EFECTOS DE AUDIO EN TIEMPO REAL CON DSPs

AUTOR:

YASSER ALEXANDER MÉNDEZ VILLABONA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICOMECANICAS
ESUELA DE INGENIERÍAS ELÉCTRICA,
ELECTRÓNICA Y TELECOMUNICACIONES
MAESTRÍA EN POTENCIA ELÉCTRICA
BUCARAMANGA
2004

EFECTOS DE AUDIO EN TIEMPO REAL CON DSPs

AUTOR:

YASSER ALEXANDER MÉNDEZ VILLABONA

TESIS DE MAESTRÍA

DIRECTOR:

Ph.D. OSCAR GUALDRÓN

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICOMECÁNICAS
ESUELA DE INGENIERÍAS ELÉCTRICA,
ELECTRÓNICA Y TELECOMUNICACIONES
MAESTRÍA EN POTENCIA ELÉCTRICA
BUCA RAMANGA
2004

CONTENIDO

LISTA DE TABLAS

Pag

LISTA DE FIGURAS

LISTA DE ANEXOS

RESUMEN

TITULO: EFECTOS DE AUDIO EN TIEMPO REAL CON DSPs [*]

AUTOR: YASSER ALEXANDER MÉNDEZ VILLABONA [**]

PALABRAS CLAVE: Efectos, Audio, DSPs, Retardo, *Flanger*, *Chorus,* Compreso, Expansor.

DESCRIPCION

El desarrollo tecnológico de la última década, ha incorporado los DSPs, dispositivos electrónicos constituidos por una arquitectura de acceso múltiple a memoria, capaces de realizar cálculos aritméticos complejos de alta velocidad, transferencia de datos desde y hacia el mundo real. Estos dispositivos junto con las herramientas del procesado de la señal, permiten el desarrollo de múltiples aplicaciones, en diferentes campos de la electrónica, la instrumentación y las telecomunicaciones. Una de las aplicaciones más atractivas y de un carácter bastante comercial, de los DSPs, es la del procesamiento de señales de audio, donde la implementación en tiempo real de algoritmos de efectos de audio, convierten a estos dispositivos en la pieza fundamental y el corazón de procesadores de audio, utilizados por músicos y profesionales de la industria del audio para la creación y mejoramiento de su obras.

Existe una gran cantidad de efectos de audio, unos basados en otros o por mezcla de varios efectos; sin embargo los efectos de audio se pueden agrupar en tres grandes grupos: efectos basados en retardos digitales, efectos de modulación de retardos y efectos basados en amplitud. En este proyecto se desarrolla la implementación de algunos de los algoritmos más representativos de estos efectos como: el retardo simple, el retardo múltiple, el *flanger,* el *chorus*, el compresor y el expansor; realizando una evaluación cualitativa y cuantitativa del funcionamiento de estos algoritmos con la simulación de los mismos en Matlab.

El presente trabajo de investigación sienta las bases para una profundización en el campo del procesamiento de señales de audio mediante el uso de los DSPs. Los resultados obtenidos para los diferentes algoritmos implementados verifican el buen desempeño de los mismos, y sugieren el desarrollo de nuevos algoritmos y el perfeccionamiento de los ya implementados, mediante la incorporación de parámetros adicionales.

---

[*] Tesis de Maestría

[**] Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones, Maestría en Potencia Eléctrica, Director: PhD. Oscar Gualdrón.

SUMMARY

TITLE: EFECTOS DE AUDIO EN TIEMPO REAL CON DSPs [*]

AUTHOR: YASSER ALEXANDER MÉNDEZ VILLABONA [**]

KEY WORDS: Effects, Audio, DSPs, Delay, Flanger, Chorus, Compressor, Expander.

DESCRIPTION:

The technological development of the last decade, has incorporated the DSPs, electronic devices constituted by architecture of multiple access to memory, they are able to carry out high-speed complex arithmetic calculations, transfer of data from and toward the real world. These devices together with the tools of the processed of the sign, they allow the development of multiple applications, in different fields of the electronics, the instrumentation and the telecommunications. One of the most attractive applications and of a quite commercial character, of the DSPs, it is that of the processing of signs of audio, where the implementation in real time of algorithms of effects of audio, they transform to these devices into the fundamental piece and the heart of processors of audio, used by musicians and professionals of the industry of the audio for the creation and improvement of their works.

A great quantity of effects exists of audio, some based in other or for mixture of several effects; however the effects of audio can be classified in three groups: digital delays, delay modulation effects and amplitude-based audio effects. In this project the implementation is developed of some of the most representative algorithms in these effects like: the simple delay, the multi-taps delays, flanger, chorus, compressor and expander; carrying out a qualitative and quantitative evaluation of the operation of these algorithms with the simulation of the same in Matlab.

The present investigation work sits down the bases for a profundización in the field of the processing of signs of audio by means of the use of the DSPs. The results obtained for the different implemented algorithms verify the their good acting, and they already suggest the development of new algorithms and the improvement of those implemented, by means of the incorporation of additional parameters.

---

[*] Tesis de Maestría

[**] Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones. Maestría en Potencia Eléctrica. Director: PhD. Oscar Gualdrón.

INTRODUCCIÓN

El campo de los procesadores de señales (DSPs) ha evolucionado desde hace ya más de una década, convirtiéndolos en piezas fundamentales de equipos electrónicos que van desde aparatos de consumo como equipos de sonido, celulares o cámaras digitales hasta equipos industriales y profesionales como osciloscopios, analizadores de espectros, equipos de instrumentación médica o productos de audio profesional, entre una larga lista de aplicaciones.

Diferentes compañías como *Analog Devices*, *Texas Instruments*, *Motorola*, *Bores* y *Lucent*, están dedicadas a la producción, comercialización y difusión de DSPs, con una amplia documentación, tutoriales, manuales, publicaciones, herramientas de desarrollo, emuladores y software de simulación colocados en la web con acceso gratuito, además de cursos en línea sobre aplicaciones específicas de sus productos y manejo de los mismos. Las ventajas que ofrecen estos DSPs como cálculos aritméticos complejos de alta velocidad, transferencia de datos desde y hacia el mundo real y su arquitectura de acceso múltiple a memoria, junto con las herramientas del procesamiento digital de señales, permiten el desarrollo de múltiples aplicaciones.

Una de las aplicaciones más comerciales se encuentra en el campo del audio digital, concretamente los efectos de audio digital, que en la actualidad hacen parte de la industria del audio profesional tanto en software de edición de audio como el Cool Edit Pro [44] o en el hardware de músicos que utilizan procesadores de efectos de audio digital en tiempo real como el SPX2000 [46], para mejorar e innovar sus obras musicales.

El trabajo con efectos de audio en tiempo real en un DSP permite observar cualitativa y cuantitativamente los cambios hechos a una señal de audio al ser procesada digitalmente por un algoritmo específico. Existe una gran cantidad de efectos de audio, unos basados en otros o por mezcla de varios efectos; sin embargo los efectos de audio se pueden agrupar en tres grandes grupos: efectos basados en retardos digitales, efectos de modulación de retardos y efectos basados en amplitud. En este proyecto se desarrolla la implementación de algunos de los algoritmos más representativos de estos efectos como: el retardo simple, el retardo múltiple, el *flanger,* el *chorus*, el compresor y el expansor; realizando una evaluación cualitativa y cuantitativa del funcionamiento de estos algoritmos con la simulación de los mismos en Matlab. Además la implementación de filtros FIR e IIR facilita la comprensión del acceso múltiple a memoria y la ejecución de múltiples instrucciones en un DSP para desarrollar las multiplicaciones y sumas que requieren estos algoritmos.

Este documento distribuido en seis capítulos y anexos de programación sirve como una introducción al mundo del audio digital aprovechando la teoría del procesamiento digital de señales y las características ofrecidas por los DSPs de la familia de *Analog Devices* de 32 bits en punto flotante.

El primer capítulo denominado el ADSP – 21065L de 32 bits de *Analog Devices* hace un breve explicación de la arquitectura de este procesador y de las herramientas de desarrollo utilizadas para la implementación de los algoritmos.

El segundo capítulo denominado efectos de audio trata de manera sencilla el marco teórico de los efectos de audio aquí implementados, el retardo simple, el retardo múltiple, el *flanger,* el *chorus*, el compresor y el expansor.

El tercer capítulo denominado implementación de los algoritmos en el DSP explica detalladamente cada uno de los algoritmos y su implementación en lenguaje *assembly* en el ADSP-21065L tanto en simulación como en tiempo real, haciendo referencia a las instrucciones del programa que ejecutan las rutinas de estos algoritmos.

El cuarto capítulo denominado evaluación de los algoritmos implementados en el DSP se muestra mediante tablas los valores de relación señal a ruido y error cuadrático medio los resultados de la comparación de los algoritmos implementados con la simulación de los mismos en Matlab; permitiendo realizar una comparación cualitativa y cuantitativa del funcionamiento de estos algoritmos.

El quinto capítulo presenta las conclusiones a las que se llega después de analizar los resultados y la evaluación de los diferentes algoritmos implementados en el DSP.

El sexto capítulo presenta una serie de recomendaciones encaminadas a abarcar una mayor cantidad de algoritmos de efectos de audio e implementar estos algoritmos en lenguaje C.

Finalmente en los anexos se muestran las líneas de código de los programas de los algoritmos de efectos de audio y los filtros FIR e IIR implementados en el DSP tanto en simulación como en tiempo real, con una breve explicación del manejo de las herramientas de desarrollo del ADSP-21065L para ejecutar estos programas.

1. EL ADSP – 21065L DE 32 BITS DE ANALOG DEVICES

El ADSP-21065L SHARC es un procesador digital de señales (DSP) de 32 bits programable, de propósito general, que trabaja tanto en punto flotante como en punto fijo con igual eficiencia; ampliamente utilizado en comunicaciones, audio digital, instrumentación y otras aplicaciones. Realiza 180 MFLOPS/ 180 MOPS (millones de operaciones por segundo) gracias a su reloj de 60 MHz. Con un caché de instrucciones, que permite al procesador ejecutar varias instrucciones en un solo ciclo, y su código es compatible con otros miembros de la familia SHARC. Sus cuatro buses independientes para datos, instrucciones y entradas/salidas (I/O) duales, y un bus interruptor de conexiones a memoria implementan la Arquitectura *Super Hardvard* del ADSP-21065L. Además es el procesador de más bajo costo que tiene en el mercado *Analog Devices.* [3]

Algunos componentes importantes de este DSP son los siguientes:

- Unidades Computacionales: ALU, MAC, *Shifter*
- 16 Registros para datos
- Secuenciador de Programa con caché de instrucciones
- Generadores de Direcciones: DAG1 (32 bit, DM) y DAG2 (24 bits, PM)
- Puerto dual SRAM con 544 kbits configurables
- Puerto externo para interfase RAM y otros chips de memoria y periféricos
- Puerto host e interfase para multiprocesadores
- Controlador DMA para soportar 10 canales
- Puerto serial con dos receptores y dos transmisores que soportan TDM y I^2S[1]
- Dos *Timers* Programables
- 12 Pines I/O propósito general programables

1. 1 DSP PARA AUDIO DIGITAL

El ADSP-21065L contiene las siguientes características para desarrollar cálculos de procesamiento digital de señales en tiempo real y que lo convierten en la mejor opción para el desarrollo de aplicaciones con procesamiento de señales de audio:

- Aritmética flexible y rápida: Cálculos en un solo ciclo para multiplicaciones con acumulación, tanto en punto flotante como en punto fijo, y un set de instrucciones estándar para operaciones aritméticas y lógicas.

- Búsqueda de dos operandos para cálculos de suma de productos en un solo ciclo: En los cálculos de suma de productos, dos operandos son necesarios en cada ciclo para realizar los cálculos. El DSP está habilitado para buscar estos dos datos en la memoria del DSP y realizar la operación de producto y suma en un solo ciclo.

[1] I^2S: Protocolo estándar de un bus serial de tres cables desarrollada por Philips para la transmisión de dos canales estéreo PCM [45].

- Soporta buffer circular en hardware: Una gran cantidad de algoritmos DSP, incluyendo filtros digitales, requieren buffers circulares de datos. ADSP-21065L está diseñado para permitir el direccionamiento automático de punteros que simplifican la implementación del buffer circular reduciendo el *overflow* y mejorando la ejecución.

- Lazos y ramificaciones eficientes para operaciones DSP repetitivas: Los algoritmos DSP son repetitivos y son lógicamente expresados en lazos. El secuenciador de programa del ADSP-21065L permite enlazar código con mínimo o cero *overhead*, incluyendo saltos y llamada a subrutinas.

1.1.1 DSP de propósito general y decodificadores para audio. Existen muchos tópicos los cuales deben ser considerados en el momento de seleccionar un DSP para una aplicación. En cualquier caso aplicaciones de audio con alto volumen y requerimientos de alta fidelidad, los diseñadores miran un número de características de diseño con el costo más bajo. Generalmente, están frecuentemente relacionadas con la velocidad del procesador, flexibilidad, tipo de datos, precisión y memoria.

• **Velocidad del procesador.** La velocidad de un procesador generalmente determina cuantas operaciones puede ejecutar un DSP en una cierta cantidad de tiempo. Hay dos unidades de medida que son típicamente usadas para describir la velocidad de un DSP: Megahertz y millones de instrucciones por segundo (o millones de operaciones por segundo). La velocidad del reloj del procesador es medida en Megahertz. Esta es la rata a la cual el DSP ejecuta operaciones en las unidades básicas de trabajo. Muchos DSPs ejecutan como mínimo una instrucción por ciclo de reloj. Los MIPS o en el caso de los procesadores SHARC de *Analog Devices* MFLOPS representa los millones de operaciones en punto flotante por segundo que puede realizar el DSP. El ADSP-21065L ejecuta 180 MFLOPS.

• **Memoria.** La memoria "*on- chip*" en un DSP es la memoria integrada dentro del DSP la cual es usada para almacenar instrucciones de programa y datos. Muchos algoritmos requieren buffers grandes para almacenar datos en memoria, por lo cual se hace necesario que el DSP provea la facilidad de accionar memoria "*off-chip*" sin reducir el rendimiento en la búsqueda de instrucciones y datos desde esta memoria. El ADSP-21065L tiene 544 kbits configurables.

• **Capacidades I/O e interfaces para procesamiento de muestras de audio.** La comunicación del DSP con el mundo real de manera rápida y eficiente sin afectar el rendimiento del procesador es otra característica importante del procesador. El ADSP-21065L cuenta con un controlador DMA que maneja la transferencia de datos entre el DSP y equipos externos. Además posee una interfase serial que soporta comunicación multicanal en modo TDM para comunicar fácilmente convertidores de audio, como el codec AD1847A SoundPort de Analog Devices que viene incluido en la tarjeta de desarrollo ADSP-21065L EZ_LAB y que permite el fácil manejo de señales de audio para su procesamiento.

1.1.2 Formato de los datos. El formato de los datos usados por el DSP determina la habilidad de este para manejar señales de diferente precisión, rango dinámico y relación señal a ruido. El ADSP-21065L trabaja con punto flotante de 32 bits, haciéndolo inmejorable para su uso en aplicaciones donde se requiere un gran rango dinámico como en el procesamiento de señales de

audio, pues ofrece un rango dinámico de 192 dB. Además este procesador trabaja con igual eficiencia tanto en punto flotante como en punto fijo.

1.1.3 El Codec. El ADSP-21065L permite a través de su interfase serial la comunicación con el codec AD1847 usando el protocolo TDM basado en la especificación AC-97 1.03 [45]. Este codec de 16 bits con entrada y salida estéreo permite trabajar con señales de audio con calidad de CD con una relación señal a ruido SNR de 96 dB y con una frecuencia de muestreo variable entre 8 kHz y 48 kHz.

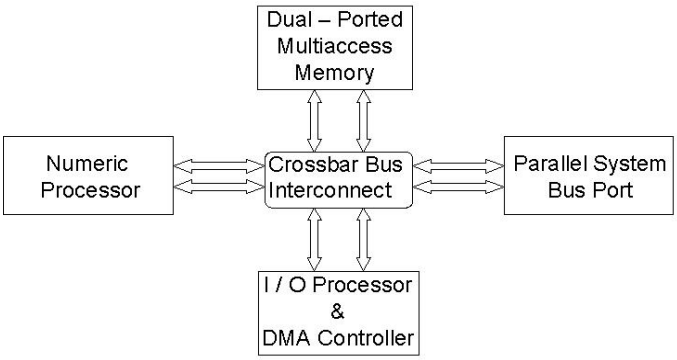
1.2 ARQUITECTURA

La arquitectura *Super Hardvard* es un término inventado por *Analog Devices* para describir la operación interna de los DSPs de la familia SHARC. Sobre la arquitectura *Hardvard*, que consiste básicamente en separar la memoria de datos e instrucciones de programa con buses separados para cada una; lo cual permite que datos e instrucciones puedan ser buscados al mismo tiempo, mejorando notablemente la velocidad sobre el diseño de un solo bus de la arquitectura *Von Neuman*; se adiciona un caché de instrucciones y un controlador de I/O dedicado [43]. En la figura 1 se muestra el diagrama de bloques de la arquitectura *Super Hardvard*, la cual consiste en un bus interruptor que interconecta el procesador numérico del corazón del DSP a un procesador independiente I/O, a un puerto de memoria dual y al bus de un sistema de puerto paralelo.

1.2.1 Corazón del DSP

• **Unidades computacionales.** El corazón del DSP contiene tres unidades computacionales independientes: ALU, Multiplicador y *Shifter*.

Figura 1. Arquitectura *Super Hardvard*

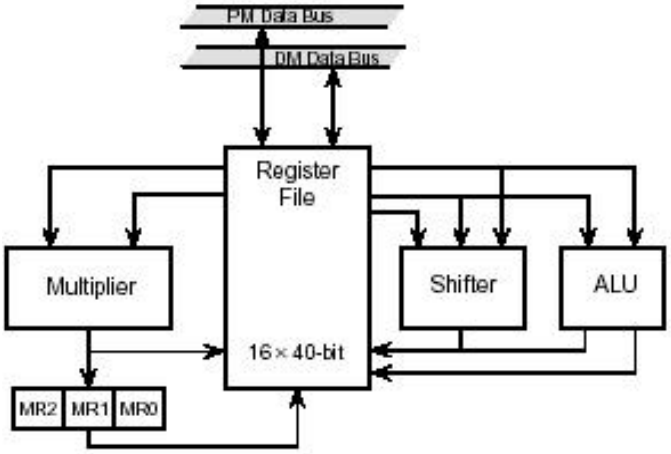


Fuente: ANALOG DEVICES. ADSP-21065L SHARC User's Manual

Las unidades computacionales están dispuestas en paralelo como se muestra en la figura 2. La salida de alguna de estas unidades puede ser la entrada de otra en el próximo ciclo. Las unidades

computacionales almacenan los operandos de entrada y resultados en el archivo de registros, el cual es accedido por el bus de datos de la memoria de programa (PMD) y el bus de datos de la memoria de datos (DMD). Ambos buses transfieren datos entre las unidades computacionales y la memoria interna, la memoria externa, y otras partes del procesador.

Figura 2. Diagrama de bloques de las unidades computacionales



Fuente: ANALOG DEVICES. ADSP-21065L SHARC User's Manual

ALU. Desarrolla un set estándar de operaciones aritméticas y lógicas en punto fijo y en punto flotante. Las instrucciones de la ALU incluyen las operaciones de suma, resta, suma y resta multifunción, y promedio, tanto en punto fijo como en flotante; manipulación de logaritmo binario, escala y mantisa en punto flotante; suma con *carry*, resta con préstamo, incremento y decremento en punto fijo; operaciones lógicas AND, OR XOR y NOT en punto fijo; las funciones valor absoluto, mínimo, máximo, comparar, pasar y cortar en punto flotante; primitivas de recíproco y recíproco de la raíz cuadrada en punto flotante. Las operaciones de la ALU toman dos operandos de entrada X y Y, los cuales pueden ser algunos de los 16 registros del archivo de registros. Las operaciones usualmente retornan un resultado excepto en la operación dual de suma y resta que retorna dos resultados, y en la operación de comparación que no retorna resultado sino que actualiza una bandera.

Las operaciones de la ALU afectan el bit 13 (ALUSAT) de saturación, el bit 15 (TRUNC) modo de redondeo y el bit 16 (RND32) redondeo límite, del registro MODE1. Además la ALU actualiza siete banderas del registro ASTAT y cuatro banderas del registro STKY al finalizar cada operación [12]. Los registros del sistema MODE1, ASTAT y STKY se explican con detalle en el apéndice E del libro de referencia técnica del DSP [7].

Multiplicador. Realiza multiplicaciones en punto fijo y punto flotante así como operaciones de multiplicación y suma, y multiplicación y resta en punto fijo mediante un acumulador. Las instrucciones en punto fijo operan en 32 bits y sus resultados se almacenan en el registro MR de 80 bits, permitiendo que el dato de entrada sea fraccionario, entero con o sin signo, y complemento a dos. Las instrucciones del multiplicador también incluyen redondeo, saturación y borrado del registro de resultado MR. El multiplicador toma dos operandos de entrada, X y Y, que pueden ser

algunos de los 16 registros del archivo de registros. El multiplicador actualiza cuatro banderas en cada operación, los bits 6, 7, 8 y 9 del registro ASTAT y cuatro banderas del registro STKY.

***Shifter*.** Trabaja con operandos de 32 bits en punto fijo y realiza las operaciones de desplazamiento izquierda y derecha, rotación; manipulación de bits, colocar, borrar y probar; extensión y depósito de campos; y soporta operaciones de conversión entre punto fijo y puno flotante. Las operaciones del *shifter* toma de uno a tres operandos de entrada, X que es la entrada operada, Y que especifica la magnitud del desplazamiento, longitud del campo de bits o posición de los bits, y Z que es el la salida actualizada. El *shifter* retorna una salida al archivo de registros y actualiza tres banderas al finalizar cada operación, los bits 11, 12 y 13 en el registro ASTAT.

• **Archivo de registro de datos.** Es una interfase entre los buses internos del procesador y las unidades computacionales. Es un almacenador temporal de operandos y resultados. Consiste de 16 registros primarios y 16 registros segundarios, de 40 bits; y usa como fuente para escribir los datos, en orden precedente, el registro universal o memoria de datos (DM), la memoria de programa (PM), la ALU, el multiplicador y el *shifter*.

En el lenguaje *assembly* los registros individuales del archivo de registro llevan un prefijo. Una “F” o “f” indica cómputos en punto flotante, mientras que una “R” o “r” indica cómputos en punto fijo. Las siguientes instrucciones utilizan algunos registros: “F0 = F1*F2“; que indica una multiplicación en punto flotante y “R0 = R1+ R2;” que indica una suma en punto fijo.

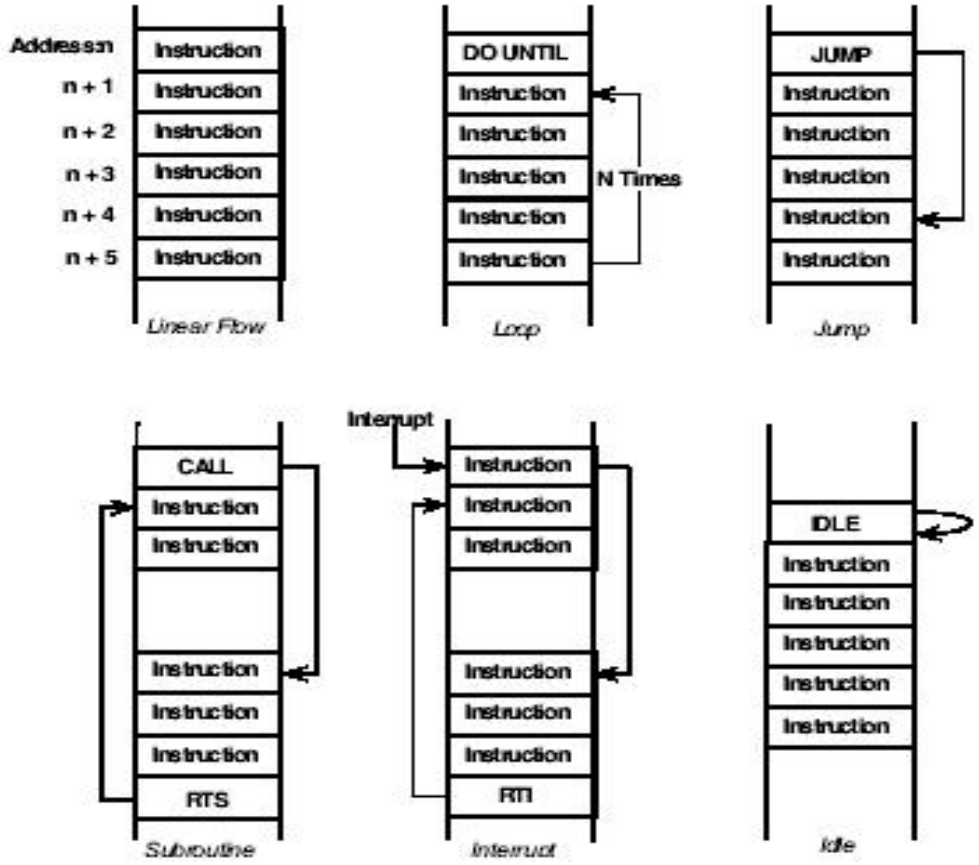
Los registros alternos o registros segundarios pueden ser activados y desactivados mediante la manipulación de bits del registro del sistema MODE1, el bit 7 (SRRFH) para los registros alternos superiores R15 a R8, y el bit 10 (SRRFL) para los registros alternos inferiores R7 a R0. Para ver detalles de las instrucciones para el manejo de estos bits ver el apéndice A del libro de referencia técnica del DSP [4].

• **Secuenciador de programa.** El secuenciador de programa junto con los generadores de direcciones de datos (DAGs) permite ejecutar las operaciones computacionales con máxima eficiencia ya que quedan libres de las unidades computacionales para procesar datos exclusivamente. Usando el caché de instrucciones, el ADSP-21065L pueden simultáneamente buscar una instrucción (desde el caché) y acceder dos operandos de datos (desde memoria).

El procesador ejecuta instrucciones de programa secuencialmente, en una línea de flujo a menos que se produzca una variación por alguna de las estructuras de programa: lazos, subrutinas, saltos, interrupciones o estado de espera (Idle).

En la figura 3 se muestra las variaciones en el flujo de programa cuando se invocan estas estructuras de programa. El secuenciador de programa suple las direcciones de las instrucciones a la memoria de programa. Este controla las iteraciones del lazo y evalúa las instrucciones condicionales. Usando un contador de lazo interno y una pila de lazo, el procesador ejecuta lazos con desbordamiento cero.

Figura 3. Variaciones del flujo de programa



Fuente: ANALOG DEVICES. ADSP-21065L SHARC User's Manual

El secuenciador de programa incluye un caché de instrucciones de 32 palabras que habilita la operación de tres buses para buscar una instrucción y dos datos de valores. El caché es selectivo, solamente instrucciones buscadas que tienen conflicto con el acceso de datos de la memoria de programa son cacheadas. Este permite la ejecución rápida de operaciones dentro de un lazo, tales como multiplicaciones y sumas de un filtro FIR o las mariposas de la FFT.

Las instrucciones del ADSP-21065L se procesan en tres ciclos de reloj: el ciclo de búsqueda, donde el procesador lee la instrucción desde el caché de instrucciones o la memoria de programa; el ciclo de decodificación, donde el procesador decodifica la instrucción, el cual genera la condición que controla la ejecución de instrucciones; y el ciclo de ejecución, donde el procesador ejecuta la instrucción completando la operación de la instrucción especificada. En el flujo de programa secuencial, mientras el corazón del procesador está buscando una instrucción, también está decodificando la instrucción buscada en el ciclo previo y ejecutando la instrucción buscada en los dos ciclos anteriores.

Lazos. El procesador soporta lazos de programa con la instrucción “DO UNTIL” y causa que el procesador repita una secuencia de instrucciones hasta que una condición especificada sea verdadera.

Ramificaciones. La instrucción “CALL” inicia una subrutina. Tanto el salto (JUMP) como la llamada a subrutina (CALL) transfieren el flujo del programa a otra posición de memoria, pero una llamada (CALL) permite el retorno de subrutina mediante la instrucción “RTS” o la instrucción “RTI” que es retorno de interrupción. Tanto el salto como las llamadas a subrutina y retornos pueden ser condicionales; indirectos, directos o relativos PC; y retardados o no retardados.

Para detalles de la operación del secuenciador de programa y cada una de las diferentes clases de flujo de programa que este soporta ver el capítulo tres del manual de usuario del DSP [14] y las instrucciones que ejecutan las estructuras de programa son tratadas en el apéndice A del libro de referencia técnica del DSP [3].

Interrupciones. Las interrupciones pueden ser internas o externas. Una interrupción fuerza a llamar a una subrutina para una dirección predefinida del vector de interrupciones. El procesador asigna un vector único para cada tipo de interrupción. El ADSP-21065L tiene cuatro interrupciones por hardware externas: tres interrupciones de propósito general IRQ_{2-0}, y una interrupción especial para el *reset*. El procesador también genera interrupción interna para el *timer*, controlador DMA, desbordamiento de *buffer* circular, desbordamiento de pila, excepciones aritméticas, vector de interrupciones para multiprocesador e interrupciones por software definidas por el usuario.

El procesador considera petición de interrupción si se cumplen las siguientes condiciones: la solicitud no es enmascarada, la interrupciones están globalmente habilitadas (registro IRPT1=1), la petición de prioridad alta no está pendiente. Una petición válida invoca una subrutina de interrupciones que salta a la dirección especificada por la interrupción. Los vectores de interrupción están espaciados por intervalos de cuatro instrucciones, pero algunas aplicaciones pueden saltar a otra región de memoria para acomodar un servicio de interrupciones mayor. La ejecución del programa retorna a la secuencia normal cuando el procesador ejecuta retorno de interrupción mediante la instrucción “RTI”.

Para habilitar o deshabilitar todas las ininterrupciones, excepto *reset*, se activa (1) o desactiva (0) el bit 12 (IRPTEN) en el registro del sistema MODE1. Los registros IRPTL y IMASK contienen todas las interrupciones del procesador. El registro IMASK controla el enmascaramiento de todas las interrupciones excepto *reset*. Enmascarar significa deshabilitar. Los bits en IMASK corresponden exactamente a los bits en el registro IRPTL. Para detalles de estos registros y la direcciones del vector de interrupciones ver el apéndice F del libro de referencia técnica del DSP [11]. Todo lo concerniente a interrupciones es tratado con profundidad en el capítulo tres del manual de usuario del DSP [15].

Cuando se trabaja con la tarjeta de evaluación se define un archivo en *assembly* “Tabla_Interrupciones.asm” que contiene las direcciones del vector de interrupciones del procesador y las variables externas que las invocan en otros archivos. Este archivo se muestra en los anexos.

Idle. Es una instrucción especial que causa que el procesador pare su operación y permanezca en estado inactivo hasta que ocurra una interrupción.

• **Generadores de direcciones (DAGs).** El procesador cuenta con dos generadores de direcciones (DAGs) que, manteniendo los punteros en memoria, simplifican la tarea de organización de datos. Los DAGs habilitan el procesador para direccionar memoria indirectamente; es decir, una instrucción especifica un registro DAG que contiene la dirección de un valor en lugar del valor. El generador de direcciones 1 (DAG1) genera direcciones de 32 bits en el bus de direcciones de la memoria de datos. El generador de direcciones 2 (DAG2) genera direcciones de 24 bits en el bus de direcciones de la memoria de programa. Los DAGs proveen soporte en hardware para algunas funciones comúnmente usadas en los algoritmos de procesamiento digital de señales. Ambos DAGs soportan *buffers* circulares de datos, los cuales permiten la implementación eficiente de líneas de retardo requeridas en filtros digitales y efectos de audio.

Cada DAG tiene cuatro tipos de registros: *Index* (I), Modificador (M), Base (B) y Longitud (L). El registro I actúa como un puntero a memoria, el registro M contiene el valor del incremento del puntero, el registro B almacena la dirección base (primera dirección) de un *buffer* circular y el registro L contiene el número de posiciones en el *buffer* circular, definiendo su longitud; si L=0 el *buffer* no es circular. Cada DAG contiene ocho registros para cada tipo de registro como se muestra en la tabla 1. Además cada registro DAG tiene un registro alterno que pueden ser activados o desactivados mediante la manipulación de los bits 3 (SRD1H), 4 (SRD1L), 5 (SRD2H) y 6 (SRD2L) en el registro del sistema MODE1. En el capítulo cuatro de manual de usuario se trata con detalle la operación de los DAG y el direccionamiento de los *buffers* circulares [16]. Las instrucciones en *assembly* referidas a los registros de los DAGs, para definir *buffer* circulares y la transferencia de datos a memoria se explican en el apéndice A del libro de referencia técnica.

Tabla 1. Registros DAG

DAG1 (32 bits DM)	***DAG2 (24 bits PM)***
B0 – B7	B8 – B15
I0 – I7	I8 – I15
M0 – M7	M8 – M15
L0 – L7	L8 – L15

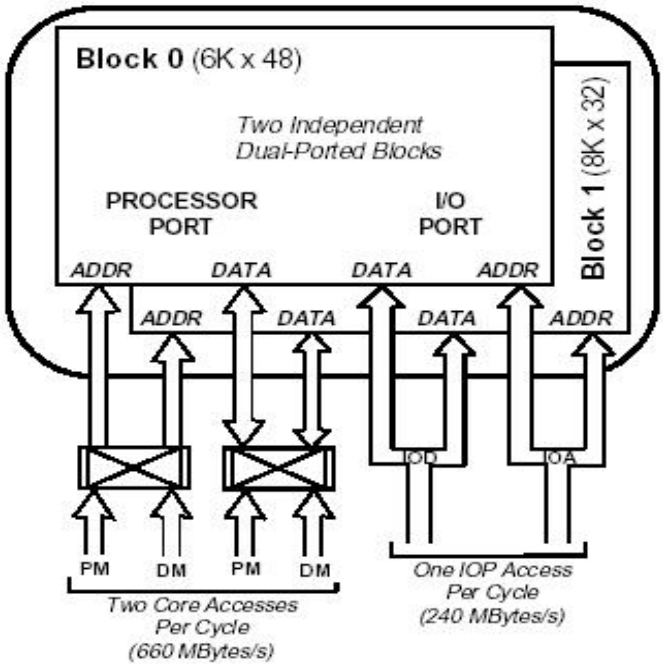
Fuente: ANALOG DEVICES. ADSP-21065L SHARC User's Manual

1.2.2 Memoria. El ADSP-21065L tiene 544 Kbits de memoria *on-chip* SRAM organizada en dos bancos: banco 0 (288 Kbits) distribuido en nueve columnas de 2Kx16 bits y el banco 1 (256 Kbits) distribuido en ocho columnas de 2Kx16 bits. Cada puerto de memoria es un puerto dual con acceso independiente por el corazón del procesador, el controlador DMA o el procesador I/O en un mismo ciclo. La memoria puede ser configurada con un máximo de 16K *words* de datos de 32 bits, 34K *words* por 16 bits de datos, 10K *words* de 48 bits de instrucciones (y 40 bits de datos) o combinaciones de diferentes tamaños de palabra hasta 544 Kbits.

El procesador tiene tres buses conectados al puerto dual de memoria: el bus DM, el bus PM y el bus I/O. Los buses DM y PM se conectan al puerto de memoria del procesador, y el bus I/O se conecta al puerto de memoria I/O como se muestra en la figura 4. Puesto que cada bloque de memoria puede almacenar combinaciones de datos y código, el acceso es más eficiente cuando

un bloque almacena datos, usando el bus DM para transferencias, y el otro bloque almacena instrucciones y datos, usando el bus PM para transferencias. Usando el bus DM o PM en esta forma, con uno dedicado a cada bloque de memoria, se asegura la ejecución en un ciclo de la transferencia de dos datos, suministrando la instrucción habilitada en el caché. La ejecución en un ciclo también se mantiene cuando un dato operando es transferido desde o hacia una memoria externa, a través del puerto externo del DSP.

Figura 4. Conexión de buses a la memoria *on-chip* SRAM



Fuente: ANALOG DEVICES. ADSP-21065L SHARC User's Manual

El procesador tiene un total de espacio de direcciones de 64M words para el mapa de memoria y está dividido en tres secciones: espacio de memoria interna, correspondiente al espacio de direccionamiento de los registros IOP del procesador, palabra normal y palabra corta; espacio de memoria para multiprocesador, correspondiente a los registros IOP de otros procesadores en un sistema multiprocesador; y el espacio de memoria externa, correspondiente a la memoria externa *off-chip* y equipos I/O mapeados en memoria. En la tabla 2 se muestra los límites de las direcciones para estos espacios en memoria separados por espacios reservados.

Los registros IOP son los registros de estado y control permanente localizados en el procesador de entradas y salidas (I/O) del procesador. Son un *set* separado de registros de control y datos mapeados en la memoria interna del procesador. Las aplicaciones de software usan los registros IOP para configurar el nivel de funciones del sistema, incluyendo puerto serial I/O, transferencias DMA, *timers* programables, puertos I/O de propósito general, vector de interrupciones e interfase con memoria externa SDRAM [10]. Las aplicaciones de software pueden usar los nombres simbólicos de los registros o bits individuales, definidos en el archivo "def21065L.h" permitiendo

correr los programas del usuario usando el simulador o la tarjeta de evaluación. Este archivo se muestra en los anexos.

Tabla 2. Secciones del mapa de memoria del procesador

Dirección de inicio	***Dirección final***	***Contenido***
0x0000 0000	0x0000 00FF	Registros IOP
0x0000 0100	0x0000 02FF	Multiprocesador
0x0000 0100	0x0000 02FF	Reservado
0x0000 8000	0x0000 9FFF	Bloque 0 normal
0x0000 A000	0x0000 BFFF	Reservado
0x0000 C000	0x0000 DFFF	Bloque 1normal
0x0000 E000	0x0000 FFF	Reservado
0x0001 0000	0x0001 3FFF	Bloque 0 *short*
0x0001 4000	0x0001 7FFF	Reservado
0x0001 8000	0x0001 BFFF	Bloque 1 *short*
0x0001 C000	0x0001 FFFF	Reservado
0x0002 0000	0x03FF FFFF	Memoria Externa

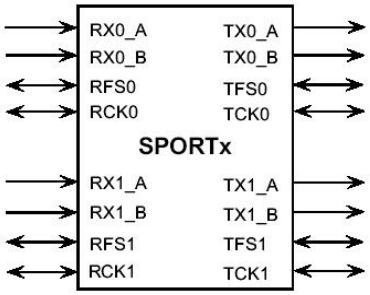
Fuente: YASSER MÉNDEZ. Autor del proyecto.

El ADSP-21065L posee además una interfase de memoria externa DRAM síncrona (SDRAM) soportando una transferencia de datos de hasta 240 Mbytes/s. Soporta SDRAM estándar de 16M, 64M y 128M y configuraciones x4, x8, x16 o x32. El uso de esta interfase permite que el usuario maneje una gran cantidad de datos que no cabrían en el segmento de datos de la memoria interna del procesador; tal es el caso de los algoritmos de efectos de audio con retardo variable en el tiempo, donde se requiere definir *buffers* circulares de gran longitud, para almacenar un periodo de los osciladores de baja frecuencia (LFO) y guardar un gran número de muestras de entrada y salida para realizar la evaluación de estos algoritmos. En las aplicaciones con la tarjeta de evaluación se inicializa la memoria externa SDRAM en el archivo *assembly* "Inicializacion_SDRAM.asm" el cual se muestra en los anexos. Además en el archivo *linker*, que contiene la definición del procesador empleado, se definen las direcciones de los espacios de memoria requeridos para la aplicación particular. El capítulo cinco del manual de usuario del DSP provee la información completa sobre el manejo de la memoria [17] y en el capítulo 10 se explica con detalle la configuración, control y operación de la interfase SDRAM [20].

1.2.3 Puertos seriales. El ADSP-21065L se caracteriza por tener dos puertos seriales síncronos e independientes, SPORT 0 y SPORT 1, que proveen una interfase I/O a equipos periféricos. Cada puerto serial tiene un *set* de registros de control y *buffers* de datos, y operan a una tasa máxima de 30 Mbits/s. Cada puerto serial tiene un *set* primario y uno segundario de canales de recepción (Rx) y transmisión (Tx). Las funciones independientes de transmisión y recepción proveen una gran flexibilidad para comunicación serial, los datos pueden ser transferidos automáticamente desde y hacia memoria a través de DMA. Cada puerto serial soporta tres modos e operación: modo estándar, modo I^2S (interfase comúnmente usada por los codec de audio), y modo multicanal TDM (multiplexado por división en el tiempo). Ofrecen además sincronización y modo de transmisión seleccionable, y compresión-expansión ley μ o ley A opcional. El reloj del puerto serial y los marcos de sincronismo pueden ser generados interna o externamente.

Las comunicaciones seriales son sincronizadas por una señal de reloj. Cada puerto serial puede generar o recibir su propia señal de reloj de transmisión (TCLK) y señal de reloj de recepción (RCLK). Las frecuencias del reloj serial generadas internamente se pueden configurar en los registros TDIVx y RDIVx del puerto serial. Los *buffers* de datos de transmisión para el puerto SPORT0 son el TX0_A y el TX0_B, y para el puerto SPORT1 son el TX1_A y el TX1_B, los cuales pueden ser cargados con datos de 32 bits por el controlador DMA o el corazón del procesador. RX0_A y RX0_B son los *buffers* de datos para el SPORT0, y RX1_A y RX1_B son los *buffers* de datos para el SPORT1. El registro de desplazamiento receptor carga automáticamente estos *buffers* de 32 bits cuando un puerto serial ha recibido una palabra entera. En la figura 5 se muestra un esquema de la configuración de las entradas y salidas de los puertos seriales.

Figura 5. Configuración entradas/salida puertos seriales



Fuente: ANALOG DEVICES. ADSP-21065L SHARC User's Manual

Los registros de control de transmisión y recepción para cada puerto serial son STCTLx y SRCTLx respectivamente. La manipulación de estos registros permite la configuración de los puertos seriales para operar en cada uno de los tres modos disponibles. En el capítulo nueve del manual de usuario se encuentra toda la información correspondiente al manejo de los puertos seriales [19] y los registros de control también son tratados en el apéndice E del libro de referencia técnica del DSP [10].

1.2.4 Interfase Host. Provee una conexión para buses de microprocesador estándar de 8, 16 y 32 bits y soporta transferencia asíncrona al máximo de la rata del reloj del procesador. La interfase *host* tiene acceso al procesador a través del puerto externo, sobre el bus externo, y está mapeado en memoria en espacio de direcciones unificadas del procesador. La trasferencia de datos y código hacia y desde el procesador se realiza mediante sobre dos canales DMA. El *host* puede leer y escribir directamente los registros IOP del procesador, incluyendo los *buffers* EPBx FIFO. Usa mensajes de interprocesador y el vector de interrupciones para asegurar que los comandos del *host* se ejecuten eficientemente. Una explicación detallada del control del *host* del procesador, la transferencia y empaquetado de los datos se presenta en el capítulo ocho del manual de usuario [18].

1.2.5 Puertos I/O. El procesador tiene 12 pines I/O $FLAG_{11-0}$ de propósito general programables, que pueden funcionar como entradas o salidas. Como salidas, estos pines pueden manejar equipos periféricos; como entradas, pueden proveer pruebas para ramificaciones condicionales. El registro del sistema MODE2 configura la funcionalidad, o dirección de los pines $FLAG_{3-0}$, y el registro ASTAT contiene el valor de estos bits. Los registros IOCTL y IOSTAT contienen los bits de control y estado de los pines $FLAG_{11-4}$. Las operaciones de manipulación de bits como BIT TST,

BIT CLR y otras, no pueden ejecutarse directamente sobre los registros IOP asociados a los pines $FLAG_{11-4}$, sino que se debe primero transferir el contenido del registro IOSTAT a un registro del archivo de registros o a otro registro universal. Para detalles del registro MODE2, ASTAT, IOCTL y IOSTAT ver el apéndice E del libro de referencia técnica del DSP [9].

1.2.6 *Timers* Programables. El ADSP-21065L provee dos bloques de *timer* programables independientes. Cada bloque puede funcionar en el modo generación de formas de onda de ancho de pulso (PWMOUNT), o en el modo contador de ancho de pulso o modo de captura (WIDTH_CNT). Cuando el *timer* en modo contador el procesador puede generar una forma de onda con un ancho de pulso arbitrario con un máximo periodo de 71.5 segundos. En modo contador y en modo de captura el procesador puede medir anchos de pulsos altos o bajos y el periodo de una forma de onda de entrada. El *timer* tiene un pin de entrada o salida, PWM_EVENTx, el cual funciona como pin de salida en el modo PWMOUNT y como pin de entrada en el modo WIDTH_CNT. Para implementar estas funciones, cada *timer* tiene tres registros: TPERIOx, TPWIDTHx y TCOUNTx. Para habilitar o deshabilitar el *timer*, se puede activar o borrar el bit TIMENx en el registro MODE2. La operación de los modos de funcionamiento del *timer* y el manejo de los bits de control del *timer* y de sus interrupciones se detallan en el capítulo 11 del manual de usuario del DSP [21].

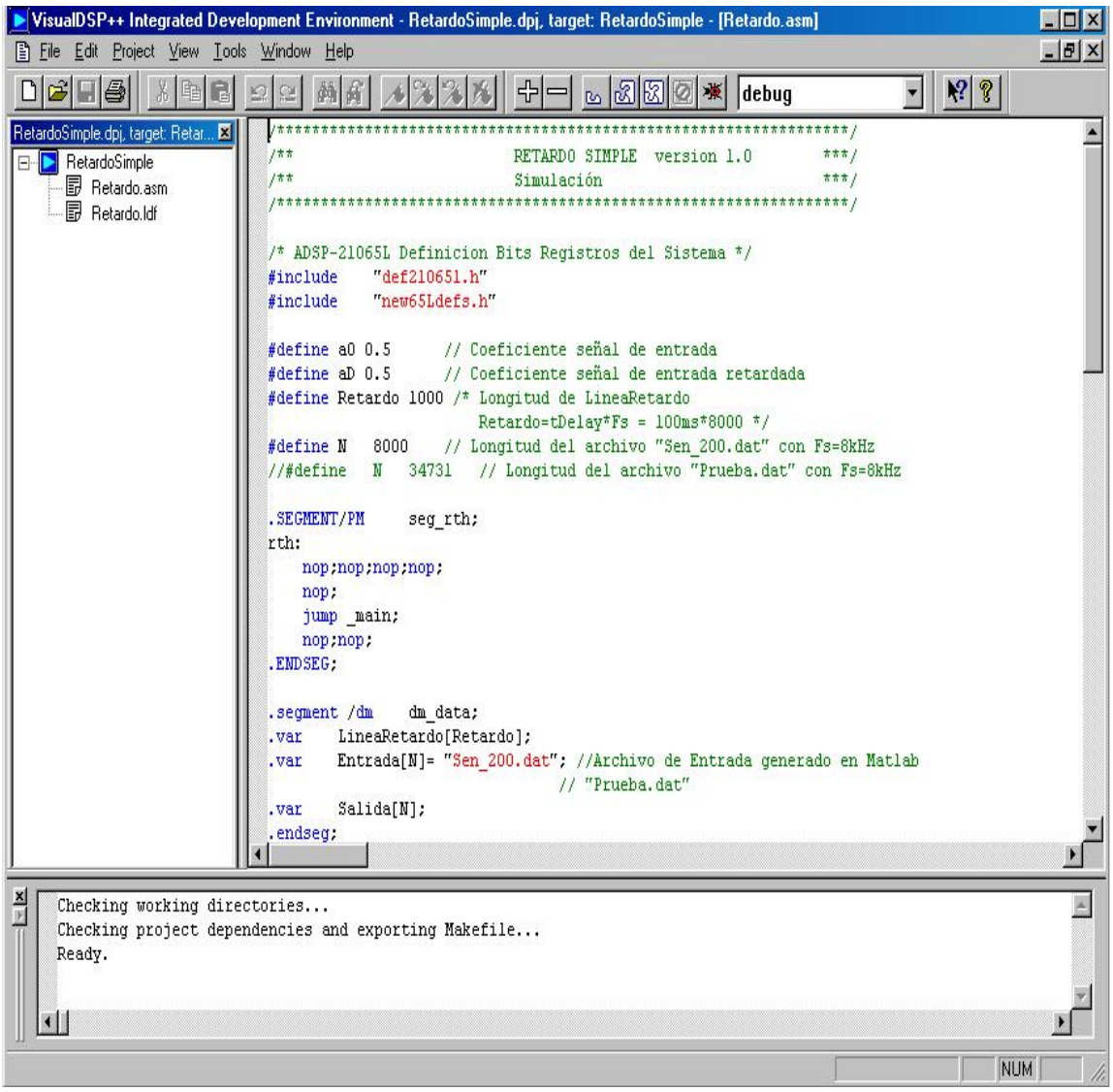
1.2.7 Controlador DMA. Permite la transferencia de datos con desbordamiento cero, operando independientemente del corazón del procesador; de tal manera que ocurren operaciones DMA, mientras el corazón está ejecutando simultáneamente un programa. Algunas aplicaciones pueden usar transferencias DMA para bajar código y datos al DSP. La transferencia DMA puede ocurrir entre la memoria interna y la memoria externa del DSP, los puertos seriales, periféricos externos o un procesador *host*; y entre la memoria externa y periféricos externos. Durante la transferencia DMA el controlador empaqueta y desempaqueta automáticamente las palabras al bus externo. El ADSP-21065L habilita los 10 canales DMA, ocho vía puerto serial y dos vía puerto externo (para otro procesador, otra memoria u otra transferencia I/O). La operación del controlador DMA, la configuración de transferencias DMA y los registros de control DMA se tratan con profundidad en el capítulo seis el manual de usuario del DSP.

1.3 HERRAMIENTAS DE DESARROLLO

1.3.1 Software Visual DSP++. El programa VisualDSP++ es un software de desarrollo para los DSP de *Analog Devices*; permite el manejo flexible de proyectos para el desarrollo de aplicaciones DSP, permitiendo al usuario ejecutar todas las actividades necesarias para crear y depurar estas aplicaciones.

El VisualDSP++ consiste de un ambiente integrado de desarrollo (IDE) y un depurador (*debugger*). El IDE provee el soporte para la edición de programas, manejo de proyectos y herramientas controladas de construcción. El editor IDE permite crear y modificar los archivos fuente, ya sea en *C* o en *assembly*; crear un proyecto (DPJ) e incluir los archivos que este requiera para funcionar correctamente; construir el proyecto permitiendo detectar los errores desplegados en la ventana de salida del editor. Por otra parte, una vez se ha construido satisfactoriamente (sin errores de compilación) un proyecto se genera un archivo ejecutable (DXE) que es corrido por el *debugger*, el cual puede ser ejecutado directamente desde el editor IDE. El proyecto puede ser depurado usando el simulador (ADSP-2106x *Family Simulator*) o la tarjeta de evaluación (EZ-KIT *Lite*) según sea el caso. En la figura 6 se muestra la ventana principal del IDE.

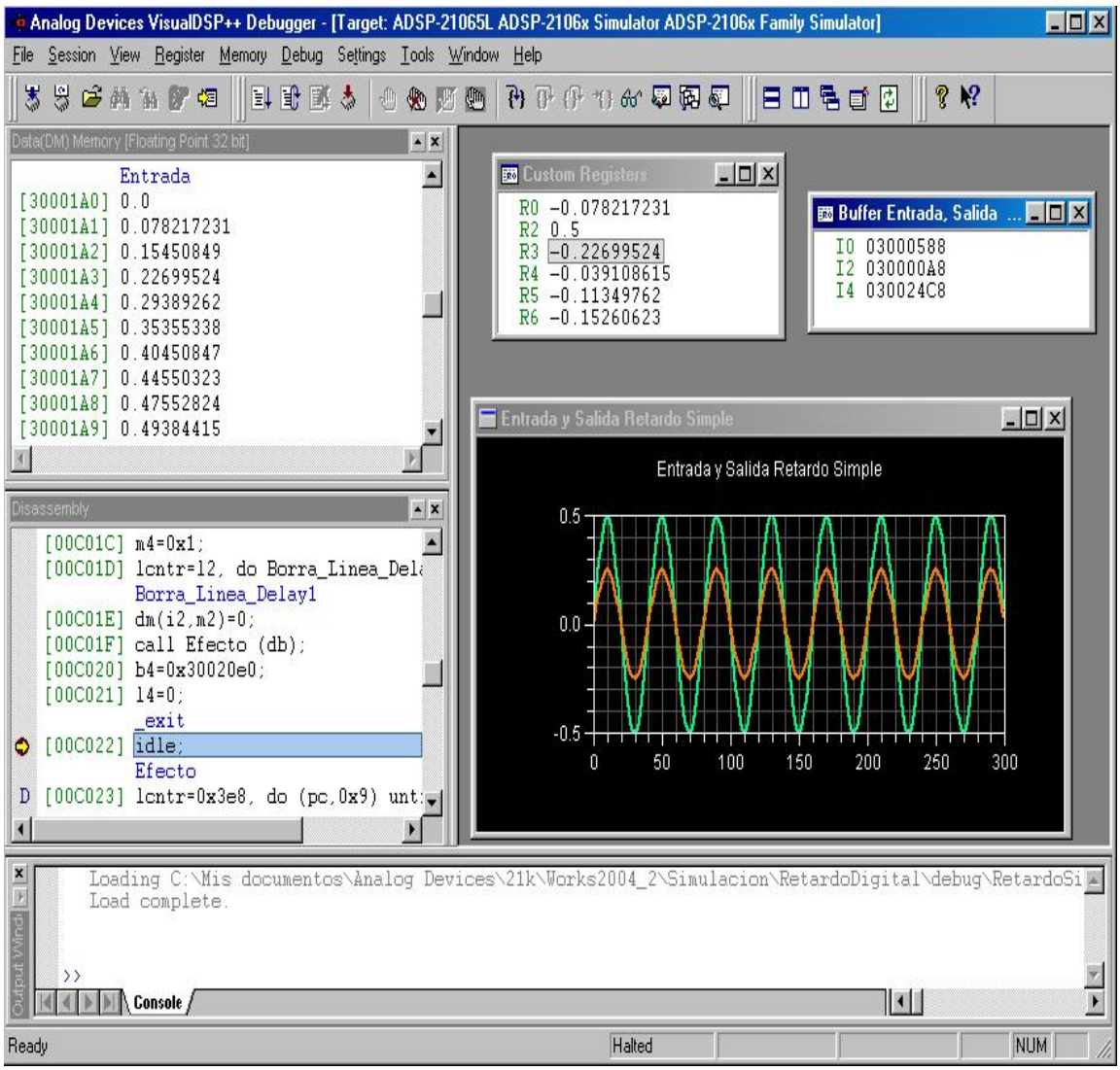
Figura 6. Ventana principal Visual DSP++ IDE



Fuente: ANALOG DEVICES. VisualDSP++ User's Guide for ADSP-21xxx Family DSPs

El *debugger* permite desplegar ventanas para visualizar el código del archivo fuente en *desasembly*, el contenido de memoria como los datos de un *buffer*, los registros temporales con formato escogido por el usuario, registros de los DAGs que definen los *buffers* utilizados, o la gráfica de los datos almacenados en un *buffer* en la memoria del DSP, entre otras opciones, como se muestra en la figura 7. Un compendio de los pasos para la creación y construcción de un proyecto, y el uso de la interfase para completar estos pasos, se encuentra en el capítulo dos de la guía del usuario del VisualDSP++ [24]; la información sobre operación de ventanas de memoria y registros, y operaciones avanzadas de depuración se presenta en el capítulo tres del mismo libro [25]. Además tanto el IDE como el *debugger* tienen un archivo de ayuda con toda la información necesaria sobre el uso de sus menús y herramientas.

Figura 7. Ventana principal *Debugger*



Fuente: ANALOG DEVICES. VisualDSP++ User's Guide for ADSP-21xxx Family DSPs

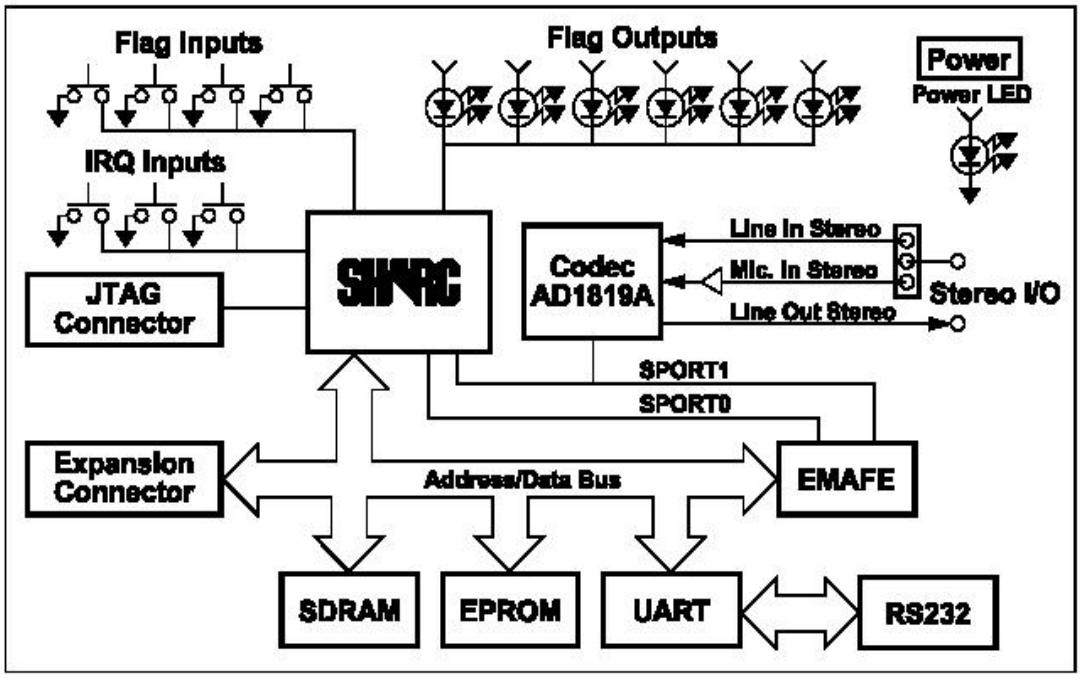
1.3.2 La tarjeta de evaluación. La tarjeta de evaluación ADSP-21065L EZ-KIT Lite está diseñada en conjunto con el VisualDSP++ y las herramientas SHARC para un completo sistema de desarrollo de código y depuración. Usando la tarjeta EZ-KITE Lite con el depurador, se pueden observar el ADSP-21065L ejecutando programas desde *on-chip RAM*, interactuar con equipos *on-board* y comunicarse con periféricos localizados en módulos adicionales. Se puede acceder al procesador a través del puerto serial (COM1 o COM2) desde el PC, o desde el emulador. El programa monitor que opera sobre el EZ-LAB, le permite acceder a la memoria interna del procesador, a través del puerto serial.

La tarjeta de evaluación ADSP-21065L EZ-KITE Lite dispone de los siguientes componentes:

- Un ADSP-21065L operando a 60 MHz
- Un Codec AD1819A de 16 bits
- Una interfase RS-232
- Una EPROM en *socket* (128K x 8 on board, o 256K x 8, 512K x 8, y 1M x 8 seleccionable)
- Una SDRAM (1M x 32)
- Cuatro pulsadores para banderas de entrada
- Tres pulsadores para entradas IRQ
- Seis *LEDs* programables por el usuario
- Fuente de poder regulada
- Conector de expansión EMAFE (*Enhanced Modular Analog Front End*)
- Conectores de expansión

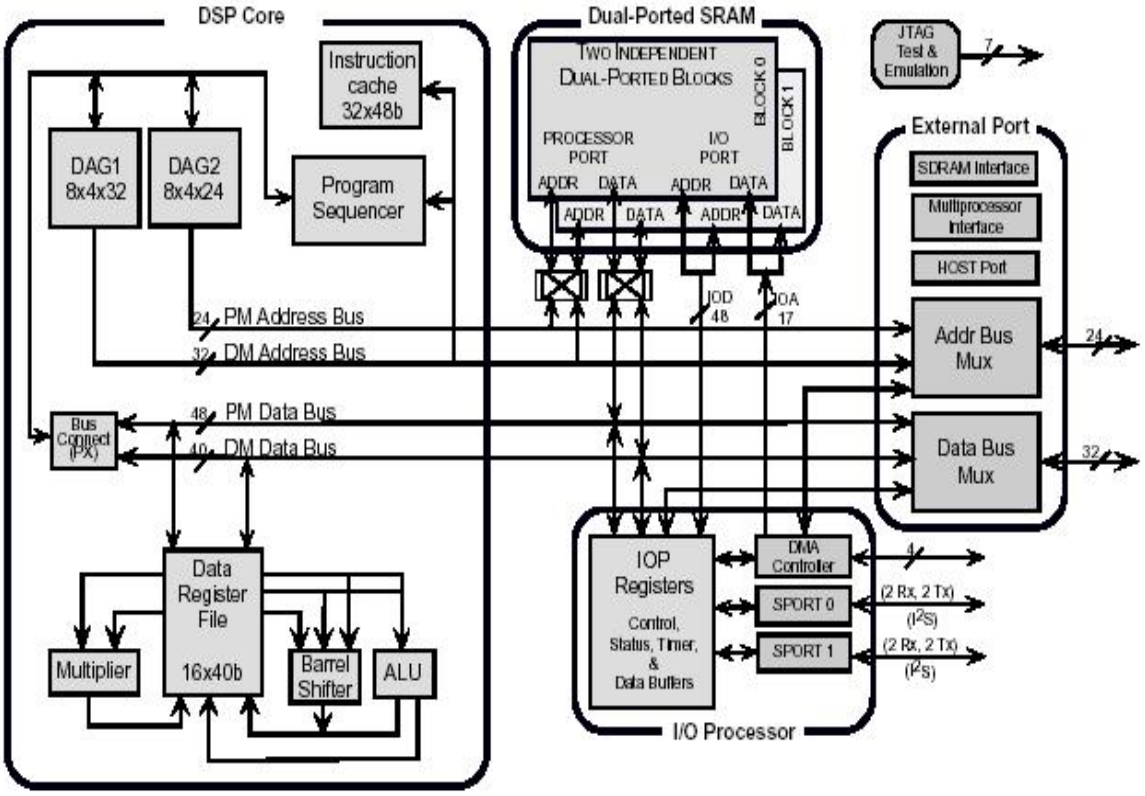
La tarjeta EZ-KIT Lite está equipada con hardware que facilita demostraciones interactivas. Los pulsadores y *LEDs* programables por el usuario permiten el control y verificación del estado de la tarjeta. Además, el Codec AD1819A *SoundPort* provee el acceso de una entrada de audio estéreo seleccionable (línea o micrófono) y una salida de audio también estéreo. Incluye un programa monitor, almacenado en una memoria no volátil, que permite al usuario cargar, ejecutar y depurar programas en el ADSP-21065L. También provee conectores de expansión que le permiten al usuario examinar las señales del procesador, así como una interfase para control de *host*. En la figura 8 se muestra la arquitectura de la tarjeta de evaluación y el diagrama detallado de la arquitectura del ADSP21-065L se muestra en la figura 9. La información sobre el procedimiento de instalación del software y hardware de este sistema de desarrollo; detalles del programa monitor, el codec y el EMAFE; y uso del depurador y programas de demostración, se encuentra en el manual de la tarjeta de evaluación [2].

Figura 8. Diagrama de bloques de ADSP-21065L EZ-KIT Lite



Fuente: ADSP-21065L EZ-KIT Lite manual rev1

Figura 9. Arquitectura del ADSP-21065L



Fuente: ANALOG DEVICES. ADSP-21065L SHARC User's Manual

2. EFECTOS DE AUDIO

2.1 DEFINICIÓN

Un efecto digital de audio es un proceso que al ser aplicado a un sonido lo modifica para mejorarlo o para reducir algún aspecto de mayor evidencia. Algunos de estos efectos provienen de la electrónica analógica. Las unidades de reverberación, son por ejemplo, un buen complemento de mezcladores y ecualizadores. Con la aparición de la electrónica digital y los computadores, se pueden controlar las diferentes unidades de efectos que modifican las señales de audio, o realizar un procesamiento sofisticado de estas señales, el cual puede ser en tiempo real.

2.2 PROCESAMIENTO EN TIEMPO REAL Y TIEMPO NO REAL

La implementación de un efecto digital de audio depende estrictamente de si el procesamiento es en tiempo real o no. Esto no significa necesariamente menor o mayor rapidez de procesado, pero si hace que se pueda o no se pueda oír el sonido al mismo tiempo que este es procesado [26].

2.2.1 Tiempo diferido. La situación en tiempo diferido ha sido el único camino del uso del computador para procesamiento digital de audio. Durante años los computadores eran lentos debido al poco desarrollo en su *hardware*. Pero la cuestión actualmente no es tanto el procesamiento sino el control. Esta es una situación diferente para el músico que quiere controlar un sonido en tiempo real. En tiempo diferido se puede componer sonidos innovadores y complejos; permite analizar las características del sonido para encontrar el mejor procesamiento. En la situación de tiempo diferido, los sonidos son grabados y luego procesados. Este tiempo puede ser lo suficientemente pequeño que se pueda oír el resultado inmediatamente después de la grabación, o también puede ser un tiempo grande, mientras el sonido pasa a través de diferentes etapas antes de obtener el sonido de salida. Hoy en día existen diversas herramientas de *software* que realizan la edición de audio, permitiendo el procesamiento de archivos de audio y un sin número de efectos; el *Cool Edit Pro* es uno de ellos [44].

2.2.2 Tiempo real. La situación en tiempo real es frecuentemente la de mayor interés. Dos cuestiones aparecen: la complejidad del proceso y el control del mismo. Esto tiene una gran influencia en la implementación del proceso: el tiempo de procesamiento de una muestra debe ser menor que el tiempo de ejecución, y el tiempo de retraso entre la entrada y la salida debe ser tal que no se perciba el retardo. El oído humano percibe retardos mayores a 70 ms.

2.3 FORMAS DE PROCESAMIENTO

Básicamente pueden aplicarse tres tipos de procesamiento:

- La técnica muestra a muestra, donde por cada muestra de entrada una muestra de salida es calculada.

- La técnica de bloques o marcos (*frames*), está muy relacionada con programas de síntesis de señales de audio como el CSound [27]. Esta ha resultado ser la más rápida para sintetizar sonidos usando *buffers*, especialmente si el programa compila o interpreta instrucciones y usa lenguajes como C, FORTRAN o Pascal.

- La técnica de vectores, donde el sonido entero de entrada es considerado como un vector, y el procesamiento es hecho sobre el vector. Lenguajes como Matlab están fuertemente orientados para tal tratamiento y son optimizados para todas las operaciones con vectores y matrices.

2.3.1 Muestra a muestra. Esta técnica puede ser llamada "simulación de equipos análogos" por que la salida es calculada muestra a muestra, dando un flujo de salida regular. Teniendo en cuenta que el retardo entre la entrada y la salida se reduce a una muestra de esta clase de procesamiento es muy conveniente para situaciones de tiempo real. El camino para escribir tales programas es para tener un único programa en el cual por una muestra de entrada una de salida. Por ejemplo un filtro puede ser fácilmente escrito como un programa "muestra a muestra", requiriendo poca memoria para guardar algunas variables provisionales. Esto no significa que instrucciones "IF" no puedan ser tomadas, pero si que, muchas direcciones dentro del programa deban ser ejecutadas en menos tiempo que el periodo de muestreo. Por ejemplo se puede implementar rectificadores o limitadores si la máquina tiene un "IF" lógico en sus instrucciones.

2.3.2 Marco a marco. Esta técnica también conocida como "procesamiento por bloques" transfiere los datos de entrada a un buffer y los procesa cada vez que el buffer se llena con los nuevos datos, es decir el sonido de salida es calculado bloque a bloque. Esto significa que el retardo entre la muestra de entrada y la muestra de salida es al menos la longitud del buffer que guarda el bloque de muestras. Comparada a la aproximación muestra a muestra, esta técnica permite una mejor distribución de la carga de computación: algunas muestras pueden requerir más tiempo que otras, o puede ser tiempo de inicialización para algunos procesos específicos. Ejemplo del uso de esta técnica son los algoritmos de la transformada rápida de Fourier. El requerimiento de tiempo de procesamiento está basado en las veces que se muestrea la memoria del buffer.

2.3.3 Vectorizado. Esta técnica está relacionada a la situación de tiempo diferido. Es usado en tiempo real solamente para aplicaciones específicas usando grandes retardos entre un sonido y su versión procesada. Este es la forma más usual para escribir programas en Matlab. Con esta clase de aproximación, un sonido de entrada es considerado como un vector. Todas las unidades de procesamiento usarán un vector como entrada y salida. Un efecto digital completo es una serie de tales procedimientos.

Las principales ventajas en términos de procesamiento son:

- Cada procedimiento es independiente y puede ser independientemente evaluado.
- El tiempo de procesamiento puede ser muy rápido según como el procedimiento esté escrito.
- Un efecto consiste solamente en una lista de procedimientos encadenados.

2.4 CLASIFICACIÓN DE LOS EFECTOS DE AUDIO

Muchos efectos se fundamentan en el mezclado de la señal original con copias retardadas o amplificadas de estas. El retardo es implementado en múltiplos enteros de la unidad de retardo. Si el retardo es mayor que una unidad de retardo, la cadena de unidades de retardos es referenciado como una línea de retardo. Si la línea de retardo y los correspondientes coeficientes son constantes, resultan los bien conocidos filtros FIR e IIR. En el caso de filtros variantes en el tiempo los coeficientes son variantes en el tiempo, pero los retardos permanecen fijos. El resultado del filtro es una superposición de amplitud modulada (AM) y copias retardadas de la señal original. Si los coeficientes son fijos y los retardos son variables, efectos de audio como el *vibrato*, *flanger* o *chorus* son producidos. Los efectos de audio se pueden clasificar en efectos de audio basados en retardos, efectos de audio basados en modulación y efectos de audio basados en amplitud, [37].

2.4.1 Efectos de audio basados en retardos. El advenimiento del audio digital, sumado al flujo de la revolución digital y procesos de grabación, ha permitido el desarrollo de sofisticados efectos de audio basados en retardos. Estos incluyen retardos simples o ecos, el *Automatic Double Tracking, Slapback Echo*, retardos multi-taps; además de efectos más sofisticados como reverberación, *chorus* o *flanger*; y todos dependen de una señal de retardo. Antes de la posibilidad de los circuitos digitales para crear estos efectos, estos podían solamente ser sintetizados usando grabadoras de retardo y complicados equipos mecánicos con circuitería analógica. Por supuesto que, la idea de utilizar un salón real como un reverberador, vía hablante y micrófono, es sumamente interesante, pero pocos se pueden dar este lujo. Aún sin un detallado entendimiento de la teoría de procesamiento de señales, es posible usar estos equipos, puesto que su operación puede ser entendida simplemente examinando la combinación de señales directas y retrasadas.

Cuando un sonido es reflejado desde una distancia, por una superficie dura, una versión retardada de la señal original aparecerá un tiempo después. Antes de introducir los DSP en el procesamiento de señales de audio, las primeras unidades de retardo fueron creadas usando retardos en las cintas magnéticas con múltiples movimientos de las cabezas de grabado, mientras otras unidades entonces producían los retardos con circuitería análoga. Para recrear esta reflexión digitalmente, unidades de efectos de retardo en DSP codifican la señal de entrada y la almacenan en un buffer de línea de retardo hasta que es requerida un tiempo después donde es decodificada a la forma análoga. Un procesador de digital de señales DSP puede producir retardos en varias formas. Unidades de retardo pueden producir resultados estéreo y resultados multi-taps [34].

Muchos procesadores de efectos implementan un retardo y usan este como base para producir efectos multiretardos y de reverberación. Las señales multiretardo pueden ser paneadas a la derecha, izquierda o mezclarlas juntas para dar la impresión de un eco estéreo rebotando desde un lado a otro.

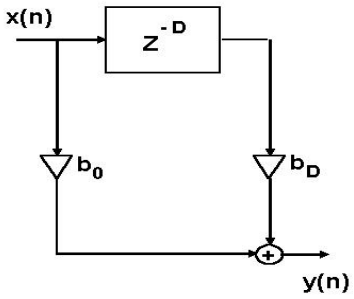
Retardo Simple. Los algoritmos de efectos de audio basados en el retardo digital consisten en un muestreo de la señal de entrada y la repetición de una copia retardada de la señal en combinación con la señal original, proporcionando un sencillo eco, [38]. Ésta retroalimentación produce una serie de repeticiones que decaen. La amplitud de la señal devuelta a la entrada determina cuantas repeticiones se oyen. El retardo digital se usa a menudo en piezas de canto cortas, en una mezcla, para obtener la apariencia de una pieza larga. También se usa normalmente en piezas musicales de un solo instrumento como saxofón o guitarra, [35]. En la figura 10 se aprecia la implementación de un retardo digital simple. Este efecto tiene solo dos parámetros la cantidad de tiempo de

retardo *D* y la amplitud relativa de la señal retrasada o ganancia b_D, y puede ser representado mediante la ecuación 1.

$$y(n) = b_0\, x(n) + b_D\, x(n - D) \qquad \text{Ecuación 1}$$

donde *D* es el retardo, b_0 es la ganancia directa y b_D es la ganancia de la entrada retrasada.

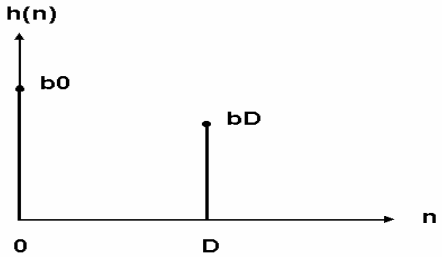
Figura 10. Diagrama de bloques retardo simple



Fuente: Yasser Méndez. Autor del proyecto

Se puede ver que la señal de entrada x(n) es adicionada a la copia retrasada de la entrada. Tanto la señal de entrada como su versión retrasada pueden ser atenuadas por un factor que es menor que 1, debido a que las superficies reflectantes, como el aire, contienen una constante de pérdida b_D debido a la absorción de la energía de la onda fuente. El retardo *D* es el tiempo total que toma la señal en retornar desde una pared reflectora. La respuesta en el tiempo consiste en dos impulsos uno correspondiente a la señal de entrada y otro a su versión retrasada como se muestra en la figura 11.

Figura 11. Respuesta al impulso retardo simple



Fuente: Yasser Méndez. Autor del proyecto

El efecto de retardo simple también puede verse como un filtro peine FIR [31], donde los picos de la respuesta en frecuencia ocurren en múltiplos de la frecuencia fundamental. Los filtros peine resultan cuando una señal de entrada es combinada con una copia de la señal retrasada. El comportamiento en el dominio del tiempo está acompañado de un interesante patrón en el dominio

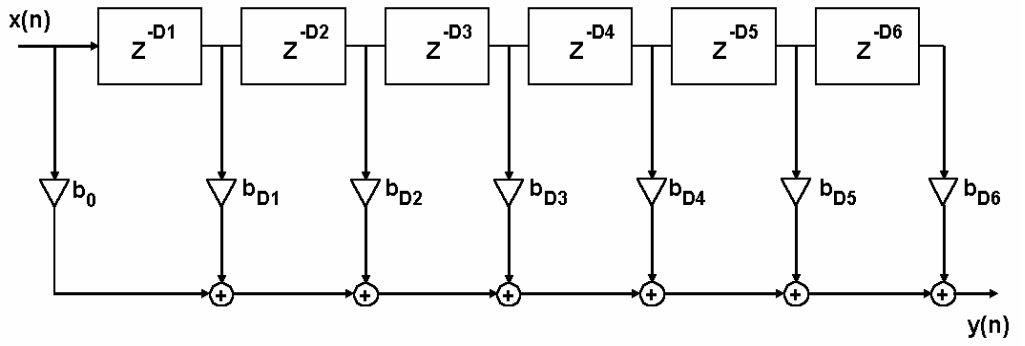
de la frecuencia. Para valores positivos de b_D, el filtro amplifica todas las frecuencias que son múltiplos de 1/T y atenúa todas las frecuencias entre ellas; para valores negativos de la ganancia de la señal de retardo ocurre el comportamiento inverso.

Los retardos digitales son frecuentemente usados sobre vocalistas en una mezcla para dar una imagen de numerosos vocalistas. Es también usado comúnmente en instrumentos como el saxofón o guitarra. El tiempo de retardo puede ser colocado al ritmo de la música, para que un eco discreto caiga sobre una octava nota, por ejemplo. El resultado es un eco que no obstruye y que produce una serie de espacios sin principios obvios. También se pueden crear un efecto doble, dando la impresión que dos instrumentos han comenzado a sonar al mismo tiempo. Tiempos de retardo entre 20-50 milisegundos trabajan bien para esto sobre instrumentos rítmicos, pero tiempos largos pueden se usados como efectos especiales sobre solos. Incrementando el retardo un ligeramente por encima de los 100 ms, se obtiene un efecto sutil. Extendiendo el retardo a un segundo y mayores, le permite la posibilidad de hacer sonar armónicas aunque solo se este tocando una sola nota.

Los retardos pueden son también importantes cuando se realiza una mezcla de instrumentos en un ambiente estéreo. Esto puede realzar la colocación de instrumentos, y haciendo que el sonido de la mezcla aumente. Un pequeño retardo puede ser más efectivo que el *panning*[2] para extender el sonido en el campo estéreo. Así, un retardo simple del orden de los 20 ms puede hacer una gran diferencia.

Retardos multi-taps. Valores de múltiples retardos de una señal de entrada pueden ser combinados fácilmente para producir reflexiones de la entrada. Como se mencionó antes, la salida es tomada después que la señal ha sido retrasada por el tiempo total de retardo. Pero también se puede tomar la salida de tal manera que solo haya sido retardada una parte del tiempo total de retardo. En otras palabras se puede obtener varios retardos mediante múltiples taps apuntando a diferentes entradas previas almacenadas en una línea de retardo, o teniendo buffers separados en memoria de diferentes tamaños, diferentes retardos, donde las muestras de entrada son almacenadas. En la figura 12 se muestra un diagrama de un retardo con seis taps.

Figura 12. Diagrama de bloques retardo multi-taps



Fuente: Yasser Méndez. Autor del proyecto

[2] Panning: Es la relación de volúmenes de salida de los canales derecho e izquierdo. Sirve para controlar el efecto estéreo de una señal de audio.

Como se observa en la figura 3 el retardo multi-taps consiste en agregar en paralelo líneas de retardo y se puede representar matemáticamente mediante la ecuación 2.

$$y(n) = b_0\, x(n) + b_{D1}\, x(n - D1) + b_{D2}\, x(n - D2) + b_{D3}\, x(n - D3) + b_{D4}\, x(n - D4) + b_{D5}\, x(n - D5) + b_{D6}\, x(n - D6)$$

Ecuación 2

El retardo multi-taps es un caso general del diseño de un retardo básico. Si se colocan todas las ganancias menos una en cero, se obtiene el retardo básico o eco visto anteriormente. El retardo multi-taps puede generalizarse aún más, agregando una línea de retroalimentación (*feedback*) desde la salida del último tap hasta el inicio de la línea de retardo. La retroalimentación permite recrear un rudimentario ambiente de reverberación, aunque en realidad este es mucho más complejo. Los retardos multi-taps son interesantes porque permiten crear patrones complejos que pueden adicionar una cualidad rítmica a un instrumento [35].

2.4.2 Efectos basados en modulación del retardo. Los efectos basados en modulación son los tipos de efectos de mayor interés y que no revisten un gasto computacional complejo. La técnica usada es comúnmente llamada Interpolación de la Línea de Retardo, donde la línea de retardo central es modulada por una señal de baja frecuencia o LFO [29]. El resultado de la interpolación/decimación de muestras dentro de la línea de retardo produce un ligero cambio de tono de la señal de entrada. Dentro de los efectos de modulación se encuentran *Flanger, Chorus, Vibrato, Pitch Shifter* y *Detune.*

El oscilador de baja frecuencia LFO puede ser una onda sinusoidal, triangular, diente de sierra o una onda randómica de baja frecuencia, entre 0.1 Hz y 10 Hz. La más usada es la onda sinusoidal y la frecuencia depende del efecto específico que se desee obtener, por ejemplo frecuencias entre los 2 Hz y 10 Hz están generalmente asociadas al efecto *flanger*; mientras que frecuencias entre 0.1 Hz y 3 Hz son usadas por el efecto *chorus* [35].

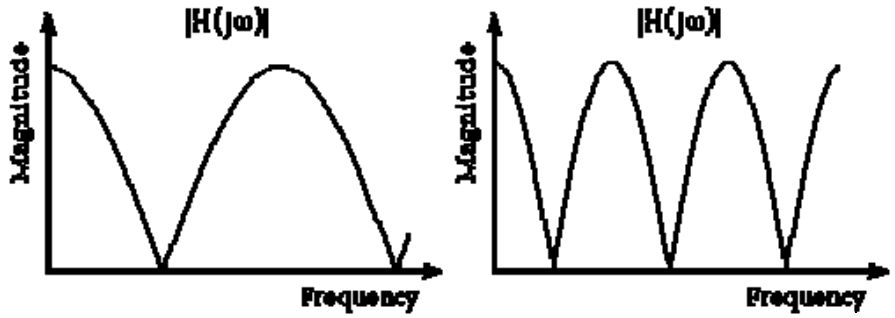
Flanger. Cuando se usa la técnica del micrófono múltiple, se combinan las señales originales con las señales retardas produciendo un efecto del filtro peine [39]. Generalmente, esto es indeseable; sin embargo, el efecto puede usarse para generar ciertos sonidos deseables. Si el tiempo de retraso constantemente se altera de forma leve, se crea un filtro con frecuencias nulas. El resultado es conocido como *flanging*. El nombre se deriva de la manera original de crear el efecto: usando un segundo grabador de cinta para retardar el sonido y reduciendo ligeramente la velocidad de las máquinas manualmente, [35].

Actualmente, los dispositivos de retardo digitales usan un oscilador para controlar el tiempo de retraso. Además, la profundidad del efecto puede controlarse mezclando la señal retardada con la señal original. Para el efecto *flanging*, el tiempo de retardo está en el rango de 0.5 a 15 ms y la tasa de modulación (que cambia el tiempo de retardo) está en el rango de 2 a 10 Hz.

Cuando se escucha el efecto del *flanging,* no se oye un eco porque el retardo es muy corto. Los tiempos de retardo típicos son de 1 a 10 milisegundos (el oído humano percibe un eco si el retardo es mayor de 50 milisegundos). En lugar de crear un eco, el retardo tiene el efecto de filtrar la señal, y este efecto crea una serie de muescas en el espectro de las frecuencias [34], como se muestra

en la figura 13, donde también se aprecia, a la izquierda del gráfico, cuando el retardo es menor. Estas muescas en la respuesta en frecuencia son creadas por interferencia destructiva. Si se escoge una onda sinusoidal pura (un solo tono) y se retrasa adicionándola a la original, la suma de las dos señales puede parecer bastante diferente. En un extremo, donde el retardo es tal que las señales están perfectamente fuera de fase, se incrementa la señal, el otro decrece la misma cantidad, desapareciendo la señal a la salida. Por supuesto, las dos señales pueden aún permanecer en fase después del retardo, doblando la magnitud de esta frecuencia (interferencia constructiva). Para un retardo dado, algunas frecuencias serán eliminadas mientras otras pasarán totalmente.

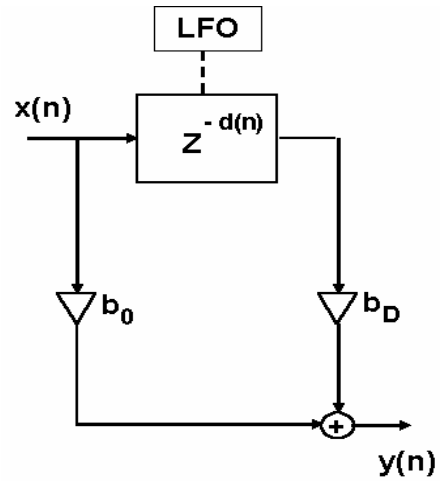
Figura 13. Respuesta en frecuencia *flanger*



Fuente: *Harmony Central*

El *flanger* es creado mezclando una señal original con una copia de la misma ligeramente retardada, donde la longitud del retardo cambia constantemente gracias a la acción de un LFO, como se muestra en la figura 14 y se puede representar mediante la ecuación 3. En algunos casos se incorpora una línea de retroalimentación de retardo medios, x(n-D/2), que produce un sonido similar al producido dentro de un tubo metálico.

Figura 14. Diagrama de bloques *flager*



Fuente: Yasser Méndez. Autor del proyecto

$$y(n) = b_0\, x(n) + b_D\, x(n - d(n))$$ Ecuación 3

El retardo variable *d(n)* se obtiene mediante la ecuación 4 y la ecuación 5.

$$d(n) = \frac{D}{2}\left(1 - LFO(n)\right)$$ Ecuación 4

$$LFO(n) = SD*\cos(2\pi n(f_{LFO}/fs))$$ Ecuación 5

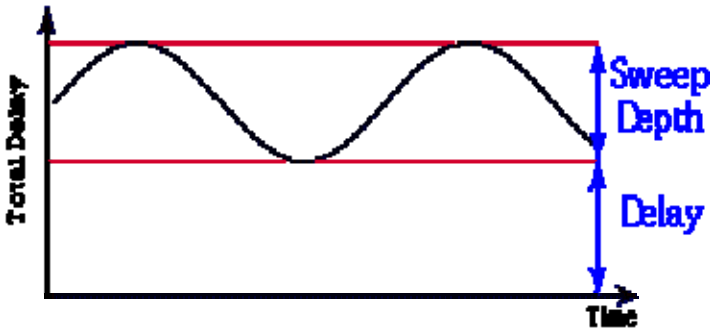
donde *D* es el retardo, *SD* es el *Sweep Depth* o amplitud, f_{LFO} es la frecuencia del LFO y *fs* es la frecuencia de muestreo.

Parámetros del efecto *flanger*

Retardo (*delay*). Especifica el mínimo retardo usado en la copia de la señal de entrada. Desde el punto de vista de la respuesta en frecuencia, este valor determina que tan ancha será la primera muesca. Al incrementar el retardo la primera muesca aparecerá más rápido. Además determina el tamaño del buffer que almacena las muestras previas de la señal de entrada en la línea de retardo.

***Sweep Depth*.** Determina el ancho del barrido en términos del retardo, en otras palabras la amplitud del LFO. Este parámetro es el máximo retardo adicional que junto con el retardo determina el máximo retardo posible, como se muestra en la figura 15. Un valor pequeño (cercano a 0) hará que la varianza en el tiempo del retardo sea pequeña, y un valor mayo (cercano a 1) causará que las muescas en la respuesta en frecuencia cubran una mayor área.

Figura 15. Relación entre retardo y *sweep depth*

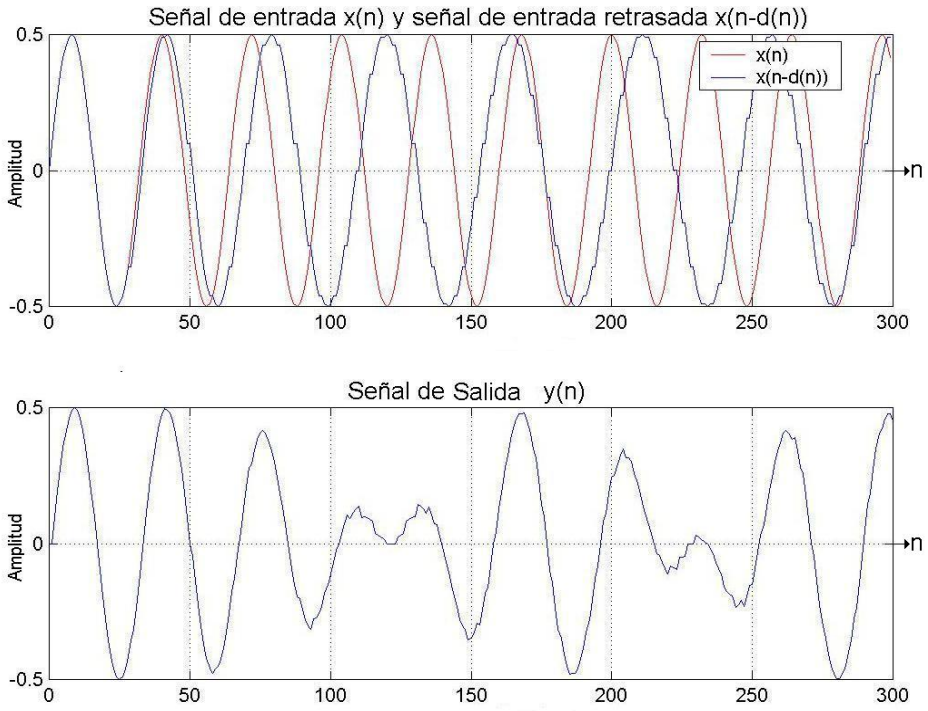


Fuente: *Harmony Central*

***Sweep Rate*.** Este parámetro permite cambiar la frecuencia del LFO.

En la figura 16 se muestra el efecto que produce el efecto *flanger* a una señal de entrada sinusoidal pura de 250 Hz con una frecuencia de muestreo de 8 kHz, utilizando un LFO sinusoidal con frecuencia de 10 Hz y un retardo de 10 ms.

Figura 16. Efecto aplicado a una *flanger* a una onda sinusoidal



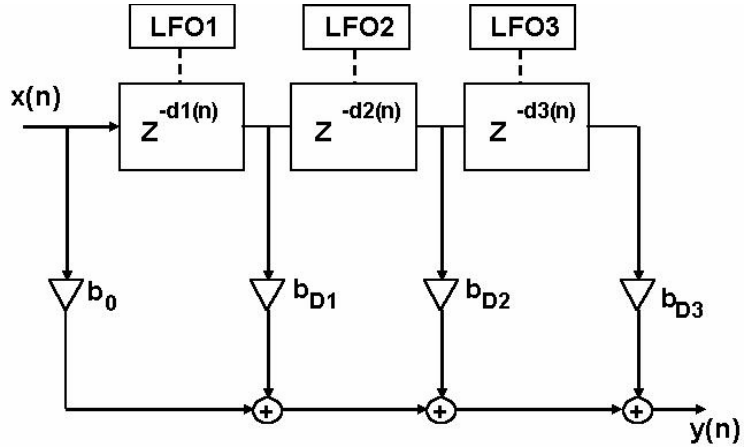
Fuente: Yasser Méndez. Autor del proyecto

Chorus. El *Chorus* consiste en simular al menos un segundo instrumento en coro con un primero. ¿Que sucede si dos personas tocan dos instrumentos juntos, o cantan juntos? No lograrán una precisa sincronización, y al sumarse los dos instrumentos se detectarán algunos retardos entre un instrumento y el otro, los cuales serán variables. Esto es justamente lo que simula el *chorus*, a partir de un solo instrumento o señal vocal. Este segundo instrumento (o voz) en coro, se genera con un algoritmo teóricamente sencillo, el mismo del *flanger,* y si quisiéramos agregar más instrumentos en coro solo habrá que agregar más etapas como ésta en paralelo, al algoritmo principal, [28]. Este algoritmo consiste en copiar la entrada a la salida y sumarle a ésta la entrada retardada. El retardo usualmente utilizado es mayor que el del *flanger,* de entre 20 ms y 30 ms. En realidad la falta de sincronismo entre los dos instrumentos (o voces), no tendrá un retardo constante entre sí, sino que será variable.

Para variar los retardos, en general, se utiliza una señal periódica, como por ejemplo un seno, así como en el *flanger*. También pueden ser utilizados una onda triangular, logarítmica o randómica. La frecuencia que se utiliza en estas señales no debe superar los 3 Hz para que de un buen efecto de coro, generalmente este efecto funciona mejor para frecuencias muy bajas, entre 0.1 Hz y 0.5 Hz. Para la generación del retardo variable se utiliza un LFO (Oscilador de Baja Frecuencia). En la

figura 17 se observa el modelo de bloques de ese algoritmo para un efecto *chorus* simulando cuatro voces o instrumentos, usando tres líneas de retardo en paralelo cada una modulada con su propio LFO.

Figura 17. Diagrama de bloques del *chorus* cuatro voces



Fuente: Yasser Méndez. Autor del proyecto

La ecuación 6 describe la situación del efecto *chorus* utilizando tres líneas de retardo controladas por LFOs diferentes.

$$y(n)=b_0\,x(n)+b_{D1}\,x(n-d_1(n))+b_{D2}\,x(n-d_2(n))+b_{D3}\,x(n-d_3(n)) \qquad \text{Ecuación 6}$$

Los retardos variables se calculan mediante la ecuación 7 y la ecuación 8.

$$d_k(n)=\frac{D}{2}\left(1+LFO_k(n)\right)\ ,\ k=1,\ 2,\ 3 \qquad \text{Ecuación 7}$$

$$LFO_k(n)=SD*\cos(2\pi n(f_{LFO_k}/fs)),\ k = 1, 2, 3 \qquad \text{Ecuación 8}$$

Se puede controlar el efecto del coro variando la frecuencia del LFO, la amplitud del LFO y su forma. Se puede utilizar una misma tabla de LFO pero desfasando esta para las otras líneas de retardo, o simplemente utilizando LFO diferentes para controlar el retardo en cada línea. También son posibles otras variaciones en el efecto coro. Por ejemplo, en lugar de usar un LFO, se puede usar un retardo cuya duración cambie en forma aleatoria, resultando útil para modelar músicos que cantan al unísono. También se puede incluir una línea de retroalimentación con un retardo constante o utilizar ganancias para las líneas de retardo que también varíen en el tiempo, mediante otro LFO.

Los parámetros del efecto *chorus* son los mismos del *flanger*, solo se adiciona la cantidad de instrumentos o voces, es decir cuantas réplicas, con retardo variable en el tiempo, de la señal de entrada se desean mezclar con la entrada original. Algunas unidades de *chorus* permiten seleccionar la cantidad de voces a usar. Generalmente, estas unidades utilizan un solo LFO para todas las voces, pero cada una tiene una fase diferente φ_k, como lo muestra la ecuación 9.

$$LFO_k(n)=SD*\cos(2\pi n(fLFO_k / fs)+\varphi_k),\ k = 1, 2, 3 \qquad \text{Ecuación 9}$$

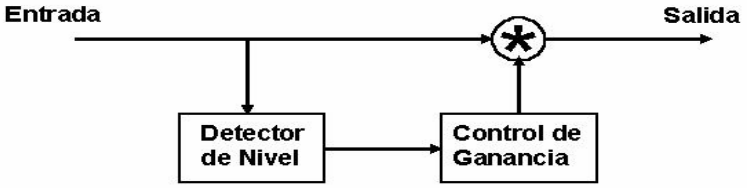
2.4.3 Efectos de audio basados en amplitud. Los efectos de audio basados en amplitud, simplemente involucran la manipulación del nivel de amplitud de la señal de audio, desde una atenuación o un incremento del volumen hasta efectos más sofisticados tales como compresión y expansión del rango dinámico , [35]. Los efectos que hacen parte de esta categoría son: Control de volumen*, Panning,* control de rango dinámico tales como *Compression, Limiting, Expansion* y *Noise Gating.*

Los algoritmos de procesamiento dinámico son usados para cambiar el rango dinámico de una señal. Este método altera la distancia en volumen entre el sonido más suave y el sonido más fuerte en una señal. Los dos tipos de algoritmos de procesamiento dinámico son: *compressors/limiters* y *expanders*.

Compresor/ Limitador. Los compresores y limitadores mantienen el nivel de una señal dentro de un rango dinámico específico, mediante una técnica llamada reducción de ganancia. Un circuito de reducción de ganancia reduce automáticamente una cantidad de ganancia adicional por encima del umbral fijado para una cierta relación de compresión. En la figura 18 se muestra el diagrama de un compresor.

El umbral o *threshold* es el nivel de la señal a la cual empieza la reducción de ganancia. La ganancia por encima de un umbral sobre la ganancia por de bajo de ese umbral se conoce como relación de compresión o *ratio*. Por ejemplo, una relación de compresión de 2:1 reduce la señal por un factor de dos cuando pasa el nivel del umbral, mientras que una relación de 20:1 reduce la señal en un factor de 20, como se observa en el ejemplo de compresor de la figura 19. Con una relación de compresión 2:1 los valores por encima del umbral 0.5 se reducen pero aún pasan; mientras que con una relación de compresión mayor, 20:1, prácticamente se limita la señal al valor umbral. El objetivo es mantener el nivel de la señal cuando se pasa de un nivel especificado. Esto puede ser usado para efectos musicales, para hacer que un sonido sobresalga en una mezcla compleja, o para propósitos de reducción de ruido en sistemas de transmisión y grabación ruidosos.

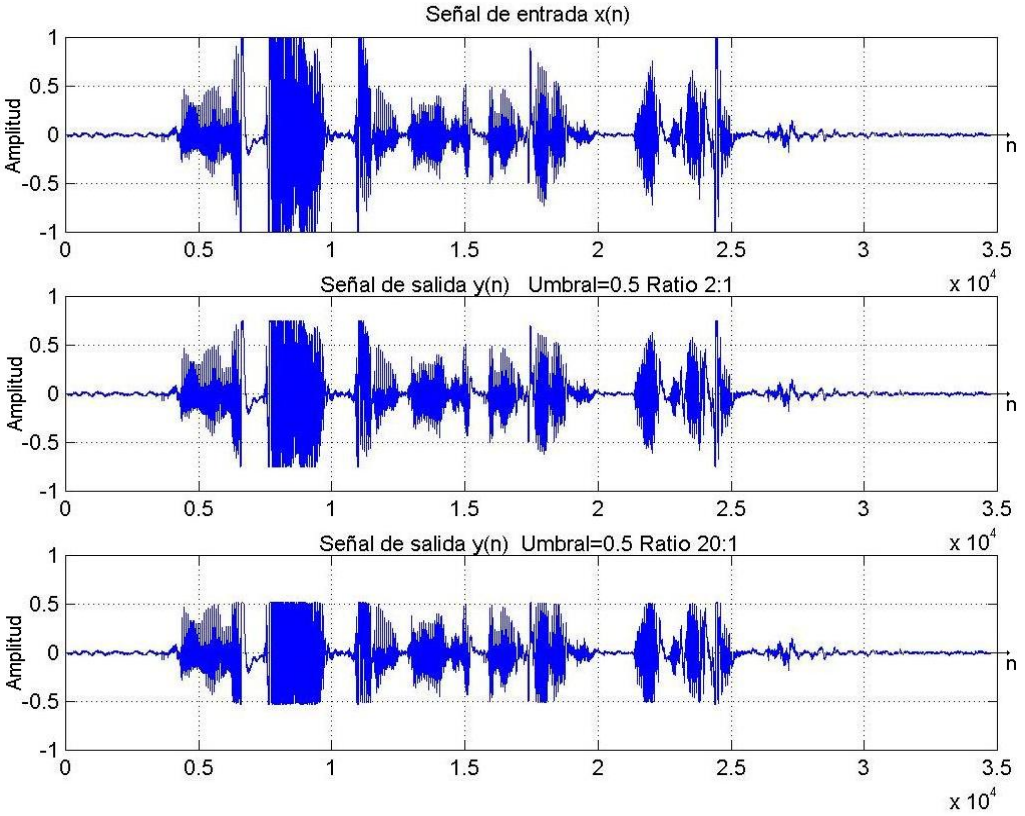
Figura 18. Diagrama de bloques del compresor



Fuente: Yasser Méndez. Autor del proyecto

La compresión puede ser empleada en alguna etapa del proceso de grabación, aunque puede sonar bastante diferente dependiendo en cual parte del proceso es usada. Esto es usualmente deseable para comprimir señales antes de ser grabadas, especialmente si la grabadora es ruidosa, puesto que la señal comprimida no contendrá ruido comprimido como parte de la señal [35].

Figura 19. Efecto compresor aplicado a una señal de voz



Fuente: Yasser Méndez. Autor del proyecto

La relación de compresión (*ratio*) se puede expresar en dB de tal forma que, en una compresión 3:1, por cada 3 dB de señal que sobrepasa el umbral (*threshold*), sólo se amplifica en 1 dB. Por tanto, cuanto mayor es la relación de compresión, más se limita la señal que sobrepasa el umbral.

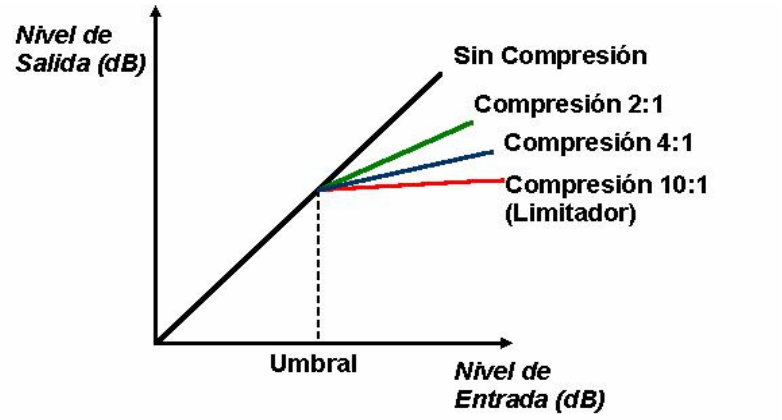
El método de compresión usa relaciones de compresión bajas como por ejemplo 2:1 o 4: 1. Si la relación es de 10:1 o mayor el método se denomina limitador, [35]. En la figura 20 se muestra como cambia la pendiente de la gráfica que relaciona la ganancia de la señal de entrada y la ganancia de la señal de salida del compresor, para diferentes relaciones de compresión.

Como se ha mencionado antes el umbral (*threshold*) es el nivel a partir del cual la señal se considera demasiado alta, y por tanto debe ser limitada por el compresor. La relación de compresión (*ratio*) es el factor que se aplica para reducir la ganancia de la señal de entrada cuando esta supera el nivel de *threshold* ajustado por el usuario. El *threshold* y *ratio* son los

parámetros más importantes del compresor, pero no los únicos. El ataque (*attack*) y el *release*, junto con el modo de detección de la señal de entrada, son también parámetros que manejan muchos compresores, implementados en software o hardware.

El ataque (*attack*) es el tiempo que tarda el compresor en empezar a atenuar la señal que ha sobrepasado el nivel de *threshold*. Con un ataque rápido (o tiempo de ataque corto) la señal es limitada inmediatamente, mientras que un ataque lento permite una transición entre la señal original y su atenuación. Por ejemplo, es muy común utilizar un ataque rápido de unos pocos ms (milisegundos) para obtener sonidos más percusivos, especialmente en guitarras e instrumentos de percusión y hace las voces más claras y comprensibles.

Figura 20. Relación de compresión



Fuente: Yasser Méndez. Autor del proyecto

Al contrario que el ataque, el *release* es lo que tarda el compresor en dejar de aplicar la limitación de ganancia y recupera el nivel original de la señal, una vez estamos por debajo del nivel de *threshold*. Un tiempo de *release* largo podría hacer que la señal no hubiera terminado de recuperar su volumen original cuando ocurriese el siguiente salto por encima de *threshold*, muy indicado en señales con muchos altibajos de volumen, para hacer una señal más constante. Normalmente un tiempo de *release* corto es poco recomendable ya que no se evitan estos altibajos.

Todo compresor dispone de un circuito o algoritmo que detecta la amplitud de la señal para saber cuando debe aplicar alguna modificación sobre ella. El detector de nivel RMS pone menos atención a los sonidos cortos y fuertes y detecta mejor sonidos continuos del mismo nivel. Este circuito tiene un funcionamiento muy similar al del oído humano, por tanto ofrece unos resultados más naturales, pero tiende a ignorar picos cortos, como los que pueden ocurrir en grabaciones digitales y que hay que evitar a toda costa.

El modo de funcionamiento *peak*, en cambio, es capaz de trabajar con zonas de la señal muy cortas y proporciona mayor control sobre la señal. Es más indicado que RMS para trabajar con sonidos percusivos, pistas sueltas de percusión y *clipping*, es decir, picos producidos por ruido en grabaciones digitales y que podremos eliminar utilizando el compresor como limitador trabajando en modo *peak*.

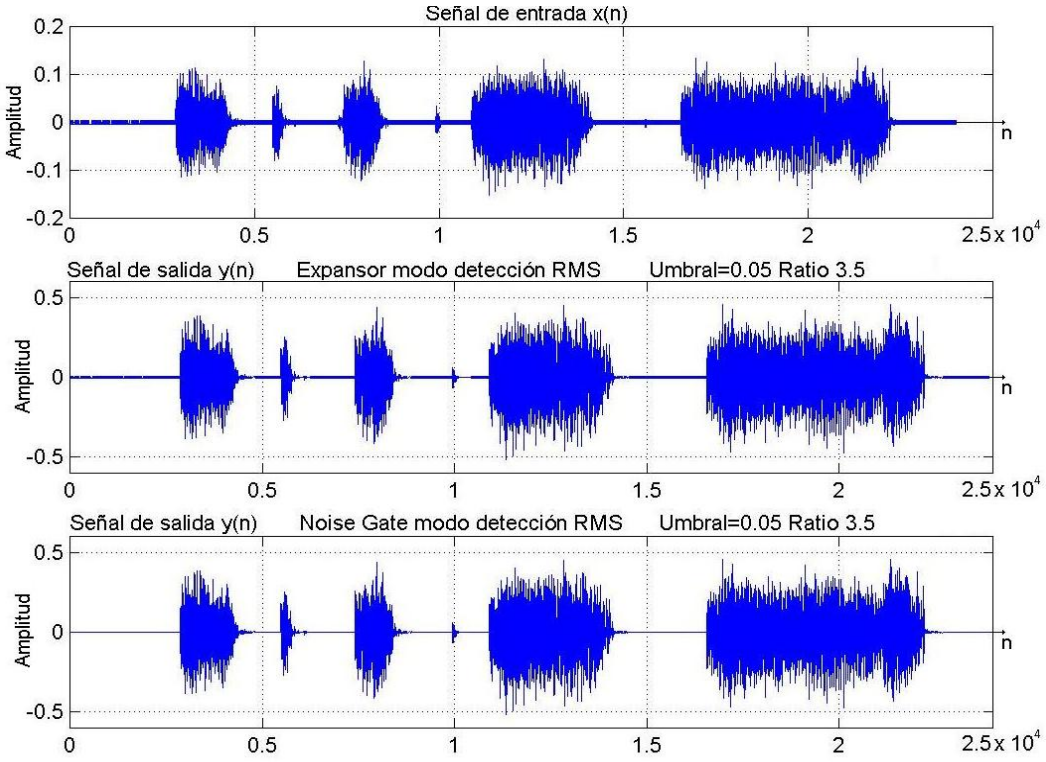
Expansor. La expansión es el proceso complementario de la compresión, es decir incrementa el rango dinámico de la señal. Cuando la señal cae por debajo de un umbral (*threshold*), la ganancia es reducida, haciendo que el sonido sea muy suave. Por ejemplo, una señal con un rango dinámico de 70 dB quizá pase a través de un expansor y salga con un nuevo rango dinámico de 100 dB. Esto puede ser usado para restaurar una señal que haya pasado por un compresor.

Cuando la ganancia se reduce tanto que el sonido se convierte en silencio o inaudible el efecto se denomina *Noise Gate* [35]. El expansor y el *Noise Gate* se utilizan combinados para no expandir el ruido de fondo de una señal.

Los parámetros del expansor (*expander*) son los mismos del compresor (*compressor*) pero con la diferencia que existe una relación compresión (entre 0 y 1) para las señales por debajo del umbral, reduciendo así la ganancia de la señal; y una relación de expansión (mayor a 1) para las señales que están por encima del umbral, aumentando la ganancia la señal. En el caso del *Noise Gate* no se requiere una relación de compresión que reduzca la ganancia de la señal si está por debajo del umbral, simplemente se hace cero.

En la figura 21 se muestra una señal de entrada afectada por un expansor y un *Noise Gate* para un umbral de 0.05, una relación de expansión de 3.5 y modo de detección RMS. La señal de entrada contiene un bajo nivel de ruido, el cual es reducido por el expansor y totalmente eliminado por el *Noise Gate*.

Figura 21. Efecto expansor y *noise gate* aplicado a una seña de voz

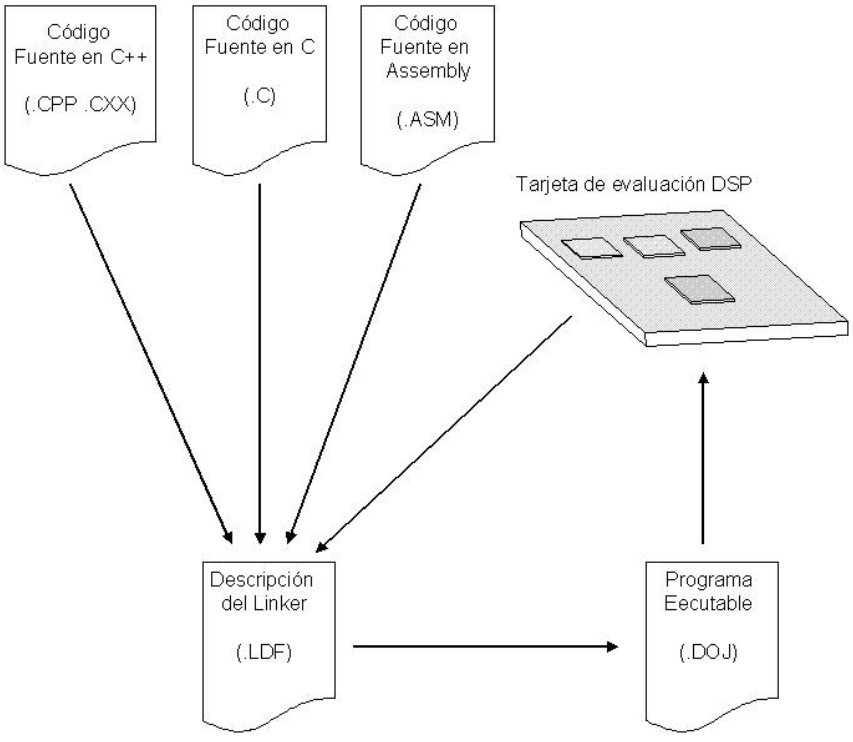


Fuente: Yasser Méndez. Autor del proyecto

3. IMPLEMENTACIÓN DE LOS ALGORITMOS EN EL DSP

Antes de implementar los diferentes algoritmos de efectos de audio y los filtros en la tarjeta de desarrollo, se simula cada uno de los algoritmos utilizando el simulador del DSP (ADSP-2106x Simulator). El resultado de esta simulación se compara con el resultado del algoritmo en Matlab empleando la misma entrada. Para la fase de simulación, se crea un proyecto DPJ que contiene dos archivos, un archivo LDF (*Linker Description Files*) donde se especifican los segmentos de memoria (datos y programa) y el tipo de procesador que se usa (ADSP-21065L) y un archivo fuente (.*C* o *asm*.) que contiene el código del algoritmo específico escrito en *C* o en *Assembly*. Los algoritmos descritos a continuación se implementan en lenguaje Assembly. El archivo LDF es muy importante pues es el que interpreta el código del archivo fuente compilado, es decir el archivo DOJ y genera el archivo ejecutable DXE el cual se carga en el *Debugge*r para finalmente ser ejecutado por este [23], como se muestra en la figura 22.

Figura 22. Archivo LDF y proceso de enlazado

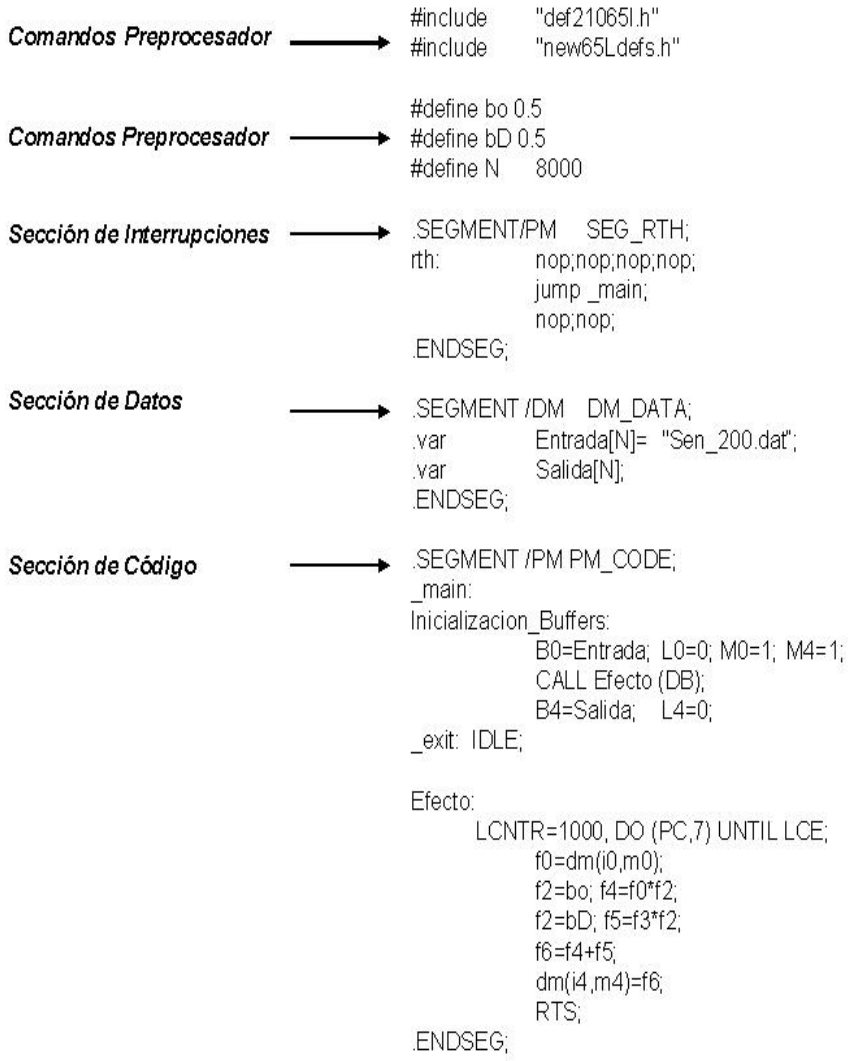


Fuente: ANALOG DEVICES. Linker and Utilities Manual for ADSP-21XXX Family DSP´s

El archivo fuente escrito en lenguaje Assembly está dividido básicamente en seis partes como se muestra en la figura 23. La primera sección es la del comando directo del preprocesador *#include*, que inserta un archivo *header*, el cual puede ser del sistema o creado por el usuario [22]. Estos

incluyen todos los archivos necesarios para que funcione adecuadamente el programa, tales como definición de registros del procesador, uso de interrupciones, librerías matemáticas, etc. La segunda sección corresponde al comando de preprocesador *#define*, que define macros [22]. En el caso particular de los algoritmos que se tratan aquí, se usa *#define* para definir los valores de los parámetros de los diferentes efectos de audio, cantidad de muestras de entrada, tamaño de los *buffers* circulares para las líneas de retardo y los coeficientes de las ecuaciones en diferencias. La tercera sección es el vector de interrupciones necesario para inicializar el programa en el DSP y las interrupciones requeridas. La cuarta sección es la de datos en la memoria de datos (DM) donde se ubican las variables que necesita el programa. La quinta sección es la de datos en la memoria de programa (PM) donde también se ubican variables y es muy útil cuando se implementan los filtros, pues permite usar toda la capacitada de procesamiento del DSP. Finalmente, la sexta sección corresponde al código de programa donde se inicializan los buffers y se desallora el algoritmo particular para cada efecto de audio o filtro digital.

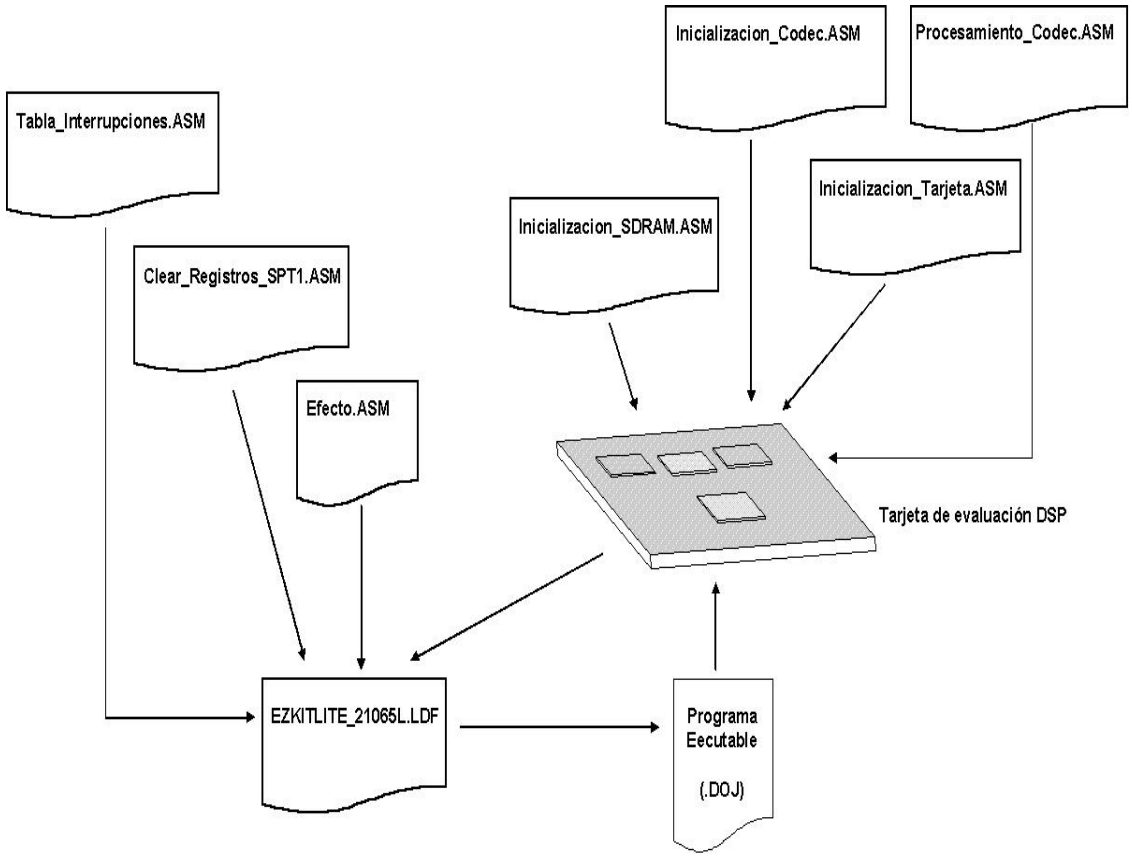
Figura 23. Estructura del programa en Assembly



Fuente: YASSER MÉNDEZ. Autor del proyecto

La implementación de estos algoritmos en tiempo real, empleando la tarjeta de evaluación ADSP-21065L EZ-KIT Lite requiere de la creación de un proyecto DPJ que contiene un archivo LDF (*Linker Description Files*) para ser empleado cuando se usa la tarjeta de evaluación, un archivo de inicialización de la tarjeta, un archivo de inicialización del codec, un archivo de inicialización de la memoria externa, un archivo que contiene la tabla de interrupciones, un archivo de limpieza de los registros del puerto serial, un archivo que contiene la rutina de servicio de interrupciones del codec y finalmente un archivo principal con el código del algoritmo en particular como se muestra en la figura 24.

Figura 24. Proyecto DPJ para tiempo real

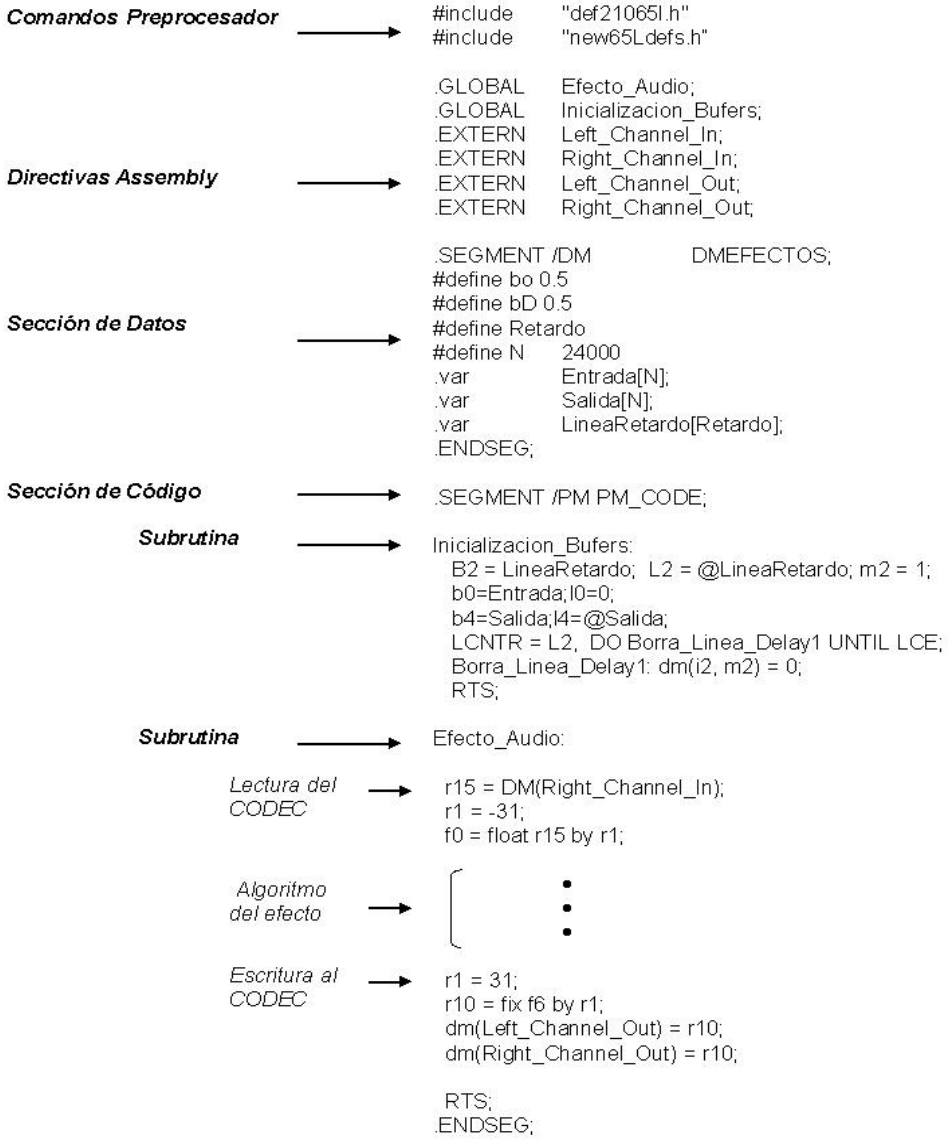


Fuente: YASSER MÉNDEZ. Autor del proyecto

El archivo principal está compuesto de seis partes como se muestra en la figura 25. La primera corresponde a los *#include*, que contiene los archivos *header* con la definición de los bits y direcciones de los registros IOP y del sistema para el ADSP-21065L. La segunda parte es la directiva *assembly .GLOBAL* que convierte un símbolo local en uno global [22], habilitándolo para ser referenciado en otros archivos, y en este caso corresponde a las subrutinas del efecto que se implementa y la inicialización de los *buffers*. La tercera parte es la directiva *assembly .EXTERN* que permite hacer referencia a un símbolo global [22], y corresponde a las variables que almacenan las muestras de entrada y salida de los dos canales del codec. La cuarta parte es la

sección de datos que contiene el comando de preprocesador *#define* que define los coeficientes de la ecuación en diferencias y el tamaño de los buffers circulares correspondientes a la líneas de retardos requeridas por el efecto; y se definen los buffers de entrada, salida y líneas de retardo, y otras variables o propias de cada efecto en particular. La quinta parte es la sección de datos en la memoria de programa que solo es utilizada en la mplementación de filtros FIR e IIR. Finalmente la sexta parte es el segmento de código de programa que contiene la subrutina de inicialización de los *buffers* circulares y la subrutina del efecto que se implementa que incluye la lectura de muestra de entrada desde el *Codec*, el código del efecto en particular y la escritura de la muestra de salida al *Codec*. El código completo para cada uno de los algoritmos que se implementan se muestra en los anexos.

Figura 25. Estructura del programa en Assembly para tiempo real

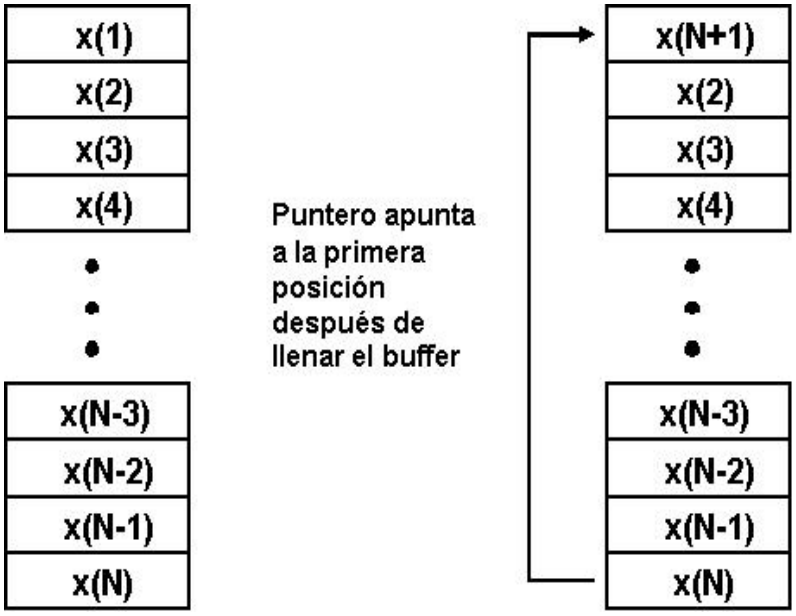


Fuente: YASSER MÉNDEZ. Autor del proyecto

3.1 IMPLEMENTACIÓN DE EFECTOS DE AUDIO BASADOS EN RETARDOS DIGITALES

La implementación digital de retardos digitales es relativamente simple. Se debe contar con una cierta cantidad de memoria para almacenar muestras de la señal de entrada. En cada periodo de muestreo se lee una muestra previamente almacenada y se escribe la muestra a una posición de memoria determinada. En el siguiente periodo de muestreo, se lee y escribe a la siguiente posición en memoria; y cuando se llega al final de la memoria, se debe apuntar a la primera posición de la misma como se muestra en la figura 26, sobrescribiendo la muestra más vieja por la muestra más reciente. Esto en procesamiento de señales se conoce como *buffer* circular. Es así como, para obtener un retardo con una sola reflexión o eco, se almacenan muestras de la señal de entrada en una línea de retardo, mediante un *buffer* circular. Si se desean tener más reflexiones se pueden agregar punteros a difrentes posiciones de memoria o más buffer circulares para las nuevas líneas de retardo, requiriendo esto de un mayor consumo de memoria.

Figura 26. Buffer circular de longitud N



Fuente: YASSER MÉNDEZ. Autor del proyecto

3.1.1 Implementación Retardo Simple. El retardo de un solo tap o retardo simple se define con la ecuación 10, simplemente se suma a la señal de entrada, una versión retrasada de la misma. Esta versión retardada de la señal de entrada, es la que se lee de la línea de retardo. El tamaño del *buffer* circular y la frecuencia de muestreo permiten determinar el tiempo del retardo que se quiere efectuar de acuerdo a la ecuación 11.

$$y(n) = b_0 * x(n) + b_D * x(n - D) \qquad \text{Ecuación 10}$$

$$t_{retardo} = \frac{\text{Re}\,tardo}{Fs}$$ Ecuación 11

Donde *Retardo* es el tamaño del *buffer* circular y *Fs* es la frecuencia de muestreo en Hz. Así para un retardo típico de 100 ms y una frecuencia de muestreo de 8000 Hz, se requiere de un *buffer* circular que almacene 800 muestras.

La implementación del buffer circular en el DSP se hace mediante el uso de los generadores automáticos de direcciones (DAGs) de datos que automáticamente generan e incrementan punteros para acceder memoria. Cada DAG usa cuatro registros: Base (B) que almacena la primera dirección del buffer circular, Index (I) que actúa como un puntero a memoria, Modificador (M) que contiene el valor para incrementar el puntero y Longitud (L) que define la longitud del buffer circular [16].

• **Simulación.** En la programación del efecto de retardo de un tap primero se definen los parámetros del retardo simple: los coeficientes de la ecuación en diferencias, bo y b_D; y el tamaño de la linea de retardo, Retardo (D). Luego se definen la cantidad de muestras procesadas, N; los buffers que almacenan las muetras de entrada y salida, Entrada[N] y Salida[N] respectivamente; y el buffer de la línea de retardo el cual es un buffer circular implementado con el DAG1, donde el registro base (B2) lleva el mismo nombre de la línea de retardo, el registro modificador (M2) es uno, el registro longitud (L3) es el tamaño del buffer circular determinado por la ecuación 11 y el registro Index (I2) se usa para apuntar a la línea de retardo.

En la tabla 3 se muestran los diferentes registros usados para inicializar el buffer circular de la línea de retardo y los buffers no circulares que almacenan la muestras de entrada y de salida.

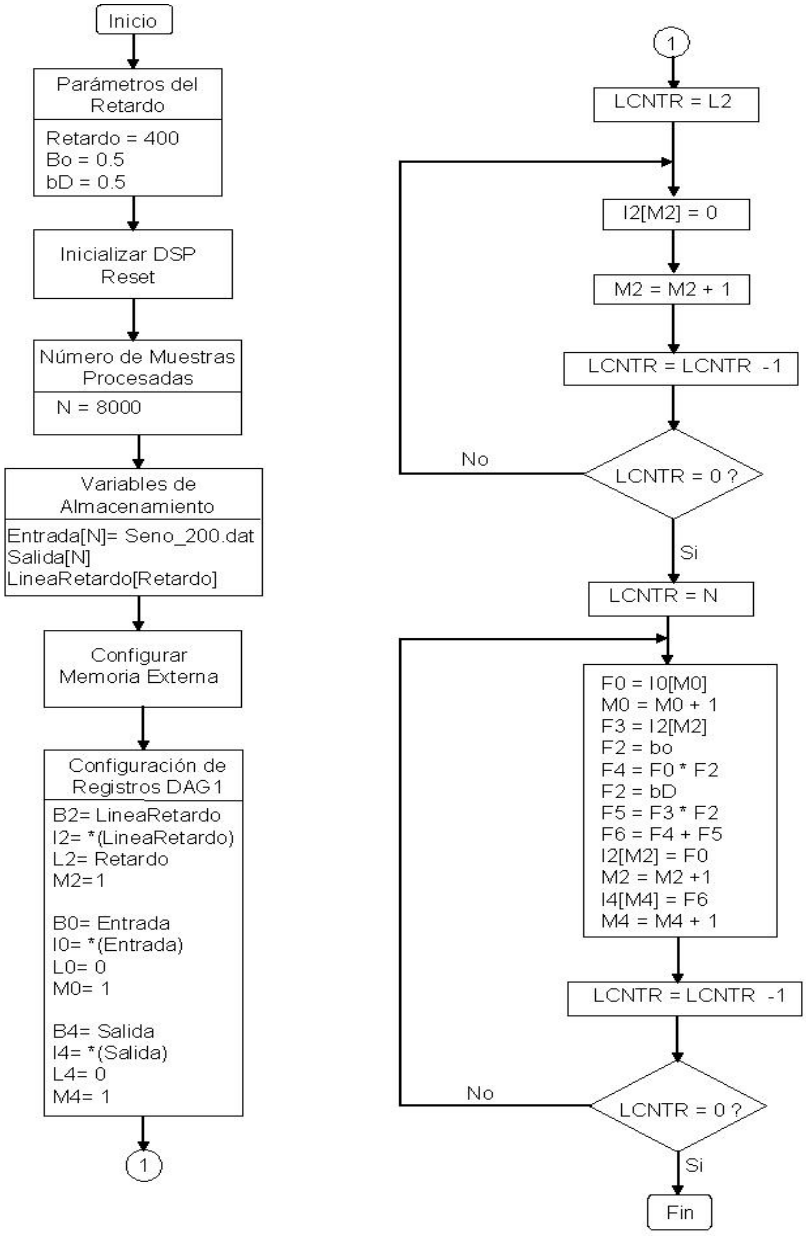
Tabla 3. Registros del DAG1 que inicializan los buffers del algoritmo de retardo simple

Registro Base	Registro Index	Registro Longitud	Registro Modificador	Nombre del buffer	Descripción
B2	I2	L2=Retardo	M2=1	LineaRetardo	Circular
B0	I0	L0=0	M0=1	Entrada	No circular
B4	I4	L4=0	M4=1	Salida	No circular

Fuente: YASSER MÉNDEZ. Autor del proyecto

En la figura 27 se muestra el diagrama de bloques del algoritmo de retardo simple donde se aprecia que primero se lee de la línea de retardo la muestra almacenada previamente, sin actualizar el puntero, es decir con un incremento nulo (M2=0) y luego si se escribe en la línea de retardo la muestra actual, actualizando el puntero, es decir con incremento uno (M2=1). El programa completo para la simulación del retardo simple "Retardo.asm" junto con el archivo *linker* "Retardo.ldf" se muestran en los anexos.

Figura 27. Diagrama de bloques algoritmo simulación retardo simple

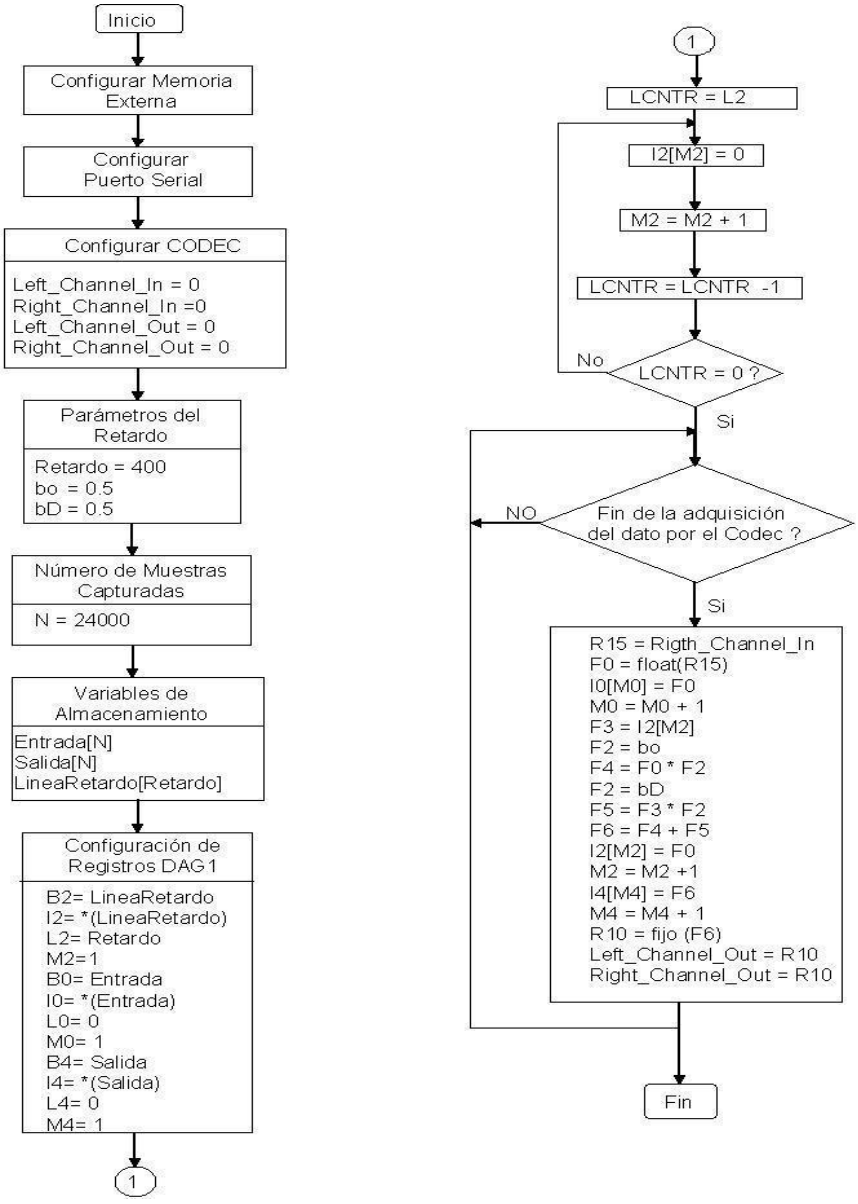


Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Implementación en tiempo real.** La subrutina del efecto retardo simple en tiempo real es prácticamente la misma subrutina de la simulación; difieren en que, se cambia el lazo de control por la interrupción del Codec; la muestra de entrada ya no se obtiene de un buffer almacenado en memoria sino que se obtiene desde el Codec; se realiza su conversión de punto fijo a punto flotante y se almacena en un buffer de entrada; y la muestra de salida es convertida de punto flotante a punto fijo para luego ser sacada por el Codec. Además se deben definir las variables de lectura (*Left_Channel_In* y *Right_Channel_In*) y escritura (*Left_Channel_Out* y

Right_Channel_Out) para cada uno de los canales del Codec, mediante la directiva en *assembly ".EXTERN"*. La definición de buffers y su inicialización es la misma de la simulación y mostrasdos en la tabla 3. El diagrama de bloques del algoritmo de retardo simple en tiempo real se muestra en la figura 28. El código del programa de la implementación en tiempo real del retardo simple, el archivo "RetardoSimple.asm" y los otros archivos necesarios para correr el programa en la tarjeta de evaluación se muestra en los anexos.

Figura 28. Diagrama de bloques algoritmo en tiempo real retardo simple



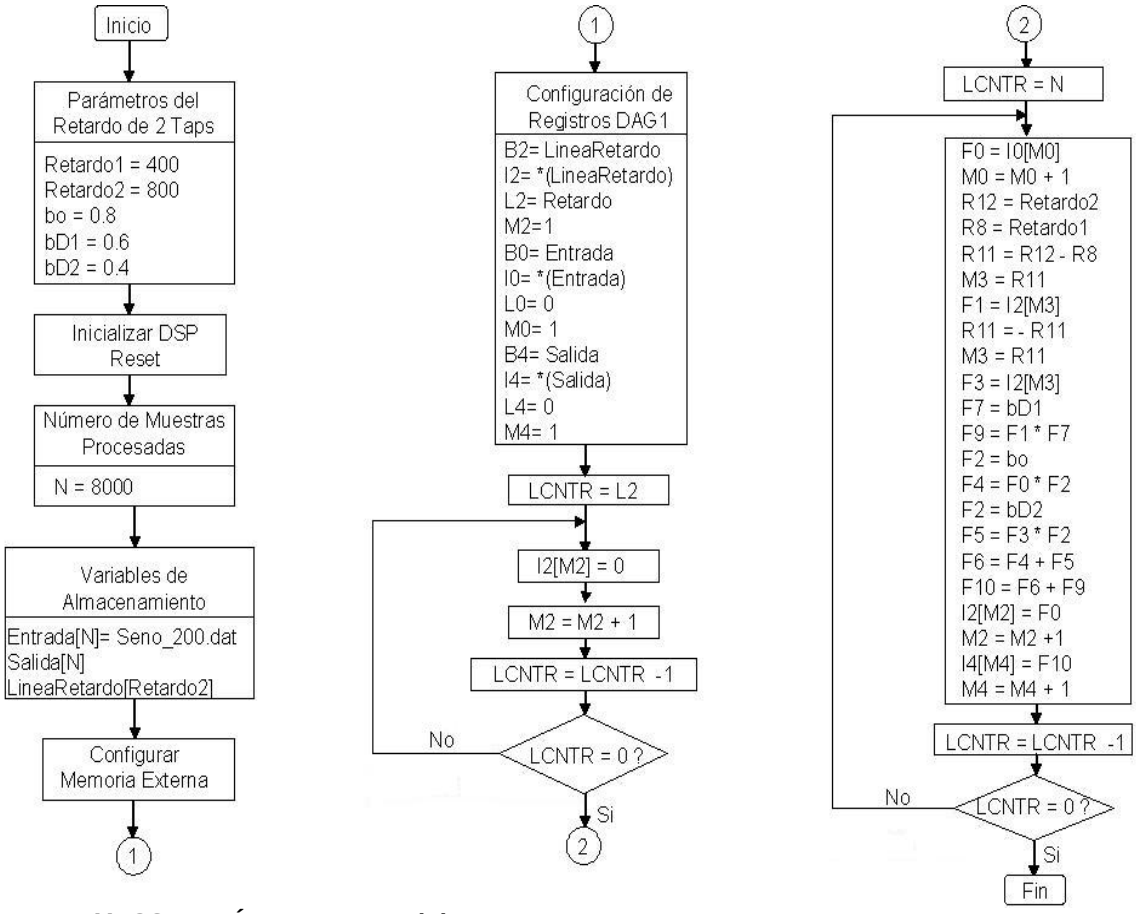
Fuente: YASSER MÉNDEZ. Autor del proyecto

3.1.2 Implementación Retardo Múltiple. Si se desea implementar un efecto con múltiples retardos se pueden usar varias líneas de retardo, mediante buffer circulares separados de diferentes longitudes, de acuerdo al tiempo de retardo que se requiera, o utilizando diferentes punteros de lectura para acceder a cada uno de los *taps* de retardo. La ecuación 12 representa un retardo con dos taps, el cual es primero implementado usando una sola línea de retardo y luego dos líneas de retardo con buffers diferentes.

$$y(n) = b_0 * x(n) + b_{D1} * x(n - D_1) + b_{D2} * x(n - D_2) \qquad \text{Ecuación 12}$$

• **Simulación.** Partiendo del programa de retardo simple se puede implementar un retardo con dos *taps* usando la misma línea de retardo, agregando un nuevo modificador de puntero y 14 nuevas líneas de programa. La longitud del buffer de la línea de retardo corresponde al mayor retardo. El incremento de puntero para el retardo más grande es de uno, mientras que el incremento del puntero de menor retardo corresponde a la diferencia entre el retardo mayor y el retardo menor. En la figura 29 se muestra el diagrama de bloques para el retardo de dos *taps* con una sola línea de retardo, pero empleando dos modificadores de puntero diferentes, uno para cada retardo.

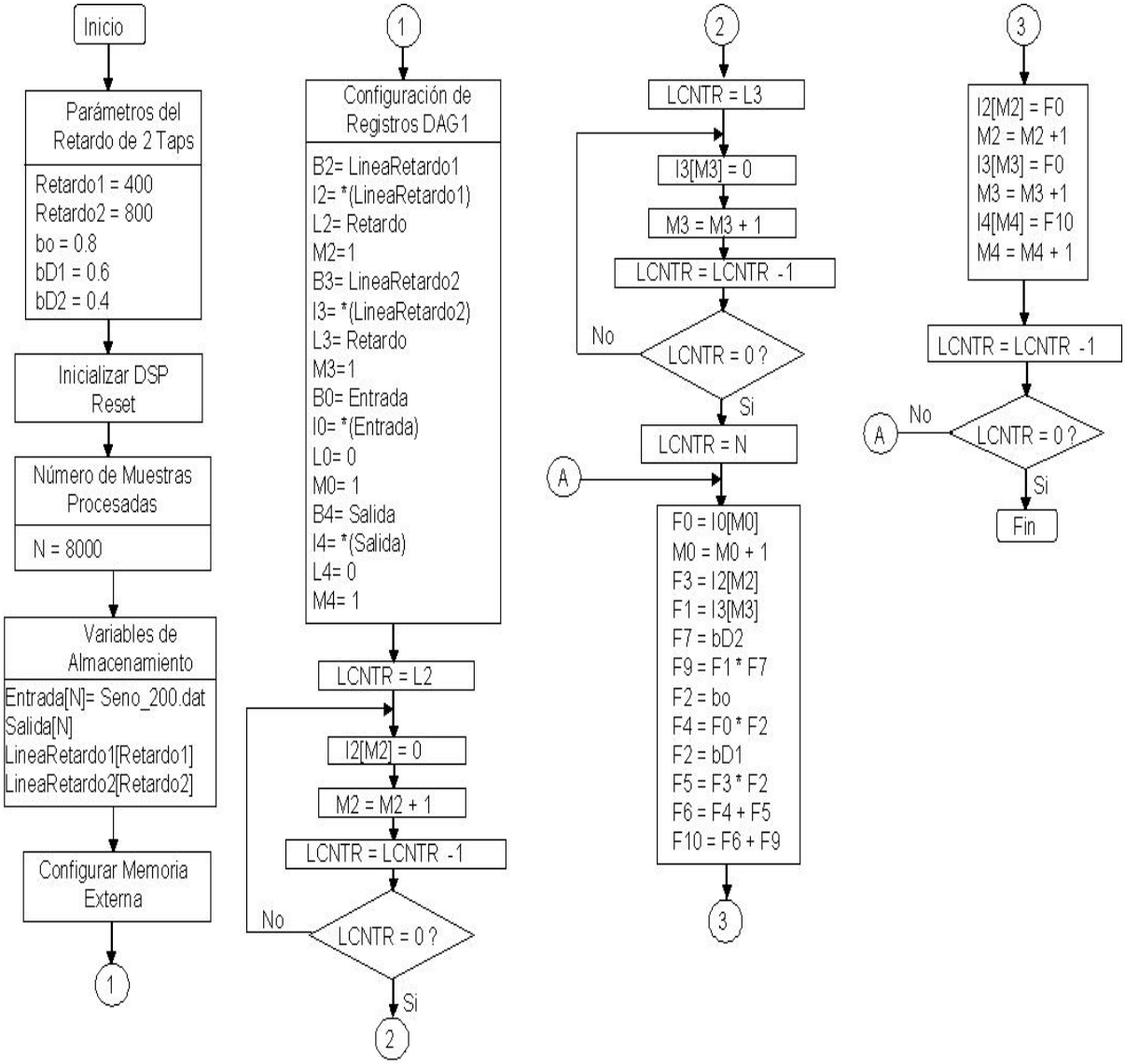
Figura 29. Diagrama de bloques algoritmo simulación retardo 2 taps y una sola línea de retardo



Fuente: YASSER MÉNDEZ. Autor del proyecto

También se puede obtener un efecto de retardo digital de dos *taps* empleando dos líneas de retardo diferentes, una para cada retardo. Se requiere de más espacio en memoria, pues hay que crear un segundo buffer circular que almacene las muestras de entrada para el nuevo retardo. Sin embargo el gasto computacional es menor, ya que el lazo de control de flujo realiza solo 14 instrucciones, mientras que el algoritmo de dos *taps* con una sola línea de retardo emplea 21 instrucciones. Se requiere definir un nuevo buffer circular para la segunda línea de retardo. En la la figura 30 se muestra el diagrama de bloques para la simulación de un retardo de dos taps con dos líneas de retardo. El programa completo del retardo de dos taps con dos lineas de retardo "Retardo_2Taps.asm" y su correspondiente archivo *linker* "Retardo_2Taps.ldf" se muestra en los anexos.

Figura 30. Diagrama de bloques algoritmo simulación retardo 2 taps y dos líneas de retardo



Fuente: YASSER MÉNDEZ. Autor del proyecto

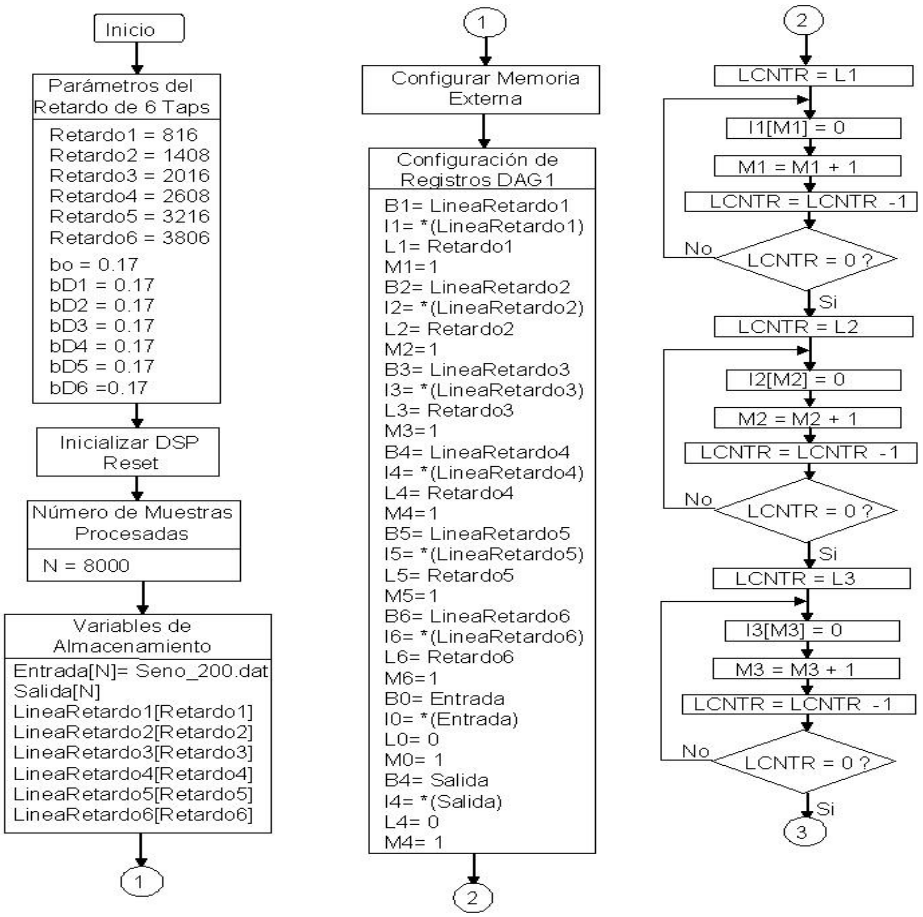
Usando esta técnica se propone implementar un algoritmo de Retardo Múltiple de 6 Taps descrito por la ecuación 13.

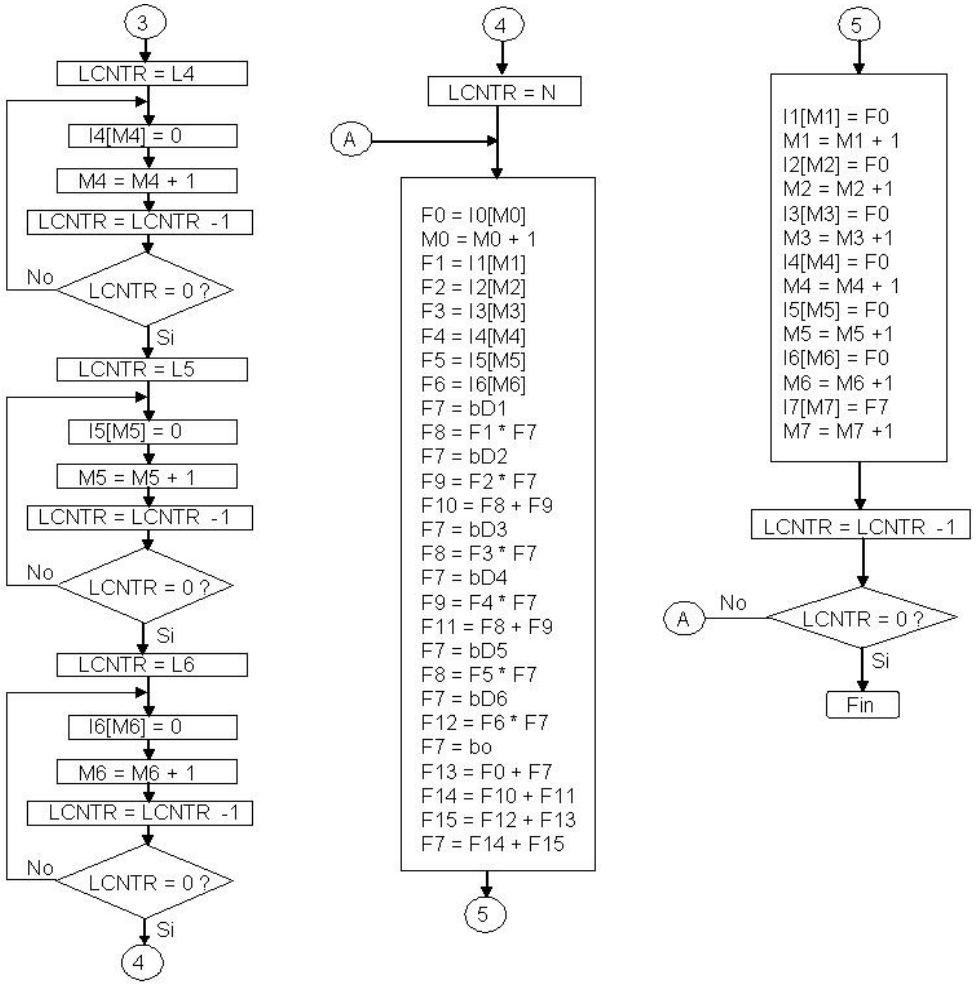
$$y(n) = b_0 * x(n) + b_{D1} * x(n - D_1) + b_{D2} * x(n - D_2) + b_{D3} * x(n - D_3) + b_{D4} * x(n - D_4) + b_{D5} * x(n - D_5) + b_{D6} * x(n - D_6)$$

Ecuación 13

Se usan seis líneas de retardo mediante la totalidad de los registros del generador automático de direcciones $DAG1_{0\text{-}7}$ habilitados en la memoria de datos (DM), donde se trabaja con los registros B0, I0 y M0 para definir el buffer de entrada, y los registros B7, I7 y M7 para definir el buffer de salida; los buffers de las líneas de retardo se definen con los registros restantes $DAG1_{1\text{-}6}$. La tabla 4 muestra los difrentes registros del DAG1 utilizado s para inicializar los buffers. En la figura 31 se muestra el diagrama de bloques para el algoritmo de simulación de retardo múltiple con seis líneas de reatardo. El programa completo "Retardo_6Tpas.asm" y el archivo *linker* "Retardo_6Taps.ldf" se muestra en los anexos.

Figura 31. Diagrama de bloques algoritmo simulación retardo seis taps y seis líneas de retardo





Fuente: YASSER MÉNDEZ. Autor del proyecto

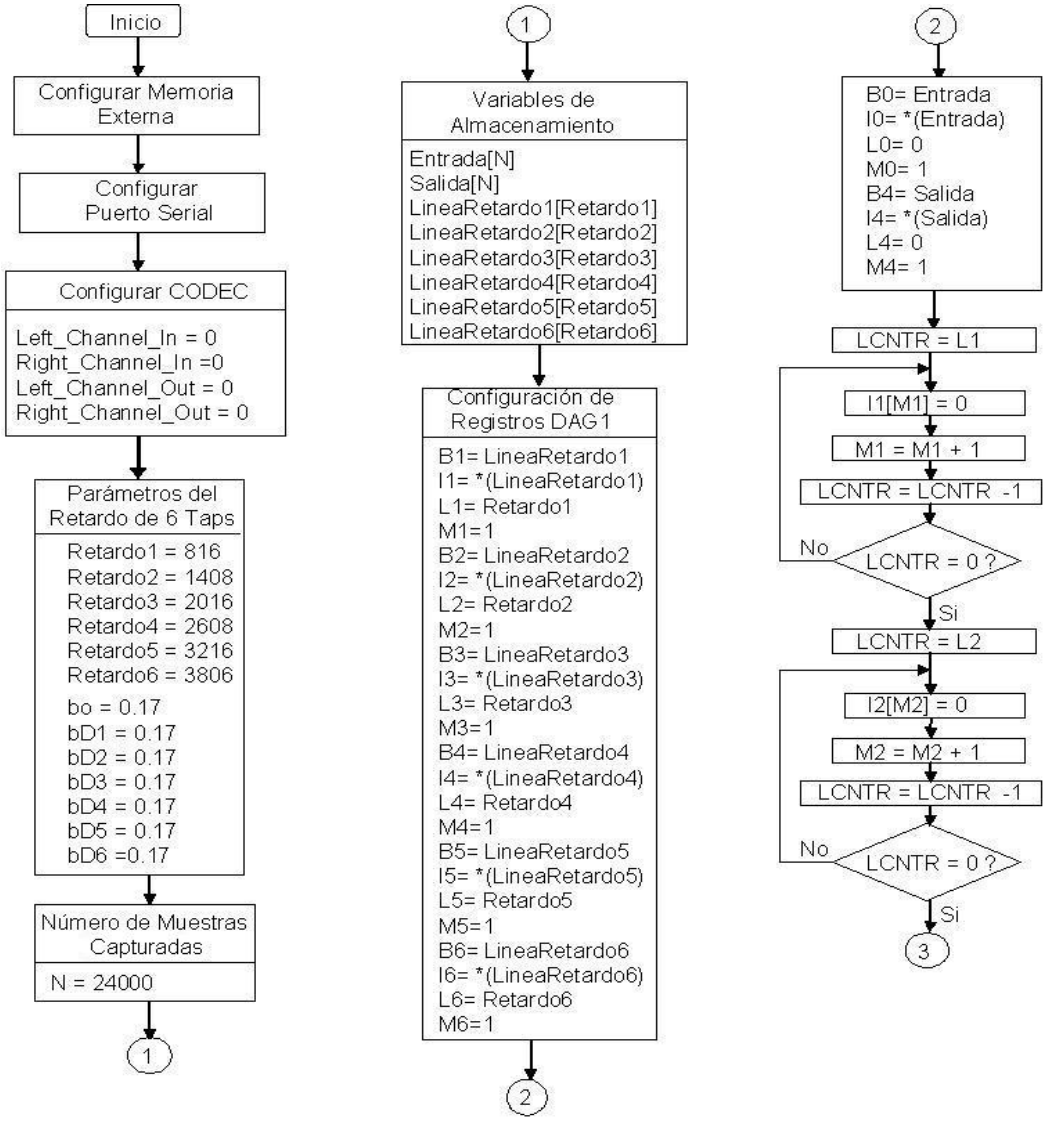
Tabla 4. Registros del DAG1 que inicializan los buffers del algoritmo de retardo múltiple de seis taps y seis líneas de retardo

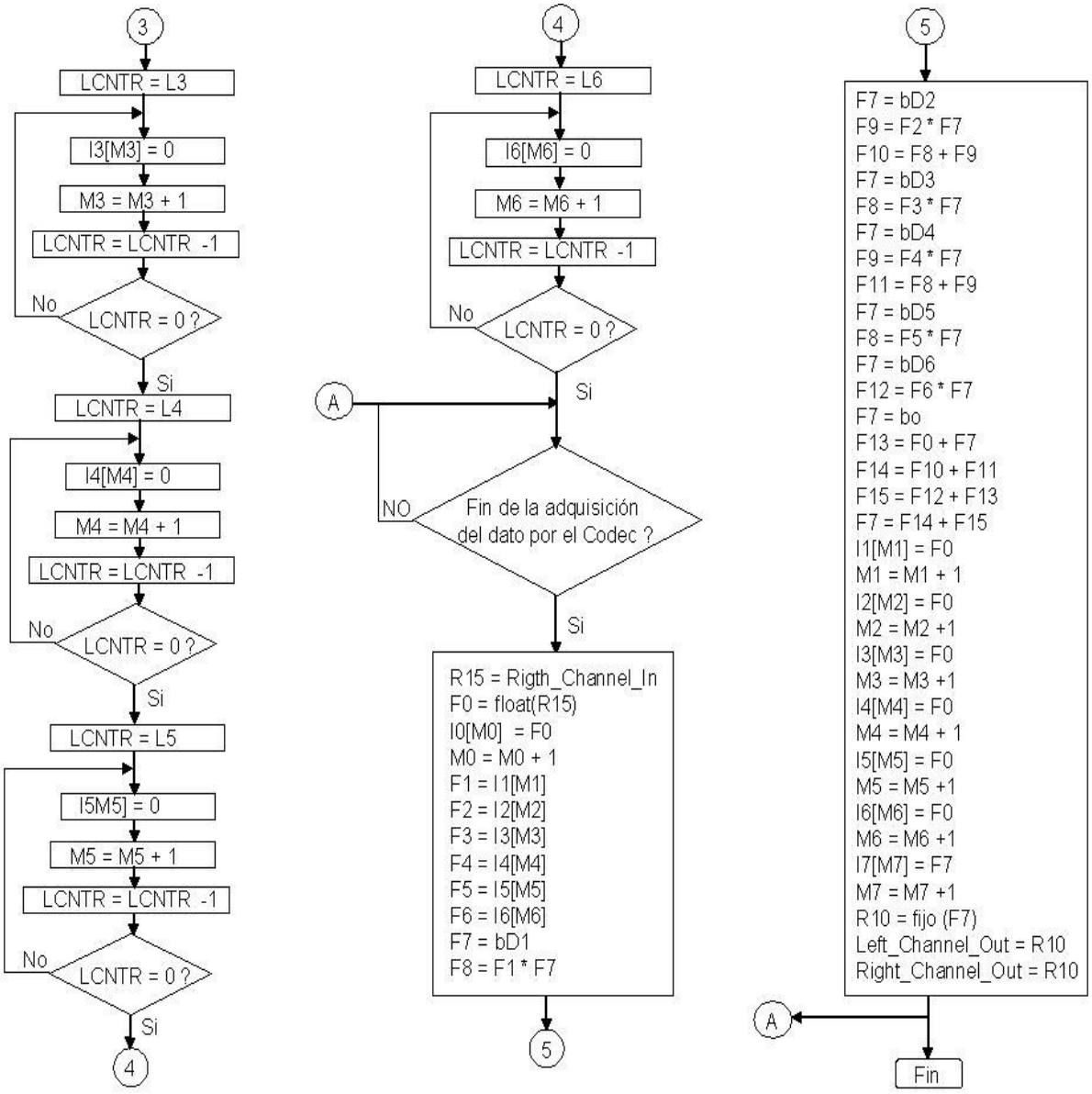
Registro Base	Registro Index	Registro Longitud	Registro Modificador	Nombre del buffer	Descripción
B1	I1	L1=Retardo1	M1=1	LineaRetardo1	Circular
B2	I2	L2=Retardo2	M2=1	LineaRetardo	Circular
B3	I3	L3=Retardo3	M3=1	LineaRetardo	Circular
B4	I4	L4=Retardo4	M4=1	LineaRetardo	Circular
B5	I5	L5=Retardo5	M5=1	LineaRetardo	Circular
B6	I6	L6=Retardo6	M6=1	LineaRetardo	Circular
B0	I0	L0=0	M0=1	Entrada	No circular
B7	I7	L7=0	M7=1	Salida	No circular

Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Implementación en tiempo real.** La implementación en tiempo real de los algoitmos de retardo múltiple de 2 taps con una sola línea de retardo y dos modificadores de puntero; retardo de 2 taps con dos líneas de retardo; y retardo de 6 taps con seis líneas de retardo difieren de la simulación en que la muestra de entrada no se obtiene de un buffer almacenado en memoria sino de la lectura del codec y su conversión a punto flotante; el almacenamiento de esta muestra en punto flotante en un buffer de entrada; y la conversión de punto flotante a punto fijo de la muestra de salida para ser sacada por el codec; así como se explica en la implementación en tiempo real del retardo simple. En la figura 32 se muestra el diagrama de bloques para la implementación en tiempo real del algoritmo de retardo múltiple de seis taps y seis líneas de retardo. El programa completo del retardo múltiple de 6 taps en tiempo real, el archivo "RetardoMultiple6.asm" se muestra en los anexos.

Figura 32. Diagrama de bloques algoritmo en tiempo real retardo múltiple de seis taps y seis líneas de retardo





Fuente: YASSER MÉNDEZ. Autor del proyecto

3.2 IMPLEMENTACIÓN DE EFECTOS DE AUDIO BASADOS EN MODULACIÓN DEL RETARDO

Los efectos de audio basados en modulación involucran un retardo variable en el tiempo, el cual se consigue mediante un oscilador de baja frecuencia (LF0) que es dispuesto en un *buffer* circular en la memoria del DSP. El retardo variable toma valores entre cero y un retardo máximo, que corresponde al tamaño del buffer circular de la línea de retardo, donde se almacenan las muestras de entrada. La muestra de entrada de la línea de retardo es adicionada a la muestra de entrada actual para obtener la muestra de salida, como lo determina la ecuación 14.

$$y(n) = b_0 * x(n) + b_D * x(n - d(n))$$ Ecuación 14

A partir de esta ecuación se pueden realizar diferentes efectos de audio basados en modulación, que dependen de la línea de retardo variable y del número de líneas que se utilicen, tales como el efecto *flanger* y el efecto *chorus*.

3.2.1 Implementación del efecto flanger. El efecto *flanger* puede ser implementado con la ecuación 14, donde las variaciones del tiempo de retardo son periódicas y se controlan en el DSP usando una tabla de valores de un oscilador de baja frecuencia, que es colocada en memoria. El retardo variable se calcula mediante la ecuación 15 y la ecuación 16.

$$d(n) = \frac{D}{2} - \frac{D}{2} * LFO(n)$$ Ecuación 15

$$LFO(n) = \cos\left(2 * pi * \frac{f_{LFO}}{f_s} * n\right)$$ Ecuación 16

La ecuación 16 es fácilmente implementada en el DSP guardando en un *buffer* circular un periodo de la tabla del LFO. Para ello, se dispone de 4000 posiciones de memoria, que almacenan valores en punto flotante de un periodo de una onda coseno de 2 Hz, generada en matlab con una frecuencia de muestreo de 8000 Hz. En cada periodo de muestreo el valor del LFO es actualizado y se calcula el correspondiente valor del retardo que es usado como modificador del puntero a la línea de retardo. Se debe tener presente que el retardo es un valor entero por lo tanto el valor en punto flotante calculado del retardo es redondeado al entero más cercano mediante el comando en *assembly fix* [5].

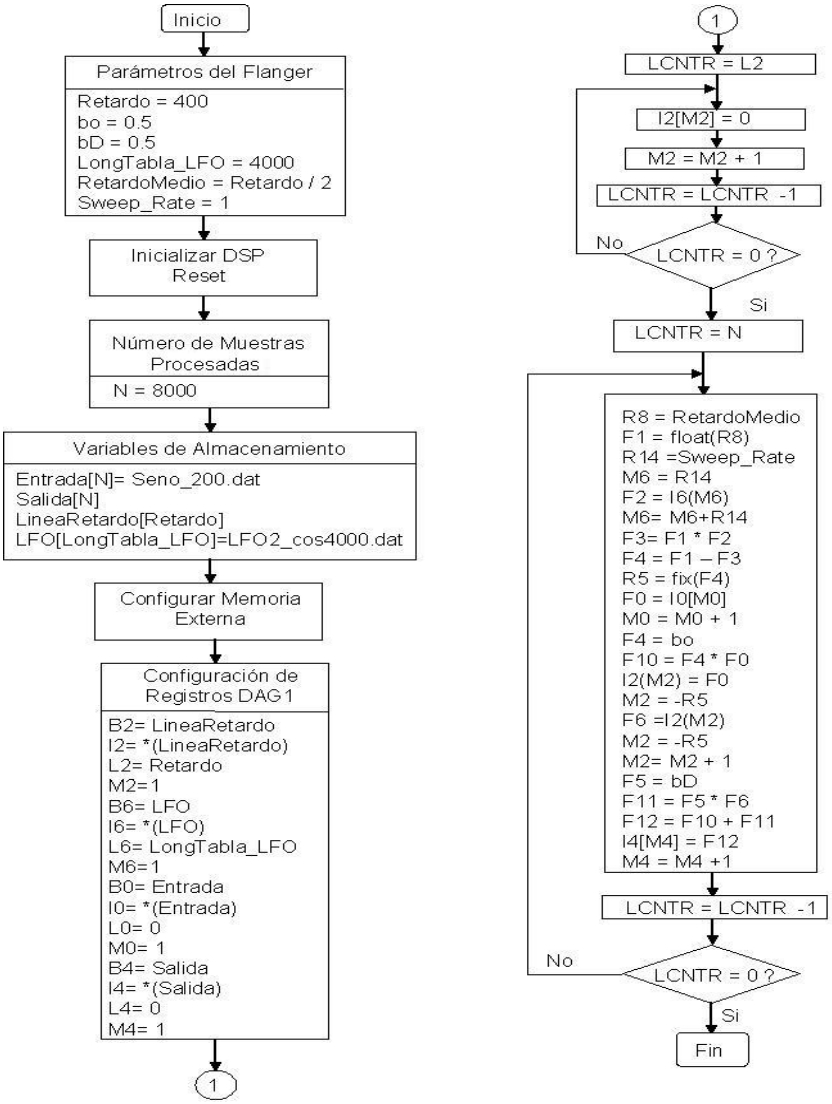
Mediante el parámetro *Sweep Rate* se puede controlar la frecuencia del LFO que varía entre 2 y 10 Hz, para múltiplos de la frecuencia base que es 2 Hz. Este parámetro en la rutina de retardo variable es el modificador del puntero que actualiza el valor del LFO. Así, cuando el *Sweep Rate* toma el valor de uno, el incremento del puntero que busca el valor del LFO es uno y se tiene entonces un LFO de 2 Hz; si se aumenta *Sweep Rate* a dos, el incremento de puntero es dos y el LFO aumenta la frecuencia a 4 Hz. De esta manera se pueden obtener LFOs con frecuencias de 2, 4, 6, 8 y 10 Hz, de acuerdo a la ecuación 17.

$$f_{LFO} = \frac{Sweep\ Rate * f_s}{Longitud\ Tabla\ LFO}$$ Ecuación 17

En cada periodo de muestreo las muestras de entrada son almacenadas en una línea de retardo, implementada con un *buffer* circular de longitud igual al máximo retardo posible. Teniendo en cuenta que el efecto *flanger* se caracteriza por retardos pequeños entre 2 y 10 ms, si se usa una frecuencia de muestreo de 8000 Hz, estos tiempos de retardo corresponden a *buffers* entre 16 y 80 posiciones de memoria, respectivamente.

• **Simulación.** El algoritmo de simulación del efecto *flanger*, mostrada en el diagrama de bloques de la figura 33, se puede realizar en cuatro etapas. La primera consiste en el cálculo del retardo variable implementando la ecuación 15 como se mencionó anteriormente. La segunda parte consiste en obtener la muestra de entrada actual y almacenarla en la línea de retardo. En la tercera parte se busca en la línea de retardo, la muestra retrasada correspondiente al retardo calculado en la primera parte. Finalmente, se halla la muestra de salida, sumando la muestra actual ponderada con la muestra retrasada ponderada antes hallada. Como se muestra en el segmento de programa, correspondiente a la rutina *flanger* para la simulación de este efecto, primero se almacena la muestra de entrada actual en la línea de retardo sin actualizar el puntero y posteriormente se lee la muestra retrasada usando el modificador de puntero que contiene el valor del retardo calculado anteriormente. El programa completo "Flanger_sim.asm" y su correspondiente archivo *linker* "Flanger_sm.ldf" se muestra en los anexos.

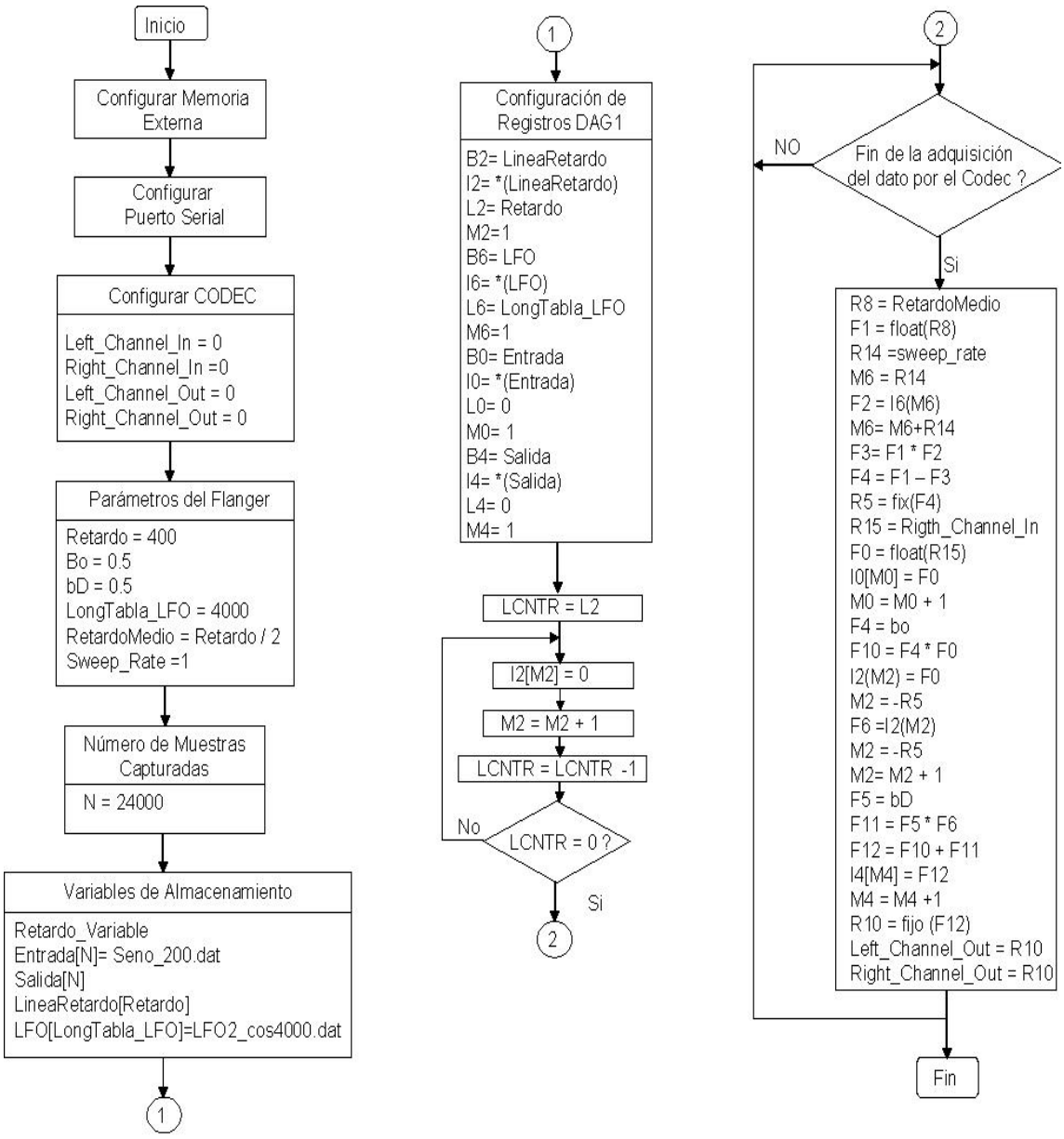
Figura 33. Diagrama de bloques algoritmo de simulación *flanger*



Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Implementación en tiempo real.** La rutina *flanger* para usarse en la tarjeta de evaluación solo difiere en la lectura de la muestra de entrada del codec y su conversión a punto flotante y la escritura de la muestra de salida en punto fijo al codec, como ya se ha mencionado antes. En la figura 34 se muestra el diagrama de bloques para tiempo real del efecto *flanger*. El programa completo "Flanger.asm" con la definición de los coeficientes y los parámetros del *flanger* (Retardo y *Sweep Rate*) se puede ver en los anexos.

Figura 34. Diagrama de bloques algoritmo en tiemo real *flanger*



Fuente: YASSER MÉNDEZ. Autor del proyecto

3.2.2 Implementación del efecto chorus. El efecto *chorus* se puede realizar implementando la ecuación 18 y consiste en agregar líneas de retardo en paralelo al esquema del efecto *flanger*. El control de los retardos variables se hace con un LFO de baja frecuencia (menor que la frecuencia usada para el *flanger*), máximo 3 Hz. La ecuación 19 describe como se calcula el retardo variable y al igual que en el efecto *flanger* se puede implementar en el DSP usando un buffer circular que almacena un periodo de una onda sinusoidal.

$$y(n) = b_0 * x(n) + b_{D1} * x(n - d_1(n)) + b_{D2} * x(n - d_2(n)) + b_{D3} * x(n - d_3(n)) \qquad \text{Ecuación 18}$$

$$d(n) = \frac{D}{2} + \frac{D}{2} * LFO(n) \qquad \text{Ecuación 19}$$

Se implementa un algoritmo que contiene tres líneas de retardo, cada una controlada por un LFO de frecuencia base 0.1 Hz, la cual puede ser aumentada hasta 3 Hz en pasos de 0.1 Hz, mediante los parámetros *Sweep_Rate1*, *Sweep_Rate2 y Sweep_Rate3*. La amplitud de los LFOs puede ser modificada mediante los parámetros *Sweep_Depth1, Sweep_Depth2, y Sweep_Depth3*, que varía entre cero y uno. Estos parámetros junto con el retardo dan el máximo retardo que puede tener cada línea de retardo. La tabla de cada LFO se implementa en el DSP guardando en un *buffer* circular 80000 posiciones de memoria, que almacenan valores en punto flotante de un periodo de una onda seno de 0.1 Hz, generada en matlab con una frecuencia de muestreo de 8000 Hz. En cada periodo de muestreo el valor de cada LFO es actualizado y se calcula el correspondiente valor del retardo entero, almacenado en las variables Retardo_Variable1, Retardo_Variable2 y Retardo_Variable3, que es usado como modificador del puntero a la correspondiente línea de retardo. La muestra de entrada actual se guarda en cada una de las líneas de retardo, que son buffer circulares de longitud entre 120 y 280, correspondientes a retardo máximos típicos entre 15 ms y 35 ms, respectivamente, usando una frecuencia de muestreo de 8000 Hz.

• **Simulación.** El programa de simulación para el algoritmo *chorus* con tres líneas de retardo requiere el uso de tres buffers circulares para almacenar las muestras retrasadas y tres buffer circulares para almacenar las tablas de cada LFO. Además se usan dos buffers para almacenar las muestras de entrada y salida requeridas para evaluación del algoritmo. Los diferentes registros del DAG1 usados para inicializar cada uno de los buffers empleados se muestran en la tabla 5.

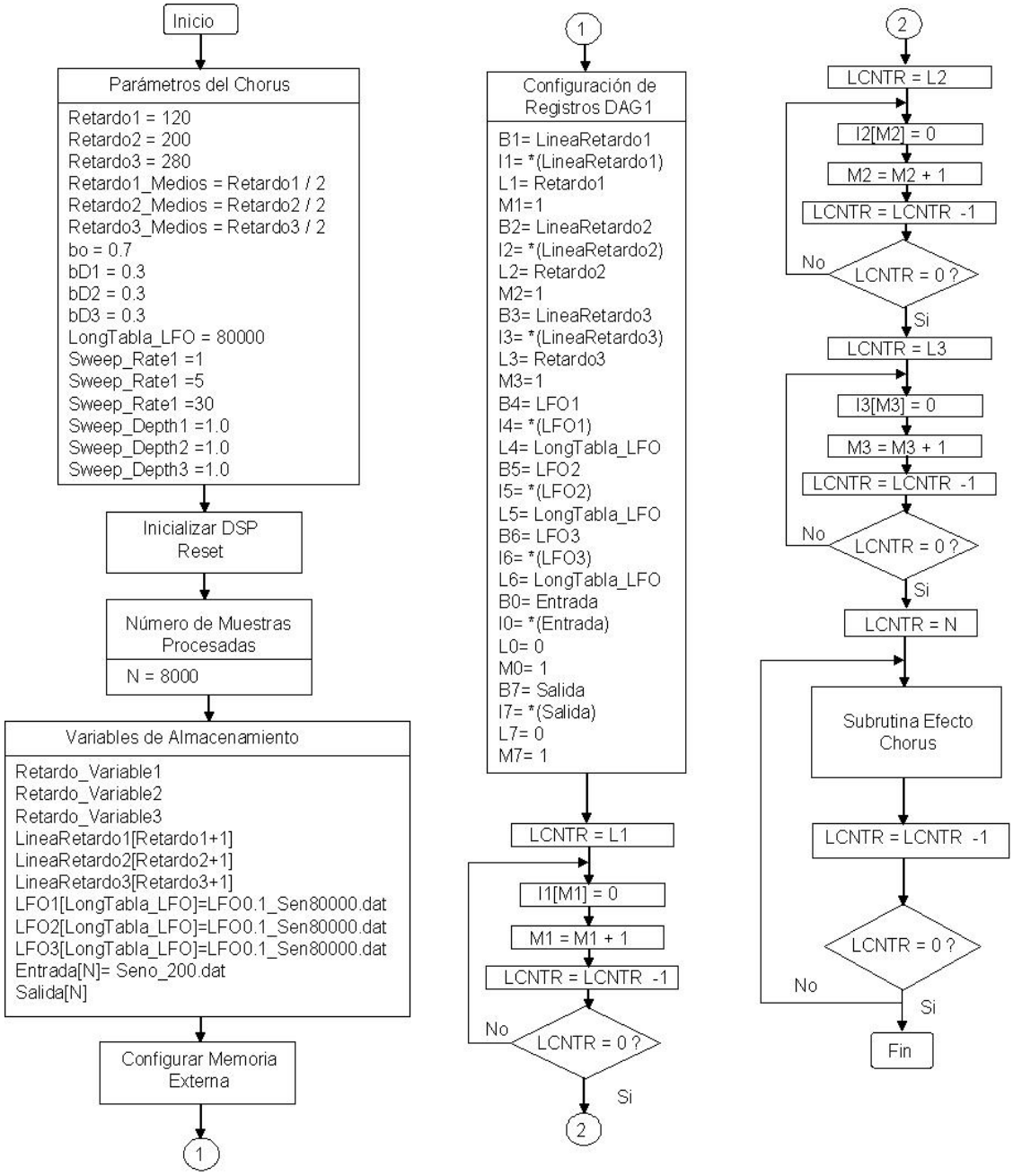
Tabla 5. Registros DAG1 empleados en el algoritmo *Chorus*

Registro Base	Registro Index	Registro Longitud	Registro Modificador	Nombre del buffer	Descripción
B1	I1	L1=Retardo1	M1=1	LineaRetardo1	Circular
B2	I2	L2=Retardo2	M2=1	LineaRetardo2	Circular
B3	I3	L3=Retardo3	M3=1	LineaRetardo3	Circular
B4	I4	L4= LongTabla_LFO		LFO1	Circular
B5	I5	L5= LongTabla_LFO		LFO2	Circular
B6	I6	L6= LongTabla_LFO		LFO3	Circular
B0	I0	L0=0	M0=1	Entrada	No circular
B7	I7	L7=0	M7=1	Salida	No circular

Fuente: YASSER MÉNDEZ. Autor del proyecto

En el diagrama de bloques de la figura 35 se muestra los parámetros del algoritmo *chorus*, las varables de almacenamiento, la configuración de los registros del DAG1, la limpieza de los buffers circulares correspondientes a las tres líneas de retardo y la subrutina del efecto *chorus*.

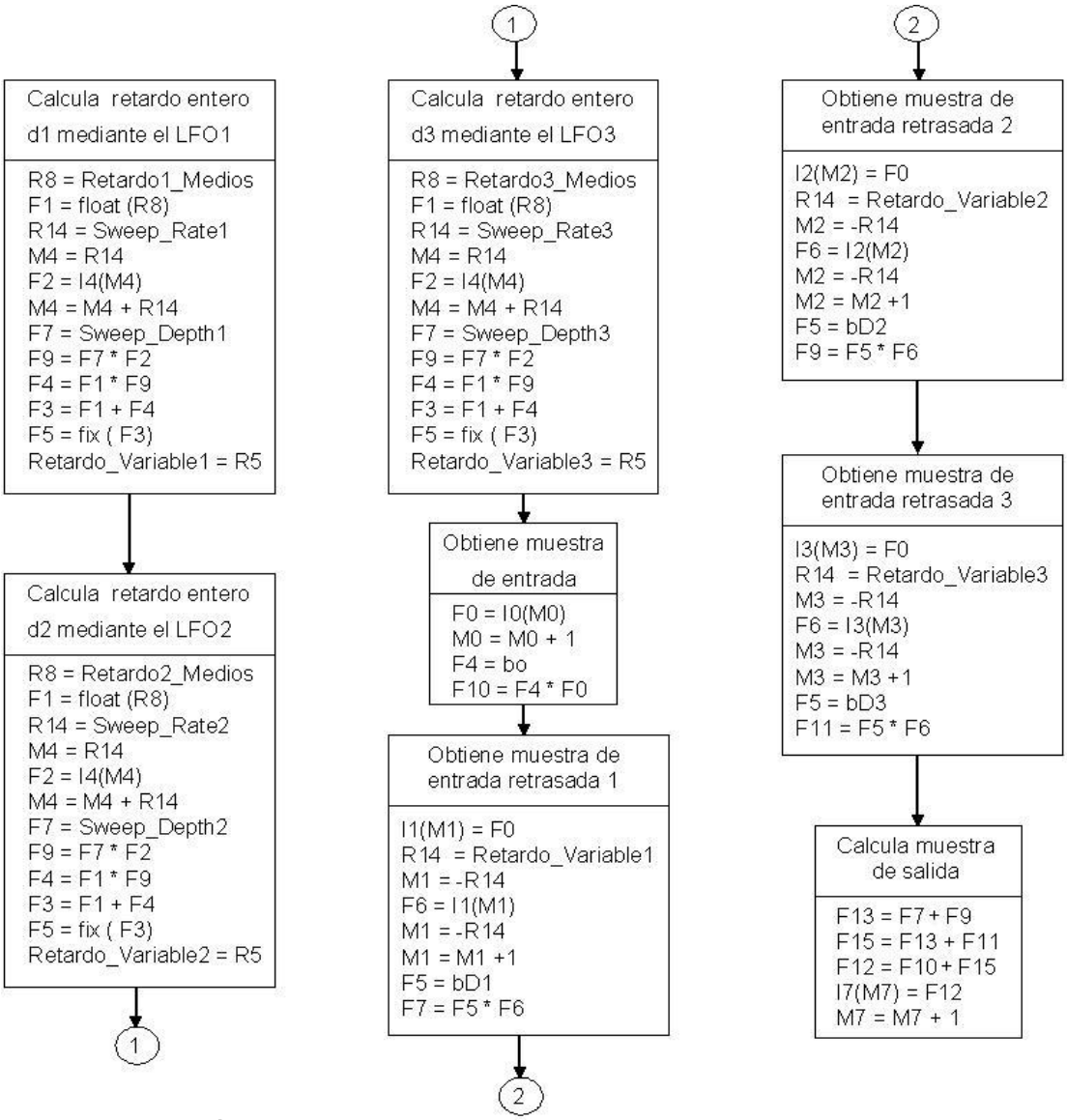
Figura 35. Diagrama de bloques algoritmo de simulación *chorus*



Fuente: YASSER MÉNDEZ. Autor del proyecto

La subrutina del efecto *chorus* se divide en seis partes como se muestra en el diagrama de bloques de la figura 36. La primera parte calcula el retardo entero D_1, D_2 y D_3 y los almacena en las variables Retardo_Variable1, Retardo_Variable2, Retardo_Variable3 respectivamente. La segunda parte consiste en obtener la muestra de entrada actual y ponderarla por el coeficente de entrada. La tercera parte guarda la muestra actual de entrada en el buffer circular LineaRetardo1 sin actualizar el puntero, luego actualiza el modificador de puntero m1 usando el valor de Retardo_Variable1 mediante el registro R14, posteriormente obtiene la muestra de retrasada $x(n-d_1(n))$ y actualiza el puntero de la LineaRetardo1, y finalmente pondera la muestra retrasada por el respectivo coeficiente b_{D1} almacenándola en el registro F7.

Figura 36. Diagrama de bloques subrutina de simulación efecto *chorus*



Fuente: YASSER MÉNDEZ. Autor del proyecto

La cuarta parte guarda la muestra actual de entrada x(n) en el buffer circular LineaRetardo2 sin actualizar el puntero, luego actualiza el modificador de puntero m2 usando el valor de Retardo_Variable2 mediante el registro R14, posteriormente obtiene la muestra de retrasada x(n-d_2(n)) y actualiza el puntero de la LineaRetardo2, y finalmente pondera la muestra retrasada por el respectivo coeficiente b_{D2} almacenándola en el registro F9. La quinta parte guarda la muestra actual de entrada en el buffer circular LineaRetardo3 sin actualizar el puntero, luego actualiza el modificador de puntero m3 usando el valor de Retardo_Variable3 mediante el registro R14, posteriormente obtiene la muestra de retrasada x(n-d_3(n)) y actualiza el puntero de la LineaRetardo3, y finalmente pondera la muestra retrasada por el respectivo coeficiente b_{D3} almacenándola en el registro F11. En última parte se suman la muestra de entrada actual ponderada y las muestras retrasadas ponderadas obteniendo así la muestra de salida ene el registro F12 y que se almacena en el buffer de salida.

El programa completo de simulación para el efecto *chorus* "Chorus3_sim.asm", con la definición de los parámetros (Retardo1, Retardo2, Retardo3, Sweep_Rate1, Sweep_Rate2, Sweep_Rate3, Sweep_Depth1, Sweep_Depth2 Sweep_Depth3), buffers de entrada, salida y las líneas de retardo; así como el archivo *linker* "Chorus3_sim.ldf" se puede ver en los anexos.

- **Implementación en tiempo real.** La subrutina *chorus* para ser usada en la tarjeta de evaluación simplemente cambia la obtención de la muestra de la muestra de entrada desde un buffer por la lectura desde el Codec y su conversión a punto flotante y posterior almacenamiento de esta muestra en el buffer de entrada; y adiciona las líneas correspondientes a colocar la muestra de salida en el codec, convirtiendo a punto fijo.

En el diagrama de bloques de la figura 37 se puede observar el diagrama de bloques del algoritmo *chorus* para ser ejecutado con la tarjeta de evaluación y en la figura 38 se muestra los cambios para la subrutina *chorus* en tiempo real. El programa completo del efecto *chorus* en tiempo real "Chorus3.asm", con la definición de los coeficientes (bo, b_{D1}, b_{D2} y b_{D3}), los parámetros del *chorus* (Retardo1, Retardo2, Retardo3, Sweep_Rate1, Sweep_Rate2, Sweep_Rate3, Sweep_Depth1, Sweep_Depth2 Sweep_Depth3), los buffers circulares de las tres líneas de retardo y los buffers de almacenamiento de las muestras de entrada y salida; y los demás archivos en *assembly* y el archivo del *linker* requeridos para ejecutar el programa en la tarjeta de desarrollo se pueden ver en los anexos.

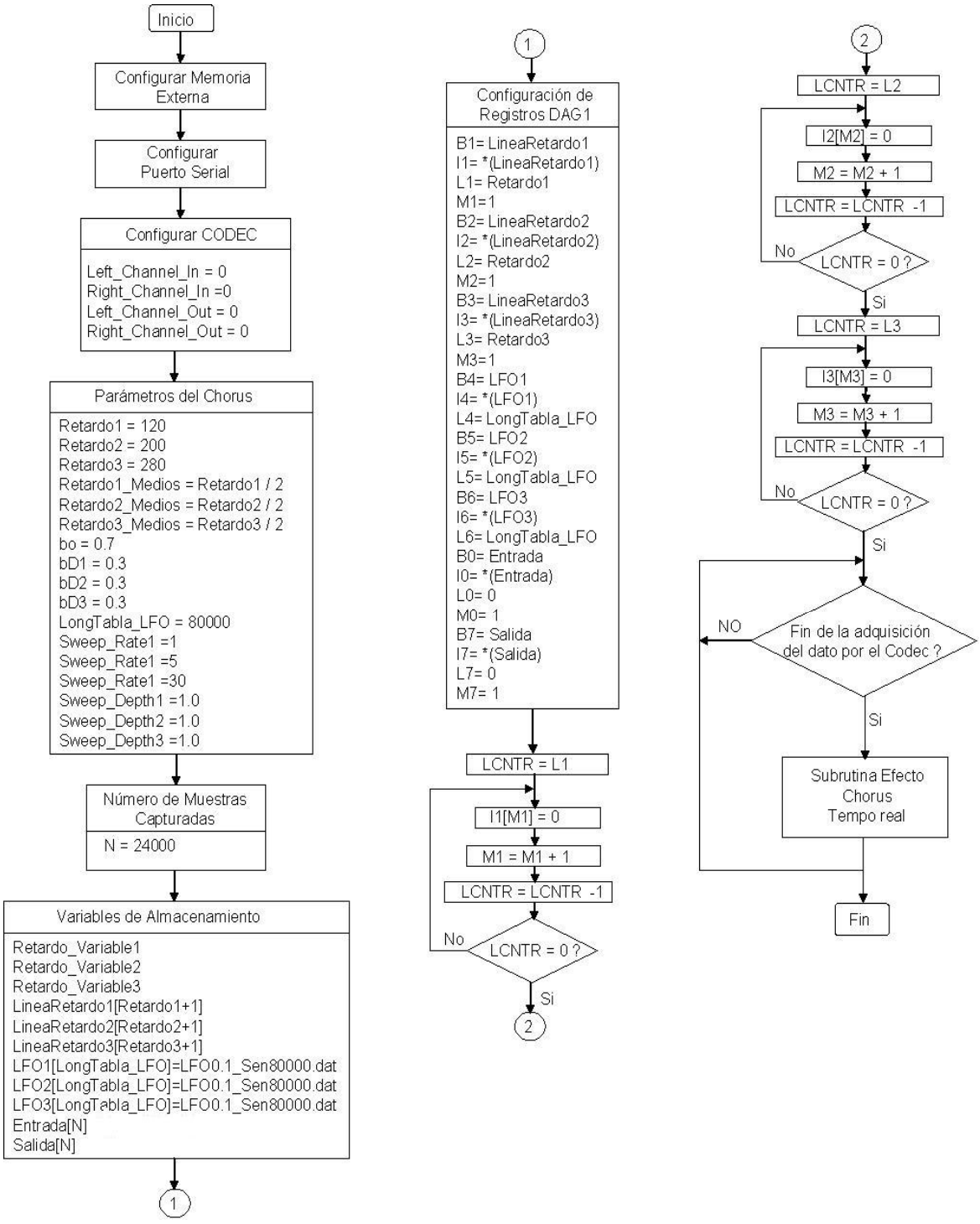
3.3 IMPLEMENTACIÓN DE EFECTOS DE AUDIO BASADOS EN AMPLITUD

3.3.1 Implementación del compresor. El algoritmo compresor con detección pico consiste en comparar la muestra de entrada con un umbral; si el valor absoluto de la muestra de entrada es menor que el valor del umbral, entonces la muestra de salida es la misma muestra de entrada. En caso contrario la muestra de salida es igual a la muestra de entrada afectada por un factor de compresión según las ecuaciones 20 y 21, si la muestra de entrada es mayor o menor a cero, respectivamente. La comparación se implementa en el DSP mediante la instrucción en *assembly "comp(Fx,Fy)"* y el uso del salto condicional mediante la instrucción en *assembly "IF Condición JUMP Etiqueta"* [3].

$$y(n)= \text{Umbral} + (x(n)-\text{Umbral})* \text{Ratio}, \quad x(n)>0 \qquad \text{Ecuación 20}$$

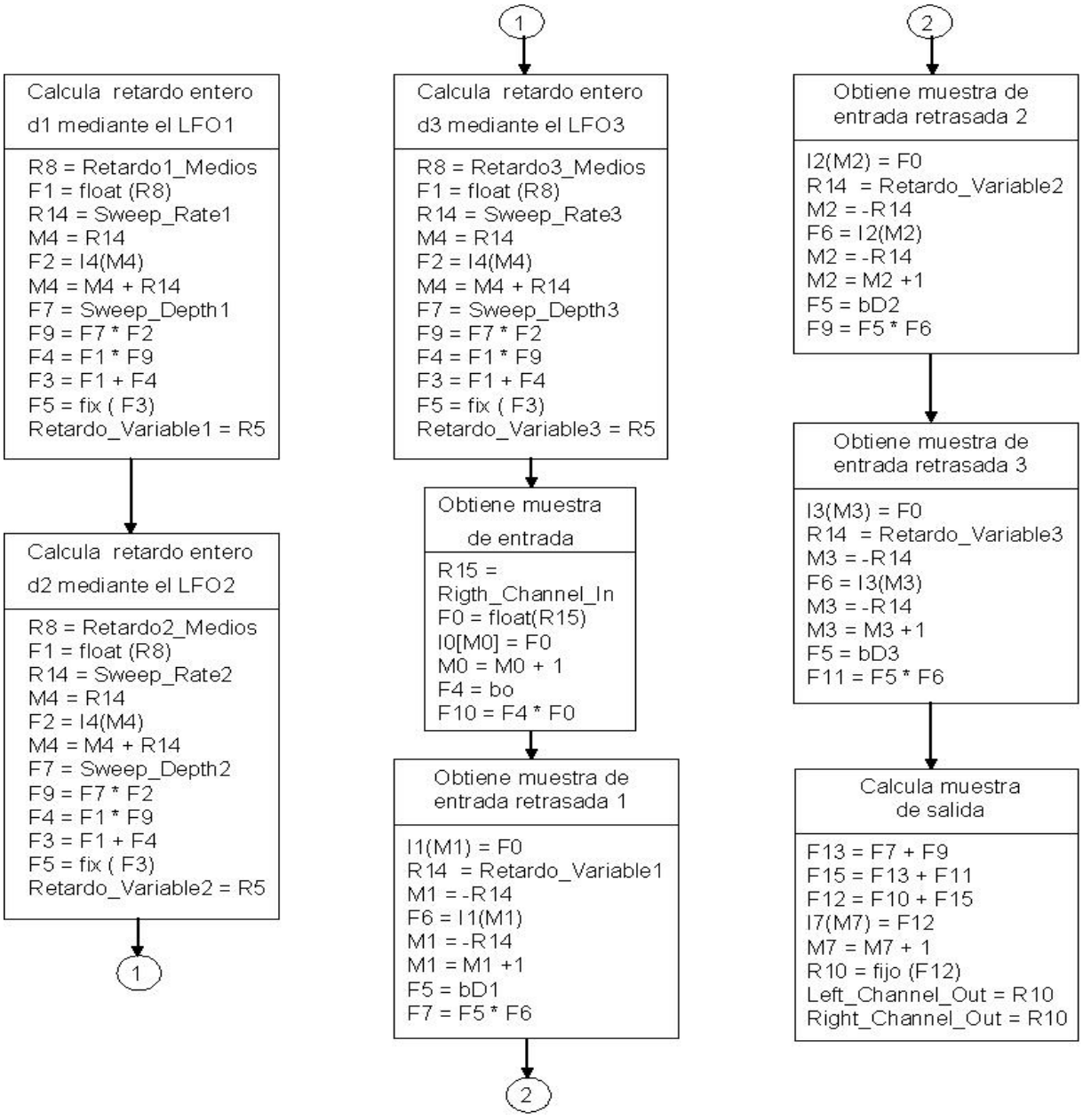
$$y(n)= -\text{Umbral} + [x(n)+\text{Umbral}]* \text{Ratio}, \quad x(n)<0 \qquad \text{Ecuación 21}$$

Figura 37. Diagrama de bloques algoritmo en tiempo real *chorus*



Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 38. Diagrama de bloques subrutina en tiempo real efecto *chorus*

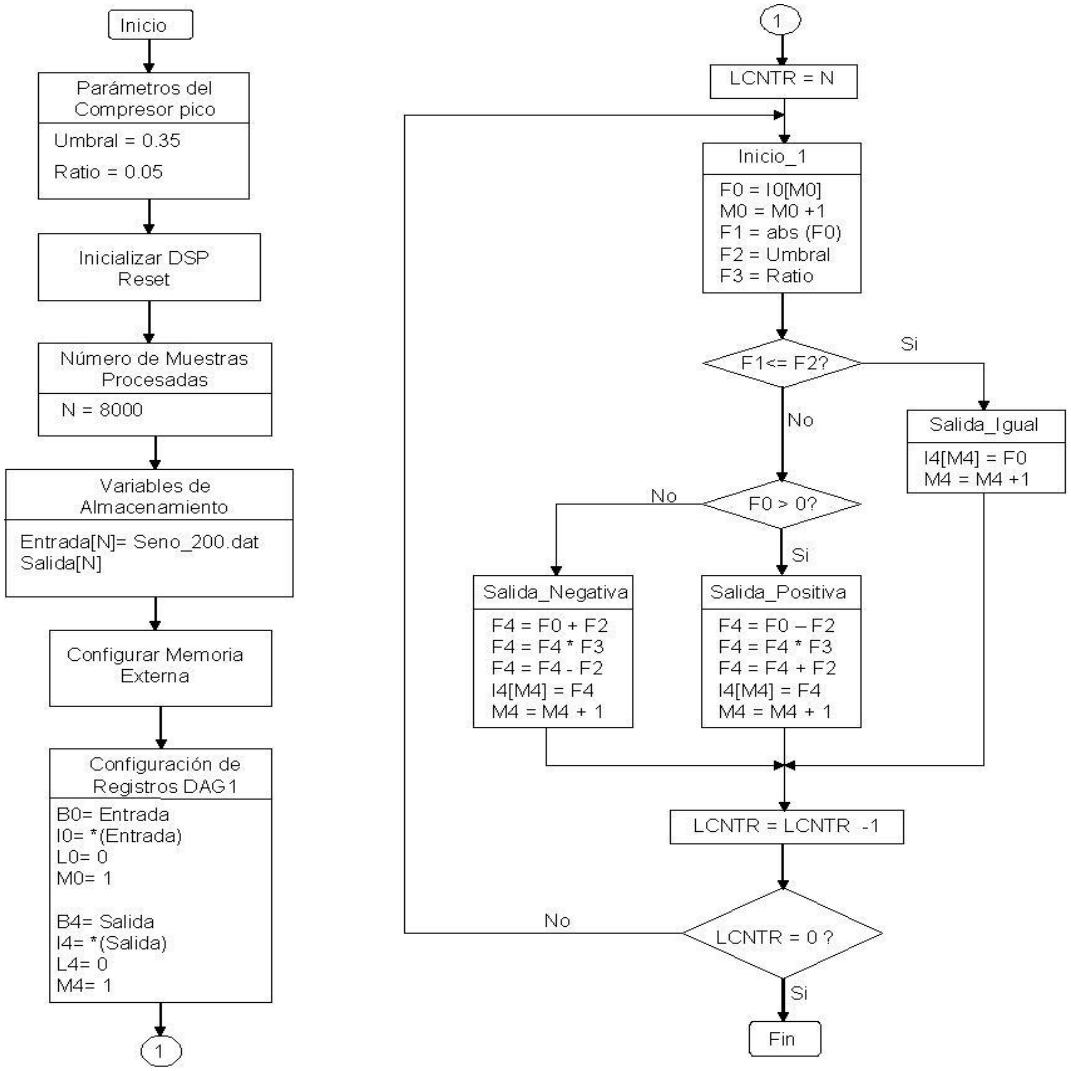


Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Simulación.** La rutina de simulación del compresor por detección pico se divide en cuatro subrutinas. La primera subrutina contiene siete instrucciones y es la de "Inicio_1", donde se obtiene la muestra de entrada, se saca su valor absoluto y lo compara con el umbral, ejecutando un salto a la subrutina *"Salida_Igual"* si la condición es verdadera o un salto a un comparador con cero, si la condición es falsa. La segunda subrutina es la de *"Salida_Igual"* que almacena la muestra de salida igual a la muestra de entrada, en un buffer de salida, y realiza un salto a la subrutina

"Inicio_1". El comparador con cero evalúa si la muestra de entrada, mayor que el umbral, es mayor o menor que cero, en caso afirmativo ejecuta un salto a la subrutina *"Salida_Positiva"*, en caso contrario realiza un salto a la subrutina "Salida_Negativa". La tercera subrutina es *"Salida_Positiva"*, que calcula la salida siguiendo la ecuación 20, almacenándola en un buffer de salida y realizando un salto a la subrutina *"Inicio_1"*. Finalmente la cuarta subrutina es "*Salida_Negativa*", que calcula la salida siguiendo la ecuación 21, almacenándola en un buffer de salida y realizando un salto a la subrutina *"Inicio_1"*. En la figura 39 se muestra el diagrama de bloques para el algoritmo de simulación del compresor pico y el programa completo "Compresor.asm" con la definición de los parámetros del compresor (Umbral y Ratio), definición de los buffers de entrada y salida, se puede ver en los anexos.

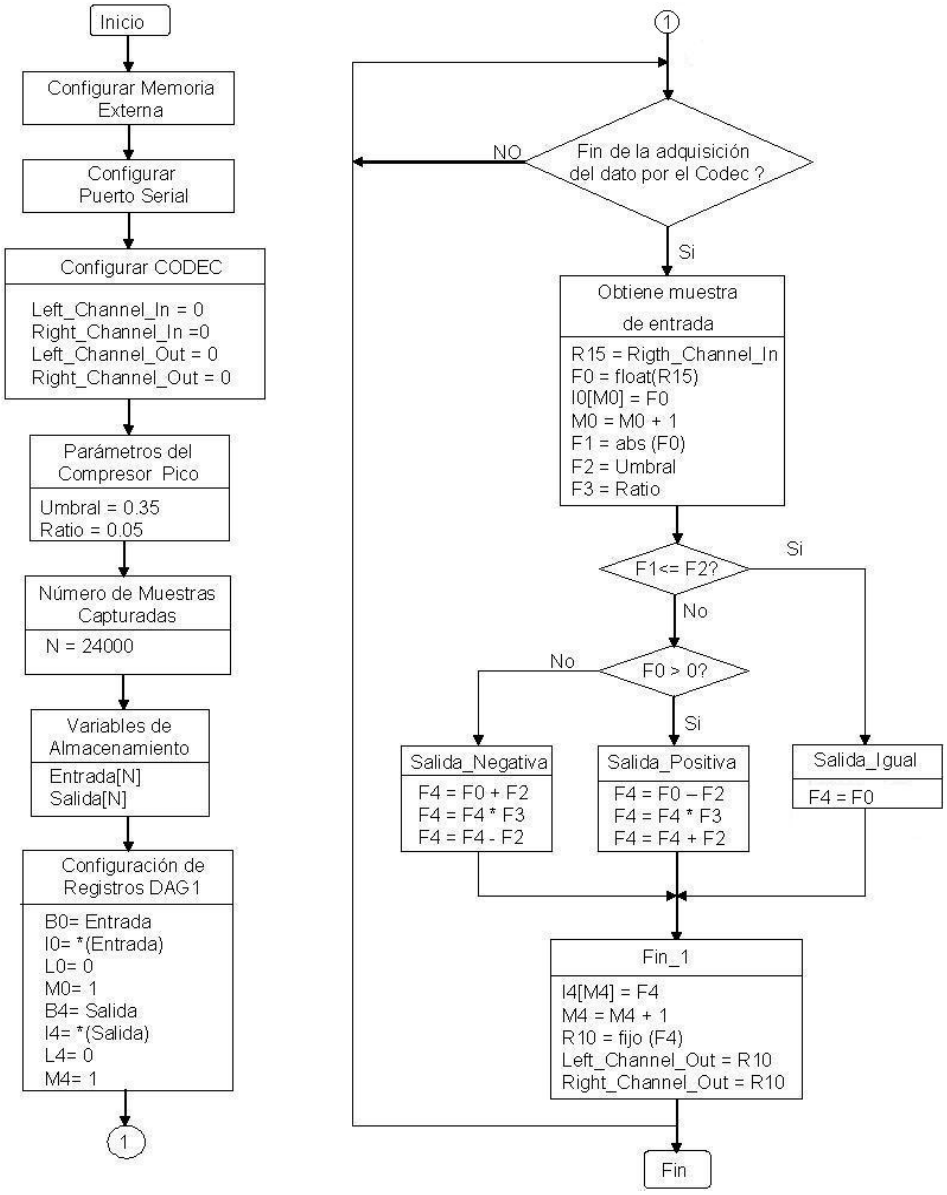
Figura 39. Diagrama de bloques algoritmo de simulación compresor pico



Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Implementación en tiempo real.** La rutina de compresor por detección pico en tiempo real para correr en la tarjeta de evaluación, cuyo diagrama de bloques se muestra en la figura 40, tiene las mismas subrutinas de la simulación "Salida_Igual", "Salida_Positiva" y "Salida_Negativa", pero se eliminan las instrucciones que almacenan la muestra de salida en un buffer y se sustituyen los saltos al inicio del programa, por saltos a la subrutina "Fin_1" que realiza la conversión de la muestra de salida de punto flotante a punto fijo para colocarla a la salida del codec. Además se elimina la línea del lazo y se agregan las instrucciones para leer la muestra de entrada del codec, su conversión e punto fijo a punto flotante y almacenamieno en el buffer de entrada como en los anteriores programas para tiempo real.

Figura 40. Diagrama de bloques algoritmo en tiempo real compresor pico



Fuente: YASSER MÉNDEZ. Autor del proyecto

El programa completo del efecto compresor pico en tiempo real “Compresor_TR.asm”, con la definición de los parámetros del compresor, la definición de los buffers de entrada y salida, se puede ver en los anexos. Los archivos de inicialización de la tarjeta, la memoria externa y el Codec, el archivo de borrado de los registros del puerto serial, el archivo de procesamiento del Codec, el archivo de la tabla de interrupciones y el archivo del linker (LDF), necesarios para que el programa se ejecute en la tarjeta de evaluación, son los mismos que los usados en los anteriores efectos, y se muestran en los anexos.

También se puede implementar una rutina de compresor con detección RMS, mediante un buffer circular que almacena M muestras, y para cada muestra de entrada x(n) se calcula el valor RMS de las últimas M muestras de entrada. Esto requiere el uso de la función en lenguaje Assembly, Fn=RSQRTS Fx, que calcula la primitiva del recíproco de la raíz cuadrada de Fx; y adicionando una serie de 19 instrucciones, correspondientes al algoritmo de iteración de Newton-Raphson [6], se obtiene la raíz cuadrada de un número en punto flotante, necesaria para determinar el valor RMS según la ecuación 22.

$$x_{RMS}(n) = \sum_{k=n-(M-1)}^{n} x^2(k) * (1/M) \qquad \text{Ecuación 22}$$

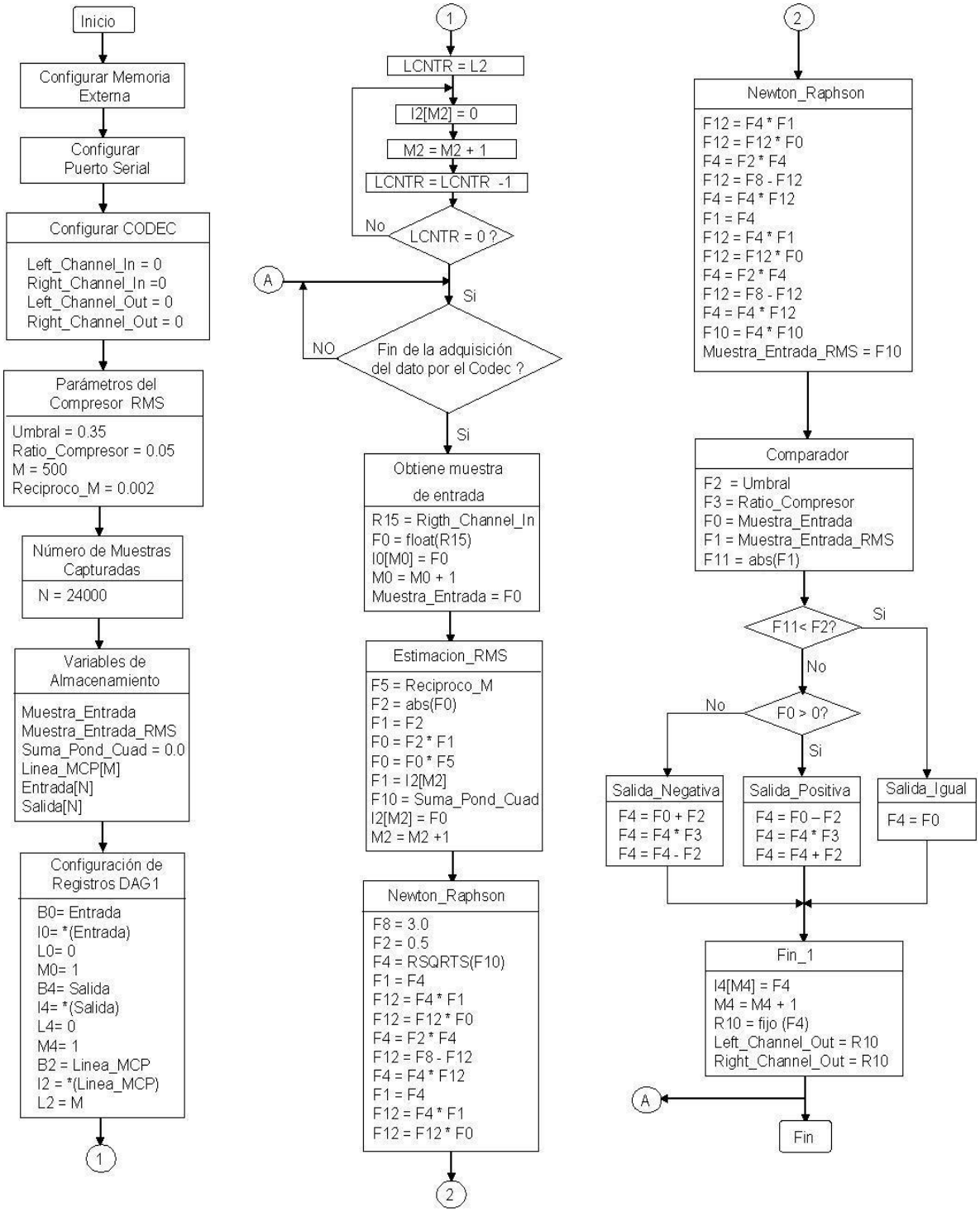
donde M es el número de muestras para estimar el valor RMS y $x_{RMS}(n)$ es el valor RMS de las últimas M muestras.

Adicionando algunas líneas al programa del compresor por detección pico en tiempo real descrito anteriormente, se obtiene el archivo “Compresor_RMS_TR.asm” que ejecuta la rutina de compresor por detección RMS en tiempo real. Se debe definir los parámetros de Umbral y Ratio; el número de muestras para estimar el valor RMS de la muestra de entrada, M; el recíproco de M, 1/M; la variable “Muestra_Entrada”, que almacena la muestra de entrada actual x(n); la variable “Muestra_Entrada_RMS”, que guarda la estimación RMS de la muestra de entrada; la variable “Suma_Pond_Cua”, que guarda el valor de la suma de entradas al cuadrado ponderadas; el número de muestras de entrada y salida, N, que se desean almacenar; los buffers de entrada y salida; y la definición del buffer que almacena las ultimas M muestras de entrada al cuadrado ponderadas, “Línea_MCP[M]”.

La rutina correspondiente al compresor por detección RMS en tiempo real inicia con la obtención de la muestra de entrada x(n) del Codec, su conversión a punto flotante y almacenamiento en el buffer de entrada y en la variable “Muestra_Entrada”. La estimación del valor RMS de de la muestra de entrada se realiza en dos partes: la primera corresponde a determinar la suma de las ultimas M entradas al cuadrado ponderadas; y la segunda parte a calcular la raíz cuadrada de este valor usando el método de iteración de Newton-Rapshon. Una vez se ha estimado el valor RMS de la muestra de entrada se procede a la comparación con el umbral para determinar si la muestra de entrada es afectada por el factor de compresión o es enviada directamente a la salida del codec. Las subrutinas “Salida_Igual”, “Salida_Compresor”, “Salida_Positiva”, “Salida_Negativa” y “Fin_1” son las mismas del programa compresor por detección pico.

En la figura 41 se muestra el diagrama de bloques para el algoritmo compresor con detección RMS en tiempo real. El programa completo del efecto compresor con detección RMS en tiempo real “Compresor_RMS_TR.asm” se muestra en los anexos.

Figura 41. Diagrama de bloques algoritmo en tiempo real compresor RMS

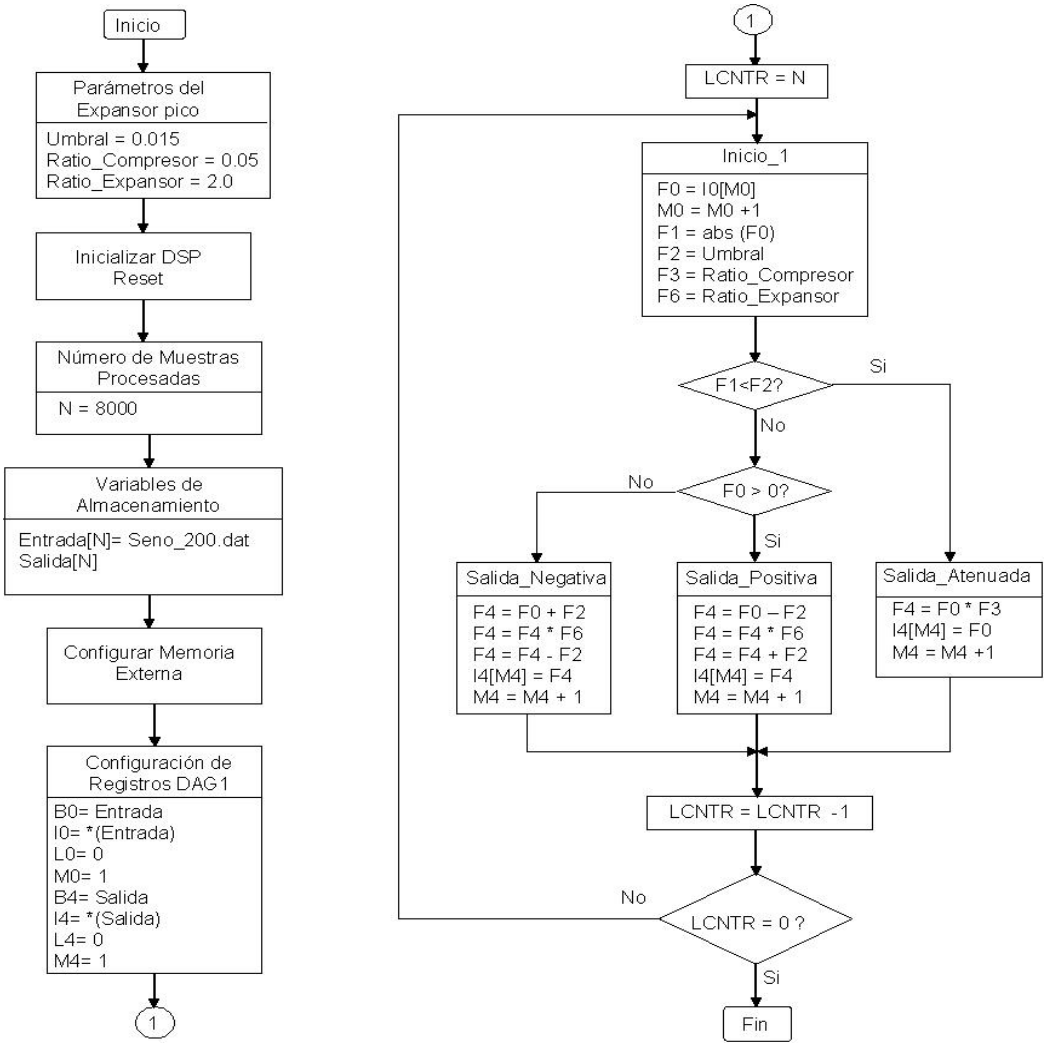


Fuente: YASSER MÉNDEZ. Autor del proyecto

3.3.2 Implementación del expansor. Básicamente el expansor es lo contrario del compresor, las señales con amplitud por encima de un umbral aumentan su ganancia al ser afectadas por un factor de expansión que es mayor de uno; mientras que las señales con amplitud por debajo del umbral reducen su ganancia al ser afectadas por un factor de compresión (entre cero y uno).

• **Simulación.** La rutina de simulación para el efecto expansor es básicamente la misma del compresor y solo requiere de la adición de una nueva variable, "Ratio_Expansor" y cambiar la subrutina "Salida_Igual" por la subrutina "Salida_Atenuada" y adicionar la variable con el valor de la relación de expansión y cambiar en la subrutinas "Salida_Positiva" y "Salida_Negativa" la relación de compresión por la relación de expansión, como se muestra en el diagrama de bloques de la figura 42. El programa completo "Expansor.asm" se muestra en los anexos.

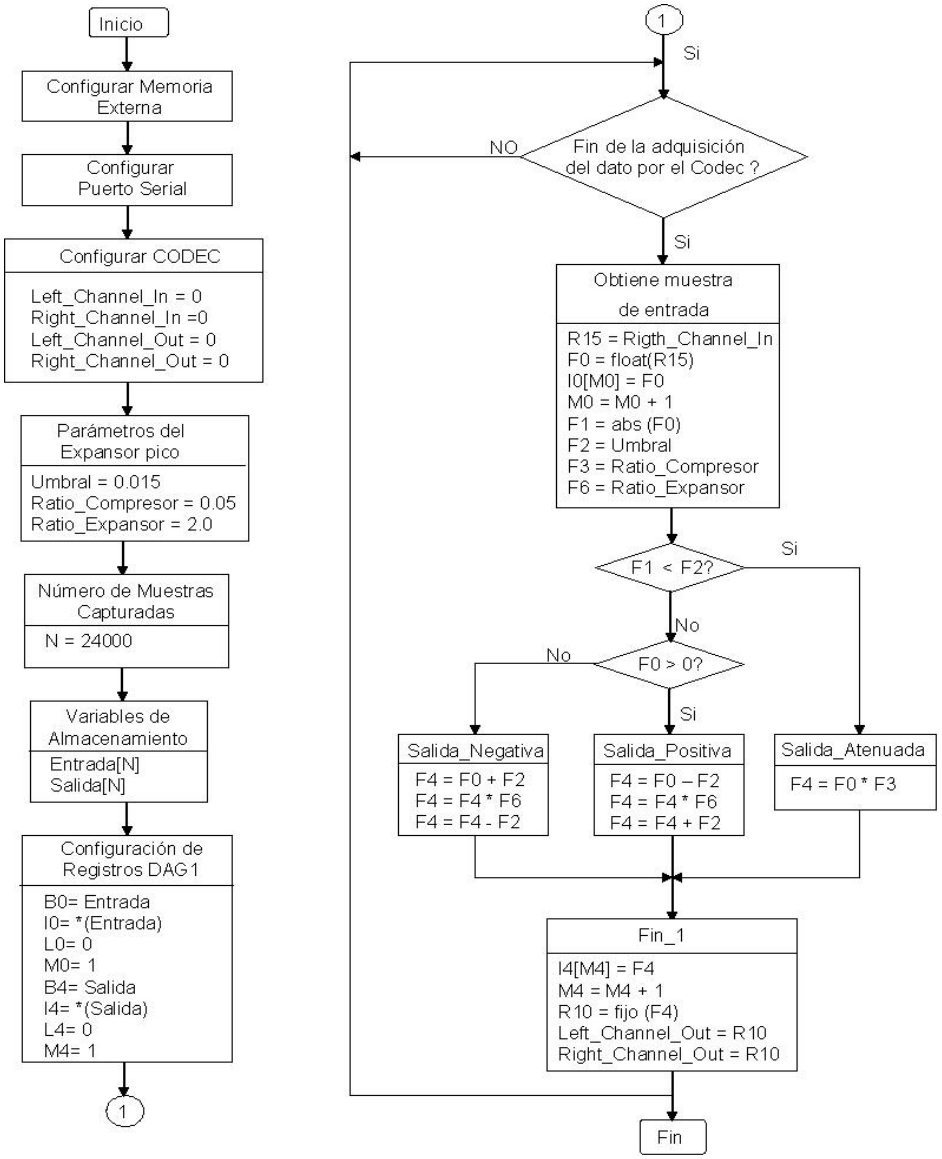
Figura 42. Diagrama de bloques algoritmo de simulación Expansor pico



Fuente: YASSER MÉNDEZ. Autor del proyecto

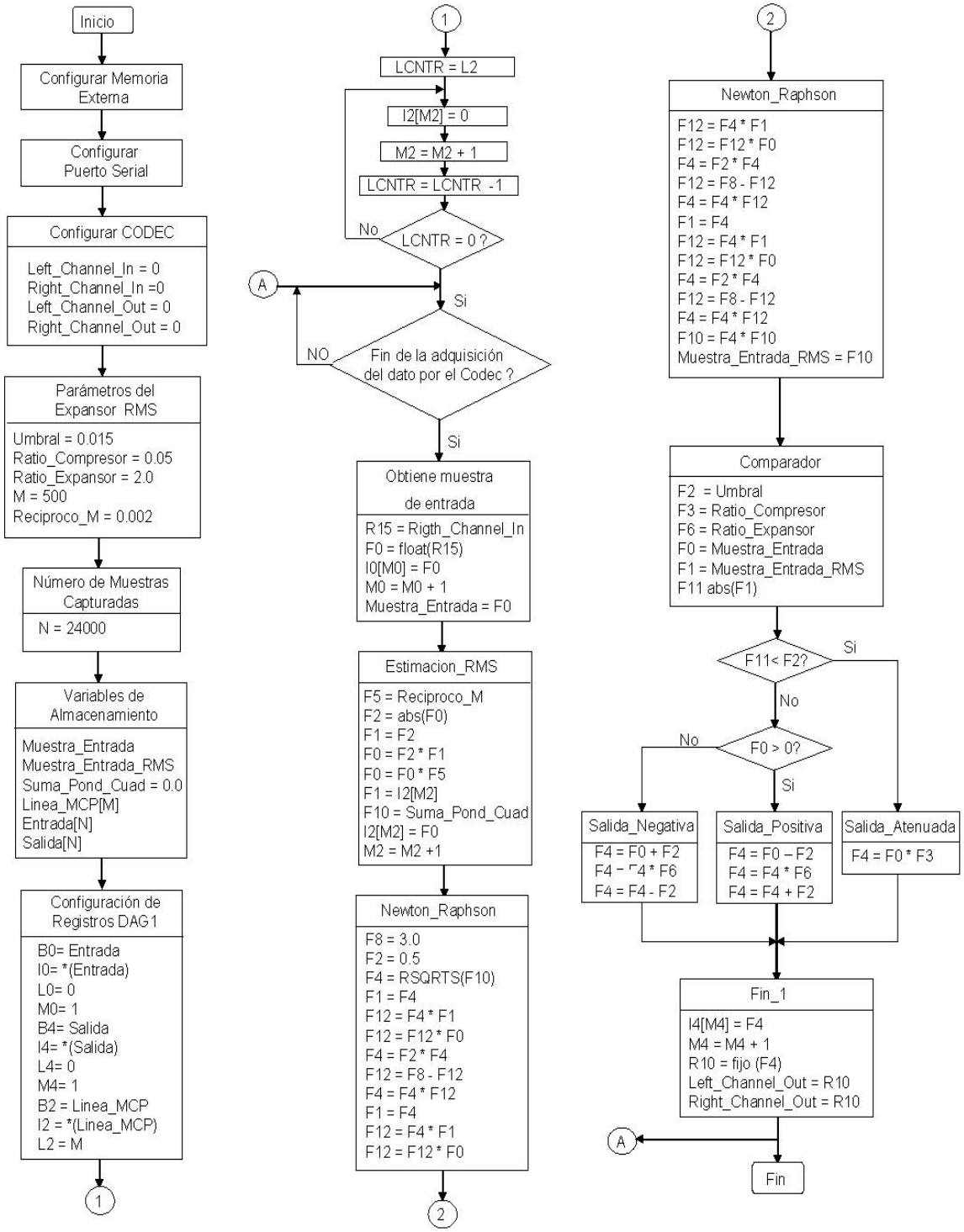
• **Implementación en tiempo real.** La rutina del expansor (detección pico y detección RMS) en tiempo real elimina las líneas que guarda la muestra de salida en un buffer y cambia los saltos a "Inicio_1" por salto a la subrutina "Fin_1", de la misma forma que se hace en la implementación en tiempo real del compresor y como se puede observar en el diagrama de bloques ed la figura 43 para el expansor con detección pico y en el diagrama de bloques de la figura 44 para el expansor con detección RMS. Los programas completos del expansor pico expansor_TR.asm" y el expansor RMS "Expansor_RMS _TR.asm", con la definción de parámetros y buffers, así como los otros archivos necesarios para ejecutar el programa en la tarjeta de evaluación se muestra en los anexos.

Figura 43. Diagrama de bloques algoritmo en tiempo real Expansor pico



Fuente: YASSER MÉNDEZ. Autor del proyecto

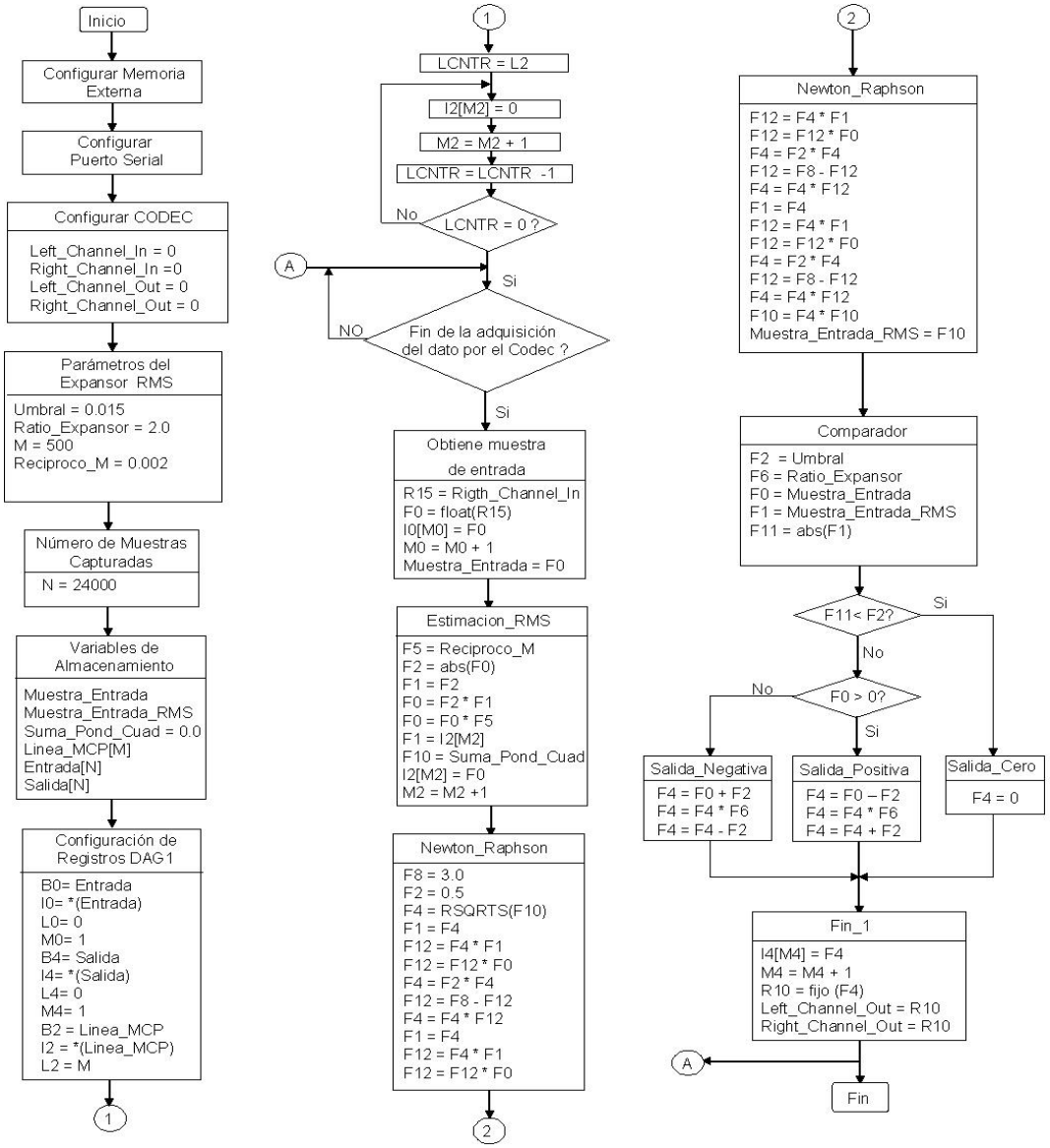
Figura 44. Diagrama de bloques algoritmo en tiempo real Expansor RMS



Fuente: YASSER MÉNDEZ. Autor del proyecto

Una variación del expansor es el Noise Gate, que elimina totalmente la señal por debajo del umbral y la implementación en tiempo real (generalmente se usa la detección RMS) es mucho más sencilla y prácticamente es el mismo algoritmo del expansor, con los parámetros de umbral y relación de expansión adecuados; solo se cambia la subrutina “Salida_Atenuada “por la subrutina “”Salida_Cero” como se muestra en el diagrama de bloques de la figura 45. El programa completo, “NoiseGate_TR.asm” con la definición de parámetros y buffers, y los otros necesarios para ejecutar el programa en la tarjeta de evaluación se muestra en los anexos

Figura 45. Diagrama de bloques algoritmo en tiempo real *Noise Gate* RMS



Fuente: YASSER MÉNDEZ. Autor del proyecto

3.4 IMPLEMENTACIÓN DE FILTROS EN EL DSP

3.4.1 Implementación de filtros FIR. Un filtro FIR es la suma finita ponderada de una entrada desplazada en el tiempo y puede ser definido por la ecuación 23.

$$y(n)=\sum_{k=0}^{M} b_k\, x(n-k) \qquad \text{Ecuación 23}$$

donde x(n-k) representa la entrada al filtro desplazada en el tiempo, b_k representa los coeficientes del filtro y y(n) representa la salida del filtro. El código del filtro FIR es una implementación en software de esta ecuación. El filtro FIR puede ser caracterizado completamente si los coeficientes son conocidos.

Las especificaciones en frecuencia del filtro que determinan si es un pasabajos, un pasaltos o un pasabanda, y los parámetros de frecuencia de la banda de paso, frecuencia de la banda eliminada, y los correspondientes factores de atenuación y rizado de estas bandas, permiten que la respuesta en frecuencia del filtro se lo más cercana a la ideal. Esto depende principalmente del método usado para calcular los coeficientes. Los métodos más comunes son el ventaneo, el muestreo en frecuencia y el de Park- McClellan (o algoritmo Remez) [40].

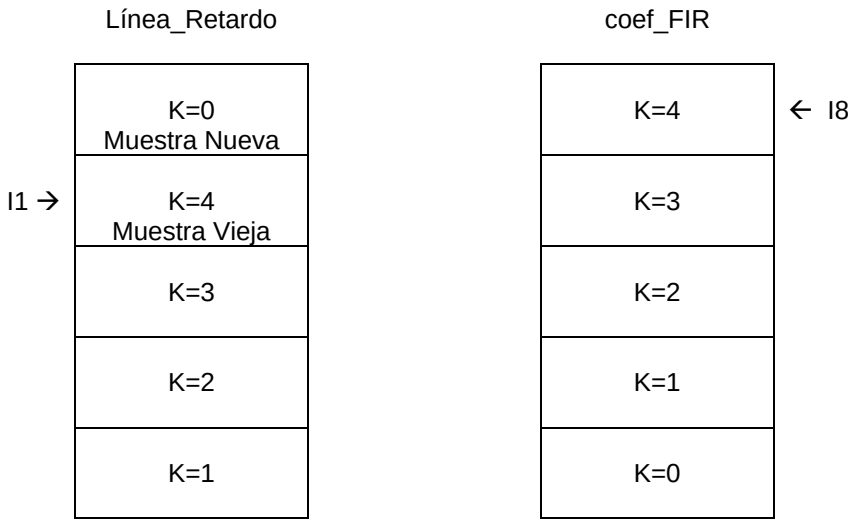
Los filtros FIR pueden ser diseñados para tener una respuesta en fase lineal aprovechando las características de simetría. Para el cálculo de los coeficientes del filtro que se implementa se usa el algoritmo de Remez mediante la función de Matlab *REMEZORD*, que estima el orden del filtro y los parámetros de las bandas de frecuencia, amplitudes y pesos para ser usados con la función *REMEZ*, que calcula los coeficientes reales y simétricos de un filtro FIR de fase lineal. Una vez se tienen los coeficientes del filtro se listan en un archivo ".dat" para ser luego usado por el algoritmo implementado en el DSP.

• **Simulación.** En el programa de simulación del filtro FIR primero se definen seis variables. Estas variables son el número de muestras de la señal de entrada que se van a utilizar, N (línea 1); el número de coeficientes del filtro, Taps: un buffer circular que almacena las muestras previas de la señal de entrada, Linea_Retardo[Taps], de longitud igual al número de Taps; y un buffer circular que almacena los coeficientes del filtro, coef_FIR[Taps], de longitud igual al número de Taps; dos buffers no circulares donde se almacenan las muestras de la señal de entrada que van a ser filtradas, Entrada[N] ,y las muestras de la señal de salida, Salida[N]. Los buffers de entrada y de salida, así como el buffer circular de la línea de retardo, se definen en el segmento de datos de la memoria de datos del DSP; mientras que el buffer circular que contiene los coeficientes del filtro, se define en el segmento de datos de la memoria de programa del DSP. La inicialización de los buffers sigue la estructura descrita en los algoritmos anteriores.

La rutina FIR inicia con leer la muestra de entrada x(n) de buffer correspondiente, luego usa las instrucciones XOR y PASS del lenguaje assembly del DSP para colocar en ceros los registros de datos R12 y F8 que almacenan los resultados del producto y suma respectivamente, que son el corazón del algoritmo de filtrado. La instrucción XOR corresponde a una operación de la ALU que puede ser ejecutada junto con el movimiento de un dato a memoria, en una misma instrucción aprovechando esta característica de los DSP de la familia ADSP-21xxx. En esta instrucción el

puntero de la línea de retardo es actualizado. Por esta razón cuando no se usan coeficientes simétricos, es necesario ordenar el vector de coeficientes de tal manera que el correspondiente a Kmax este ubicado en la primera posición del vector. El modificador (M1) mueve el puntero de la línea de retardo (I1) al valor viejo en ella. La muestra de entrada x(n-k) que ya ha sido escrita en la línea de retardo, no es leída hasta que resto del buffer es accesado como se muestra en la Figura 46.

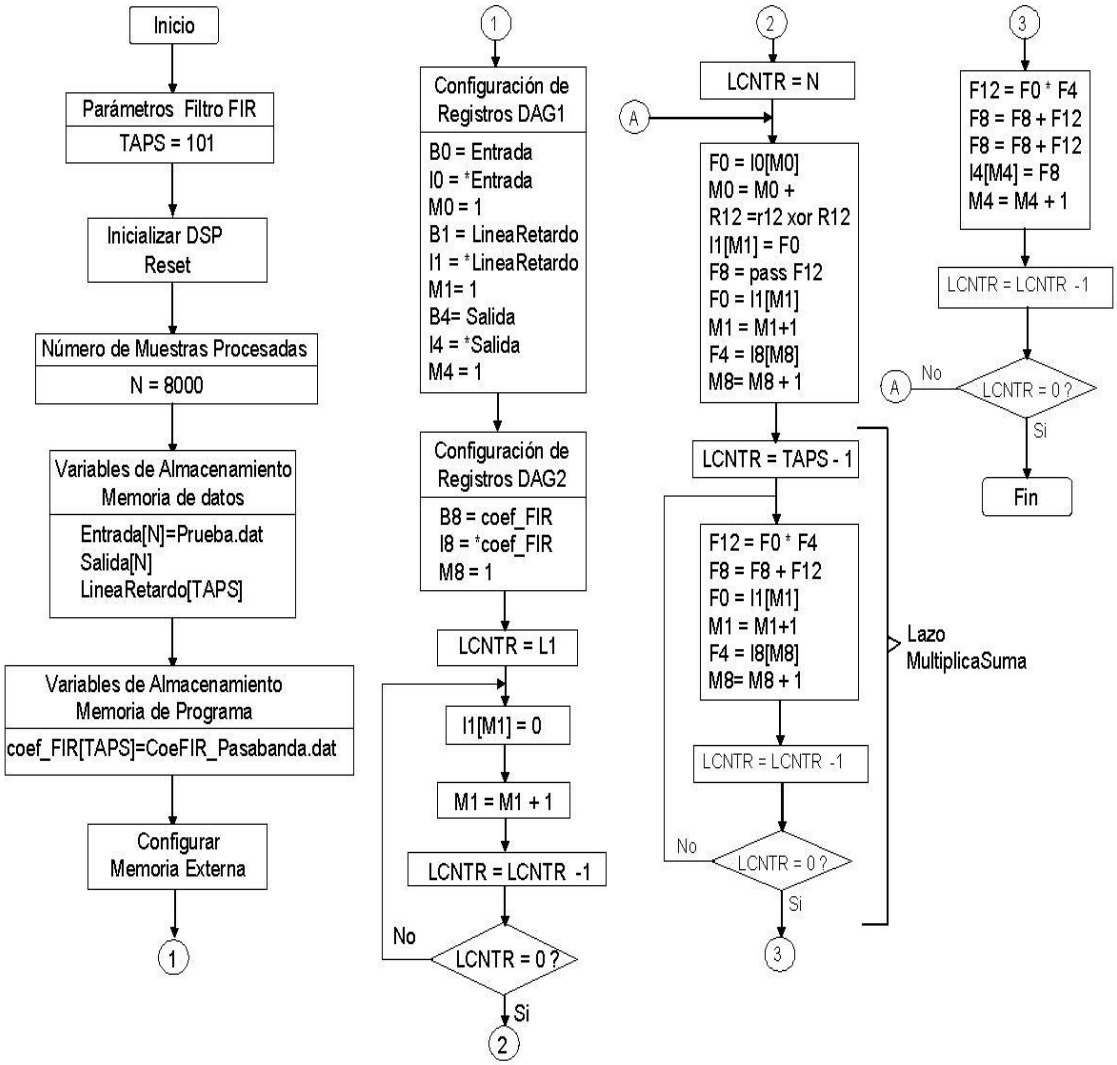
Figura 46. Buffers circulares linea de retardo y coeficentes del filtro FIR



Fuente: ANALOG DEVICES. ADSP-21000 Familiy Application Handbook Volume 1

El lazo MultiplicaSuma calcula la suma de productos según la ecuación 23, se ejecuta Taps-1 veces y usa eficientemente la capacidad multifunción de la arquitectura SHARC. Una instrucción multifunción puede buscar dos operandos y ejecutar una operación de multiplicación y suma en un solo ciclo. La multiplicación trabaja con operando buscados en un ciclo previo [1]. Por lo tanto, el coeficiente b_k almacenado en el registro F4 y la muestra de entrada previa x(n-k) almacenada en el registro F0, deben ser leídos la primera vez fuera del lazo de suma de productos. Finalmente el resultado de la última suma y producto fuera del lazo generan la muestra de salida y(n) en el registro de datos F8 que es almacenada en el buffer de salida. Este proceso se repite para cada muestra de entrada dentro un lazo, con un total de 8 instrucciones. En la figura 47 se muestra el diagrama de bloques edl algoritmo de simulación para el Filtro FIR. El programa completo de simulación del filtro FIR "Filtro_FIR.asm" junto con el corrspondiente archivo *linker* "Filtro_FIR.ldf" se muestra en los anexos.

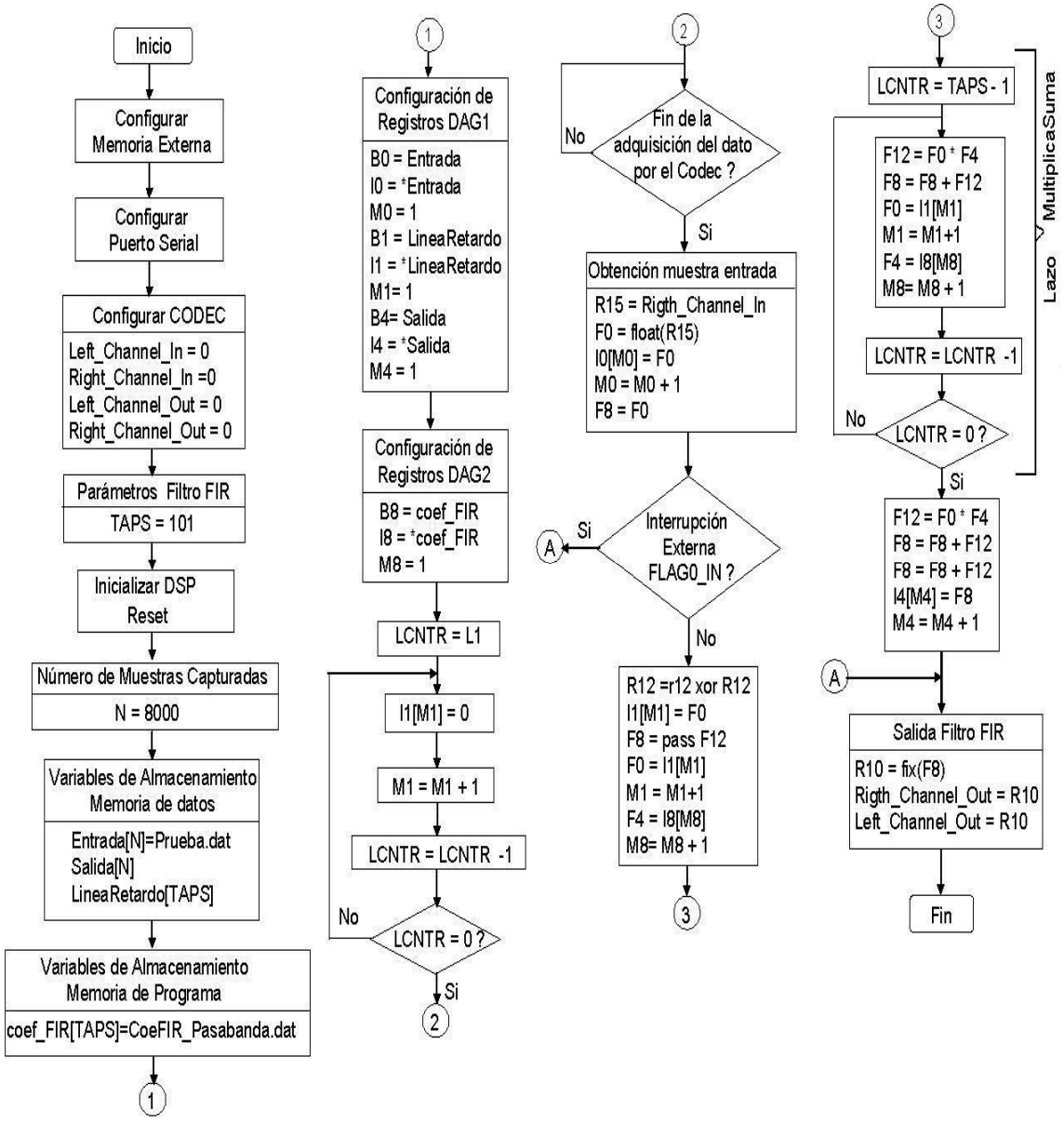
Figura 47. Diagrama de bloques algoritmo de simulación Filtro FIR



Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Implementación en tiempo real.** Realizando unos ligeros cambios al código usado en la simulación se logra implementar el algoritmo del filtro FIR en tiempo real. Estos cambios básicamente corresponden a la adquisición de la muestra de entrada desde el codec y la colocación de la muestra de salida en el codec, como se ha mencionado antes en los otros algoritmos. Adicionalmente se incluyen cinco nuevas instrucciones dentro de la rutina del filtro, que hacen referencia a usar una interrupción por hardware (Flag0), para obtener a la salida del filtro la misma señal de entrada cuando la interrupción es activada manualmente en la tarjeta. Es decir, que mientras el programa se está ejecutando, se puede mantener oprimido el botón correspondiente a Flag0, para obtener a la salida del Codec la misma señal de entrada sin filtrar. En la figura 48 se muestra el diagrama de bloques del algoritmo del Filtro FIR implementado en tiempo real. El programa completo del filtro FIR en tiempo real “FiltroFIR_TR.asm”, con la definición de variables y buffers se muestra en los anexos.

Figura 48. Diagrama de bloques algoritmo Filtro FIR en tiempo real

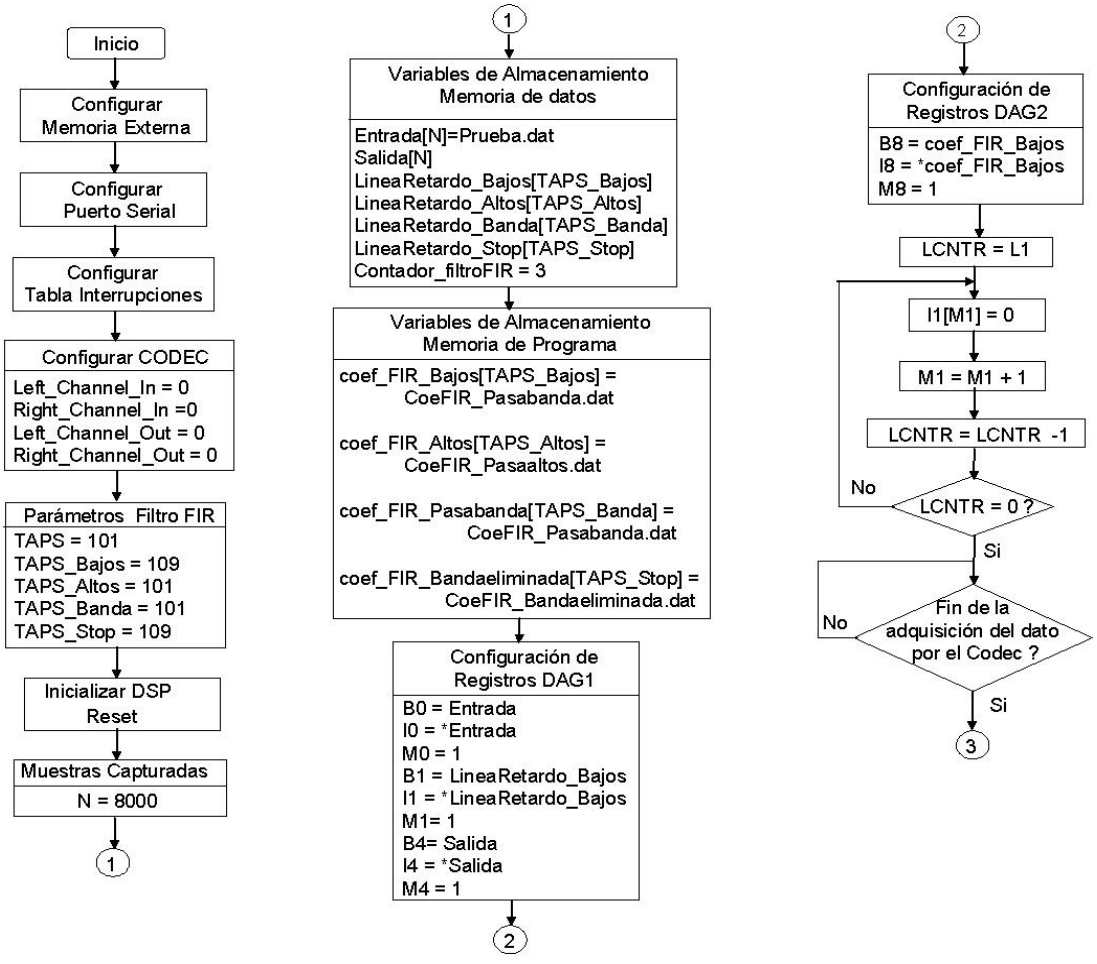


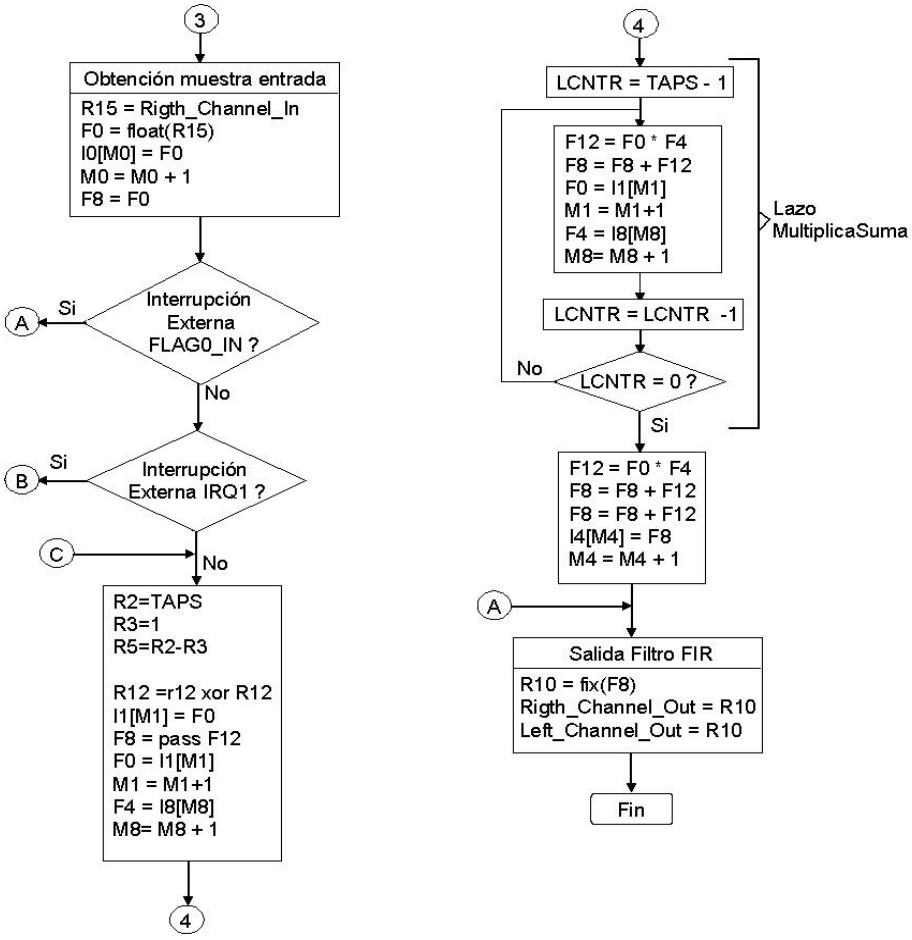
Fuente: YASSER MÉNDEZ. Autor del proyecto

Si se desea implementar diferentes filtros, pasa bajos, pasa altos, pasa banda o banda eliminada, basta con cambiar el archivo ".DAT" que contiene los coeficientes de dicho filtro y el número de TAPS; y luego volver a compilar el proyecto DPJ para que genere un nuevo archivo DXE y pueda ser ejecutado por el Debugger. Sin embargo, aprovechando la posibilidad de utilizar una rutina de interrupciones IRQ1 del ADSP-21065L se pueden implementar estos cuatro filtros en un solo programa, agregando algunas instrucciones y la rutina de interrupción al programa anterior. Entonces, se definen los taps para cada tipo de filtro, TAPS_Bajos, TAPS_Altos, TAPS_Banda y

TAPS_Stop. Se crea la variable TAPS en la memoria de datos que guarda el número de taps según el tipo de filtro. Se definen los buffers circulares para las líneas de retardo, una para cada tipo de filtro, cada una con longitud diferente, LineaRetardo_Bajos[TAPS_Bajos],LineaRetardo_Altos[TAPS_Altos], LineaRetardo_Banda[TAPS_Banda] y LineaRetardo_Stop[TAPS_Stop]. También se definen cuatro buffers para guardar los coeficientes de cada tipo de filtro y de longitud diferente, coef_FIR_Bajos[TAPS_Bajos], coef_FIR_Altos[TAPS_Altos], coef_FIR_Pasabanda[TAPS_Banda] y coef_FIR_Bandaeliminada[TAPS_Stop]. Cada una de estos buffers tiene asociado un archivo ".DAT" que contiene los coeficientes de cada filtro. Sin embargo solo se inicializa un buffer circular para la línea de retardo que guarda las muestras previas de entrada y un buffer circular para guardar los coeficientes del filtro. Esto se debe a que sólo se usa un tipo de filtro a la vez y no simultáneamente los cuatro filtros. De esta manera se definen ocho variables, para las líneas de retardo y los coeficientes del filtro, pero solo se implementan dos buffers circulares, que se inicializan por defecto como el filtro pasa bajos. También se define una variable contador, Contador_filtroFIR, que determina el tipo de filtro que se usa. El contador se inicializa con el valor de 3 que corresponde al filtro pasa bajos. En la figura 49 se muestra el diagrama de bloques para el algoritmo del filtro FIR en tiempo real con rutina de interrupciones para cambiar el tipo de filtro.

Figura 49. Diagrama de bloques algoritmo Filtro FIR con selección de tipo de filtro en tiempo real





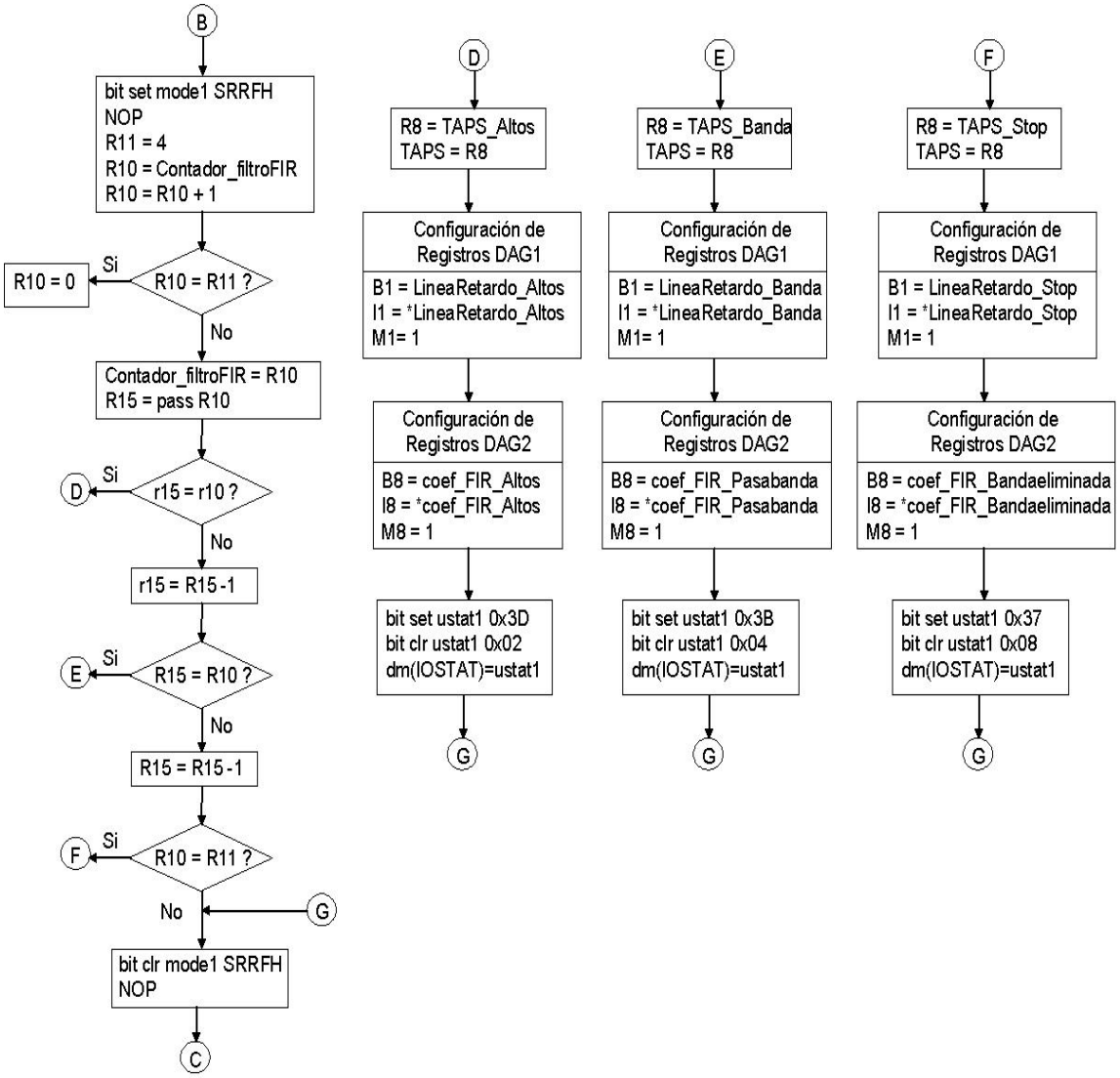
Fuente: YASSER MÉNDEZ. Autor del proyecto

En la rutina del filtro FIR se introducen dos nuevas líneas, que consisten en: leer desde memoria la variable TAPS y guardar este valor en el registro “R2; y calcular TAPS-1 y guardar este valor en el registro “R5”, para luego ser usado en el lazo de control de la suma de productos. El resto de la rutina es la misma que la de simulación, al igual que la inicialización de buffers.

La rutina de interrupción IRQ1, etiquetada con el nombre “Cambio_coeficientes_filtroFIR” inicia habilitando los registros de datos segundarios R8 a R15 del Registro del Sistema MODE1, mediante la instrucción “*bit set*” [4], seguido de la instrucción “NOP”, debido al ciclo de latencia que requiere el registro MODE1 para ser escrito [13]. Luego se define el número de interrupciones que se hacen, es decir el número de tipos de filtro que se implementan; se lee el valor contador del filtro0; se incrementa el contador, se compara este valor con el valor máximo y si es igual se vuelve a al valor inicial; se salva el valor del contador en la variable Contador_filtroFIR, de la memoria de datos. Posteriormente se realiza un chequeo del valor del contador, cada vez que se presiona el botón IRQ1 de la tarjeta de evaluación, generando la interrupción y preparando el programa para ejecutar el tipo de filtro. Para cada tipo de filtro seleccionado se realiza un salto a su subrutina correspondiente. En la figura 50 se muestra el diagrama de bloques de la subruina de interrupciones IRQ1.

Esta subrutina realiza las siguientes operaciones: carga en el registro “R8” el valor del número de taps del filtro seleccionado y luego traslada ese valor a la variable TAPS, para ser usada en la rutina principal del filtro; se inicializan los buffers de la línea de retardo y coeficientes del filtro escogido; se modifican los bits del Registro de Código Universal USTAT1 [21], que activan y desactivan las banderas de software de propósito general o de salida (LEDS, Flags 4, 5, 6 y 7) según el filtro que se está ejecutando; finalmente se hace un salto a la salida de la subrutina de interrupciones. La subrutina de interrupciones IRQ1 finaliza deshabilitando los registros de datos segundarios R8 a R15 del Registro del Sistema MODE1, mediante la instrucción “*bit clr*” [4], y retornado a la rutina principal del filtro. El progama completo del filtro FIR en tiempo real con rutina de interrupciones “FiltroFIR_TR_IRQ1.asm”se muestra en los anexos.

Figura 50. Diagrama de bloques subrutina interrupciones IRQ1 Filtro FIR en tiempo real



Fuente: YASSER MÉNDEZ. Autor del proyecto

La subrutina de interrupciones IRQ1 es referenciada mediante la directiva *.GLOBAL* para ser luego invocada en el archivo de la tabla de interrupciones (Tabla_Interupciones.asm). Por lo tanto este archivo debe ser modificado para que la rutina de ininterrupción tenga efecto. El archivo que contiene el vector de interrupciones invoca la rutina de interrupciones IRQ1, "Cambio_coeficientes_filtroFIR", mediante la directiva *.EXTERN*.

```
.EXTERN        Cambio_coeficientes_filtroFIR;        // Nueva línea
```

Esta etiqueta debe ser ubicada en la dirección en memoria de la interrupción IRQ1, *0x1Ck,* en una de las cuatro posiciones disponibles para la interrupción [15]:

```
/* 0x1C - IRQ1 Interrupt Service Routine (ISR)  */
irq1_svc:      jump Cambio_coeficientes_filtroFIR;    RTI;    RTI;    RTI;
```

Los otros archivos necesarios para ejecutar el filtro FIR en la tarjeta de evaluación, no se modifican y se pueden ver en los anexos.

3.4.2 Implementación de filtros IIR. Los filtros de Respuesta al Impulso Infinita (IIR) son muy utilizados cuando la rapidez es necesaria la característica de fase lineales aceptable. Los filtros IIR son computacionalmente más eficientes que los filtros FIR ya que requieren pocos coeficientes por el hecho de usar retroalimentación o polos. Sin embargo, esta retroalimentación puede resultar en filtro inestable si los coeficientes se desvían de sus valores verdaderos. Esto puede pasar durante el escalado o cuantización. Un filtro IIR puede ser expresado por la ecuación 24:

$$y(n)=\sum_{k=0}^{N} b_k\, x(n-k)+\sum_{k=0}^{M} a_k\, y(n-k) \qquad \text{Ecuación 24}$$

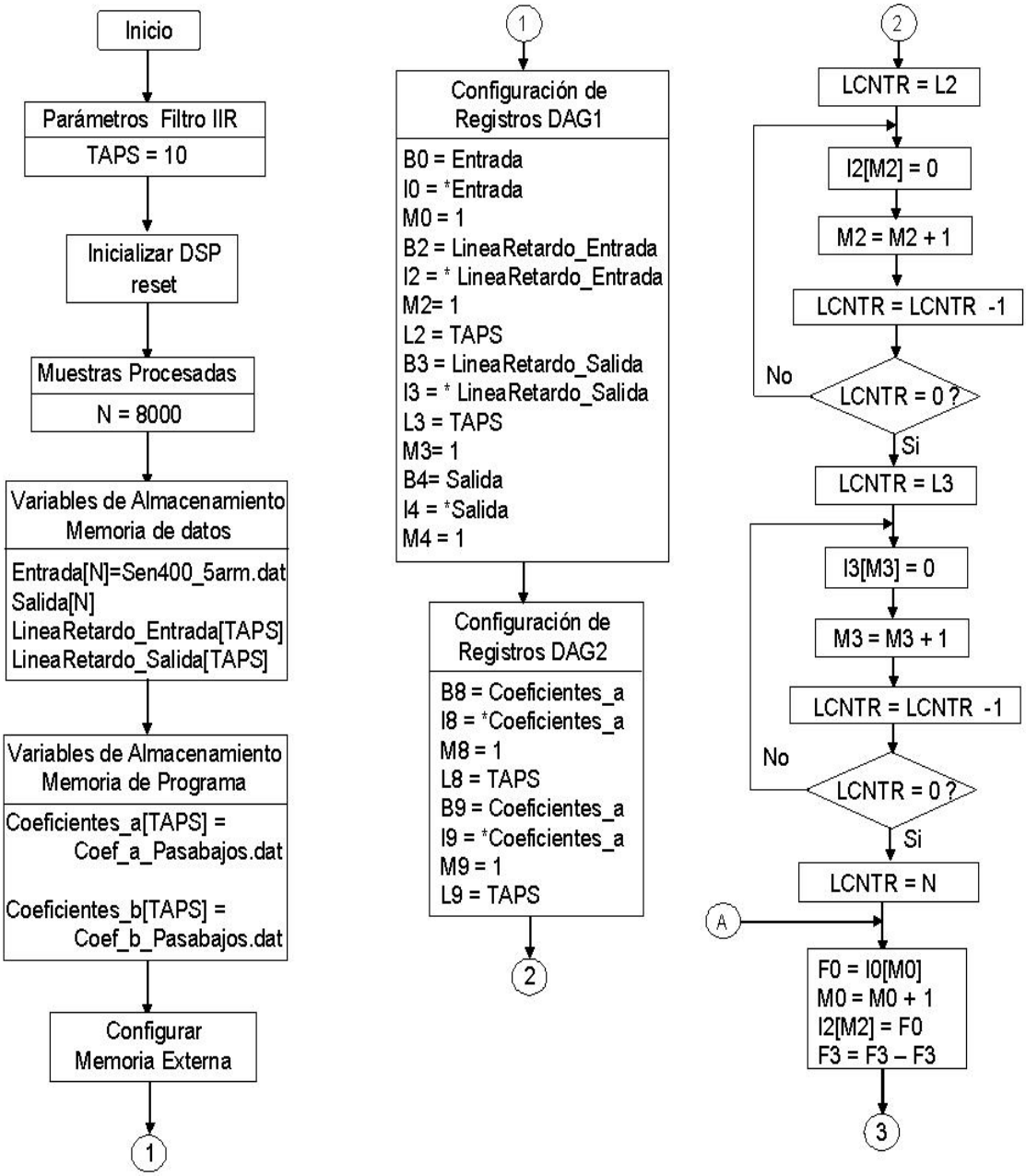
donde a_k y b_k son los coeficientes del filtro, y(n-k) representa la retroalimentación y x(n-k) la entrada desplazada en el tiempo. La ecuación 24 se conoce como estructura de Forma Directa I.

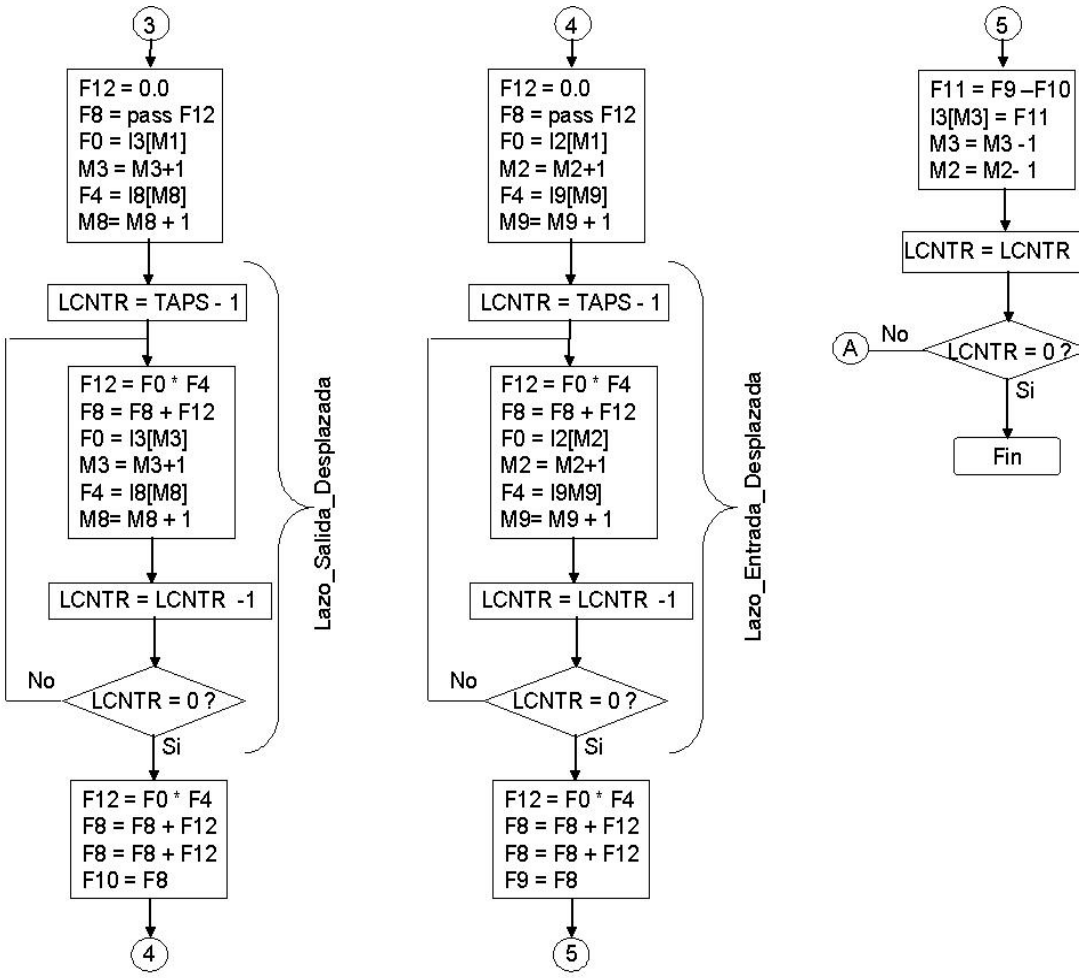
Los coeficientes pueden ser calculados por varios métodos: aproximación de derivadas, invarianza impulsional, transformación bilineal, transformación z adaptada, o la transformación en frecuencia [41]. Estos métodos parten de filtros analógicos Butterworth, Chebyshev, Elípticos y Bessel. Mediante las funciones (BUTTER, BESSELF, ELLIP, CHEBY1 y CHEBY2) de la *toolbox* de procesamiento de señales de Matlab se pueden calcular los coeficientes del filtro IIR a implementar, especificando el orden del filtro y las características de la respuesta en frecuencia que se desea obtener.

• **Simulación.** La implementación del filtro IIR en la Forma Directa I es similar a la implementación del filtro FIR. El algoritmo de simulación requiere de cuatro buffers circulares: uno para la línea de retardo que almacena la muestras previas de la señal de entrada, LineaRetardo_Entrada[TAPS]; uno para la línea de retardo que almacena la muestras previas de la señal de salida, LineaRetardo_Salida[TAPS], definidos en la memoria de datos; dos buffers circulares definidos en la memoria de datos de programa, que almacenan los coeficientes del filtro a_k y b_k,

Coeficientes_a[TAPS] y Coeficientes_b[TAPS] respectivamente. Se definen dos buffers no circulares para almacenar las muestras de entrada y salida. La longitud de los coeficientes o número de TAPS y el número de muestras de entrada y salida, N, en la memoria de datos. La inicialización de los buffers sigue la estructura descrita en los algoritmos anteriores. En la figura 51 se muestra el diagrama de bloques del algoritmo de simulación del filtro IIR.

Figura 51. Diagrama de bloques algoritmo de simulación Filtro IIR



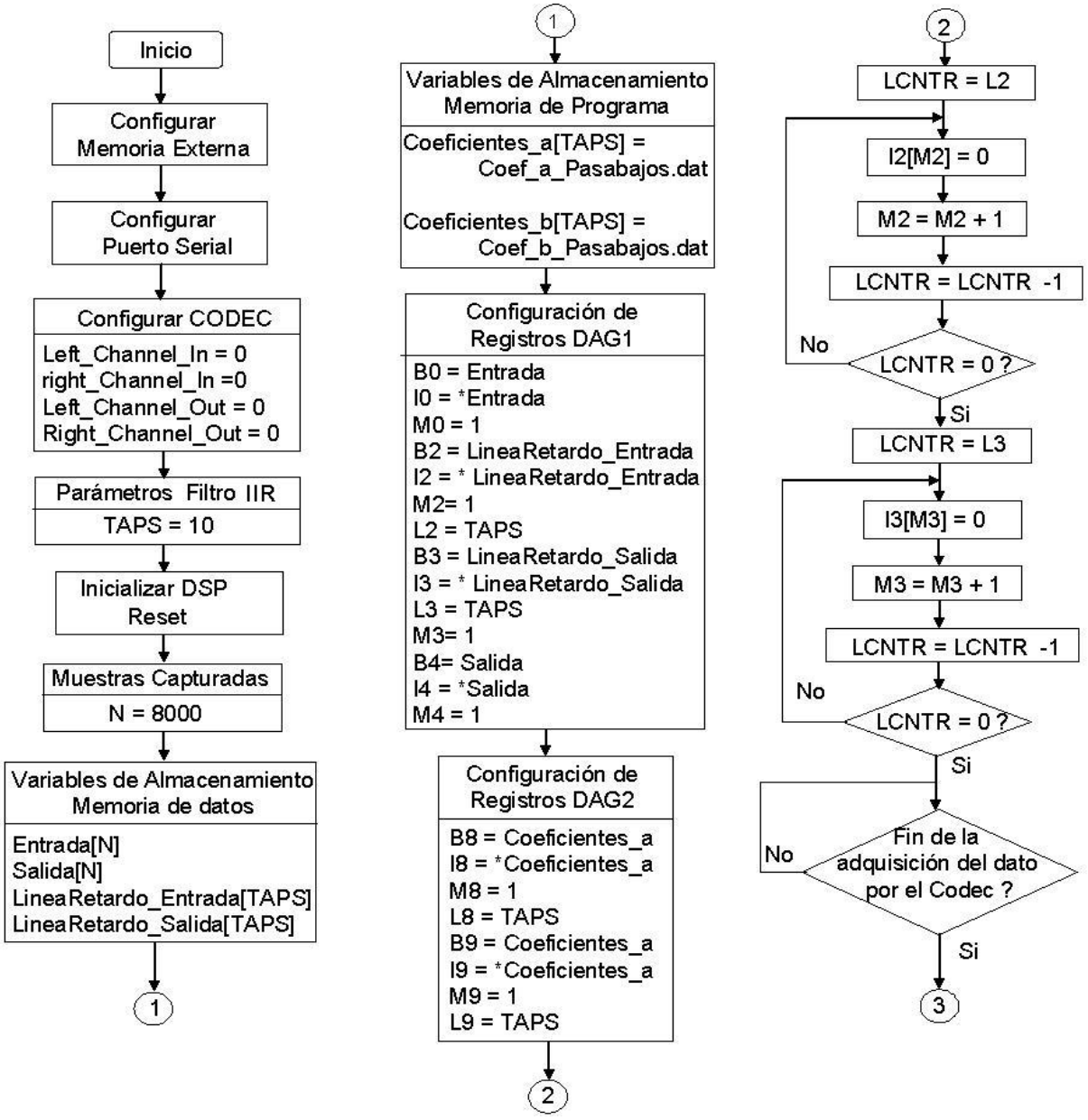


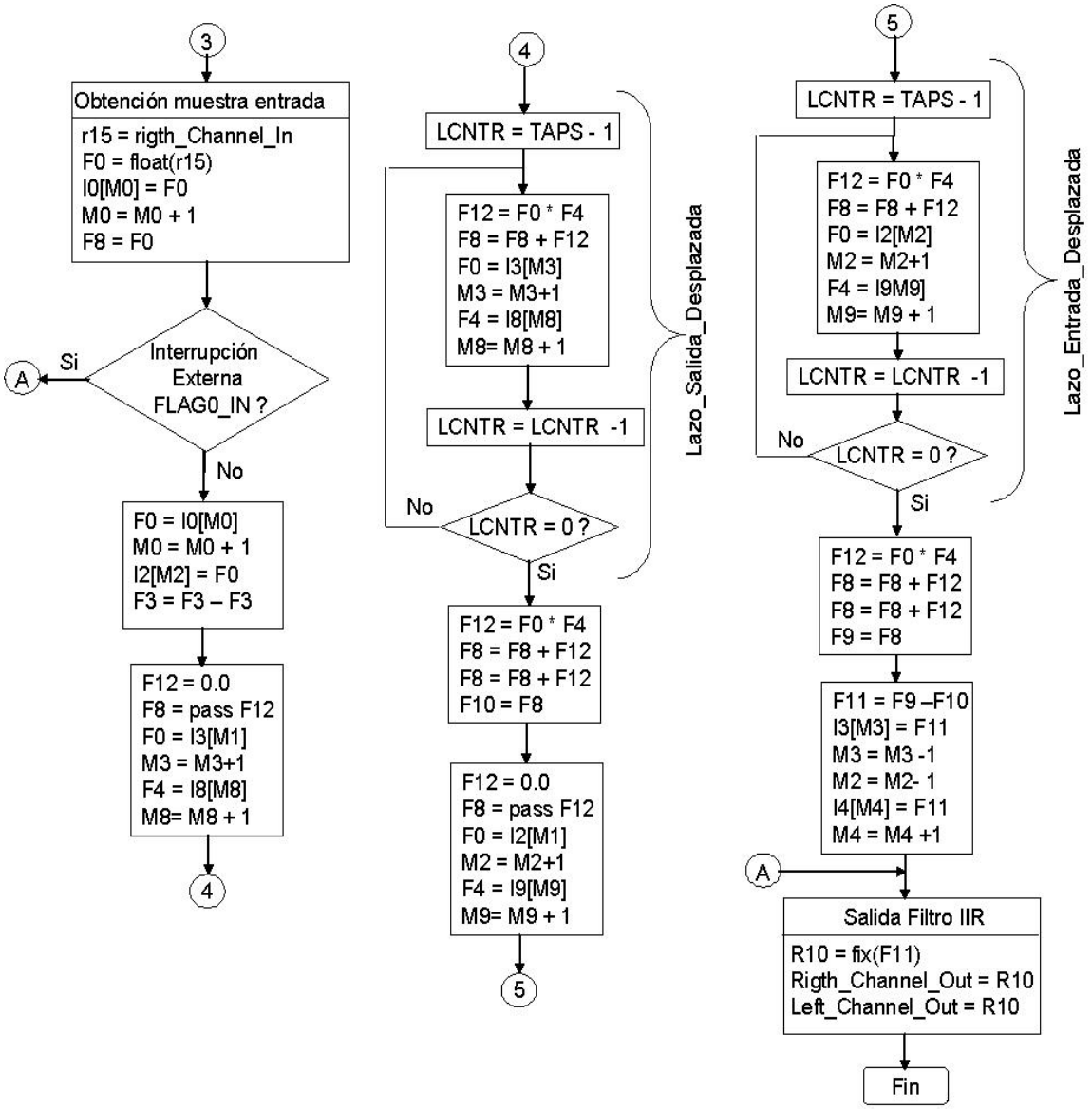
Fuente: YASSER MÉNDEZ. Autor del proyecto

La rutina del filtro IIR en la Forma Directa I se ejecuta en un lazo de 23 instrucciones para las N muestras de entrada. La rutina inicia obteniendo del buffer de entrada de la memoria de datos la muestra de entrada x(n) mediante el registro F15 y pasándola al buffer de la línea de retardo de entrada sin actualizar el puntero. Puesto que el coeficiente a_0=1, este coeficiente es redundante y realmente no es usado en el cálculo, ya que siempre resultan multiplicaciones por cero; por lo tanto se coloca en cero la primera posición de la línea de retardo de salida sin actualizar el puntero. Luego se borran los registros F12 y F8, que guardan los resultados de los productos entre coeficientes a_k y muestras previas de la línea de retardo de salida *y(n-k)* y la suma de estos productos; y usando una multifunción se obtienen el primer coeficiente de salida y la primera muestra de salida previa. El lazo que calcula esta suma de productos es llamado "Lazo_Salida_Desplazada" y se ejecuta TAPS-1 veces. Fuera de este lazo se realiza el último producto y la última suma y se guarda este resultado en el registro F10 para ser usado posteriormente. Este procedimiento se repite para la suma de productos de entradas previas y el resultado se guarda en el registro F9. Finalmente se calcula la muestra de salida en el registro F11 y se almacena en el buffer de la línea de retardo de salida, modificando el puntero mediante un decremento. Finalmente se modifica el puntero de la línea de retardo de entrada mediante un decremento y se guarda la muestra de salida y(n) en el buffer de salida modificando el puntero.

• **Implementación en tiempo real.** es similar al usado en la simulación con los correspondientes cambios para la lectura y escritura de las muestras de entrada y salida, respectivamente, en el Codec; y las instrucciones para usar la interrupción por hardware FLAG0, que colocan la muestra de entrada x(n) directamente a la salida del Codec, sin realizar el filtrado; así como se explica en la implementación en tiempo real del filtro FIR. En la figura 52 se muestra el diagrama de bloques del algoritmo del filtro IIR en tiempo real. El programa completo del filtro IIR en tiempo real, "FiltroIIR_TR.asm", se muestra en los anexos.

Figura 52. Diagrama de bloques algoritmo Filtro IIR en tiempo real





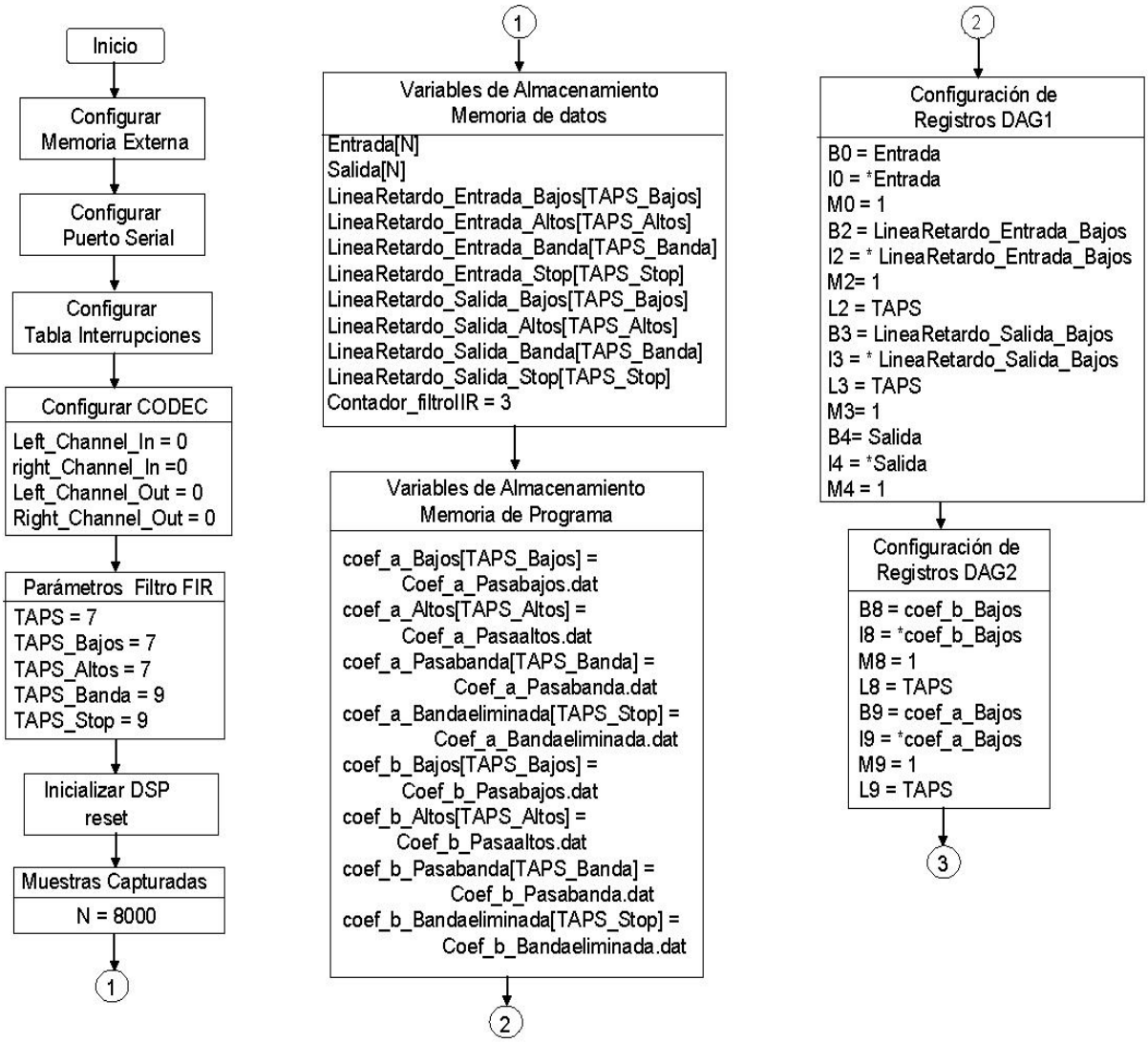
Fuente: YASSER MÉNDEZ. Autor del proyecto

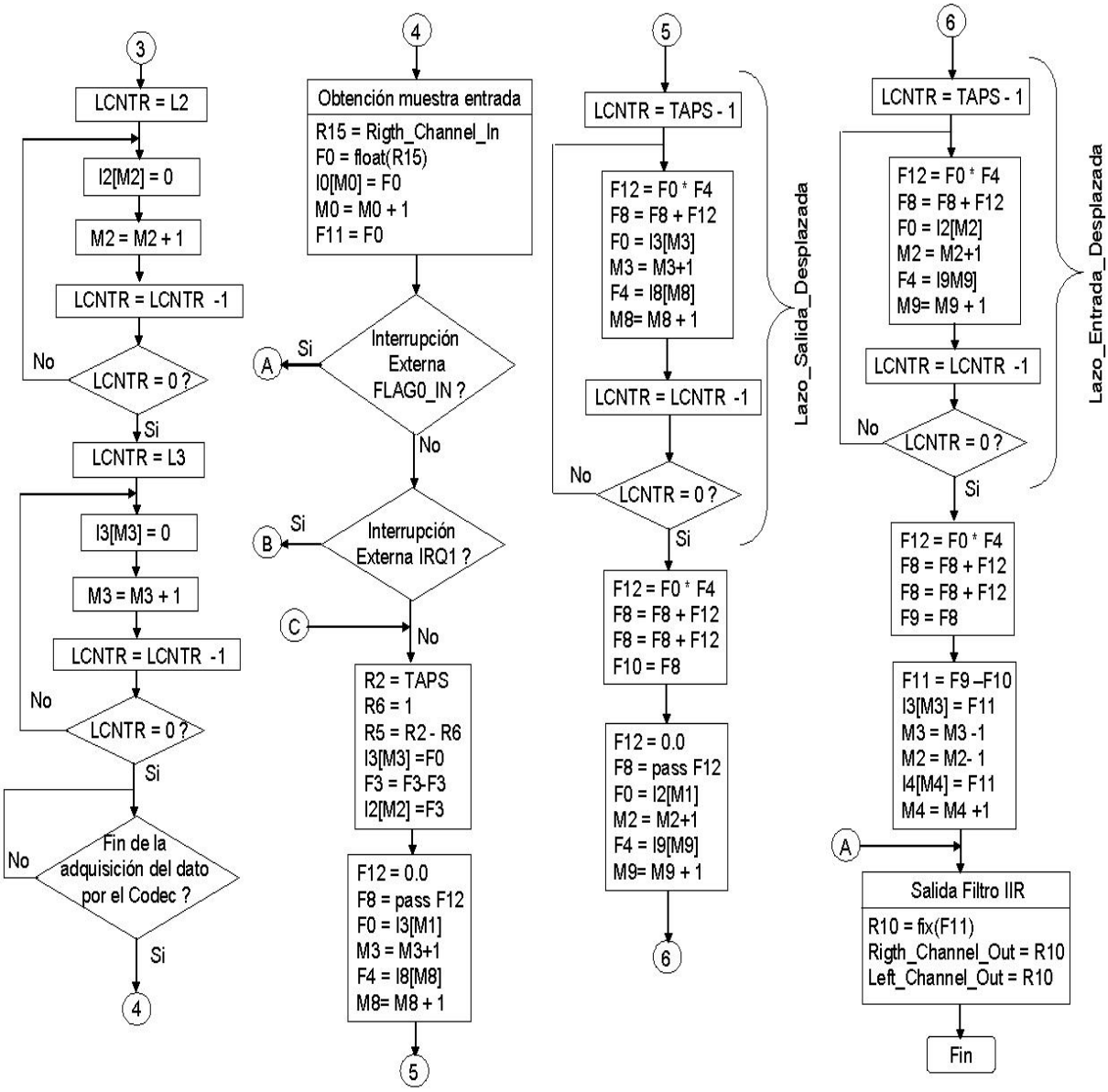
Cambiando el contenido de los archivos *"Coefientes_a.dat"* y "*Coeficientes_b.dat"* se puede implementar un filtro pasa bajos, pasa altos, pasa banda o banda eliminada. Estos cuatro tipos de filtro pueden ser imlementados en un solo programa adicionando una subrutina reinterrupción IRQ1 como se hizo con el filtro FIR. Así, se defininen en las variables TAPS_Bajos, TAPS_Altos, TAPS_Banda y TAPS_Stop, el número de coeficientes para cada uno de los filtros. Para cada tipo de filtro se requiere de una línea de retardo de entrada, una línea de retardo de salida, y dos buffers circulares que contiene los coeficientes del filtro. También se crea la variable TAPS en la memoria de datos que guarda el número de taps según el tipo de filtro y Contador_filtroIIR, que determina el tipo de filtro que se usa y se inicializa con el valor de 3 que corresponde al filtro IIR pasa bajos.

La rutina del filtro IIR en tiempo real con selección del tipo de filtro, adiciona tres nuevas líneas correspondientes a leer desde memoria la variable TAPS y guardar este valor en el registro "R2"; y calcular TAPS-1 y guardar este valor en el registro "R5", para luego ser usado en los lazos de control de la suma de productos. El resto de la rutina es la misma que para el filtro IIR en tiempo real descrita anteriormente. En la figura 53 se muestra el diagrama de bloques del algoritmo del filtro IIR en tiempo real con selección del tipo de filtro.

La subrutina de interrupción IRQ1 que cambia los coeficientes del filtro para que funcione como pasa bajos, pasa altos, pasa banda o banda eliminada, se llama *"Cambio_coeficientes_filtroIIR"* e inicia habilitando los registros segundarios R8 a R15; luego chequea el valor del contador, *Contador_filtroIIR*, y realiza un salto a la subrutina del tipo de filtro correspondiente.

Figura 53. Diagrama de bloques algoritmo Filtro IIR en tiempo real con selcción del tipo de filtro





Fuente: YASSER MÉNDEZ. Autor del proyecto

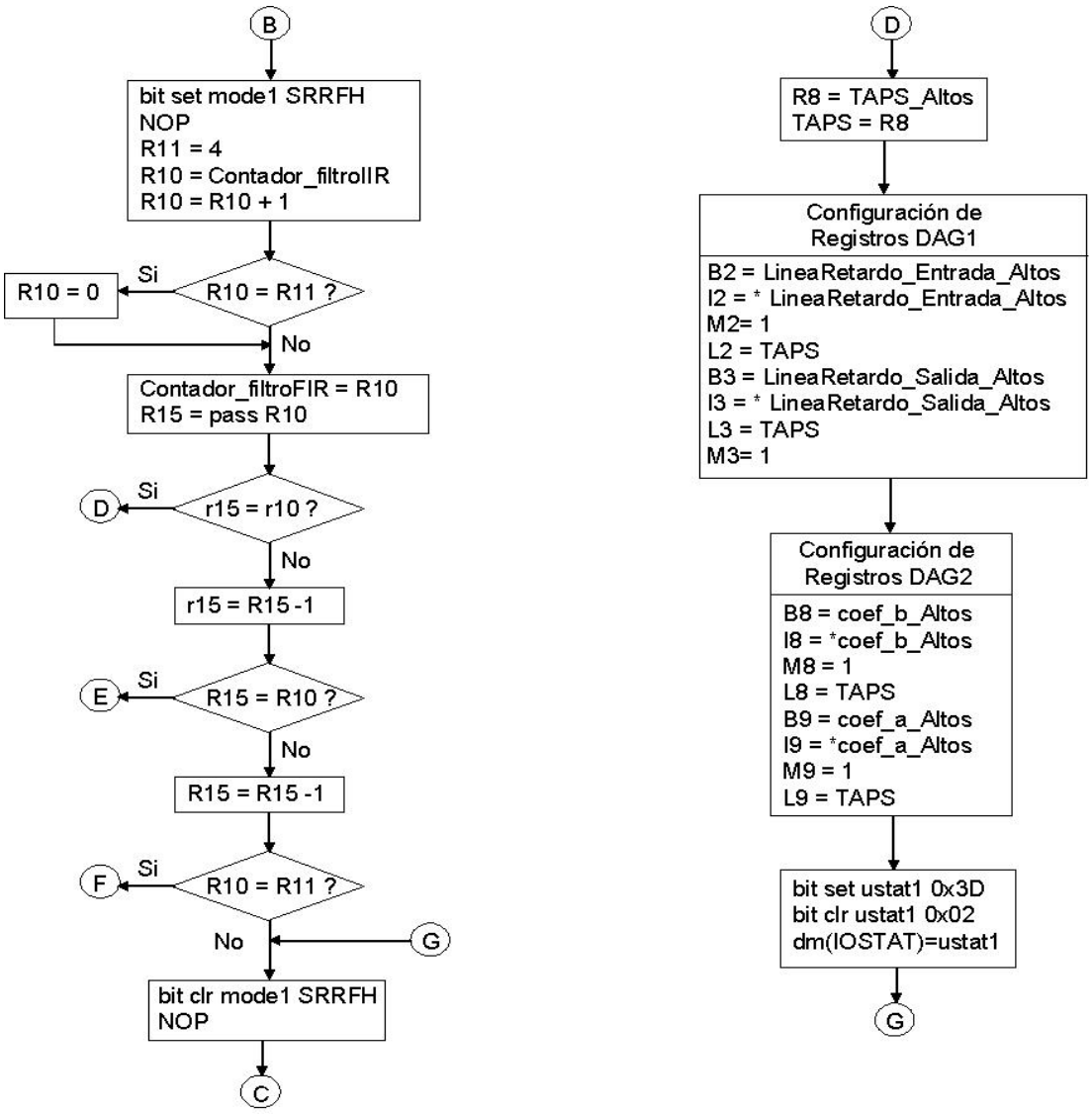
Las cuatro subrutinas, una por cada tipo de filtro contienen 14 instrucciones: Primero lee el valor del número de taps del filtro seleccionado y lo guarda el la variable TAPS. Después inicializa los buffers circulares de las líneas de retardo de entrada y salida, definiendo su longitud, según el tipo de filtro, mediante los registros B2, L2 y B3, L3 del generador de direcciones automáticas en la memoria de datos (DAG1); también inicializa los buffers circulares que almacenan los coeficientes del filtro a_k *y* b_k, mediante registros B8, L8 y B9, L9 del generador de direcciones automáticas en la memoria de programa (DAG2). Posteriormente modifica los bits del registro del sistema, USTAT1, que activan y desactivan las banderas de salida o leds (Flags 4, 5, 6 y 7) según el filtro que se está ejecutando. Finalmente se guarda el valor del registro USTAT1 en el Registro de Estado del Puerto Programable I/O (IOSTAT) y se realiza un salto al fin de la subrutina de interrupción. En la figura

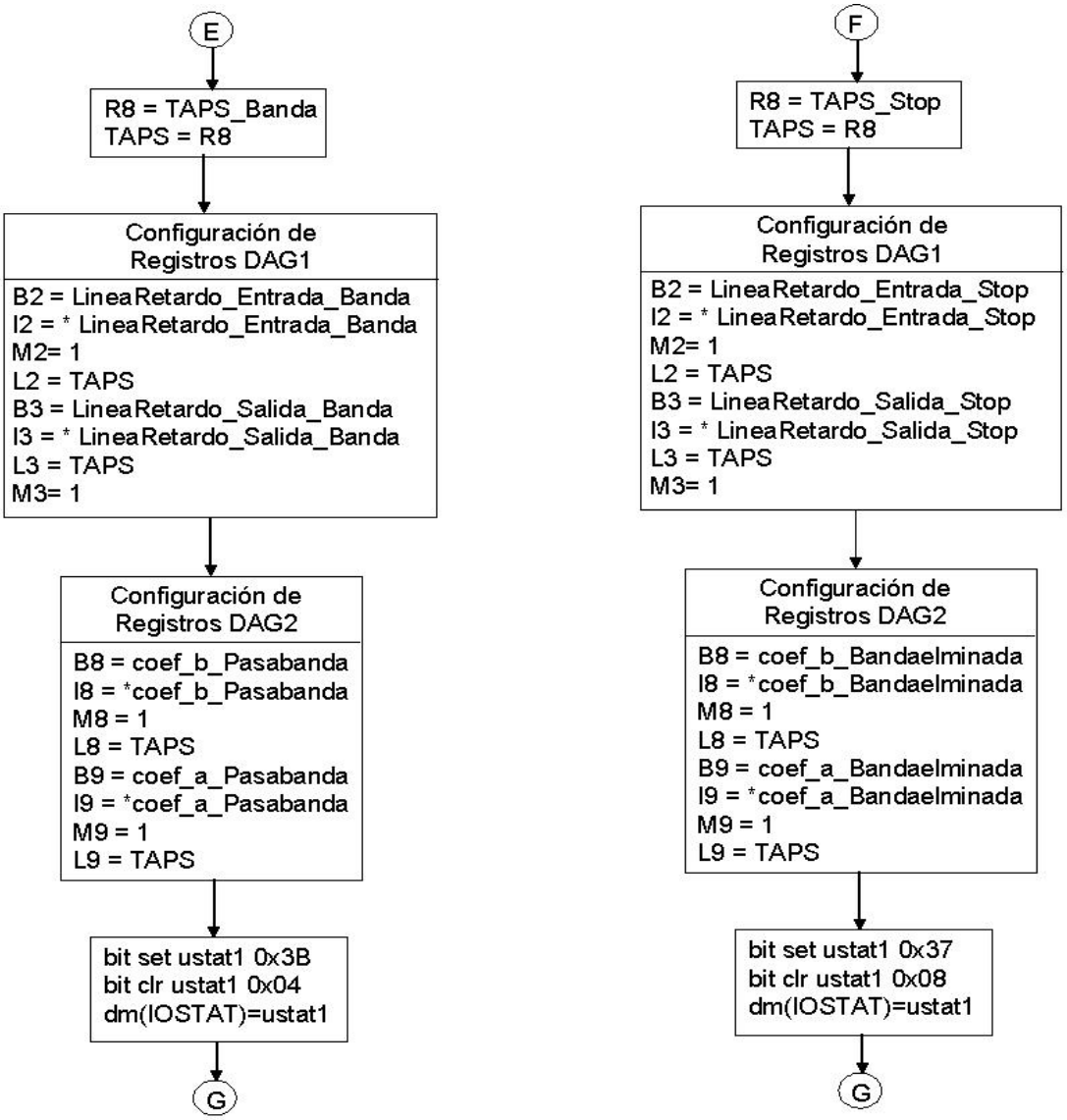
54 se muestra el diagrama de bloques de la subrutina de interrupción IRQ1 del filtro IIR en tiempo real.

El archivo de la tabla de interrupciones “Tabla_Interupciones.asm”, debe ser modificado para que la rutina de interrupción tenga efecto. El archivo que contiene el vector de interrupciones invoca la rutina de interrupciones IRQ1, “Cambio_coeficientes_filtroIIR”, mediante la directiva *.EXTERN*:

.EXTERN Cambio_coeficientes_filtroIIR; // Nueva línea

Figura 54. Diagrama de bloques subrutina interrupciones IRQ1 Filtro IIR en tiempo real





Fuente: YASSER MÉNDEZ. Autor del proyecto

Esta etiqueta debe ser ubicada en la dirección en memoria de la interrupción IRQ1, *0x1Ck,* en una de las cuatro posiciones disponibles para la interrupción [15]:

irq1_svc: jump Cambio_coeficientes_filtroIIR; RTI; RTI; RTI;

Los otros archivos necesarios para ejecutar el filtro IIR en la tarjeta de evaluación, no se modifican y se pueden ver en los anexos, donde también se encuentra el progama completo del filtro IIR en tiempo real con rutina de interrupciones “FiltroIIR_TR_IRQ1.asm”.

4. EVALUACIÓN DE LOS ALGORITMOS IMPLEMENTADOS EN EL DSP

La evaluación de los efectos de audio puede ser vista desde una perspectiva subjetiva y otra objetiva. Subjetiva, si la señal de salida de un efecto se compara auditivamente con una señal conocida de antemano como la señal obtenida por el algoritmo de simulación de Matlab. Objetiva, si se tratan las señales de entrada y salida de cada efecto de una manera cualitativa, observando las formas de onda de estas señales y comparando con patrones conocidos, bajo los mismos parámetros especificados por el efecto de audio examinado; y cuantitativa, si a partir de estas formas de onda se determina valores que permitan medir la calidad de las señales, como la relación señal a ruido (SNR) o el error cuadrático medio (Erms). La validación de los efectos de audio implementados en DPSs no está definida por parámetros establecidos, sin embargo aquellos proyectos realizados en este campo presentan sus conclusiones a partir de comparaciones cualitativas con simulaciones en matlab [32], [33], [42]; otros, mediante comparación de los valores obtenidos para las muestras de salida de los algoritmos de simulación [36] presentan la validez de los mismos; también, de manera subjetiva se describe el cambio auditivo que se manifiesta al aplicarse un efecto determinado a un sonido y como estos cambios en el sonido son similares a los obtenidos con matlab [36], [30].

La evaluación de los algoritmos implementados en el DSP se realiza en dos etapas; una concerniente al algoritmo de simulación que permite verificar que el efecto diseñado realmente hace lo deseado; y la otra etapa, la correspondiente al efecto ejecutado en tiempo real en la tarjeta de evaluación EZ-KIT Lite 21065L. Como se muestra en el capítulo de implementación, los algoritmos de simulación no difieren mucho de los implementados para usarse con la tarjeta de evaluación; básicamente son los mismos y generalmente solo cambian en la fase de adquisición y salida de datos del Codec. Por tal razón el buen desempeño del algoritmo en la etapa de simulación garantiza en buena medida el correcto funcionamiento en tiempo real del algoritmo.

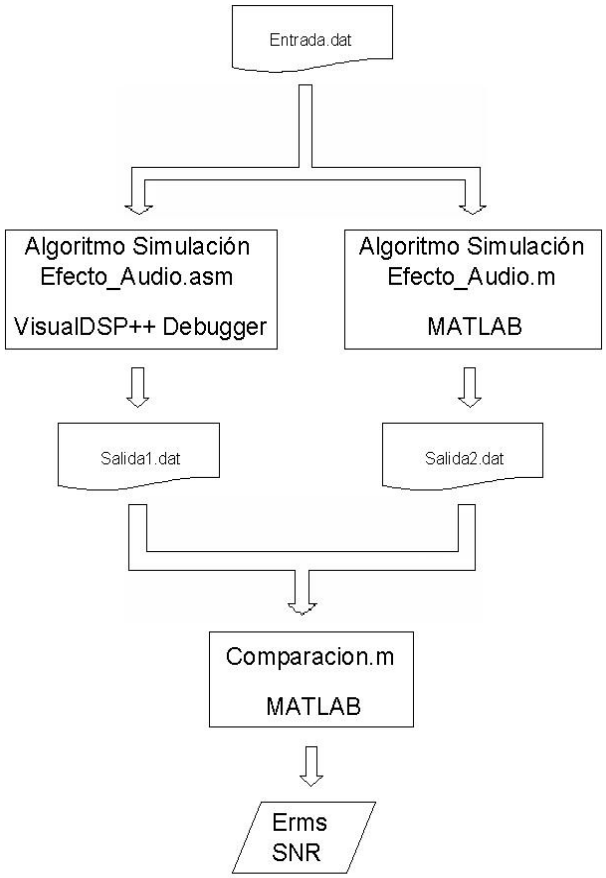
En la etapa de simulación se emplea como entrada dos ficheros ".DAT" que son almacenados en un buffer en la memoria de datos externa SDRAM del DSP. Un archivo ".DAT" corresponde a una onda sinusoidal generada por Matlab y el otro a una señal ".WAV" generada por el Cool Edit Pro 2.0 y convertida a ".DAT" con Matlab. La señal sinusoidal generada por Matlab es muestreada a 8 kHz y se almacena 1 segundo que corresponde a 8000 posiciones de memoria; la señal de voz generada por el Cool Edit Pro 2.0 es muestreada a 8 kHz y se almacena aproximadamente 4 segundos correspondientes a 34731 posiciones de memoria. La señal de salida es salvada en un buffer, en la memoria de datos del DSP, la cual es exportada a un archivo ".DAT" y se compara con la señal de salida generada por el algoritmo implementado en Matlab, utilizando la misma entrada. La comparación se hace con base en la medida de la Relación Señal a Ruido (SNR) y el Error Cuadrático Medio (Erms) según las ecuaciones 25 y 26 respectivamente.

$$SNR = 10\log\sum\left(\frac{Señal\_Calculada}{Señal\_Capturada - Señal\_Calculada}\right)^2 \qquad \text{Ecuación 25}$$

$$E_{RMS} = \sum\frac{\left(Señal\_Capturada - Señal\_Calculada\right)^2}{Número\_Muestras} \qquad \text{Ecuación 26}$$

Para la fase de simulación se almacenan en buffers no circulares en la memoria externa del DSP las 8000 muestras de la señal de entrada y la señal de salida. Estas muestras almacenadas son guardadas en un archivo ".dat" mediante el uso de la función *Dump* del *Debugger*, que convierte los datos almacenados en la memoria de datos del DSP en un fichero ".dat", especificando su longitud y formato del dato, que en este caso es punto flotante de 32 bits [24]. La señal de entrada convertida en un fichero ".dat" se usa como entrada al algoritmo hecho en Matlab y la salida del mismo se compara con la salida del DSP. Los resultados de la Relación Señal a Ruido (SNR) y Error Cuadrático Medio (Erms) se muestran en tablas para cada uno de los algoritmos implementados, teniendo en cuenta diversas configuraciones de los parámetros requeridos por cada efecto. En la figura 55 se muestra el proceso para obtener los resultados de Erms y SNR que sirven para evaluar los algoritmos de simulación.

Figura 55. Proceso de evaluación algoritmos de simulación

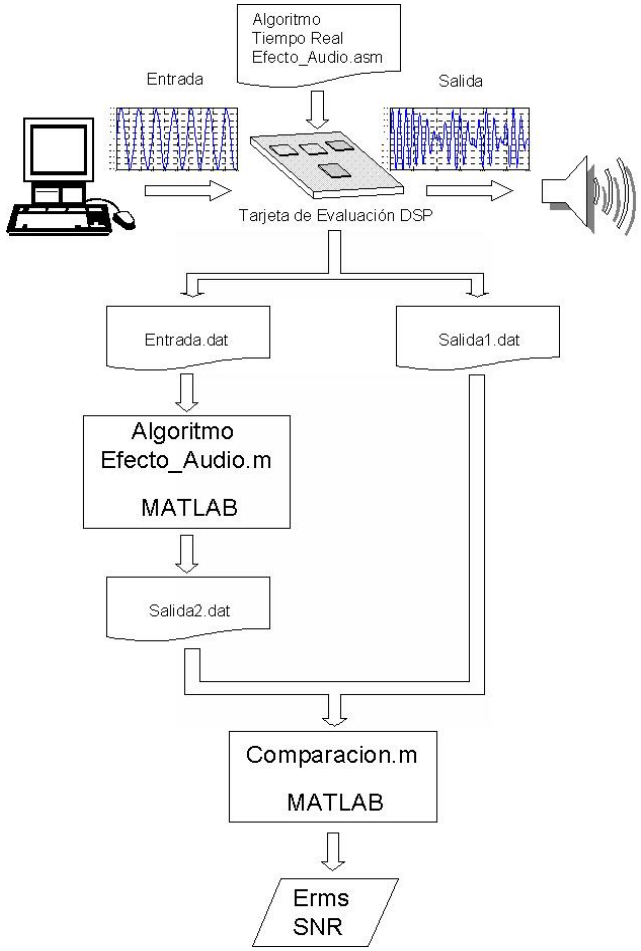


Fuente: YASSER MÉNDEZ. Autor del proyecto

En la evaluación de los algoritmos en tiempo real se emplea la tarjeta de audio del PC, que sirve como equipo generador de la señal de entrada al Codec de la Tarjeta de Evaluación. Las señales de entrada usadas son una onda sinusoidal y un archivo de voz o música muestreadas a 8 kHz, de un solo canal, de 16 bits, generada con el programa Cool Edit Pro 2.0. Se debe tener especial cuidado de controlar adecuadamente el volumen de salida de la tarjeta audio del PC y del

reproductor del archivo ".wav", para evitar saturar la señal a la entrada del Codec y originando señales erróneas. El reproductor usado es el Winamp 2.91 y el control de volumen se ajusta al 50% mientras que el de la tarjeta de sonido del PC se ajusta al 100%, lo cual permite obtener una señal de entrada almacenada en la memoria del DSP, con una amplitud aproximada de 0.5 volts pico. Las características de frecuencia y cantidad de muestras de las señales de entrada usadas para evaluar cada uno del os algoritmos se indica en su respectivo apartado. En la figura 56 se muestra el proceso para obtener los resultados de Erms y SNR que sirven para evaluar los algoritmos en tiempo real.

Figura 56. Proceso de evaluación algoritmos en tiempo real



Fuente: YASSER MÉNDEZ. Autor del proyecto

4.1 EVALUACIÓN DE LOS EFECTOS DE AUDIO BASADOS EN RETARDOS DIGITALES

4.1.1 Evaluación del efecto retardo simple. El algoritmo del retardo simple se puede describir mediante la ecuación 27, donde y(n) es la muestra de salida, x(n) es la muestra de entrada actual, x(n-D) es la muestra de entrada retrasada, b_0 y b_D son los coeficientes que representan las

ganancias de la señal de entrada y la línea de retardo respectivamente, y D es el retardo en muestras.

$$y(n) = b_0 * x(n) + b_D * x(n - D)$$ Ecuación 27

Para evaluar la fase simulación se emplean los archivos "Sen_200.dat" y "Prueba.dat" almacenados en la memoria externa del DSP, que corresponden a la señal sinusoidal y al archivo de voz respectivamente. Para la entrada sinusoidal se almacenan 8000 muestras, mientras que para el archivo de voz se almacenan 34731 muestras. Los coeficientes que modifican la señal de entrada y la línea de retardo son: b_0= 0.5 y b_D = 0.5. En la tabla 6 se muestran los valores de la Relación Señal a Ruido (SNR) y el Error Cuadrático Medio (Erms) con diferentes retardos para el algoritmo de Retardo simple o de un solo tap para cada una de las entradas.

Para la evaluación del algoritmo de Retardo Simple en tiempo real, usando la tarjeta de evaluación, se almacenan las primeras 24000 muestras de la señal de entrada, en un buffer no circular de Entrada (B0= Entrada y L0=0) y sus correspondientes 24000 muestras de salida, en un buffer circular de Salida (B4= Salida y L4= @Salida). Puesto que la señal de entrada "Sen_200.wav" es una señal periódica, se pueden utilizar sólo las primeras 8000 muestras (es decir 1 segundo) de las señales de entrada y salida, para realizar la comparación con Matlab y calcular la Relación Señal a Ruido (SNR) y el Error Cuadrático Medio (Erms).

Tabla 6. Valores SNR y Erms para retardo simple simulación

Entrada "Sen_200.dat"		
Retardo (ms)	***SNR (dB)***	***Erms (%)***
52	151.82	8.52e-018
76	153.33	3.63e-017
125	170.58	9.91e-019

Entrada "Prueba.dat"		
Retardo (ms)	***SNR (dB)***	***Erms (%)***
76	158.29	3.07e-018
125	157.86	3.17e-018
300	157.76	3.11e-018

Fuente: YASSER MÉNDEZ. Autor del proyecto

Cuando se evalúa el algoritmo con la señal de entrada "Prueba.wav" se almacenan nuevamente 24000 muestras (3 segundos) en los buffers de entrada y salida, teniendo especial cuidado de parar la ejecución del programa justo a los tres segundos, garantizando así que las muestras de salida corresponden a las muestras de entrada. Sin embargo, en el momento de guardar en un archivo ".dat" las muestras de entrada almacenadas en el buffer de entrada en la memoria externa del DSP, solo se utilizan las primera 8000 muestras En la Tabla 7 se puede observar los resultados de SNR y Erms para los mismos retardos evaluados en la simulación, 52, 76 y 96 ms, utilizando

como entrada la señal “Sen_200.wav”, y 52, 76 y 125 ms, utilizando como entrada la señal “Prueba.wav”.

Tabla 7. Valores SNR y Erms para retardo simple tiempo real

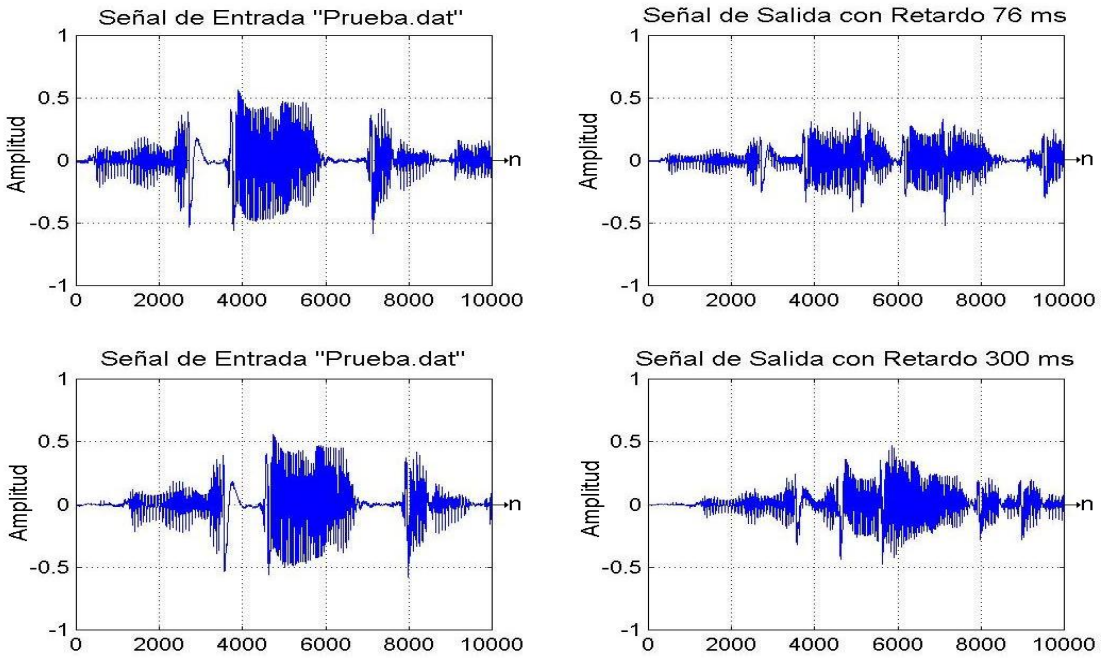
Entrada “Sen_200.wav”		
Retardo (ms)	***SNR (dB)***	***Erms (%)***
52	151.18	7.47e-018
76	151.19	7.74e-018
125	159.20	1.03e-017

Entrada “Prueba.wav”		
Retardo (ms)	***SNR (dB)***	***Erms (%)***
76	154.89	1.80e-018
125	154.64	1.81e-018
300	154.34	1.87e-018

Fuente: YASSER MÉNDEZ. Autor del proyecto

En la figura 57 se observa un segmento de 10000 muestras de la señal de entrada correspondiente al archivo de voz y la salida del algoritmo Retardo Simple para los retardos de 76 ms y 300 ms.

Figura 57. Efecto retardo simple



Fuente: YASSER MÉNDEZ. Autor del proyecto

4.1.2 Evaluación del efecto retardo múltiple. El algoritmo de Retardo Múltiple con 6 Taps o líneas de retardo se puede describir mediante la ecuación 28, donde y(n) es la muestra de salida, x(n) es la muestra de entrada actual, $x(n-D_1)$, $x(n-D_2)$, $x(n-D_3)$, $x(n-D_4)$, $x(n-D_5)$ y $x(n-D_6)$ son las muestras de entrada retrasada para cada retardo, b_0 es ganancia de la señal de entrada y b_{D1}, b_{D2}, b_{D3}, b_{D4}, b_{D5} y b_{D6} son los coeficientes de las líneas de retardo respectivamente, y D_1, D_2, D_3, D_4, D_5 y D_6 son los retardos en muestras.

$$y(n) = b_0 * x(n) + b_{D1} * x(n - D_1) + b_{D2} * x(n - D_2) + b_{D3} * x(n - D_3) + b_{D4} * x(n - D_4) + b_{D5} * x(n - D_5) + b_{D6} * x(n - D_6)$$

Ecuación 28

El algoritmo de Retardo Múltiple con 6 Taps se evalúa para un conjunto de retardos pequeños entre 20 y 50 ms, para un conjunto de retardos grandes entre 100 y 500 ms, y para un conjunto de retardos pequeños y grandes entre 20 y 300 ms. Los coeficientes de la ecuación en diferencias o ganancias de la línea de retardo y ganancia directa son los mismos para los tres conjuntos de retardos, y su valor es 0.17.

Para evaluar el algoritmo en la simulación se utilizan las entradas "Sen_200.dat" de 8000 muestras y el archivo de voz "Prueba.dat" de 34731 muestras. En la tabla 8 se muestran los valores de SNR y Erms para el algoritmo de Retardo Múltiple de 6 Taps, con coeficientes $b_0=b_{D1}=b_{D2}=b_{D3}=b_{D4}=b_{D5}=b_{D6}$= 0.17, para ambas entradas.

Tabla 8. Valores SNR y Erms para retardo múltiple 6 taps simulación

Entrada "Sen_200.dat"							
Retardos (ms)						***SNR (dB)***	***Erms (%)***
1	***2***	***3***	***4***	***5***	***6***		
22	26	32	36	42	46	146.51	1.67e-016
102	176	252	326	402	476	146.67	1.07e-016
22	36	76	102	176	298	146.85	8.58e-017

Entrada "Prueba.dat"							
Retardos (ms)						***SNR (dB)***	***Erms (%)***
1	***2***	***3***	***4***	***5***	***6***		
20	25	30	35	40	45	147.16	1.75e-017
100	175	250	325	400	475	146.90	1.39e-017
20	35	75	100	175	300	146.97	1.49e-017

Fuente: YASSER MÉNDEZ. Autor del proyecto

.

La evaluación del algoritmo de Retardo Múltiple con 6 Taps, empleando la tarjeta del DSP, requiere el almacenamiento de 24000 muestras en los buffers de entrada y salida en la memoria externa del DSP. Se utilizan como entradas las señales de audio, "Sen_200.wav" y "Prueba.wav". El procedimiento para determinarlas muestras necesarias para realizar la comparación con el algoritmo en Matlab y determinar SNR y Erms, es el mismo que el descrito para la evaluación del algoritmo de Retardo Simple. En la tabla 9 se muestran los valores de SNR y Erms para los tres conjuntos de retardos descritos anteriormente y usando como entradas las señales de audio

“Sen_200.wav” y “Prueba.wav”. Los coeficientes de la señal de entrada y las seis líneas de retardo son los mismos empleados en la simulación.

Tabla 9. Valores SNR y Erms para retardo múltiple 6 taps tiempo real

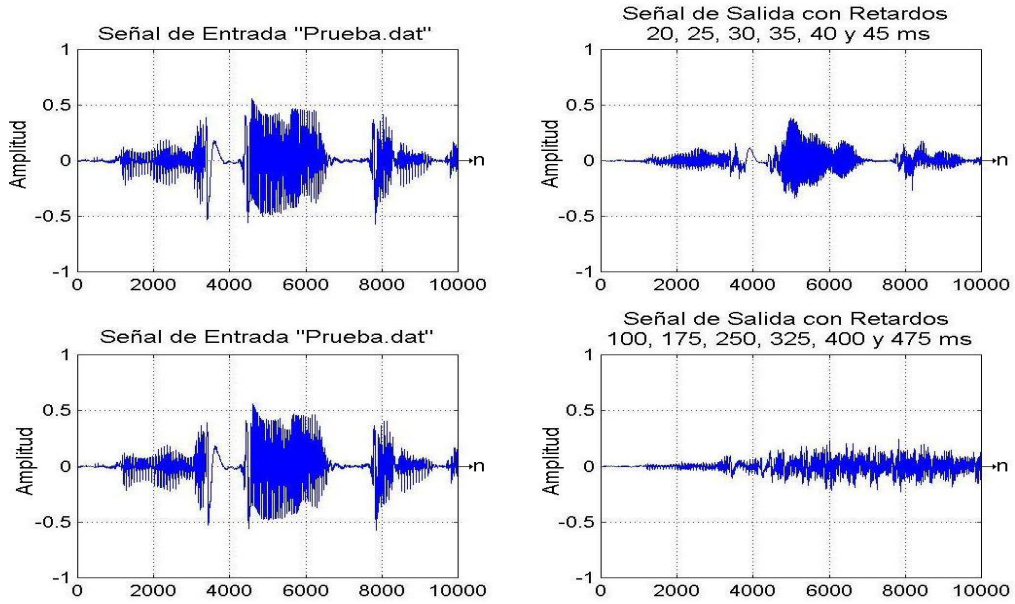
Entrada “Sen_200.wav”							
Retardos (ms)						***SNR (dB)***	***Erms (%)***
1	***2***	***3***	***4***	***5***	***6***		
22	26	32	36	42	46	146.44	1.16e-017
102	176	252	326	402	476	145.96	8.62e-018
22	36	76	102	176	298	145.37	8.46e-018

Entrada “Prueba.wav”							
Retardos (ms)						***SNR (dB)***	***Erms (%)***
1	***2***	***3***	***4***	***5***	***6***		
20	25	30	35	40	45	145.17	7.33e-018
100	175	250	325	400	475	144.85	5.63e-018
20	35	75	100	175	300	144.78	6.00e-019

Fuente: YASSER MÉNDEZ. Autor del proyecto

En la figura 58 se muestra el efecto que produce un el algoritmo de Retardo Múltiple con seis retardos cuando la entrada es el archivo de voz “Prueba.wav”, con un conjunto de retardos pequeños (20, 25, 30, 35, 40 y 45 ms) y un conjunto de retardos grandes (100, 175,250, 325, 400 y 475 ms) y con el valor de 0.17 para todos los coeficientes de la línea de entrada y las líneas de retardo.

Figura 58. Efecto retardo múltiple 6 taps



Fuente: YASSER MÉNDEZ. Autor del proyecto

4.2 EVALUACIÓN DE LOS EFECTOS DE AUDIO BASADOS EN MODULACIÓN

4.2.1 Evaluación del efecto *flanger*. El algoritmo *flanger* se describe por las ecuaciones 29, 30 y 31 donde b_0 y b_D son los coeficientes de entrada y entrada retrasada respectivamente, *D* es el retardo y f_{LFO} es la frecuencia del oscilador de baja frecuencia (LFO) y *fs* es la frecuencia de muestreo.

$$y(n) = b_0 * x(n) + b_D * x(n - d(n)) \qquad \text{Ecuación 29}$$

$$d(n) = \frac{D}{2} - \frac{D}{2} * LFO(n) \qquad \text{Ecuación 30}$$

$$LFO(n) = \cos\left(2 * pi * \frac{f_{LFO}}{f_s} * n \right) \qquad \text{Ecuación 31}$$

Para la evaluación del efecto *flanger* se varían los parámetros de retardo *D* entre 2 y 10 ms; y *Sweep Rate* que controla la frecuencia del LFO (f_{LFO}) entre 2 y 10 Hz. Las ganancias se mantienen constantes $b_0 = b_D = 0.5$; y se trabaja con una frecuencia de muestreo de 8 kHz. Los resultados de la comparación con Matlab para la simulación y tiempo real se muestran en las tablas 10 y 11 respectivamente, usando como entrada una señal sinusoidal de 250 Hz.

Tabla 10. Valores SNR y Erms para *flanger* simulación

Entrada "Sen_250.dat"		***LFO 2 Hz***
Retardo (ms)	***SNR(dB)***	***Erms (%)***
2	164.36	9.17e-018
6	163.94	1.01e-017
10	163.88	1.02e-017

Entrada "Sen_250.dat"		***LFO 6 Hz***
Retardo (ms)	***SNR(dB)***	***Erms***
2	164.34	9.20e-018
6	164.02	9.91e-018
10	163.87	1.02e-017

Entrada "Sen_250.dat "		***LFO 10 Hz***
Retardo (ms)	***SNR(dB)***	***Erms***
2	164.35	9.23e-018
6	164.00	9.97e-018
10	164.0	9.83e-018

Fuente: YASSER MÉNDEZ. Autor del proyecto

Tabla 11. Valores SNR y Erms para *flanger* tiempo real

Entrada "Sen_250.wav"		***LFO 2 Hz***
Retardo (ms)	***SNR(dB)***	***Erms (%)***
2	158.20	7.13e-018
6	157.71	8.05e-018
10	158.27	7.06e-018

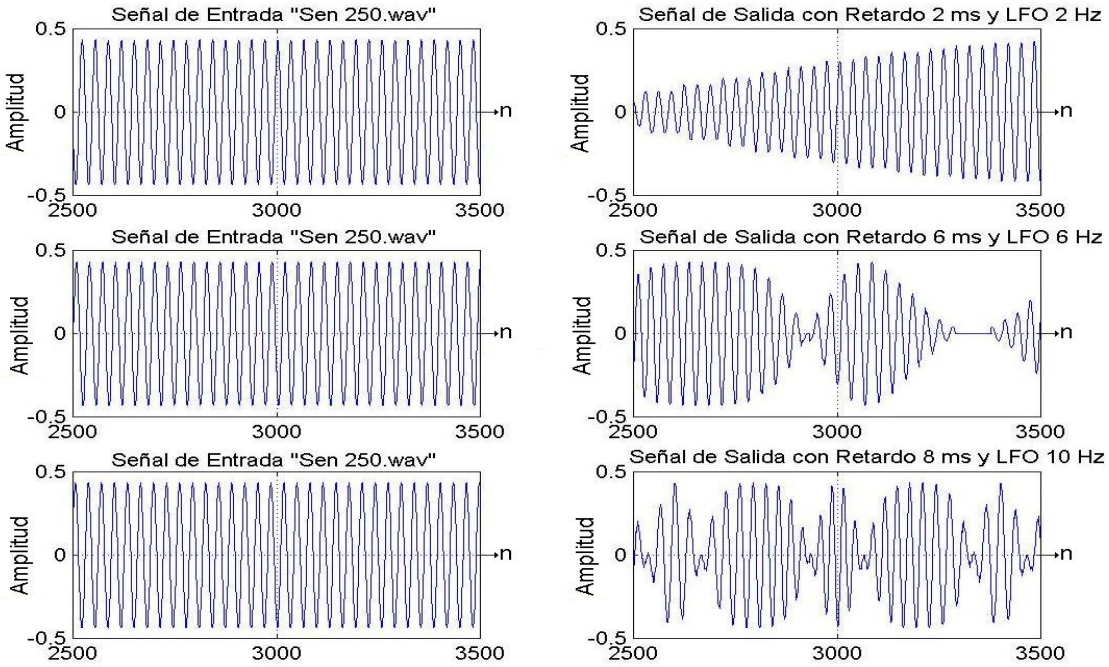
Entrada "Sen_250.wav"		***LFO 6 Hz***
Retardo (ms)	***SNR(dB)***	***Erms***
2	158.08	7.26e-018
6	157.73	7.84e-018
10	157.63	8.01e-018

Entrada "Sen_250.wav"		***LFO 10 Hz***
Retardo (ms)	***SNR(dB)***	***Erms***
2	158.35	6.93e-018
6	151.52	4.27e-018
10	151.57	4.22e-018

Fuente: YASSER MÉNDEZ. Autor del proyecto

El efecto de modulación que produce el *flanger* a la señal de entrada sinusoidal de 250 Hz se puede observaren la figura 59, para tres configuraciones de parámetros de retardo y LFO diferentes.

Figura 59. Efecto *flanger*



Fuente: YASSER MÉNDEZ. Autor del proyecto

4.2.2 Evaluación del efecto *chorus.* El efecto *chorus* agrega líneas de retardo variable en paralelo al esquema del efecto *flanger*, esto se traduce en una modulación más acentuada de la señal de entrada. Si la entrada es una onda sinusoidal, el número de picos debido a la modulación aumenta a medida que aumente la frecuencia de la señal de entrada.

Para la evaluación se mantienen fijos algunos parámetros del efecto *chorus* implementado para facilitar la toma de datos y la comparación con el algoritmo en Matlab. Estos parámetros son el coeficiente de la señal de entrada ($b_0 = 0.7$), los coeficientes de las líneas de retardo ($b_{D1} = b_{D2} = b_{D3} = 0.3$) y la amplitud del los LFOs (Sweep_Depth1 = Sweep_Depth2 = Sweep_Depth3 =1.0). Se varían los parámetros de los retardos *D1, D2 y D3* entre 15 y 35 ms; y los *Sweep Rate* que controlan la frecuencia de los tres LFOs (f_{LFO}) para variaciones entre 0.1 y 3 Hz. En la tabla 12 se muestran los resultados de la comparación con Matlab de la simulación del efecto *chorus*, usando las mismas entradas del efecto *flanger*.

Tabla 12. Valores SNR y Erms para *chorus* simulación

Entrada "Sen_250.dat" b_0=0.7 b_{D1}=b_{D2}=b_{D3}=0.3 Sweep_Depth1, 2 y 3=1.0							
LFO 1 (Hz)	***LFO 2 (Hz)***	***LFO 3 (Hz)***	***D1 (ms)***	***D2 (ms)***	***D3 (ms)***	***SNR (dB)***	***Erms (%)***
0.1	0.2	0.3	15	20	25	144.51	1.31e-015
1	2	3	25	30	35	144.84	1.32e-015
0.1	1	2	15	20	25	145.11	1.38e-015
0.1	0.5	3	15	25	35	144.90	1.32e-015

Entrada "Prueba.dat" b_0=0.7 b_{D1}=b_{D2}=b_{D3}=0.3 Sweep_Depth1, 2 y 3=1.0							
LFO 1 (Hz)	***LFO 2 (Hz)***	***LFO 3 (Hz)***	***D1 (ms)***	***D2 (ms)***	***D3 (ms)***	***SNR (dB)***	***Erms (%)***
0.1	0.2	0.3	15	20	25	145.06	9.28e-017
1	2	3	25	30	35	145.32	1.04e-016
0.1	1	2	15	20	25	145.27	1.02e-016
0.1	0.5	3	15	25	35	145.07	8.98e-017

Fuente: YASSER MÉNDEZ. Autor del proyecto

En la implementación en tiempo real se usan las mismas entradas de la evaluación del algoritmo de simulación y con la misma configuración de parámetros. En la tabla 13 se muestran los resultados de la comparación con Matlab del algoritmo en tiempo real del efecto *chorus.*

En las figura 60 se aprecian las modulaciones producidas por el efecto *chorus* con tres líneas de retardo aplicado a una onda sinusoidal, con LFO de muy baja frecuencia (0.1, 0.2 y 0.3 Hz) con retardos típicos (15, 20 y 25); y LFO de frecuencias máximas (1, 2 y 3 Hz) con retardos típicos (25, 30 y 35); los coeficientes de entrada y de las líneas de retardo se mantienen fijos (b_0=0.7, b_{D1} =0.3, b_{D2} = 0.3 y b_{D3} = 0.3). También se mantienen fijas las ampliudes de los LFOs (Sweep_Depth1 = 1.0, Sweep_Depth2 = 1.0 y Sweep_Depth3 =1.0).

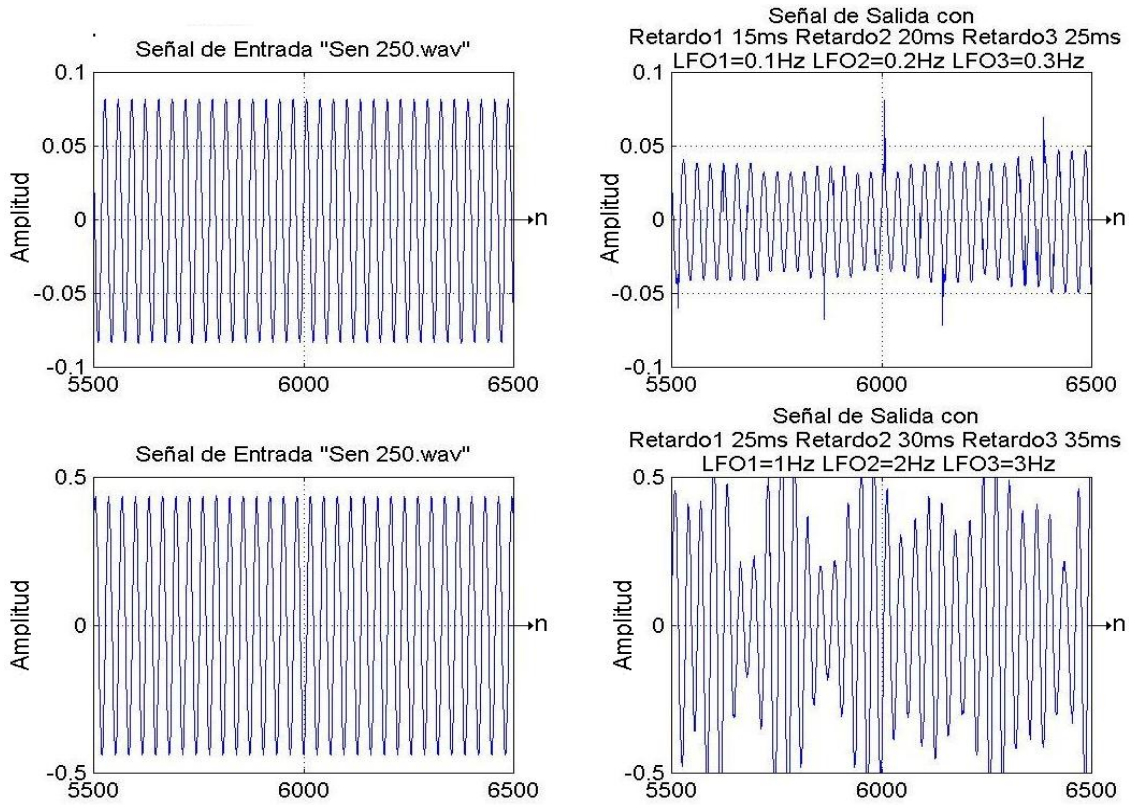
Tabla 13. Valores SNR y Erms para *chorus* tiempo real

Entrada "Sen_250.wav" b_0=0.7 b_{D1}=b_{D2}=b_{D3}=0.3 Sweep_Depth1, 2 y 3=1.0							
LFO 1 (Hz)	***LFO 2 (Hz)***	***LFO 3 (Hz)***	***D1 (ms)***	***D2 (ms)***	***D3 (ms)***	***SNR (dB)***	***Erms (%)***
0.1	0.2	0.3	15	20	25	146.39	5.81e-018
1	2	3	25	30	35	145.71	2.07e-016
0.1	1	2	15	20	25	146.28	2.01e-016
0.1	0.5	3	15	25	35	145.89	2.01e-016

Entrada "Prueba.wav" b_0=0.7 b_{D1}=b_{D2}=b_{D3}=0.3 Sweep_Depth1, 2 y 3=1.0							
LFO 1 (Hz)	***LFO 2 (Hz)***	***LFO 3 (Hz)***	***D1 (ms)***	***D2 (ms)***	***D3 (ms)***	***SNR (dB)***	***Erms (%)***
0.1	0.2	0.3	15	20	25	145.51	4.72e-017
1	2	3	25	30	35	145.85	4.76e-017
0.1	1	2	15	20	25	146.00	4.64e-017
0.1	0.5	3	15	25	35	145.77	4.69e-017

Fuente: YASSER MÉNDEZ. Autor del proyecto

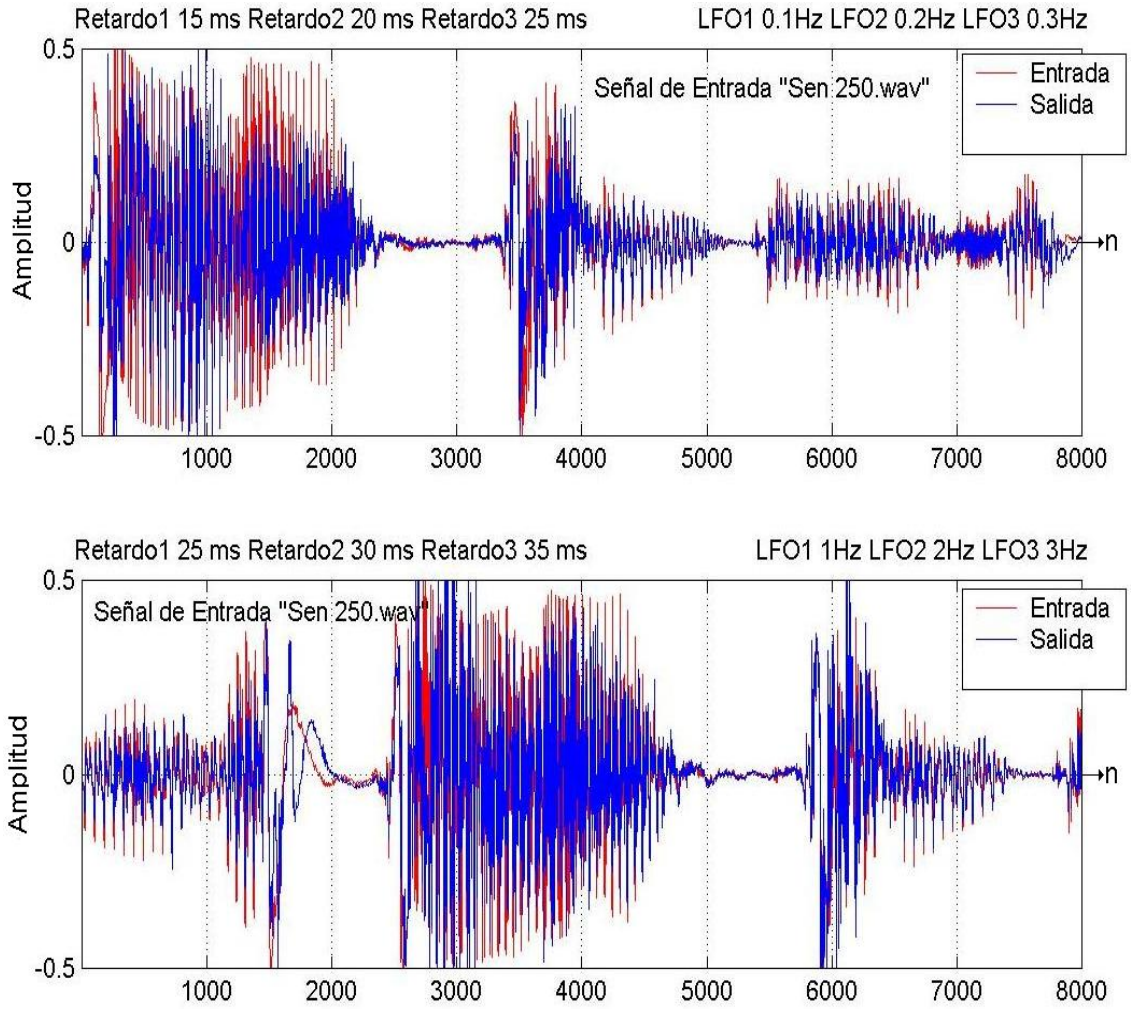
Figura 60. Efecto *chorus* señal sinusoidal



Fuente: YASSER MÉNDEZ. Autor del proyecto

En la figura 61 se muestra el efecto que produce el algoritmo *chorus* cuando la entrada es una señal de audio, usando la misma configuración de parámetros que en el caso de la entrada sinusoidal.

Figura 61. Efecto *chorus* señal de voz



Fuente: YASSER MÉNDEZ. Autor del proyecto

4.3 EVALUACIÓN DE LOS EFECTOS DE AUDIO BASADOS EN AMPLITUD

4.3.1 Evaluación del compresor. Se evalúa el algoritmo compresor con detección pico para simulación y en tiempo real; y el compresor con detección RMS se evalúa para su implementación en tiempo real. En las tablas 14 y 15 se muestran los resultados de SNR y Erms del efecto compresor con detección pico para diferentes umbrales y relaciones de compresión, usando como entrada una señal sinusoidal de 200 Hz y una señal de voz, las mismas usadas en la evaluación del los retardos.

Tabla 14. Valores SNR y Erms para compresor pico simulación

Entrada "Sen_200.dat"			
Umbral (V)	***Relación de Compresión***	***SNR (dB)***	***Erms***
0.15	0.5	143.89	2.44e-016
0.25	0.1	147.64	9.38e-017
0.35	0.05	145.91	2.21e-016

Entrada "Prueba.dat"			
Umbral (V)	***Relación de Compresión***	***SNR (dB)***	***Erms***
0.15	0.5	151.36	1.26e-017
0.25	0.1	147.84	2.19e-017
0.35	0.05	147.39	3.12e-017

Fuente: YASSER MÉNDEZ. Autor del proyecto

Tabla 15. Valores SNR y Erms para compresor pico tiempo real

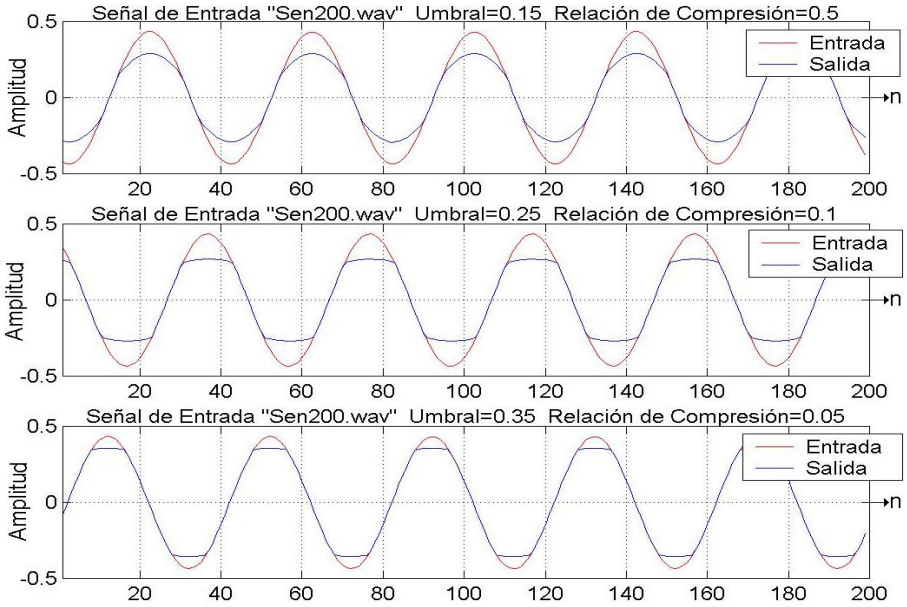
Entrada "Sen_200.wav"			
Umbral (V)	***Relación de Compresión***	***SNR (dB)***	***Erms***
0.15	0.5	148.62	6.70e-017
0.25	0.1	150.21	4.81e-017
0.35	0.05	151.61	5.37e-017

Entrada "Prueba.wav"			
Umbral (V)	***Relación de Compresión***	***SNR (dB)***	***Erms***
0.15	0.5	150.67	1.11e-017
0.25	0.1	151.78	9.19e-018
0.35	0.05	154.15	7.05e-018

Fuente: YASSER MÉNDEZ. Autor del proyecto

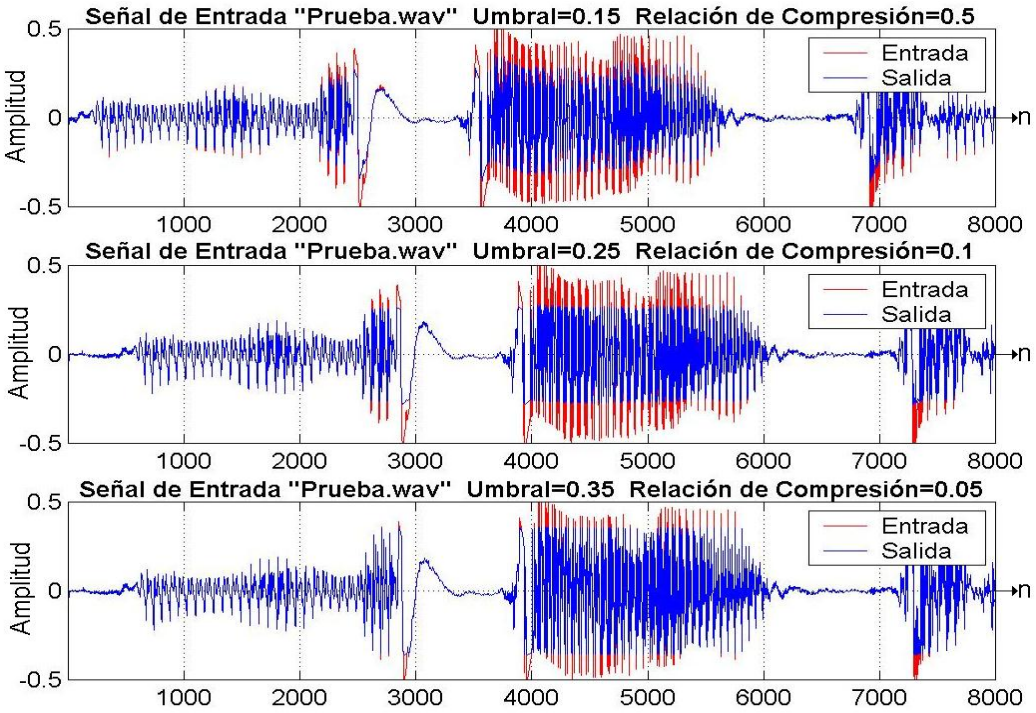
Las figuras 62 y 63 muestran la reducción de la ganancia de la señal sinusoidal "Sen_200.wav" y de la señal de audio "Prueba.wav" respectivamente, para las diferentes relaciones de compresión y umbral definidos en la tabla 15.

Figura 62. Efecto compresor con detección pico señal sinusoidal



Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 63. Efecto compresor con detección pico señal de voz



Fuente: YASSER MÉNDEZ. Autor del proyecto

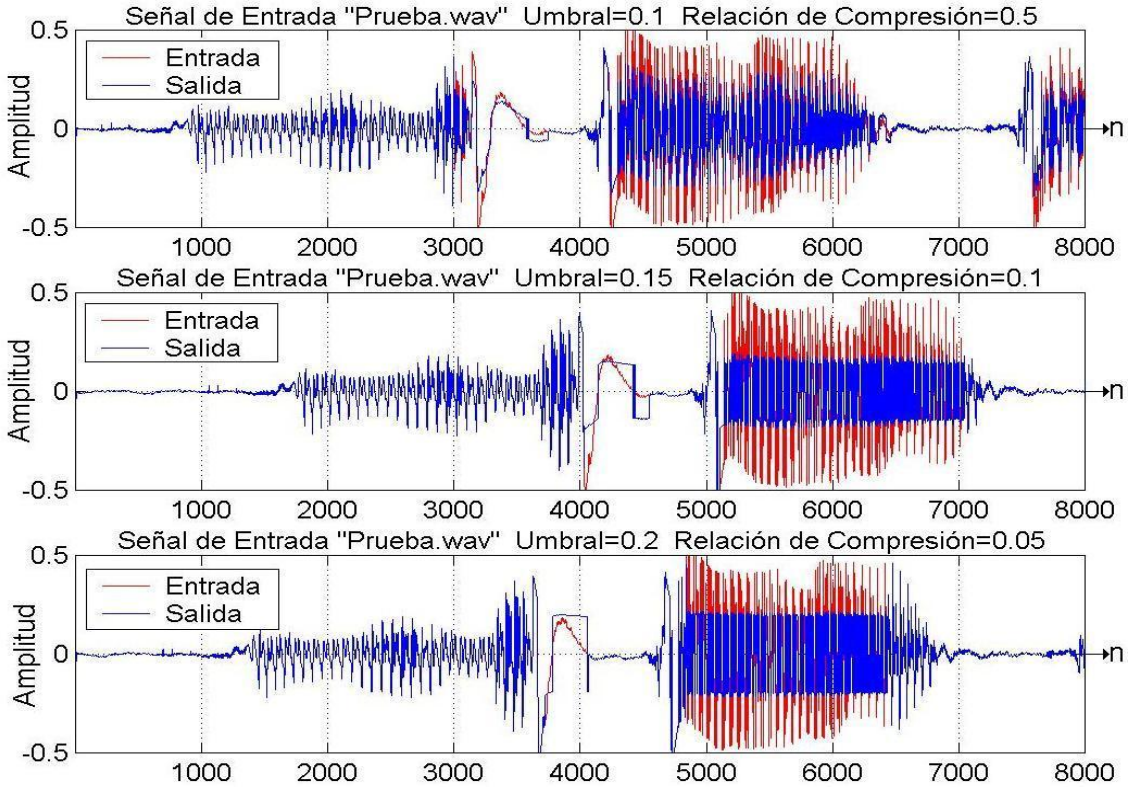
Para la evaluación del compresor con detección RMS se emplea la señal de audio “Prueba.wav” y los resultados de la comparación con Matlab en términos de SNR y Erms se muestran en la tabla 13 para diferentes valores de umbral y relación de compresión. En la evaluación del compresor con detección RMS se deben emplear valores de umbral más pequeños puesto que la comparación se hace a partir de la estimación RMS de la señal de entrada y no con el valor pico. Por esta razón aunque el valor pico de la señal de entrada sobrepasa el valor umbral, si su estimación RMS es menor al umbral, la señal no sufre compresión; esto se traduce en una compresión más suave como se observa en la figura 64, donde se aplica la compresión con detección RMS a la señal de audio “Prueba.wav” para los valores de umbral y relaciones de compresión de la tabla 16.

Tabla 16. Valores SNR y Erms para compresor RMS tiempo real

Entrada “Prueba2.wav”			
Umbral (V)	***Relación de Compresión***	***SNR (dB)***	***Erms***
0.1	0.5	150.61	1.03e-017
0.15	0.1	145.07	3.43e-017
0.2	0.05	146.29	3.83e-017

Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 64. Efecto compresor con detección RMS señal de voz



Fuente: YASSER MÉNDEZ. Autor del proyecto

4.3.2 Evaluación del expansor. El algoritmo expansor con detección pico se evalúa en la simulación y su implementación en tiempo real usando como entrada una señal sinusoidal de 200 Hz y amplitud pico 0.25; y una señal de voz. Los archivos de la simulación son "Sen_200b.dat" y "Prueba2.dat" y para tiempo real se usan los archivos "Sen_200.wav" y "Prueba2.wav". Los resultados de la comparación con Matlab, en términos de SNR y Erms, se muestran en la tabla 17 para la simulación y en la tabla 18 para la implementación en tiempo real.

Tabla 17. Valores SNR y Erms para expansor pico simulación

Entrada "Sen_200b.dat"				
Umbral (V)	***Relación de Compresión***	***Relación de Expansión***	***SNR (dB)***	***Erms***
0.08	0.05	2.0	146.99	1.62e-016
0.1	0.125	1.7	144.60	2.04e-016
0.12	0.25	1.5	150.45	4.29e-017

Entrada "Prueba2.dat"				
Umbral (V)	***Relación de Compresión***	***Relación de Expansión***	***SNR (dB)***	***Erms***
0.05	0.5	1.5	138.98	7.65e-017
0.02	0.1	1.7	141.90	5.88e-017
0.015	0.05	2.0	139.07	1.59e-016

Fuente: YASSER MÉNDEZ. Autor del proyecto

Tabla 18. Valores SNR y Erms para expansor pico tiempo real

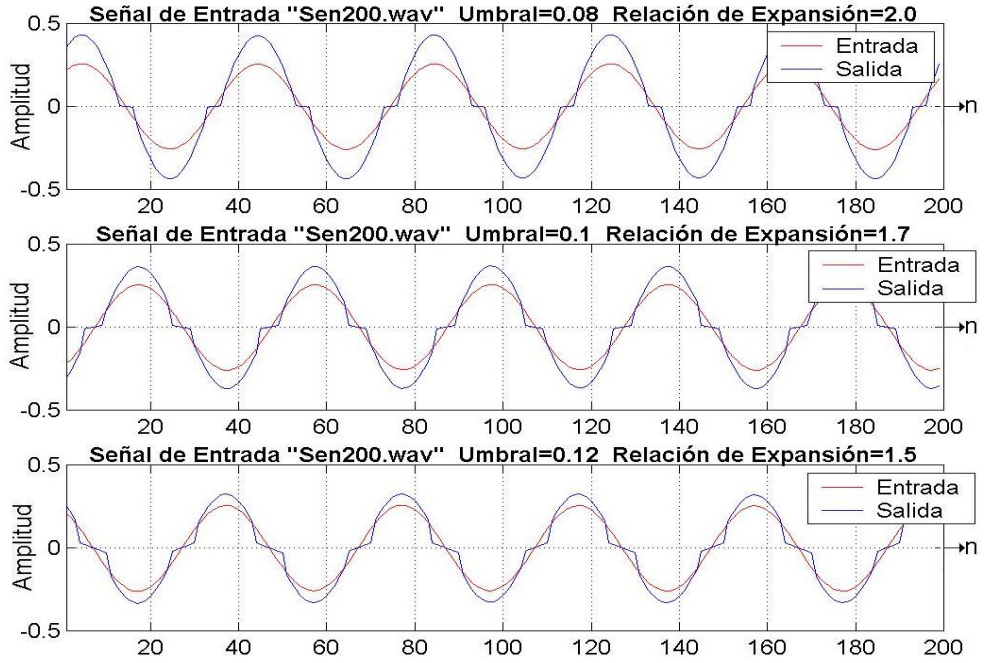
Entrada "Sen_200.wav"				
Umbral (V)	***Relación de Compresión***	***Relación de Expansión***	***SNR (dB)***	***Erms***
0.08	0.05	2.0	146.59	1.87e-016
0.1	0.125	1.7	146.34	1.42e-016
0.12	0.25	1.5	153.80	1.98e-017

Entrada "Prueba2.wav"				
Umbral (V)	***Relación de Compresión***	***Relación de Expansión***	***SNR (dB)***	***Erms***
0.05	0.5	1.5	145.04	6.99e-018
0.02	0.1	1.7	144.86	1.36e-017
0.015	0.05	2.0	150.35	5.47e-018

Fuente: YASSER MÉNDEZ. Autor del proyecto

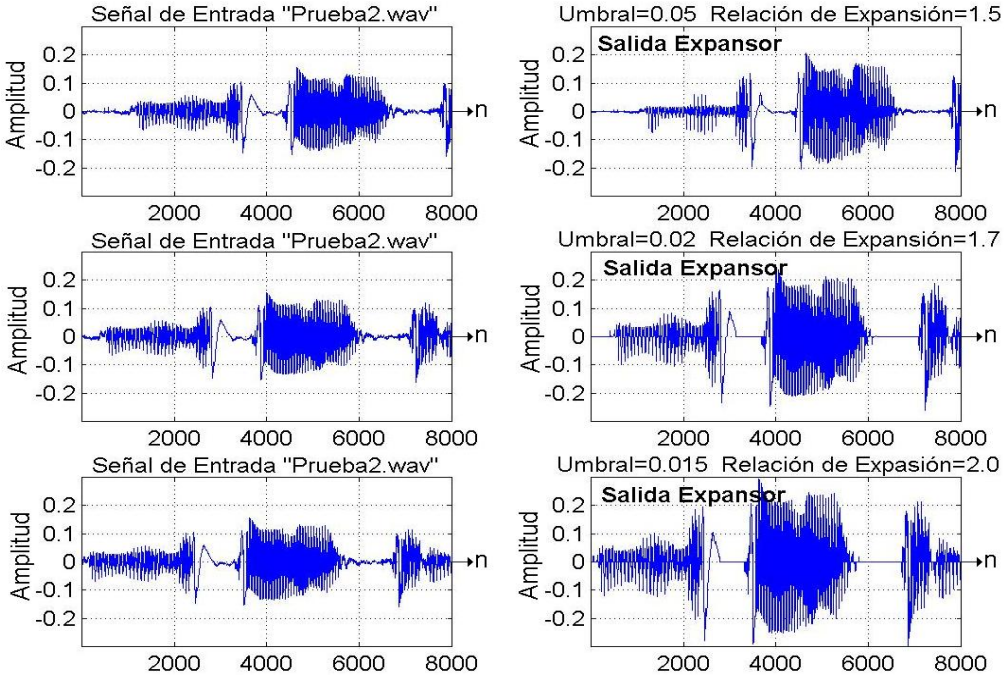
En la figura 65 se muestra el efecto del expansor cuando la señal de entrada es la onda sinusoidal de 200 Hz y en la figura 66 cuando la entrad es una señal de voz, usando los parámetros de umbral, relación de compresión y relación de expansión de la tabla 17.

Figura 65. Efecto Expansión con detección pico señal sinusoidal



Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 66. Efecto Expansión con detección pico señal de voz



Fuente: YASSER MÉNDEZ. Autor del proyecto

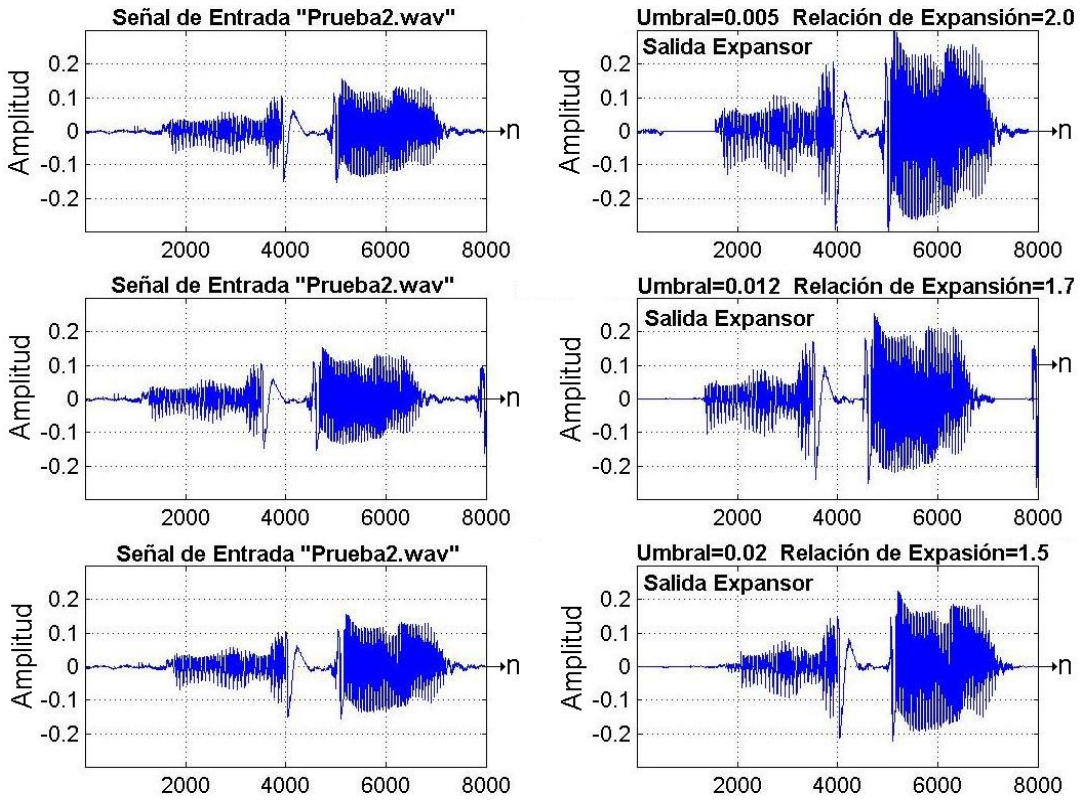
La evaluación del expansor con detección RMS se hace solo en tiempo real y se usa como entrada la señal de voz “Prueba2.wav” para los valores de umbral de la tabla 19 manteniendo constantes la relación de compresión y la relación de expansión. En esta tabla se muestran los resultados en términos de SNR y Erms de la comparación con Matlab; y en la figura 67 se observa el efecto del expansor con detección RMS cuando se usa como entrada una señal de voz.

Tabla 19. Valores SNR y Erms para expansor RMS tiempo real

Entrada “Prueba2.wav”				
Umbral (V)	***Relación de Compresión***	***Relación de Expansión***	***SNR (dB)***	***Erms***
0.005	0. 05	2.0	145.28	1.76e-017
0.012	0.125	1.7	145.14	1.29e-017
0.02	0.25	1.5	144.46	1.02e-017

Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 67. Efecto Expansión con detección RMS señal de voz



Fuente: YASSER MÉNDEZ. Autor del proyecto

En la tabla 20 se muestran los valores de SNR y Erms de la salida del efecto Noise Gate (detección RMS) usando como entrada la señal de voz “Prueba2.wav” y los mismos valores de

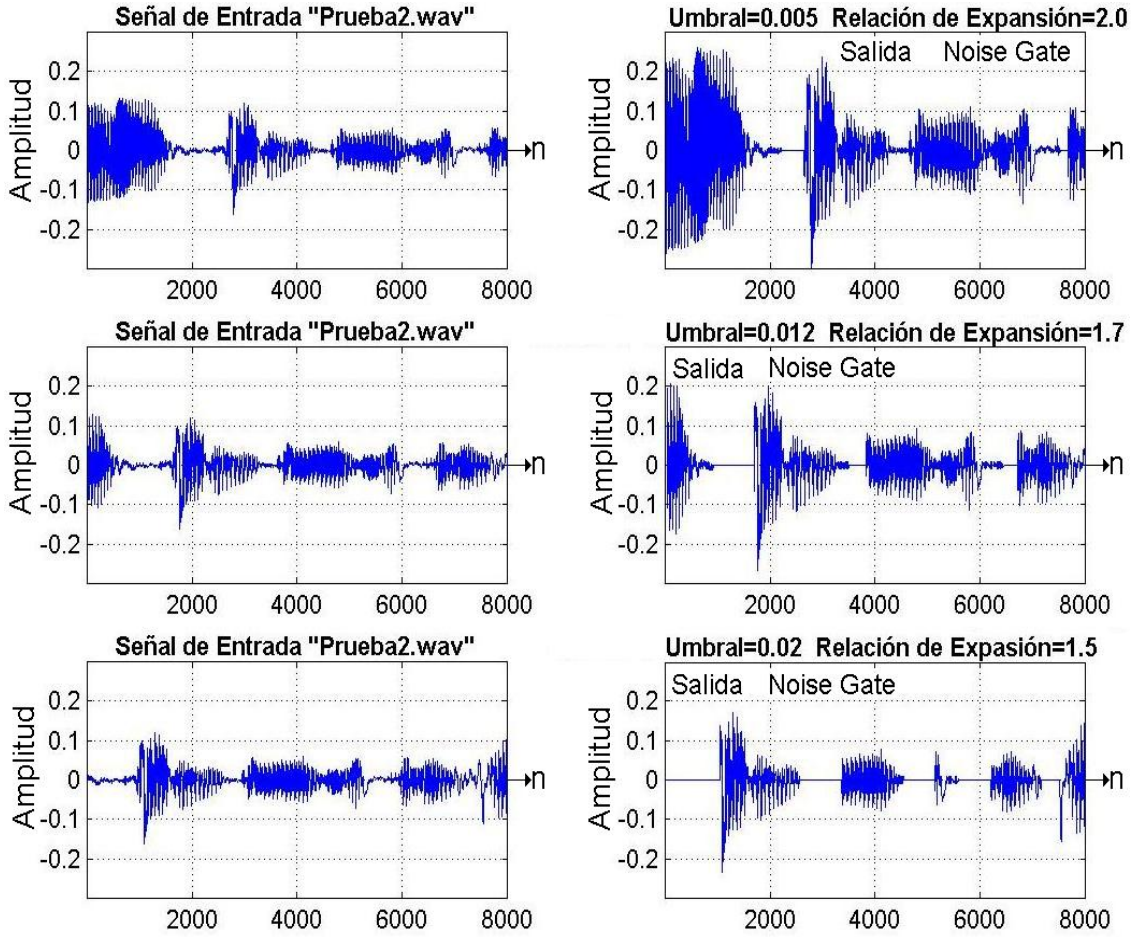
umbral y relación de expansión del expansor RMS. La comparación de la señal de entrada y la señal de salida para el Noise Gate con los parámetros de la tabla 20 se muestra en la figura 68, donde se puede apreciar que la salida es nula para aquellos valores de la estimación RMS menores que el umbral establecido.

Tabla 20. Valores SNR y Erms para Noise Gate RMS tiempo real

Entrada "Prueba2.wav"			
Umbral (V)	***Relación de Expansión***	***SNR (dB)***	***Erms***
0.005	2.0	145.15	1.25e-017
0.012	1.7	144.26	5.30e-018
0.02	1.5	144.76	2.97e-018

Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 68. Efecto Noise Gate RMS señal de voz



Fuente: YASSER MÉNDEZ. Autor del proyecto

4.4 EVALUACIÓN DE LOS FILTROS FIR E IIR IMPLEMENTADOS

Para la evaluación los algoritmos de simulación de los filtros IIR y FIR se utiliza como señales de entrada un impulso unitario generado en Matlab, una suma de cinco sinusoidales armónicamente relacionadas, un archivo de ruido blanco de 8 kHz y un archivo de voz convertido en un archivo ".DAT". Para los algoritmos en tiempo real se usan los archivos *"Sen400_5arm.wav"*, *"RuidoBlanco8kHz.wav"* y "Prueba.wav" a través de la tarjeta de audio del PC. El primero corresponde a la suma de cinco sinusoidales de diferentes frecuencias armónicamente relacionadas y muestreadas a 8 kHz, generadas con el programa Cool Edit Pro 2.0, el segundo a ruido blanco muestreado a 8 kHz generado por el mismo programa y el tercero el la señal de voz utilizada en los anteriores efectos.

4.4.1 Evaluación filtro FIR. El algoritmo del filtro FIR se evalúa primero en su etapa de simulación y para ello se usa en primera instancia como entrada un impulso unitario generado en Matlab. La longitud del impulso unitario es la misma del número de taps del filtro, la primera muestra vale uno y las restantes cero. El archivo ".DAT" que contiene los coeficientes del filtro FIR, puede ser cambiado por CoeFIR_Pasabajos.dat, CoeFIR_Pasaaltos.dat, CoeFIR_Pasabanda.dat o CoeFIR_Bandaeliminada.dat, según sea un filtro FIR pasa bajos, pasa altos, pasa banda o banda eliminada, respectivamente.

Las características en frecuencia de los filtros FIR implementados se definen en la tabla 21, donde todos usan una frecuencia de muestreo de 8000 Hz. Estos son los datos necesarios para ser usados en los archivos de Matlab que generan los coeficientes de cada tipo de filtro usando las funciones REMEZORD y REMEZ. Los coeficientes usados para el algoritmo de simulación y la implementación en tiempo real son los mismos. En los anexos se pueden ver los programas de Matlab que calculan los coeficientes para los diferentes filtros FIR usados en la simulación y la tarjeta de evaluación.

Tabla 21. Características de los filtros FIR implementados

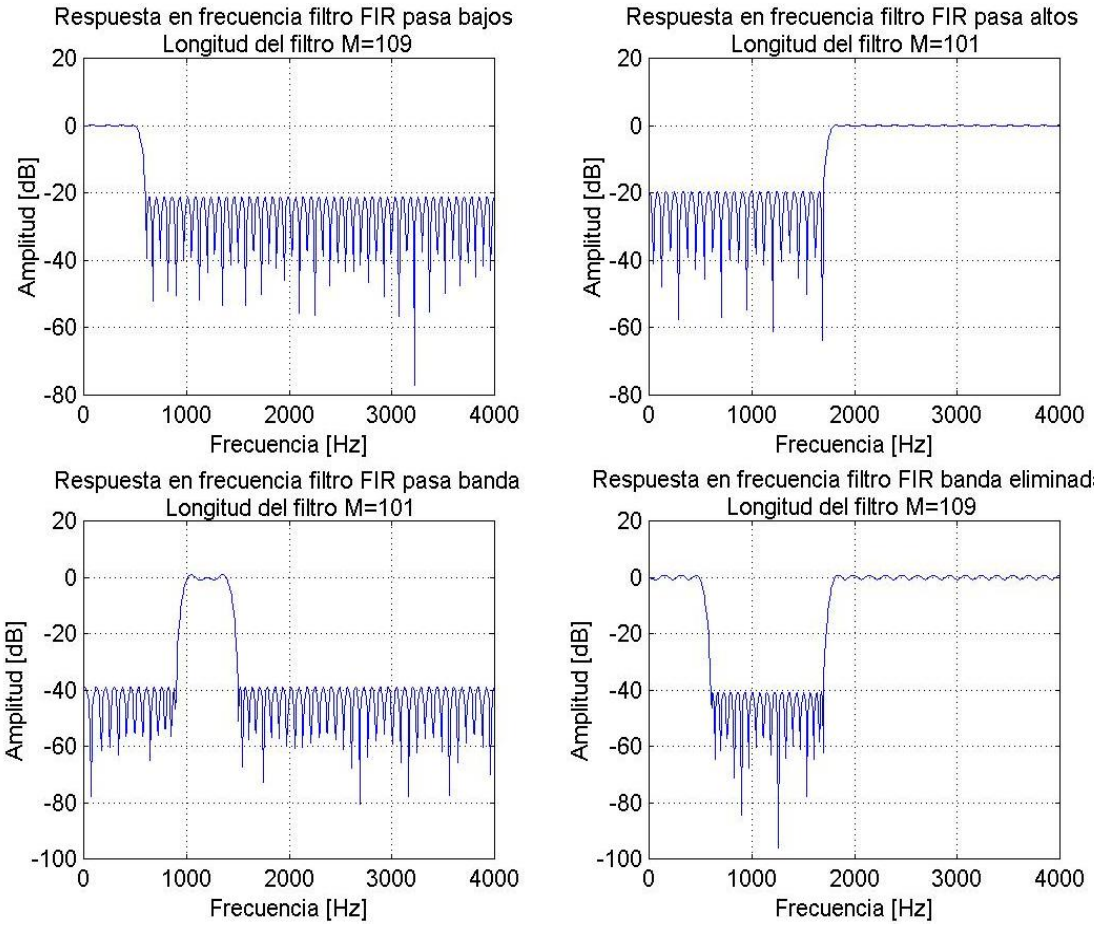
Tipo de Filtro FIR	***Frecuencia de la banda de paso (Hz)***	***Frecuencia de la banda eliminada (Hz)***	***Amplitud de la banda de paso***	***Amplitud de la banda eliminada***	***Rizado de la banda de paso***	***Rizado de la banda eliminada***
Pasa bajos	500	600	1	0	0.01	0.1
Pasa altos	1800	1700	0	1	0.1	0.01
Pasa banda	1000 y1400	900 y 1500	1	0 y 0	0.01	0.1 y 0.1
Banda eliminada	500 y 1800	600 y 1700	1 y 1	0	0.01 y 0.01	0.1

Fuente: YASSER MÉNDEZ. Autor del proyecto

En la figura 69 se muestra la respuesta en frecuencia para cada uno de los filtros FIR. Los coeficientes de los filtros FIR pasa bajos y banda eliminada tienen una longitud de 109, mientras que los coeficientes de los filtros FIR pasa altos y pasa banda tienen una longitud de 101. Cada vez que se desee ejecutar un filtro diferente simplemente se cambia el número de taps en la

variable definida como TAPS y el archivo ".DAT" correspondiente en la variable definida como coef_FIR[TAPS].

Figura 69. Respuesta en frecuencia de los filtros FIR implementados

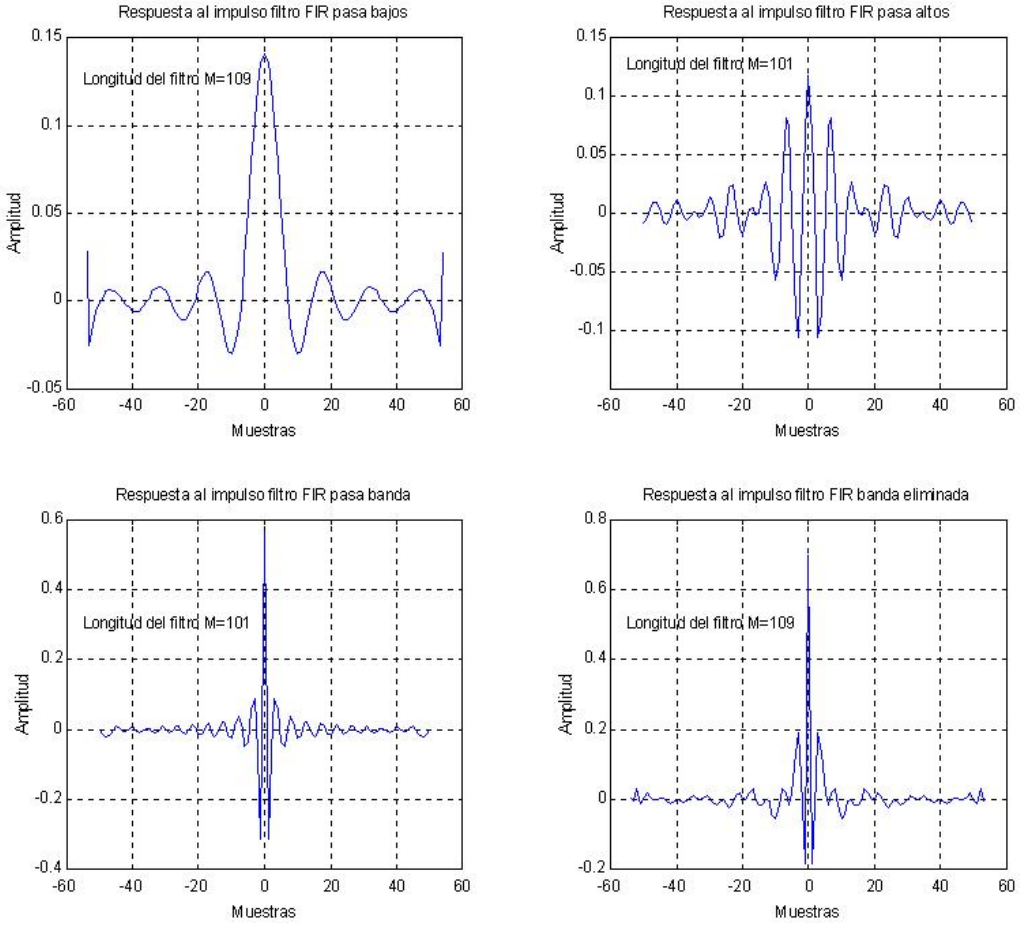


Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Simulación.** Según el número de taps se utiliza como entrada, en primera instancia, para probar el filtro respuestas al impulso de 101 y 109 muestras de longitud, que se guardan en los archivos "Impulso101.dat" e "Impulso109.dat". Por lo tanto la variable definida como N (muestras de entrada) debe tener un valor de 101 y 109 según la entrada que se use. La salida del filtro FIR es almacenada en el buffer de salida y se transfiere a un archivo ".DAT", para ser luego comparada con la salida del algoritmo en Matlab. En la figura 70 se observa que la salida para cada filtro cuando la entrada es un impulso unitario corresponde a la gráfica de los coeficientes del filtro respectivo, es decir la respuesta al impulso.

El segundo tipo de entrada corresponde a una suma de sinusoides armónicamente relacionadas de frecuencia base 400 Hz y cinco primeros armónicos, muestreada a 8000 Hz, generado con el Cool Edit Pro 2 y convertido con matlab en el archivo de 8000 muestras, "Sen400_5arm.dat".

Figura 70. Respuesta al impulso filtros FIR a implementar



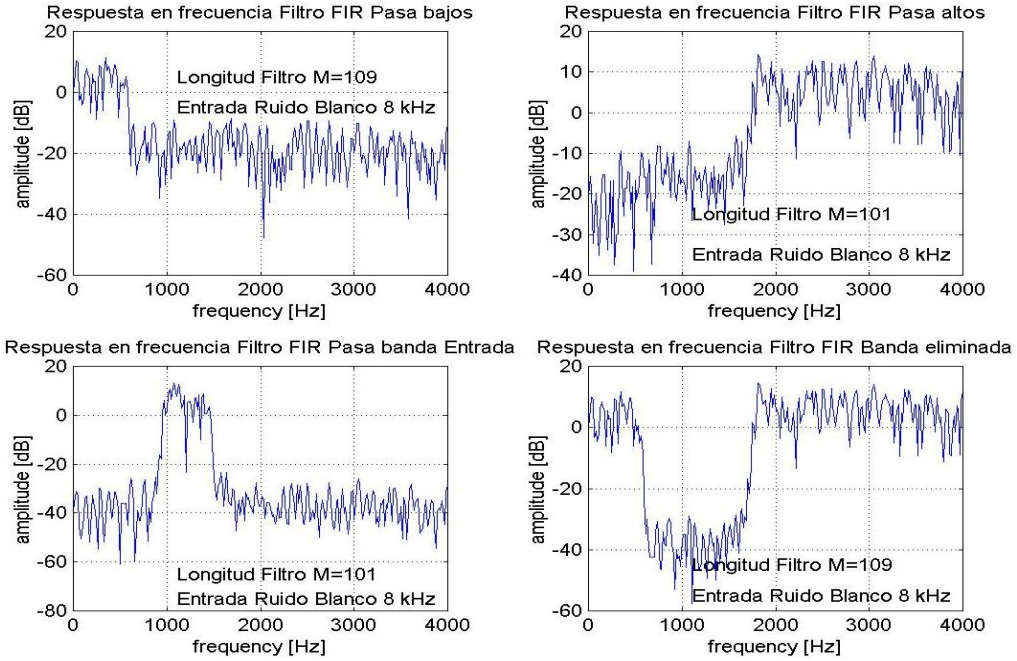
Fuente: YASSER MÉNDEZ. Autor del proyecto

También se utiliza como entrada ruido blanco muestreado a 8000 Hz, generado con el Cool Edit Pro 2 y convertido con matlab en el archivo de 8000 muestras, “RuidoBlanco8kHz.dat”. El archivo de salida de cada filtro se pasa al dominio de la frecuencia mediante la (FFT) de 512 puntos y se utiliza una ventana Bartlett para suavizar la señal. De esta manera, se determina que la salida del filtro en el dominio de la frecuencia, corresponde a su respuesta en frecuencia, como se muestra en la figura 71, aunque el rizado tanto en la banda de paso como en la banda eliminada es bastante fuerte.

Por último se utiliza como entrada la señal de audio “Prueba.wav” con frecuencia de muestreo de 8 kHZ y convertida por matlab en el archivo, “Prueba.dat” que tiene una longitud de 34731 muestras.

En la tabla 22 se muestran los resultados de la Relación Señal a Ruido (SNR) y Error Cuadrático Medio (Erms) para cada filtro FIR implementado usando como entradas el impulso unitario, la suma de sinusoides armónicamente relacionadas, el ruido blanco de 8 kHz y la señal de voz.

Figura 71. Salida filtro FIR para entrada Ruido Blanco de 8 kHz



Fuente: YASSER MÉNDEZ. Autor del proyecto

Tabla 22. Valores SNR y Erms Filtro FIR simulación

Entrada Impulso unitario		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	143.66	5.53e-018
Pasa altos	143.34	2.59e-017
Pasa banda	144.70	3.64e-018
Banda eliminada	153.82	2.62e-018

Entrada "Sen400_5arm.dat"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	139.71	2.16e-016
Pasa altos	140.06	1.96e-016
Pasa banda	140.75	1.59e-016
Banda eliminada	142.27	2.37e-016

Entrada "RuidoBlanco8kHz.dat"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	139.26	5.99e-017
Pasa altos	138.97	2.73e-016
Pasa banda	140.35	3.85e-017
Banda eliminada	143.49	1.163e-016

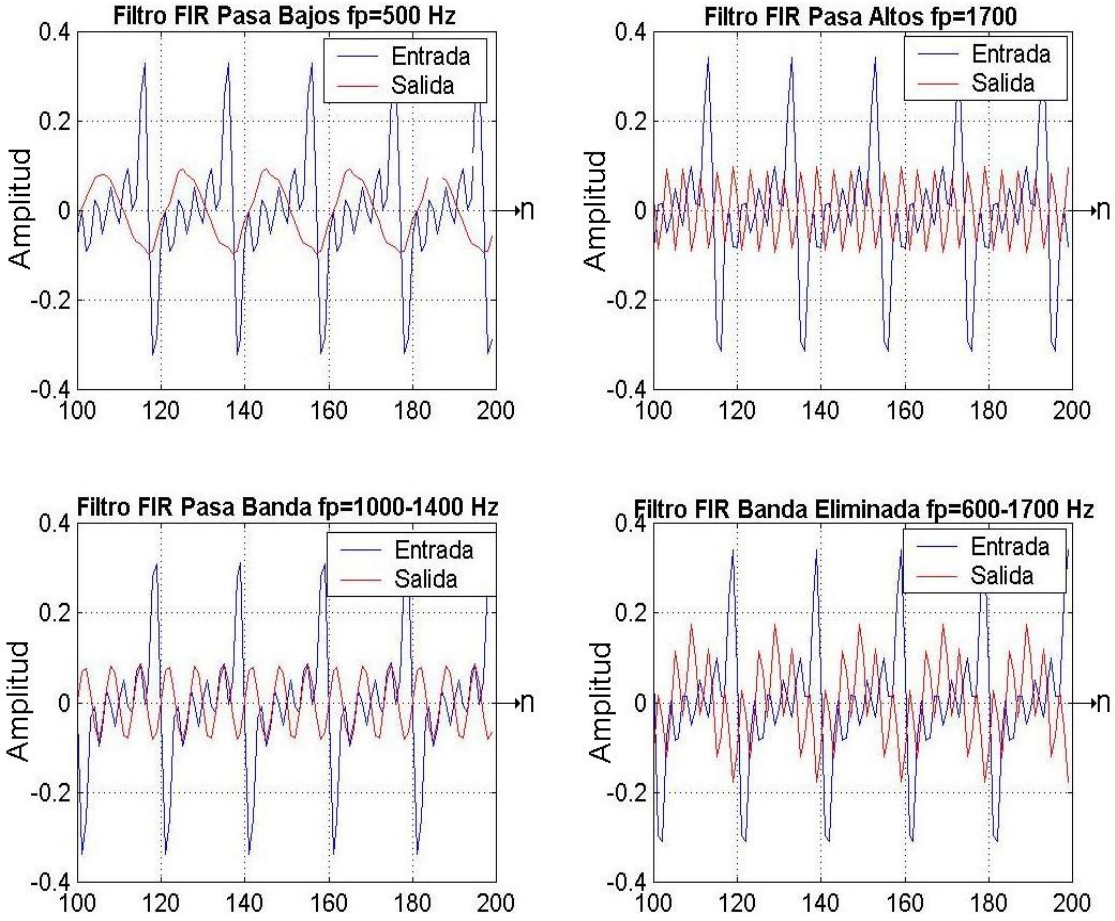
Entrada "Prueba.dat"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	140.04	4.61e-016
Pasa altos	139.00	1.06e-017
Pasa banda	139.99	2.31e-018
Banda eliminada	144.14	1.68e-016

Fuente: YASSER MÉNDEZ. Autor del proyecto

• **Tiempo real.** Para la evaluación del filtro FIR en tiempo real usando el ADSP21065L - EZ-KITE Lite se emplean como entradas los archivos “Sen400_5arm.wav”, “RuidoBlanco8kHz.wav” y “Prueba.wav”, con frecuencia de muestreo de 8000 Hz, mediante la salida de la tarjeta de audio del PC , la cual es conectada a la entrada del Codec en la tarjeta de evaluación del DSP.

En la figura 72 se puede observar un segmento de la señal de entrada y la respectiva salida del filtro FIR pasa bajos, pasa altos, pasa banda y banda eliminada, cuando la entrada es la suma de los cinco primeros armónicos de una señal sinusoidal de frecuencia fundamental (primer armónico) 400 Hz y con una frecuencia de muestreo de 8 kHz.

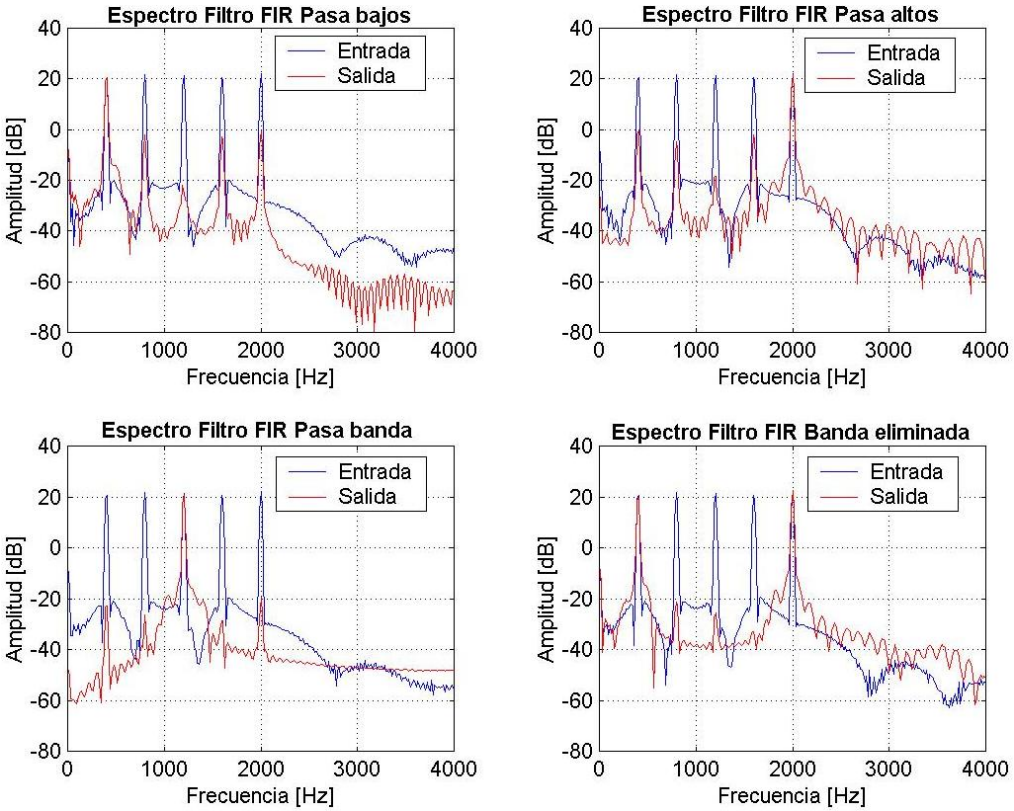
Figura 72. Filtro FIR entrada suma sinusoidales



Fuente: YASSER MÉNDEZ. Autor del proyecto

En la figura 73 se muestra el espectro de la señal de entrada y la señal de salida para cada uno de los tipos del filtro FIR implementado, donde se puede observar cuales son la frecuencias eliminadas por el filtro. La señal de entrada es la suma de sinusoidales armónicamente relacionadas “Sen400_5arm.wav”. Los resultados de la comparación con Matlab se muestran en la tabla 23.

Figura 73. Espectro entrada suma sinusoidales y salida Filtro FIR



Fuente: YASSER MÉNDEZ. Autor del proyecto

Tabla 23. Valores SNR y Erms Filtro FIR tiempo real

Entrada "Sen400_5arm.wav"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	134.03	1.58e-016
Pasa altos	134.08	1.60e-016
Pasa banda	134.16	1.36e-016
Banda eliminada	134.83	2.66e-016

Entrada "RuidoBlanco8kHz.wav"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	133.67	4.23e-017
Pasa altos	134.01	1.27e-016
Pasa banda	134.34	2.84e-017
Banda eliminada	134.23	1.52e-016

Entrada "Prueba.wav"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	134.15	6.62e-016
Pasa altos	131.50	2.58e-017
3Pasa banda	134.14	3.49e-017
Banda eliminada	134.46	5.38e-016

Fuente: YASSER MÉNDEZ. Autor del proyecto

4.4. 2 Evaluación filtro IIR. Para la evaluación de cada uno de los filtros IIR los archivos que contienen los coeficientes del filtro IIR y que se almacenan en las variables "Coeficientes_a[TAPS]" y "Coeficientes_b[TAPS]" son reemplazados por coef_a_Pasabajos.dat y coef_b_Pasabajos.dat; coef_a_Pasaaltos.dat y coef_b_Pasaaltos.dat; coef_a _Pasabanda.dat y coef_b _Pasabanda.dat; coef_b _Bandaeliminada.dat y coef_b _Bandaeliminada.dat, según sea un filtro IIR que se desea evaluar, pasa bajos, pasa altos, pasa banda o banda eliminada, respectivamente.

Las características en frecuencia de los filtros IIR implementados son las mismas definidas para el filtro FIR y se muestran en la tabla 24, con una frecuencia de muestreo de 8000 Hz. En esta tabla aparecen los datos necesarios (frecuencia de la banda de paso, frecuencia de la banda eliminada, rizado de la banda de paso, rizado de la banda eliminada y orden del filtro) que son usados en los archivos de Matlab, para generar los coeficientes de cada tipo de filtro. Para esto se usan las funciones CHEB1ORD Y CHEBY1 para hallar los coeficientes del filtro pasa bajos y pasa banda; y las funciones ELLIPORD y ELLIP para calcular los coeficientes del filtro pasa altos y banda eliminada. Los coeficientes usados para el algoritmo de simulación y la implementación en tiempo real son los mismos. En los anexos se pueden ver los programas de Matlab que calculan los coeficientes para los diferentes filtros IIR usados en la simulación y la tarjeta de evaluación.

Tabla 24. Características de los filtros IIR a implementar

Tipo de Filtro IIR	***Frecuencia de la banda de paso (Hz)***	***Frecuencia de la banda eliminada (Hz)***	***Rizado de la banda de paso (dB)***	***Rizado de la banda eliminada (dB)***	***Orden del Filtro N***
Pasa bajos	500	600	3	50	6
Pasa altos	1800	1700	3	50	7
Pasa banda	1000 y1400	900 y 1500	3	50	4
Banda eliminada	500 y 1800	600 y 1700	3	40	5

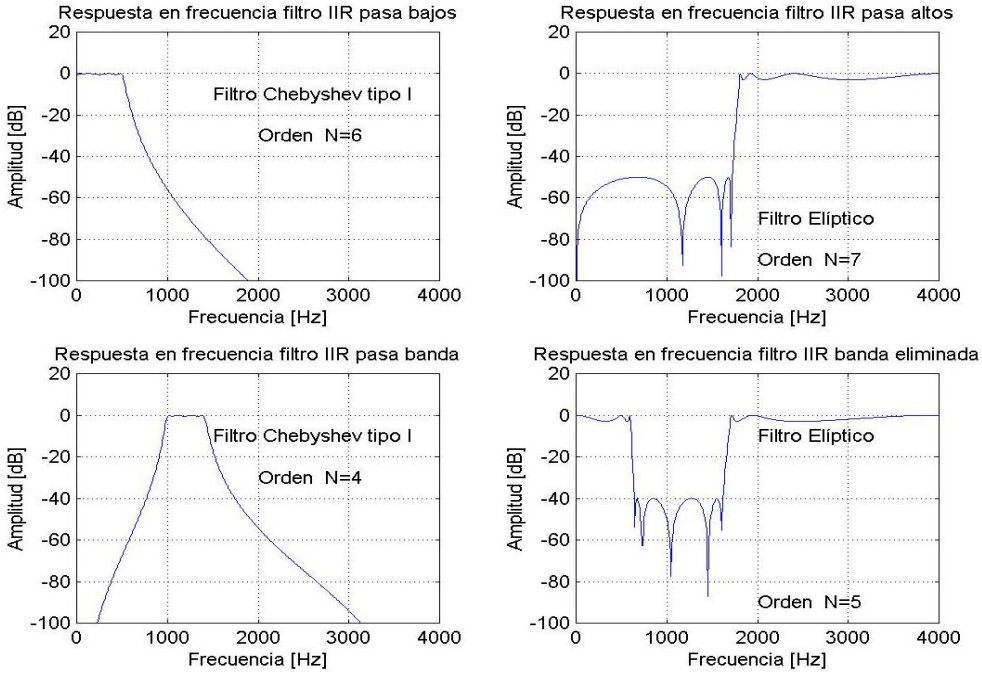
Fuente: YASSER MÉNDEZ. Autor del proyecto

En la figura 74 se muestra la respuesta en frecuencia para cada uno de los filtros IIR implementados. Los coeficientes de los filtros IIR pasa bajos y pasa altos tienen una longitud de N+1; mientras que los coeficientes de los filtros IIR pasa banda y banda eliminada tienen una longitud de 2N+1.

• **Simulación.** El algoritmo simulación del filtro IIR se comprueba usando como entrada el impulso unitario de longitud 300 ("Impulso_IIR.dat"), la suma de cinco sinusoidales armónicamente relacionadas con frecuencia base 400 Hz y muestreada a 8 kHz ("Sen400_5arm.dat"), la señal de voz ("Prueba.dat") y el ruido blanco de 8 kHz ("RuidoBlanco8kHz.dat"). Las salidas del filtro para las entradas suma de sinusoidales y ruido son almacenadas en *buffers* de 8000 muestras de longitud; mientras que para la entrada señale de voz se utiliza *buffers* de 34731 muestras de longitud. Los resultados de la comparación con Matlab, en términos de SNR y Erms se muestran en la tabla 25 para cada uno de los tipos de filtro.

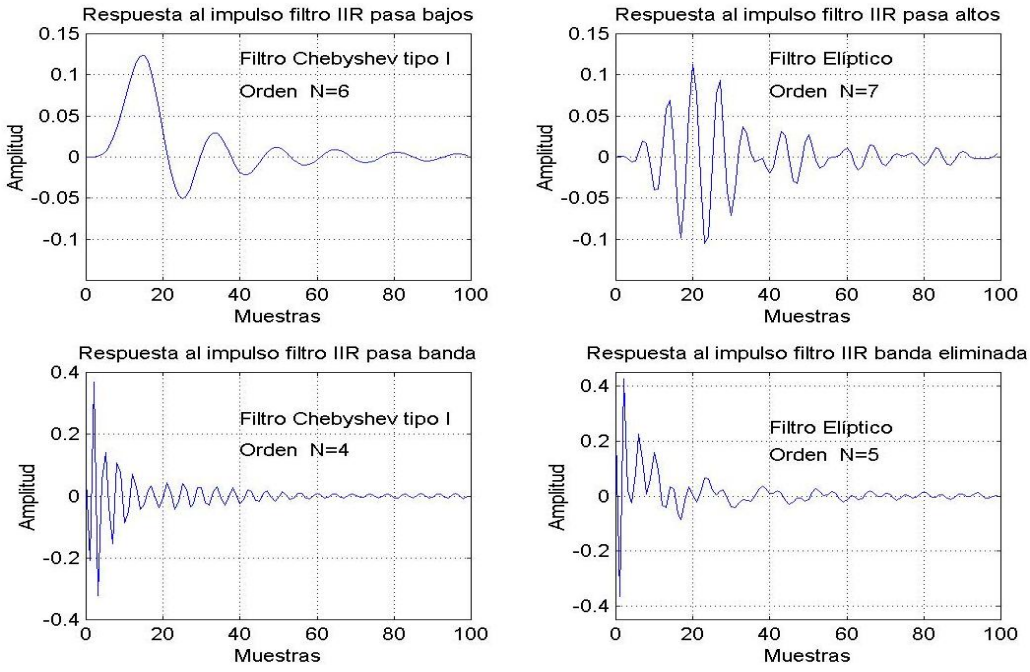
La respuesta al impulso de cada filtro IIR se obtiene cuando la entrada es el impulso unitario de 300 muestras y se puede observar en la figura 75.

Figura 74. Respuesta en frecuencia de los filtros IIR implementados



Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 75. Respuesta al impulso de los filtros IIR implementados



Fuente: YASSER MÉNDEZ. Autor del proyecto

Los resultados de la comparación con Matlab en términos de SNR y Erms para cada uno de los filtro IIR implementados en la simulación usando como entrada el impulso unitario, la suma de sinusoides armónicamente relacionadas, la señal de voz y el ruido blanco de 8 kHz se muestran en la tabla 25.

Tabla 25. Valores SNR y Erms Filtro IIR simulación

Entrada Impulso unitario		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	54.41	1.50e-009
Pasa altos	113.55	5.75e-015
Pasa banda	68.60	4.89e-011
Banda eliminada	74.06	6.88e-011

Entrada "Sen400_5arm.dat"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	54.30	6.97e-008
Pasa altos	112.01	7.80e-014
Pasa banda	70.36	1.65e-009
Banda eliminada	79.79	3.14e-010

Entrada "RuidoBlanco8kHz.dat"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	54.46	1.62e-008
Pasa altos	114.39	5.43e-014
Pasa banda	68.75	5.55e-010
Banda eliminada	73.34	9.17e-010

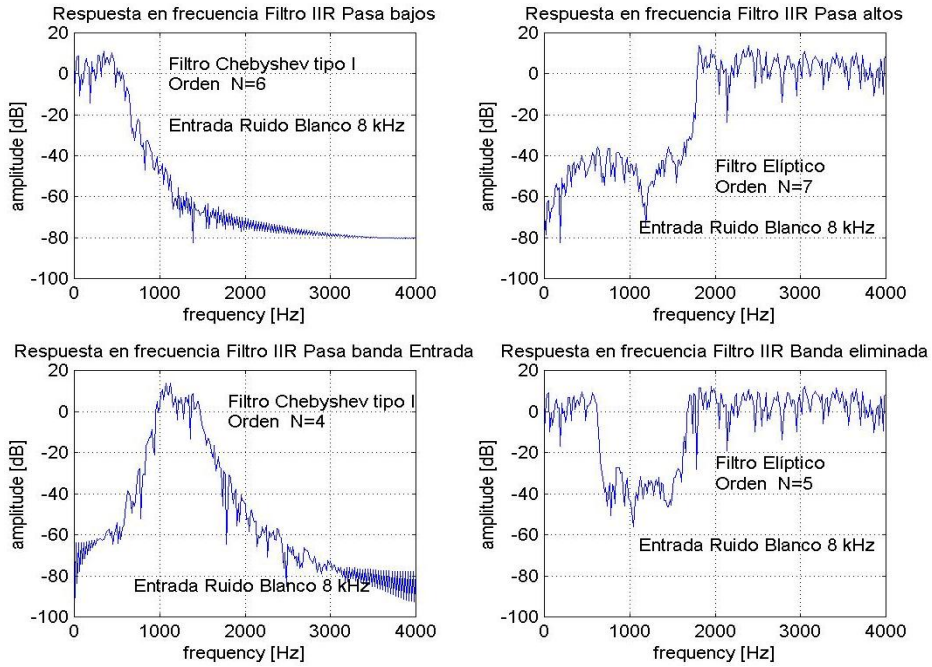
Entrada "Prueba.dat"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	54.68	9.66e-008
Pasa altos	112.26	3.20e-015
Pasa banda	69.33	1.31e-010
Banda eliminada	71.75	1.82e-009

Fuente: YASSER MÉNDEZ. Autor del proyecto

La salida de cada tipo filtro IIR cuando la entrada es el ruido blanco de 8 kHz se puede pasar al dominio de la frecuencia mediante una FFT de 512 puntos y una ventana Bartlett; obteniendo una respuesta en frecuencia muy similar a la respuesta en frecuencia del filtro diseñado, aunque con un nivel de ruido bastante notorio, como lo muestra la figura 76.

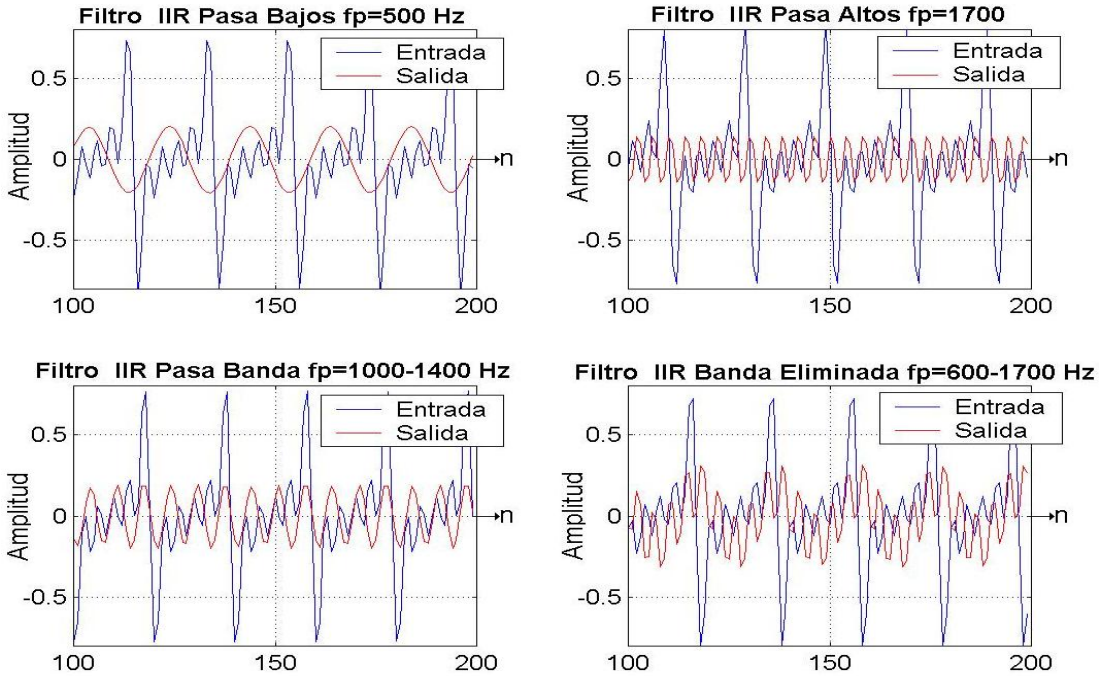
• **Tiempo real.** La evaluación de los filtros IIR implementados en tiempo real se realiza con las mismas entradas empleadas en la comprobación del filtro FIR: "Sen400_5arm.wav", "RuidoBlanco8kHz.wav" y "Prueba.wav"; todas con una frecuencia de muestreo de 8000 Hz. En la figura 77 se muestra el filtrado de la suma de sinusoidales y en la figura 778 el espectro de las mismas.

Figura 76. Salida filtro IIR para entrada Ruido Blanco de 8 kHz



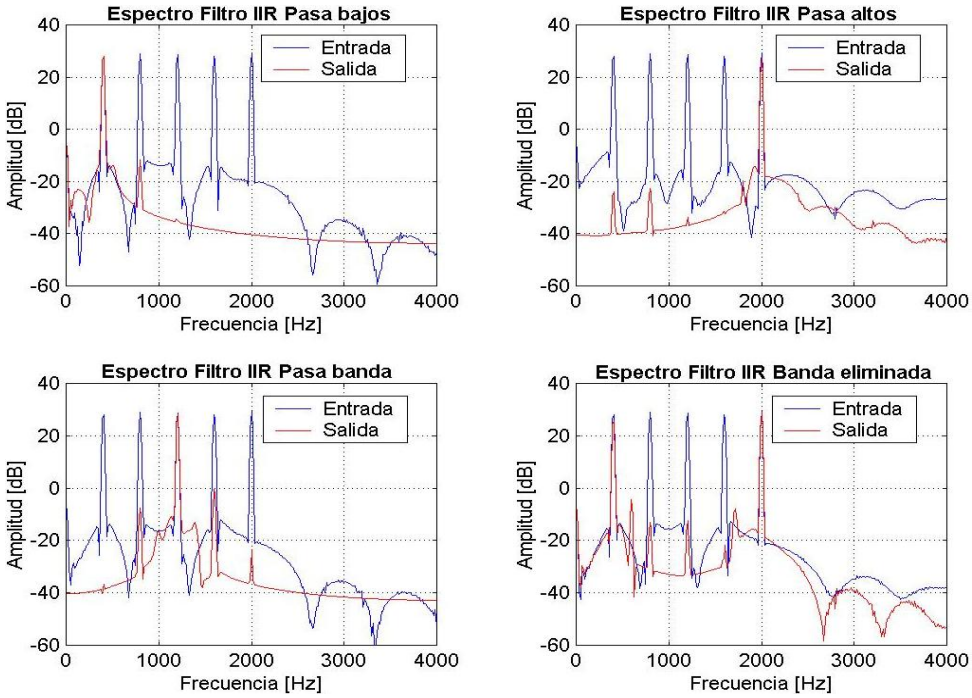
Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 77. Filtro IIR entrada suma sinusoidales



Fuente: YASSER MÉNDEZ. Autor del proyecto

Figura 78. Espectro entrada suma sinusoidales y salida Filtro IIR



Fuente: YASSER MÉNDEZ. Autor del proyecto

Los resultados de la comparación con Matlab del filtro IIR en tiempo real se muestran en la tabla 26 para las mismas entradas con que se evalúa el filtro FIR.

Tabla 26. SNR y Erms Filtro IIR tiempo real

Entrada "Sen400_5arm.wav"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	53.81	8.59e-008
Pasa altos	104.64	4.92e-013
Pasa banda	68.11	2.91e-009
Banda eliminada	67.68	5.78e-009

Entrada "RuidoBlanco8kHz.wav"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	54.07	3.23e-009
Pasa altos	104.08	7.98e-014
Pasa banda	66.57	1.70e-010
Banda eliminada	64.36	1.07e-009

Entrada "Prueba.wav"		
Tipo de Filtro FIR	***SNR (dB)***	***Erms (%)***
Pasa bajos	54.00	5.99e-008
Pasa altos	101.67	7.64e-015
Pasa banda	67.64	2.03e-011
Banda eliminada	68.35	2.07e-009

Fuente: YASSER MÉNDEZ. Autor del proyecto

4.5 ANÁLISIS DE RESULTADOS

Los valores de relación señal a ruido (SNR) y error cuadrático medio (Erms) de la comparación con el algoritmo de simulación de retardo simple en Matlab, son muy similares tanto en la simulación con el VisualDSP++ como en su implementación en tiempo real con la tarjeta de desarrollo. Estos valores están entre de 151 dB y 170 dB para la relación señal a ruido y entre $9x10^{-19}$% y $1x10^{-17}$% para el error cuadrático medio.

Los valores de relación señal a ruido (SNR) y error cuadrático medio (Erms) de la comparación con el algoritmo de simulación de retardo múltiple en Matlab, son muy similares tanto en la simulación con el VisualDSP++ como en su implementación en tiempo real con la tarjeta de desarrollo. Sin embargo el SNR es menor y el Erms es mayor que el del retardo simple. Estos valores están entre de 144 dB y 146 dB para la relación señal a ruido y entre $6x10^{-19}$% y $1x10^{-16}$% para el error cuadrático medio.

La envolvente de la forma de onda de una señal de voz, tiende a conservarse al ser procesada por los algoritmos de retardo simple y retardo múltiple, para valores pequeños de retardo, menores a 50 ms; mientras que valores de retardo grandes, superiores a 100 ms, hacen que la envolvente de la forma de onda de la señal de voz cambie considerablemente.

Los resultados de la comparación con Matlab para la simulación del algoritmo *flanger* están alrededor de 164 dB para SNR y $9x10^{-18}$% para Erms usando como entrada una señal sinusoidal pura de 250 Hz.

Los valores de SNR y Erms obtenidos de la comparación del algoritmo *flanger* en tiempo real y Matlab son muy diferentes a los obtenidos en la simulación, están entre 30 dB y 85 dB para SNR y entre $2x10^{-9}$% y $4x10^{--5}$% para Erms. Esto se debe a la presencia de ruido aleatorio que aparece al utilizar buffers para almacenar gran cantidad de muestras de entrada y salida para realizar la comparación. Aunque los valores de SNR son bajos comparados con los obtenidos en la simulación, los valores de Erms siguen siendo muy pequeños, lo que quiere decir que la forma de onda de la señal de salida es muy similar a la deseada.

Al corregir la señal de salida del algoritmo *flanger* el valor de SNR aumenta considerablemente y se encuentra entre 151 dB y 158 dB; mientras que el valor de Erms disminuye notablemente y se encuentra entre $4x10^{-18}$% y $8x10^{-18}$%, para los diferentes valores de retardo y frecuencia del LFO.

El efecto *chorus* produce una mayor modulación de la señal de entrada debido a que utiliza líneas de retardo variable en paralelo al algoritmo del *flanger*. Para frecuencias del LFO mayores a 0.5 Hz el efecto audible producido por el *chorus* es muy similar al *flanger*. Utilizando LFOs con frecuencias entre 0.1 Hz y 0.5 Hz el efecto audible del *chorus* se asemeja más a varias voces o instrumentos sonando al mismo tiempo.

Los valores de SNR y Erms obtenidos para el algoritmo de simulación *chorus* comparado con el algoritmo de Matlab, se encuentran alrededor 144 dB para SNR y $1x10^{-15}$% para Erms cuando la señal de entrada es la onda seno de 250 Hz; y alrededor 145 dB para SNR y $1x10^{-16}$% para Erms

cuando la señal de entrada es la señal de voz. Esto quiere decir que el algoritmo funciona correctamente independiente de la frecuencia de la señal de entrada.

La señal de salida del algoritmo *chorus* en tiempo real debe ser corregida, pues presenta un ruido aleatorio ocasionando que el valor de SNR este entre 25 dB y 34 dB, y el Erms este entre $7x10^{-6}$% y $3x10^{-5}$%, difiriendo de los valores obtenido en la simulación. Sin embargo los valores pequeños de Erms garantizan que la forma de onda de la señal de salida sea bastante similar a la deseada.

La corrección de la señal de salida del algoritmo *chorus* en tiempo real mejora el valor de SNR entre 145 dB y 146 dB, y reduce el Erms entre $5x10^{-18}$% y $2x10^{-16}$%, siendo muy similares a los obtenidos con la simulación.

Los valores de SNR y Erms obtenidos en la comparación del algoritmo de simulación del compresor y el algoritmo compresor en tiempo real con la simulación en Matlab, son muy similares y se encuentran entre 143 dB y 154 dB para SNR y con valores de Erms menores a $3x10^{-16}$%, lo que muestra el alto grado de funcionamiento de este algoritmo.

El comportamiento de los algoritmos de expansor y *noise gate* presentan también valores muy altos de SNR alrededor de 145 dB y valores muy pequeños de Erms menores a $3x10^{-16}$%, tanto en la simulación como en tiempo real.

La comparación del algoritmo de simulación del filtro FIR implementado usando diversas entradas, con Matlab, arroja valores de SNR entre 138 dB y 153 dB y valores de Erms menores a $5x10^{-16}$%, lo que corrobora el buen funcionamiento de este algoritmo, independiente de los coeficientes del filtro, es decir del tipo de filtro que se ejecute. Estos valores para la situación de tiempo real están alrededor de 134 dB para SNR y $7x10^{-16}$% para Erms.

Los valores de SNR y Erms obtenidos de la comparación del algoritmo del filtro IIR en la forma directa I en simulación y tiempo real con el algoritmo en Matlab, para los diferentes tipos de filtro implementados, son similares pero menores para SNR y mayores para Erms, que los obtenido con el algoritmo de filtro FIR. El algoritmo presenta el mayor SNR (110 dB) y el menor Erms ($7x10^{-14}$%) para un filtro pasa altos *Chebyshev* tipo 1 de orden 7; mientras que para un filtro pasa bajos *Chebyshev* tipo 1 de orden 6 se obtiene el menor SNR (54 dB) y el mayor Erms ($10x10^{-8}$%). Los valores de SNR para los filtros pasa banda y banda eliminada, elípticos de orden 4 y 5 respectivamente, están alrededor de 70 dB y con Erms menores a $1x10^{-9}$%.

5. CONCLUSIONES

Los efectos de audio hacen parte de una amplia gama de aplicaciones del procesamiento digital de señales en el campo del audio. En la actualidad existen numerosos programas sofisticados de edición de audio como el *Cool Edit Pro* (y otros como *Audacity, Sound Forge* o *WavePad*) que permite procesar señales de audio aplicando diversos efectos pero no en tiempo real, sino procesando la señal por vectores. Si la idea es procesar señales de audio en tiempo real, procesamiento muestra a muestra, la aparición de los DSPs ha logrado convertir a los efectos de audio en un sector con gran proyección comercial. Son muchas las casas fabricantes de procesadores de audio (*Yamaha, Behringer, Sabine*) que tienen a los DPSs como la pieza fundamental y el corazón de estos dispositivos, dotando a los músicos y productores de audio profesional de una herramienta excepcional para aplicar numerosos efectos de audio en tiempo real a sus creaciones musicales. Este auge comercial de los procesadores de audio le da al usuario un menú muy variado de posibilidades a la hora de seleccionar uno u otro dispositivo, según sus requerimientos particulares; sin embargo al momento de indagar sobre la implementación en tiempo real de los algoritmos correspondientes a los efectos de audio más utilizados, la información a la cual se tiene acceso es limitada y se reduce a unos pocos autores [28], [29], [34], siendo esto apenas natural si se tiene en cuenta que la rama de procesamiento de audio es bastante explotada comercialmente, y cada fabricante busca impactar con sofisticados equipos cada vez más potentes, con más efectos y control de parámetros de los mismos. Es por esta razón que el presente proyecto tiene gran relevancia al trabajar con la implementación de algoritmos de efectos de audio en tiempo real usando un DSP de 32 bits de coma flotante de la familia SHARC de Analog Devices.

La evaluación de los algoritmos de efectos de audio implementados en tiempo real en el ADSP-21065L, producen resultados satisfactorios, como se observa en los valores obtenidos para los parámetros de evaluación, Error Cuadrático Medio (Erms) son muy bajos, menores a 10^{-5}%; y los valores de Relación Señal a Ruido (SNR) son altos, mayores a 90 dB para los efectos de retardo; mayores a 70 dB para los efectos de modulación; y mayores a 80 dB para los efectos de amplitud. Además, la comparación cualitativa de estos algoritmos implementados en el DSP con los algoritmos desarrollados en Matlab, permiten verificar el correcto funcionamiento de cada uno de los mismos, por cuanto las formas de ondas de las señales obtenidas en Matlab y las señales capturadas en la tarjeta de evaluación DSP son altamente similares. Por otra parte, la posibilidad de trabajar con un DSP de 32 bits en coma flotante aprovechando la rapidez de cálculos, la velocidad de acceso a memoria y transmisión de datos, y el manejo de *buffers* circulares que este tipo de procesadores utilizan, facilita la implementación de una gran cantidad de algoritmos de efectos de audio y combinación de estos, tal y como se puede encontrar en el mercado de los procesadores de audio con una amplia gama de efectos con control sobre los diferentes parámetros característicos. De igual manera la implementación de filtros FIR e IIR es muy sencilla ya que se aprovecha al máximo la estructura hardware del DSP, la cual permite guardar en memoria de programa los coeficientes del filtro previamente calculados, y en memoria de datos las muestras retrasadas de la señal de entrada. Una vez almacenados los datos en memoria, se ejecutan las operaciones de lectura de la muestra de entrada retrasada, lectura del coeficiente, producto de los datos anteriores y suma de los productos, en un solo ciclo.

Teniendo en cuenta que la comparación con Matlab de cada uno de los algoritmos implementados en el DSP tanto cualitativa como cuantitativa, son coherentes y revelan el buen desempeño de los mismos, se hace sumamente interesante seguir desarrollando aplicaciones en este campo del

procesamiento de audio; perfeccionando los algoritmos desarrollados mediante la incorporación de aquellos parámetros que no se tuvieron en cuenta en el presente proyecto; la posibilidad de agregar interfases de usuario que permitan el control de parámetros; la implementación de otros efectos como reverberación, mezclado, ecualizadores, *Vibrato, Pitch Shifter Detune* entre otros; así como la programación de estos algoritmos en lenguaje C soportado por el ADPS-21065L.

Además, en el presente proyecto se puede observar que la programación de los algoritmos utilizando datos de punto flotante, se facilita gracias al archivo de 16 registros temporales para realizar las diferentes operaciones requeridas por los algoritmos. Desde el punto de vista se puede concluir que:

- ✓ La rutina que calcula el retardo simple es fácilmente implementada en tiempo real y requiere solo ocho instrucciones en assembly para su ejecución. La cantidad de retardo puede ser modificada aumentando el tamaño del buffer circular que contiene las muestras previas de la señal de entrada o variando la frecuencia de muestreo. A una mayor frecuencia de muestreo se requiere de un buffer circular de mayor longitud para la línea de retardo.

- ✓ La rutina de retardo múltiple usando líneas de retardo diferentes para cada retardo, adiciona tres instrucciones a la rutina de retardo simple, por cada retardo. Así, la rutina de retardo múltiple con seis taps requiere un total de 33 instrucciones para su implementación. El ADSP 21065L permite implementar hasta un máximo de seis líneas de retardo diferentes utilizando la totalidad de los registros del DAG1 para inicializar los buffers que definen las líneas de retardo.

La evaluación de los diferentes algoritmos implementados se hace usando una frecuencia de muestreo de 8 kHz, sin embargo al utilizar frecuencias de muestreo mayores hasta la máxima soportada por el codec, 48 kHz, el efecto audible es el mismo que al usar una frecuencia de muestreo de 8 kHz.

En cuanto a los resultados obtenidos para SNR y Erms en los algoritmos de los efectos de retardo, se puede afirmar que:

- ✓ Usando como señal de entrada una onda sinusoidal pura de 200 Hz y una señal de voz que tiene un contenido de frecuencias más amplio, los valores de SNR y Erms son similares. Es decir, los algoritmos de retardo simple y retardo múltiple funcionan correctamente para cualquier tipo de señal de audio, independiente de la frecuencia.

Analizando las señales de salida de los algoritmos de los efectos de audio basados en modulación(*Flanger* y *Chorus*) se observa que:

- ✓ El efecto *flanger* produce una modulación de la señal de entrada, que aumenta a medida que se aumenta el retardo y la frecuencia del oscilador de baja frecuencia (LFO). Si la entrada es una señal sinusoidal pura, un solo tono, el efecto se traduce en una variación del tono del tono, el cual varía más rápido a medida que se aumenta el retardo y la frecuencia del LFO.

- ✓ Aunque la teoría sobre el efecto *chorus* sugiere el uso de LFO randómicos, esta situación se simuló en Matlab sin obtener resultados audibles que correspondieran a lo que el efecto *chorus* debe realizar. Por esta razón no se llevó a cabo su implementación en el DSP.

- ✓ En general los algoritmos de efecto *chorus* desarrollados para otros DSP como Motorola o Texas, al igual que lo programas de edición de audio como Cool Edit Pro, usan solo un LFO para determinar el retardo variable en las diferentes líneas de retardo, pero cada una con un desfase predeterminado. Este no es un parámetro del efecto *chorus* por lo cual el usuario o programador no puede cambiar.

Por otra parte, el trabajo con los algoritmos de efectos de audio basados en amplitud (*Compresor, Expansor* y *Noise* Gate) indican que:

- ✓ Las señales de salida del algoritmo compresor permiten visualizar fácilmente la compresión de la señal de entrada a partir de un umbral determinado. El efecto audible es muy notorio en el sentido de percibir una reducción del volumen de la señal como una disminución de la ganancia de la señal, para diferentes tramos de la misma.

- ✓ Las señales de salidas obtenidas del algoritmo expansor evidencian claramente el aumento de la ganancia cuando la señal de entrada supera un umbral determinado, y la reducción de la misma cuando la señal de entrada está por debajo de este umbral.

- ✓ El algoritmo expansor presenta el inconveniente de que si la señal de entrada tiene cierto nivel ruido en aquellos espacios de silencio, este ruido aumenta su ganancia si está por encima del umbral seleccionado. Por lo tanto es recomendable usar el algoritmo de *noise gate* para este tipo de señales o utilizar valores de umbral pequeños.

Finalmente, en el análisis de los resultados obtenidos para los algoritmos de los filtros FIR e IIR implementados, se concluye que:

- ✓ En la evaluación de los diferentes tipos de filtros FIR implementados, pasa bajos, pasa altos, pasa banda y banda eliminada, se pudo constatar que la señal de salida corresponde a la frecuencia seleccionada por cada tipo de filtro, cuando se usa como entrada una suma de señales sinusoidales de frecuencias armónicamente relacionadas. El efecto audible claramente es un tono de diferente frecuencia y la visualización del espectro de las señales de salida verifican esta situación.

- ✓ El algoritmo del filtro IIR implementado en la forma directa I no funciona bajo ciertas condiciones: para filtros pasa bajos *Butterworth* de orden mayor a 10, elípticos de orden mayor a 6 y *Chebyshev* tipo 1 de orden superior a 8; para filtros pasa altos *Butterworth* de orden mayor a 40, elípticos de orden mayor a 10 y *Chebyshev* tipo 1 de orden superior a 16; para filtros pasa banda *Butterworth* de orden mayor a 7, elípticos de orden mayor a 5 y *Chebyshev* tipo 1 de orden superior a 5; para filtros

banda eliminada *Butterworth* de orden mayor a 5, elípticos de orden mayor a 5 y *Chebyshev* tipo 1 de orden superior a 7.

- ✓ El efecto audible de la señal de salida del filtro IIR no se diferencia claramente del obtenido con el filtro FIR a pesar que este presenta un buen funcionamiento independiente del orden del filtro, mejor SNR y valores Erms menores. Sin embargo, al comparar los espectros de las señales de salida de los filtros FIR e IIR estos últimos presentan un mejor comportamiento en la atenuación de la banda de rechazo.

- ✓ El procesador utilizado tiene un espacio para datos en la memoria de programa de 2047 posiciones para almacenar 2047 coeficientes de un filtro FIR, permitiendo implementar un filtro FIR de orden 2046, lo que hace posible tener un ancho de banda entre la frecuencia de paso y la frecuencia de corte, para un filtro pasa bajos o pasa altos, de hasta 10 Hz, usando una frecuencia de muestreo de 8 kHz, o hasta 35 Hz usando una frecuencia de muestreo de 48 kHz, permitiendo así discriminar frecuencias muy cercanas.

6. RECOMENDACIONES

El presente trabajo de investigación ha sentado las bases para una profundización en el campo del procesamiento de señales de audio mediante el uso de los DSPs. Los resultados obtenidos para los diferentes algoritmos implementados verifican el buen desempeño de los mismos, y sugieren el desarrollo de nuevos algoritmos y el perfeccionamiento de los ya implementados, mediante la incorporación de parámetros adicionales.

Las siguientes son las recomendaciones generales que hacen con el fin de seguir explorando el área del procesamiento de señales de audio y explotando los grandes beneficios y características de los DSPs:

- El algoritmo del efecto *flanger* puede ser modificado para agregar una línea de retroalimentación, dándole un efecto más metalizado. Esta línea de retroalimentación también puede ser agregada a los algoritmos de retardo y *chorus*.

- En los algoritmos de efecto de audio en amplitud, compresor y expansor, no se implementan los parámetros de tiempo de ataque y tiempo *release*, quedando como una mejora que se puede adicionar a estos algoritmos.

- Se puede crear una biblioteca de efectos de audio en lenguaje *assembly* para ejecutarse con la tarjeta de desarrollo EZ-KIT Lite ADSP-21065L, la cual puede ser completada implementando los algoritmos para los efectos de retardo: reverberación, *Automatic Double* y *Slapback Echo*; los algoritmos de efectoS de modulación: Doppler, Pitch Shifting, *Detune*, *Doubling*, *Vibrado*; y los algoritmos de efectos basados en amplitud, Trémolo, control de volumen.

- También se sugiere implementar los algoritmos de filtros combo FIR e IIR, filtros paramétricos, y filtros IIR en la forma directa II.

- Los algoritmos aquí tratados, así como todos los sugeridos, pueden ser también implementados en lenguaje C y establecer así una comparación en cuanto a rendimiento computacional con los implementados en lenguaje *assembly*.

- Puesto que el software utilizado para editar, compilar y depurar estos algoritmos es un versión vieja, se presentan inconvenientes al momento de determinar errores de programación que pueden pasar desapercibidos por el compilador. Por lo que se sugiere actualizar el VisualDSP++ a su más reciente versión.

BIBLIOGRAFÍA

[1] ANALOG DEVICES. ADSP-21000 Familiy Application Handbook Volume 1. Norwood, Mass. 1994. p. 90-111.

[2] _____ ADSP-21065L EZ-KIT Lite manual rev1. Norwood, Mass. 2000. p.1-88.

[3] _____ ADSP-2065L SHARC Technical Reference. Norwood, Mass. 1998. p. A_1-A_83.

[4] _____ . _____ Norwood, Mass. 1998. p. A_71-A_72.

[5] _____ . _____ Norwood, Mass. 1998. p. B_39-B_40.

[6] _____ . _____ Norwood, Mass. 1998. p. B_44-B_45.

[7] _____ . _____ Norwood, Mass. 1998. p. E_8-E_11, E_16-E_20, E_27-E_30.

[8] _____ . _____ Norwood, Mass. 1998. p. E_68-E_79.

[9] _____ . _____ Norwood, Mass. 1998. p. E_75-E_77.

[10] _____ . _____ Norwood, Mass. 1998. p. E_78-E_98.

[11] _____ . _____ Norwood, Mass. 1998. p. F_1-F_5.

[12] _____ ADSP-21065L SHARC User's Manual. Norwood, Mass. 1998. p. 2_12-2_25.

[13] _____ . _____ Norwood, Mass. 1998. p.2_41-2_49.

[14] _____ . _____ Norwood, Mass. 1998. p.3_1-3_34.

[15] _____ . _____ Norwood, Mass. 1998. p.3_35-3_49.

[16] _____ . _____ Norwood, Mass. 1998. p.4_1-4_17.

[17] _____ . _____ Norwood, Mass. 1998. p. 5_1-5_68.

[18] _____ . _____ Norwood, Mass. 1998. p. 8_1-8_52.

[19] _____ . _____ Norwood. Mass. 1998. p. 9_1-9_95.

[20] _____ . _____ Norwood, Mass. 1998. p. 10_1-10_41.

[21] _____ . _____ Norwood, Mass. 1998. p. 11_1-11_14.

[22] _____ Assembler Manual for ADSP-21xxx Family DSPs. Norwood, Mass. 1998. p. 3_1-3_30.

[23] _____ Linker and Utilities Manual for ADSP-21xxx Family DSPs. Norwood, Mass. 1998. p. 2_1-2_115.

[24] _____ VisualDSP++ User's Guide for ADSP-21xxx Family DSPs. Norwood, Mass. 1998. p. 2_1-2_78.

[25] _____ . _____ Norwood, Mass. 1998. p. 3_1-3_174.

[26] ARFIB, Daniel. Different Ways to Write Digital Audio Effects Programas. CNRS-LMA, 31 Chemin Joseph Aiguier. Marseille. 1997. p. 1-4.

[27] BOULANGER, Richard. The Csound Book: Perspectives inSoftware Syntesis, Sound Design, Siganl Processing and Programming. The MIT Press. p. 782.

[28] DATORRO, J. Effects Design, Part 2: Delay-Line Modulation and Chorus. J. Audio Eng. Soc, Vol. 45, No. 10. 1997. p. 764-788.

[29] DISCH, Sasha and ZÖLZER, Udo. Modulation and Delay Line Based Digital Audio Effects. Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects. Trandheim. 1999. p. 1-4.

[30] DUARTE, Rodrigo y HAO, Kei. Proyecto DSP: Filtro Notch Adaptivo. Procesadores Digitales de Señales. Instituto de Ingeniería Eléctrica, Facultad Ingeniería, Universidad de la República, Montevideo, Internet: http://iie.fing.edu.uy/ense/asign/sisdsp/proyectos/1998/lms/index.html. 1998.

[31] DUTILLEUX, Pierre. Filter, Delay y Modulation a Tutorial. ZMK, Institute for Music and Acousitcs. 1997. p. 1-8.

[32] FROCHT, Gustavo; BORCA Javier y FORCELLATI, Tabare. Proyecto DSP: Efectos de Audio en DSP Motorola 5600. Procesadores Digitales de Señales. Instituto de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad de la República. Montevideo, Internet: http://iie.fing.edu.uy/ense/asign/sisdsp/proyectos/1999/flanging/informe.htm. 1998.

[33] GUAUS, Enric; ROSSELL, Ivana y VALLEJOS, Lucas. Implementación de efectos para señales de audio. Dpto de acústica, Ingeniería La Salle, Universidad Ramon Llull. XXX Jornadas Nacionales de Acústica, Tecniacúsica 99, y Encuentro Ibérico de Acústica Avila. Barcelona. 1999

[34] LEHMAN, Scott. Harmony Central Effects Explained, Internet: http://www.harmony-central.com/Effects/Articcles. 1996.

[35] LOPEZ, Fernando. Fundamentals of Computer Generated Sound, Center for Computer Research in Music and Acoustics, Universidad de Stanford, Internet: http://ccrma-www.stanford.edu/CCRMA/Courses/220a/. 1996.

[36] MARQUEZ Pereira, David. Generación del efecto Flanger con un DSP TMS320C31. Departamento de Electrónica y Telecomunicaciones. Universidad de Aveiro, Portugal, Internet: http://webct.ua.pt/public/multimedia/Docs/Projectos/Proj5/relatorio.pdf. 2002.

[37] ORFANIDIS, S. J. .Introduction to Signal Processing, Prentice Hall. Englewood Cliffs, NJ. 1996. p. 325-330.

[38] _____ . _____ Prentice Hall. Englewood Cliffs, NJ. 1996. p. 355-383.

[39] PROAKIS, Jhon G. y MANOLAKIS, Dimitris G. Tratamiento digital de señales, Editorial Prentice Hall, España. 1997. p. 349-353.

[40] _____ . _____ Editorial Prentice Hall, España. 1997. p. 629-674.

[41] _____ . _____ Editorial Prentice Hall, España. 1997. p. 674-720.

[42] RAVASCHIO, Alexis. Proyecto DSP: Biblioteca de Efectos. Procesadores Digitales de Señales. Instituto de Ingeniería Eléctrica, Facultad Ingeniería, Universidad de la República. Montevideo, Internet: http://iie.fing.edu.uy/ense/asign/sisdsp/proyectos/2002/efectos/Inicio.html. 2002.

[43] SMITH, Steven W. The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Publishing, 1999. p. 503-514.

[44] SYNTRILLIUM, Software Corporation, Internet: http://www.syntrillium.com. 1997

[45] TOMARAKOS, Jhon. Analog Devices DSP Applications. Norwood, Mass. 1998. p. 1-16.

[46] YAMAHA. Procesadores de Audio Multiefecos, Internet: http://www.yamaha-europe.com. 2004

ANEXOS

CÓDIGO ALGORITMO DE SIMULACIÓN RETARDO SIMPLE

Archivo Fuente: *Retardo.asm*

```
/*****************************************************************************************************/
/***                              RETARDO SIMPLE      versión 1.0                                  ***/
/***                              Simulación                                                       ***/
/***                              Retardo.asm                                                      ***/
/***                                                                                               ***/
/*****************************************************************************************************/

/* ADSP-21065L Definición de Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

#define N       8000    // Longitud del archivo "Sen_200.dat" con Fs=8kHz
//#define       N       34731   // Longitud del archivo "Prueba.dat" con Fs=8kHz

/*Parámetros del Efecto Retardo*/
#define bo 0.5          // Coeficiente señal de entrada
#define bD 0.5          // Coeficiente señal de entrada  retardada
#define Retardo 416 /* Longitud de LineaRetardo
                                        Retardo=tDelay*Fs = 100ms*8000 */

.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;
        nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;
.var    LineaRetardo[Retardo];
.var    Entrada[N]=     "Sen_200.dat"; //Archivo de Entrada generado en Matlab
                                        //        "Prueba.dat"; //
.var    Salida[N];
.endseg;

.segment /pm pm_code;
_main:
                CALL setup_SDRAM (db);
                r0 = 0x3A2;             /* SDRDIV setting */
                r1 = 0x8E762800;        /* IOCTL setting  */
                ustat1=dm(SYSCON);  /* IMDW0=1 acceso block0 */
                bit set ustat1 IMDW0X; /* Son 40 bits (3 columnas) para compatibilidad*/
```

```
                    dm(SYSCON)=ustat1;

Inicializacion_Buffers:
B2 = LineaRetardo;  // Apunta a la primera dirección de LineaRetardo
L2 = @LineaRetardo; // Longitud de LineaRetardo
m2 = 1;                              // Incrementa el puntero en 1

B0=Entrada;     L0=0;    M0=1;
                                              M4=1;

LCNTR = L2,  DO Borra_Linea_Delay1 UNTIL LCE;
Borra_Linea_Delay1: dm(i2, m2) = 0;

                    CALL Efecto (DB);         /* Example delayed call instruction */
                    B4=Salida;
                    L4=0;
_exit:  IDLE;

/*---------------------------------------------------------------------*/
//Rutina para Delay simple  Procesamiento solo canal derecho

Efecto:
LCNTR=1000, DO (PC,9) UNTIL LCE;
f0=dm(i0,m0);   // Lee la muestra de entrada actual x(n)
f3=dm(i2,0);    // Lee muestra retardada en la LineaRetardo sin actualizar el puntero
f2=bo;              // Lee Coeficiente señal de entrada
f4=f0*f2;           // a0*x(n)
f2=bD;              // Lee Coeficiente señal de entrada    retardada
f5=f3*f2;           // aD*x(n-D)
f6=f4+f5;           // y(n) = a0*x(n) + aD*x(n-D)
dm(i2,m2)=f0;   // Escribe muestra actual en la LineaRetardo actualizando el puntero
dm(i4,m4)=f6;   // Almacena muestra de salida y(n)
RTS;

setup_SDRAM:
                            RTS (db);
                            dm(SDRDIV) = r0;
                            dm(IOCTL)  = r1;
.endseg;
```

Archivo *Linker*: *Retardo.ldf*

```
ARCHITECTURE(ADSP-21065L)
SEARCH_DIR( $ADI_DSP\21k\lib )
$LIBRARIES = lib060.dlb, libc.dlb, libio32.dlb, bmtools.dlb ;
$OBJECTS = $COMMAND_LINE_OBJECTS;   // No C code so do not include run time header

MEMORY
{
   seg_rth  { TYPE(PM RAM) START(0x00008000) END(0x000080ff) WIDTH(48) }
   seg_init { TYPE(PM RAM) START(0x0000c000) END(0x0000c00F) WIDTH(48) }
   seg_pmco { TYPE(PM RAM) START(0x0000c010) END(0x0000cfff) WIDTH(48) }
   seg_pmda { TYPE(PM RAM) START(0x0000d800) END(0x0000dfff) WIDTH(32) }
   seg_dmda { TYPE(DM RAM) START(0x03000000) END(0x030fdeff) WIDTH(32) }
     seg_heap { TYPE(DM RAM) START(0x030fdf00) END(0x030feeff) WIDTH(32) }
     seg_stak { TYPE(DM RAM) START(0x030fef00) END(0x030ffeff) WIDTH(32) }
     seg_bnk3 { TYPE(DM RAM) START(0x030FFF00) END(0x030FFFFF) WIDTH(32) }
 }

PROCESSOR p0
{
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS
   {
     seg_rth
     {
       INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
     } >seg_rth

     seg_init
     {
       INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
     } >seg_init

     seg_pmco
     {
       INPUT_SECTIONS($OBJECTS(seg_pmco) $OBJECTS(pm_code) $LIBRARIES(seg_pmco))
     } >seg_pmco

     seg_pmda
     {
      INPUT_SECTIONS( $OBJECTS(seg_pmda) $OBJECTS(pm_data) $LIBRARIES(seg_pmda))
     } >seg_pmda

     seg_bnk3
     {
      INPUT_SECTIONS( $OBJECTS(seg_bnk3) $LIBRARIES(seg_bnk3))
     } >seg_bnk3

     seg_dmda
     {
```

```
      INPUT_SECTIONS( $OBJECTS(seg_dmda) $OBJECTS(dm_data) $LIBRARIES(seg_dmda))
    } > seg_dmda

    stackseg
    {
      ldf_stack_space = .;
      ldf_stack_length = MEMORY_SIZEOF(seg_stak);
    } > seg_stak

    heap
    {
      ldf_heap_space = .;
      ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(seg_heap) - 1;
     ldf_heap_length = ldf_heap_end - ldf_heap_space;
    } > seg_heap

  }
}
```

CÓDIGO ALGORITMO EN TIEMPO REAL RETARDO SIMPLE

Archivo Fuente: *RetardoSimple.asm*

```
/*****************************************************************************************/
/**                                RETARDO SIMPLE      versión 1.1                  ***/
/**                                usando la tarjeta de evaluación                   **/
/**                                                                                  **/
/*****************************************************************************************/

/* ADSP-21065L Definicion Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL          Efecto_Audio;
.GLOBAL          Inicializacion_Bufers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;


.segment /dm    dmEfectos;

#define N       24000

/*      Parámetros Retardo simple       */
#define bo 0.5          // Coeficiente señal de entrada
#define bD 0.5          // Coeficiente señal de entrada  retrasada
#define Retardo         1000/* Longitud de LineaRetardo
                                        Retardo=tDelay*Fs = 100ms*12500 */

/*      Variables de almacenamiento    */
.var    Entrada[N];
.var    Salida[N];
.var    LineaRetardo[Retardo];

.endseg;

.segment /pm pm_code;

Inicializacion_Bufers:
B2 = LineaRetardo;              // Apunta a la primera dirección de LineaRetardo
L2 = @LineaRetardo;              // Longitud de LineaRetardo
m2 = 1;                         // Incrementa el puntero en 1

b0=Entrada;     l0=0;
b4=Salida;      l4=@Salida;

LCNTR = L2,  DO Borra_Linea_Delay1 UNTIL LCE;
Borra_Linea_Delay1: dm(i2, m2) = 0;
```

```
RTS;

/*------------------------------------------------------------------------*/
//Rutina para Delay simple  Procesamiento solo canal derecho

Efecto_Audio:
//Obtiene muestras entrada CODEC
r15 = DM(Right_Channel_In);              //Obtiene muestra de entrada canal derecho
r1 = -31;
f0 = float r15 by r1;                    //convierte a flotante

dm(i0,1)=f0;      //Almacena la muestra x(n) en el buffer de entrada
f3=dm(i2,0);    /* Lee muestra retrasada en la LineaRetardo1 sin actualizar
                   el puntero, el puntero, es decir obtiene x(n-D)*/
f2=bo;              // Lee Coeficiente señal de entrada
f4=f0*f2;           // bo*x(n)
f2=bD;              // Lee Coeficiente señal de entrada    retardada
f5=f3*f2;           // bD*x(n-D)
f6=f4+f5;           // y(n) = bo*x(n) + bD*x(n-D)
dm(i2,m2)=f0;       // Escribe muestra actual en la LineaRetardo actualizando el puntero
dm(i4,1)=f6;        // Almacena la muestra y(n) en el buffer de salida

r1 = 31;
r10 = fix f6 by r1;  //convierte a punto fijo
//Escribe la muestra de salida al CODEC
dm(Left_Channel_Out) = r10; {Muestra de salida  canal Izquierdo}
dm(Right_Channel_Out) = r10;{Muestra de salida  canal Derecho}

RTS;

.endseg;
```

CÓDIGO ALGORITMO DE SIMULACIÓN RETARDO MÚLTIPLE

Archivo Fuente: *Retardo_2Taps.asm*

```
/*****************************************************************************/
/**                    RETARDO 2 TAPS    versión 1.0                    ***/
/***                    Dos líneas de retardo                            ***/
/***                    Retardo_2Taps.asm                                ***/
/***                                                                     ***/
/*****************************************************************************/

/* ADSP-21065L Definición Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

#define N       23740   // Longitud del archivo "Prueba.dat" con Fs=8kHz

/*      Parámetros Retardo Múltiple 2 Taps*/
#define bo              0.3     // Coeficiente señal de entrada
#define bD1             0.35    // Coeficiente señal de entrada retrasada con D1
#define bD2             0.35    // Coeficiente señal de entrada retrasada con D2
#define Retardo1        1000    // Retardo1= Longitud de LineaRetardo1
#define Retardo2        480     /* Retardo2 = Longitud de LineaRetardo2
                                                        Retardo=tDelay*Fs = 64ms*12500 */

.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;
        nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;
.var    LineaRetardo1[Retardo1];        /*Define el buffer que almacenan las muestras
                                          de entrada para el Retardo1*/
.var    LineaRetardo2[Retardo2];        /*Define el buffer que almacenan las muestras
                                          de entrada para el Retardo2*/
.var    Entrada[N]=     "Prueba.dat"; //Archivo de Entrada generado en Matlab
                                      //  "Seno_200.dat"
.var    Salida[N];
.endseg;

.segment /pm pm_code;
_main:
                CALL setup_SDRAM (db);
                r0 = 0x3A2;                 /* SDRDIV setting */
                r1 = 0x8E762800;            /* IOCTL setting  */
                ustat1=dm(SYSCON);      /* IMDW0=1 acceso block0  */
                bit set ustat1 IMDW0X;  /* son 40 bits (3 columnas) para compatibilidad*/
                dm(SYSCON)=ustat1;
```

```
Inicializacion_Bufers:
B2 = LineaRetardo1;              // Apunta a la primera dirección de LineaRetardo1
L2 = @LineaRetardo1;             // Longitud de LineaRetardo1
M2 = 1;                          // Incrementa el puntero en 1

B3 = LineaRetardo2;              // Apunta a la primera dirección de LineaRetardo2
L3 = @LineaRetardo2;             // Longitud de LineaRetardo2
M3 = 1;                          // Incrementa el puntero en 1

B0 = Entrada;   L0=0;   M0=1;
                        M4=1;

LCNTR = L2,  DO Borra_LineaRetardo1 UNTIL LCE;
Borra_LineaRetardo1: dm(i2, m2) = 0;

LCNTR = L3,  DO Borra_LineaRetardo2 UNTIL LCE;
Borra_LineaRetardo2: dm(i3,m3) = 0;


                CALL Efecto (DB);
                B4=Salida;
                L4=0;
_exit:  IDLE;

/*-------------------------------------------------------------------------------------------*/
//Rutina para Retardo de 2 Taps  Procesamiento solo canal derecho

Efecto:
 LCNTR=N, DO (PC,14) UNTIL LCE;
 f0=dm(i0,m0);           // Lee muestra actual x(n)
 f3=dm(i2,0);            /* Lee muestra retrasada en la LineaRetardo1 sin actualizar el puntero,
                            es decir obtiene x(n-D1)*/
 f1=dm(i3,0);           /* Lee muestra retrasada en la LineaRetardo2 sin actualizar el puntero,
                            es decir obtiene x(n-D2)*/
 f7=bD2;                 // Lee Coeficiente señal de entrada retrasada con Retardo2
 f9=f1*f7;               // bD2*x(n-D2)
 f2=bo;                  // Lee Coeficiente señal de entrada
 f4=f0*f2;               // bo*x(n)
 f8=bD1;                 // Lee Coeficiente señal de entrada retrasada con Retardo1
 f5=f3*f8;               // bD1*x(n-D1)
 f6=f4+f5;               // bo*x(n) + bD*x(n-D1)
 f10=f6+f9;              // y(n) = bo*x(n) + bD1*x(n-D1) + bD2*x(n-D2)
 dm(i2,m2)=f0;           // Escribe muestra actual en la LineaRetardo1 actualizando el puntero
 dm(i3,m3)=f0;           // Escribe muestra actual en la LineaRetardo2 actualizando el puntero
 dm(i4,m4)=f10;          // Almacena muestra de salida y(n)
RTS;

setup_SDRAM:
                        RTS (db);
                        dm(SDRDIV) = r0;
                        dm(IOCTL)  = r1;
.endseg;
```

Archivo Fuente: *Retardo_6Taps.asm*

```
/*****************************************************************************/
/**                        RETARDO MULTIPLE  versión 1.1          ***/
/**                        6 TAPS                                 ***/
/**                        Simulación                             ***/
/**                                                               ***/
/*****************************************************************************/

/* ADSP-21065L Definición Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

//#define       N       34731           // Muestras de la señal de entrada
#define N       8000            // Muestras de la señal de entrada

/*      Parámetros Retardo 6 Taps       */
#define bo              0.17            // Coeficiente señal de entrada
#define bD1             0.17            // Coeficiente señal de entrada  retrasada con D1
#define bD2             0.17            // Coeficiente señal de entrada  retrasada con D2
#define bD3             0.17            // Coeficiente señal de entrada  retrasada con D3
#define bD4             0.17            // Coeficiente señal de entrada  retrasada con D4
#define bD5             0.17            // Coeficiente señal de entrada  retrasada con D5
#define bD6             0.17            // Coeficiente señal de entrada  retrasada con D6

#define Retardo1        816     /*      Retardo1= Longitud de LineaRetardo1
                                        Retardo=tDelay*Fs = 64ms*12500      */

#define Retardo2        1408    // Retardo2= Longitud de LineaRetardo2
#define Retardo3        2016    // Retardo3= Longitud de LineaRetardo3
#define Retardo4        2608    // Retardo1= Longitud de LineaRetardo4
#define Retardo5        3216    // Retardo1= Longitud de LineaRetardo5
#define Retardo6        3808    // Retardo1= Longitud de LineaRetardo6

.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;
        nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;
.var    LineaRetardo1[Retardo1];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo1*/
.var    LineaRetardo2[Retardo2];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo2*/
.var    LineaRetardo3[Retardo3];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo3*/
.var    LineaRetardo4[Retardo4];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo4*/
.var    LineaRetardo5[Retardo5];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo5*/
```

```
.var     LineaRetardo6[Retardo6];      /*Define el buffer que almacenan las muestras de entrada
                                              para el Retardo5*/

.var     Entrada[N]="Sen_200.dat";     //Archivo de Entrada generado en Matlab "Prueba.dat";
.var     Salida[N];                    // Buffer que almacena la salida y(n)
.endseg;


.segment /pm pm_code;
_main:
                CALL setup_SDRAM (db);
                r0 = 0x3A2;                /* SDRDIV setting */
                r1 = 0x8E762800;           /* IOCTL setting  */
                ustat1=dm(SYSCON);      /* IMDW0=1 acceso block0  */
                bit set ustat1 IMDW0X;     /* son 40 bits (3 columnas) para compatibilidad*/
                dm(SYSCON)=ustat1;

Inicializacion_Bufers:
 B1 = LineaRetardo1;          // Apunta a la primera dirección de LineaRetardo
 L1 = @LineaRetardo1;         // Longitud de LineaRetardo1
 m1 = 1;                      // Incrementa el puntero en 1 para Retardo1

 B2 = LineaRetardo2;          // Apunta a la primera dirección de LineaRetardo2
 L2 = @LineaRetardo2;         // Longitud de LineaRetardo2
 m2 = 1;                     // Incrementa el puntero en 1 para Retardo2

 B3 = LineaRetardo3;          // Apunta a la primera dirección de LineaRetardo3
 L3 = @LineaRetardo3;          // Longitud de LineaRetardo3
 m3 = 1;                     // Incrementa el puntero en 1 para Retardo3

 B4 = LineaRetardo4;          // Apunta a la primera dirección de LineaRetardo4
 L4 = @LineaRetardo4;          // Longitud de LineaRetardo4
 m4 = 1;                     // Incrementa el puntero en 1 para Retardo4

 B5 = LineaRetardo5;          // Apunta a la primera dirección de LineaRetardo5
 L5 = @LineaRetardo5;         // Longitud de LineaRetardo5
 m5 = 1;                     // Incrementa el puntero en 1 para Retardo5

 B6 = LineaRetardo6;          // Apunta a la primera dirección de LineaRetardo6
 L6 = @LineaRetardo6;         // Longitud de LineaRetardo6
 m6 = 1;                     // Incrementa el puntero en 1 para Retardo6

 B0=Entrada;                  // Apunta a la primera dirección de Entrada
 L0=0;                        // Entrada no es un buffer circular
 M0=1;                        // Incrementa el puntero de Entrada en 1
 M7=1;                        // Incrementa el  puntero de Salida en 1


 LCNTR = L1,  DO Borra_LineaRetardo1 UNTIL LCE;
 Borra_LineaRetardo1: dm(i1, m1) = 0;

 LCNTR = L2,  DO Borra_LineaRetardo2 UNTIL LCE;
 Borra_LineaRetardo2: dm(i2,m2) = 0;
```

```
 LCNTR = L3,  DO Borra_LineaRetardo3 UNTIL LCE;
 Borra_LineaRetardo3: dm(i3, m3) = 0;

 LCNTR = L4,  DO Borra_LineaRetardo4 UNTIL LCE;
 Borra_LineaRetardo4: dm(i4,m4) = 0;

 LCNTR = L5,  DO Borra_LineaRetardo5 UNTIL LCE;
 Borra_LineaRetardo5: dm(i5, m5) = 0;

 LCNTR = L6,  DO Borra_LineaRetardo6 UNTIL LCE;
 Borra_LineaRetardo6: dm(i6,m6) = 0;

                    CALL Efecto (DB);
                    B7=Salida;
                    L7=0;
_exit:  IDLE;

/*---------------------------------------------------------------------*/
//Rutina para Retardo Multiple  Procesamiento solo canal derecho

Efecto:
 LCNTR=N, DO (PC,34) UNTIL LCE;
 f0=dm(i0,m0);              // Lee muestra actual x(n)

 f1=dm(i1,0);       /* Lee muestra retrasada en la LineaRetardo1 sin actualizar  el puntero, es decir
obtiene x(n-D1)*/
 f2=dm(i2,0);       /* Lee muestra retrasada en la LineaRetardo2 sin actualizar  el puntero, es decir
obtiene x(n-D2)*/
 f3=dm(i3,0);       /* Lee muestra retrasada en la LineaRetardo3 sin actualizar  el puntero, es decir
obtiene x(n-D3)*/
 f4=dm(i4,0);       /* Lee muestra retrasada en la LineaRetardo4 sin actualizar  el puntero, es decir
obtiene x(n-D4)*/
 f5=dm(i5,0);       /* Lee muestra retrasada en la LineaRetardo5 sin actualizar  el puntero, es decir
obtiene x(n-D5)*/
 f6=dm(i6,0);       /* Lee muestra retrasada en la LineaRetardo6 sin actualizar  el puntero, es decir
obtiene x(n-D6)*/

 f7=bD1;                        // Lee Coeficiente señal de entrada retrasada con Retardo1
 f8=f1*f7;                      // bD1*x(n-D1)
 f7=bD2;                        // Lee Coeficiente señal de entrada retrasada con Retardo2
 f9=f2*f7;                      // bD2*x(n-D2)
 f10=f8+f9;                     // bD1*x(n-D1) + bD2*x(n-D2)

 f7=bD3;                        // Lee Coeficiente señal de entrada retrasada con Retardo3
 f8=f3*f7;                      // bD3*x(n-D3)
 f7=bD4;                        // Lee Coeficiente señal de entrada retrasada con Retardo4
 f9=f4*f7;                      // bD4*x(n-D4)
 f11=f8+f9;                     // bD3*x(n-D3) + bD4*x(n-D4)
 f7=bD5;                        // Lee Coeficiente señal de entrada retrasada con Retardo5
 f8=f5*f7;                      // bD5*x(n-D5)
 f7=bD6;                        // Lee Coeficiente señal de entrada retrasada con Retardo6
 f9=f6*f7;                      // bD6*x(n-D6)
 f12=f8+f9;                     // bD5*x(n-D5) + bD6*x(n-D6)
 f7=bo;                         // Lee Coeficiente señal de entrada
```

```
f13=f0*f7;                          // bo*x(n)

f14=f10+f11;            // bD1*x(n-D1) + bD2*x(n-D2) + bD3*x(n-D3) + bD4*x(n-D4)
f15=f12+f13;            // bD5*x(n-D5) + bD6*x(n-D6) + bo*x(n)
f7=f14+f15;             /*y(n)=bo*x(n)+bD1*x(n-D1)+bD2*x(n-D2)+bD3*x(n-D3)+
                                bD4*x(n-D4)+bD5*x(n-D5)+bD6*x(n-D6)+bo*x(n)*/

dm(i1,m1)=f0;           // Escribe muestra actual en la LineaRetardo1 actualizando el puntero
dm(i2,m2)=f0;           // Escribe muestra actual en la LineaRetardo2 actualizando el puntero
dm(i3,m3)=f0;           // Escribe muestra actual en la LineaRetardo3 actualizando el puntero
dm(i4,m4)=f0;           // Escribe muestra actual en la LineaRetardo4 actualizando el puntero
dm(i5,m5)=f0;           // Escribe muestra actual en la LineaRetardo5 actualizando el puntero
dm(i6,m6)=f0;           // Escribe muestra actual en la LineaRetardo6 actualizando el puntero

dm(i7,m7)=f7; // Almacena muestra de salida y(n)

RTS;

setup_SDRAM:
                        RTS (db);
                        dm(SDRDIV) = r0;
                        dm(IOCTL)  = r1;
.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL RETARDO MÚLTIPLE

Archivo Fuente: *RetardoMultimple6.asm*

```
/*****************************************************************************************/
/**             RETARDO MULTIPLE  version 1.1                                          ***/
/**             6 TAPS                                                                 ***/
/**             usando la tarjeta de evaluación                                         **/
/**             RetardoMultimple6.asm                                                   **/
/*****************************************************************************************/

/* ADSP-21065L Definicion Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL          Efecto_Audio;
.GLOBAL         Inicializacion_Bufers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;


.segment /dm    dmEfectos;
#define N       24000           // Muestras de la señal de entrada

/*      PARÁMETROS RETARDO MÚLTIPLE 6 TAPS */
#define bo              0.17            // Coeficiente señal de entrada
#define bD1             0.17            // Coeficiente señal de entrada  retrasada con D1
#define bD2             0.17            // Coeficiente señal de entrada  retrasada con D2
#define bD3             0.17        // Coeficiente señal de entrada      retrasada con D3
#define bD4             0.17        // Coeficiente señal de entrada      retrasada con D4
#define bD5             0.17            // Coeficiente señal de entrada  retrasada con D5
#define bD6             0.17     // Coeficiente señal de entrada  retrasada con D6

#define Retardo1        176     /* Retardo1= Longitud de LineaRetardo1+
                                   Retardo=tDelay*Fs = 64ms*12500 */
#define Retardo2        288      // Retardo2= Longitud de LineaRetardo2
#define Retardo3        608     // Retardo1= Longitud de LineaRetardo3
#define Retardo4        816     // Retardo1= Longitud de LineaRetardo4
#define Retardo5        1408    // Retardo1= Longitud de LineaRetardo5
#define Retardo6        2384    // Retardo1= Longitud de LineaRetardo6

/*      Variables de almacenamiento*/
.var    LineaRetardo1[Retardo1];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo1*/
.var    LineaRetardo2[Retardo2];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo2*/
.var    LineaRetardo3[Retardo3];        /*Define el buffer que almacenan las muestras de entrada
                                                para el Retardo3*/
```

```
.var     LineaRetardo4[Retardo4];      /*Define el buffer que almacenan las muestras de entrada
                                             para el Retardo4*/
.var     LineaRetardo5[Retardo5];      /*Define el buffer que almacenan las muestras de entrada
                                             para el Retardo5*/
.var     LineaRetardo6[Retardo6];      /*Define el buffer que almacenan las muestras de entrada
                                             para el Retardo5*/

.var     Entrada[N];                   // Señal de entrada x(n)
.var     Salida[N];                    // Buffer que almacena la salida y(n)
.endseg;

.segment /pm pm_code;

Inicializacion_Bufers:
 B1 = LineaRetardo1;          // Apunta a la primera dirección de LineaRetardo
 L1 = @LineaRetardo1;         // Longitud de LineaRetardo1
 m1 = 1;                      // Incrementa el puntero en 1 para Retardo1

 B2 = LineaRetardo2;          // Apunta a la primera dirección de LineaRetardo2
 L2 = @LineaRetardo2;         // Longitud de LineaRetardo2
 m2 = 1;                     // Incrementa el puntero en 1 para Retardo2

 B3 = LineaRetardo3;          // Apunta a la primera dirección de LineaRetardo3
 L3 = @LineaRetardo3;         // Longitud de LineaRetardo3
 m3 = 1;                     // Incrementa el puntero en 1 para Retardo3

 B4 = LineaRetardo4;          // Apunta a la primera dirección de LineaRetardo4
 L4 = @LineaRetardo4;         // Longitud de LineaRetardo4
 m4 = 1;                     // Incrementa el puntero en 1 para Retardo4

 B5 = LineaRetardo5;          // Apunta a la primera dirección de LineaRetardo5
 L5 = @LineaRetardo5;          // Longitud de LineaRetardo5
 m5 = 1;                     // Incrementa el puntero en 1 para Retardo5

 B6 = LineaRetardo6;          // Apunta a la primera dirección de LineaRetardo6
 L6 = @LineaRetardo6;         // Longitud de LineaRetardo6
 m6 = 1;                     // Incrementa el puntero en 1 para Retardo6

 LCNTR = L1,  DO Borra_LineaRetardo1 UNTIL LCE;
 Borra_LineaRetardo1: dm(i1, m1) = 0;

 LCNTR = L2,  DO Borra_LineaRetardo2 UNTIL LCE;
 Borra_LineaRetardo2: dm(i2,m2) = 0;

 LCNTR = L3,  DO Borra_LineaRetardo3 UNTIL LCE;
 Borra_LineaRetardo3: dm(i3, m3) = 0;

 LCNTR = L4,  DO Borra_LineaRetardo4 UNTIL LCE;
 Borra_LineaRetardo4: dm(i4,m4) = 0;

 LCNTR = L5,  DO Borra_LineaRetardo5 UNTIL LCE;
 Borra_LineaRetardo5: dm(i5, m5) = 0;

 LCNTR = L6,  DO Borra_LineaRetardo6 UNTIL LCE;
```

```
 Borra_LineaRetardo6: dm(i6,m6) = 0;


 B0=Entrada;            // Apunta a la primera dirección de Entrada
 L0=0;                  // Entrada no es un buffer circular
 M0=1;                  // Incrementa el puntero de Entrada en 1
 B7=Salida;             // Apunta a la primera dirección de Salida
 L7=@Salida;            // Salida no es un buffer circular
 M7=1;                  // Incrementa el  puntero de Salida en 1

RTS;

/*------------------------------------------------------------------------*/
//Rutina para Retardo Multiple  Procesamiento solo canal derecho

Efecto_Audio:
//Obtiene muestras entrada CODEC
r15 = DM(Right_Channel_In);          //Obtiene muestra de entrada canal derecho
r1 = -31;
f0= float r15 by r1;                 //convierte a flotante
dm(i0,m0)=f0;                        // Almacena muestra de entrada x(n)
 f1=dm(i1,0);      /* Lee muestra retrasada en la LineaRetardo1 sin actualizar  el puntero, es decir
                     obtiene x(n-D1)*/
 f2=dm(i2,0);      /* Lee muestra retrasada en la LineaRetardo2 sin actualizar  el puntero, es decir
                     obtiene x(n-D2)*/
 f3=dm(i3,0);      /* Lee muestra retrasada en la LineaRetardo3 sin actualizar  el puntero, es decir
                     obtiene x(n-D3)*/
 f4=dm(i4,0);      /* Lee muestra retrasada en la LineaRetardo4 sin actualizar  el puntero, es decir
                     obtiene x(n-D4)*/
 f5=dm(i5,0);      /* Lee muestra retrasada en la LineaRetardo5 sin actualizar  el puntero, es decir
                     obtiene x(n-D5)*/
 f6=dm(i6,0);      /* Lee muestra retrasada en la LineaRetardo6 sin actualizar  el puntero, es decir
                     obtiene x(n-D6)*/

 f7=bD1;                    // Lee Coeficiente señal de entrada retrasada con Retardo1
 f8=f1*f7;                  // bD1*x(n-D1)
 f7=bD2;                    // Lee Coeficiente señal de entrada retrasada con Retardo2
 f9=f2*f7;                  // bD2*x(n-D2)
 f10=f8+f9;                 // bD1*x(n-D1) + bD2*x(n-D2)
 f7=bD3;                    // Lee Coeficiente señal de entrada retrasada con Retardo3
 f8=f3*f7;                  // bD3*x(n-D3)
 f7=bD4;                    // Lee Coeficiente señal de entrada retrasada con Retardo4
 f9=f4*f7;                  // bD4*x(n-D4)
 f11=f8+f9;                 // bD3*x(n-D3) + bD4*x(n-D4)
 f7=bD5;                    // Lee Coeficiente señal de entrada retrasada con Retardo5
 f8=f5*f7;                  // bD5*x(n-D5)
 f7=bD6;                    // Lee Coeficiente señal de entrada retrasada con Retardo6
 f9=f6*f7;                  // bD6*x(n-D6)
 f12=f8+f9;                 // bD5*x(n-D5) + bD6*x(n-D6)
 f7=bo;                     // Lee Coeficiente señal de entrada
 f13=f0*f7;                 // bo*x(n)

 f14=f10+f11;           // bD1*x(n-D1) + bD2*x(n-D2) + bD3*x(n-D3) + bD4*x(n-D4)
 f15=f12+f13;           // bD5*x(n-D5) + bD6*x(n-D6) + bo*x(n)
```

```
 f7=f14+f15;              // y(n)=bo*x(n)+bD1*x(n-D1)+bD2*x(n-D2)+bD3*x(n-D3)+
                                  bD4*x(n-D4)+bD5*x(n-D5)+bD6*x(n-D6)+bo*x(n)

 dm(i1,m1)=f0;            // Escribe muestra actual en la LineaRetardo1 actualizando el puntero
 dm(i2,m2)=f0;            // Escribe muestra actual en la LineaRetardo2 actualizando el puntero
 dm(i3,m3)=f0;            // Escribe muestra actual en la LineaRetardo3 actualizando el puntero
 dm(i4,m4)=f0;            // Escribe muestra actual en la LineaRetardo4 actualizando el puntero
 dm(i5,m5)=f0;            // Escribe muestra actual en la LineaRetardo5 actualizando el puntero
 dm(i6,m6)=f0;            // Escribe muestra actual en la LineaRetardo6 actualizando el puntero

dm(i7,m7)=f7; // Almacena muestra de salida y(n)
r1 = 31;
r10 = fix f7 by r1; //convierte a punto fijo
//Escribe la muestra de salida al CODEC
dm(Left_Channel_Out) = r10; {Muestra de salida  canal Izquierdo}
dm(Right_Channel_Out) = r10;{Muestra de salida  canal Derecho}

rts;

.endseg;
```

CÓDIGO ALGORITMO DE SIMULACIÓN FLANGER

Archivo Fuente: *Flanger_sim.asm*

```
/*****************************************************************************/
/**                        FLANGER       versión 2.0                        ***/
/**                        LFO 2Hz cos4000                                  ***/
/**                        Simulación                                        **/
/**                        Flanger_sim.asm                                   **/
/*****************************************************************************/

/* ADSP-21065L Definición Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

#define N       8000            // Muestras de la señal de entrada

/*      Parámetros Flanger */
#define bo              0.5             // Coeficiente señal de entrada
#define bD              0.5             // Coeficiente señal de entrada  retrasada
#define Retardo          16             /* Retardo = Longitud de LineaRetardo
                                            Retardo=tDelay*Fs = 10ms*8000 */
#define RetardoMedio                    Retardo/2
#define LongTabla_LFO                   4000            //6250


.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;
        nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;

.var    LineaRetardo[Retardo+1];
.var    LFO[LongTabla_LFO]="LFO2_cos4000.dat";          //"LFO2_cos6250.dat";
.var    Sweep_Rate=4;                   // Parámetro del flanger que cambia la frecuencia del LFO
.var    Retardo_Variable;

.var    Entrada[N]="Sen_250.dat";       //Archivo de Entrada generado en Matlab "Prueba.dat";
.var    Salida[N];                      // Buffer que almacena la salida y(n)

.var    D_entero[N];
.var    Entrada_Retrasada[N];
.endseg;


.segment /pm pm_code;
_main:
                CALL setup_SDRAM (db);
```

```
                r0 = 0x3A2;                 /* SDRDIV setting */
                r1 = 0x8E762800;            /* IOCTL setting  */
                ustat1=dm(SYSCON);          /* IMDW0=1 acceso block0  */
                bit set ustat1 IMDW0X;      /* son 40 bits (3 columnas) para compatibilidad*/
                dm(SYSCON)=ustat1;

Inicializacion_Bufers:
B2 = LineaRetardo;          // Apunta a la primera dirección de LineaRetardo
L2 = @LineaRetardo;         // Longitud de LineaRetardo
m2 = 1;                     // Incrementa el puntero en 1 para retardo D2

B6=LFO;                     L6=@LFO;

LCNTR = L2,  DO Borra_LineaRetardo UNTIL LCE;
Borra_LineaRetardo: dm(i2,m2) = 0;

B0=Entrada;                 L0=0;   M0=1;
                                    M4=1;
B3=D_entero;                L3=0;   m3=1;
B1=Entrada_Retrasada;       L1=0;   m1=1;


                CALL Efecto (DB);
                B4=Salida;
                L4=0;
_exit:  IDLE;

/*-------------------------------------------------------------------------------------------------------------*/
//Rutina para Flanger  Procesamiento solo canal derecho

Efecto:
LCNTR=N, DO (PC,25) UNTIL LCE;

r8=RetardoMedio;            // Lee Retardo/2 (entero)
f1=float r8;                // Convierte en flotante
m6 = dm(Sweep_Rate);        // Lee incremento para buscar en la tabla LFO
f2=dm(i6,m6);               // Lee valor de LFO actualizando el puntero
f3=f1*f2;                   // D/2 * LFO(n)
f4=f1-f3;                   // D/2 - D/2 * LFO(n)
r5=fix f4;                  // Convierte a entero el retardo d(n)
dm(Retardo_Variable)=r5;

dm(i3,m3)=r5;

f0=dm(i0,m0);
f4=bo;                      // Lee coeficiente de entrada bo
f10=f4*f0;                  // bo*x(n)
dm(i2,0)=f0;                /* Almacena la muestra de entrada actual en la
                               línea de retardo sin actualizar el puntero*/
r14=dm(Retardo_Variable);
r14=-r14;
m2=r14;                     // Carga  modificador de puntero con -retardo
modify(i2,m2);              // Actualiza el puntero
r14=-r14;
```

```
m2=r14;                         // Carga  modificador de puntero con +retardo

f6=dm(i2,m2);                   // Lee la muestra retrasada con retardo d(n)
modify(i2,1);                   // Incrementa puntero
f5=bD;                          // Lee coeficiente de entrada retrasada bD

f11=f5*f6;                      // bD*x(n - d(n))
f12=f10+f11;                    // y(n) =  bo*x(n) + bD*x(n - d(n))
dm(i4,m4)=f12;

RTS;

setup_SDRAM:
                        RTS (db);
                        dm(SDRDIV) = r0;
                        dm(IOCTL)  = r1;
.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL FLANGER

Archivo Fuente: *Flanger.asm*

```
/*****************************************************************************************/
/**                    FLANGER          versión 2.0                                    ***/
/**                    Para uso con laTarjeta de Evaluación                            **/
/**                    Fs= 8kHz LFO es Onda Cos de 2 Hz 4000 muestras                  **/
/***                   Flanger.asm                                                     **/
/*****************************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Bufers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;

.segment /dm    dmEfectos;
#define N       32000
/*      Parámetros Flanger      */
#define bo              0.5             // Coeficiente señal de entrada
#define bD              0.5             // Coeficiente señal de entrada  retrasada
#define Retardo         48              /* Retardo = Longitud de LineaRetardo
                                           Retardo=tDelay*Fs = 10ms*8000 */
#define RetardoMedio            Retardo/2
#define LongTabla_LFO           4000            //6250

/*      Variables de almacenamiento*/
.var    LineaRetardo[Retardo+1];
.var    LFO[LongTabla_LFO]="LFO2_cos4000.dat";
.var    Sweep_Rate=1;           // Parámetro del flanger que cambia la frecuencia del LFO
.var    Retardo_Variable;

.var    Entrada[N];
.var    Salida[N];
.endseg;


/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Bufers:
B2 = LineaRetardo;  // Apunta a la primera dirección de LineaRetardo
L2 = @LineaRetardo; // Longitud de LineaRetardo
m2 = 1;                         // Incrementa el puntero en 1 para retardo D2

B6=LFO;                 L6=@LFO;

LCNTR = L2,  DO Borra_LineaRetardo UNTIL LCE;
```

```
Borra_LineaRetardo: dm(i2,m2) = 0;
B0=Entrada;     L0=@Entrada;          M0=1;
B4=Salida;      L4=@Salida;           M4=1;

RTS;

/*-------------------------------------------------------------------*/
//Rutina para FLANGER Procesamiento canal derecho
Efecto_Audio:

r8=RetardoMedio;                      // Lee Retardo/2 (entero)
f1=float r8;                          // Convierte en flotante
m6 = dm(Sweep_Rate);                  // Lee incremento para buscar en la tabla LFO
f2=dm(i6,m6);                         // Lee valor de LFO actualizando el puntero
f3=f1*f2;                             // D/2 * LFO(n)
f4=f1-f3;                             // D/2 - D/2 * LFO(n)
r5=fix f4;                            // Convierte a entero el retardo d(n)
dm(Retardo_Variable)=r5;

r15 = DM(Right_Channel_In);           //Obtiene entrada canal derecho
r1 = -31;
f0 = float r15 by r1;

dm(i0,m0)=f0;

f4=bo;                          // Lee coeficiente de entrada bo
f10=f4*f0;                      // bo*x(n)

dm(i2,0)=f0;                    /* Almacena la muestra de entrada actual en la
                                   línea de retardo sin actualizar el puntero*/

r14=dm(Retardo_Variable);
r14=-r14;
m2=r14;                         // Carga  modificador de puntero con -retardo
modify(i2,m2);                  // Actualiza el puntero
r14=-r14;
m2=r14;                         // Carga  modificador de puntero con +retardo
f6=dm(i2,m2);                   // Lee la muestra restrasada con retardo d(n)
modify(i2,1);                   // Incrementa puntero

f5=bD;                          // Lee coeficiente de entrada retrasada bD
f11=f5*f6;                      // bD*x(n - d(n))
f12=f10+f11;                    // y(n) =  bo*x(n) + bD*x(n - d(n))

dm(i4,m4)=f12;

r1 = 31;
r10 = fix f12 by r1;
DM(Left_Channel_Out)=r10;
DM(Right_Channel_Out)=r10;

RTS;

.endseg;
```

CÓDIGO ALGORITMO DE SIMULACIÓN CHORUS

Archivo Fuente: *Chorus3_sim.asm*

```
/*****************************************************************************/
/**             CHORUS        versión 2.0                                   ***/
/**             LFO 0.5 Hz 16000 muestras                                   ***/
/**             Simulación                                                   **/
/**             Chorus3_sim.asm                                              **/
/*****************************************************************************/

/* ADSP-21065L Definición Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

#define N       34731           // Muestras de la señal de entrada

/*      Parámetros Chorus*/
#define Retardo1                        120   // Retardo = retardo en ms*Fs
#define Retardo2                        200
#define Retardo3                        280
#define Retardo1_Medios                 Retardo1/2
#define Retardo2_Medios                 Retardo2/2
#define Retardo3_Medios                 Retardo3/2
#define LongTabla_LFO                   80000
#define bo      0.7
#define bD1     0.3
#define bD2     0.3
#define bD3     0.3

.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;
        nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;
.var    LineaRetardo1[Retardo1+1];                      // Linea de Retardo1
.var    LineaRetardo2[Retardo2+1];                      // Linea de Retardo2
.var    LineaRetardo3[Retardo3+1];                      // Linea de Retardo3
.var    LFO1[LongTabla_LFO]="LFO0.1_Sen80000.dat";      // 0.1 Hz
.var    LFO2[LongTabla_LFO]="LFO0.1_Sen80000.dat";      // 0.1 Hz
.var    LFO3[LongTabla_LFO]="LFO0.1_Sen80000.dat";      // 0.1 Hz
.var  Sweep_Rate1=1;
.var  Sweep_Rate2=5;                    //cambia la frecuencia del LFO de 0.5 a 2 Hz
.var  Sweep_Rate3=30;
.var  Sweep_Depth1=1.0;                 // 0.0 <Sweep_Depth1 <=1.0
.var  Sweep_Depth2=1.0;
.var  Sweep_Depth3=1.0;
.var  Retardo_Variable1;
```

```
.var      Retardo_Variable2;
.var      Retardo_Variable3;

.var      Entrada[N]="Prueba.dat";        //"Sen_250.dat"; //Archivo de Entrada generado en Matlab
.var      Salida[N];                      // Buffer que almacena la salida y(n)
.endseg;


.segment /pm pm_code;
_main:
                  CALL setup_SDRAM (db);
                  r0 = 0x3A2;           /* SDRDIV setting */
                  r1 = 0x8E762800;      /* IOCTL setting  */
                  ustat1=dm(SYSCON);    /* IMDW0=1 acceso block0  */
                  bit set ustat1 IMDW0X; /* son 40 bits (3 columnas) para compatibilidad*/
                  dm(SYSCON)=ustat1;

Inicializacion_Bufers:
B1 = LineaRetardo1;           L1 = @LineaRetardo1;          m1 = 1;
B2 = LineaRetardo2;           L2 = @LineaRetardo2;          m2 = 1;
B3 = LineaRetardo3;           L3 = @LineaRetardo3;          m3 = 1;
B4 = LFO1;                    L4 = @LFO1;
B5 = LFO2;                    L5 = @LFO2;
B6 = LFO3;                    L6 = @LFO3;

LCNTR = L1;  DO Borra_LineaRetardo1 UNTIL LCE;
Borra_LineaRetardo1: dm(i1, m1) = 0;

LCNTR = L2;  DO Borra_LineaRetardo2 UNTIL LCE;
Borra_LineaRetardo2: dm(i2, m2) = 0;

LCNTR = L3;  DO Borra_LineaRetardo3 UNTIL LCE;
Borra_LineaRetardo3: dm(i3, m3) = 0;

B0=Entrada;    L0=0;          M0=1;
                              M7=1;

                  CALL Efecto (DB);
                  B7=Salida;
                  L7=0;
_exit:  IDLE;

/*------------------------------------------------------------------------*/
//Rutina para Chorus  de 2 Lineas de Retardo  Procesamiento solo canal derecho

Efecto:
LCNTR=N, DO (PC,70) UNTIL LCE;

/*        PARTE Ia: Cálculo  retardo entero d1 mediante el LFO1                */
          r8=Retardo1_Medios;                 // 0.5*Retardo1=0.5*D1
          f1=float r8;                        // Convierte a flotante
          m4 = dm(Sweep_Rate1);               // Lee el incremento de frecuencia LFO1
          f2=dm(i4,m4);                       // Lee el valor del LFO1(n)
          f7=dm(Sweep_Depth1);                // Lee amplitud del LFO1
```

```
        f9=f7*f2;                       // Sweep_Depth1*LFO1(n)
        f4=f1*f9;                       // 0.5*D1*LFO1(n)
        f3=f1+f4;                       // d1(n)=0.5*D1 + 0.5*D1*LFO1(n)
        r5=fix f3;                      // Convierte al entero más cercano
        dm(Retardo_Variable1) = r5;     // Almacena en memoria d1(n) entero

/*      PARTE Ib: Cálculo  retardo entero d2 mediante el LFO2          */
        r8=Retardo2_Medios;             // 0.5*Retardo2=0.5*D2
         f1=float r8;                   // Convierte a flotante
         m5 = dm(Sweep_Rate2);          // Lee el incremento de frecuencia LFO2
         f2=dm(i5,m5);                  // Lee el valor del LFO2(n)
         f7=dm(Sweep_Depth2);           // Lee amplitud del LFO2
         f9=f7*f2;                      // Sweep_Depth2*LFO2(n)
         f4=f1*f9;                      // 0.5*D2*LFO2(n)
         f3=f1+f4;                      // d2(n)=0.5*D2 + 0.5*D2*LFO2(n)
         r5=fix f3;                     // Convierte al entero más cercano
        dm(Retardo_Variable2) = r5;     // Almacena en memoria d2(n) entero

/*      PARTE Ic: Cálculo  retardo entero d3 mediante el LFO3          */
        r8=Retardo3_Medios;             // 0.5*Retardo3=0.5*D3
        f1=float r8;                    // Convierte a flotante
         m6 = dm(Sweep_Rate3);          // Lee el incremento de frecuencia LFO3
         f2=dm(i6,m6);                  // Lee el valor del LFO3(n)
         f7=dm(Sweep_Depth3);           // Lee amplitud del LFO3
         f9=f7*f2;                      // Sweep_Depth3*LFO3(n)
         f4=f1*f9;                      // 0.5*D3*LFO3(n)
         f3=f1+f4;                      // d3(n)=0.5*D3 + 0.5*D3*LFO3(n)
         r5=fix f3;                     // Convierte al entero más cercano
        dm(Retardo_Variable3) = r5;     // Almacena en memoria d3(n) entero


/*      PARTE II: Obtiene muestra de entrada x(n)       */
        f0=dm(i0,m0);
        f5=bo;                          // Lee coeficiente de entrada bo
        f10=f5*f0;                      // bo*x(n)

/*      PARTE III: Obtiene muestra de entrada retrasada 1       */
        dm(i1,0)=f0;                    /* Almacena muestra actual de entrada x(n)
                                           en la línea de retardo 1, sin actualizar el puntero*/
        r14=dm(Retardo_Variable1);      // Obtiene el retardo entero d1(n)
         r14=-r14;
         m1=r14;
         modify(i1,m1);
         r14=-r14;
         m1=r14;
         f6=dm(i1,m1);                  // Obtiene la muestra retrasada x(n-d1(n))
         modify(i1,1);
         f5=bD1;                        // Lee ganancia entrada retradasa 1
        f11=f5*f6;                      // bD1*x(n-d1(n))

/*      PARTE IV: Obtiene muestra de entrada retrasada 2       */
        dm(i2,0)=f0;                    /* Almacena muestra actual de entrada x(n)
                                           en la línea de retardo 2, sin actualizar el puntero*/
        r14=dm(Retardo_Variable2);      // Obtiene el retardo entero d2(n)
```

```
        r14=-r14;
        m2=r14;
        modify(i2,m2);
        r14=-r14;
        m2=r14;
        f6=dm(i2,m2);                   // Obtiene la muestra retrasada x(n-d2(n))
        modify(i2,1);
        f5=bD2;                         // Lee ganancia entrada retradasa 2
        f13=f5*f6;                      // bD2*x(n-d2(n))

/*      PARTE V: Obtiene muestra de entrada retrasada 3       */
        dm(i3,0)=f0;                    /* Almacena muestra actual de entrada x(n)
                                           en la línea de retardo 3, sin actualizar el puntero*/
        r14=dm(Retardo_Variable3);      // Obtiene el retardo entero d3(n)
        r14=-r14;
        m3=r14;
        modify(i3,m3);
        r14=-r14;
        m3=r14;
        f6=dm(i3,m3);                   // Obtiene la muestra retrasada x(n-d3(n))
        modify(i3,1);
        f5=bD3;                         // Lee ganancia entrada retradasa 3
        f9=f5*f6;                       // bD3*x(n-d3(n))

/*      PARTE VI: Calcula salida y(n)   */
        f14=f11+f13;                    // bD1*x(n-d1(n)) + bD2*x(n-d2(n))
        f15=f14+f9;                     // bD1*x(n-d1(n)) + bD2*x(n-d2(n)) +  bD3*x(n-d3(n))
        f12=f15+f10;          // y(n)=bo*x(n) + bD1*x(n-d1(n)) + bD2*x(n-d1(n)) +  bD3*x(n-d3(n))
        dm(i7,m7)=f12;

RTS;

setup_SDRAM:
                        RTS (db);
                        dm(SDRDIV) = r0;
                        dm(IOCTL)  = r1;
.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL CHORUS

Archivo Fuente: *Chorus3.asm*

```
/*****************************************************************************/
/**             CHORUS        versión 2.0                                    ***/
/**             LFO 0.1 Hz 80000 muestras                                    ***/
/**             Tres líneas de Retardo                                       ***/
/**             Tarjeta de Evaluación                                        ***/
/**             Chorus3.asm                                                 ***/
/*****************************************************************************/

/* ADSP-21065L Definicion Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Bufers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;

.segment /dm    dmEfectos;

#define N       24000           // Muestras de la señal de entrada

/*      Parámetros Chorus*/
#define Retardo1                120             // Retardo = retardo en ms*Fs
#define Retardo2                160
#define Retardo3                200
#define Retardo1_Medios         Retardo1/2
#define Retardo2_Medios         Retardo2/2
#define Retardo3_Medios         Retardo3/2
#define LongTabla_LFO           80000
#define bo      0.7
#define bD1     0.3
#define bD2     0.3
#define bD3     0.3

.var    LineaRetardo1[Retardo1+1];                      // Linea de Retardo1
.var    LineaRetardo2[Retardo2+1];                       // Linea de Retardo2
.var    LineaRetardo3[Retardo3+1];                      // Linea de Retardo3
.var    LFO1[LongTabla_LFO]="LFO0.1_Sen80000.dat";      // 0.1 Hz
.var    LFO2[LongTabla_LFO]="LFO0.1_Sen80000.dat";      // 0.1 Hz
.var    LFO3[LongTabla_LFO]="LFO0.1_Sen80000.dat";      // 0.1 Hz
.var  Sweep_Rate1=1;
.var  Sweep_Rate2=2;                    //cambia la frecuencia del LFO de 0.5 a 2 Hz
.var  Sweep_Rate3=3;
.var  Sweep_Depth1=1.0;                 // 0.0 <Sweep_Depth1 <=1.0
.var  Sweep_Depth2=1.0;
.var  Sweep_Depth3=1.0;
```

```
.var      Retardo_Variable1;
.var      Retardo_Variable2;
.var      Retardo_Variable3;

.var      Entrada[N];                    //Archivo de Entrada generado en Matlab "Prueba.dat";
.var      Salida[N];                     // Buffer que almacena la salida y(n)
.endseg;


/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Bufers:
B1 = LineaRetardo1;          L1 = @LineaRetardo1;          m1 = 1;
B2 = LineaRetardo2;          L2 = @LineaRetardo2;          m2 = 1;
B3 = LineaRetardo3;          L3 = @LineaRetardo3;          m3 = 1;
B4 = LFO1;                   L4 = @LFO1;
B5 = LFO2;                   L5 = @LFO2;
B6 = LFO3;                   L6 = @LFO3;

LCNTR = L1;  DO Borra_LineaRetardo1 UNTIL LCE;
Borra_LineaRetardo1: dm(i1, m1) = 0;

LCNTR = L2;  DO Borra_LineaRetardo2 UNTIL LCE;
Borra_LineaRetardo2: dm(i2, m2) = 0;

LCNTR = L3;  DO Borra_LineaRetardo3 UNTIL LCE;
Borra_LineaRetardo3: dm(i3, m3) = 0;

B0=Entrada;L0=0;M0=1;
B7=Salida; L7=0;M7=1;

RTS;



/*-------------------------------------------------------------------------------------------------------------------*/
//Rutina para CHORUS Procesamiento solo canal derecho

Efecto_Audio:

/*        PARTE Ia: Cálculo  retardo entero d1 mediante el LFO1                    */
          r8=Retardo1_Medios;              // 0.5*Retardo1=0.5*D1
          f1=float r8;                     // Convierte a flotante
          m4 = dm(Sweep_Rate1);            // Lee el incremento de frecuencia LFO1
          f2=dm(i4,m4);                    // Lee el valor del LFO1(n)
          f7=dm(Sweep_Depth1);             // Lee amplitud del LFO1
          f9=f7*f2;                        // Sweep_Depth1*LFO1(n)
          f4=f1*f9;                        // 0.5*D1*LFO1(n)
          f3=f1+f4;                        // d1(n)=0.5*D1 + 0.5*D1*LFO1(n)
          r5=fix f3;                       // Convierte al entero más cercano
          dm(Retardo_Variable1) = r5;      // Almacena en memoria d1(n) entero
```

```
/*      PARTE Ib: Cálculo  retardo entero d2 mediante el LFO2          */
        r8=Retardo2_Medios;             // 0.5*Retardo2=0.5*D2
         f1=float r8;                   // Convierte a flotante
         m5 = dm(Sweep_Rate2);          // Lee el incremento de frecuencia LFO2
         f2=dm(i5,m5);                  // Lee el valor del LFO2(n)
         f7=dm(Sweep_Depth2);           // Lee amplitud del LFO2
         f9=f7*f2;                      // Sweep_Depth2*LFO2(n)
         f4=f1*f9;                      // 0.5*D2*LFO2(n)
         f3=f1+f4;                      // d2(n)=0.5*D2 + 0.5*D2*LFO2(n)
         r5=fix f3;                     // Convierte al entero más cercano
        dm(Retardo_Variable2) = r5;     // Almacena en memoria d2(n) entero

/*      PARTE Ic: Cálculo  retardo entero d3 mediante el LFO3          */
        r8=Retardo3_Medios;             // 0.5*Retardo3=0.5*D3
         f1=float r8;                   // Convierte a flotante
         m6 = dm(Sweep_Rate3);          // Lee el incremento de frecuencia LFO3
         f2=dm(i6,m6);                  // Lee el valor del LFO3(n)
         f7=dm(Sweep_Depth3);           // Lee amplitud del LFO3
         f9=f7*f2;                      // Sweep_Depth3*LFO3(n)
         f4=f1*f9;                      // 0.5*D3*LFO3(n)
         f3=f1+f4;                      // d3(n)=0.5*D3 + 0.5*D3*LFO3(n)
         r5=fix f3;                     // Convierte al entero más cercano
        dm(Retardo_Variable3) = r5;     // Almacena en memoria d3(n) entero

/*      PARTE II: Obtiene muestra de entrada x(n)                      */
        r15 = DM(Right_Channel_In);     //Obtiene entrada canal derecho
        r1 = -31;
        f0 = float r15 by r1;

        dm(i0,m0)=f0;
        f5=bo;                          // Lee coeficiente de entrada bo
        f10=f5*f0;                      // bo*x(n)

/*      PARTE III: Obtiene muestra de entrada retrasada 1              */
        dm(i1,0)=f0;                    /* Almacena muestra actual de entrada x(n)
                                           en la línea de retardo 1, sin actualizar el puntero*/
        r14=dm(Retardo_Variable1);      // Obtiene el retardo entero d1(n)
         r14=-r14;
         m1=r14;
         modify(i1,m1);
         r14=-r14;
         m1=r14;
         f6=dm(i1,m1);                  // Obtiene la muestra retrasada x(n-d1(n))
         modify(i1,1);
         f5=bD1;                        // Lee ganancia entrada retradasa 1
        f11=f5*f6;                      // bD1*x(n-d1(n))

/*      PARTE IV: Obtiene muestra de entrada retrasada 2               */
        dm(i2,0)=f0;                    /* Almacena muestra actual de entrada x(n)
                                           en la línea de retardo 2, sin actualizar el puntero*/
        r14=dm(Retardo_Variable2);      // Obtiene el retardo entero d2(n)
        r14=-r14;
        m2=r14;
        modify(i2,m2);
```

```
        r14=-r14;
        m2=r14;
        f6=dm(i2,m2);                       // Obtiene la muestra retrasada x(n-d2(n))
        modify(i2,1);
        f5=bD2;                             // Lee ganancia entrada retradasa 2
        f13=f5*f6;                          // bD2*x(n-d2(n))

/*      PARTE V: Obtiene muestra de entrada retrasada 3       */
        dm(i3,0)=f0;                        /* Almacena muestra actual de entrada x(n)
                                               en la línea de retardo 3, sin actualizar el puntero*/
        r14=dm(Retardo_Variable3);     // Obtiene el retardo entero d3(n)
        r14=-r14;
        m3=r14;
        modify(i3,m3);
        r14=-r14;
        m3=r14;
        f6=dm(i3,m3);                       // Obtiene la muestra retrasada x(n-d3(n))
        modify(i3,1);
        f5=bD3;                             // Lee ganancia entrada retradasa 3
        f9=f5*f6;                           // bD3*x(n-d3(n))

/*      PARTE VI: Calcula salida y(n)   */
        f14=f11+f13;                        // bD1*x(n-d1(n)) + bD2*x(n-d2(n))
        f15=f14+f9;             // bD1*x(n-d1(n)) + bD2*x(n-d2(n)) +  bD3*x(n-d3(n))
        f12=f15+f10;            // y(n)=bo*x(n) + bD1*x(n-d1(n)) + bD2*x(n-d1(n)) +  bD3*x(n-d3(n))

        dm(i7,m7)=f12;

        r1 = 31;
        r10 = fix f12 by r1;
        DM(Left_Channel_Out)=r10;
        DM(Right_Channel_Out)=r10;

        rts;

.endseg;
```

CÓDIGO ALGORITMO DE SIMULACIÓN COMPRESOR PICO

Archivo Fuente: *Compresor.asm*

```
/*****************************************************************************/
/**                COMPRESOR PICO     versión 1.0                  ***/
/**                Simulación                                      ***/
/***               Compresor.asm                                   ***/
/*****************************************************************************/

/* ADSP-21065L Definicion Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

/*      Parámetros Compresor*/
#define Umbral 0.35             // Threshold  o Umbral donde empieza a comprimir
#define Ratio  0.05             // Factor de Compresión

#define N       8000            //34731         //  Muestras de Entrada


.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;
        nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;
.var    Entrada[N]="Sen_200.dat";//"Prueba.dat";//
.var    Salida[N];
.endseg;

.segment /pm pm_code;
_main:
                CALL setup_SDRAM (db);
                r0 = 0x3A2;                     /* SDRDIV setting */
                r1 = 0x8E762800;                /* IOCTL setting  */
                ustat1=dm(SYSCON);              /* IMDW0=1 acceso block0  */
                bit set ustat1 IMDW0X;          /* son 40 bits (3 columnas) para compatibilidad*/
                dm(SYSCON)=ustat1;

Inicializacion_Bufers:

B0=Entrada;     L0=@Entrada;  M0=1;
                              M4=1;

                CALL Efecto (DB);
                B4=Salida;
                L4=@Salida;
_exit:  IDLE;
```

```
/*-----------------------------------------------------------------------*/
//Rutina para Compresor Procesamiento solo canal derecho

Efecto:
LCNTR=5, DO (PC,7) UNTIL LCE;//20
Inicio_1:
f0=dm(i0,m0);                   // Lee la muestra de entrada actual x(n)
f1=abs f0;                      // Toma el valor absoluto de la muestra x(n)
f2=Umbral;                      // Lee el valor del Umbral
f3=Ratio;                       // Lee el valor del Factor de Compresión
comp(f1,f2);                    // x(n) < Umbral ?
if LT jump Salida_Igual;        // Si-->Salida_Igual
Jump Salida_Comprimida;         // No-->Salida_Comprimida

Salida_Igual:
dm(i4,m4)=f0;                   // Almacena muestra de salida en un buffer
jump Inicio;                    // Regresa al inicio del programa

Salida_Comprimida:              // 4 Instrucciones
f5=f1-f1;
comp(f0,f5);                    // x(n)>0
if GT jump Salida_Positiva;     // Si --> Salida_Positiva
jump Salida_Negativa;           // No --> Salida_Negativa

Salida_Positiva:
f4=f0-f2;                       // x(n)-Umbral
f4=f4*f3;                       // (x(n)-Umbral)* Ratio
f4=f4+f2;                       // y(n)= Umbral + (x(n)-Umbral)* Ratio
dm(i4,m4)=f4;                   // Almacena muestra de salida en un buffer
jump Inicio;                    // Regresa al inicio del programa

Salida_Negativa:
f4=f0+f2;                       // x(n)-Umbral
f4=f4*f3;                       // (x(n)+Umbral)* Ratio
f4=f4-f2;                       // y(n)= -Umbral + (x(n)+Umbral)* Ratio
dm(i4,m4)=f4;                   // Almacena muestra de salida en un buffer
jump Inicio;                    // Regresa al inicio del programa

RTS;

setup_SDRAM:
                        RTS (db);
                        dm(SDRDIV) = r0;
                        dm(IOCTL)  = r1;
.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL COMPRESOR PICO

Archivo Fuente: *Compresor_TR.asm*

```
/*****************************************************************************/
/**              COMPRESOR  version 2.0          EZ-KIT          ***/
/**              Compresor_TR.asm                                  **/
/*****************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"


.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Buffers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;

.segment /dm    dmEfectos;
#define         N                       24000

/*          Parámetros Compresor*/
#define         Umbral                  0.35    // Threshold  o Umbral donde empieza a comprimir
#define         Ratio_Compresor         0.05    // Factor de Compresión
.var            Entrada[N];
.var            Salida[N];
.endseg;

/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
b0=Entrada;             l0=0;                   m0=1;
b4=Salida;              l4=@Salida;             m4=1;

RTS;

/*-----------------------------------------------------------------------*/
//Rutina para Compresor RMS Canal Derecho

Efecto_Audio:
r15 = DM(Right_Channel_In);    //Obtiene entrada canal derecho
r1 = -31;
f0 = float r15 by r1;
dm(i0,m0)=f0;

Inicio_1:
f1=abs f0;                              // Toma el valor absoluto de la muestra x(n)
f2=Umbral;                              // Lee el valor del Umbral
f3=Ratio_Compresor;                     // Lee el valor del Factor de Compresion
```

```
comp(f1,f2);                    // x(n) < Umbral ?
if LT jump Salida_Igual;        // Si--> Fin: y(n) = x(n)
Jump Salida_Compresor;          // No-->Salida

Salida_Igual:
f4=f0;                          // y(n) = x(n)
jump Fin_1;

Salida_Compresor:
f5=f1-f1;
comp(f0,f5);                    // x(n) > 0 ?
if GT jump Salida_Positiva;     // Si--> Salida_Positiva
jump Salida_Negativa;           // No--> Salida_Negativa

Salida_Positiva:
f4=f0-f2;                       // x(n)-Umbral
f4=f4*f3;                       // [x(n)-Umbral]* Ratio
f4=f4+f2;                       // y(n)= Umbral + [x'(n)-Umbral ]* Ratio
jump Fin_1;

Salida_Negativa:
f4=f0+f2;                       // x(n)+Umbral
f4=f4*f3;                       // [x'(n)+Umbral]* Ratio
f4=f4-f2;                       // y(n)= -Umbral + (x'(n)+Umbral)* Ratio
jump Fin_1;

//------------------------------------------------------------------------

Fin_1:
dm(i4,m4)=f4;                   // Almacena muestra de salida en un buffer
r1 = 31;
r10 = fix f4 by r1;
DM(Left_Channel_Out)=r10;
DM(Right_Channel_Out)=r10;
rts;

.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL COMPRESOR RMS

Archivo Fuente: *Compresor_RMS_TR.asm*

```
/*****************************************************************************/
/**                    COMPRESOR  versión 2.0          EZ-KIT    ***/
/**                    Detección RMS                               ***/
/**                    Compresor_RMS_TR.asm                        ***/
/**                                                                ***/
/*****************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"


.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Buffers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;

.segment /dm    dmEfectos;
#define         N               8000        // Muestras de entrada y salida
#define         M               500         // Muestras para estimar RMS
#define         Umbral          0.1         // Threshold o umbral donde empieza a comprimir
#define         Ratio_Compresor 0.05        // Factor de Compresión
#define         Reciproco_M     0.002       // 1/M, ponderación

.var            Muestra_Entrada;                    // Muestra de entrada actual x(n)
.var            Muestra_Entrada_RMS;                // Estimación RMS de x(n)
.var            Suma_Pond_Cua = 0.0;                // SUM[x^2(k)*(1/M)], n-(M-1)<= k <= n
.var            Linea_MCP[M];                       // línea entrada ponderada al cuadrado
.var            Entrada[N];                         // Buffer de entrada
.var            Salida[N];                          // Buffer de salida
.endseg;


/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
B0=Entrada;             L0=0;               M0=1;
B4=Salida;              L4=@Salida;         M4=1;
B2=Linea_MCP;           L2=@Linea_MCP;      M2=1;

f13=0.0;
LCNTR = M, DO  Borra_Linea_RMS UNTIL LCE;
Borra_Linea_RMS: dm(i2, m2) = f13;

RTS;
```

```
/*-------------------------------------------------------------------------------------------------------------*/
// Rutina para Compresor RMS Canal Derecho

Efecto_Audio:

        r15 = DM(Right_Channel_In);             // Obtiene entrada canal derecho
        r1 = -31;
        f0 = float r15 by r1;
        dm(i0,m0)=f0;
        DM(Muestra_Entrada) = f0;

Estimacion_RMS:
        f5 = dm(Reciproco_M);           // 1/M
        f2 = abs f0;                    // abs x(n)
        f1 = f2;                        // F1= abs x(n)
        f0 = f2 * f1;                   // F0 = x(n)*x(n)
        f0 = f0 * f5;                   // F0=   x(n)*x(n)*(1/M)
        f1 = dm(i2,0);                  /* Obtiene el valor viejo de la línea de entradas
                                           al cuadrado ponderadas, sin actualizar el puntero*/
        f10 = DM(Suma_Pond_Cua);        // Obtiene el resultado de la última suma
        f10 = f10 + f0;                 // Suma la nueva entrada al cuadrado ponderada
        f10 = f10 - f1;                 // Resta la más vieja entrada al cuadrado ponderada
        DM(Suma_Pond_Cua) = f10;        // Guarda el resultado de la nueva suma
        dm(i2,m2) = f0;                 /* Guarda la entrada actual al cuadrado ponderada,
                                           actualizando el puntero*/

Newton_Raphson:
        f8 = 3.0;
        f2 = 0.5;
        f4 = RSQRTS f10;                        // 1/ [(F10)^0.5]
        f1 = f4;
        f12 = f4 * f1;                          // F12=X0^2
        f12 = f12 * f0;                         // F12=C*X0^2
        f4 = f2 * f4, f12 = f8 - f12;           // F4=.5*X0, F10=3-C*X0^2
        f4 = f4 * f12;                          // F4=X1=.5*X0(3-C*X0^2)
        f1 = f4;
        f12 = f4 * f1;                          // F12=X1^2
        f12 = f12 * f0;                         // F12=C*X1^2
        f4 = f2 * f4, f12 = f8 - f12;           // F4=.5*X1, F10=3-C*X1^2
        f4 = f4 * f12;                          // F4=X2=.5*X1(3-C*X1^2)
        f1 = f4;
        f12 = f4 * f1;                          // F12=X2^2
        f12 = f12 * f0;                         // F12=C*X2^2
        f4 = f2 * f4, f12 = f8 - f12;           // F4=.5*X2, F10=3-C*X2^2
        f4 = f4 * f12;                          // F4=X3=.5*X2(3-C*X2^2)
        f10 = f4 * f10;                         // X=sqrt(Y)=Y/sqrt(Y)
        DM(Muestra_Entrada_RMS) = f10;
        //dm(i3,m3)=f10;

Comparador:
        f2=Umbral;                              // Lee el valor del Umbral
        f3=Ratio_Compresor;                     // Lee el valor del Factor de Compresión
        f0 = DM(Muestra_Entrada);               // Obtiene la muestra actual de entrada x(n)
        f1=DM(Muestra_Entrada_RMS);             // F1= x(n)rms
```

```
        f11 = abs f1;                           // abs    x(n)rms
        comp(f11,f2);                           // x(n)rms < Umbral ?
        if LT jump Salida_Igual;                // Si--> Fin: y(n) = x(n)
        Jump Salida_Compresor;                  // No-->Salida

Salida_Igual:
f4=f0;
jump Fin;

Salida_Compresor:
f5=f0-f0;
comp(f0,f5);
if GT jump Salida_Positiva;
jump Salida_Negativa;

Salida_Positiva:
f4=f0-f2;
f4=f4*f3;
f4=f4+f2;
jump Fin;

Salida_Negativa:
f4=f0+f2;
f4=f4*f3;
f4=f4-f2;
jump Fin;

//------------------------------------------------------------------------

Fin:
dm(i4,m4)=f4;                           // Almacena muestra de salida en un buffer
r1 = 31;
r10 = fix f4 by r1;
DM(Left_Channel_Out)=r10;
DM(Right_Channel_Out)=r10;
rts;

.endseg;
```

CÓDIGO ALGORITMO DE SIMULACIÓN EXPANSOR PICO

Archivo Fuente: *Expansor.asm*

```
/*****************************************************************************/
/**                        EXPANSOR PICO      version 1.0                   ***/
/**                        Simulación                                       ***/
/**                        Expansor.asm                                     ***/
/**                                                                         ***/
/*****************************************************************************/

/* ADSP-21065L Definicion Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

#define Umbral 0.015                    // Threshold  o Umbral donde empieza a comprimir
#define Ratio_Compresor  0.05           // Factor de Compresión
#define Ratio_Expansor 2.0              // Factor de Expansión
#define N       34731                   //  Muestras de Entrada //8000


.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;
        nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;
.var    Entrada[N]="Prueba2.dat";       //"Sen_200b.dat";//
.var    Salida[N];
.endseg;

.segment /pm pm_code;
_main:
                CALL setup_SDRAM (db);
                r0 = 0x3A2;         /* SDRDIV setting */
                r1 = 0x8E762800;    /* IOCTL setting  */
                ustat1=dm(SYSCON);          /* Ensure that IMDW0=1 so block0 accesses */
                bit set ustat1 IMDW0X; /* are 40 bits (3 columns) for compatability*/
                dm(SYSCON)=ustat1;

Inicializacion_Bufers:

B0=Entrada;             L0=0;                   M0=1;
                                                M4=1;

                CALL Efecto (DB);       /* Example delayed call instruction */
                B4=Salida;
                L4=0;
_exit:  IDLE;
```

```
/*-----------------------------------------------------------------------------------------------------*/
//Rutina para Delay simple  Procesamiento solo canal derecho

Efecto:
LCNTR=N, DO (PC,8) UNTIL LCE;//20
Inicio_1:
f0=dm(i0,m0);                   // Lee la muestra de entrada actual x(n)
f1=abs f0;                      // Toma el valor absoluto de la muestra x(n)
f2=Umbral;                      // Lee el valor del Umbral
f3=Ratio_Compresor;             // Lee el valor del Factor de atenuación
f6=Ratio_Expansor;              // Lee el valor del Factor de compresión
comp(f1,f2);                    // x(n) < Umbral ?
if LT jump Salida_Atenuada;     // Si-->Salida_Igual
Jump Salida_Expansor;           // No-->Salida_Comprimida

Salida_Atenuada:
 f4=f0*f3;                      // y(n)=x(n)*Ratio_Compresor
 dm(i4,m4)=f4;
 jump Inicio_1;

Salida_Expansor:
 f5=f1-f1;
 comp(f0,f5);                           // x(n) > 0 ?
 if GT jump Salida_Positiva;            // Si--> Salida_Positiva2
 jump Salida_Negativa;                  // No--> Salida_Negativa2

Salida_Positiva:
 f4=f0-f2;                              // x(n)-Umbral
 f4=f4*f6;                              // [x(n)-Umbral]* Ratio_Expansor
 f4=f4+f2;                              // y(n)= Umbral + [x(n)-Umbral ]* Ratio_Expansor
dm(i4,m4)=f4;
 jump Inicio_1;

Salida_Negativa:
 f4=f0+f2;                              // x(n)+Umbral
 f4=f4*f6;                              // [x(n)+Umbral]* Ratio_Expansor
 f4=f4-f2;                              // y(n)= -Umbral + (x(n)+Umbral)* Ratio_Expansor
 dm(i4,m4)=f4;
 jump Inicio_1;


RTS;

setup_SDRAM:
                        RTS (db);
                        dm(SDRDIV) = r0;
                        dm(IOCTL)  = r1;
.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL EXPANSOR PICO

Archivo Fuente: *Expansor_TR.asm*

```
/*****************************************************************************/
/**                    EXPANSOR    versión 2.0          EZ-KIT    ***/
/**                    Detección Pico                              ***/
/**                    Expansor_TR.asm                             ***/
/**                                                                ***/
/*****************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"


.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Buffers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;

.segment /dm    dmEfectos;
#define N                       24000
#define Umbral                  0.12        // Threshold  o Umbral donde empieza a comprimir
#define Ratio_Compresor         0.25        // Factor de Compresión
#define Ratio_Expansor          1.5         // Factor de Expansión

.var            Entrada[N];
.var            Salida[N];
.endseg;


/* -------------Codigo Memoria de Programa-----------------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
b0=Entrada;             l0=@Entrada;    m0=1;
b4=Salida;              l4=@Salida;             m4=1;

RTS;

/*-------------------------------------------------------------------------------------------------*/
//Rutina para Compresor RMS Canal Derecho

Efecto_Audio:
r15 = DM(Right_Channel_In);    //Obtiene entrada canal derecho
r1 = -31;
f0 = float r15 by r1;

dm(i0,m0)=f0;
Inicio__1:
```

```
f1=abs f0;                      // Toma el valor absoluto de la muestra x(n)
f2=Umbral;                              // Lee el valor del Umbral
f3=Ratio_Compresor;             // Lee el valor del Factor de atenuación
f6=Ratio_Expansor;              // Lee el valor del Factor de compresión
comp(f1,f2);                    // x(n) < Umbral ?
if LT jump Salida_Atenuada;     // Si-->Salida_Igual
Jump Salida_Expansor;           // No-->Salida_Comprimida

Salida_Atenuada:
f4=f0*f3;                                       // y(n) = x(n)*Ratio_Compresor
jump Fin_1;

Salida_Expansor:
f5=f1-f1;
comp(f0,f5);                            // x(n) > 0 ?
if GT jump Salida_Positiva;     // Si--> Salida_Positiva
jump Salida_Negativa;           // No--> Salida_Negativa

Salida_Positiva:
f4=f0-f2;                                       // x(n)-Umbral
f4=f4*f6;                                       // [x(n)-Umbral]* Ratio
f4=f4+f2;                                       // y(n)= Umbral + [x'(n)-Umbral ]* Ratio
jump Fin_1;

Salida_Negativa:
f4=f0+f2;                                       // x(n)+Umbral
f4=f4*f6;                                       // [x'(n)+Umbral]* Ratio
f4=f4-f2;                                       // y(n)= -Umbral + (x'(n)+Umbral)* Ratio
jump Fin_1;

//-----------------------------------------------------------------------

Fin_1:
dm(i4,m4)=f4;                   // Almacena muestra de salida en un buffer
r1 = 31;
r10 = fix f4 by r1;
DM(Left_Channel_Out)=r10;
DM(Right_Channel_Out)=r10;
rts;

.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL EXPANSOR RMS

Archivo Fuente: *Expansor_RMS_TR.asm*

```
/*****************************************************************************/
/**             EXPANSOR     version 2.0              EZ-KIT    ***/
/**             RMS                                            ***/
/**             Expansor_RMS_TR.asm                            ***/
/**                                                            ***/
/*****************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"


.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Buffers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;

.segment /dm    dmEfectos;
#define         N                       8000        // Muestras de entrada y salida
#define         M                       500         // Muestras para estimar RMS
#define         Umbral                  0.1         // Threshold o umbral
#define         Ratio_Compresor          0.05       // Factor de Compresión
#define         Ratio_Expansor          1.7         // Factor de Compresión
#define         Reciproco_M             0.002       // 1/M, ponderación

.var            Muestra_Entrada;                    // Muestra de entrada actual x(n)
.var            Muestra_Entrada_RMS;                // Estimación RMS de x(n)
.var            Suma_Pond_Cua = 0.0;                // SUM[x^2(k)*(1/M)], n-(M-1)<= k <= n
.var            Linea_MCP[M];                       // linea entrada`ponderada al cuadrado
.var            Entrada[N];                         // Buffer de entrada
.var            Salida[N];                          // Buffer de salida
.endseg;


/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
B0=Entrada;             L0=0;               M0=1;
B4=Salida;              L4=@Salida;         M4=1;
B2=Linea_MCP;           L2=@Linea_MCP;      M2=1;

f13=0.0;
LCNTR = M, DO  Borra_Linea_RMS UNTIL LCE;
Borra_Linea_RMS: dm(i2, m2) = f13;

RTS;
```

```
/*-------------------------------------------------------------------------------------------------------*/
//Rutina para Compresor RMS Canal Derecho
Efecto_Audio:
        r15 = DM(Right_Channel_In);             // Obtiene entrada canal derecho
        r1 = -31;
        f0 = float r15 by r1;
        dm(i0,m0)=f0;
        DM(Muestra_Entrada) = f0;

Estimacion_RMS:
        f5 = dm(Reciproco_M);                   // 1/M
        f2 = abs f0;                            // abs x(n)
        f1 = f2;                                // F1= abs x(n)
        f0 = f2 * f1;                           // F0 = x(n)*x(n)
        f0 = f0 * f5;                           // F0=   x(n)*x(n)*(1/M)
        f1 = dm(i2,0);                         /* Obtiene el valor viejo de la línea de entradas al
                                                  cuadrado ponderadas, sin actualizar el puntero
        f10 = DM(Suma_Pond_Cua);                // Obtiene el resultado de la última suma
        f10 = f10 + f0;                         // Suma la nueva entrada al cuadrado ponderada
        f10 = f10 - f1;                  // Resta la más vieja entrada al cuadrado ponderada
        DM(Suma_Pond_Cua) = f10;                // Guarda el resultado de la nueva suma
        dm(i2,m2) = f0;                         /* Guarda la entrada actual al cuadrado ponderada,
                                                   actualizando el puntero*/

Newton_Raphson:
        f8 = 3.0;
        f2 = 0.5;
        f4 = RSQRTS f10;                                // 1/ [(F10)^0.5]
        f1 = f4;
        f12 = f4 * f1;                                  // F12=X0^2
        f12 = f12 * f0;                                 // F12=C*X0^2
        f4 = f2 * f4, f12 = f8 - f12;                   // F4=.5*X0, F10=3-C*X0^2
        f4 = f4 * f12;                                  // F4=X1=.5*X0(3-C*X0^2)
        f1 = f4;
        f12 = f4 * f1;                                  // F12=X1^2
        f12 = f12 * f0;                                 // F12=C*X1^2
        f4 = f2 * f4, f12 = f8 - f12;                   // F4=.5*X1, F10=3-C*X1^2
        f4 = f4 * f12;                                  // F4=X2=.5*X1(3-C*X1^2)
        f1 = f4;
        f12 = f4 * f1;                                  // F12=X2^2
        f12 = f12 * f0;                                 // F12=C*X2^2
        f4 = f2 * f4, f12 = f8 - f12;                   // F4=.5*X2, F10=3-C*X2^2
        f4 = f4 * f12;                                  // F4=X3=.5*X2(3-C*X2^2)
        f10 = f4 * f10;                                 // X=sqrt(Y)=Y/sqrt(Y)
        DM(Muestra_Entrada_RMS) = f10;

Comparador:
        f2=Umbral;                                      // Lee el valor del Umbral
        f6=Ratio_Expansor;                              // Lee el valor del Factor de Expansión
        f3=Ratio_Compresor;                             // Lee el valor del Factor de Compresion
        f0 = DM(Muestra_Entrada);                       // Obtiene la muestra actual de entrada x(n)
        f1=DM(Muestra_Entrada_RMS);                     // F1= x(n)rms
        f11 = abs f1;                                   // abs    x(n)rms
```

```
        comp(f11,f2);                       // x(n)rms < Umbral ?
        if LT jump Salida_Atenuada;         // Si--> Fin: y(n) = Ratio_Compresor*x(n)
        Jump Salida_Expansor;               // No-->Salida

Salida_Atenuada:
f4=f0*f3;
jump Fin;
Salida_Expansor:
f5=f0-f0;
comp(f0,f5);
if GT jump Salida_Positiva;
jump Salida_Negativa;

Salida_Positiva:
f4=f0-f2;
f4=f4*f6;
f4=f4+f2;
jump Fin;

Salida_Negativa:
f4=f0+f2;
f4=f4*f6;
f4=f4-f2;
jump Fin;


//-------------------------------------------------------------------------------------------------------------

Fin:
dm(i4,m4)=f4;                       // Almacena muestra de salida en un buffer
r1 = 31;
r10 = fix f4 by r1;
DM(Left_Channel_Out)=r10;
DM(Right_Channel_Out)=r10;
rts;

.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL NOISE GATE RMS

Archivo Fuente: *NoiseGate_TR.asm*

```
/*****************************************************************************/
/**                    NOISE GATE   version 2.0          EZ-KIT    ***/
/**                    Detección RMS                                ***/
/**                    NoiseGate_TR.asm                             ***/
/***                                                                 ***/
/*****************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Buffers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;

.segment /dm    dmEfectos;
#define         N                   8000        // Muestras de entrada y salida
#define         M                   500         // Muestras para estimar RMS
#define         Umbral              0.1         // Threshold o umbral
#define         Ratio_Expansor      1.7         // Factor de Compresión
#define         Reciproco_M         0.002       // 1/M, ponderación

.var            Muestra_Entrada;                // Muestra de entrada actual x(n)
.var            Muestra_Entrada_RMS;            // Estimación RMS de x(n)
.var            Suma_Pond_Cua = 0.0;            // SUM[x^2(k)*(1/M)], n-(M-1)<= k <= n
.var            Linea_MCP[M];                   // linea entrada`ponderada al cuadrado
.var            Entrada[N];                     // Buffer de entrada
.var            Salida[N];                      // Buffer de salida
.endseg;


/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
B0=Entrada;             L0=0;               M0=1;
B4=Salida;              L4=@Salida;         M4=1;

B2=Linea_MCP;           L2=@Linea_MCP;      M2=1;

f13=0.0;
LCNTR = M, DO  Borra_Linea_RMS UNTIL LCE;
Borra_Linea_RMS: dm(i2, m2) = f13;

RTS;
```

```
/*-------------------------------------------------------------------------------------------------------------*/
//Rutina para Compresor RMS Canal Derecho
Efecto_Audio:

        r15 = DM(Right_Channel_In);         // Obtiene entrada canal derecho
        r1 = -31;
        f0 = float r15 by r1;
        dm(i0,m0)=f0;
        DM(Muestra_Entrada) = f0;

Estimacion_RMS:
        f5 = dm(Reciproco_M);               // 1/M
        f2 = abs f0;                        // abs x(n)
        f1 = f2;                            // F1= abs x(n)
        f0 = f2 * f1;                       // F0 = x(n)*x(n)
        f0 = f0 * f5;                       // F0=   x(n)*x(n)*(1/M)
        f1 = dm(i2,0);                     /* Obtiene el valor viejo de la línea de entradas al
                                              cuadrado ponderadas, sin actualizar el puntero
        f10 = DM(Suma_Pond_Cua);            // Obtiene el resultado de la última suma
        f10 = f10 + f0;                     // Suma la nueva entrada al cuadrado ponderada
        f10 = f10 - f1;                    // Resta la más vieja entrada al cuadrado
ponderada
        DM(Suma_Pond_Cua) = f10;            // Guarda el resultado de la nueva suma
        dm(i2,m2) = f0;                     /* Guarda la entrada actual al cuadrado ponderada,
                                               actualizando el puntero*/

Newton_Raphson:
        f8 = 3.0;
        f2 = 0.5;
        f4 = RSQRTS f10;                            // 1/ [(F10)^0.5]
        f1 = f4;
        f12 = f4 * f1;                              // F12=X0^2
        f12 = f12 * f0;                             // F12=C*X0^2
        f4 = f2 * f4, f12 = f8 - f12;               // F4=.5*X0, F10=3-C*X0^2
        f4 = f4 * f12;                              // F4=X1=.5*X0(3-C*X0^2)
        f1 = f4;
        f12 = f4 * f1;                              // F12=X1^2
        f12 = f12 * f0;                             // F12=C*X1^2
        f4 = f2 * f4, f12 = f8 - f12;               // F4=.5*X1, F10=3-C*X1^2
        f4 = f4 * f12;                              // F4=X2=.5*X1(3-C*X1^2)
        f1 = f4;
        f12 = f4 * f1;                              // F12=X2^2
        f12 = f12 * f0;                             // F12=C*X2^2
        f4 = f2 * f4, f12 = f8 - f12;               // F4=.5*X2, F10=3-C*X2^2
        f4 = f4 * f12;                              // F4=X3=.5*X2(3-C*X2^2)
        f10 = f4 * f10;                             // X=sqrt(Y)=Y/sqrt(Y)
        DM(Muestra_Entrada_RMS) = f10;

Comparador:
        f2=Umbral;                          // Lee el valor del Umbral
        f6=Ratio_Expansor;                  // Lee el valor del Factor de Expansión
        f0 = DM(Muestra_Entrada);           // Obtiene la muestra actual de entrada x(n)
        f1=DM(Muestra_Entrada_RMS);         // F1= x(n)rms
```

```
        f11 = abs f1;                   // abs    x(n)rms
        comp(f11,f2);                   // x(n)rms < Umbral ?
        if LT jump Salida_Cero;         // Si--> Fin: y(n) =0
        Jump Salida_NoiseGate;          // No-->Salida

Salida_Cero:
f4=f0-f0;
jump Fin;

Salida_NoiseGate:
f5=f0-f0;
comp(f0,f5);
if GT jump Salida_Positiva;
jump Salida_Negativa;

Salida_Positiva:
f4=f0-f2;
f4=f4*f6;
f4=f4+f2;
jump Fin;

Salida_Negativa:
f4=f0+f2;
f4=f4*f6;
f4=f4-f2;
jump Fin;


//-------------------------------------------------------------------------------------------

Fin:
dm(i4,m4)=f4;                       // Almacena muestra de salida en un buffer
r1 = 31;
r10 = fix f4 by r1;
DM(Left_Channel_Out)=r10;
DM(Right_Channel_Out)=r10;
rts;

.endseg;
```

CÓDIGO ALGORITMO DE SIMULACIÓN FILTRO FIR

Archivo Fuente: *Filtro_FIR.asm*

```
/*****************************************************************************/
/**                    FILTRO FIR    version 1.0    Simulación    ***/
/***                   Filtro_FIR.asm                             ***/
/***                                                              ***/
/*****************************************************************************/

/* ADSP-21065L Definicion Bits Registros del Sistema */
#include        "def21065l.h"
#include        "new65Ldefs.h"

#define N               34731           //  Muestras de Entrada
#define TAPS            101             //  Numero de taps del filtro FIR


.SEGMENT/PM     seg_rth;
rth:
        nop;nop;nop;nop;nop;
        jump _main;
        nop;nop;
.ENDSEG;

.segment /dm    dm_data;
.var    Entrada[N]="Prueba.dat";        //"RuidoBlanco8kHz.dat";        //"Sen400_5arm.dat";
                                        // "Impulso109.dat";    // "Impulso101.dat";//
.var    Salida[N];
.var    LineaRetardo[TAPS];
.endseg;

.SEGMENT/PM seg_pmda;
.VAR    coef_FIR[TAPS]="CoeFIR_Pasabanda.dat";    //  "CoeFIR_Pasaaltos.dat";
                                                  // "CoeFIR_Bandaeliminada.dat";
                                                  //  "CoeFIR_Pasabajos.dat";//
.ENDSEG;

.segment /pm pm_code;
_main:
                CALL setup_SDRAM (db);
                r0 = 0x3A2;             /* SDRDIV setting */
                r1 = 0x8E762800;        /* IOCTL setting  */
                ustat1=dm(SYSCON);      /* IMDW0=1 acceso block0  */
                bit set ustat1 IMDW0X; /* son 40 bits (3 columnas) para compatibilidad*/
                dm(SYSCON)=ustat1;

Inicializacion_Buffers:

B0=Entrada;             L0=@Entrada;            M0=1;
```

```
B1=LineaRetardo;        L1=@LineaRetardo;      M1=1;
                                               M4=1;
B8=coef_FIR;   L8=@coef_FIR;M8=1;
        f5=0.0;
        lcntr=TAPS, do Borra_LineaRetardo until lce;
        Borra_LineaRetardo: dm(i1,1)=f5;
                CALL Fir (DB);
                B4=Salida;
                L4=0;
_exit:  IDLE;

/*-------------------------------------------------------------------------------------------*/
/*****                    Rutina Filtro FIR                                    ****/

Fir:
LCNTR=N, DO (PC,8) UNTIL LCE;

        f0 =dm(i0,m0);          //      Lee muestra de entrada x(n)

        r12=r12 xor r12,        dm(i1,m1)=f0;
        f8=pass f12,            f0=dm(i1,m1), f4=pm(i8,m8);

        lcntr=TAPS-1, do MultiplicaSuma until lce;
        MultiplicaSuma:   f12=f0*f4, f8=f8+f12, f0=dm(i1,m1), f4=pm(i8,m8);

        f12=f0*f4, f8=f8+f12;
        f8=f8+f12;

        dm(i4,m4) = f8;         // Guarda muestra de salida y(n) en buffer

RTS;

setup_SDRAM:
                        RTS (db);
                        dm(SDRDIV) = r0;
                        dm(IOCTL)  = r1;
.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL FILTRO FIR

Archivo Fuente: *FiltroFIR_TR.asm*

```
/*****************************************************************************************************/
/**             Filtro FIR              version 2.0                                                 **/
/**             Para uso con la Tarjeta de Evaluación                                               **/
/**             Usa la interrupción Flag0 para y(n)=x(n)                                            **/
/**             Fs= 8kHz                                                                            **/
/**                                                                                                 **/
/**             FIltroFIR_TR.asm                                                                    **/
/**                                                                                                ***/
/*****************************************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL         Efecto_Audio;
.GLOBAL             Inicializacion_Bufers;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;
.EXTERN             RX_left_flag;

.segment /dm    dmEfectos;
#define N           24000           //  Muestras de Entrada y Salida
#define TAPS        101             //      Numero de taps del filtro FIR

.var    LineaRetardo[TAPS];
.var    Entrada[N];
.var    Salida[N];
.endseg;


.SEGMENT/PM pm_data;
.VAR    coef_FIR[TAPS]="CoeFIR_Pasabanda.dat";      // "CoeFIR_Pasaaltos.dat";
                                                    //  "CoeFIR_Bandaeliminada.dat";
                                                    //"CoeFIR_Pasabajos.dat";//
.ENDSEG;



/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Bufers:
        B0=Entrada;             L0=0;                   M0=1;
        B1=LineaRetardo;        L1=@LineaRetardo;       M1=1;
        B4=Salida;              L4=@Salida;             M4=1;
        B8=coef_FIR;            L8=@coef_FIR;M8=1;
```

```
        f5=0.0;
        lcntr=TAPS, do Borra_LineaRetardo until lce;
        Borra_LineaRetardo: dm(i1,1)=f5;


RTS;

/*-------------------------------------------------------------------------------------------------------------------*/
//Rutina para el filtro FIR Procesamiento canal derecho
Efecto_Audio:

//--------------------------//
Entrada_filtroFIR:
r15 = DM(Right_Channel_In);             //Obtiene entrada canal derecho
r1 = -31;
f0 = float r15 by r1;

dm(i0,m0)=f0;           //      Lee muestra de entrada x(n)

f8 = f0;
if FLAG0_IN jump Salida_filtroFIR;

Fir:    r12=r12 xor r12,        dm(i1,m1)=f0;
        f8=pass f12,            f0=dm(i1,m1), f4=pm(i8,m8);

        lcntr=TAPS-1, do MultiplicaSuma until lce;
        MultiplicaSuma:   f12=f0*f4, f8=f8+f12, f0=dm(i1,m1), f4=pm(i8,m8);

        f12=f0*f4, f8=f8+f12;
        f8=f8+f12;

        dm(i4,m4) = f8;         // Guarda muestra de salida y(n) en buffer

Salida_filtroFIR:
        r1 = 31;
        r10 = fix f8 by r1;
        DM(Left_Channel_Out)=r10;
        DM(Right_Channel_Out)=r10;

/*-------------------------------------------------------------------------------------------------------------------*/
                rts (db);
                r4 = 0;
                dm(RX_left_flag) = r4;          // RX_left_flag una vez procesado el dato

                rts;

.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL FILTRO FIR CON RUTINA IRQ1

Archivo Fuente: *FiltroFIR_TR_IRQ1.asm*

```
/*****************************************************************************************************************/
/**                     Filtro FIR              version 3.0                                                     **/
/**                     Para uso con la Tarjeta de Evaluación                                                   **/
/**                     Usa la interrupción Flag0 para y(n)=x(n)                                                **/
/**                     Usa interrupción IRQ1 para cambiar coeficientes                                         **/
/**                     pasa bajos, pasa altos, pasa banda o banda eliminada                                    **/
/**                     Fs= 8kHz                                                                                **/
/**                                                                                                             **/
/**                     Filtro_TR_IRQ1.asm                                                                      **/
/**                                                                                                             **/
/*****************************************************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL          Efecto_Audio;
.GLOBAL             Inicializacion_Buffers;
.GLOBAL             Cambio_coeficientes_filtroFIR;
.EXTERN      Left_Channel_In;
.EXTERN      Right_Channel_In;
.EXTERN      Left_Channel_Out;
.EXTERN      Right_Channel_Out;
.EXTERN             RX_left_flag;

.segment /dm    dmEfectos;

#define N               24000           //  Muestras de Entrada y Salida
#define TAPS_Bajos      109             //  Numero de taps del filtro FIR Pasa bajos
#define TAPS_Altos      101             //  Numero de taps del filtro FIR Pasa altos
#define TAPS_Banda      101             //  Numero de taps del filtro FIR Pasa bandas
#define TAPS_Stop       109             //  Numero de taps del filtro FIR Banda eliminada

.var    TAPS=109;

.var    LineaRetardo_Bajos[TAPS_Bajos];
.var    LineaRetardo_Altos[TAPS_Altos];
.var    LineaRetardo_Banda[TAPS_Banda];
.var    LineaRetardo_Stop[TAPS_Stop];

.VAR    Contador_filtroFIR=3;
.var    Entrada[N];
.var    Salida[N];
.endseg;


.SEGMENT/PM pm_data;
.VAR    coef_FIR_Bajos[TAPS_Bajos]="CoeFIR_Pasabajos.dat";
.VAR    coef_FIR_Altos[TAPS_Altos]="CoeFIR_Pasaaltos.dat";
```

```
.VAR    coef_FIR_Pasabanda[TAPS_Banda]="CoeFIR_Pasabanda.dat";
.VAR    coef_FIR_Bandaeliminada[TAPS_Stop]="CoeFIR_Bandaeliminada.dat";
.ENDSEG;


/* ----------------------------        Codigo Memoria de Programa   ----------------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
        B0=Entrada;                 L0=0;                                   M0=1;
        B1=LineaRetardo_Bajos;      L1=@LineaRetardo_Bajos;                 M1=1;
        B4=Salida;                  L4=@Salida;                             M4=1:
        B8=coef_FIR_Bajos;          L8=@coef_FIR_Bajos;                     M8=1;

        f5=0.0;
        lcntr=TAPS, do Borra_LineaRetardo until lce;
        Borra_LineaRetardo: dm(i1,1)=f5;


RTS;

/*-------------------------------------------------------------------------------------------------------------------------------------------------------*/
//Rutina para el filtro FIR Procesamiento canal derecho

Efecto_Audio:

Entrada_filtroFIR:
        r15 = DM(Right_Channel_In);         //Obtiene entrada canal derecho
        r1 = -31;
        f0 = float r15 by r1;

        dm(i0,m0)=f0;           //      Lee muestra de entrada x(n)

        f8 = f0;
        if FLAG0_IN jump Salida_filtroFIR;

Fir:    r2=dm(TAPS);
        r3=1;
        r5=r2-r3;

        r12=r12 xor r12,        dm(i1,m1)=f0;
        f8=pass f12,            f0=dm(i1,m1), f4=pm(i8,m8);

        lcntr=r5, do MultiplicaSuma until lce;
        MultiplicaSuma:   f12=f0*f4, f8=f8+f12, f0=dm(i1,m1), f4=pm(i8,m8);

        f12=f0*f4, f8=f8+f12;
        f8=f8+f12;

        dm(i4,m4) = f8;         // Guarda muestra de salida y(n) en buffer
```

```
Salida_filtroFIR:
        r1 = 31;
        r10 = fix f8 by r1;
        DM(Left_Channel_Out)=r10;
        DM(Right_Channel_Out)=r10;

/*-------------------------------------------------------------------------------------------------------------------*/
        rts (db);
        r4 = 0;
        dm(RX_left_flag) = r4;          // RX_left_flag una vez proesado el dato

        rts;

/* ------------------------------------------------------------------------------------------------ */
/*  Rutina de Interrupción IRQ1 para cambiar los coeficientes del filtro FIR          */
/*  Cambia entre un filtro pasaltos, pasabajo y pasabanda.                            */
/* ------------------------------------------------------------------------------------------------ */

Cambio_coeficientes_filtroFIR:
   bit set mode1 SRRFH;             // Habilta Registros de datos segundarios R8 a R15
        NOP;                        // 1 ciclo de latencia paraescribir en el registro MODE1

        r11 = 4;                        // Tipos de Filtros
        r10 = DM(Contador_filtroFIR);   // Lee el último valor del contador desde memoria
        r10 = r10 + 1;                  // Incrementa el contador
        comp (r10, r11);            // compara el valor actual  del contador con el valor máximo
        if ge r10 = r10 - r10;// Si el valor del contador es igual al máximo reseta y vuelve a empezar
        DM(Contador_filtroFIR) = r10;   // Salva el valor actual del contador

        r15 = pass r10;
        if eq jump FiltroFIR_Pasaaltos;                 // Chequea el contador = 0,
                                                        //entonces está en Pasa altos
        r15 = r15 - 1;
        if eq jump FiltroFIR_Pasabandas;                // Chequea el contador = 1,
                                                        // entonces está en Pasa banda
        r15 = r15 - 1;
        if eq jump FiltroFIR_Bandaeliminada;            // Chequea el contador = 2,
                                                        //entonces está en Banda
eliminada

FiltroFIR_Pasabajos:                            // Si esta acá el contador = 3
        r8=TAPS_Bajos;
        dm(TAPS)=r8;
        b1=LineaRetardo_Bajos;                  // Inicializa el buffer circular para la línea de
                                                //retardo del filtro pasabajos
        l1=@LineaRetardo_Bajos;
        b8 = coef_FIR_Bajos;    l8 = @coef_FIR_Bajos;
        bit set ustat1 0x3E;       // Enciende el LED Flag4
        bit clr ustat1 0x01;
        dm(IOSTAT)=ustat1;
        jump Final_Rutina_IRQ1;

FiltroFIR_Pasaaltos:
        r8=TAPS_Altos; dm(TAPS)=r8;
```

```
        b1=LineaRetardo_Altos;   l1=@LineaRetardo_Altos;
        b8 = coef_FIR_Altos;     l8 = @coef_FIR_Altos;
        bit set ustat1 0x3D;       // Enciende el LED Flag5
        bit clr ustat1 0x02;
        dm(IOSTAT)=ustat1;
        jump Final_Rutina_IRQ1;


FiltroFIR_Pasabandas:
        r8=TAPS_Banda; dm(TAPS)=r8;
        b1=LineaRetardo_Banda;          l1=@LineaRetardo_Banda;
        b8 = coef_FIR_Pasabanda;  l8 = @coef_FIR_Pasabanda;
        bit set ustat1 0x3B;         // Enciende el LED Flag6
        bit clr ustat1 0x04;
        dm(IOSTAT)=ustat1;
        jump Final_Rutina_IRQ1;

FiltroFIR_Bandaeliminada:
        r8=TAPS_Stop; dm(TAPS)=r8;
        b1=LineaRetardo_Stop;                   l1=@LineaRetardo_Stop;
        b8 = coef_FIR_Bandaeliminada; l8 = @coef_FIR_Bandaeliminada;
        bit set ustat1 0x37;                  // Enciende el LED Flag7
        bit clr ustat1 0x08;
        dm(IOSTAT)=ustat1;
         jump Final_Rutina_IRQ1;


Final_Rutina_IRQ1:
        rti(db);
        bit clr mode1 SRRFH;   // Deshabilita los registros de datos segundarios R8 a R15
        nop;

.endseg;
```

CÓDIGO ALGORITMO DE SIMULACIÓN FILTRO IIR

Archivo Fuente: *Filtro_IIR.asm*

```
/*****************************************************************************************************************/
/**                                  FILTRO IIR      version 1.0       Simulación                        ***/
/**                                                                                                      ***/
/*        FILTRO IIR PASABANDA BASADO EN UN FILTRO BUTTER DE 2º ORDEN                                    ***/
/**       CON Fs 48kHz  Y FRECUENCIAS DE CORTE PASABANDA Fs*0.3 Y Fs*.4                                  ***/
/***                                                                                                    ***/
/*****************************************************************************************************************/

/* ADSP-21065L Definicion Bits Registros del Sistema */
#include          "def21065l.h"
#include          "new65Ldefs.h"

#define           TAPS            11              // 11     9        8         7
#define           N               8000            //300     8000


/*---------------------------------------------------------------------------------------------------------------*/
.SEGMENT/PM     seg_rth;
rth:
          nop;nop;nop;nop;
          nop;
          jump _main;
          nop;nop;
.ENDSEG;



/*---------------------------------------------------------------------------------------------------------------*/


.SEGMENT/DM seg_dmda;
.VAR    LineaRetardo_Entrada[TAPS];
.VAR    LineaRetardo_Salida[TAPS];
.VAR
Entrada[N]="Sen400_5arm.dat";//"Prueba.dat";//"RuidoBlanco8kHz.dat";//"Impulso_IIR.dat";//
.VAR    Salida[N];
.endseg;


/*---------------------------------------------------------------------------------------------------------------*/

.SEGMENT/PM seg_pmda;
.VAR
Coeficientes_a[TAPS]="Coef_a_Pasabajos1.dat";//"Coef_a_Pasabanda.dat";//"Coef_a_Bandaelimi
nada.dat";//"Coef_a_Pasaaltos.dat";//
.VAR
          Coeficientes_b[TAPS]="Coef_b_Pasabajos1.dat";//"Coef_b_Pasabanda.dat";//"Coef_b_Ban
daeliminada.dat";//"Coef_b_Pasaaltos.dat";//
```

```
.endseg;
/*-----------------------------------------------------------------------------------------------------------------------*/

.SEGMENT/PM seg_pmco;

_main:
                    CALL setup_SDRAM (db);
                    r0 = 0x3A2;           /* SDRDIV setting */
                    r1 = 0x8E762800;      /* IOCTL setting  */
                    ustat1=dm(SYSCON);    /* IMDW0=1 acceso block0  */
                    bit set ustat1 IMDW0X; /* son 40 bits (3 columnas) para compatibilidad*/
                    dm(SYSCON)=ustat1;

Inicializacion_Buffers:

          b0=Entrada;                 l0=0;                          m0=1;
                                                                     m4=1;

          b3=LineaRetardo_Salida;     l3=@LineaRetardo_Salida;       m3=1;
          b2=LineaRetardo_Entrada;    l2=@LineaRetardo_Entrada;      m2=1;

          b8 = Coeficientes_a;        l8 =@Coeficientes_a;           m8=1;
          b9 = Coeficientes_b;        l9 =@Coeficientes_b;           m9=1;

          f2=0.0;

          lcntr=TAPS, do Borra_LineaRetardo_Salida until lce;
          Borra_LineaRetardo_Salida:    dm(i2,m2)=f2;

          lcntr=TAPS, do Borra_LineaRetardo_Entrada until lce;
          Borra_LineaRetardo_Entrada:    dm(i3,m3)=f2;

          CALL IIR (DB);
          B4=Salida;
          L4=0;

_exit:  IDLE;


IIR:
lcntr=N, do (pc,21) until lce;


f0 =dm(i0,m0);  // lee x(n)
dm(i2,0)=f0;
f3=f3-f3, dm(i3,0)=f3;

f12=0.0;
f8=pass f12,      f0=dm(i3,m3), f4=pm(i8,m8);
lcntr=TAPS-1, do Lazo_Salida_Desplazada until lce;          /*  a1*y(n-1)+a2*y(n-2)+...+
                                                                aM*y(n-M), M=TAPS-1    */

Lazo_Salida_Desplazada:   f12=f0*f4, f8=f8+f12, f0=dm(i3,m3), f4=pm(i8,m8);
f12=f0*f4, f8=f8+f12;
```

```
f8=f8+f12;
f10=f8;             //ak* y(n-k)

f12=0.0;
f8=pass f12,      f0=dm(i2,m2), f4=pm(i9,m9);
lcntr=TAPS-1, do Lazo_Entrada_Desplazada until lce;
Lazo_Entrada_Desplazada:   f12=f0*f4, f8=f8+f12, f0=dm(i2,m2), f4=pm(i9,m9);
f12=f0*f4,f8=f8+f12;
f8=f8+f12;

f9=f8;                          // bk *x(n-k)

f11=f9-f10;       // y(n)= bk *x(n-k)-ak* y(n-k)

dm(i3,-1)=f11;
modify(i2,-1);

dm(i4,m4)=f11; // salida

RTS;


setup_SDRAM:
                    RTS (db);
                    dm(SDRDIV) = r0;
                     dm(IOCTL)  = r1;
.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL FILTRO IIR

Archivo Fuente: *FiltroIIR_TR.asm*

```
/*****************************************************************************************************************/
/**                     Filtro IIR              version 2.0                                                     **/
/**                     Para uso con laTarjeta de Evaluación                                                    **/
/**                     Usa la interrupción Flag0 para y(n)=x(n)                                                **/
/**                     Fs= 8kHz                                                                                **/
/**                                                                                                             **/
/**                     FiltroIIR_TR1.asm                                                                       **/
/**                                                                                                             **/
/*****************************************************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"


.GLOBAL         Inicializacion_Buffers;
.GLOBAL         Efecto_Audio;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;
.EXTERN         RX_left_flag;


.segment /dm    dmEfectos;
#define N               24000           //  Muestras de Entrada y Salida
#define TAPS            11              //  Numero de taps del filtro IIR

.VAR    LineaRetardo_Entrada[TAPS];
.VAR    LineaRetardo_Salida[TAPS];
.var    Entrada[N];
.VAR    Salida[N];
.endseg;


.SEGMENT/PM pm_data;
.VAR    Coeficientes_a[TAPS]="Coef_a_Bandaeliminada.dat";        //"Coef_a_Pasabanda.dat";
                                                                 //"Coef_a_Pasaaltos.dat";
                                                                 //"Coef_a_Pasabajos.dat";//

.VAR Coeficientes_b[TAPS]="Coef_b_Bandaeliminada.dat";           //"Coef_b_Pasabanda.dat";
                                                                 //"Coef_b_Pasaaltos.dat";
                                                                 //"Coef_b_Pasabajos.dat";//
.ENDSEG;
```

```
/* ------------          Codigo Memoria de Programa                    ---------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
B0=Entrada;                         L0=Entrada;                         M0=1;
B4=Salida;                          L4=@Salida;                         M4=1;

B2=LineaRetardo_Salida;             L2=@LineaRetardo_Salida;            M2=1;
B3=LineaRetardo_Entrada;            L3=@LineaRetardo_Entrada;           M3=1;

B8 = Coeficientes_a;                L8 =@Coeficientes_a;                M8=1;
B9 = Coeficientes_b;                L9 =@Coeficientes_b;                M9=1;

f2=0.0;

lcntr=TAPS, do Borra_LineaRetardo_Salida until lce;
Borra_LineaRetardo_Salida:    dm(i2,m2)=f2;

lcntr=TAPS, do Borra_LineaRetardo_Entrada until lce;
Borra_LineaRetardo_Entrada:    dm(i3,m3)=f2;


RTS;

/*-------------------------------------------------------------------------------------------------------------------------*/

//Rutina para el filtro IIR Procesamiento canal derecho

Efecto_Audio:


Entrada_filtroIIR:
        r15 = DM(Right_Channel_In);             //Obtiene entrada canal derecho
        r1 = -31;
        f0 = float r15 by r1;

        dm(i0,m0)=f0;           //      Lee muestra de entrada x(n)

        f11 = f0;
        if FLAG0_IN jump Salida_filtroIIR;

        dm(i3,0)=f0;
        f3=f3-f3, dm(i2,0)=f3;

        f12=0.0;
        f8=pass f12,    f0=dm(i2,m2), f4=pm(i8,m8);
        lcntr=TAPS-1, do Lazo_Salida_Desplazada until lce;      // a1*y(n-1)+a2*y(n-2)+...+
                                                                // aM*y(n-M), M=TAPS-1
        Lazo_Salida_Desplazada:   f12=f0*f4, f8=f8+f12, f0=dm(i2,m2), f4=pm(i8,m8);
        f12=f0*f4, f8=f8+f12;
        f8=f8+f12;
        f10=f8;         //ak* y(n-k)
```

```
        f12=0.0;
        f8=pass f12,      f0=dm(i3,m3), f4=pm(i9,m9);
        lcntr=TAPS-1, do Lazo_Entrada_Desplazada until lce;
        Lazo_Entrada_Desplazada:   f12=f0*f4, f8=f8+f12, f0=dm(i3,m3), f4=pm(i9,m9);
        f12=f0*f4,f8=f8+f12;
        f8=f8+f12;
        f9=f8;                          // bk *x(n-k)

        f11=f9-f10;        // y(n)= bk *x(n-k)-ak* y(n-k)
        dm(i2,-1)=f11;
        modify(i3,-1);

        dm(i4,m4) = f11;                    // Guarda muestra de salida y(n) en buffer

Salida_filtroIIR:
        r1 = 31;
        r10 = fix f11 by r1;
        DM(Left_Channel_Out)=r10;
        DM(Right_Channel_Out)=r10;

/*-------------------------------------------------------------------------------------------------------------------------------------------*/
        rts (db);
        r4 = 0;
        dm(RX_left_flag) = r4;              // RX_left_flag en cero una vez proesado el dato

        rts;

.endseg;
```

CÓDIGO ALGORITMO EN TIEMPO REAL FILTRO IIR CON RUTINA IRQ1

Archivo Fuente: *FiltroIIR_TR_IRQ1.asm*

```
/*****************************************************************************************************************/
/**                     Filtro IIR              version 3.0                                                     **/
/**                     Para uso con laTarjeta de Evaluación                                                    **/
/**                     Usa la interrupción Flag0 para y(n)=x(n)                                                **/
/**                     Usa interrpción IRQ1 para cambiar coeficientes                                          **/
/**                     pasa bajos, pasa altos, pasa banda o banda eliminada                                    **/
/**                     Fs= 8kHz                                                                                **/
/**                                                                                                             **/
/**                     FiltroIIR_TR_IRQ1.asm                                                                   **/
/**                                                                                                             **/
/*****************************************************************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL         Efecto_Audio;
.GLOBAL         Inicializacion_Buffers;
.GLOBAL         Cambio_coeficientes_filtroIIR;
.EXTERN         Left_Channel_In;
.EXTERN         Right_Channel_In;
.EXTERN         Left_Channel_Out;
.EXTERN         Right_Channel_Out;
.EXTERN         RX_left_flag;

.segment /dm    dmEfectos;
#define         N               24000           // Muestras de Entrada y Salida
#define         TAPS_Bajos      7               //  Numero de taps del filtro IIR Pasa bajos
#define         TAPS_Altos      7               // Numero de taps del filtro IIR Pasa altos
#define         TAPS_Banda      9               // Numero de taps del filtro IIR Pasa bandas
#define         TAPS_Stop       9               // Numero de taps del filtro IIR Banda eliminada

.var    TAPS=7;

.var    LineaRetardo_Entrada_Bajos[TAPS_Bajos];
.var    LineaRetardo_Entrada_Altos[TAPS_Altos];
.var    LineaRetardo_Entrada_Banda[TAPS_Banda];
.var    LineaRetardo_Entrada_Stop[TAPS_Stop];

.var    LineaRetardo_Salida_Bajos[TAPS_Bajos];
.var    LineaRetardo_Salida_Altos[TAPS_Altos];
.var    LineaRetardo_Salida_Banda[TAPS_Banda];
.var    LineaRetardo_Salida_Stop[TAPS_Stop];

.VAR    Contador_filtroIIR=3;

.endseg;
```

```
.SEGMENT/PM pm_data;
.VAR    coef_a_Bajos[TAPS_Bajos]="Coef_a_Pasabajos.dat";
.VAR    coef_a_Altos[TAPS_Altos]="Coef_a_Pasaaltos.dat";
.VAR    coef_a_Pasabanda[TAPS_Banda]="Coef_a_Pasabanda.dat";
.VAR    coef_a_Bandaeliminada[TAPS_Stop]="Coef_a_Bandaeliminada.dat";
.VAR    coef_b_Bajos[TAPS_Bajos]="Coef_b_Pasabajos.dat";
.VAR    coef_b_Altos[TAPS_Altos]="Coef_b_Pasaaltos.dat";
.VAR    coef_b_Pasabanda[TAPS_Banda]="Coef_b_Pasabanda.dat";
.VAR    coef_b_Bandaeliminada[TAPS_Stop]="Coef_b_Bandaeliminada.dat";

.ENDSEG;



/* -------------Codigo Memoria de Programa---------------------------------*/
.segment /pm pm_code;

Inicializacion_Buffers:
        B0=Entrada;             L0=@Entrada;                                    M0=1;
        B4=Salida;              L4=@Salida;                                     M4=1;


        B2=LineaRetardo_Salida_Bajos;    L2=@LineaRetardo_Salida_Bajos;         M2=1;
        B3=LineaRetardo_Entrada_Bajos;   L3=@LineaRetardo_Entrada_Bajos;        M3=1;

        B8 = coef_a_Bajos;               L8 =@coef_a_Bajos;                     M8=1;
        B9 = coef_b_Bajos;               L9 =@coef_b_Bajos;                     M9=1;

        f2=0.0;

        lcntr=TAPS, do Borra_LineaRetardo_Salida until lce;
        Borra_LineaRetardo_Salida:    dm(i2,m2)=f2;

        lcntr=TAPS, do Borra_LineaRetardo_Entrada until lce;
        Borra_LineaRetardo_Entrada:    dm(i3,m3)=f2;

RTS;

/*-----------------------------------------------------------------------*/
//Rutina para el filtro IIR Procesamiento canal derecho
Efecto_Audio:

//-------------------------------------------------------------------------------------------------------------------------------------------//
Entrada_filtroIIR:
        r15 = DM(Right_Channel_In);             //Obtiene entrada canal derecho
        r1 = -31;
        f0 = float r15 by r1;

        dm(i0,m0)=f0;           //      Lee muestra de entrada x(n)

        f11 = f0;
        if FLAG0_IN jump Salida_filtroIIR;

        r2=dm(TAPS);
```

```
        r6=1;
        r5=r2-r6;

        dm(i3,0)=f0;
        f3=f3-f3, dm(i2,0)=f3;

        f12=0.0;
        f8=pass f12,    f0=dm(i2,m2), f4=pm(i8,m8);
        lcntr=r5, do Lazo_Salida_Desplazada until lce;
        Lazo_Salida_Desplazada:   f12=f0*f4, f8=f8+f12, f0=dm(i2,m2), f4=pm(i8,m8);
        f12=f0*f4, f8=f8+f12;
        f8=f8+f12;
        f10=f8;          //ak* y(n-k)

        f12=0.0;
        f8=pass f12,    f0=dm(i3,m3), f4=pm(i9,m9);
        lcntr=r5, do Lazo_Entrada_Desplazada until lce;
        Lazo_Entrada_Desplazada:   f12=f0*f4, f8=f8+f12, f0=dm(i3,m3), f4=pm(i9,m9);
        f12=f0*f4,f8=f8+f12;
        f8=f8+f12;
        f9=f8;                  // bk *x(n-k)

        f11=f9-f10;             // y(n)= bk *x(n-k)-ak* y(n-k)
        dm(i2,-1)=f11;
        modify(i3,-1);

        dm(i4,m4) = f11;                // Guarda muestra de salida y(n) en buffer

Salida_filtroIIR:
        r1 = 31;
        r10 = fix f11 by r1;
        DM(Left_Channel_Out)=r10;
        DM(Right_Channel_Out)=r10;
/*---------------------------------------------------------------------*/
        rts (db);
        r4 = 0;
        dm(RX_left_flag) = r4;          // RX_left_flag en cero una vez proesado el dato

        rts;
/* ------------------------------------------------------------------------------ ----------------------------*/
/*  Rutina de Interrupción IRQ1 para cambiar los coeficientes del filtro IIR                                    */
/*  Cambia entre un filtro pasaltos, pasabajo y pasabanda.                                                      */
/* ------------------------------------------------------------------------------------------------------------*/

Cambio_coeficientes_filtroIIR:
   bit set mode1 SRRFH;             // Habilta Registros de datos segundarios R8 a R15
        NOP;                        // 1 ciclo de latencia paraescribir en el registro MODE1

        r11 = 4;                            // Tipos de Filtros
        r10 = DM(Contador_filtroIIR);       // Lee el último valor del contador desde memoria
        r10 = r10 + 1;                      // Incrementa el contador
        comp (r10, r11);            // compara el valor actual  del contador con el valor máximo
        if ge r10 = r10 - r10;// Si el valor del contador es igual al máximo reseta y vuelve a empezar
        DM(Contador_filtroIIR) = r10;       // Salva el valor actual del contador
```

```
        r15 = pass r10;
        if eq jump FiltroIIR_Pasaaltos;          // Chequea el contador = 0,
                                                 //entonces está en Pasa altos
        r15 = r15 - 1;
        if eq jump FiltroIIR_Pasabandas;         // Chequea el contador = 1,
                                                 //  entonces está en Pasa banda
        r15 = r15 - 1;
        if eq jump FiltroIIR_Bandaeliminada;     // Chequea el contador = 2,
                                                 //  entonces está en Banda eliminada


FiltroIIR_Pasabajos:                                    // Si esta acá el contador = 3
        r8=TAPS_Bajos;          dm(TAPS)=r8;
        B2=LineaRetardo_Salida_Bajos;   L2=@LineaRetardo_Salida_Bajos;
        B3=LineaRetardo_Entrada_Bajos;  L3=@LineaRetardo_Entrada_Bajos;

        B8 = coef_a_Bajos;              L8 =@coef_a_Bajos;
        B9 = coef_b_Bajos;              L9 =@coef_b_Bajos;
        bit set ustat1 0x3E;                    // Enciende el LED Flag4
        bit clr ustat1 0x01;
        dm(IOSTAT)=ustat1;
        jump Final_Rutina_IRQ1;

FiltroIIR_Pasaaltos:
        r8=TAPS_Altos; dm(TAPS)=r8;
        B2=LineaRetardo_Salida_Altos;   L2=@LineaRetardo_Salida_Altos;
        B3=LineaRetardo_Entrada_Altos;  L3=@LineaRetardo_Entrada_Altos;

        B8 = coef_a_Altos;              L8 =@coef_a_Altos;
        B9 = coef_b_Altos;              L9 =@coef_b_Altos;
        bit set ustat1 0x3D;                    // Enciende el LED Flag5
        bit clr ustat1 0x02;
        dm(IOSTAT)=ustat1;
        jump Final_Rutina_IRQ1;


FiltroIIR_Pasabandas:
        r8=TAPS_Banda; dm(TAPS)=r8;
        B2=LineaRetardo_Salida_Banda;   L2=@LineaRetardo_Salida_Banda;
        B3=LineaRetardo_Entrada_Banda;  L3=@LineaRetardo_Entrada_Banda;

        B8 = coef_a_Pasabanda;          L8 =@coef_a_Pasabanda;

        B9 = coef_b_Pasabanda;          L9 =@coef_b_Pasabanda;
        bit set ustat1 0x3B;                    // Enciende el LED Flag6
        bit clr ustat1 0x04;
        dm(IOSTAT)=ustat1;
        jump Final_Rutina_IRQ1;
```

```
FiltroIIR_Bandaeliminada:
        r8=TAPS_Stop; dm(TAPS)=r8;
        B2=LineaRetardo_Salida_Stop;   L2=@LineaRetardo_Salida_Stop;
        B3=LineaRetardo_Entrada_Stop;  L3=@LineaRetardo_Entrada_Stop;
        B8 = coef_a_Bandaeliminada;     L8 =@coef_a_Bandaeliminada;
        B9 = coef_b_Bandaeliminada;     L9 =@coef_b_Bandaeliminada;
        bit set ustat1 0x37;                    // Enciende el LED Flag7
        bit clr ustat1 0x08;
        dm(IOSTAT)=ustat1;
         jump Final_Rutina_IRQ1;


Final_Rutina_IRQ1:
        rti(db);
        bit clr mode1 SRRFH;            // Deshabilita los registros de datos segundarios R8 a R15
        nop;

.endseg;
```

ARCHIVOS COMUNES PARA EJECUTAR LOS EFECTOS DE AUDIO EN LA TARJETA DE EVALUACIÓN EZ KIT LITE 21065L

Rutina Inicialización Tarjeta

Archivo: *Inicializacion_Tarjeta.asm*

```
/*****************************************************************************/
/*                      INICIALIZACION DE LA TARGETA                         */
/*****************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL                 _main;
.GLOBAL                 Init_DSP;
.EXTERN                 Init_65L_SDRAM_Controller;
.EXTERN         Blink_LEDs_Test;
.EXTERN                 Program_SPORT1_Registers;
.EXTERN                 Program_DMA_Controller;
.EXTERN                 AD1819_Codec_Initialization;
.EXTERN                 Inicializacion_Bufers;

/*---------------------------------------------------------------------------*/
.SEGMENT/dm dm_data;


.endseg;

/*---------------------------------------------------------------------------*/

.segment /pm pm_code;

_main:
        call Init_65L_SDRAM_Controller;         /* Inicializa Memoria Externa */
        call Program_SPORT1_Registers;          /* Inicializa Puerto SPOT1 para
                                                   comunicación con el Codec */
        call Program_DMA_Controller;            /* Inicializa Puerto Serial 1  tx and rx
                                                   Transferencia DMA */
        call AD1819_Codec_Initialization;       /* Inicializa y  programa AD1819 */
        call Inicializacion_Bufers;

        bit set imask SPT1I;                    /* Comienza Procesamiento de audio*/

        call Blink_LEDs_Test;

        wait_forever:
        call wait_flag1;
```

```
        bit tgl ustat1 0x3F;                                    /* toggle flag 4-9 LEDs */
        dm(IOSTAT)=ustat1;
        jump wait_forever;

/*-------------------------------------------------------------------------------------*
*                              Subrutinas                                              *
*-------------------------------------------------------------------------------------*/
wait_flag1:
        /* wait for flag 1 button press and release */
        if flag1_in jump wait_flag1;       /* espera por presión de un pulsador/
release:
        if not flag1_in jump release;      /* espera se libere el pulsador
        rts;

/*-------------------------------------------------------------------------------------------------------------------
*/
/* Nota: Esta rutina es la primera llamada por el Vector Reset en la tabla de inetrrupciones          */
/*-------------------------------------------------------------------------------------------------------------------*/

Init_DSP:
        ustat1=0x3F;                            /* flags 4 a 9 son salidas para LEDs */
        dm(IOCTL)=ustat1;
        bit set ustat1 0x3F;                    /*  toggle 4-9 LEDs */
        dm(IOSTAT)=ustat1;                                  /*  enciende todos los LEDs */
        bit clr mode2 FLG2O | FLG1O | FLG0O;        /* flag 3, 2 y 0 Entradas  */
        bit set mode2 IRQ1E | IRQ2E;                /* irqx  sensitividad ligera          */
        bit clr mode2 IRQ0E;              /* reservA irq1 para nivel de sensitividad  para UART */
        IRPTL = 0x00000000;                         /* borra interrupciones pendientes */
        bit set mode1 IRPTEN | NESTM;               /*  habilita   interrupts  globales  y  enlaces  */

        bit set imask IRQ0I | IRQ1I | IRQ2I;       /* irq1 y irq2 habiltados,
                                                    reserva irq0 habiltado para UART */
        bit set mode1 ALUSAT;                       /* habilita modo saturacion ALU  */

        L0 = 0;
        L1 = 0;
        L2 = 0;
        L3 = 0;
        L4 = 0;
        L5 = 0;
        L6 = 0;
        L7 = 0;
        L8 = 0;
        L9 = 0;
        L10 = 0;
        L11 = 0;
        L12 = 0;
        L13 = 0;
        L14 = 0;
        L15 = 0;

        rts;

.endseg;
```

Rutina de Inicialización Memoria Externa SDRAM

Archivo: *Inicializacion_SDRAM.asm*

```
/*****************************************************************/
/*                  Inicializacion Interface SDRAM               */
/*            Test I/O Programables Flags LEDs                   */
/*****************************************************************/
#include "def21065L.h"
#include "new65Ldefs.h"

#define sdram_size 0xfffff

.SEGMENT/DM segsdram;

.ENDSEG;

.SEGMENT/PM pm_code;


/*************************************************************************************
        Setup  SDRAM
        Asume SDRAM part# MT48LC1M16A1TG-12 S
                SDCLK=60MHz
                tCK=15ns min @ CL=2 -> SDCL=2
                tRAS=72ns min       -> SDTRAS=5
                tRP=36ns min        -> SDTRP=3
                tREF=64ms/4K rows   -> SDRDIV=(2(30MHz)-CL-tRP-4)64ms/4096=937cycles

        2 SDRAMs by 16 bits wide total=1Mbitx32
        Mapeado a MS3 addresses 0x03000000-0x030fffff
*************************************************************************************/

.GLOBAL     Init_65L_SDRAM_Controller;
.GLOBAL     Blink_LEDs_Test;


Init_65L_SDRAM_Controller:
        ustat1=dm(WAIT);
        bit clr ustat1 0x000f8000;          /* borra modo y estado de espera MS3 */
        dm(WAIT)=ustat1;

        ustat1=937;                         /*refresca tasa*/
        dm(SDRDIV)=ustat1;

        ustat1=dm(IOCTL);                   /*enmascara settings SDRAM */
        bit set ustat1 SDPSS|SDBN2|SDBS3|SDTRP3|SDTRAS5|SDCL2|SDPGS256|SDDCK1;
        dm(IOCTL)=ustat1;

        rts;
```

```
/*---------------------------------------------------------------------------------*/

Blink_LEDs_Test:
                    /* Setup FLAG salidas */
                    ustat1=dm(IOCTL);
                    bit set ustat1 FLG4O|FLG5O|FLG6O|FLG7O|FLG8O|FLG9O;
                    dm(IOCTL)=ustat1;               /*flag 5-9 son salidas*/
                    ustat1=0x3f;                    /*borra flags para arrancar*/
                    dm(IOSTAT)=ustat1;

                    /* Cinco parpadeos flags  */
                    lcntr=10, do blink_loop until lce;
                    lcntr=15000000;
                    do delay until lce;
                    delay:   nop;

                    ustat1=dm(IOSTAT);
                    bit tgl ustat1 FLG4|FLG5|FLG6|FLG7|FLG8|FLG9;
blink_loop:         dm(IOSTAT)=ustat1;
                    rts;
.ENDSEG;
```

Rutina de Inicialización Codec

Archivo: *Inicializacion_Codec.asm*

```
/*****************************************************************/
/**       Inicialicacion CODEC                                  **/
/*****************************************************************/
#include        "def21065l.h"
#include        "new65Ldefs.h"

/*#define       MIC_INPUT     0*/
#define         LINE_INPUT    1

.GLOBAL             Program_SPORT1_Registers;
.GLOBAL             Program_DMA_Controller;
.GLOBAL             AD1819_Codec_Initialization;
.GLOBAL             tx_buf;
.GLOBAL             rx_buf;
.EXTERN             Clear_All_SPT1_Regs;
.EXTERN             spt1_svc;

/* AD1819 Definicion de Direcciones de los Registros del Codec  */
#define         REGS_RESET                      0x0000
#define         MASTER_VOLUME                   0x0200
#define         RESERVED_REG_1                  0x0400
#define         MASTER_VOLUME_MONO              0x0600
#define         RESERVED_REG_2                  0x0800
#define         PC_BEEP_Volume                  0x0A00
#define         PHONE_Volume                    0x0C00
#define         MIC_Volume                      0x0E00
#define         LINE_IN_Volume                  0x1000
#define         CD_Volume                       0x1200
#define         VIDEO_Volume            0x1400
#define         AUX_Volume                      0x1600
#define         PCM_OUT_Volume                  0x1800
#define         RECORD_SELECT                   0x1A00
#define         RECORD_GAIN                             0x1C00
#define         RESERVED_REG_3                  0x1E00
#define         GENERAL_PURPOSE                 0x2000
#define         THREE_D_CONTROL_REG             0x2200
#define         RESERVED_REG_4                  0x2400
#define         POWERDOWN_CTRL_STAT             0x2600
#define         SERIAL_CONFIGURATION    0x7400
#define         MISC_CONTROL_BITS               0x7600
#define         SAMPLE_RATE_GENERATE_0          0x7800
#define         SAMPLE_RATE_GENERATE_1          0x7A00
#define         VENDOR_ID_1                     0x7C00
#define         VENDOR_ID_2                     0x7E00

/* Bits para enmascarar en el registro de configuración serial
   para el acceso de los registros de alguno de los 3 codecs */
#define         MASTER_Reg_Mask                 0x1000
```

```
#define         SLAVE1_Reg_Mask                 0x2000
#define         SLAVE2_Reg_Mask                 0x4000
#define         MASTER_SLAVE1                   0x3000
#define         MASTER_SLAVE2                   0x5000
#define         MASTER_SLAVE1_SLAVE2    0x7000

/* Macros para colocar los  Bits 15, 14 and 13 en Slot 0 Tag Phase */
#define         ENABLE_VFbit_SLOT1_SLOT2        0xE000
#define         ENABLE_VFbit_SLOT1              0xC000

/* Definicion  Timeslot TDM AD1819 */
#define         TAG_PHASE                       0
#define         COMMAND_ADDRESS_SLOT 1
#define         COMMAND_DATA_SLOT               2
#define         STATUS_ADDRESS_SLOT             1
#define         STATUS_DATA_SLOT        2
#define         LEFT                            3
#define         RIGHT                           4

#define         AD1819_RESET_CYCLES     60
                        /* ad1819 RESETb spec = 1.0(uS) min */
                        /* 60(MIPs) = 16.67 (nS) cycle time, -->  >=    40 cycles */
#define         AD1819_WARMUP_CYCLES    60000
                        /* ad1819 warm-up = 1.0(mS) */
                        /* 60(MIPs) = 16.67 (nS) cycle time, --> >= 40000 cycles */


/*-------------------------------------------------------------------------------*/
.segment /dm    dm_codec;
.var rx_buf[5];   /* buffer recpetor */

                                        /* buffer transmisor*/
.var tx_buf[7] = ENABLE_VFbit_SLOT1_SLOT2,          /* coloca los bit vaklidos slot 0, 1, y 2 */
          SERIAL_CONFIGURATION,             /* direccion registro configuaracion serial */
           0xFF80, /* inicialmente coloca en modo slot de 16 bit para compatibilidad ADI SPORT */
           0x0000,                                                  /* rellena los otros slot con ceros */
           0x0000,
          0x0000,
          0x0000;

.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* recpetor tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 7, 1, 0;      /* transmisor tcb */

/* Incializacion Registros Codec */
#define Select_LINE_INPUTS   0x0404 /* LINE IN - 0X0404, Mic In - 0x0000 */
#define Select_MIC_INPUT     0x0000
#define Line_Level_Volume    0x0000 /* 0 dB para linea de entrada */
#define Mic_Level_Volume     0x0F0F
#define Sample_Rate1         23456
#define Sample_Rate2         8000//48000

.var Init_Codec_Registers[34] =
          MASTER_VOLUME, 0x0000,            /* Master Volume sin atenuacion attenuation */
MASTER_VOLUME_MONO,    0x8000,          /* Master Mono volume es mudo */
```

```
PC_BEEP_Volume,             0x8000,         /* PC volume es mudo */
PHONE_Volume,               0x8008,         /* Phone Volume es mudo*/
MIC_Volume,                 0x8008,         /* MIC Input analog loopback es mudo */
LINE_IN_Volume,             0x8808,         /* Line Input analog loopback es mudo*/
CD_Volume,                  0x8808,         /* CD Volume es mudo */
VIDEO_Volume,               0x8808,         /* Video Volume es mudo */
AUX_Volume,                 0x8808,          /* AUX Volume es mudo */
PCM_OUT_Volume,             0x0000,/* 0x0808,*/      /* salida PCM  del DACs es de ganancia 0
db  para ambos canales */

/* Usado para cambiar entre  MIC input y Line input */
#ifdef MIC_INPUT
        RECORD_SELECT, Select_MIC_INPUT,   /*  seleccion Lineas de entrada L/R canales */
        RECORD_GAIN,        Mic_Level_Volume,      /* gananacia 0 dB on  ambos  L/R canales*/
#else
        RECORD_SELECT,Select_LINE_INPUTS,  /*  seleccion Lineas de entrada L/R canales */
        RECORD_GAIN,Line_Level_Volume,   /* gananacia 0 dB on  ambos  L/R canales*/
#endif

        GENERAL_PURPOSE,            0x0000,
        THREE_D_CONTROL_REG,    0x0000,
        MISC_CONTROL_BITS,          0x0000,                 /* usa SR0 para ambos canales */
        SAMPLE_RATE_GENERATE_0,Sample_Rate2,        /* tasa de muestrero seleccionable
                                                            por el usuario */
        SAMPLE_RATE_GENERATE_1,         48000;

.var Codec_Init_Results[34] =
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0,
        0,              0;

.endseg;

.SEGMENT /pm        pm_code;
```

```
/* ---------------------------------------------------------------------------------------------------------*/
/* Programando registro de control Sport1                                                                   */
/* ---------------------------------------------------------------------------------------------------------*/

Program_SPORT1_Registers:
        /* Requerido par deshabilitar configuracion  SPORT para EZLAB RS232 debugger */
        CALL Clear_All_SPT1_Regs;    /* Clear and Reset SPORT1 and DMAs */

        /* registro de control recpetor sport1  */
        R0 = 0x0F8C40F0;       /* 16 chans, int rfs, ext rclk, slen = 15, sden & schen habilitados */
        dm(SRCTL1) = R0;                    /* sport 0 receive control register */

        /* registro receptor sport1  frame sync divide  */
        R0 = 0x00FF0000;                    /* SCKfrq(12.288M)/RFSfrq(48.0K)-1 = 0x00FF */
        dm(RDIV1) = R0;

        /* registro de control transmisor sport1  */
        R0 = 0x001C00F0;       /* 1 cyc mfd, data depend, slen = 15, sden & schen enabled */
        dm(STCTL1) = R0;                    /* sport 0 transmit control register */

        /* habilita registros receptor  y transmisor palabra multicanal sport1  */
        R0 = 0x0000001F;                    /* habilita canales repecion  0-4 */
        dm(MRCS1) = R0;
        R0 = 0x0000007F;                    /* habilita canales transmision 0-6 */
        dm(MTCS1) = R0;

        /* habilita registros receptor  y transmisor companding multicanal sport1  */
        R0 = 0x00000000;                    /* no companding */
        dm(MRCCS1) = R0;                    /* no companding en recepcion */
        dm(MTCCS1) = R0;                    /* no companding en transmision */

        RTS;

/*---------------------------------------------------------------------------------*/
/*                                DMA Controller Programming For SPORT1            */
/*---------------------------------------------------------------------------------*/

Program_DMA_Controller:
        r1 = 0x0001FFFF;                    /*enmascara registro cpx */

        /* registro de punteros enlazados sport1 dma control tx  */
        r0 = tx_buf;
        dm(xmit_tcb + 7) = r0;
        r0 = 1;
        dm(xmit_tcb + 6) = r0;
        r0 = 7;
        dm(xmit_tcb + 5) = r0;
        r0 = xmit_tcb + 7;
        r0 = r1 AND r0;
        r0 = BSET r0 BY 17;
        dm(xmit_tcb + 4) = r0;
        dm(CPT1A) = r0;

        /* registro de punteros enlazados sport1 dma control rx  */
```

```
        r0 = rx_buf;
        dm(rcv_tcb + 7) = r0;        /* direccion interna dma usado para encadenar */
        r0 = 1;
        dm(rcv_tcb + 6) = r0;        /* modificador de memoria interna DMA  */
        r0 = 5;
        dm(rcv_tcb + 5) = r0;        /* contador de buffer de memoria interna DMA */
        r0 = rcv_tcb + 7;
        r0 = r1 AND r0;                                  /* enmascra punteors */
        r0 = BSET r0 BY 17;                              /* coloca el bit pci */
        dm(rcv_tcb + 4) = r0;          /* ecribe al bloque receptor DMA cambia punteros a  TCB
buffer*/
        dm(CPR1A) = r0;               /* cambia puntero bloque receptor, inicializa rx0 tranferencia
DMA  */

        RTS;

/* ---------------------------------------------------------------------------- */
/*                    Inicializacion   Codec  AD1819A                    */
/* ---------------------------------------------------------------------------- */

AD1819_Codec_Initialization:
        bit set imask SPT1I;     /* habilta interrupcion sport0 x-mit */

Wait_Codec_Ready:                            /* Wait for CODEC Ready State */
        R0 = DM(rx_buf + 0);                 /* obtien bit 15 status bit del AD1819 tag phase slot
0 */
        R1 = 0x8000;                         /*enmascara salida codec ready bit in tag phase */
        R0 = R0 AND R1;                      /* prueba codec ready status flag bit */
        IF EQ JUMP Wait_Codec_Ready;         /* Si flag esta en bajo espera por un alto*/

        idle;                        /* espera para un acople de marcos de audio TDM par pasar */
        idle;

Initialize_1819_Registers:
        i4 = Init_Codec_Registers;     /* apunta a comando de inicilizacion codec */
        r15 = ENABLE_VFbit_SLOT1_SLOT2;  /* habilita los bits validos de marcos, y slots 1 y 2  */

        LCNTR = 17, DO Codec_Init UNTIL LCE;
                dm(tx_buf + TAG_PHASE) = r15;   /* coloca los bits validos de slot  en tag phase
                                                   para slots 0, 1 , 2 */
                r1 = dm(i4, 1);              /* busca la direccion de codigo de registro siguiente */
                dm(tx_buf + COMMAND_ADDRESS_SLOT) = r1;         /* coloca dato de código de
                                                                   registro buscado en slot 1
*/
                r1 = dm(i4, 1);                          /*busca contenido  de registro de datos  */
                dm(tx_buf + COMMAND_DATA_SLOT) = r1;     /* coloca dato de código de registro
                                                            buscado en slot 2 */
Codec_Init:     idle;                        /* espera hasta que el marco  TDM es transmitido*/
```

```
/*------------------------------------------------------------------------------------------------------------*/
/* Verifica la integridad del estado de los registros de control indexados del  AD1819a   */
/* para ver si la comunicacion fue satisfactoria                                                                */
/*------------------------------------------------------------------------------------------------------------*/


verify_reg_writes:
        i4 = Init_Codec_Registers;
        m4 = 2;
        i5 = Codec_Init_Results;
        r15 = ENABLE_VFbit_SLOT1;           /* habilita los bits validos de marcos, y slots 1 y 2  */

        LCNTR = 17, Do ad1819_register_status UNTIL LCE;
                dm(tx_buf + TAG_PHASE) = r15;       /* coloca los bits validos de slot  en tag
                                                     phase para slots 0, 1 , 2 */
                r1 = dm(i4,2);                      /*obtiene la direción de registros indexados
                                                     que van a ser inspeccionados*/
                r2 = 0x8000;                        /* coloca el bit 15 para lectura en comando
                                                     de direccion de palabra*/
                r1 = r1 OR r2;
                dm(tx_buf + COMMAND_ADDRESS_SLOT) = r1;     /* envia el valor de salida
                                                            del comandado de direccion timeslot */
                idle;
                idle;
                r3 = dm(rx_buf + STATUS_ADDRESS_SLOT);
                dm(i5,1) = r3;
                r3 = dm(rx_buf + STATUS_DATA_SLOT);     /* busca el valor el dato del registro
                                                            indexado solicitado*/
                dm(i5,1) = r3;                      /* almacena resultados en buffer */
                ad1819_register_status: nop;


PowerDown_DACs_ADCs:
        idle;
        r15 = ENABLE_VFbit_SLOT1_SLOT2;             /* habilita los bits validos de
marcos,
                                                    y slots 1 y 2  */
        dm(tx_buf + TAG_PHASE) = r15;               /* coloca los bits validos de slot en
                                                     tag phase para slots 0, 1 , 2 */
        r0=POWERDOWN_CTRL_STAT;
        dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
        r0=0x0300;                                  /* enciende todos los  DACs/ADCs */
        dm(tx_buf + COMMAND_DATA_SLOT) = r0;
        idle;
        idle;

        LCNTR = AD1819_RESET_CYCLES-2, DO reset_loop UNTIL LCE;
        reset_loop:     NOP;                        /* espera para un reset */

        idle;
        r15 = ENABLE_VFbit_SLOT1_SLOT2;  /* habilita los bits validos de marcos, y slots 1 y 2  */
        dm(tx_buf + TAG_PHASE) = r15;       /* coloca los bits validos de slot
                                            en tag phase para slots 0, 1 , 2 */
```

```
        r0=POWERDOWN_CTRL_STAT;         /* direccion para escribir  */
        dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
        r0=0;                                   /* enciende toodo los DACs/ADCs */
        dm(tx_buf + COMMAND_DATA_SLOT) = r0;
        idle;
        idle;

        LCNTR = AD1819_WARMUP_CYCLES-2, DO warmup_loop2 UNTIL LCE;
warmup_loop2: NOP;                      /* espera para calentamiento del AD1819


/* ------------------------------------------------------------------------------------------------- */

        bit clr imask SPT1I;                    /* deshabilita sport1 xmit  */

Install_ISR_SPORT1_Tx_ISR:
        PX2 = 0x063e0000;                               /* Upper 32 bit Opcode para instrucciones
        PX1 = Procesamiento_Muestras_Audio;             /* Lower 16 bits de Opcode contiene
                                                           drecciones jump */
        PM(spt1_svc) = PX;                              /*  copia  a  la  localizacion  0x34  -  SPORT1
del
                                                           vector de interrupciones*/

        RTS;                                            /* Finaliza inicializacion  AD1819A  */

/* ------------------------------------------------------------------------------------------- */
 .endseg;
```

Rutina para Limpiar los Registros SPT1

Archivo: *Clear_Registros_SPT1.asm*

```
/*****************************************************************/
/**              Limpia Registros        SPT1            ***/
/*****************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"


.GLOBAL                 Clear_All_SPT1_Regs;

.SEGMENT /pm            pm_code;

Clear_All_SPT1_Regs:
        IRPTL = 0x00000000;             /* borra interrupciones pendientes */
        bit clr imask SPT1I;

        R0 = 0x00000000;
        dm(SRCTL1) = R0;                /* registro de control receptor sport1  */

        dm(RDIV1) = R0;                 /* registro de control receptor sport1 frame sync divide*/

        dm(STCTL1) = R0;                /* registro de control transmisor sport 0  */
        dm(MRCS1) = R0;         /* habilita registro de contol receptor palabra multicanal sport1   */
        dm(MTCS1) = R0;         /* habilita registro de contol transmisor palabra multicanal sport1  */
        dm(MRCCS1) = R0;/* habilita registro de contol receptor companding multicanal sport1 */
        dm(MTCCS1) = R0;/* habilita registro de contol transmisor companding multicanal sport1 */

        /* reset SPORT1 DMA  */
        R1 = 0x1FFFF; dm(IIR1A) = R1;
        R1 = 0x0001;    dm(IMR1A) = R1;
        R1 = 0xFFFF;    dm(CR1A)  = R1;
        R1 = 0x1FFFF; dm(CPR1A) = R1;
        R1 = 0x1FFFF; dm(GPR1A) = R1;
        R1 = 0x1FFFF; dm(IIT1A) = R1;
        R1 = 0x0001;    dm(IMT1A) = R1;
        R1 = 0xFFFF;    dm(CT1A)  = R1;
        R1 = 0x1FFFF; dm(CPT1A) = R1;
        R1 = 0x1FFFF; dm(GPT1A) = R1;

        RTS;

.ENDSEG;
```

Rutina de Procesamiento del Codec

Archivo: *Procesamiento_Codec.asm*

```
/*****************************************************************/
/**       Procesamiento  CODEC                                  **/
/*****************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"

/* Definicion AD1819 TDM Timeslot  */
#define         TAG_PHASE                       0
#define         COMMAND_ADDRESS_SLOT 1
#define         COMMAND_DATA_SLOT               2
#define         STATUS_ADDRESS_SLOT             1
#define         STATUS_DATA_SLOT        2
#define         LEFT                            3
#define         RIGHT                           4

/* Bits validos Left y Right ADC valid Bits usado prueba ed datos ed audio validos en  TDM frame
actual */
#define         M_Left_ADC                      12
#define         M_Right_ADC                     11
#define         DAC_Req_Left            0x80
#define         DAC_Req_Right                   0x40

.GLOBAL         Procesamiento_Muestras_Audio;
.GLOBAL         Left_Channel_In;
.GLOBAL         Right_Channel_In;
.GLOBAL         Left_Channel_Out;
.GLOBAL         Right_Channel_Out;
.EXTERN                 tx_buf;
.EXTERN                 rx_buf;
.EXTERN                 Efecto_Audio;

.segment /dm    dm_codec;

/* Almacena datos canlaes estereo AD1819a  - usado para procesamiento de audio de datos
recibidos desde el codec */
.VAR            Left_Channel_In;
.VAR            Right_Channel_In;
.VAR            Left_Channel_Out;
.VAR            Right_Channel_Out;
.VAR            ADC_valid_bits;


/* contador audio frame/ISR AC'97  , para proósitos de depueración */
.VAR            audio_frame_timer = 0;


.endseg;
```

```
.segment /pm pm_code;

Procesamiento_Muestras_Audio:
        /* Información Tag Phase Slot */
        r0 = 0x8000;            /* Coloca Marco válido  bit 15 en etiqueta de Phase Slot  0   */
        dm(tx_buf + TAG_PHASE) = r0;                    /* Escribe etiqueta a tx-buf ASAP
                                                        antes de cambiar la salida de SPORT! */
        r0 = 0;                                         /* Borra  slots Marcos salida de audio AC97  */
        dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
        dm(tx_buf + COMMAND_DATA_SLOT) = r0;
        dm(tx_buf + LEFT) = r0;
        dm(tx_buf + RIGHT) = r0;

check_ADCs_for_valid_data:
   r0 = dm(rx_buf + TAG_PHASE);                         /* Obtiene bits validos ADC desde slot*/
   r1 = 0x1800;                                         /* examina datos validos para L/R ADC  */
   r2 = r0 and r1;                                      /* enmascara otro bits en etiqueta */
        dm(ADC_valid_bits) = r2;

set_tx_slot_valid_bits:
        r1 = dm(tx_buf + TAG_PHASE);            /* set tx valid bits based on ADC valid bits info */
        r3 = r2 or r1;          /* coloca bits etiqueta canal derecho e izquierdo si es requerido */

        dm(tx_buf + TAG_PHASE) = r3;            /* Escribe etiqueta a tx-buf ASAP
                                                   antes de cambiar la salida de SPORT! */
check_AD1819_ADC_left:
        BTST r2 by M_Left_ADC;                  /* Chequea  bit valido Master derecho ADC  */
        IF sz JUMP check_AD1819_ADC_right;   /* Si dato valido entonces salva muestra ADC */
          r6 = dm(rx_buf + LEFT);         /* obtiene muestra entrada Master canal izquierdo 1819*/
          r6 = lshift r6 by 16;   /* cambia a MSBs para preservar el signo en formato 1.31  */
          dm(Left_Channel_In) = r6;       /* Salva el dato almacenado pra el procesamiento */

check_AD1819_ADC_right:
        BTST r2 by M_Right_ADC;         /* Chequea  bit valido Master derecho ADC  */
        If sz rti;                              /* Si dato valido entonces salva muestra ADC */
          r6 = dm(rx_buf + RIGHT);      /* obtiene muestra entrada Master canal derecho 1819*/
          r6 = lshift r6 by 16;     /* cambia a MSBs para preservar el signo en formato 1.31  */
          dm(Right_Channel_In) = r6;    /* Salva el dato almacenado pra el procesamiento */

/*****************************************************************************************************************/
user_applic:
        CALL Efecto_Audio;
/* ---- el procesmiento DSP ha finalizado, entonces el resultado es colocado en AD1819 ---- */

playback_audio_data:
        /* Transmite datos validos derecho e izquierdo toda vez que ADCs tiene dato valido */
        r2 = dm(ADC_valid_bits);

tx_AD1819_DAC_left:
        BTST r2 by M_Left_ADC;                  /* Chequea para ver si necesitamos enviar
                                                    muestra DAC izquierdo */
        IF sz JUMP tx_AD1819_DAC_right;   /* si dato es valido entones transmite muestra DAC  */

        r15 = dm(Left_Channel_Out);             /* obtiene resultado salida canal 1*/
```

```
        r15 = lshift r15 by -16;                /* bits 0..15 for SPORT tx */
        dm(tx_buf + LEFT) = r15;                /* resultado salida izquierda a AD1819a  Slot 3 */

tx_AD1819_DAC_right:
        BTST r2 by M_Right_ADC;                 /* Chequea para ver si necesitamos enviar
                                                   muestra DAC derecho */
        If sz jump tx_done;             /* si dato es valido entones transmite muestra DAC  */
        r15 = dm(Right_Channel_Out);            /* obtiene resultado salida canal 2*/
        r15 = lshift r15 by -16;                /*  bits 0..15 para SPORT tx */
        dm(tx_buf + RIGHT) = r15;               /* resultado salida derecha a AD1819a Slot 4 */

tx_done:
        r0=dm(audio_frame_timer);                       /* Obtien el último conteo */
        rti(db);                                /* Retorna de la interrupción, salto retardado */
        r0=r0+1;                                /* incrementa contador */
        dm(audio_frame_timer)=r0;               /* salva contador actualiazdo */


.endseg;
```

Tabla de Interruciones

Archivo: *Tabla_Interrupciones.asm*

```
/* ********************************************************************************/
/*                    VECTOR INTERRUPCIONES  ADSP-21065L                          */
/* ********************************************************************************* */


.EXTERN             Procesamiento_Muestras_Audio;
.EXTERN             _main;
.EXTERN             Init_DSP;
.GLOBAL             spt1_svc;

.SEGMENT/PM   isr_tbl;                  /* 21065L Interrupt Service Table */

/* 0x00 Reserved Interrupt */
/*                 0x00    0x01    0x02    0x03    0x04 */
/* reserved_0:  NOP;    NOP;    NOP;    NOP;    NOP; */

/* *** Reset vector *** */
/* 0x05 - reset vector starts at location 0x8005 */
rst_svc:        call Init_DSP;
                NOP;
                jump _main;

/* 0x08 - Reserved interrupt */
reserved_0x8:  NOP;    NOP;    NOP;    NOP;

/* 0x0C - Vector for status stack/loop stack overflow or PC stack full: */
sovfi_svc:      RTI;    RTI;    RTI;    RTI;

/* 0x10 - Vector for high priority timer interrupt: */
tmzhi_svc:      RTI;    RTI;    RTI;    RTI;

/* 0x14 - Vectors for external interrupts: */
vrpti_svc:      RTI;    RTI;    RTI;    RTI;

/* 0x18 - IRQ2 Interrupt Service Routine (ISR)  */
irq2_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x1C - IRQ1 Interrupt Service Routine (ISR)  */
irq1_svc:       RTI;    RTI;    RTI;    RTI;

/* IRQ0 INTERRUPT VECTOR */

/* *** 0x20 - IRQ0 Interrupt Service Routine (ISR) , 4 locations max *** */
irq0_svc:
        /* JUMP KERNEL_UART_ISR; */ RTI;   RTI;    RTI;    RTI;

/* 0x24  - Reserved interrupt     */
reserved_0x24: NOP;    NOP;    NOP;    NOP;
```

```
/* 0x28 - Vectors for Serial Port 0 Receive A & B DMA channels 0/1 */
spr0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x2C - Vectors for Serial Port 1 Receive A & B DMA channels 2/3 */
spr1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x30 - Vectors for Serial Port 0 Transmint A & B DMA channels 4/5 */
spt0_svc:       RTI;    RTI;    RTI;    RTI;

spt1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x38 - Reserved Interrupt */
reserved_0x38: RTI;    RTI;    RTI;    RTI;

/* 0x3C - Reserved Interrupt */
reserved_0x3c: RTI;    RTI;    RTI;    RTI;

/* 0x40 - Vector for External Port DMA channel 8 */
ep0_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x44 - Vector for External Port DMA channel 9 */
ep1_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x48 - Reserved Interrupt */
reserved_0x48: RTI;    RTI;    RTI;    RTI;

/* 0x4C - Reserved Interrupt */
reserved_0x4c: RTI;    RTI;    RTI;    RTI;

/* 0x50 - Reserved Interrupt */
reserved_0x50: RTI;    RTI;    RTI;    RTI;

/* 0x54 - Vector for DAG1 buffer 7 circular buffer overflow */
cb7_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x58 - Vector for DAG2 buffer 15 circular buffer overflow */
cb15_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x5C - Vector for lower priority timer interrupt */
tmzl_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x60 - Vector for fixed-point overflow */
fix_svc: RTI;    RTI;    RTI;    RTI;

/* 0x64 - Floating-point overflow exception */
flt0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x68 - Floating-point underflow exception */
fltu_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x6C - Floating-point invalid exception */
flti_svc: RTI;    RTI;    RTI;    RTI;
```

```
/* 0x70 - User software interrupt 0 */
sft0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x74 - User software interrupt 1 */
sft1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x78 - User software interrupt 2 */
sft2_svc:       RTI;    RTI;    RTI;    RTI;

#define KERNEL_SWI3_ISR  0x9002

sft3_svc: JUMP KERNEL_SWI3_ISR;    RTI;      RTI;    RTI;

.ENDSEG;
```

Archivo Linker para ser usado con la tarjeta de evaluación en la ejecución de los algoritmos efectos de audio

Archivo: *EZKITLITE_21065L.ldf*

```
// ************************************************************************************/
// *                                                                                  */
// *                  21065L EZ-LAB LINKER DESCRIPTION FILE                           */
// *                                                                                  */
// ************************************************************************************ */

ARCHITECTURE(ADSP-21065L)

SEARCH_DIR( $ADI_DSP\21k\lib )

// The lib060.dlb must come before libc.dlb because libc.dlb has some 21020
// specific code and data
$LIBRARIES = lib060.dlb;

// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
$OBJECTS = $COMMAND_LINE_OBJECTS;

MAP(Efectos_memory.map)
MEMORY
{
//      IRQ0 Interrupt 0x20 - 0x23 reserved by EZ-LAB UART Monitor Program */
//      SWI3 User Interrupt 0x7c - 0x7f reserved by EZ-LAB UART Monitor Program */
        isr_tbl    { TYPE(PM RAM) START(0x00008005) END(0x0000807f) WIDTH(48) }
        pm_code         { TYPE(PM RAM) START(0x00008100) END(0x00008bff) WIDTH(48) }
        pm_data         { TYPE(PM RAM) START(0x00009400) END(0x000097ff) WIDTH(32) }
//      krnl_code       { TYPE(PM RAM) START(0x00009000) END(0x000097ff) WIDTH(48) }
        dm_data         { TYPE(DM RAM) START(0x0000c000) END(0x0000dfff) WIDTH(32) }
        EMAFE_addr    { TYPE(DM RAM) START(0x01000000) END(0x01000000) WIDTH(32) }
        EMAFE_data    { TYPE(DM RAM) START(0x01000001) END(0x01000001) WIDTH(32) }
        UART_regs     { TYPE(DM RAM) START(0x01000008) END(0x0100000F) WIDTH(32) }
        codec_reset   { TYPE(DM RAM) START(0x01000010) END(0x01000010) WIDTH(32) }
        seg_dm_sdram { TYPE(DM RAM) START(0x03000000) END(0x030ffeff) WIDTH(32) }
        krnl_ext_res { TYPE(DM RAM) START(0x030fff00) END(0x030fffff) WIDTH(32) }
}

PROCESSOR p0
{
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS
   {
        // .text output section
        isr_tbl
        {
                INPUT_SECTIONS( $OBJECTS(isr_tbl) $LIBRARIES(isr_tbl))
        } >isr_tbl
```

```
        pm_code
        {
                INPUT_SECTIONS( $OBJECTS(pm_code) $LIBRARIES(pm_code))
        } >pm_code


        pm_data
        {
                INPUT_SECTIONS( $OBJECTS(pm_data) $LIBRARIES(pm_data))
        } >pm_data


        dm_data
        {
                INPUT_SECTIONS( $OBJECTS(dm_data dm_codec) $LIBRARIES(dm_data))
        } > dm_data


        dm_sdram //SHT_NOBITS
        {
                INPUT_SECTIONS( $OBJECTS(dmEfectos segsdram))
        } > seg_dm_sdram
    }
}
```

ARCHIVOS PARA EJECUTAR LOS FILTROS FIR E IIR
EN LA TARJETA DE EVALUACIÓN EZ KIT LITE 21065L

Rutina Procesamiento del Codec para usarse con los filtros FIR e IIR

Archivo: *Procesamiento_Codec.asm*

```
/*****************************************************************/
/**        Procesamiento  CODEC                                 **/
/*****************************************************************/

#include        "def21065l.h"
#include        "new65Ldefs.h"

/* Definicion AD1819 TDM Timeslot  */
#define         TAG_PHASE                       0
#define         COMMAND_ADDRESS_SLOT            1
#define         COMMAND_DATA_SLOT               2
#define         STATUS_ADDRESS_SLOT             1
#define         STATUS_DATA_SLOT                2
#define         LEFT                            3
#define         RIGHT                           4

/* Bits validos Left y Right ADC valid Bits usado prueba ed datos ed audio validos en  TDM frame actual */
#define         M_Left_ADC                      12
#define         M_Right_ADC                     11
#define         DAC_Req_Left                    0x80
#define         DAC_Req_Right                   0x40

.GLOBAL         Procesamiento_Muestras_Audio;
.GLOBAL         Left_Channel_In;
.GLOBAL         Right_Channel_In;
.GLOBAL         Left_Channel_Out;
.GLOBAL         Right_Channel_Out;
.EXTERN                 tx_buf;
.EXTERN                 rx_buf;

.GLOBAL                 RX_left_flag;           // Nueva Línea
.GLOBAL                 RX_right_flag;          // Nueva Línea

.EXTERN                 Efecto_Audio;


.segment /dm    dm_codec;

/* Almacena datos canales estereo AD1819a  - usado para procesamiento de audio de datos recibidos desde el codec */
.VAR            Left_Channel_In;
.VAR            Right_Channel_In;
.VAR            Left_Channel_Out;
```

```
.VAR            Right_Channel_Out;
.VAR            ADC_valid_bits;

.VAR            RX_left_flag;           // Nueva Línea
.VAR            RX_right_flag;          // Nueva Línea

/* contador audio frame/ISR AC'97  , para proósitos de depuración */
.VAR            audio_frame_timer = 0;


.endseg;
.segment /pm pm_code;

Procesamiento_Muestras_Audio:
        /* Información Tag Phase Slot */
        r0 = 0x8000;            /* Coloca Marco válido  bit 15 en etiqueta de Phase Slot  0   */
        dm(tx_buf + TAG_PHASE) = r0;                    /* Escribe etiqueta a tx-buf ASAP
                                                        antes de cambiar la salida de SPORT! */
        r0 = 0;                                         /* Borra  slots Marcos salida de audio AC97  */
        dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
        dm(tx_buf + COMMAND_DATA_SLOT) = r0;
        dm(tx_buf + LEFT) = r0;
        dm(tx_buf + RIGHT) = r0;

check_ADCs_for_valid_data:
   r0 = dm(rx_buf + TAG_PHASE);                         /* Obtiene bits validos ADC desde slot*/
   r1 = 0x1800;                                         /* examina datos validos para L/R ADC  */
   r2 = r0 and r1;                                      /* enmascara otro bits en etiqueta */
        dm(ADC_valid_bits) = r2;

set_tx_slot_valid_bits:
        r1 = dm(tx_buf + TAG_PHASE);            /* set tx valid bits based on ADC valid bits info */
        r3 = r2 or r1;          /* coloca bits etiqueta canal derecho e izquierdo si es requerido */

        dm(tx_buf + TAG_PHASE) = r3;            /* Escribe etiqueta a tx-buf ASAP
                                                   antes de cambiar la salida de SPORT! */

{check_AD1819_ADC_left:
        BTST r2 by M_Left_ADC;                  /* Chequea  bit valido Master derecho ADC  */
        IF sz JUMP check_AD1819_ADC_right;   /* Si dato valido entonces salva muestra ADC */
          r6 = dm(rx_buf + LEFT);         /* obtiene muestra entrada Master canal izquierdo 1819*/
          r6 = lshift r6 by 16;   /* cambia a MSBs para preservar el signo en formato 1.31  */
          dm(Left_Channel_In) = r6;       /* Salva el dato almacenado pra el procesamiento */ }

// ----------------------------------Nuevas Líneas--------------------//
check_AD1819_ADC_left:
        BTST r2 by M_Left_ADC;
        IF sz JUMP check_AD1819_ADC_right;
          r6 = dm(rx_buf + LEFT);
          r6 = lshift r6 by 16;
          dm(Left_Channel_In) = r6;
          r4 = 1;
          dm(RX_left_flag) = r4;
// ------------------------------------------------------------------------//
```

```
{check_AD1819_ADC_right:
        BTST r2 by M_Right_ADC;       /* Chequea  bit valido Master derecho ADC  */
        If sz rti;                    /* Si dato valido entonces salva muestra ADC */
          r6 = dm(rx_buf + RIGHT);    /* obtiene muestra entrada Master canal derecho 1819*/
          r6 = lshift r6 by 16;     /* cambia a MSBs para preservar el signo en formato 1.31  */
          dm(Right_Channel_In) = r6;  /* Salva el dato almacenado pra el procesamiento */}


// ----------------------------------Nuevas Líneas--------------------//
check_AD1819_ADC_right:
        BTST r2 by M_Right_ADC;
        If sz jump user_applic;
          r6 = dm(rx_buf + RIGHT);
          r6 = lshift r6 by 16;
          dm(Right_Channel_In) = r6;
          r4 = 1;
          dm(RX_right_flag) = r4;
// ------------------------------------------------------------------------//


user_applic:
        //CALL Efecto_Audio;

// ----------------------------------Nuevas Líneas--------------------//
        r4 = 0x0;
        dm(RX_right_flag) = r4;
        r4 = dm(RX_left_flag);
        r4 = pass r4;
        if eq jump playback_audio_data;
        call (pc, Efecto_Audio);

/*************************************************************************************************************/

playback_audio_data:
        /* Transmite datos validos derecho e izquierdo toda vez que ADCs tiene dato valido */
        r2 = dm(ADC_valid_bits);

tx_AD1819_DAC_left:
        BTST r2 by M_Left_ADC;              /* Chequea para ver si necesitamos enviar
                                               muestra DAC izquierdo */
        IF sz JUMP tx_AD1819_DAC_right;   /* si dato es valido entones transmite muestra DAC  */

        r15 = dm(Left_Channel_Out);         /* obtiene resultado salida canal 1*/
        r15 = lshift r15 by -16;            /* bits 0..15 for SPORT tx */
        dm(tx_buf + LEFT) = r15;            /* resultado salida izquierda a AD1819a  Slot 3 */

tx_AD1819_DAC_right:
        BTST r2 by M_Right_ADC;             /* Chequea para ver si necesitamos enviar
                                               muestra DAC derecho */
        If sz jump tx_done;               /* si dato es valido entones transmite muestra DAC  */
        r15 = dm(Right_Channel_Out);        /* obtiene resultado salida canal 2*/
        r15 = lshift r15 by -16;            /*  bits 0..15 para SPORT tx */
        dm(tx_buf + RIGHT) = r15;           /* resultado salida derecha a AD1819a Slot 4 */
```

```
tx_done:
        r0=dm(audio_frame_timer);               /* Obtien el último conteo */
        rti(db);                                /* Retorna de la interrupción, salto retardado */
        r0=r0+1;                                /* incrementa contador */
        dm(audio_frame_timer)=r0;               /* salva contador actualiazdo */


.endseg;
```

Tabla de Interrupciones para ser usada por el filtro FIR con la rutina IRQ1

Archivo: *Tabla_ Interrupciones.asm*

```
/* ********************************************************************************/
/*                    VECTOR INTERRUPCIONES  ADSP-21065L                          */
/*                         Filtto FIR con selleción del tipo de filtro            */
/* ********************************************************************************* */


.EXTERN         Procesamiento_Muestras_Audio;
.EXTERN         _main;
.EXTERN         Init_DSP;
.EXTERN         Cambio_coeficientes_filtroFIR;      //Nueva Línea
.GLOBAL         spt1_svc;


.SEGMENT/PM   isr_tbl;              /* 21065L Interrupt Service Table */

/* 0x00 Reserved Interrupt */
/*              0x00    0x01    0x02    0x03    0x04 */
/* reserved_0:  NOP;    NOP;    NOP;    NOP;    NOP; */

/* *** Reset vector *** */
/* 0x05 - reset vector starts at location 0x8005 */
rst_svc:        call Init_DSP;
                NOP;
                jump _main;

/* 0x08 - Reserved interrupt */
reserved_0x8:  NOP;    NOP;    NOP;    NOP;

/* 0x0C - Vector for status stack/loop stack overflow or PC stack full: */
sovfi_svc:      RTI;    RTI;    RTI;    RTI;

/* 0x10 - Vector for high priority timer interrupt: */
//tmzhi_svc:    jump genera_onda_seno;      RTI;    RTI;    RTI;
tmzhi_svc:      RTI;    RTI;    RTI;    RTI;

/* 0x14 - Vectors for external interrupts: */
vrpti_svc:      RTI;    RTI;    RTI;    RTI;

/* 0x18 - IRQ2 Interrupt Service Routine (ISR)  */
irq2_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x1C - IRQ1 Interrupt Service Routine (ISR)  */
irq1_svc:       jump Cambio_coeficientes_filtroFIR;    RTI;    RTI;    RTI;


/* IRQ0 INTERRUPT VECTOR */
/* ------------------------------------------------------- */
/* IRQ0 Interrupt, Used by the UART and Debug Monitor Program */
```

```
/* *** 0x20 - IRQ0 Interrupt Service Routine (ISR) , 4 locations max *** */
irq0_svc:
        /* JUMP KERNEL_UART_ISR; */ RTI;   RTI;     RTI;     RTI;

/* 0x24  - Reserved interrupt     */
reserved_0x24: NOP;    NOP;    NOP;    NOP;

/* 0x28 - Vectors for Serial Port 0 Receive A & B DMA channels 0/1 */
spr0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x2C - Vectors for Serial Port 1 Receive A & B DMA channels 2/3 */
spr1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x30 - Vectors for Serial Port 0 Transmint A & B DMA channels 4/5 */
spt0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x34 - Vectors for Serial Port 1 Transmit A & B DMA channels 6/7 */
spt1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x38 - Reserved Interrupt */
reserved_0x38: RTI;    RTI;    RTI;    RTI;

/* 0x3C - Reserved Interrupt */
reserved_0x3c: RTI;    RTI;    RTI;    RTI;

/* 0x40 - Vector for External Port DMA channel 8 */
ep0_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x44 - Vector for External Port DMA channel 9 */
ep1_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x48 - Reserved Interrupt */
reserved_0x48: RTI;    RTI;    RTI;    RTI;

/* 0x4C - Reserved Interrupt */
reserved_0x4c: RTI;    RTI;    RTI;    RTI;

/* 0x50 - Reserved Interrupt */
reserved_0x50: RTI;    RTI;    RTI;    RTI;

/* 0x54 - Vector for DAG1 buffer 7 circular buffer overflow */
cb7_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x58 - Vector for DAG2 buffer 15 circular buffer overflow */
cb15_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x5C - Vector for lower priority timer interrupt */
tmzl_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x60 - Vector for fixed-point overflow */
fix_svc: RTI;    RTI;    RTI;    RTI;

/* 0x64 - Floating-point overflow exception */
```

```
flt0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x68 - Floating-point underflow exception */
fltu_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x6C - Floating-point invalid exception */
flti_svc: RTI;    RTI;    RTI;    RTI;

/* 0x70 - User software interrupt 0 */
sft0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x74 - User software interrupt 1 */
sft1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x78 - User software interrupt 2 */
sft2_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x7C - User software interrupt 3 */
#define KERNEL_SWI3_ISR  0x9002

sft3_svc: JUMP KERNEL_SWI3_ISR;    RTI;      RTI;    RTI;

.ENDSEG;
```

Tabla de Interrupciones para ser usada por el filtro IIR con la Rutina IRQ1

Código: *Tabla_ Interrupciones.asm*

```
/* ********************************************************************************/
/*                    VECTOR INTERRUPCIONES  ADSP-21065L                          */
/*                          Filtto FIR con selleción del tipo de filtro           */
/* ******************************************************************************** */


.EXTERN             Procesamiento_Muestras_Audio;
.EXTERN             _main;
.EXTERN             Init_DSP;
.EXTERN             Cambio_coeficientes_filtroIIR;          //Nueva Línea
.GLOBAL             spt1_svc;


.SEGMENT/PM   isr_tbl;                  /* 21065L Interrupt Service Table */

/* 0x00 Reserved Interrupt */
/*                0x00    0x01    0x02    0x03    0x04 */
/* reserved_0:   NOP;    NOP;    NOP;    NOP;    NOP; */

/* *** Reset vector *** */
/* 0x05 - reset vector starts at location 0x8005 */
rst_svc:          call Init_DSP;
                  NOP;
                  jump _main;

/* 0x08 - Reserved interrupt */
reserved_0x8:  NOP;    NOP;    NOP;    NOP;

/* 0x0C - Vector for status stack/loop stack overflow or PC stack full: */
sovfi_svc:        RTI;     RTI;     RTI;     RTI;

/* 0x10 - Vector for high priority timer interrupt: */
//tmzhi_svc:      jump genera_onda_seno;        RTI;     RTI;     RTI;
tmzhi_svc:        RTI;     RTI;     RTI;     RTI;

/* 0x14 - Vectors for external interrupts: */
vrpti_svc:        RTI;     RTI;     RTI;     RTI;

/* 0x18 - IRQ2 Interrupt Service Routine (ISR)  */
irq2_svc:         RTI;     RTI;     RTI;     RTI;

/* 0x1C - IRQ1 Interrupt Service Routine (ISR)  */
irq1_svc:         jump Cambio_coeficientes_filtroIIR;      RTI;     RTI;     RTI;


/* IRQ0 INTERRUPT VECTOR */
/* ------------------------------------------------------- */
/* IRQ0 Interrupt, Used by the UART and Debug Monitor Program */
```

```
/* *** 0x20 - IRQ0 Interrupt Service Routine (ISR) , 4 locations max *** */
irq0_svc:
        /* JUMP KERNEL_UART_ISR; */ RTI;   RTI;     RTI;     RTI;

/* 0x24  - Reserved interrupt     */
reserved_0x24: NOP;    NOP;    NOP;    NOP;

/* 0x28 - Vectors for Serial Port 0 Receive A & B DMA channels 0/1 */
spr0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x2C - Vectors for Serial Port 1 Receive A & B DMA channels 2/3 */
spr1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x30 - Vectors for Serial Port 0 Transmint A & B DMA channels 4/5 */
spt0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x34 - Vectors for Serial Port 1 Transmit A & B DMA channels 6/7 */
spt1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x38 - Reserved Interrupt */
reserved_0x38: RTI;    RTI;    RTI;    RTI;

/* 0x3C - Reserved Interrupt */
reserved_0x3c: RTI;    RTI;    RTI;    RTI;

/* 0x40 - Vector for External Port DMA channel 8 */
ep0_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x44 - Vector for External Port DMA channel 9 */
ep1_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x48 - Reserved Interrupt */
reserved_0x48: RTI;    RTI;    RTI;    RTI;

/* 0x4C - Reserved Interrupt */
reserved_0x4c: RTI;    RTI;    RTI;    RTI;

/* 0x50 - Reserved Interrupt */
reserved_0x50: RTI;    RTI;    RTI;    RTI;

/* 0x54 - Vector for DAG1 buffer 7 circular buffer overflow */
cb7_svc:        RTI;    RTI;    RTI;    RTI;

/* 0x58 - Vector for DAG2 buffer 15 circular buffer overflow */
cb15_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x5C - Vector for lower priority timer interrupt */
tmzl_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x60 - Vector for fixed-point overflow */
fix_svc: RTI;    RTI;    RTI;    RTI;

/* 0x64 - Floating-point overflow exception */
flt0_svc:       RTI;    RTI;    RTI;    RTI;
```

```
/* 0x68 - Floating-point underflow exception */
fltu_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x6C - Floating-point invalid exception */
flti_svc: RTI;    RTI;    RTI;    RTI;

/* 0x70 - User software interrupt 0 */
sft0_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x74 - User software interrupt 1 */
sft1_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x78 - User software interrupt 2 */
sft2_svc:       RTI;    RTI;    RTI;    RTI;

/* 0x7C - User software interrupt 3 */
#define KERNEL_SWI3_ISR  0x9002

sft3_svc: JUMP KERNEL_SWI3_ISR;    RTI;      RTI;    RTI;

.ENDSEG;
```