19 y 20 de la documentación del módulo McASP [50]. En ellas puede verse como los registros RRVRS, RROT, RMASK, RPBIT y RPAD son aplicados en la transmisión y la recepción de datos. Las figuras 29 y 30 sirven como guía para configurar estos registros en función de la representación interna de los datos en el DSP y del formato de salida que se necesite. Los bits enviados por el codec son una representación en complemento a dos, en el caso de este trabajo con una longitud de palabra de 24 bits. Dado que el bit más significativo llega primero al serializador se configura la unidad de formato de recepción para que invierta los bits. Igualmente, dado que en el código posterior los bits a transmitir se introducirán en el buffer del serializador de transmisión alineados a la derecha y que el codec espera recibir el bit más significativo primero es necesario hacer una rotación de 24 posiciones hacia la derecha e invertir los bits, tal y como puede verse en la figura 29 de la documentación.

10.8 Módulo EDMA3

En el listado 10.15 se muestra el contenido del archivo *EDMA_defines.h*. Este archivo contiene la definición de la estructura de datos usada para acceder a los registros de configuración de los PaRAM, la dirección del mapa de memoria en la que se encuentra el primer PaRAM y el desplazamiento para acceder a los PaRAM sucesivos, es decir, el

```
1  #include "EDMA.h"
2
3  void Init_Interrupts()
4  {
5      ISTP = 0x11800000;
6  }
7
8  void Enable_Interrupts()
9  {
10     IER |= 0x0102;
11     ICR = 0xffff;
12     CSR |= 1;
13 }
```

Listado 10.16: Contenido del archivo EDMA.c.

tamaño en bytes de la estructura que almacena los registros para configurar un PaRAM. Con estos datos puede accederse a cualquier PaRAM usando la macro *EDMA_0_PARAM(x)*.

Además, se incluyen al comienzo del archivo las macros que definen los eventos provenientes del módulo McASP que pueden potencialmente sincronizar una transferencia EDMA3. Estos eventos están asociados a un determinado número de PaRAM. En este trabajo se usarán los eventos de recepción y transmisión del módulo McASP, asociados respectivamente a los PaRAM 0 y 1, tal y como se menciona en la sección 6.9.1 de la hoja de datos del procesador [47].

El módulo McASP generará un evento cada vez que se termine de transmitir (recibir) un *slot*. El módulo EDMA3 recibirá este evento y activará la transferencia (recepción) de un bloque A hacia (desde) el buffer del serializador de transmisión (recepción) desde (hacia) la dirección de una variable de memoria. Este proceso resultará más claro cuando se configuren los PaRAM en un fragmento de código posterior.

Es necesario distinguir claramente entre los eventos enviados por el módulo McASP al módulo EDMA3 y las interrupciones generadas por el módulo EDMA3 cuando se completa la transferencia de todos los bloques. Son cosas totalmente distintas. Los eventos del módulo McASP sirven para sincronizar las transferencias EDMA3 mientras que las interrupciones generadas por el módulo EDMA3 son interrupciones del procesador.

Finalmente, se incluyen en el mismo archivo macros para acceder a otros registros de configuración del módulo.

En el listado 10.16 se muestra el contenido del archivo *EDMA.c*. Una vez más se omite el archivo de cabecera, ya que este solo contiene los prototipos de las funciones presentes en el archivo de implementación. El propósito de estas funciones es configurar varios registros relacionados con las interrupciones que serán generadas por el módulo EDMA3.

La función *Init_Interrupts()* configura el registro ISTP. Como se mencionó anteriormente, este registro contiene la dirección de la tabla de interrupciones.

La función *Enable_Interrupts()* configura varios registros:

- El IER (Interrupt Enabled Register) indica que interrupciones están enmascaradas. En el caso de este trabajo es necesario dejar activada la interrupción de transferencia completa del controlador de canal EDMA3. De esta forma podrá crearse posteriormente una rutina que tome las acciones oportunas cuando termine una transferencia directa a memoria. En la documentación técnica de referencia [52], tabla 18-8, se indica que esta interrupción es la número 8 en el módulo de interrupciones del DSP.
- El ICR (Interrupt Clear Register) permite limpiar manualmente las interrupciones activas en el registro IFR (Interrupt Flag Register), que indica cuando se ha producido una interrupción.
- El CDR (Control Status Register) contiene bits de control y estado. El bit menos significativo es el GIE (Global Interrupt Enabled) y es necesario activarlo para habilitar las interrupciones enmascarables.

Nótese que estos registros no son accesibles mediante el mapa de memoria. Se trata de registros especiales definidos en la cabecera *c6x.h*.

10.9 Tabla de interrupciones

En el listado 10.17 puede verse el contenido del archivo *vectors.asm*, en él que se implementan las rutinas asociadas a cada interrupción. Cada una de las rutinas debe tener un tamaño total de 32 bytes para que los desplazamientos que usa el sistema de interrupciones coincidan.

La directiva *.sect* se usa para indicar al compilador en que sección de las definidas en el archivo CMD debe ubicarse la tabla. La directiva *.ref* se usa para hacer referencia a un símbolo definido en otro archivo. En este caso se hace referencia a dos símbolos: Al punto de entrada de la aplicación, *_c_int00*, y a una función llamada *Audio_ISR* que se definirá en código posterior. Esta función será la encargada de realizar las acciones necesarias cada vez que el módulo EDMA3 genere una interrupción de transferencia completa.

La primera interrupción es la interrupción de reseteo. Todo lo que se hace en ella es cargar la dirección del punto de entrada en el registro B0 y saltar a dicha dirección mediante la instrucción B de salto no condicionado. Nótese que tanto en las instrucciones de carga como en la de salto se indica la unidad funcional en la que deben ejecutarse, en este caso la unidad funcional .S2.

En la interrupción 8, asociada a una transferencia EDMA3 completada, se realiza un salto no condicionado a la función *Audio_ISR*.

```
1       .ref    Audio_ISR           ;audio rutine
2       .ref    _c_int00            ;entry point
3       .sect   "vectors"           ;section name
4           .nocmp                  ;use 32 bit instructions so everything remains aligned
5   RESET_RST:
6       MVKL .S2 _c_int00, B0   ;return to entry point at reset
7       MVKH .S2 _c_int00, B0
8       B    .S2 B0
9           NOP
10          NOP
11          NOP
12          NOP
13          NOP
14  NMI_RST: b NMI_RST
15          NOP
16          NOP
17          NOP
18          NOP
19          NOP
20          NOP
21          NOP
22  RESV1:  b RESV1
23          NOP
24          NOP
25          NOP
26          NOP
27          NOP
28          NOP
29          NOP
30  RESV2:  b RESV2
31          NOP
32          NOP
33          NOP
34          NOP
35          NOP
36          NOP
37          NOP
38  INT4:   b INT4
39          NOP
40          NOP
41          NOP
42          NOP
43          NOP
44          NOP
45          NOP
46  INT5:   b INT5
47          NOP
48          NOP
49          NOP
50          NOP
51          NOP
52          NOP
53          NOP
54  INT6:   b INT6
55          NOP
56          NOP
57          NOP
58          NOP
59          NOP
60          NOP
61          NOP
62  INT7:   b INT7
63          NOP
64          NOP
65          NOP
66          NOP
67          NOP
68          NOP
69          NOP
70  INT8:   b Audio_ISR     ;branch to audio rutine on EDMA3 transfer complete interruption
71          NOP
72          NOP
73          NOP
74          NOP
75          NOP
76          NOP
77          NOP
78  INT9:   b INT9
79          NOP
80          NOP
81          NOP
82          NOP
83          NOP
84          NOP
85          NOP
86  //Se han omitido interrupciones
```

Listado 10.17: Contenido del archivo vectors.asm.

```
1  #ifndef BUFFER_PROC_H_
2  #define BUFFER_PROC_H_
3  void Zero_Buffers();
4  void Process_Buffer();
5  int IsBufferReady();
6  void EDMA3_Config_Params();
7  void EDMA3_Config_Events();
8  #endif /* BUFFER_PROC_H_ */
9  }
```

Listado 10.18: Contenido del archivo *buffer_proc.h*.

En el resto de interrupciones se realiza un salto no condicionado a la etiqueta de la propia interrupción, entrando así el programa en un bucle infinito. La aplicación no hace uso de estas interrupciones por lo que en teoría no deberían suceder nunca. Si se produce un error que genera una interrupción no prevista el bucle infinito en el que la aplicación se queda atrapada hace más sencilla la depuración.

10.10 Procesamiento de buffers

El propósito de estos bloques de código es manejar los buffers y las transferencias EDMA3.

En el listado 10.18 se muestra el contenido del archivo *buffer_proc.h*, en el que se declaran varias funciones. El propósito de estas funciones resultará claro cuando se explique el archivo de implementación.

En el listado 10.19 se muestra la primera parte del código del archivo *buffer_proc.c*. En el listado 10.20 se muestra la segunda parte.

A partir de la linea 11 se definen macros para los parámetros del buffer. Estas incluyen el tamaño de cada uno de los buffers en número de muestras, la longitud total del buffer y el número de buffers que serán usados. Nótese que la longitud total del buffer será igual al número de muestaras multiplicado por dos, ya que un solo buffer recibe muestras de ambos canales de audio.

A partir de la linea 16 se definen varias macros para los parámetros de transmisión y recepción de los PaRAM. En cuanto a la transmisión:

- En el campo de opciones se activa el bit TCINTEN (El número 20), que habilita la generación de interrupciones cuando una transferencia finaliza. En el campo TCC (bits 12 a 17) se escribe un 1. Este código de 6 bits será usado para habilitar el bit de la interrupción en el IPR (Interrupt Pending Register).
- En los campos BCNT y ACNT se escribe la longitud de los bloques B (La longitud total de los buffers) y la longitud de los bloques A (El tamaño en bytes de una muestra de un buffer). En este caso, cada muestra de un buffer es un float de 4 bytes.

```
#include <ti/csl/tistdtypes.h>
#include "GPIO.h"
#include "McASP.h"
#include "EDMA_defines.h"
#include "buffer_proc.h"
#include "EffectVector.h"
#include <stdio.h>
#include "SineLFO.h"
#include <float.h>

//Buffer
#define BUFFER_COUNT        256
#define BUFFER_LENGTH       BUFFER_COUNT*2
#define NUM_BUFFERS         3

//Parametros de transmision
#define PARAM_TX_OPTION               0x00101000
#define PARAM_TX_AB_COUNT             ((BUFFER_LENGTH << 16) + 4)
#define PARAM_TX_DEST_ADDR            ((Uint32)(&(MCASP_0->XBUF13)))
#define PARAM_TX_SRC_DEST_B_INDEX     ((0 << 16) + 4)

//Parametros de recepcion
#define PARAM_RX_OPTION               0x00100000
#define PARAM_RX_AB_COUNT             ((BUFFER_LENGTH << 16) + 4)
#define PARAM_RX_SRC_ADDR            ((Uint32)(&(MCASP_0->RBUF14)))
//#define PARAM_RX_SRC_ADDR             ((Uint32)(&(MCASP_0->RBUF12)))     //loopback
#define PARAM_RX_SRC_DEST_B_INDEX     ((4 << 16) + 0)

#pragma DATA_SECTION (buffer, "SDRAM");
Int32 buffer[NUM_BUFFERS][BUFFER_LENGTH];
volatile Int16 buffer_ready = 0, ready_index = 0;

void Zero_Buffers()
{
    Int32 i = BUFFER_LENGTH * NUM_BUFFERS;
    Int32 *p = (Int32 *)buffer;
    while(i--)
        *p++ = 0;
}

void Process_Buffer()
{
    const float Q = 1.0f / 0x80000000;  //1/(2^31)
    Int32 *pBuf = buffer[ready_index];
    static float Left[BUFFER_COUNT];
    float *pL = Left;
    Int32 i;

    Write_LEDS(0);

    //Read only left channel, shift to put sign bit in place, normalize to the interval [-1, 1]
    for(i = 0;i < BUFFER_COUNT;i++) {
        *pBuf++; *pL++ = ((Int32)(*pBuf++ << 8)) * Q;
    }

    pL = Left;
    pBuf = buffer[ready_index];

    //Process data using effect vector
    ProcessBuffer256(Left);

    for(i = 0;i < BUFFER_COUNT;i++) {
            if(*pL >= 1) *pL = 1.0 - FLT_EPSILON;  else if(*pL < -1) *pL = -1;
            *pBuf++ =  (_spint(*pL * (1 << 23)) & 0x00FFFFFF);
            *pBuf++ =  (_spint(*pL * (1 << 23)) & 0x00FFFFFF);
            pL++;
    }

    Write_LEDS(1);
    buffer_ready = 0;
}

int IsBufferReady()
{
    return buffer_ready;
}

interrupt void Audio_ISR()
{
    *(volatile Uint32 *)EDMA3_0_CC_ICR = 1;
    if(++ready_index >= NUM_BUFFERS)
        ready_index = 0;
    buffer_ready = 1;
}
```

Listado 10.19: Contenido del archivo *buffer_proc.c*.

```
85  void EDMA3_Config_Params()
86  {
87      EDMA3_Param* param;
88
89      //PaRAM encadenados para transmision de muestras al puerto McASP
90      param = (EDMA3_Param*)EDMA3_0_PARAM(EDMA3_EVENT_MCASP0_TX);
91      param->option = PARAM_TX_OPTION;
92      param->srcAddr = (Uint32)(&buffer[2][0]);
93      //param->srcAddr = (Uint32)(&aux[0]); //loopback
94      param->aCntbCnt = PARAM_TX_AB_COUNT;
95      param->dstAddr = PARAM_TX_DEST_ADDR;
96      param->srcDstBidx = PARAM_TX_SRC_DEST_B_INDEX ;
97      param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(64) & 0xFFFF);
98      param->cCnt = 1;
99
100     param = (EDMA3_Param*)EDMA3_0_PARAM(64);
101     param->option = PARAM_TX_OPTION;
102     param->srcAddr = (Uint32)(&buffer[0][0]);
103     //param->srcAddr = (Uint32)(&aux[0]); //loopback
104     param->aCntbCnt = PARAM_TX_AB_COUNT;
105     param->dstAddr = PARAM_TX_DEST_ADDR;
106     param->srcDstBidx = PARAM_TX_SRC_DEST_B_INDEX ;
107     param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(65) & 0xFFFF);
108     param->cCnt = 1;
109
110     param = (EDMA3_Param*)EDMA3_0_PARAM(65);
111     param->option = PARAM_TX_OPTION;
112     param->srcAddr = (Uint32)(&buffer[1][0]);
113     //param->srcAddr = (Uint32)(&aux[0]); //loopback
114     param->aCntbCnt = PARAM_TX_AB_COUNT;
115     param->dstAddr = PARAM_TX_DEST_ADDR;
116     param->srcDstBidx = PARAM_TX_SRC_DEST_B_INDEX ;
117     param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(66) & 0xFFFF);
118     param->cCnt = 1;
119
120     param = (EDMA3_Param*)EDMA3_0_PARAM(66);
121     param->option = PARAM_TX_OPTION;
122     param->srcAddr = (Uint32)(&buffer[2][0]);
123     //param->srcAddr = (Uint32)(&aux[0]); //loopback
124     param->aCntbCnt = PARAM_TX_AB_COUNT;
125     param->dstAddr = PARAM_TX_DEST_ADDR;
126     param->srcDstBidx = PARAM_TX_SRC_DEST_B_INDEX ;
127     param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(64) & 0xFFFF);
128     param->cCnt = 1;
129
130     //PaRAM encadenados para transmision de muestras desde el puerto McASP
131     param = (EDMA3_Param*)(EDMA3_0_PARAM(EDMA3_EVENT_MCASP0_RX));
132     param->option = PARAM_RX_OPTION;
133     param->srcAddr = PARAM_RX_SRC_ADDR;
134     param->aCntbCnt = PARAM_RX_AB_COUNT;
135     param->dstAddr = (Uint32)(&buffer[1][0]);
136     param->srcDstBidx = PARAM_RX_SRC_DEST_B_INDEX;
137     param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(67) & 0xFFFF);
138     param->cCnt = 1;
139
140     param = (EDMA3_Param*)EDMA3_0_PARAM(67);
141     param->option = PARAM_RX_OPTION;
142     param->srcAddr = PARAM_RX_SRC_ADDR;
143     param->aCntbCnt = PARAM_RX_AB_COUNT;
144     param->dstAddr = (Uint32)(&buffer[2][0]);
145     param->srcDstBidx = PARAM_RX_SRC_DEST_B_INDEX;
146     param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(68) & 0xFFFF);
147     param->cCnt = 1;
148
149     param = (EDMA3_Param*)EDMA3_0_PARAM(68);
150     param->option = PARAM_RX_OPTION;
151     param->srcAddr = PARAM_RX_SRC_ADDR;
152     param->aCntbCnt = PARAM_RX_AB_COUNT;
153     param->dstAddr = (Uint32)(&buffer[0][0]);
154     param->srcDstBidx = PARAM_RX_SRC_DEST_B_INDEX;
155     param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(69) & 0xFFFF);
156     param->cCnt = 1;
157
158     param = (EDMA3_Param*)EDMA3_0_PARAM(69);
159     param->option = PARAM_RX_OPTION;
160     param->srcAddr = PARAM_RX_SRC_ADDR;
161     param->aCntbCnt = PARAM_RX_AB_COUNT;
162     param->dstAddr = (Uint32)(&buffer[1][0]);
163     param->srcDstBidx = PARAM_RX_SRC_DEST_B_INDEX;
164     param->linkBcntrld = (BUFFER_LENGTH << 16) + (EDMA3_0_PARAM(67) & 0xFFFF);
165     param->cCnt = 1;
166 }
167
168 void EDMA3_Config_Events()
169 {
170     *(volatile Uint32 *)EDMA3_0_CC_ECR  = 3;
171     *(volatile Uint32 *)EDMA3_0_CC_EESR = 3;
172     *(volatile Uint32 *)EDMA3_0_CC_DRAE1 = 3;
173     *(volatile Uint32 *)EDMA3_0_CC_IESR = 1;
174 }
```

Listado 10.20: Contenido del archivo *buffer_proc.c*.

- En el campo DST se escribe la dirección hacia la que se transfieren los datos, en este caso la dirección del buffer del serializador 13 del módulo McASP. Recordemos que este serializador es usado para transmitir muestras al codec.
- En los campos DSTBIDX y SRCBIDX se escriben los incrementos de la dirección de destino y de origen cada vez que se finaliza la transferencia de un bloque A. El destino no debe moverse en absoluto ya que la siguiente muestra se escribe igualmente en el buffer del serializador. La dirección de origen debe incrementarse 4 bytes para que quede sobre la siguiente muestra a enviar.

En cuanto a la recepción, la configuración es análoga:

- Los bits del campo de opciones indican que no escribe ningún bit en el registro IPR cuando la transferencia finaliza.
- En los campos BCNT y ACNT se escribe la longitud de los bloques B (La longitud total de los buffers) y la longitud de los bloques A (El tamaño en bytes de una muestra de un buffer). En este caso, cada muestra de un buffer es un float de 4 bytes.
- En el campo SRC se escribe la dirección desde la que se reciben los datos, en este caso la dirección del buffer del serializador 14 del módulo McASP. Recordemos que este serializador es usado para recibir muestras del codec.
- En los campos DSTBIDX y SRCBIDX se escriben los incrementos de la dirección de destino y de origen cada vez que se finaliza la transferencia de un bloque A. El origen no debe moverse en absoluto ya que la siguiente muestra se recibe igualmente del buffer del serializador. La dirección de destino debe incrementarse 4 bytes para que quede sobre la siguiente posición a escribir con la muestra recibida.

En la linea 29 se declaran los buffers en forma de array bidimensional. La directiva del compilador *DATA_SECTION* permite especificar la sección de una variable. Recordemos que la sección SDRAM se asocio en el archivo CMD con la región de memoria correspondiente a la memoria RAM externa. Por tanto, para realizar el procesamiento será necesario en primer lugar traer los datos desde la memoria RAM, a la que serán transferidos por DMA mediante el módulo EDMA3.

La función *Zero_Buffers()* recorre los arrays escribiendo un 0 en todas las posiciones. En las siguientes secciones se explican el resto de funciones definidas en el archivo.

10.10.1 Función Process_Buffer()

Esta función será llamada cada vez que un buffer esté listo para el procesamiento. Existe un entero *ready_index* que indica el número de buffer que está listo para el procesamiento e irá rotando cada vez que se llame a la rutina de interrupción, como se verá a continuación.

En la linea 52 se copia el buffer a un array declarado en el interior de la función. Al haberse indicado en el archivo CMD que todas las secciones de código están situadas en la memoria interna L2 esta operación supone una transferencia del buffer desde la memoria RAM hacia la memoria interna del chip. Nótese que por cada copia al array interno el puntero al array en la memoria RAM se incrementa dos veces. Como se mencionó anteriormente, este array almacena en posiciones alternativas muestras del canal izquierdo y el canal derecho. En este trabajo solo se procesan las muestras del canal izquierdo, mientras que las otras son descartadas.

Para realizar el procesamiento en coma flotante cada una de las muestras es normalizada y convertida a un float en el rango $[-1, 1 - FLT\_EPSILON]$, donde $FLT\_EPSILON$ es el float más pequeño mayor que 0. Para ello, primero se desplazan los 24 bits de la muestra de entrada 8 bits hacia la izquierda. Esto sitúa el bit de signo en su lugar. Posteriormente, se divide el entero resultante entre el mayor entero positivo representable en complemento a dos haciendo uso de 32 bits, es decir, 2^{31}.

Puede comprobarse que fácilmente que recibiendo enteros de 24 bits representados en complemento a dos el menor float que es posible obtener tras el proceso anterior es -1.0, mientras que el mayor es $1 - FLT\_EPSILON$.

En la linea 60 se procesan las muestras normalizadas haciendo uso de una función cuyo código será mostrado en el capítulo siguiente.

Tras el procesamiento es necesario convertir de nuevo las muestras a enteros de 24 bits, redondeando el resultado de ser necesario. En primer lugar es una buena idea comprobar que las muestras resultantes se encuentran dentro del rango disponible, que puede haber sido excedido tras el procesamiento.

Para convertir las muestras entre -1 y $1 - FLT\_EPSILON$ a 24 bits simplemente se multiplican por el mayor entero representable con 24 bits en complemento a dos, es decir, 2^{23}. El resultado debe ser redondeado al entero más próximo. Para ello, se usa la función intrínseca del compilador *_spint*. Adicionalmente, se enmascaran los 8 bits más significativos. Realmente no es necesario hacer esto ya que la unidad de formato de transmisión del módulo McASP ha sido configurada para enmascarar estos bits antes de la transmisión.

Finalmente, se indica que el procesamiento ha finalizado poniendo a 0 el entero $buffer\_ready$.

Nótese que antes de empezar la transferencia de las muestras desde la memoria RAM y realizar el procesamiento se apaga un LED, que es encendido de nuevo al terminar el procesamiento. Esto permite medir con un osciloscopio el tiempo empleado para procesar

cada buffer, cosa que será hecha en un capítulo posterior.

10.10.2 Función Audio_ISR()

Esta función es llamada por la rutina de interrupción cada vez que se completa una transferencia EDMA3, es decir, cada vez que se transmiten todos los bloques definidos en un PaRAM, o lo que es lo mismo en este caso, cada vez que un buffer está lleno. Como puede verse la rutina es muy corta, lo único que hace es limpiar el registro IPR mediante el registro ICR (Interrupt Clear Register) e incrementar el marcador *ready_index*, que como se mencionó anteriormente indica el número de buffer listo para el procesamiento. Finalmente, se indica que hay un buffer listo para procesar mediante el entero $buffer\_ready$.

10.10.3 Función EDMA3_Config_Params()

Para entender esta función es necesario recordar que el PaRAM 0 está asociado al evento de transmisión del módulo McASP. Es decir, cada vez que el módulo McASP envía un *slot* genera un evento que sirve para sincronizar esta transferencia. El módulo EDMA3 recibirá el evento y transferirá un nuevo bloque al buffer del serializador de transmisión. La linea 95 escribe el campo SRC, que indica la dirección desde la que empiezan a transmitirse las muestras, en este caso la dirección de la primera muestra del buffer 2. Recordemos que cada vez que la transferencia es sincronizada esta dirección se incrementa lo indicado en el campo SRCBIDX, en este caso 4 bytes, apuntando a la siguiente muestra.

¿Que ocurre cuando el buffer se queda sin muestras? Para empezar, el módulo EDMA3 genera una interrupción de transferencia completa, al haber llegado el campo BCNT a 0. Recordemos que BCNT se decrementa cada vez que termina la transferencia de un bloque A.

A continuación se recarga el campo BCNT con el número indicado en el campo BCNTRLD, configurado en la linea 100. En este caso se carga de nuevo el tamaño del buffer.

Posteriormente se cargan el resto de parámetros desde el PaRAM situado en la dirección indicada en el campo LINK, configurado también en la linea 100. Este campo contiene la dirección del PaRAM 64, cuyo campo SRC contiene la dirección del buffer 0.

Cuando este último buffer también se agota se carga el PaRAM 65, cuyo campo SRC contiene la dirección del buffer 1. Este proceso continua hasta que finalmente el PaRAM 66 hace que las muestras provengan nuevamente del buffer 2. Cuando este se agota, se carga nuevamente el PaRAM 64. Esto da lugar a un bucle infinito en el que los buffers son constantemente rotados. Es importante resaltar que el PaRAM usado para la transferencia

es siempre el mismo, lo que sucede es que los datos de otros PaRAM son automáticamente transferidos al PaRAM 0 al finalizar la transferencia.

El proceso para los PaRAM de transmisión de muestras desde el puerto McASP es análogo, con la excepción de que ahora son los eventos que genera el módulo McASP al recibir un *slot* los que sincronizan la transferencia EDMA3.

Estos dos bucles de transmisión y recepción constituyen un sistema de tres estados. En el primer estado se están transmitiendo las muestras desde el buffer 2 mientras que el buffer 1 está recibiendo muestras. El buffer 0 esta libre y por lo tanto sus muestras pueden procesarse mientras se produce la transferencia.

Cuando se completa la primera transferencia EDMA3 la rutina de interrupción llama a la función *Audio_ISR()* y esta señala que hay un buffer listo para el procesamiento e incrementa el indicador *ready_index*, que pasa a valer 1. Ahora las muestras se envían desde el buffer 0, cuyo procesamiento tuvo lugar en el estado anterior, y el buffer 2 recibe nuevas muestras. Dado que el buffer 1 está lleno y no esta siendo usado para recepción ni para transmisión puede ser aprovechado este tiempo para realizar el procesamiento mientras la transferencia EDMA3 prepara el siguiente buffer.

En el último estado el buffer 1 transmite muestras procesadas y el buffer 0 recibe nuevas muestras. Mientras tanto se procesan las muestras almacenadas en el buffer 2. Cuando esta transferencia termina se vuelve al primer estado, en el que podrán procesarse las muestras del buffer 0.

Nótese que la latencia en muestras es igual al tamaño del buffer. Con una frecuencia de muestreo de 48 KHz se tiene una latencia teórica de aproximadamente 5 milisegundos. Este es también el tiempo del que se dispone para realizar el procesamiento de un buffer y dejarlo listo para su transmisión.

10.10.4 Función EDMA3_Config_Events()

En esta función se configuran varios registros relacionados con los eventos que sincronizan la transferencia EDMA3 y con la interrupción de transferencia completa:

- Los eventos de sincronización de transferencia son capturados en el registro ER (Event Register). Para limpiar los bits de este registro se usa el registro ECR (Event Clear Register). En este caso se ponen a 0 los bits asociados a los eventos McASP.
- El registro EER (Event Enable Register) indica los eventos que están habilitados. Los bits de este registro se cambian escribiendo en el bit correspondiente del registro EESR (Event Enable Set Register). En este caso se ponen a 1 los bits asociados a los

eventos McASP.

- Los dos registros restantes fueron mencionados con anterioridad y se configuran de acuerdo a la figura 12 de la documentación del módulo [49] para habilitar la interrupción INT0 cuando finaliza una transferencia. Esta interrupción está asociada a la rutina de interrupción 8 de la tabla de rutinas de interrupción.

10.11 Conclusiones

En este capítulo se ha presentado código que sirve para configurar los módulos del OMAP-L138. La implementación realizada permite recibir muestras del codec, procesarlas y enviarlas de nuevo al codec usando un sistema de transferencia directa a memoria.

CAPÍTULO 11

Implementación del retardo modulado en el C674x

En este capítulo se explica el código escrito en C para hacer funcionar el algoritmo de retardo modulado en el núcleo C674x del procesador OMAP-L138. El código mostrado funciona en conjunto con los controladores implementados en el capítulo anterior para completar el sistema de procesamiento en tiempo real. Todo el código de este capítulo puede encontrarse en la ruta *./Proyecto/Retardo Modulado/C674x/ModulatedDelay*.

11.1 Vector de efectos

El propósito de este código es procesar un buffer de 256 muestras como los usados en el capítulo anterior para las transferencias EDMA3. Esto se lleva a cabo mediante un vector de efectos. Cada una de las posiciones de este vector contiene un efecto y las muestras fluyen desde el primer efecto hasta el último, siendo procesadas en cadena.

Aunque en el contexto de este trabajo esta implementación puede resultar extraña o innecesaria, hacerlo de este modo permitirá en un futuro introducir otros efectos en el vector. Además, el núcleo C674x podrá cambiar en tiempo de ejecución el contenido del vector de efectos de acuerdo a mensajes externos.

Aunque como ya dije esta idea queda fuera del marco de este trabajo, mi intención es implementar un sistema de comunicación con el núcleo ARM9. El núcleo ARM9 controlará una interfaz de usuario. Cuando reciba una señal de los periféricos escribirá un mensaje para el núcleo C674x en la memoria compartida situada en el interior del chip y generará una interrupción para el DSP. El DSP recibirá la interrupción, señalará la existencia de un mensaje y enmascarará futuras interrupciones del núcleo ARM9 hasta el procesamiento del siguiente buffer. Sin embargo, el núcleo ARM9 podrá seguir ampliando o actualizando el mensaje con las acciones del usuario. Al comienzo del procesamiento de cada bloque el núcleo C674x comprobará si tiene mensajes y de ser así generará una interrupción para el

```
#ifndef EFFECTVECTOR_H_
#define EFFECTVECTOR_H_
#include "ModulatedDelay.h"
#include <ti/csl/tistdtypes.h>
#define SAMPLE_RATE 48000

#define E_VECTOR_SIZE 5
#define MODULATED_DELAY_TYPE 1
#define MODULATED_DELAY_PARAM_FREQ 1
#define MODULATED_DELAY_PARAM_DEPTH 2
#define MODULATED_DELAY_CH 294

typedef struct effect {
    UInt16 type;
    Ptr data_object;
} effect_t;

void AddToEffectVector(Uint16 index, UInt16 type);
void DeleteFromEffectVector(Uint16 index);
void ChangeEffectParamF(Uint16 index, Uint16 param_number, float newValue);
void ProcessBuffer256(float input[]);
#endif /* EFFECTVECTOR_H_ */
```

Listado 11.1: Contenido del archivo *EffectVector.h*.

núcleo ARM9 que le indicará a este que no escriba en la región de memoria compartida hasta que reciba una nueva interrupción. Las acciones del usuario durante este periodo de tiempo serán almacenadas en la memoria interna del núcleo ARM9 para enviarlas a la memoria compartida cuando sea posible. Posteriormente, el núcleo C674x comprobará la zona de mensajes y actualizará el vector de efectos de acuerdo al mensaje existente. Finalmente, enviará una interrupción al núcleo ARM9 para señalizar que ya puede escribir de nuevo en la región de memoria compartida. De este modo, los parámetros de los efectos son actualizados con un periodo teórico de aproximadamente 5 milisegundos si el buffer es de 256 muestras y la frecuencia de muestreo de 48 Khz. Este tiempo es lo suficientemente pequeño para dar una sensación de continuidad.

En el listado 11.1 se muestra el contenido del archivo *EffectVector.h*. En las primeras lineas se definen macros para la frecuencia de muestreo, el número de posiciones del vector de efectos y los parámetros iniciales para el retardo modulado.

Posteriormente se define una estructura para almacenar los datos de cada efecto. Esta contendrá un entero sin signo de 16 bits que representará el tipo de efecto y un puntero a una estructura de datos que contendrá los parámetros del efecto.

En cuanto a las funciones:

- La función *AddToEffectVector(...)* añade un efecto al vector de efectos en la posición indicada por el primer parámetro con el tipo indicado en el segundo parámetro.
- La función *DeleteFromEffectVector(...)* elimina del vector el efecto en la posición indicada.
- La función *ChangeEffectParamF(...)* modifica un parámetro de tipo float en un efecto. Toma como parámetros la posición del efecto en el vector, el número de parámetro y el nuevo valor del parámetro.

```
#include "EffectVector.h"
#include "ModulatedDelay.h"

effect_t effect_array[E_VECTOR_SIZE];

void AddToEffectVector(Uint16 index, UInt16 type){
    effect_array[index].type = type;
    switch(type){
    case MODULATED_DELAY_TYPE:
        effect_array[index].data_object = malloc(sizeof(ModulatedDelay));
        ModulatedDelay_Init((ModulatedDelay*)effect_array[index].data_object, MODULATED_DELAY_CH);
        break;
    default:
        break;
    }
}

void DeleteFromEffectVector(Uint16 index){
   switch(effect_array[index].type){
   case MODULATED_DELAY_TYPE:
       ModulatedDelay_Free((ModulatedDelay*)effect_array[index].data_object);
       break;
   default:
       break;
   }
   effect_array[index].type = 0;
   free(effect_array[index].data_object);
}

void ChangeEffectParamF(Uint16 index, Uint16 param_number, float newValue){
   switch(effect_array[index].type){
   case MODULATED_DELAY_TYPE:
       switch(param_number){
       case MODULATED_DELAY_PARAM_FREQ:
           SineLFO_SetFrequency((&((ModulatedDelay*)effect_array[index].data_object)->lfo),newValue);
           break;
       case MODULATED_DELAY_PARAM_DEPTH:
           SineLFO_SetDepth((&((ModulatedDelay*)effect_array[index].data_object)->lfo),newValue);
           break;
       default:
           break;
       }
       break;
   default:
       break;
   }
}

void ProcessBuffer256(float input[]){
   UInt16 i;
   for(i=0;i<E_VECTOR_SIZE;i++){
       switch(effect_array[i].type){
       case MODULATED_DELAY_TYPE:
           ModulatedDelay_ProcessBuffer256((ModulatedDelay*)effect_array[i].data_object,input);
           break;
       default:
           break;
       }
   }
}
```

Listado 11.2: Contenido del archivo *EffectVector.c*.

- La función *ProcessBuffer256(...)* procesa un buffer de 256 muestras usando el vector de efectos.

En el listado 11.2 se muestra el contenido del archivo *EffectVector.c*. El vector de efectos, llamado *effect_array*, se compone de varias estructuras del tipo *effect_t* definido en la cabecera.

Para añadir un efecto al vector se usa un switch dependiente del tipo de efecto. En este caso hay una sola posibilidad, que el efecto sea un retardo modulado. La primera acción para crear el efecto es hacer que el puntero $data\_object$ de la estructura $effect\_t$ en la posición indicada apunte a una estructura de tipo *ModulatedDelay*. Esta estructura será definida en otra parte del código y contendrá todos los parámetros necesarios para el funcionamiento

del retardo modulado. La función *ModulatedDelay_Init* también será definida más adelante. Su propósito es inicializar los contenidos de una estructura de tipo *ModulatedDelay*.

Para eliminar un efecto del vector se sigue el mismo proceso de selección del tipo de efecto. En el caso del retardo modulado, se llama primero a la función *ModulatedDelay_Free* pasándole el puntero *data_object* como parámetro. El propósito de esta función es liberar la memoria ocupada por las estructuras contenidas en una estructura de tipo *ModulatedDelay*. Posteriormente, se libera la memoria ocupada por la estructura a la que apunta *data_object* y se hace que el tipo de la estructura *effect_t* sea igual a cero, lo cual indica que la posición no contiene ningún efecto.

Para cambiar un parámetro de un efecto se anidan dos switch. El primero selecciona el tipo de efecto, el segundo el número de parámetro. A las funciones de cambio de parámetros del retardo modulado se les pasa una referencia al oscilador de baja frecuencia contenido en la estructura de datos de tipo *ModulatedDelay* y el nuevo valor para el parámetro.

Para procesar un buffer el procedimiento es análogo. Para cada uno de los efectos se selecciona el tipo del efecto y se llama a su función de procesamiento, que almacena el resultado en el mismo array pasado como entrada. La función de procesamiento del retardo modulado, *ModulatedDelay_ProcessBuffer256(...)*, será definida en una sección posterior.

11.2 Buffer Circular

Es posible hacer una implementación más eficiente de un buffer circular que la descrita en la sección 7.2. El inconveniente de esta implementación es que el tamaño del buffer debe ser siempre una potencia de 2.

Cuando el tamaño del buffer es una potencia de 2 puede definirse una máscara que aísle los bits menos significativos de un índice entero, manteniendo el resultado siempre dentro del rango del buffer.

Tomemos como ejemplo el caso en el que el tamaño del buffer es $s = 2^6 = 64$. La máscara será $m = s - 1 = 63$. Representada en binario con un entero de 16 bits queda como sigue: $0000000000111111b$. Si se hace un AND lógico de cualquier entero de 16 bits con esta máscara el resultado estará dentro del intervalo $[0,63]$. Además, todos los enteros representables posibles quedan asociados a un indice en este intervalo. Si tomamos por ejemplo el entero 64, tenemos que su representación en binario es $0000000001000000b$. Haciendo un AND lógico bit a bit con la máscara, el indice resultante es 0. Si tomamos 65, tenemos que su representación en binario es $0000000001000001b$. Haciendo un AND lógico con la máscara, el indice resultante es 1, y así sucesivamente. De este modo,

```
1  #ifndef CIRCULARBUFFER_H_
2  #define CIRCULARBUFFER_H_
3
4  #include <stdlib.h>
5  #include <ti/csl/tistdtypes.h>
6
7  typedef struct CircularBuffer {
8    Int32 writeIndex;
9    float* first;
10   float* last;
11   Int32 mask;
12 }CircularBuffer;
13
14 void CircularBuffer_Init(CircularBuffer* cb, Uint32 size);
15 void CircularBuffer_Free(CircularBuffer* cb);
16 void CircularBuffer_Push(CircularBuffer* cb, const float value);
17 void CircularBuffer_PushRight(CircularBuffer* cb, const float value);
18 float CircularBuffer_Get(CircularBuffer* cb, const int position);
19 float CircularBuffer_PushDown(CircularBuffer* cb);
20
21 #endif /* CIRCULARBUFFER_H_ */
```

Listado 11.3: Contenido del archivo *CircularBuffer.h*.

podemos incrementar y decrementar un índice sin hacer ningún tipo de comprobación de límites.

Además, si estamos usando una representación en complemento a dos y hacemos una negación lógica de un entero el resultado será el mismo que cambiar el signo del entero original y restarle 1. Por ejemplo, tomemos el entero 63 y neguémoslo lógicamente. Esto resulta en la cadena binaria $1111111111000000b$, es decir, -64 representado en complemento a dos.

Por tanto, si queremos acceder a una posición del buffer circular basta con negar la posición, sumarla al valor actual del índice y enmascarar el resultado con la máscara del buffer.

En el listado 11.3 se muestra el contenido del archivo *CircularBuffer.h*. La única función nueva es *CircularBuffer_PushDown(...)*. Esta función devuelve el último elemento del buffer circular (el más antiguo) y posteriormente lo desplaza a la primera posición, moviendo el resto de muestras hacia la derecha.

Como puede verse, la estructura *CircularBuffer* contiene los datos necesarios para el funcionamiento del buffer circular, incluyendo un puntero a la posición de memoria del primer y el último elemento.

En el listado 11.4 se muestra el contenido del archivo *CircularBuffer.c*. La función de inicialización, *CircularBuffer_Init(...)*, se usa para inicializar los datos de la estructura de buffer circular, reservar la memoria necesaria para el buffer y escribir un 0 en todas las posiciones usando la función *memset(...)*.

El resto de funciones son implementaciones triviales basadas en las explicaciones dadas anteriormente sobre el funcionamiento del buffer.

Estas funciones se han escrito en un archivo separado por motivos de claridad en las

```
#include "CircularBuffer.h"
#include <string.h>
#include <stdio.h>

void CircularBuffer_Init(CircularBuffer* cb, Uint32 size)  {
  cb->mask = size-1;
  cb->writeIndex = 0;
  cb->first = malloc(sizeof(float)*size);
  if (cb->first == 0)
  {
      printf("ERROR: Out of memory\n");
      exit(0);
  }
  cb->last = cb->first + cb->mask;
  memset(cb->first, 0, sizeof(float)*size);
}

void CircularBuffer_Free(CircularBuffer* cb)  {
  free(cb->first);
}

void CircularBuffer_Push(CircularBuffer* cb, const float value) {
  cb->first[(cb->writeIndex++) & cb->mask] = value;
}

void CircularBuffer_PushRight(CircularBuffer* cb, const float value) {
  cb->first[(--cb->writeIndex) & cb->mask] = value;
}

float CircularBuffer_Get(CircularBuffer* cb, int position) {
  return cb->first[(cb->writeIndex + (~position)) & cb->mask];
}

float CircularBuffer_PushDown(CircularBuffer* cb){
  return cb->first[(cb->writeIndex++) & cb->mask];
}
```

Listado 11.4: Contenido del archivo *CircularBuffer.c*.

explicaciones. Sin embargo, como se verá posteriormente, su código será introducido directamente en el bucle de procesamiento en lugar de introducir llamadas a las funciones aquí implementadas. Esto se debe a que el sistema de desenrollamiento de bucles del procesador no puede funcionar adecuadamente si hay una llamada a función dentro de un bucle, ya que se rompe la linealidad de la ejecución.

11.3 Oscilador

La implementación de este oscilador es análoga a la realizada en la sección 7.7, aunque existen algunas diferencias.

En el listado 11.5 se muestra el contenido del archivo *SineLFO.h*. Como puede verse, se define una estructura de datos que contiene los parámetros necesarios para hacer funcionar el oscilador. La función *SineLFO_Init(...)* sirve para inicializar los datos de la estructura pasada como parámetro.

En este caso el oscilador no genera las muestras de una en una sino que rellena directamente un array contenido en una posición de memoria. Para ello se usa la función *SineLFO_FillArray256(...)*.

Nótese que en todas las funciones se pasa como parámetro un puntero a una estructura de

```
1  #ifndef SINELFO_H_
2  #define SINELFO_H_
3  #include <ti/csl/tistdtypes.h>
4
5  typedef struct SineLFO {
6    Uint32 sample_rate;
7    float frequency;
8    float phstep;
9    float phase;
10   float depth;
11   Bool change_depth;
12   float new_depth;
13 }SineLFO;
14
15 void SineLFO_Init(SineLFO* lfo, int sample_rate);
16 void SineLFO_SetFrequency(SineLFO* lfo, float frq);
17 void SineLFO_SetPhase(SineLFO* lfo, float phase);
18 void SineLFO_SetDepth(SineLFO* lfo, float depth);
19 void SineLFO_FillArray256(SineLFO* lfo, float output[]);
20 #endif /* SINELFO_H_ */
```

Listado 11.5: Contenido del archivo *SineLFO.h*.

datos de tipo *SineLFO*. Las funciones toman de ella los datos para generar un resultado o los modifican en el caso de las funciones *set*. Este patrón se repetirá en el resto de los archivos, constituyendo una estructura de pseudo-clases en C.

En el listado 11.6 se muestra el contenido del archivo *SineLFO.c*. La implementación es en su mayoría análoga a la realizada en la sección 7.7. La mayor diferencia reside en la implementación de la función *SineLFO_FillArray256(...)*.

En la linea 41 se generan mediante un bucle 256 valores para calcular posteriormente el seno de todos ellos. Si el booleano *change_depth* de la estructura de datos recibida como parámetro es positivo es necesario cambiar el parámetro *depth* de la estructura que contiene los datos del oscilador. Como se mencionó en la sección 11.6 este parámetro solo se modifica cuando la fáse es mayor que 2π o 0. En este caso, se deja indicado el número de índice para realizar el cambio posteriormente.

Nótese el uso de la directiva *MUST_ITERATE*. Esta es la directiva que permite indicar al compilador el número de iteraciones de un bucle. El primer parámetro es el número de iteraciones mínimo, el segundo es el número de iteraciones máximo y el tercero indica que el número de iteraciones es un múltiplo de su valor.

En la linea 52 se llama a la función *sinsp_v(...)*. Esta función es una función de la librería *mathlib* optimizada para ejecutar un conjunto de senos y guardar los resultados en un vector de salida. El array *a* contiene los valores de entrada generados anteriormente, el array *output* almacenará los resultados y el último parámetro es el tamaño de ambos arrays.

Una vez calculados los senos se multiplican por el parámetro *depth*. El valor *cindex* almacenado anteriormente permite modificar el valor de dicho parámetro en el índice de iteración adecuado.

Esta forma de generar los valores sinusoidales es más eficiente que generarlos uno a uno, ya

```
#include <ti/mathlib/mathlib_rts.h>
#include "SineLFO.h"

#define M_PI 3.14159265358979323846
#define D_PI (M_PI*2)

void SineLFO_Init(SineLFO* lfo, int sample_rate){
    lfo->sample_rate = sample_rate;
    lfo->frequency = 2.0f;
    lfo->phase = 0.0f;
    lfo->depth = 1.0f;
    lfo->phstep = (2.0f/sample_rate) * D_PI;
    lfo->change_depth = 0;
}

void SineLFO_SetSampleRate(SineLFO* lfo, Uint32 sample_rate){
    lfo->sample_rate = sample_rate;
    SineLFO_SetFrequency(lfo, lfo->frequency);
}

void SineLFO_SetFrequency(SineLFO* lfo, float frq){
    lfo->frequency = frq;
    lfo->phstep = (frq/lfo->sample_rate) * D_PI;
}

void SineLFO_SetPhase(SineLFO* lfo, float phase){
    lfo->phase = phase;
}

void SineLFO_SetDepth(SineLFO* lfo, float depth){
    lfo->new_depth = depth;
    lfo->change_depth = 1;
}

void SineLFO_FillArray256(SineLFO* lfo, float output[]){
    Uint32 i;
    Uint32 cindex;
    Bool c = 0;
    float a[256];
    float* p = a;
    #pragma MUST_ITERATE(256,256,2)
    for(i = 0; i<256; i++){
        *p++ = lfo->phase;
        lfo->phase += lfo->phstep;
        if(lfo->change_depth == 1 && (lfo->phase > D_PI || lfo->phase == 0)) {
            cindex = i;
            c = 1;
        }
        while (lfo->phase > D_PI)
            lfo->phase -= D_PI;
    }
    sinsp_v(a, output, 256);
    p = output;
    #pragma MUST_ITERATE(256,256,2)
    for(i = 0; i<256; i++){
        *p++ = *p * lfo->depth;
        if(c && i == cindex){
            lfo->depth = lfo->new_depth;
            lfo->change_depth = 0;
            c = 0;
        }
    }
}
```

Listado 11.6: Contenido del archivo *SineLFO.c*.

```
1  #ifndef FRACTIONALDELAY_H_
2  #define FRACTIONALDELAY_H_
3
4  #include "CircularBuffer.h"
5  #define DIFSIZE 63
6  #define DIFSIZEH ((DIFSIZE-1)/2)
7
8  typedef struct  {
9      Int32 start;
10     CircularBuffer extBuffer;
11     CircularBuffer stageA;
12     CircularBuffer stageB;
13     CircularBuffer stageC;
14     CircularBuffer stageD;
15     CircularBuffer stageE;
16     CircularBuffer stageF;
17     CircularBuffer stageG;
18 } FractionalDelay;
19
20 void FractionalDelay_Init(FractionalDelay* fd, Int32 start);
21 void FractionalDelay_Free(FractionalDelay* fd);
22
23 #endif /* FRACTIONALDELAY_H_ */
```

Listado 11.7: Contenido del archivo *FractionalDelay.h*.

que el compilador puede hacer uso de las estructuras de paralelismo del núcleo, incluyendo el sistema de desenrollamiento de bucles. Además, las funciones de la librería *mathlib* están optimizadas directamente en lenguaje ensamblador y en general es una buena idea usarlas siempre que sea posible. La documentación de esta librería puede encontrarse en [54].

11.4 Retardo Fraccionario

En el listado 11.7 se muestra el contenido del archivo *FractionalDelay.h*. Como puede verse, en este caso el código del retardo fraccionario solo define una estructura en la que se almacenan los buffers circulares necesarios para su funcionamiento.

Además, se declaran dos funciones usadas para reservar y liberar la memoria necesaria y se definen en las primeras lineas macros para el tamaño del diferenciador.

En el listado 11.8 se muestra el archivo de implementación. En él se definen los coeficientes del diferenciador y las funciones de reserva y liberación de memoria. Lo único que hacen estas funciones es llamar a su vez a las funciones de inicialización o liberación de cada buffer circular. En el caso de la función de inicialización el parámetro *start* indica la posición actual del retardo fraccionario en el primer buffer.

11.5 Retardo Modulado

En el listado 11.9 se muestra el contenido del archivo *ModulatedDelay.h*. En él se define una estructura que contiene los datos necesarios para el funcionamiento de un retardo

```
#include "FractionalDelay.h"

float h_dif[DIFSIZE] = {
0.000101123672523732 04f, -0.000214909017491092 83f,  0.000389259420547352 78f, -0.000642902137881814 03f,
        0.000997815945163807 67f,
-0.001479429111007423 9f,  0.002116804961747320 1f,  -0.002942827259786639 5f,  0.003994405390933085 7f,
     -0.005312729598867354 6f,
0.006943620203454814 8f,  -0.008938033460947968 8f,  0.011352812992477129 f,  -0.014251813650764486 f,
         0.017707581183683881 f,
-0.021803857757955208 f,  0.026639320637763239 f,  -0.032333185422459339 f,  0.039033683421326598 f,
     -0.046931083823310953 f,
0.056278133040736561 f,  -0.067423067241917489 f,  0.080864922469061115 f,  -0.097350576551591669 f,
         0.118055153953430 86f,
-0.144942921361081 39f,  0.181561198412249 88f,  -0.235024854859070 63f,  0.321969163952363 75f,
     -0.492359889629482 85f,
0.996160593220573 71f};

void FractionalDelay_Init(FractionalDelay* fd, Int32 start)  {
  CircularBuffer_Init(&fd->extBuffer, 1024);
  CircularBuffer_Init(&fd->stageA, 256);
  CircularBuffer_Init(&fd->stageB, 256);
  CircularBuffer_Init(&fd->stageC, 128);
  CircularBuffer_Init(&fd->stageD, 128);
  CircularBuffer_Init(&fd->stageE, 64);
  CircularBuffer_Init(&fd->stageF, 64);
  fd->start = start;
}

void FractionalDelay_Free(FractionalDelay* fd)  {
  CircularBuffer_Free(&fd->extBuffer);
  CircularBuffer_Free(&fd->stageA);
  CircularBuffer_Free(&fd->stageB);
  CircularBuffer_Free(&fd->stageC);
  CircularBuffer_Free(&fd->stageD);
  CircularBuffer_Free(&fd->stageE);
  CircularBuffer_Free(&fd->stageF);
}
```

Listado 11.8: Contenido del archivo *FractionalDelay.c*.

```
#ifndef MODULATEDDELAY_H_
#define MODULATEDDELAY_H_

#include "SineLFO.h"
#include "FractionalDelay.h"

typedef struct ModulatedDelay {
    int CHORUS_WIDTH;
    int NOMINAL_DELAY;
    Int32 oldi;
    FractionalDelay md_fd;
    SineLFO lfo;
}ModulatedDelay;

void ModulatedDelay_Init(ModulatedDelay* md, int CH);
void ModulatedDelay_Free(ModulatedDelay* md);
void ModulatedDelay_ProcessBuffer256(ModulatedDelay* md, float* input);

#endif /* MODULATEDDELAY_H_ */
```

Listado 11.9: Contenido del archivo *ModulatedDelay.h*.

modulado: los parámetros de centro y amplitud de modulación; la posición actual del retardo fraccionario; la estructura que contiene sus buffers; y el oscilador.

A la función de inicialización es necesario pasarle como parámetro, además de un puntero a la estructura a inicializar, la amplitud máxima de la oscilación en muestras.

En los listados 11.10,11.11 y 11.12 se muestra el contenido del archivo *ModulatedDelay.c*. Las funciones de inicialización y liberación son triviales, pero la función de procesamiento requiere de una explicación en profundidad.

Esta función es una optimización del código escrito en capítulos previos. Para entender el código usado en las operaciones con buffers circulares recomiendo al lector que tome como referencia las funciones implementadas en la sección 11.2. Para entender las explicaciones también puede ser de gran utilidad la figura 5.6. Cada uno de los buffers circulares debe interpretarse como una columna de esta figura. Las muestras entran por la parte superior y salen por la parte inferior. Cada linea verde abarca un número de muestras igual al número de coeficientes del diferenciador. Cuando el retardo fraccionario se desplaza sobre el primer buffer o entra en él una nueva muestra las acciones a realizar tienen como objetivo generar nuevos valores para la primera posición del resto de los buffers.

A grandes rasgos, la función de procesamiento hace las mismas operaciones que el retardo modulado implementado en un capítulo anterior para el plugin de audio. La diferencia es que ahora todas las operaciones se llevan a cabo en una sola función por motivos de eficiencia.

En las lineas 24 y 25 se declara un array para contener la señal sinusoidal de oscilación y se rellena usando la función creada anteriormente para ello pasando como parámetro la estructura con los datos del oscilador del retardo modulado.

A continuación se declara un puntero a la estructura que contiene los datos del retardo fraccionario perteneciente a la estructura de datos del retardo modulado y otra serie de punteros y variables cuyo propósito resultará claro posteriormente.

En la linea 66 se introduce una muestra en el primer buffer circular del retardo fraccionario. Como se mencionó anteriormente, las operaciones con el buffer se llevan a cabo de forma directa, sin llamar a ninguna función, por motivos de eficiencia.

Posteriormente se calculan $frac$ e i. Recuérdese que la función *_spint()* redondea un float al entero más próximo. Si la antigua posición del retardo fraccionario, $oldi$, es menor que la nueva posición, entonces tan solo hay que desplazar el retardo fraccionario cambiando el valor de $start$. En caso contrario es necesario llevar a cabo el proceso de actualización de buffers.

```
#include "FractionalDelay.h"
#include <ti/csl/tistdtypes.h>
#include "ModulatedDelay.h"
#include "SineLFO.h"
#include <math.h>
#include <stdio.h>

extern float h_dif[];

void ModulatedDelay_Init(ModulatedDelay* md, Int32 CH){
    md->CHORUS_WIDTH = CH;      //294
    md->NOMINAL_DELAY = CH;     //294
    md->oldi = CH;              //294
    FractionalDelay_Init(&md->md_fd, CH);
    SineLFO_Init(&md->lfo, 48000);       //Sample rate is 48000 Hz
}

void ModulatedDelay_Free(ModulatedDelay* md){
    FractionalDelay_Free(&md->md_fd);
}

void ModulatedDelay_ProcessBuffer256(ModulatedDelay* md, float* input){
    //Set up lfo array
    float lfo_samples[256];
    SineLFO_FillArray256(&md->lfo, lfo_samples);
    //Internal loop variables
    Int32 i;
    Uint32 j;
    Uint32 k;
    float frac;
    FractionalDelay* fd = &md->md_fd;
    float* startExt;
    float* startA;
    float* startB;
    float* startC;
    float* startD;
    float* startE;
    float* startF;
    float* PstartExt;
    float* PstartA;
    float* PstartB;
    float* PstartC;
    float* PstartD;
    float* PstartE;
    float* PstartF;
    float* PendExt;
    float* PendA;
    float* PendB;
    float* PendC;
    float* PendD;
    float* PendE;
    float* PendF;
    float rExt;
    float rA;
    float rB;
    float rC;
    float rD;
    float rE;
    float rF;
    float* hP;
    float aux = 0;
    float res = 0;
    #pragma MUST_ITERATE(256,256,2)
    for(j = 0; j<256; j++){
        //Do the processing
        fd->extBuffer.first[(fd->extBuffer.writeIndex++) & fd->extBuffer.mask] = input[j];
        lfo_samples[j] *= md->CHORUS_WIDTH;
        lfo_samples[j] += md->NOMINAL_DELAY;
        i = _spint(lfo_samples[j]);
        frac = lfo_samples[j] - i;
        if (md->oldi < i) {
            md->oldi++;
            fd->start = md->oldi;
        }else{
            while (md->oldi >= i){
                fd->start = md->oldi;
                //Stages set up
                startExt = &(fd->extBuffer.first[(fd->extBuffer.writeIndex + (~fd->start)) & fd->extBuffer.
                    mask]);
                startA = &(fd->stageA.first[(fd->stageA.writeIndex++) & fd->stageA.mask]);
                startB = &(fd->stageB.first[(fd->stageB.writeIndex++) & fd->stageB.mask]);
                startC = &(fd->stageC.first[(fd->stageC.writeIndex++) & fd->stageC.mask]);
                startD = &(fd->stageD.first[(fd->stageD.writeIndex++) & fd->stageD.mask]);
                startE = &(fd->stageE.first[(fd->stageE.writeIndex++) & fd->stageE.mask]);
                startF = &(fd->stageF.first[(fd->stageF.writeIndex++) & fd->stageF.mask]);
```

Listado 11.10: Contenido del archivo *ModulatedDelay.c*, primera parte.

```
85                  PstartExt = startExt;
86                  PstartA = startA;
87                  PstartB = startB;
88                  PstartC = startC;
89                  PstartD = startD;
90                  PstartE = startE;
91                  PstartF = startF;
92                  PendExt = &(fd->extBuffer.first[(fd->extBuffer.writeIndex + (~(fd->start+DIFSIZE-1))) & fd
                        ->extBuffer.mask]);
93                  PendA = &(fd->stageA.first[(fd->stageA.writeIndex + (~(DIFSIZE-1))) & fd->stageA.mask]);
94                  PendB = &(fd->stageB.first[(fd->stageB.writeIndex + (~(DIFSIZE-1))) & fd->stageB.mask]);
95                  PendC = &(fd->stageC.first[(fd->stageC.writeIndex + (~(DIFSIZE-1))) & fd->stageC.mask]);
96                  PendD = &(fd->stageD.first[(fd->stageD.writeIndex + (~(DIFSIZE-1))) & fd->stageD.mask]);
97                  PendE = &(fd->stageE.first[(fd->stageE.writeIndex + (~(DIFSIZE-1))) & fd->stageE.mask]);
98                  PendF = &(fd->stageF.first[(fd->stageF.writeIndex + (~(DIFSIZE-1))) & fd->stageF.mask]);
99                  *startA = 0;
100                 *startB = 0;
101                 *startC = 0;
102                 *startD = 0;
103                 *startE = 0;
104                 *startF = 0;
105
106                 //Loop variables set up
107                 hP = h_dif;
108                 rExt = 0;
109                 rA = 0;
110                 rB = 0;
111                 rC = 0;
112                 rD = 0;
113                 rE = 0;
114                 rF = 0;
115
116                 #pragma MUST_ITERATE(DIFSIZEH,DIFSIZEH,)
117                 for(k = 0;k<DIFSIZEH;k++){
118                         rExt += (*PstartExt-- - *PendExt++) * *hP;
119                         rA += (*PstartA-- - *PendA++) * *hP;
120                         rB += (*PstartB-- - *PendB++) * *hP;
121                         rC += (*PstartC-- - *PendC++) * *hP;
122                         rD += (*PstartD-- - *PendD++) * *hP;
123                         rE += (*PstartE-- - *PendE++) * *hP;
124                         rF += (*PstartF-- - *PendF++) * *hP;
125                         hP++;
126                         if(PstartExt < fd->extBuffer.first)
127                             PstartExt = fd->extBuffer.last;
128                         if(PstartA < fd->stageA.first)
129                             PstartA = fd->stageA.last;
130                         if(PstartB < fd->stageB.first)
131                             PstartB = fd->stageB.last;
132                         if(PstartC < fd->stageC.first)
133                             PstartC = fd->stageC.last;
134                         if(PstartD < fd->stageD.first)
135                             PstartD = fd->stageD.last;
136                         if(PstartE < fd->stageE.first)
137                             PstartE = fd->stageE.last;
138                         if(PstartF < fd->stageF.first)
139                             PstartF = fd->stageF.last;
140                         if(PendExt > fd->extBuffer.last)
141                             PendExt = fd->extBuffer.first;
142                         if(PendA > fd->stageA.last)
143                             PendA = fd->stageA.first;
144                         if(PendB > fd->stageB.last)
145                             PendB = fd->stageB.first;
146                         if(PendC > fd->stageC.last)
147                             PendC = fd->stageC.first;
148                         if(PendD > fd->stageD.last)
149                             PendD = fd->stageD.first;
150                         if(PendE > fd->stageE.last)
151                             PendE = fd->stageE.first;
152                         if(PendF > fd->stageF.last)
153                             PendF = fd->stageF.first;
154                 }
155
156                 //Arrange results
157                 *startA = rExt * -1.0f;
158                 *startB = ( *startA * *h_dif + rA ) * -0.5f;
159                 *startC = ( *startB * *h_dif + rB ) * -0.3333333333f;
160                 *startD = ( *startC * *h_dif + rC ) * -0.25f;
161                 *startE = ( *startD * *h_dif + rD ) * -0.2f;
162                 *startF = ( *startE * *h_dif + rE ) * -0.1666666667f;
163                 res = ( *startF * *h_dif + rF ) * -0.1428571429f;
164                 md->oldi--;
165             }
166             md->oldi++;
167         }
```

Listado 11.11: Contenido del archivo *ModulatedDelay.c*, segunda parte.

```
168             //Interpolate the sample
169              aux = res * frac;
170              aux += fd->stageF.first[(fd->stageF.writeIndex + (~(DIFSIZEH-1))) & fd->stageF.mask]; aux *= frac;
171              aux += fd->stageE.first[(fd->stageE.writeIndex + (~(DIFSIZEH*2-1))) & fd->stageE.mask]; aux *=
                      frac;
172              aux += fd->stageD.first[(fd->stageD.writeIndex + (~(DIFSIZEH*3-1))) & fd->stageD.mask]; aux *=
                      frac;
173              aux += fd->stageC.first[(fd->stageC.writeIndex + (~(DIFSIZEH*4-1))) & fd->stageC.mask]; aux *=
                      frac;
174              aux += fd->stageB.first[(fd->stageB.writeIndex + (~(DIFSIZEH*5-1))) & fd->stageB.mask]; aux *=
                      frac;
175              aux += fd->stageA.first[(fd->stageA.writeIndex + (~(DIFSIZEH*6-1))) & fd->stageA.mask]; aux *=
                      frac;
176              aux += fd->extBuffer.first[(fd->extBuffer.writeIndex + (~(fd->start + DIFSIZEH*7-1))) & fd->
                      extBuffer.mask];
177              input[j] = aux;
178          }
179      }
```

Listado 11.12: Contenido del archivo *ModulatedDelay.c*, tercera parte.

En este último caso, en primer lugar se toma un puntero a la primera posición del primer buffer del retardo fraccionario en la linea 78. Para ello, se suma al puntero de escritura del array la posición actual del retardo fraccionario y se aplica al resultado la máscara del buffer. Si tomamos la figura 5.6 como referencia, esta posición se corresponde con la primera muestra abarcada por la linea verde del primer buffer.

Posteriormente se toma un puntero a la primera posición de cada uno de los buffers a partir de la linea 79. A la vez se rota cada buffer una posición, haciendo que la muestra más antigua pase a ser la primera y desplazando el resto de muestras. Todos estos punteros son copiados a otra variable a partir de la linea 85. Esta operación puede interpretarse como un desplazamiento hacia abajo de todas las muestras en los buffers de la figura 5.6, dejando la primera posición libre. Como puede verse en el código, a partir de la linea 99 se almacena un 0 en todas estas posiciones menos en la correspondiente al primer buffer.

A continuación, a partir de la linea 92, se toma un puntero a la posición 62 de cada buffer. Si tomamos la figura 5.6 como referencia esta posición se corresponde con la última muestra abarcada por las lineas verdes.

En la linea 107 se almacena en el puntero *hP* la posición de memoria del array que contiene los coeficientes del diferenciador. Posteriormente, a partir de la linea 108, se inicializan varios enteros con valor cero que servirán como acumuladores para calcular convoluciones.

Recapitulando, en este punto del código se tienen:

- Punteros a las primeras posiciones de todos los buffers. En todas estas posiciones menos en la perteneciente al primer buffer se ha introducido un 0.
- Punteros a la posición 62 de cada buffer.

Con estas variables es posible calcular una convolución parcial de las muestras iniciales de cada buffer con los coeficientes del diferenciador. Tomando como referencia la figura 5.6,

la operación que va a realizarse consistiría en hacer una convolución de los coeficientes del diferenciador con cada uno de los grupos de muestras abarcados por las lineas verdes ignorando siempre la primera posición (en la que se ha introducido un cero).

En la linea 117 empieza el bucle que lleva a cabo esta convolución. El puntero que indica la posición inicial para cada buffer se decrementa mientras que el que indica la posición final se incrementa. El contenido de ambas posiciones se resta, al ser el filtro antisimétrico, y el resultado se multiplica por el coeficiente correspondiente del diferenciador. El resultado final obtenido se va sumando a un acumulador para cada buffer. Posteriormente, a partir de la linea 126, se hacen las comprobaciones de límites para cada puntero.

Mi idea con este método es que el compilador aproveche al máximo el paralelismo del nucleo. Se están llevando a cabo dentro de un mismo bucle 7 convoluciones que incluyen operaciones de suma, resta y multiplicación. Con esta implementación el compilador debería poder aprovechar al máximo las unidades funcionales y el sistema de desenrollamiento de bucles. La contrapartida es un tamaño de código mucho mayor.

Nótese que no es posible introducir en la operación la primera muestra de cada buffer, ya que esta depende del resultado de la convolución de los coeficientes del diferenciador con las muestras del buffer anterior.

A partir de la linea 157 la idea es organizar los resultados obtenidos y añadir al resultado de cada convolución parcial la primera muestra de cada buffer. En el caso del segundo buffer, la primera posición será igual al resultado de la primera convolución multiplicado por $-1/2$. En el caso del tercer buffer, se añade al acumulador del segundo buffer el resultado de multiplicar el valor de la primera posición del segundo buffer por el primer coeficiente del diferenciador y por último se multiplica el resultado de esta suma por $-1/3$. En caso de que esta parte del código no resulte clara puede revisarse la figura 4.8. El resultado de cada aplicación del filtro $G(z)$, es decir, el diferenciador, pasa al siguiente buffer multiplicado por una fracción dependiente de la etapa.

Finalmente, a partir de la linea 168, se hacen las operaciones necesarias para aproximar la muestra requerida de acuerdo al valor de $frac$. Para ello, simplemente se sigue una vez más el esquema de la figura 4.8.

11.6 Función main()

En el listado 11.13 se muestra el contenido del archivo *main.c*. En él se hace uso del código mostrado en las secciones anteriores para escribir la función principal del programa.

```
#include "EDMA.h"
#include "I2C.h"
#include "AIC3106.h"
#include "McASP.h"
#include "GPIO.h"
#include "buffer_proc.h"
#include "EffectVector.h"
#include "SYSCFG.h"
#include "SineLFO.h"


void syscfg(void){
    PINMUX0 = PINMUX0_VALUE;
    PINMUX1 = PINMUX1_VALUE;
    PINMUX2 = PINMUX2_VALUE;
    PINMUX5 = PINMUX5_VALUE;
    PINMUX13 = PINMUX13_VALUE;
}

/**
 * main.c
 */
int main(void)
{
    //Configure effect vector
    AddToEffectVector(0, MODULATED_DELAY_TYPE);
    ChangeEffectParamF(0, MODULATED_DELAY_PARAM_FREQ, 2.0f);
    ChangeEffectParamF(0, MODULATED_DELAY_PARAM_DEPTH, 0.5f);
    //System set up
    syscfg();
    Init_GPIO_LEDS();
    Zero_Buffers();
    EDMA3_Config_Params();
    EDMA3_Config_Events();
    Init_Interrupts();
    Init_I2C();
    if(!Init_AIC3106()) {
            while(1);
    }
    Init_McASP0();
    Enable_Interrupts();
    while(1) {
        if(IsBufferReady())
            Process_Buffer();
    }
}
```

Listado 11.13: Contenido del archivo *main.c*.

La función *syscfg()* escribe en los registros de configuración de multiplexores los valores definidos en el archivo *SYSCFG.h*. Esta función será llamada en la linea 30, justo antes de comenzar la configuración de los módulos.

A partir de la linea 26 se hace uso de las funciones del vector de efectos para añadir un efecto de tipo retardo modulado en la posición 0 del vector de efectos y configurar sus parámetros de frecuencia y profundidad.

A partir de la linea 31 se llama a las funciones de configuración de módulos. Estas funciones fueron explicadas en el capítulo de implementación de controladores, por lo que no insistiré en ellas.

En la linea 42 comienza el bucle principal del programa. Este es un bucle infinito que llama constantemente a la función *IsBufferReady()*, definida en el archivo *buffer_proc.c*. Esta función devolverá 1 si existe un buffer listo para procesar, en cuyo caso se llamará a la función *Process_Buffer()*, definida también en el archivo *buffer_proc.c*.

11.7 Conclusiones

Con el código presentado se consideran cumplidos los objetivos marcados para este trabajo en cuanto al efecto de retardo modulado. El código compilado puede cargarse directamente en la memoria del procesador mediante la sonda de depuración con la ayuda del IDE de Texas Instruments. Al comenzar una sesión de depuración el depurador situará el contador de programa en el punto de entrada, comenzando la ejecución. En el siguiente capítulo se explican las pruebas realizadas.

CAPÍTULO 12

Pruebas del retardo modulado en el C674x

Este capítulo pertenece al proceso de pruebas del retardo modulado diseñado. En este caso se analizará el software final ejecutándose en el núcleo C674x del OMAP-L138.

12.1 Comparación del espectro de frecuencias

Para analizar la respuesta en frecuencia del sistema y compararla con el procesamiento enteramente digital mostrado en el capítulo 8 hice uso de la configuración que se describe a continuación.

En primer lugar convertí una señal sinusoidal de 1 KHz a formato mp3 y la guardé en un reproductor con salida jack de 3.6 mm. Posteriormente, conecté este reproductor a la entrada de linea del codec de la placa de desarrollo. A su vez, conecté la salida del codec a una de las entradas de linea de la interfaz de audio. En Reaper configuré una pista para recibir audio de esta entrada de linea y en la misma pista añadí el plugin *Frequency Spectrum Analyzer Meter* usado en el capítulo 8.

Esta configuración permite analizar en el dominio de la frecuencia la respuesta del sistema a una señal de 1 KHz. La conversión a formato mp3, las impedancias de entrada y salida, las posibles interferencias y los procesos de conversión analógico-digital introducirán ruido y distorsión que impedirá realizar un análisis adecuado del sistema. Sin embargo, considero que esta prueba es suficiente para verificar que el software funciona adecuadamente, al margen de las consideraciones que puedan hacerse sobre el hardware.

En la figura 12.1 se muestran los resultados obtenidos. El comportamiento es el esperado, el pico en la frecuencia principal oscila de izquierda a derecha y es posible variar su velocidad y amplitud de oscilación cambiando los parámetros en el código fuente. Sin embargo, tal y como era de esperar, el nivel de ruido es elevado.

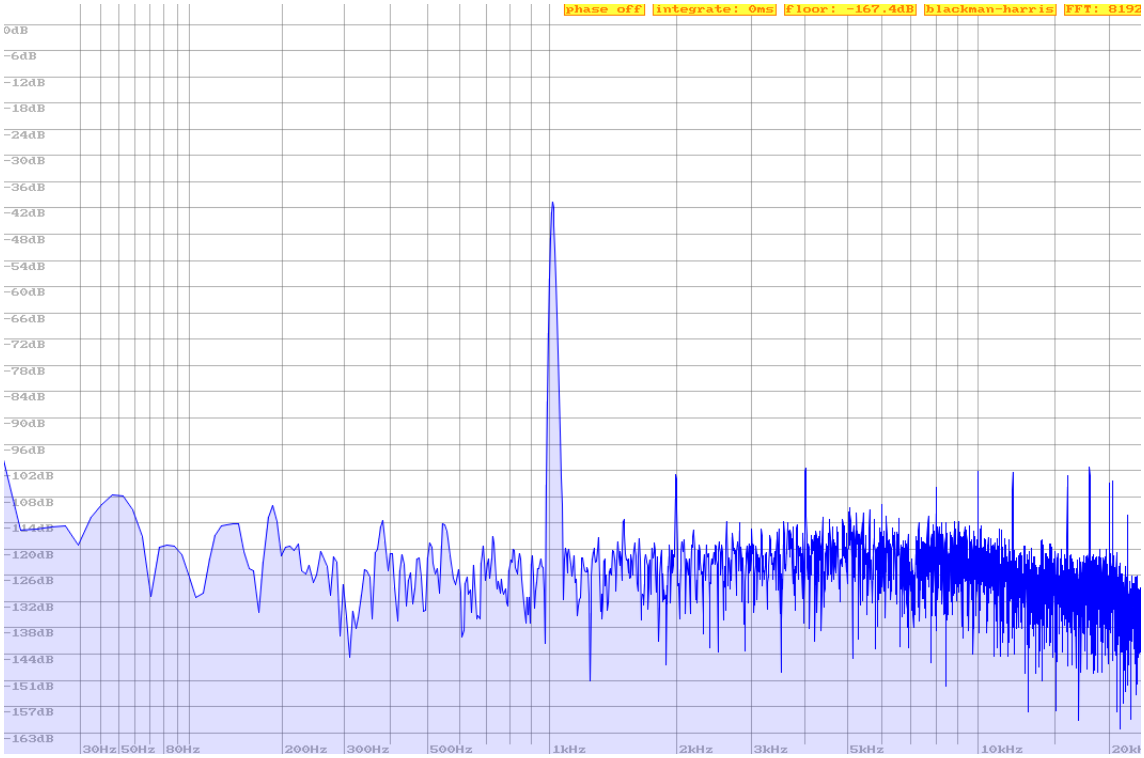


Figura 12.1: Respuesta del software implementado a una señal de 1046.502 Hz.

12.2 Tiempo de procesamiento

En la figura 12.2 se muestra la forma de onda obtenida con el osciloscopio al medir la señal de voltaje en el LED usado para obtener el tiempo de procesamiento. Como puede verse, el periodo de la señal coincide con la latencia calculada en la sección 10.10.3. Cada flanco de subida se corresponde con una transferencia EDMA3 completada, es decir, un buffer listo para su procesamiento. Cada flanco de bajada se corresponde con la finalización del procesamiento de todas las muestras de un buffer.

En cuanto al tiempo de procesamiento, con la implementación realizada es de tan solo 840 $\mu s/buffer$. Hay que tener en cuenta que el procesador está funcionando con la frecuencia de reloj configurada por defecto, 300 MHz. Esta frecuencia puede aumentarse hasta 456 MHz.

La primera versión del programa era fruto de una implementación descuidada en cuanto a eficiencia, siendo esta una mera traducción del código en C++. El resultado era desastroso, el núcleo ni siquiera tenía tiempo de completar el procesamiento antes de la llegada del siguiente buffer.

La conclusión que extraigo es que para procesamientos complejos es muy importante hacer implementaciones eficientes, a ser posible haciendo uso del sistema de desenrollamiento

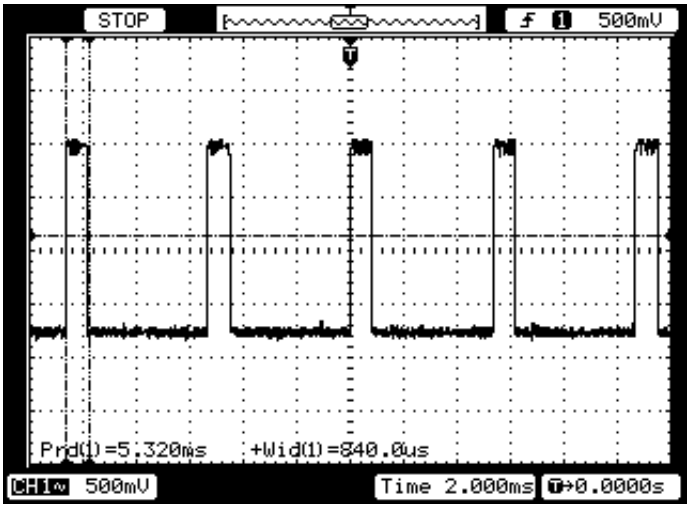


Figura 12.2: Tiempo de procesamiento (anchura de pulso) y latencia de buffers (periodo) en el software implementado.

de bucles, aún a costa de sacrificar abstracción y tamaño de código.

Obviamente estas pruebas se han hecho con el programa cargado en la memoria interna L2. Lo único que tiene que desplazarse desde la memoria RAM es el buffer transferido mediante EDMA3 que vaya a procesarse.

Si en un futuro se incluyen más efectos o código adicional es posible que el tamaño final del programa exceda el tamaño de la memoria interna L2, en cuyo caso habría que implementar un sistema de caché o hacer uso del sistema operativo TI-RTOS, ya que no sería posible tener el programa completo almacenado en la memoria interna del chip en todo momento. Texas Instruments proporciona ejemplos de código muy asequibles por lo que la migración no sería complicada. Obviamente, los archivos de código del efecto no tendrían que ser modificados.

12.3 Conclusiones

Con los resultados obtenidos en este capítulo se consideran cumplidos los objetivos de este proyecto en lo relativo a la implementación de un efecto de retardo modulado. Ambas versiones del software producen resultados satisfactorios y los tiempos de procesamiento se encuentran dentro de los límites aceptables.

CAPÍTULO 13

Introducción a los filtros WDF

En este capítulo he elaborado una introducción a la simulación de circuitos mediante filtros digitales de ondas o WDF (*Wave Digital Filters*). Esta introducción es un enfoque propio matemáticamente intuitivo. Para una introducción formal puede consultarse [57], por ejemplo. En la segunda parte de este capítulo he elaborado una introducción al método de interpolación de Fritsch–Carlson [58], cuya utilidad práctica en este trabajo resultará clara en los siguientes capítulos. Recojo de este modo en un solo capítulo toda la revisión bibliográfica restante como paso previo a las explicaciones sobre la implementación de la simulación del circuito en tiempo real.

13.1 Simulación de circuitos mediante filtros digitales de ondas

En el dominio W existen dos tipos de componentes, llamados en este trabajo elementos y adaptadores. Los elementos modelan componentes reales del circuito mientras que los adaptadores modelan conexiones en serie o en paralelo. Un elemento no es un adaptador pero un adaptador sí es un elemento, ya que puede a su vez conectarse a otro adaptador creando árboles de conexiones en serie y en paralelo.

Un elemento tiene un puerto que sirve para conectarlo a un adaptador. Un adaptador tiene dos o más puertos que sirven para conectar elementos entre ellos. A cada uno de los elementos se asigna una resistencia de puerto, R_p. Esta resistencia puede interpretarse como la impedancia que ve el adaptador al que está conectado el elemento.

Cuando un elemento se conecta a un adaptador la resistencia de puerto del adaptador en el puerto de conexión adquiere el valor de la resistencia de puerto del elemento.

Cada adaptador tiene un puerto cuya resistencia se adapta para que sea la suma de las resistencias de los elementos conectados al adaptador. Esto permite conectarlo a otro adaptador que verá la impedancia total a través del puerto de conexión.

Las siguientes secciones contienen una introducción más detallada a este tipo de filtros así

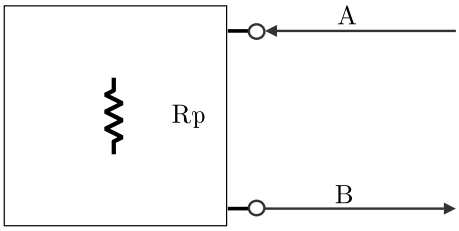


Figura 13.1: Símbolo de resistencia en el dominio W.

como una descripción de los adaptadores y elementos que serán usados en este trabajo.

13.1.1 Variables en el dominio W

A grandes rasgos, la idea es convertir las variables del análisis de circuitos clásico en el dominio de Kirchhoff a ondas de corriente y voltaje en el dominio W. Para ello se definen dos ondas por elemento, una de entrada y otra de salida, denotadas como A y B:

$$A = V + IR_p \tag{13.1}$$

$$B = V - IR_p \tag{13.2}$$

Donde R_p es una variable introducida en el sistema llamada “Resistencia de puerto”. Esta resistencia puede interpretarse como la impedancia visible para el adaptador al que está conectado el componente. Despejando de las ecuaciones anteriores se obtienen las transformaciones para pasar del dominio W al dominio K:

$$V = \frac{1}{2}(A + B) \tag{13.3}$$

$$I = \frac{1}{2R_p}(A - B) \tag{13.4}$$

En la figura 13.1 se muestra un ejemplo de una resistencia en el dominio W. La resistencia recibe una onda A y refleja una onda B. Para que una estructura WDF sea computable esta

no puede contener elementos no causales, es decir, la onda B reflejada puede depender de valores anteriores de la onda A pero nunca del valor actual o de valores futuros.

13.1.2 Discretización de elementos reactivos

Para elaborar una estructura WDF que simule un circuito es necesario discretizar sus elementos reactivos. En este trabajo se usa para ello la transformación bilineal:

$$s = \frac{2}{T}\frac{z-1}{z+1} \tag{13.5}$$

Donde T es el periodo de muestreo.

13.1.3 Resistencia

Tomando la ley de Ohm y las ecuaciones 13.1 y 13.2 es posible derivar una expresión para el valor de B:

$$R = \frac{V}{I} = \frac{\frac{1}{2}(A+B)}{\frac{1}{2R_p}(A-B)} \tag{13.6}$$

De donde se obtiene, despejando B:

$$B = \frac{R-R_p}{R+R_p}a \tag{13.7}$$

Y haciendo $R_p = R$:

$$B = 0 \tag{13.8}$$

Así, la onda B reflejada en una resistencia será siempre cero. Nótese que se ha logrado el objetivo de causalidad dando un valor adecuado a R_p.

El hecho de que la onda reflejada sea siempre 0 no quiere decir que la resistencia no afecte al resto de la estructura. Como se verá posteriormente, el valor que se elija para R_p tendrá

efectos en los coeficientes de dispersión de los adaptadores usados para unir elementos WDF.

13.1.4 Condensador

Describiendo la impedancia del condensador como una función de la variable de Laplace s podemos hacer un desarrollo análogo al anterior:

$$Z_c = \frac{1}{sC} = \frac{V}{I} = \frac{\frac{1}{2}(A+B)}{\frac{1}{2R_p}(A-B)} \tag{13.9}$$

Despejando B de la expresión anterior:

$$B = \frac{1-R_p sC}{1+R_p sC}A \tag{13.10}$$

Aplicando la transformación bilineal:

$$B = \frac{1-R_p\frac{2}{T}\frac{z-1}{z+1}C}{1+R_p\frac{2}{T}\frac{z-1}{z+1}C}A \tag{13.11}$$

Operando y dividiendo nominador y denominador entre z^{-1}:

$$B = \frac{T-2CR_p+(T+2CR_p)z^{-1}}{T+2CR_p+(T-2CR_p)z^{-1}}A \tag{13.12}$$

Dividiendo nominador y denominador entre el periodo de muestreo T:

$$B = \frac{(1-2F_sCR_p)+(1+2F_sCR_p)z^{-1}}{(1+2F_sCR_p)+(1-2F_sCR_p)z^{-1}}A \tag{13.13}$$

Donde F_s es la frecuencia de muestreo utilizada. Si transformamos esta expresión al

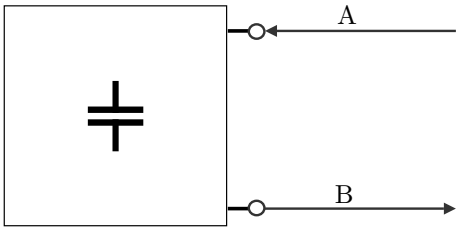


Figura 13.2: Símbolo de condensador en el dominio W.

dominio del tiempo queda:

$$B[n] = \frac{1}{1+2F_sCR_p}((1-2F_sCR_p)A[n]+(1+2F_sCR_p)A[n-1]-(1-2F_sCR_p)B[n-1]) \tag{13.14}$$

Para que la onda reflejada $B[n]$ no dependa de la onda incidente actual $A[n]$ es necesario hacer cero el término $A[n]$. Para ello se le da a R_p el valor:

$$R_p = \frac{1}{2F_sC} \tag{13.15}$$

Sustituyendo se obtiene la expresión final de la onda reflejada:

$$B = z^{-1}A \tag{13.16}$$

Es decir, la onda reflejada es la onda incidente retrasada una muestra. En la figura 13.2 se muestra el símbolo de un condensador en el dominio W.

13.1.5 Fuente de voltaje ideal

En una fuente de voltaje ideal el voltaje entre los dos terminales será igual al voltaje de la fuente, es decir:

$$V = V_s = \frac{1}{2}(A+B) \tag{13.17}$$

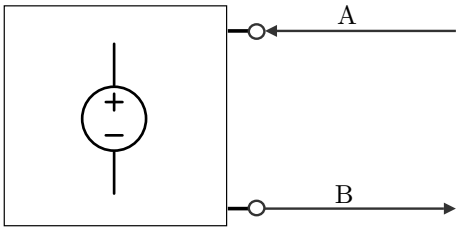


Figura 13.3: Símbolo de fuente de voltaje ideal en el dominio W.

Y por lo tanto:

$$B = 2V_s - A \tag{13.18}$$

En este caso no es posible lograr el objetivo de causalidad. Este problema se tratará posteriormente.

En la figura 13.3 se muestra el símbolo de una fuente de voltaje ideal en el dominio W.

13.1.6 Fuente de voltaje real

En una fuente de voltaje real el voltaje entre los dos terminales será igual al voltaje de la fuente ideal más el voltaje que caiga en la resistencia de la fuente, es decir:

$$V = V_s + R_s I \tag{13.19}$$

Sustituyendo las expresiones (13.3) y (13.4) y despejando B resulta:

$$B = 2V_s + \frac{2R_s}{2R_p}(A - B) - A \tag{13.20}$$

Si hacemos $R_p = R_s$ la onda B reflejada es:

$$B = V_s \tag{13.21}$$

En la figura 13.4 se muestra el símbolo de una fuente de voltaje real en el dominio W.

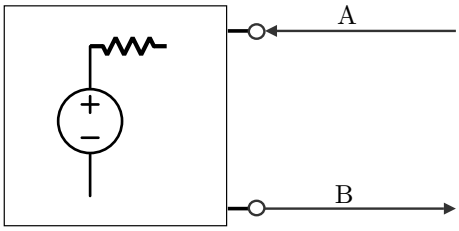


Figura 13.4: Símbolo de fuente de voltaje real en el dominio W.

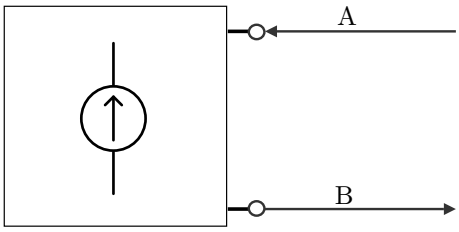


Figura 13.5: Símbolo de fuente de corriente ideal en el dominio W.

13.1.7 Fuente de corriente ideal

De acuerdo a la ecuación (13.4), en una fuente de corriente ideal se tiene:

$$I_s = \frac{1}{2R_p}(A - B) \tag{13.22}$$

Despejando B:

$$B = A - 2R_p I_s \tag{13.23}$$

Al igual que en la fuente de voltaje ideal, no es posible lograr el objetivo de causalidad.

En la figura 13.5 se muestra el símbolo de una fuente de corriente ideal en el dominio W.

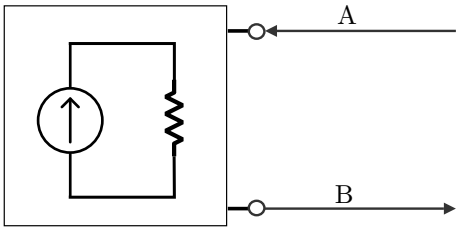


Figura 13.6: Símbolo de fuente de corriente real en el dominio W.

13.1.8 Fuente de corriente real

Podemos escribir una expresión para la onda reflejada si sustituimos en (13.23) I_s por la intensidad total, es decir:

$$B = A - 2R_p(\frac{V}{R_s} - I_s) \tag{13.24}$$

Donde V/R_s es la intensidad que circula por la rama de la resistencia interna de la fuente de corriente. Sustituyendo con (13.3):

$$B = A - \frac{R_p}{R_s}(A+B) + 2I_sR_p \tag{13.25}$$

Si hacemos $R_p = R_s$ en la expresión anterior obtenemos la expresión final para calcular una onda reflejada sin hacer uso de los valores de la onda incidente:

$$B = R_sI_s \tag{13.26}$$

En la figura 13.6 se muestra el símbolo de una fuente de corriente real en el dominio W.

13.1.9 Adaptador en serie

Siguiendo las leyes de Kirchhoff se tiene que para varios componentes conectados en serie se cumple:

$$V_1 + V_2 + ... + V_n = 0 \tag{13.27}$$

Donde V_i es la caída de potencial en cada componente. Además:

$$I_1 = I_2 = ... = I_n \tag{13.28}$$

Es decir, la misma corriente atraviesa todos los componentes. Despejando B de (13.4):

$$B = A - I2R_p \tag{13.29}$$

Y sustituyendo en (13.3):

$$V = A - IR_p \tag{13.30}$$

Sustituyendo esta última expresión en (13.27):

$$(A_1 - I_1R_{p1}) + (A_2 - I_2R_{p2}) + ... + (A_n - I_nR_{pn}) = 0 \tag{13.31}$$

Dado que todas las corrientes son iguales la anterior expresión puede convertirse en:

$$A_1 + A_2 + ... + A_n - I(R_{p1} + R_{p2} + ... + R_{pn}) = 0 \tag{13.32}$$

Y despejando I:

$$I = I_i = \frac{A_1 + A_2 + ... + A_n}{R_{p1} + R_{p2} + ... + R_{pn}} \tag{13.33}$$

Sustituyendo (13.4) en esta última expresión:

$$\frac{1}{2R_{pi}}(A_i - B_i) = \frac{A_1 + A_2 + ... + A_n}{R_{p1} + R_{p2} + ... + R_{pn}} \tag{13.34}$$

De donde puede finalmente despejarse B_i, la onda reflejada en un determinado puerto:

$$B_i = A_i - \frac{2R_{pi}(A_1 + A_2 + ... + A_n)}{R_{p1} + R_{p2} + ... + R_{pn}} \tag{13.35}$$

Nótese que la onda reflejada en un puerto del adaptador será la onda que incida en el componente que esté conectado a ese puerto.

En la última expresión puede definirse un coeficiente de dispersión γ como:

$$\gamma_i = -\frac{2R_{pi}}{R_{p1} + R_{p2} + ... + R_{pn}} \tag{13.36}$$

Quedando la expresión de la onda reflejada como sigue:

$$B_i = A_i + \gamma_i(A_1 + A_2 + ... + A_n) \tag{13.37}$$

Normalmente, los adaptadores en una estructura WDF forman un árbol en el que cada adaptador se conecta a un adaptador de nivel superior usando uno de sus puertos. En el caso de un adaptador en serie la impedancia de este puerto debe adaptarse para que sea la suma de las impedancias del resto de los puertos, de tal forma que la impedancia observada por el adaptador de nivel superior sea la impedancia total de los elementos en serie:

$$R_{pn} = R_{p1} + R_{p2} + ... + R_{pn-1} \tag{13.38}$$

Donde R_{pn} es la impedancia del puerto que se adapta. Nótese que en este caso el coeficiente de dispersión del puerto adaptado resulta ser:

$$\gamma_n = -\frac{2R_{pn}}{R_{p1} + R_{p2} + ... + R_{pn}} = -\frac{2R_{pn}}{2R_{pn}} = -1 \tag{13.39}$$

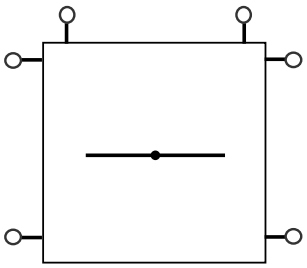


Figura 13.7: Símbolo de adaptador en serie de tres puertos en el dominio W.

Y por lo tanto la onda reflejada queda en el puerto adaptado como sigue:

$$B_n = -A_1 - A_2 - ... - A_{n-1} \tag{13.40}$$

Nótese que ahora la onda reflejada no depende en absoluto de la onda incidente en este puerto, sino tan solo de la onda incidente en todos los demás. Gracias a esto una onda puede propagarse hacia arriba en un árbol de adaptadores, como se verá posteriormente.

En al figura 13.7 se muestra el símbolo de un adaptador en serie de tres puertos en el dominio W.

13.1.10 Adaptador en paralelo

Siguiendo las leyes de Kirchhoff se tiene que para varios componentes conectados en paralelo se cumple:

$$V_1 = V_2 = ... = V_n = V \tag{13.41}$$

Donde V_i es la caída de potencial en cada componente. Además:

$$I_1 + I_2 + ... + I_n = 0 \tag{13.42}$$

Es decir, la suma de corrientes es cero. Despejando I de (13.30):

$$I_i = \frac{1}{R_{pi}}(V - A_i) \tag{13.43}$$

Sustituyendo en (13.42):

$$\frac{1}{R_{p1}}(V - A_1) + \frac{1}{R_{p2}}(V - A_2) + ... + \frac{1}{R_{pn}}(V - A_n) = 0 \tag{13.44}$$

Despejando V:

$$V = \frac{\frac{A_1}{R_{p1}} + \frac{A_2}{R_{p2}} + ... + \frac{A_n}{R_{pn}}}{\frac{1}{R_{p1}} + \frac{1}{R_{p2}} + ... + \frac{1}{R_{pn}}} \tag{13.45}$$

Sustituyendo esta última expresión en (13.3) se tiene:

$$\frac{1}{2}(A_i + Bi) = \frac{\frac{A_1}{R_{p1}} + \frac{A_2}{R_{p2}} + ... + \frac{A_n}{R_{pn}}}{\frac{1}{R_{p1}} + \frac{1}{R_{p2}} + ... + \frac{1}{R_{pn}}} \tag{13.46}$$

De donde finalmente se obtiene despejando una expresión para la onda reflejada en un puerto del adaptador:

$$Bi = \frac{2(\frac{A_1}{R_{p1}} + \frac{A_2}{R_{p2}} + ... + \frac{A_n}{R_{pn}})}{\frac{1}{R_{p1}} + \frac{1}{R_{p2}} + ... + \frac{1}{R_{pn}}} - A_i \tag{13.47}$$

Al igual que con el adaptador en serie se define un coeficiente de dispersión γ como:

$$\gamma_i = \frac{2\frac{1}{R_{pi}}}{\frac{1}{R_{p1}} + \frac{1}{R_{p2}} + ... + \frac{1}{R_{pn}}} \tag{13.48}$$

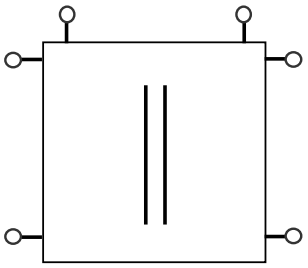


Figura 13.8: Símbolo de adaptador en paralelo de tres puertos en el dominio W.

Quedando ahora la expresión de la onda reflejada como sigue:

$$Bi = \gamma_1 A_1 + \gamma_2 A_2 + ... + \gamma_n A_n - A_i \tag{13.49}$$

Si se quiere adaptar la impedancia de un puerto para que un adaptador de nivel superior visualice la impedancia total del adaptador, tal y como se hizo con el adaptador en serie, debemos sumar las impedancias del resto de los puertos en paralelo, es decir:

$$R_{pn} = \frac{1}{\frac{1}{R_{p1}} + \frac{1}{R_{p2}} + ... + \frac{1}{R_{pn-1}}} \tag{13.50}$$

En este caso, el coeficiente de dispersión en el puerto adaptado será:

$$\gamma_n = 1 \tag{13.51}$$

Y por lo tanto la onda reflejada será:

$$B_n = \gamma_1 A_1 + \gamma_2 A_2 + ... + \gamma_{n-1} A_{n-1} \tag{13.52}$$

Una vez más, se obtiene un puerto libre de reflexión, es decir, independiente de la onda incidente en ese puerto. En la figura 13.8 se muestra el símbolo de un adaptador en paralelo de tres puertos en el dominio W.

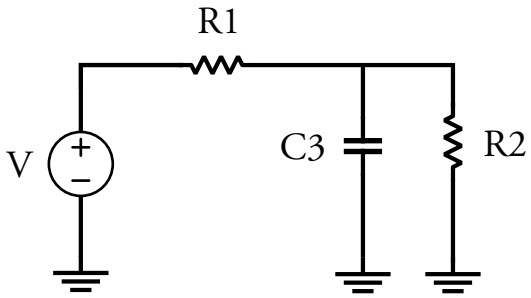


Figura 13.9: Filtro paso bajo con carga en el dominio K.

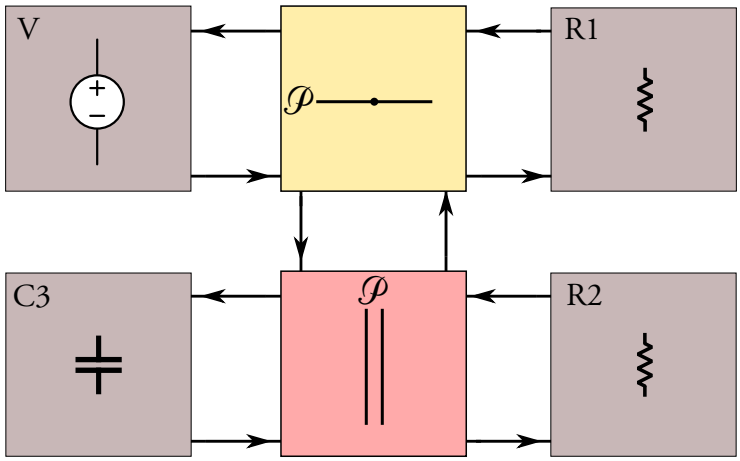


Figura 13.10: Filtro paso bajo con carga en el dominio W.

13.1.11 Ejemplo

En las figuras 13.9 y 13.10 se muestra un filtro paso bajo con una resistencia de carga en su salida representado en los dominios K y W.

R2 y C3 están conectados en paralelo, por lo que ambos se conectan a un adaptador en paralelo de tres puertos en la estructura WDF. El puerto con impedancia adaptada del adaptador se indica con una $\mathscr{P}$ en el gráfico. Este puerto se usa para conectar otro adaptador al que a su vez se conectan los elementos en serie del circuito, R1 y V.

Para computar una estructura WDF se usa un procesamiento en forma de árbol. Las hojas son los nodos terminales del gráfico, los nodos intermedios son los adaptadores y la raíz se escoge de tal modo que la estructura sea computable. En este caso, es escoge como raíz la fuente ideal de voltaje. Los pasos a seguir son:

- Propagar las ondas reflejadas hacia arriba en el árbol desde las hojas; en este caso C3,R2 y R1. Dado que ambos adaptadores tienen un puerto con impedancia adaptada no necesitamos conocer el valor de la onda que viaja en sentido opuesto para computar la propagación.

- Calcular la onda reflejada en la raíz. Nótese que la raíz puede ser un elemento no causal, ya que se conoce el valor de la onda incidente en el momento de calcular la onda reflejada.
- Propagar la onda desde la raíz hasta las hojas a través de los adaptadores.

Usando las expresiones (13.3) y (13.4) en cualquier momento es posible conocer el voltaje o la intensidad en uno de los componentes del circuito.

Este tipo de estructuras permiten introducir en el nodo raíz del árbol elementos no lineales tal y como se ha hecho con la fuente de voltaje ideal.

Los diodos son un ejemplo de resistencia no lineal. Dado que la resistencia de un diodo depende del voltaje entre sus extremos no es posible construir un puerto libre de reflexión instantánea.

Aunque quedan fuera del marco de este trabajo, existen actualmente formas de computar árboles WDF con múltiples elementos no lineales [59].

13.2 Interpolación de Fritsch-Carlson

En esta sección se presenta una revisión del método de interpolación de Fritsch-Carlson y del algoritmo propuesto en el documento original [58]. Este método produce una función de interpolación cúbica y monótona que se ajusta a un conjunto de datos de entrada monótono. El interés para este trabajo radica en que permite aproximar razonablemente bien curvas corriente-voltaje, evitando las ondulaciones típicas de otros métodos de interpolación como los splines al asegurar la monotonía del interpolador en cada subintervalo.

Hagamos $\pi : a = x_1 < x_2 < ... < x_n = b$ una partición del intervalo $[a,b]$. Hagamos $f_i : i = 1,2,...,n$ un conjunto de datos monótono en los puntos de partición. El objetivo es construir en π una función cúbica a trozos $p(x)$ tal que:

$$p(x_i) = f_i \quad \text{i=1,2,...,n} \tag{13.53}$$

En cada subintervalo $[x_i, x_{i+1}]$, $p(x)$ será un polinomio cúbico. Para desarrollar una expresión que lo defina denotemos las derivadas de este polinomio en los extremos del

subintervalo como d_i y d_{i+1}. Ahora hagamos un cambio de variable como sigue:

$$p(t) = at^3 + bt^2 + ct + d \tag{13.54}$$

donde a,b,c y d son los coeficientes del polinomio tras el cambio de variable y

$$x = t(x_{i+1} - x_i) + x_i \tag{13.55}$$

De tal forma que t toma valores en el intervalo $[0,1]$ y la función anterior los transforma en un punto del subintervalo $[x_i, x_{i+1}]$.

Construyamos ahora el siguiente sistema de ecuaciones para obtener a,b,c y d:

$$\begin{cases} p(0) = d = f_i \\ p'(0) = d_i = c \\ p(1) = a + b + c + d = f_{i+1} \\ p'(1) = d_{i+1} = 3a + 2b + c \end{cases} \tag{13.56}$$

Nótese que el cambio de variable simplifica mucho los cálculos al corresponder los extremos del subintervalo con los valores 0 y 1. Resolviendo el anterior sistema de ecuaciones se llega a que:

$$\begin{aligned} a &= 2f_i - 2f_{i+1} + d_i + d_{i+1} \\ b &= -3f_i + 3f_{i+1} - 2d_i - d_{i+1} \\ c &= d_i \\ d &= f_i \end{aligned} \tag{13.57}$$

Con estos resultados el polinomio puede expresarse en forma matricial como sigue:

$$p(t) = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix} \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} f_i \\ f_{i+1} \\ d_i \\ d_{i+1} \end{pmatrix} \tag{13.58}$$

Para realizar el cambio de variable a la inversa hay que tener en cuenta que:

$$p(x) = a\left(\frac{x-x_i}{x_{i+1}-x_i}\right)^3 + b\left(\frac{x-x_i}{x_{i+1}-x_i}\right)^2 + c\left(\frac{x-x_i}{x_{i+1}-x_i}\right) + d \tag{13.59}$$

Si hacemos $h_i = x_{i+1} - x_i$ y derivamos respecto a x resulta que:

$$p'(x) = \left(3a\left(\frac{x-x_i}{x_{i+1}-x_i}\right)^2 + 2b\left(\frac{x-x_i}{x_{i+1}-x_i}\right) + c\right)\frac{1}{h_i} = p'(t)\frac{1}{h_i} \tag{13.60}$$

Es decir, la derivada de la función es escalada un factor $1/h_i$ al realizar el cambio de variable, hecho que es necesario compensar multiplicando las derivadas por h_i. Teniendo esto en cuenta puede escribirse una expresión final para el polinomio en forma matricial:

$$p(x) = \begin{pmatrix} \left(\frac{x-x_i}{x_{i+1}-x_i}\right)^3 & \left(\frac{x-x_i}{x_{i+1}-x_i}\right)^2 & \left(\frac{x-x_i}{x_{i+1}-x_i}\right) & 1 \end{pmatrix} \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} f_i \\ f_{i+1} \\ d_i h_i \\ d_{i+1} h_i \end{pmatrix}$$

Denotemos $\Delta_i = (f_{i+1} - f_i)/h_i$. Desarrollando el anterior producto de matrices se llega a que:

$$p(x) = \left[\frac{d_i + d_{i+1} - 2\Delta_i}{h_i^2}\right](x - x_i^3) + \left[\frac{-2d_i - d_{i+1} + 3\Delta_i}{h_i^2}\right](x - x_i^2) + d_i(x - x_i) + f_i$$

Se ha presentado hasta este punto una derivación completa del polinomio que se examina

en el documento de Fritsch y Carlson y se ha expresado aquí con la misma notación que se usa en el documento original. El examen es relativamente sencillo y consiste en estudiar las condiciones para que el polinomio sea monótono utilizando las derivadas primera y segunda.

Denotemos $\alpha_i = d_i/\Delta_i$ y $\beta_i = d_{i_{i+1}}/\Delta_i$. La conclusión a la que se llega es que existe una región claramente definida en el plano (α, β), a la que se llama $\mathscr{M}$, dentro de la cual el polinomio en el subintervalo es monótono creciente o decreciente. Un subconjunto de esta región es la intersección del círculo con centro en el origen y radio 3 con el primer cuadrante, al que se llama $\mathscr{S}_2$. Si α_i o β_i son negativas, entonces los valores de entrada f_i no son estrictamente monótonos.

El procedimiento para obtener un interpolador monótono a trozos queda por tanto reducido a ajustar d_i y d_{i+1} en cada subintervalo de tal forma que el punto (α_i,β_i) se encuentre dentro de la región de monotonía. Para ello se propone el siguiente algoritmo en el documento original:

1. Inicializar las derivadas d_i de tal forma que $sgn(d_i) = sgn(d_{i+1}) = sgn(\Delta_i)$. Si $\Delta_i = 0$, $d_i = d_{i+1} = 0$.
2. Para cada subintervalo I_i en el que $(\alpha_i, \beta_i) \notin \mathscr{M}$, modificar d_i y d_{i+1} a d_i^* y d_{i+1}^* de tal forma que $(\alpha_i^*, \beta_i^*) \in \mathscr{M}$, donde $\alpha_i^* = d_i^*/\Delta_i$ y $\beta_i^* = d_{i_{i+1}}^*/\Delta_i$
   Para implementar este segundo paso se remarcan en el documento original las interacciones entre los intervalos adyacentes, es decir, $\beta_{i-1}\Delta_{i-1} = d_i = \alpha_i\Delta_i$. Al modificar d_i para producir monotonía en I_i también estamos modificando β_{i-1} y esto debe ser tenido en consideración para preservar la monotonía en I_{i-1}. Una forma de conseguir esto es seleccionar un subconjunto $\mathscr{S} \subset \mathscr{M}$ tal que:
   a) Si $(\alpha_i, \beta_i) \in \mathscr{S}$, entonces $(\alpha_i^*, \beta_i^*) \in \mathscr{S}$ siempre que $0 \leq \alpha^* \leq \alpha$ y $0 \leq \beta^* \leq \beta$.
   b) Si $(\alpha_i, \beta_i) \in \mathscr{S}$ entonces $(\beta_i, \alpha_i) \in \mathscr{S}$
   Aunque la segunda propiedad de simetría no es esencial, está presente en $\mathscr{M}$ y parece ser intuitivamente deseable. A la luz de lo expuesto anteriormente, este paso puede convertirse en:

2A. Para cada subintervalo I_i en el que $(\alpha_i, \beta_i) \notin \mathscr{S}$, modificar d_i y d_{i+1} a d_i^* y d_{i+1}^* de tal forma que $0 \leq \alpha^* \leq \alpha$, $0 \leq \beta^* \leq \beta$ y $(\alpha_i^*, \beta_i^*) \in \mathscr{S}$.

Un procedimiento para modificar las derivadas en el paso 2A consiste en construir la linea que une el origen con el punto (α_i, β_i) y hacer el punto (α_i^*, β_i^*) la intersección de esta linea con los límites de $\mathscr{S}$. Entonces, $d_i^* = \alpha_i^*\Delta_i$ y $d_{i+1}^* = \beta_i^*\Delta_i$. Para $\mathscr{S} = \mathscr{S}_2$, $\alpha_i^* = \tau_i\alpha_i$, $\beta_i^* = \tau_i\beta_i$, donde $\tau_i = 3(\alpha_i^2 + \beta_i^2)^{-1/2}$.

13.3 Conclusiones

En la primera parte de este capítulo se ha presentado una aproximación matemáticamente intuitiva a la simulación de circuitos electrónicos mediante filtros digitales de ondas. En la segunda parte se ha presentado una revisión del método de interpolación de Fritsch-Carlson, que será usado en posteriores capítulos para aproximar curvas corriente-voltaje.

CAPÍTULO 14

Análisis del efecto zendrive

Este capítulo pertenece al proceso de análisis del efecto zendrive. En él hago un breve análisis del circuito del efecto zendrive. Este es un paso previo a la conversión del circuito en una estructura WDF que ayuda a entender mejor su funcionalidad. El código Scilab mostrado puede encontrarse en la ruta *./Proyecto/Saturación/Diseño*.

14.1 Análisis del circuito

En la figura 14.1 se muestra el esquema del circuito a simular. Este esquema ha sido dibujado a partir de la información obtenida en foros de internet proporcionada por gente que ha intentado replicar el circuito original. Es normal que los creadores de este tipo de circuitos eliminen mediante raspado la serigrafía de los componentes, en especial de los circuitos impresos, en parte para dificultar las copias de los diseños originales.

Este diseño es muy simular al del efecto *Tube Screamer*, llamado así por la similitud del sonido resultante con la saturación de un tubo de vacío. La estrategia en estos circuitos consiste en introducir elementos limitadores no lineales en el lazo de realimentación del amplificador operacional, produciendo una caída en la ganancia de la etapa a medida que aumenta la diferencia de potencial entre la entrada inversora y la salida. Esto produce evidentemente una deformación de la onda de entrada, como se verá a continuación.

14.1.1 Fuente de alimentación y etapa de entrada

La fuente de alimentación se muestra en la parte superior derecha. La entrada de 9 voltios es usada para proporcionar un segundo voltaje V2 de 4.5 voltios mediante un divisor de tensión formado por R1 y R2. Los condensadores C1 y C2 son condensadores de desacoplo que eliminan el posible rizado.

Hacia la parte izquierda se encuentra la etapa de entrada. La entrada no inversora del amplificador operacional es polarizada usando el voltaje V2. El condensador C6 y la

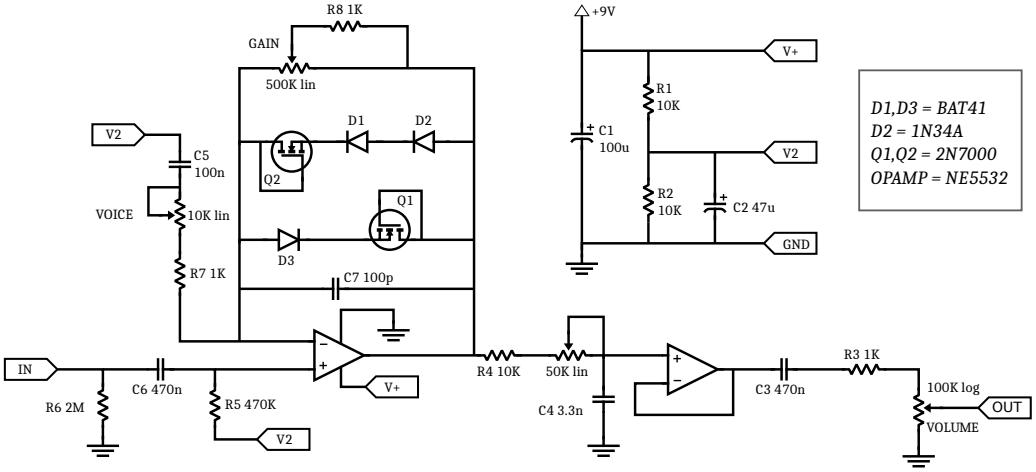


Figura 14.1: Clon del efecto zendrive.

resistencia R5 forman un filtro paso alto cuya frecuencia de corte será aproximadamente:

$$f_{-3db} = \frac{1}{2\pi R5C6} = 0.72Hz \tag{14.1}$$

Es evidente que el circuito ha sido diseñado para que esta primera etapa no interfiera con la magnitud de ninguna frecuencia de interés. En otros circuitos similares la frecuencia de corte es algo superior, sobre los 15 o 20 Hz. El condensador C6 también funciona como condensador de acoplo, bloqueando cualquier componente de continua en ambas direcciones y protegiendo el circuito de entrada de posibles fallos.

En cuanto a la resistencia R6, su función no es evidente. Este tipo de circuitos son frecuentemente conectados y desconectados. En teoría, el condensador C6 bloquea totalmente las componentes continuas de voltaje. Sin embargo, en el mundo real muchos condensadores presentan fugas y un pequeño voltaje puede aparecer en la entrada del circuito. Al conectar a la entrada un circuito con una baja impedancia de salida y sin ninguna polarización en continua, como es el caso de las pastillas de guitarra, existe el riesgo de que este pequeño voltaje se descargue a través del camino de la señal. Dado que la salida del circuito suele estar conectada a amplificadores con ganancias muy elevadas esto suele llevar a un chasquido desagradable en el altavoz. El objetivo de la resistencia R6 es precisamente proporcionar un camino de descarga para el voltaje que aparece en la entrada debido a posibles fugas en el condensador de acoplo.

14.1.2 Amplificador limitador

Esta etapa es la parte más importante del circuito. Está formada por un amplificador no inversor, los componentes limitadores de ganancia y dos filtros en el lazo de realimentación; uno paso bajo y otro paso alto.

En primer lugar nótese que C5 está conectado al voltaje V2. Dado que todo el voltaje continuo cae en el condensador podemos considerar a efectos de polarización que esta rama está conectada a tierra. La ventaja es que el ruido presente en el voltaje de alimentación aparecerá tanto en la entrada inversora como en la no inversora y puesto que la ganancia en modo común es mínima esto ayudará a reducir el ruido en la salida.

Ignoremos por el momento las ramas con diodos y analicemos la etapa como un amplificador no inversor con impedancia compleja en la rama de realimentación. Denotemos:

$$R_G = \frac{1}{\frac{1}{500000a} + \frac{1}{1000}} + 500000(1-a) \tag{14.2}$$

Esta es la resistencia total de la rama del potenciómetro Gain, donde el parámetro a es un valor entre 0 y 1 que determina la posición del potenciómetro. Si $a = 1$ entonces $R_G \approx 1000$. Si $a = 0$ entonces $R_G = 500000$.

Para obtener la impedancia total sumamos a R_G la impedancia en paralelo del condensador C7. Denotamos esta impedancia total como:

$$Z_r = \frac{1}{\frac{1}{Rg} + \frac{1}{\frac{-j}{2\pi f C7}}} \tag{14.3}$$

Por otro lado, en la rama del condensador C5 tenemos que la impedancia total es:

$$Z_v = Rv - \frac{j}{2\pi f C5} \tag{14.4}$$

Donde R_v es la suma del valor del potenciómetro Voice y la resistencia R7. Haciendo las típicas suposiciones de ganancia infinita [60, pp. 64] la ganancia de esta etapa sin tener en

```
//Variación de ganancia
f = 1:1:20000;
C5=100*10^-9;
C7=100*10^-12;
Rg= 100000;
Rv= 6000;
for j = 1:5;
Zr(j,:) = 1./( 1/(Rg*j) + 1./(-%i./(2*%pi*f*C7) ) );
end;
Zv= Rv - %i./(2*%pi*f*C5);
for j = 1:5;
G(j,:) = 1+Zr(j,:)./Zv;
end;
mag = abs(G);
subplot(211);
xset('font size',3)
ylabel("Ganancia","fontsize",3);
xlabel("Frecuencia (Hz)","fontsize",3);
plot(f,mag);
hl=legend(['Rg = 100K';'Rg = 200K';'Rg = 300K';'Rg =
   400K';'Rg = 500K'],'in_upper_right');
arg = atan(imag(G),real(G));
phi = arg*180 ./%pi;
subplot(212);
xset('font size',3)
ylabel("Fase(Grados)","fontsize",3);
xlabel("Frecuencia (Hz)","fontsize",3);
plot(f,phi);
```

Listado 14.1: Análisis de la variación del potenciómetro Gain en el efecto zendrive

cuenta los diodos y los MOSFETS será:

$$G = 1 + \frac{Z_r}{Z_v} \tag{14.5}$$

Para visualizar como afecta la red de resistencias y condensadores a la ganancia sin tener en cuenta las ramas con diodos he escrito el código en Scilab de los listados 14.1 y 14.2. Este código calcula la ganancia mediante las ecuaciones anteriores usando diferentes valores para cada potenciómetro y posteriormente separa la información en magnitud y fase. El resultado de la ejecución puede verse en las figuras 14.2 y 14.3. En el caso de la primera figura el valor de Rv se fija en 6K, mientras que en el caso de la segunda el valor de Rg se

```
scf(1);
//Variación de voz
f = 1:1:20000;
C5=100*10^-9;
C7=100*10^-12;
Rg= 250000;
Rv= 1000;
Zr = 1./( 1/(Rg) + 1./(-%i./(2*%pi*f*C7) ) );
for j = 1:5;
Zv(j,:) = Rv*j*2 - %i./(2*%pi*f*C5);
end;
for j = 1:5;
G(j,:) = 1+Zr./Zv(j,:);
end;
mag = abs(G);
subplot(211);
xset('font size',3)
ylabel("Ganancia","fontsize",3);
xlabel("Frecuencia (Hz)","fontsize",3);
plot(f,mag);
hl=legend(['Rv= 2K';'Rv = 4K';'Rv = 6K';'Rv = 8K';'Rv =
    10K'],'in_upper_right');
arg = atan(imag(G),real(G));
phi = arg*180 ./%pi;
subplot(212);
xset('font size',3)
ylabel("Fase(Grados)","fontsize",3);
xlabel("Frecuencia (Hz)","fontsize",3);
plot(f,phi);
```

Listado 14.2: Análisis de la variación del potenciómetro Voice en el efecto zendrive

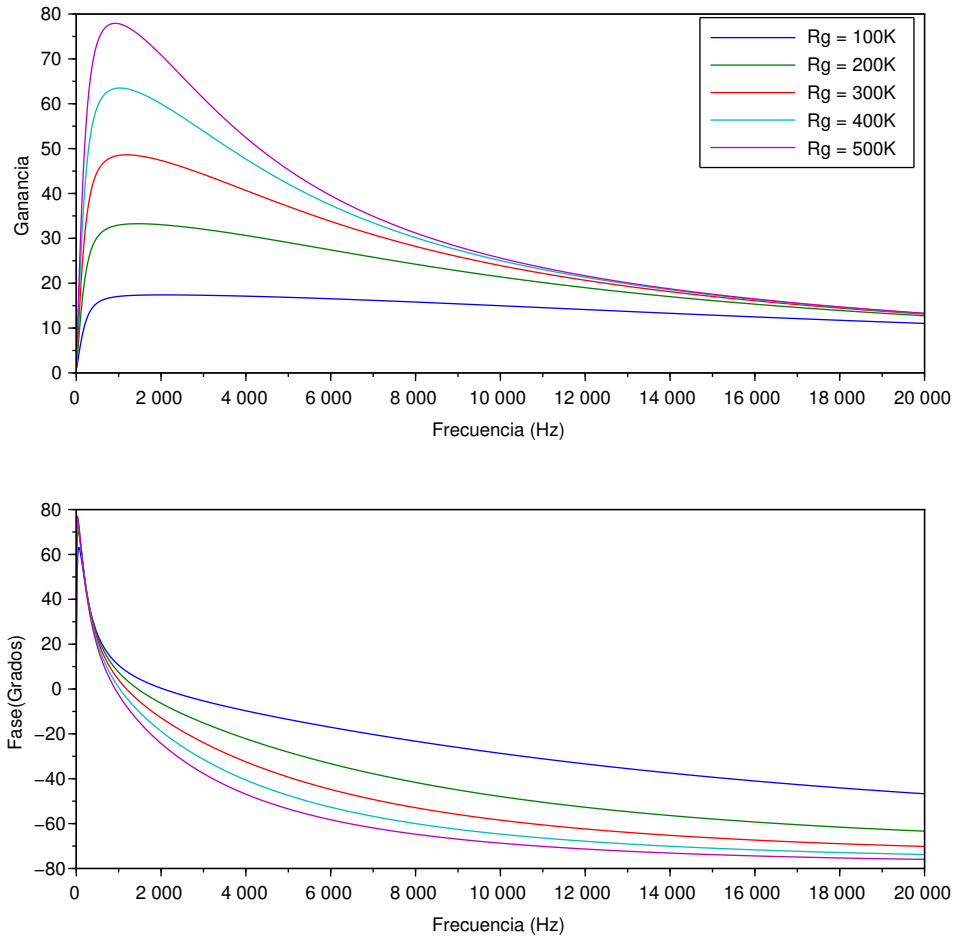


Figura 14.2: Efecto del potenciómetro Gain en el efecto zendrive.

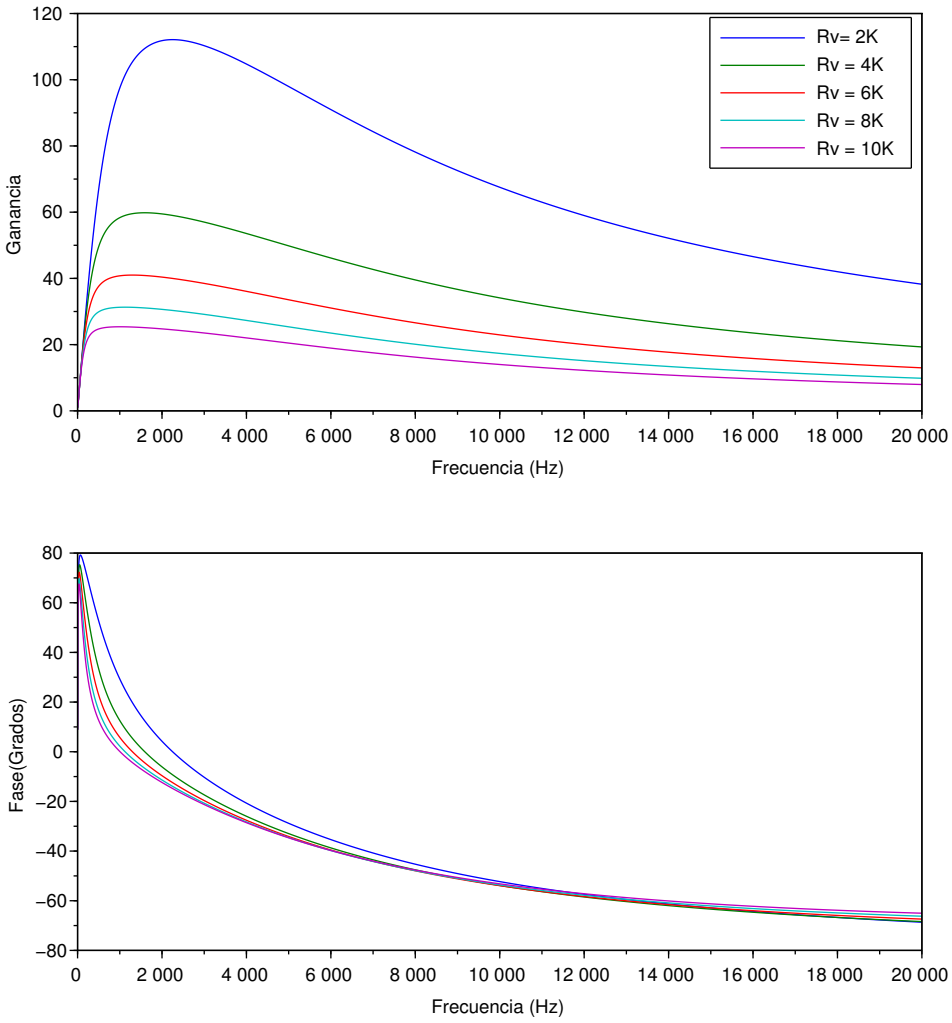


Figura 14.3: Efecto del potenciómetro Voice en el efecto zendrive.

fija en 250K.

El potenciómetro de ganancia afecta más a las frecuencias bajas y medias debido al efecto del condensador C7. También tiene un cierto efecto sobre la curva de fase. El potenciómetro Voice altera la ganancia de forma más uniforme y también varía la frecuencia de corte del filtro paso alto, tal y como puede verse en el gráfico.

14.1.3 Efecto de los diodos y MOSFETs

Añadamos ahora los diodos y los MOSFETs al análisis. Lo primero que hay que considerar es que en este circuito los MOSFETs actúan como resistencias no lineales. Habitualmente se usa la siguiente ecuación para aproximar la corriente en el drenador de un MOSFET:

$$i_d = k_n(V_{GS} - V_T - \frac{1}{2}V_{DS})V_{DS} \tag{14.6}$$

Donde i_d es la corriente en el drenador, k_n es una constante dependiente de la geometría del transistor y V_T la tensión umbral que induce la carga necesaria para producir un canal entre drenador y fuente. Una derivación completa de este modelo a partir de la geometría del transistor así como una descripción de las diferentes regiones de trabajo de un MOSFET puede encontrarse en un video de la Dra. Cristina Crespo [61].

Si consideramos que el drenador está conectado a la puerta entonces $V_{DS} = V_{GS}$ y por lo tanto siempre que $V_{GS} \geq V_T$ se cumple la condición de saturación $V_{DS} \geq V_{GS} - V_T$. En este caso la ecuación (14.6) se convierte en:

$$i_d = \frac{1}{2}k_n(V_{GS} - V_T)^2 \tag{14.7}$$

Suele llamarse a un transistor conectado de este modo “transistor conectado como diodo”, aunque la curva corriente-voltaje es más cuadrática que exponencial [60, pp. 312].

Los diodos y MOSFETs tienen el efecto de limitar la ganancia del amplificador, como se mencionó anteriormente. Cuando la ganancia supere un cierto umbral la diferencia de potencial entre la entrada inversora y la salida será alta y la resistencia equivalente en la rama de los diodos caerá. Debido a ello la ganancia disminuirá enormemente, como se deduce al sustituir Z_r por 0 en la expresión (14.5).

Por otro lado, dado que la etapa amplificadora altera la fase, la limitación de ganancia

no coincidirá con la forma de la onda de salida sino que estará desfasada en función de la frecuencia. Dicho de otro modo, la máxima diferencia de potencial entre la entrada inversora y la salida no se dará necesariamente en el instante en que más alto sea el voltaje de salida ya que los voltajes en ambos puntos estarán fuera de fase. En una onda sinusoidal esta limitación fuera de fase produce un resultado en forma de "cresta de ola" que también es típica del efecto *Tube Screamer*.

Por último, nótese que la limitación de ganancia es asimétrica. El BAT41 es un diodo de silicio mientras que el 1N34A es un diodo de germanio añadido en la rama superior, lo que incrementa el voltaje a partir del cual la ganancia empieza a ser limitada. Esta forma de distorsión asimétrica tiende a crear formas de onda con más armónicos de orden impar.

El uso de MOSFETs conectados de este modo para limitar la ganancia de un amplificador es algo que se viene haciendo desde hace años con este tipo de efectos. La curva corriente-voltaje es más suave que la de un diodo por lo que las variaciones de ganancia son menos bruscas. En la práctica, esto produce un efecto sobre la onda de entrada más suave y "redondo", con más presencia de armónicos de orden par. Suele decirse que estos suenan en general mejor o menos discordantes, ya que se corresponden con las octavas de una nota musical. Aunque esto es cierto en lo relativo a la reproducción de piezas de música, en realidad tanto los armónicos pares como los impares tienen su importancia en los sonidos de guitarra eléctrica.

14.1.4 Control de tono

El control de tono está formado por la resistencia R4, el condensador C4 y un potenciómetro. Es en realidad un filtro paso bajo que atenúa la intensidad de los armónicos creados por el amplificador-limitador a partir de una cierta frecuencia. El filtro está seguido de un seguidor de tensión que lo aísla de la salida. Con el potenciómetro al mínimo su frecuencia de corte será:

$$f_{-3db} = \frac{1}{2\pi 10KC4} \approx 4823Hz \tag{14.8}$$

Mientras que con el potenciómetro al máximo:

$$f_{-3db} = \frac{1}{2\pi 60KC4} \approx 804Hz \tag{14.9}$$

14.1.5 Control de volumen

El control de volumen es nuevamente un condensador de acoplo que forma un filtro paso bajo con la resistencia R3 y el potenciómetro. Dado que este tipo de efectos suelen estar conectados en su salida a amplificadores o a efectos similares con una muy alta impedancia de entrada a efectos prácticos puede suponerse una impedancia de entrada infinita en el siguiente circuito de la cadena.

La frecuencia de corte será aproximadamente:

$$f_{-3db} = \frac{1}{2\pi(R3 + 100000)C3} \approx 3.35Hz \tag{14.10}$$

Al igual que en la etapa de entrada, el filtro no afecta a la magnitud de ninguna frecuencia de interés.

El potenciómetro logarítmico funciona como un divisor de tensión que atenúa el voltaje de salida.

14.2 Conclusiones

Se ha presentado en este capítulo un análisis completo del circuito que se pretende simular mediante filtros digitales de ondas. El análisis se ha centrado en la etapa limitadora. Para visualizar el efecto de los potenciómetros se han graficado diferentes curvas de ganancia y fase para diferentes valores de impedancia en el lazo de realimentación.

CAPÍTULO 15

Diseño del plugin oakdrive

He decidido llamar “Oakdrive” a mi versión en forma de plugin del efecto zendrive. En este capítulo construyo un esquema para simular el circuito en SPICE. Posteriormente, convierto el esquema en una estructura WDF y explico el método utilizado para simular las ramas con MOSFETs y diodos. Los archivos de LTSpice utilizados pueden encontrarse en la ruta *./Proyecto/Saturación/SPICE*.

15.1 Modelo SPICE

Para simular el circuito supondré que los amplificadores operacionales son ideales. Haciendo esta suposición el voltaje en la entrada inversora será igual al de la entrada no inversora y la impedancia de entrada será en ambos casos infinita.

El esquema de la primera etapa se muestra en la figura 15.1. E1 es una fuente de voltaje controlada por voltaje con ganancia 1 que toma como entrada el voltaje que cae en R5 sumado al voltaje de polarización V2. Esta fuente de voltaje está conectada en serie con el condensador C5, la resistencia R7 y el potenciómetro de voz. R_v representa en el esquema la suma de la resistencia del potenciómetro de voz y R7.

Es posible modelar el lazo de realimentación como una fuente de corriente en paralelo con

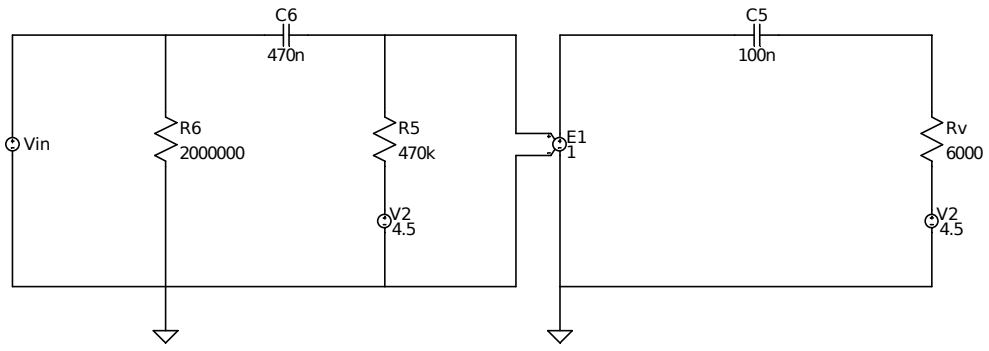


Figura 15.1: Etapa de entrada del efecto zendrive para simulación en SPICE.

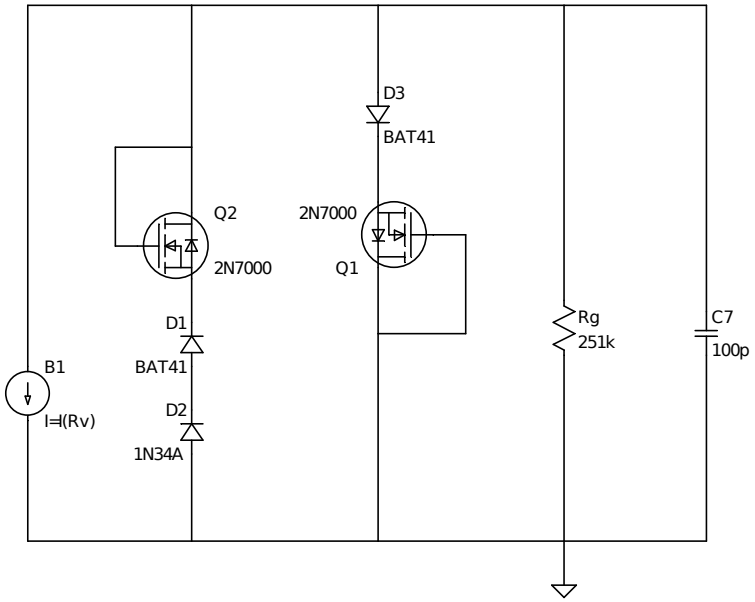


Figura 15.2: Lazo de realimentación en el amplificador operacional del efecto zendrive para simulación en SPICE.

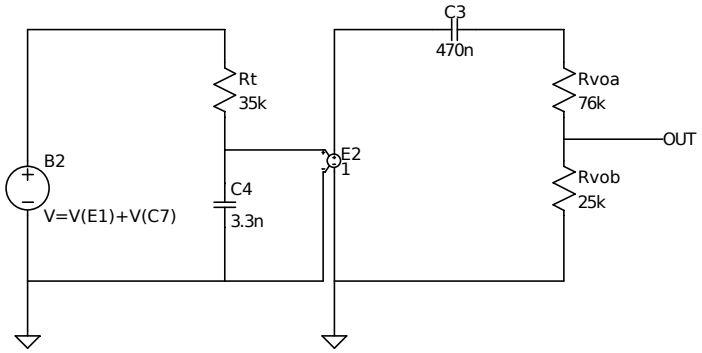


Figura 15.3: Etapa de salida del efecto zendrive para simulación en SPICE.

todos los componentes del lazo. El esquema se muestra en la figura 15.2. B1 es una fuente de corriente que toma el valor de la intensidad que circula por R_v. Se ha llamado R_g a la suma de la resistencia del potenciómetro de ganancia y R8.

El voltaje de salida del amplificador operacional será igual al voltaje en la entrada inversora sumado al voltaje que cae en el modelo del lazo de realimentación de la figura 15.2. Este voltaje de salida puede aplicarse al filtro paso bajo formado por R4,C4 y el potenciómetro de tono. Finalmente, el voltaje que cae en C4 puede aplicarse a la etapa de salida formada por C3,R3 y el potenciómetro de volumen.

El esquema se muestra en la figura 15.3. B2 es una fuente de voltaje cuya salida es la suma de V(E1), el voltaje en la entrada inversora, y V(C7), el voltaje que cae en el lazo de realimentación negativa.Se ha llamado R_t a la suma de la resistencia del potenciómetro de

tono y R4. Las resistencias R_{voa} y R_{vob} forman el divisor de tensión de la etapa de salida.

El circuito se ha dibujado en el capturador de esquemas de LTSpice XVII. Si se pulsa la tecla Ctrl y se hace clic derecho en la fuente de voltaje de entrada es posible introducir en el campo de configuración *value* una linea de tipo $wavefile = test.wav$. Esto permite usar un archivo wav como entrada para la fuente de voltaje.

También es posible hacer algo similar con el nodo de salida si se añade una directiva SPICE de tipo ".wave ruta 16 44.1k V(out)", donde "ruta" es la ruta completa a un archivo wav. Esto hace que durante la simulación los resultados se almacenen en un archivo wav con la frecuencia de muestreo indicada, en este caso 44100 Hz.

Ambas funciones serán usadas en el capítulo de pruebas para comparar el resultado del procesamiento realizado usando el plugin con el procesamiento en SPICE.

15.1.1 Modelos SPICE para los componentes

Los modelos SPICE usados para el MOSFET 2N7000 y el diodo 1N34A vienen incluidos con la versión de LTSpice elegida.

El modelo usado para el diodo BAT41 es el proporcionado por ST Microelectronics en la página `http://www.st.com/content/st_com/en/products/diodes-and-rectifiers/schottky-barrier-diodes/signal-schottky/bat41.html`.

15.2 Estructura WDF

La conversión de los esquemas de la sección anterior en una estructura WDF es inmediata usando lo expuesto en el capítulo de revisión bibliográfica sobre filtros digitales de ondas.

En primer lugar se transforma la etapa de entrada de la figura 15.1 en las estructuras mostradas en las figuras 15.4 y 15.5. El voltaje de polarización y la resistencia R5 se modelan como una fuente de voltaje no ideal llamada V2. El voltaje V^- que cae en este componente es el mismo voltaje que el de la fuente E1 en la figura 15.1.

Este voltaje V^- se usa para definir el valor de la fuente de voltaje ideal de la figura 15.5. La resistencia Rv y el voltaje V2 se modelan nuevamente como una fuente de voltaje no ideal llamada Vb.

En la figura 15.6 se muestra la estructura WDF equivalente al esquema de la figura 15.2. La fuente de corriente $B1$ y la resistencia R_g se han modelado en conjunto como una fuente de corriente no ideal llamada I_v en la figura. Las dos ramas con MOSFETs y diodos se

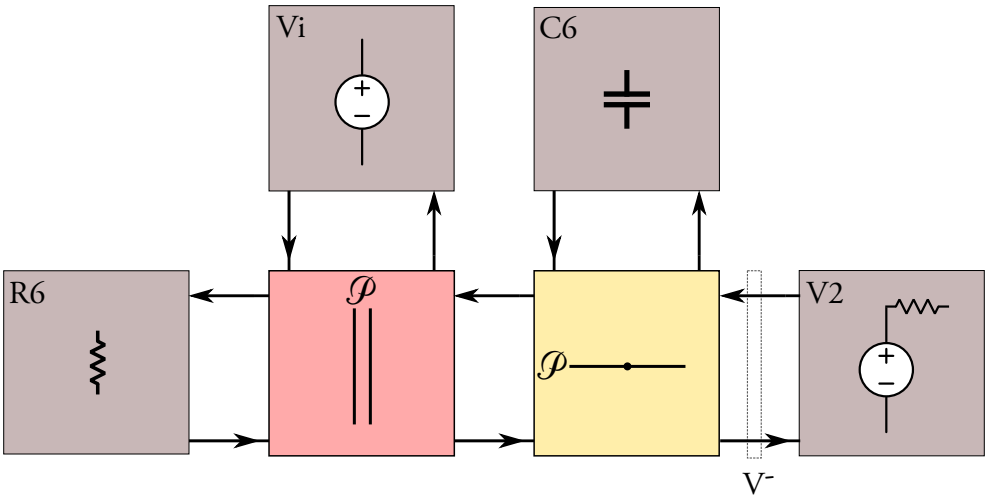


Figura 15.4: Estructura WDF para simular la etapa de entrada del efecto zendrive.

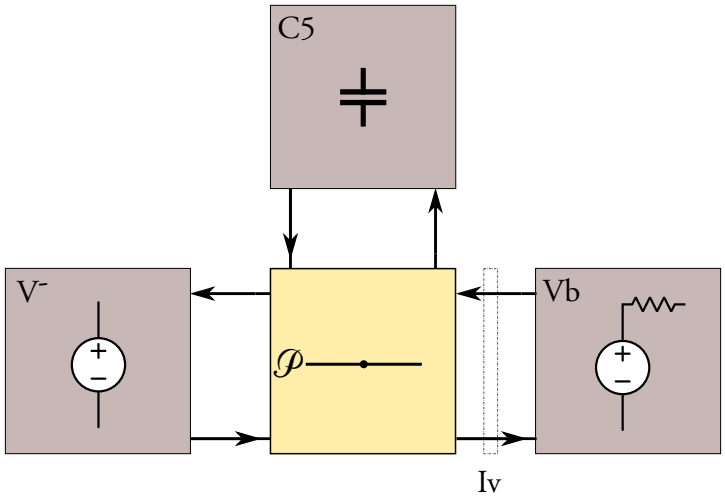


Figura 15.5: Estructura WDF para simular el filtro de voz del efecto zendrive.

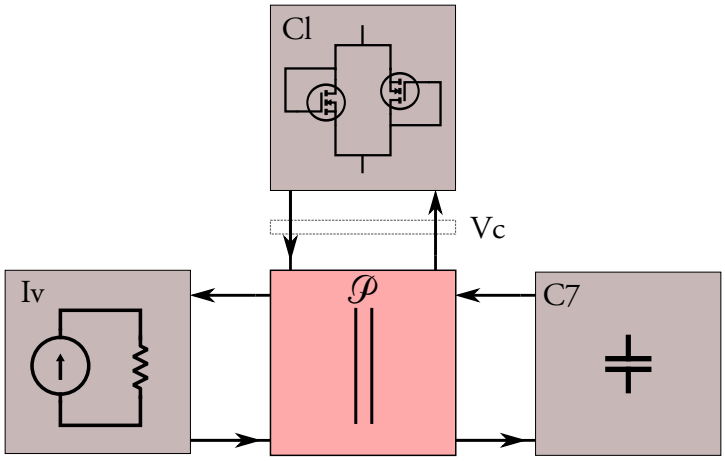


Figura 15.6: Estructura WDF para simular el lazo de realimentación del efecto zendrive.

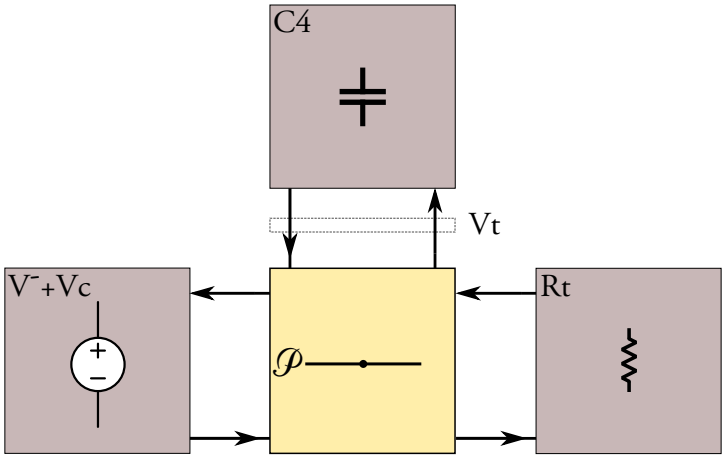


Figura 15.7: Estructura WDF para simular el control de tono del efecto zendrive.

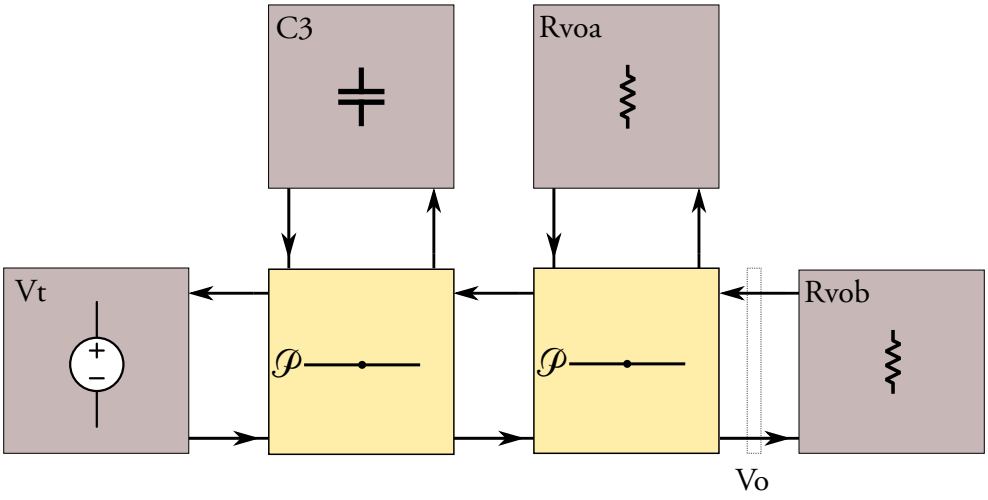


Figura 15.8: Estructura WDF para simular el control de volumen del efecto zendrive.

han modelado usando un solo componente WDF al que he llamado “Cl”, abreviatura de *Clipper*. El procesamiento que este componente hace de las ondas será explicado en la siguiente sección. El voltaje V_c será en cada instante la diferencia de potencial entre la entrada inversora y la salida.

En la figura 15.7 se muestra la estructura WDF equivalente al lazo izquierdo en el esquema de la figura 15.3. La fuente de voltaje ideal toma el valor de voltaje en la entrada inversora sumado a la caída de potencial en el lazo de realimentación negativa.

En la figura 15.8 se muestra la parte restante del esquema. La fuente de voltaje ideal toma como valor la caída de potencial en el condensador C4. El voltaje V_o es el voltaje en la salida del circuito.

Estos 4 árboles WDF forman en su conjunto un algoritmo que simula el circuito. En cada periodo de muestreo los árboles serán procesados tal y como fue descrito en la sección

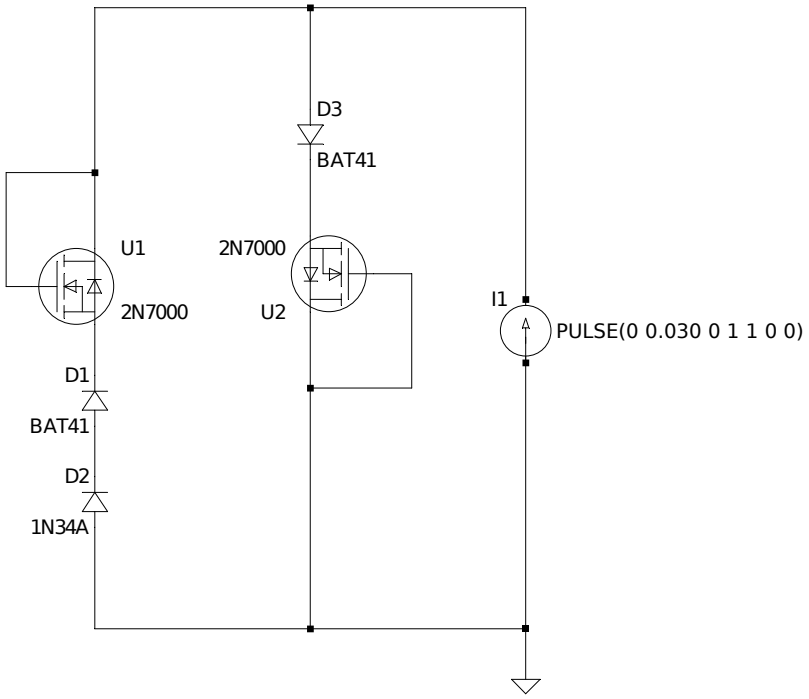


Figura 15.9: Circuito limitador con MOSFETs y fuente de corriente.

13.1.11. El orden de procesamiento será el necesario para hacer posible la ejecución. Al principio se tendrá un voltaje de entrada V_i y por lo tanto podrá procesarse el primer árbol y obtener V^-. Usando V^- podrá procesarse el siguiente árbol, obteniendo I_v. Este proceso continuará hasta que se obtenga el voltaje de salida V_o.

15.3 Modelo WDF de un limitador con MOSFETs y diodos

En esta sección se desarrolla el componente “Cl”, usado en la sección anterior para simular las ramas con MOSFETs y diodos en el lazo de realimentación del amplificador operacional.

Existen en la actualidad varios modelos WDF para simular limitadores con diodos, si bien la mayoría de ellos solo son útiles cuando la limitación es simétrica. Además, los modelos están basados en la ecuación de Shockley por lo que no resultan de utilidad en este caso. De entre ellos, es de especial interés el modelo de K.J. Werner y otros [62]. En él se usa la función $\mathscr{W}$ de Lambert para construir una ecuación que describe las ondas reflejadas en limitadores simétricos con un número arbitrario de diodos iguales en serie. Si es del interés del lector, recomiendo usar el método de Fukushima para computar la función $\mathscr{W}$ de Lambert en aplicaciones que requieran rapidez y precisión [63].

En este trabajo la aproximación es diferente. Dado que el limitador contiene diodos de

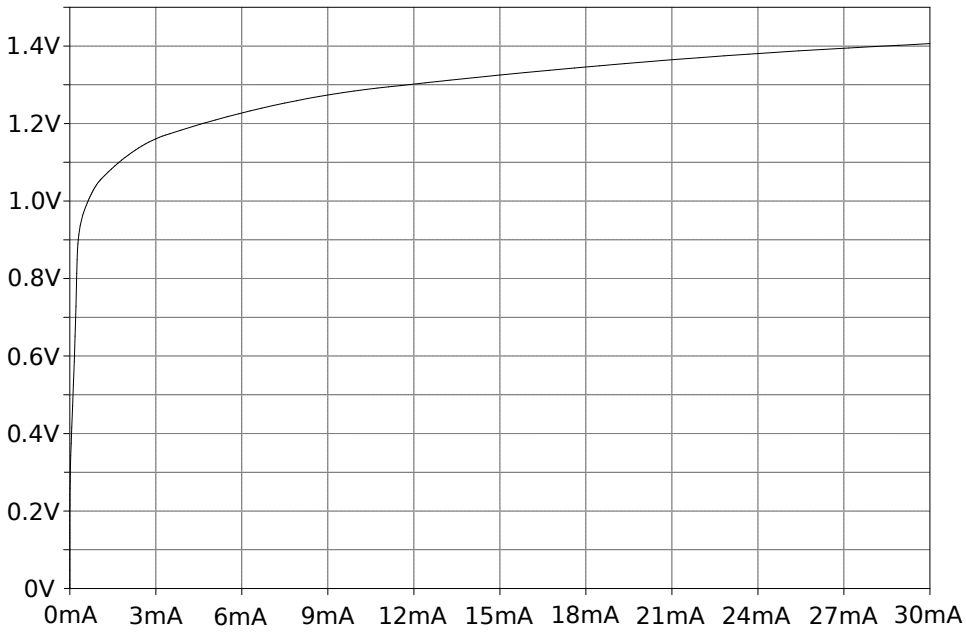


Figura 15.10: Curva corriente-voltaje de un MOSFET y un diodo en serie.

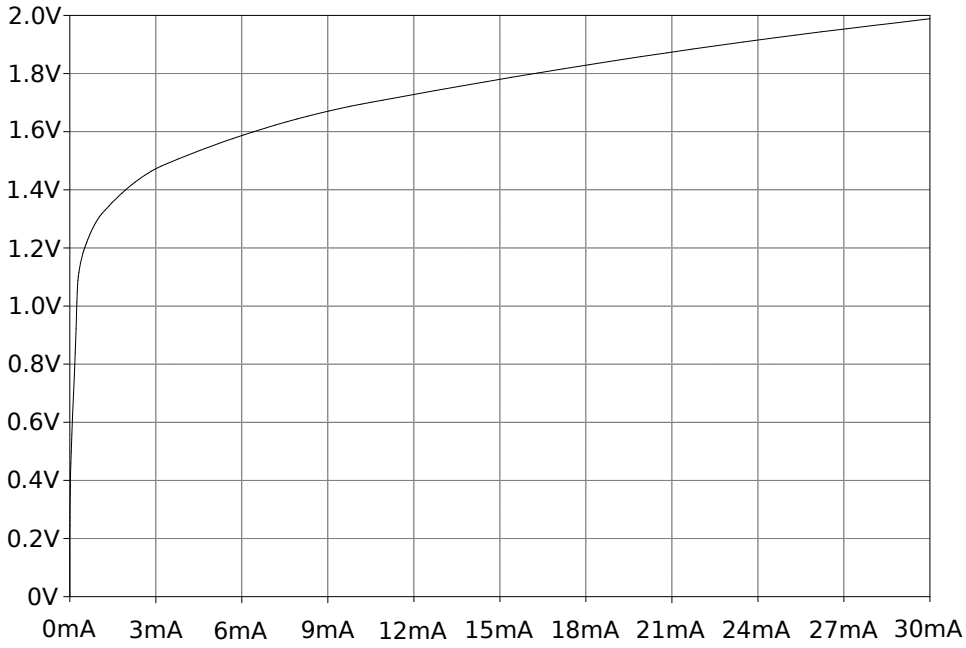


Figura 15.11: Curva corriente-voltaje de un MOSFET y dos diodos de silicio y germanio en serie.

```
res = csvRead("E:\Mis documentos\Proyectos\Pedales\
    Zendrive\Clipper V-I\negVnegI.csv");
negI = res(:,1)*0.03;
csvWrite(negI,"E:\Mis documentos\Proyectos\Pedales\
    Zendrive\Clipper V-I\negIValues.csv");
negV = res(:,2);
csvWrite(negV,"E:\Mis documentos\Proyectos\Pedales\
    Zendrive\Clipper V-I\negVValues.csv");

res = csvRead("E:\Mis documentos\Proyectos\Pedales\
    Zendrive\Clipper V-I\posVposI.csv");
posI = res(:,1)*0.03;
csvWrite(posI,"E:\Mis documentos\Proyectos\Pedales\
    Zendrive\Clipper V-I\posIValues.csv");
posV = res(:,2);
csvWrite(posV,"E:\Mis documentos\Proyectos\Pedales\
    Zendrive\Clipper V-I\posVValues.csv");
```

Listado 15.1: Código para organizar los datos de corriente y voltaje.

diferentes tipos y MOSFETs derivar un modelo matemático sería complejo y probablemente no se obtendría el nivel de exactitud deseado. En su lugar, he decidido obtener puntos de la curva corriente-voltaje del limitador mediante simulación en SPICE y usarlos para construir un interpolador de Fritsch-Carlson siguiendo el procedimiento explicado en la sección 13.2. La precisión de este modelo es limitada, ya que ignora factores como las capacitancias parásitas y los tiempos de conmutación. Además, la corriente está limitada a un máximo de 30 mA. No obstante, estimo que es adecuado para la aplicación.

Para obtener los puntos de datos he usado el esquema que se muestra en la figura 15.9. La fuente de corriente aumenta desde 0 hasta una intensidad máxima de 30 mA en el transcurso de un segundo. Por tanto, multiplicando el tiempo por 0.030 se obtiene la intensidad en amperios en un determinado instante.

El sentido de la fuente de corriente puede cambiarse para obtener así las curvas corriente-voltaje de ambas ramas. Obviamente es necesario cambiar también el punto de conexión a tierra si se quiere obtener en ambos casos un voltaje positivo. Los resultados de las simulaciones se muestran en las figuras 15.10 y 15.11. Los datos de ambas curvas pueden exportarse a ficheros CSV desde LTSpice y serán usados en el capítulo siguiente para construir el interpolador.

Dado que el formato con el que LTSpice almacena los datos no es adecuado para definir un array de floats en C++ es necesario seguir un cierto procedimiento. En primer lugar he

usado el código Scilab del listado 15.1 para separar los archivos CSV de ambas ramas en listados de corriente y voltaje. Lo único que hace el código, además de leer y escribir los archivos, es multiplicar la columna de tiempo por 0.03 para obtener la corriente.

Posteriormente he usado el programa Sublime Text 3 para buscar y reemplazar texto con expresiones regulares. La siguiente expresión regular encuentra todos los números representados con notación exponencial:

```
([-+]?[0-9]*\.?[0-9]+(?:[e][-+]?[0-9]+)?)
```

Esta expresión está entre paréntesis para ser asignada a un grupo. En programas de tratamiento de textos como Sublime Text 3 estos grupos pueden usarse en el campo de reemplazo y representan la ocurrencia original. En este caso he usado "$1*f*," en el campo de reemplazo, lo cual representa la ocurrencia original seguida de una f y una coma. El resultado al buscar y reemplazar todas las ocurrencias es una cadena de números separada por comas apta para inicializar un array de floats.

15.4 Conclusiones

Se ha derivado en este capítulo un modelo completo para simular el circuito. Para ello, se ha dibujado el circuito en LTSpice usando las típicas suposiciones de ganancia infinita y posteriormente se han convertido los esquemas en una estructura WDF. Este modelo será la base de la implementación en capítulos posteriores. Además, el circuito dibujado en LTSpice será de utilidad para procesar señales y comprar los resultados en el capítulo de pruebas.

CAPÍTULO 16

Implementación del plugin oakdrive

Este capítulo pertenece al proceso de codificación del efecto zendrive. En primer lugar, explico la implementación realizada del método de interpolación de Fritsch-Carlson descrito en la sección 13.2. Posteriormente, expongo la implementación de un conjunto de clases que aportan las funcionalidades necesarias para procesar estructuras WDF. Finalmente, explico la implementación de una clase principal que haga uso de las anteriores para simular el efecto, lo cual es trivial tras lo expuesto en anteriores capítulos. El código de este capítulo puede encontrarse en la ruta *./Proyecto/Saturación/Plugin VST/Source*.

16.1 Interpolador de Fritsch-Carlson

En el listado 16.1 se muestra la cabecera de la clase *FCInterpolator*. Los miembros *mX*, *mY* y *mD* son punteros a arrays que contendrán los puntos de partición, x_i; los valores de la función de entrada en dichos puntos, f_i; y los valores de las derivadas para realizar la interpolación, d_i; respectivamente. El miembro *mN* indica el tamaño de los arrays anteriores.

Debe ponerse especial cuidado al usar esta clase, ya que los miembros *mX* y *mY* apuntan a direcciones no locales. Esto se ha hecho así por motivos de eficiencia, pero seguramente

```
1  #pragma once
2  #include <stdexcept>
3  #include <math.h>
4  class FCInterpolator {
5  private:
6          const float* mX;
7          const float* mY;
8          const float* mD;
9          const int mN;
10 public:
11         FCInterpolator(float* x, float* y, const int n);
12         ~FCInterpolator();
13         float interpolate(float x);
14 };
```

Listado 16.1: Contenido del archivo *FCInterpolator.h*.

```
#include "FCInterpolator.h"

	FCInterpolator::FCInterpolator(float* x, float* y, const int n) : mN(n) {
		mX = x;
		mY = y;

		float* s = new float[n - 1];
		float* d = new float[n];

		//Compute s, the slope of the secants.
		for (int i = 0; i < n - 1; i++) {
			float h = x[i + 1] - x[i];
			if (h <= 0) {
				throw std::invalid_argument("Partition points are not valid.");
			}
			s[i] = (y[i + 1] - y[i]) / h;
		}

		//Initialize derivatives as the average slope of the secants
		d[0] = s[0];
		for (int i = 1; i < n - 1; i++) {
			d[i] = (s[i - 1] + s[i]) * 0.5f;
		}
		d[n - 1] = s[n - 2];

		// Update derivatives using S2, subset of M
		for (int i = 0; i < n - 1; i++) {
			if (s[i] == 0) {
				d[i] = 0;
				d[i + 1] = 0;
			}
			else {
				float a = d[i] / s[i];
				float b = d[i + 1] / s[i];
				if (a >= 0 && b >= 0) {     //Update only if input values are monotone
					float h = hypot(a, b);
					if (h > 3.0f) {
						float tau = 3.0f / h;
						d[i] = tau * a * s[i];
						d[i + 1] = tau * b * s[i];
					}
				}
			}
		}
		delete[] s;
		mD = d;          //Deleted in the destructor
	}

	FCInterpolator::~FCInterpolator() {
		delete[] mD;
	}
```

Listado 16.2: Contenido del archivo *FCInterpolator.c*, constructor y destructor.

```
1   float FCInterpolator::interpolate(float x) {
2           if (isnan(x)) {
3                   return x;
4           }
5           if (x >= mX[mN - 1]) {
6                   return mY[mN - 1];
7           }
8           if (x <= mX[0]) {
9                   return mY[0];
10          }
11
12          //Find the closest partition point greater than x
13          int i = 0;
14          while (x >= mX[i + 1]) {
15                  i += 1;
16                  if (x == mX[i]) {
17                          return mY[i];
18                  }
19          }
20
21          float h = mX[i + 1] - mX[i];
22          float t = (x - mX[i]) / h;
23          return (mY[i] * (1 + 2 * t) + h * mD[i] * t) * (1 - t) * (1 - t) +
24                      (mY[i + 1] * (3 - 2 * t) + h * mD[i + 1] * (t - 1)) * t * t;
25  }
```

Listado 16.3: Contenido del archivo *FCInterpolator.c*, función de interpolación.

el coste de añadir una operación de copia de los arrays externos a miembros locales sería despreciable. No obstante, hay que tener en cuenta que esta clase deberá ser creada cada vez que se modifique el parámetro de ganancia, como se verá posteriormente.

En el listado 16.2 se muestra el constructor y el destructor de la clase *FCInterpolator*. Como puede verse, el código del constructor empieza calculando la pendiente de las secantes, denotada en el desarrollo de la sección 13.2 como Δ_i. Posteriormente, inicializa las derivadas en los puntos de partición calculando el promedio de la pendiente de las secantes en subintervalos adyacentes. Por último, las derivadas son actualizadas usando la región $\mathscr{S}_2$. Para ello, primero se calcula la longitud del vector que une el origen de coordenadas con el punto (α, β). Si esta es mayor que 3, entonces el punto (α, β) se encuentra fuera de la región de monotonía elegida, $\mathscr{S}_2$, y es necesario actualizar las derivadas tal y como se describió en la sección 13.2.

El código no actualiza las derivadas en el subintervalo correspondiente si alguno de los valores de entrada f_i rompe la monotonía, pero tampoco arroja ningún tipo de error o excepción. Simplemente deja las derivadas en ese subintervalo como el promedio de la pendiente de las secantes aún en el caso de que el punto (α, β) esté fuera de la región de monotonía. Esto relaja las condiciones sobre los datos de entrada.

En el listado 16.3 se muestra el código de la función *interpolate*. El retorno es simplemente una aplicación de la ecuación (13.58) usando las derivadas calculadas en el constructor.

```
#pragma once
class WDFElement {
    protected:
        double mR = 100.0;
        double mA;
        double mB;
        WDFElement* mParent;
    public:
        virtual double getReflectedWave() = 0;
        virtual void setIncidentWave(double A) = 0;
        virtual void updateImpedance(double R) = 0;
        virtual void linkTree(WDFElement* parent) = 0;
                WDFElement();
                double getVoltage() const{
                        return (mA+mB)/2;
                }
                double getCurrent() const{
                        return (mA-mB)/(2*mR);
                }
                double getR() const{
                        return mR;
                }
                double getA() const{
                        return mA;
                }
                double getB() const{
                        return mB;
                }
};
```

Listado 16.4: Contenido del archivo *WDFElement.h*.

16.2 Clases WDF

16.2.1 Clase abstracta para elementos WDF

En el listado 16.4 se muestra la cabecera de esta clase. El objetivo es proporcionar una ascendencia común a todas las clases de un árbol WDF. Todos los elementos WDF comparten un conjunto de atributos:

- *mR*, la resistencia de puerto.
- *mA*, la onda incidente.
- *mB*, la onda reflejada.
- *mParent*, un puntero al elemento padre.

Y un conjunto de operaciones, declaradas como funciones virtuales que deben ser implementadas en cada clase hija:

- *getReflectedWave*, obtener la onda reflejada.
- *setIncidentWave*, definir la onda incidente.
- *updateImpedance*, actualizar la impedancia del puerto.
- *linkTree*, aportar un puntero al elemento padre.

Además, hay un conjunto de operaciones que no solo son comunes a todos los elementos WDF sino que además siempre realizan el mismo procedimiento. Estas son:

- *getVoltage*, obtener el voltaje actual en el dominio K.
- *getCurrent*, obtener la corriente actual en el dominio K.

```
#include "WDFElement.h"
WDFElement::WDFElement() {
        mA = 0;
        mB = 0;
        mR = 0;
        mParent = nullptr;
}
```

Listado 16.5: Contenido del archivo *WDFElement.c*.

```
#pragma once
#include "WDFElement.h"
class Capacitor : public WDFElement {
        private:
                const double mC;
                const int mFs;
        public:
                Capacitor(double C, double Fs);
                double getReflectedWave();
                void setIncidentWave(double A);
                void updateImpedance(double R);
                void linkTree(WDFElement* parent);
};
```

Listado 16.6: Contenido del archivo *Capacitor.h*.

- *getR*, obtener la impedancia del puerto.
- *getA*, obtener la onda incidente.
- *getB*, obtener la onda reflejada.

Estas operaciones se implementan directamente en la cabecera de la clase abstracta.

He decidido usar doble precisión para implementar estas clases, aunque en el contexto de este trabajo no es necesario. En la versión para el núcleo C674x usaré en cambio precisión simple por motivos de eficiencia.

La idea general es que cada elemento WDF contenga un puntero a su padre de tal forma que la función *updateImpedance* pueda notificar automáticamente al elemento padre cualquier cambio en la impedancia. Esto resultará claro a medida que se vaya realizando la implementación de los diferentes elementos.

En el listado 16.5 se muestra la implementación de esta clase. Aunque por definición esta es una clase abstracta, proporciona un constructor base que da un valor inicial a todos los miembros. Este constructor será llamado por todas las clases hijas antes de empezar su propio constructor.

16.2.2 Condensador

La implementación de la clase para el condensador es inmediata a partir de lo expuesto en la sección 13.1.4. En el listado 16.6 se muestra la cabecera de la clase. Los miembros *mC* y *mFs* almacenan la capacidad y la frecuencia de muestreo, respectivamente.

```
#include "Capacitor.h"
        Capacitor::Capacitor(double C, double Fs) : mC(C), mFs(Fs) {
                mR = 1 / (2 * Fs*C);
        }
        double Capacitor::getReflectedWave() {
                mB = mA;
                return mB;
        }
        void Capacitor::setIncidentWave(double A) {
                mA = A;
        }
        void Capacitor::updateImpedance(double R) {
                mR = 1 / (2 * mFs*R);
                if (mParent)
                        mParent->updateImpedance(mR);
        }
        void Capacitor::linkTree(WDFElement* parent) {
                mParent = parent;
        }
```

Listado 16.7: Contenido del archivo *Capacitor.c*.

```
#pragma once
#include "WDFElement.h"
class Resistor : public WDFElement {
        public:
                Resistor(double R);
                double getReflectedWave();
                void setIncidentWave(double A);
                void updateImpedance(double R);
                void linkTree(WDFElement* parent);
};
```

Listado 16.8: Contenido del archivo *Resistor.h*.

En el listado 16.7 se muestra la implementación de la clase. El constructor calcula la impedancia de puerto mediante la expresión desarrollada en la sección 13.1.4. La función *updateImpedance* toma una nueva capacidad y actualiza la impedancia de puerto. Si el condensador tiene un elemento padre en el árbol llama a la función *updateImpedance* del padre pasando como argumento la nueva impedancia de puerto. El padre debe tomar esta información y actuar en consecuencia, como se verá en una sección posterior.

```
#include "Resistor.h"
        Resistor::Resistor(double R) {
                mR = R;
        }
        double Resistor::getReflectedWave() {
                mB = 0;
                return mB;
        }
        void Resistor::setIncidentWave(double A) {
                mA = A;
        }
        void Resistor::updateImpedance(double R) {
                mR = R;
                if (mParent)
                        mParent->updateImpedance(mR);
        }
        void Resistor::linkTree(WDFElement* parent) {
                mParent = parent;
        }
```

Listado 16.9: Contenido del archivo *Resistor.c*.

```
#pragma once
#include "WDFElement.h"
class IdealVoltageSource : public WDFElement {
        private:
                double mV;
        public:
                IdealVoltageSource(double V, double R);
                double getReflectedWave();
                void setIncidentWave(double A);
                void setV(double V);
                void updateImpedance(double R);
                void linkTree(WDFElement* parent);
};
```

Listado 16.10: Contenido del archivo *IdealVoltageSource.h*.

```
#include "IdealVoltageSource.h"
        IdealVoltageSource::IdealVoltageSource(double V, double R) {
                mV = V;
                mR = R;
        }
        double IdealVoltageSource::getReflectedWave(){
                mB = 2 * mV - mA;
                return mB;
        }
        void IdealVoltageSource::setIncidentWave(double A) {
                mA = A;
        }
        void IdealVoltageSource::setV(double V) {
                mV = V;
        }
        void IdealVoltageSource::updateImpedance(double R) {
                mR = R;
                if (mParent)
                        mParent->updateImpedance(mR);
        }
        void IdealVoltageSource::linkTree(WDFElement* parent) {
                mParent = parent;
        }
```

Listado 16.11: Contenido del archivo *IdealVoltageSource.c*.

16.2.3 Resistencia

La implementación de la clase para la resistencia es inmediata a partir de lo expuesto en la sección 13.1.3. En el listado 16.8 se muestra la cabecera de la clase. En el listado 16.9 se muestra el código de la clase.

16.2.4 Fuente de voltaje ideal

La implementación de la clase para la fuente de voltaje ideal es inmediata a partir de lo expuesto en la sección 13.1.5. En el listado 16.10 se muestra la cabecera de la clase. El miembro mV almacena el voltaje de la fuente y la función adicional *setV* sirve para modificarlo. En el listado 16.11 se muestra el código de la clase.

16.2.5 Fuente de voltaje real

La implementación de la clase para la fuente de voltaje real es inmediata a partir de lo expuesto en la sección 13.1.6. En el listado 16.12 se muestra la cabecera de la clase. El miembro mV almacena el voltaje de la fuente y la función adicional *setV* sirve para

```
#pragma once
#include "WDFElement.h"
class ResistiveVoltageSource: public WDFElement {
        private:
                double mV;
        public:
                ResistiveVoltageSource(double V, double R);
                double getReflectedWave();
                void setIncidentWave(double A);
                void setV(double V);
                void updateImpedance(double R);
                void linkTree(WDFElement* parent);
};
```

Listado 16.12: Contenido del archivo *ResistiveVoltageSource.h*.

```
#include "ResistiveVoltageSource.h"
        ResistiveVoltageSource::ResistiveVoltageSource(double V, double R) {
                mV = V;
                mR = R;
        }
        double ResistiveVoltageSource::getReflectedWave() {
                mB = mV;
                return mB;
        }
        void ResistiveVoltageSource::setIncidentWave(double A) {
                mA = A;
        }
        void ResistiveVoltageSource::setV(double V) {
                mV = V;
        }
        void ResistiveVoltageSource::updateImpedance(double R) {
                mR = R;
                if (mParent)
                        mParent->updateImpedance(mR);
        }
        void ResistiveVoltageSource::linkTree(WDFElement* parent) {
                mParent = parent;
        }
```

Listado 16.13: Contenido del archivo *ResistiveVoltageSource.c*.

```
#pragma once
#include "WDFElement.h"
class ResistiveCurrentSource : public WDFElement {
        private:
                double mI;
        public:
                ResistiveCurrentSource(double I, double R);
                double getReflectedWave();
                void setIncidentWave(double A);
                void setCurrent(double I);
                void updateImpedance(double R);
                void linkTree(WDFElement* parent);
};
```

Listado 16.14: Contenido del archivo *ResistiveCurrentSource.h*.

```
#include "ResistiveCurrentSource.h"
        ResistiveCurrentSource::ResistiveCurrentSource(double I, double R) {
                mI = I;
                mR = R;
        }
        double ResistiveCurrentSource::getReflectedWave() {
                mB = mR*mI;
                return mB;
        }
        void ResistiveCurrentSource::setIncidentWave(double A) {
                mA = A;
        }
        void ResistiveCurrentSource::setCurrent(double I) {
                mI = I;
        }
        void ResistiveCurrentSource::updateImpedance(double R) {
                mR = R;
                if(mParent)
                        mParent->updateImpedance(mR);
        }
        void ResistiveCurrentSource::linkTree(WDFElement* parent) {
                mParent = parent;
        }
```

Listado 16.15: Contenido del archivo *ResistiveCurrentSource.c*.

modificarlo. En el listado 16.13 se muestra el código de la clase.

16.2.6 Fuente de corriente real

La implementación de la clase para la fuente de voltaje real es inmediata a partir de lo expuesto en la sección 13.1.8. En el listado 16.14 se muestra la cabecera de la clase. El miembro mI almacena la corriente de la fuente y la función adicional *setCurrent* sirve para modificarla. En el listado 16.15 se muestra el código de la clase.

16.2.7 Adaptador en serie de tres puertos

La implementación de la clase para un adaptador en serie de tres puertos se hace tomando como guía lo expuesto en la sección 13.1.9, pero requiere de un comentario pormenorizado. En el listado 16.16 se muestra la cabecera de la clase. Se añaden algunos miembros y una función privada adicional:

- $mLeft$, un puntero al miembro conectado al puerto izquierdo del adaptador.
- $mRight$, un puntero al miembro conectado al puerto derecho del adaptador.

```
#pragma once
#include "WDFElement.h"
class SeriesThreePortAdaptor: public WDFElement
{
public:
    SeriesThreePortAdaptor(WDFElement *left, WDFElement *right);
    double getReflectedWave();
    void setIncidentWave(double A);
    void updateImpedance(double R);
    void linkTree(WDFElement* parent);
private:
        WDFElement *mLeft;
        WDFElement *mRight;
        double gamma;
        double gamma_left;
        double gamma_right;
        void impedanceMatch();
};
```

Listado 16.16: Contenido del archivo *SeriesThreePortAdaptor.h*.

```
#include "SeriesThreePortAdaptor.h"
    SeriesThreePortAdaptor::SeriesThreePortAdaptor(WDFElement *left, WDFElement *right) :
        mLeft(left), mRight(right)
        {
        impedanceMatch();
    }
    void SeriesThreePortAdaptor::impedanceMatch(){
        mR = mLeft->getR() + mRight->getR();
        gamma = - 2*mR / (mR+ mLeft->getR() +mRight->getR());
        gamma_left = -(2*mLeft->getR()) / (mR+mLeft->getR()+ mRight->getR());
        gamma_right = -(2*mRight->getR()) / (mR+mLeft->getR() + mRight->getR());
    }
    double SeriesThreePortAdaptor::getReflectedWave() {
        mB = mA + gamma * (mA+mLeft->getReflectedWave()+mRight->getReflectedWave());
        return mB;
    }
    void SeriesThreePortAdaptor::setIncidentWave(double A) {
        mA = A;
        mLeft->setIncidentWave(mLeft->getB() + gamma_left * (A+mLeft->getB()+mRight->getB()));
        mRight->setIncidentWave(mRight->getB() + gamma_right  * (A+mLeft->getB()+mRight->getB()));
    }
    void SeriesThreePortAdaptor::updateImpedance(double R) {
        impedanceMatch();
        if(mParent)
                        mParent->updateImpedance(mR);
    }
    void SeriesThreePortAdaptor::linkTree(WDFElement* parent) {
        mParent = parent;
        mLeft->linkTree(this);
        mRight->linkTree(this);
    }
```

Listado 16.17: Contenido del archivo *SeriesThreePortAdaptor.c*.

```
#pragma once
#include "WDFElement.h"
class ParallelThreePortAdaptor : public WDFElement {
public:
        ParallelThreePortAdaptor(WDFElement* left, WDFElement* right);
        double getReflectedWave();
        void setIncidentWave(double A);
        void updateImpedance(double R);
        void linkTree(WDFElement* parent);
private:
        WDFElement* mLeft;
        WDFElement* mRight;
        double gamma;
        double gamma_left;
        double gamma_right;
        void impedanceMatch();
};
```

Listado 16.18: Contenido del archivo *ParallelThreePortAdaptor.h*.

- *gamma*, el coeficiente de dispersión para el puerto adaptado, llamado γ_n en el desarrollo de la sección 13.1.9.
- *gamma_left*, el coeficiente de dispersión para el puerto izquierdo.
- *gamma_right*, el coeficiente de dispersión para el puerto derecho.
- *impedanceMatch*, esta función calcula la impedancia del puerto adaptado y los coeficientes de dispersión siempre que es llamada a partir de las expresiones de la sección 13.1.9.

En el listado 16.17 se muestra el código de la clase. Nótese que la función *impedanceMatch* es llamada en dos casos: En el constructor, tras dar un valor a los punteros *mLeft* y *mRight*; y en la función *updateImpedance*, dado que una llamada a esta sucederá por parte de los hijos cuando modifiquen su resistencia de puerto y sea por tanto necesario actualizar la impedancia del puerto adaptado y los coeficientes de dispersión.

Por otro lado, cuando se actualiza la onda incidente en el puerto adaptado mediante la función *setIncidentWave* esta usa los coeficientes de dispersión para reflejar la onda en ambos hijos, de acuerdo a las expresiones expuestas en la sección 13.1.9 para calcular las ondas *B*. Algo análogo sucede cuando se solicita la onda reflejada en el puerto adaptado mediante la función *getReflectedWave*.

Nótese que aunque esto puede generar cierta confusión, la onda B reflejada en el puerto de un adaptador es la onda A incidente en el elemento conectado a ese puerto. Llamamos A o B a la onda dependiendo del punto de vista.

16.2.8 Adaptador en paralelo de tres puertos

La implementación del adaptador en paralelo es análoga a la realizada para el adaptador en serie. La única diferencia es que se usan las expresiones desarroladas en la sección 13.1.10 para calcular las ondas y los coeficientes de dispersión. En el listado 16.18 se muestra la

```
#include "ParallelThreePortAdaptor.h"

        ParallelThreePortAdaptor::ParallelThreePortAdaptor(WDFElement* left, WDFElement* right) {
                mLeft = left;
                mRight = right;
                impedanceMatch();
        }

        void ParallelThreePortAdaptor::impedanceMatch() {
                mR = 1.0 / (1.0 / mLeft->getR() + 1.0 / mRight->getR());
                gamma = 2 * (1.0 / mR) / (1.0 / mR + 1.0 / mLeft->getR() + 1.0 / mRight->getR());
                gamma_left = 2 * (1.0 / mLeft->getR()) / (1.0 / mR + 1.0 / mLeft->getR() + 1.0 / mRight->
                        getR());
                gamma_right = 2 * (1.0 / mRight->getR()) / (1.0 / mR + 1.0 / mLeft->getR() + 1.0 / mRight->
                        getR());
        }

        double ParallelThreePortAdaptor::getReflectedWave() {
                mB = -mA + gamma*mA + gamma_left*mLeft->getReflectedWave() + gamma_right*mRight->
                        getReflectedWave();
                return mB;
        }

        void ParallelThreePortAdaptor::setIncidentWave(double A) {
                mA = A;
                mLeft->setIncidentWave(-mLeft->getB() + gamma*A + gamma_left*mLeft->getB() + gamma_right*
                        mRight->getB());
                mRight->setIncidentWave(-mRight->getB() + gamma*A + gamma_left*mLeft->getB() + gamma_right*
                        mRight->getB());
        }

        void ParallelThreePortAdaptor::updateImpedance(double R) {
                impedanceMatch();
                if (mParent)
                        mParent->updateImpedance(mR);
        }

        void ParallelThreePortAdaptor::linkTree(WDFElement* parent) {
                mParent = parent;
                mLeft->linkTree(this);
                mRight->linkTree(this);
        }
```

Listado 16.19: Contenido del archivo *ParallelThreePortAdaptor.c*.

cabecera de la clase. En el listado 16.19 se muestra el código de la clase.

16.2.9 Limitador con MOSFETs

En esta sección se implementa el elemento *Cl* usado en la figura 15.6 para simular el limitador. En el listado 16.20 se muestra la cabecera de la clase. Además de los miembros típicos de un elemento WDF se añaden otros miembros privados:

- Las constantes *posV* y *posIvalues* contienen los valores extraidos de la curva corriente-voltaje de la figura 15.10.
- Las constantes *negV* y *negIvalues* contienen los valores extraidos de la curva corriente-voltaje de la figura 15.11.
- Los arrays *posAvalues* y *posBvalues* contendrán en cada momento una representación en el dominio *W* de las curvas corriente-voltaje almacenadas en los arrays *posV* y *posIvalues*.
- Los arrays *negAvalues* y *negBvalues* contendrán en cada momento una representación en el dominio *W* de las curvas corriente-voltaje almacenadas en los arrays *negV* y *negIvalues*.

```
#pragma once
#include "WDFElement.h"
#include "FCInterpolator.h"
#include <memory>
class MosfetClipperSpline : public WDFElement {
public:
        MosfetClipperSpline(double R);
        double getReflectedWave();
        void setIncidentWave(double A);
        void updateImpedance(double R);
        void linkTree(WDFElement* parent);
private:
        static const int mLn = 76;
        static const float posV[];
        static const float negV[];
        static const float posIvalues[];
        static const float negIvalues[];

        float posAvalues[mLn];
        float posBvalues[mLn];
        float negAvalues[mLn];
        float negBvalues[mLn];

        std::unique_ptr<FCInterpolator> spiPos;
        std::unique_ptr<FCInterpolator> spiNeg;
};
```

Listado 16.20: Contenido del archivo *MosfetClipperSpline.h*.

- Los miembros *spiPos* y *spiNeg* son punteros a interpoladores de Fritsch-Carlson construidos en cada momento con los valores contenidos en las curvas A-B que permitirán obtener el valor de la onda reflejada.

En el listado 16.21 se muestra el código de la clase. Los datos de las curvas corriente-voltaje se han omitido en el listado por economía de espacio. En el constructor de la clase se usa la impedancia del adaptador para calcular las curvas A-B usando las expresiones (13.1) y (13.2) e inicializar con ellas los interpoladores de Fritsch-Carlson.

Cuando se lee la onda reflejada mediante la función *getReflectedWave* se selecciona un interpolador u otro en función del signo de la onda incidente. Esto puede resultar confuso ya que se está asumiendo que el signo de A es igual al signo de la corriente. De hecho, esto no es así en general pero si en el caso concreto de un conjunto de diodos antiparalelos. Para una demostración puede consultarse [62, pp. 8].

El mayor problema de esta clase es que cuando se actualiza la impedancia mediante la función *updateImpedance* es necesario reconstruir los adaptadores y los interpoladores. Dado que el coste de la operación es relativamente alto el número de veces por segundo que puede realizarse estará limitado, como se verá una una sección posterior.

16.3 Clase de simulación

En el listado 16.22 se muestra la cabecera de la clase *OakdriveSim*. Para evitar confusiones expongo a continuación un listado con la correspondencia entre los nombres de miembros en esta clase y los nombres usados en los elementos de las estructuras WDF presentadas en

```
#include "MosfetClipperSpline.h"
        const float MosfetClipperSpline::posV[] = {
                //Se han omitido los datos
        };
        const float MosfetClipperSpline::negV[] = {
                //Se han omitido los datos
        };
        const float MosfetClipperSpline::posIvalues[] = {
                //Se han omitido los datos
        };
        const float MosfetClipperSpline::negIvalues[] = {
                //Se han omitido los datos
        };
        MosfetClipperSpline::MosfetClipperSpline(double R)
        {
                mR = R;
                for (int i = 0; i < mLn; i++) {
                        posAvalues[i] = posV[i] + (float)(posIvalues[i] * R);
                        negAvalues[i] = negV[i] + (float)(negIvalues[i] * R);
                        posBvalues[i] = posV[i] - (float)(posIvalues[i] * R);
                        negBvalues[i] = negV[i] - (float)(negIvalues[i] * R);
                }
                spiPos = std::make_unique<FCInterpolator>(posAvalues, posBvalues, mLn);
                spiNeg = std::make_unique<FCInterpolator>(negAvalues, negBvalues, mLn);
        }


        double MosfetClipperSpline::getReflectedWave() {
                float aA = abs(mA);
                if (aA == 0)
                        return 0;
                if (mA > 0) {
                        mB = spiPos->interpolate(aA);
                        return mB;
                }
                else {
                        mB = -spiNeg->interpolate(aA);
                        return mB;
                }
        }

        void  MosfetClipperSpline::setIncidentWave(double A) {
                mA = A;
        }

        void  MosfetClipperSpline::updateImpedance(double R) {
                mR = R;
                for (int i = 0; i<mLn; i++) {
                        posAvalues[i] = posV[i] + (float)(posIvalues[i] * R);
                        negAvalues[i] = negV[i] + (float)(negIvalues[i] * R);
                        posBvalues[i] = posV[i] - (float)(posIvalues[i] * R);
                        negBvalues[i] = negV[i] - (float)(negIvalues[i] * R);
                }
                spiPos = std::make_unique<FCInterpolator>(posAvalues, posBvalues, mLn);
                spiNeg = std::make_unique<FCInterpolator>(negAvalues, negBvalues, mLn);
                if(mParent)
                        mParent->updateImpedance(mR);
        }

        void MosfetClipperSpline::linkTree(WDFElement* parent) {
                mParent = parent;
        }
```

Listado 16.21: Contenido del archivo *MosfetClipperSpline.c*.

```
#pragma once
#include "IdealVoltageSource.h"
#include "ResistiveVoltageSource.h"
#include "Resistor.h"
#include "Capacitor.h"
#include "SeriesThreePortAdaptor.h"
#include "ParallelThreePortAdaptor.h"
#include "MosfetClipperSpline.h"
#include "ResistiveCurrentSource.h"

class OakdriveSim {

private:
        const double mFs;
        //Input
        IdealVoltageSource vin;
        ResistiveVoltageSource bias;
        Resistor R6;
        Capacitor C6;
        SeriesThreePortAdaptor S1;
        ParallelThreePortAdaptor P1;
        //Voice filter
        double mVoice;
        double voiceresistorvalue;
        Capacitor C5;
        ResistiveVoltageSource voicebias;
        IdealVoltageSource opampvin;
        SeriesThreePortAdaptor S2;
        //Clipper
        double mGain;
        double gainresistorvalue;
        MosfetClipperSpline mclipper;
        ResistiveCurrentSource clippercurrent;
        Capacitor C7;
        ParallelThreePortAdaptor P2;
        //Low pass
        double mTone;
        double toneresistorvalue;
        IdealVoltageSource vclippersource;
        Resistor tonepot;
        Capacitor C4;
        SeriesThreePortAdaptor S3;
        //High pass
        double mVolume;
        double volumeresistorvalue;
        double volumeVdiv;
        IdealVoltageSource vlp;
        Resistor volumeresistorVdiv;
        Resistor volumeresistor;
        Capacitor C3;
        SeriesThreePortAdaptor S4;
        SeriesThreePortAdaptor S5;
public:
        OakdriveSim(double Fs, double voice, double gain, double tone, double volume);
        double getSampleRate();
        float oakdriveSimulation(float sample);
        void changeVoice(float value);
        void changeGain(float value);
        void changeTone(float value);
        void changeVolume(float volume);
};
```

Listado 16.22: Contenido del archivo *OakdriveSim.h*.

la sección 15.2. Se omitirán los nombres de las resistencias y condensadores que coincidan. También se omitirán los nombres de los adaptadores, ya que la correspondencia es obvia.

- vin $\rightarrow$ Vi
- bias $\rightarrow$ V2
- opampvin $\rightarrow V^-$
- voicebias $\rightarrow$ Vb
- clippercurrent $\rightarrow$ Iv
- mclipper $\rightarrow$ Cl
- vclippersource $\rightarrow V^- + Vc$
- tonepot $\rightarrow$ Rt
- vlp $\rightarrow$ Vt
- volumeresistorVdiv $\rightarrow$ Rvoa
- volumeresistor $\rightarrow$ Rvob

Para almacenar la posición de los potenciómetros se declaran tres miembros que contendrán valores en el intervalo $[0,1]$: *mVoice*, *mGain*, *mTone* y *mVolume*. En cuanto a las funciones, su propósito es el siguiente:

- *OakdriveSim(double Fs, double voice, double gain, double tone, double volume)*: El constructor de la clase. Toma en los parámetros la frecuencia de muestreo y un valor inicial para todos los potenciómetros.
- *double getSampleRate()*: Devuelve la frecuencia de muestreo que está utilizando el simulador.
- *float oakdriveSimulation(float sample)*: Toma una muestra de entrada y devuelve una muestra de salida cada vez que es llamado. Es la función que será usada para realizar el procesamiento muestra a muestra.

El resto de funciones tienen como propósito cambiar los valores de los potenciómetros, por lo que no es necesario hacer más comentarios sobre ellas.

En el listado 16.23 se muestra el constructor de la clase. Como puede verse, cada resistencia de la estructura WDF se construye usando dos resistencias en serie. Una de ellas será el valor del potenciómetro multiplicado por el valor de posición en el intervalo $[0,1]$. La otra tendrá el valor de la resistencia correspondiente en el circuito.

Las fuentes de voltaje ideales se inicializan con voltaje 0 e impedancia de puerto 1000. Dado que la impedancia es adaptada en todos los nodos raíz de un árbol WDF esta impedancia de puerto inicial es irrelevante. El resto de elementos son inicializados con sus respectivos valores. Nótese que a cada adaptador se le pasa una referencia a sus hijos izquierdo y derecho, que almacenará en punteros internos.

```
#include "OakdriveSim.h"

OakdriveSim::OakdriveSim(double Fs, double voice, double gain, double tone, double volume) : mFs(Fs),
        mVoice(voice),
        voiceresistorvalue(10000 * voice + 1000),
        mGain(gain),
        gainresistorvalue(1000 + 500000 * gain),
        mTone(tone),
        toneresistorvalue(20000 + 50000 * tone),
        mVolume(volume),
        volumeresistorvalue(100000 * volume),
        volumeVdiv((100000 - volumeresistorvalue) + 1000),

        //Input stage
        vin(0, 1000),
        bias(4.5f, 470000),
        R6(2000000),
        C6((470 * pow(10, -9)), Fs),
        S1(&C6, &bias),
        P1(&R6, &S1),

        //Voice filter
        C5((100 * pow(10, -9)), Fs),
        voicebias(4.5f, voiceresistorvalue),
        opampvin(0, 1000),
        S2(&C5, &voicebias),

        //Clipper
        mclipper(100),
        C7((100 * pow(10, -12)), Fs),
        clippercurrent(0.0, gainresistorvalue),
        P2(&clippercurrent, &C7),

        //Low pass
        vclippersource(0, 1000),
        tonepot(toneresistorvalue),
        C4((3.3*pow(10, -9)), Fs),
        S3(&tonepot, &C4),

        //High pass
        vlp(0, 1000),
        volumeresistorVdiv(volumeVdiv),
        volumeresistor(volumeresistorvalue),
        C3((470 * pow(10, -9)), Fs),
        S4(&volumeresistorVdiv, &C3),
        S5(&volumeresistor, &S4)
        {
                P1.linkTree(&vin);
                P1.updateImpedance(0);
                S2.linkTree(&opampvin);
                S2.updateImpedance(0);
                P2.linkTree(&mclipper);
                P2.updateImpedance(0);
                S3.linkTree(&vclippersource);
                S3.updateImpedance(0);
                S5.linkTree(&vlp);
                S5.updateImpedance(0);
        }
```

Listado 16.23: Contenido del archivo *OakdriveSim.c*, constructor.

Una vez terminadas las inicializaciones es necesario construir el árbol. Para ello se usa la función *linkTree* de cada adaptador para conectar su puerto adaptado al elemento padre. Esta función llamará a su vez a las funciones *linkTree* de todos los hijos pasando un puntero al propio adaptador y produciendo una cadena de llamadas que se propagará hacia abajo en el árbol.

Posteriormente se llama a la función *updateImpedance*, que automáticamente actualizará las impedancias y los coeficientes de dispersión. Internamente, esta función acabará también llamando a todas las funciones *updateImpedance* de los padres de los adaptadores, produciendo una actualización de su impedancia de puerto.

Si en cualquier momento de la ejecución se llama a la función *updateImpedance* de uno de los elementos hoja la propagación sucederá desde la parte inferior del árbol, actualizando todas las impedancias de puerto de los elementos de nivel superior. Imaginemos que, por ejemplo, se llama a la función *updateImpedance* del miembro *tonepot* con un nuevo valor de resistencia tras modificar la posición del potenciómetro. Al haber llamado a la función *linkTree* en el constructor el miembro *tonepot* dispone internamente de un puntero a su padre por lo que llamará a su vez a la función *updateImpedance* de $S2$ que a su vez actualizará todas las impedancias de puerto y volverá a calcular los coeficientes de dispersión.

Este es uno de los problemas de las estructura WDF, cualquier actualización de impedancia es costosa, más aún si tenemos en cuenta que es necesario reconstruir el interpolador para las curvas corriente-voltaje del limitador en caso de que la actualización sea en el potenciómetro de ganancia.

En el listado 16.24 se muestra la función de simulación y las funciones de actualización de potenciómetros. En estas últimas se realizan las operaciones necesarias para volver a calcular la resistencia total de un elemento cuando se actualiza un potenciómetro.

Para entender la función de simulación tomemos como ejemplo la etapa de entrada. En primer lugar se actualiza el voltaje de la fuente *vin*. Una vez hecho esto, se actualiza la onda incidente en la fuente de voltaje ideal pasando como argumento el resultado de la llamada a la función *getReflectedWave* del adaptador. Esta función llama internamente a las funciones *getReflectedWave* de los hijos, por lo que una vez más la llamada se propaga hacia abajo en el árbol. Una vez se obtienen los resultados, la primera función de la cadena de llamadas usa los coeficientes de dispersión para calcular la onda reflejada. A continuación se pasa a la función *setIncidentWave* del adaptador el resultado de llamar a la función *getReflectedWave* de la fuente de voltaje ideal. Una vez más, la función *setIncidentWave* del adaptador llamará a las funciones *setIncidentWave* de todos los hijos,

```
59  float OakdriveSim::oakdriveSimulation(float sample){
60                  //Input stage
61                  vin.setV(sample);
62                  vin.setIncidentWave(P1.getReflectedWave());
63                  P1.setIncidentWave(vin.getReflectedWave());
64
65                  //Voice filter
66                  opampvin.setV(bias.getVoltage());
67                  opampvin.setIncidentWave(S2.getReflectedWave());
68                  S2.setIncidentWave(opampvin.getReflectedWave());
69
70                  //Clipper
71                  clippercurrent.setCurrent(voicebias.getCurrent());
72                  mclipper.setIncidentWave(P2.getReflectedWave());
73                  P2.setIncidentWave(mclipper.getReflectedWave());
74
75                  //Low pass
76                  vclippersource.setV(mclipper.getVoltage() + bias.getVoltage());
77                  vclippersource.setIncidentWave(S3.getReflectedWave());
78                  S3.setIncidentWave(vclippersource.getReflectedWave());
79
80                  //High pass
81                  vlp.setV(C4.getVoltage());
82                  vlp.setIncidentWave(S5.getReflectedWave());
83                  S5.setIncidentWave(vlp.getReflectedWave());
84
85                  return -volumeresistor.getVoltage();
86          }
87
88          double OakdriveSim::getSampleRate()
89          {
90                  return mFs;
91          }
92
93          void OakdriveSim::changeVoice(float value) {
94                  mVoice = value;
95                  voiceresistorvalue = 10000 * mVoice + 1000;
96                  voicebias.updateImpedance(voiceresistorvalue);
97          }
98          void OakdriveSim::changeGain(float value) {
99                  mGain = value;
100                 gainresistorvalue = 1000 + 500000 * mGain;
101                 clippercurrent.updateImpedance(gainresistorvalue);
102         }
103         void OakdriveSim::changeTone(float value) {
104                 mTone = value;
105                 toneresistorvalue = 10000 + 50000 * mTone;
106                 tonepot.updateImpedance(toneresistorvalue);
107         }
108         void OakdriveSim::changeVolume(float value) {
109                 mVolume = value;
110                 volumeresistorvalue = 100000 * mVolume;
111                 volumeVdiv = (100000 - volumeresistorvalue) + 1000;
112                 volumeresistorVdiv.updateImpedance(volumeVdiv);
113                 volumeresistor.updateImpedance(volumeresistorvalue);
114         }
```

Listado 16.24: Contenido del archivo *OakdriveSim.c*, funciones.

pasándoles las ondas reflejadas usando los coeficientes de dispersión.

Este proceso se repite para cada etapa. Cada vez que se procesa una etapa, es posible tomar un valor para la siguiente tal y como se mostró en las figuras de la sección 15.2. Por ejemplo, al terminar el procesamiento de la primera etapa el miembro *bias* contendrá el voltaje necesario para actualizar el voltaje de *opampvin*. El voltaje final devuelto será la caida de potencial en el miembro *volumeresistor*.

16.4 Procesador y editor

Las clases para el editor y el procesador son en esencia idénticas a las presentadas en la sección 7.11. La única diferencia es que los *Sliders* creados se corresponden con los parámetros del efecto que nos ocupa en lugar de con los parámetros del retardo modulado, a saber: Voz, Ganancia, Tono y Volumen. Algo análogo sucede con el constructor del procesador, en el que habrá que inicializar los parámetros como se hizo en el listado 7.25 pero usando los nombres de los parámetros de este efecto.

Sin embargo, la cabecera de la clase de procesamiento y la implementación de algunas funciones contienen cambios significativos que vale la pena comentar. En el listado 16.25 se muestra la cabecera de la clase.

Como puede verse, la clase hereda de una clase nueva llamada *Timer*. Como su nombre indica, esta clase otorga la funcionalidad de un temporizador controlable mediante las funciones *startTimer* y *stopTimer*. Si el temporizador está activo, se llamará a la función *timerCallback* cada vez que transcurra el tiempo determinado en milisegundos en el parámetro de *startTimer*. El temporizador será usado para limitar la cantidad de veces que los parámetros pueden actualizarse a 100 por segundo. Esto es suficiente para dar una sensación de continuidad y de hecho podría reducirse bastante más.

En cuanto a los miembros, la clase contiene punteros a instancias de las clases de interpolación y diezmado, como en el plugin de retardo modulado. En este caso, dado que este tipo de procesamientos generan gran cantidad de armónicos, no se dará la opción al usuario de elegir. La interpolación estará siempre activada.

Los miembros *parameterchange* y *parameteraux* son arrays de booleanos y floats, respectivamente, usados para el cambio de parámetros asíncrono. Cuando se llame a la función *parameterChanged* el nuevo valor del parámetro será guardado en la posición correspondiente del array *parameteraux* y se marcará como verdadera la misma posición en el array *parameterchange*. Cuando el temporizador llame a la función *timerCallback* esta

```
#pragma once
#include "../JuceLibraryCode/JuceHeader.h"
#include "OakdriveSim.h"
#include "Resampling\Interpolatorx2.h"
#include "Resampling\Decimatorx2.h"

class OakdriveAudioProcessor : public AudioProcessor,
                                                                public AudioProcessorValueTreeState::Listener,
                                                                public Timer
{
public:
    //==============================================================================
    OakdriveAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif

    void processBlock (AudioSampleBuffer&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    double getTailLengthSeconds() const override;

    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;

    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;
        //Listener
        virtual void parameterChanged(const String &parameterID, float newValue) override;
        //Timer
        virtual void timerCallback() override;
private:
        std::unique_ptr<Interpolatorx2> ix;
        std::unique_ptr<Decimatorx2> dx;
        bool parameterchange[4];
        float parameteraux[4];
        std::unique_ptr<OakdriveSim> mOakdrivesim;
        void createSimulator(double sampleRate);
        void deleteSimulator();
        AudioProcessorValueTreeState parameters;
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (OakdriveAudioProcessor)
};
```

Listado 16.25: Contenido del archivo *PluginProcessor.h*.

```
1  void OakdriveAudioProcessor::parameterChanged(const String &parameterID, float newValue) {
2          parameteraux[parameterID.getTrailingIntValue()-1] = newValue;
3          parameterchange[parameterID.getTrailingIntValue()-1] = true;
4  }
5
6  void OakdriveAudioProcessor::timerCallback() {
7          if(parameterchange[0]){
8                  mOakdrivesim->changeVoice(parameteraux[0]);
9                  parameterchange[0] = false;
10         }
11         if (parameterchange[1]) {
12                 mOakdrivesim->changeGain(parameteraux[1]);
13                 parameterchange[1] = false;
14         }
15         if (parameterchange[2]) {
16                 mOakdrivesim->changeTone(parameteraux[2]);
17                 parameterchange[2] = false;
18         }
19         if (parameterchange[3]) {
20                 mOakdrivesim->changeVolume(parameteraux[3]);
21                 parameterchange[3] = false;
22         }
23 }
```

Listado 16.26: Funciones *parameterChanged* y *timerCallback*.

comprobará si los parámetros deben ser actualizados mediante el array de booleanos y actuará en consecuencia.

Por último, la clase contiene un puntero a una instancia de *OakdriveSim* para realizar el procesamiento de las muestras y dos funciones adicionales, *createSimulator* y *deleteSimulator*, que permiten crear y destruir el simulador y las clases de interpolación y diezmado cuando sea necesario.

En el listado 16.26 se muestra el código de las funciones responsables de actualizar los parámetros. Cuando un parámetro cambia de valor se usa el id del parámetro para acceder a las posiciones de los arrays *parameteraux* y *parameterchange* y modificar sus valores. Cuando se llama la función *timerCallback* esta comprueba todos los valores del array *parameterchange* y si alguno es verdadero llama a la función correspondiente del simulador para realizar el cambio.

En el listado 16.27 se muestra el código de las funciones *prepareToPlay* y *releaseResources*. Si antes de empezar la reproducción existe el simulador y su frecuencia de muestreo no es igual a la frecuencia de muestreo actual multiplicada por dos este se elimina llamando a la función *deleteSimulator*. Esta función detiene el temporizador y elimina el simulador y las instancias del interpolador y el diezmador.

Si no existe el simulador, bien porque no existía o porque se ha eliminado, este se crea llamando a la función *createSimulator*. Esta última función crea una instancia de la clase de simulación pasando al constructor el valor actual de cada parámetro del plugin. Posteriormente, crea instancias de las clases de interpolación y diezmado. Finalmente, inicia el temporizador de actualización de parámetros.

Cuando el procesamiento termina simplemente se llama a la función *deleteSimulator*, que

```
1  void OakdriveAudioProcessor::prepareToPlay(double sampleRate, int samplesPerBlock)
2  {
3          if (mOakdrivesim && sampleRate != (mOakdrivesim->getSampleRate()/2)) {
4                  deleteSimulator();
5          }
6          if (!mOakdrivesim) {
7                  createSimulator(sampleRate);
8          }
9  }
10
11 void OakdriveAudioProcessor::releaseResources()
12 {
13         if (mOakdrivesim) {
14                 deleteSimulator();
15         }
16 }
17
18 void OakdriveAudioProcessor::createSimulator(double sampleRate)
19 {
20         mOakdrivesim = std::make_unique<OakdriveSim>(sampleRate*2,
21                 parameters.getParameter("1")->getValue(),
22                 parameters.getParameter("2")->getValue(),
23                 parameters.getParameter("3")->getValue(),
24                 parameters.getParameter("4")->getValue()
25         );
26         ix = std::make_unique<Interpolatorx2>();
27         dx = std::make_unique<Decimatorx2>();
28         startTimer(10);
29 }
30
31 void OakdriveAudioProcessor::deleteSimulator()
32 {
33         stopTimer();
34         mOakdrivesim.reset();
35         ix.reset();
36         dx.reset();
37 }
```

Listado 16.27: Funciones *prepareToPlay*, *releaseResources* y funciones auxiliares.

lleva a cabo las operaciones descritas anteriormente.

En el listado 16.28 se muestra el código de la función *processBlock*. El procesamiento de cada bloque es trivial. Como en la implementación del retardo modulado, en primer lugar se limpia el canal derecho. Posteriormente, se introduce una muestra en el interpolador y se extraen dos para ser procesadas por el simulador. Finalmente, ambas muestras procesadas se introducen en el diezmador y se extrae el resultado final.

16.5 Conclusiones

El código presentado constituye la implementación de un plugin de saturación basado en los modelos del circuito derivados en los capítulos anteriores. El código servirá además como guía cuando se realice la implementación en C. Las clases implementadas para simular los elementos del circuito permiten su uso directo como nodos en un árbol WDF. En futuros trabajos sería interesante encapsular el árbol en una clase de tal modo que su construcción no requiriera todas las acciones llevadas a cabo en la clase de simulación.

```
1  void OakdriveAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
2  {
3
4      //Clean right channel
5          float* channelData = buffer.getWritePointer(1);
6          for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
7          {
8                  channelData[sample] = 0;
9          }
10
11         //Process left channel
12         channelData = buffer.getWritePointer(0);
13         if(mOakdrivesim)
14                 for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
15                 {
16                                 ix->inputValue(channelData[sample]);
17                                 dx->inputValue(mOakdrivesim->oakdriveSimulation(ix->getNext()));
18                                 dx->inputValue(mOakdrivesim->oakdriveSimulation(ix->getNext()));
19                                 channelData[sample] = dx->getNext();
20                 }
21 }
```

Listado 16.28: Función *processBlock*.

CAPÍTULO 17

Pruebas del plugin oakdrive

Este capítulo pertenece al proceso de pruebas del efecto zendrive. En él hago pruebas con el plugin implementado y lo comparo con el procesamiento en SPICE de una muestra de audio. El archivo dll resultante de compilar el código del capítulo anterior así como instrucciones para probarlo pueden encontrarse en la ruta *./Proyecto/Saturación/DLL*.

17.1 Valor de los potenciómetros

Todas las pruebas han sido realizadas con los potenciómetros en las siguientes posiciones: Volumen, 25 %. Ganancia, 50 %. Voz, 50 %. Tono, 50 %. De este modo los valores de las resistencias coinciden con las de los esquemas en la sección 15.1.

17.2 Comparación con SPICE

Esta prueba se llevó a cabo como se describe a continuación. Usando el modelo SPICE diseñado en la sección 15.1 procesé en LTSpice una señal de guitarra eléctrica contenida en un archivo wave. Posteriormente, cree un nuevo proyecto en Reaper y añadí en una pista con el plugin implementado en la lista de inserciones el mismo archivo wave. Junto a esta pista cree otra para situar en ella el archivo resultante del procesamiento mediante SPICE.

Al "congelar" pistas en Reaper se aplican todos los procesamientos de las inserciones a las muestras contenidas en ellas y se copia el resultado a una nueva pista. Una vez configuradas las pistas como se decribió anteriormente congelé la pista en la que estaba insertado el plugin. Así, pude comparar las tres señales en el dominio del tiempo: la señal en limpio, la señal procesada mediante SPICE y la señal procesada usando el plugin. Para poder mostrar los resultados en esta memoria hice una captura de pantalla de las formas de onda y las vectoricé usando el programa Inkscape. El resultado se muestra en la figura 17.1.

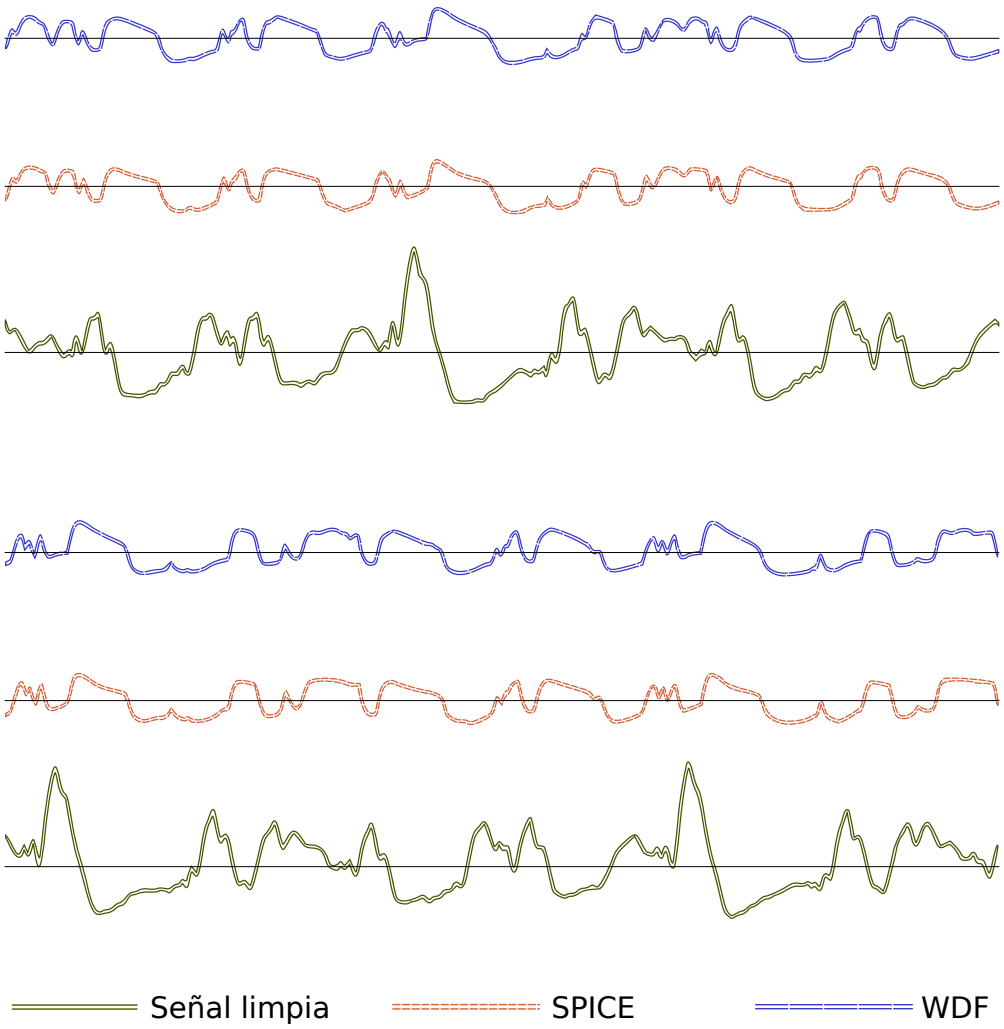


Figura 17.1: Comparación de procesamiento de una señal mediante SPICE y un filtro digital de ondas.

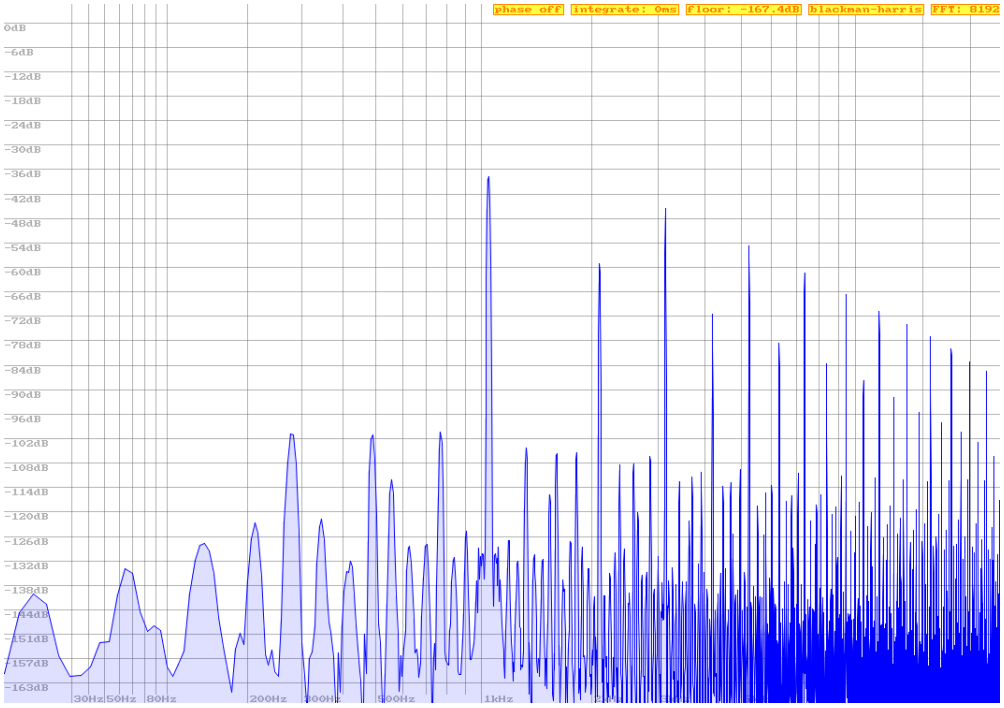


Figura 17.2: Respuesta del plugin oakdrive a una señal de 1046.502 Hz.

En mi opinión, el resultado es muy bueno. No obstante, sería interesante comparar el procesamiento con el realizado por el circuito real. Al oido, no se aprecian diferencias entre el procesamiento SPICE y el realizado por el plugin.

17.3 Análisis del espectro

Siguiendo un procedimiento análogo al de la sección 8.1 introduje en el plugin una señal de 1046.502 Hz para visualizar el espectro. El resultado se muestra en la figura 17.2. Como puede verse, predominan los armónicos de orden impar. También aparecen frecuencias no relacionadas con la original, sospecho que debido principalmente al aliasing. Una dicusión sobre el problema del aliasing en los efectos de saturación digitales puede encontrarse en un artículo al respecto de Pablo Fernández-Cid [64]. Existen actualmente métodos para reducir la distorsión de aliasing en efectos de saturación que sería interesante considerar en futuros trabajos [65].

17.4 Comentarios adicionales

En ocasiones, al introducir el plugin como inserción en una pista, se produce un chasquido desagradable que puede visualizarse como un aumento súbito en el marcador de dB de la

pista. En futuras versiones este inconveniente debería investigarse, ya que no he sido capaz de encontrar el motivo. Se debe, probablemente, a los valores iniciales de voltaje, corriente e impedancia dados a los componentes WDF.

17.5 Conclusiones

El plugin implementado simula el circuito en la banda de frecuencias de interés con bastante precisión. No obstante, se ha supuesto un amplificador operacional ideal lo cual posiblemente resta carácter al sonido. En futuros trabajos sería interesante construir un modelo más complejo del amplificador operacional. El análisis del espectro muestra una elevada distorsión de aliasing a pesar de la interpolación, que no obstante se considera aceptable y hasta cierto punto inevitable en los efectos digitales de saturación.

CAPÍTULO 18

Implementación del efecto oakdrive en el C674x

En este capítulo se explica el código escrito en C para hacer funcionar el algoritmo del efecto oakdrive en el núcleo C674x del procesador OMAP-L138. El código mostrado funciona en conjunto con los controladores implementados en el capítulo 10 para completar el sistema de procesamiento en tiempo real. El código de este capítulo puede encontrarse en la ruta *./Proyecto/Saturación/C674x*.

18.1 Interpolador y diezmador

Dado que el procesamiento sin interpolación no es aceptable, en este caso es necesario implementar versiones en C del interpolador y el diezmador. Para ello se usará el buffer circular implementado en la sección 11.2 y las operaciones asociadas. Las implementaciones realizadas son análogas a las mostradas en las secciones 7.5 y 7.6, aunque existen algunas diferencias.

En el listado 18.1 se muestra el contenido del archivo *Interpolatorx2.h*. En él se define una estructura que contiene un solo buffer circular y se declaran tres funciones usadas para la

```
1  #ifndef RESAMPLING_INTERPOLATORX2_H_
2  #define RESAMPLING_INTERPOLATORX2_H_
3
4  #include <ti/csl/tistdtypes.h>
5  #include "CircularBuffer.h"
6
7  #define IX2_B_SIZE 64
8  #define IX2_F_SIZE 25
9
10 typedef struct Interpolatorx2 {
11    CircularBuffer filter;
12 }Interpolatorx2;
13
14 void Interpolatorx2_Init(Interpolatorx2* ix2);
15 void Interpolatorx2_Free(Interpolatorx2* ix2);
16 void Interpolatorx2_Interpolatex2_256(Interpolatorx2* ix2,float* input, float* output);
17
18 #endif /* RESAMPLING_INTERPOLATORX2_H_ */
```

Listado 18.1: Contenido del archivo *Interpolatorx2.h*.

```
1  #include "Interpolatorx2.h"
2  float int_filterC_1[25] = { 0.000535f,  -0.0005685f,   0.0008606f,  -0.001243f,
3  0.001733f, -0.0023495f,   0.0031137f,  -0.0040499f,   0.0051839f,  -0.006548f,   0.0081774f,
4  -0.0101163f, 0.01242f,  -0.0151585f,   0.0184269f,  -0.0223577f,   0.0271442f,  -0.0330818f,
5  0.0406496f,-0.0506774f,   0.0647381f,  -0.086212f,   0.1239052f,  -0.21014f,   0.6359283f };
6
7  void Interpolatorx2_Init(Interpolatorx2* ix2)  {
8    CircularBuffer_Init(&ix2->filter, IX2_B_SIZE);
9  }
10
11 void Interpolatorx2_Free(Interpolatorx2* ix2)  {
12   CircularBuffer_Free(&ix2->filter);
13 }
14
15 void Interpolatorx2_Interpolatex2_256(Interpolatorx2* ix2, float* input, float* output){
16    Int32 i;
17    Int16 j;
18     #pragma MUST_ITERATE(256,256,)
19    for(i=0; i<256;i++){
20        float res = 0;
21        ix2->filter.first[(ix2->filter.writeIndex++) & ix2->filter.mask] = *input++;
22        float* spointer = &(ix2->filter.first[(ix2->filter.writeIndex + (~(0))) & ix2->filter.mask]);
23        float* epointer = &(ix2->filter.first[(ix2->filter.writeIndex + (~(IX2_F_SIZE*2 - 1))) & ix2->filter
              .mask]);
24        float* cpointer = int_filterC_1;
25        #pragma MUST_ITERATE(IX2_F_SIZE,IX2_F_SIZE,)
26        for(j=0;j<IX2_F_SIZE;j++){
27            res += (*spointer-- + *epointer++) * *cpointer++;
28            if(spointer < ix2->filter.first) spointer = ix2->filter.last;
29            if(epointer > ix2->filter.last) epointer = ix2->filter.first;
30        }
31        *output++ = res;
32        *output++ = ix2->filter.first[(ix2->filter.writeIndex + (~(IX2_F_SIZE-1))) & ix2->filter.mask];
33    }
34 }
```

Listado 18.2: Contenido del archivo *Interpolatorx2.c*.

inicialización, la eliminación y el procesamiento.

En el listado 18.2 se muestra el contenido del archivo *Interpolatorx2.c*. En el array *int_filterC_l* se guardan los coeficientes de la componente polifásica del filtro paso bajo. Como se mencionó en un capítulo anterior, al ser el filtro simétrico solo hace falta almacenar la mitad de los coeficientes. La única operación que realizan las funciones de inicialización y eliminación es llamar a las funciones de inicialización o eliminación del buffer circular.

En cuanto al procesamiento, este se hace usando bloques de 256 muestras. Cuando se llama a la función de procesamiento es necesario pasarle un puntero a una estructura de tipo *Interpolatorx2* que contenga el buffer circular a utilizar. El puntero *input* debe apuntar al array de entrada y el puntero *output* al de salida. Este último array debe tener un tamaño igual al doble del tamaño del array de entrada.

El procesamiento dentro del bucle empieza introduciendo la muestra de entrada en el buffer e incrementando el puntero, de tal forma que apunte a la siguiente muestra del array de entrada. Posteriormente, se declaran los punteros *spointer* y *epointer*. El primero de ellos apunta a la primera posición del buffer circular. El segundo apunta a la posición 49. También se declara un puntero al primero de los coeficientes del filtro.

Dentro del bucle interno se calcula el resultado de aplicar el filtro simétrico. Para ello, se usa el acumulador *res* para almacenar los resultados intermedios. En cada iteración se incrementan *cpointer* y *epointer* y se decrementa *spointer*. Como siempre, es necesario

```
1  #ifndef RESAMPLING_DECIMATORX2_H_
2  #define RESAMPLING_DECIMATORX2_H_
3
4  #include <ti/csl/tistdtypes.h>
5  #include "CircularBuffer.h"
6
7  #define DX2_B_SIZE 64
8  #define DX2_F_SIZE 25
9
10 typedef struct Decimatorx2 {
11   CircularBuffer filter;
12   CircularBuffer delay;
13   float a;
14   float b;
15 }Decimatorx2;
16
17 void Decimatorx2_Init(Decimatorx2* dx2);
18 void Decimatorx2_Free(Decimatorx2* dx2);
19 void Decimatorx2_Decimatex2_256(Decimatorx2* dx2,float* input, float* output);
20
21 #endif /* RESAMPLING_DECIMATORX2_H_ */
```

Listado 18.3: Contenido del archivo *Decimatorx2.h*.

comprobar que los punteros se encuentren dentro de los límites del array usado para almacenar el buffer circular.

Finalmente se guardan dos muestras consecutivas en el array de salida, una por cada componente polifásica del filtro. Nótese que se ha utilizado un solo buffer circular. Dado que una de las componentes polifásicas es un simple retardo, tal y como se mencionó en la sección 5.3.2, no es necesario usar un segundo buffer circular en este caso.

Nótese que el tamaño del buffer circular debe ser una potencia de 2, de ahí que se haya usado un tamaño total de 64 muestras. En realidad solo se usan 50 muestras para aplicar el filtro, de ahí que *spointer* apunte a la posición 49 al comienzo del bucle de procesamiento.

En cuanto al diezmador, en el listado 18.3 se muestra el contenido del archivo *Decimatorx2.h*. Nótese que en este caso la estructura de datos contiene dos buffers circulares, además de dos variables auxiliares *a* y *b*. Por otra parte, el array de entrada debe tener el doble de tamaño que el array de salida, en este caso 512 muestras.

En el listado 18.4 se muestra el contenido del archivo *Decimatorx2.c*. El procesamiento se realiza sin cambios hasta la linea 35. La diferencia es que ahora el resultado de aplicar el filtro se guarda en la variable *a* y se introduce la siguiente muestra de entrada en el buffer circular *delay*. Posteriormente, se toma una muestra retrasada 25 posiciones de este buffer y se guarda en la variable *b*. Esta se multiplica por 0.5 y se suma a la variable *a* para obtener el resultado final que se almacena en el array de salida. Para una explicación más detallada consultar la sección 5.3.2.

```
#include "Decimatorx2.h"

float dec_filterC_1[25] = { 0.0002675f,  -0.0002843f,   0.0004303f,
                     -0.0006215f,   0.0008665f,  -0.0011748f,   0.0015569f,  -0.0020249f,   0.0025919f,
                          -0.003274f,
                      0.0040887f,  -0.0050582f,   0.00621f,  -0.0075792f,   0.0092134f,  -0.0111788f,
                           0.0135721f,
                     -0.0165409f,   0.0203248f,  -0.0253387f,   0.032369f,  -0.043106f,   0.0619526f,
                          -0.10507f,
                      0.3179641f };

void Decimatorx2_Init(Decimatorx2* dx2)  {
  CircularBuffer_Init(&dx2->filter, DX2_B_SIZE);
  CircularBuffer_Init(&dx2->delay, DX2_B_SIZE/2);
}

void Decimatorx2_Free(Decimatorx2* dx2)  {
  CircularBuffer_Free(&dx2->filter);
  CircularBuffer_Free(&dx2->delay);
}

void Decimatorx2_Decimatex2_256(Decimatorx2* dx2,float* input, float* output){
   Int32 i;
   Int16 j;
   #pragma MUST_ITERATE(256,256,)
   for(i=0; i<256;i++){
       float res = 0;
       dx2->filter.first[(dx2->filter.writeIndex++) & dx2->filter.mask] = *input++;
       float* spointer = &(dx2->filter.first[(dx2->filter.writeIndex + (~(0))) & dx2->filter.mask]);
       float* epointer = &(dx2->filter.first[(dx2->filter.writeIndex + (~(DX2_F_SIZE*2 - 1))) & dx2->filter
            .mask]);
       float* cpointer = dec_filterC_1;
       #pragma MUST_ITERATE(DX2_F_SIZE,DX2_F_SIZE,)
       for(j=0;j<DX2_F_SIZE;j++){
           res += (*spointer-- + *epointer++) * *cpointer++;
           if(spointer < dx2->filter.first) spointer = dx2->filter.last;
           if(epointer > dx2->filter.last) epointer = dx2->filter.first;
       }
       dx2->a = res;
       dx2->delay.first[(dx2->delay.writeIndex++) & dx2->delay.mask] = *input++;
       dx2->b = dx2->delay.first[(dx2->delay.writeIndex + (~(DX2_F_SIZE))) & dx2->delay.mask];
       *output++ = dx2->a + dx2->b*0.5f;
   }
}
```

Listado 18.4: Contenido del archivo *Decimatorx2.c*.

```
1  #ifndef WDF_WDFELEMENT_H_
2  #define WDF_WDFELEMENT_H_
3
4  #include <ti/csl/tistdtypes.h>
5  #include <stdlib.h>
6
7  typedef struct _WDFElement WDFElement;
8
9  typedef float (*fPtr_WDF)(WDFElement*);
10 typedef void (*vPtr_WDF_float)(WDFElement*,float);
11 typedef void (*vPtr_WDF_WDF)(WDFElement*,WDFElement*);
12 typedef void (*vPtr_WDF)(WDFElement*);
13
14 typedef struct _WDFElement{
15     Ptr derived;
16     vPtr_WDF del;
17     float mR;
18     float mA;
19     float mB;
20     struct _WDFElement *mParent;
21     fPtr_WDF getV;
22     fPtr_WDF getC;
23     fPtr_WDF getReflectedWave;
24     vPtr_WDF_float setIncidentWave;
25     vPtr_WDF_float updateImpedance;
26     vPtr_WDF_WDF linkTree;
27 }WDFElement;
28
29 float getVoltage(WDFElement* e);
30 float getCurrent(WDFElement* e);
31
32 WDFElement* new_WDFElement();
33 void delete_WDFElement(WDFElement* e);
34
35 #endif /* WDF_WDFELEMENT_H_ */
```

Listado 18.5: Contenido del archivo *WDFElement.h*.

18.2 Pseudo-Clases en C para elementos WDF

Para implementar el plugin usando C se usará en este caso una estructura de pseudo-clases y un procesamiento muestra a muestra. Dado que el nivel de abstracción de las estructuras WDF es alto considero que vale la pena probar esta implementación y evaluar los resultados.

A grandes rasgos, la idea consiste en definir en una cabecera una estructura de datos que contenga todos los miembros y punteros a las funciones de la clase. De este modo, incluso es posible representar algo similar a la herencia.

Tomemos como ejemplo el caso de la clase abstracta *WDFElement* implementada en la sección 16.2.1. En el listado 18.5 se muestra el archivo de cabecera en C.

El código empieza asignando el nombre *WDFElement* al tipo *struct _WDFElement*. Esto se hace para poder declarar un puntero dentro de una estructura que apunte a otra estructura del mismo tipo, como se verá posteriormente.

A continuación se asignan nombres a varios tipos de punteros a función. Por ejemplo, en la linea 9 se asigna el nombre *fPtr_WDF* al tipo de un puntero a una función que devuelve un float y toma como argumento un puntero a una estructura de tipo *WDFElement*.

De forma análoga, en la linea 10 se da el nombre *vPtr_WDF_float* al tipo de un puntero a una función sin retorno que toma como parámetros un puntero a una estructura de tipo

```
1  #include "WDFElement.h"
2
3  WDFElement* new_WDFElement(vPtr_WDF del, fPtr_WDF gRW, vPtr_WDF_float sIW, vPtr_WDF_float uI, vPtr_WDF_WDF
       lT){
4      WDFElement* ne = malloc(sizeof(WDFElement));
5      ne->mA = 0;
6      ne->mB = 0;
7      ne->mR = 0;
8      ne->del = del;
9      ne->mParent = NULL;
10     ne->getV = &getVoltage;
11     ne->getC = &getCurrent;
12     ne->getReflectedWave = gRW;
13     ne->setIncidentWave = sIW;
14     ne->updateImpedance = uI;
15     ne->linkTree = lT;
16     return ne;
17 }
18
19 void delete_WDFElement(WDFElement* e){
20     free(e);
21 }
22
23 float getVoltage(WDFElement* e){
24     return (e->mA+e->mB)/2;
25 }
26 float getCurrent(WDFElement* e){
27     return (e->mA-e->mB)/(2*e->mR);
28 }
```

Listado 18.6: Contenido del archivo *WDFElement.c*.

WDFElement y un float. En general, la convención utilizada es empezar el nombre del tipo con la inicial del tipo devuelto, añadir las letras Ptr y a continuación separar el nombre de cada tipo de los parámetros de la función con guiones bajos.

A partir de la linea 14 se define el contenido de una estructura *WDFElement*. El tipo *Ptr* se encuentra definido en el archivo *tistdtypes.h* y es un alias de un puntero a un tipo indefinido. La estructura contiene un elemento de este tipo, llamado *derived*, que puede usarse para apuntar a otra estructura que contenga los datos de una pseudo-clase que herede de la actual. Así, una pseudo-clase completa que herede de *WDFElement* contendrá una estructura *WDFElement* cuyo puntero *derived* apuntará a otra estructura que contendrá a su vez los miembros y funciones de la pseudo-clase hija.

El puntero *del* apuntará al destructor de la pseudo-clase. El resto de punteros a función apuntarán a funciones que realicen las operaciones definidas en la sección 16.2.1. Igualmente, los miembros de la estructura tienen exactamente el mismo propósito que los definidos en la sección 16.2.1.

Finalmente, en la cabecera se declaran las funciones comunes a todas las clases WDF. Estas son por un lado las funciones de obtención de voltaje y corriente y por el otro el constructor y el destructor, llamados *new_WDFElement* y *delete_WDFElement*, respectivamente.

Pasemos ahora a explicar la implementación. En el listado 18.6 se muestra el código correspondiente.

En primer lugar, habrá que pasarle al constructor punteros a todas las funciones exceptuando las definidas en la propia clase abstracta. El nombre de estas funciones se ha abreviado

```
1  #ifndef WDF_CAPACITOR_H_
2  #define WDF_CAPACITOR_H_
3
4  #include <ti/csl/tistdtypes.h>
5  #include "WDFElement.h"
6
7  typedef struct  {
8      float mC;
9      float mFs;
10     WDFElement* wdfelement;
11 } Capacitor;
12
13 WDFElement* newCapacitor(float C, int Fs);
14 void deleteCapacitor(WDFElement* e);
15
16 float capacitor_getReflectedWave(WDFElement* e);
17 void capacitor_setIncidentWave(WDFElement* e, float A);
18 void capacitor_updateImpedance(WDFElement* e, float R);
19 void capacitor_linkTree(WDFElement* e, WDFElement* parent);
20
21 #endif /* WDF_CAPACITOR_H_ */
```

Listado 18.7: Contenido del archivo *Capacitor.h*.

en los parámetros, pero dadas las asignaciones que se hacen en el constructor es evidente la correspondencia.

La primera acción que realiza el constructor es reservar memoria para una estructura de tipo *WDFElement*. Posteriormente inicializa los miembros de la pseudo-clase y asigna a cada puntero a función uno de los argumentos. Por ejemplo, *gRW* es asignado a *getReflectedWave*. Los punteros *getV* y *getC* son en cambio asignados a referencias de implementaciones que proporciona la clase abstracta, incluidas en el mismo archivo.

De este modo, cuando se llame a una función mediante un puntero contenido en una estructura de tipo *WDFElement* se estará llamando realmente a una función implementada en el archivo de alguna pseudo-clase heredera que habrá pasado una referencia de su implementación a este pseudo-constructor.

Tomemos como ejemplo el caso del condensador. El código de la cabecera se muestra en el listado 18.7. En ella se define una estructura llamada *Capacitor* que contiene los miembros adicionales del condensador, la capacidad y la frecuencia de muestreo, y un puntero a la estructura *WDFElement* que contendrá el resto de datos y funciones. Por otro lado, se declaran las funciones propias del condensador.

Estas funciones son definidas en el archivo de implementación mostrado en el listado 18.8. Como puede verse, el constructor empieza reservando memoria para una estructura de tipo *Capacitor*. Tras asignar los valores de los parámetros a los miembros de esta estructura llama al constructor de la pseudo-clase abstracta *WDFElement* pasándole referencias a las funciones implementadas en este archivo. Finalmente, se le da a la impedancia de puerto el valor correspondiente, se hace que el puntero *derived* de la estructura *WDFElement* apunte a la estructura *Capacitor* recién creada y se devuelve la estructura *WDFElement* completa.

Las acciones del destructor consisten en liberar las dos estructuras de datos que forman la

```
1  #include "Capacitor.h"
2
3  WDFElement* newCapacitor(float C, int Fs){
4      Capacitor* cap = malloc(sizeof(Capacitor));
5      cap->mC = C;
6      cap->mFs = Fs;
7      cap->wdfelement = new_WDFElement(&deleteCapacitor, &capacitor_getReflectedWave, &
           capacitor_setIncidentWave, &capacitor_updateImpedance, &capacitor_linkTree);
8      cap->wdfelement->mR = 1 / (2 * Fs*C);
9      cap->wdfelement->derived = cap;
10     return cap->wdfelement;
11 }
12
13 void deleteCapacitor(WDFElement* e){
14     free(e->derived);
15     free(e);
16 }
17
18 float capacitor_getReflectedWave(WDFElement* e){
19     e->mB = e->mA;
20     return e->mB;
21 }
22
23 void capacitor_setIncidentWave(WDFElement* e,float A){
24     e->mA = A;
25 }
26
27 void capacitor_updateImpedance(WDFElement* e, float R){
28     Capacitor* cap = (Capacitor*)(e->derived);
29     e->mR = 1/(2*cap->mFs*e->mR);
30     if(e->mParent)
31         e->mParent->updateImpedance(e->mParent,e->mR);
32 }
33
34 void capacitor_linkTree(WDFElement* e,WDFElement* parent){
35     e->mParent = parent;
36 }
```

Listado 18.8: Contenido del archivo *Capacitor.c*.

pseudo-clase.

Nótese que finalmente se tiene una estructura *WDFElement* con punteros a funciones que realizan las operaciones de un condensador. Esta estructura contiene además un puntero *derived* que apunta a una estructura donde se encuentran los miembros de la pseudo-clase *Capacitor*.

Una vez se entiende el patrón, realizar la traducción de todas las clases es tedioso pero sencillo. Por tanto, no mostraré aquí el código del resto de archivos que puede no obstante encontrarse en los adjuntos a este documento.

18.3 Interpolador de Fritsch-Carlson

En esta sección se muestra la implementación realizada en C del intepolador de Fritsch-Carlson. En el listado 18.9 se muestra la cabecera. En ella se define, como en otras ocasiones, una estructura que contendrá los datos del interpolador. También se declaran las funciones de inicialización, borrado e interpolación. La implementación de estas funciones es practicamente idéntica a la realizada en la sección 16.1, por lo que no insistiré en ella. De hecho, la única diferencia es el error en caso de que los puntos de partición no sean válidos y la forma de eliminar los datos en el destructor.

```
1  #ifndef WDF_MATH_FCINTERPOLATOR_H_
2  #define WDF_MATH_FCINTERPOLATOR_H_
3
4  #include <ti/csl/tistdtypes.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <math.h>
8
9  typedef struct  {
10     float* mX;
11     float* mY;
12     float* mD;
13     Int16 mN;
14 } FCInterpolator;
15
16 FCInterpolator* newFCInterpolator(float* x, float* y, int N);
17 void deleteFCInterpolator(FCInterpolator* si);
18 float interpolate(FCInterpolator* si, float x);
19
20 #endif /* WDF_MATH_FCINTERPOLATOR_H_ */
```

Listado 18.9: Contenido del archivo *FCInterpolator.h*.

18.4 Simulador en C

En esta sección se implementa el simulador del circuito en C mediante un árbol WDF. Existen diferencias con la implementación realizada en la sección 16.3 ya que en este caso es necesario que el simulador pueda incluirse en el vector de efectos implementado en la sección 11.1.

En el listado 18.10 se muestra el contenido del archivo *OakdriveSimulation.h*. Como puede verse, los elementos incluidos dentro del tipo de estructura *OakdriveSimulation* coinciden casi totalmente con los miembros de la clase implementada en la sección 16.3. La única diferencia es que esta estructura incluye además un interpolador y un diezmador para realizar los procesos de interpolación y diezmado junto con el propio procesamiento llevado a cabo por el efecto. En el archivo de cabecera se declaran además cuatro funciones, dos dedicadas a inicializar y liberar la estructura y dos dedicadas a procesamiento. Una de las funciones de procesamiento realiza el procesamiento tomando como parámetro un buffer de 256 muestras. La otra realiza el procesamiento muestra a muestra.

En el listado 18.11 se muestra la función de inicialización. Esta es practicamente idéntica al constructor implementado en la sección 16.3. La única diferencia es que además se llama a las funciones de inicialización del interpolador y el diezmador al final del código.

Nótese que se llama a los constructores de las pseudo-clases de diferentes tipos y el resultado se almacena siempre en un puntero a una estructura *WDFElement*. Esto hace posible construir un árbol de elementos WDF tal y como se hizo en la implementación en C++.

En el listado 18.12 se muestra la función de liberación. Recordemos que el puntero *del* de la estructura *WDFElement* siempre apunta al destructor adecuado. Un inconveniente de las pseudo-clases implementadas es que utilizan memoria dinámica, por lo que su destructor

```
#ifndef OAKDRIVESIMULATION_H_
#define OAKDRIVESIMULATION_H_

#include <ti/csl/tistdtypes.h>
#include "WDF/WDFElement.h"
#include "WDF/IdealVoltageSource.h"
#include "WDF/ResistiveVoltageSource.h"
#include "WDF/Resistor.h"
#include "WDF/Capacitor.h"
#include "WDF/SeriesThreePortAdaptor.h"
#include "WDF/ParallelThreePortAdaptor.h"
#include "WDF/MosfetClipperSpline.h"
#include "WDF/ResistiveCurrentSource.h"
#include "Resampling/Interpolatorx2.h"
#include "Resampling/Decimatorx2.h"

typedef struct {
    Int32 fs;
    //Input
    WDFElement* vin;
    WDFElement* bias;
    WDFElement* R6;
    WDFElement* C6;
    WDFElement* S1;
    WDFElement* P1;

    //Voice filter
    float voice;
    float voiceresistorvalue;
    WDFElement* C5;
    WDFElement* voicebias;
    WDFElement* opampvin;
    WDFElement* S2;

    //Clipper
    float gain;
    float gainresistorvalue;
    WDFElement* mclipper;
    WDFElement* clippercurrent;
    WDFElement* C7;
    WDFElement* P2;

    //Low pass
    float tone;
    float toneresistorvalue;
    WDFElement* vclippersource;
    WDFElement* tonepot;
    WDFElement* C4;
    WDFElement* S3;

    //High pass
    float volume;
    float volumeresistorvalue;
    float volumeVdiv;
    WDFElement* vlp;
    WDFElement* volumeresistorVdiv;
    WDFElement* volumeresistor;
    WDFElement* C3;
    WDFElement* S4;
    WDFElement* S5;
    Interpolatorx2 ix2;
    Decimatorx2 dx2;
}OakdriveSimulation;

void OakdriveSimulation_Init(OakdriveSimulation* os, Int32 fs);
void OakdriveSimulation_Free(OakdriveSimulation* os);
void OakdriveSimulation_ProcessBuffer256(OakdriveSimulation* os, float* input);
float OakdriveSimulation_ProcessBufferSample(OakdriveSimulation* os, float input);

#endif /* OAKDRIVESIMULATION_H_ */
```

Listado 18.10: Contenido del archivo *OakdriveSimulation.h*.

```
void OakdriveSimulation_Init(OakdriveSimulation* os, Int32 fs){
    os->fs = fs;
    os->voice = 0.5f;
    os->voiceresistorvalue = 10000 * os->voice + 1000;
    os->gain = 0.5f;
    os->gainresistorvalue = 1000 + 500000 * os->gain;
    os->tone = 0.5f;
    os->toneresistorvalue = 20000 + 50000 * os->tone;
    os->volume = 0.75f;
    os->volumeresistorvalue = 100000 * os->volume;
    os->volumeVdiv = (100000 - os->volumeresistorvalue) + 1000;

    //Input stage
    os->vin = newIdealVoltageSource(0,1000);
    os->bias = newResistiveVoltageSource(4.5f, 470000);
    os->R6 = newResistor(2000000);
    os->C6 = newCapacitor((470 * pow(10, -9)), os->fs);
    os->S1 = newSeriesThreePortAdaptor(os->C6, os->bias);
    os->P1 = newParallelThreePortAdaptor(os->R6, os->S1);

    //Voice filter
    os->C5 = newCapacitor((100 * pow(10, -9)), fs);
    os->voicebias = newResistiveVoltageSource(4.5f, os->voiceresistorvalue);
    os->opampvin = newIdealVoltageSource(0, 1000);
    os->S2 = newSeriesThreePortAdaptor(os->C5, os->voicebias);

    //Clipper
    os->mclipper = newMosfetClipperSpline(100);
    os->C7 = newCapacitor((100 * pow(10, -12)), fs);
    os->clippercurrent = newResistiveCurrentSource(0.0, os->gainresistorvalue);
    os->P2 = newParallelThreePortAdaptor(os->clippercurrent, os->C7);

    //Low pass
    os->vclippersource = newIdealVoltageSource(0, 1000);
    os->tonepot = newResistor(os->toneresistorvalue);
    os->C4 = newCapacitor((3.3*pow(10, -9)), os->fs);
    os->S3 = newSeriesThreePortAdaptor(os->tonepot, os->C4);

    //High pass
    os->vlp = newIdealVoltageSource(0, 1000);
    os->volumeresistorVdiv = newResistor(os->volumeVdiv);
    os->volumeresistor = newResistor(os->volumeresistorvalue);
    os->C3 = newCapacitor((470 * pow(10, -9)), os->fs);
    os->S4 = newSeriesThreePortAdaptor(os->volumeresistorVdiv, os->C3);
    os->S5 = newSeriesThreePortAdaptor(os->volumeresistor, os->S4);

    os->P1->linkTree(os->P1,os->vin);
    os->P1->updateImpedance(os->P1,0);
    os->S2->linkTree(os->S2,os->opampvin);
    os->S2->updateImpedance(os->S2,0);
    os->P2->linkTree(os->P2,os->mclipper);
    os->P2->updateImpedance(os->P2,0);
    os->S3->linkTree(os->S3,os->vclippersource);
    os->S3->updateImpedance(os->S3,0);
    os->S5->linkTree(os->S5,os->vlp);
    os->S5->updateImpedance(os->S5,0);

    Interpolatorx2_Init(&os->ix2);
    Decimatorx2_Init(&os->dx2);
}
```

Listado 18.11: Función de inicialización contenida en el archivo *OakdriveSimulation.c*.

```
1  void OakdriveSimulation_Free(OakdriveSimulation* os){
2      //Input stage
3      os->vin->del(os->vin);
4      os->bias->del(os->bias);
5      os->R6->del(os->R6);
6      os->C6->del(os->C6);
7      os->S1->del(os->S1);
8      os->P1->del(os->P1);
9
10     //Voice filter
11     os->C5->del(os->C5);
12     os->voicebias->del(os->voicebias);
13     os->opampvin->del(os->opampvin);
14     os->S2->del(os->S2);
15
16     //Clipper
17     os->mclipper->del(os->mclipper);
18     os->C7->del(os->C7);
19     os->clippercurrent->del(os->clippercurrent);
20     os->P2->del(os->P2);
21
22     //Low pass
23     os->vclippersource->del(os->vclippersource);
24     os->tonepot->del(os->tonepot);
25     os->C4->del(os->C4);
26     os->S3->del(os->S3);
27
28     //High pass
29     os->vlp->del(os->vlp);
30     os->volumeresistorVdiv->del(os->volumeresistorVdiv);
31     os->volumeresistor->del(os->volumeresistor);
32     os->C3->del(os->C3);
33     os->S4->del(os->S4);
34     os->S5->del(os->S5);
35
36     Interpolatorx2_Free(&os->ix2);
37     Decimatorx2_Free(&os->dx2);
38 }
```

Listado 18.12: Función de liberación contenida en el archivo *OakdriveSimulation.c*.

siempre debe ser llamado.

En el listado 18.13 se muestran las funciones de procesamiento. La función *OakdriveSimulation_ProcessBuffer256* procesa un array de 256 muestras. Como puede verse, empieza declarando un array de 512 muestras en el que se almacena el resultado de la interpolación. Este array se usa posteriormente para realizar el procesamiento muestra a muestra, guardando el resultado del procesamiento de cada muestra en la misma posición del array. Finalmente, en la linea 35 del listado, la dirección del array es pasada la función de diezmado junto con el puntero *dx*2 a la estructura que contiene los datos necesarios. El resultado final será almacenado en el mismo array de entrada.

La función *OakdriveSimulation_ProcessBufferSample* procesa muestras una a una sin realizar interpolación y se ha implementado para realizar pruebas. No tiene ninguna utilidad para el procesamiento.

18.5 Vector de efectos

Para poder añadir el simulador al vector de efectos deben realizarse algunas modificaciones.

En el listado 18.14 se muestra el archivo de cabecera. Como puede verse, la única modificación realizada es que se añaden macros para el tipo de efecto oakdrive y para cada uno

```
void OakdriveSimulation_ProcessBuffer256(OakdriveSimulation* os, float* input){
    int i;
    float interpolated_a[512];
    float* interpolated = interpolated_a;
    Interpolatorx2_Interpolatex2_256(&os->ix2,input, interpolated_a);
    #pragma MUST_ITERATE(512,512,)
    for(i = 0; i<512; i++){
        //Input stage
        ((IdealVoltageSource*)os->vin->derived)->mV = *interpolated;
        os->vin->setIncidentWave(os->vin,os->P1->getReflectedWave(os->P1));
        os->P1->setIncidentWave(os->P1,os->vin->getReflectedWave(os->vin));

        //Voice filter
        ((IdealVoltageSource*)os->opampvin->derived)->mV = os->bias->getV(os->bias);
        os->opampvin->setIncidentWave(os->opampvin,os->S2->getReflectedWave(os->S2));
        os->S2->setIncidentWave(os->S2,os->opampvin->getReflectedWave(os->opampvin));

        //Clipper
        ((ResistiveCurrentSource*)os->clippercurrent->derived)->mI = os->voicebias->getC(os->voicebias);
        os->mclipper->setIncidentWave(os->mclipper, os->P2->getReflectedWave(os->P2));
        os->P2->setIncidentWave(os->P2,os->mclipper->getReflectedWave(os->mclipper));

        //Low pass
        ((IdealVoltageSource*)os->vclippersource->derived)->mV = os->mclipper->getV(os->mclipper) + os->
            bias->getV(os->bias);
        os->vclippersource->setIncidentWave(os->vclippersource, os->S3->getReflectedWave(os->S3));
        os->S3->setIncidentWave(os->S3,os->vclippersource->getReflectedWave(os->vclippersource));

        //High pass
        ((IdealVoltageSource*)os->vlp->derived)->mV = os->C4->getV(os->C4);
        os->vlp->setIncidentWave(os->vlp,os->S5->getReflectedWave(os->S5));
        os->S5->setIncidentWave(os->S5,os->vlp->getReflectedWave(os->vlp));

        *interpolated++ = -os->volumeresistor->getV(os->volumeresistor);
    }
    Decimatorx2_Decimatex2_256(&os->dx2,interpolated_a, input);
}

float OakdriveSimulation_ProcessBufferSample(OakdriveSimulation* os, float input){
    //Input stage
    ((IdealVoltageSource*)os->vin->derived)->mV = input;
    os->vin->setIncidentWave(os->vin,os->P1->getReflectedWave(os->P1));
    os->P1->setIncidentWave(os->P1,os->vin->getReflectedWave(os->vin));

    //Voice filter
    ((IdealVoltageSource*)os->opampvin->derived)->mV = os->bias->getV(os->bias);
    os->opampvin->setIncidentWave(os->opampvin,os->S2->getReflectedWave(os->S2));
    os->S2->setIncidentWave(os->S2,os->opampvin->getReflectedWave(os->opampvin));

    //Clipper
    ((ResistiveCurrentSource*)os->clippercurrent->derived)->mI = os->voicebias->getC(os->voicebias);
    os->mclipper->setIncidentWave(os->mclipper, os->P2->getReflectedWave(os->P2));
    os->P2->setIncidentWave(os->P2,os->mclipper->getReflectedWave(os->mclipper));

    //Low pass
    ((IdealVoltageSource*)os->vclippersource->derived)->mV = os->mclipper->getV(os->mclipper) + os->bias->
        getV(os->bias);
    os->vclippersource->setIncidentWave(os->vclippersource, os->S3->getReflectedWave(os->S3));
    os->S3->setIncidentWave(os->S3,os->vclippersource->getReflectedWave(os->vclippersource));

    //High pass
    ((IdealVoltageSource*)os->vlp->derived)->mV = os->C4->getV(os->C4);
    os->vlp->setIncidentWave(os->vlp,os->S5->getReflectedWave(os->S5));
    os->S5->setIncidentWave(os->S5,os->vlp->getReflectedWave(os->vlp));

    float aux = -os->volumeresistor->getV(os->volumeresistor);
    return aux;
}
```

Listado 18.13: Funciones de procesamiento contenidas en el archivo *OakdriveSimulation.c*.

```
#ifndef EFFECTVECTOR_H_
#define EFFECTVECTOR_H_

#include <ti/csl/tistdtypes.h>
#include <stdlib.h>
#include "OakdriveSimulation.h"

#define SAMPLE_RATE 48000

#define E_VECTOR_SIZE 5

#define MODULATED_DELAY_TYPE 1
#define MODULATED_DELAY_PARAM_FREQ 1
#define MODULATED_DELAY_PARAM_DEPTH 2
#define MODULATED_DELAY_CH 294

#define OAKDRIVE_TYPE 2
#define OAKDRIVE_PARAM_VOICE 1
#define OAKDRIVE_PARAM_GAIN 2
#define OAKDRIVE_PARAM_TONE 3
#define OAKDRIVE_PARAM_VOLUME 4

typedef struct effect {
    UInt16 type;
    Ptr* data_object;
} effect_t;

void AddToEffectVector(Uint16 index, UInt16 type);
void DeleteFromEffectVector(Uint16 index);
void ChangeEffectParamF(Uint16 index, Uint16 param_number, float newValue);
void ProcessBuffer256(float input[]);

#endif /* EFFECTVECTOR_H_ */
```

Listado 18.14: Contenido del archivo *EffectVector.h*.

de sus parámetros.

En el listado 18.15 se muestra el archivo de implementación. Las modificaciones realizadas son triviales. Si existe cualquier duda puede tomarse como referencia la explicación sobre el funcionamiento del código dada en la sección 11.1. En este caso se ha omitido implementar funciones para cambiar los parámetros del efecto, ya que la modificación de parámetros no se contempla en este trabajo.

18.6 Conclusiones

Con el código presentado en este capítulo se consideran concluidos los trabajos de codificación del efecto de saturación. El código compilado puede cargarse directamente en la memoria del procesador mediante la sonda de depuración con la ayuda del IDE de Texas Instruments. Al comenzar una sesión de depuración el depurador situará el contador de programa en el punto de entrada, comenzando la ejecución. En el siguiente capítulo se explican las pruebas realizadas.

```
#include "EffectVector.h"

effect_t effect_array[E_VECTOR_SIZE];

void AddToEffectVector(Uint16 index, UInt16 type){
    effect_array[index].type = type;
    switch(type){
    case MODULATED_DELAY_TYPE:
        //MODULATED DELAY
        break;
    case OAKDRIVE_TYPE:
        effect_array[index].data_object = malloc(sizeof(OakdriveSimulation));
        OakdriveSimulation_Init((OakdriveSimulation*)effect_array[index].data_object, SAMPLE_RATE);
        break;
    default:
        break;
    }
}

void DeleteFromEffectVector(Uint16 index){
   switch(effect_array[index].type){
   case MODULATED_DELAY_TYPE:
       //MODULATED DELAY
       break;
   case OAKDRIVE_TYPE:
      OakdriveSimulation_Free((OakdriveSimulation*)effect_array[index].data_object);
      break;
   default:
       break;
   }
   effect_array[index].type = 0;
   free(effect_array[index].data_object);
}

void ChangeEffectParamF(Uint16 index, Uint16 param_number, float newValue){
   switch(effect_array[index].type){
   case MODULATED_DELAY_TYPE:
       switch(param_number){
       case MODULATED_DELAY_PARAM_FREQ:
           //MODULATED DELAY CHANGE FREQ
           break;
       case MODULATED_DELAY_PARAM_DEPTH:
           //MODULATED DELAY CHANGE DEPTH
           break;
       //TODO: Oakdrive params
       default:
           break;
       }
       break;
   default:
       break;
   }
}

void ProcessBuffer256(float input[]){
   UInt16 i;
   for(i=0;i<E_VECTOR_SIZE;i++){
       switch(effect_array[i].type){
       case MODULATED_DELAY_TYPE:
          //MODULATED DELAY PROCESSING
          break;
       case OAKDRIVE_TYPE:
          OakdriveSimulation_ProcessBuffer256((OakdriveSimulation*)effect_array[i].data_object,input);
          break;
       default:
          break;
       }
   }
}
```

Listado 18.15: Contenido del archivo *EffectVector.c*.

CAPÍTULO 19

Pruebas del efecto oakdrive en el C674x

Este capítulo pertecene al proceso de pruebas del efecto zendrive. En este caso se analizará el software final ejecutándose en el núcleo C674x del procesador OMAP-L138.

19.1 Análisis del espectro de frecuencias

Dada la naturaleza del efecto, el principal análisis en este caso consiste en una visualización del espectro de frecuencias obtenida con la misma configuración que en la sección 12.1.

Como puede verse en la figura 19.1, el ruido impide visualizar en este caso la distorsión de aliasing. Lo importante es que los resultados del procesamiento usando la placa de desarrollo coinciden en la forma de las componentes armónicas de la señal, con una intensidad ligeramente mayor en los armónicos de orden impar.

19.2 Tiempo de procesamiento

En la figura 19.2 se muestra la forma de onda obtenida con el osciloscopio al medir la señal de voltaje en el LED usado para obtener el tiempo de procesamiento. Como puede verse, el periodo de la señal coincide con la latencia calculada en la sección 10.10.3. Cada flanco de subida se corresponde con una transferencia EDMA3 completada, es decir, un buffer listo para su procesamiento. Cada flanco de bajada se corresponde con la finalización del procesamiento de todas las muestras de un buffer.

Considero que el nivel de abstracción usado en la implementación ha tenido su repercusión en el tiempo de procesamiento, ya que este abarca alrededor de 3/4 del tiempo disponible. En cualquier caso, recordemos que el DSP está funcionando con una frecuencia de 300 MHz y que es posible aumentarla hasta 456 Mhz.

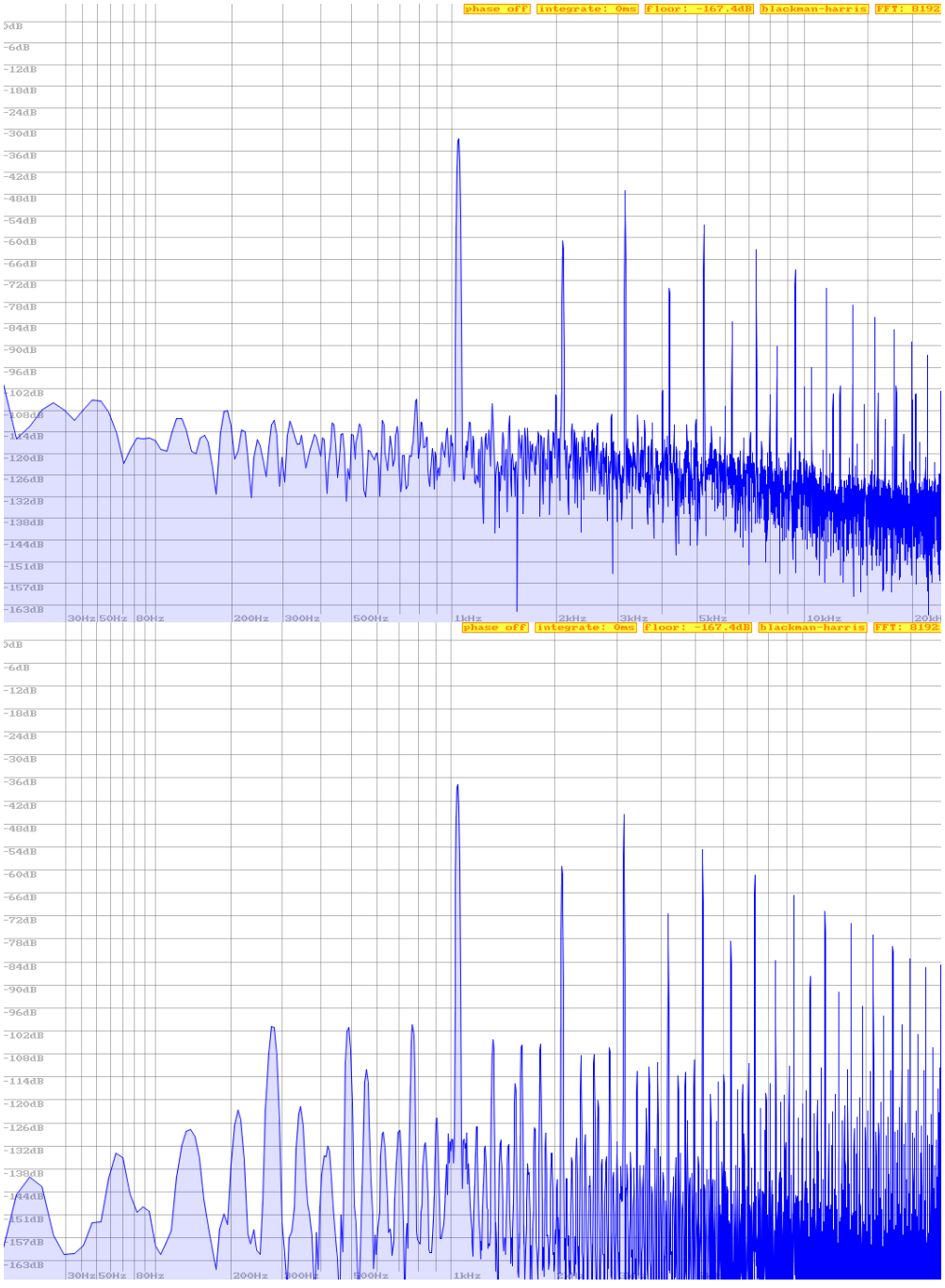


Figura 19.1: Comparación entre una señal de 1046.502 Hz procesada mediante la placa de desarrollo (arriba) y mediante el plugin oakdrive (abajo).

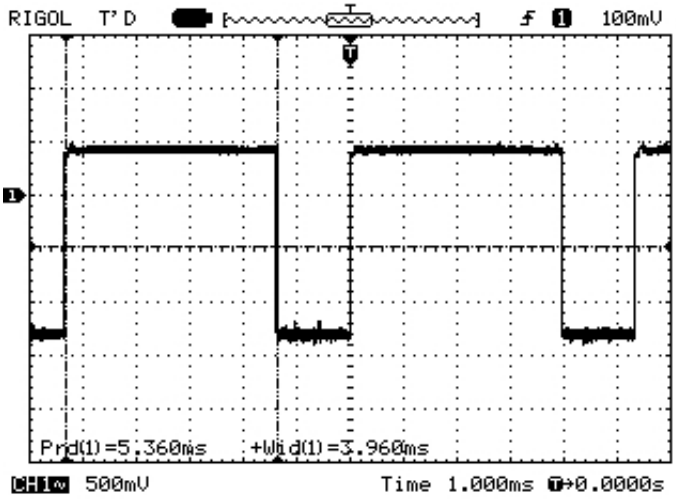


Figura 19.2: Tiempo de procesamiento (anchura de pulso) y latencia de buffers (periodo) en el software implementado.

Por otra parte, un hardware con este procesador no tiene que ser necesariamente un multi-efectos, podría centrarse en un efecto individual, en cuyo caso el tiempo de procesamiento sería aceptable.

19.3 Conclusiones

El resultado del software implementado es aceptable, aunque el tiempo invertido en el procesamiento sería excesivo si tuvieran que ejecutarse más procesamientos en cadena. Sería interesante en futuros trabajos realizar una implementación más eficiente y comparar los resultados.

CAPÍTULO 20

Planificación y costes del proyecto

En este capítulo se describen las tareas realizadas en el proyecto, así como su duración, los retrasos respecto a la planificación original y sus motivos. En el último apartado se hace un análisis aproximado de los costes del proyecto.

20.1 Planificación y tareas

A continuación se detallan las diferentes tareas realizadas en el proyecto. Las tareas se denotan como TRi, donde R es una letra que define un subconjunto de tareas del trabajo e $i = 1, ..., n$, donde n es el número de tareas del subconjunto.

El trabajo completo se divide en tres subconjuntos de tareas. El primero abarca todas las tareas relacionadas con el retardo modulado. El segundo abarca todas las tareas relacionadas con el efecto de distorsión. El tercero abarca todas las tareas independientes. Para el primero de ellos, $R = M$. Para el segundo, $R = D$. Para el tercero, $R = I$. Las diferentes tareas se describen a continuación:

Tarea TM1. Estudio teórico de los retardos modulados, sus métodos de diseño y las ventajas e inconvenientes de cada uno.

Tarea TM2. Diseño de un retardo modulado a partir de la investigación realizada, análisis de los resultados obtenidos en el dominio de la frecuencia.

Tarea TM3. Implementación y pruebas de un plugin VST a partir del diseño realizado. Evaluación de resultados.

Tarea TM4. Implementación y pruebas de un programa ejecutable en el núcleo C674x a partir del diseño realizado. Evaluacion de resultados.

Tarea TD1. Análisis del circuito a simular y estudio teórico de los métodos de simulación de circuitos: posibilidades, ventajas e inconvenientes.

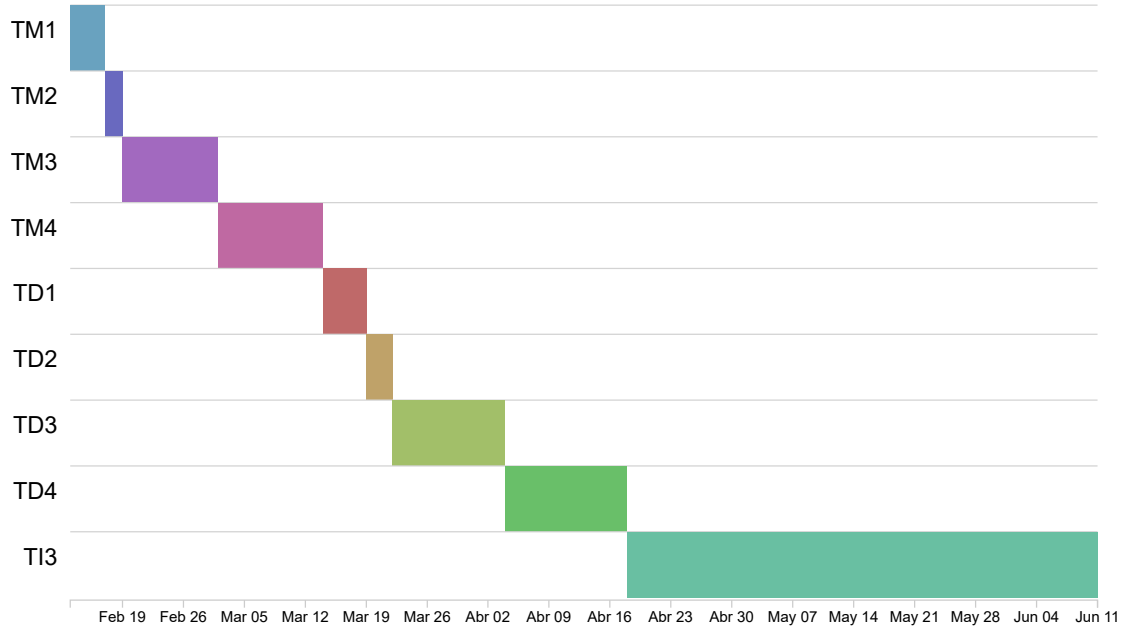


Figura 20.1: Planificación de tareas inicial.

Tarea TD2. Diseño de un simulador del circuito a partir de la investigación realizada.

Tarea TD3. Implementación y pruebas de un plugin VST a partir del diseño realizado. Evaluación de resultados.

Tarea TD4. Implementación y pruebas de un programa ejecutable en el núcleo C674x a partir del diseño realizado. Evaluacion de resultados.

Tarea TI1. Estudio de la documentación del procesador OMAP-L138. Búsqueda de alternativas para alcanzar los objetivos definidos.

Tarea TI2. Implementación de un sistema de controladores que permita realizar los procesamientos digitales diseñados.

Tarea TI2. Escritura de la memoria

Dado el fracaso absoluto de la planificación inicial del proyecto, presento a continuación dos diagamas de Gantt diferentes con una escala temporal distinta en el eje horizontal. En el de la figura 20.1 se muestra la planificación inicial del proyecto. En el de la figura 20.2 se muestra el tiempo real invertido en cada tarea.

Nótese en primer lugar que en la figura 20.1 no aparecen las tareas $TI2$ y $TI2$. Al realizar la planificación al principio del proyecto no tenía claro cual iba a ser el proceso a seguir para implementar los programas en la placa de desarollo, por lo que no tomé en consideración

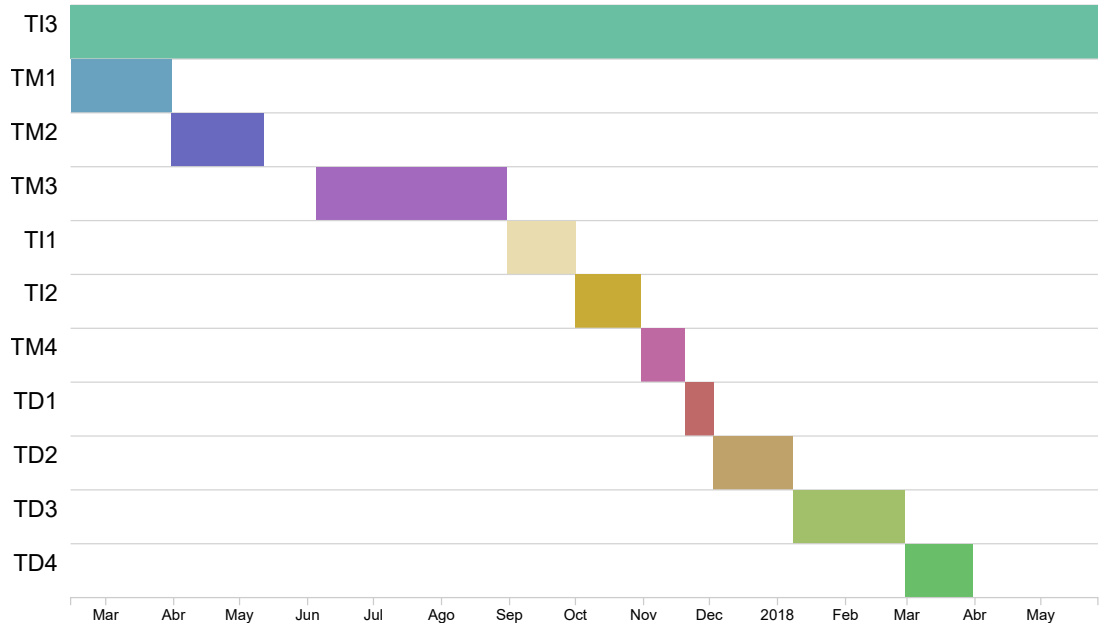


Figura 20.2: Duración real de las tareas.

estas dos tareas. Cuando llego el momento de afrontarlo, me di cuenta de que había varias alternativas pero ninguna de ellas era una tarea trivial.

Nótese en segundo lugar que las tareas $TM1$ y $TM2$ se alargan mucho en el gráfico de la figura 20.2 en comparación con el de la figura 20.1. En un principio tenía pensado usar una simple interpolación lineal para diseñar el retardo modulado. A medida que investigué sobre el tema, me di cuenta de que había alternativas mucho mejores si quería conseguir un buen resultado. Fue entonces cuando decidí intentar usar un retardo fraccionario para realizar el diseño.

En general, subestimé enormemente la duración de las tareas y sobreestimé la cantidad de tiempo que podía dedicarles. Es por eso que todas ellas acabaron alargándose en el tiempo. Esto se nota también, en menor medida, en el diseño e implementación de la simulación del circuito.

Finalmente, nótese que la escritura de la memoria se alarga durante toda la duración del proyecto. En todas las etapas se fue añadiendo información y revisando lo que ya estaba escrito.

20.2 Costes del proyecto

En la figura 20.3 se muestran los costes materiales del proyecto. Los únicos elementos de los que no disponía antes de realizar el proyecto eran la placa de desarrollo TMDSLCDK138 y

Elemento	Coste	Comentarios
Reaper 5.35	51.5 €	Una licencia no comercial fue suficiente para la realización del proyecto.
JUCE	0 €	Una licencia educativa fue suficiente para la realización del proyecto.
Visual Studio 2017	0 €	La licencia fue obtenida gracias al programa Microsoft Imagine.
Code Composer Studio	0 €	La licencia gratuita es suficiente para este trabajo.
LTSpiceXVII	0 €	Se trata de software gratuito.
Scilab	0 €	Se trata de software gratuito bajo licencia GNU.
Texmaker	0 €	Se trata de software gratuito.
Inkscape	0 €	Se trata de software gratuito bajo licencia GPL.
Sublime Text 3	68.6 €	
Rigol DS1052E	300 €	
UltraLite-mk3-Hybrid	555 €	
TMDSLCDK138	168 €	
XDS100V2	85 €	
TOTAL	1228 €	

Figura 20.3: Tabla de costes.

la sonda de depuración XDS100V2. Aún así, he incluido en los costes todos los elementos utilizados.

Por otro lado, aunque es complicado hacer una estimación precisa, calculo que he invertido cerca de 500 horas en este proyecto. No obstante, aproximadamente la mitad de estas horas fueron invertidas en tareas de investigación y estudio de los métodos elegidos. Suponiendo un sueldo de 15 €/*hora* el trabajo ha requerido 7500 €.

20.3 Conclusiones

El fracaso de la primera planificación fue debido sobre todo a la falta de un análisis en profundidad de las tareas a realizar. Este factor influyó sobre todo en los retrasos de las tareas relacionadas con la placa de desarrollo. Por otro lado, la investigación de alternativas también influyó de forma importante en los retrasos. No se puede tampoco despreciar el coste en tiempo de la escritura de la memoria, ya que la presentación de los resultados y la escritura de las revisiones bibliográficas han requerido un considerable esfuerzo de síntesis y constantes revisiones de lo ya escrito.

CAPÍTULO 21

Conclusiones y futuros trabajos

En este capítulo expongo las conclusiones que pueden extraerse del trabajo y algunos de los trabajos futuros que podrían realizarse.

21.1 Conclusiones

En la primera parte del trabajo se ha usado un retardo fraccionario integrado en una estructura propia para producir una modulación continua de las frecuencias de entrada. La estructura consiste en un buffer circular a lo largo del cual se desplaza el retardo fraccionario. Este método ha permitido un diseño en el dominio de la frecuencia, obteniendo resultados muy precisos. La limitación del desplazamiento por muestra del retardo fraccionario sobre el buffer circular ha permitido hacer computable la estructura con un número de operaciones mínimo. Para evitar la aparición de aliasing tras el procesamiento se han diseñado un interpolador y un diezmador y ambos se han implementado en forma de estructura polifásica.

A lo largo de esta parte del trabajo se ha usado Scilab para realizar los diseños de los sistemas y obtener sus respuestas en frecuencia. Además, se han presentado revisiones de los métodos utilizados en cada diseño. Se ha incluido también un capítulo completo de introducción al tratamiento digital de señales.

Para realizar la implementación en forma de plugin de audio se han usado Visual Studio y la librería JUCE. Para implementar una versión en forma de software ejecutable en el núcleo C674x se ha usado Code Composer Studio.

Se ha obtenido como resultado de esta primera parte un plugin de audio que aplica el efecto a una señal de entrada cuando funciona como inserción en un DAW compatible con el estándar VST de Steinberg. Se ha obtenido también software ejecutable sobre el núcleo C674x del procesador OMAP-L138. Este software puede procesar los datos recibidos desde el codec integrado en la placa de desarrollo del procesador y enviar el resultado

del procesamiento otra vez al codec para ser reproducido por un equipo amplificador en tiempo real.

Se han realizado pruebas de ambas versiones del software, encontrándolas satisfactorias en cuanto a resultados del procesamiento y capacidades de funcionamiento en tiempo real.

En la segunda parte se han usado filtros digitales de ondas para simular un circuito de saturación. Se ha analizado la funcionalidad del circuito y posteriormente se ha construido un modelo simplificado, haciendo suposiciones de ganancia infinita para el amplificador operacional. Posteriormente se ha convertido este modelo en filtros digitales de ondas y se ha procedido a su implementación. Para simular la parte más complicada del circuito se ha usado el método de interpolación de Fritsch-Carlson para aproximar curvas corriente-voltaje obtenidas mediante simulación en SPICE.

A lo largo de esta segunda parte del trabajo se han usado Scilab y LTSpice como programas de apoyo para analizar y simular el circuito.

Los elementos utilizados para la implementación, los resultados obtenidos y las pruebas realizadas en esta segunda parte son análogos a los de la primera parte.

Además de lo expuesto, se han incluido en el trabajo un capítulo de revisión bibliográfica con explicaciones sobre el núcleo C674x y los módulos del procesador OMAP-L138 y otro en el que se implementan controladores para hacer posible la configuración de los módulos y el procesamiento sin la necesidad de un sistema operativo.

El diseño de un retardo modulado con el método elegido ofrece una respuesta en frecuencia muy precisa a costa de aumentar en gran medida el coste de procesamiento en comparación con otros métodos de interpolación.

El uso de filtros digitales de ondas para la simulación de efectos de audio parece prometedor, sobre todo trás las últimas propuestas para simular circuitos con múltiples elementos no lineales [59].

21.2 Futuros trabajos

21.2.1 Interfaz física para cambio de parámetros

Mi intención es desarrollar una interfaz física que permita al usuario interactuar con los plugins mediante potenciómetros, botones y una pantalla LCD.

Para ello, el primer paso es implementar un sistema de comunicación con el núcleo ARM9. A continuación se expone una visión general de esta idea.

El núcleo ARM9 recibirá los eventos de la interfaz física mediante algún módulo adecuado del procesador, posiblemente el módulo GPIO. También controlará la pantalla LCD. Cuando reciba una evento de la interfaz escribirá un mensaje para el núcleo C674x en la memoria compartida situada en el interior del chip y generará una interrupción para el DSP. El DSP recibirá la interrupción, señalará la existencia de un mensaje y enmascarará futuras interrupciones del núcleo ARM9 hasta el procesamiento del siguiente buffer. Sin embargo, el núcleo ARM9 podrá seguir ampliando o actualizando el mensaje con las acciones del usuario. Al comienzo del procesamiento de cada bloque el núcleo C674x comprobará si tiene mensajes y de ser así generará una interrupción para el núcleo ARM9 que le indicará a este que no escriba en la región de memoria compartida hasta que reciba una nueva interrupción. Las acciones del usuario durante este periodo de tiempo serán almacenadas en la memoria interna del núcleo ARM9 para enviarlas a la memoria compartida cuando sea posible. Posteriormente, el núcleo C674x comprobará la zona de mensajes y actualizará el vector de efectos de acuerdo al mensaje existente. Finalmente, enviará una interrupción al núcleo ARM9 para señalizar que ya puede escribir de nuevo en la región de memoria compartida.

21.2.2 Conversor A/D y entrada de alta impedancia

Existen conversores A/D para aplicaciones de audio de mayor calidad que el codec incluido en la placa, por ejemplo, el PCM4202 de Texas Instrumens. El uso de un formato de 24 bits en este trabajo no es casual, se ha hecho con la intención de que el puerto McASP pueda recibir en un futuro muestras de este conversor.

Por otro lado, para poder conectar al circuito una guitarra eléctrica sería necesario diseñar una etapa de entrada de alta impedancia, dado que las pastillas de guitarra tienen una impedancia de salida de alrededor de 10K y producen señales muy débiles. Para ello, en la hoja de datos del conversor A/D mencionado [66] aparece el esquema de un circuito de entrada usando el amplificador diferencial OPA1632. Sería interesante evaluar si resulta adecuado para la aplicación.

21.2.3 Diseño de una placa propia

Sería interesante diseñar una placa propia que incluyera las mejoras mencionadas en las anteriores secciones. Aunque carezco de conocimientos para diseño de PCBs a las velocidades a las que funcionan algunas partes del circuito, el hecho de que los esquemas de la placa sean distribuidos de forma gratuita seguramente aliviaría la carga de trabajo.

21.2.4 Mejora de los efectos

Los efectos presentados en este trabajo pueden simplificarse. Sobre todo en el caso del retardo modulado, es posible usar otros métodos de interpolación menos costosos y aún así obtener resultados aceptables. Para encontrar un balance, sería interesante realizar una investigación en profundidad sobre métodos de interpolación.

Por otro lado, sería interesante implementar un chorus, un flanger y otros efectos de modulación haciendo uso del retardo modulado diseñado.

En el caso de la saturación, sería interesante usar un modelo no ideal del amplificador operacional y un algoritmo para el control de la distorsión de aliasing como el presentado en [65].

21.2.5 Desarrollo de nuevos efectos

En el futuro podrían añadirse más efectos al software, como por ejemplo un amplificador de tubos de vacío. La válvula o tubo de vacío sigue siendo a día de hoy el dispositivo de preferencia para amplificadores de instrumentos musicales y sonido profesional, ya que la linealidad y mejor rendimiento teórico de los transistores da como resultado en circuitos de audio sonidos muy fríos y con poco carácter [67, pp. 7]. La simulación digital de amplificadores de tubos de vacío tiene gran utilidad práctica, ya que permite aproximar las características sonoras de estos aparatos haciendo uso de menos espacio y energía; ofreciendo mayor durabilidad y fiabilidad; y reduciendo considerablemente los costes materiales.

Referencias

[1] J. G. Proakis y D. G. Manolakis, *Tratamiento digital de señales*, 4.ª ed. Pearson Educación, S.A., 2007 (vid. págs. 27, 49, 51, 52, 54, 55, 66, 85).

[2] U. Zolzer *et al.*, *DAFX - Digital Audio Effects*, 1.ª ed. John Wiley and Sons, Ltd, 2002 (vid. págs. 27, 29, 85, 87).

[3] I.Sinclair *et al.*, *Audio Engineering*, 1.ª ed. Newnes, 2009 (vid. pág. 28).

[4] M. A. P. García, *Instrumentación electrónica*, 1.ª ed. Ediciones Paraninfo, S.A., 2014 (vid. pág. 28).

[5] J. D.Reiss y A. P. McPherson, *Audio effects, Theory, Implementation and Application*, 1.ª ed. CRC Press, 2015 (vid. págs. 29, 31, 43, 87).

[6] D. T. Yeh, "Digital Implementation of Musical Distortion Circuits by Analysis and Simulation", Ph.D. dissertation, Department Of Electrical Engineering, Stanford University, 2009 (vid. pág. 30).

[7] K. Huang. (2007). King of Tone Guitar Pedal Modeling With Nodal Analysis and Table Preconstruction Method, [Online]. Dirección: `https://ccrma.stanford.edu/~kaichieh/KingOfTone.pdf` (vid. pág. 30).

[8] J. Pakarinen y D. T. Yeh, "A Review of Digital Techniques for Modeling Vacuum-Tube Guitar Amplifiers)", *Computer Music Journal*, vol. 33, n.º 2, págs. 85-100, 2009 (vid. pág. 30).

[9] E. Hamidovic, *The systematic mixing guide*, 1.ª ed. Systematic Productions, 2012 (vid. pág. 31).

[10] M. Shafaati y H. Mojallali, "Modified Firefly Optimization for IIR System Identification", *Journal of Control Engineering and Applied Informatics*, vol. 14, n.º 4, págs. 59-69, 2012 (vid. pág. 34).

[11] T. B. Welch, C. H. G. Wright y M. G. Morrow, *Real-Time Digital Signal Processing from MATLAB to C with the TMS320C6x DSPs*, 3.ª ed. CRC Press, 2016 (vid. págs. 36, 175).

[12] S. G. Palomo y E. M. Gil, *Aproximación a la ingeniería de software*, 1.ª ed. Editorial Centro de estudios Ramón Areces S.A., 2014 (vid. pág. 39).

[13] R. S. Pressman, *Ingeniería del software, un enfoque práctico*, 7.ª ed. Mc.Graw-Hill, 2010 (vid. pág. 39).

[14] Reaper. (2017). Página web de Reaper, [Online]. Dirección: `http://www.reaper.fm/` (vid. pág. 42).

[15] Motu. (2017). Página web de Ultralite-mk3 Hybrid, [Online]. Dirección: http://motu.com/products/motuaudio/ultralite-mk3 (vid. pág. 44).

[16] TI. (2017). Página del procesador OMAP-L138, [Online]. Dirección: http://www.ti.com/product/OMAP-L138 (vid. pág. 45).

[17] S. D. Canto, J. S. Moreno y V. S. Prat, *Ingeniería de computadores II*, 1.ª ed. Sanz y Torres, 2011 (vid. págs. 45, 46).

[18] ARM. (2001). ARM Architecture Reference Manual, [Online]. Dirección: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/third-party/ddi0100e_arm_arm.pdf (vid. pág. 45).

[19] TI. (2017). TMS320C674x DSP CPU and Instruction Set Reference Guide, [Online]. Dirección: http://www.ti.com/lit/ug/sprufe8b/sprufe8b.pdf (vid. págs. 45, 163).

[20] TI. (2017). OMAP-L138 DSP ARM9 Development Kit, [Online]. Dirección: http://www.ti.com/lit/ml/sprt634/sprt634.pdf (vid. pág. 46).

[21] S. C. D. Roy. (2008). Lecture - 7 FIR and IIR Recursive and Non Recursive, Department of Electrical Engineering, IIT Delhi, [Online]. Dirección: https://www.youtube.com/watch?v=GpqMAzGEXXk#t=21m00s (vid. pág. 54).

[22] L. R. Marín, *Temas de matemáticas*, 1.ª ed. Sanz y torres, 2010 (vid. págs. 61, 91, 95).

[23] B. R. Kusse y E. A. Westwig, *Mathematical Physics*, 2.ª ed. Wiley-Vch, 2006 (vid. pág. 63).

[24] B. Osgood. (2007). Lecture notes for the Fourier transform and its Applications, [Online]. Dirección: https://see.stanford.edu/materials/lsoftaee261/book-fall-07.pdf (vid. pág. 65).

[25] J. O. Smith, *Spectral audio signal processing*, 1.ª ed. Stanford University, Center for Computer Research in Music and Acoustics (vid. pág. 65).

[26] (2017). Convolution theorem, Society of Exploration Geophysicists, [Online]. Dirección: http://wiki.seg.org/wiki/Dictionary:Convolution_theorem (vid. pág. 65).

[27] L.Wanhammar, *DSP Integrated Circuits*, 1.ª ed. Academic Press, 1999 (vid. pág. 67).

[28] T. Saramäki, "Finite Impulse Response Filter Design", en *Handbook for Digital Signal Processing*. John Wiley and Sons, Ltd, 1993 (vid. pág. 67).

[29] R. E. Crochiere y L. R. Rabiner, *Multirate Digital Signal Processing*, 1.ª ed. Prentice Hall, 1983 (vid. págs. 70, 73).

[30] R. J. Pumphrey, "Upper Limit of Frequency for Human Hearing", *Nature*, vol. 166, n.º 4222, pág. 571, 1950 (vid. pág. 80).

[31] J. Dattorro, "Effect Design Part 2: Delay-Line Modulation and Chorus", *Journal of the Audio Engineering Society*, vol. 45, n.º 10, págs. 764-788, 1997 (vid. págs. 88, 107).

[32] J. Diaz-Carmona y G. J. Dolecek, "Fractional Delay Filters", en *Applications of MATLAB in Science and Engineering*. InTech, 2011 (vid. pág. 89).

[33] V. Välimäki, "Discrete-Time Modeling of Acoustic Tubes Using Fractional Delay Filters", Dissertation for the degree of Doctor of Technology, Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology, 1995 (vid. pág. 89).

[34] S. Pei y C. Tseng, "An Efficient Design of a Variable Fracional Delay Filter Using a First-Order Differentiator", *IEEE Signal Processing Letters*, vol. 10, n.º 10, págs. 307-310, 2003 (vid. págs. 89, 91-93).

[35] G. Mollova, "Compact formulas for least-squares design of digital differentiators", *Electronic Letters*, vol. 35, n.º 20, págs. 1695-1697, sep. de 1999 (vid. págs. 90, 93, 94, 97, 99).

[36] E. J. Barbeau, *Polynomials*, 1.ª ed. Springer, 1989 (vid. pág. 93).

[37] H. Flanders, "Differentiation Under the Integral Sign", *The American Mathematical Monthly*, vol. 80, n.º 6, págs. 615-627, 1973, ISSN: 00029890, 19300972. [Online]. Dirección: `http://www.jstor.org/stable/2319163` (vid. pág. 95).

[38] J. Rincón. (2004-07). Derivación bajo la integral, Department of Applied Mathematics, University of Granada, [Online]. Dirección: `http://canizo.org/tex/dbi.pdf` (vid. pág. 95).

[39] Scilab. (2017-04). Scilab Manual, [Online]. Dirección: `https://www.scilab.org/product/man/` (vid. pág. 99).

[40] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, 3.ª ed. Center for Computer Research in Music and Acoustics, Stanford University, 2007 (vid. pág. 106).

[41] T. Saramäki y T. Ramstad. (2010-10). Design of Efficient FIR filters based on multirate and complementary filtering, [Online]. Dirección: `https://www.cs.tut.fi/~ts/Part4_short_article.pdf` (vid. pág. 116).

[42] JUCE. (2018-02). Tutorial: The AudioProcessorValueTreeState class, [Online]. Dirección: `https://juce.com/doc/tutorial_audio_processor_value_tree_state` (vid. pág. 120).

[43] Microsoft. (2018-03). Expresiones lambda en C++, [Online]. Dirección: https://msdn.microsoft.com/es-es/library/dd293608.aspx (vid. pág. 151).

[44] JUCE. (2018-03). Tutorial: The Slider class, [Online]. Dirección: https://docs.juce.com/master/tutorial_slider_values.html (vid. pág. 154).

[45] TI. (2011-03). OMAPL138 LCDK board schematics, BOM, layout and Gerber files, [Online]. Dirección: http://www.ti.com/lit/zip/sprcaf3 (vid. pág. 161).

[46] TI. (2009-09). TMS320C674x DSP Cache User's Guide, [Online]. Dirección: http://www.ti.com/lit/ug/sprug82a/sprug82a.pdf (vid. pág. 161).

[47] TI. (2017-01). OMAP-L138 Datasheet, [Online]. Dirección: http://www.ti.com/lit/ds/symlink/omap-l138.pdf (vid. págs. 161, 162, 192).

[48] TI. (2010-06). TMS320C674x CPU and Instruction Set Reference Guide, [Online]. Dirección: http://www.ti.com/lit/ug/sprufe8b/sprufe8b.pdf (vid. págs. 162, 163).

[49] TI. (2010-04). TMS320C6748/46/42 and OMAP-L138 Processor EDMA3 Controller User's Guide, [Online]. Dirección: http://www.ti.com/lit/ug/sprugp9b/sprugp9b.pdf (vid. págs. 164-166, 202).

[50] TI. (2009-08). TMS320C674x/OMAP-L1x Processor McASP User's Guide, [Online]. Dirección: http://www.ti.com/lit/ug/sprufm1/sprufm1.pdf (vid. págs. 167-171, 191).

[51] TI. (2010-06). TMS320C674x/OMAP-L1x GPIO User's Guide, [Online]. Dirección: http://www.ti.com/lit/ug/spruf18b/spruf18b.pdf (vid. pág. 171).

[52] TI. (2016-09). OMAP-L138 Technical Reference Manual, [Online]. Dirección: http://www.ti.com/lit/ug/spruh77c/spruh77c.pdf (vid. págs. 173, 193).

[53] TI. (2014-12). TLV320AIC3106 Datasheet, [Online]. Dirección: http://www.ti.com/lit/ds/symlink/tlv320aic3106.pdf (vid. págs. 173, 182, 184, 186).

[54] TI. (2018-04). Math Library for Floating Point Devices, [Online]. Dirección: http://processors.wiki.ti.com/index.php/Software_libraries#MathLIB (vid. págs. 178, 211).

[55] TI. (2013-09). OMAP-L132/L138 TMS320C6742/6/8 Pin Multiplexing Utility, [Online]. Dirección: http://www.ti.com/litv/zip/sprab63b (vid. pág. 178).

[56] TI. (2011-03). TMS320C674x/OMAP-L1x I2C Module User's Guide, [Online]. Dirección: http://www.ti.com/lit/ug/spruf19d/spruf19d.pdf (vid. págs. 180, 181).

[57] A. Fettweis, “Wave digital filters: Theory and practice”, *Proceedings of the IEEE*, vol. 74, n.º 2, págs. 270-327, feb. de 1986 (vid. pág. 225).

[58] R. E. Carlson y F. N. Fritsch, “Monotone Piecewise Cubic Interpolation”, *SIAM Journal on Numerical Analysis*, vol. 17, n.º 2, págs. 238-246, abr. de 1980 (vid. págs. 225, 239).

[59] K. J. Werner *et al.*, “Resolving Wave Digital Filters With Multiple/Multiport Nonlinearities”, en *Proceedings of the Eighteenth International Conference on Digital Audio Effects (DAFx-15)*, Trondheim, Noruega: P. Svensson y U. Kristiansen, 2015, págs. 387-394 (vid. págs. 239, 318).

[60] N. R. Malik, *Circuitos electrónicos. Análisis, diseño y simulación.* 1.ª ed. Prentice Hall, 1996 (vid. págs. 247, 252).

[61] C. Crespo. (2017). MOSFET Transistor: Derivation of MOSFET Equations, [Online]. Dirección: https://www.youtube.com/watch?v=S46mT6ZYD6o (vid. pág. 252).

[62] K. J. Werner *et al.*, “An Improved and Generalized Diode Clipper Model for Wave Digital Filters”, en *Audio Engineering Society Convention 139*, New York, USA, oct. de 2015. [Online]. Dirección: http://www.aes.org/e-lib/browse.cfm?elib=17918 (vid. págs. 260, 277).

[63] T. Fukushima, “Precise and fast computation of Lambert W-functions without transcendental function evaluations”, *Journal of Computational and Applied Mathematics*, vol. 244, págs. 77-89, 2013, ISSN: 0377-0427. DOI: 10.1016/j.cam.2012.11.021 (vid. pág. 260).

[64] P. Fernández-Cid. (2017). ¿Por qué suena peor la distorsión digital que la analógica?, [Online]. Dirección: https://www.hispasonic.com/tutoriales/suena-peor-distorsion-digital-analogica/42993 (vid. pág. 291).

[65] F. Esqueda, V. Välimäki y S. Bilbao, “Aliasing reduction in soft-clipping algorithms”, en *23rd European Signal Processing Conference (EUSIPCO)*, ago. de 2015, págs. 2014-2018. DOI: 10.1109/EUSIPCO.2015.7362737 (vid. págs. 291, 320).

[66] TI. (2013-09). PCM4202 Datasheet, [Online]. Dirección: http://www.ti.com/lit/ds/symlink/pcm4202.pdf (vid. pág. 319).

[67] M. P. Hernández. (2005). Electroacústica, Departamento de electrónica y comunicaciones,Escuela de Ingeniería de Bilbao, [Online]. Dirección: http://aholab.ehu.es/users/imanol/akustika/IkasleLanak/Amplificadores%20de%20audio.pdf (vid. pág. 320).

Siglas, abreviaturas y acrónimos

A/D	Analógico-Digital
D/A	Digital-Analógico
CCS	Code Composer Studio
WDF	Wave Digital Filter
FIR	Finite Impulse Response
IIR	Infinite Impulse Response
FFT	Fast Fourier Transform
VST	Virtual Studio Technology
DAW	Digital Audio Workstation
IDE	Integrated Development Environment
DLL	Dynamic Link Library
LCDK	Low Cost Development Kit
ALU	Arithmetic Logic Unit
VLIW	Very Long Instruction Word
RISC	Reduced Instruction Set Computer
RAM	Random Access Memory
SoC	System on Chip
EDMA3	Enhanced Direct Memory Access
PaRAM	Parameter RAM
McASP	Multichannel Audio Serial Port
TDM	Time Division Multiplexing
PLL	Phase Locked Loop
AGC	Automatic Gain Control
I2C	Inter Integrated Circuit
GPIO	General Purpose Input/Output
DSP	Digital Signal Processor